UNIVERSIDADE D
COIMBRA

Tiago Miguel Matrola Simões

APLICAÇÃO BASEADA EM MICROSSERVIÇOS PARA FINS CIENTÍFICOS

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em engenharia de software, orientada pelos Professores Doutores Filipe Araújo e Jorge Cardoso e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

julho de 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Tiago Miguel Matrola Simões

APLICAÇÃO BASEADA EM MICROSSERVIÇOS PARA FINS CIENTÍFICOS

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em engenharia de software, orientada pelos Professores Doutores Filipe Araújo e Jorge Cardoso e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

julho de 2023

Resumo

As aplicações de microsserviços são sistemas compostos por vários serviços independentes, autónomos, com capacidade de integrar várias e distintas linguagens de programação e tecnologias, estão interligados e conseguem comunicar entre si através de APIs (Interfaces de Programação de Aplicações), garantido, desta forma, a continuidade do bom funcionamento das mesmas sem quaisquer interrupções.

As aplicações desenvolvidas sob o padrão arquitetural de microsserviços, disponíveis para fins científicos continuam, contudo, a ser escassas, semelhantes, de reduzida dimensão, com poucas tecnologias e incompletas, comprometendo, desta forma o acesso a dados de monitorização capazes de emular um ambiente real.

O reconhecimento destas limitações impulsionou o desenvolvimento e implementação de uma aplicação de *website* de *e-commerce* que integra três sistemas, o **Publicitaki (**anuncia artigos publicados por lojas), a **Loja** (vende os artigos) e o **Banco** (efetua transações bancárias). A sua demarcação passará pela sua ampla diversidade tecnológica (*frameworks* e linguagens de programação) funcionalidades (comunicação síncrona e assíncrona), escalabilidade e suporte nativo para a *Cloud* (Amazon AWS), com execução a baixo custo.

A aplicação assegurará que a produção de dados será efetiva e completa e fornecerá recursos de monitorização, que serão disponibilizados para conceder informações relevantes, precisas e acessíveis para futuras investigações e análises da comunidade científica.

Palavras-Chave

Monolítico, Arquitetura Orientada a Serviços, Microsserviços, Cloud, Docker, Amazon AWS, Terraform, API REST, MQTT, gRPC

Abstract

Microservices applications are systems composed of several independent, autonomous services, with the ability to integrate several and distinct programming languages and technologies, are interconnected and can communicate with each other through APIs (Application Programming Interfaces), thus ensuring the continuity of their proper functioning without any interruptions.

The applications developed under the microservices architectural pattern, available for scientific purposes are still, however, scarce, similar, small, with few technologies and incomplete, thus compromising the access to monitoring data capable of emulating a real environment.

The recognition of these limitations drove the development and implementation of an e-commerce website application that integrates three systems, the Publicitaki (advertises articles published by shops), the Store (sells the products) and the Bank (performs banking transactions). Its demarcation will be through its wide technological diversity (frameworks and programming languages) functionalities (synchronous and asynchronous communication), scalability and native support for the Cloud (Amazon AWS), with low-cost execution.

The application will ensure that the production of data will be effective and complete and will provide monitoring resources, which will be made available to grant relevant, precise, and accessible information for future research and analysis by the scientific community.

Keywords

Monolith, Service Oriented Architecture, Microservices, Cloud, Docker, Amazon AWS, Terraform, API REST, MQTT, gRPC

Agradecimentos

O sentimento de dever cumprido, quando chegado ao fim de um trabalho, só é possível com o suporte de pilares que apesar de menos visíveis, são cruciais para assegurar os tão ambicionados resultados. A minha gratidão aos incansáveis e sempre disponíveis orientadores Doutores Filipe Araújo e Jorge Cardoso, os sustentáculos que me acompanharam e me orientaram na escolha dos melhores caminhos a trilhar, e a quem ficarei eternamente grato pela dedicação, ensinamentos e confiança que sempre em mim depositaram.

Aos meus pais o meu muito obrigado por todo o carinho e apoio incondicional com que sempre me presentearam e que tanto contribuiu para a concretização deste projeto.

Índice

Acrónimos

API	-	Application programming interface
AMQP	-	Advanced Message Queuing Protocol
CPU	-	Central Processing Unit
CRUD	-	Create, Read, Update e Delete
CQRS	-	Command and Query Responsibility Segregation
HTTP	-	Hypertext Transfer Protocol
HTML	-	Hypertext Markup Language
IDE	-	Integrated development environment
IaaS	-	Infrastructure as a Service
IaC	-	Infrastructure as Code
IT	-	Information Technology
FaaS	-	Function as a Service
JPEG	-	Joint Photographic Experts Group
JWT	-	JSON Web Token
JPA	-	Java Persistence API
JAR	-	Java Archive
JDBC	-	Java Database Connectivity
JSON	-	JavaScript Object Notation
PaaS	-	Platform as a Service
REST	-	Representational State Transfer
RPC	-	Remote Procedure Call
SaaS	-	Software as a Service
SOA	-	Service-Oriented Architecture
SDL	-	Schema Definition Language
SQL	-	Structured Query Language
URI	-	Uniform Resource Identifier
URL	-	Uniform Resource Locator
XML	-	Extensible Markup Language

Lista de Figuras

Lista de Tabelas

Capítulo 1
Introdução

A presente tese insere-se no âmbito do programa de Mestrado em Engenharia Informática, do ramo de Engenharia de Software, do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

O trabalho abraçou o estudo e análise de aplicações já existentes no mercado, destinadas a fins científicos, sob um paradigma de desenvolvimento de microsserviços. Procedeu-se, também, ao levantamento e definição dos requisitos para o desenvolvimento de uma nova aplicação. Por último foram realizadas análises dos resultados obtidos do desenrolar dos trabalhos.

A elaboração do presente relatório tem como propósito descrever, a progressão dos trabalhos desenvolvidos durante a criação da aplicação, e resultados alcançados.

Este capítulo apresenta conteúdos que abordam o enquadramento, motivação, objetivo dos trabalhos e estruturamento do documento.

1.1 Enquadramento

O enquadramento apresenta-se decomposto em enquadramento geral (uma visão mais abrangente das aplicações de microsserviços), e em enquadramento central (desafios enfrentados pelas aplicações de microsserviços no contexto científico).

1.1.1. Enquadramento geral

As aplicações monolíticas [1], pela sua intrínseca natureza centralizada, enfrentam obstáculos quando confrontadas com a necessidade de escalabilidade do *software*. Fortemente acoplados como um único serviço, têm registado dificuldades que se têm traduzido em fragilidades, como falhas, dificuldades em escalar, etc., que põe em causa a qualidade das suas operacionalidades, com o envolvimento de custos adicionais não previstos, resultantes de alterações nos sistemas, que implicam a construção e respetivo *deployment* de uma nova versão de todo o sistema.

Diante de um cenário como este impunha-se a necessidade de introdução de um novo padrão arquitetural capaz de operar, de forma independente, sem o comprometimento dos sistemas, mesmo aquando da presença de eventuais falhas.

A arquitetura baseada em microsserviços apresentou-se como a solução capaz de certificar a continuidade do bom funcionamento dos sistemas, independentemente da sua complexidade, através da fragmentação dos sistemas em pequenos serviços independentes, mas com comunicações interligadas. A autonomia e especialização que caracterizam os microsserviços conferem uma segurança e eficácia ininterruptas, que impedem o efeito "dominó" (impactos em cadeia) característico do padrão monolítico.

A capacidade dos microsserviços [2] de se conseguirem integrar num vasto leque de sistemas distribuídos (agilidade, escalabilidade, facilidade de implementação, independência tecnológica, resiliência e código reutilizável) beneficia todo o processo de desenvolvimento, a nível financeiro, de celeridade, de independência das equipas de trabalho e suas tarefas.

Assim, é fulcral um estudo prévio sobre eventuais desafios a levar em consideração tais como a compilação, a testagem, o controlo de versões, a implementação, a geração de *logs*, a monitorização, as operações de *Debugging* e a comunicação, entre outros, para poder então haver uma decisão assertiva sobre o efetivo padrão de arquitetura a definir.

A ascensão das aplicações tem vindo a ser acompanhada por uma igual ascensão da sua complexidade, frequentemente, avaliada tendo por base o grau de acoplamento entre os seus componentes, que, por sua vez, está diretamente relacionado com a ocorrência de falhas e com a facilidade de manutenção.

O acoplamento [3] reflete o quão interligados estão os componentes de um software, pelo número de classes e/ou módulos que se encontram dependentes entre si. Esta métrica é relevante aquando da análise de qualquer software uma vez que à medida que o acoplamento aumenta, a tarefa de compreender, manter e modificar o código do software acaba por se tornar mais complicada, resultando num aumento da probabilidade de o programador introduzir defeitos (bugs).

A classificação do nível de complexidade de um software assenta nas métricas SFIN (*structural fan-in*) que expressa o número de procedimentos (*procedures*) que invocam um procedimento e SFOUT (*structural fan-out*), que assenta no número de procedimentos requisitados por um determinado procedimento. Valores elevados destas métricas conduzirá a um aumento da complexidade e dificuldade de manutenção do software. Estes indicadores são cruciais durante o processo de design e desenvolvimento de qualquer software.

É, também, importante sublinhar a existência de abordagens arquiteturais mistas que empregam uma combinação de componentes de blocos monolíticos, conhecidos como "*large blocks*", juntamente com microsserviços. Estas arquiteturas mistas visam alcançar um equilíbrio entre a granularidade, e a coesão. Os *large blocks*, em contraste com os microsserviços, englobam funcionalidades mais abrangentes e podem ser implementados como módulos monolíticos ou serviços maiores. São utilizados em casos em que certas partes do sistema requerem uma abordagem mais integrada e coesa, enquanto os microsserviços se concentram em funções específicas e são projetados para serem independentes e autónomos.

1.1.2. Enquadramento Central

A comunidade científica tem encontrado nas aplicações de microsserviços, que se encontram disponíveis para fins científicos, a principal fonte de apoio para os seus trabalhos de investigação.

No entanto, as aplicações de microsserviços oferecidas no mercado não têm vindo a corresponder às necessidades e expectativas de quem as experimenta, independentemente da área do conhecimento em causa, não só pelo escasso número de aplicações disponíveis como também por lacunas (são similares, genéricas, sem soluções de inovação tecnológica, com um número de linguagens restrito, não funcionam na *Cloud* [4], limitação de recursos, adaptabilidade e capacidade de

integração com o ecossistema científico, falta de realismo, não são simples de executar, de baixo custo, etc.), que travam a confiança e o consequente avanço das investigações e consequentemente a viabilidade e fiabilidade dos resultados.

Consciente das lacunas que as aplicações disponíveis no mercado têm apresentado, a escolha do tema pareceu ser uma boa oportunidade para avançar com o desafio de desenvolver uma nova aplicação capaz de oferecer ao mundo científico uma alternativa, uma aplicação de microsserviços que fosse de encontro áquilo que a comunidade científica há muito aguardava, uma aplicação realista, completa, simples de executar, preparada para funcionar na *Cloud* e de baixo custo.

Desta forma, a nova aplicação será desenvolvida, tendo em consideração aspetos fundamentais, que vão assegurar a sua demarcação e êxito relativamente às outras aplicações. Isso inclui a análise e compreensão das fragilidades, correntes, neste tipo de aplicações, durante a sua conceção e desenvolvimento; uma arquitetura que deverá corresponder às exigências do ambiente científico (robusta e escalável); estar preparada para funcionar na *Cloud*, a implementação de algoritmos de deteção e localização de falhas para evitar interrupções; custos mais reduzidos; melhor desempenho relativamente às aplicações concorrentes no mercado, através de testes e potencial para ser experimentada e aperfeiçoada pela comunidade científica.

1.2 Motivação

A comunidade científica tem encontrado dificuldades, persistentes, na sua tentativa de conseguir identificar, para os seus trabalhos de investigação, aplicações *open-source* baseadas em microsserviços, capazes de contribuir para a qualidade e progresso das investigações. Estas aplicações, para além de escassas, apresentam várias lacunas, nomeadamente não disporem de funcionalidades suficientes para uma correta análise de falhas, serem pequenas em escala, contudo, dispendiosas.

Este cenário motivou o desenvolvimento de uma aplicação de microsserviços capaz de se afirmar como uma alternativa às aplicações de microsserviços, para fins de investigação, já existentes no mercado, por ser mais abrangente, acessível, dinâmica, realista, completa, de fácil implementação, com custos reduzidos e inovadora.

A nova aplicação de microsserviços apresenta-se como um *website* de *e-commerce* constituído por três sistemas interligados e suportados por vários serviços: o **Publicitaki**, responsável por expor os artigos vendidos pelas lojas; **a Loja** que efetua a venda dos artigos e o **Banco** que processa as transações bancárias. A diversidade de tecnologias e de comunicações, a escalabilidade, o *deployment* local ou para a *Cloud* e a testagem de carga são os pontos fortes da nova aplicação.

A disponibilidade da nova aplicação de microsserviços na comunidade científica permitirá que a mesma possa ser experimentada, partilhada e aperfeiçoada por vários investigadores, contribuindo, desta forma, para uma notória evolução no futuro dos resultados a obter nas investigações, que se traduzirão num importante avanço do conhecimento e da inovação. A nova aplicação também servirá de referência para o desenvolvimento de novas aplicações de microsserviços para fins científicos, no mercado, independentemente da área considerada.

1.3 Objetivos

O projeto, inicialmente, visava o enriquecimento de uma aplicação já concebida com a inserção de um ou mais sistemas de microsserviços, não tendo, no entanto, avançado pelo facto do mesmo ainda se encontrar, na altura, em processo de conclusão.

Em consonância, com os orientadores, foi estabelecido, para o presente projeto, a criação de raiz de uma nova aplicação.

O trabalho objetiva o desenvolvimento e implementação de três sistemas (*website* que publicita artigos publicados por lojas, denominado **Publicitaki**, **Loja** que vende artigos e **Banco** que efetua transações bancárias) decompostos sob um padrão arquitetural baseado em microsserviços, que independentes e com capacidade de comunicação entre si, através de uma *Application Programming Interface* (API), conseguem englobar todas as fases de gestão de anúncios e vendas de artigos de lojas (submissão de encomendas, notificações, transações financeiras, consultas de artigos, publicação de opiniões, entre muitas outras).

É intenção no término do trabalho, a efetivação da produção de dados e facultar dados de monitorização para a comunidade científica.

Assim, foram delineados os **objetivos principais e secundários** dos trabalhos a alcançar para o cumprimento do projeto:

Objetivos principais:

- Investigação de aplicações baseadas em microsserviços para fins científicos já existentes;
- Proposta do tema da nova aplicação a desenvolver;
- Análise e levantamento dos requisitos;
- Desenvolvimento da arquitetura;
- Definição dos recursos (*Uniform Resource Identifier* - URI) da API REST;
- Definição e implementação dos modelos de dados;
- Monitorização e testagem;

Objetivos secundários:

- Redação de um **artigo científico** com os resultados obtidos na experiência;
- Realização e respetiva documentação de uma ***Peer-review*** (*software-review*) ao código relativo ao modelo de dados;
- **Avaliação** da **usabilidade** da interface gráfica, recorrendo às **Heurísticas de Jakob Nielsen** e respetiva documentação dos resultados;
- Injeção de falhas.

É conveniente salientar que os **objetivos secundários**, apesar de não constituírem o foco principal do desenvolvimento do projeto, envolvem atividades complementares, que embora possam, ou não, vir a ser realizáveis, por

incapacidades temporais, não deixam, por isso, de ser uma mais-valia para o enriquecimento do mesmo.

1.4 Visão do produto

O presente projeto propõe-se à criação de raiz de uma aplicação para fins científicos, pelo que todo o trabalho desenvolvido será pautado por uma transparência e clareza dos objetivos e disponibilização dos requisitos necessários e indispensáveis para o bom desempenho da investigação.

O trabalho principiou por um estudo aprofundado e minucioso da identificação das principais fragilidades das aplicações desenvolvidas sob o padrão arquitetural baseado em microsserviços já existentes, para posteriormente, avançar com o desenvolvimento da aplicação, também ela desenvolvida sob um padrão arquitetural baseado em microsserviços, mas já habilitada a prever e a colmatar eventuais fragilidades.

A aplicação a desenvolver *Publicitaki*, assemelhar-se-á a um *website* reconhecido no mercado de compras *online* denominado Kuantokusta (https://www.kuantokusta.pt/), onde lojistas expõem os seus artigos para venda e clientes dispõem de diversas lojas onde lhes é dada a oportunidade de compararem artigos e preços das mais diversas categorias, antes de efetuarem qualquer compra.

A aplicação incorporará **três sistemas**, o ***Publicitaki***, que anuncia os artigos publicados pelas lojas, a ***Loja*** que vende os artigos e o ***Banco*** que efetua as transações bancárias.

A aplicação será detentora de diversos serviços interdependentes que se comunicarão sob **padrões de comunicação distintos** (comunicação síncrona e assíncrona), **instrumentada**, **monitorizada** e com **aplicações de carga** e injeção de falhas (opção no futuro).

Os resultados serão disponibilizados para a comunidade científica que os utilizará nos trabalhos de pesquisa como uma mais-valia para o progresso das investigações.

Dado que a aplicação funciona como uma ferramenta *benchmark* haverá uma autonomia na seleção da escalabilidade (funciona tanto em ambientes pequenos como em ambiente de grande escala) e da reprodutibilidade (possível instalar em diferentes ambientes e contextos) que melhor se adequa à respetiva pesquisa.

Persona

Os resultados serão disponibilizados para a comunidade científica que os utilizará nos trabalhos de pesquisa como uma mais-valia para o progresso das investigações.

Programadores de *software*, académicos, organizações e até mesmo *apaixonados* pelas tecnologias, serão o público-alvo da aplicação, por esta lhes permitir explorar e avaliar os aspetos relacionados com microsserviços, comunicações, e o seu enquadramento em diferentes contextos.

Para uma melhor identificação dos potenciais utilizadores da aplicação, prossegue-se à descrição geral de cada um deles:

- **Programadores de *software*:** ansiosos e expectantes por adquirir novos conhecimentos, nomeadamente diferentes linguagens de programação e

abordagens de comunicação em microsserviços, que lhes permitam encontrar as melhores práticas e desafios na implementação de sistemas distribuídos;

- **Académicos e entusiastas de tecnologia:** podem utilizar a aplicação como uma plataforma educacional e experimental;
- **Organizações:** interessadas em *abraçar* a arquitetura de microsserviços ou refinar a estrutura já existente, podem utilizar a aplicação como um suporte para avaliar a escalabilidade e a reprodutibilidade das suas próprias soluções e identificar eventuais melhorias;

Para uma melhor e mais esclarecedora identificação dos eventuais utilizadores interessados na aplicação, foi elaborada uma *persona* detalhada, que pode ser consultada no **Anexo A: Persona**.

1.5 Resultados

Os resultados obtidos confirmaram que o desenvolvimento da nova aplicação, apostada numa arquitetura modular e escalável, alcançou os objetivos definidos, por ter conseguido ultrapassar as limitações e fragilidades encontradas nas aplicações de microsserviços para fins científicos, já existentes. A nova aplicação desenvolvida está preparada para funcionar na *Cloud*, mais especificamente na plataforma Amazon AWS. Foram realizados testes de carga para avaliar o desempenho e a escalabilidade da aplicação. Os resultados obtidos permitiram retirar ilações acerca do funcionamento da aplicação num ambiente distribuído, nomeadamente na capacidade de escalar consoante os níveis de carga recebidos, garantindo, assim, a disponibilidade e confiabilidade da aplicação.

1.6 Estruturação do documento

O relatório encontra-se organizado da seguinte forma:

- **Capítulo 1** – Introdução (enquadramento, motivação, objetivo dos trabalhos, cronograma, metodologia *Waterfall*, estrutura do relatório);
- **Capítulo 2** – Estado da Arte (padrões arquiteturais: monolítico, SOA, microsserviços, vantagens e desvantagens dos microsserviços, API REST, *Framework* de desenvolvimento *Web*, comparação entre sistemas de gestão de bases de dados, computação na *Cloud* e *software containers*);
- **Capítulo 3** – Especificação de requisitos de *software* (casos de uso, requisitos funcionais, requisitos não funcionais, restrições de negócio, restrições técnicas, gestão de riscos e *code review*);
- **Capítulo 4** – Opções arquiteturais e tecnológicas (digramas representativos da arquitetura da aplicação, nomeadamente, diagramas de contexto, *containers* e de componentes, e opções tecnológicas tomadas);
- **Capítulo 5** – Implementação;
- **Capítulo 6 –** Metodologia e Planeamento**;**
- **Capítulo 7** – Conclusão e Trabalhos Futuros.

No capítulo seguinte (**Capítulo 2 - Conceitos e Estado da arte**) serão abordados os principais conceitos, como os estilos arquiteturais utilizados no desenvolvimento de software, as fragilidades e limitações das aplicações baseadas em microsserviços disponibilizadas a investigadores, as *frameworks* de desenvolvimento *Web* existentes, e uma introdução à *Cloud*, Docker, e Kubernetes.

Capítulo 2
Conceitos e Estado da arte

No presente capítulo são apresentadas e descritas as características inerentes dos padrões de arquitetura de *software* para uma melhor compreensão do padrão considerado para o desenvolvimento do projeto a desenvolver.

2.1 Estilos arquiteturais

A definição dos padrões de arquitetura, abaixo identificados, permitem reconhecer a adequação do padrão de microsserviços ao projeto, pela análise e avaliação da arquitetura, seus benefícios, limitações e forma de comunicação entre si, quer seja pelo protocolo de Transferência de Hipertexto (HTTP), quer seja por mensagens.

2.1.1. Arquitetura monolítica

O padrão monolítico foi, durante muitos anos, correspondendo às necessidades que se impunham, no entanto, a ascensão da complexidade dos sistemas, veio originar uma dificuldade na gestão dos mesmos, reportando-se algumas debilidades que comprometem a sua capacidade de resposta.

A arquitetura monolítica alicerçada a um único bloco de código que suporta todas as funcionalidades e componentes dos sistemas, sem serviços independentes, torna a sua operacionalidade, totalmente, dependente, provocando um efeito "dominó" altamente perigoso, que obriga à construção e ao *deployment* de uma nova versão de todo o sistema em caso de necessidade de introdução de uma qualquer alteração ou adição de uma nova funcionalidade, como no caso de se pretender fazer um *deployment* na *Cloud*.

2.1.2. Arquitetura Orientada aos Serviços (SOA)

A arquitetura orientada a serviços [5] é um padrão de arquitetura de *software* que se apresentou como uma solução alternativa ao padrão monolítico.

Este padrão objetiva que funcionalidades implementadas pelas aplicações sejam dispostas sob a forma de serviços, supervisionados por um *Enterprise Service Bus* (ESB), com as interfaces a fornecerem um acoplamento flexível.

2.1.3. Arquitetura microsserviços

A arquitetura baseada em microsserviços é uma arquitetura que se impõe em sistemas que pela sua complexidade se tornam mais vulneráveis a eventuais falhas que poderão comprometer todo o sistema.

Por intermédio de uma API os sistemas são suportados por pequenos serviços (microsserviços) que comunicam entre si, e que pela sua independência são capazes

de se protegerem do contágio de problemas imprevistos, para continuarem a operar sem que o utilizador se aperceba.

2.1.4. Principais características

Os microsserviços distinguem-se, essencialmente, pelas seguintes características:

- Fragmentação do sistema em pequenos serviços;
- Foco nos recursos e na lógica dos serviços (daí a serem distribuídos por equipas multifuncionais que podem intervir sempre que julguem necessário);
- Descentralização de dados (cada serviço tem uma Base de Dados independente);
- Tolerância a falhas (as falhas não interditam a continuidade da operacionalidade do sistema);
- Transações distribuídas;
- Escalabilidade;
- Reprodutibilidade.

As operações que envolvem a composição de microsserviços devem acautelar eventuais dificuldades em conseguirem garantir a consistência entre os diversos serviços.

Para suplantar este problema é necessário inserir estratégias de gestão de transações distribuídas e de implementação de padrões de consistência eventual, que irão garantir que os dados serão sincronizados e consistentes em todo o sistema.

2.1.5. Protocolos de comunicação

Embora independentes uns dos outros, acarretando, por vezes, diferentes tecnologias e linguagens, os microsserviços relacionam-se entre si, através de comunicações asseguradas e protegidas por protocolos.

HTTP (*Hyper Text Transfer Protocol*)

O Protocolo de Transferência de Hipertexto [6] é aquele que é mais usado em *websites* na *Internet* pela capacidade de realização de uma comunicação síncrona entre o cliente que envia o pedido e o servidor que envia uma resposta.

Os sistemas desenvolvidos em arquiteturas de microsserviços aplicam este protocolo para a comunicação entre os serviços, por, frequentemente, elegerem o REST (*Representational State Transfer*), aquando da criação dos diferentes serviços.

AMQP (*Advanced Message Queuing Protocol*)

Este protocolo [7] está direcionado para comunicações assíncronas, onde as comunicações não são realizadas de forma direta, sem desvios e intermediários.

As comunicações por mensagem são reguladas por este protocolo dado que para serem entregues aos destinatários necessitam de intermediários como o Spring AMQP Framework e o Apache Kafka.

MQTT (*Message Queuing Telemetry Transport*)

O MQTT [8] é um protocolo de comunicação assíncrona, leve e eficiente, desenvolvido para a troca de mensagens, sob o padrão *publish/subscribe*, entre dispositivos na rede.

A sua ampla utilização em aplicações de *Internet of Things* (IoT), embora não exclusiva, é justificada pela baixa sobrecarga de rede, capacidade de suportar cenários de comunicação de baixo consumo de energia, bem como o seu uso em outras áreas e cenários de comunicação em tempo real, onde a troca de mensagens leve e assíncrona é desejada.

gRPC (*Google Remote Procedure Calls*)

O gRPC [9] é uma *framework* de comunicação desenvolvido pela Google que permite uma comunicação eficiente entre diferentes serviços via **RPC** (Remote Procedure Call).

Utilizado em várias linguagens de programação, o gRPC oferece recursos avançados, como *streaming* bidirecional e serialização eficiente de dados usando o protocolo *protobuf*.

2.1.6. Formas de interação

As interações protagonizadas pelos pequenos serviços pautam-se por uma sincronização assegurada, não apenas pelo protocolo de comunicação, mas, igualmente, pela forma de interação [10], que define a metodologia a seguir durante a execução das tarefas.

Coreografia

Os microsserviços executam as suas tarefas de forma autónoma, assíncrona, prescindindo de assistência, muito embora providenciem um serviço que poderá vir a ser partilhado por outros pequenos serviços.

Rapidez, maior agilidade e reutilização dos microsserviços são os ganhos mais visíveis e garantidos na aplicação desta forma de comunicação.

Orquestração

Comparável a uma orquestra, pela necessidade de um suporte de orientação, que garanta a boa sincronização do todo, esta forma de comunicação integra um *orquestrador* que vai conduzir e orientar os pequenos serviços durante o desenrolar das tarefas.

É, no entanto, relevante notar que a comunicação com cada pequeno serviço acarreta uma demora de resposta por parte de cada um deles, criando uma latência na rede que poderá comprometer a disponibilidade do sistema, podendo mesmo vir a agravar-se, ainda mais, aquando da utilização de centenas de microsserviços, com os sistemas a tornarem-se quase numa arquitetura monolítica distribuída, de alto acoplamento.

2.1.7. Vantagens e desvantagens da arquitetura de microsserviços

Os benefícios dos microsserviços passam pela interação entre os serviços, (apesar de cada serviço poder possuir tecnologias diferentes, que poderá o uso originar diferentes linguagens de programação), pela sua resiliência (permite que cada serviço resista a falhas isoladamente), pela possibilidade de escalar apenas os microsserviços pretendidos, e de intervir numa funcionalidade sem precisar que o *deployment* abranja todo o sistema, como é o caso do padrão monolítico.

Apesar do reconhecimento dos muitos benefícios dos microsserviços é importante distinguir em que projetos se podem realmente enquadrar, de forma a evitar custos de projeto, e desenvolvimento inconvenientes.

É de notar que a fragmentação dos sistemas em pequenos serviços, a operarem em diferentes *containers*, originam um desaproveitamento dos recursos naturais, nomeadamente no processamento, memória e armazenamento, uma repetição de ficheiros, bibliotecas e processos, latência na rede e dificuldades em assegurar a integridade dos dados aquando da realização de transações entre as bases de dados dos microsserviços de forma distribuída (*Two-phase commit* [11]). Também deverão ser levados em conta gastos referentes à segurança dos serviços, dado que determinados estados armazenados, como transações bancárias, não podem estar sujeitos a ataques/intrusões por entidades maliciosas.

Tendo em conta as considerações acima mencionadas, constata-se que a arquitetura baseada em microsserviços revela vantagens por ser modular, flexível a nível tecnológico, escalável, resiliente e acessível em manutenção. No entanto, apresenta fragilidades em termos de complexidade de comunicação, transações distribuídas, latência, e dificuldade na realização de testes de deteção de defeitos.

2.2 REST (Representational State Transfer)

A API REST [12] representa um estilo de arquitetura que enquadra os princípios e restrições que devem assistir aquando da criação de sistemas com *interfaces* bem definidas usados nos *webservices*.

2.2.1. Princípios e restrições

- **Arquitetura Cliente-servidor** (as responsabilidades do cliente e do servidor devem manter-se separadas, para permitir a evolução de cada um deles);
- **Protocolo *Stateless*** (os pedidos devem ser independentes e não estarem relacionados com pedidos anteriores);
- **Arquitetura em camadas (*N-tier*)** (Os sistemas são desenvolvidos em camadas independentes e distribuídas por várias máquinas, que fortalecem e garantem o seu desacoplamento);
- **Interação uniforme** (os sistemas possuem *interfaces* uniformes reguladas por convenções)

- **Utilização de *cache*** (quando as respostas aos pedidos dos clientes são armazenáveis em *cache*, a *cache* poderá encaixar os dados da resposta em novos pedidos similares, de forma a evitar processamentos desnecessário, contribuindo desta forma para o aumento do desempenho).

O cliente sempre que formula um pedido ao servidor deverá assegurar-se de que o faz de uma forma explícita, completa e independente de eventuais pedidos anteriores, para que quando o cliente efetuar um pedido REST e estabelecer uma ligação TCP/IP com o servidor, enviando-lhe uma requisição HTTP GET a partir do endereço indicado, o servidor possa enviar uma resposta HTTP ao pedido do cliente de uma forma correta, que vá de encontro ao seu pedido. O servidor assegura o atendimento não apenas de um cliente, mas sim de vários clientes, independentemente de ambos estarem ou não em ambientes separados;

2.2.2. Métodos HTTP

O sistema REST compreende oito tipos de mensagens que são seguidas e interpretadas tanto pelo cliente como pelo servidor. As mensagens mais correntes são as de leitura (HEAD, OPTIONS, GET) e as mensagens de gravação (POST, PUT, DELETE). As operações CRUD (*Create, Read, Update* e *Delete*) são executadas pelo REST considerando os métodos HTTP.

2.2.3. Arquitetura REST

Este estilo de arquitetura tem essencialmente como "ponto forte" a autonomia dos diferentes componentes no desenvolvimento dos sistemas, que lhes permite adaptarem-se, posteriormente, a novas e diferentes exigências (e.g. alteração da linguagem de programação), quer sejam no *backend* quer sejam no *frontend*, sem, contudo, se prejudicarem mutuamente. Os benefícios também se refletem pela independência e adaptação dos sistemas REST relativamente aos tipos de plataformas e linguagens de programação e tecnologias a adotar.

Os componentes podem ser independentes, com *interfaces* bem definidas e padronizadas, permitindo, desta forma que os trabalhos possam ser desenvolvidos em paralelo, com maior flexibilidade e agilidade, sem criar dependências rígidas entre os respetivos componentes.

Em eventuais necessidades de alterações na linguagem de programação utilizada num componente, a arquitetura REST faculta essa transição, dado que não comprometerá os outros componentes. Esta autonomia permitirá uma escalabilidade e uma manutenção eficiente dos sistemas ao longo do tempo.

2.3 Framework Web

Existem, atualmente, no mercado diversas *frameworks* para a implementação de *webservices*, desenvolvidas, especificamente para cada tipo de linguagem de programação.

Seguidamente, serão apresentadas, de forma mais detalhada, as *frameworks* mais populares e adotadas pela indústria para o desenvolvimento de aplicações *web*.

2.3.1. Spring Boot *Model-View-Controller* (MVC)

Desenvolvida para aplicações *web*, a *Framework* Spring MVC [12] é uma das *frameworks* mais utilizadas, nomeadamente na linguagem de programação Java pela sua "leveza" e contínua evolução. Este tipo de padrão (MVC) permite a reutilização dos códigos e a separação de conceitos em três camadas interligadas, onde o *frontend* é separado do *backend*.

Esta solução possibilita a execução do *webservice* REST (correspondente à lógica de negócio do sistema desenvolvido) em qualquer plataforma, sob a forma de uma aplicação autónoma (é produzido um ficheiro *jar* com todas as dependências e o serviço web Tomcat embutido).

Para a operacionalidade do Spring MVC recorre-se ao DispatcherServlet onde é percetível o fluxo percorrido desde o início do pedido do cliente até à obtenção da resposta dada pelo servidor através do HTTP.

Depois de apresentados e descritos os principais conceitos desta Framework, serão seguidamente apresentadas as suas principais características.

Análise

Framework criada pela **Pivotal Software**, e utilizada no desenvolvimento de aplicações Web. Tal como o seu concorrente, o Microsoft ASP.NET Core, utiliza o padrão MVC (Modelo, Vista, Controlador), para isolar o código.

Desvantagens:

- Dificuldade e tempo desperdiçado na atualização de projetos antigos;
- Projetos com grande tamanho elevado;
- Controlo limitado da aplicação.

Vantagens:

- Reduz o tempo de desenvolvimento das aplicações. Evita a necessidade de escrita de código desnecessário ou duplicado;
- Boa integração com outras Frameworks. Por exemplo, os testes, poderão ser realizados recorrendo à Framework JUnit para testes, ou à Framework do SpringBoot a SpringTest;
- Disponibilização de serviços;
- Boa integração com as linguagens **Java** e Groovy;
- Integração com os servidores Apache Tomcat e Jetty;
- Desnecessidade de configurar ficheiros de configuração XML;
- Multiplataforma.

2.3.2. Quarkus

Concebidas para aplicações web, o Quarkus [13] é uma framework utilizada no desenvolvimento de aplicações Java que, tem ganhado, muito recentemente, popularidade na indústria pela sua ascensão no desenvolvimento de aplicações desenhadas sob o padrão de **microsserviços** para a Cloud.

Popularizada pelo seu elevado nível de eficiência, a *framework* Quarkus possui funcionalidades que permitem, não só, a minimização do tempo de iniciação da aplicação, como também a otimização de recursos de processamento e memória, que contribuem para uma maior eficiência da aplicação.

O Quarkus é, particularmente, vantajoso no âmbito de desenvolvimento de aplicações que seguem a metodologia ágil, onde ocorrem modificações frequentes no código da aplicação. Esta razão deve-se ao tempo de inicialização extremamente reduzido e menor consumo de recursos proporcionados pelo Quarkus. Estas características tornam a utilização do Quarkus especialmente adequada para aplicações desenhadas para a *Cloud* ou que serão executadas em *containers*.

Outra vantagem do Quarkus encontra-se no facto de possuir suporte nativo para a execução de aplicações Java em modo nativo, que por sua vez, permite, não só, o aumento de forma abismal do nível de desempenho como também a redução do tamanho da imagem da aplicação, quando comparado, por exemplo, com o Spring Boot.

2.3.3. Django

Publicado no ano de 2005, Django [14] (denominação originária do músico de jazz Django Reinhardt) é uma *Framework* criada para o desenvolvimento de aplicações *Web*, escritas na linguagem Python, sob o padrão MVC. Esta *Framework* utiliza o princípio "DRY" (*don´t repeat yourself*) que consiste na reutilização de código existente, sem necessidade de o voltar a rescrever.

Abaixo são apresentados alguns dos proveitos a retirar com a utilização do Django:

- *Framework* gratuita e de código aberto;
- Extras estendem as suas capacidades que vão permitir abranger tarefas habitualmente operadas (autenticação do utilizador, a gestão dos conteúdos, mapas dos *sites*, *feeds* RSS entre outras);
- Proteção contra futuros problemas de segurança (*SQL injection*, *Cross-site Scripting*, e *clickjacking*);
- Escalonamento rápido e flexível;
- A sua polivalência compreende desde a construção de sistemas de gestão de conteúdo a redes sociais e plataformas de computação científica.

2.3.4. Node.js

O Node.js [15] é uma plataforma de desenvolvimento de aplicações *web*, baseada na linguagem Javascript, amplamente adotada pela indústria. A sua popularidade deve-se, principalmente, à sua capacidade de criar aplicações leves e escaláveis com um elevado nível de desempenho.

Uma das principais características do Node.js é o seu modelo de programação orientado a eventos, que permite tratar elevados níveis de carga de forma eficiente, o que torna esta *framework* especialmente útil no desenvolvimento de aplicações onde o tempo de resposta é fulcral, tal como videoconferências ou videojogos online.

O Node.js possui um vasto leque de bibliotecas que permitem a fácil construção de aplicações mais robustas. Suporta, igualmente, o desenvolvimento de aplicações sob o padrão arquitetural de microsserviços, aplicações desenhadas para a Cloud e a criação de APIs do tipo REST.

2.3.5. Microsoft ASP.NET Core

A Framework .NET, .NET Core, Xamarin, ASP.NET e Web API [16], entre outros, são tecnologias .NET que por vezes se confundem. Para uma melhor compreensão e distinção de cada uma destas tecnologias são, seguidamente, descritas as suas principais características.

.NET

O .NET caracteriza todo o ambiente envolto no desenvolvimento de aplicações construídas sobre tecnologias Microsoft, como as linguagens de programação (*e.g.* C#), *runtimes*, compiladores e outros.

Esta plataforma, estabelece um conjunto de APIs que as suas implementações deverão utilizar, com o objetivo de permitir a criação de bibliotecas que funcionem em todas as tecnologias .NET (como acontece com o .NET Framework, .NET Core e Xamarin). Esta restrição garante a compatibilidade entre as diferentes implementações, visto que permite uma maior facilidade de partilha de código entre programas desenvolvidas sob outras implementações da família .NET.

O .NET oferece, não só, uma ampla biblioteca de *standards* para o desenvolvimento de aplicações *Web*, mas também de um servidor *Web* denominado de Kestrel. Fornece, igualmente, diversos recursos, como a autenticação, o padrão MVC, a ORM Entity Framework Core, a compressão da resposta de pedidos efetuados, a filtragem, e o roteamento, sempre com um desempenho muito elevado. A documentação relativa às tecnologias .NET descritas, encontra-se disponível no *website* da Microsoft.

.NET Framework

O .NET Framework consiste numa implementação de um *runtime* que tem como objetivo a conversão do código Command-line Interface (CIL) para código nativo. É importante salientar que esta *Framework* é exclusiva, apenas, para desenvolvimento de aplicações Windows.

.NET Core

O .NET Core é uma nova implementação do .NET Framework. Contrariamente ao .NET Framework, o .NET Core é uma Framework de código aberto que tem como objetivo permitir criar aplicações para os sistemas operativos Windows, macOS e Linux. A principal desvantagem prende-se no facto desta *Framework* apenas permitir o desenvolvimento de aplicações consola e aplicações *Web*.

Dada as vantagens do .NET Core, muitos creem que a .NET Framework seja, eventualmente, no futuro, descontinuada.

.NET MVC

O ASP.NET MVC é uma funcionalidade disponibilizada pela Framework .NET Core que permite a utilização do padrão MVC nos projetos de aplicações *Web*.

.NET Web API

A .NET WebAPI é uma funcionalidade disponibilizada pela Framework .NET Core que permite o desenvolvimento de APIs REST de forma simples e mais célere.

Recentemente, as tecnologias .NET MVC e .NET Web API uniram-se para formar o ASP.NET Core que permite devolver vistas (*frontend Views*) e dados recorrendo aos mesmos Controllers.

Xamarin

O Xamarin é uma plataforma cujo principal objetivo é a criação de aplicações para os sistemas operativos móveis iOS e Android.

Análise

Após apresentados e descritos os principais conceitos desta Framework, serão seguidamente apresentadas as suas principais características.

Framework criada pela Microsoft Corporation, e utilizada no desenvolvimento de aplicações Web. Emprega o padrão MVC (Modelo, Vista, Controlador), que permite o isolamento do código correspondente a cada uma das camadas.

Desvantagens:

- Restrito aos IDE (*Integrated Development Environment*) **Visual Studio** e Visual Code, e às linguagens **C#, VB.NET, C++, C e F#**;
- O programador dever-se-á encontrar familiarizado com HTML para gerir as mesmas que são autocriadas pelo *webservise*;
- Pouca portabilidade.

Vantagens:

- Rápido, moderno, e necessita de poucos recursos (dado que o projeto é compilado em código nativo);
- Elevado nível de segurança;
- Fácil integração com o Docker;
- Integração com os servidores IIS, Nginx e Apache Tomcat
- Alta integração com os outros serviços, dado que foi concebido para a linguagem C# e para SQL Server.
- *Webservice* em contante atualização, pela Microsoft;
- Documentação excelente e em abundância;
- Existência de monotorização;
- Multiplataforma (Windows, macOS, Linux).

2.3.6. React + Vite

O React [17] caracteriza-se como uma biblioteca JavaScript amplamente adotada para o desenvolvimento de interfaces gráficas dinâmicas para plataformas *web*.

A sua abordagem baseada em componentes reutilizáveis, permite a criação de interfaces complexas de forma mais eficiente, recorrendo a abstrações que possibilitam a construção de componentes encapsulados, semelhantes a *tags* HTML, simplificando, assim, a estruturação e organização do código. Esta abstração dos componentes em blocos isolados aumenta a legibilidade e facilita a manutenção do código, além de promover a reutilização de funcionalidades em diferentes partes da aplicação.

É usual na indústria combinar a utilização do React com a ferramenta Vite, uma vez que esta ferramenta permite efetuar, de forma mais célere, o carregamento dos módulos usados pelo React durante a fase de desenvolvimento da aplicação, evitando, assim, a necessidade de recompilação da aplicação aquando de uma alteração, facilitando, assim o processo de desenvolvimento e de testagem da mesma.

2.4 Sistema de Gestão de Base de dados

Apresentadas as linguagens e *frameworks web* utilizadas no projeto, parte-se para a apresentação dos sistemas de gestão de bases de dados a selecionar para o projeto.

Para a seleção do sistema de gestão de bases de dados (SGBD) que melhor se enquadrasse nas exigências relativas ao desempenho, segurança, concorrência, preço, vantagens e desvantagens, entre outras, foram analisadas as características dos principais SGBD no mercado, nomeadamente o SQLite [18], MySQL[19], Oracle[20], PostgreSQL [21] e o SQL SERVER [22].

Realizada a análise dos diferentes SGBD, foram obtidos os resultados que se encontram sintetizados na **Tabela** 2-1.

Tabela 2-1: Análise comparativa entre os diferentes tipos de sistemas de gestão de bases de dados

Característica	SQLite	MySQL	Oracle	PostgreSQL	SQL Server
Preço	Gratuito	Gratuito	Comercial	Gratuito	Comercial
Complexidade	Simples	Simples	Complexo	Médio	Médio
Limite de armazenamento	Limitada (até 1TB)	Alta (256 TB)	Alta (128 TB)	Média (64 TB)	Muito alta (524 PB)
Suporte a acessos concorrentes	Não	Sim	Sim	Sim	Sim
Multiplataforma	Sim	Sim	Parcial (Windows, Linux, Solaris)	Sim	Parcial (Windows, Linux)
Apoio documental	Médio	Alto	Alto	Alto	Alto
Segurança (autenticação)	Não	Sim	Sim	Sim	Sim
Desempenho	Alto	Alto	Alto	Médio	Alto
Popularidade	Alta	Alta	Média	Alta	Alta

2.5 Comunicação entre a lógica e o SGBD

Foram analisas as possíveis abordagens de interação entre a lógica do sistema e o SGBD.

2.5.1. Comunicação em ASP.NET Core

Desenvolvida pela Microsoft Corporation, a Entity Framework Core, é um tipo de Object Relational Mapper Entity (ORM) desenvolvido, especificamente, para realizar o mapeamento das entidades presentes na base de dados com as classes do código desenvolvidos ASP.NET recorrendo a uma abstração com o intuito de evitar a utilização de *queries* diretas ao SGBD. Inicialmente, esta ORM era parte integral da Framework .NET, todavia a partir da versão seis tornou-se independente da Framework .NET.

2.5.2. Comunicação em Spring Boot e Quarkus

Quando desenvolvida uma aplicação Spring Boot, existe a possibilidade de escolha entre duas API distintas, a API JDBC (Java Database Connectivity) ou a JPA (Java Persistence API). A API JDBC permite uma interação de "baixo nível" com o SGBD, através de comandos SQL, o que permite aumentar o desempenho e manter a compatibilidade do código do modelo de negócio com qualquer SGBD. O mapeamento de cada tabela do SGBD para um objeto de uma determinada classe, deve ser efetuado manualmente coluna-a-coluna, o que torna o processo mais

"enfadonho" para o programador. Em contraste, a API JPA permite o mapeamento de cada uma das tabelas (entidades) para um objeto de uma classe representativa dessa mesma tabela, o que permite um maior nível de abstração, do SGBD por parte do programador. Enquanto a API JDBC foi desenhada para utilização por qualquer plataforma que suporte a linguagem Java, a API JPA foi concebida especificamente para o Java Enterprise e para a Framework Spring Boot.

Definição da API JPA

O JPA é um conjunto de regras (um *standard*), que define classes e métodos (não implementados), que permitem interagir com uma ORM.

A implementação desta API deve ser tratada recorrendo a *Framework* de terceiros que sigam as regras definidas pelo JPA, sendo as mais conhecidas o Hibernate (da RedHat) e o EclipseLink (do Eclipse).

Objetivo da API JPA

A API foi concebida pela Oracle, e tem como objetivo reduzir, o mais possível, a escrita de código de interação com a base de dados, por parte do programador. Assim, o programador interage com a base de dados, como se estivesse a atuar sob objetos alocados localmente, evitando, assim, o uso de *queries*.

Definição da API JDBC

O JDBC é uma API que permite estabelecer uma ligação entre uma aplicação Java e uma ou mais bases de dados, através de código SQL. Para o efeito, é necessário recorrer a um driver específico para cada tipo de base de dados, dado que sistemas distintos recorrem a protocolos de comunicação distintos.

Comparação JDBC *versus* JPA

O JDBC é uma API que permite estabelecer uma ligação entre uma aplicação Java e uma ou mais bases de dados, através de código SQL. Para o efeito, é necessário recorrer a um driver específico para cada tipo de base de dados, dado que sistemas distintos recorrem a protocolos de comunicação distintos. Na Tabela 2-2 encontra-se a análise comparativa entre as duas APIs.

Tabela 2-2: Análise comparativa entre as APIs JDBC e JPA

Critério	JDBC	JPA Hibernante
Complexidade	Simples	Complexo
Manipulação de dados	Simples	Complexo
Mapeamento para classes	Manual	Automático
Linguagem de obtenção de dados	SQL "puro"	HSQL (SQL do JPA)
Permite *Caching*	Não	Sim
Concorrência	Sim	Não
Desempenho Geral	Elevado	Baixo
Desempenho na leitura de dados (*read*)	Elevado	Baixo
Desempenho na inserção de novos registos (*insert)*	Médio	Médio
Desempenho na atualização de registos (*update*)	Elevado	Médio
Desempenho na remoção (*delete*) em cascata	Baixo	Médio

Complexidade: o JPA é uma API com uma aprendizagem simples, em que o programador interage diretamente sobre os objetos mapeados, evitando, assim, recorrer à sintaxe SQL, no entanto, perde o controlo rigoroso sobre base de dados, que o JDBC oferece.

Desempenho: ao obter um objeto DTO, o programador, obtém também referências para os objetos resultantes dos relacionamentos, o que implica um elevado desperdício de desempenho. Por outro lado, o JDBC obtém, apenas, os dados que o programador realmente pretende. O JPA tenta diminuir um pouco esta perda de desempenho, colocando alguns resultados em cache, o que permite economizar alguns recursos, apesar do desempenho ficar muito longe do JDBC.

Remoção em Cascata: o JPA permite mais facilmente apagar registos em cascata do que o JDBC. Todo o processo é efetuado automaticamente pelo JPA.

Concorrência: No JPA, nenhum dos DTO, pode ser utilizado, para escrita e/ou leitura, por várias *threads* em simultâneo, não existindo, por isso, concorrência.

Arquitetura JDBC *versus* JPA

O JDBC permite uma interação a baixo nível entre a base de dados e a aplicação, dando ao programador maior controlo sobre as pesquisas. Ao contrário do JDBC, o JPA, permite omitir a maioria das instruções SQL, fazendo mapeamento automático das tabelas para a aplicação, facilitando o trabalho do programador. No

entanto, o trabalho automático pode custar perda de desempenho ao programador caso este pretenda construir uma aplicação com um tamanho elevado.

O JPA omite as instruções SQL e abstrai o programador de lidar com o JDBC, no entanto utiliza-os internamente.

Conclusão

Em suma, **recomenda-se o uso do JPA**, nos casos em que **organização** e a **arquitetura** do **código** é **fundamental**, em detrimento do desempenho da aplicação. Para o caso oposto, onde o **desempenho** é **crucial** e o controlo da aplicação é menos importante, **é recomendada a utilização do JDBC**.

2.5.3. Comunicação em Node.js

O **Sequelize** caracteriza-se com uma ORM (Object-Relational Mapping) desenvolvida, especificamente, para mapear os registos presentes na base de dados em objetos JavaScript na plataforma Node.js. Esta abordagem elimina o trabalho maçador por parte do programador ao ter de escrever *queries* SQL, de forma manual.

Esta ORM, suporta os SGBD mais conhecidos como o MySQL, PostgreSQL, SQL Server entre outros.

2.6 Computação em *Cloud*

A computação em *Cloud*, cuja denominação se deveu à adoção do logotipo em forma de nuvem, começou a ganhar destaque no início dos anos 2000, quando empresas como a Amazon e a Google disponibilizaram serviços baseados na *nuvem (Cloud)*.

Presentemente, continua a afirmar-se como uma das grandes propulsoras da boa *performance* e desenvolvimento empresarial, pela sua capacidade de armazenamento, redes e processamentos mais robustos como a linguagem natural e a inteligência artificial.

A omnipresença é, efetivamente, a sua maior mais-valia pela capacidade de disponibilizar ao utilizador, através de uma interface com base na *web*, uma total liberdade de localização, substituindo a sua infraestrutura de TI do plano físico para o plano digital, e aceder a toda a informação em diferentes computadores, através de um acesso remoto.

As aplicações desenvolvidas sob o padrão de microsserviços, pela sua independência e diversificação de tecnologias abarcam variados alojamentos. Os serviços da *Cloud* mais difundidos na área industrial são a AWS da Amazon [23], o Azure da Microsoft e a Oracle Cloud que suportam as bases de dados, o *middleware* e o *hardware*, ficando os clientes incumbidos, apenas, de disponibilizarem a aplicação, sem se aperceberem que a mesma poderá se encontrar repartida em pequenos serviços alojados em diferentes origens (*e.g*. uma aplicação pode conter um microsserviço na Microsoft Azure e outro microsserviço da mesma aplicação na Amazon AWS).

É, no entanto, importante salientar eventuais desafios que podem interferir no desempenho e segurança dos serviços alojados em *Cloud* e sistemas distribuídos,

que poderão impedir ou dificultar o acesso dos utilizadores aos sistemas, como o surgimento de falhas não consideradas, manutenções, lotação de solicitações e falta de rede, que podem comprometer o tempo de resposta a um cliente. De forma a colmatar este tipo de ocorrências, as aplicações deverão, antecipadamente, conseguir prevê-las para estarem munidas de ferramentas capazes de as prevenir, (e.g. repetição dos pedidos, reenvio das mensagens).

2.6.1. Modelos de Serviços de computação na *Cloud*

Existem três opções de tipos de serviços:

- ***Infrastructure as a Service*** **(IaaS)** – As organizações pagam um serviço a uma entidade externa que lhes concedem os recursos necessários para albergarem a sua aplicação nos seus *datacenters*, evitando, desta forma, investimentos com novas aquisições de hardware. Os exemplos de IaaS mais conhecidas no mercado são a Amazon AWS, a Microsoft Azure, e a Google Cloud Platform.
- ***Software as a Service*** **(SaaS) -** As entidades externas fornecedoras de serviços alojam e gerem as aplicações de software e infraestruturas, cuidando, igualmente, das manutenções, atualizações de software e aplicação de *patches* de segurança. Os custos restringem-se unicamente à utilização efetivada e àquilo que necessita, podendo amplificar os serviços à medida das suas necessidades. Os exemplos mais conhecidos de SaaS são o Office 365, da Microsoft, e a Google G Suite, da Google.
- ***Platform as a Service*** **(PaaS)** – Serviços que fornecem ambientes onde é possível desenvolver, testar, fornecer e gerir aplicações de *software.* Os programadores desenvolvem as aplicações web ou móveis sem precisarem de se preocupar com a gestão ou configuração das infraestruturas, armazenamento, rede e base de dados, por estes requisitos já estarem assegurados pelos serviços. Os exemplos que mais se destacam são o AWS Elastic Beanstalk, a Microsoft Azure App Service, e a Google App Engine.
- ***Function as a Service*** **(FaaS)** – Serviços da computação na *Cloud* que permitem a criação, execução e gestão de pacotes de aplicações como funções, sem necessidade de manutenção da infraestrutura, executados em *containers stateless*. Os FaaS que mais se destacam são a AWS Lambda da Amazon, a Google Cloud Functions da Google, a Azure Functions da Microsoft, a Open FaaS e IBM Cloud Functions.

2.6.2. Características da *Cloud*

A computação *Cloud* é composta por características capitais que a definem:

- **Agrupamento de recursos** – Os recursos são partilhados, em tempo real, por vários clientes de acordo com as necessidades de cada um deles (armazenamento, processamento, serviços em banda larga);
- **Autoatendimento sob demanda** – Consente que o cliente monitorize a atividade do servidor e o armazenamento de rede alocado e controle as habilidades de computação segundo as suas aspirações;

- **Facilidade de manutenção** – Não são exigidos aos servidores grandes esforços, os períodos de inatividade são, por vezes, baixos ou nulos, os recursos são potencializados e otimizados pelas atualizações regulares;
- **Rapidez de escalabilidade e elasticidade** – Cenários de trabalho extenso, que necessitam, num curto prazo de tempo, acomodar um elevado número de servidores, são compensados pela rapidez de escalabilidade, que garante o desempenho e *performance*;
- **Económico** – O cliente financia, apenas, os recursos que realmente utilizou, havendo alguns recursos de âmbito gratuito;
- **Segurança** – Para proteção dos dados armazenados são geradas cópias, que asseguram a sua recuperação, como sejam arquivos inesperadamente corrompidos;
- **Resiliência** – O serviço tem capacidade de, aquando de uma interrupção, se reiniciar e recuperar de eventuais danos, sem grandes demoras, graças à sua omnipresença que permite o acesso remoto.

2.6.3. Tipos de computação na *Cloud*

Os custos, a disponibilidade, o desempenho e expectativas são os principais agentes considerados aquando da escolha de um dos três tipos de computação *Cloud* disponíveis.

- ***Cloud* pública** – Tudo está disponível na web e compartilhado por vários utilizadores, que embora separados, a usam em simultâneo. Os recursos computacionais, como servidores, e armazenamento estão disponíveis para os particulares e para as organizações. Todas as movimentações e inserções são da responsabilidade dos seus intervenientes, ficando o *provider* encarregado da manutenção e gestão de todos os recursos;
- ***Cloud* privada** – Focada em organizações regidas pela segurança e privacidade de dados e informações. As organizações restringem o acesso aos funcionários e outros de seu interesse, personalizam as funções e sustenta as suas conveniências relacionadas com a sua atividade;
- ***Cloud* híbrida** – Permite compartilhar dados e aplicações entre a *cloud* pública e a *Cloud* privada. Dependendo das necessidades assim será definida a seleção da *Cloud*.

2.6.4. Benefícios da *Cloud*

A computação em *Cloud* confere às organizações inúmeros proveitos:

- **Infraestrutura** – Uma infraestrutura partilhada impede a necessidade de investimento em custos adicionais relacionados com a aquisição extra de vários *hardwares*;
- **Redimensionamento** – Possibilidade de melhoria dos processamentos, autonomia na escolha do número de dados armazenados e gestão dos recursos de computação;

- **Automação** – Suspende a necessidade de investimentos extra na gestão de atualizações de *software* ou de compatibilidades de versões com diferentes sistemas operativos e bases de dados;
- **Mobilidade** – A *Cloud* pode ser acedida em qualquer parte, a qualquer hora (omnipresença), bastando para isso ter acesso a uma base *web*;
- **Subscrição** – a renovação contínua das subscrições reduz os gastos inerentes das primeiras subscrições.

2.6.5. Arquitetura da *Cloud*

Um dos principais focos, aquando do desenvolvimento de um sistema, é a definição da arquitetura a implementar, por forma a impedir potenciais surgimentos de códigos desorganizados (*big ball of mud*).

As primeiras arquiteturas, eram denominadas de arquiteturas unitárias e centravam-se no cliente-servidor, onde um servidor central rececionava os pedidos dos clientes e processava as respostas que eram enviadas a esses mesmos clientes. As arquiteturas modulares cliente-servidor denominadas de arquitetura *3-tier* ou *n-tier* diferenciavam-se pela separação dos dados, lógica e apresentação, independentemente de a três camadas estarem ou não em execução em máquinas separadas.

A arquitetura da *Cloud* é composta por quatro camadas:

- ***Hardware*** – processador (CPU), memória, disco e rede;
- **Infraestrutura** – computação, virtualização e armazenamento;
- **Plataforma** – armazenamento e bases de dados;
- **Aplicações** – aplicações com a lógica de negócio e *webservices*.

2.7 Docker e software containers

2.7.1. Docker

Docker [23] é uma plataforma que permite a criação, *deployment* e execução de aplicações albergadas em *containers* que representam ambientes isolados que permitem a hospedagem de aplicações, a fim de facilitar a sua portabilidade e testagem. Recorrendo a esta plataforma, os componentes de um sistema podem ser agrupados e integrados em vários *containers* de forma eficiente para garantir o correto funcionamento do sistema.

Esta plataforma permite evitar possíveis conflitos de compatibilidade de versões entre programadores em cenários onde, hipoteticamente, dois programadores encontram-se a desenvolver uma aplicação Java. Um dos programadores utiliza o macOS com PostgreSQL 15 e JDK 17, enquanto o outro utiliza o Windows com PostgreSQL 14 e o JDK 1.8. Se ambos executassem a aplicação de forma nativa nos seus respetivos computadores, poderiam surgir problemas e conflitos significativos. No entanto, recorrendo ao Docker, é possível evitar esta situação, uma vez que esta plataforma proporciona um ambiente isolado e consistente para cada programador, onde as dependências e respetivas configurações específicas do

sistema operativo são encapsuladas em *containers* independentes, garantindo, assim, a compatibilidade entre os ambientes de desenvolvimento.

A utilização de bibliotecas do núcleo (*kernel*) do sistema operativo, em comum com outros *containers*, permite que os *containers* sejam portáteis e consequentemente que o trabalho possa ser realizado em conjunto e em ambientes diferentes (*e.g.* localmente no *datacenter* do cliente, num provedor de serviços externo ou na *Cloud*).

Apesar de direcionados, essencialmente, para as arquiteturas de microsserviços, os *containers* também podem ser utilizados para aplicações desenvolvidas sob uma arquitetura monolítica. De salientar que, por vezes, os benefícios de inclusão de *containers* em arquiteturas de microsserviços de baixa complexidade, não são deveras significativos, pelo que a sua exclusão é assentida.

Os Docker *containers*, são compostos por:

- ***Docker Client*** (cria, gere e executa aplicações contidas nos *containers*);
- ***Docker Host*** (recebe e atende os pedidos, gere as imagens e os *containers*);
- ***Registry*** (armazena as imagens de sistemas operativos de forma isolada ou com tecnologias/*Framework* já previamente instaladas no sistema operativo. As imagens dos sistemas operativos poderão ser transferidas diretamente do *website* Docker Hub, ou criadas através de um ficheiro denominado de Dockerfile).

A **Figura** 2-1 permite uma mais fácil identificação dos componentes que integram o Docker e interpretação de toda o funcionamento que os envolve.

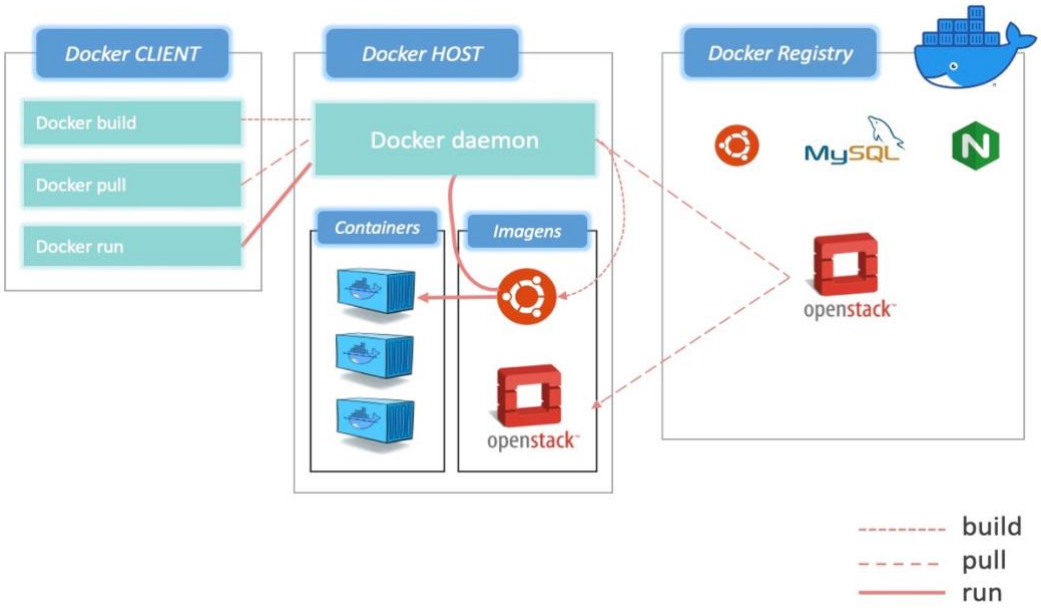


Figura 2-1: Funcionamento da plataforma Docker

2.7.2. Docker *versus* máquina virtual

Frequentemente confundido na indústria com as máquinas virtuais, o Docker apresenta notórias vantagens.

O Docker e as máquinas virtuais aproximam-se nos objetivos, mas afastam-se, substancialmente, no desempenho, portabilidade, segurança, memória, armazenamento e suporte a sistemas operativos. Enquanto os *containers* do Docker partilham as bibliotecas do núcleo do sistema operativo do *Host*, as máquinas virtuais possuem o seu sistema operativo isolado do sistema operativo da máquina do *Host*. A **Tabela** 2-3 ilustra os principais contrastes verificados entre ambos.

Tabela 2-3: Comparação entre Docker e Máquina Virtual

Tecnologia	Desempenho	Portabilidade	Segurança	Memória	Armazenamento	Sistema operativo
Docker	Elevado	Muito alta	Média	Quantidade moderada de memória	Baixo (Megabytes, em média)	Dependente do *hardware*
Máquina Virtual	Muito baixo	Baixa	Muito alta	Quantidade muito elevada de memória	Elevado (Gigabytes, em média)	Independente do *hardware*

Em suma o Docker é uma plataforma de desenvolvimento e execução de aplicações em *containeirs* isolados do resto do sistema. Cada *container* dispõe de todos os recursos que precisa para funcionar de uma forma autónoma (*self-contained).* Por esta razão, o Docker dá a ilusão que cada aplicação está a correr na sua própria máquina virtual, no entanto, do ponto de vista técnico, esta terminologia não é correta, dado que o Docker reutiliza os recursos do núcleo (*kernel*) da máquina do *host*, contrariamente às máquinas virtuais. É, por isso, possível terem-se, na mesma máquina, diferentes aplicações a correr nos seus próprios *containers* Docker sem que estas interferiram entre si.

O Docker é, essencialmente, utilizado, na área de desenvolvimento de aplicações *Web*, uma vez que é possível colocar as aplicações em qualquer servidor sem a preocupação de instalação e gestão de dependências, que poderiam, eventualmente, provocar, entre outros, conflitos ou erros na execução das aplicações.

A maior fragilidade do Docker prende-se no facto de os *containers* apenas funcionarem sob o mesmo tipo de sistema operativo do *Host, limitando, a*ssim, as máquinas Windows a *containers* Windows e Linux (o suporte ao Linux deve-se à funcionalidade Windows Subsystem for Linux (WSL)), as máquinas macOS limitadas a *containers* Unix e o Linux a *containers* Linux.

2.7.3. Kubernetes

O sistema de orquestração de *containers* Kubernetes [24] permite simplificar e automatizar a gestão de aplicações constituídas por múltiplos *containers*. Este sistema disponibiliza as funcionalidades de escalabilidade, distribuição de carga (*load balancing*) e de recuperação de falhas, que asseguram o correto funcionamento e disponibilidade dos *containers*. Quando detetada uma falha ou um problema num dos *containers*, o Kubernetes tem a responsabilidade de substituí-lo por um *container* substituto de forma automática, retirando a responsabilidade de

realização desta tarefa ao programador. Disponibiliza, também, outras funcionalidades como o balanceamento e monitorização do índice de carga e o *rollback* de *deployments*, apesar de a sua principal tarefa ser a gestão de "vida" dos *containers*.

Vantagens do Kurbunetes

Como referido anteriormente, e a fim de evitar eventuais latências que se traduziriam em períodos de inatividade, pela "queda" de *containers*, o Kubernetes assegura a continuidade do bom funcionamento através de uma estrutura que executa sistemas distribuídos de uma forma resiliente, trata do escalonamento, recupera falhas, oferece padrões de implementação, entre outros.

De seguida encontram-se identificados os principais proveitos que podem ser retirados do Kubernetes:

- Capacidade de controlar e distribuir um *container* que possua um elevado índice de carga;
- Liberdade na escolha do sistema de armazenamento que mais satisfaz os requisitos dos *containers*;
- Poderá ser automatizado para criação de novos containers, para remoção dos *containers* existentes e adotar todos os seus recursos para a alocação de um novo *container*;
- Poder-lhe-á ser fornecido um *cluster* de nós que poderá ser utilizado para executar tarefas nos *containers*, com a indicação, para isso, da quantidade de processamento (CPU) e de memória que será necessária para cada container. Para um melhor uso dos recursos poderá encaixar containers nos seus nós;
- Substitui *containers*, reinicia-os quando falham e elimina-os sempre que estes não correspondam;
- Gere informações confidenciais, como palavras-passe e *tokens* de autenticação, atualiza e configura aplicações, sem necessidade de reconstrução da imagem do container e sem divulgar o seu conteúdo durante as configurações.

Seguidamente serão analisados os principais tipos de deployment de aplicações.

Deployment tradicional

As aplicações são executadas em servidores físicos, sem imposição de limite de recursos, o que compromete a boa gestão e distribuição dos recursos em servidores usados por várias aplicações. Uma só aplicação pode açambarcar um elevado número de recursos, privando outras aplicações dos recursos necessários e consequentemente limitando o desempenho das mesmas. O investimento em novos servidores físicos poucas vezes é opção, para colmatar o problema, por não estar contemplado no orçamento das organizações.

Deployment virtualizado

As máquinas virtuais encontram-se em execução numa única unidade de processamento central (CPU) de um servidor físico, onde aplicações estarão em

funcionamento no seu próprio sistema operativo e *hardware* (virtual) de forma isolada.

Container Deployment

Apesar de independentes, como as máquinas virtuais, os *containers* utilizam as bibliotecas e recursos do sistema operativo do *host*, tornando-os assim mais leves quando comparados com as máquinas virtuais. Os *containers* possuem o seu próprio sistema operativo, e partilham recursos de processamento, memória, e espaço com o servidor físico. Por estarem separados da infraestrutura subjacente são portáteis e poderão, assim, ser facilmente utilizados na *Cloud*.

Comparação entre *Deployments*

Na Figura 2-2 encontra-se retratado o antes e o depois da introdução do Kubernetes, para um melhor entendimento e compreensão da mais-valia da sua integração.

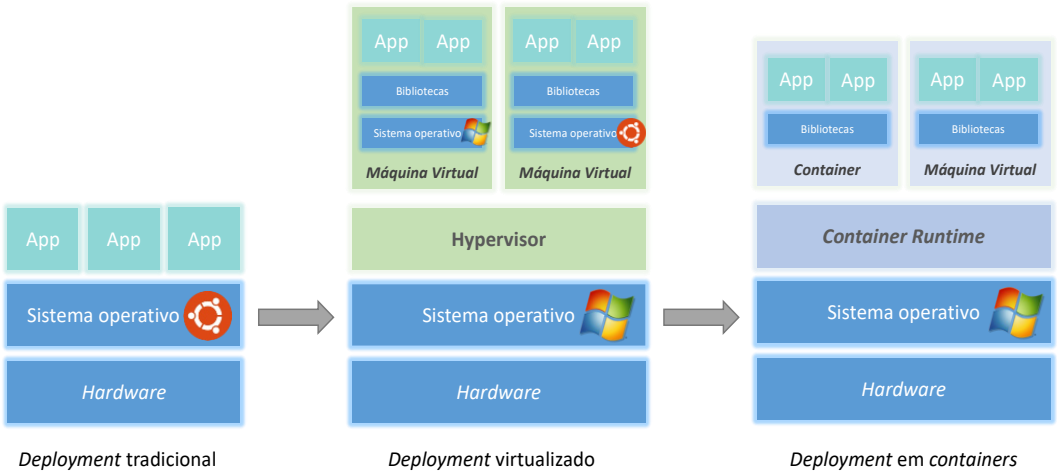


Figura 2-2: Comparação entre os diferentes tipos de deployment de aplicações

Principais vantagens do uso de *containers*

Seguidamente são enumeradas as principais vantagens do uso de *containers*:

- Simplicidade e facilidade no desenvolvimento e implementação de aplicações, por os *containers* permitirem a utilização de imagens pré-configuradas, ao contrário das imagens usadas nas máquinas virtuais onde o processo é mais moroso;
- Fases de desenvolvimento, implementação e integração contínuas;
- Possibilidade de criação de imagens de *containers* das aplicações aquando da sua finalização o que permite desacoplar as aplicações da infraestrutura.
- Consistência entre ambientes de desenvolvimento, teste e produção, uma vez que a mesma imagem funcionará da mesma forma em qualquer máquina;
- Portabilidade, visto que o *container* poderá ser executado localmente, ou numa infraestrutura *Cloud*;
- Abstração do sistema operativo;

- Microsserviços pouco acoplados, distribuídos e escaláveis;
- Isolamento dos recursos dos *containers*, o que permitirá um previsível desempenho das aplicações;
- Reutilização de recursos.

2.7.4. Docker Swarm e comparação com Kubernetes

O Docker Swarm é uma ferramenta disponibilizada pelo Docker que permite a orquestração de *containers* Docker. Esta ferramenta permite a consolidação de *containers* Docker numa estrutura denominada por *cluster* (conjunto de *containers*), que permite simplificar a criação e gestão de serviços distribuídos em *containers*.

Na **Tabela** 2-4 são apontadas as características mais relevantes entre o Kubernetes e o Docker Swarm.

Tabela 2-4: Comparação entre o Kurbunetes e o Docker Swarm

Tecnologia	Instalação	Escalabilidade	Cluster	Carga	Monitorização
Kubernetes	Exigente	Automática	Simples	Manual	Ferramentas integradas
Docker Swarm	Acessível	Inexistente	Complexa	Automática	Ferramentas de terceiros

2.7.5. Computação na *Cloud* com a Amazon Web Services (AWS)

A AWS é uma plataforma de serviços de computação na *Cloud*, detentora de uma infraestrutura composta por uma alta escalabilidade e por uma rede de servidores com um elevado nível de segurança e estabilidade. Lançada em 2006, pela Amazon, é explorada por todo o tipo de clientes, desde as mais variadas organizações a governos e particulares, independentemente da sua geografia, setor e tamanho ficando estes, apenas, obrigados ao pagamento dos serviços que foram contratados. O seu elevado número de serviços e recursos, (computação, o armazenamento, base de dados, *machine learning* e inteligência artificial, *data leaks*, análises e internet das coisas (IoT)), não envolve a necessidade de uma nova aprendizagem, dado que não exige a rescrição das aplicações já existentes e permiti manter os modelos de programação, idiomas e sistemas operativos vigentes, contribuindo, desta forma para que as aplicações acedam, à *Cloud*, sem quaisquer dificuldades, de forma rápida e sem custos expressivos.

2.7.6. Kubernetes versus AWS

Os contrastes mais visíveis entre o Kubernetes e a AWS focam-se na segurança integrada que compõe o Kubernetes, que desenvolve uma rede privada própria com a sua própria rede isolada e segura, independente de fornecedor e executada em qualquer *Cloud*. As cargas de trabalho são portáteis e suportam estratégias híbridas

e *multicloud*, que podem ser executadas dentro da AWS (e.g. Amazon Elastic Kubernetes (EKS)).

2.8 Testagem de serviços

Uma aplicação nunca poderá ser considerada como concluída sem uma rigorosa análise e avaliação, capaz de assegurar, com confiança e segurança, a integridade do seu bom funcionamento. Esse apuramento será efetivado através de uma testagem dos serviços, que determinará os níveis de desempenho da aplicação. Nas subsecções seguintes são reveladas algumas das técnicas e ferramentas disponíveis para o apuramento de resultados.

2.8.1. Locust

O Locust [25] é uma ferramenta utilizada para a realização de teste de carga, *load testing*, para aplicações *Web*, codificada na linguagem Python, de fácil utilização, disponível para várias tecnologias, programável através de *scripts* e escalonável (sustenta, em simultâneo, inúmeros utilizadores). É, também, detentora de uma interface gráfica que disponibiliza, em tempo real, os resultados dos testes e, mesmo durante a execução dos testes, autoriza alterações na carga de dados que, eventualmente, se apresentem como necessárias.

Contrariamente a outras ferramentas para *load testing*, que não possuem capacidade de gerar cargas realistas em *websites* mais abrangentes (*e.g.* páginas diferenciadas para cada utilizador) e utilizam interfaces gráficas não intuitivas ou ficheiros de configuração necessários para a realização dos testes, o Locust obtém uma estrutura Python que lhe permite definir, com segurança, os comportamentos através do código Python.

Os proveitos com a utilização desta ferramenta são, efetivamente, irrefutáveis, contudo podem ser observadas algumas limitações que deverão ser levadas em conta, como a falta de suporte nativo para asserções (*assertions*) e a dificuldade de criação de padrões de *ramp-up* personalizáveis para cargas específicas, por falta de experiência, o que acarreta um dispêndio de tempo muito maior.

Como alternativa ao Locust podem ser encontradas as ferramentas JMeter e o Goose.

2.8.2. *Pipeline* de monitorização de dados

A monitorização *pipeline* de dados assegura que os dados são submetidos a uma observação, que vai assegurar uma maior confiança na qualidade e exatidão dos mesmos. Todavia, é importante salientar que a compreensão semântica dos dados desempenha um papel importante na realização de uma monitorização eficaz. A compreensão da estrutura dos dados possibilita estabelecer métricas adequadas, definir regras de validação e deteção de defeitos, o que contribui, desta forma, para uma monitorização mais consistente.

As organizações que possuem dados de diferentes fontes, sem uma observabilidade contínua poderiam colocar em risco a viabilidade dos dados e consequentemente o sucesso dos objetivos (*e.g.* pipeline de dados na *Cloud*).

A **observabilidade de dados é composta**, entre outros, por **três pilares**, dominantes, denominados de **métricas, *logs* e trace**.

- **Métricas**:
    - Sistemas e aplicações concebem informações sobre os desempenhos;
    - Permite verificar a integridade dos dados;
    - O tipo de métrica a utilizar depende do tipo de observabilidade de dados pretendida;
    - Integra enormes quantidades de informação dispersas por inúmeras fontes (dificuldade em adquirir *insights* úteis sem um sistema adequado para organizar e interpretar métricas).
- ***Logs***:
    - Mantêm informação sobre quase todos os sistemas;
    - Alternativa às métricas por fornecerem informações mais detalhadas;
    - Sustentam enormes quantidades de informação (dificulta a boa gestão das informações, que pode ser ultrapassada utilizando uma ferramenta para guardar, gerir e reunir os *logs* mais relevantes).
- ***Traces***:
    - Rastreios (avaliam a qualidade dos dados de aplicações específicas);
    - Não oferece muito contexto;
    - Fornecem informação sobre a aplicação de origem dos dados, mas excluem a infraestrutura;
    - Imagem completa só é assegurada utilizando outras ferramentas conjuntamente com os dados de rastreio.

A observabilidade quando auxiliada por ferramentas como o painel de monitorização de *pipeline* de dados melhora, substancialmente, a qualidade da sua observabilidade, refletindo-se numa segurança maior nos dados obtidos, o que vai ajudar as organizações a preverem e a prevenirem-se de imprevistos indesejáveis, a otimizar os custos, a melhorar os negócios e a acompanhar permanentemente os dados. Uma auditoria deve ser, também, considerada como garantia de que o sistema está a funcionar corretamente, sem erros.

2.8.3. OpenTelemetry

Desenvolvida, em maio de 2019, pela empresa Cloud Native Computing Foundation (CNCF) responsável pela criação do Kubernetes, a Open Telemetry (OTel) é uma ferramenta de instrumentação que contém um conjunto de APIs e SDKs que permitem a recolha, a exportação e a geração de *traces*, *logs* e métricas, em aplicações com serviços distribuídos. A recolha dos dados de eventos realizados nos microsserviços permite ao programador compreender melhor o seu comportamento e desempenho.

A OpenTelemetry é independente de linguagem ou de plataforma e opera sob uma biblioteca que captura toda esta informação de uma forma unificada sob uma só especificação e envia-a para a localização desejada (*e.g.* por exemplo para um coletor ou para uma outra aplicação).

Esta ferramenta incorpora três requisitos essenciais para uma boa monitorização, nomeadamente o *tracing*, *logs* e métricas.

Como referido anteriormente, são denominadas de observáveis, as aplicações cuja monitorização é realizada através de *logs*, *traces* e métricas.

Tipos de Telemetria (*Telemetry*)

A telemetria é uma técnica que tem como objetivo a análise dos dados gerados pelas aplicações (através de métricas, *logs* e traces). Esta técnica permite o diagnóstico de sistemas distribuídos de uma forma mais simples e célere.

Logs

Os *logs* são mensagens deixadas nas aplicações para serem lidas mais tarde, para verificação do comportamento e desempenho da aplicação. A realização desta tarefa em sistemas distribuídos acaba, contudo, por ser maçadora por acabar por existir um vasto rasto de *logs* (ficam em cadeia) que vai impossibilitar a sua análise de forma clara.

Métricas

As métricas permitem uma visão mais superficial (alto-nível) do estado das aplicações, relativamente ao seu bom funcionamento. Apesar disso, são realmente eficazes na revelação de eventuais alterações comportamentais das aplicações, embora só seja possível identificar a aplicação que sofre a alteração do seu comportamento (*e.g.* alto uso de processamento da CPU). Por falta de informação suficiente, também, não é possível compreender a razão de não se conseguir identificar a raiz do problema, para que o mesmo possa ser colmatado.

A OpenTelemetry permite a realização de *tracings* de forma distribuída, que em conjunto com os *logs* e as métricas, ajudam a solucionar este problema.

Benefícios da OpenTelemetry

O mecanismo e o formato de recolha dos dados gerados pelas aplicações, raramente são consistentes de uma aplicação para a outra, tornando-se numa angústia para os programadores que tentam compreender a integridade da sua aplicação, nomeadamente o seu comportamento e desempenho.

A OpenTelemetry fornece um *standard* destinado a permitir a instrumentação de aplicações nativas da *Cloud*, o que permite às organizações evitar perdas de tempo desnecessárias na criação de mecanismos de recolha de dados das aplicações.

Dada a sua proximidade com o Kubernetes, a OpenTelemetry está a tornar-se a norma da indústria, uma vez que facilita às organizações a realização das implementações de *containers*, dado que não precisam de criar a sua própria plataforma de orquestração.

2.9 Aplicações baseadas em microsserviços para estudo científico

Como, anteriormente, referido na secção **1.1.2 - Enquadramento Central**, foi efetuado um estudo/investigação de aplicações já existentes e desenvolvidas como exemplo de aplicações baseadas em microsserviços. As aplicações em análise, para este projeto, incidiram sobre a Robot Shop da Instana [27], a Sock Shop da Weaveworks e Container Solutions [28], o DeathStarBench da Universidade de Cornell [29], a Tea Store da Universidade de Würzburg [30], a Web Shop de Justus Bogner [31] e por fim a Piggy Metric de Alexander Lukyanchikov [32].

A Tabela 2-5 apresenta de forma sumária as aplicações em análise e a sua respetiva hiperligação do seu projeto GitHub.

Tabela 2-5: Aplicações de microsserviços analisadas no projeto

Nome da aplicação	**Descrição e hiperligação para o repositório GitHub**
Robot Shop	• Loja *online* de venda de *robots* e artigos relacionados; • Comporta vários microsserviços (catálogo de produtos, carrinho de compras, processamento de pedidos e autenticação); • Disponível como recurso didático sobre sistemas baseados em microsserviços; • **Linguagens de programação:** Node.js, Python e Go; • **Bases de dados:** MongoDB e MySQL; • **Orquestração e gestão de *containers***: Docker e Kubernetes; • Mantida pela última vez a 27 de maio de 2021.
	https://github.com/instana/robot-shop
Sock Shop	• Loja *online* de venda de meias; • Comporta vários microsserviços (catálogo de meias, carrinho de compras, processamento de pedidos e autenticação); • Disponível como recurso didático sobre sistemas baseados em microsserviços; • **Linguagens de programação:** Java, Python e Ruby; • **Bases de dados:** MongoDB, MySQL e PostgreSQL; • **Orquestração e gestão de *containers***: Docker e Kubernetes; • Mantida pela última vez a 22 de janeiro de 2021.
	https://github.com/microservices-demo/microservices-demo
DeathStarBench	• Rede social e *website* de reserva de hotéis;

	• Suíte de *benchmark* de referência para sistemas de microsserviços; • Fornece uma plataforma para avaliar o desempenho, a escalabilidade e a eficiência de sistemas baseados em microsserviços; • Insere uma série de cenários de teste que simulam cargas de trabalho realistas e desafios comuns encontrados na arquitetura de microsserviços; • Avalia e compara o desempenho de diferentes implementações e configurações de sistemas de microsserviços; • Ainda não se encontra totalmente finalizada; • **Linguagens de programação:** C++, Java, Python; • **Bases de dados:** Apache Cassandra; • **Orquestração e gestão de *containers***: Docker; • Mantida pela última vez a 25 de janeiro de 2022.
	https://github.com/delimitrou/DeathStarBench
Tea Store	• Loja *online* de venda de chá; • Comporta vários microsserviços (catálogo de chás, carrinho de compras, processamento de pedidos, pagamentos e autenticação de utilizadores); • Elucidativo sobre os conceitos e padrões de desenvolvimento de sistemas distribuídos com microsserviços; • **Linguagens de programação:** Java e Javascript; • **Bases de dados:** MongoDB e MySQL; • **Orquestração e gestão de *containers***: Docker e Kubernetes; • Mantida pela última vez a 10 de novembro de 2020.
	https://github.com/DescartesResearch/TeaStore
Web Shop	• Loja *online* baseada em serviços; • Diferentes serviços são responsáveis por funcionalidades específicas (gestão de produtos, carrinho de compras e processamento de pedidos); • Permite explorar os conceitos e benefícios da arquitetura de microsserviços no contexto de uma loja virtual; • **Linguagens de programação:** Javascript; • **Bases de dados:** MongoDB; • **Orquestração e gestão de *containers***: não incluída; • Mantida pela última vez a 22 de janeiro de 2020.

	https://github.com/xJREB/service-based-web-shop-v1
Piggy Metric	• Aplicação de gestão financeira; • Utiliza as tecnologias Spring Boot e Spring Cloud; • Inclui recursos que permitem a monitorização de todo o sistema. • **Linguagens de programação:** Java; • **Bases de dados:** MongoDB; • **Orquestração e gestão de *containers***: não incluída; • Mantida pela última vez a 23 de janeiro de 2018.
	https://github.com/sqshq/piggymetrics

O estudo minucioso destas aplicações permitiu denotar que as mesmas apresentavam variadas fragilidades, já demarcadas em vastos estudos *online* que compararam estas conhecidas aplicações, como é o caso da DeathStarBench que se encontra incompleta, uma vez que os serviços não interagem entre si e ainda por se se encontrar em fase de desenvolvimento. Foi, também, analisado o artigo ***Evaluating the Effectiveness of Proposed Service-based Maintainability Metrics for Microservices*** de Marcel Szidlovszky [33] que elaborou um estudo comparativo e pormenorizado das características e avaliação qualitativa e quantitativa das aplicações em análise.

Para avaliar quantitativamente as aplicações Szidlovszky levou em conta o seu tamanho e granularidade, o seu acoplamento, e a sua coesão. Já para avaliar qualitativamente as aplicações Szidlovszky (apenas para as aplicações Sock Shop, Tea Shop e Robot Shop) criou cenários comuns de teste, nomeadamente os cenários "Adicionar método de pagamento", "Adicionar categoria para um item", "Alteração dos dados do utilizador" e "Adição de nova moeda (*currency*)".

Granularidade

Na experiência Szidlovszky verificou que, quanto à granularidade, a Piggy Metrics foi a que mais se destacou, seguida da Sock Shop por apresentarem os melhores resultados relativamente ao tamanho e abrangência de cada serviço individual, contrariamente à Tea Store que se evidenciou pela pior granularidade, por ter funcionalidades mais amplas e menos modularizadas.

Serviços com granularidade inadequada (muito grandes ou complexos) podem tornar a manutenção mais difícil na presença de problemas, dado que uma alteração ou retificação numa funcionalidade, em particular, pode implicar modificações numa parte significativa do sistema. Por outro lado, serviços muito pequenos ou demasiadamente fragmentados podem aumentar a complexidade de comunicação e coordenação entre os mesmos.

Acoplamento

No que concerne ao acoplamento a Sock Shop foi aquela que apresentou o melhor resultado, em contraste com a Tea Store que obteve o pior resultado. Szidlovszky explica que o resultado obtido para a Tea Store se prende, fundamentalmente, com o facto de esta aplicação possuir uma arquitetura interdependente, no entanto, há peritos que não levam em consideração esta situação neste tipo de avaliação. Quando esta consideração não é levada em conta, para avaliar a aplicação quanto ao seu acoplamento, a pior aplicação será a Robot Shop a este nível. Altos valores de acoplamento originam dificuldades de manutenção.

Coesão

Relativamente à coesão, o resultado obtido pelas aplicações ficou muito próximo do satisfatório, com a Sock Shop a destacar-se com o melhor resultado. O pior resultado foi apresentado pela Tea Store. Uma alta coesão permite uma melhor manutenção.

Conclusão da avaliação quantitativa

Após analisar o estudo de Szidlovszky, verificou-se que quanto à avaliação quantitativa as aplicações que se destacaram, por evidenciarem os melhores resultados, foram a Sock Shop e a Robot Shop. As restantes aplicações encontram-se com uma avaliação muito próxima, embora sempre com valores satisfatórios. Por outro lado, nas três métricas estabelecidas a Tea Store foi sempre aquela que apresentou os piores resultados e quase sempre muito abaixo da média das restantes aplicações.

Conclusão da avaliação qualitativa

Na avaliação qualitativa das três aplicações, Szidlovszky verificou que aquela que apresentou o melhor resultado para os quatro cenários foi a Tea Store, embora esta aplicação possua o maior esforço (*effort*). Szidlovszky constatou igualmente que a distância da avaliação não era muito distante entre a Robot Shop e Sock Shop, dado que estas duas aplicações possuem arquiteturas semelhantes.

Nova aplicação a desenvolver

Verificaram-se algumas **fragilidades** habituais neste tipo de aplicações baseadas em microsserviços, como sejam a utilização de poucas **linguagens de programação** ou *frameworks web*, um alto nível de **acoplamento** entre microsserviços, o uso de um único **SGBD**, e exclusividade da utilização de **comunicação síncrona,** e de execução **somente local** via Docker, sendo que caso se deseje colocá-la em **funcionamento na *Cloud***, esta tarefa terá de ser **realizada manualmente** (não possuí uma Infraestrutura como Código IaC).

Depois de conhecidas as principais fragilidades das aplicações, baseadas em microsserviços já disponíveis foi, então, possível desenvolver, com mais confiança e segurança, uma aplicação capaz de as colmatar.

Para proporcionar uma maior flexibilidade tecnológica, serão incorporadas diversas linguagens de programação, *frameworks* e opções de SGBD e será projetada para funcionar como uma Infraestrutura como Serviço (IaaS) com recursos de monitoração. Serão adicionadas ferramentas de carga para fins de testagem e validação do sistema.

Desta forma, a aplicação a conceber apresentar-se-á mais robusta, escalável, adaptável e abrangente, capaz de superar os desafios e detentora de um conjunto de recursos avançados que poderão vir a ser explorados para fins científicos e de investigação.

As comunicações entre os microsserviços serão efetuadas, não só, através de mensagens (*e.g. publish-subscribe*), mas também através de **comunicações síncronas e assíncronas** mediante pedidos Remote Procedure Call (RPC).

É importante frisar que a base de estudo não se limitou apenas à exploração de exemplos realistas, como as aplicações acima mencionadas e descritas na Tabela 2-5, mas também se focou em exemplos reais como o *website* KuantoKusta.

No capítulo seguinte (**Capítulo 3 - Especificação de requisitos**) serão abordados os casos de uso, requisitos funcionais e não-funcionais, e as restrições da aplicação a desenvolver.

Capítulo 3
Especificação de requisitos

Evidenciados os principais conceitos e tecnologias envolvidas no desenvolvimento de aplicações suportadas por arquiteturas em microsserviços, prossegue-se com a apresentação do tema da aplicação a desenvolver.

Serão desenvolvidos três sistemas que estarão interligados e que comunicarão entre si. Estes sistemas consistirão num *website* de anúncios de artigos vendidos por várias lojas (***Publicitaki***), numa **loja** e num **banco**. Este cenário assenta numa abordagem de microsserviços e permite integrar tanto comunicações **síncronas** (REST) como comunicações **assíncronas** (MQTT para transações bancárias).

É, no entanto, importante referir que a seleção do tema teve por base um estudo aprofundado sobre o seu enquadramento no contexto e adequação a uma aplicação que seria desenvolvida sob uma arquitetura de microsserviços.

O trabalho de pesquisa junto de aplicações similares, já existentes, mais concretamente *websites de e-commerce online* (umas fictícias e outras reais), permitiram fazer comparações e retirar ilações, que muito contribuíram para uma melhor reflexão e compreensão de quais os requisitos que seriam essenciais e mais adequados a recolher.

Desta forma, procedeu-se à escolha das características e funcionalidades a implementar no desenvolvimento dos sistemas com a identificação e especificação dos casos de uso, requisitos funcionais, requisitos não funcionais e restrições a serem considerados.

3.1 Casos de uso

Os casos de uso exprimem todas as interações dos atores com os sistemas, pelo que uma boa definição e descrição de cada um é fundamental.

3.1.1. Atores

Neste projeto foram identificados os atores **Cliente**, **Administrador do Publicitaki**, **Dono da Loja** e o **Cliente do banco**.

Seguidamente são descritas as tarefas de cada ator nos três sistemas.

Cliente

- **Acede** ao **Publicitaki** gratuitamente, sem precisar de se registar para visualizar artigos e lojas correspondentes, necessitando de se registar, apenas, no caso de querer ter uma ficha pessoal, com acesso ao seu histórico e poder dar a sua opinião relativamente à sua satisfação.
- **Acede**, também, à **Loja** gratuitamente, sem precisar de se registar para visualizar os artigos e suas características e consultar preços, necessitando de se registar, apenas, no caso de querer efetuar uma compra, consultar encomendas, efetuar pagamentos ou publicar a sua opinião acerca de um artigo;

Em suma, o ator ***Cliente*** interage com sistemas **Publicitaki** e a **Loja**. Utiliza o *website* Publicitaki para encontrar as lojas que vendem os artigos desejados, que o reencaminhará para o *website* da loja, que vende o respetivo artigo, onde poderá efetuar a sua compra.

Administrador do Publicitaki

Responsável por toda a gestão do *website* Publicitaki, nomeadamente o registo, atualização e remoção de artigos.

Dono da Loja

Anuncia os seus artigos e dá-se a conhecer registando-se **no Publicitaki**, pagando, para isso, uma subscrição. **Na sua loja** regista os artigos disponíveis para venda, altera artigos, e consulta as suas vendas.

Cliente do Banco

Engloba **todos os atores que realizam pagamentos**, efetuam adição de fundos e que consultam os movimentos da conta. Os atores, que são *clientes do banco*, são o os ***Clientes da loja***, que efetuam o pagamento da compra de artigos, e o ***Dono da loja*** que efetua o pagamento da sua subscrição, para poder publicitar os seus artigos no Publicitaki, e por fim o ***Administrador do Publicitaki*** que recebe o pagamento efetuado pelo dono da loja da sua subscrição.

3.1.2. Sistemas

Os atores apresentados na subsecção **3.1.1 - Atores**, irão interagir com os sistemas denominados **Publicitaki**, **Loja** e **Banco**. Seguidamente são descritos, de forma mais detalhada, estes três sistemas.

Sistema Publicitaki

O sistema Publicitaki tem como objetivo ser um *motor de pesquisa* de **comparação** de preços de **artigos publicitados e disponibilizados por Lojas**. Este sistema permite aos clientes procurar artigos e comparar os preços oferecidos pelas lojas, ajudando-os a encontrar as melhores ofertas disponíveis.

Os clientes que pretendem adquirir novos artigos recorrem a este sistema para encontrar a loja que oferece o melhor preço desse artigo e caso esse artigo seja selecionado encaminha o cliente para o *website* dessa Loja (que no âmbito do projeto será o sistema Loja).

Sistema Loja

O sistema Loja é uma plataforma online especializada na venda de artigos, que atua como um *website* de uma loja *online*. Neste *website*, os clientes encaminhados pelo Publicitaki, podem consultar e adquirir uma vasta gama de artigos.

A Loja apresentará aos clientes uma interface intuitiva, onde podem facilmente visualizar as informações relativas aos artigos disponibilizados de forma detalhada, com a sua respetiva descrição, imagem e preço. Além disso, os clientes podem realizar pesquisas filtradas por categoria e utilizar filtros para encontrar exatamente o que desejam, proporcionando-lhes uma experiência de compra personalizada.

O processo da realização do **pagamento** de encomendas na Loja é efetuado de **forma assíncrona**, entre a Loja e **o sistema Banco**. Durante esta interação, são trocadas as informações necessárias para efetuar o pagamento de forma segura existindo, sempre, a confirmação por parte do banco a notificar a loja de que a transação foi efetuada corretamente.

Sistema Banco

O sistema Banco permite realizar todas as transações bancárias, solicitadas pelos sistemas Publicitaki e Loja, de forma assíncrona. Este sistema, dispõe, ainda, de uma interface gráfica simples para a uma fácil consulta dos dados bancários dos seus clientes.

O sistema Banco desempenha um papel essencial uma vez que é responsável por processar todas as transações bancárias solicitadas pelos sistemas Publicitaki e Loja, de forma assíncrona. Este sistema possui uma interface gráfica que possibilita a consulta e gestão dos dados bancários dos clientes.

Para uma melhor perceção das interações atribuídas a cada ator entre os sistemas foi desenhado um diagrama exemplificativo, conforme demonstrado, pela Figura 3-1 presente na secção **3.1.3 - Diagrama de Casos de Uso - Nível 0**.

3.1.3. Diagrama de Casos de Uso - Nível 0

A **Figura** 3-1 representa os três sistemas a desenvolver, o Publicitaki, a Loja e o Banco, que serão suportados por uma arquitetura de microsserviços que garantirá o pleno funcionamento de cada um dos serviços, pela independência inerente de cada um deles.

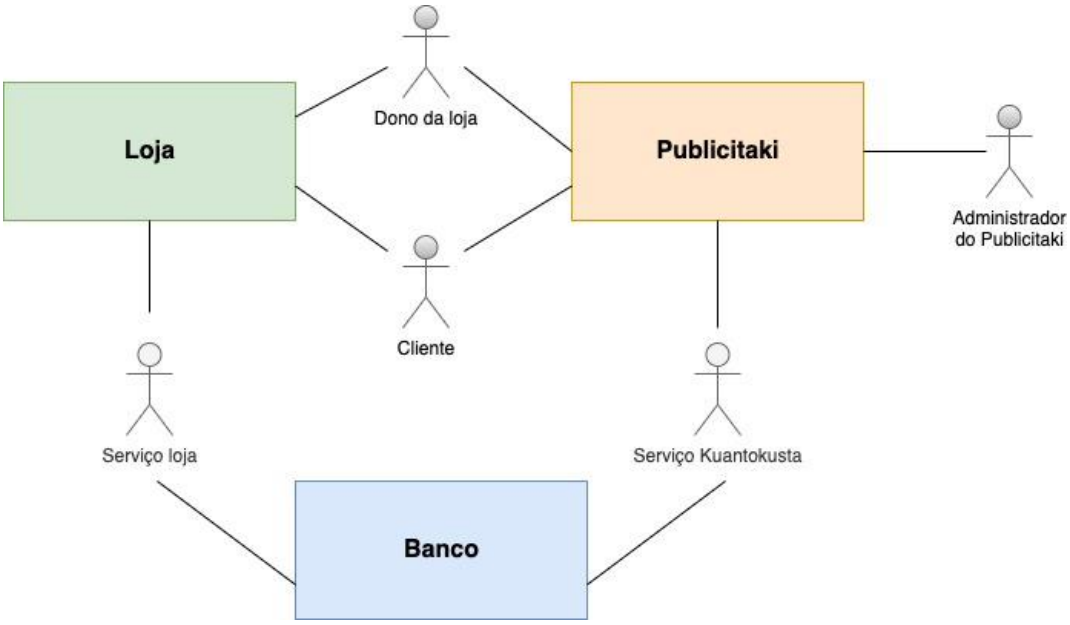


Figura 3-1: Nível 0 do diagrama de casos de uso

3.1.4. Diagrama de Casos de Uso - Nível 1 - Publicitaki

A **Figura** 3-2 representa o sistema "Publicitaki" onde o cliente desta plataforma terá acesso aos artigos disponibilizados e publicitados pelas respetivas lojas, onde poderá, eventualmente, vir a fazer as suas compras na respetiva Loja selecionada. A Loja poderá anunciar os seus artigos e conseguir novos clientes que serão encaminhados, por este sistema, para o *website* da sua loja para poderem comprar os seus artigos.

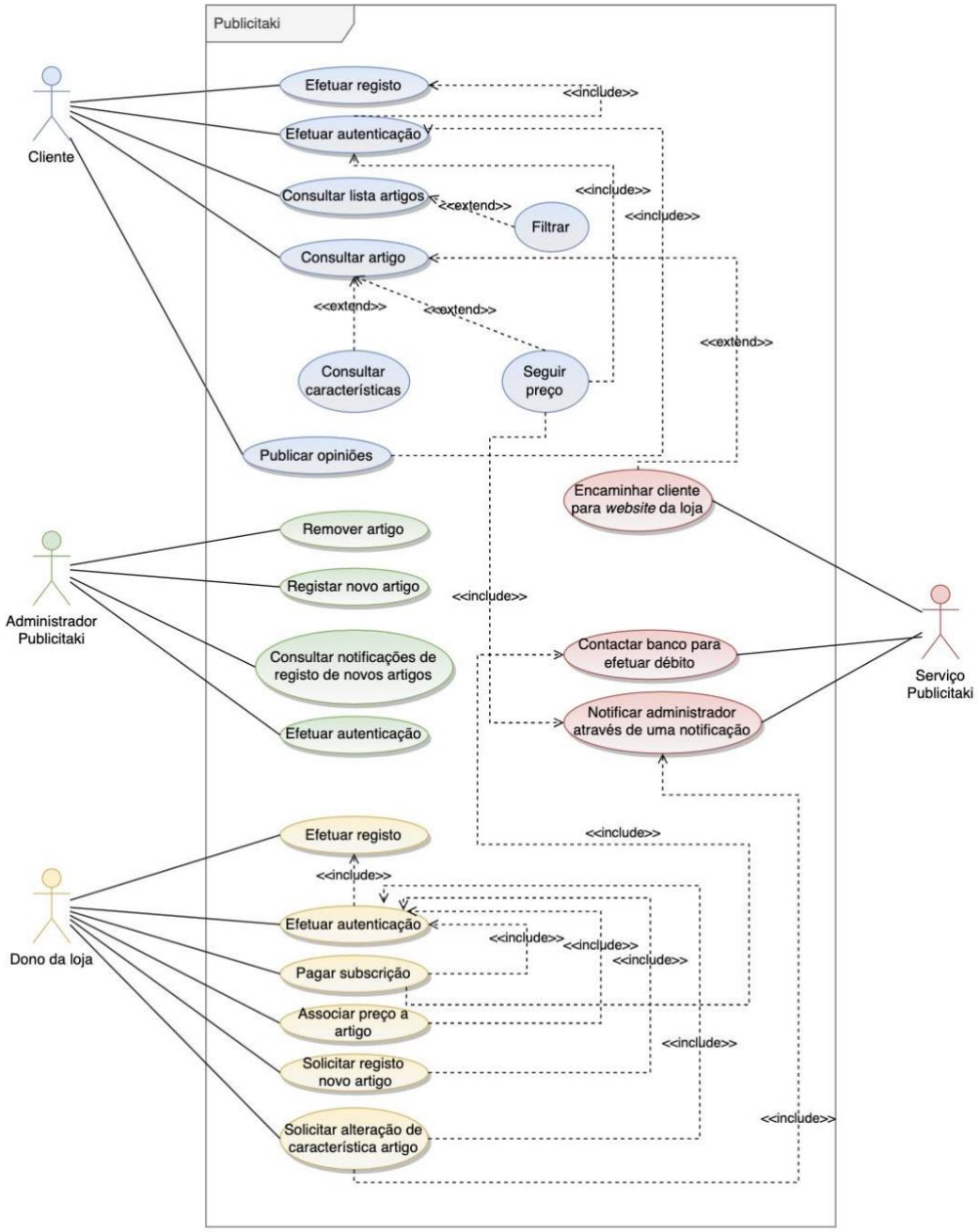


Figura 3-2: Nível 1 do diagrama de casos de uso do sistema "Publicitaki"

Os títulos dos casos de uso a seguir assinalados encontram-se evidenciados com a sua **descrição completa** no **Anexo B: Casos de Uso – nível 1 (Publicitaki)**:

Cliente:

- Efetua registo
- Efetua autenticação;
- Consulta lista de artigos;
- Consulta artigo;
- *Segue* um artigo (equivalente a adicionar artigo aos favoritos)
- Publica opiniões relativas a um determinado artigo.

Administrador:

- Remove artigos;
- Regista novos artigos;
- Consulta notificações de registos de novos artigos.

Loja:

- Efetua registo;
- Efetua autenticação;
- Paga subscrição;
- Associa produto;
- Solicita, ao Administrador, o registo novo produto;
- Solicita, ao Administrador, a alteração das características de um artigo.

Serviço:

- Encaminha o cliente para o *website* da loja;
- Notifica o Administrador aquando da solicitação de registo ou alteração de um artigo.

3.1.5. Diagrama de Casos de Uso - Nível 1 – Loja

A **Figura** 3-3 representa o sistema Loja onde o cliente final tem a possibilidade de consultar os artigos desejados e concretizar as suas compras. A Loja contactará com o sistema Banco para proceder à efetivação do pagamento dos clientes na compra dos artigos.

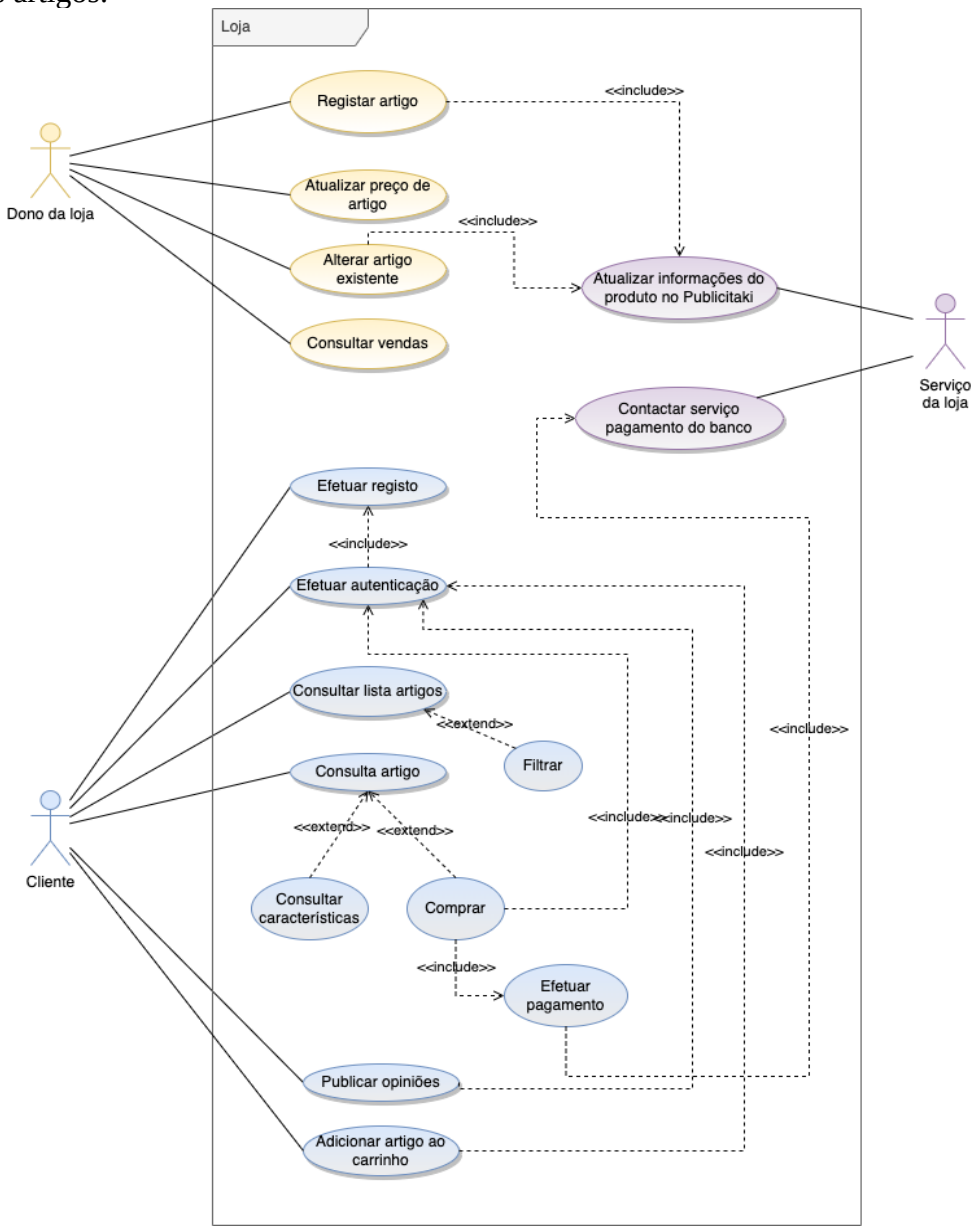


Figura 3-3: Nível do diagrama de casos de uso do sistema "Loja"

Descrição completa dos casos de uso encontram-se evidenciados no **Anexo C: Casos de uso - nível 1 (Loja)**, sendo apenas apresentados os seus títulos a seguir:

Cliente:

- Efetua registo;
- Efetua autenticação;

- Consulta lista de artigos;
- Consulta artigo;
- Publica opiniões;
- Consulta preços;

Loja:

- Regista os artigos;
- Altera características dos artigos;
- Atualiza preço associado ao artigo;
- Consulta as suas vendas.

Serviço:

- Contacta o serviço de pagamento do Banco.

3.1.6. Diagramas de Casos de Uso - Nível 1 - Banco:

A **Figura** 3-4 representa o sistema Banco que permitirá aos clientes procederem ao pagamento das suas compras na loja, e à Loja de efetuar o pagamento pela utilização dos serviços do Publicitaki.

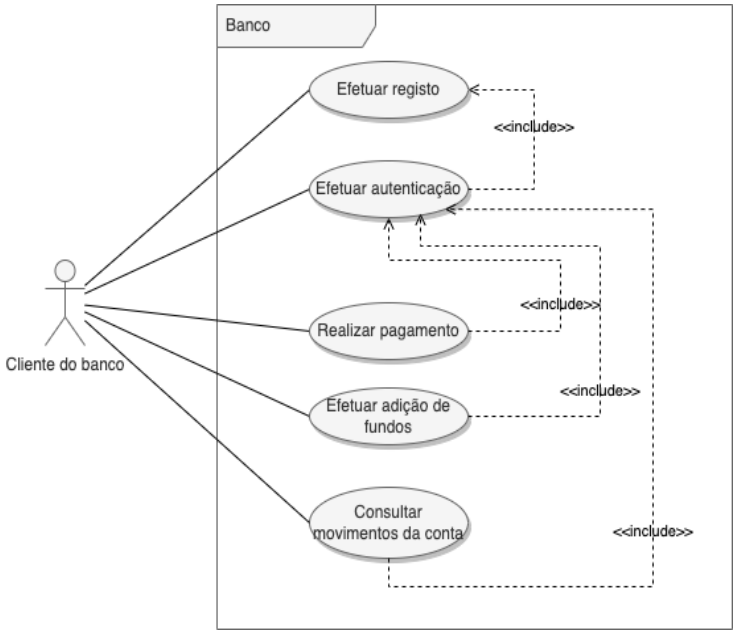


Figura 3-4: Nível 1 do diagrama de casos de uso do sistema "Banco"

Descrição completa dos casos de uso encontram-se evidenciados no **Anexo D: Casos de uso – nível 1 (Banco)**, sendo apenas apresentados os seus títulos a seguir:

Cliente do Banco:

- Efetua registo
- Efetua autenticação

- Realiza pagamentos;
- Efetua adição de fundos;
- Consulta os movimentos da conta.

3.2 Requisitos Funcionais

Após a análise dos casos de uso procedeu-se às especificações dos requisitos funcionais. Nas subsecções seguintes 3.2.1, 3.2.2, e 3.2.3 são apresentados os requisitos funcionais levantados para os três sistemas, Publicitaki, Loja e Banco.

3.2.1. Requisitos funcionais - Publicitaki

Para o sistema **Publicitaki** foram identificados os seguintes requisitos funcionais:

- **RF1** – O cliente e a loja efetuam o seu registo;
- **RF2** – O cliente consulta todos os artigos;
- **RF3** – O cliente consulta um artigo específico;
- **RF4** – O cliente publica as suas opiniões;
- **RF5** – O administrador consulta, insere, remove ou atualiza os artigos registados;
- **RF6** – A loja efetua o pagamento da sua subscrição;
- **RF7** – API de contacto com o sistema externo do Banco;
- **RF8** – API de envio de notificações.

A descrição completa de todos requisitos funcionais deste sistema encontra-se evidenciada no **Anexo E: Requisitos Funcionais (Publicitaki)**.

3.2.2. Requisitos funcionais - Loja

Para o sistema **Loja** foram identificados os seguintes requisitos funcionais:

- **RF9** – O cliente efetua uma encomenda;
- **RF10** – A loja consulta, insere, remove ou atualiza artigos registados no sistema;
- **RF11** – A loja gere as vendas dos artigos disponíveis nas lojas;
- **RF12** – O cliente e a loja consultam as opiniões sobre os artigos comprados;
- **RF13** – O cliente consulta todos os artigos disponíveis para venda no sistema;
- **RF14** – O cliente consulta um artigo específico;
- **RF15** – O cliente regista-se na loja;
- **RF16** – O cliente edita dados pessoais;

- **RF17** – O cliente consulta todas as encomendas realizadas, pendentes, devolvidas e trocadas;
- **RF18** – O cliente publica opiniões sobre os artigos comprados;
- **RF19** – API de envio de contacto com o sistema externo Banco.

A descrição completa de todos os requisitos funcionais encontra-se evidenciada no **Anexo F: Requisitos Funcionais (Loja).**

3.2.3. Requisitos funcionais - Banco

Para o sistema **Banco** foram identificados os seguintes requisitos funcionais:

- **RF20** – O Banco consulta, insere, remove ou atualiza a conta do cliente registada no sistema;
- **RF21** – O Banco gere os pagamentos do cliente e da loja;
- **RF22** – O Banco consulta os movimentos do cliente e da loja.

A descrição completa de todos os requisitos funcionais encontra-se evidenciada no **Anexo G: Requisitos Funcionais (Banco)**.

Serão igualmente efetuados testes funcionais, nomeadamente, testes unitários e de integração.

3.3 Requisitos não-funcionais

Foram, também, identificados os requisitos não funcionais (ou atributos de qualidade) que deveriam ser adotados nos três sistemas. A interação dos sistemas com os utilizadores é fundamental e primordial, pelo que é imperativo não esquecer que a prioridade assenta na capacidade de resposta às necessidades dos utilizadores que são quem atesta, avalia e aprova a viabilidade do sistema.

Desta forma, a escolha recaiu sobre os requisitos não-funcionais que mais se enquadravam nas suas pretensões. Os **requisitos não-funcionais identificados** foram os seguintes:

- **RNF-1: Usabilidade** (facilidade na compreensão e manejo dos sistemas);
    - **Avaliação do requisito:** *Walkthrough* cognitivo e heurísticas de Jakob Nielsen;
- **RNF-2: Segurança** (o sistema está assegurado contra ameaças que possam impedir a sua boa execução);
    - **Avaliação do requisito:**
        - **(Cenário A)** simulação de ataque de *brute-force*;
        - **(Cenário B)** seleção de 10 tipos de palavras-passe inseguras;
        - **(Cenário C)** verificação dos *timestamps* dos pedidos REST.
- **RNF-3: Desempenho** (em caso de congestionamento de acesso ao website, o seu desempenho e tempo de espera não ficam comprometidos);

- **Avaliação do requisito:**
  - **(Cenário A)** retirar o acesso à Internet durante 15 segundos, a fim de exceder o *timeout* do *MQTT*;
  - **(Cenário B)** executar 10 pedidos aleatórios às bases de dados e verificar se o tempo de resposta é inferior àquele estabelecido;

Seguidamente são apresentados, de forma mais detalhada, os requisitos não-funcionais identificados. Na descrição consta os atores envolvidos no requisito não-funcional, a razão da sua escolha e uma hipotética ocorrência da sua ativação (com o estímulo, a fonte, o ambiente, o artefacto envolvido, a forma como o sistema deverá responder e a medida de resposta do estímulo).

3.3.1. RNF-1: Usabilidade

Para garantir a qualidade da interação dos utilizadores com os sistemas, as componentes de *frontend* deverão ser submetidas a testes de usabilidade (**Tabela** 3-1).

Para a aprovação dos testes de usabilidade, a interface, deverá passar com a classificação de, pelo menos, **80% nas 10 heurísticas** de **Jakob Nielsen**.

Tabela 3-1: Requisito não-funcional "usabilidade"

Atores: Cliente, Administrador do Publicitaki, Loja e cliente do banco	**ID:** RNF-1
Atributo de qualidade: Usabilidade	
Razão da escolha do atributo: Tornar os sistemas intuitivos e, assim, aumentar a celeridade da sua utilização.	
História do requisito não-funcional **Estímulo:** pretende aceder aos sistemas e interagir com os mesmos de forma rápida sem perdas de tempo a tentar compreendê-lo ou a ficar sem saber o que deve fazer; **Fonte do estímulo:** Cliente, Administrador do Publicitaki, Loja e cliente do banco; **Ambiente:** funcionamento normal; **Artefacto**: os sistemas; **Resposta**: A interface gráfica da aplicação deverá ser agradável, fácil de utilizar e intuitiva. **Medida da resposta**: Cada uma das dez heurísticas necessitará de obter uma pontuação avaliada numa percentagem igual ou superior a 80%.	

Observações: Para validar os sistemas, o seu respetivo *frontend* será submetido a um teste de usabilidade de forma a verificar se estes respeitam as heurísticas de **Jakob Nielsen**. Será igualmente realizado um **Walkthrough cognitivo**.

Para uma melhor e mais esclarecedora definição das dez heurísticas, procede-se à discriminação individual de cada uma delas.

Heurística n. º 1 - Visibilidade do sistema

Descrição: Perceção e interpretação do ambiente. O utilizador deverá saber onde se encontra, as tarefas que pode e deve realizar no sistema.

Heurística n. º 2 - Correspondência entre o sistema e o mundo real

Descrição: A presença de terminologia técnica deverá ser sempre dispensada na apresentação de mensagens ao utilizador. A intercomunicação (utilizador e sistema) deverá ser coerente e contextualizada com o "modelo mental" do utilizador.

Heurística n. º 3 - Liberdade e controlo do utilizador

Descrição: O utilizador poderá controlar o sistema livremente, sem quaisquer condicionantes limitativas e imposições. O sistema deverá apresentar sugestões (quando o utilizador se sente desorientado ou em situações inesperadas), e/ou disponibilizar alternativas, não restringir movimentos, nem proibir saídas não previstas.

Heurística n. º 4 - Consistência e padrões

Descrição: A consistência visual e de linguagem deverão ser garantidas, bem como os padrões de interação independentemente dos seus contextos. O tipo de linguagem deverá enquadrar-se na do utilizador, os ícones não poderão ser alterados no decorrer de uma mesma ação e deverá haver uma sintonia de tratamento em situações similares para uma mais fácil identificação e melhor interpretação do sistema.

Heurística n. º 5 - Prevenção de Erros

Descrição: Ações críticas como alterações da palavras-passe e remoção de informação do sistema, deverão ser sempre sinaladas e confirmadas antes da sua realização, e deverá ser permitida a sua reposição. Jakob Nielsen já dizia que: *"Melhor que uma boa mensagem de erro é um design cuidadoso que possa prevenir esses erros".*

Heurística n. º 6 - Reconhecer ao invés de lembrar

Descrição: O utilizador deverá ser dispensado da obrigatoriedade de reter as lembranças das suas ações para as poder reconhecer. A interface deverá conseguir identificar as mais diversas situações e orientar o utilizador nas suas ações.

Heurística n. º 7 - Flexibilidade e eficiência de uso

Descrição: O sistema deverá possibilitar aos utilizadores experientes a possibilidade de realizarem as tarefas habituais de forma mais eficaz e célere. A

interface do sistema deverá, igualmente, permitir a sua personalização de acordo com os critérios do utilizador.

Heurística n. º 8 - Estética e design minimalista

Descrição: A interface deverá ter uma linguagem simples e corrente para não confundir o utilizador e um aspeto visual leve e apelativo (*e.g.* evitar a utilização de cores excessivas), com uma coloração adequada para também não confundir o utilizador.

Heurística n. º 9 - Ajudar utilizadores a reconhecer, diagnosticar e recuperar erros

Descrição: Todas as mensagens de erro deverão ser reportadas ao utilizador numa linguagem simples e acessível, no entanto, os códigos de erro identificados com intuito de diagnosticar os problemas deverão ser, sempre, evitados.

Heurística n. º 10 - Ajuda e documentação

Descrição: É aconselhável que, para compreender funcionamento do sistema, não sejam necessárias explicações ou indicações adicionais. O sistema deverá, também, fornecer toda a documentação necessária para poder ajudar o utilizador a completar as suas ações.

Áreas de foco

Como já referido, os testes de usabilidade serão aplicados às duas áreas de foco. As áreas de foco consideradas serão as páginas de interação com os sistemas Publicitaki e Loja e.

Walkthrough Cognitivo

Reveladas as heurísticas de Jakob Nielsen para a avaliação da usabilidade, prossegue-se à explicação de como será orientada e efetuada a avaliação das duas áreas de foco.

Pretende-se com o *Walkthrough* Cognitivo analisar e avaliar, a usabilidade da *interface* do sistema, com o intuito de conseguir encontrar incoerências e erros de usabilidade, onde o utilizador executa a tarefa que lhe é solicitada de forma natural, sem que este necessite de pensar nela. Seguidamente é apresentado um exemplo do tipo de questões que serão utilizadas aquando da realização do *Walkthrough* Cognitivo:

Área de foco - Publicitaki
Questão 1: Quem são os utilizadores?
Questão 2: Na página principal do Publicitaki, o dono da loja percebe facilmente, para onde o botão “Associar preço de artigo” o irá redirecionar?
Questão 3: Na página de listagem de artigos, o cliente percebe que pode filtrar os artigos pelo seu tipo ou características?

Questão 4: O utilizador sabe como, nas páginas de consulta de um artigo, regressar à página anterior?

Questão 5: O dono da loja dispõe de uma opção para consultar a listagem dos seus artigos registados no sistema?

Questão 6: O utilizador (dono da loja e cliente) a qualquer momento pode terminar a sua sessão?

Quando finalizado o *Walkthrough* Cognitivo, será solicitado a um perito convidado que avalie de zero a cem por cento, cada uma das três áreas de foco de acordo com as dez heurísticas de Jakob Nielsen.

Se os sistemas alcançarem uma classificação média superior a oitenta por cento, poder-se-á concluir que os sistemas cumprem o requisito não-funcional "usabilidade".

Os resultados obtidos durante o teste poderão ser consultados no **capítulo 5.1.5 - Testes de usabilidade**.

3.3.2. RNF-2A: Segurança (Cenário A)

Para garantir a qualidade a confidencialidade e a privacidade dos dados dos utilizadores, os sistemas, deverão possuir, obrigatoriamente, um sistema de autenticação (**Tabela** 3-2).

Para a aprovação deste requisito, os dados dos utilizadores deverão estar protegidos de entidades externas e/ou maliciosas (apenas os próprios têm acesso) e as operações críticas nesses sistemas deverão requerer autenticação.

Tabela 3-2: Requisito não-funcional "segurança (cenário A)"

Atores: Cliente, Loja e cliente do banco	**ID:** RNF-2**A**
Atributo de qualidade: Segurança (Confidencialidade)	
Razão da escolha do atributo: Evitar que uma entidade não-autorizada tenha acesso às informações registadas nos sistemas	
<u>História do requisito não-funcional</u> **Estímulo:** os atores autenticam-se, no sistema para usufruírem das suas funcionalidades; **Fonte do estímulo:** Cliente, Administrador do Publicitaki, Loja e cliente do banco **Ambiente:** funcionamento normal; **Artefacto**: acesso ao sistema; **Resposta**: o sistema deverá restringir o acesso a entidades não-autorizadas, garantindo a privacidade dos dados de cada utilizador registado no sistema, ou seja, apenas permitindo acesso aos atores registados; **Medida da resposta**: O sistema deverá apresentar uma página de autenticação onde	

os atores terão de inserir o seu nome e palavra-passe. De seguida, deverá avaliar as credenciais e verificar se estas estão ou não corretas. Caso estejam corretas, autêntica os atores, caso contrário avisa os atores de que as suas credenciais não estão corretas.
Observações: Os sistemas poderão ficar vulneráveis a ataques de *brute-force*.

3.3.3. RNF-2B: Segurança (Cenário B)

O cenário B do requisito "segurança" prevê que os sistemas, possuam um mecanismo de verificação de "qualidade" das palavras-passe escolhidas pelos utilizadores, **não permitindo** que **escolham palavras-passe "inseguras"** (**Tabela** 3-3). Serão classificadas como seguras, as palavras-passe com **oito ou mais caracteres**, os quais deverão ter, pelo menos, **uma letra**, **um número** e **um caracter especial**.

Resumidamente, os sistemas não deverão permitir o registo de utilizadores que escolham palavras-passe inseguras e deverão informá-los da razão pela qual a palavra-passe escolhida não está a ser autorizada. É importante referir que este mecanismo poderá, eventualmente, diminuir a qualidade da usabilidade dos sistemas, uma vez que o utilizador poderá ter de "inventar" uma nova palavra-passe ou que este requisito de oito caracteres faça com que a palavra-passe seja difícil de memorizar.

Tabela 3-3: Requisito não-funcional "segurança (cenário B)"

Atores: Cliente, Loja e Cliente do banco;	**ID**: RNF2-**B**
Atributo de qualidade: Segurança	
Razão da escolha do atributo: Evitar que os atores escolham uma palavra-passe insegura aquando do seu registo no sistema.	
<u>História do requisito não-funcional</u> **Estímulo:** escolha de uma palavra-passe insegura; **Fonte do estímulo:** Cliente, Loja e Cliente do banco; **Ambiente:** fase de registo do ator no sistema (funcionamento normal); **Artefacto**: campo palavra-passe do registo; **Resposta**: palavra-passe insegura rejeitada e solicitada outra alternativa; **Medida da resposta**: mínimo 8 caracteres, um dos quais numérico e outro especial;	
Observações:	

- Poderá existir a **perda de usabilidade** (o ator terá de perder tempo a tentar escolher outra palavra-passe);
- A palavra-passe escolhida pelo ator poderá ser difícil de memorizar, dado que as restrições propostas implicam o uso de uma palavra-passe comprida (esta sensibilidade será mitigada, através da opção de recuperação da palavra-passe).

3.3.4. RNF-2C: Segurança (Cenário C)

O cenário C do requisito "segurança" prevê que o sistema "Publicitaki" associe a data e hora do registo ao artigo registado pela Loja, de forma automática (**Tabela** 3-4). A data e hora deverão serão obtidas pelo relógio do sistema.

Tabela 3-4: Requisito não-funcional "segurança (cenário C)"

Atores: Loja	**ID:** RNF2-**C**
Atributo de qualidade: Segurança (não-repudiação)	
Razão da escolha do atributo: Evitar que a Loja indique uma data e hora incorretas aquando da associação do seu artigo no sistema (Publicitaki).	
História do requisito não-funcional **Estímulo:** a Loja indica uma data e hora falsa aquando da associação do seu artigo no sistema (Publicitaki); **Fonte do estímulo:** a Loja; **Ambiente:** quando efetua a associação do seu artigo no sistema (funcionamento normal); **Artefacto**: o artigo que foi atribuído o preço; **Resposta**: o sistema deverá colocar os campos da data e hora naquele momento, automaticamente, quando for efetuada a associação do preço ao artigo; **Medida da resposta**: Os campos de data e hora são preenchidos automaticamente sem opção da sua edição.	
Observações: A data e hora são as do momento da associação do artigo no sistema.	

3.3.5. RNF3-A: Desempenho (Cenário A)

Para garantir o célere desempenho das transações bancárias efetuadas pelo serviço da loja, o sistema "Banco" deverá ter um *timeout* com um valor máximo da sua

realização (**Tabela** 3-5). Caso este *timeout* seja atingido a transação ficará sem efeito.

Tabela 3-5: Requisito não-funcional "desempenho (cenário A)"

Atores: Serviços da Loja e do Banco	**ID:** RNF-3**A**
Atributo de qualidade: Desempenho	
Razão da escolha do atributo: Evitar que os sistemas fiquem à espera eternamente pela resposta.	
História do requisito não-funcional **Estímulo:** realização de uma transferência bancária por um dos atores (ou seja, é colocada uma nova mensagem num tópico MQTT); **Fonte do estímulo:** os sistemas mencionados; **Ambiente:** funcionamento normal; **Artefacto:** transferência bancária (mensagem MQTT); **Resposta:** A mensagem deverá ter um *timeout* máximo para a realização das transferências bancárias. Caso seja excedido este *timeout* a transação ficará sem efeito. **Medida da resposta:** A transferência deverá ser efetuada em menos de uma hora.	
Observações: -	

3.3.6. RNF3-B: Desempenho (Cenário B)

Para garantir o célere desempenho dos sistemas, as operações de leitura e escrita às bases de dados deverão ter um prazo máximo da sua realização, de forma a evitar atrasos em cadeia nos microsserviços interiores (**Tabela** 3-6). Para minimizar a realização destas operações, as bases de dados irão adotar a abordagem *Command and Query Responsibility Segregation* (CQRS).

Tabela 3-6: Requisito não-funcional "desempenho (cenário B)"

Atores: Todos os atores	**ID:** RNF-3**B**
Atributo de qualidade: Desempenho	

Razão da escolha do atributo: Evitar atrasos no tempo de resposta dos sistemas.
<u>História do requisito não-funcional</u> **Estímulo:** realização de uma operação de CRUD nas bases de dados (*e.g.* registo de um novo utilizador ou de um novo artigo). **Fonte do estímulo:** sistemas Publicitaki, Loja e Banco; **Ambiente:** funcionamento normal; **Artefacto:** pedido realizado às bases de dados; **Resposta:** As operações de leitura e escrita à base de dados deverão ter um prazo mínimo de realização. Caso este prazo seja ultrapassado a operação ficará sem efeito e o ator será notificado do sucedido. **Medida da resposta:** As operações de leitura terão um prazo máximo de 400 milissegundos e as operações de escrita um prazo máximo de 900 milissegundos.
Observações: Este requisito será satisfeito recorrendo à abordagem **<u>CQRS</u>**.

3.4 Restrições

A classificação das restrições de negócio e técnicas foi, igualmente, considerada por ajudar na identificação dos fatores que poderiam condicionar o alcance dos objetivos e metas definidos, caso fossem desrespeitados.

3.4.1. Restrições de Negócio

As restrições de negócio estabelecidas no projeto, pelos orientadores, foram as seguintes:

- **RN1** – A aplicação será desenvolvida em 10 meses;
- **RN2** – As tecnologias serão de caráter gratuito;
- **RN3** – Deverão existir mecanismos de autenticação para todos os atores registados no sistema.

As restrições de negócio do projeto encontram-se evidenciadas, em maior detalhe, no **Anexo H: Restrições de Negócio**.

3.4.2. Restrições Técnicas

As restrições técnicas estabelecidas no projeto, pelos orientadores, foram as seguintes:

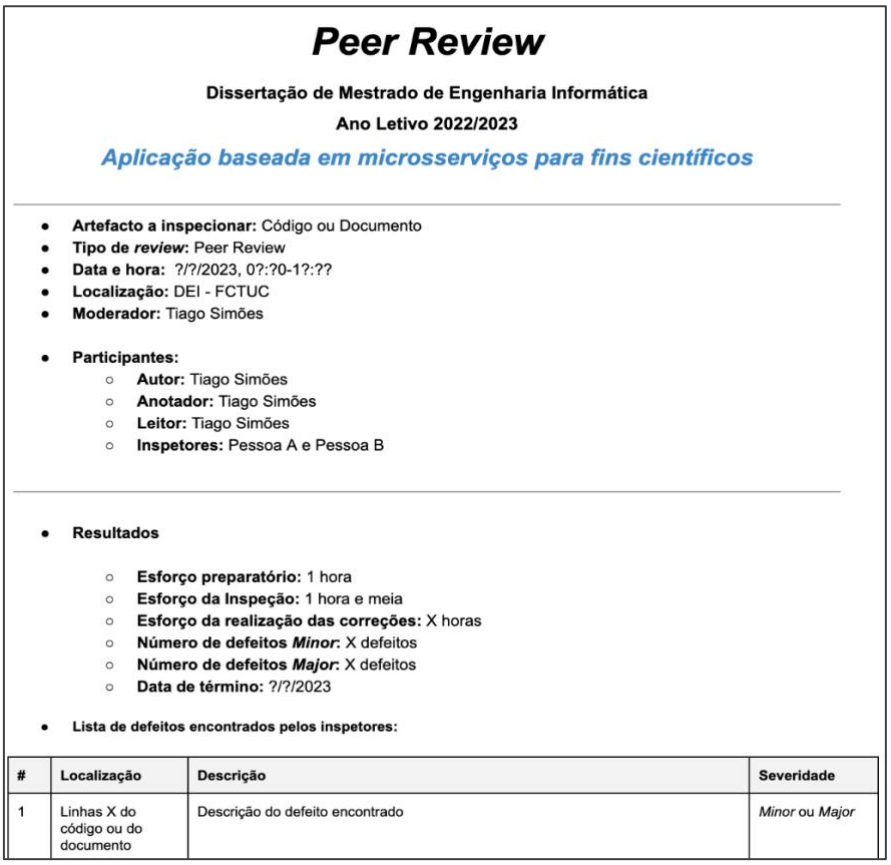
- **RT1** – As aplicações deverão estar encapsuladas em *containers* com sistemas UNIX (excluindo sistemas Windows);
- **RT2** – O sistema deverá funcionar em dispositivos móveis e em computadores *Desktop*;
- **RT3** – A arquitetura deverá ser baseada em microsserviços;
- **RT4** – As palavras-passe deverão ser seguro e deverão, igualmente, estar encriptadas;
- **RT5** –As aplicações deverão estar encapsuladas em *containers* Docker e deverão também ser orquestradas recorrendo ao Amazon AWS;
- **RT6** – As aplicações deverão obedecer ao padrão de mensagens ***Publish-subscribe*.** A tecnologia usada deverá ser o MQTT;

As restrições técnicas do projeto encontram-se evidenciadas, em maior detalhe, no **Anexo I: Restrições Técnicas.**

3.5 *Peer Review*

Para garantir uma maior confiabilidade dos três sistemas foi determinada a realização de uma *review*. Por imposição do presente projeto, este será desenvolvido por uma única pessoa o que impossibilita a realização de uma *review* mais formal denominada de inspeção de *software* de Michael Fagan. Foi, então, decido realizar uma *review* informal denominada de *peer review*. Neste tipo de *review* é solicitado a um ou mais colegas, externos ao projeto, com um mesmo nível de conhecimento técnico equiparado ao autor, a avaliação de um artefacto desenvolvido, podendo este ser um documento ou uma porção de código. Durante a redação do presente documento estes colegas serão apelidados de inspetores, muito embora esta terminologia não seja a mais adequada, visto que poderão apenas poderem ser assim designados nos casos em que a *review* é uma inspeção.

O documento desta *review*, ilustrado pela Figura 3-5 contará com a descrição do artefacto a inspecionar, o tipo de *review* que será realizado, a data, hora, e localização da sua realização. No documento constará os nomes de todos os participantes, nomeadamente o nome do autor/anotador/leitor/moderador e os inspetores. Nesta *review* os defeitos serão classificados de acordo com a sua severidade, como graves (*Major*) ou leves (*Minor*). Serão classificados como graves os defeitos que possam causar uma avaria, e como leves aqueles que não causarão uma avaria. Uma avaria ocorre quando o utilizador se apercebe de um problema existente no sistema ou quando o sistema não responde de forma adequada, ou seja, uma avaria acontece sempre que um problema se torna visível ao utilizador.



Peer Review

Dissertação de Mestrado de Engenharia Informática

Ano Letivo 2022/2023

Aplicação baseada em microsserviços para fins científicos

- **Artefacto a inspecionar:** Código ou Documento
- **Tipo de *review*:** Peer Review
- **Data e hora:** ?/?/2023, 0?:?0-1?:??
- **Localização:** DEI - FCTUC
- **Moderador:** Tiago Simões

- **Participantes:**
  - **Autor:** Tiago Simões
  - **Anotador:** Tiago Simões
  - **Leitor:** Tiago Simões
  - **Inspetores:** Pessoa A e Pessoa B

- **Resultados**
  - **Esforço preparatório:** 1 hora
  - **Esforço da Inspeção:** 1 hora e meia
  - **Esforço da realização das correções:** X horas
  - **Número de defeitos *Minor*:** X defeitos
  - **Número de defeitos *Major*:** X defeitos
  - **Data de término:** ?/?/2023

- **Lista de defeitos encontrados pelos inspetores:**

#	Localização	Descrição	Severidade
1	Linhas X do código ou do documento	Descrição do defeito encontrado	*Minor* ou *Major*

Figura 3-5: Documento a utilizar na *review*

Antes da realização da *review*, será dispensado um esforço preparatório de uma hora. Posteriormente, será enviado aos inspetores a porção do código a inspecionar. Está previsto um esforço de uma hora e meia na realização da *review*. Após a sua conclusão, serão efetivadas as correções dos defeitos apontados pelos inspetores.

3.6 Plano de Riscos

Depois de levantados os requisitos foi possível enumerar os possíveis riscos que, eventualmente, poderiam surgir. Os riscos onde a multiplicação da Probabilidade com o Impacto (P * I) forem superiores a 16 serão aqueles que terão uma ação de mitigação associada. A listagem completa dos riscos identificados encontra-se apresentados na **Tabela** 3-7.

Tabela 3-7: Listagem dos ricos identificados

ID	**Probabilidade**	**Impacto**	**Declaração do risco (Condição e Consequência)**		**Ação**
RS-1	4	4	**Risco pessoal:** A pouca familiarização com o Docker e o Amazon AWS.	**Consequência:** Aumento do número de erros de programação (bugs) e/ou demora no tempo de execução de tarefas, comprometendo assim os prazos estipulados.	EM-1
RS-2	2	3	**Risco tecnológico:** As diferentes versões do Docker poderão dificultar a fase de deployment da aplicação.	**Consequência:** A aplicação poderá não funcionar corretamente quando colocada em funcionamento	-
RS-3	3	5	**Risco pessoal:** A inexperiência na estimação das tarefas (na fase de planeamento) poderá fazer com que os prazos não sejam bem definidos.	**Consequência:** Possível atraso na entrega final da aplicação.	-
RS-4	4	5	**Risco pessoal:** Comprometimento na gestão do tempo originado por outras unidades curriculares.	**Consequência:** Demora no tempo de execução de tarefas, comprometendo assim os prazos estipulados.	EM-2

EM-1 Estratégia de Minimização [RS-1]

Para minimizar o impacto do risco RS-1 foi decidido recorrer aos tutoriais disponíveis *online* para poder compreender melhor a utilização do Docker e do Kurbunetes. Caso surjam dúvidas, estas poderão ser esclarecidas junto dos orientadores.

EM-2 Estratégia de Minimização [RS-4]

Para minimizar o impacto do risco RS-4 foi decidido estabelecer dias em concreto para a realização de cada trabalho de cada unidade curricular.

No capítulo seguinte (**Capítulo 4 - Arquitetura**) serão apresentados os diagramas descritivos da arquitetura da aplicação e as opções tecnológicas, desde as linguagens de programação até aos sistemas de gestão de bases de dados selecionados.

Capítulo 4
Arquitetura do sistema

Depois de efetuadas as especificações dos requisitos de software dos três sistemas, foi possível avançar para a fase de especificação da arquitetura dos sistemas e escolha das tecnologias.

4.1 Arquitetura

Para uma boa compreensão da arquitetura dos três sistemas, e respeitando as exigências estabelecidas na fase de levantamento dos requisitos, recorreu-se ao modelo C4 de Simon Brown firmado em diagramas de Contexto, de Containers e de Componentes.

Este modelo permite, retratar as atividades través da sua ramificação em vários diagramas, para melhor interpretar e compreender a arquitetura desenvolvida.

4.1.1. Nível 1 – Diagrama de Contexto

O diagrama de contexto, identificado na Figura 4-1, evidencia as arquiteturas gerais dos três sistemas (Publicitaki, Loja e Banco), nomeadamente o ambiente que os envolve (no centro rodeados por um picotado) e os quatro atores que, à sua volta, vão interagir com os sistemas.

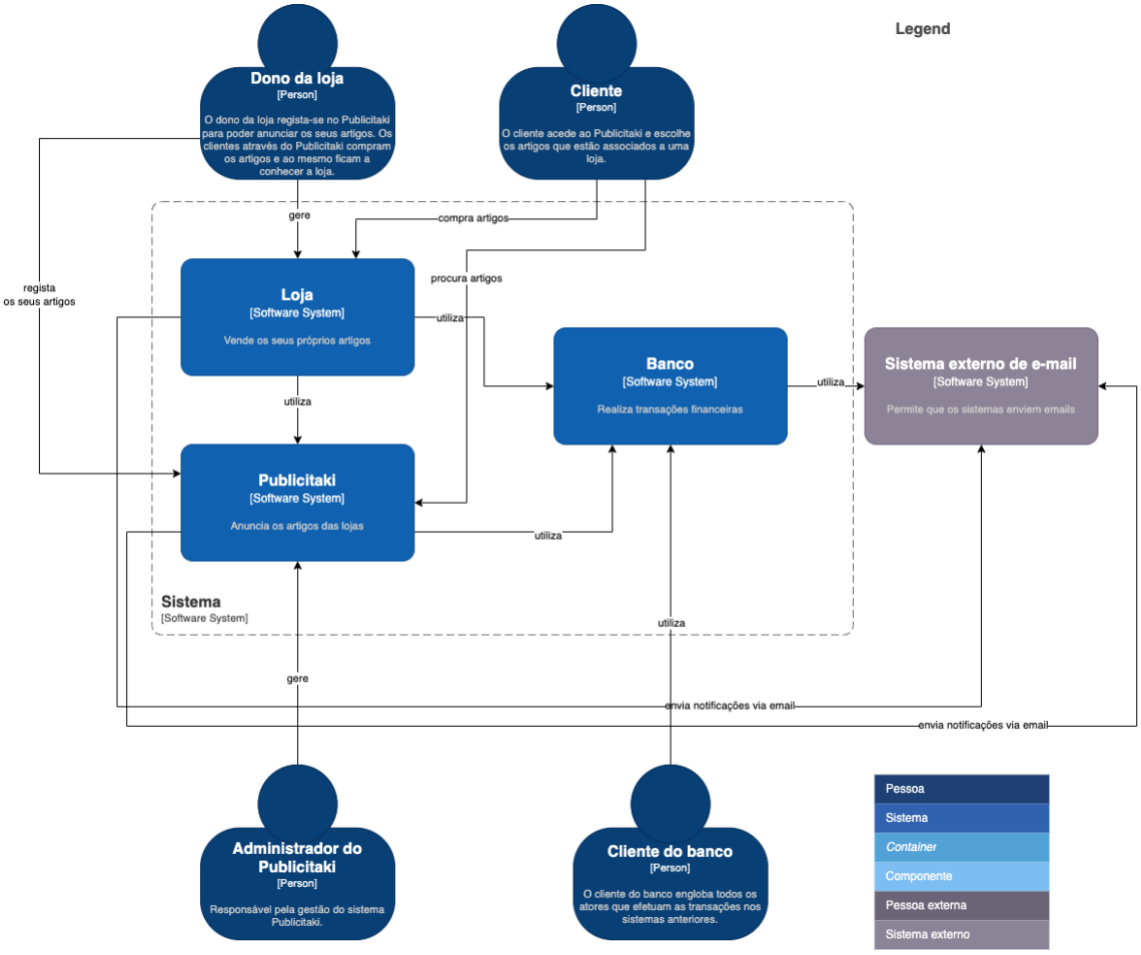


Figura 4-1: Diagrama de Contexto

A legenda, presente neste e em todos os diagramas, tem o propósito de identificar as cores e diferentes tonalidades inseridas nos diagramas. No presente diagrama pode-se observar a azul-escuro os atores, a cinzento-claro os sistemas externos e num azul intermédio o sistema de *software* a desenvolver.

A utilização do sistema externo de *e-mail*, nos três sistemas, foi fundamental para o envio de notificações de confirmação de registos, de promoções e de opiniões.

É importante salientar que todas as comunicações efetuadas entre os sistemas serão comunicações RESTful, exceto a comunicação entre os sistemas Publicitaki e Loja com o Banco, onde as comunicações serão efetuadas de forma assíncrona recorrendo ao modelo *publish-subscribe*, e a comunicação do microsserviço de gestão de registos de utilizadores da Loja que utiliza a comunicação gRPC.

4.1.2. Nível 2 – Diagrama de *Containers* (Banco)

O diagrama de *containers*, representado pela Figura 4-2, identifica o *container* que vai compor o sistema Banco e a sua fragmentação em três pequenos serviços independentes, (API Serviço Gestão Registos, API Serviço Gestão Transações e API Serviço Gestão Notificações), cada um com a sua própria Base de Dados. Podem, igualmente, ser observadas as interações do ator com o *container*, a interação deste como os três microsserviços, bem como a interação dos microsserviços entre si e a relação com o sistema externo.

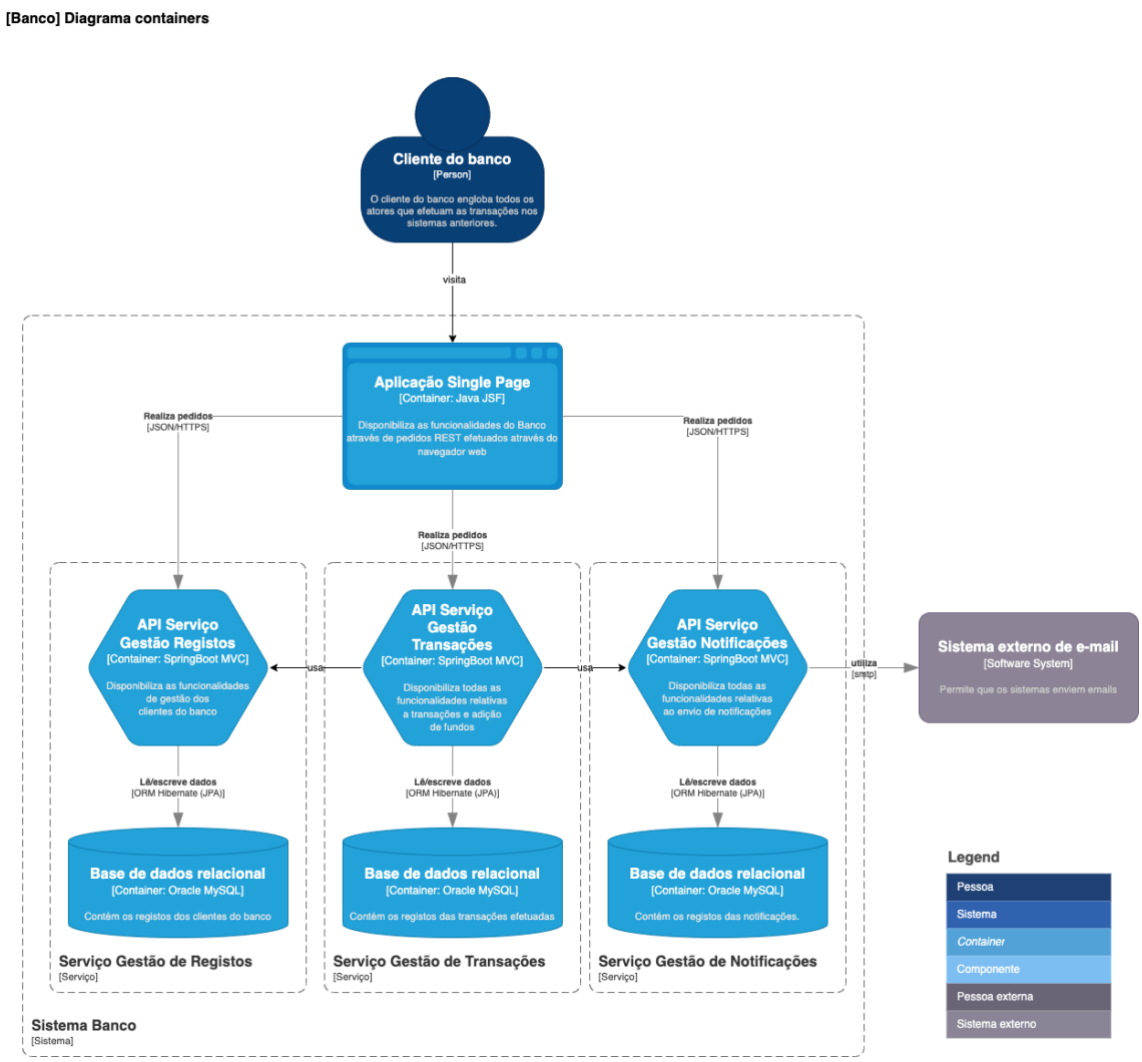


Figura 4-2: Diagrama de *containers* do sistema "Banco"

Uma análise mais detalhada ao diagrama permite verificar que o cliente acede às funcionalidades do sistema através do *container* **Aplicação *Single Page*** que disponibiliza essas mesmas funcionalidades através de pedidos REST efetuados através do navegador web. Seguidamente, e de acordo com o pedido do cliente, o *container* vai interagir com o microsserviço que se enquadra nesse mesmo pedido, para a realização, então, do pedido.

Para o envio de notificações o Serviço Gestão Notificações conta com o sistema externo de *e-mail* que se encarrega de enviar/receber os seus comunicados e os dos clientes (e.g. registos dos clientes).

Na **Tabela** 4-1 são apresentadas as especificidades de cada um dos microsserviços que compõe o *container* Aplicação *Single Page*:

Tabela 4-1: Descrição dos microsserviços presentes no sistema "Banco"

ID Microsserviço	Nome	Descrição
MS-01	API Serviço Gestão Registos	Disponibiliza as funcionalidades de gestão dos clientes do Banco
MS-02	API Serviço Gestão Transações	Disponibiliza todas as funcionalidades relativas a transações e adição de fundos
MS-03	API Serviço Gestão Notificações	Disponibiliza todas as funcionalidades relativas ao envio de notificações e para isso recorre a um sistema externo de *e-mail*.

4.1.3. Nível 2 – Diagrama de *Containers* (Loja)

O diagrama de *containers*, representado pela **Figura** 4-3, identifica o *container* que vai compor o sistema Loja e a sua fragmentação em quatro pequenos serviços independentes, (API Serviço Gestão Artigos, API Serviço Gestão Registos, API Serviço Gestão Pagamentos e API Serviço Gestão Notificações), cada um com a sua própria Base de Dados. Podem, igualmente, ser observadas as interações dos atores (Dono da loja e Cliente) com o *container*, a interação deste como os quatro microsserviços, bem como a interação dos microsserviços entre si e a relação destes com o sistema externo e com o sistema Banco.

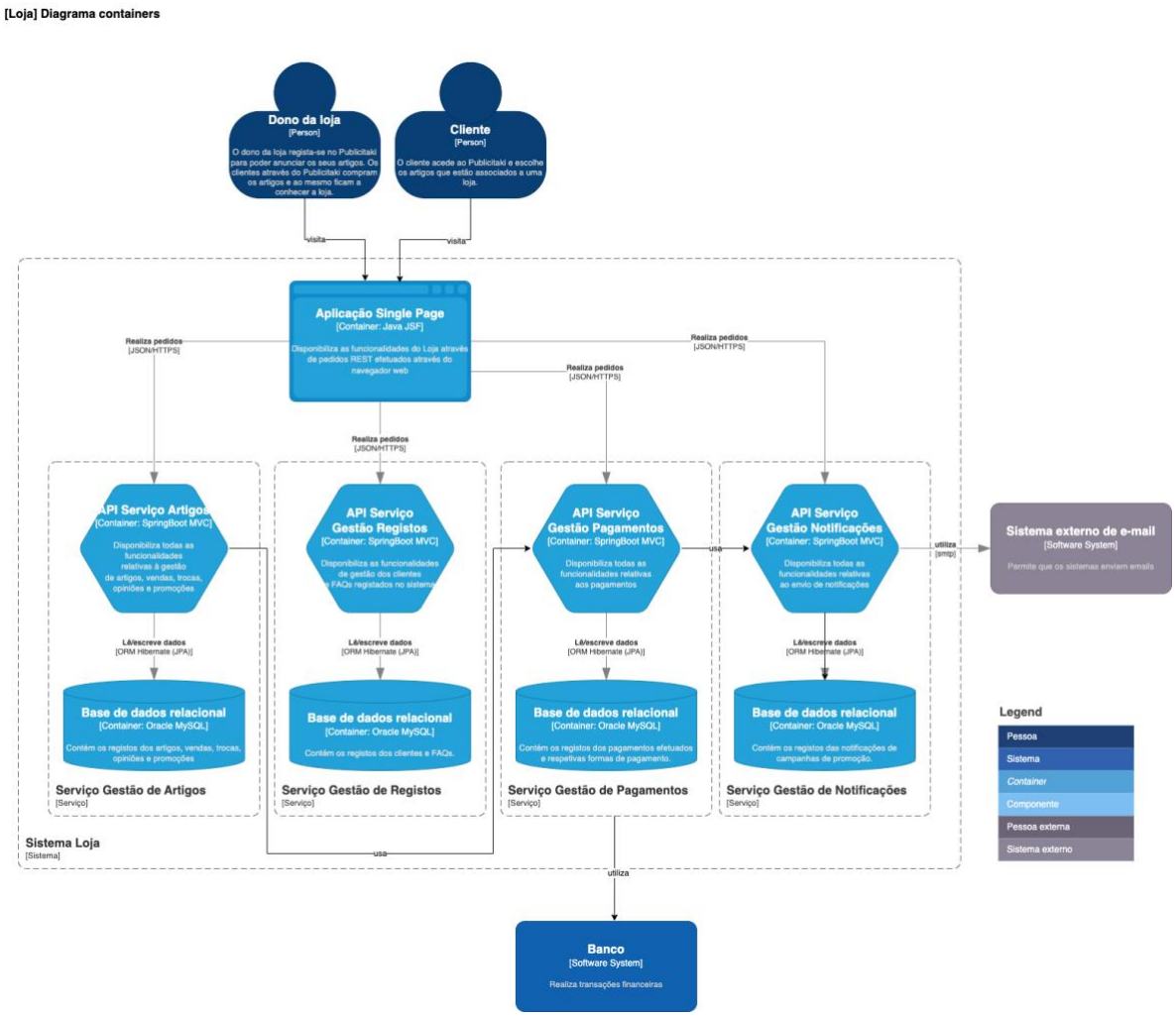


Figura 4-3: Diagrama de *containers* do sistema "Loja"

Uma análise mais detalhada ao diagrama permite verificar que o Dono da Loja e o Cliente acedem às funcionalidades do sistema através do *container* **Aplicação *Single Page*** que disponibiliza essas mesmas funcionalidades através de pedidos REST efetuados através do navegador web. Seguidamente, e de acordo com os pedidos do Dono da Loja e do Cliente, o container vai interagir com os microsserviços que se enquadram nesses mesmos pedidos, para a realização, então, dos pedidos.

O Serviço Gestão Artigos usa o Serviço Gestão Pagamentos para disponibilizar os pagamentos, que por sua vez utiliza o sistema Banco que processa os pedidos.

O Serviço Gestão Pagamentos usa o Serviço Gestão Notificações que conta com o sistema externo de *e-mail* que se encarrega de enviar/receber os seus comunicados e os dos clientes (e.g. registos dos Donos das Lojas e dos Clientes).

Na **Tabela** 4-2 são apresentadas as especificidades de cada um dos microsserviços que compõe o *container* Aplicação *Single Page*:

Tabela 4-2: Descrição dos microsserviços presentes no sistema "Loja"

ID Microsserviço	**Nome**	**Descrição**
MS-01	API Serviço Gestão Artigos	Disponibiliza todas as funcionalidades relativas à gestão de artigos, vendas, trocas, opiniões e promoções
MS-02	API Serviço Gestão Registos	Disponibiliza as funcionalidades de gestão dos clientes e FAQs registados no sistema
MS-03	API Serviço Gestão Pagamentos	Disponibiliza todas as funcionalidades relativas aos pagamentos
MS-04	API Serviço Gestão Notificações	Disponibiliza todas as funcionalidades relativas ao envio de notificações e para isso recorre a um sistema externo de *e-mail*.

4.1.4. Nível 2 – Diagrama de *Containers* (Publicitaki)

O diagrama de *containers*, acima representado pela **Figura** 4-4, identifica o *container* que vai compor o sistema Publicitaki e a sua fragmentação em quatro pequenos serviços independentes, (API Serviço Gestão Artigos, API Serviço Gestão Registos, API Serviço Gestão Pagamentos e API Serviço Gestão Notificações), cada um com a sua própria Base de Dados. Podem, igualmente, ser observadas as interações dos atores (Administrador do Publicitaki, Cliente e Dono da loja) com o *container*, a interação deste como os quatro microsserviços, bem como a interação dos microsserviços entre si e a relação destes com o sistema externo.

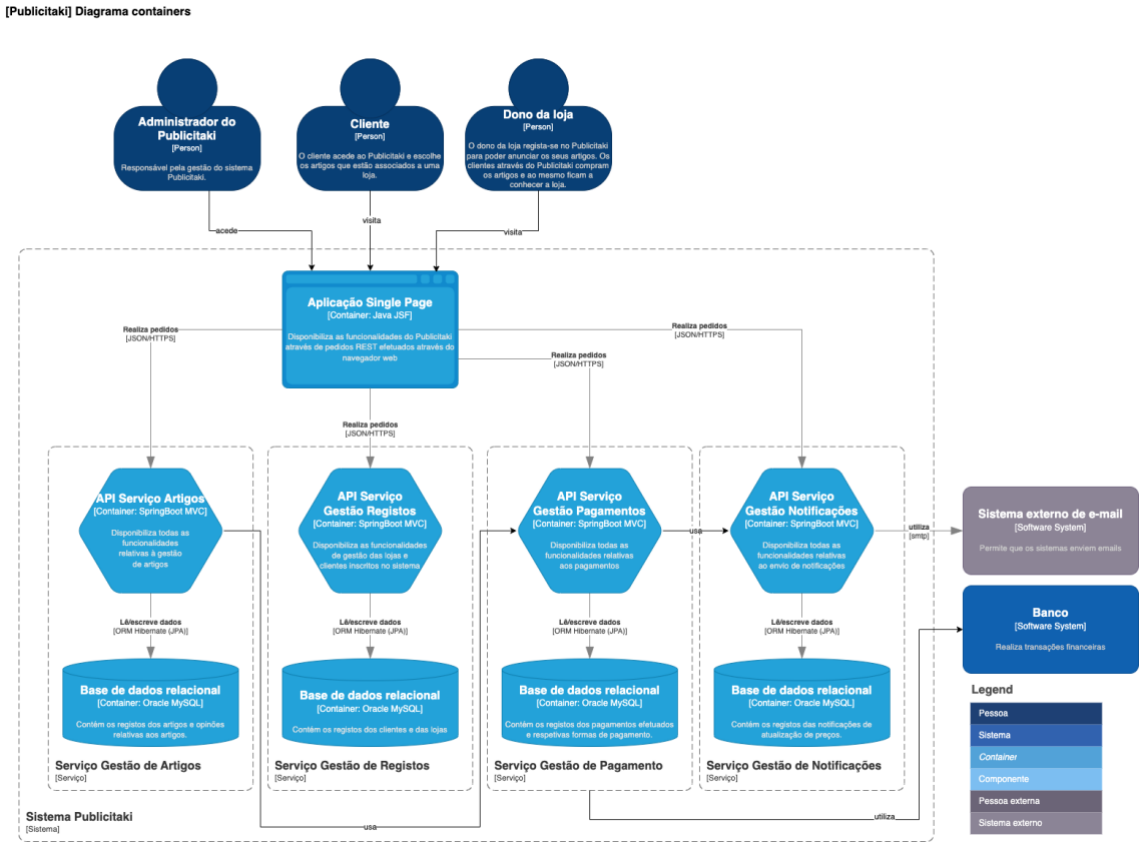


Figura 4-4: Diagrama de containers do sistema "Publicitaki"

Uma análise mais detalhada ao diagrama permite verificar que o Administrador do Publicitaki, Cliente e Dono da Loja acedem às funcionalidades do sistema através do *container* **Aplicação *Single Page*** que disponibiliza essas mesmas funcionalidades através de pedidos REST efetuados através do navegador web. Seguidamente, e de acordo com os pedidos do Administrador do Publicitaki, Cliente e Dono da Loja, o *container* vai interagir com os microsserviços que se enquadram nesses mesmos pedidos, para a realização, então, dos pedidos.

O Serviço Gestão Artigos usa o Serviço Gestão Pagamentos para disponibilizar os pagamentos, que por sua vez utiliza o sistema Banco que processa os pedidos.

O Serviço Gestão Pagamentos usa o Serviço Gestão Notificações que conta com o sistema externo de *e-mail* que se encarrega de enviar/receber os seus comunicados e os dos clientes (e.g. registos dos Donos das Lojas e dos Clientes).

Na **Tabela** 4-3 são apresentadas as especificidades de cada um dos microsserviços que compõe o *container* Aplicação *Single Page*:

Tabela 4-3: Descrição dos microsserviços presentes no sistema "Publicitaki"

ID Microsserviço	**Nome**	**Descrição**
MS-01	API Serviço Gestão Artigos	Disponibiliza todas as funcionalidades relativas à gestão de artigos.
MS-02	API Serviço Gestão Registos	Disponibiliza as funcionalidades de gestão das lojas e clientes inscritos no sistema.
MS-03	API Serviço Gestão Pagamentos	Disponibiliza todas as funcionalidades relativas aos pagamentos e para isso recorre ao sistema Banco.
MS-04	API Serviço Gestão Notificações	Disponibiliza todas as funcionalidades relativas ao envio de notificações e para isso recorre ao sistema externo de *e-mail.*

4.1.5. Nível 3 – Diagrama de Componentes (Banco)

No diagrama de componentes, representado pela **Figura** 4-5, são salientadas as especificidades do *container* **Aplicação *Single-Page*** e suas funcionalidades decompostas em três *controllers*. A cada *controller* é atribuído um componente independente que irá realizar o respetivo pedido.

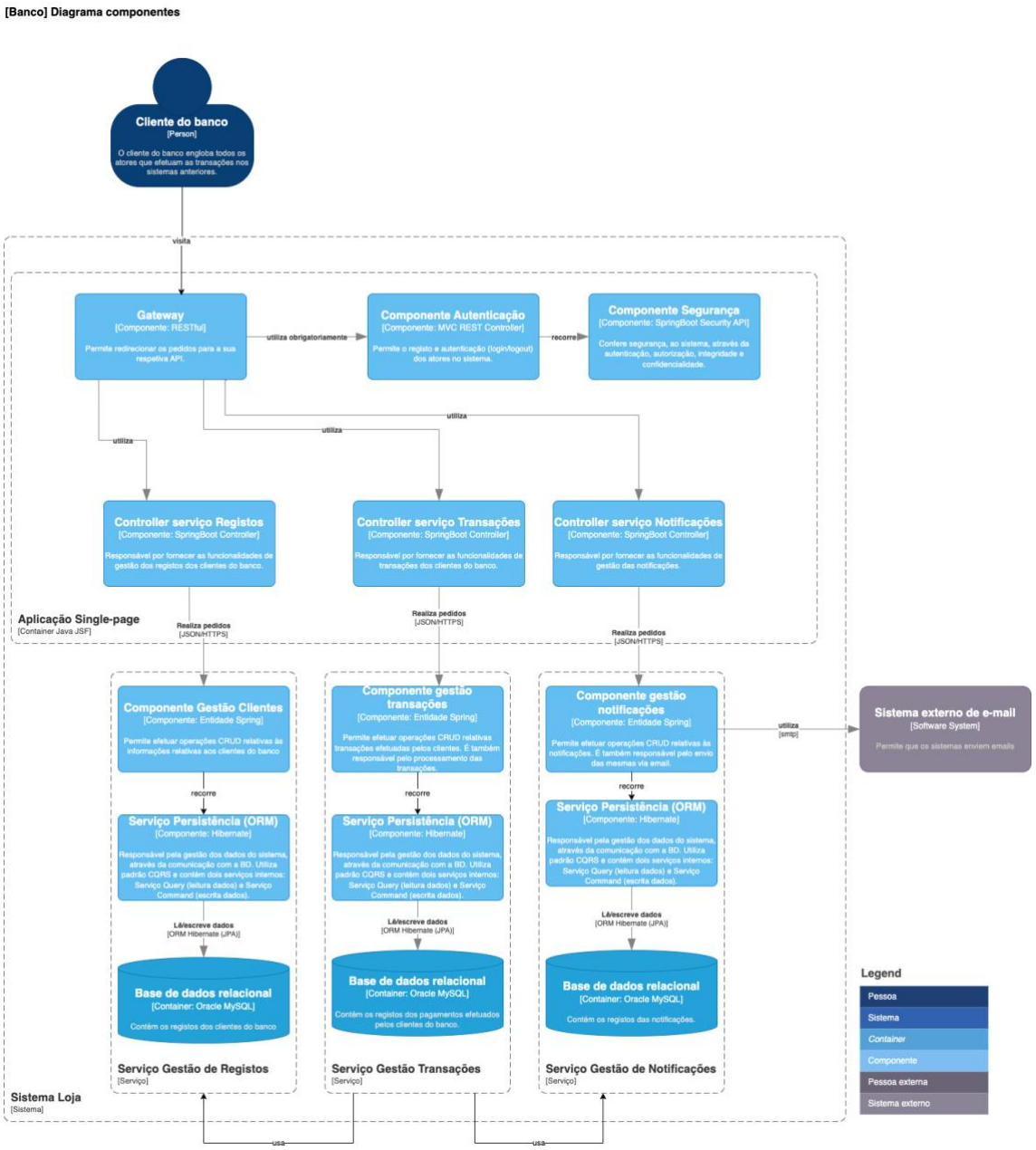


Figura 4-5: Diagrama de componentes do sistema "Banco"

Os pedidos HTTPS recolhidos pelo *container* **Aplicação *Single-Page*** são organizados pelo componente API Gateway (API RESTful) que vai servir de ponte de ligação com os componentes Autenticação e Segurança e com os *controllers* Serviço Registos, Serviço Transações e Serviço Notificações.

Deste modo, os pedidos dos clientes do Banco são encaminhados para o componente Autenticação, que os reencaminhará para o componente Segurança para uma avaliação desses mesmos pedidos, nomeadamente se o *token* é ou não válido. São, depois, encaminhados para os *controllers* Serviço Registos, Serviço Transações e Serviço Notificações que, através dos respetivos componentes, realizam os pedidos recorrendo aos Serviços Persistência (ORM), responsáveis pela gestão dos dados dos sistemas através da comunicação com a BD.

O *Controller* Serviço Registos realiza os pedidos através do componente Gestão Clientes que efetua as operações CRUD relativas às informações de registo dos clientes do Banco.

O *Controller* Serviço Transações realiza os pedidos através do componente Gestão Transações que efetua as operações CRUD relativas às transações realizadas pelos clientes.

O *Controller* Serviço Notificações realiza os pedidos através do componente Gestão Notificações que efetua as operações CRUD relativas às notificações enviadas via *e-mail* aos clientes. Para o envio dos *e-mails* este componente necessita de recorrer ao sistema externo de *e-mails*.

Na **Tabela** 4-4 são apresentadas as especificidades de cada um dos componentes que compõe o diagrama.

Tabela 4-4: Descrição dos componentes presentes no sistema "Banco"

ID Componente	Nome	Descrição
CP-01	API Gateway	Organiza os pedidos dos clientes e serve de ponte de ligação com os componentes Autenticação e Segurança e com os *Controllers* Serviço Registos, Serviço Transações e Serviço Notificações.
CP-02	Componente Autenticação	Processa os registos e autenticações dos clientes. No processo de autenticação recorre ao componente Segurança para assegurar a validade dos *tokens* dos utilizadores e limite máximo de tentativas de introdução das credenciais.
CP-03	Componente Segurança	Garante a segurança das operações executadas no sistema (validade dos *tokens* e número de tentativas de autenticação).
CP-04	*Controller* Serviço Registos	Fornece as funcionalidades de gestão dos registos dos clientes do Banco.
CP-05	*Controller* Serviço Transações	Fornece as funcionalidades de gestão das transações dos clientes do Banco.
CP-06	*Controller* Serviço Notificações	Fornece as funcionalidades de gestão das notificações. O componente Gestão de Notificações que realiza os pedidos necessita de recorrer a um serviço externo de envio de *e-mails*.

4.1.6. Nível 3 – Diagrama de Componentes (Loja)

No diagrama de componentes, representado pela **Figura** 4-6, são salientadas as especificidades do *container* **Aplicação *Single-Page*** e suas funcionalidades decompostas em quatro *Controllers* que, através de componentes independentes, irão realizar os respetivos pedidos.

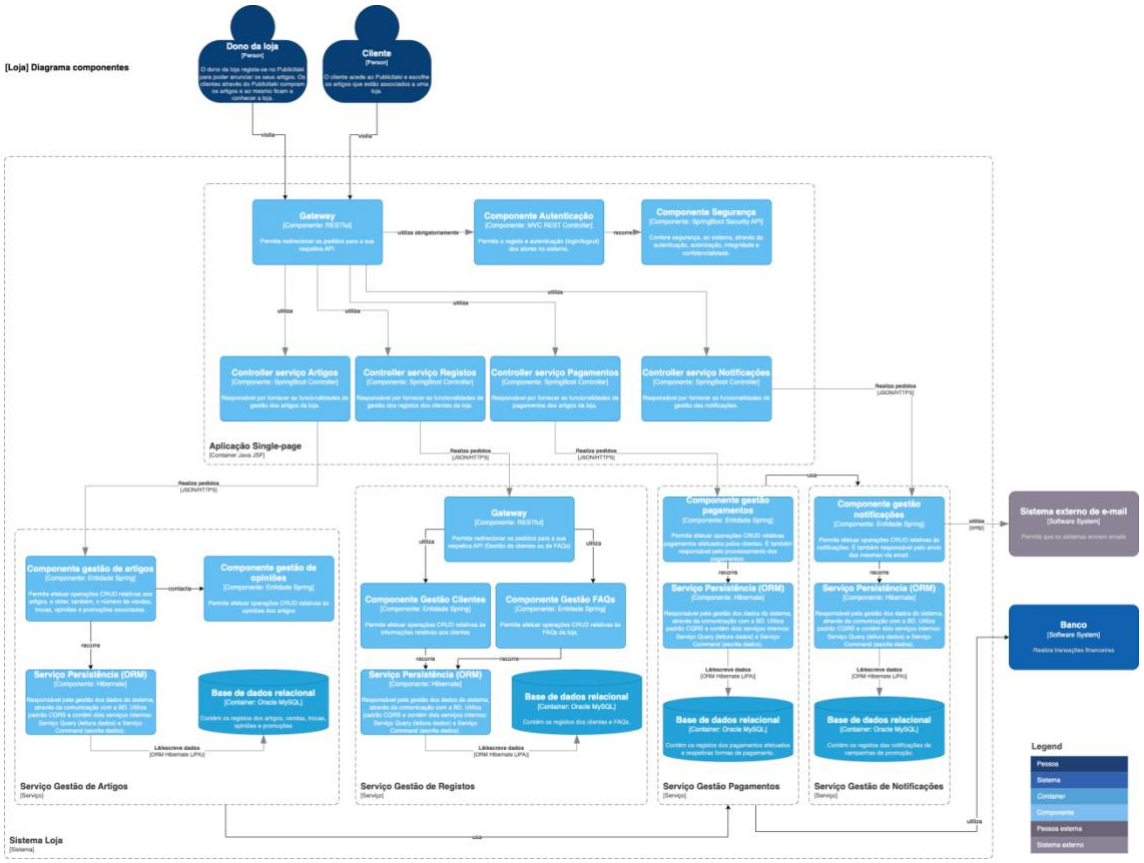


Figura 4-6: Diagrama de componentes do sistema "Loja"

Os pedidos HTTPS recolhidos pelo *container* **Aplicação *Single-Page*** são organizados pelo componente API Gateway (API RESTful) que vai servir de ponte de ligação com os componentes Autenticação e Segurança e com os *controllers* Serviço Artigos, Serviço Registos, Serviço Pagamentos e Serviço Notificações.

Deste modo, os pedidos dos clientes da Loja são encaminhados para o componente Autenticação, que os reencaminhará para o componente Segurança para uma avaliação desses mesmos pedidos, nomeadamente se o *token* é ou não válido. São, depois, encaminhados para os *controllers* Serviço Artigos, Serviço Registos, Serviço Pagamentos e Serviço Notificações que, através dos respetivos componentes, realizam os pedidos, recorrendo aos Serviços Persistência (ORM), responsáveis pela gestão dos dados dos sistemas através da comunicação com a BD.

O ***Controller* Serviço Artigos** é fragmentado em dois componentes, o componente Gestão de Artigos e o componente Gestão de Opiniões.

O componente Gestão de Artigos é quem efetua as operações CRUD relativas aos artigos e obtém o número de vendas, trocas, opiniões e promoções. Para as

operações CRUD relativas às opiniões dos artigos o componente Gestão de Artigos recorre ao componente Gestão de Opiniões.

O ***controller*** **Serviço Registos** é fragmentado em dois componentes, o componente Gestão Clientes e o componente Gestão FAQs. O *controller* Serviço Registos realiza os pedidos através do Gateway que vai servir de elo de ligação com os componentes Gestão Clientes e Gestão de FAQs da loja.

O componente Gestão Clientes é quem efetua as operações CRUD relativas às informações dos clientes.

O componente Gestão FAQs é quem efetua as operações CRUD relativas às FAQs da Loja.

O ***Controller*** **Serviço Pagamentos** realiza os pedidos através do componente Gestão Pagamentos que efetua as operações CRUD relativas aos pagamentos realizados pelos clientes.

O ***Controller*** **Serviço Notificações** realiza os pedidos através do componente Gestão Notificações que efetua as operações CRUD relativas às notificações enviadas via *e-mail* aos clientes. Para o envio dos *e-mails* este componente necessita de recorrer ao sistema externo de *e-mails*.

Para as operações CRUD relativas às notificações, o componente Gestão Pagamentos recorre ao componente Gestão Notificações.

O Serviço Gestão Artigos usa o Serviço Gestão Pagamentos para efetuar os pagamentos, que por sua vez usa o sistema Banco que realiza as transações financeiras.

Na **Tabela** 4-5 são apresentadas as especificidades de cada um dos componentes que compõe o diagrama:

Tabela 4-5: Descrição dos componentes presentes no sistema "Loja"

ID Componente	Nome	Descrição
CP-01	API Gateway	Organiza os pedidos dos clientes e serve de ponte de ligação com os componentes Autenticação e Segurança e com os *Controllers* Serviço Artigos, Serviço Registos, Serviço Pagamentos e Serviço Notificações.
CP-02	Componente Autenticação	Processa os registos e autenticações dos clientes. No processo de autenticação recorre ao componente Segurança para assegurar a validade dos *tokens* dos utilizadores e limite máximo de tentativas de introdução das credenciais.
CP-03	Componente Segurança	Garante a segurança das operações executadas no sistema (validade dos *tokens* e número de tentativas de autenticação).
CP-04	*Controller* Serviço Artigos	Fornece as funcionalidades de gestão dos artigos da Loja.
CP-05	*Controller* Serviço Registos	Fornece as funcionalidades de gestão dos registos dos clientes da Loja.
CP-06	*Controller* Serviço Pagamentos	Fornece as funcionalidades de gestão dos pagamentos realizados pelos clientes da Loja e para isso recorre ao sistema Banco.
CP-07	*Controller* Serviço Notificações	Fornece as funcionalidades de gestão das notificações. O componente Gestão de Notificações que realiza os pedidos necessita de recorrer a um serviço externo de envio de *e-mails.*

4.1.7. Nível 3 – Diagrama de Componentes (Publicitaki)

No diagrama de componentes, acima representado, são salientadas as especificidades do *container* **Aplicação *Single-Page*** e suas funcionalidades decompostas em quatro *controllers* que, através de componentes independentes, irão realizar os respetivos pedidos.

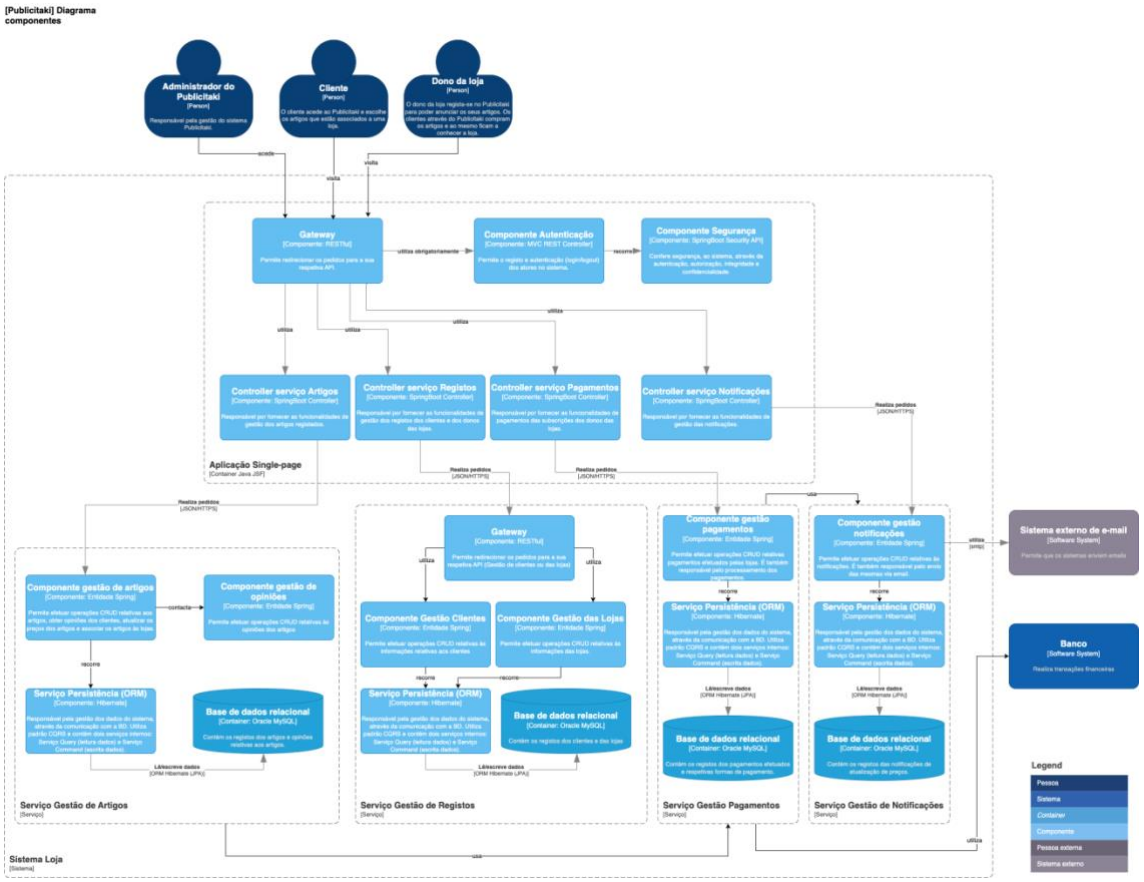


Figura 4-7: Diagrama de componentes do sistema "Publicitaki"

Os pedidos HTTPS recolhidos pelo *container* **Aplicação *Single-Page*** são organizados pelo componente API Gateway (API RESTfull) que vai servir de ponte de ligação com os componentes Autenticação e Segurança e com os *controllers* Serviço Artigos, Serviço Registos, Serviço Pagamentos e Serviço Notificações. O Gateway tem, igualmente, a responsabilidade de processar os pedidos de transferências bancárias (pedidos efetuados pelos sistemas Loja e Publicitaki de forma automática) que são rececionados no tópico Kafka.

Deste modo, os pedidos dos clientes do Publicitaki são encaminhados para o componente Autenticação, que os reencaminhará para o componente Segurança para uma avaliação desses mesmos pedidos, nomeadamente se o *token* é ou não válido. São, depois, encaminhados para os *controllers* Serviço Artigos, Serviço Registos, Serviço Pagamentos e Serviço Notificações que, através dos respetivos componentes, realizam os pedidos, recorrendo aos Serviços Persistência (ORM), responsáveis pela gestão dos dados dos sistemas através da comunicação com a BD.

O ***controller* Serviço Artigos** é fragmentado em dois componentes, o componente Gestão de Artigos e o componente Gestão de Opiniões.

O componente Gestão de Artigos é quem efetua as operações CRUD relativas aos artigos, obtém as opiniões dos clientes, atualiza os preços e associa os artigos às lojas. Para as operações CRUD relativas às opiniões dos artigos o componente Gestão de Artigos contacta o componente Gestão de Opiniões.

O ***controller* Serviço Registos** é fragmentado em dois componentes, o componente Gestão Clientes e o componente Gestão das Lojas. O *controller* Serviço Registos realiza os pedidos através do Gateway que vai servir de elo de ligação com os componentes Gestão Clientes e Gestão das lojas.

O componente Gestão Clientes efetua as operações CRUD relativas às informações dos clientes e o componente Gestão das Lojas efetua as operações CRUD relativas às informações das lojas.

O ***controller* Serviço Pagamentos** realiza os pedidos através do componente Gestão Pagamentos que efetua as operações CRUD relativas aos pagamentos realizados pelos clientes.

O ***controller* Serviço Notificações** realiza os pedidos através do componente Gestão Notificações que efetua as operações CRUD relativas às notificações enviadas via *e-mail* aos clientes. Para o envio dos *e-mails* este componente necessita de recorrer ao sistema externo de *e-mails*.

Para as operações CRUD relativas às notificações o componente Gestão Pagamentos recorre ao componente Gestão Notificações.

O Serviço Gestão Artigos usa o Serviço Gestão Pagamentos que por sua vez usa o sistema Banco que realiza as transações financeiras.

Na **Tabela** 4-6 são apresentadas as especificidades de cada um dos componentes que compõe o diagrama:

Tabela 4-6: Descrição dos componentes presentes no sistema "Publicitaki"

ID Componente	Nome	Descrição
CP-01	API Gateway	Organiza os pedidos dos clientes e serve de ponte de ligação com os componentes Autenticação e Segurança e com os *controllers* Serviço Artigos, Serviço Registos, Serviço Pagamentos e Serviço Notificações.
CP-02	Componente Autenticação	Processa os registos e autenticações dos clientes. No processo de autenticação recorre ao componente Segurança para assegurar a validade dos *tokens* dos utilizadores e limite máximo de tentativas de introdução das credenciais.
CP-03	Componente Segurança	Garante a segurança das operações executadas no sistema (validade dos *tokens* e número de tentativas de autenticação).
CP-04	Controller Serviço Artigos	Fornece as funcionalidades de gestão dos artigos do Publicitaki.
CP-05	*Controller* Serviço Registos	Fornece as funcionalidades de gestão dos registos dos clientes.
CP-06	*Controller* Serviço Pagamentos	Fornece as funcionalidades de gestão dos pagamentos realizados pelos clientes e para isso recorre ao sistema Banco.
CP-07	*Controller* Serviço Notificações	Fornece as funcionalidades de gestão das notificações. O componente Gestão de Notificações que realiza os pedidos necessita de recorrer a um serviço externo de envio de *e-mails*.

4.2 Opções tecnológicas

Depois de desenhada a arquitetura do sistema, é, então, possível prosseguir para a decisão das tecnologias a utilizar.

4.2.1. *Frameworks web*

Ao analisar a arquitetura desenhada e as tecnologias disponíveis para o desenvolvimento de aplicações Web (**subsecção 2.3 - *Framework Web***) foi, então, possível escolher as que mais se adequavam ao projeto.

Desta forma, determinaram-se quais as *frameworks* que mais se adequam às camadas de *backend* e *frontend* das aplicações *web*. Baseado nesse estudo, foram escolhidas as seguintes *frameworks* e linguagens de programação:

- ***Backend***:
    - Spring Boot (Java)
    - Quarkus (Java)
    - Django (Python)
    - Node.js (Typescript)
- ***Frontend***:
    - ASP.NET Core MVC (C#)
    - React + Vite (Typescript)

A escolha destas tecnologias sustentou-se na intenção de utilizar as *frameworks* mais populares e ativamente adotadas pela indústria. Foram escolhidas uma vez que se pretende demonstrar várias aplicações, escritas em linguagens de programação e *frameworks* distintas, a comunicar entre si, simulando, assim, o comportamento real de um sistema constituído por vários microsserviços.

Em suma esta tecnologias foram selecionadas dado o seu baixo nível de complexidade, e elevada popularidade e apoio documental.

4.2.2. Sistema de Gestão de Base de dados

Depois de um meticuloso e aprofundado estudo efetuado das soluções disponíveis (**subsecção 2.5 - Comunicação entre a lógica e o SGBD**), optou-se pelos SGBD, **MySQL** e **PostgreSQL**, dadas as suas irrefutáveis popularidades e baixo consumo de recursos.

4.2.3. Comunicação entre o SGBD e a lógica

Foi, igualmente, decidido que a comunicação entre o SGBD e as camadas de lógica, na Framework Spring Boot, será efetuada recorrendo à **API JPA** com a utilização da implementação **Hibernate**. Para a *framework* .NET Core a escolha recaiu no uso da ORM **Entity Framework**, e para a *framework* Node.js foi selecionada a ORM Sequelize.

4.2.4. Protocolos de comunicação

Como referido no início do capítulo, foi selecionado o estilo arquitetural **RESTful** para a maioria das comunicações efetuadas pelos microsserviços integrantes dos sistemas. No entanto, as comunicações entre os sistemas Publicitaki e Loja com o Banco serão realizadas de **forma assíncrona**, utilizando o modelo *publish-subscribe*. Para tal, foi selecionado o protocolo **MQTT** devido à sua simplicidade, uma vez que os sistemas a desenvolver são apenas uma prova de conceito para fins científicos, não sendo necessário utilizar um protocolo mais complexo como o Kafka. Também foi utilizada a *framework* **gRPC** no microsserviço do sistema Loja.

4.3 Tecnologias e ferramentas

Na Tabela 4-7: Tecnologias e ferramentas utilizadas encontram-se discriminadas as ferramentas que serão utilizadas para a implementação da solução proposta.

Tabela 4-7: Tecnologias e ferramentas utilizadas

	Tecnologia ou ferramenta	**Versão**	**Descrição**
Sistemas de gestão de bases de dados (SGBD)	MySQL	8.0.31	Base de dados
	PostgreSQL	15.1	Base de dados
	MySQL WorkBench	8.0.31	Sistema de design e gestão de bases de dados MySQL
Ambientes de desenvolvimento (IDE)	Oracle NetBeans	12.5	IDE utilizado durante das modelos de negócio que utilizarão a linguagem Java e a *Framework* Spring Boot
	Visual Studio	2022	IDE utilizado durante das modelos de negócio que utilizarão a linguagem C# e a *Framework* .NET Core
***Framework* e máquinas virtuais**	Spring Boot	3.0.1	Framework w*eb* utilizada no desenvolvimento das aplicações Spring Boot
	Quarkus	2.1.19	Framework w*eb* utilizada no desenvolvimento das aplicações Quarkus
	.NET Framework	4.8.1	Framework *web* utilizada no desenvolvimento das aplicações ASP.NET
	Java JDK	(Oracle) 1.8.0_351-b10	Máquina virtual Java
	Python / Django	3.11 / 4.1	Aplicações Django
	Node.js	18	Aplicações Node.js
Gestão e desenvolvimento das aplicações	Docker	20.10.13	Plataforma de desenvolvimento de aplicações

Capítulo 5
Implementação da aplicação

Definidos os requisitos, arquitetura e tecnologias foi, então, possível avançar para a fase da implementação dos três sistemas.

Este capítulo abordará os tópicos relativos à fase de deployment da aplicação, os testes de carga realizados, e os resultados obtidos na avaliação heurística e na *software-review*.

5.1 *Deployment*

Conjuntamente com os orientadores, foi deliberada a criação de criar duas versões de *deployment* para os três sistemas, designadamente a versão de *deployment* para Docker e para Amazon AWS (como IaaS).

O propósito da utilização de duas versões de *deployment* e de não apenas uma (somente a versão AWS), prende-se mormente com os seguintes fatores:

- **Portabilidade**: a existência de uma versão Docker permite, não só, a fácil alteração do sistema operativo do container, mas também garante uma maior flexibilidade da alteração da plataforma de *deployment* na *Cloud*, como a migração de Amazon AWS para Microsoft Azure.
- **Testagem**: a versão Docker permite uma maior facilidade no processo de testagem, uma vez que a correção de defeitos poderá ser realizada localmente evitando, desta forma, delongas e recursos no seu *re-deployment*. Esta abordagem permite, igualmente, quando necessário, a fácil alteração das versões de dependências, como o caso da Amazon AWS, em que variadas vezes, não dispõe de versões atualizadas das suas tecnologias, como a *framework* ASP. NET Core, que, à data da redação da presente tese (maio de 2023), permanecia na versão seis enquanto era ultrapassada pela versão sete já lançada pela Microsoft em novembro de 2022.
- **Custos**: a versão Docker permite evitar gastos adicionais quando é necessária a correção de código, visto que, qualquer *re-eployment* na Amazon AWS, aumenta o número de pedidos registados (tipos *Amazon Simple Storage Service*, *Amazon Relational Database Service*, entre outros) que aumenta a fatura mensal final.

Consequentemente, qualquer modificação do código e posterior testagem deverá ser sempre realizada no ambiente Docker, e quando finalizado, ser então feito o seu *deployment* para a Amazon AWS, evitando assim custos e tempo desnecessários.

5.1.1. Versão Docker

A versão Docker de *deployment* dos três sistemas, foi estruturada ou decomposta em três pastas denominadas *Publicitaki*, Loja e Banco, representativas dos respetivos sistemas. Cada sistema contém na sua pasta, não só, as subpastas com o código dos seus respetivos microsserviços internos, mas também um ficheiro "docker-compose" que permite a execução destes mesmos microsserviços. A **Figura** 5-1 ilustra, de forma sintetizada, a estrutura de *deployment* dos três sistemas da versão Docker.

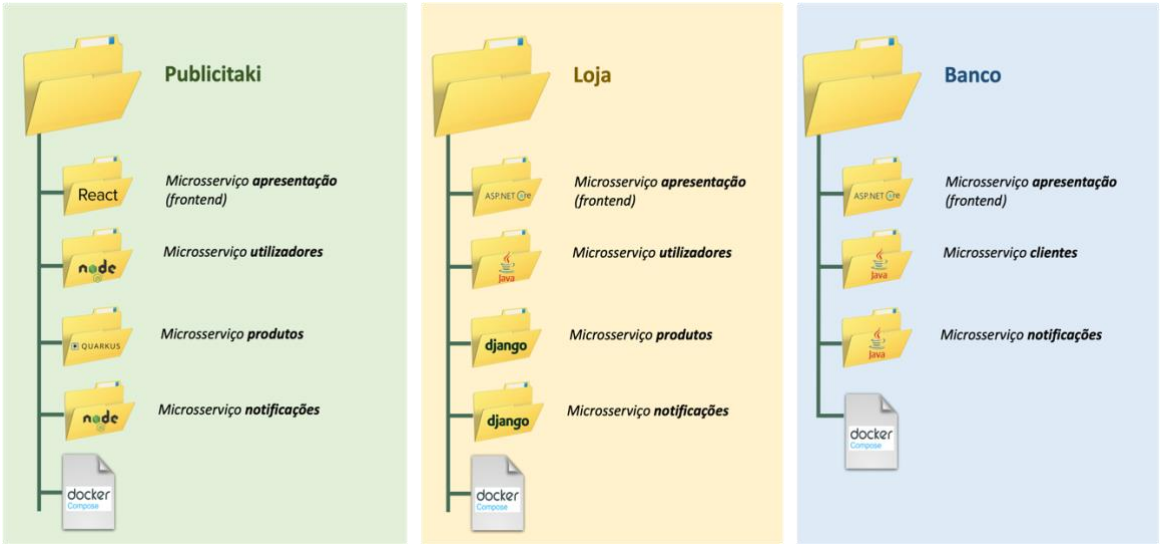


Figura 5-1: Estrutura do deployment da versão Docker

Funcionamento

Para colocar em funcionamento a versão Docker, o utilizador deverá instalar na sua máquina o *software* Docker Desktop a partir do *website* https://www.docker.com/products/docker-desktop. Seguidamente, deverá entrar em cada uma das pastas representativas dos sistemas e iniciá-los, recorrendo ao seguinte comando:

```
docker-compose up -d
```

Código 1 - Docker compose up no modo *detached*

Ao executar o comando anterior, o Docker iniciará os microsserviços necessários para iniciar o respetivo sistema, ficando cada microsserviço exposto à máquina local (*localhost*) através de encaminhamento de portos (*port-forwarding*). Para aceder à camada de apresentação do respetivo sistema, o utilizador deverá aceder ao microsserviço denominado por "apresentação/*frontend*" através do seu navegador (*browser*), no URL apresentada pelo Código 2:

```
localhost:<porto-encaminhado-do-microsserviço-frontend>
```

Código 2 - Hiperligação do *website* do sistema

Depois de efetuada a iniciação dos microsserviços, é possível aceder à sua página web (através do URL do *frontend*) para interagir com o sistema, efetuar testes à API

através dos seus *endpoints* (recorrendo às plataformas Postman ou Insomnia), testes de carga, injeção de falhas, entre outras operações.

Por **convenção** foi decidido **nomear** os containers dos três sistemas da seguinte forma:

- ***Containers* com aplicações:** *<nome_sistema-nome_aplicação>*, por exemplo **"pub-users"** representa o *container* responsável pela **gestão de utilizadores** do sistema **Publicitaki**;
- ***Containers* com bases de dados:** *<nome_sistema-nome_aplicação-**db**>*, por exemplo **"pub-users-db"** representa o *container* que contém a base de dados dos utilizadores e que comunica pelo *container* **"pub-users"** do sistema **Publicitaki**.

Docker-compose do Publicitaki

Os ficheiros *docker-compose* permitem ao programador uma maior facilidade na definição e controlo da sua aplicação constituída por diversos *containers*. Neste ficheiro definem-se os serviços, redes e volumes necessários para executar e interligar os *containers* constituintes da aplicação.

A **Figura** 5-2 ilustra um excerto do código do ficheiro docker-compose utilizado na construção do sistema Publicitaki.

```
version: '3'
services:

  #---- NOTIFICATIONS SERVICE [NodeJS] ----
  pub-notifications-db:
    image: mongo:latest
    command: mongod --quiet --logpath /dev/null
    container_name: pub_notifications_db
    restart: always
    ports:
      - "7000:27017"
    volumes:
      - ./pub-notifications-db/data:/data/db
    environment:
      - MONGO_INITDB_DATABASE=notifications-db

  pub-notifications:
    build: ./pub-notifications
    container_name: pub_notifications
    depends_on:
      - pub-notifications-db
    restart: on-failure
    ports:
      - "7001:3000"
    environment:
      - MONGODB_URI=mongodb://pub-notifications-db:27017/notification-db

  # ---- USERS SERVICE [NodeJS] ----
  pub-users-db:
    image: postgres:15.2-alpine
    container_name: pub_users_db
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      POSTGRES_DB: users-db
      POSTGRES_SSLMODE: disable
    ports:
      - "7002:5432"
    volumes:
      - ./pub-users-db/postgresql.conf:/etc/postgresql.conf
      - ./pub-users-db/pg-data:/var/lib/postgresql/data

  pub-users:
    build: ./pub-users
    container_name: pub_users
```

Figura 5-2: Excerto do ficheiro Docker-compose do sistema Publicitaki

A **Figura** 5-2 esboça a constituição do sistema Publicitaki suportado pelos microsserviços *pub-notifications-db* (correspondente à base de dados do microsserviço responsável por gerir as notificações), *pub-notifications* (microsserviço de gestão de notificações), *pub-users-db* (base de dados do microsserviço de gestão os utilizadores), e, parcialmente, o microsserviço *pub-users* (microsserviço de gestão os utilizadores). Ainda que não presentes na figura foram igualmente incluídos no sistema os microsserviços *pub-products*, *pub-products-db* e *pub-frontend*. O ficheiro foi escrito de acordo com a **versão 3 do Docker-compose**.

Seguidamente, encontra-se descrito, com maior detalhe, o código dos serviços presentes no ficheiro Docker-compose do **Publicitaki**, representados parcialmente pela **Figura** 5-2:

- **Pub-notifications-db**: *container* denominado por *pub_notifications_db* (para uma mais fácil identificação na rede), baseado na **imagem** com a versão mais recente do **SGBD Mongo**, configurado para que sempre que ocorra um erro volte a reiniciar, possui, ainda, um volume dedicado para persistir a informação contida na base de dados e expõe o seu porto 27017, através de *port-forwarding*, para o ***localhost*** no **porto 7000**;
- **Pub-notifications**: *container* denominado por *pub_notifications*, contém uma aplicação Node.js e é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. Este ficheiro Dockerfile define a utilização de uma imagem baseada na versão **Alpine do Node 18**, instala as dependências através do comando *npm*, copia o código da aplicação para o *container*, e expõe o porto 3000. O *container* expõe, igualmente, o porto 3000 através de *port-forwarding* para o ***localhost*** no **porto 7001**. A variável de ambiente **MONGODB_URI** permite a comunicação com o *container* que contém a base de dados deste microsserviço no *endpoint* ***mongodb://pub-notifications-db:27017/notification-db***;
- **Pub-users-db**: *container* denominado por *pub_users_db*, baseado na **imagem** com a versão 15.2 do **SGBD PostgreSQL**. Através de variáveis de ambiente, com os nomes reservados *POSTGRES*, é possível definir o nome da base de dados, utilizador, palavra-passe e ativar ou desativar o modo SSL. O *container* conta com um volume dedicada para persistir a informação contida na base de dados e expõe o seu porto 5432, através de *port-forwarding*, para o ***localhost*** no **porto 7002**;
- **Pub-users**: *container* denominado por *pub_users* contém uma aplicação Node.js que é construída a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. Este ficheiro Dockerfile define a utilização de uma imagem baseada na versão **Alpine do Node 18**, instala as dependências através do comando *npm*, copia código da aplicação para o *container*, e expõe o porto 3000. O *container* encontra-se configurado para sempre que ocorra uma avaria volte a reiniciar. O *container* expõe, igualmente, o porto 3000 através de *port-forwarding* para o ***localhost*** no **porto 7003**. A variável de ambiente **POSTGRES_URI** permite a comunicação com privilégios de administrador, ao *container* "pub-users-db" que contém a base de dados deste microsserviço no *endpoint* ***postgres://admin:admin@pub-users-db:5432/users-db***;

- **Pub-products-db**: *container* denominado por *pub_products_db*, baseado na **imagem** com a versão 8.0.32 do **SGBD MySQL**. Através de variáveis de ambiente, com os nomes reservados *MYSQL*, é possível definir o nome da base de dados, utilizador *root* e palavra-passe. O *container* possui um volume destinado a persistir a informação contida na base de dados e expõe o porto 3306 do MySQL, através de *port-forwarding*, para o ***localhost*** no **porto 7004**;
- **Pub-products**: *container* denominado por *pub_products*, contém uma aplicação **Quarkus**, e é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. O Dockerfile utilizado neste microsserviço é fornecido pela Redhat em projetos **Quarkus** e define a utilização de uma imagem baseada na **versão 8** do ***Red Hat Universal Base Image*** **(UBI)** que utiliza a **versão 11** do **OpenJDK**, copia o código para o *container*, expõe o porto 8080 e utiliza variáveis de ambiente que permitem a correta iniciação do *container*. O *container* expõe o porto 8080 através de *port-forwarding* para o ***localhost*** no **porto 7005**. Foi criado um volume para permitir a persistência das dependências do Maven evitando assim voltar a descarregá-las da Internet;
- **Pub-frontend**: *container* denominado por *pub_frontend* é **responsável pela componente de interação do utilizador com a aplicação**, construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. O Dockerfile utilizado neste microsserviço baseia-se numa imagem com a **versão 18** do **Node.js**, copia código o *container*, expõe o porto 3000 e instala as dependências do Node através do comando *npm*. O *container* contém uma aplicação **React + Vite** que expõe o porto 3000 através de *port-forwarding* para o ***localhost*** no **porto 7007**.

Os **ficheiros docker-compose** dos sistemas **Loja** e **Banco** foram, igualmente, descritos e encontram-se detalhados no **Anexo J:**.

Inserção de dados

Após a inicialização dos três sistemas, é essencial fornecer-lhes conjuntos de dados para viabilizar uma testagem mais adequada dos mesmos. Desta forma, é proposta a realização dos seguintes procedimentos:

- **Registar contas no Banco**:
    - **Método 1:** aceder a um cliente REST como o Insomnia ou o Postman (ou através de um comando *curl*). Realizar um pedido REST do tipo POST ao endereço http://localhost:6001/accounts, fornecendo um objeto JSON com os dados de registo de uma nova conta, que incluem, o balanço inicial (*balance*), nome (*holderName*), email (*holderEmail*), e *flag* de validação (*valid*), que indica se a conta está ou não ativa. Caso o pedido seja aceite, deverá devolver uma resposta 201 (*created*) "*A new account for {holderName} was opened on the bank*".
    - **Método 2:** aceder à página principal do Banco no endereço http://localhost:6003 (*endpoint* da camada de apresentação do

sistema Banco). Posteriormente, na barra de navegação superior, selecionar a opção *Contas* (caso seja apresentada a página de autenticação, introduzir o nome *admin* e a palavra-passe *admin*). Seguidamente clicar no botão de criação de uma nova conta, preencher os dados da nova conta e por fim clicar no botão de confirmação;

- **Registar produtos na Loja**:
    - **Método 1:** aceder a um cliente REST como o Insomnia ou o Postman (ou através de um comando *curl*) e realizar, sequencialmente, os **três** pedidos REST do tipo **POST**:
        - **Registar uma categoria**: realizar um pedido ao endereço http://localhost:5005/categories e submeter objeto JSON com o nome da categoria (*name*) e descrição da categoria (*description*). Dois hipotéticos exemplos poderiam ser "Playstation" ou "TV";
        - **Registar uma marca**: realizar um pedido ao endereço http://localhost:5005/brands e submeter objeto JSON com o nome da marca (*name*) e descrição da marca (*description*). Dois hipotéticos exemplos poderiam ser "Sony" ou "Toshiba";
        - **Registar o produto**: realizar um pedido ao endereço http://localhost:5005/products e submeter objeto JSON com os dados do produto, nomeadamente o seu nome (*name*), preço (*price*), descrição (*description*), imagem associada (*image*) e sua categoria (*category*) e marca (*brand*) (com os identificadores que foram definidos nos pedidos anteriores).
    - **Método 2:** os passos mencionados no método anterior podem ser efetuados recorrendo ao navegador com a interface gráfica providenciada pelo Django;
- **Registar produtos no Publicitaki**:
    - Aceder ao endereço http://localhost:7007 através do navegador (*browser*) e clicar na opção *Login* da barra de navegação. Na página de autenticação entrar como administrador e utilizar o nome *admin*, a palavra-passe *admin* e o perfil (*role*) *administrator*. Na página da conta "administrador" selecionar a opção de registo de um novo artigo. Preencher os campos necessários para o registo do artigo e clicar no botão de confirmação.
- **Registar a lista de países na Loja**:
    - Todos os clientes registados na loja deverão ter um país de origem atribuído. Assim, torna-se necessário adicionar ao microsserviço *store-users* da loja uma lista com os países que a loja trabalha. Este serviço apenas atende somente pedidos **gRPC**. Para facilitar esta tarefa foi criado um projeto escrito na linguagem C# que quando executado cria esta lista de países. Este código só deve ser executado uma só vez, caso contrário os países ficarão duplicados.

- Todos os clientes registados na loja deverão ter atribuído um país de origem, pelo que é fundamental fornecer ao microsserviço *store-users* da loja uma lista dos países. Este serviço foi exclusivamente projetado para atender pedidos gRPC. Com o intuito de facilitar esta tarefa, foi desenvolvido um pequeno projeto na linguagem C#, que, quando executado, cria essa lista de países. É importante salientar que o código deverá ser executado somente uma vez, caso contrário, ocorrerá duplicação de países.

É, igualmente, importante salientar que, no contexto da **Loja**, aquando do registo de um cliente, o formulário de registo, solicita o número da sua conta bancária. Este **número de conta** deverá ser o **identificador (*id*) da sua conta no sistema Banco**. Caso este número seja inválido, as suas **encomendas permanecerão em estado pendente**. É possível alterar o número da conta bancária do cliente na sua página da Loja.

Testagem de *endpoints*

Conforme mencionado anteriormente, para iniciar a versão Docker dos três sistemas, é necessário executar o comando *docker-compose up*. Após a inicialização, torna-se possível testar a API dos microsserviços dos três sistemas. Seguidamente, são apresentados alguns exemplos de testes aos *endpoints* destes microsserviços recorrendo ao **Insomnia**, uma aplicação de cliente REST que permite a testagem e análise de APIs.

O primeiro exemplo consiste num pedido **REST** do tipo **GET** ao recurso ***accounts*** do microsserviço ***bank-clients*** do sistema **banco** e é acedido por através de *port-forwarding* no porto 6001 do *localhost*. Este microsserviço é responsável pela gestão das contas dos clientes e das suas respetivas transações. Conforme ilustrado pela **Figura** 5-3, a resposta desse pedido resulta uma lista em formato **JSON**, que contém todas as contas registadas no banco.

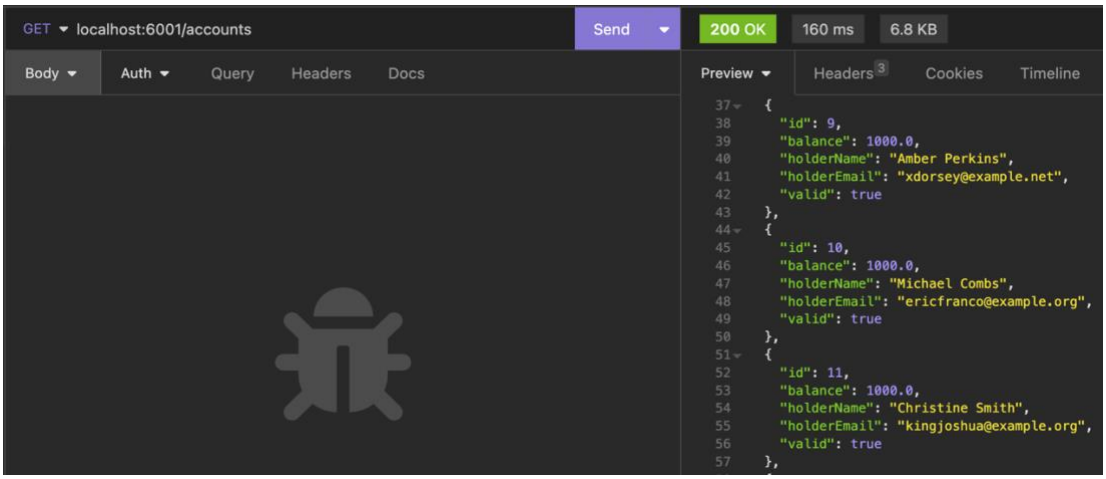


Figura 5-3: Pedido GET ao recurso *accounts* do microsserviço *bank-clients* do sistema banco

Cada conta é identificada por um identificador automático do cliente (*id*), e possui o seu balanço (*balance*), e informações acerca do cliente como nome (*holderName*), e-mail (*holderEmail*) e uma *flag* "*valid*" que indica se a conta do cliente ainda se encontra ativa no banco.

Todas as aplicações desenvolvidas em Spring Boot incluiem, obrigatoriamente, a biblioteca Swagger. Esta biblioteca oferece uma interface interativa e automática para a documentação de APIs REST desenvolvidas com o Spring Boot. Ao aceder através de um navegador (*browser*) à página no URL http://localhost:6001/swagger-ui/#/, é possível visualizar a página criada por esta biblioteca que permite uma testagem mais simples e célere da API, conforme demonstrado pela **Figura** 5-4.

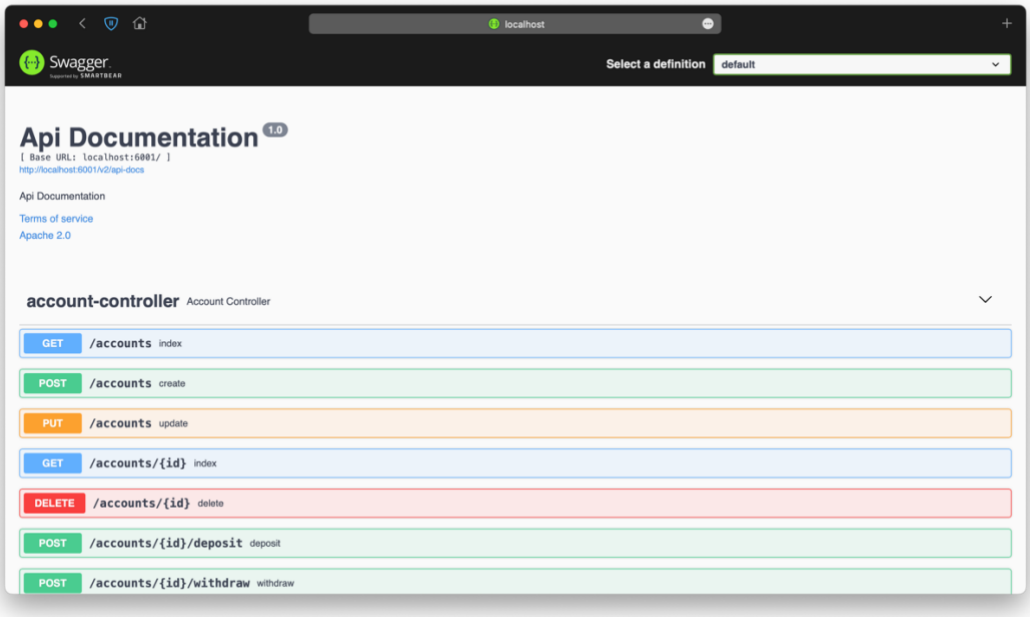


Figura 5-4: Página Swagger UI do microsserviço *bank-clients*

O exemplo seguinte consiste igualmente num pedido **REST** do tipo **POST** ao recurso ***products/videogames*** do microsserviço ***pub-products*** do sistema **Publicitaki** e é acedido por através de *port-forwarding* no porto 7005 do *localhost*. Este microsserviço é responsável pela gestão dos artigos registados no Publicitaki.

O pedido apresentado na **Figura** 5-5, permite o registo de um artigo do tipo videojogo no Publicitaki, sendo necessário fornecer o seu nome (*name*), marca (*brand*), descrição (*description*), plataforma (*plataform*), ano de publicação (*release-year*) bem como o ficheiro de imagem do artigo (*file*) e a sua respetiva extensão (*file-extension*).

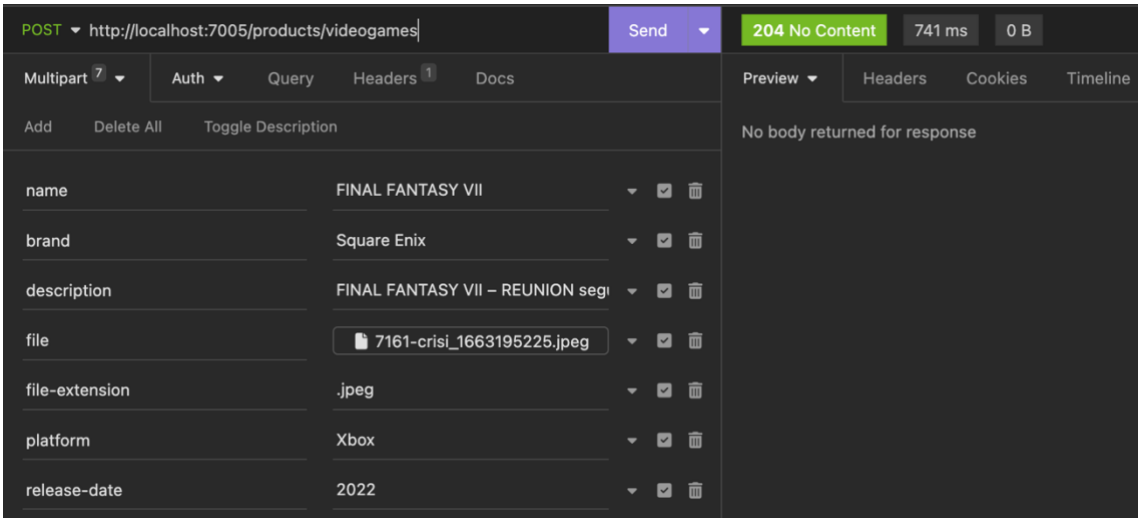


Figura 5-5: Pedido GET ao recurso *products/videogames* do microsserviço *pub-products* do sistema Publicitaki

Verifica-se que o pedido apresentado na Figura 5-5 devolve uma resposta com o código 204 (*no content*) que indica que o artigo foi guardado no sistema com sucesso.

No âmbito deste projeto, foi estabelecido que todos os **pedidos REST** que **recorram à utilização de ficheiros**, em particular ficheiros de imagem, representativos de imagens de artigos, serão tratados através do **protocolo Multipart**.

O último exemplo de um pedido **REST** consiste num pedido do tipo **DELETE** ao recurso ***stores/{id-da-loja}*** do microsserviço ***pub-users*** do sistema **Publicitaki** e é acedido por através de *port-forwarding* no porto 7003 do *localhost*. Este microsserviço é responsável pela gestão dos utilizadores registados no Publicitaki, que poderão ser clientes (*customers*) ou lojas (*stores*).

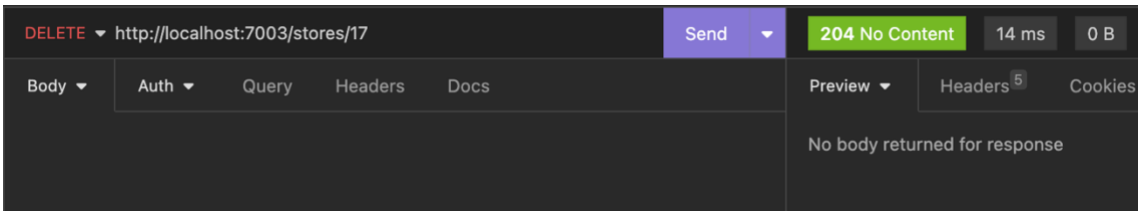


Figura 5-6: Pedido GET ao recurso *stores/{store-id}* do microsserviço *pub-products* do sistema Publicitaki

O pedido ilustrado pela **Figura** 5-6 remove a loja possui o dado identificador, que, neste caso, foi o **identificador número 17**, devolvendo uma resposta com o código 204 (*no content*) a indicar que a loja foi removida com sucesso do sistema.

5.1.2. Versão Amazon AWS

Concluído o desenvolvimento da versão Docker, foi concebida uma versão de *deployment* para uma Infraestrutura como Serviço (IaaS). Esta decisão deveu-se, em grande parte, a um dos **principais desafios enfrentados pelo Docker**, a **falta de tolerância a falhas**.

Amazon AWS

Para contornar a omissão de tolerância a falhas, opções como Docker Swarm ou Kubernetes poderiam ser utilizadas. No entanto, esta abordagem revelar-se-ia trabalhosa e dispendiosa, uma vez que a implementação de um *cluster* que recorresse a este tipo de ferramentas exigiria um investimento considerável, chegando a um custo mensal de aproximadamente cem euros apenas para a instância Master, sem considerar os nós de computação adicionais.

Desta forma, a escolha recaiu sobre a Amazon AWS, pela sua ampla gama de benefícios e vantagens oferecidas. A AWS apresenta uma infraestrutura escalável e altamente disponível, juntamente com recursos avançados de orquestração e gestão de *containers*. Além disso, a plataforma proporciona opções flexíveis de escalabilidade, monitorização e segurança, o que a torna uma escolha adequada para atender aos requisitos de implantação estabelecidos para os três sistemas.

Terraform

Definida a IaaS a utilizar, prosseguiu-se com a seleção da ferramenta de Infraestrutura como Código (IaC) mais apropriada ao projeto tendo sido consideradas as ferramentas CloudFormation e Terraform. Após uma meticulosa análise das características e capacidades oferecidas por ambas as ferramentas, optou-se pelo uso da ferramenta Terraform como a escolha que mais se enquadrava ao âmbito do projeto.

A preferência pelo Terraform relativamente ao CloudFormation fundamentou-se em dois critérios principais. Em primeiro lugar, porque o Terraform é conhecido pela sua abordagem agnóstica relativa aos fornecedores de serviços da *Cloud*, o que significa que pode ser utilizado não apenas com a Amazon Web Services (AWS), mas também com outros provedores, como Microsoft Azure e Google Cloud Platform. Essa flexibilidade proporciona uma vantagem significativa em termos de portabilidade e possibilita a adoção de uma abordagem *multicloud*, caso assim seja necessário. Em segundo lugar, porque o Terraform se destaca pela sua vasta documentação disponibilizada *online*, o que permite uma mais fácil e célere aprendizagem.

Estes fatores foram determinantes na **escolha do Terraform** em detrimento do CloudFormation. A capacidade do Terraform em trabalhar com **diferentes fornecedores de serviços *Cloud***, aliada à sua comunidade ativa e recursos *online* acessíveis, proporciona uma base sólida para a implementação eficaz da IaC, tornando-o, também por isso, na ferramenta mais apropriada para atender às necessidades do projeto, por ser capaz de garantir **flexibilidade**, **suporte** e **facilidade de utilização**.

Infraestrutura como Código (IaC)

As ferramentas de IaC como o Terraform [54] permitem ao programador construir e configurar as infraestruturas albergadas na *Cloud* de forma declarativa. Invés de realizar manualmente a configuração de cada componente da infraestrutura, o programador tem a prerrogativa de definir a configuração da mesma num ficheiro, que, por conseguinte, é executado pela ferramenta de Infraestrutura como Código (IaC) com o intuito de prover e administrar a infraestrutura na *Cloud*.

Seguidamente encontra-se descrita em maior detalhe a estrutura aplicada no desenvolvimento dos scripts Terraform usados para o *deployment* dos três sistemas para a plataforma Amazon AWS.

Código Terraform

Na fase de codificação dos scripts do Terraform, optou-se por adotar uma estratégia análoga àquela empregada na versão Docker, onde existe um ficheiro Docker-compose responsável pela criação de cada sistema. Desta forma, cada sistema possui um script Terraform responsável pela sua criação, existindo um total de três *scripts*.

A versão AWS foi estruturada de forma que a atribuir uma pasta específica para cada um dos três sistemas, cujos nomes coincidem com os das respetivas pastas. Cada pasta possui um ***script* principal** denominado de ***main.tf***, responsável pelo *deployment* do sistema e um outro ***script* secundário** denominado de ***variables.tf***, que permite a definição de variáveis que são solicitadas ao utilizador aquando da execução do script principal, nomeadamente parâmetros como a nome e palavra-passe usados nas bases de dados. A pasta do sistema conta ainda com uma subpasta interna intitula *executables* que contém os ficheiros executáveis dos microsserviços do sistema em causa (e.g., executáveis *jar* para as aplicações Java).

Para explicar a semântica utilizada na linguagem do Terraform foi selecionado como exemplo o *script* do sistema Banco.

a) *Provider AWS*

A primeira porção de código em análise encontra-se ilustrada na **Figura** 5-7.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.70.0"
    }
  }
}
#  Provider
provider "aws" {
  region = "eu-north-1"
```

```
}
```

Figura 5-7: Excerto inicial do código Terraform referente aos *providers*

O bloco de código da **Figura** 5-7 estabelece as dependências necessárias para a correta execução do código Terraform. A instrução *required_providers*, define a exigência da utilização de um *provider* específico para a plataforma AWS, neste caso, o fornecido pela HashiCorp, cuja fonte é definida como "hashicorp/aws". É, igualmente, expressa a indicação de utilização da versão 3.70.0 desse *provider*.

A instrução *provider aws*, permite a configuração do *provider aws*, sendo que a região selecionada no âmbito do projeto foi a região ***eu-north-1*** em **Estocolmo, Suécia**. Esta restrição particular implica que tanto as operações como os recursos pertinentes ao referido *provider* serão implementados e executados, exclusivamente, na região geográfica especificamente mencionada.

A **seleção desta região foi motivada**, primordialmente, por se destacar pelos seus **custos** relativamente mais baixos quando comparados com as restantes regiões disponíveis. A escolha também teve em conta a **localização geográfica** uma vez que a Suécia tal como os outros membros da união europeia partilham as mesmas normas no que concerne ao tratamento de dados.

b) Amazon S3

A segunda porção de código apresentado na **Figura** 5-8, cria, primeiramente, um *bucket* Amazon S3 privado. Os *buckets* da Amazon S3 permitem o armazenamento e gestão de dados submetidos para a plataforma AWS na *Cloud*, como código executável (e.g., ficheiros *jar*), imagens, vídeos, ficheiros estáticos entre outros tipos de recursos, com o intuito de simplificar a sua utilização aquando da criação de ambientes de desenvolvimento (*Development Environments*).

Depois de criado o *bucket* do banco, são lhe adicionados três objetos que contêm os ficheiros das aplicações a utilizar pelos microsserviços do banco, *frontend*, *clients* e *notifications*.

```
# Bucket
resource "aws_s3_bucket" "s3_bucket_fctuc_bank" {
  bucket = "fctuc-bank-bucket"
  acl    = "private"
}

# Object: Bank frontend - C#
resource "aws_s3_bucket_object" "s3_bucket_object_fctuc_bank_frontend" {
  bucket = aws_s3_bucket.s3_bucket_fctuc_bank.id
  key    = "beanstalk/fctuc-bank-frontend"
  source = "executables/Archive.zip" # published asp.net code
}

# Object: Bank clients - Java
resource "aws_s3_bucket_object" "s3_bucket_object_fctuc_bank_clients" {
  bucket = aws_s3_bucket.s3_bucket_fctuc_bank.id
  key    = "beanstalk/fctuc-bank-clients"
  source = "executables/clients-0.0.1-SNAPSHOT.jar" # jar path on localhost
}

# Object: Bank notifications - Java
resource "aws_s3_bucket_object" "s3_bucket_object_fctuc_bank_notifications" {
  bucket = aws_s3_bucket.s3_bucket_fctuc_bank.id
  key    = "beanstalk/fctuc-bank-notifications"
  source = "executables/notifications-0.0.1-SNAPSHOT.jar" # jar path
}
```

Figura 5-8: Excerto do código Terraform relativo aos *buckets* utilizados pelos microsserviços do Banco

c) Amazon Security Groups

Os grupos de segurança na AWS assemelham-se a uma *firewall*, porém adaptada à *Cloud* que permite controlar o acesso e filtrar o tráfego aos recursos da AWS, como instâncias Amazon EC2 e bases de dados Amazon RDS. Por outras palavras, os grupos de segurança da AWS oferecem um mecanismo flexível e efetivo para garantir a segurança e o controlo de acesso aos recursos na AWS.

O terceiro excerto de código apresentado na **Figura** 5-9, tem como objetivo a criação de dois grupos de segurança, sendo um para as instâncias EC2 dos ambientes de desenvolvimento (onde se encontram em execução os microsserviços) e o outro para a base de dados. O grupo de segurança das instâncias EC2, denominado de *ec2_security_group*, visa garantir o acesso aos portos TCP, 80 do HTTP (para permitir a realização de pedidos REST por parte dos clientes) e 22 do

SSH (para permitir o acesso aos microsserviços através da consola). Já o grupo de segurança *rds_mysql_security_group* permite o acesso da base de dados através da abertura do porto 3306 do MySQL.

```
#  Permissions (PORTS)
resource "aws_security_group" "ec2_security_group" {
  name_prefix = "ec2-"
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
resource "aws_security_group" "rds_mysql_security_group" {
  name_prefix = "rds-mysql-"
  ingress {
    from_port   = 3306
    to_port     = 3306
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Figura 5-9: Excerto do código Terraform relativo aos *grupos de segurança* utilizados pelos microsserviços do Banco

d) Bases de dados Amazon RDS

Existem **duas** possíveis **abordagens** a considerar na criação de uma base de dados na plataforma AWS. A primeira consiste em criar um ambiente de desenvolvimento baseado, por exemplo, numa imagem do Docker e submeter um ficheiro Dockerfile que irá gerar um *container* com o SGBD desejado, como o MySQL. A segunda abordagem envolve a criação de uma base de dados gerida completamente pela AWS, conhecida como Amazon RDS.

A principal vantagem na utilização da **Amazon RDS** reside no facto da AWS assumir integralmente a responsabilidade da gestão da base de dados, nomeadamente a gestão da infraestrutura, atualizações e a realização de backups automáticos de forma regular, libertando, assim, o programador da realização

manual destas tarefas. Justamente por esta razão, **optou-se por eleger esta abordagem** para o presente projeto.

O quarto excerto de código apresentado na **Figura** 5-10**,** permite criar uma base de dados gerida pela Amazon RDS com a versão 8.0.32 do SGBD MySQL.

Observa-se, igualmente, que a base de dados se apoia numa imagem *t3 micro* que se baseia num *container* com poucos recursos atribuídos de processamento e memória. A sua seleção prende-se com a sua inclusão na componente *Free Tier* oferecida pela Amazon AWS.

Um outro aspeto a considerar é o nome do utilizador e palavra-passe da base de dados são determinados a partir de variáveis definidas no ficheiro *variables.tf*. Esta técnica permite ao utilizador definir o valor das variáveis, aquando da execução do *script*.

Por fim que é possível gerir o acesso à base de dados recorrendo à instrução *vpc_security_group* que permite definir quais os portos e endereços de IP autorizados a aceder à base de dados. Na **Figura** 5-9 verifica-se a atribuição do grupo de segurança, através do identificador *rds_security_group*, criado para esta base de dados na **Figura** 5-10.

```
# Bank's database (MySQL)
resource "aws_db_instance" "rds_fctuc_bank" {
  engine              = "mysql"
  engine_version      = "8.0.32"
  instance_class      = "db.t3.micro"
  allocated_storage   = 20
  storage_type        = "gp2"
  identifier          = "fctuc-bank-db"
  name                = "bank_clients" # database name
  username            = var.bank_db_username
  password            = var.bank_db_password
  parameter_group_name = "default.mysql8.0"
  skip_final_snapshot  = true

  vpc_security_group_ids = [aws_security_group.rds_mysql_security_group.id]
}
```

Figura 5-10: Excerto do código Terraform relativo à criação da base de dados RDS do Banco

e) Ambientes de desenvolvimento Amazon Elastic Beanstalk

Os ambientes de desenvolvimento do Elastic Beanstalk da AWS oferecem aos programadores uma abordagem simplificada para a gestão do *deployment* de aplicações para a *Cloud* na AWS. Esta plataforma é responsável pela automatização de todas as tarefas relacionadas com a gestão e configuração dos recursos atribuídos à aplicação, abrangendo a monitorização e a alocação do número de instâncias EC2 e de distribuidores de carga denominados por *load balancers*.

Toda a informação relativa ao número de instâncias ou de réplicas é gerida no ambiente de desenvolvimento Elastic Beanstalk.

Num eventual cenário de um aumento significativo de carga, o *load balancer* inicia, automaticamente, uma nova instância da aplicação. Estas réplicas, que se assemelham a *containers* Docker, são classificadas como instâncias EC2.

Aquando da criação de um ambiente de desenvolvimento, é possível selecionar a Framework (e.g. ambientes .NET, Java, Node, etc.) que melhor se enquadra na aplicação a submeter, bem como a versão dessa mesma *framework*. Após a seleção, o Elastic Beanstalk prepara um ambiente para albergar a aplicação.

A gestão de diferentes versões da aplicação, permitindo a alternância de *deployments* é uma das primazias do Elastic Beanstalk (é possível ter a aplicação em funcionamento com a versão 2.0 e revertê-la para a versão anterior 1.0).

Deste modo, o último excerto de código do *script* do Banco, representado na **Figura** 5-11, contém um bloco inicial que cria um ambiente de desenvolvimento Elastic Beanstalk para o microsserviço de gestão de clientes do banco, *bank_clients*, através da instrução *aws_elastic_beanstalk_application*.

O segundo bloco de código, *aws_elastic_beanstalk_application_version*, permite definir a versão da aplicação, que, neste caso, será a versão 1.0.0 e associar o *bucket* com o objeto criado na **Figura** 5-8 que contém executável da aplicação do microsserviço *bank_clients*.

O terceiro e último bloco de código, *aws_elastic_beanstalk_application*, é responsável por definir a configuração do ambiente de desenvolvimento, nomeadamente a *framework* a utilizar, recorrendo, para tal, à instrução *solution_stack_name* que no código apresentado foi selecionada uma máquina com o sistema operativo Amazon Linux com a versão 11 do Java JDK. Recorrendo à instrução *setting* é possível configurar as variáveis de ambiente da aplicação, como o *endpoint* da base de dados, o tipo de instância, ou o grupo de segurança.

```
# BANK - Clients (Java Spring Boot)
resource "aws_elastic_beanstalk_application" "beanstalk_fctuc_bank_clients" {
  name        = "fctuc-bank-clients"
  description = "Bank clients' micro-service"
}
resource "aws_elastic_beanstalk_application_version" "beanstalk_fctuc_bank_clients_version" {
  application = aws_elastic_beanstalk_application.beanstalk_fctuc_bank_clients.name
  bucket      = aws_s3_bucket.s3_bucket_fctuc_bank.id                    # bucket's id
  key         = aws_s3_bucket_object.s3_bucket_object_fctuc_bank_clients.id # id of bucket with jar file
  name        = "fctuc-bank-clients-1.0.0"
}
resource "aws_elastic_beanstalk_environment" "beanstalk_fctuc_bank_clients_env" {
  name                = "fctuc-bank-clients-dev"
  application         = aws_elastic_beanstalk_application.beanstalk_fctuc_bank_clients.name
  solution_stack_name = "64bit Amazon Linux 2 v3.4.7 running Corretto 11"
  version_label       = aws_elastic_beanstalk_application_version.beanstalk_fctuc_bank_clients_version.name

  # database url, name, password, db-name
  setting {
    namespace = "aws:elasticbeanstalk:application:environment"
```

```
  name      = "BANK_CLIENTS_DATABASE_URL"
  value     = aws_db_instance.rds_fctuc_bank.endpoint
 }
 setting {
  namespace = "aws:elasticbeanstalk:application:environment"
  name      = "BANK_CLIENTS_DATABASE_USERNAME"
  value     = aws_db_instance.rds_fctuc_bank.username
 }
 setting {
  namespace = "aws:elasticbeanstalk:application:environment"
  name      = "BANK_CLIENTS_DATABASE_PASSWORD"
  value     = aws_db_instance.rds_fctuc_bank.password
 }
 setting {
  namespace = "aws:elasticbeanstalk:application:environment"
  name      = "BANK_CLIENTS_DATABASE_NAME"
  value     = aws_db_instance.rds_fctuc_bank.name
 }

 setting {
  name      = "SERVER_PORT"
  namespace = "aws:elasticbeanstalk:application:environment"
  value     = "5000"
 }
 setting {
  namespace = "aws:ec2:instances"
  name      = "InstanceTypes"
  value     = "t3.micro"
 }

 setting {
  namespace = "aws:autoscaling:launchconfiguration"
  name      = "IamInstanceProfile"
  value     = "aws-elasticbeanstalk-ec2-role"
 }

 setting {
  namespace = "aws:autoscaling:launchconfiguration"
  name      = "SecurityGroups"
  value     = aws_security_group.ec2_security_group.name
 }
}
```

Figura 5-11: Excerto do código Terraform relativo à criação do ambiente de desenvolvimento Elastic Beanstalk do microsserviço *bank-clients* do Banco

f) Fase de *deployment*

Finalizada a apresentação da sintaxe do Terraform, torna-se, assim, possível prosseguir para a fase de execução.

Para efetuar o *deployment* dos três sistemas para a plataforma AWS, é imperativo assegurar, em primeiro lugar, que as dependências do Terraform se encontram presentes nas pastas dos respetivos sistemas. Na eventualidade da sua inexistência, dever-se-á executar em cada uma das três pastas, que contêm o *script main.tf*, o comando:

```
terraform init
```

O passo seguinte será executar o comando do efetivo *deployment* dos três sistemas para a AWS através do comando:

```
terraform apply -y
```

Depois de executado o comando anterior para os três sistemas, o Terraform irá realizar o *deployment* dos mesmos para a plataforma AWS. É, contudo, importante referir que esta operação revelar-se-á um pouco morosa podendo levar entre cinco e dez minutos. Concluído o *deployment*, os três sistemas estarão prontos para atender os pedido dos clientes.

Para evitar custos desnecessários, dever-se-á remover o *deployment* dos sistemas recorrendo, para tal, ao comando:

```
terraform destroy -y
```

Para facilitar a realização de todas estas tarefas foram adicionados ao repositório GitHub do projeto dois *scripts*, um destinado à realização do *deployment* dos três sistemas e outro à sua remoção.

5.1.3. Funcionamento dos sistemas

Concluída a apresentação da estrutura da fase de *deployment* da versão Amazon AWS dos sistemas, é, então, possível prosseguir para a fase de apresentação do funcionamento dos três sistemas. Pela sua extensão descritiva apresenta-se somente o sistema Banco, encontrando-se o funcionamento dos sistemas **Loja** e **Publicitaki** detalhado no **Anexo K:**.

Sistema Banco

Como mencionado no **Capítulo 3**, o sistema **Banco** é usado pelos sistemas Loja e Publicitaki com o intuito de permitir a realização de transações bancárias, nomeadamente, pagamento de encomendas pelo cliente e pagamento de subscrições por parte do dono da loja no Publicitaki.

Os outros dois sistemas comunicam diretamente com os *endpoints* dos microsserviços, e para facilitar o processo de testagem foi desenvolvida uma interface gráfica.

Para aceder à página Banco o utilizador deverá indicar o endereço do *endpoint* do microsserviço *bank-frontend*, no seu navegador *web*. Ao aceder à página o utilizador visualizará uma página idêntica àquela exibida na **Figura** 5-12.

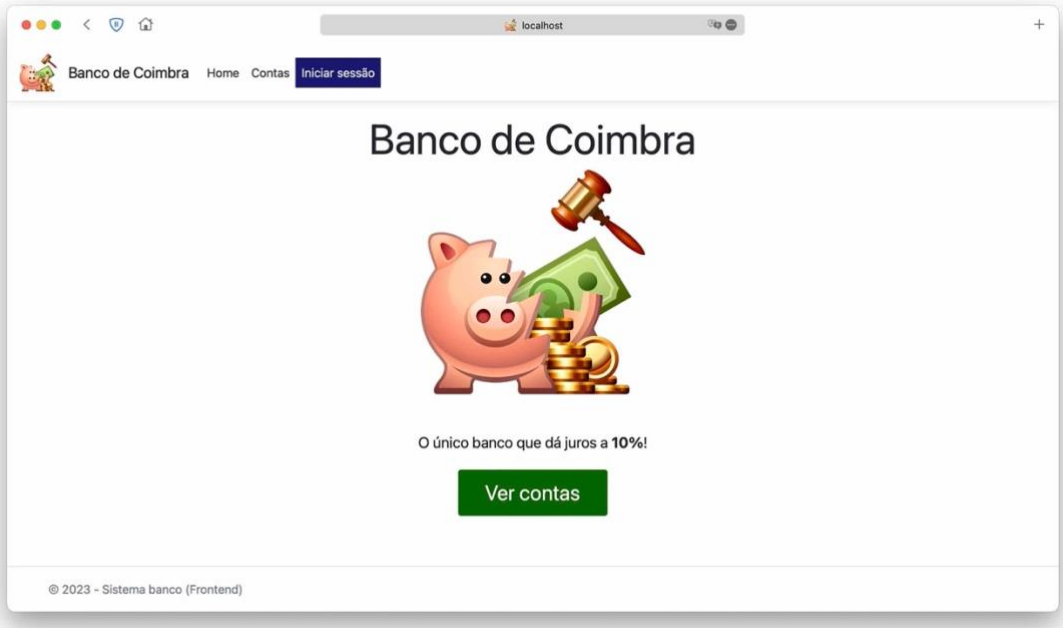


Figura 5-12: Página principal do sistema Banco

Nesta página, o utilizador, deverá autenticar-se no sistema como administrador para consultar e gerir as contas dos clientes. Para tal deverá clicar "Iniciar sessão" e inserir as suas credenciais de acesso, nomeadamente o nome a palavra-passe, conforme ilustrado na **Figura** 5-13.

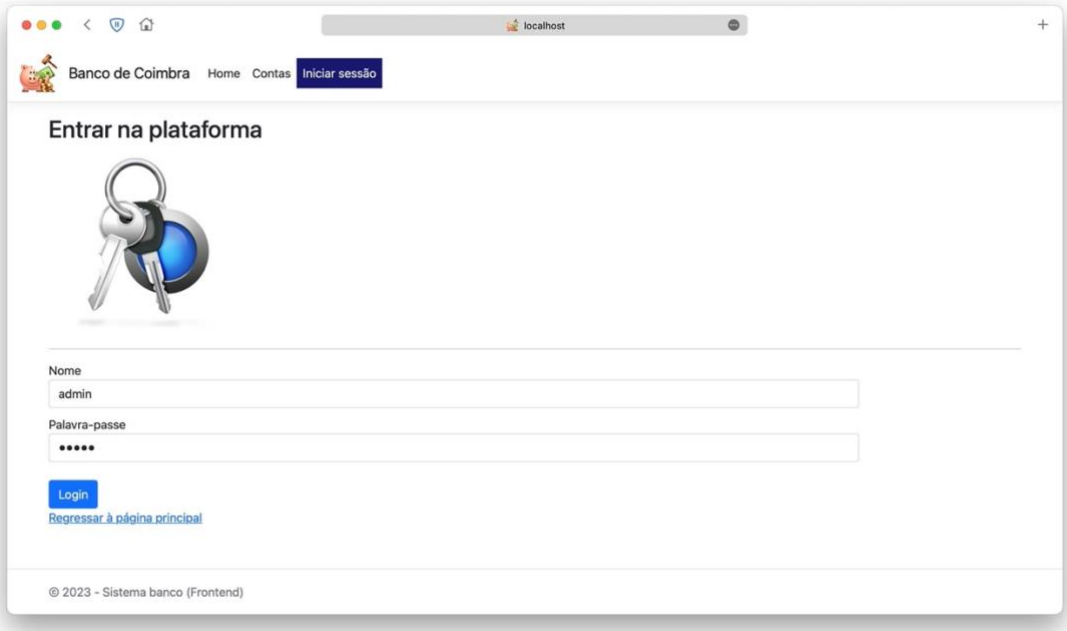


Figura 5-13: Página de autenticação do administrador no sistema Banco

Efetuada a autenticação o utilizador, poderá aceder à página responsável pela gestão das contas bancárias, clicando no separador "contas" presente na barra de navegação, onde será encaminhado para a página apresentada na **Figura** 5-14.

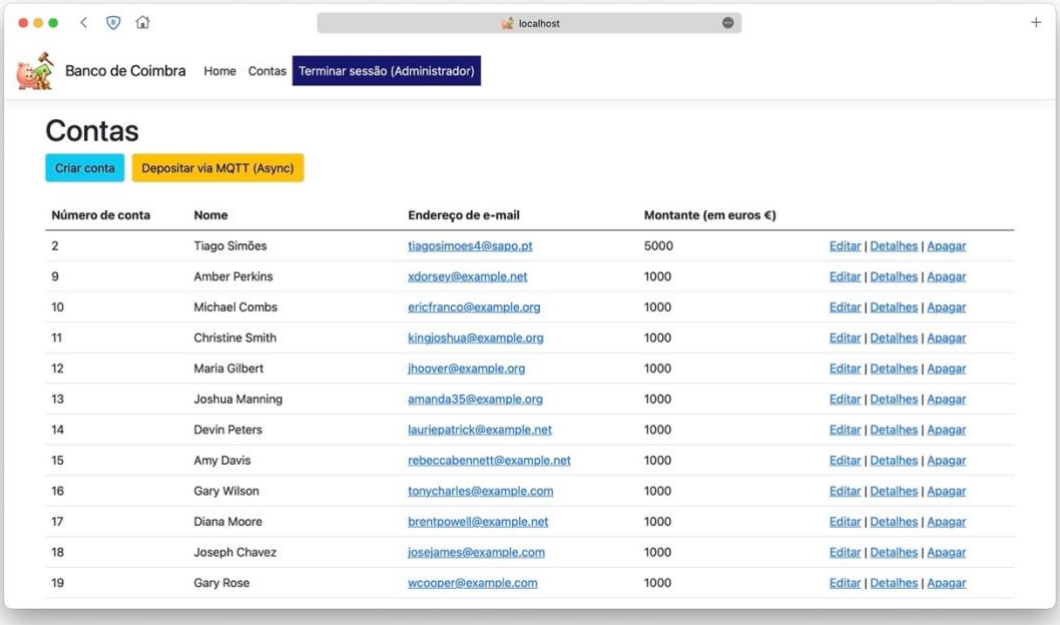


Figura 5-14: Página de gestão de contas bancárias do sistema Banco

Nesta página o utilizador, poderá e visualizar as conta existentes, editar, visualizar ou apagar uma determinada conta. Poderá ainda registar uma nova conta clicando no botão "criar conta" e ainda adicionar fundos a uma conta existente, clicando no botão "depositar via MQTT".

Ao clicar no botão "depositar via MQTT" o utilizador é encaminhado para a página de depósitos, conforme apresentado na **Figura** 5-15.

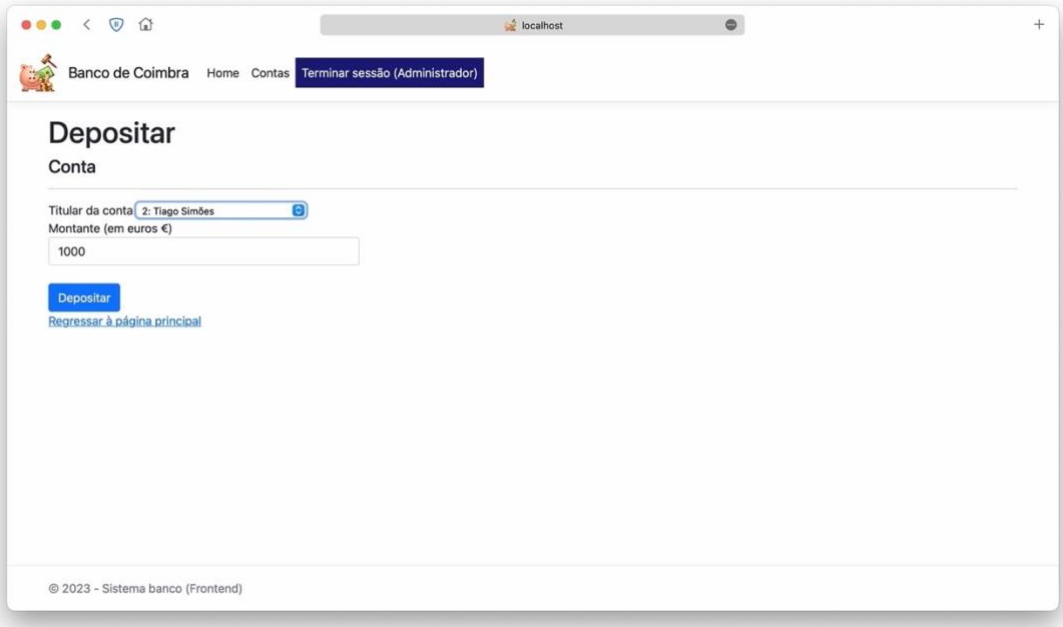


Figura 5-15: Página de depósitos do sistema Banco

Nesta página o utilizador seleciona a conta para a qual pretende efetuar o depósito (*titular da conta*) (selecionando, na lista *drop-down*, a conta que prende escolher), e a respetiva quantia (*montante*). Preenchidos os campos o utilizador deverá efetuar a depósito clicando no botão "depositar".

Relativamente à arquitetura, a comunicação entre o Banco e os restantes sistemas é efetuado sob o **padrão de comunicação assíncrono** *publish-subscribe* recorrendo ao **HiveMQTT**.

É importante salientar que a página do Banco não comunica diretamente com o microsserviço de *backend* (denominado de *bank-clients*) para efetuar a transação. Em vez disso, a comunicação ocorre via MQTT, sendo que o microsserviço *bank-clients* se encontra em **constante monitorização pela receção de mensagens** classificadas como *transações bancárias*. Assim que uma mensagem MQTT é detetada na fila, o evento é reconhecido e a transação é realizada.

Efetuada o depósito, o utilizador é encaminho novamente para a página de gestão de contas da **Figura** 5-14. É importante referir que o montante da conta para a qual foi feito o depósito ainda aparecerá com o mesmo valor uma vez que a operação anterior foi realizada de forma assíncrona, sendo necessário refrescar a página segundos mais tarde até ser efetuado o depósito.

5.1.4. Testes de carga

Finalizada a explicação do funcionamento dos sistemas, é possível partir-se para o tema seguinte relativo à fase de realização dos testes de carga.

Para a realização dos testes de carga foi necessário, tal como mencionado no capítulo anterior, **Capítulo 2.8.1**, a ferramenta de testagem Locust.

Outras opções, como o JMeter, foram consideradas, no entanto, optou-se pela ferramenta Locust devido à sua reduzida curva de aprendizagem e à facilidade de escrita dos *scripts* de teste, uma vez que a ferramenta recorre à linguagem Python.

Sintaxe do Locust

Para utilizar o Locust, dever-se-á primeiramente, realizar a instalação do Python e, posteriormente, a instalação do Locust através do comando *pip locust*. Depois de instalado, dever-se-á criar um ficheiro Python e importar o módulo Locust.

O ficheiro Python deverá conter uma classe que estenda a classe HttpUser do módulo Locust, contendo um método com a anotação *@task* para cada pedido REST que se pretende testar.

A **Figura** 5-16 evidencia um simples ficheiro de teste Locust cuja tarefa é a de realizar um pedido GET para obter todos os clientes, outro pedido GET para obter um cliente pelo seu identificador, e um pedido POST para registar um novo cliente.

```python
import random
from locust import HttpUser, task, between

class MyUser(HttpUser):
    wait_time = between(1, 2)

    @task
    def get_all_customers(self):
        self.client.get("/customers")

    @task
    def get_customer_by_id(self):
        customer_id = random.randint(1, 100) # random customer ID between 1 and 100
        self.client.get(f"/customers/{customer_id}", name="Customer Details")


    @task
    def create_customer(self):
        data = {
            "name": "John Doe",
            "email": "johndoe@example.com",
            "password": "password123"
        }
        self.client.post("/customers", json=data)
```

Figura 5-16: Exemplo de um ficheiro de teste de carga Locust

Para iniciar o teste deve-se correr o seguinte comando:

```
locust -f <ficheiro-de-teste>.py
```

Ao executar o comando é possível aceder ao serviço de testagem do Locust abrindo o navegador no endereço http://localhost:8089 como apresentado na **Figura** 5-17.

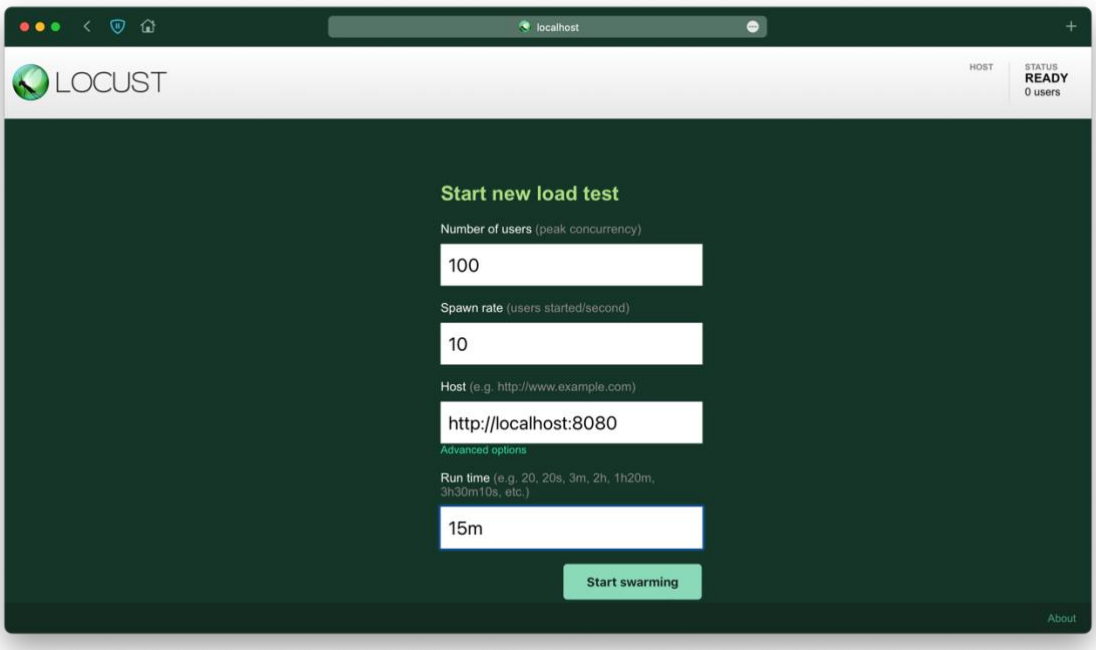


Figura 5-17: Página web do serviço Locust

Nesta consola define-se o número máximo de utilizadores (*number of users - peak concorrency*), o número de utilizadores a incrementar por segundo (*spawn rate – users per second*), o endpoint da aplicação a testar (*host*) e o tempo de execução. No teste apresentado na **Figura** 5-17 constata-se o pico do número de utilizadores foi cem e a taxa de incremento foi de dez utilizadores por segundo durante quinze minutos.

Depois de selecionados os parâmetros, clica-se no botão *Start swarming* para dar início ao teste. Depois de finalizado o teste pode-se gerar um relatório com o sumário do teste.

Experiência

Para a realização da testagem foi necessário definir os critérios e cenários a utilizar, tendo a escolha sido definida do seguinte modo:

- **Critério de testagem:** realização de pedidos REST, dos tipos GET, POST e DELETE;
- **Duração de cada execução:** 15 minutos;
- **Número de pedidos (de clientes REST) por microsserviço:**
  - **Cenário n.º 1** – carga **baixa**: 10 pedidos (aumento de 1 em 1);
  - **Cenário n.º 2** – carga **moderada**: 100 pedidos (aumento de 10 em 10);

    - **Cenário n.º 3** – carga **elevada:** 1000 pedidos (aumento de 10 em 10);

- **Microsserviços submetidos ao processo de testagem**: todos os microsserviços dos três sistemas;
- **Número de *scripts* de testagem**: um *script* atribuído por cada microsserviço;
- **Distribuidor de carga (*load balancer)*:**
    - Número **mínimo** de instâncias: **uma** instância;
    - Número **máximo** de instância: **quatro** instâncias;

É importante realçar que, durante o processo de testagem de carga, **o número de pedidos REST aumentou de forma gradual**, conforme especificado **para cada cenário**, sendo **incrementado** de acordo com as quantidades acima mencionadas.

A seleção dos valores mínimo e máximo de instâncias do *load balancer* deve-se, fundamentalmente, à necessidade de contenção de custos, a fim de evitar gastos inesperados e excessivos decorrentes da alocação descontrolada de instâncias EC2.

Resultados

Após a execução do *script* de testagem de cada microsserviço, obteve-se um ficheiro HTML que contém a descrição sumária dos resultados da experiência. Este ficheiro contém detalhadamente os pedidos REST realizados e o respetivo número de pedidos, o tempo de resposta mínimo, médio, e máximo, tamanho do pedido em bytes e a taxa de avarias por segundo.

No presente documento serão destacados os resultados de maior relevância, situando-se os restantes no repositório GitHub do projeto.

Durante o processo de testagem, observou-se que em situações de baixa carga (cenário nº 1), o distribuidor de carga recorreu somente a uma instância EC2 para atender os pedidos, revelando-se adequado com base nos tempos de resposta obtidos.

Ao realizar os cenários de carga moderada, constatou-se que o *load balancer* do Elastic Beanstalk (*load balancer*) inicia, primeiramente, uma única instância da aplicação, no entanto, à medida que o número de pedidos aumenta e essa única instância não consegue processar os pedidos em tempo útil, o *load balancer* inicia automaticamente uma nova instância da aplicação. Na generalidade, o número de instâncias criadas durante o cenário de carga moderada situou-se entre duas e três instâncias.

Esta ocorrência é, ainda, mais evidente no cenário de carga elevada, onde, geralmente, até ao final da experiência, são utilizadas quatro instâncias EC2, atingindo, assim, o limite máximo estabelecido de instâncias. Na maioria dos microsserviços testados no cenário de carga elevada foram lançadas quatro instâncias EC2.

O gráfico da **Figura** 5-18 retrata, parcialmente, a evolução do tempo médio de resposta no atendimento dos pedidos gerados pelo Locust para o microsserviço "pub-products" sob carga moderada. Observa-se que é **atingido um pico no tempo**

de resposta de 1300 milissegundos no intervalo entre 2:57:53 PM e 2:58:23 PM. Nesse intervalo, torna-se evidente que a única instância EC2 existente, da aplicação, **não é capaz de lidar com os pedidos** do Locust de forma eficiente. No entanto, após esse período, ocorre uma queda abrupta no tempo de resposta para 150 milissegundos, que pode ser explicado pelo facto de o ***load balancer* ter iniciado**, momentos antes, **uma nova instância EC2** para partilhar a carga da aplicação.

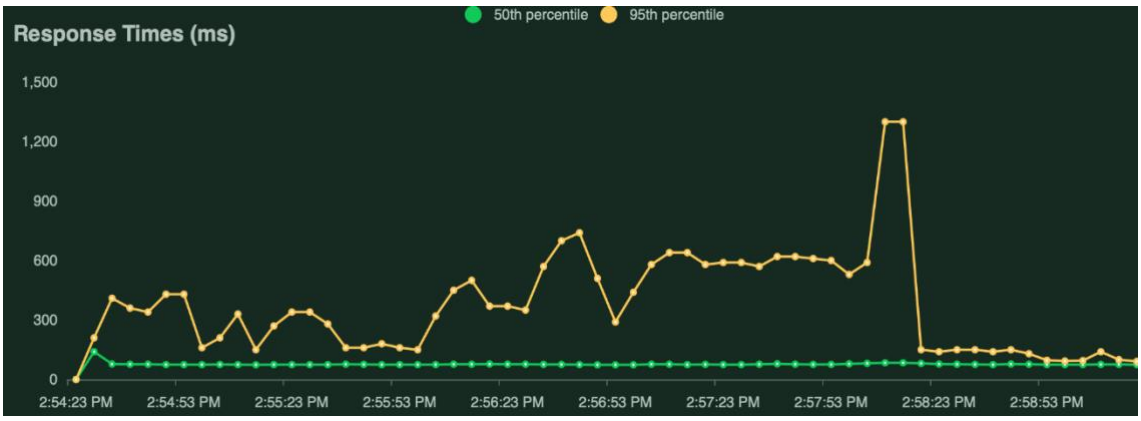


Figura 5-18: Representação gráfica parcial dos tempos de resposta, em milissegundos, do microsserviço "pub-products" do Publicitaki, no cenário de carga moderada.

Durante a realização da experiência, verificou-se, igualmente, que o *load balancer* tem, não só, a capacidade de atribuição de novas instâncias para atender ao aumento do tráfego como para situações em que o tráfego diminui. Foi efetuada uma testagem adicional com o intuito de avaliar a diminuição do tráfego do número de clientes, e constatou-se que o número de instâncias diminuiu proporcionalmente. Esta situação ocorre pelo facto de o *load balancer* detetar a redução na demanda de pedidos, o que o leva a tomar a decisão de destruir as instâncias que são consideradas desnecessárias.

Conclusão

Após a realização dos testes de carga, é possível inferir que a plataforma Amazon AWS oferece um serviço eficaz de distribuição de carga (*load balacing*), que se adapta às necessidades requisitadas na fase de *deployment* de aplicações *web*.

Uma das principais vantagens da utilização de uma infraestrutura como serviço (IaaS) é, sem dúvida, a presença de um serviço de distribuição de carga que se destaca pela sua notável capacidade de criar instâncias adicionais de forma eficiente quando há um aumento significativo no tráfego recebido por uma aplicação web. Isto é, sobretudo, relevante em cenários particulares, como a época de Natal, altura em que as lojas online enfrentam um tráfego, habitualmente, elevado. Ao adotar uma plataforma Amazon AWS para o deployment de uma aplicação web, o sistema é capaz de lidar com o aumento exponencial dos pedidos dos clientes, garantindo que o seu serviço não fique sobrecarregado.

É, igualmente, importante referir que o distribuidor de carga também atua de forma eficiente em cenários opostos, quando existe um tráfego baixo de clientes. Nesses casos, o distribuidor de carga deteta a baixa demanda e reduz o número de instâncias atribuídas à aplicação, diminuindo, assim, custos desnecessários.

Em suma, a plataforma Amazon AWS fornece um serviço de distribuição de carga eficiente, que se adapta de forma dinâmica às exigências do tráfego recebido pelas aplicações, seja aumentando ou diminuindo o número de instâncias da mesma, com o intuito de otimizar o desempenho e minimizar custos desnecessários.

5.1.5. Testes de usabilidade

Para avaliar o atributo de qualidade de *usabilidade*, foi selecionado um **utilizador convidado** que possui competências básicas na área de informática. Este participante foi escolhido, intencionalmente, por representar um típico utilizador, capaz de realizar tarefas comuns, como compras *online*. No entanto, é importante salientar que o participante não possui experiência específica na área de engenharia informática.

Assim, seguidamente, são apresentados os resultados obtidos no *Walkthrough* Cognitivo:

Área de foco 1 - Publicitaki

Questão 1: Quem são os utilizadores?

Resposta: São o administrador, o cliente e a loja

Questão 2: Na página principal do Publicitaki, o dono da loja percebe facilmente, para onde o botão "Associar preço de artigo" o irá redirecionar?

Resposta: Sim, o dono da loja percebe que será encaminhado para a página onde define o preço de um artigo que vende na sua loja.

Questão 3: Na página de listagem de artigos, o cliente percebe que pode filtrar os artigos pelo seu tipo ou características?

Resposta: Sim, o utilizador percebe que pode usar os separadores da barra superior, para selecionar a categoria do artigo e pode clicar num artigo específico para visualizar os seus detalhes.

Questão 4: O utilizador sabe como, nas páginas de consulta de um artigo, regressar à página anterior?

Resposta: Sim, o utilizador percebe que pode utilizar o botão com a seta esquerda do seu navegador para regressar à página anterior ou clicar no botão no fundo da página *Return to previous page.*

Questão 5: O dono da loja dispõe de uma opção para consultar a listagem dos seus artigos registados no sistema?

Resposta: Sim, o utilizador percebe que pode clicar no separador *All products* para poder visualizar todos os artigos registados no Publicitaki.

Questão 6: O utilizador (dono da loja e cliente) a qualquer momento pode terminar a sua sessão?

Resposta: Sim, o utilizador, no canto superior direito, poderá clicar em *Log out*.

Área de foco 2 - Loja

Questão 1: Quem são os utilizadores?

Resposta: São o cliente da loja

Questão 2: O utilizador sabe que pode adicionar associar vários artigos a uma só encomenda?

Resposta: Sim, é possível realizar esta tarefa adicionando os artigos desejados ao carrinho de compras.

Questão 3: O utilizador apercebe-se que ao terminar a sessão não perde os artigos anteriormente adicionados ao carrinho?

Resposta: Sim, ao terminar a sessão e, posteriormente, voltar a autenticar-se, o utilizador percebe que os seus artigos ainda se encontram no carrinho.

Questão 4: O utilizador percebe que pode remover artigos do seu carrinho?

Resposta: Sim, o utilizador percebe que pode remover artigos clicando no botão vermelho de remoção de artigos, e que pode, igualmente, alterar o número de quantidades desse artigo.

Questão 5: O utilizador entende o que significa o estado pendente de uma encomenda?

Resposta: Sim, o utilizador percebe que o pagamento ainda não foi efetuado.

Questão 6: O utilizador percebe que pode publicar uma opinião?

Resposta: Sim, o utilizador percebe que pode publicar uma opinião relativa a um artigo e também visualizar outras opiniões efetuadas por outros utilizadores.

É importante referir que o sistema Banco não foi submetido ao teste de usabilidade uma vez que apenas será utilizado como ferramenta de testagem e não será utilizado por utilizadores finais.

Após a finalização do *Walkthrough* Cognitivo, foi solicitado a um **perito** que avaliasse os sistemas Loja e Publicitaki recorrendo às heurísticas de Jakob Nielsen tendo sido atribuídos os seguintes resultados:

Heurística	Percentagem atribuída por perito
Heurística n. º 1 - Visibilidade do sistema	***80%***
Heurística n. º 2 - Correspondência entre o sistema e o mundo real	***80%***

Heurística n. º 3 - Liberdade e controlo do utilizador	***70%***
Heurística n. º 4 - Consistência e padrões	***80%***
Heurística n. º 5 - Prevenção de Erros	***80%***
Heurística n. º 6 - Reconhecer ao invés de lembrar	***80%***
Heurística n. º 7 - Flexibilidade e eficiência de uso	***75%***
Heurística n. º 8 - Estética e design minimalista	***80%***
Heurística n. º 9 - Ajudar utilizadores a reconhecer, diagnosticar e recuperar erros	***65%***
Heurística n. º 10 - Ajuda e documentação	***90%***

Tendo em consideração os resultados apresentados, torna-se possível afirmar que os sistemas passaram nos testes de usabilidade.

5.1.6. *Peer review*

Tal como referido no Capítulo 3.5, foi efetuada uma *software review* aos componentes mais críticos ao correto funcionamento dos sistemas.

Assim, foram selecionados os seguintes componentes:

- Ficheiro AccountService.java, pertencente ao microsserviço *bank-clients*, do Banco;
- Ficheiro Views.py, pertencente ao microsserviço *store-products*, da Loja;
- Ficheiro StoreProducts.tsx, pertencente ao microsserviço do Publicitaki;

Os **resultados** da *review* podem ser consultados nos **documentos anexados** ao presente relatório.

5.1.7. Riscos

Durante a execução do projeto, foi elaborado um plano de riscos com o intuito de identificar e mitigar possíveis ocorrências inesperadas que pudessem comprometer negativamente o seu desenvolvimento. No entanto, é importante salientar que, ao longo do projeto, nenhum dos riscos identificados se concretizou.

Concluída a apresentação da implementação dos sistemas, prossegue-se para o capítulo seguinte, **Capítulo 6**, referente às fases de planeamento do projeto.

Capítulo 6
Metodologia e Planeamento

O presente capítulo apresenta a decomposição das tarefas realizadas nas quatro fases do projeto e a metodologia aplicada e a razão da sua escolha.

6.1 Cronograma

O projeto tinha como plano inicial a continuidade de uma aplicação já concebida. No entanto, não foi possível a sua concretização, pelas razões apontadas no subcapítulo **1.3 Objetivos** . Desta forma, foi necessário introduzir alguns ajustes à proposta inicialmente estabelecida.

Proposta inicial

1º Semestre:

- T1 – Conhecimento da aplicação existente (1 mês);
- T2 – Avaliação do estado da arte e definição de uma nova aplicação (1 mês);
- T3 – Definição dos Requisitos e arquitetura do sistema (1 mês);
- T4 – Desenvolvimento dos microsserviços (1 mês);
- T5 – Redação do relatório intermediário (1 mês).

2º Semestre:

- T6 – Implementação da aplicação (2 meses)
- T7 – Testagem e avaliação do desempenho do sistema, testes com o utilizador (1 mês)
- T8 – Reunião e disponibilização de dados (1 mês)
- T9 – Redação do relatório final (1 mês)

Proposta concretizada

Por ainda se encontrar em fase de conclusão não foi possível avançar com a proposta inicial, tendo sido deliberado a criação de uma aplicação nova de raiz.

Pela necessidade de alterações ao escopo foi decidido passar a tarefa quatro ("**Desenvolvimento dos microsserviços**") para o segundo semestre. Também a tarefa um ("**Conhecimento da aplicação existente**") foi alterada e anexada à tarefa dois ("**Avaliação do estado da arte e definição de uma nova aplicação**").

Assim foi necessário reformular as fases de desenvolvimento do projeto, tendo sido decidido decompô-lo em quatro fases:

1º Semestre:

- Fase 1 - Avaliação do estado da arte e definição de uma nova aplicação;
- Fase 2 - Definição dos requisitos e arquitetura do sistema;

2º Semestre:

- Fase 3 - Desenvolvimento e implementação da aplicação;
- Fase 4 - Avaliação e testagem.

As fases um e dois foram desenvolvidas durante o primeiro semestre e as fases três e quatro no segundo semestre.

Fase 1 – Avaliação do estado da arte e definição de uma nova aplicação

A primeira fase corresponde a todas as tarefas referentes à análise do estado da arte e da definição do tema da aplicação a desenvolver. Na **Tabela** 6-1 encontram-se apresentadas as tarefas realizadas nesta fase.

Tabela 6-1: Tarefas realizadas durante a fase n.º um com as suas respetivas datas de início e término

Fase 1 - Avaliação do estado da arte e definição da nova aplicação			
T1 - Estudo do padrão arquitetural de microserviços	100%	9/8/22	9/15/22
T2 - Aprendizagem do Docker	100%	9/15/22	9/22/22
T3 - Aprendizagem das tecnologias da Cloud	100%	9/22/22	9/27/22
T4 - Análise do microserviço Sock Shop	100%	9/27/22	10/1/22
T5 - Análise do microserviço Robot Shop	100%	10/1/22	10/5/22
T6 - Análise do microserviço Tea Store	100%	10/5/22	10/6/22
T7 - Análise do microserviço Web Shop (versão 1)	100%	10/6/22	10/7/22
T8 - Análise do microserviço Piggy Metric	100%	10/7/22	10/8/22
T9 - Análise do microserviço DeathStarBench	100%	10/8/22	10/12/22
T10 - Definição da nova aplicação	100%	10/12/22	10/16/22

Como ilustrado na tabela, esta fase iniciou-se com o estudo comparativo entre os principais padrões arquiteturais de *software* existentes, como o padrão arquitetural monolítico, a arquitetura orientada a serviços, e os microsserviços (**tarefa um**).

Depois de analisadas as principais características destes padrões arquiteturais de *software*, procedeu-se à aprendizagem do Docker e das principais tecnologias *Cloud* usadas na fase de *deployment* de microsserviços (**tarefas dois e três**, respetivamente).

Prosseguiu-se, depois, para o estudo e análise de algumas aplicações exemplificativas de microsserviços já desenvolvidas (**tarefas quatro a nove**) o que muito contribuiu para uma ideia mais clara do funcionamento deste tipo de aplicações e conhecimento dos seus principais pontos fortes e respetivas debilidades (**tarefas quatro a nove**).

Com base nos elementos recolhidos foi, então, possível definir, com clareza, o tema a desenvolver no presente projeto (**tarefa dez**).

Fase 2 – Definição dos Requisitos e Arquitetura do sistema

A segunda fase corresponde a todas as tarefas referentes à definição dos casos de uso, ao levantamento dos requisitos e à documentação da arquitetura da aplicação a desenvolver. Na **Tabela** 6-2 encontram-se discriminadas todas as tarefas realizadas nesta fase.

Tabela 6-2: Tarefas realizadas durante a fase n.º dois com as suas respetivas datas de início e término

Fase 2 - Definição dos Requisitos e Arquitetura do sistema			
T11 - Escolha da metodologia e das tecnologias	100%	10/13/22	10/15/22
T12 - Documentação da fase anterior	100%	10/15/22	10/22/22
T13 - Definição dos casos de uso	100%	10/22/22	11/5/22
T14 - Levantamento dos requisitos funcionais	100%	11/5/22	11/17/22
T15 - Levantamento dos requisitos não-funcionais e das restrições	100%	11/17/22	11/24/22
T16 - Definição da arquitetura do sistema (C4)	100%	11/24/22	12/2/22
T17 - Redação do relatório intermédio	100%	12/2/22	1/11/23

Após a definição do tema da aplicação a desenvolver iniciou-se a segunda fase do projeto. A primeira tarefa a realizar nesta fase foi a escolha da metodologia de desenvolvimento de *software* e das tecnologias (*Framework* e linguagens de programação) a utilizar na fase três de implementação dos microsserviços (**tarefa onze**).

Depois de delineadas as escolhas realizadas na tarefa onze procedeu-se à documentação/redação das tarefas realizadas na fase um, nomeadamente a comparação entre os padrões arquiteturais, a explicação do funcionamento do Docker e das tecnologias usadas na *Cloud*, e da análise comparativa entre as aplicações de exemplo de microsserviços (**tarefa doze**).

De seguida prosseguiu-se para a definição dos diagramas UML de casos de uso dos três sistemas contidos na aplicação a desenvolver (consultar subcapítulo **3.1 Casos de Uso**) (**tarefa treze**).

Definidos os casos de uso, procedeu-se ao levantamento e documentação dos requisitos funcionais da aplicação com as suas respetivas descrições individuais (consultar o subcapítulo 3.2 **Requisitos Funcionais**) (**tarefa catorze**).

Concluído o levantamento dos requisitos funcionais da aplicação realizou-se não só o levantamento dos requisitos não-funcionais da aplicação, mas também a enumeração das restrições técnicas e de negócio impostas pelo projeto. Nesta tarefa foi também delineada a forma como o requisito não-funcional da usabilidade da interface gráfica da aplicação será avaliada (consultar subcapítulos 3.3 **Requisitos não-funcionais** e **3.4 Restrições**) (**tarefa quinze**).

Depois de definidos os casos de uso e os requisitos funcionais e não-funcionais da aplicação e as suas respetivas restrições, foi assim possível prosseguir para a

definição da arquitetura da aplicação (consultar o subcapítulo 4.1 **Arquitetura**) (**tarefa dezasseis**). Nesta tarefa, a arquitetura da aplicação foi detalhada individualmente para os três sistemas, recorrendo aos diagramas C4 de Simon Brown.

A presente fase, foi fechada com a realização da tarefa de redação do presente documento remente à primeira parte do projeto do primeiro semestre.

Fase 3 – Desenvolvimento e implementação da aplicação

A terceira fase remete a todas as tarefas referentes à implementação dos três sistemas da aplicação. Na **Tabela** 6-3 encontram-se expostas as tarefas a realizar nesta fase, no segundo semestre.

Tabela 6-3: Tarefas a realizar, futuramente, durante a fase n.º três

Fase 3 - Desenvolvimento e implementação
T18 - Definição dos recursos da API REST
T19 - Implementação do microsserviço "Gestão de Registos" do Banco
T20 - Implementação do microsserviço "Gestão de Transações" do Banco
T21 - Implementação do microsserviço "Gestão de Notificações" do Banco
T22 - Implementação do microsserviço "Gestão de Artigos" da Loja
T23 - Implementação do microsserviço "Gestão de Registos" da Loja
T24 - Implementação do microsserviço "Gestão de Pagamentos" da Loja
T25 - Implementação do microsserviço "Gestão de Notificações" da Loja
T26 - Implementação do microsserviço "Gestão de Artigos" do Publicitaki
T27 - Implementação do microsserviço "Gestão de Registos" do Publicitaki
T28 - Implementação do microsserviço "Gestão de Pagamentos" do Publicitaki
T29 - Implementação do microsserviço "Gestão de Notificações" do Publicitaki

A terceira fase (2º semestre) terá início com a tarefa referente à definição dos recursos URI da API REST dos três sistemas da aplicação (**tarefa dezoito**).

De seguida proceder-se-á à implementação dos microsserviços do sistema Banco (**tarefas dezanove a vinte e um**), do sistema Loja (**tarefas vinte e dois a vinte e cinco**), e do sistema Publicitaki (**tarefas vinte e seis a vinte e nove**).

Quando terminada a implementação dos microsserviços dos três sistemas será dada como concluída a fase três.

Fase 4 – Avaliação e testagem da aplicação

A quarta e última fase remete a todas as tarefas referentes à avaliação, testagem e monitorização dos três sistemas da aplicação. Na **Tabela** 6-4 encontram-se expostas as tarefas a realizar nesta fase, no segundo semestre.

Tabela 6-4: Tarefas a realizar, futuramente, durante a fase n.º quatro

Phase 4 - Avaliação e testagem
T30 - Testagem do sistema Banco
T31 - Testagem do sistema Loja
T32 - Testagem do sistema Publicitaki
T33 - Escrita do relatório científico
T34 - Redação do relatório final

A quarta fase (2º semestre) contará com as tarefas referentes à testagem, avaliação e monitorização dos microsserviços desenvolvidos na fase anterior (tarefas trinta a trinta e três) e será fechada com a redação de um relatório científico (com os resultados da experiência) e com o relatório final do projeto. A testagem envolverá a realização de testes unitários e de integração dos três sistemas. Já a avaliação consistirá, não só, na realização de uma *peer-review* aos modelos de dados de cada sistema, mas também na realização de um teste ao requisito não-funcional "usabilidade".

Prazos realizados nas fases um e dois

A **fase um** teve início a 8 de setembro e foi concluída a 16 de outubro. Nesta fase, a realização das tarefas referentes à análise e investigação às aplicações de microsserviços exemplificativas, foram aquelas que exigiram um dispêndio de tempo maior.

A **fase dois** teve início a 13 de outubro e foi concluída a 11 de janeiro. Já nesta fase, a realização das tarefas referentes à especificação dos casos de uso, ao levantamento de requisitos, à definição da arquitetura e à redação do relatório intermédio, foram aquelas que exigiram um dispêndio de tempo maior.

Previsão da realização das fases três e quatro

Neste momento, está previsto a **fase três** ter um **prazo de realização de dois meses** e a **fase quatro** ter um **prazo de três meses**.

A tarefa referente à definição dos URI, terá um prazo de conclusão de uma semana, e as tarefas referentes à implementação dos sistemas, a um prazo próximo de sete semanas.

As tarefas da fase quatro referentes à testagem dos sistemas, terão um prazo de quatro semanas, a escrita do relatório terá um prazo de quatro semanas e a redação do relatório final terá um prazo de duas semanas.

Concretização das fases três e quatro

Todas as previsões para o cumprimento dos objetivos delineadas no final do primeiro semestre foram cumpridas.

Estas tarefas incluíram, durante o mês de fevereiro, a definição dos recursos da API REST dos três sistemas, a aprendizagem da *framework* ASP.NET Core, e a implementação do sistema Banco.

No mês seguinte, procedeu-se ao desenvolvimento do sistema Loja, à aprendizagem do protocolo gRPC, das *frameworks* Django, Quarkus e Node.js, e à aprendizagem das bases de dados não-relacionais NoSQL (nomeadamente a MongoDB);

Durante o mês de abril foi realizado o desenvolvimento do sistema Publicitaki e respetiva testagem dos três sistemas.

O mês de maio implicou o desenvolvimento da versão de *deployment* para Docker e a aprendizagem do funcionamento da Amazon AWS. Também foi realizada uma investigação às ferramentas *Infrastructure as Code* (IaC) Terraform e CloudFormation, e, posteriormente efetuada a aprendizagem da sintaxe do Terraform. Após a realização destas tarefas, foi implementa a versão de deployment dos três sistemas para a Amazon AWS.

No decorrer dos meses de junho e julho foi realizada a aprendizagem da ferramenta de testagem de carga Locust, efetuados os testes de carga aos três sistemas recorrendo ao Locust, e organizados os ficheiros de *deployment* das versões Docker e Amazon AWS. Foram, também, criados os repositórios GitHub das versões Docker e Amazon AWS do projeto, e redigido o relatório final.

6.2 Metodologia *Waterfall*

Para que um projeto resulte é primordial que os requisitos sejam levantados de uma forma, não apenas, realista, mas também real, para que a definição de qual a metodologia de desenvolvimento a ser seguida seja a mais apropriada. Sistemas reais e similares de *websites* de *e-commerce* como o KuantoKusta foram essenciais para permitir com maior facilidade e clareza o levantamento dos requisitos, o que contribuiu para a decisão final de aplicação da metodologia ***Waterfall*** para o desenvolvimento do projeto. Orientado e supervisionado pelos orientadores Doutores Filipe Araújo e Jorge Cardoso, o projeto foi, então, desenvolvido com base na metodologia ***Waterfall***, que assenta no rigor e disciplina, como forma de evitar e/ou minimizar quaisquer riscos que possam vir a acarretar custos inerentes a uma hipotética alteração aquando do desenvolvimento do projeto. Ao seguir a metodologia *Waterfall* (**Figura** 6-1**)** é imperativo que, logo de início, sejam bem definidas, clarificadas e estabelecidas as exigências e liberdades que devem ser trilhadas ao longo de todo o trabalho, nomeadamente quais as fases a desenvolver e sua ordem, a identificação de todos os requisitos que devem ser cumpridos, as limitações a alterações ao processo, e alterações ao *design* na fase inicial, assim como a promoção de organização das entidades envolvidas.

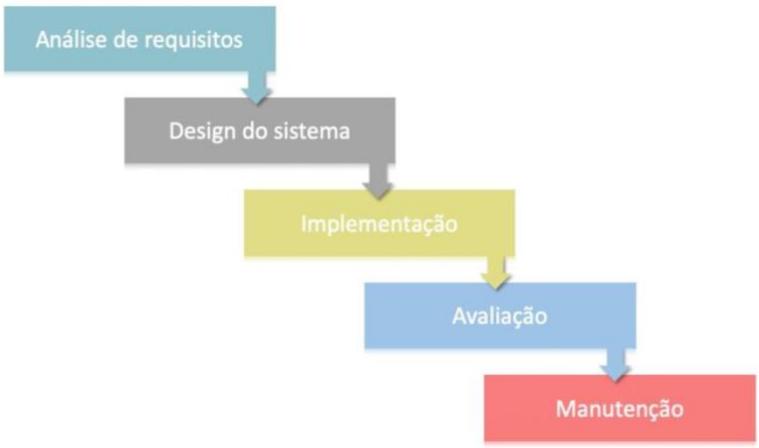


Figura 6-1: Metodologia *Waterfall*

Firmado nestas premissas foi estabelecido a realização de reuniões quinzenais de acompanhamento, orientação e avaliação das tarefas desenvolvidas e delineamento das tarefas para as semanas seguintes.

Capítulo 7
Conclusão

O trabalho, documentado no presente relatório, visou o desenvolvimento de uma aplicação de microsserviços direcionada para a comunidade científica, como resposta à procura de aplicações que possam servir de base a experiências que sejam realistas, de dimensão adequada e de baixo custo.

Neste capítulo são apresentadas, as conclusões decorrentes do trabalho desenvolvido e perspetivas futuras para o avanço do conhecimento e inovação, através de uma síntese da descrição de cada atividade percorrida, desde o levantamento de requisitos até à implementação e conclusão do projeto. Serão, também, sugeridas soluções para o aperfeiçoamento e expansão do trabalho futuro.

7.1 Apreciação final

O projeto foi bastante enriquecedor e gratificante, tanto no aspeto técnico como no pessoal. A nível técnico, foi enriquecedor pela qualidade do *software* desenvolvido, com a implementação de uma arquitetura altamente escalável preparada para *deployment* na *Cloud* e a inclusão de diversas tecnologias. A nível pessoal foi gratificante porque foram adquiridos novos conhecimentos e competências de planeamento e organização de trabalho que muito contribuíram para a boa progressão e conclusão do projeto.

O trabalho foi sendo desenvolvido de forma ordenada e sequencial, respeitando, sempre, as atividades designadas como forma de alcançar, mais rapidamente, os objetivos traçados, como o levantamento e a análise dos requisitos, a arquitetura, tecnologia e testes, até ao seu término com a entrega do relatório.

Antes de dar início ao trabalho, propriamente dito, foram identificados e analisados os diferentes estilos arquiteturais, por forma a compreender as características, benefícios e malefícios de cada um deles e suas inter-relações.

O trabalho iniciou-se com a análise e compreensão do modelo vigente e identificação dos problemas que o envolvem e persistem (falta de escalabilidade, dificuldade na deteção, localização e resolução de falhas, a limitação de tecnologias e funcionalidades, funcionam localmente sem suporte para a *Cloud*, sem ferramentas para a realização de testes de carga, falta de flexibilidade e adaptabilidade (e.g. adaptação consoante o tráfego recebido)), por forma a tirar as ilações necessárias dos pós e contras a ter em consideração, para não cometer os mesmos erros no novo modelo a ser implementado.

Na especificação de requisitos foram identificados os atores e delineados os casos de uso através de diagramas de contexto para uma melhor compreensão de como o sistema se iria apresentar. Também foram detalhados os requisitos funcionais e os requisitos não-funcionais e restrições que são uma importante ajuda para prevenir eventuais problemas de futuro.

Para a definição da arquitetura recorreu-se ao modelo C4 de Simon Brown, utilizando diagramas de contexto, *containers* e componentes para uma mais fácil compreensão das interações entre os intervenientes envolvidos. Foram, igualmente, caracterizadas várias opções tecnológicas, bem como as tecnologias e ferramentas.

Na fase de implementação da aplicação, foi decidido criar duas versões de *deployment*: uma para funcionar localmente, recorrendo ao Docker, e outra para funcionar na *Cloud*, para a plataforma Amazon AWS. A opção de ter múltiplas opções de *deployment* permite uma maior flexibilidade e adaptabilidade do sistema às necessidades e preferências dos utilizadores.

Constatou-se com o *deployment* da aplicação na *Cloud* e os subsequentes testes de carga, que a aplicação demonstra ser capaz de escalar de acordo com a carga, além de ser altamente eficiente em termos de desempenho e apresentar custos de operação na *Cloud* reduzidos. Em cenários com alto volume de pedidos provenientes de vários clientes, a infraestrutura da AWS é capaz de ativar e desativar instâncias conforme necessário, maximizando, assim, os recursos e garantindo uma alocação eficiente dos mesmos para a aplicação. Recorrer a imagens de baixo custo, designadas de *micro*, permitiu aproveitar melhor a capacidade disponível, evitando a alocação desnecessária de recursos em instâncias mais potentes. A possibilidade de dimensionar os recursos atribuídos à aplicação de forma dinâmica na *Cloud* permite um melhor ajuste aos picos de tráfego e garante a disponibilidade e o desempenho adequados da aplicação.

Foi elaborado um cronograma representativo das tarefas definidas (levantamento de requisitos, *design* da arquitetura, implementação da aplicação e realização de testes), com o prazo de concretização estabelecido para cada uma delas. Foi, também, adotada a metodologia *Waterfall* para que o trabalho seguisse uma orientação ordenada e sequencial. Relativamente à gestão do tempo, o levantamento de requisitos, *design* da arquitetura, implementação da aplicação, testes e conclusão do relatório foram cumpridos dentro do prazo estipulado.

7.2 Trabalho futuro

O trabalho reúne as condições para poder, no futuro, vir a ser explorado e aperfeiçoado por outros investigadores. Os principais pontos a serem considerados incluem a realização de **testes de carga mais aprofundados** (avaliar a capacidade da aplicação em lidar com volumes ainda maiores de pedidos), melhorias na **segurança** (análise de vulnerabilidades), o aprofundamento da **monitorização** (com vista a permitir encontrar possíveis pontos de melhoria em termos de tempo de resposta, escalabilidade e eficiência), a expansão da cobertura de testes e a exploração da **injeção de falhas**.

A injeção de falhas pode ser uma abordagem interessante para testar a resiliência do sistema, permitindo a simulação controlada de falhas e a validação da capacidade de recuperação e tolerância a falhas dos microsserviços, tornando a aplicação mais confiável com vista a enfrentar desafios reais do ambiente de produção.

A nova aplicação de microsserviços para fins científicos poderá tornar-se um modelo para o desenvolvimento futuro de aplicações mais eficientes no desempenho, flexíveis e adequadas às necessidades da comunidade científica,

devido às suas características inovadoras, como a modularidade, escalabilidade, arquitetura avançada e novas tecnologias.

Referências

[1] Martin Fowler. "Monolithic Architecture". In: Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002

[2] Sam Newman. "Building Microservices: Designing Fine-Grained Systems". O'Reilly Media, 2015.

[3] Robert Martin. "Clean Architecture: A Craftsman's Guide to Software Structure and Design". Prentice Hall, 2017.

[4] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. "Cloud Computing: Concepts, Technology & Architecture". Prentice Hall, 2013.

[5] Thomas Erl. "SOA Principles of Service Design". Prentice Hall, 2007.

[6] David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, and Sailu Reddy. "HTTP: The Definitive Guide". O'Reilly Media, 2002.

[7] Andrew L. Goettl. "RabbitMQ Essentials". Packt Publishing, 2014.

[8] Andy Stanford-Clark, Arlen Nipper. "MQTT Essentials - A Lightweight IoT Protocol". O'Reilly Media, 2017.

[9] Kasun Indrasiri, Danesh Kuruppu. "gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes". O'Reilly Media, 2018.

[10] Daniel Bryant, Abraham Marín-Pérez, and Chris Richardson. "Microservices Patterns: With Examples in Java". Manning Publications, 2021.

[11] Gerard Tel. "Introduction to Distributed Algorithms". Cambridge University Press, 2000.

[12] Roy Fielding, Richard N. Taylor. "Architectural Styles and the Design of Network-based Software Architectures". University of California, Irvine, 2000.

[13] Craig Walls. "Spring in Action". Manning Publications, 2018.

[14] John Doyle, Jason Porter, Andy Guibert. "Quarkus Cookbook: Harness the Power of Quarkus to Develop Scalable and Efficient Java Applications". Packt Publishing, 2021.

[15] David Herron. "Node.js: Web Development with Node and Express: Leveraging the JavaScript Stack". Manning Publications, 2018.

[16] Adam Freeman. "Pro ASP.NET MVC Framework". Apress, 2019.

[17] Roy Fielding, Richard N. Taylor. "Architectural Styles and the Design of Network-based Software Architectures". University of California, Irvine, 2000.

[18] Grant Allen, Mike Owens, and Apress. "Beginning SQLite for Mobile Development". Apress, 2015.

[19] Paul DuBois. "MySQL Cookbook: Solutions for Database Developers and Administrators". O'Reilly Media, 2014.

[20] Thomas Kyte, and Darl Kuhn. "Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions". Apress, 2010.

[21] Bruce Momjian. "PostgreSQL: Introduction and Concepts". Addison-Wesley Professional, 2001.

[22] Itzik Ben-Gan, Dejan Sarka, and Ron Talmage. "T-SQL Fundamentals". Microsoft Press, 2016.

[23] Jeff Barr. "AWS for Developers For Dummies". For Dummies, 2020.

[24] Adrian Mouat. "Using Docker: Developing and Deploying Software with Containers". O'Reilly Media, 2016.

[25] Kelsey Hightower, Brendan Burns, Joe Beda. "Kubernetes: Up and Running: Dive into the Future of Infrastructure". O'Reilly Media, 2019.

[26] Abraham Marín-Pérez, Isaac Rodríguez-Cuevas, José Manuel Cantera Fonseca. "Performance Testing with Locust: An End-to-End Guide". Packt Publishing, 2020.

[27] Steve Waterworth. Instana. Stan's Robot Shop – a Sample Microservice Application. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/. Acedido em setembro de 2022.

[28] Weaveworks e Container Solutions. Sock Shop - a Microservices Demo Application. https://microservices-demo.github.io. Acedido em setembro de 2022

[29] Cornell University. DeathStarBench - Open-source benchmark suite for cloud microservices. https://github.com/delimitrou/DeathStarBench. Acedido em setembro de 2022

[30] University of Würzburg. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. https://github.com/DescartesResearch/TeaStore. Acedido em setembro de 2022

[31] Justus Bogner. Service-Based WebShop Version 1. https://github.com/xJREB/service-based-web-shop-v1. Acedido em setembro de 2022

[32] Alexander Lukyanchikov. Piggy Metrics. https://github.com/sqshq/piggymetrics. Acedido em setembro de 2022

[33] Szidlovszky Marcel, Evaluating the Effectiveness of Proposed Service-based

Maintainability Metrics for Microservices. https://d-nb.info/1202043526/34. Acedido em outubro de 2022

[34] Microsoft Corporatiom, ASP.NET Documentation. https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0. Acedido em outubro de 2022

[35] Dev Media, Processo de teste ágil x tradicional. https://www.devmedia.com.br/processo-de-teste-agil-x-tradicional/36854. Acedido em outubro de 2022

[36] S. Gillis Alexander, Guide to building an enterprise API strategy. https://www.techtarget.com/searchapparchitecture/definition/RESTful-API. Acedido em outubro de 2022

[37] McKenzie Cameron, Definition of Java Database Connectivity (JDBC). https://www.theserverside.com/definition/Java-Database-Connectivity-JDBC. Acedido em outubro de 2022

[38] Tyson Matthew, What is JPA? Introduction to Java persistence. https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html. Acedido em outubro de 2022

[39] Srivastav Prakhar, Docker for beginners https://docker-curriculum.com. Acedido em outubro de 2022

[40] PhoenixNap, What is Docker?. https://phoenixnap.com/kb/what-is-docker. Acedido em outubro

[41] Wayne Segar, What is OpenTelemetry? An open-source standard for logs, metrics, and traces. https://www.dynatrace.com/news/blog/what-is-opentelemetry-2/. Acedido em dezembro

[42] Red Hat, What is Kubernetes? https://www.redhat.com/en/topics/containers/what-is-kubernetes. Acedido em outubro

[43] Sirish Raghuram, 4 reasons you should use Kubernetes. https://www.infoworld.com/article/3173266/4-reasons-you-should-use-kubernetes.html. Acedido em outubro

[44] GeekHunter, O que é Kubernetes? Tudo que você precisa saber sobre!. https://blog.geekhunter.com.br/kubernetes-a-arquitetura-de-um-cluster/. Acedido em outubro

[45] Microsoft Corporation, Command and Query Responsibility Segregation (CQRS) pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs. Acedido em outubro

[46] Kreps Jay, Introducing Kafka Streams: Stream Processing Made Simple.

[47] Richardson Chris, Introduction to Microservices. https://www.nginx.com/blog/introduction-to-microservices/, 2015

[48] Richards Mark, Microservices vs. Service-Oriented Architecture, 2016

[49] Townsend Erik, The 25-Year History of Service-Oriented Architecture, 2008

[50] Michael L. Ronayne e Erik S. Townsend, Distributed Object Technology at Wells Fargo Bank, 1996

[51] Erik S. Townsend, The Cushing Group, Inc., Distributed Object Technology at Wells Fargo Bank, 1996

[52] Hohpe Gregor, Your Coffee Shop Doesn't Use Two-Phase Commit, 2005

[53] Team Aspecto, What is OpenTelemetry? A Straightforward Guide, 2022

[54] Yevgeniy Brikman. "Terraform: Up & Running: Writing Infrastructure as Code". O'Reilly Media, 2017.

Anexos

Anexo A:Persona

Nome: Alcina Fernandes

Formação: Doutoramento em sistemas de informação e computação,

Profissão: Investigadora de sistemas de informação e computação no Centro de Estudo Tecnológico de Coimbra

Competências: investigadora responsável por vários projetos em ciência da computação, com vários livros e artigos publicados sobre o panorama, presente e futuro, das arquiteturas de sistemas distribuídos e escalabilidade.

Projetos: Coordena uma equipa que investiga a eficiência e o desempenho de diferentes arquiteturas de microsserviços em ambientes distribuídos.

Desafios: Descobrir uma ferramenta que seja capaz de se adaptar e responder às exigências de benchmarking (testes e medições de comparação, em várias linguagens de programação, diferentes tipos de comunicação (síncrona e assíncrona)), e que permita, uma reprodução dos resultados em cenários e contextos distintos, para que as experiências realizadas possam garantir credibilidade e fiabilidade, não obstante a escolha das linguagens de programação ou critérios de comunicação avançados.

Benefícios da nova aplicação: Realização de experiências de comparação, em diferentes cenários de escalabilidade e reprodutibilidade, disponibilidade para executar microsserviços em variadas linguagens de programação e operar com diferentes tipos de comunicação. Pode, também, executar testes e conquistar métricas significativas sobre o desempenho e eficiência dos microsserviços num contexto científico e para além disso, dispõe de autonomia para ajustar a aplicação às exigências da investigação.

Anexo B: Casos de Uso – nível 1 (Publicitaki)

Descrição detalhada dos casos de uso do sistema Publicitaki:

UC1 - Efetuar registo	Permite ao cliente registar-se no sistema com os seus dados pessoais. A criação da conta é obrigatória para o cliente se poder autenticar no sistema.
UC2 - Efetuar autenticação	Permite ao cliente entrar com a sua conta no sistema inserindo as suas credenciais. A criação da conta, referente ao UC1, é obrigatória para o cliente se poder autenticar no sistema.
UC3 - Consultar lista artigos	Permite ao cliente consultar a lista de artigos que pretende visualizar tendo opcionalmente a opção de os poder filtrar pela sua categoria (*e.g.* eletrodomésticos, telemóveis, roupa, ...) ou por uma loja em específico.
UC4 - Consultar artigo	Permite ao cliente consultar os preços de um artigo específico, podendo opcionalmente consultar as suas características, todo o seu histórico de preços, e receber notificações de quedas de preço (o recebimento das notificações implica a autenticação do cliente no sistema). O cliente também terá a opção de consultar o artigo na loja que selecionou, sendo que o sistema deverá encaminhá-lo para o *website* da loja que o cliente escolheu (UC16).
UC5 - Publicar opiniões	Permite ao cliente submeter um registo de satisfação/reclamação relativa à funcionalidade do Publicitaki.
UC6 - Remover artigo	Permite ao administrador remover artigos sempre que encontrar alguma irregularidade, devendo o serviço do Publicitaki notificar a loja dessa ocorrência, tal como descrito pelo UC17.
UC7 - Registar artigo	Permite ao administrador registar novos artigos a pedido das lojas.
UC8 - Consultar notificações de registo de novos artigos	Permite ao administrador consultar notificações das lojas a pedir o registo de novos artigos.
UC9 - Efetuar autenticação	Permite ao administrador efetuar autenticação para poder efetuar os casos de uso UC 6, 7 e 8.
UC10 - Efetuar registo	Permite à Loja registar-se no sistema com os seus dados. A criação da conta é obrigatória para a Loja se poder autenticar no sistema.
UC11 - Efetuar autenticação	Permite à Loja entrar com a sua conta no sistema inserindo as suas credenciais. A criação da conta, referente ao UC10, é obrigatória para a Loja se poder autenticar no sistema.
UC12 - Pagar subscrição	Permite à Loja realizar o pagamento da sua subscrição no sistema a fim de poder efetuar o caso de uso UC13.
UC13 - Associar artigo	Permite à Loja associar o preço de um artigo que vende na sua loja. Este artigo deverá já estar registado anteriormente pelo administrador (no caso de uso UC 7).
UC14 - Solicitar registo novo	Permite à Loja solicitar ao administrador o registo de novos artigos, não

artigo	existentes no sistema.
UC15 - Alterar características artigo existente	Permite ao Loja alterar o preço e as características dos seus artigos registados no sistema.
UC16 - Encaminhar cliente para website da loja	Permite ao serviço do sistema encaminhar o cliente para o website da loja que possui o artigo que escolheu no UC4.
UC17 - Notificar as lojas via email	Permite ao serviço do sistema notificar as lojas, via email, de que um artigo foi removido do sistema pelo administrado quando executado o UC6.
UC18 - Contactar banco para efetuar débito	Permite ao serviço do sistema contactar o sistema representativo do Banco para poder efetuar o débito da subscrição da Loja (referente ao UC12).
UC19 - Notificar cliente de mudança de preço do artigo via email	Permite ao serviço do sistema, aquando da ocorrência do UC15, notificar os clientes que se encontram a seguir um determinado artigo de que o preço desse artigo foi alterado.

Anexo C: Casos de uso - nível 1 (Loja)

Descrição detalhada dos casos de uso do sistema Loja:

UC1 - Inserir descontos semanais	Permite à Loja atribuir, ao cliente registado no sistema, descontos semanais nos artigos que pretende.
UC2 - Registar artigo	Permite à Loja registar, no sistema, um novo artigo.
UC3 - Alterar artigo existente	Permite à Loja alterar as características, como a descrição e preço do artigo no sistema.
UC4 - Consultar vendas	Permite à Loja consultar o número de vendas de cada um dos artigos registados no sistema.
UC5 - Gerir FAQs	Permite à Loja inserir, atualizar ou remover as questões e respetivas respostas para as perguntas mais comuns dos clientes.
UC6 - Efetuar registo	Permite ao cliente registar-se no sistema com os seus dados pessoais. A criação da conta é obrigatória para o cliente se poder autenticar no sistema.
UC7 - Efetuar autenticação	Permite ao cliente autenticar-se no sistema com as suas credenciais. A autenticação será necessária para o cliente poder comprar artigos e poder efetuar também os casos de uso UC 10 e 11.
UC8 - Consultar lista de artigos	Permite ao cliente consultar a lista de artigos para visualizar o artigo desejado. Opcionalmente, poderá filtrar os artigos por categoria.
UC9 - Consultar artigo	Permite ao cliente consultar um artigo específico, podendo opcionalmente consultar as suas características. O cliente tem também a opção de comprá-lo, necessitando, para isso de estar autenticado no sistema (UC 7). O processamento do pagamento pelo cliente será da responsabilidade do serviço do sistema (UC 15).
UC10 - Consultar FAQs	Permite ao cliente consultar as perguntas e respostas mais frequentemente colocadas. As FAQs apresentadas são aquelas que foram registadas no sistema pela Loja (UC5).
UC11 - Publicar opiniões	Permite ao cliente publicar opiniões relativas a um artigo comprado no sistema. Para poder executar este caso de uso o cliente deverá ter feito, anteriormente, a sua autenticação no sistema (UC 7).
UC12 - Consultar descontos	Permite ao cliente consultar os descontos que lhe foram atribuídos pelo sistema (UC 13).
UC13 - Atribuir descontos a clientes	Permite ao serviço do sistema atribuir os descontos semanais que foram registados pela Loja (UC1) aos clientes (UC12).
UC14 - Atualizar artigo no Publicitaki	Permite ao serviço do sistema alterar o preço do artigo no sistema externo “Publicitaki” quando ocorre o caso de uso UC3.
UC15 - Contactar serviço	Permite ao serviço do sistema contactar com o sistema externo “Banco” para

pagamento do banco	poder proceder ao pagamento do artigo por parte do cliente.

Anexo D: Casos de uso – nível 1 (Banco)

Descrição detalhada dos casos de uso do sistema Banco:

UC1 - Efetuar registo	Permite ao cliente do banco registar-se no sistema com os seus dados pessoais. A criação da conta é obrigatória para o cliente se poder autenticar no sistema.
UC2 - Efetuar autenticação	Permite ao cliente do banco autenticar-se no sistema com as suas credenciais. A autenticação será necessária para o cliente poder realizar pagamentos (UC 3), adicionar fundos à sua conta (UC 4), e consultar os movimentos da sua conta (UC 5).
UC3 - Realizar pagamento	Permite ao cliente do banco efetuar um pagamento.
UC4 - Efetuar adição de fundos	Permite ao cliente do banco adicionar fundos (créditos) na sua conta.
UC5 - Consultar movimentos da conta	Permite ao cliente do banco consultar todas as transações e movimentos efetuados na sua conta, tendo a opção de os poder filtrar por data.

Anexo E: Requisitos Funcionais (Publicitaki)

RF01 - Efetuar registo

Descrição operacional: Permite aos atores registarem-se no sistema Publicitaki	
Atores envolvidos: Loja e Cliente	**ID:** RF1 **Sistema Publicitaki**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite aos atores criarem um registo/uma conta no sistema, permitindo-lhes, posteriormente, autenticarem-se para poderem aceder a todas as suas funcionalidades. As funcionalidades menos críticas não requerem o registo do ator.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O cliente gostaria de guardar nos favoritos os artigos para comprar mais tarde ou para aguardar uma baixa de preços, receber notificações de quaisquer atualizações dos artigos das lojas e centralizar toda esta informação na sua conta. A Loja gostaria de fazer, no sistema, uma gestão dos artigos da sua loja de forma centralizada e organizada por categorias.
Dados de entrada (*inputs*) fornecidos	**Cliente**: Tipo de cliente, nome, palavra-passe, e-mail, sexo, idade e preferências (categorias de artigos, marcas e lojas). **Loja**: Tipo de cliente, nome da loja, palavra-passe, e-mail, categorias de artigos que vende.
Descrever contexto operacional	O ator, através do seu browser, acede à página principal do website (representativo do sistema). Na página principal, o ator seleciona a opção de criação de um novo registo e é encaminhado para a página de registo. Na página de registo, é-lhe apresentado um formulário com os parâmetros mencionados nos inputs. Se todos os campos foram corretamente preenchidos com valores válidos, o ator é encaminhado para uma página a informá-lo de que se encontra registado no sistema e redireciona-o para a página principal. Na página principal o ator fica automaticamente autenticado.
Resposta do sistema	A página principal deverá ter uma opção para encaminhar o ator para uma página de registo. Na página de registo um formulário de preenchimento dos parâmetros será apresentado ao ator. Após o preenchimento, o ator confirma o registo, e o sistema guarda a conta do ator.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O sistema irá guardar na sua base de dados a informação submetida pelo ator e notificará o mesmo do sucesso da submissão do registo.

Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado não só para registo, mas também para permitir ao ator usufruir das funcionalidades mais críticas do sistema.

RF02 - Consultar todos os artigos

Descrição operacional: Permite ao ator consultar todos os artigos disponíveis para venda no sistema.	
Atores envolvidos: Cliente	**ID:** RF2 **Sistema Publicitaki**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator consultar todos os artigos que estão para venda em todas as lojas registadas no sistema.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de consultar os artigos por categoria, data de publicação, preço, loja, e marca para escolher aquele que oferece as melhores condições àquilo que procura para então efetuar a compra.
Dados de entrada (*inputs*) fornecidos	Categoria do artigo, marca, data de publicação, garantia, preço, loja
Descrever contexto operacional	O ator, através do seu browser, acede à página principal do website (representativo do sistema). Na página principal, o ator indica o tipo de artigo que pretende pesquisar. Depois de indicado o artigo que se deseja pesquisar, o sistema encaminha o ator para uma página com a listagem de todos os artigos que se possam enquadrar no tipo de artigo indicado. Nesta página o ator pode, opcionalmente, filtrar nesta lista os parâmetros mencionados nos *inputs*. O ator posteriormente seleciona o artigo que pretende e é encaminhado para a página desse artigo.
Resposta do sistema	A página principal deverá ter uma opção para encaminhar o ator para uma página de consulta de artigos. Na página de consulta de artigos o sistema deverá apresentar uma lista com os artigos que se enquadram no tipo de artigos que o ator indicou. Quando o ator seleciona o artigo que pretende visualizar, o sistema encaminha-o para a página desse artigo.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a lista de artigos que se enquadram no tipo de artigo selecionado, tendo em conta as opções de filtragem indicadas.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado pelo ator que através da lista dos artigos selecionará o artigo desejado.

RF03 - Consultar um artigo em específico

Descrição operacional: Permite ao ator consultar um artigo específico.	
Atores envolvidos: Cliente	**ID:** RF3 **Sistema Publicitaki**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator consultar um artigo específico que está para venda em todas as lojas registadas no sistema.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de verificar as lojas onde o artigo está mais barato, quais as lojas que o têm, guardá-lo nos favoritos para, eventualmente, comprar quando o preço estiver mais baixo, consultar em maior detalhe as suas características, o seu histórico de preços e receber via e-mail notificações de descida de preço.
Dados de entrada (*inputs*) fornecidos	Identificador do artigo selecionado
Descrever contexto operacional	Depois de selecionado o artigo que pretende visualizar a partir da lista referente ao RF2, o sistema encaminha o ator para a página do artigo. Nesta página o ator poderá consultar as características do artigo (nome, marca, garantia, especificações), o seu histórico de preços, e as lojas que o têm disponível com os respetivos preços. O ator posteriormente seleciona a loja onde pretende comprar o artigo e é encaminhado para a página desse artigo na respetiva loja.
Resposta do sistema	O sistema deverá apresentar a página do artigo com as suas respetivas características (nome, marca, garantia, especificações), o histórico de preços, e as lojas que o têm disponível com os respetivos preços. Quando o ator seleciona a loja onde pretende comprar o artigo, o sistema encaminha-o para a página desse artigo na respetiva loja.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será o artigo selecionado.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado pelo ator para poder escolher a loja onde pretende comprar o artigo.

RF04 - Publicitar opiniões

Descrição operacional: Permite ao ator publicar opiniões sobre o sistema.	
Atores envolvidos: Cliente	**ID:** RF4 **Sistema Publicitaki**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator expressar as suas opiniões sobre a sua operacionalidade e diversidade de produtos para venda.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de expressar as suas opiniões sobre a sua operacionalidade e diversidade de produtos para venda, para melhoria do sistema, eventualmente introdução de novas funcionalidades e análise e avaliação por parte dos outros clientes.
Dados de entrada (*inputs*) fornecidos	Título da opinião, a descrição, número de estrelas (1 a 5) com a respetiva data e hora.
Descrever contexto operacional	Durante a utilização do sistema o ator vai detetando algumas falhas comparando as mesmas com opiniões já publicadas. Seguidamente o ator vai à página de publicação e publica a sua opinião. O administrador do sistema analisa a opinião rececionada e avalia se deverá ou não avançar com o pedido, enviando uma notificação para o cliente a informá-lo da sua decisão.
Resposta do sistema	O sistema deverá apresentar uma página de publicação de opiniões com um formulário por preencher com os inputs mencionados acima. Depois de submetida a opinião pelo ator, o sistema regista-a para poder ser posteriormente analisada pelo administrador.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a opinião do ator.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados e será utilizado para análise por parte do administrador e dos clientes.

RF05 - Gerir artigos

Descrição operacional: Permite aos atores consultar, inserir, remover ou atualizar artigos registados no sistema.	
Atores envolvidos: Loja e Administrador	**ID:** RF5 **Sistema Publicitaki**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite aos atores inserir, remover ou atualizar artigos das lojas registados no sistema.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	**Administrador:** O ator gostaria de poder consultar a lista de todos os artigos registados no sistema e quando notificado pelas lojas inserir novos artigos e/ou removê-los e atualizá-los. **Loja:** O ator gostaria de poder consultar a lista de todos os seus artigos registados no sistema, atualizar os respetivos preços, e notificar o administrador para inserir novos artigos e/ou removê-los.
Dados de entrada (*inputs*) fornecidos	**Administrador:** **Inserção/Alteração de artigo:** id, fotografia, nome, especificações e marca **Remoção:** identificador do artigo **Loja:** **Inserção/Alteração de artigo:** preço **Remoção:** identificador do artigo
Descrever contexto operacional	A loja gostaria de poder registar o seu artigo no sistema, no entanto, este ainda não se encontra criado, por isso, a loja seleciona a opção de aviso que faz com que o sistema notifique o administrador para este criar o novo artigo. O administrador regista o novo artigo no sistema com os inputs mencionados e na conta da loja aparece uma notificação a dizer que o artigo já foi introduzido no sistema. A loja insere no artigo o preço de venda.
Resposta do sistema	O sistema quando avisado pela loja de que um artigo não existe deverá notificar o administrador para o registar no sistema. O sistema deverá também permitir que a loja associe o preço ao artigo.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será o artigo criado pelo administrador e o preço associado pela loja.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados e será utilizado para consulta por parte dos clientes.

RF06 - Efetuar pagamento de subscrição

Descrição operacional: Permite ao ator efetuar o pagamento da utilização do sistema.	
Atores envolvidos: Loja	**ID:** RF6 **Sistema Publicitaki**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator realizar o pagamento da utilização da plataforma para poder anunciar os artigos da sua loja.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de efetuar o pagamento das suas publicações no sistema através de débito direto.
Dados de entrada (*inputs*) fornecidos	Número do IBAN da loja, quantia a pagar, Número do IBAN do Publicitaki
Descrever contexto operacional	O ator autenticado na sua conta efetua os pagamentos dentro do próprio sistema fornecendo para isso os seus dados bancários. O sistema depois contacta o sistema externo "Banco" para efetuar o débito. O sistema depois de receber o pagamento envia uma notificação a confirmar o pagamento.
Resposta do sistema	O sistema apresenta um formulário para preenchimento das coordenadas bancárias. Depois de o ator preencher os dados bancários, o sistema fica automaticamente autorizado a proceder aos débitos do ator. O sistema depois de receber o pagamento envia uma notificação na área da sua conta no sistema, a confirmar o pagamento.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a efetivação do pagamento.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será utilizado para efetuar o pagamento da subscrição.

Anexo F: Requisitos Funcionais (Loja)

RF09 – Cliente efetua uma encomenda

Descrição operacional: Permite ao ator efetuar uma encomenda.	
Atores envolvidos: Cliente e serviço da loja	**ID:** RF9 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator comprar os artigos que pretende na Loja.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de poder encomendar os artigos que pretende comprar na Loja.
Dados de entrada (*inputs*) fornecidos	Artigos selecionados para compra e respetivas quantidades.
Descrever contexto operacional	A loja regista os artigos disponíveis para compra no sistema Loja. O cliente consulta a lista de artigos disponíveis e adiciona ao carrinho aqueles que pretende comprar. Depois de selecionados, procede com o efetivação da compra.
Resposta do sistema	Depois de efetuada a encomenda, o sistema envia uma mensagem para o Banco para efetuar o pagamento da encomenda, e fica a aguardar pela confirmação, colocando a encomenda no estado pendente. Depois de recebida a confirmação por parte do Banco, o sistema Loja coloca a encomenda no estado finalizada.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a alteração do estado da encomenda de pendente para finalizada.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados e será utilizado para visualização do cliente que efetuou a encomenda.

RF10 - Gerir artigos

Descrição operacional: Permite ao ator consultar, inserir, remover ou atualizar artigos registados no sistema.	
Atores envolvidos: Loja	**ID:** RF10 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator inserir, remover ou atualizar artigos na sua loja.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de poder consultar a lista de todos os seus artigos registados no sistema, inserir, e atualizar as suas características e preços.
Dados de entrada (*inputs*) fornecidos	**Inserção/Alteração de artigo:** nome, marca, categoria, fotografia, características **Remoção:** identificador do artigo
Descrever contexto operacional	A loja regista o novo artigo no sistema com os inputs mencionados ficando automaticamente disponível no sistema para venda aos clientes.
Resposta do sistema	O sistema deverá apresentar um formulário com os inputs mencionados. Depois deverá informar o ator de que o registo do artigo foi bem-sucedido.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será o artigo criado pelo ator.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados e será utilizado para consulta por parte dos clientes.

RF11 - Gerir Vendas

Descrição operacional: Permite ao ator gerir as vendas dos artigos disponíveis na loja.	
Atores envolvidos: Loja	**ID:** RF5 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator gerir as vendas dos artigos disponíveis na loja.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de poder gerir as suas vendas desde a receção das encomendas até à sua entrega. Também gostaria de gerir eventuais devoluções ou trocas. A inserção dos artigos na loja, atribuição de preço, descontos e formas de pagamento e escolha dos locais e formas de entrega seriam igualmente geridas pelo ator.
Dados de entrada (*inputs*) fornecidos	Número de cliente, número da encomenda, data da encomenda, nome e marca do artigo
Descrever contexto operacional	O ator gostaria de poder acompanhar as suas vendas de início ao fim. O sistema envia uma notificação da entrada de uma nova encomenda com a identificação do cliente, artigo encomendado, número da encomenda e data. O ator, depois de informado pelo Banco do pagamento da encomenda, entra na página encomendas e seleciona a encomenda pelo número que lhe foi atribuído pelo sistema e prepara-a para envio. O ator regista no sistema o envio e notifica o cliente do seu envio. Depois de confirmada a entrega pelo transportador o ator envia uma notificação ao cliente para este registar no sistema a sua opinião sobre o artigo comprado.
Resposta do sistema	O sistema sempre que receciona uma encomenda deverá atribuir-lhe, sempre, um número de encomenda que vai associar ao número de cliente.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será toda a gestão relacionada com as vendas dos artigos.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados e será utilizado para venda dos artigos.

RF12 - Consultar opiniões

Descrição operacional: Permite ao ator consultar opiniões sobre os artigos comprados.	
Atores envolvidos: Cliente, Loja	**ID:** RF4 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite aos atores consultarem as opiniões dos clientes sobre os artigos comprados
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	Cliente: o ator gostaria de consultar as opiniões dos outros clientes por data e artigo Loja: o ator gostaria de consultar as opiniões de todos os clientes por número de cliente, datas e artigos
Dados de entrada (*inputs*) fornecidos	Título da opinião, a descrição, número de estrelas (1 a 5) com a respetiva data e hora da publicação.
Descrever contexto operacional	Antes de efetuar uma encomenda o ator vai à página de consulta de opiniões e consulta as opiniões dos outros clientes sobre o artigo selecionado e decide se deve ou não avançar com a efetivação da encomenda.
Resposta do sistema	O sistema deverá apresentar uma página de consultas de opiniões com uma lista de opiniões dadas por vários clientes, organizada por ordem de chegada, com os inputs acima mencionados de cada cliente
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a consulta das opiniões dadas pelos clientes.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado servirá para análise e avaliação do artigo selecionado.

RF13 - Consultar todos os artigos

Descrição operacional: Permite ao ator consultar todos os artigos disponíveis para venda no sistema.	
Atores envolvidos: Cliente	**ID:** RF14 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator consultar todos os artigos que estão para venda na loja.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de consultar os artigos por categoria, data de publicação, preço, e marca para escolher aquele que mais lhe agrada, para, eventualmente, efetuar uma compra.
Dados de entrada (*inputs*) fornecidos	Categoria do artigo, marca, data de publicação, garantia e preço.
Descrever contexto operacional	Na página principal, o ator indica o tipo de artigo que pretende pesquisar. Depois de indicado o artigo que se deseja pesquisar, o sistema encaminha o ator para uma página com a listagem de todos os artigos que se possam enquadrar no tipo de artigo indicado. Nesta página o ator pode, opcionalmente, filtrar nesta lista os parâmetros mencionados nos *inputs*. O ator posteriormente seleciona o artigo que pretende e é encaminhado para a página desse artigo.
Resposta do sistema	A página principal deverá ter uma opção para encaminhar o ator para uma página de consulta de artigos. Na página de consulta de artigos o sistema deverá apresentar uma lista com os artigos que se enquadram no tipo de artigos que o ator indicou. Quando o ator seleciona o artigo que pretende visualizar, o sistema encaminha-o para a página desse artigo.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a lista de artigos que se enquadram no tipo de artigo selecionado, tendo em conta as opções de filtragem indicadas.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado pelo ator que através da lista dos artigos selecionará o artigo desejado.

RF14 e 15 - Consultar um artigo em específico

Descrição operacional: Permite ao ator consultar um artigo específico.	
Atores envolvidos: Cliente	**ID:** RF3 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator consultar um artigo específico que está para venda.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de verificar qual o artigo que está mais barato, guardá-lo nos favoritos para, eventualmente, comprar quando o preço estiver mais baixo, consultar em maior detalhe as suas características e preço e receber via e-mail notificações de descida de preço.
Dados de entrada (*inputs*) fornecidos	Identificador do artigo selecionado
Descrever contexto operacional	Depois de selecionado o artigo que pretende visualizar a partir da lista referente ao RF14, o sistema encaminha o ator para a página do artigo. Nesta página o ator poderá consultar as características do artigo (nome, marca, garantia, especificações), e o preço.
Resposta do sistema	O sistema deverá apresentar a página do artigo com as suas respetivas características (nome, marca, garantia, especificações) e o respetivo preço. Quando o ator seleciona a opção comprar, o sistema encaminha-o para a página de pagamento.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será o artigo selecionado.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado pelo ator para poder escolher o artigo que pretende comprar.

RF16 - Efetuar registo

Descrição operacional: Permite aos atores registar-se na loja	
Atores envolvidos: Cliente	**ID:** RF1 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator criar um registo/uma conta no sistema, permitindo-lhe, posteriormente, autenticar-se para poder aceder a todas as suas funcionalidades. As funcionalidades menos críticas não requerem o registo do ator.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O cliente gostaria de guardar nos favoritos os artigos para comprar mais tarde ou para aguardar uma baixa de preços, receber notificações de quaisquer atualizações dos artigos das lojas e centralizar toda esta informação na sua conta.
Dados de entrada (*inputs*) fornecidos	**Cliente**: nome, palavra-passe, e-mail, sexo, idade, morada, telemóvel e preferências (categorias de artigos, marcas e lojas).
Descrever contexto operacional	O ator, através do seu browser, acede à página principal do website (representativo do sistema). Na página principal, o ator seleciona a opção de criação de um novo registo e é encaminhado para a página de registo. Na página de registo, é-lhe apresentado um formulário com os parâmetros mencionados nos inputs. Se todos os campos foram corretamente preenchidos com valores válidos, o ator é encaminhado para uma página a informá-lo de que se encontra registado no sistema e redireciona-o para a página principal. Na página principal o ator fica automaticamente autenticado.
Resposta do sistema	A página principal deverá ter uma opção para encaminhar o ator para uma página de registo. Na página de registo um formulário de preenchimento dos parâmetros será apresentado ao ator. Após o preenchimento, o ator confirma o registo, e o sistema guarda a conta do ator.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O sistema irá guardar na sua base de dados a informação submetida pelo ator e notificará o mesmo do sucesso da submissão do registo.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado não só para registo, mas também para permitir ao ator usufruir das funcionalidades mais críticas do sistema.

RF17 - Editar dados pessoais

Descrição operacional: Permite ao ator editar os seus dados pessoais	
Atores envolvidos: Cliente	**ID:** RF1 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator editar os seus dados pessoais, como a morada, número de telemóvel, e-mail, preferências, dados bancários, palavra-passe.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O cliente gostaria de editar os seus dados pessoais sempre que o desejasse e receber uma notificação com a confirmação das edições.
Dados de entrada (*inputs*) fornecidos	**Cliente**: nome, palavra-passe, e-mail, sexo, idade, morada, telemóvel, dados bancários e preferências (categorias de artigos, marcas).
Descrever contexto operacional	Na página da sua conta o ator seleciona a opção Editar e é encaminhado para a página Editar. Na página Editar, é-lhe apresentado um formulário com os parâmetros mencionados nos inputs. O ator preenche os campos que deseja editar e é encaminhado para uma página que o informa de que os parâmetros editados foram bem-sucedidos e redireciona-o para a página da sua conta. Na página da conta do ator os dados já se encontram editados.
Resposta do sistema	A página da conta do ator deverá ter a opção Editar para encaminhar o ator para a página de Editar. Na página Editar aparece o formulário anteriormente preenchido, onde o ator deverá editar os parâmetros que assim entender. Após editar os Parâmetros o ator confirma a edição e o sistema guarda os parâmetros editados
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O sistema irá guardar na sua base de dados a edição de dados submetida pelo ator e notificará o mesmo do sucesso da submissão dos parâmetros editados.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado para a edição de alguns dos parâmetros mencionados no input

RF18 - Consultar encomendas

Descrição operacional: Permite ao ator consultar todas as encomendas realizadas, pendentes, devolvidas e trocadas.	
Atores envolvidos: Cliente	**ID:** RF2 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator consultar na sua conta todas as encomendas realizadas, pendentes, devolvidas e trocadas.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de consultar, pelas datas, as encomendas, as devoluções e as trocas já encerradas, bem como as encomendas, as devoluções e as trocas ainda pendentes.
Dados de entrada (*inputs*) fornecidos	Encomendas, devoluções e trocas já encerradas datadas Encomendas, devoluções e trocas pendentes datadas
Descrever contexto operacional	Na página da sua conta o ator seleciona a opção encomendas e é encaminhado para a página encomendas. Na página encomendas, são-lhe apresentadas as opções de escolha de consulta de encomendas, devoluções e trocas já encerradas por data e também as opções de consulta de encomendas, devoluções e trocas pendentes por data. O ator escolhe a opção desejada e faz a sua consulta. Terminada a consulta o ator pode voltar para a página da sua conta ou voltar para a página principal.
Resposta do sistema	A página da conta do ator deverá ter a opção de consulta de encomendas para encaminhar o ator para a página de consultar encomendas. Na página de consulta de encomendas o sistema deverá apresentar opções de escolha de consulta de encomendas, devoluções e trocas já encerradas por data e também as opções de consulta de encomendas, devoluções e trocas pendentes por data. Quando o ator seleciona uma das opções que pretende consultar, o sistema encaminha-o para a página dessa opção.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido serão as consultas, por datas, das encomendas, devoluções e trocas já terminadas e as consultas, por datas, das encomendas, devoluções e trocas pendentes
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output é usado pelo ator através das opções de consultas, por datas, das encomendas, devoluções e trocas terminadas e pendentes.

RF19 - Publicar opiniões

Descrição operacional: Permite ao ator publicar opiniões sobre os artigos comprados.	
Atores envolvidos: Cliente	**ID:** RF4 **Sistema Loja**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator expressar as suas opiniões sobre os artigos comprados
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de expressar as suas opiniões sobre os artigos já comprados, como a qualidade, características, preço, a demora na receção, a fidelidade à fotografia, se reececionou com algum defeito e se precisou de trocar ou devolver por alguma insatisfação
Dados de entrada (*inputs*) fornecidos	Título da opinião, a descrição, número de estrelas (1 a 5) com a respetiva data e hora da publicação.
Descrever contexto operacional	Depois de realizadas as encomendas o sitema envia notificações para o ator dar as suas opiniões relativamente aos artigos comprados. O ator vai à página de publicação e publica a sua opinião que fica disponível para todos os clientes verem. A loja será notificada pelo sistema de que o ator publicou a sua opinião e quando ela não é favorável a loja notifica o ator, por e-mail e SMS.
Resposta do sistema	O sistema deverá apresentar uma página de publicação de opiniões com um formulário por preencher com os inputs mencionados acima. Depois de submetida a opinião pelo ator, o sistema regista-a e notifica a loja.
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a opinião do ator.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados para ser visualizado pelos clientes e analisado pela loja.

Anexo G:Requisitos Funcionais (Banco)

RF20 - Gerir a conta do cliente

Descrição operacional: Permite ao ator consultar, inserir, remover ou atualizar a conta do cliente registada no sistema.	
Atores envolvidos: Banco	**ID:** RF5 **Sistema Banco**

Considerações operacionais	**Resposta**
Descrição geral da funcionalidade	Esta funcionalidade permite ao ator inserir, remover ou atualizar a conta do cliente registada no sistema. Também permite enviar ao cliente da loja as referências para pagamento da encomenda.
O que o(s) ator(es) gostaria(m) de poder fazer no futuro	O ator gostaria de poder consultar a conta do cliente da loja que está registada no sistema, bem como inserir novos dados desse cliente, atualizá-los ou encerrar a conta. Também gostaria de poder enviar ao cliente da loja as referências para pagamento da encomenda.
Dados de entrada (*inputs*) fornecidos	Cartão de cidadão, fotografia, nome, idade, sexo, IBAN do cliente da loja, montante a debitar ao cliente da loja e IBAN da loja para creditar o valor na sua conta. O número da encomenda também é disponibilizado para o ator enviar referências de pagamento no caso do cliente da loja querer pagar por multibanco.
Descrever contexto operacional	O ator recebe os parâmetros descritos no input e debita o montante. O Banco notifica o cliente de que já recebeu o montante. O cliente(loja) envia uma notificação por e-mail e SMS para o seu cliente onde confirma a receção do pagamento.
Resposta do sistema	O sistema notifica o cliente do sucesso da operação de pagamento enviando um e-mail e SMS
Dados de saída (*outputs*) produzidos como resultado da operação no sistema.	O output produzido será a gestão da conta do cliente.
Quem utilizará os *outputs* produzidos e para que serão utilizados	O output usado será gravado na base de dados e será utilizado para futuros pagamentos.

Anexo H: Restrições de Negócio

RN1 - Duração do desenvolvimento

Descrição	**ID:** RN1
A aplicação deverá ser desenvolvida num período máximo de 10 meses, com entrega prevista em junho de 2023.	

RN2 - Tecnologias de carácter gratuito

Descrição	**ID:** RN2
As tecnologias usadas não devem atingir os limites que superem a sua gratuitidade.	

RN3 - Mecanismo de autenticação

Descrição	**ID:** RN3
Cada ator terá a sua conta para poder aceder aos sistemas dentro do que o seu “role” permite. Cada ator possuirá o seu *role* no sistema (cliente, administrador ou dono de loja) o que o restringirá apenas às suas funcionalidades no sistema.	

Anexo I: Restrições Técnicas

RT1 - Sistemas operativos

Descrição	**ID:** RT1
As aplicações deverão estar encapsuladas em *containers* com sistemas operativos UNIX.	

RT2 - Plataformas

Descrição	**ID:** RT2
A aplicação deverá funcionar corretamente em dispositivos móveis e em computadores Desktop.	

RT3 - Arquitetura

Descrição	**ID:** RT3
A arquitetura dos sistemas a desenvolver será baseada em microsserviços.	

RT4 - Proteção de palavras-passe

Descrição	**ID:** RT4
Palavras-passe dos utilizadores deverão ser seguras e deverão também estar encriptadas.	

RT5 - Tecnologias envolvidas

Descrição	**ID:** RT5
As aplicações deverão estar encapsuladas em containers Docker e deverão ser orquestradas recorrendo ao Amazon AWS.	

RT6 - Padrão de comunicação

Descrição	**ID:** RT6
As aplicações deverão obedecer ao padrão de mensagens *publish-subscribe*.	

Anexo J: Docker-compose

Docker-compose da Loja

Descrito o ficheiro docker-compose do **Publicitaki**, prossegue-se para a explicação do docker-compose da **Loja**.

A **Figura** J-1 contém um excerto do código do ficheiro docker-compose utilizado na construção do sistema Loja.

```yaml
version: '3'
services:

  #---- NOTIFICATIONS SERVICE [Python] ----
  store-notifications-db:
    image: postgres:15.2-alpine
    container_name: store_notifications_db
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      POSTGRES_DB: notifications-db
    ports:
      - "5000:5432"
    volumes:
      - ./store_notifications_db:/var/lib/postgresql/data/

  store-notifications:
    build: ./store_notifications
    container_name: store_notifications
    links:
      - store-notifications-db
    depends_on:
      - store-notifications-db
    restart: on-failure
    ports:
      - "5001:8000"
    command: sh -c "python manage.py makemigrations && python manage.py migrate && python manage.py runserver 0.0.0.0:8000"
    environment:
      DB_ENGINE: django.db.backends.postgresql
      DB_HOST: store-notifications-db
      DB_PORT: 5432
      DB_NAME: notifications-db
      DB_USER: admin
      DB_PASSWORD: admin

  # ---- USERS SERVICE [Java] ----
  store-users-db:
    image: postgres:15.2-alpine
    container_name: store_users_db
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      POSTGRES_DB: users-db
      POSTGRES_SSLMODE: disable
```

Figura J-1: Excerto do ficheiro Docker-compose do sistema Loja

Uma análise à **Figura** J-1 permite identificar quais os microsserviços que compõe o sistema Loja como sejam o ***store-notifications-db*** (correspondente à base de dados do microsserviço responsável por gerir as notificações), ***store-notifications*** (microsserviço de gestão de notificações), e parcialmente ***store-users-db*** (base de dados do microsserviço de gestão dos utilizadores). Os microsserviços ***store-users*** (microsserviço de gestão dos utilizadores e que comunica via gRPC), ***store-products*** (responsável pela gestão dos artigos e das encomendas), ***store-products-db*** (base de dados do microsserviço ***store-products***) e ***store-frontend*** (responsável pela camada visual da aplicação) também fazem parte do sistema Loja. Tal como o ficheiro do Publicitaki, o ficheiro da Loja foi escrito de acordo com a versão três do Docker-compose.

Seguidamente, encontra-se descrito, com maior detalhe, os serviços presentes no ficheiro docker-compose da **Loja**, representados parcialmente pela **Figura** J-1:

- **Store-notifications-db**: *container* denominado por *store_notifications_db*, baseado na **imagem** com a versão 15.2 do **SGBD PostgreSQL**. Através das variáveis de ambiente reservadas pelo PostgresSQL é possível configurar o nome e palavra-passe do utilizador e nome da base de dados, e expor o seu porto 5432, através de *port-forwarding*, para o ***localhost*** no **porto 5000**. O *container* possui um volume dedicado a fim de permitir a persistência da informação contida na base de dados.
- **Store-notifications**: *container* denominado por *store_notifications*, contém uma aplicação **Django**, e é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. Este ficheiro Dockerfile define a utilização de uma imagem baseada na versão **Alpine do Python 3.11.2**, instala as dependências definidas no ficheiro *requirements.txt* recorrendo ao comando *pip*, instala o cliente do PostgreSQL, e copia código da aplicação para o *container*. O *container* expõe, igualmente, o porto 8000 através de *port-forwarding* para o ***localhost*** no **porto 5001**. As variáveis de ambiente permitem a comunicação com o *container* que contém a base de dados deste microsserviço (o *container store-notifications-db*). Existe no docker-compose um comando que permite, através da *Shell*, a criação e a realização das migrações para que possam ser aplicadas à base de dados.
- **Store-users-db**: *container* denominado de *store_users_db*, baseado na **imagem** com a versão 15.2 do **SGBD PostgreSQL**. Através de variáveis de ambiente, com os nomes reservados *POSTGRES*, é possível definir o nome da base de dados, utilizador, palavra-passe e ativar ou desativar o modo SSL. O *container* conta com um volume dedicada para persistir a informação contida na base de dados e expõe o seu porto 5432, através de *port-forwarding*, para o ***localhost*** no **porto 5002**;
- **Store-users**: *container* denominado de *store_users*, que contém uma aplicação **Maven** preparada para atender pedidos gRPC, e que é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. Este ficheiro Dockerfile define a utilização de uma imagem baseada na versão **11 do OpenJDK**, copia o código da aplicação para o *container*, e expõe o porto 50051 (porto por defeito do gRPC). O *container* encontra-se configurado para sempre que ocorra uma avaria volte a reiniciar. O *container* expõe o porto 50051 através de *port-forwarding* para o ***localhost*** no **porto 5003**.
- **Store-products-db**: *container* denominado de *store_products_db*, baseado na **imagem** com a versão 8.0.32 do **SGBD MySQL**. Através de variáveis de ambiente, com os nomes reservados *MYSQL*, é possível definir o nome da base de dados, utilizador *root* e palavra-passe. O *container* possui um volume destinado a persistir a informação contida na base de dados e expõe o porto 3306 do MySQL, através de *port-forwarding*, para o ***localhost*** no **porto 5004**;
- **Store-products**: *container* denominado de *store_products*, contém uma aplicação **Django**, e é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. O Dockerfile utilizado neste microsserviço baseia-se numa imagem com a versão **3.11** do **Python**, copia código para o *container*, instala as dependências necessárias (como o Django) recorrendo ao comando *pip* e expõe o porto 8000. O *container* expõe o porto 8000 através de *port-forwarding* para o ***localhost*** no **porto 5005**.

- **Store-frontend**: *container* denominado por *store_frontend* é **responsável pela componente de interação do utilizador com a aplicação**, construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. O Dockerfile utilizado neste microsserviço baseia-se numa imagem com a **versão 7** da *framework* **.NET**, copia o código para o *container*, e expõe o porto 80. O *container* contém uma aplicação **ASP.NET Core MVC** que expõe o porto 80 através de *port-forwarding* para o ***localhost*** no **porto 5006**.

Docker-compose do Banco

Especificados os ficheiros docker-compose do Publicitaki e da **Loja**, prossegue-se para a explicação do Docker-compose da **Banco**.

A **Figura** J-2 contém um excerto do código do ficheiro Docker-compose utilizado na construção do sistema Banco.

```yaml
version: '3'
services:

  bank-db:
    image: mysql:8.0.32
    container_name: bank_db
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_DATABASE: bank_clients
      MYSQL_USER: tiago
      MYSQL_PASSWORD: 1234
    ports:
      - 6000:3306
    volumes:
      - ./bank_db:/var/lib/mysql

  bank-notifications:
    depends_on:
      - bank-db
    build: ./bank_notifications
    container_name: bank_notifications
    restart: on-failure
    ports:
      - 6002:8080

  bank-clients:
    depends_on:
      - bank-db
      - bank-notifications
    build: ./bank_clients
    container_name: bank_clients
    restart: on-failure
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://bank_db:3306/bank_clients
      SPRING_DATASOURCE_USERNAME: tiago
      SPRING_DATASOURCE_PASSWORD: 1234
    ports:
      - 6001:8080
```

Figura J-2: Excerto do ficheiro Docker-compose do sistema Banco

A análise à **Figura** J-2 permite verificar, parcialmente, a constituição do sistema Banco pelos microsserviços ***bank-db*** (correspondente à base de dados do banco), ***bank-notifications*** (microsserviço de gestão de notificações), e ***bank-clients*** (microsserviço de gestão de clientes e transações do banco). O microsserviço ***bank-frontend*** (responsável pela camada visual da

aplicação) também faz parte do sistema Banco. Tal como o ficheiro da Loja, o ficheiro do Banco foi escrito de acordo com a versão três do docker-compose.

Seguidamente, encontra-se descrito, em maior detalhe, os serviços presentes no ficheiro docker-compose do Banco, representados parcialmente pela **Figura** J-2Figura 5-2:

- **Bank-db**: *container* denominado por *bank_db*, baseado na **imagem** com a versão 8.0.32 do **SGBD MySQL**. Através das variáveis de ambiente reservadas pelo MySQL é possível configurar o nome e palavra-passe do utilizador e nome da base de dados, e expõe o seu porto 3306, através de *port-forwarding*, para o ***localhost*** no **porto 6000**. O *container* possui um volume dedicado a fim de permitir a persistência da informação contida na base de dados.
- **Bank-notifications**: *container* denominado por *bank_notifications*, contém uma aplicação **Spring Boot**, e é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. Este ficheiro Dockerfile define a utilização de uma imagem baseada na versão **11 do OpenJDK**, e copia o código da aplicação para o *container*. O *container* expõe, igualmente, o porto 8080 através de *port-forwarding* para o ***localhost*** no **porto 6002**.
- **Bank-clients**: *container* denominado de *bank_users*, que contém uma aplicação **Spring Boot**, depende dos containers *bank-db* e *bank-notifications* antes de ser iniciado, e é construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. Este ficheiro Dockerfile define a utilização de uma imagem baseada na versão **11 do OpenJDK**, copia o código da aplicação para o *container*, e expõe o porto 8080. O *container* encontra-se configurado para que sempre que ocorra uma avaria volte a reiniciar. O *container* expõe o porto 8080 através de *port-forwarding* para o ***localhost*** no **porto 6001**.
- **Bank-frontend**: *container* denominado por *bank_frontend* é **responsável pela componente de interação do utilizador com a aplicação**, construído a partir do Dockerfile que se encontra na diretoria que contém o código deste microsserviço. O Dockerfile utilizado neste microsserviço baseia-se numa imagem com a **versão 6** da *framework* **.NET**, copia código o *container*, e expõe o porto 80. O *container* contém uma aplicação **ASP.NET Core MVC** que expõe o porto 80 através de *port-forwarding* para o ***localhost*** no **porto 6003**.

Anexo K: Funcionamento dos sistemas

Sistema Loja

Efetuada a apresentação do Banco, prossegue-se para a presentação do sistema Loja.

Para aceder à página Loja, o utilizador, no seu navegador, deverá inserir o endereço do *endpoint* do microsserviço *store-frontend*. Ao aceder à página, o utilizador, visualizará uma página idêntica àquela exibida na **Figura** K-1.

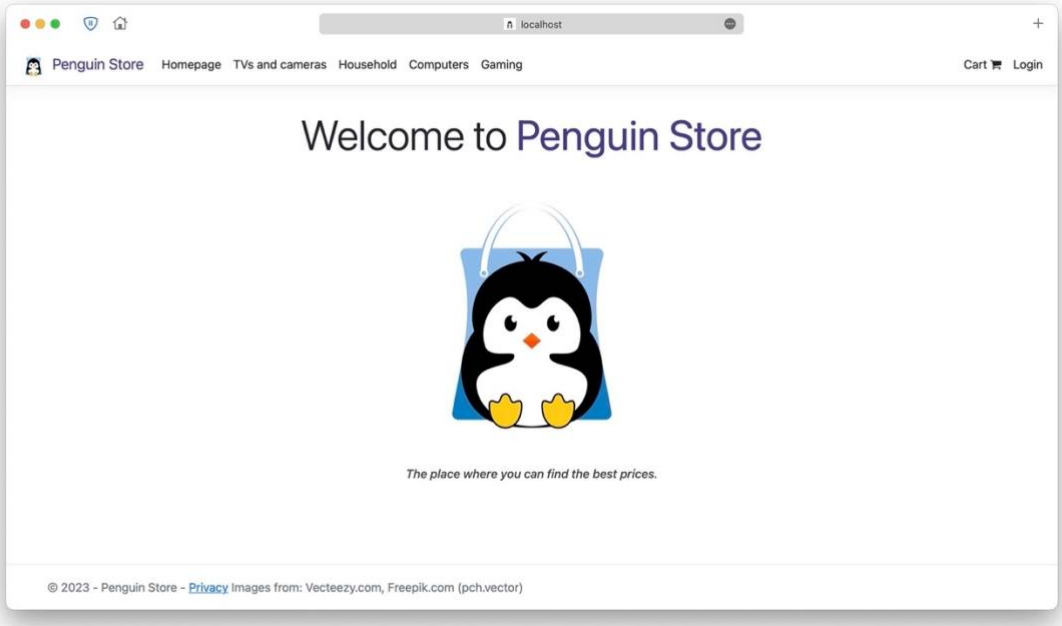


Figura K-1: Página principal do sistema Loja

Este sistema simula um *website* de uma Loja de eletrodomésticos, onde o cliente da loja poderá comprar diversos tipos de artigos desde televisões a videojogos.

Analisando a barra de navegação é possível observar-se a existência de separadores correspondentes ao tipo de artigos que a loja oferece. No lado direito da barra verifica-se a existência de dois separadores, correspondentes ao carrinho (*cart*) e à página da conta do cliente, que neste caso ainda não está autenticado no sistema (*login*).

Para visualizar a lista de artigos, o utilizador, deverá selecionar o tipo de artigo que deseja na barra de navegação. Ao selecionar o tipo de artigo desejado, o utilizador é encaminhado para a página de seleção de categorias dos artigos do tipo de artigo selecionado, conforme ilustrado pelas figuras **Figura** K-2 e **Figura** K-3, das respetivas categorias *Gaming* e *TVs and Cameras*.

Nestas páginas o utilizador poderá selecionar a categoria do tipo de artigo que pretende visualizar.

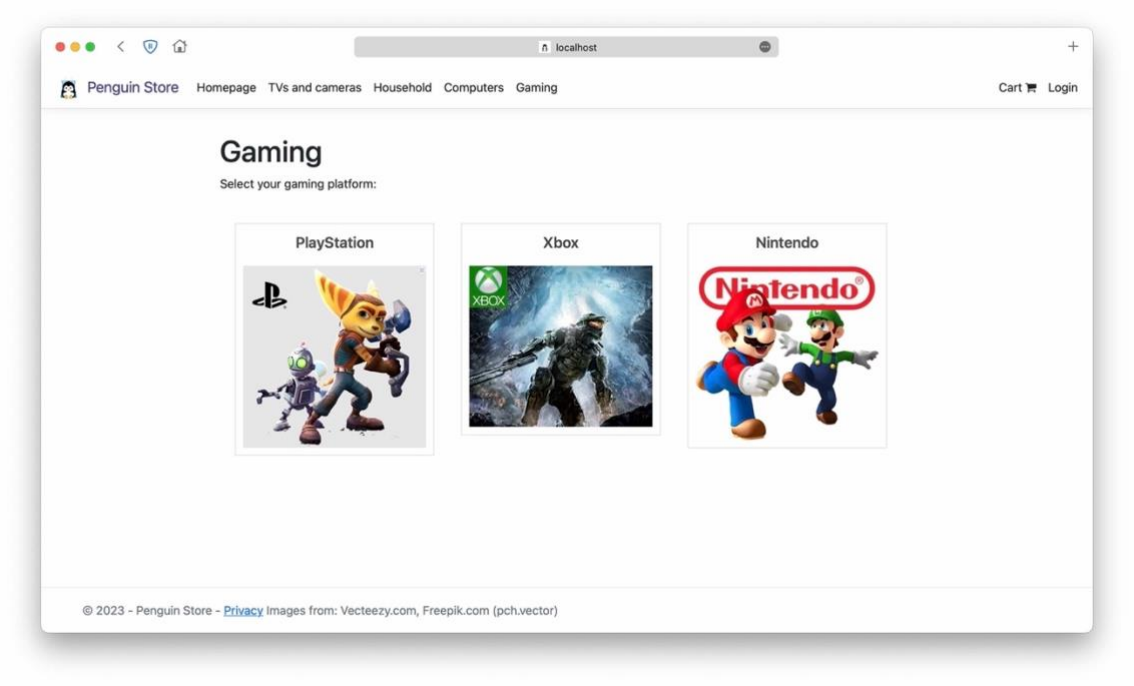


Figura K-2: Página da categoria dos artigos *Gaming*

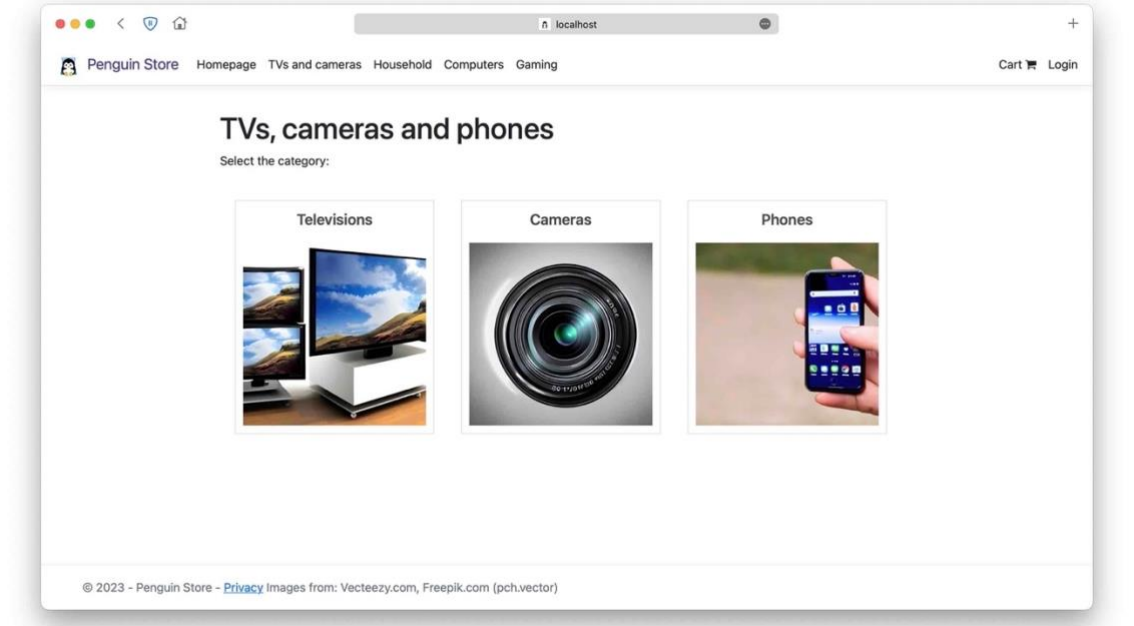


Figura K-3: Página da categoria dos artigos *TVs and Cameras*

Para exemplificar o funcionamento do *website* foi escolhido o tipo de artigo *Gaming*. Ao escolher a plataforma desejada, por exemplo *PlayStation*, o utilizador será encaminhado para a página com a listagem de todos os artigos do tipo videojogos (*Gaming*) da plataforma *PlayStation*, conforme apresentado na **Figura** K-4.

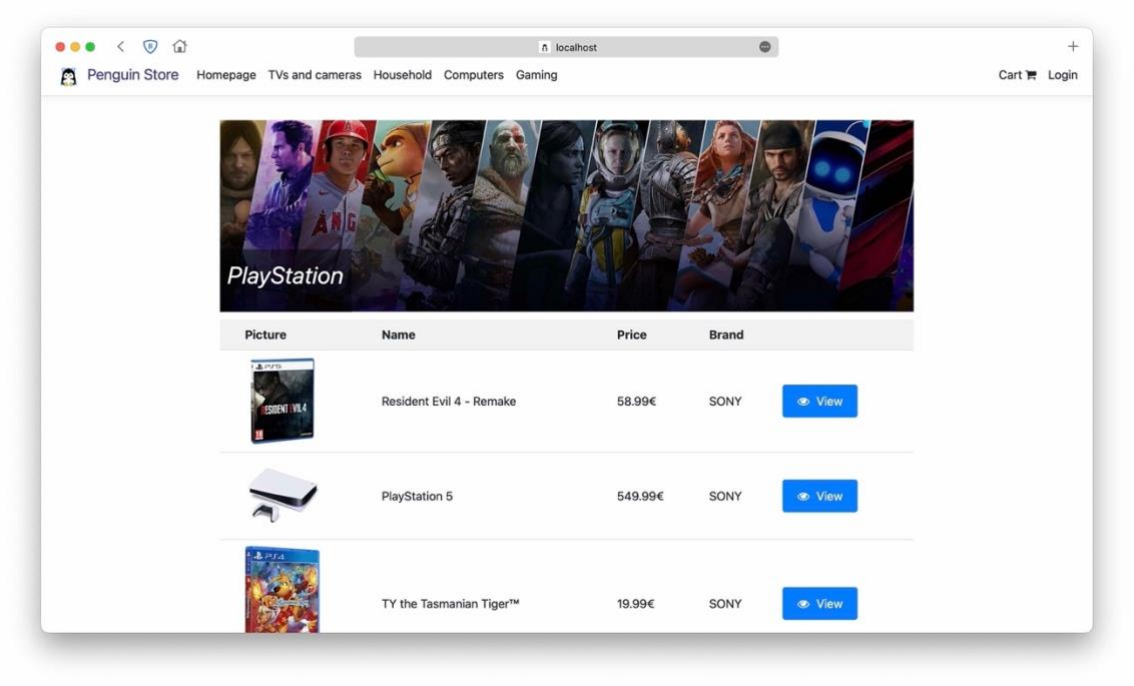


Figura K-4: Página de listagem de artigos *Gaming* da plataforma *PlayStation*

Nesta página, o utilizador pode consultar a lista detalha de todos os artigos disponíveis para compra na Loja da categoria *Gaming*, e respetivos preços. Quando o utilizador quer consultar em maior detalhe um determinado artigo presente na lista deverá clicar no botão *View* que o encaminhará para página de detalhes do artigo, tal como ilustrado na **Figura** K-5.

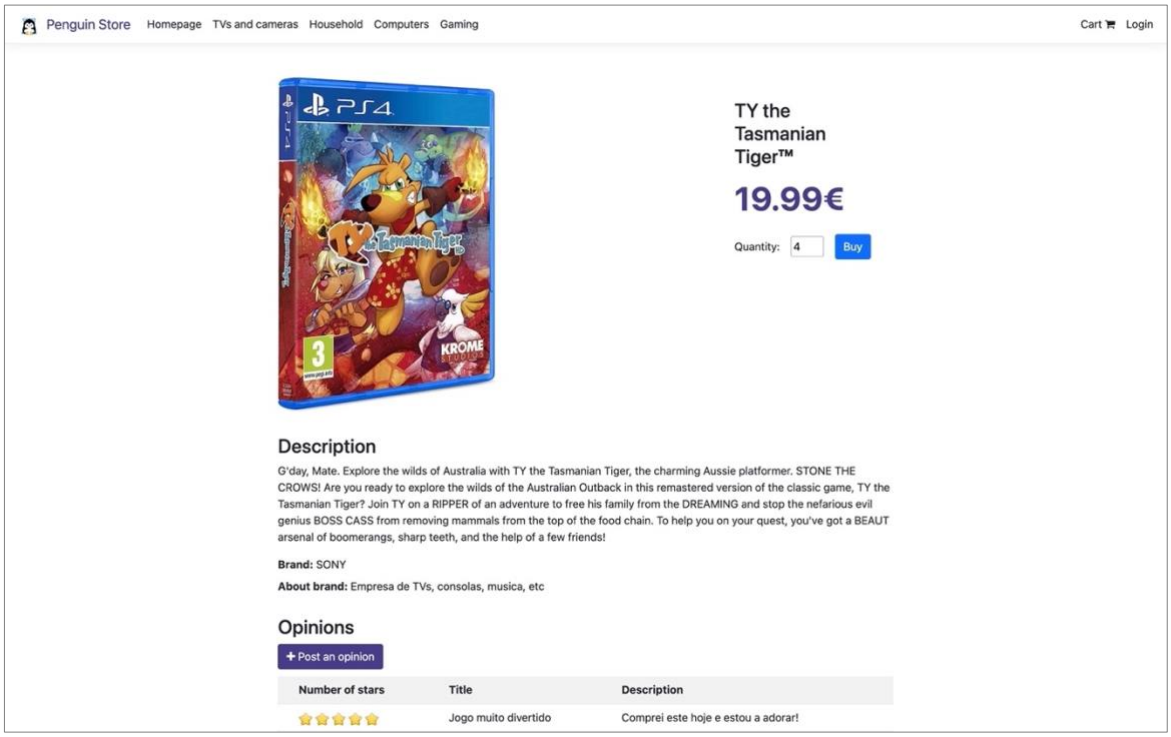


Figura K-5: Página de detalhes de um artigo da Loja

Na página de detalhes do artigo, o utilizador, poderá visualizar toda a informação relativa ao artigo em questão, como sejam o título, fotografia, descrição, marca e preço, e efetuar a compra adicionando o artigo ao carrinho e selecionando o número de unidades desejadas.

Nesta página, o utilizador pode, também, consultar todas as opiniões publicadas por outros clientes acerca do artigo, e publicar a sua opinião, clicando no botão intitulado de *Post an opinion*. As opiniões sobre os artigos são classificadas por estrelas, refletindo uma estrela um elevado grau de insatisfação e cinco estrelas um elevado grau de satisfação.

Seguidamente, procede-se à análise do processo de registo e autenticação na página da Loja. Para o registo na Loja, o cliente deverá clicar no separador *login* presente na barra de navegação, que o encaminhará para a página representada na **Figura** K-6.

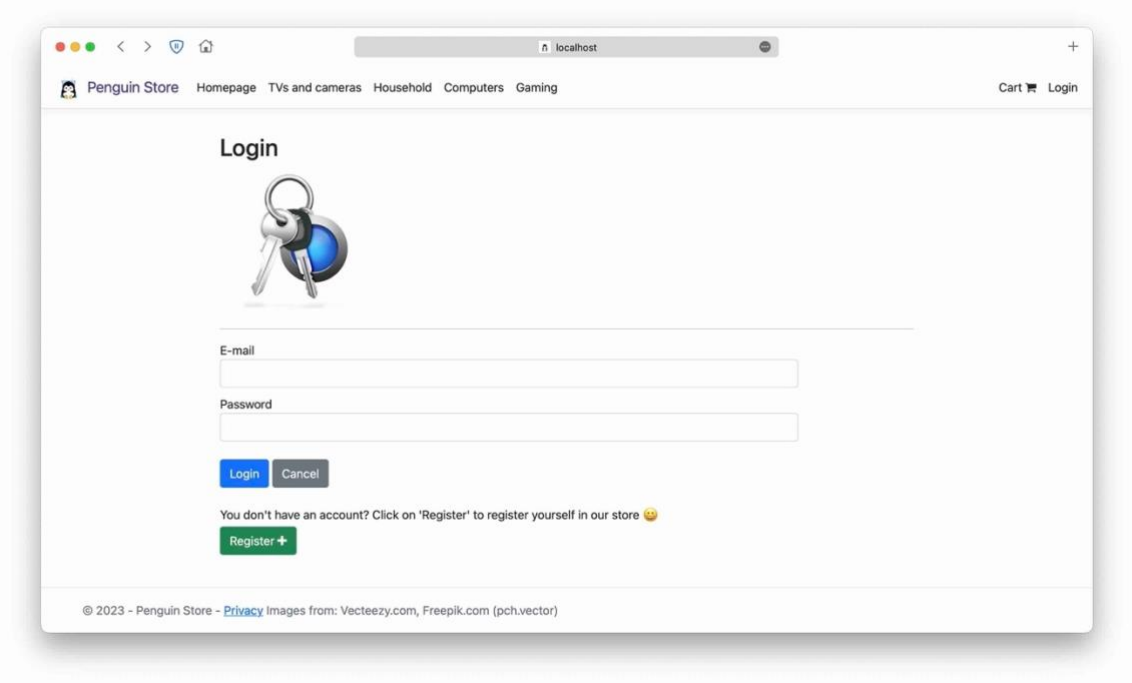


Figura K-6: Página de autenticação da Loja

Nesta página, o utilizador poderá entrar na Loja com uma conta já existente, ou registar-se na Loja clicando no botão de registo *Register*, que o encaminhará para a página ilustrada na **Figura** K-7.

Para criar a sua conta de cliente, o utilizador, deverá preencher o formulário apresentado, nomeadamente, o seu nome, endereço de *email*, palavra-passe, telefone, morada, número de conta bancária e selecionar o seu país. Depois de preenchidos todos os campos, o utilizador deverá clicar no botão *Register* para finalizar a criação da sua conta de cliente da Loja.

É importante referir que o número de conta bancária corresponde ao identificador do cliente no sistema Banco. No exemplo apresentado na **Figura** K-7 verifica-se que o número de conta bancária selecionado foi o número dois, uma vez que a conta bancária do cliente *Tiago Simões* possui esse identificador no sistema Banco, conforme ilustrado anteriormente na **Figura** 5-14.

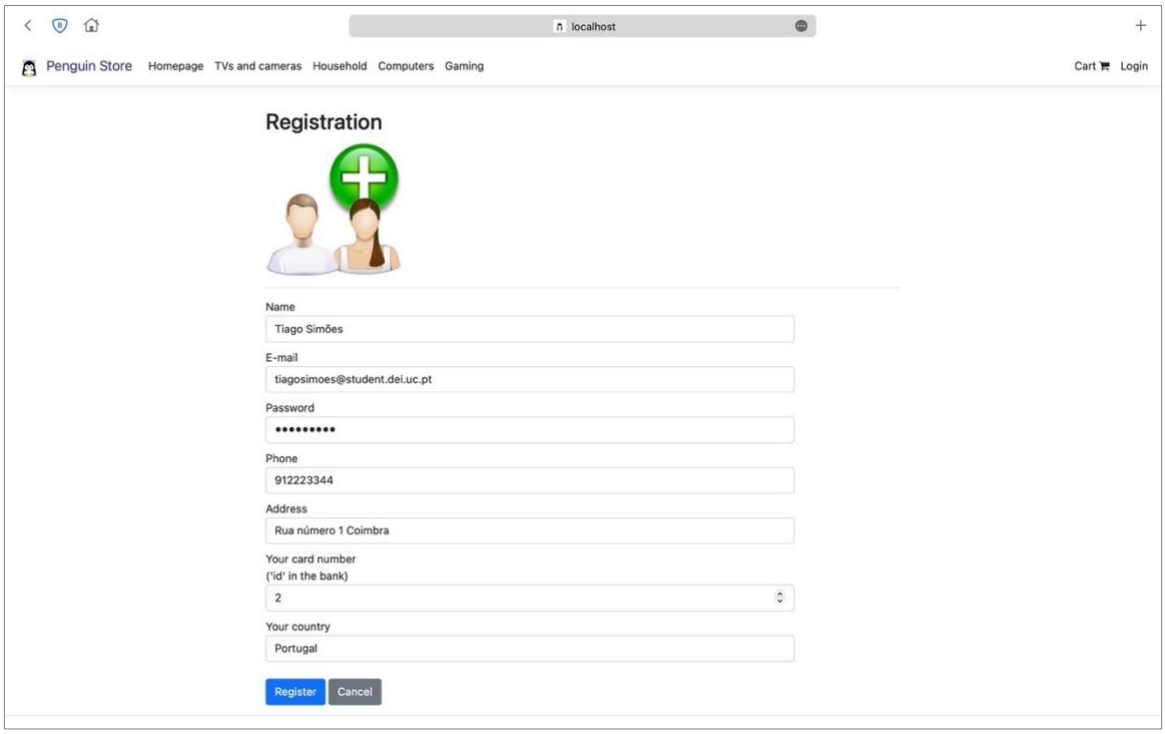


Figura K-7: Página de registo da conta do cliente na Loja

Finalizado o registo, a barra de navegação apresenta o separador do cliente intitulado de *Hello Tiago Simões!*, que quando clicado redireciona o utilizador para a sua página de gestão da sua conta na Loja, conforme ilustrado na **Figura** K-8.

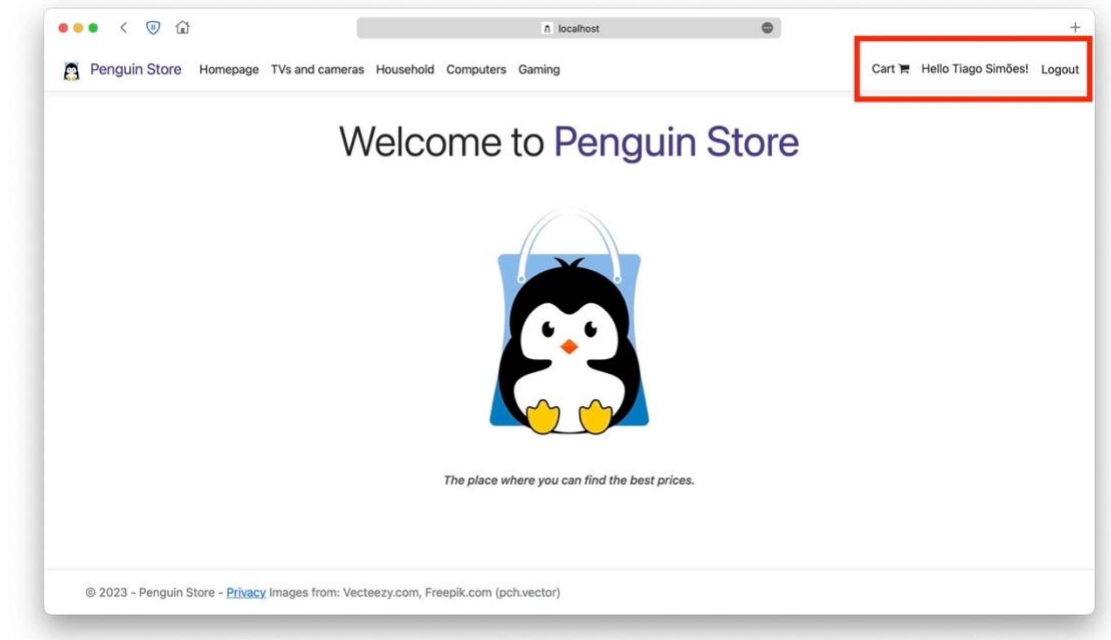


Figura K-8: Página principal com o cliente *Tiago Simões* autenticado

Concluída a autenticação do utilizador, prossegue-se para a consulta da sua página de gestão da sua conta de cliente, onde poderá visualizar as suas encomendas (*my orders*), os seus dados (*my*

account), alterar os seus dados (*edit my account*) e apagar a sua conta do sistema (*delete my account*), conforme ilustrado na **Figura** K-9.

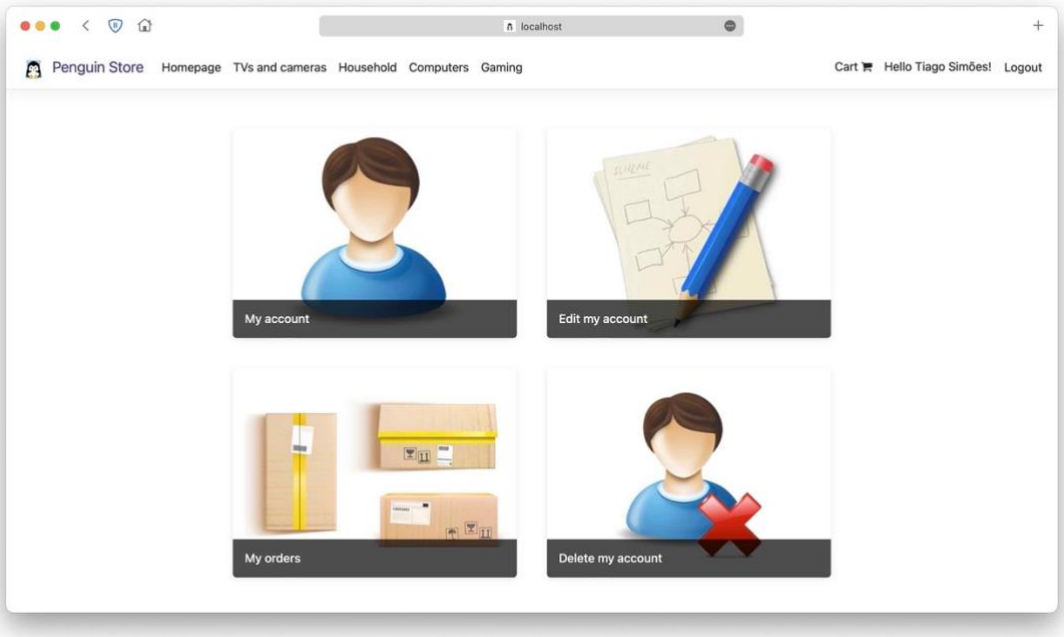


Figura K-9: Página de gestão da conta de cliente da Loja

Para efetuar uma encomenda de artigos na Loja, o utilizador deverá estar autenticado com a sua conta de cliente para aceder à página do artigo que pretende comprar.

Depois de adicionar ao cesto os artigos que pretende comprar o utilizador deverá aceder ao seu carrinho de compras. Para isso, deverá, na barra de navegação, selecionar o separador *Cart*, que o encaminhará para a página onde poderá visualizar o seu carrinho de compras e verificar se todos os artigos e preços dos mesmos foram corretamente atribuídos.

Conforme representado na **Figura** K-10 a página dispõe de três painéis, onde o primeiro denominado *Cart* ilustra os artigos adicionados ao carrinho, a respetiva quantidade e custo de cada um deles, e o valor total final da respetiva compra. O segundo painel apresenta os dados de envio e faturação da encomenda (*delivery address*), como a morada, e o número de conta do cliente. O terceiro painel (lado direito) apresenta um botão que permite ao utilizador remover o artigo do seu carrinho.

Para efetuar a encomenda o utilizador deverá clicar no botão de compra *Checkout*.

De referir que sempre que o utilizador termina a sessão no *website* da Loja o seu carrinho é guardado pelo sistema. Quando volta a autenticar-se o seu carrinho é novamente restaurado.

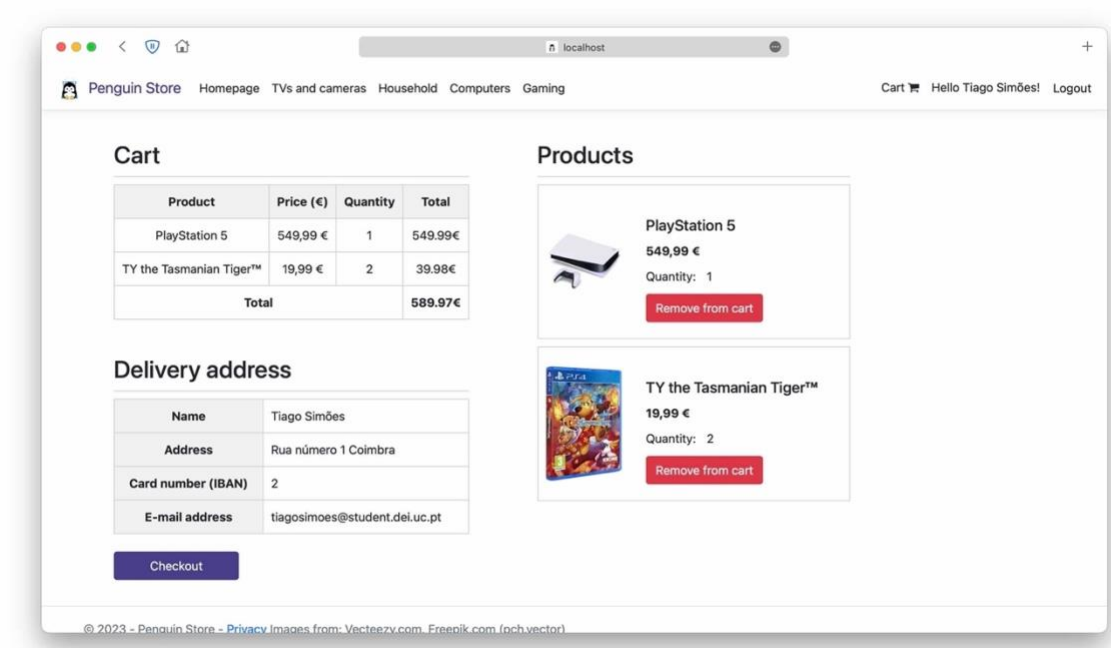


Figura K-10: Página do carrinho de compras do cliente da Loja

Efetivada a encomenda é exibida uma mensagem ao utilizador a informá-lo de que a compra foi efetuada com sucesso, tal como ilustrado na **Figura** K-11.

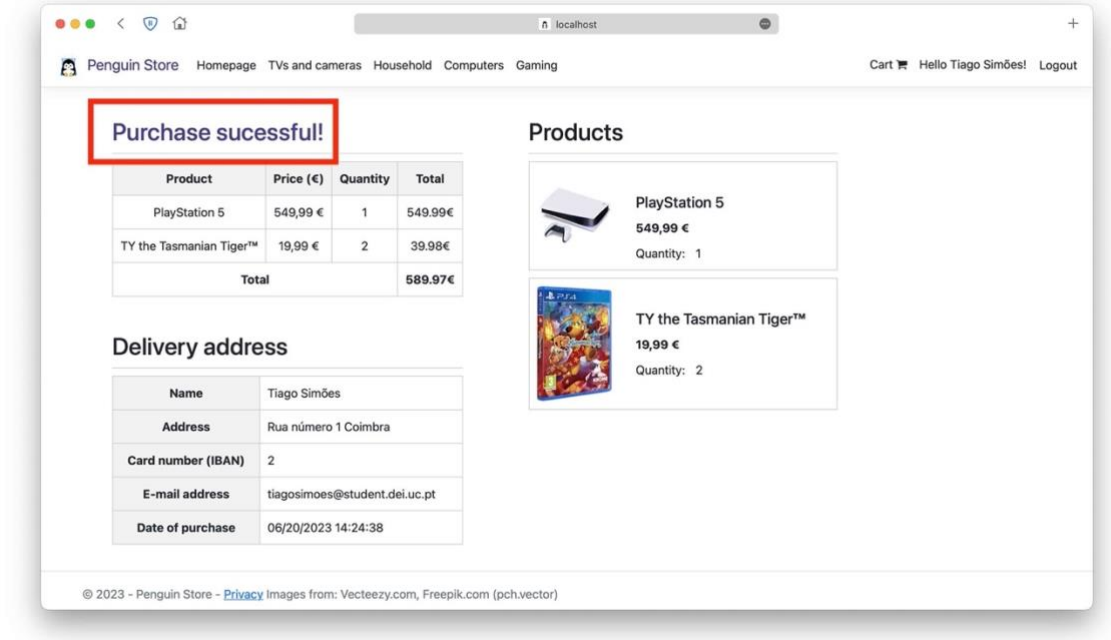


Figura K-11: Página do carrinho de compras do cliente da Loja, após efetivada a encomenda

É possível consultar o estado da encomenda, a partir da página de gestão da conta do cliente, na página de encomendas. A página de encomendas encontra-se apresentada na **Figura** K-12. O utilizador tem a opção de visualizar em maior detalhe a encomenda clicando no botão *Check*, que o encaminhará para a página da **Figura** K-13.

É possível consultar o estado da encomenda, a partir da página de gestão da conta do cliente, na página de encomendas (**Figura** K-12). O utilizador tem a opção de visualizar em maior detalhe a encomenda clicando no botão *Check*, (**Figura** K-13).

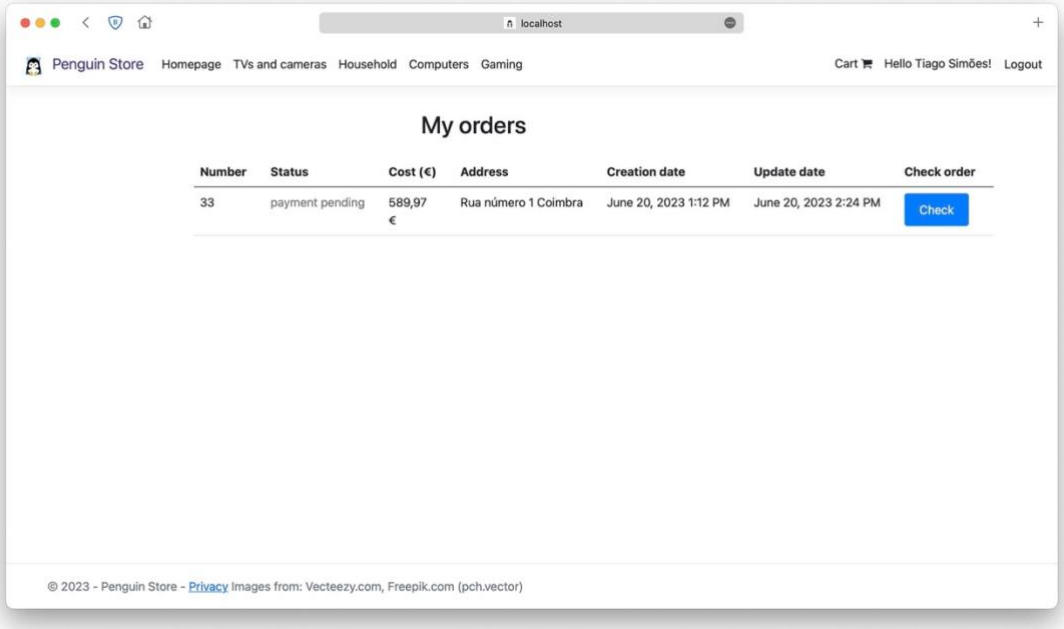


Figura K-12: Página de encomendas do cliente

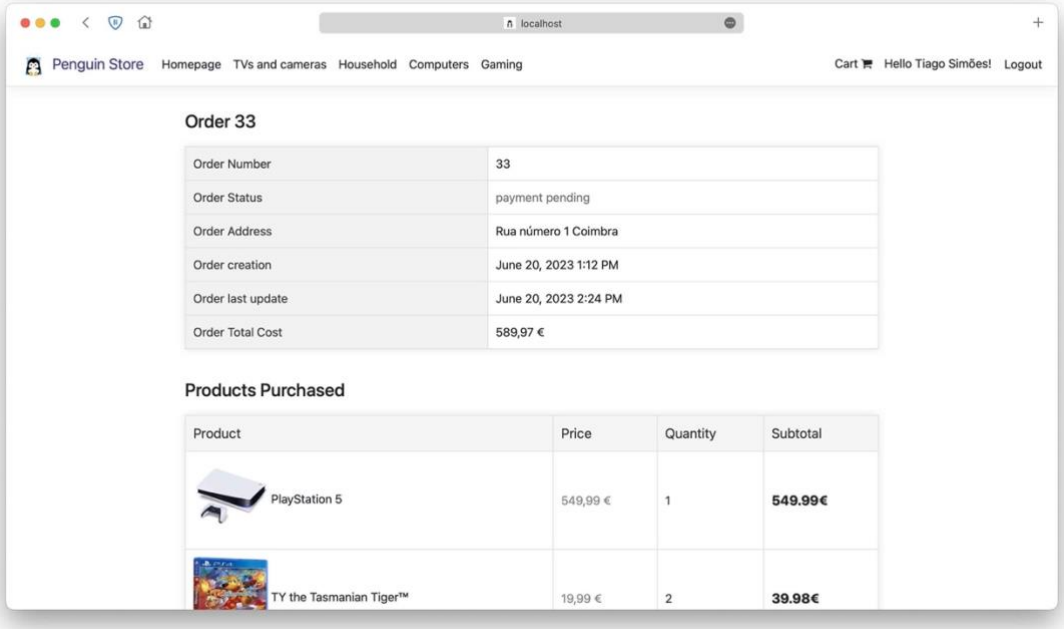


Figura K-13: Página de detalhes de uma encomenda

É possível verificar que a encomenda se encontra num estado pendente, uma vez que a Loja realiza a operação de cobrança de forma assíncrona. Ou seja, a Loja envia, via MQTT, uma

mensagem representativa da transação bancária que será, posteriormente, recebida pelo microsserviço de *backend* do sistema Banco. Aquando da receção da mensagem da transação bancária pelo microsserviço do Banco, este efetua o débito na conta do cliente e, se for bem-sucedida, envia de volta à Loja uma mensagem de confirmação da realização da transação. Até que a confirmação chegue à Loja marcará a encomenda como pendente (*payment pending*) e quando a recebe atualiza o seu estado para pagamento recebido (*payment received*), tal como apresentado na **Figura** K-14.

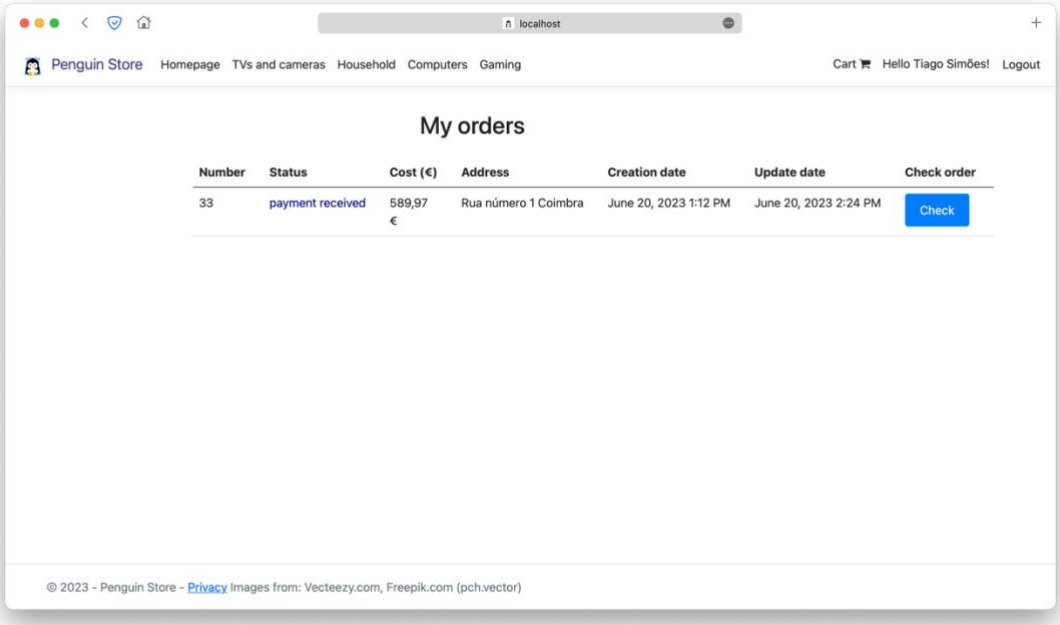


Figura K-14: Página de encomendas do cliente com o estado atualizado de pagamento recebido

Sistema Publicitaki

Concluída a apresentação da Loja, prossegue-se à apresentação do sistema Publicitaki.

Para aceder à página Publicitaki, o utilizador aceder ao *endpoint* do microsserviço *pub-frontend*, através do seu navegador *web*. Ao aceder à página, o utilizador visualizará uma página idêntica àquela exibida na **Figura** K-15.

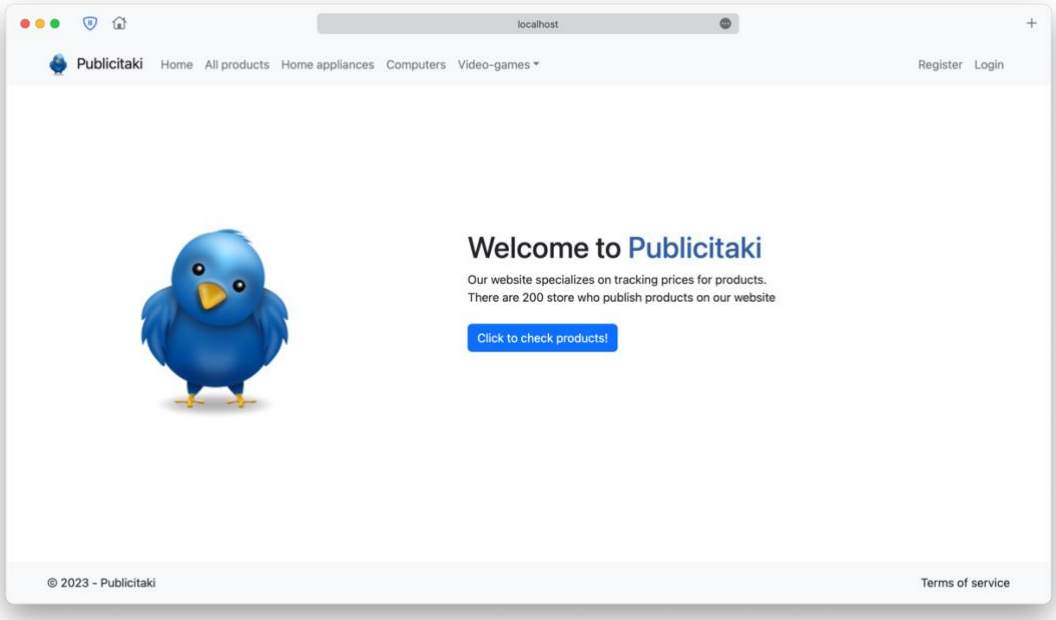


Figura K-15: Página principal do sistema Publicitaki

A partir da página principal o utilizador poderá selecionar a categoria dos artigos vendidos pelas lojas, selecionando a categoria pretendida a partir da barra de navegação.

Como referido no Capítulo 3, no Publicitaki existem três utilizadores distintos, o cliente que irá efetuar a compra na loja, a loja e o administrador do *website*.

Assim, inicia-se a apresentação das tarefas do administrador.

O administrador, tal como os restantes utilizadores, deverá autenticar-se com as suas credenciais para aceder às suas funcionalidades, selecionando o separador *Login* da barra de navegação, que o irá encaminhar para a página de autenticação comum para todos os utilizadores (**Figura** K-16).

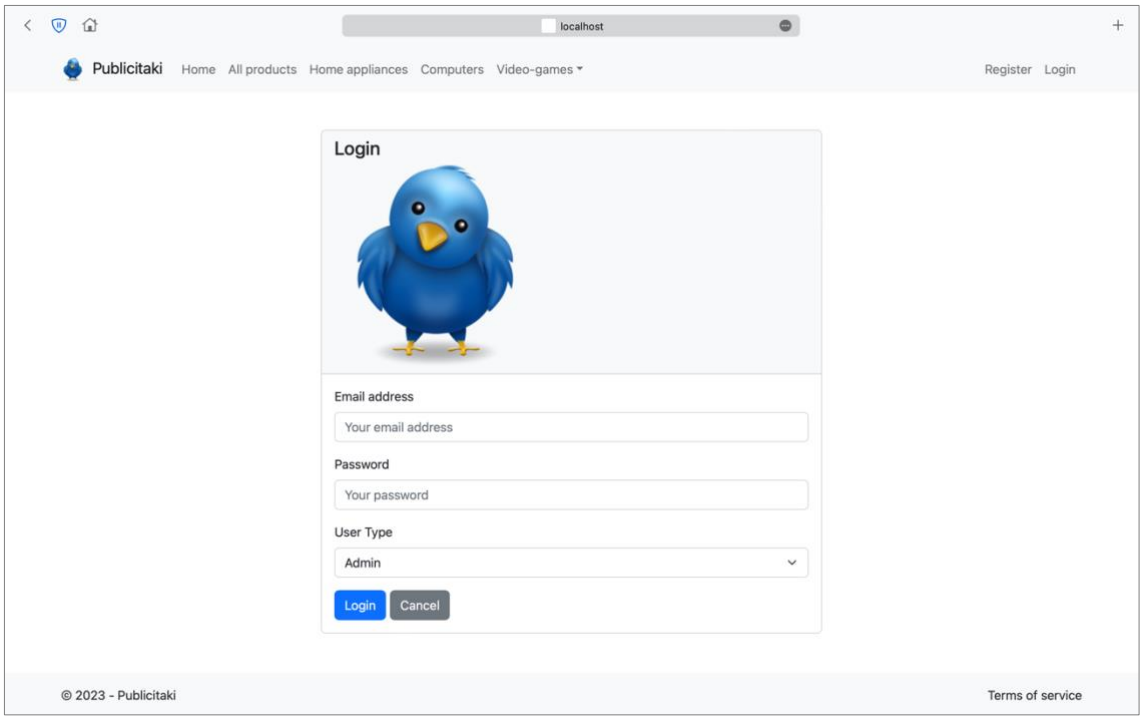


Figura K-16: Página de autenticação dos utilizadores no Publicitaki

Na página de autenticação da **Figura** K-16 o utilizador, deverá inserir o seu endereço de email, palavra-passe e selecionar o seu perfil de utilizador. Os perfis de utilizador correspondem ao tipo de utilizador, que poderá ser cliente (*customer*), loja (*store*) ou administrador (*administrator*).

Ao efetuar a autenticação como administrador, o utilizador poderá consultar a página de gestão da sua conta, clicando para tal, no separador *Hello admin*, que o encaminhará para a página apresentada na **Figura** K-17.

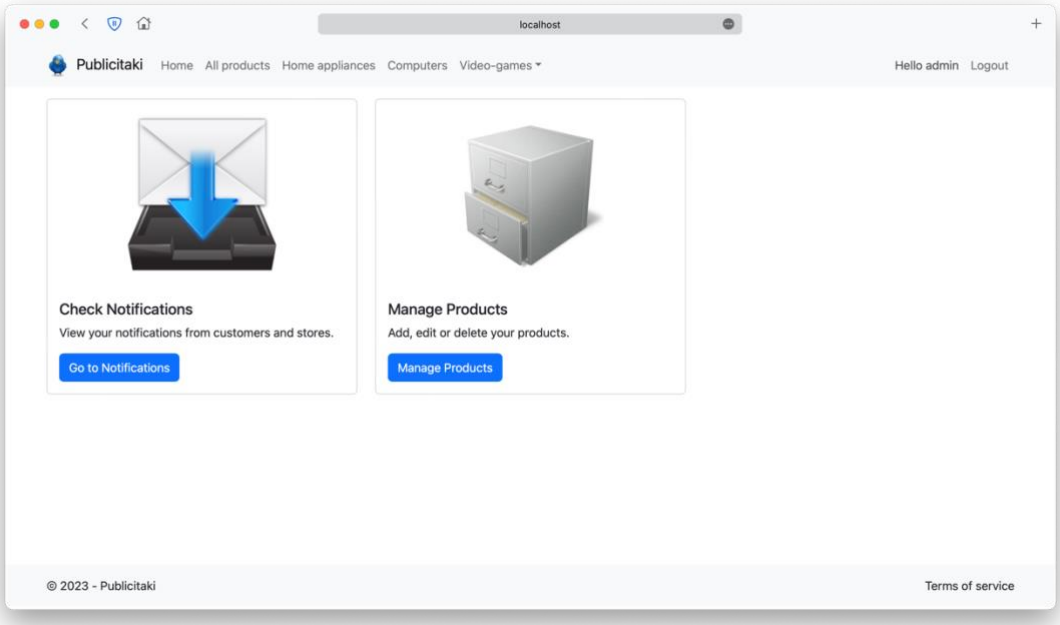


Figura K-17: Página da conta do administrador do Publicitaki

Nesta página o administrador poderá gerir os artigos publicitados no Publicitaki (opção *Manage Products*) e consultar as notificações de registo de inserção ou alteração de artigos que são requisitados pelas lojas para registo no *website* por parte do administrador (opção *Check Notifications*).

Ao clicar na opção *Manage Products*, o administrador é encaminhado para a página de gestão dos artigos publicado, conforme ilustrado na **Figura** K-18.

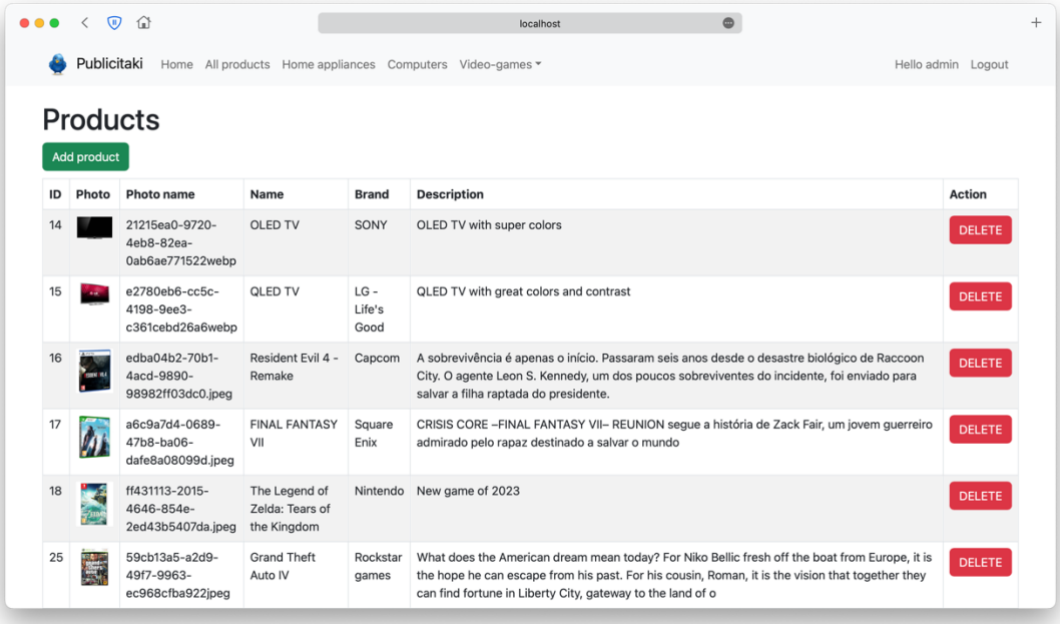


Figura K-18: Página de gestão de artigos

Esta página permite ao administrador gerir os artigos que se encontram publicitados no Publicitaki, sendo que as lojas apenas poderão associar o seu preço se o artigo já tiver sido registado pelo administrador anteriormente, caso contrário, as lojas deverão enviar para o administrador uma notificação de registo de um novo artigo através da sua conta no Publicitaki.

Para registar um novo artigo para ser publicitado no Publicitaki, o administrador deverá clicar no botão *Add product*, encaminhando-o para a página ilustrada na **Figura** K-19.

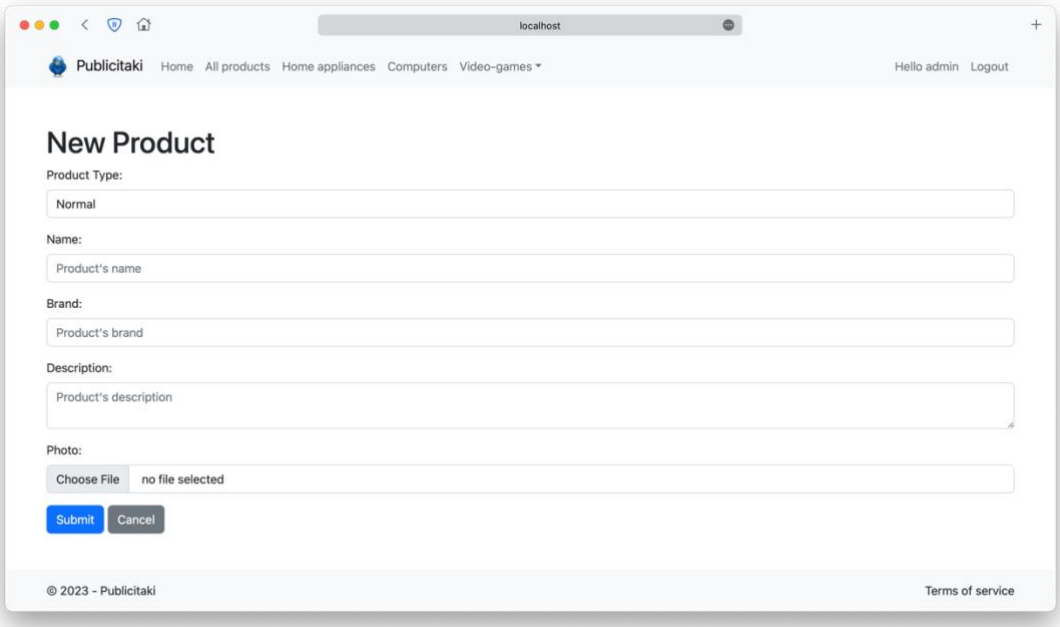


Figura K-19: Página de registo de artigos

Para registar um artigo, o administrador deverá, primeiramente, selecionar a categoria do artigo (*Product Type*) tendo a opção de registar um artigo, computador (*computer*), videojogo (*videogame*), eletrodoméstico (*home appliance*) ou um artigo genérico que não corresponda às anteriores categorias. Selecionada a categoria, o administrador deverá preencher o nome, marca, descrição e fotografia do artigo. Poderá ser necessário preencher outros parâmetros dependendo da categoria do artigo selecionada.

Depois de efetuado o preenchimento, o administrador deverá clicar em *Submit* para registar o artigo no Publicitaki.

Seguidamente serão apresentadas as tarefas da loja no Publicitaki.

Tal como o administrador, a loja também se deve autenticar no sistema para efetuar as suas tarefas. Caso não possua uma conta no Publicitaki, a loja deverá clicar no separador de registo (*Register*) presente na barra de navegação, que encaminhará o utilizador para a página apresentada na **Figura** K-20.

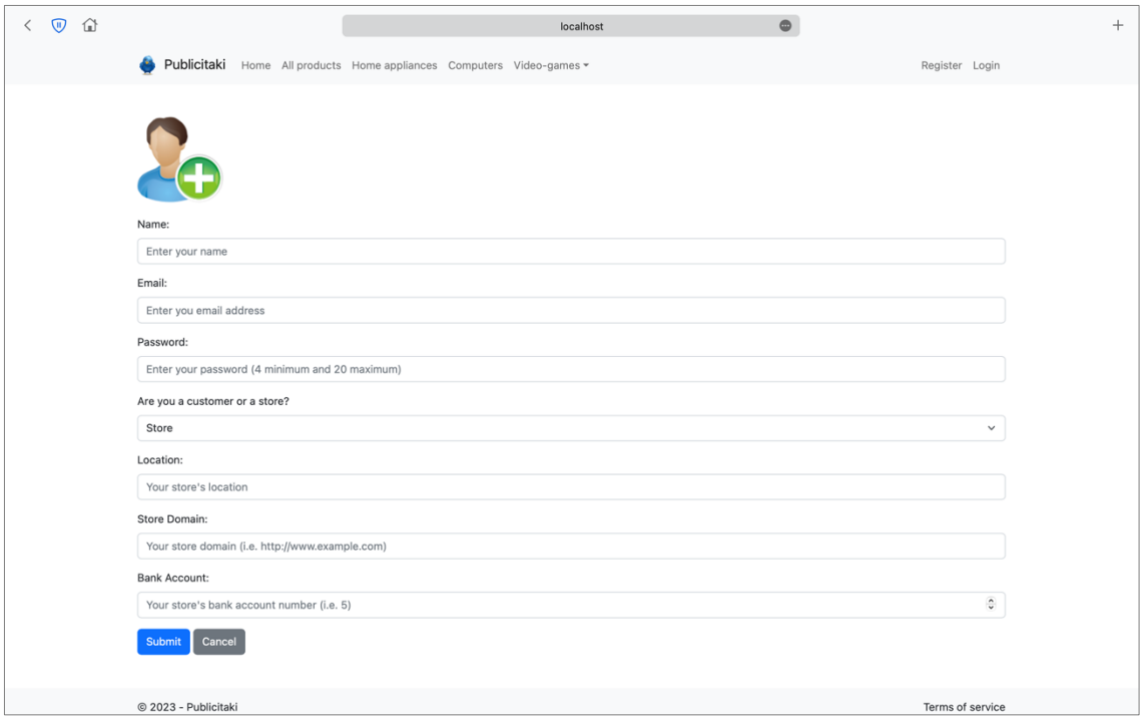


Figura K-20: Página de registo de conta com perfil *Loja* selecionado

A página permite a criação de contas cliente (*customer*) e loja, bastando para isso que o utilizador insira o seu nome, email, palavra-passe e que selecione o perfil da sua conta (*cliente* ou *loja*). Se for selecionado o perfil *loja* será necessário preencher os campos de localização, domínio e número de conta bancária da loja (correspondente ao número de conta da loja no sistema Banco).

Finalizado o registo da conta do utilizador *loja*, este ficará, automaticamente, autenticado, podendo aceder à página de gestão da sua conta na barra de navegação, que o encaminhará para a página ilustrada na **Figura** K-21.

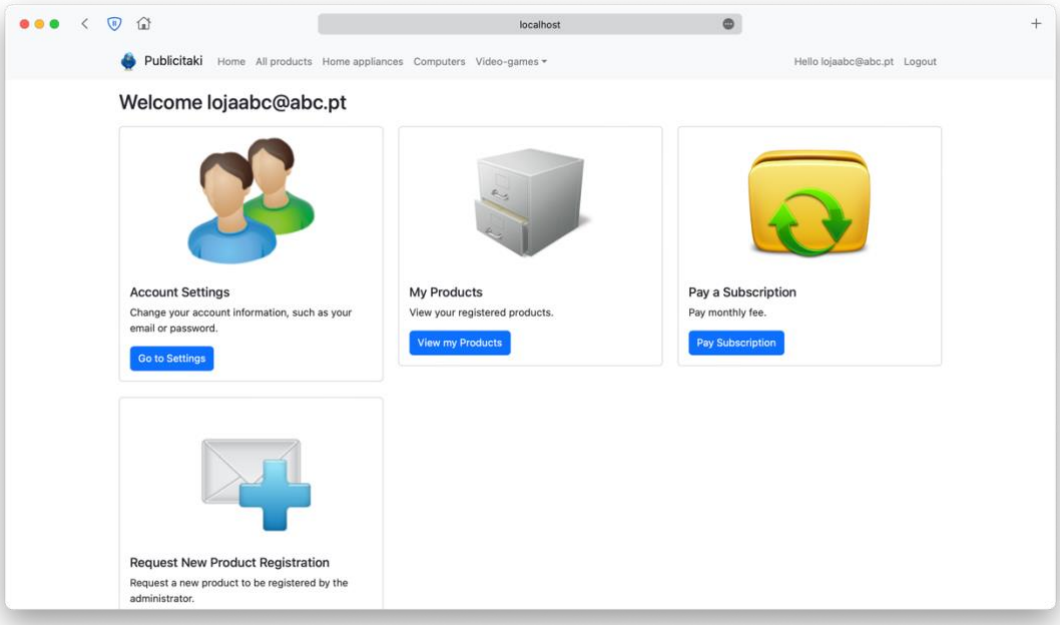


Figura K-21: Página da conta da Loja no Publicitaki

Conforme demonstrado na **Figura** K-21 a loja poderá alterar os dados da sua conta (inclusive apagar a sua conta), visualizar os artigos para os quais atribuiu o seu preço ou atribuir novos preços a determinados artigos registados pelo administrador, pagar a sua subscrição (para poder atribuir preços aos seus artigos) e requisitar um registo de um novo artigo no Publicitaki por parte do administrador. Ao clicar na opção de gestão de artigos (*View my products*) o utilizador é encaminhado para a página de gestão de artigos onde pode atribuir um preço a um artigo, conforme apresentado na **Figura** K-22.

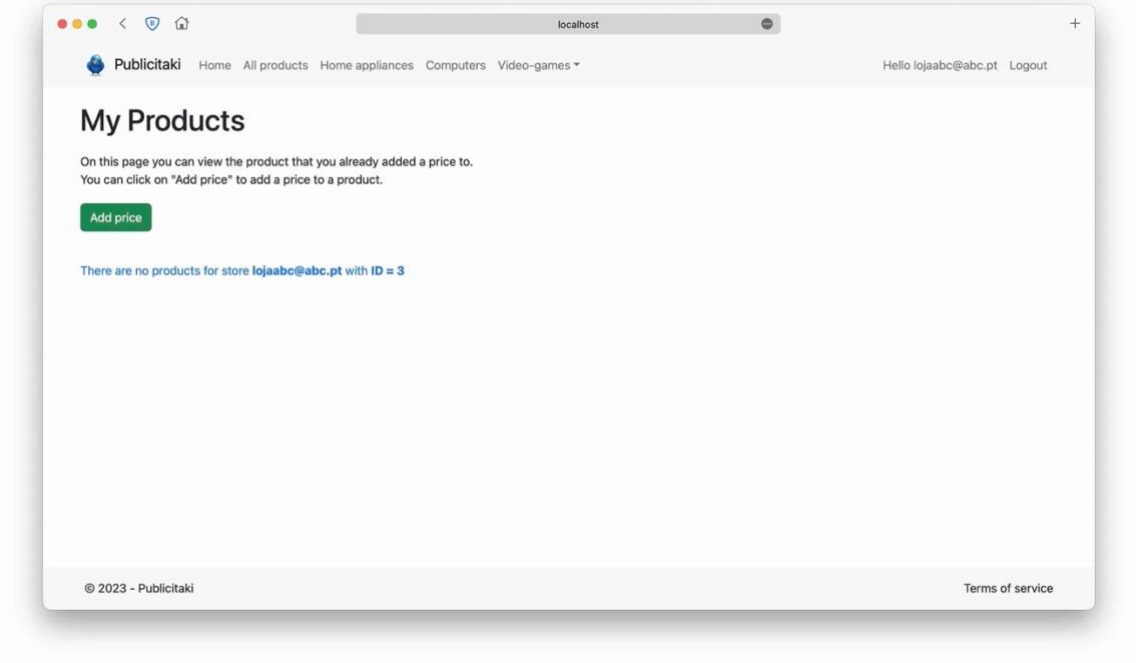


Figura K-22: Página de gestão de artigos da loja no Publicitaki

Ao analisar a **Figura** K-22 verifica-se que a loja, neste momento, ainda não atribui preços a quaisquer artigos. Para atribuir o preço a um artigo, o utilizador deverá clicar no botão *Add Price*, que o encaminhará para a página da **Figura** K-23.

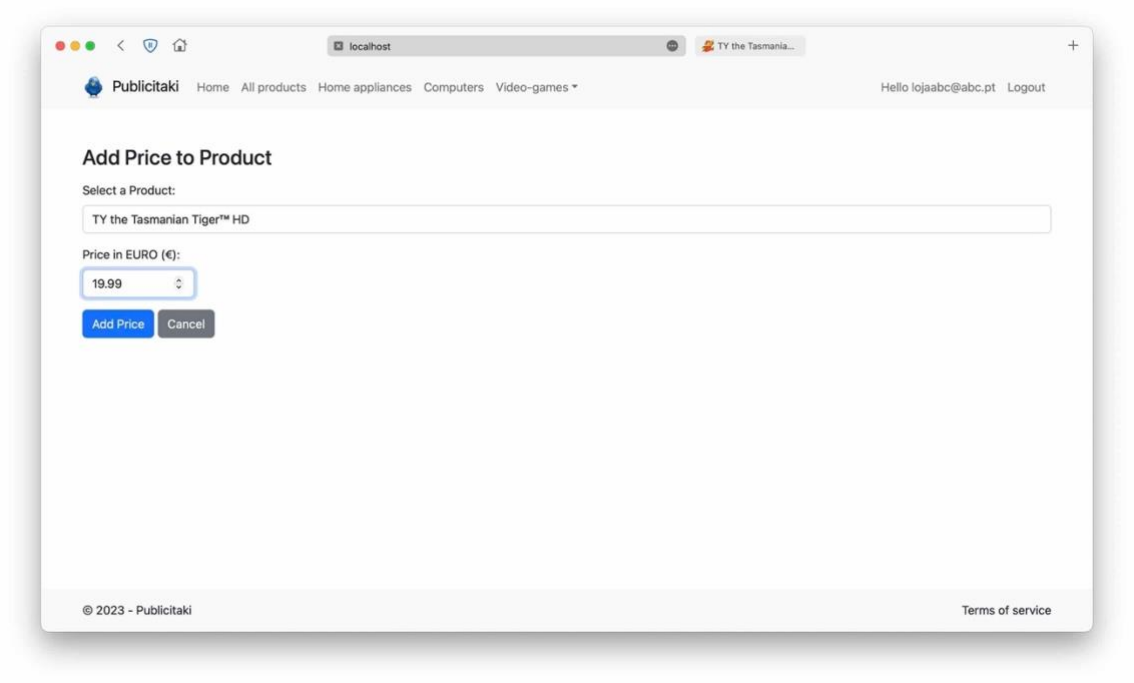


Figura K-23: Página de atribuição de preço a artigo

Como ilustrado na **Figura** K-23, para atribuir o preço a um artigo, o utilizador deverá selecionar o seu artigo clicando na lista *drop-down* que lhe apresenta a listagem de todos os artigos publicados na loja para definir o seu preço.

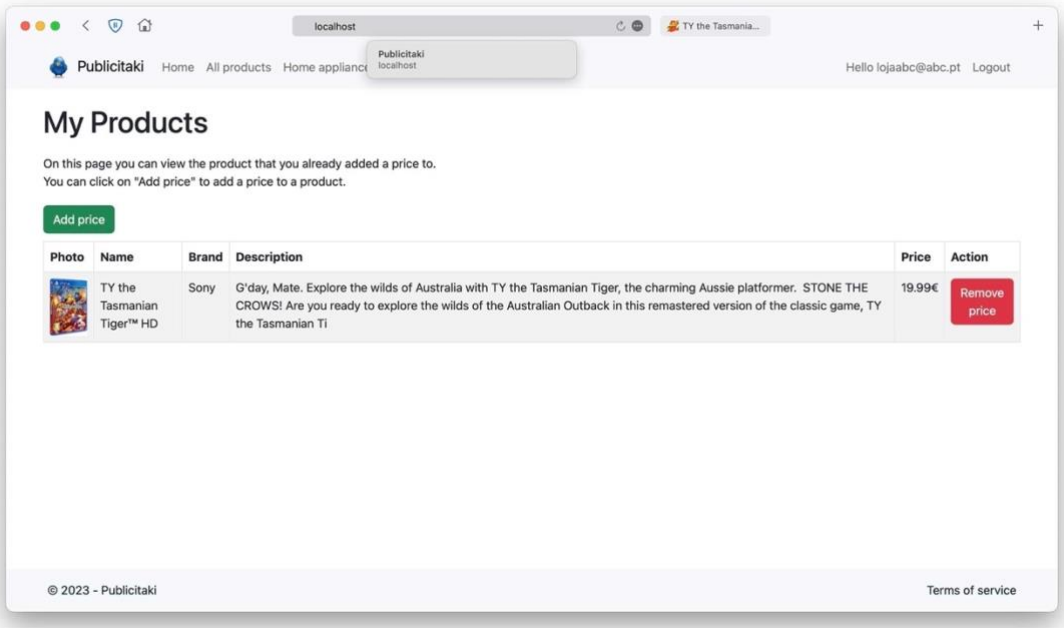


Figura K-24: Página de gestão de artigos da conta da loja, depois de atribuído o novo preço, no Publicitaki

Após a atribuição do preço ao artigo, por parte da loja, será possível aos clientes visualizarem na página de listagem de artigos a publicitação do artigo por parte da loja (loja *lojaabc*).

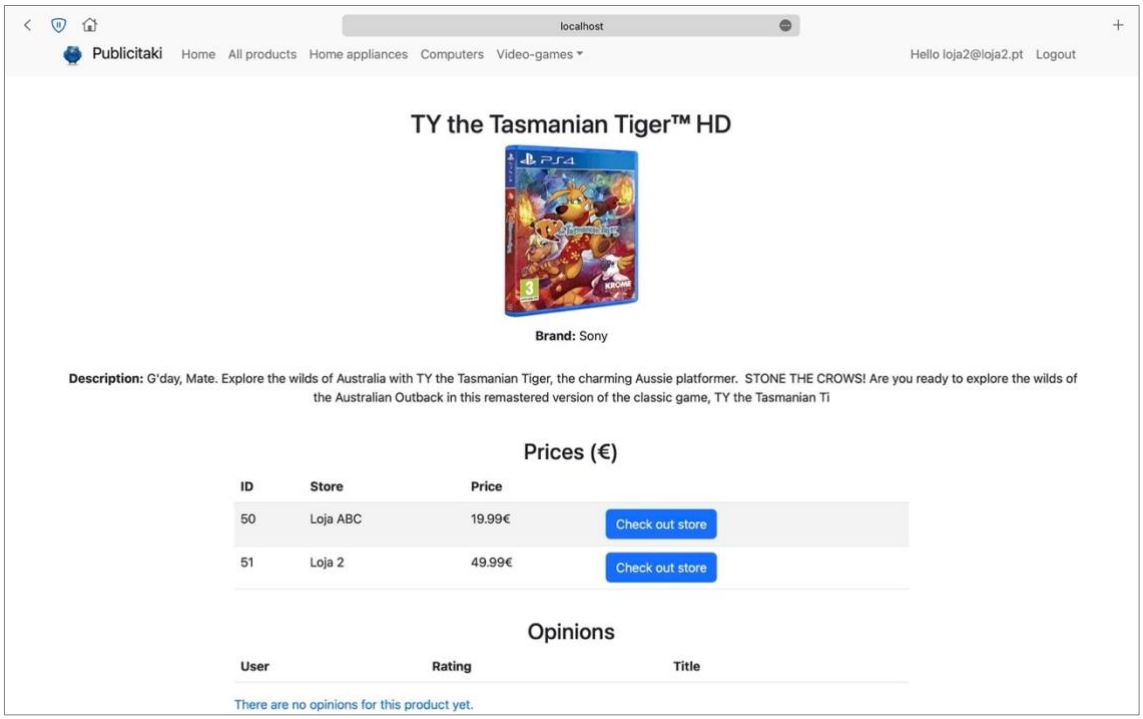


Figura K-25: Página de listagem de artigos

Ilustradas as tarefas da loja no Publicitaki, prossegue-se para a apresentação das tarefas do cliente.

Ao contrário dos restantes utilizadores, o utilizador cliente não é obrigado a registar-se no *website*, no entanto, deverá registar-se para poder *seguir* um artigo ou publicar uma opinião do mesmo. Sem estar autenticado o cliente, apenas pode visualizar os preços dos artigos nas respetivas lojas e ser redirecionado para essas mesmas lojas e consultar opiniões publicadas por utilizadores autenticados.

Assumindo que o cliente se registou e, posteriormente, se autenticou no sistema, ao aceder à página de conta de cliente, será apresentada ao utilizador cliente, a página ilustrada na **Figura** K-26.

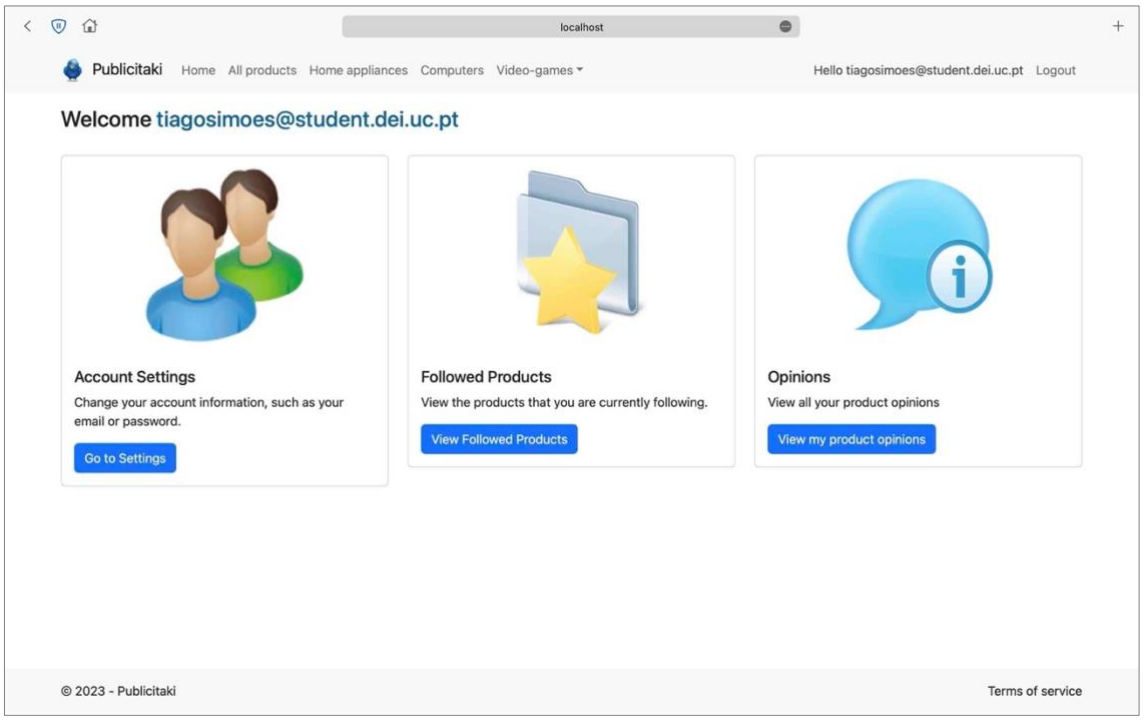


Figura K-26: Página do cliente no Publicitaki

Na página do cliente, o utilizador poderá realizar as operações de gestão da sua conta (*account settings*), nomeadamente, editar ou apagar a sua conta, seguir artigos (*follow products*), e consultar as opiniões que publicou anteriormente.

Depois de autenticado no sistema, o utilizador cliente ao aceder à página de um determinado artigo, terá à sua disposição funcionalidades exclusivas para clientes autenticados, nomeadamente as opções de seguir ou publicar uma opinião sobre um determinado artigo, como demonstrado na **Figura** K-27.

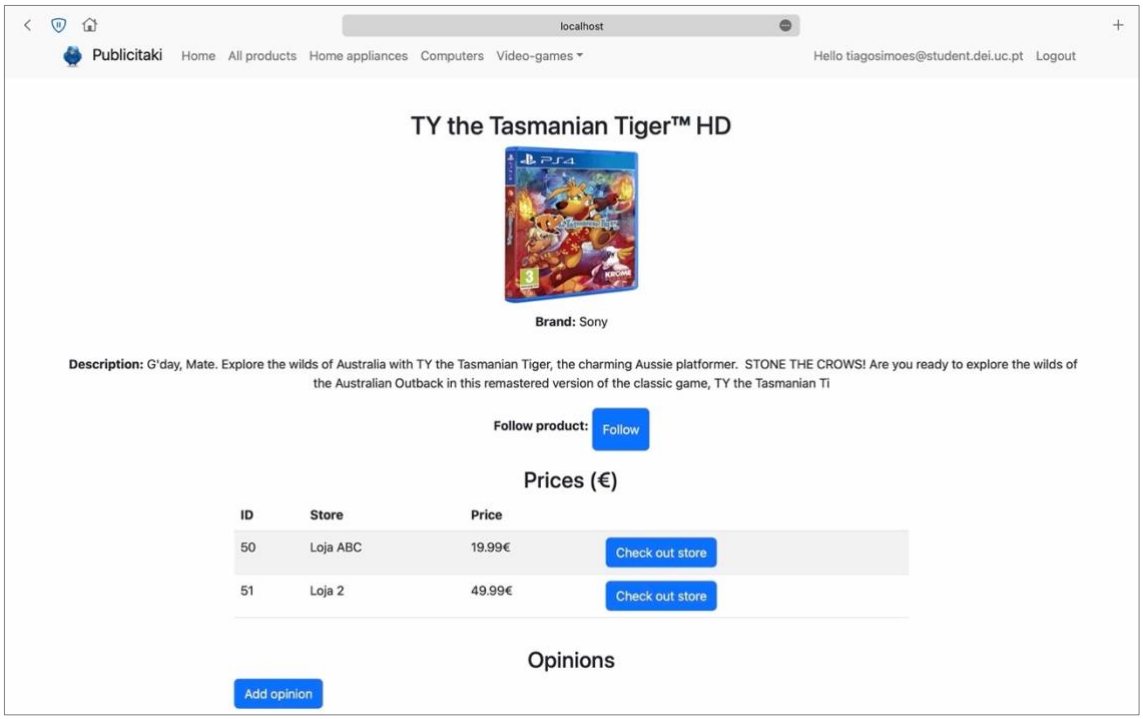


Figura K-27: Página de detalhes de um artigo, com o cliente autenticado

Caso o utilizador deseje seguir um artigo deverá clicar no botão *follow*, e automaticamente poderá a seguir esse artigo. O utilizador poderá consultar na sua conta os artigos que se encontra a seguir e, caso deseje deixar de os seguir, tal como demonstrado na **Figura** K-28.

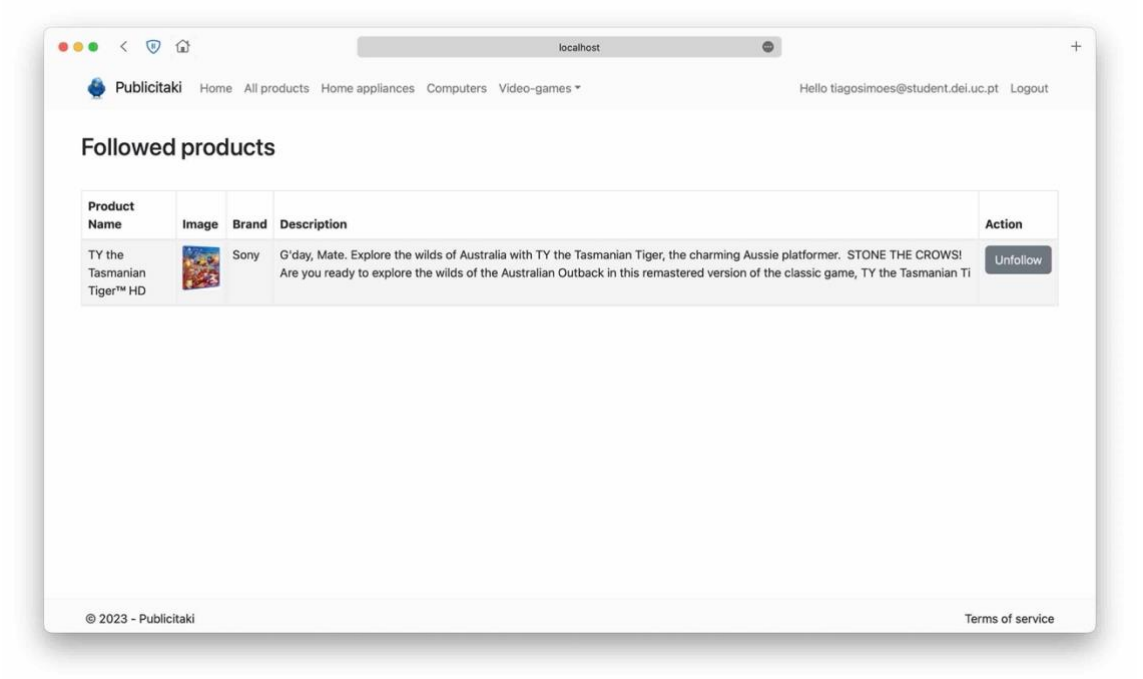


Figura K-28: Página de seguimento de artigos da conta do cliente

Como ilustrado, anteriormente, na **Figura** K-27 o utilizador tem a opção de publicar uma opinião relativa a um artigo, preenchendo uma página onde lhe é solicitado a atribuição de uma avaliação, título e descrição da opinião (**Figura** K-29).

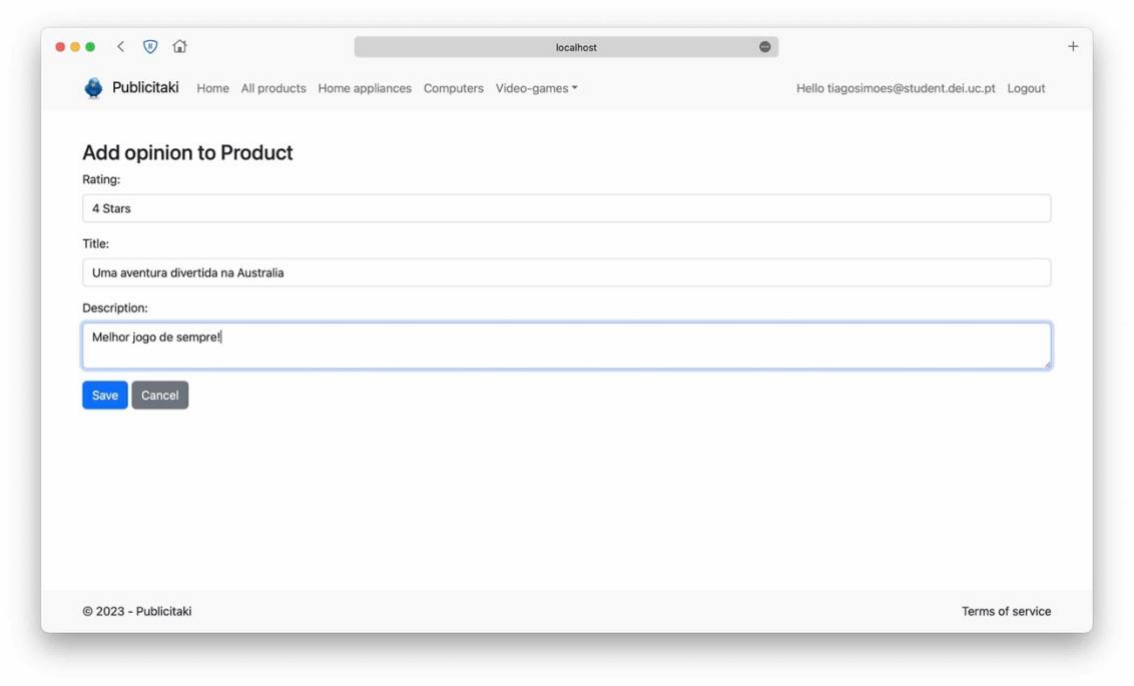


Figura K-29: Formulário de publicação de opinião de um artigo por parte do cliente

Depois de publicada a opinião, esta será apresentada na página do artigo para que possa ser consultada por outros clientes no Publicitaki, como apresentado na **Figura** K-30.

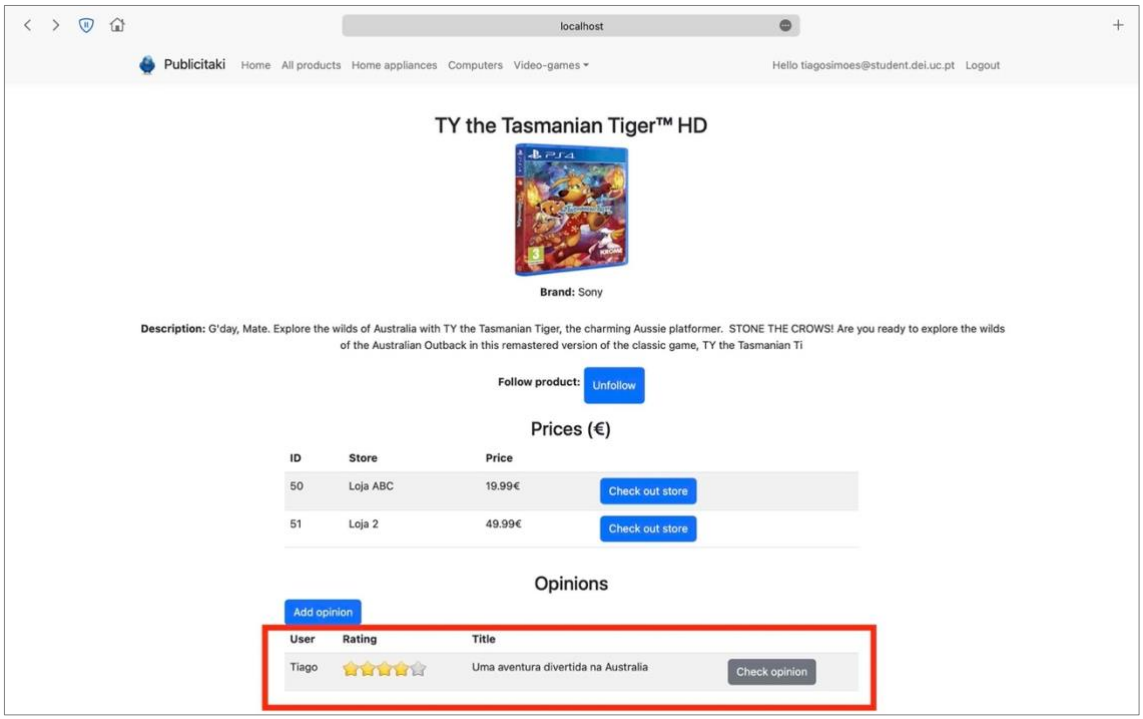


Figura K-30: Página de detalhes do artigo com a publicação da opinião pelo cliente