UNIVERSIDADE
BEIRA INTERIOR

Metodologias de Desenvolvimento para Aplicações Baseadas em Microserviços

Sérgio José Saraiva Gonçalves

Relatório do Projeto de Estágio
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Nuno Coelho Pombo
Co-orientador: Eduardo Rodrigues de Almeida

Covilhã, junho de 2023

Declaração de Integridade

Eu, Sérgio José Saraiva Gonçalves, que abaixo assino, estudante com o número de inscrição M10668 de/o Mestrado em Engenharia Informática da Faculdade de Engenharias, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o Código de Integridades da Universidade da Beira Interior.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 08/06/2023

Sérgio José Saraiva Gonçalves

Agradecimentos

A conclusão deste relatório projeto de estágio descrito neste documento, bem como a iminente conclusão de uma das etapas mais importantes da minha vida, não seria possível sem o apoio de todas as pessoas que estiveram presentes ao longo da minha vida.

Primeiramente quero agradecer à minha família, especialmente aos meus pais, pois sempre lutaram juntamente comigo pelos meus sonhos, sempre estiveram lá nos bons e nos maus momentos, sempre com um conselho para me dar, este meu feito não é só meu como também é deles. Devo-lhes tudo, devido ao que me tornei e ao que fizeram por mim.

De seguida, quero agradecer ao professor doutor Nuno Pombo, sem ele nada disto seria possível, foi sempre prestável e presente, sempre apoiou o progresso deste relatório e deu-me sempre bons conselhos. Obrigado professor doutor Nuno Pombo, por ter aceite ser o meu orientador, assim como toda a paciência que foi necessária neste percurso. Como não podia deixar de ser, agradeço a todos os meus amigos, um obrigado não chega por todo o companheirismo durante a minha vida.

Quero agradecer ao meu orientador na empresa Eduardo Rodrigues de Almeida, por fazer-me crescer tanto neste estágio, pela maturidade que me transmitiu, pelos conselhos que me deu, por sempre me ajudar quando assim precisei, assim como à empresa de acolhimento e a toda a minha equipa de desenvolvimento composta por João Pereira, Daniel Matos, Arnaldo Costa, Pedro Silva, Sérgio Silva, Nuno Silva, João Vieira, Daniel Pereira e Eduardo Rodrigues. Um enorme obrigado a todos vocês.

Por fim, um obrigado aos meus companheiros de casa, que se tornaram pessoas importantes na minha vida, por todas as horas passadas juntos a estudar, por todo o companheirismo, por todos os conselhos, sem esquecer todo o carinho com que me receberam, então obrigado Rafael Silva, Xavier Silva e João Tinoco e à minha namorada Maria Loureiro.

Resumo

Este relatório incide-se sobre o estágio profissional realizado no Fundão, na empresa ReadinessIT, com o principal objetivo de manter e implementar novas funcionalidades utilizando microserviços na área de telecomunicações. Para um melhor entendimento, é apresentado uma mapa cronológico desenvolvido, com o planeamento do estágio, pela empresa de acolhimento para um melhor desenvolvimento do estagiário e são descritos os objetivos a atingir durante o mesmo, estes definidos pela empresa de acolhimento. Tendo estes em conta, são apresentados artigos com literatura relevante com comparações entre diferentes tipos de arquitetura para desenvolvimento de aplicações, entre outros.

São apresentadas as tecnologias que serão usadas ao longo do estágio e a metodologia de trabalho, sendo também apresentado um mapa cronológico sobre o desenvolvimento dos principais domínios no decorrer do mesmo. Tendo em conta que todo o desenvolvimento efetuado faz parte da propriedade intelectual da empresa de acolhimento e da empresa cliente, é apenas apresentada uma arquitetura abstrata para cada uma das funcionalidades implementadas para o cliente e uma justificação geral dos objetivos de cada uma destas.

Por fim, é descrito o trabalho efetuado por objetivos e apresentadas as conclusões do estagiário após este estágio.

Palavras-chave

Microserviços, ReadinessIT, Estágio, Spring Boot

Abstract

This document focuses on the professional internship carried out in Fundão, in the ReadinessIT company, with the main objective of maintaining and implementing new functionalities using microservices in the telecommunications area. For a better understanding, it is presented a chronological map developed, with the internship planning, by the host company for a better development of the trainee and the objectives to be achieved during the internship are described, these defined by the host company too. Taking these into account, articles with relevant literature are presented with comparisons between different types of architecture for application development, among others.

The technologies that will be used throughout the internship and the work methodology are presented, as well as a chronological map on the development of the main objetives during the internship. Taking into account that all the development done is part of the intellectual property of the host company and the client company, it is only presented an abstract architecture for each of the functionalities implemented for the client and a general justification of the objectives of each of these.

Finally, the work done by objectives is described and the trainee's conclusions after this internship are presented.

Keywords

Microserviços, ReadinessIT, Internship, Spring Boot

Índice

1 Introdução **1**

1.1 Contextualização 1
1.2 Caracterização Geral da Organização 1
1.3 Objetivos do Estágio 2
1.4 Projeto de Estágio 3
1.5 Propriedade Intelectual 3
1.6 Organização do Documento 3

2 Estado da Arte **5**

2.1 Introdução 5
2.2 *Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review* 5
2.3 *A Comparative Review of Microservices and Monolithic Architectures* 6
2.4 Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems 8
2.5 *Dynamic Microservices to Create Scalable and Fault Tolerance Architecture* 8
2.6 *Kubernetes as an Availability Manager for Microservice Applications* 10
2.7 Conclusão 10

3 Metodologias e Ferramentas Utilizadas **13**

3.1 Introdução 13
3.2 Metodologias 13
3.2.1 *Agile/Scrum* 13
3.2.2 Vantagens da Metodologia Agile/Scrum 14
3.2.3 Utilização da Metodologia Agile/Scrum 15
3.3 Ferramentas Utilizadas 16
3.3.1 Postman 16
3.3.2 Apache JMeter 17
3.3.3 SoapUI 17
3.3.4 Draw.IO 18
3.3.5 Jira 19
3.3.6 Confluence 19
3.3.7 Git/GitLab 21
3.3.8 IntelliJ IDEA 22
3.3.9 PostgreSQL 23
3.3.10 pgAdmin 24
3.3.11 Docker Desktop 24
3.3.12 Kafka 25
3.3.13 RabbitMQ 26

3.3.14 Maven . . . 28
3.3.15 Java . . . 29
3.3.16 JUnit . . . 30
3.3.17 MongoDB . . . 30
3.3.18 Redis . . . 31
3.3.19 SonarQube . . . 32
3.3.20 Spring Boot . . . 33
3.4 Conclusão . . . 35

4 Análise Comparativa de Arquiteturas de Desenvolvimento de Aplicações 37
4.1 Introdução . . . 37
4.2 Arquitetura Monolítica . . . 37
4.3 Arquitetura Orientada a Serviços . . . 40
4.4 Microserviços . . . 42
4.5 Conclusão . . . 44

5 Planeamento do Estágio 47
5.1 Introdução . . . 47
5.2 Plano do Estágio . . . 47
5.2.1 Academia Especializada . . . 48
5.2.2 Formações e Certificações . . . 48
5.2.3 Projeto Interno . . . 49
5.2.4 Período de Integração . . . 49
5.2.5 Projeto Cliente . . . 50
5.2.6 Avaliação . . . 50
5.3 Conclusão . . . 50

6 Implementação 51
6.1 Introdução . . . 51
6.2 Plano de Trabalho . . . 51
6.2.1 Domínio do *Customer* . . . 52
6.2.2 Domínio de *Billing* . . . 53
6.2.3 Criação de Ordem . . . 54
6.3 Conclusão . . . 57

7 Conclusão 59
7.1 Conclusões Principais . . . 59
7.2 Trabalho Futuro . . . 60

Bibliografia 61

Lista de Figuras

2.1 Resultados do Primeiro Teste [1]. 7
2.2 Resultados do Primeiro Teste em Relação ao Tempo de Resposta [1]. 7
2.3 Resultados do Segundo Teste [1]. 8
2.4 Arquitetura Proposta [2]. 9

3.1 Metodologia Srcum/Agile [3]. 14
3.2 Postman IDE [4]. 17
3.3 Apache JMeter IDE [5]. 18
3.4 Exemplo de resposta de uma API no SoapUI [6]. 19
3.5 Exemplo de utilização do Draw.IO [7]. 20
3.6 Exemplo de um quadro do Jira [8]. 20
3.7 Exemplo de uma página do *Confluence* [9]. 21
3.8 Exemplo dos *branches* do Git [10]. 22
3.9 Interface do IntelliJ IDEA[11]. 23
3.10 Exemplo de utilização do pgAdmin [12]. 24
3.11 Docker Desktop *workspace* [13]. 25
3.12 Arquitetura do Apache Kafka [14]. 26
3.13 Arquitetura do RabbitMQ [15]. 27
3.14 Funcionamento geral do *maven*. 29
3.15 Interface do *Mongo Express* [16]. 31
3.16 Caso de uso da implementação do redis com mongoDB. 32
3.17 Interface do *SonarQube* [17]. 33
3.18 Arquitetura do *flow* do *Spring Boot* [18]. 34

4.1 Exemplo de uma Arquitetura Monolítica. 38
4.2 Diferenças entre a Arquitetura Orientada a Serviços e Monolítica [19]. 40
4.3 Papeis Fundamentais da Arquitetura Orientada a Serviços. 41
4.4 Diferenças entre as Arquiteturas SOA, Microserviços e Monolítica [19]. 43

5.1 Mapa Cronológico do Planeamento do Estágio. 47

6.1 Mapa Cronológico do Desenvolvimento do Projeto. 52
6.2 Arquitetura do Domínio de *Customer*. 53
6.3 Arquitetura do Domínio de *Billing*. 54
6.4 Arquitetura da Junção dos Domínios. 55
6.5 Arquitetura da Criação de Ordem. 55
6.6 Arquitetura do Catálogo. 56

Lista de Acrónimos

AMF	*Availability Management Framework*
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
B2B	*Business-to-business*
CPQ	*Configure, Price, Quote*
CRM	*Customer Relationship Management*
DES	*Data Encryption Standard*
FIFO	*First-In-First-Out*
HTTP	*Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
MSA	*Microservice Architectures*
PM	*Project Manager*
OSS/BSS	*Operations & Business Support Systems*
POC	*Proof of concept*
REST	*Representational State Transfer*
SLR	*Systematic Literature Review*
SOA	*Service-oriented architecture*
SOAP	*Simple Object Access Protocol*
TL	*Team Leader*
UBI	Universidade da Beira Interior
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

Capítulo 1

Introdução

Pretende-se com este capítulo introdutório contextualizar este relatório de projeto de estágio, clarificando o seu âmbito, objetivo, empresa de acolhimento, introdução aos projetos em que fui inserido durante o estágio, e por fim, apresentar a estrutura do mesmo.

1.1 Contextualização

Este relatório foi desenvolvido no âmbito da unidade curricular de dissertação ou estágio em engenharia informática, onde esta pertence ao segundo ano do Mestrado de Engenharia Informática no Departamento de Informática, na Universidade da Beira Interior (UBI).

O estágio foi acolhido pela organização Readiness IT, no Fundão, assentando-se na vertente de Java júnior *developer* com microserviços, dando uso à *framework Java Spring Boot*, iniciado a 27 de junho de 2022 e finalizado a 27 de junho de 2023. Tendo como objetivo integrar a equipa de desenvolvimento por microserviços, aprender o funcionamento e implementar os mesmos na área de telecomunicações.

Os orientadores foram o Engenheiro Eduardo Rodrigues Almeida por parte da Readiness IT e o Professor Doutor Nuno Pombo por parte do Departamento de Informática da Universidade da Beira Interior.

1.2 Caracterização Geral da Organização

A Readiness IT foi fundada em 2015, por profissionais com mais de 25 anos de experiência na área de telecomunicações. Esta oferece serviços com alta qualidade e soluções, que desafiam as atuais necessidades dos clientes e aceleram a inovação da empresa.

Neste momento, a Readiness IT encontra-se em funcionamento em 8 países, dos quais: Portugal, Espanha, Servia, Chile, Peru, Estados Unidos da América, Canadá e Nova Zelândia.

O objetivo principal desta é acelerar a inovação digital e a integração por meio da tecnologia e são especializados no uso da tecnologia digital para melhorar processos de negócios, aumentar a fidelidade do cliente, reduzir o custo operacional, aumentar a receita e as margens e adaptar-se rapidamente às realidades do mercado em constante mudança.

Desenvolve produtos nas seguintes áreas, sendo bastante reconhecida em algumas delas:

- *OSS/BSS Operations & Business Support Systems* (OSS/BSS) *System Integration*;
- *Marketplace and Partner Ecosystem deployment*;
- *Low-code/No-code*;
- *Configure, Price, Quote* (CPQ) & Catalog / *Order Management* / *Customer Relationship Management* (CRM);
- Serviços Administrados;
- Serviços Salesforce;
- Automação *Business-to-business* (B2B);
- Microserviços;
- Serviços Cloud;
- Consultoria Digital Telco;
- DevOps;
- Experiência Digital (UX/UI/CX/DX).

1.3 Objetivos do Estágio

O estágio teve como objetivo aprender, desenvolver e implementar microserviços, sejam estas: novas funcionalidades, tratamento de *bugs/defects*, testes integrados e unitários e aprender a fazer documentação interna. De uma forma sucinta, tornar o estagiário o mais autónomo possível dentro do mercado de trabalho, na área de microserviços.

O início do estágio passou pela aprendizagem numa academia especializada na empresa de acolhimento, onde foram dados conceitos mais técnicos de SQL, Java e uma introdução à *framework Java Spring Boot*. Posteriormente, fui colocado num projeto com uma equipa de 9 pessoas a trabalhar para uma equipa cliente. Durante o primeiro mês passei pelo tempo de adaptação, onde os integrantes seniores da equipa davam conceitos de programação, boas práticas, pontos críticos de como trabalhar em equipa e tarefas de análise a ferramentas e tecnologias, que foram utilizadas no decorrer do projeto para que, desta forma, preparassem o começo de desenvolvimento do mesmo.

Este projeto tinha como principal objetivo expor API's relacionadas ao negócio de telecomunicações da empresa cliente, através da utilização de microserviços, principalmente em 3 domínios: faturação, carrinho de compras e cliente. Onde foram desenvolvidos conceitos de negócio como: verificar os produtos disponíveis para um determinado cliente, descontos associados, obter dados de conta, faturas, entre outros. Por fim, os microserviços desenvolvidos no projeto foram utilizados por uma aplicação *web* desenvolvida por uma equipa da empresa cliente.

1.4 Projeto de Estágio

O projeto, como referido na secção 1.3, tem como objetivo criar o *backend*, através da utilização de microserviços, para uma empresa de telecomunicações. Este começou a ser desenvolvido por uma outra equipa contratada pelo cliente, mas acabaram por contratar os serviços da Readiness IT para apoiar o desenvolvimento do mesmo.

Neste, ficou definido 3 principais objetivos:

- definir o domínio de faturação e cliente, a sua estrutura de dados, lógicas de negócio e expor API's de forma a gerir os mesmos;
- desenvolver o carrinho de compras, estrutura de dados, lógicas de negócio relativas a compras online e expor API's para os mesmos;
- e por fim, fazer suporte ao desenvolvimento das outras equipas relacionadas ao projeto.

De notar que, neste projeto não fizemos parte direta no desenvolvimento da arquitetura do projeto, nem das suas lógicas de negócio. Estas foram feitas pela equipa de arquitetos funcionais do cliente, tanto como o *frontend* pertenceu também a uma equipa de desenvolvimento do mesmo.

1.5 Propriedade Intelectual

Razões relativas ao direito de propriedade intelectual justificam limitações da descrição do trabalho, de forma, a proteger os interesses do cliente e da empresa de acolhimento. Com isto, boa parte do processo de desenvolvimento, arquiteturas e informação não pode ser partilhada.

1.6 Organização do Documento

De modo a refletir o trabalho feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta a empresa de acolhimento, os objetivos do estágio, projeto, propriedade intelectual e a organização do documento.
2. O segundo capítulo – **Estado da Arte** – apresenta artigos sobre microserviços, comparação entre esta com outras arquiteturas existentes, a sua utilização com *kubernetes*, bem como alguma literatura relacionada com o projeto.
3. O terceiro capítulo - **Metodologias e Ferramentas Utilizadas** - apresenta metodologias e tecnologias utilizadas no desenvolver do estágio, bem como uma explicação da necessidade do uso da mesma.

4. O quarto capítulo - **Análise Comparativa de Arquiteturas de Desenvolvimento de Aplicações** - apresenta, analisa e compara microserviços, arquitetura orientada a serviços (SOA) e arquiteturas monolíticas, com o objetivo de proporcionar uma compreensão abrangente das suas principais características, vantagens e desvantagens e considerações sobre as mesmas;
5. O quinto capítulo - **Planeamento do Estágio** - apresenta sobre o planeamento que a empresa definiu para o estagiário e a forma como este foi definido;
6. O sexto capítulo - **Implementação** - explica os desenvolvimentos que o projeto teve e a função e responsabilidade do estagiário em cada uma delas;
7. O sétimo capítulo - **Conclusão** - apresenta as conclusões resultantes da elaboração do estágio e do relatório, bem como o trabalho futuro a realizar no projeto cliente, resultado da renovação do cliente com a ReadinessIT.

Capítulo 2

Estado da Arte

2.1 Introdução

Este capítulo tem como finalidade apresentar artigos desenvolvidos sobre microserviços, sejam estes uma comparação de arquiteturas existentes para desenvolvimento de aplicações, a sua utilização com kubernetes, ou uma forma de contextualizar melhor a arquitetura de desenvolvimento por microserviços. Com o objetivo de se perceber o porquê de atualmente se utilizar cada vez mais este tipo de arquitetura, o que são, como se integram, entre outros.

Desta forma, neste capítulo são apresentados 5 artigos:

1. ***Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review*** [20] – apresenta um estudo e os resultados de uma revisão sistemática para explicar profundamente o estado da arte do *Microservice Architectures* (MSA) e identificar os desafios enfrentados na sua utilização;

2. ***A Comparative Review of Microservices and Monolithic Architectures***[1] – apresenta uma comparação entre dois tipos de arquiteturas para desenvolvimento de aplicações, a monolítica e microserviços, e as suas performances;

3. ***Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems***[21] – apresenta uma comparação entre dois tipos de arquiteturas, microserviços e monolítico, enquanto tentam fornecer perspetivas tanto à investigação como à indústria;

4. ***Dynamic Microservices to Create Scalable and Fault Tolerance Architecture***[2] – aborda os desafios levantados pela necessidade de desenvolver um sistema escalável e tolerante a falhas baseado em microserviços, propondo um sistema que possa ser a solução para os mesmos;

5. ***Kubernetes as an Availability Manager for Microservice Applications***[22] – apresenta uma investigação feita para o problema de disponibilidade dos microserviços, bem como várias soluções com a utilização de *kubernetes*.

2.2 *Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review*

Neste artigo, fizeram um estudo e apresentaram os resultados de uma revisão sistemática para explicar profundamente o estado da arte do *Microservice Architectures* (MSA) e

identificar os desafios enfrentados na sua utilização. Levaram em consideração os estudos publicados desde a introdução do MSA em 2014, identificaram 85 dos 3.842 artigos descobertos como estudos primários. Desta forma, conseguem explicar a teoria de funcionamento e o que são os microserviços.

O foco deste *Systematic Literature Review* (SLR) foi principalmente nos desafios encontrados ao aplicar o MSA. Assim, identificaram nove categorias básicas de problemas que foram discutidos nos estudos primários selecionados. Estes problemas são:

1. descoberta de serviços;
2. gestão de dados e consistência;
3. testes;
4. previsão, medição e otimização de desempenho;
5. comunicação e integração;
6. orquestração de serviços;
7. segurança;
8. monitorização, rastreamento e registo;
9. decomposição.

Sintetizaram e explicaram cada um e as direções a tomar, de forma abrangente, utilizando diagramas para fornecer uma visão geral dos problemas identificados e das soluções sugeridas. Além disso, as semelhanças e diferenças entre as soluções disponíveis para o mesmo desafio são informações úteis para escolher uma solução em vez de outra.

Concluíram que o uso de MSA está se a tornar cada vez mais popular e traz muitas soluções e benefícios importantes para aplicações orientadas a serviços e em *cloud*.

2.3 *A Comparative Review of Microservices and Monolithic Architectures*

Este artigo, apresenta uma comparação entre dois tipos de arquiteturas utilizadas para desenvolvimento de aplicações, microserviços e monolítico. Esta comparação apresenta as vantagens da utilização dos microserviços em relação à arquitetura monolítica e as desvantagens da utilização desta última.

Em termos de performance, para determinar o desempenho destas arquiteturas em diferentes cenários, utilizaram diferentes configurações de teste.

Concluíram com a análise dos resultados do primeiro cenário de teste, que estas podem ter um desempenho semelhante sob uma carga normal na aplicação. No caso de uma pequena carga com menos de 100 utilizadores, a aplicação monolítica pode ter um desempenho um pouco melhor do que a aplicação de microserviços. Assim, a aplicação monolítica é recomendada para pequenas aplicações, utilizadas apenas por alguns utilizadores. Estes resultados encontram-se na figura 2.1, que apresenta o numero de request que consegue responder em relação ao numero de *threads* (estas começam em 100 e vão subindo gradualmente até aos 7000), enquanto que a figura 2.2 apresenta o tempo de resposta a cada *thread*.

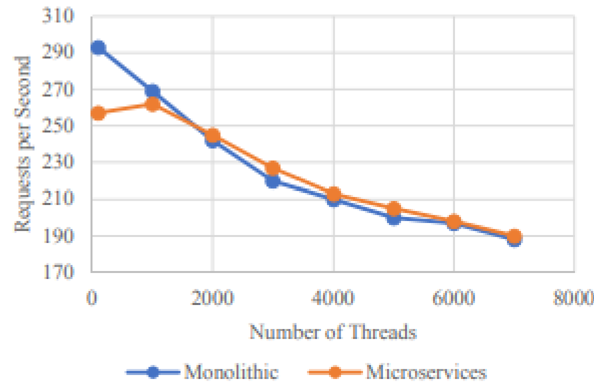

Figura 2.1: Resultados do Primeiro Teste [1].

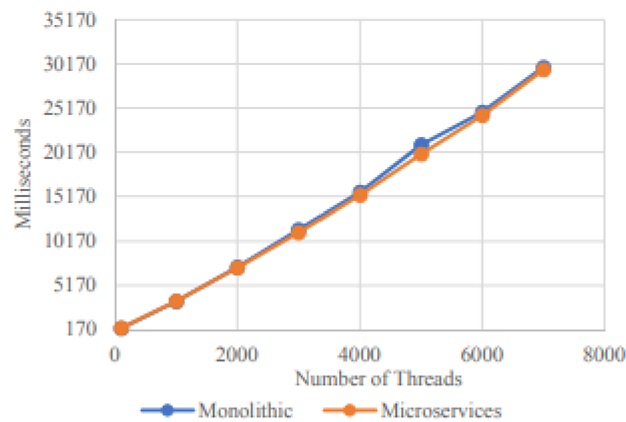

Figura 2.2: Resultados do Primeiro Teste em Relação ao Tempo de Resposta [1].

No segundo cenário de teste, os resultados foram diferentes em termos de performance, apresentados na figura 2.3. O número de pedidos foi fixado de modo a encontrar o número exato de pedidos que uma aplicação pode tratar por segundo, ou seja, encontraram o número máximo de pedidos que ambas conseguem tratar por segundo. A aplicação monolítica mostrou um rendimento mais elevado em média. Assim, a aplicação monolítica pode tratar os pedidos de uma forma mais rápida, concluindo que, esta pode ser utilizada quando o se pretende especialmente é que a aplicação trate os pedidos de uma forma mais rápida, sem pensar noutros fatores.

O último cenário de teste, incluiu uma comparação entre duas aplicações de microservi-

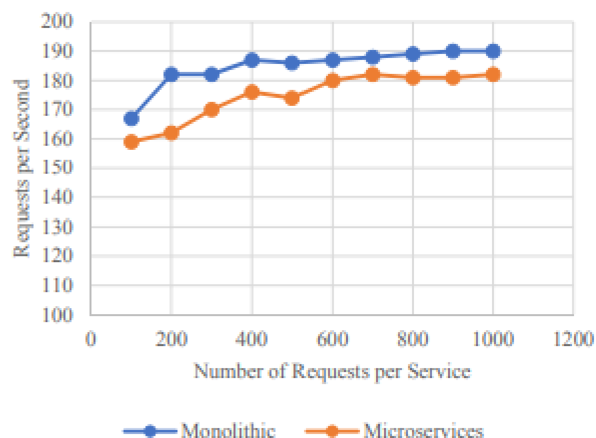

Figura 2.3: Resultados do Segundo Teste [1].

ços com diferentes tecnologias de descoberta de serviços, tais como *Eureka* e *Consul*. Os resultados deste teste indicaram que a aplicação de microserviços com *Consul* teve melhor desempenho do que a aplicação com tecnologia de *Eureka*, em termos de performance ou número de pedidos tratados por segundo, apresentou uma melhoria de 4% com a utilização do primeiro.

2.4 Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems

Este documento, visa desmistificar o uso ambíguo de termos e significados de *Service-oriented architecture (SOA)* e microserviços. Apresenta as duas arquiteturas detalhadamente e especifica características e diferenças entre as mesmas, apontando os seus pontos fortes e fracos. Enquanto tentam fornecer perspetivas tanto à investigação como à indústria, para apontar os pontos fortes e fracos de ambas as direções arquitetónicas, sempre salientando desvantagens e vantagens, em ambas as abordagens.

Concluíram que ambas as arquiteturas abordam a integração de sistemas, mas a indústria avança em direção a um aumento da utilização de microserviços, deixando a SOA como termo legado (passado). O principal crédito para esta tendência pode ser dado à capacidade de implementação de microserviços independentes e à escalabilidade elástica. Estes podem ser integrados com *Internet of Things (IoT)*, *Cloud computing* e sistemas para tratamento de grandes dados.

2.5 *Dynamic Microservices to Create Scalable and Fault Tolerance Architecture*

Este artigo, aborda os desafios levantados pela necessidade de desenvolver um sistema escalável e tolerante a falhas baseado em microserviços. Realizaram experiências onde con-

sideraram dois tipos de microserviços, simples e estendidos, e a solução proposta mostra-se inovadora principalmente pelo seu comportamento dinâmico.

A arquitetura proposta, demonstrada na figura 2.4, permite a criação de novos microserviços de forma bastante simples, que são automaticamente integrados no sistema construído. O elemento inovador desta arquitetura é a facilidade com que se pode escalar, sendo que a cada novo cliente é lhe atribuído uma tarefa pelo servidor, de acordo com a estratégia seguida. O servidor lida com a distribuição dinâmica de tarefas entre clientes, fornecendo um escalonamento dinâmico com base em vários parâmetros (o número de chamadas a uma tarefa, o seu tempo de execução ou combinações dos mesmos).

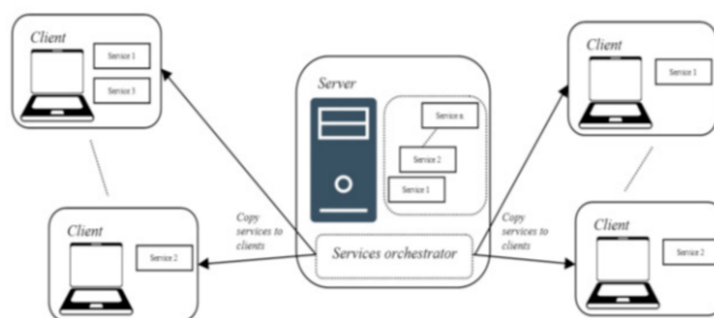

Figura 2.4: Arquitetura Proposta [2].

O protocolo de comunicação servidor-cliente é realizado nas seguintes etapas:

1. O cliente conecta-se ao servidor e inicia o protocolo de troca de chaves. Este também notifica ao servidor a porta a que está ligado;
2. O servidor notifica o cliente com uma tarefa a ser executada (uma tarefa é representada por um par (microserviço, dados de entrada));
3. O cliente recebe a tarefa e, em seguida, notifica o servidor que a transmissão e o *upload* foram bem-sucedidos ou falharam;
4. Uma vez estabelecida a conexão entre as duas entidades, o servidor envia os dados em formato *JavaScript Object Notation*(JSON), criptografados com *Data Encryption Standard* (DES) ao cliente para fazer o processamento;
5. O cliente recebe os dados e retorna a resposta também no formato JSON criptografado com DES.
6. A cada minuto o cliente notifica o servidor se está disponível ou não, para receber novas tarefas.

A arquitetura proposta pode ser aprimorada e expandida a qualquer momento, com disponibilidade para múltiplas plataformas (por exemplo, telemóvel, aplicação web, entre outros).

Concluíram, como direção futura os que utilizadores fornecessem o poder de processamento em troca de remuneração (por exemplo, o sistema *BITCOIN*), ou seja, aplicação é desenvolvida para executar microserviços em determinados computadores.

2.6 *Kubernetes as an Availability Manager for Microservice Applications*

Este artigo, apresenta uma das maiores preocupações em relação aos microserviços, que é a sua disponibilidade. A plataforma *Kubernetes* é de código aberto e define um conjunto de blocos de construção, que coletivamente fornecem mecanismos para a implementação, manutenção e disponibilidade. Assim, este esconde a complexidade da microorquestração de serviços enquanto gerem a sua disponibilidade.

Num trabalho preliminar, avaliaram o *Kubernetes*, usando a sua configuração padrão, a partir da perspetiva de disponibilidade em cenários privados de *cloud*.

Investigaram mais arquiteturas e realizaram mais experiências, com configurações diferentes, para avaliar a disponibilidade que o *Kubernetes* fornece para os microserviços que o próprio gere. Apresentam também diferentes arquiteturas para *clouds* públicas e privadas, avaliaram a disponibilidade alcançável através da capacidade de *healing* de *Kubernetes*, verificaram o impacto da adição de redundância na disponibilidade dos microserviços e, por fim, realizaram uma avaliação comparativa com a *Availability Management Framework* (AMF), que é uma *framework* comprovada como um serviço de *middleware* para a gestão de alta disponibilidade.

Concluíram, que o *Kubernetes* é mais adaptado para nuvens públicas do que para nuvens privadas. Descobriram que as ações de *healing* não são suficientes para proporcionar disponibilidade, especialmente alta disponibilidade. Observaram também que a adição de redundância pode diminuir significativamente o tempo de paragem desde que o *Kubernetes* deteta a falha até o serviço ser totalmente recuperado.

2.7 Conclusão

Neste capítulo, são apresentados artigos que explicam o tema de microserviços de forma detalhada, desde o inicio do seu uso até à atualidade, uma comparação deste com outras arquiteturas de desenvolvimento de aplicações, como o caso da arquitetura orientada por serviços e a monolítica. Por fim, apresenta as vantagens de integrar o *kubernetes* com a arquitetura de microserviços, proporcionando disponibilidade e escalabilidade aos projetos.

Concluindo, nesta investigação por informação sobre microserviços ou aplicações que utilizem esta arquiteturas demonstrou que as empresas de telecomunicações, que utilizam esta

arquitetura, não disponibilizam informação de como estes foram utilizados nas suas aplicações, de forma a não expor a sua lógica de negócio. Esta serviu também como base para a realização do capítulo 4, de forma a manter toda a explicação coerente e estruturada.

Capítulo 3

Metodologias e Ferramentas Utilizadas

3.1 Introdução

Neste capítulo, são apresentadas e discutidas as metodologias implementadas (secção 3.2) durante o decorrer do estágio, bem como as ferramentas já utilizadas no desenvolver do projeto. Algumas destas ferramentas foram escolhidas pela empresa cliente e membros presentes na criação das arquiteturas implementadas e outras apenas pela nossa equipa.

3.2 Metodologias

3.2.1 *Agile/Scrum*

O Agile/scrum é uma metodologia de desenvolvimento ágil utilizada no desenvolvimento de *software*, baseada em processos iterativos e incrementais e é uma evolução do *Agile Management*, representada na figura 3.1. Possui uma estrutura ágil/adaptável, rápida, flexível e eficaz, projetada para fazer entregas ao cliente durante todo o desenvolvimento do projeto.

O principal objetivo do *scrum* é satisfazer a necessidade do cliente, por meio de um ambiente de transparência na comunicação, responsabilidade coletiva e progresso contínuo. O desenvolvimento parte de uma ideia geral do que precisa ser construído, elaborando uma lista de características ordenadas por prioridade (*product backlog*), que o dono do produto deseja obter. [3] Todo este processo é utilizado por uma equipa *scrum*, que normalmente consiste nos seguintes papéis:

- Dono do produto (*product owner*) – é o representante direto do cliente. Concentram-se mais na parte comercial e são responsáveis pelo retorno do inventimento no projeto. Eles traduzem a visão do projeto para a equipa, validam os benefícios em *stories* a serem incorporadas ao *Product Backlog* e priorizam os mesmos regularmente;

- *Scrum Master* – é a pessoa que lidera a equipa orientando-a para o cumprimento das regras e processos da metodologia. Este gere a redução dos impedimentos do projeto e trabalha com o dono do produto para maximizar o retorno do investimento;

- Equipa *Scrum* – é a equipa responsável pelo desenvolvimento de *stories* presentes no product backlog.

O Scrum é executado em blocos temporários curtos e periódicos, chamados de *sprints*, que geralmente variam de 1 a 4 semanas, que é o termo para feedback, reflexão, desenvolvimento

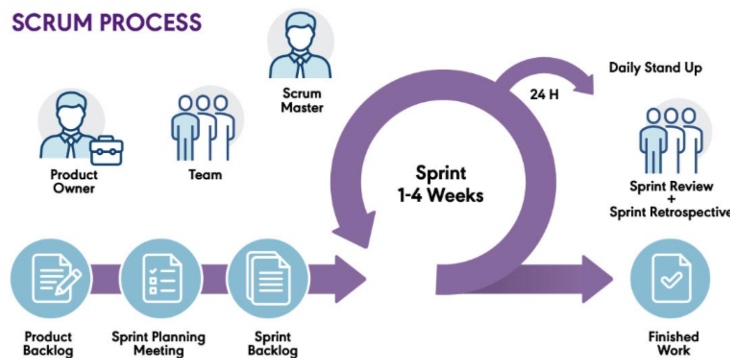

Figura 3.1: Metodologia Srcum/Agile [3].

e entrega. Nestes *sprints* definem-se os objetivos e quais as *stories* que devem ser desenvolvidas tendo em conta a disponibilidade da equipa.

O processo tem como ponto de partida uma lista de objetivos/requisitos que compõem o plano do projeto. É o cliente que prioriza esses objetivos, assim são determinadas as iterações e consequentes entregas. Todas estas interações são chamadas de eventos. Estes podem ser do tipo:

- *Sprint* – é a unidade básica de trabalho. Essa é a principal característica que marca a diferença entre o *scrum* e outros modelos de desenvolvimento ágil;
- *Sprint Planning* – onde se define o que será feito no decorrer do *sprint* e como isso será feito. Essa reunião é realizada no início de cada *sprint* e é definido como será abordado o projeto vindo das etapas e prazos do *product backlog*;
- *Daily Scrum* – o objetivo desta é avaliar o progresso do *sprint*, onde se explicam as atividades e se cria um plano para as próximas 24 horas. É uma breve reunião que ocorre diariamente durante o período da *sprint*. Normalmente três perguntas acabam por ser respondidas individualmente: O que eu fiz ontem? O que eu vou fazer hoje? Que ajuda eu preciso? O *Scrum Master* ou a equipa devem tentar resolver os problemas ou obstáculos que surgirem;
- Revisão do *sprint* – mostra qual trabalho foi concluído em relação ao *backlog do produto*. O *sprint* é revisto e já deve haver um avanço claro e tangível no produto a ser apresentado ao cliente;
- Retrospetiva do *sprint* – é quando a equipa revê os objetivos cumpridos do *sprint* finalizado e fala-se sobre a performance de cada elemento da equipa. Esta etapa serve para implementar melhorias do ponto de vista do processo de desenvolvimento.

3.2.2 Vantagens da Metodologia Agile/Scrum

Esta metodologia tem muitas vantagens sobre outras de desenvolvimento ágil. Atualmente

é a *framework* de referência mais utilizada e confiável na indústria de software. Em seguida encontra-se as suas principais vantagens:

- Facilmente escalável – os processos *scrum* são iterativos e são tratados em períodos de trabalho específicos, o que torna mais fácil para a equipa focar-se em funcionalidades definidas para cada período. Isso não só tem o benefício de alcançar melhores entregas de acordo com as necessidades do utilizador, mas também permite que estas dimensionem os módulos em termos de funcionalidade, design, desenvolvimento e características de maneira ordenada, transparente e simples;

- Cumprimento das expectativas – o cliente estabelece as suas expectativas indicando o valor que cada requisito/*story* do projeto traz, a equipa faz uma estimação e com essas informações o dono do produto estabelece a sua prioridade. Regularmente, nas demonstrações do *sprint*, este verifica se os requisitos foram atendidos e transmite *feedback* para a equipa;

- Flexível a mudanças – reação rápida a mudanças de requisitos gerados pelas necessidades do cliente ou desenvolvimentos de mercado. A metodologia é projetada para se adaptar às mudanças de requisitos que envolvem projetos complexos;

- *Redução do* Time to Market – o cliente pode começar a utilizar as funcionalidades mais importantes do projeto antes que o produto esteja totalmente pronto;

- Melhor qualidade de software – o método de trabalho e a necessidade de obter uma versão funcional após cada iteração, ajuda a obter um software de melhor qualidade;

- *Timely Prediction* – permite conhecer a velocidade média da equipa por *sprint (story points)*, com a qual, consequentemente, é possível estimar quando determinada funcionalidade, que ainda está no *backlog*, estará disponível;

- Redução de riscos – o fato de realizar as funcionalidades mais prioritárias em primeiro lugar e conhecer a velocidade com que a equipa avança no projeto, permite eliminar os riscos com eficácia e antecedência.

3.2.3 Utilização da Metodologia Agile/Scrum

Neste projeto, a minha equipa está a utilizar *sprints* com duração de duas semanas, ou seja, a equipa tem esse período de tempo para realizar as *stories* definidas no *sprint planning*. Durante o *sprint* são realizados todos os dias depois de almoço uma reunião (*Daily Standup*) para a equipa ficar a par sobre o trabalho de cada membro. Nesta reunião, cada membro informa o que fez desde a ultima reunião até à presente, o que vai fazer depois da mesma e se tem problemas ou está bloqueado em algum ponto da *story* (isto pode ajudar a que estes problemas sejam resolvidos mais depressa, com maior eficiência). Também durante o *sprint* é feita uma revisão ao estado atual da *sprint*, e esta tem o propósito de criar de novas *stories* (tarefas /funcionalidades), que são guardadas no *product backlog*, se estas forem necessárias. Cada *story* tem explicada a lógica do negócio e comportamento esperado

quando finalizada. No final da explicação é atribuída a cada *story* um valor que indica a sua dificuldade/complexidade, valor este que é a média da votação de todos os membros da equipa.

No final de cada *sprint*, é necessário efetuar uma revisão e retrospetiva. A revisão tem como finalidade rever e analisar o resultado do desenvolvimento feito no *sprint* em questão, enquanto a retrospetiva serve para analisar o desempenho dos membros da equipa, de forma a identificar o que correu bem, o que correu mal e sugerir uma ação em conformidade.

Por fim, é feita uma demonstração do desenvolvimento efetuado durante a *sprint*. Para o caso de o desenvolvimento atual não ser suficiente, a demonstração pode ser adiada com o consentimento do dono do produto até conter funcionalidades suficientes que provem o funcionamento mínimo da *sprint*.

3.3 Ferramentas Utilizadas

Nesta secção são apresentadas as ferramentas (estas foram escolhidas pelo cliente e pelos arquitetos funcionais) utilizadas no desenvolvimento deste projeto, durante o estágio, e algumas extensões que são essenciais, de modo a facilitar o desenvolvimento e os testes que lhe estão associados.

3.3.1 Postman

O postman [23] é uma plataforma para os utilizadores projetarem, construírem, testarem e iterarem as API's desenvolvidas. Esta permite também monitorizar todos os pedidos realizados, pode ser integrada em repositórios para manter o código fonte, entre outras funcionalidades que não são utilizadas pela equipa de desenvolvimento.

No entanto, neste projeto o Postman foi utilizado meramente para obter as respostas atuais das API's, realizar testes integrados e construir uma coleção de pedidos que cada microserviços pode ter, de forma a facilitar o trabalho dos *testers* da equipa do cliente. Visto que a coleção de pedidos são utilizadas por eles para testarem os comportamentos dos microserviços implementados. Isto para se poder estruturar os modelos corretamente e também para resolução de defeitos.

O Postman permite ainda guardar os pedidos já efetuados, bem como os seus requisitos (*header, body, authentication*, entre outros) por ambiente, que podem posteriormente ser utilizados para pesquisa de defeitos.

Por fim, este oferece mais funcionalidades, mas estas, por decisão da equipa, foram utilizadas por outra aplicação chamada *apache jmeter*, que foi utilizada principalmente para realizar testes de carga e gerar relatórios e gráficos sobre esses testes de carga.

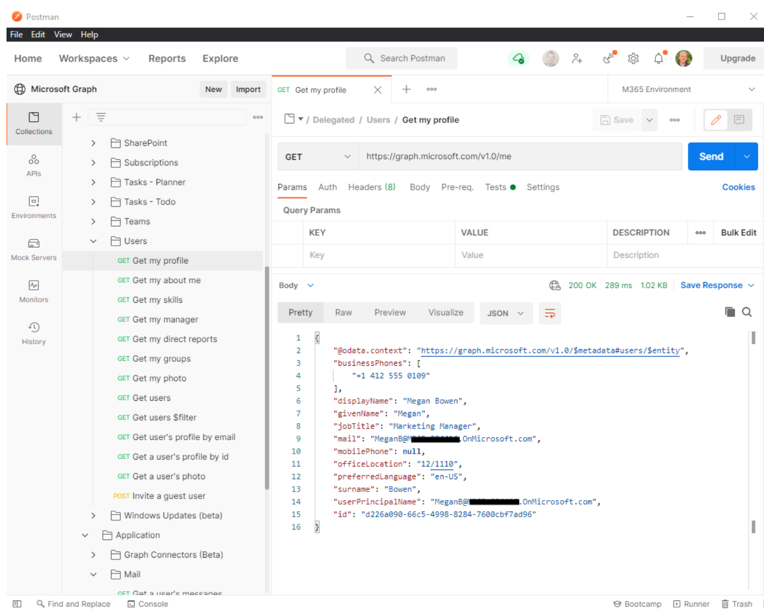

Figura 3.2: Postman IDE [4].

3.3.2 Apache JMeter

O apache jmeter[24] é uma aplicação que foi desenvolvida inicialmente com objetivo de realizar testes em aplicações (tal como o Postman), mas atualmente é possível utilizá-lo principalmente para testes em recursos diferentes, como base de dados, servidores de email, requisições *Hypertext Transfer Protocol* (HTTP) e *Simple Object Access Protocol* (SOAP) e outros.

Com este, foram realizados principalmente testes de performance e carga, tornando possível medir a performance de uma determinada *Application Programming Interface* (API) (ou da lógica toda por detrás desta). Assim conseguimos saber o quão otimizada esta se encontra, se está dentro dos padrões do cliente ou se esta parte com alguma carga de pedidos.

Um exemplo de utilização, na interface apresentada na figura 3.3, é o teste de todas as requisições que um utilizador faria, simulando chamadas aos microserviços. Essas ações ficam gravadas no jmeter, numa estrutura conhecida como "grupo de teste". Após a gravação destas ações, o jmeter possibilita disparar múltiplas chamadas destas ações, simulando um grupo de utilizadores. No final, a resposta do servidor para cada solicitação feita é guardada e, com base nessas respostas, as estatísticas são calculadas e as métricas de performance são geradas. O objetivo do jmeter é simular cenários de testes mais reais possíveis.

3.3.3 SoapUI

O SoapUI é uma ferramenta que se concentra na segurança e qualidade de API's e serviços web. Esta suporta métodos mais conhecidos como *Representational State Transfer* (REST) e SOAP e pode ser utilizada para testes funcionais, bem como testes de desempenho (teste de carga).

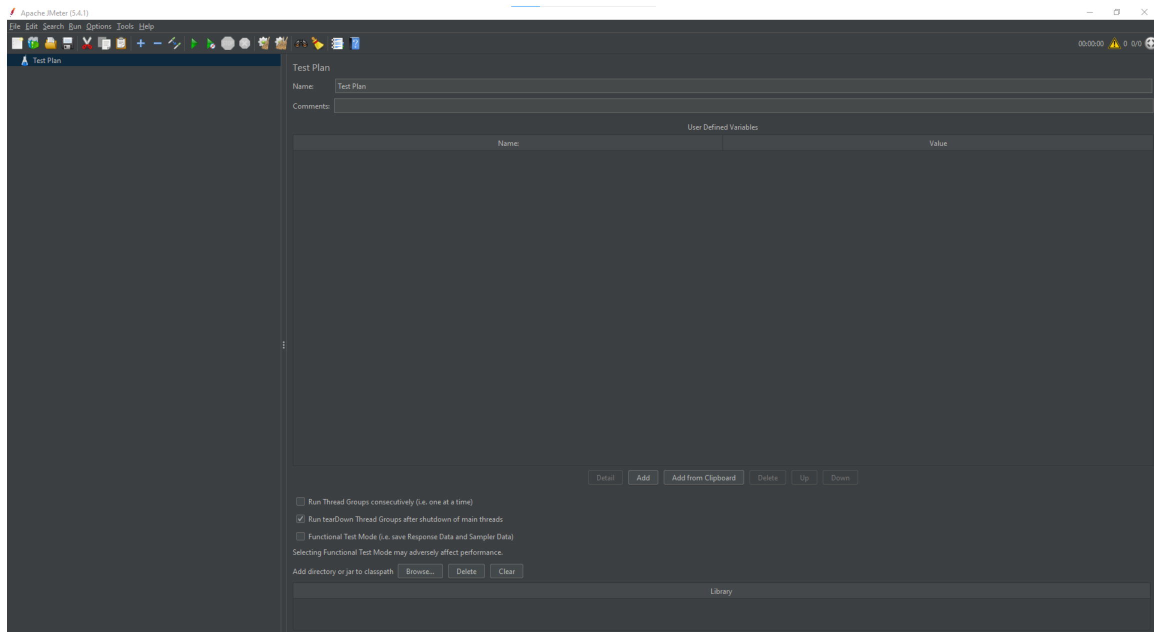

Figura 3.3: Apache JMeter IDE [5].

A seguir estão listados alguns dos principais recursos fornecidos pelo SoapUI:

- Testes funcionais de API e serviços web;
- Simulação de serviços web;
- Testes de segurança e controle de qualidade;
- Teste de carga.

Este foi escolhido pela equipa, para simular respostas de outros microserviços ou de serviços externos. Desta forma, permite criar uma *mock* de uma API, simulando a sua resposta sempre que este é invocado. Isto permite utilizar e testar um microserviço de forma isolada sem ter que correr os microserviços integrados com este, poupando recursos do computador, tempo de implementação e testar toda a implementação do mesmo apenas alterando a mensagem e o *HTTP Status* da resposta, como se pode ver na figura 3.4.

3.3.4 Draw.IO

O draw.io[25] é um software de desenho gráfico e a sua interface pode ser utilizada para criar diagramas como fluxogramas, *wireframes*, diagramas *Unified Modeling Language* (UML), organogramas e diagramas de rede, como se pode verificar no exemplo da figura 3.5.

Este foi utilizado no projeto para desenhar arquiteturas, diagramas de fluxo e atividade, para ter uma documentação atualizada no *Confluence*. Assim todas as equipas do projeto estão a par da solução, como o microserviço funciona ou determinada funcionalidade está

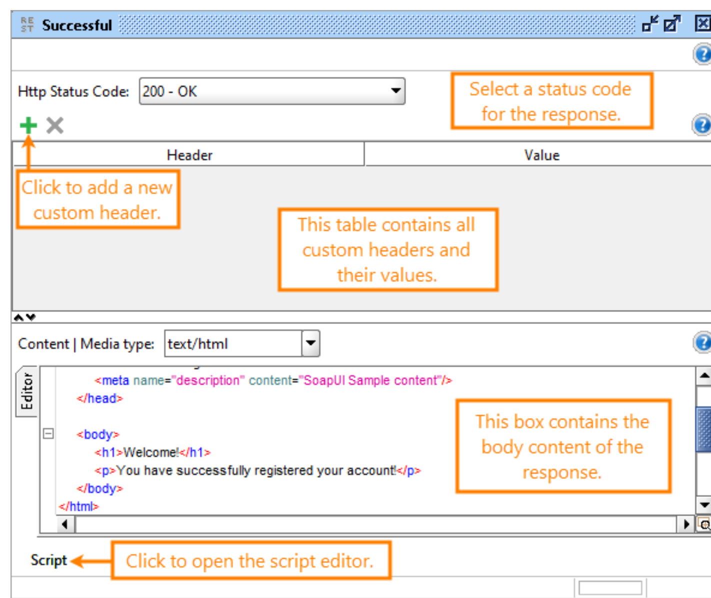

Figura 3.4: Exemplo de resposta de uma API no SoapUI [6].

feita. Serve também de auxilio a outros membros do projeto que não interagiram no desenvolvimento de uma certa funcionalidade ou microserviço.

3.3.5 Jira

O Jira [8] é uma ferramenta que permite monitorizar tarefas e acompanhar projetos garantindo a gestão de todas as suas atividades num único lugar. Este ajuda a manter a metodologia *agile/scrum*. Através dela consegue-se ter a noção do progresso de cada *story*.

Na figura 3.6 pode se ver o *active board* ou *current sprint*, onde temos 4 colunas, *TO DO*, *In Progress*, *Code Review* e *Done*. Na coluna *TO DO* são colocadas as *stories* para o *sprint* ou o *sprint backlog*, ou seja, são colocados os *sprints* planeados com todas as *tasks* associadas. Na *In Progress* estão os *stories* que estão a ser desenvolvidas por algum membro da equipa. Na coluna *Code Review* são colocadas as *stories* terminadas, mas que aguardam aprovação de um membro responsável por subir as *features* para o *branch* dev. Na última coluna, (*Done*) são colocadas as que foram declaradas como feitas ou canceladas. Na coluna da esquerda podemos ver o *backlog*, onde estão as que se encontram no *product backlog*, ou seja, as *stories* que ainda faltam desenvolver.

Esta aplicação permite aos membros da equipa, bem como ao *Project Manager* (PM) e *Team Leader* (TL), uma melhor gestão e eficiência do projeto, como ter o processo todo do projeto registado numa plataforma de gestão de projetos.

3.3.6 Confluence

O *Confluence* é uma *wiki* desenvolvida pela empresa de software e é utilizada como uma ferramenta de partilha de informação/conhecimento, onde por norma as organizações criam

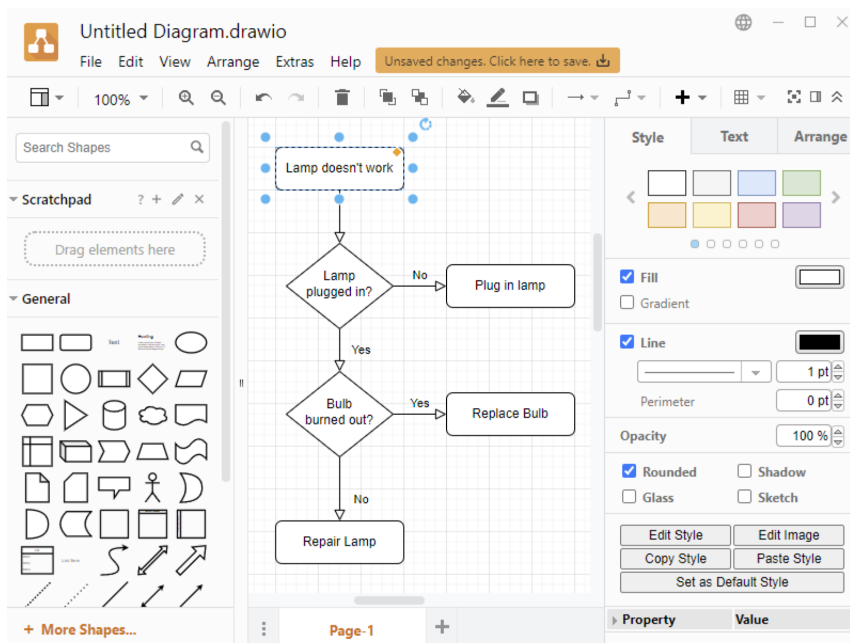

Figura 3.5: Exemplo de utilização do Draw.IO [7].

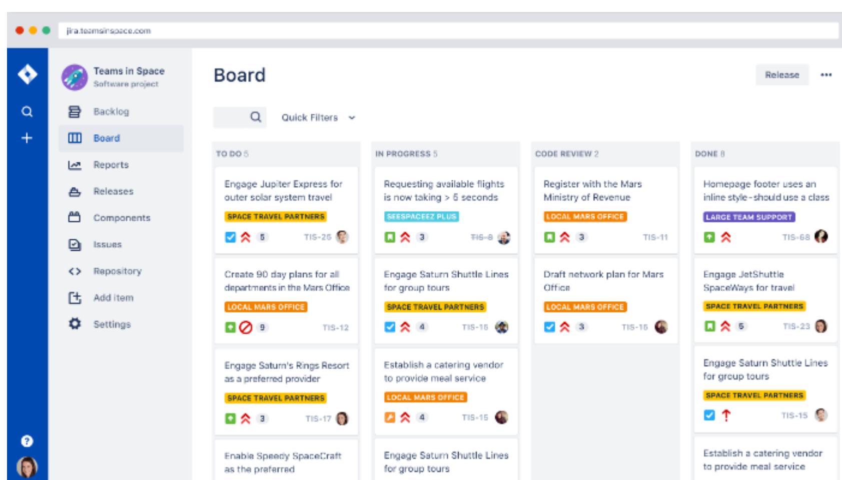

Figura 3.6: Exemplo de um quadro do Jira [8].

várias páginas dedicadas a múltiplos assuntos, de outra forma, é onde se encontra toda a documentação do projeto. Na imagem 3.7 pode se ver o exemplo de uma página de documentação [26].

Neste caso, este contém informação desde sobre como preparar o computador para poder correr o projeto, informação sobre as arquiteturas desenvolvidas, que depois viram *stories* ou *features*. Esta serve de documentação para os membros da equipa consultarem durante o desenvolvimento. Por norma, esta documentação é feita inicialmente pelos arquitetos funcionais e depois concluída pelos membros que desenvolveram a *story*. Basicamente, é uma plataforma com toda a informação sobre o projeto e empresa.

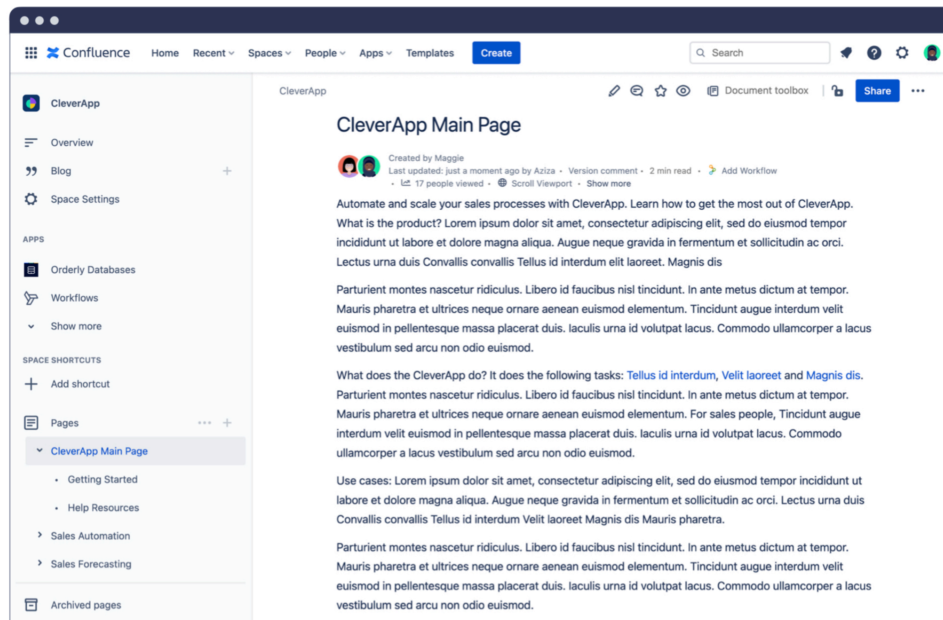

Figura 3.7: Exemplo de uma página do *Confluence* [9].

3.3.7 Git/GitLab

O Git é um sistema de controlo de versões utilizado para gerir projetos de software. Este permite acompanhar as alterações do código ao longo do tempo, colaborar em projetos com outros membros, e facilmente reverter as alterações, se necessário. Funciona criando instâncias do código, e depois armazenando essas instâncias num repositório.

O Git controla então vários repositórios, onde se encontra armazenado o código fonte da aplicação e os *branches* (ramos) a ser trabalhos no momento.
O fluxo dos *branches* (ramos) do repositório é relativamente semelhante ao mostrado na figura 3.8. Nesta conseguimos identificar os cinco ramos principais: o *master* (ou *main*), *develop*, *feature*, *release* e *hotfix*, onde fica a faltar a nomenclatura *fix* que tem um comportamento semelhante às *features* (funcionalidades).

Por outro lado, o Gitlab é um gestor de repositório Git, que fornece funcionalidades adicionais sobre o mesmo. Uma das suas principais características são os *pipelines CI/CD*, que permitem testar e implementar automaticamente alterações de código à medida que são feitas, ajudando a assegurar que o código está sempre em estado de funcionamento. O Gitlab também fornece características: como rastreio de problemas, revisão de código, e páginas *wiki*, tornando-o uma solução abrangente para gerir projetos de software. Estas páginas, foram utilizadas, neste projeto, para criar documentação interna apenas para equipa de desenvolvimento da *ReadinessIT*, de forma a todas as tecnologias, implementações, bibliotecas estarem descritas e como as utilizar [27].

Uma outra vantagem do Gitlab é que pode ser auto-hospedado, o que significa que as or-

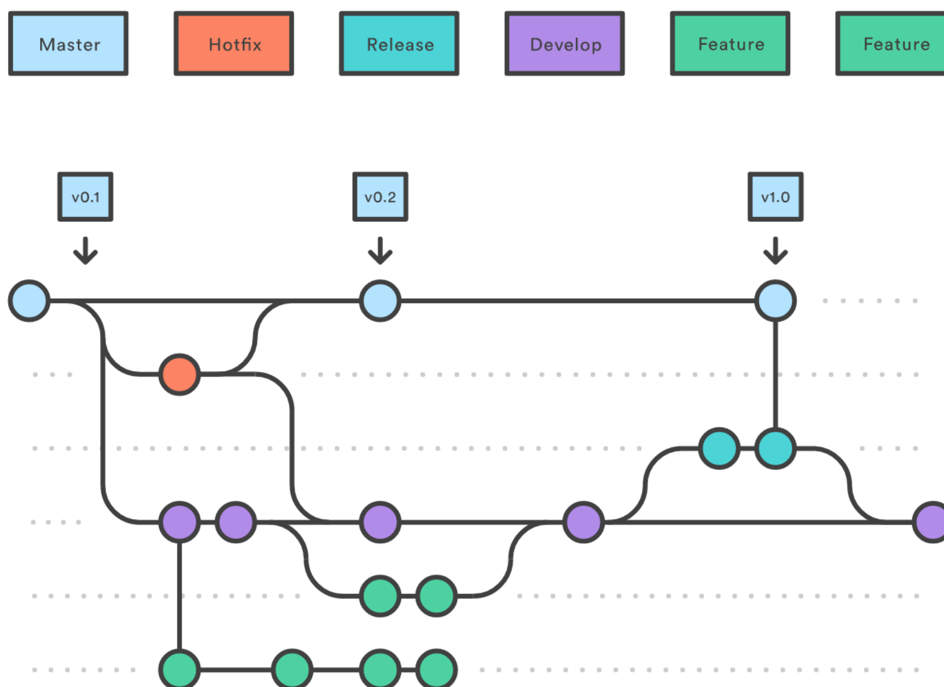

Figura 3.8: Exemplo dos *branches* do Git [10].

ganizações podem instalar e executar as suas próprias instâncias do Gitlab nos seus próprios servidores. Isto dá às organizações mais controlo sobre o seu código e dados, e pode também ajudar a melhorar o desempenho e reduzir custos.

Globalmente, o Git e o Gitlab são ferramentas que podem ajudar a gerir os seus projetos de forma mais eficiente e colaborativa. Ao utilizar estas ferramentas, assegura-se que o código está sempre em estado de funcionamento, e que as alterações são acompanhadas e revistas de forma controlada.

3.3.8 IntelliJ IDEA

O IntelliJ IDEA é um *Integrated Development Environment* (IDE) para programar na linguagem Java. Tornou-se uma das ferramentas mais utilizadas para o desenvolvimento em Java, devido à sua interface de fácil utilização e características/funcionalidades, e ainda fornece suporte para *frameworks* como *Spring* e *Hibernate*, esta interface encontra-se apresentada na figura 3.9 [28].

Uma das suas características principais é a funcionalidade inteligente de completar código. Pode prever a próxima linha de código, sugerir nomes de variáveis, e até completar blocos de código inteiros. Isto poupa uma quantidade significativa de tempo e esforço e também possui uma variedade de ferramentas de *debugging* e testes, facilitando a identificação e correção de erros no código. Integra-se com estruturas de teste populares como *JUnit*, permitindo assim executar testes unitários diretamente a partir do mesmo e visualizar os resultados em tempo real.

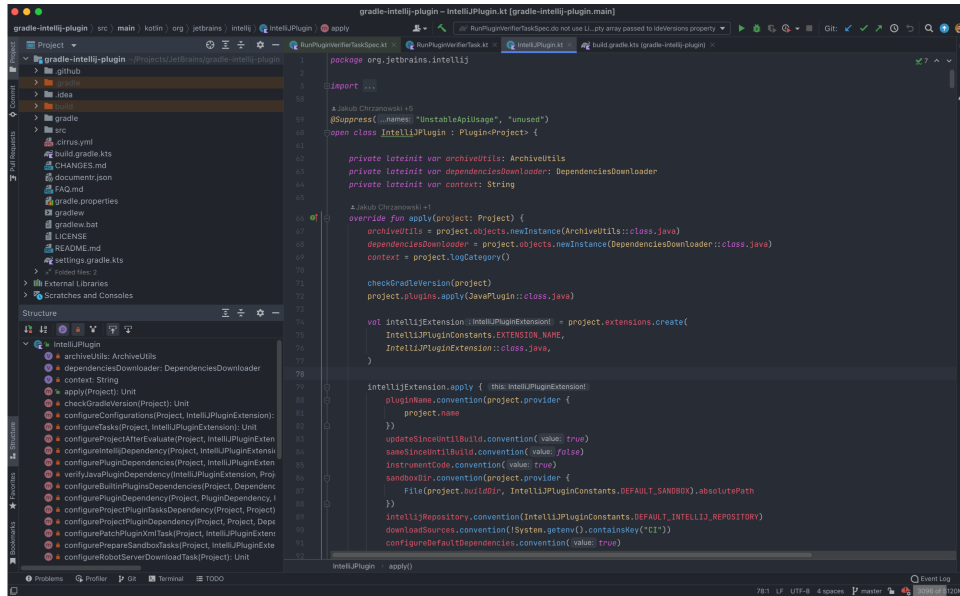

Figura 3.9: Interface do IntelliJ IDEA[11].

Possui uma vasta gama de *plugins*, que fornecem funcionalidades adicionais, tais como a integração com sistemas de controlo de versões como Git, suporte para outras linguagens de programação, ferramentas de base de dados, analisadores de código, entre outros.

Em geral, é uma ferramenta essencial para desenvolvimento em Java, oferecendo uma gama de funcionalidades que ajudam a simplificar o processo de desenvolvimento e a melhorar a qualidade do código.

3.3.9 PostgreSQL

O PostgreSQL é uma ferramenta que funciona como sistema de gestão de base de dados relacionais e o seu principal objetivo é permitir a implementação da linguagem SQL em estruturas de dados relacionais.

A utilização deste tem crescido consideravelmente nos últimos tempos, muito por conta da sua facilidade de utilização e pela sua alta compatibilidade com diferentes padrões de linguagem. Permite a armazenar informações de forma segura e, se necessário, restaurá-las sempre que houver solicitação de outras aplicações integradas [29].

Este foi escolhido para este projeto, porque é um sistema que lida bem com altos volumes de solicitações e com cargas de trabalho grandes, ou seja, funciona muito bem para aplicações com enorme intensidade de acessos. Na área de telecomunicações é um ótimo exemplo de estrutura, que precisa desse sistema para ter um desempenho otimizado, devido ao alto número de acessos simultâneos recebidos.

3.3.10 pgAdmin

O pgAdmin é o software gráfico, *open source*, com mais funcionalidades de desenvolvimento para a linguagem PostgreSQL. Esta permite fazer todas as tarefas necessárias de administração de base de dados. O pgAdmin é instalado automaticamente quando é instalado o PostgreSQL[12].

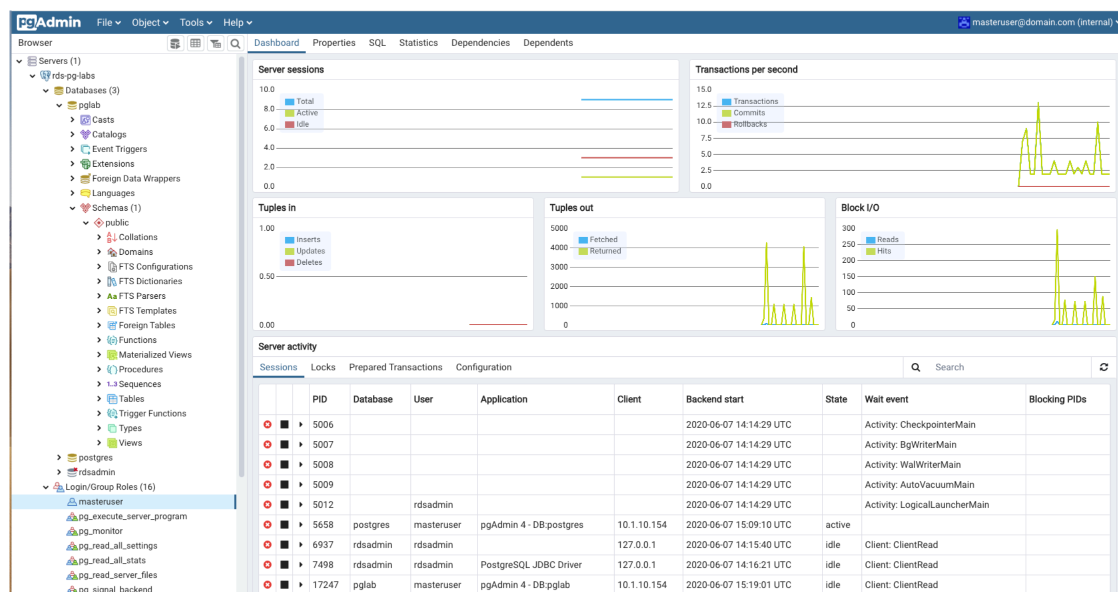

Figura 3.10: Exemplo de utilização do pgAdmin [12].

Este foi escolhido pela equipa e pelo cliente, porque quando instalado como um *docker container* não sobrecarrega tanto o computador, como outro software gráfico que suporte a linguagem *PostgreSQL*, e este foi desenvolvido para lidar com todas as funcionalidades desta linguagem, de forma a termos o total aproveitamento da mesma.

3.3.11 Docker Desktop

O *Docker* é uma aplicação, que permite aos utilizadores executar e gerir aplicações em *containers* nas suas máquinas locais, e esta pode ser utilizada através da linha de comandos ou da interface disponível *docker desktop*. Fornece uma interface fácil de utilizar para construir, testar, e implementar aplicações utilizando *containers* [30].

A aplicação *desktop*, apresentada na figura 3.11, fornece uma interface gráfica ao utilizador que facilita a criação, arranque, paragem e remoção de *containers* de forma fácil e intuitiva. Também inclui ferramentas para gerir imagens, redes, e volumes.

Uma das principais vantagens da sua utilização é a sua capacidade de proporcionar um ambiente de desenvolvimento consistente e isolado. Podem se criar *containers* com exatamente as mesmas configurações e dependências que os ambientes de produção, assegurando que as aplicações funcionem sem problemas quando forem implementadas nos ambientes. Oferece também uma vasta gama de ferramentas e características, que simplificam o processo

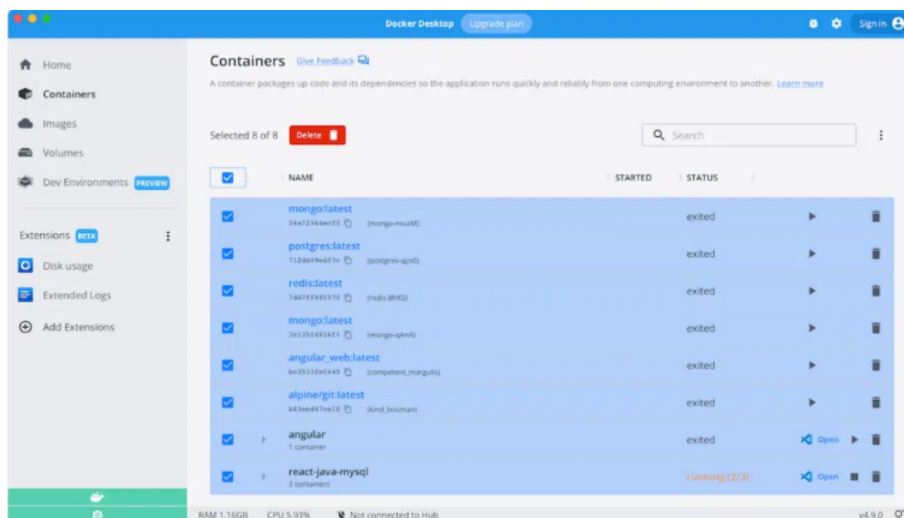

Figura 3.11: Docker Desktop *workspace* [13].

de criação e gestão de *containers*.

Possui a capacidade de trabalhar sem problemas com outras ferramentas e serviços *Docker*. Por exemplo, pode integrar-se com o *Docker Hub*, que é utilizado para armazenar e partilhar imagens de *containers*, como *Docker Composes*, que é uma ferramenta utilizada para definir e executar imagens multi*containers*.

Neste projeto, foi utilizado de forma a correr aplicações como o *kafka*, *rabbitMQ*, *postgreSQL*, entre outros, para não sobrecarregar tanto o computador devido à utilização de várias tecnologias/aplicações, para realizar testes integrados a alguma funcionalidade ou microserviço.

3.3.12 Kafka

O *kafka* é uma plataforma de *streaming* de eventos. Foi desenvolvido para lidar com grandes volumes de fluxos de dados em tempo real e para fornecer mensagens fiáveis, escaláveis e tolerantes a falhas entre diferentes sistemas [31].

A arquitetura do *Kafka* é baseada no modelo *publish-subscribe*, onde os que produzem o evento publicam mensagens para um tópico, e os consumidores subscrevem os tópicos para receber essas mensagens. Este é horizontalmente escalável, o que significa que pode tratar grandes quantidades de dados adicionando mais nós a um *cluster*.

Um tópico é uma categoria ou um nome para a qual as mensagens são publicadas. Os tópicos podem ser particionados em múltiplas partições, o que permite uma escalabilidade horizontal e um melhor desempenho.

Os *brokers* são os componentes centrais da arquitetura, apresentada na figura 3.12. São responsáveis pelo armazenamento e replicação das mensagens, que são publicadas para as

partições de um tópico.

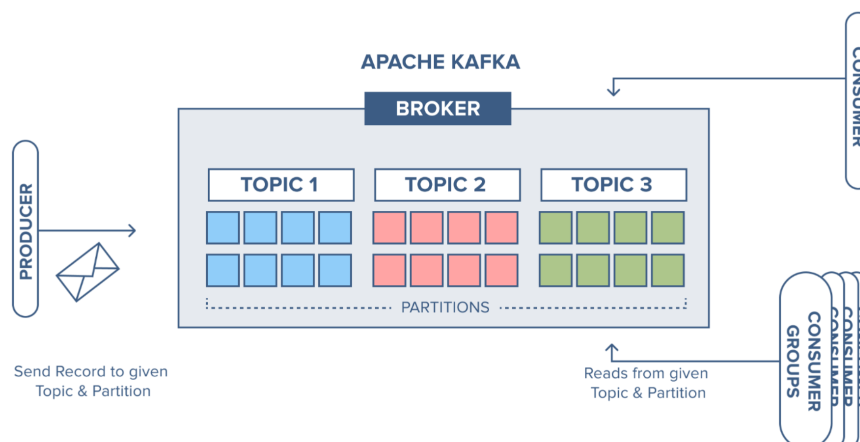

Figura 3.12: Arquitetura do Apache Kafka [14].

A arquitetura de Kafka está concebida para ser tolerante a falhas e altamente escalável. A replicação de dados entre vários *brokers* garante que os dados não se perdem em caso de falha de um destes.

O *kafka* foi utilizado principalmente para armazenar eventos de criação de ordens, por exemplo, um utilizador faz uma encomenda online, a aplicação cria um evento e publica-o no *kafka*, e outro microserviço responsável por tratar dos eventos consume esse evento e faz a sua função.

3.3.13 RabbitMQ

O RabbitMQ é um *broker* de mensagens que implementa o *Advanced Message Queuing Protocol* (AMQP). Foi desenvolvido com o objetivo de permitir o envio de mensagens fiáveis entre aplicações e sistemas, suportando uma variedade de padrões de envio de mensagens, tais como *point-to-point*, *publish-subscriber*, e resposta a pedidos [15].

Desta forma, este pode se descrever como uma fila de mensagens, o que significa que permite que as mensagens sejam armazenadas numa fila até que possam ser processadas por um consumidor. Quando uma mensagem é publicada, é entregue a uma fila, e os consumidores podem então recuperar as mensagens da fila numa pedido *first-in-first-out* (FIFO).

A arquitetura do RabbitMQ, apresentada na figura 3.13, é baseada num modelo distribuído, o que permite uma escalabilidade horizontal e um melhor desempenho. Um *cluster* do RabbitMQ consiste em vários nós, cada um dos quais é responsável pelo armazenamento e processamento de mensagens. Os nós comunicam entre si para assegurar que as mensagens são entregues de forma fiável e eficiente.

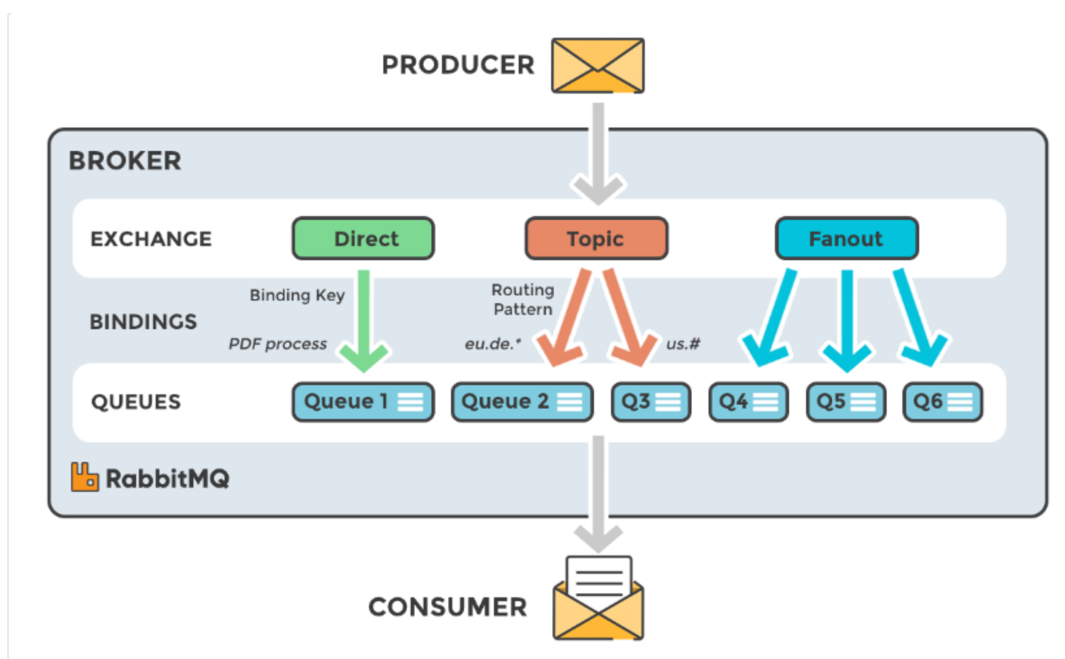

Figura 3.13: Arquitetura do RabbitMQ [15].

O RabbitMQ também suporta o reconhecimento de mensagens, o que significa que os produtores podem ser notificados quando as mensagens forem entregues e processadas com sucesso pelos consumidores. Isto ajuda a assegurar que as mensagens não sejam perdidas ou duplicadas no sistema.

Este teve uma utilização muito idêntica ao *Kafka* no projeto, dependendo da rapidez que se queira ter no microserviço, em relação ao processamento dos eventos. A diferença entre estes é explicada na subsecção seguinte.

3.3.13.1 Kafka vs RabbitMQ

O RabbitMQ e Apache Kafka são dois *brokers* de mensagens, cada um com os seus próprios pontos fortes e fracos.

Uma das principais diferenças entre RabbitMQ e Apache Kafka é a abordagem no processamento das mensagens. A fila de mensagens centralizada do RabbitMQ permite um processamento de mensagens mais fiável, enquanto que o modelo distribuído do Kafka é otimizado para um elevado rendimento e baixa latência. Além disso, Kafka é frequentemente utilizado para processamento de dados em tempo real, enquanto que o RabbitMQ é mais adequado para os padrões tradicionais de fila de mensagens.

Em seguida, mostra-se uma comparação de ferramentas entre o Kafka e o rabbitMQ:

- **Ordem de mensagem** – Enquanto o kafka tem esta ferramenta devido à sua característica de partição e por todas mensagens serem guardadas por uma *key*, o rabbitMQ não suporta esta ferramenta;

- **Tempo de vida das mensagens** – O kafka por sua natureza é um *log* e guarda as mensagens por *default* e o rabbitMQ é uma *queue*, logo perdem se as mensagens mal elas sejam consumidas;

- **Garantia de entrega** – O kafka retém todas as mensagens dentro das suas partições e esta garante que todas estas sejam consumidas com sucesso ou insucesso. Enquanto o rabbitMQ não garante esta entrega, nem quando se usa apenas uma *queue*;

- **Mensagens prioritárias** – No rabbitMQ pode se ter mensagens com maior prioridade em ser consumidas do que outras, por outro lado o kafka não suporta tal ferramenta.

Em suma, a escolha entre RabbitMQ e Apache Kafka depende das necessidades específicas da aplicação ou sistema. O RabbitMQ é um *broker* de mensagens fiável e flexível, que se adapta bem aos padrões tradicionais de mensagens e integração com múltiplos protocolos. O Kafka, por outro lado, é uma plataforma de transmissão de eventos, que é otimizada para a transmissão de dados em tempo real e elevada eficácia.

3.3.14 Maven

O *Maven* é uma ferramenta utilizada para automatizar a construção e compilação utilizada no desenvolvimento de um projeto, com a linguagem de programação Java. Fornece uma forma robusta e padrões para construir e gerir projetos, incluindo bibliotecas, estruturas, e aplicações [32].

Este segue o princípio de convenção sobre configuração, o que significa que os utilizadores adotam uma estrutura padrão de estruturação do projeto e convenção de nomes, tornando mais fácil a compreensão e manutenção dos mesmos. Utiliza uma abordagem declarativa, em que se especificam as dependências e constroem a configuração num ficheiro *extensible markup language* (XML) chamado *pom.xml*.

Este é também responsável pela gestão das dependências dos projetos, descarrega e gere automaticamente todas as bibliotecas e dependências necessárias, facilitando a utilização da linguagem Java aos utilizadores permitindo que se concentrem no código e não se preocupem com a infraestrutura subjacente, para isto utiliza um repositório central denominado repositório central *maven*. Este possui uma vasta gama de *plugins*, que alargam a sua funcionalidade. Estes podem ser utilizados para executar várias tarefas, tais como executar testes unitários, gerar documentação, distribuir artefactos para um repositório remoto, e muito mais. Parte deste funcionamento, está representado na figura 3.14, que demonstra a construção de um programa Java utilizando *maven*.

As principais funcionalidades do *maven* estão descritas a seguir:

- Gestão das dependências;

- Compilar o código fonte;

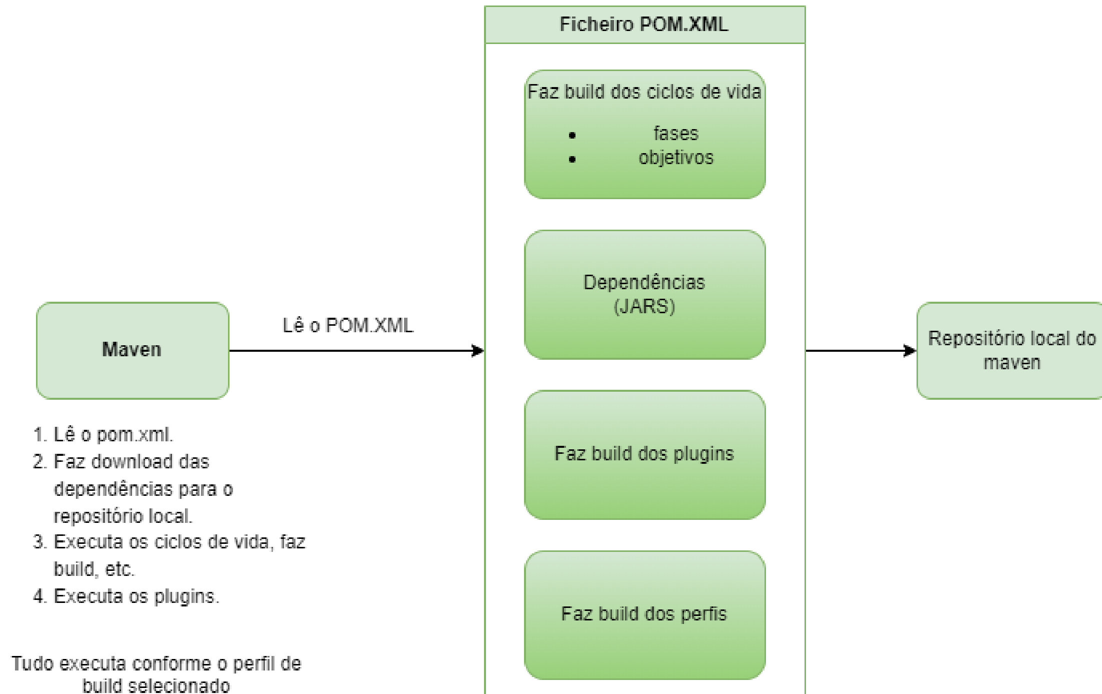

Figura 3.14: Funcionamento geral do *maven*.

- Compilar e executar os códigos de testes;
- Empacotar projetos;
- Facilitar a escrita de ficheiros de *build* usando conceitos de convenção sobre configuração;
- Realizar a gestão de dependências de projeto;
- Auxiliar todo o ciclo de construção do projeto;
- Fazer *deploy* em diferentes servidores;
- Gerar documentação (wiki);
- Verificar a qualidade do código.

Em suma, é uma ferramenta que simplifica a construção e gestão de projetos Java.

3.3.15 Java

O Java é uma linguagem orientada a objetos e centralizada na rede, que pode ser utilizada como uma plataforma em si. É uma linguagem de programação rápida, segura e confiável para codificar tudo, desde aplicações móveis e software até aplicações de *big data* e tecnologias do servidor [33].

Esta foi a linguagem de programação escolhida pela equipa e pela empresa para desenvolver e inovar na área de microserviços. O suporte que esta possui, as bibliotecas existentes, a facilidade de integração com novas tecnologias de software e a possibilidade de ser utilizada em múltiplas plataformas fomentou a sua escolha.

3.3.16 JUnit

O JUnit é uma *framework open-source*, que possibilita a criação de testes unitários em Java. A maioria dos *IDEs*, inclusive o IntelliJ, incorporam o JUnit dentro do seu ambiente de desenvolvimento, facilitando assim o uso dessa framework. Este possibilita a criação de classes de testes, que contém um ou mais métodos, para que sejam realizados os testes unitários, podendo ser organizados de forma hierárquica, de forma que o sistema seja testado em partes separadas, algumas integradas ou até mesmo todas de uma só vez. Além disso, este *framework* tem como objetivo, facilitar a criação de casos de teste, além de permitir escrever testes que retenham o valor ao longo do tempo, ou seja, que possam ser reutilizáveis [34].

Para o utilizar é necessário incluir a biblioteca no projeto Java, neste projeto tem que ser adicionado no ficheiro *pom.xml*, como explicado anteriormente. Depois pode-se criar classes de teste que contenham métodos de teste, que devem conter afirmações (*asserts*), que verifiquem o comportamento esperado do código que está a ser testado.

Por fim, este foi utilizado, por requisição do cliente, visando a qualidade do código e a sua segurança, isto é, todo o código entregue ao cliente tem que estar devidamente testado e coberto com casos de uso e testes unitários. Sendo este um ponto definido no inicio do projeto como forma der garantir que tudo o que é entregue funciona da forma esperada e correta.

3.3.17 MongoDB

O MongoDB é uma base de dados *NoSQL* que foi desenvolvida para lidar com grandes quantidades de dados não estruturados ou semi-estruturados. Este armazena dados em documentos flexíveis, como por exemplo *JSON*, o que torna esta uma boa escolha para aplicações que requerem um armazenamento de dados rápido e escalável, ao contrário das base de dados relacionais tradicionais [35].

Este foi escolhido para o desenvolver do projeto pelas seguintes razões:

1. **Modelo de dados flexível** – permite armazenar dados de uma forma flexível, tornando-o fácil de se adaptar às necessidades atuais. Pode adicionar ou remover campos dos documentos sem ter de modificar o esquema da sua base de dados;
2. **Escalabilidade** – este escala horizontalmente, o que significa que pode se adicionar mais servidores ao cluster da base de dados à medida que a aplicação/sistema cresce. Isto torna-o ideal para o tratamento de grandes quantidades de dados e cargas de tráfego elevadas;

3. **Alta disponibilidade** – As características de replicação e *failover* automático do MongoDB garantem que os seus dados estejam sempre disponíveis, mesmo no caso de falhas de hardware ou da rede;

4. **Rica linguagem de consulta** – esta permite realizar consultas complexas sobre os dados armazenados, incluindo consultas por país, pesquisas de texto, e agregações.

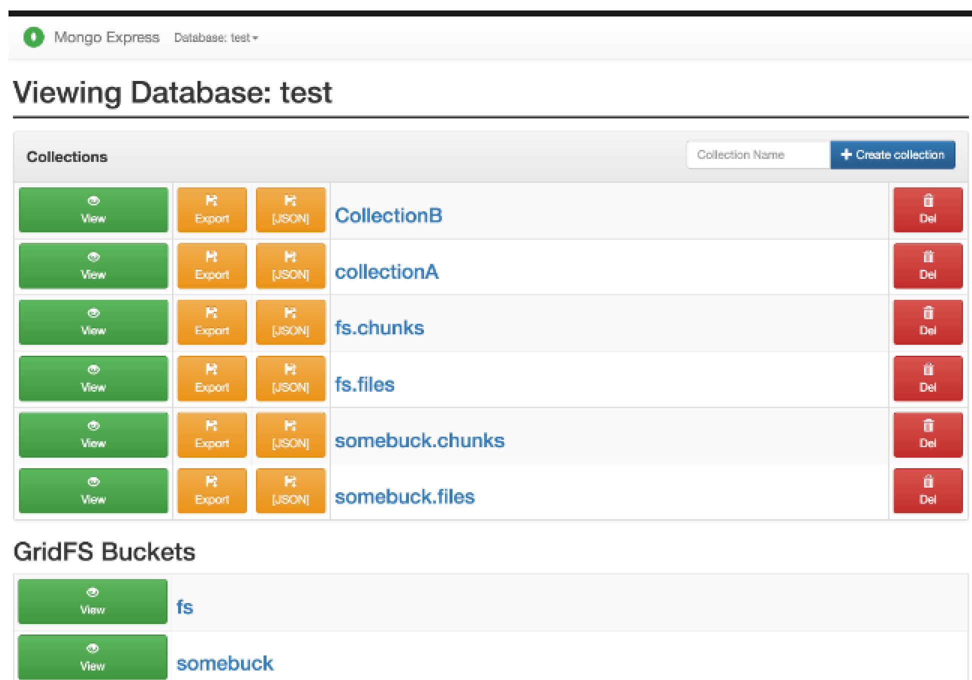

Figura 3.15: Interface do *Mongo Express* [16].

Por fim, este é muito utilizado para guardar dados não relacionais e dados que não sabemos a sua estrutura e é rápido na consulta mesmo com elevado tráfego de dados. A sua integração com o *Spring Boot* é simples e intuitiva, o que torna todo o processo de configuração quase inexistente, tendo até uma interface disponível (*Mongo Express*) através do *docker* (figura 3.15).

3.3.18 Redis

O Redis armazena em memória estrutura de dados, que é utilizado como base de dados, cache, e *broker* de mensagens. Este foi desenvolvido com o objetivo de lidar com um alto desempenho e escalabilidade, tornando-o uma ótima escolha para a construção de aplicações em tempo real, que requerem acesso e processamento rápido de dados. Suporta uma grande variedade de estruturas de dados, como strings, *hashes*, listas, conjuntos, conjuntos ordenados, e fornece um rico conjunto de comandos para manipulação e consulta dessas estruturas de dados [36].

Uma das principais vantagens do Redis é a sua rapidez, visto que armazena dados em memória, consegue fornecer velocidades de leitura e escrita extremamente rápidas, o que o torna ideal para casos de utilização que requerem acesso a dados de baixa latência, tais

como análise em tempo real, *cache*, e gestão de sessão.

Este pode também ser configurado de forma a guardar dados periodicamente em disco, assegurando que os dados não são perdidos no caso de uma falha do sistema. Este suporta replicação e *clustering*, o que permite que múltiplas instâncias sejam sincronizadas e distribuídas por múltiplos servidores, proporcionando alta disponibilidade e escalabilidade.

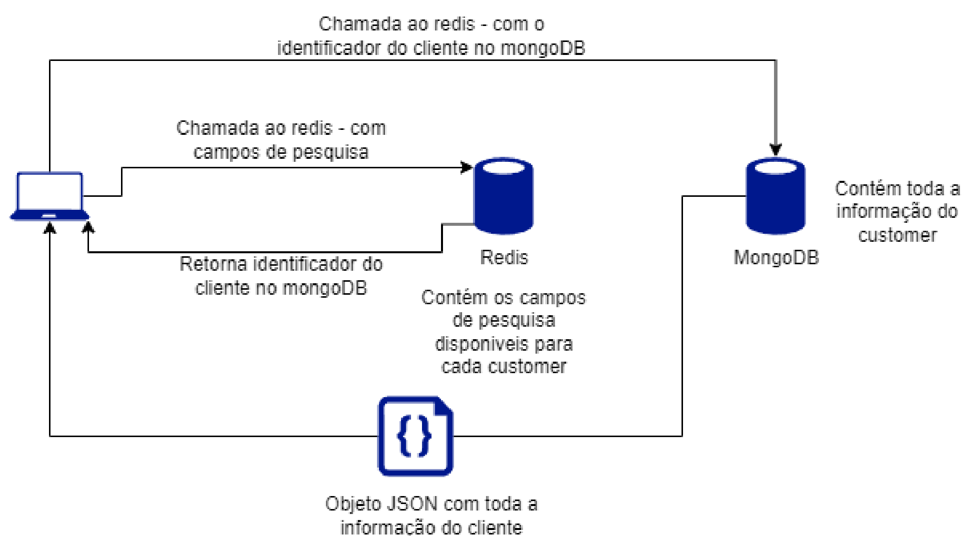

Figura 3.16: Caso de uso da implementação do redis com mongoDB.

No desenvolvimento do projeto, este foi utilizado para armazenar *cache* dos dados dos *customers*, onde são guardados campos de pesquisa, como por exemplo o *customerId*, *customerAccountId*, identificador do cliente no mongoDB ,entre outros. Estes campos de pesquisa são campos de um objeto *JSON* que contém toda a informação sobre o cliente em questão, guardado no mongoDB. O redis neste caso de uso, serve como *cache* sobre um cliente, onde se guardam campos de pesquisa sobre o mesmo e integra através do identificador do cliente com o mongoDB, que contém o resto da informação do cliente. Este uso explica-se através da vantagem de ter os campos de pesquisa de cada cliente em *cache*, porque se verifica que a pesquisa por campos é mais rápida do que consultar diretamente o próprio mongoDB. Este caso de uso encontra-se demonstrado na figura 3.16.

Concluindo, o Redis é uma ferramenta vantajosa e flexível em termos de armazenamento de dados, que pode ser utilizado para uma variedade de casos de utilização, desde o *cache* de alta velocidade até à análise e envio de mensagens em tempo real. A sua velocidade, escalabilidade, e conjunto de características fazem no uma boa escolha quando precisam de construir aplicações rápidas, fiáveis, e escaláveis.

3.3.19 SonarQube

O *SonarQube* é uma plataforma utilizada para fazer uma inspeção contínua à qualidade

do código. O objetivo desta é ajudar no desenvolvimento de software, identificar e corrigir problemas de qualidade no código, reduzindo o custo de manutenção e melhorando a fiabilidade e a segurança das aplicações que esta analisa [37].

A plataforma utiliza uma variedade de técnicas de análise de código estático, incluindo a complexidade do código, a duplicação de código e a adesão às melhores práticas e normas de programação. Também inclui funcionalidades para testar e analisar vulnerabilidades de código, falhas de segurança e conformidade com regulamentos, como por exemplo, o regulamento geral de proteção de dados [37].

O *SonarQube* fornece um painel de controlo abrangente que apresenta uma visão geral da qualidade do código, incluindo métricas como a cobertura do código, o número de erros, as vulnerabilidades, os problemas ou possíveis problemas de código e o nível de cobertura do código dado pelos testes unitários do microserviço em questão. A plataforma apresenta-se na figura 3.17.

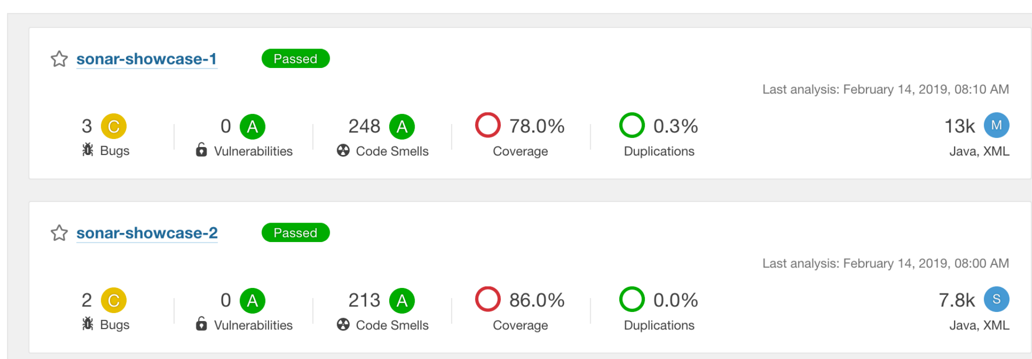

Figura 3.17: Interface do *SonarQube* [17].

Uma das principais vantagens desta tecnologia é permitir às equipas de desenvolvimento monitorizar e melhorar continuamente a qualidade do código, facilitando a deteção de problemas durante o processo de desenvolvimento, o que pode poupar tempo e recursos significativos à empresa.

Neste projeto, todas as entregas tinham que passar pela a avaliação do *SonarQube*, para que desta forma, a qualidade de entrega ser garantida ao cliente. Incluindo a percentagem de cobertura dos testes unitários ser superior ou igual a 80 %.

Em resumo, esta é uma ferramenta muito utilizada pelo mercado de trabalho para melhorar a qualidade do código, reduzir os custos de manutenção e aumentar a fiabilidade e a segurança do software.

3.3.20 Spring Boot

O Spring Boot é uma *framework* para criar e implementar código baseado na linguagem

Java. Esta foi escolhida pela empresa cliente, visto já ser a *framework* de desenvolvimento da sua equipa e por possuir uma maior agilidade no processo de desenvolvimento, uma vez que consegue reduzir o tempo gasto com as configurações iniciais. Oferece também um recurso de injeção de dependências, que permite que os objetos definam as próprias dependências onde os *Spring containers* injetam-nos posteriormente. Isso permite criar aplicações modulares, que consistem em componentes fracamente acoplados que são ideais para microserviços e aplicações distribuídas [38].

Com o *Spring Boot* pode-se abstrair e facilitar a configuração de, por exemplo:

- servidores;
- gestão de dependências;
- configurações de bibliotecas;
- integração com outras tecnologias;
- criar aplicações *standalone*;
- ligação a base de dados, entre outros.

Na figura 3.18, encontra-se um exemplo genérico de um *flow* criado no *spring boot*. Onde se pode verificar que se faz requisições *HTTP* a um controlador exposto pelo *spring boot*, depois este tem lógica associada no *service layer*. Este por sua vez tem as dependências *CRUD* para comunicar com a base de dados, onde estas dependências fazem *queries* à mesma. O *Spring data/JPA* permite criar uma estrutura de dados relacionada diretamente à base de dados para termos os dados mapeados dentro da execução da aplicação. Por fim, tudo funciona de forma abstrata ao utilizador, visto que não é necessário nenhuma configuração complexa nem extensa. Sendo esta uma das principais razões da sua escolha.

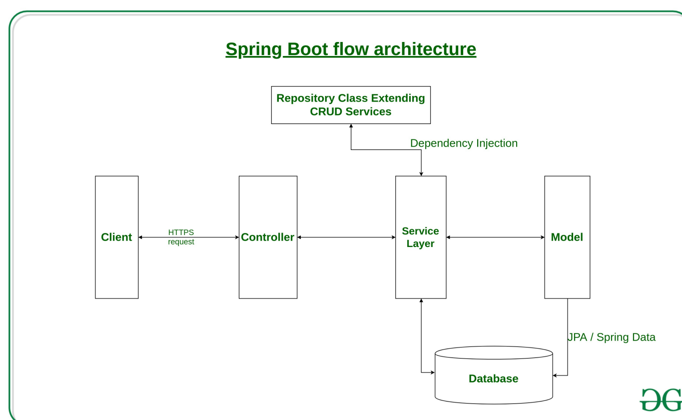

Figura 3.18: Arquitetura do *flow* do *Spring Boot* [18].

3.4 Conclusão

Em resumo, neste capítulo descrevem-se as metodologias de trabalho e tecnologias e ferramentas utilizadas na realização e desenvolvimento deste projeto. A integração num projeto implica sempre um determinado período de adaptação aos membros que nele constam, bem como às ferramentas utilizadas, que normalmente ocorre na entrada ao projeto, onde se tem um período de tempo para aprender as ferramentas que são utilizadas no desenvolver do mesmo. Neste período foram desenvolvidas várias *Proof Of Concept* (POC) para a equipa interna, de forma a provar que o estagiário estava apto para começar a pegar em *tasks*. O *Confluence* e o *Jira* requerem uma maior atenção de utilização visto que interage diretamente com a organização e estado do projeto.

Capítulo 4

Análise Comparativa de Arquiteturas de Desenvolvimento de Aplicações

4.1 Introdução

Neste capítulo, apresentam-se as diferentes arquiteturas de desenvolvimento de software, que têm surgido ao longo do tempo, com diferentes paradigmas e abordagens para dar resposta às necessidades da evolução das aplicações modernas. Analisa e compara microserviços, arquitetura orientada a serviços (SOA) e arquiteturas monolíticas, com o objetivo de proporcionar uma compreensão abrangente das suas principais características, vantagens e desvantagens e considerações sobre as mesmas.

4.2 Arquitetura Monolítica

Uma arquitetura monolítica é um modelo único e tradicional para o desenvolvimento de programas de *software*. Monolítico, neste contexto, significa ”composto numa só peça”. Este é projetado para ser independente, onde os seu componentes ou funções são fortemente acoplados (grande dependência entre si), em vez de fracamente acoplados, como nos programas de *software* modulares. Numa arquitetura monolítica, cada componente e os respetivos componentes associados devem estar presentes, para que o código seja executado ou compilado.

Estas aplicações são de camada única, o que significa que vários componentes sejam combinados numa aplicação final (dependendo do tamanho e objetivo da mesma). Consequentemente, tendem a ter grandes bases de código, que podem ser difíceis de gerir ao longo do tempo. A figura 4.1 apresenta um exemplo de uma arquitetura monolítica, onde os componentes estão todos dependentes entre si.

Além disso, se um componente do programa precisar ser atualizado ou até removido, outros elementos também podem exigir uma alteração, esta pode ter um grau de complexidade enorme, dependendo da dependência entre os elementos, e toda a aplicação deve ser recompilada e testada, visto que, uma simples alteração pode mudar o comportamento da mesma. O processo pode ser demorado e até limitar a agilidade e a velocidade das equipas de desenvolvimento de software. Apesar destes problemas, a abordagem ainda está em uso porque oferece algumas vantagens, que serão vistas mais à frente. Além disso, até ao momento há muitas aplicações que foram desenvolvidos como software monolítico, portanto, a abordagem não pode ser totalmente desconsiderada enquanto estes ainda estiverem em uso e exigirem atualizações.

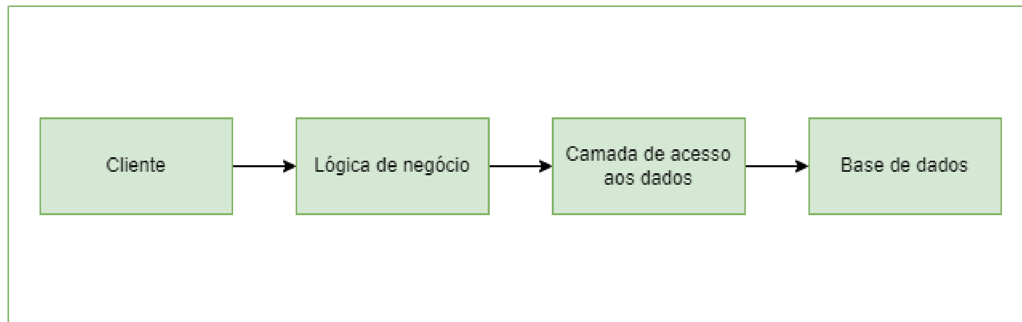

Figura 4.1: Exemplo de uma Arquitetura Monolítica.

Desta forma, esta apresenta várias vantagens, principalmente:

- Facilidade de desenvolvimento e implementação. A abordagem monolítica é uma forma padrão de criar aplicações, onde nenhum conhecimento adicional é necessário, visto que, todo o código-fonte está localizado num local, que pode ser rapidamente compreendido.
- Facilidade de depuração. O processo de depuração é simples e intuitivo, porque todo o código está localizado num só lugar, pode se seguir logicamente o fluxo de uma solicitação (seguindo o código).
- Simplicidade da fase de testes. Testa-se apenas um serviço devido à enorme dependência entre eles.
- Facilidade de uso. É gerado apenas uma unidade de implantação (por exemplo, arquivo jar). Não há outras dependências. Nos casos em que a interface do utilizador é gerida com código do *backend*, não se tem nenhuma alteração significativa, tudo existe e muda-se num só lugar.
- Facilidade da evolução da aplicação . Basicamente, este não possui nenhuma limitação do ponto de vista da lógica de negócios. Se for preciso alguns dados para um novo recurso, eles já estão lá.
- Preocupações transversais e personalizações são usadas apenas uma vez. As preocupações transversais são resolvidas apenas uma vez. Por exemplo, a segurança, registo, tratamento de exceções, monitorização, entre outros.
- Facilidade na integração de novos membros na equipa de desenvolvimento. O código-fonte está localizado num só lugar e os novos membros podem facilmente depurar algum fluxo funcional e se familiarizar com a aplicação.
- Baixo custo nas fases iniciais da aplicação . Todo o código-fonte está localizado num só lugar, empacotado numa única unidade de implantação. O que leva a não haver sobrecarga no custo de infraestruturas, nem no custo de desenvolvimento.

Devido a estas vantagens, a arquitetura monolítica geralmente é utilizada nos estágios iniciais do desenvolvimento de aplicações, porque a principal função da aplicação é ser rentável. Como resultado, é importante implementar rapidamente algumas soluções *Proof of Concept* (POC), para verificar o comportamento da aplicação no mundo real. Além disso, é importante verificar os comportamentos dos clientes no sistema e caso sejam necessárias melhorias podem ser implementadas facilmente no inicio. Outra razão prende-se ao facto dos requisitos geralmente não serem claros nos estágios iniciais de desenvolvimento.

Os problemas com a esta arquitetura começam a surgir quando o a aplicação necessita de maior complexidade. A razão para isto deve-se ao crescimento da mesma. Normalmente, após algum período de tempo, esta muda para outro tipo de arquitetura devido aos seguintes motivos:

- Velocidade lenta no processo desenvolvimento e subida de pipelines. A desvantagem mais simples está diretamente relacionada ao pipeline CI/CD. Quando a aplicação contém muitos serviços, algum nível de complexidade, e cada serviço neste tem que ser coberto com testes, que são executados para cada *Pull/Merge Request*, e é sujeita a uma pequena alteração no código-fonte, tem que se esperar muito tempo (por exemplo, 1 hora), para que o pipeline seja bem-sucedido. E o que acontece quando o pipeline falha por algum motivo? Tem que se esperar novamente outra hora. Todos os serviços estão localizados num único lugar, como se lida com conflitos no código entre os elementos da equipa? Perdendo muito tempo a comunicar este tipo de problemas em reuniões entre os elementos da equipa.

- Acoplamento de alto código. De certa forma consegue-se manter uma estrutura de serviço clara dentro do projeto, mas como mostra a prática, eventualmente, vai se acabar com um código espalhado pelo projeto e com muita dependência entre funcionalidades. Como resultado, o sistema torna se mais confuso e difícil de entender, especialmente para novos membros da equipa de desenvolvimento.

- Pouca independência no desenvolvimento. Normalmente o desenvolvimento de funcionalidades é separado entre os vários elementos da equipa, só que nesta arquitetura não existe fronteira entre essas funcionalidades, devido ao forte acoplamento, o que torna com que o trabalho de um afete o do outro.

- Alterações ou atualizações podem ser complexas. Mesmo uma pequena alteração pode afetar negativamente grande parte sistema da aplicação. Como resultado, a regressão para serviço monolítico completo é necessária.

- Limitação nas tecnologias legadas. Difícil migrar de tecnologias, visto ser necessário alterar grande parte do sistema, tornando essa ação custosa para a empresa.

- Falta de flexibilidade. As tecnologias escolhias ao inicio do desenvolvimento, são normalmente as que são utilizadas até ao final do projeto. Uma alteração de tecnologia pode trazer enormes prejuízos e perdas de tempo.

- Problemas com implantação. Uma pequena mudança, requer a reimplantação de todo o projeto.

4.3 Arquitetura Orientada a Serviços

A Arquitetura Orientada a Serviços (SOA) é considerada a evolução do desenvolvimento de aplicações utilizando a arquitetura monolítica, e resolve grande parte dos problemas encontrados na secção anterior. Este define uma forma de tornar os componentes de software reutilizáveis e independentes.

Esta é uma abordagem onde as aplicações fazem uso de serviços disponíveis na rede. Nesta, são fornecidos serviços para formar aplicações, consumindo-os através de padrões de comunicação comuns, que aceleram e agilizam as integrações de serviços nas mesmas. Cada serviço possui uma interface e é uma função de negócios completa em si, onde estes são publicados, de forma, que seja fácil para a equipa de desenvolvimento montar as aplicações, ou seja, após a definição de requisitos das lógicas de negócio, estes são separados e desenvolvidos de formas separadas e independentes uns dos outros, e posteriormente utilizados para o desenvolvimento da aplicação.

Estes serviços também podem comunicar entre si em diferentes plataformas e linguagens, pois podem ser desenvolvidos com linguagens e tecnologias diferentes, visto que, são independentes uns dos outros. Podem também reutilizar serviços em sistemas diferentes ou combinar vários serviços independentes para realizar tarefas mais complexas.

Pode-se ver as principais diferenças entre a arquitetura monolítica e a SOA na figura 4.2.

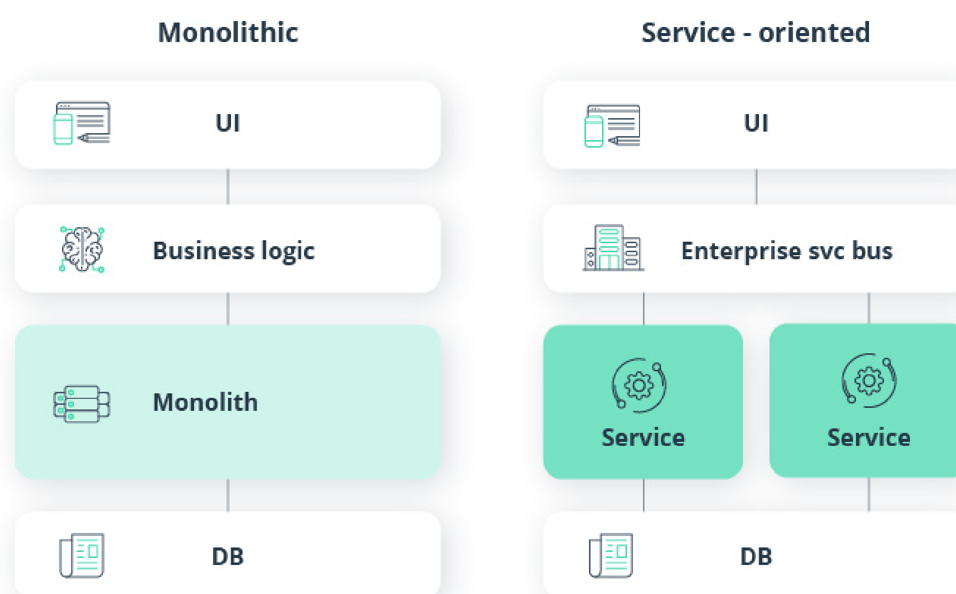

Figura 4.2: Diferenças entre a Arquitetura Orientada a Serviços e Monolítica [19].

Esta utiliza uma arquitetura de software, chamado *Enterprise Service Bus*, que conecta todos os serviços numa infraestrutura semelhante a um barramento. Este atua como centro de comunicação no SOA, permitindo vincular vários sistemas, aplicações, dados e conectar vários sistemas sem interrupção. Desta forma, surgem três papéis fundamentais em cada um dos blocos de construção desta arquitetura, como se pode ver na figura 4.3:

1. provedor de serviço, este cria serviços Web e fornece-os a um registo de serviços. O provedor de serviços é responsável pelos termos de uso do serviço, segurança, disponibilização, entre outros;

2. agente de serviço ou registo de serviço, é responsável por fornecer informações sobre o serviço a um solicitante. Um corretor pode ser público ou privado;

3. solicitante de serviço ou consumidor de serviço, encontra um serviço num intermediário de serviços ou registo de serviço e, em seguida, conecta-se ao provedor de serviço para receber o mesmo.

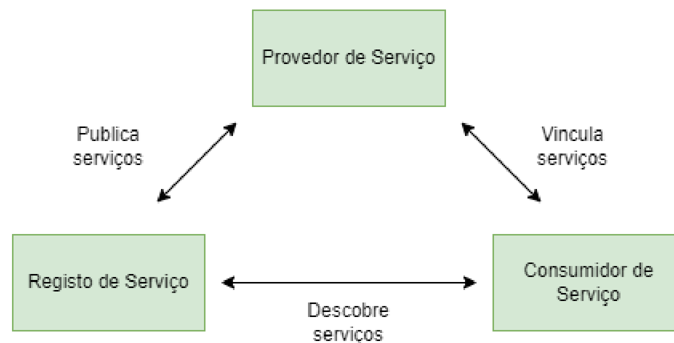

Figura 4.3: Papeis Fundamentais da Arquitetura Orientada a Serviços.

Posto isto, apresenta inúmeras vantagens em relação à arquitetura monolítica, dentro do quais são:

1. Reutilização e independência dos serviços, estes são criados pela junção de componentes funcionais pequenos, independentes e fracamente vinculados. Consequentemente, estes podem ser reutilizados para outras aplicações, poupando tempo e recursos.

2. Simples manutenção, pode ser facilmente e rapidamente atualizada ou mantida sem se preocupar com outros serviços, porque estes são independentes um dos outros. Assim, quando um serviço está em manutenção, não afeta o resto dos outros serviços, minimizando as perdas.

3. Maior fiabilidade, porque os serviços pequenos e independentes são mais simples de testar e depurar do que grandes blocos de código, sendo mais fácil de encontrar erros e corrigi-los.

4. Melhorias na escalabilidade e disponibilidade da aplicação, este pode funcionar com várias instâncias simultaneamente. Isso torna o serviço mais escalável e prontamente disponível. Desta forma, pode se escalar o serviço que estiver a ter uma maior demanda de acessos, em vez de escalar a aplicação inteira.

5. Aumento da produtividade, podem adicionar novas funcionalidades e reutilizar serviços de aplicações já previamente desenvolvidas sem começar do zero.

6. Diversidade tecnológica, cada serviço tem a sua linguagem de programação, base de dados, entre outros. Como são desenvolvidos de forma separada, não há limitação do uso tecnológico.

Em suma, a arquitetura orientada a serviços veio deste modo resolver grandes problemas colocados pela a arquitetura monolítica e revolucionar a forma como se desenvolve aplicações, mas com esta surgem três grandes problemas:

1. Servidor de alta largura de banda, visto que, o serviço de rede envia e recebe mensagens e informações com uma enorme demanda, ou seja, atinge altas solicitações por dia, o que envolve um servidor de alta velocidade, aumentando os custos com infraestruturas;

2. Enorme sobrecarga na rede, sempre que um serviço interage com outro serviço, ocorre uma validação completa de cada parâmetro de entrada. Isso aumenta o tempo de resposta e a carga da máquina e, portanto, reduz o desempenho geral;

3. Alto custo de investimento, este tipo de implementações requerem um grande investimento inicial por meio de tecnologia, desenvolvimento e recursos humanos.

4.4 Microserviços

Os microserviços são uma abordagem inspirada no SOA para o desenvolvimento de software. Nesta, uma aplicação é dividida em pequenos serviços independentes, cada um deles responsável por uma única função específica. Estes são desenvolvidos, implantados e escalados de forma independente. Uma das principais diferenças para o SOA, é a definição de serviço, enquanto neste um serviço é responsável por uma funcionalidade, por exemplo, login ou registo, nos microserviços um serviço é responsável apenas por uma função, por exemplo, para o mesmo caso anterior de login, teríamos um microserviço para gerir as contas cliente e outro para validar as credenciais colocadas pelo utilizador. A figura 4.4 apresenta a diferença entre os microserviços com as duas arquiteturas anteriores.

Enquanto no SOA, utiliza-se o *Enterprise Service Bus* para a comunicação de serviços, nos microserviços é necessário estabelecer uma comunicação eficiente entre estes, geralmente utilizando API's (Interfaces de Programação de Aplicativos) para definir como os serviços interagem entre si.

Desta forma, os microserviços apresentam várias vantagens, das quais:

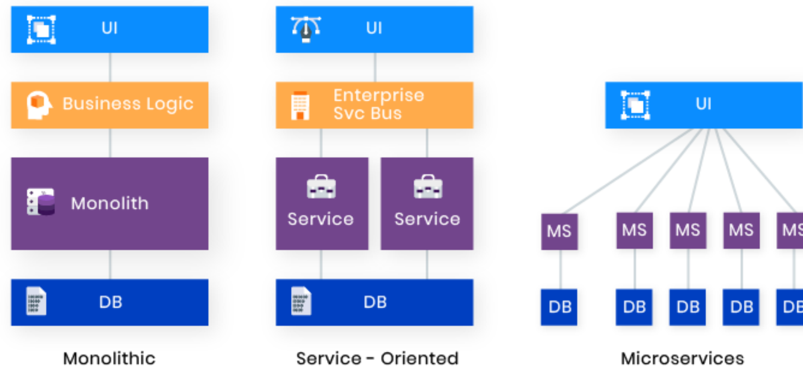

Figura 4.4: Diferenças entre as Arquiteturas SOA, Microserviços e Monolítica [19].

1. Flexibilidade e escalabilidade, como cada microserviço é independente dos outros, pode ser desenvolvido e implantado separadamente. Se existir uma maior quantidade de solicitações para um determinado serviço, pode se replicar a sua instância, para responder ao aumento da demanda;

2. Facilidade de manutenção, como o microserviço executa uma tarefa específica, apenas é necessário concentrar-se com essa função, porque não afeta o resto da aplicação;

3. Ciclo de desenvolvimento mais rápido, nestes equipas menores trabalham em conjunto para desenvolver cada microserviço, como este executa uma tarefa específica, os membros de cada equipa podem se concentrar apenas nessa tarefa. Tornando todo o desenvolvimento menos confuso e complexo;

4. Escolha de tecnologia independentes, cada microserviço pode ter a sua tecnologia, linguagem de programação e desenvolvimento. Pode-se responder aos problemas encontrados com a variedade de tecnologia que existe na atualidade;

5. Mais resiliente, a falha de um microserviço apenas afeta a respetiva funcionalidade em vez de toda a aplicação;

Embora os microserviços ofereçam muitas vantagens sobre as duas arquiteturas anteriores, é importante considerar também as desvantagens desta abordagem. Em seguida, estão algumas das principais desvantagens dos mesmos:

1. Estruturação complexa, visto ser um conceito distribuído, a estruturação do projeto é mais complexa que na monolítica, demandando integrações mesmo para funcionalidades simples;

2. Sobrecarga de comunicação, há uma maior dependência na comunicação entre os serviços. Cada solicitação pode exigir chamadas a vários serviços diferentes, o que pode resultar numa sobrecarga de comunicação;

3. Requisitos de infraestrutura, geralmente requerem uma infraestrutura de suporte mais avançada em comparação com as aplicações monolíticas. É necessário ter mecanismos

eficientes de escalabilidade, balanceamento de carga, monitorização e gestão de *containers*. Isso pode resultar em custos adicionais e maior complexidade operacional;

4. Complexidade de planeamento, este requer uma abordagem diferente em comparação com o planeamento monolítico e um planeamento mais cuidado que os SOA, por serem serviços mais pequenos. É necessário definir cuidadosamente os serviços, estabelecer uma comunicação eficiente entre eles e garantir que cada serviço funcione corretamente em conjunto. Essa complexidade pode aumentar o tempo e os esforços necessários para planear o projeto;

5. Consistência de dados, cada serviço geralmente possui a sua própria base de dados e isso pode levar a problemas de consistência de dados, já que cada serviço pode ter o seu próprio modelo de dados. É necessário implementar estratégias de sincronização e padronização de dados para garantir a consistência em todo a aplicação.

4.5 Conclusão

Em conclusão, as arquiteturas de microserviços, monolítica e SOA (Service-Oriented Architecture) apresentam formas diferentes de estruturar e desenvolver *software*, cada uma delas possui as suas vantagens e desvantagens e a escolhas destas para desenvolvimento depende sempre dos requisitos de cada projeto.

A arquitetura monolítica é uma abordagem onde toda a aplicação é desenvolvida num só bloco de código e é mais simples de gerir e implementar, apresenta desafios na escalabilidade e flexibilidade, para além de dificultar o uso de práticas e metodologias ágeis. Enquanto os microserviços dividem a aplicação em serviços independentes, onde cada um é responsável por uma função especifica, o que permite uma maior agilidade, escalabilidade e flexibilidade para as equipas de desenvolvimento, permitindo estas trabalharem de forma independente em cada serviço, mas a complexidade de gestão, planeamento e a necessidade de comunicação eficiente entre os serviços podem representar desafios adicionais. Por fim, o SOA é uma abordagem em que os serviços são desenvolvidos como componentes independentes, mas com maior ênfase na sua interoperabilidade e na partilha de serviços entre diferentes sistemas. Esta promove a reutilização de serviços e a integração entre sistemas heterogéneos, mas a implementação do SOA pode ser complexa e requerer um planeamento cuidadoso para garantir o comportamento adequado dos serviços.

Cada abordagem tem suas aplicações e utilização especifica e não há um melhor que outro, mas sim casos de usos em que um é melhor que os outros. A arquitetura monolítica pode ser mais adequada para projetos menores e com requisitos simples, enquanto os microserviços e o SOA são mais indicados para aplicações complexas e que requerem escalabilidade e flexibilidade. A escolha entre essas abordagens deve ter em consideração as necessidades do projeto, os recursos disponíveis e os objetivos a longo prazo.

Em resumo, não há uma abordagem única que seja a melhor para todos os cenários. É importante analisar cuidadosamente os prós e contras de cada uma e selecionar a que melhor se adequa às necessidades específicas de um projeto ou sistema.

Capítulo 5

Planeamento do Estágio

5.1 Introdução

Neste capítulo, apresenta o planeamento delineado pela empresa ao estagiário. Este planeamento é feito de forma individual, visto que, depende muito do desenvolvimento pessoal e técnico para o começo da integração dos projetos que a empresa possui. Por fim, apresenta o que foi feito em cada parte do planeamento de forma mais detalhada e com os períodos de realização.

5.2 Plano do Estágio

O plano de estágio, como dito anteriormente, depende muito do desenvolvimento/desempenho do estagiário durante todo o seu percurso. Este encontra-se na figura 5.1, que é um mapa cronológico, e demonstra a separação das estapas do percurso do estagiário pela duração do estágio.

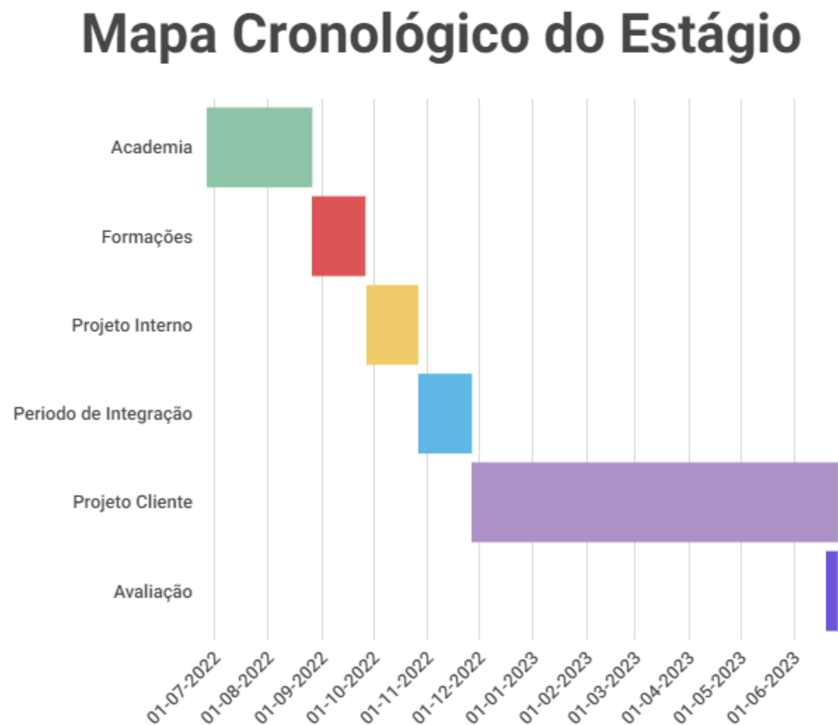

Figura 5.1: Mapa Cronológico do Planeamento do Estágio.

5.2.1 Academia Especializada

Nesta fase, o estagiário teve a oportunidade de conhecer mais sobre a empresa de acolhimento e as áreas de trabalho que esta possui, onde posteriormente escolheu a área que pretendia trabalhar e desenvolver os seus conhecimentos (microserviços).

Após a definição da área de trabalho do estágio, ocorreu o período de desenvolvimento técnico e teórico, sobretudo sobre *fullstack* (desenvolvimento simultâneo de *backend* e *frontend*), visto que a vaga preenchida pelo estagiário inicialmente seria essa. Este período foi orientado pela pessoa responsável da academia, onde ensinou os seguintes pontos:

1. CSS;
2. HTML:
3. Java;
4. SQL;
5. Base de dados;
6. *Spring boot*;
7. Conceitos teóricos e aplicações práticas sobre microserviços;
8. Variadas metodologias de trabalho que existem no mercado de trabalho, com um maior ênfase na metodologia *agile/scrum*.

Ao fim deste período de aprendizagem e desenvolvimento, foi feito uma avaliação de conhecimentos teóricos e práticos, para desta forma perceber o estado do estagiário. Seguido de um período de certificações e formações, apresentado com melhor detalhe na subsecção 5.2.2.

5.2.2 Formações e Certificações

Este período foi essencialmente incidido na certificação e formação de conhecimentos do estagiário, de forma a este provar que realmente aprendeu e desenvolveu os seus conhecimentos durante o período de academia e durante este período de formações.

Neste também foram realizados vários cursos com certificação sobre:

1. Microserviços;
2. REST API;
3. TMFORUM;
4. Java;
5. Spring boot.

As certificações e formações sobre *TMFORUM*, ao todo foram 10 incluindo formações teóricas e práticas, e incidiram mais no sentido de explicar ao estagiário o conhecimento de negócio sobre telecomunicações, e explicar as metodologias de trabalho que uma empresa, que trabalhe na área de comunicações, pode seguir para ter faturação e menos despesas.

Concluindo, este período é um dos mais importantes para a empresa durante o estágio, porque desta forma verifica-se o conhecimento do estagiário e o quanto este pretende evoluir na sua carreira. Estas certificações são extremamente importantes para a colocação da empresa no mercado de trabalho, derivado à mão de obra qualificada e certificada gerar mais entradas de projetos.

5.2.3 Projeto Interno

O projeto interno foi uma fase onde o estagiário teve oportunidade de mostrar os conhecimentos ganhos na academia e pelas formações/certificações. Este projeto é sempre um desenvolvimento de um produto interno da empresa, por exemplo, aplicação de gestão da evolução da carreira dos trabalhadores.

Nesta fase, o estagiário aprendeu e aplicou os seus conhecimentos sobre a metodologia de trabalho *agile/scrum*, funcionamento do *gitlab/git* e todos os *standards* de trabalho da empresa de acolhimento. Durante esta, o estagiário foi acompanhado por um mentor, onde este acompanhou todo o desenvolvimento das tarefas do estagiário.

Após um mês no projeto interno, o mentor colocou o estagiário num projeto cliente para uma empresa de telecomunicações.

5.2.4 Período de Integração

O período de integração, já dentro de um projeto cliente, incide na aprendizagem do estado atual do projeto, os objetivos deste, passam por ler a documentação do mesmo, aprender as tecnologias utilizadas, realizar pequenas *POCs* para a equipa (parecidas a algumas possíveis tarefas no decorrer do projeto) e a parte mais difícil, que é aprender o que já foi desenvolvido, como foi desenvolvido e o porquê de ser aquela a solução.

Neste mês, o *team leader* marcou reuniões diárias (que não eram as *dailys*) para que cada integrante do projeto explicasse aos novos os estagiários do projeto como desenvolvem e procedem uma entrega de tarefas, que fizeram desde o dia anterior.

Concluindo, esta é uma das fases mais importantes, porque é onde o estagiário aprende as metodologias e rotinas de desenvolvimento da equipa e de cada integrante da mesma. Aprende como os *standards* das tecnologias utilizadas, como se procede para resolver as tarefas que lhe são associadas e bem como mexer nas ferramentas *Jira* e *Confluence*.

5.2.5 Projeto Cliente

No projeto cliente o estagiário assumiu tarefas de desenvolvimento de novos funcionamentos, correção de *bugs*, testes integrados, testes unitários, escrever documentação, participação dos *sprints reviews*, prevenção e manutenção e por fim desenvolvimento de novas tarefas a realizar em cada *sprint*. Esta fase é aprofundada no capítulo 6, onde se explicita os vários desenvolvimentos e etapas do projeto e a responsabilidade do estagiário em cada um delas.

5.2.6 Avaliação

A avaliação do estágio é a fase final de um estágio de 12 meses, realizado na empresa de acolhimento, onde os *mentores* do estagiário, o *team leader* e o diretor da *BU* avaliam o desempenho global do estagiário, mostrando os pontos fortes e fracos do percurso do estagiário ao longo do seu estágio.

É nesta fase, que a empresa de acolhimento mostra ao estagiário o que melhorar e os futuros caminhos a percorrer pelo estagiário na sua carreira na empresa. Esta carreira é decidida também neste momento, onde a empresa apresenta uma proposta ao mesmo, ou então decide não o fazer dependendo do percurso e desenvolvimento do mesmo.

5.3 Conclusão

A concluir, é de notar que o plano de trabalho foi desenvolvido pelos recursos humanos e mentores do estagiário de acordo com a evolução técnica e teórica do mesmo, tendo sempre em conta o seu desempenho, a metodologia em uso na empresa e os seus objetivos. Sendo assim serviu como referência para o desenrolar do estágio e como método de avaliação. É ainda importante notar que este planeamento bem delineado é importante para o desenvolvimento e apoio ao mesmo para o percurso e sucesso do estagiário.

Capítulo 6

Implementação

6.1 Introdução

Neste capítulo, é apresentado a implementação dos domínio de *billing*, *customer* e a criação de ordens definidas neste projeto, mostra as arquiteturas e as tarefas que o estagiário teve em cada uma delas e explica onde se enquadram os microserviços nas arquiteturas.

Por fim, este apresenta o trabalho desempenhado durante todo o estágio, ou seja, o trabalho desempenhado neste projeto, e é importante notar que o desenvolvimento não é descrito na totalidade, pelas razões apresentadas na secção 1.5.

6.2 Plano de Trabalho

O desenvolvimento deste projeto focou-se principalmente, como dito anteriormente, na criação e definição do domínio de *billing* e *customer*, e criação de ordem. Este último originou o desenvolvimento de um *shopping cart* e de um catálogo. Todos estes desenvolvimentos serão explicados em seguida, com um maior detalhe e ênfase com recurso a exemplos de arquiteturas.

Todos os desenvolvimentos de arquitetura, lógica de negócio, *endpoints*, entre outros, foram realizados pelo cliente. Apenas o desenvolvimento e manutenção dos microserviços é que foi realizada pela nossa equipa, não tivemos interferência no desenvolvimento do negócio do cliente, apenas na sua implementação. Por isso, as definições do domínio e estrutura dos dados foram responsabilidade do cliente, onde definiu este através do uso do *swagger* a sua lógica e posteriormente utilizado para gerar o modelo de dados e os controladores do microserviço.

Por outro lado, a criação de ordem implicou o desenvolvimento do *shopping cart*, um orquestrador da ordem e o catálogo, definidos pelos arquitetos funcionais do cliente, onde neste último o modelo de dados também ficou ao critério do cliente.

Desta forma, o trabalho foi dividido por *sprints* de 2 em duas semanas onde se entregam as funcionalidades prioritárias pedidas pelo cliente, até os domínios e as restantes entregas estarem finalizadas. Na figura 6.1, encontra-se um mapa cronológico, que representa a divisão do tempo de desenvolvimento pelos objetivos definidos pelo cliente com a sua respetiva *timeline*.

Figura 6.1: Mapa Cronológico do Desenvolvimento do Projeto.

6.2.1 Domínio do *Customer*

Nesta parte do projeto, o objetivo passou pela criação de microserviços responsáveis por tratar os dados dos *customers* do nosso cliente, como por exemplo, guardar os seus dados de conta (*login* e *registo*), informações sobre compras anteriores, entre outros. Foi desenvolvido para alimentar o modelo de dados do mesmo, com funções sobre: criar, atualizar, remover e devolver informações sobre os mesmos.

A figura 6.2 representa a arquitetura desenvolvida para suportar estas funcionalidades, mencionadas anteriormente e estão representados 3 microserviços responsáveis por todo o domínio de *customer*: *customer domain*, *database adapter* e a base de dados..

O microserviço do *customer domain* não comunica de forma direta com a base de dados, para a existência deste ser agnóstica à existência desta, e por isso foi criado o segundo microserviço para interagir diretamente com a base de dados e realizar todos os mapeamentos ou lógicas para retornar os dados pedidos pelo *customer domain*. O melhor exemplo para o uso do *database adapter* é no sentido de se o *customer domain* estiver sobre uma enorme demanda, este vai escalar e criar uma nova instância do mesmo, para desta forma, conseguir lidar com a fluência dos dados. Ao escalar-mos este microserviço estaria dependente também uma instanciação de uma base de dados que é ligada diretamente a este, o que torna inútil a utilização da arquitetura de microserviços. Com esta implementação, pode se escalar o microserviço que for preciso sem nunca ter uma dependência de nova instância de uma tecnologia ou base de dados.

Neste microserviço, o estagiário foi responsável por criar o modelo de dados, bem como implementar as API´s e lógica de negócio. Também implementou testes unitários, testes

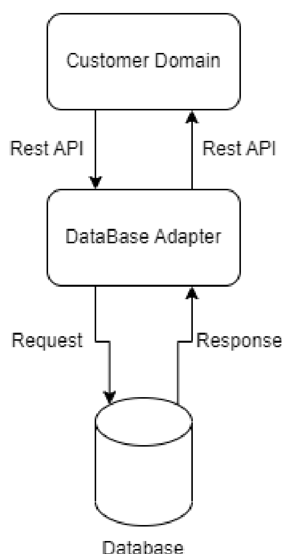

Figura 6.2: Arquitetura do Domínio de *Customer*.

integrados e resolução de *bugs* e defeitos.

6.2.2 Domínio de *Billing*

Nesta parte do projeto, o intuito passou pela criação de microserviços responsáveis por tratar dos dados de faturação e cobrança do nosso cliente, como por exemplo, cobrar um cliente, enviar os dados de uma fatura para esta ser gerada e enviada por *email*, entre outros. Foi desenvolvido para alimentar o modelo de dados do mesmo, com funções sobre: criar, atualizar, remover e devolver informações sobre faturação e cobrança.
A figura 6.3 representa a arquitetura desenvolvida para suportar estas funcionalidades, mencionadas anteriormente.

A implementação deste domínio partiu da mesma lógica do anterior. A duração da entrega deste domínio demorou tanto como a anterior, porque foi desenvolvida uma camada superior a estes dois domínios de forma a conseguir registar ambos os domínios com um só pedido. Esta implementação deve-se ao facto que grande parte dos clientes nesta área não se encontram registados na base de dados da empresa, nem têm conta, e no primeiro contacto que estes normalmente têm com um operador, subscrevem planos ou compram produtos, o que acaba por demorar mais tempo ao cliente e ao operador para registar a conta e depois efetuar a faturação e cobrança. Assim com apenas um pedido, com toda a informação da ordem do cliente com os seus dados, este cria a conta do mesmo e ativa o domínio de faturação e cobrança. Esta arquitetura apresenta-se na figura 6.4.

Concluindo, estes domínios foram desenvolvidos através da documentação feita pela cliente, realizada pelos arquitetos funcionais dos mesmos. Estes são responsáveis por tratar e gerir toda a informação do cliente e faturação/cobrança. O *API layer*, foi desenvolvido para um cenário especifico, quando o cliente não possui uma conta criada ou a mesma não pos-

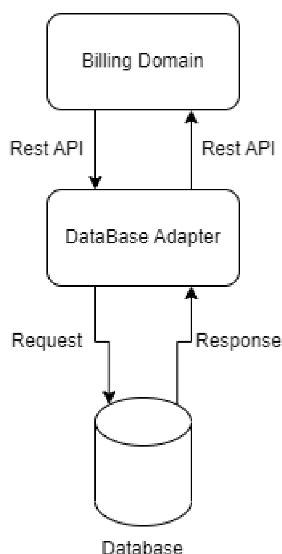

Figura 6.3: Arquitetura do Domínio de *Billing*.

sui informação suficiente para realização da ordem pedida. Esta é utilizada num cenário de compra, onde recebe todas as informações juntas (conta do cliente e dados de faturação e cobrança) e posteriormente são enviadas para os domínios respetivos, sendo todos os dados armazenados e autenticados. Este acaba por ser uma forma efetiva e rápida de criação de conta e faturação/cobrança, poupando recursos e tempo ao cliente e ao operador do sistema, que está a suportar a compra/ordem do cliente.

Neste microserviço, o estagiário foi responsável por criar o modelo de dados, bem como implementar as API´s e lógica de negócio. Também implementou testes unitários, testes integrados e resolução de *bugs* e defeitos.

6.2.3 Criação de Ordem

A funcionalidade de criação de ordem tem como objetivo lidar com os pedidos/compras dos clientes. Este é responsável por fazer com que um pedido passe por todos os processos necessários para este ser entregue ao cliente, desde iniciar a encomenda, até esta estar totalmente completa. Por exemplo, esta é responsável por tratar um pedido de atualização do plano de internet de um cliente, este inicializa a ordem, processa a parte logística, comunica com as entidades/parceiros (seja da parte logística ou de software) que sejam necessários para esta entrega, e quando tudo estiver pronto completa a encomenda.

A arquitetura desta funcionalidade encontra-se na figura 6.5. Nesta podemos verificar que a ordem pode ser feita de duas formas possíveis: através de um carrinho de compras que é realizado pelo *customer* (exemplo de um site de compras online), que é a forma mais tradicional de venda mais conhecida atualmente, ou fazer em contacto direto com uma operador/a da empresa de telecomunicações. Este último caso surge muitas vezes por subscrições feitas por chamada com o/a operador/a e esta ordem não faz sentido ser criada por um carrinho de

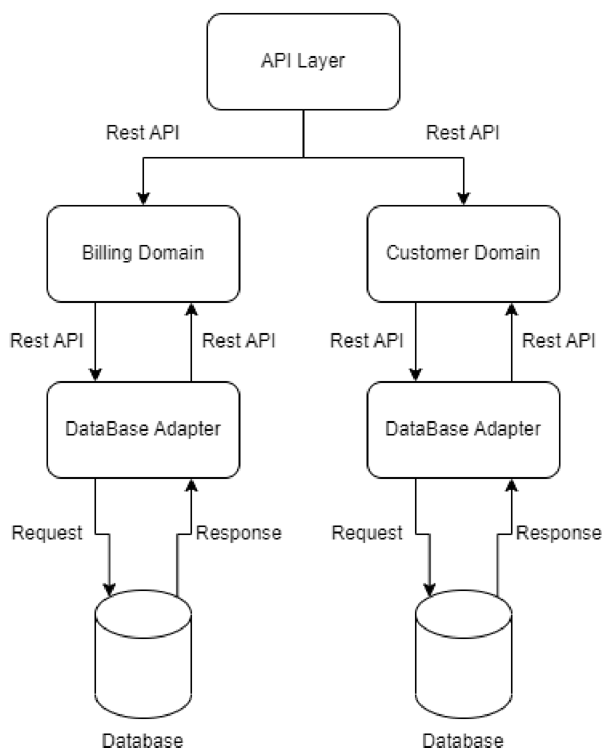

Figura 6.4: Arquitetura da Junção dos Domínios.

compras, mas sim por um pedido feito diretamente no microserviço responsável pela criação da ordem. No caso desta ordem ser realizada no carrinho de compras, todos os produtos presentes neles são validados contra os que estão no catalogo antes da criação da ordem.

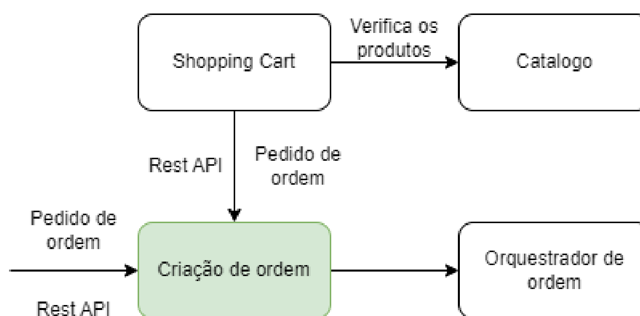

Figura 6.5: Arquitetura da Criação de Ordem.

Nesta figura 6.5, podemos ver que esta implementação teve consequentemente a implementação de mais 3 microserviços. O *shopping cart*, como o nome sugere é o conhecido carrinho de compras de um cliente, que este pode editar, adicionar mais produtos, entre outras funcionalidades. O catalogo é o microserviço responsável por contactar diretamente com o catalogo da empresa, fazer consultas a produtos, editar, adicionar e remover produtos do mesmo. Por fim, o orquestrador da ordem, é representado como um microserviço único, mas a implementação real é um conjunto de microserviços que são responsáveis por cada

processo e estado da ordem. Este não pode ser explicado com profundidade devido à exposição da lógica de negócio do cliente, defendido pela secção 1.5. Todas estas implementações consequentes da criação de ordem serão explicadas nas seguintes subsecções.

Neste microserviço, o estagiário foi responsável por criar o modelo de dados, bem como implementar as API´s. Também implementou testes unitários, testes integrados e resolução de *bugs* e defeitos.

6.2.3.1 Catálogo

O catálogo é responsável por gerir todos os produtos disponibilizados pela empresa cliente, desde criar produtos, remover, editar, atualizar, aplicar descontos, mostrar os produtos disponíveis para um determinado *customer*, entre outros. De outra forma, é o microserviço responsável por gerir toda a informação presente no catálogo, que é disponibilizado ao *customer*.

Este é utilizado para validar o carrinho de compras do *customer*, e é através deste que se apresentam os descontos disponíveis e verificar se os mesmos estão disponíveis para o respetivo *customer*, ou seja, o *customer* cria o seu carrinho de compras com os produtos que estão disponíveis pela interface e quando este submete o carrinho para finalização do mesmo, valida todos os produtos com os que estão no catálogo, para desta forma, prevenir venda de produtos indisponíveis.

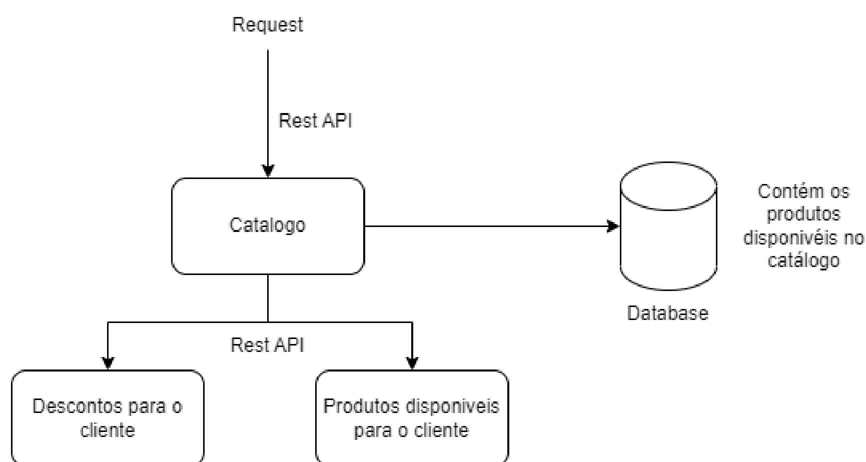

Figura 6.6: Arquitetura do Catálogo.

Neste microserviço, o estagiário foi responsável por criar o modelo de dados, bem como implementar as API´s. Também implementou testes unitários, testes integrados e resolução de *bugs* e defeitos.

6.2.3.2 *Shopping Cart*

O *Shopping Cart* permite ao utilizador criar um carrinho de compras com os produtos escolhidos pelo mesmo. Este tem funcionalidades como, adicionar produtos, remover, editar o carrinho de compras e a quantidade dos produtos. Este tem o mesmo comportamento que um carrinho de compras de produtos online. Este quando é criado faz a associação automática dos dados do *customer* e *billing*, deixando apenas o utilizador tratar dos produtos que pretende.

Neste microserviço, o estagiário foi responsável por implementar regras e validações, testes unitários, testes integrados e resolução de *bugs* e defeitos.

6.2.3.3 Orquestrador de Ordem

O orquestrador de ordem é responsável por tratar de todo o processo da ordem, desde a ativação de todos os produtos escolhidos pelo utilizador, como verificar a disponibilidade dos mesmo. De outra forma, este é responsável pelo ciclo da criação de ordem até a entrega do produto ao cliente.

Este desempenha um papel essencial na gestão eficiente e otimizada dos processos relacionados à prestação de serviços de telecomunicações. Permite coordenar, automatizar e monitorizar diversas atividades e recursos, garantindo a entrega eficaz de serviços de telecomunicações. Atua como um ponto centralizado de controlo, permitindo que as operadoras de telecomunicações consigam gerir a ativação, modificação e desativação de serviços para os clientes, ou seja, simplifica e agiliza a execução de tarefas operacionais, reduzindo a intervenção manual e os erros associados.

Uma das suas principais funcionalidades é a capacidade de criar e gerir os fluxos necessários de trabalho automatizados. Isso permite que as operadoras criem processos personalizados e padronizados para lidar com diferentes tipos de solicitações de serviços.

Neste microserviço, o estagiário foi responsável por expor as API's definidas, implementar testes unitários, testes integrados, resolver *bugs* e defeitos e lógica de negócio.

6.3 Conclusão

Neste capítulo, foram apresentadas as implementações realizadas durante o estágio e o que o estagiário fez em cada uma destas. Estas implementações não se encontram totalmente descritas derivado ao direito de propriedade intelectual do cliente 1.5.

Neste, são apresentadas as arquiteturas abstratas, sem lógica de negócio ou detalhes da sua implementação, mas demonstra claramente o que foi desenvolvido, através das tecnologias e metodologias apresentadas na secção 3, e qual o seu objetivo no projeto. De notar, que

todo este processo foi acompanhado pelos elementos mais seniores da equipa, que levou ao estagiário adaptar-se da melhor possível e aprender de forma mais rápida e assertiva, o que levou com o passar do tempo a que o estagiário fosse implementando tarefas com maior dificuldade.

Capítulo 7

Conclusão

7.1 Conclusões Principais

Em conclusão, o estágio na ReadinessIT teve um maior foco no desenvolvimento e implementação de microserviços na área de telecomunicações e foi uma experiência enriquecedora e altamente benéfica para o meu crescimento enquanto pessoa e profissional. Durante todo este período, pude aprender e aprofundar os meus conhecimentos no desenvolvimento de *software* e nas metodologias de trabalho, principalmente na metodologia *agile/scrum*, que é das metodologias mais utilizadas no mercado de trabalho, aprendendo também sobre os benefícios e desafios do desenvolvimento por microserviços.

Ao longo deste estágio, tive a oportunidade de colaborar com uma profissionais experientes, com muitos anos na área de telecomunicações e altamente qualificados. Através dos desafios diários e das suas resoluções, pude aplicar conceitos teóricos aprendidos na universidade e na academia da empresa, de forma adquirir habilidades práticas necessárias para o mercado de trabalho e desenvolvimento da carreira.

A implementação de microserviços permitiu uma arquitetura modular e escalável, trazendo benefícios significativos para o desenvolvimento deste projeto. A divisão das funcionalidades em serviços mais pequenos, menos complexos e independentes facilitou a manutenção (tratamento de erros e *bugs*) e atualização dos sistemas, além de promover maior flexibilidade e agilidade no desenvolvimento de novas funcionalidades.

Durante o estágio, também tive a oportunidade de aprender sobre *standards* de boas práticas de desenvolvimento de microserviços, como o uso de *containers* e orquestração de serviços com ferramentas como *Docker Desktop* e *Kubernetes*. Essas práticas foram essenciais para garantir a disponibilidade e a confiabilidade dos serviços implementados.

Além disso, pude aprimorar as minhas habilidades de trabalho em equipa, comunicação e resolução de problemas. Trabalhar neste ambiente colaborativo e organizado ajudou me a entender a importância da cooperação e da troca de conhecimento para o sucesso de um projeto.

Por fim, adquiri algum conhecimento na área de telecomunicações, que é uma área bastante complexa, sendo esta a maior área de negócio da ReadinessIT, o que torna este conhecimento bastante importante e valorizado pela mesma. Esta experiência certamente impulsionou o meu crescimento e conhecimento profissional e preparou-me e motivou-me para

futuros desafios na área.

7.2 Trabalho Futuro

O desenvolvimento após a duração do estágio, neste projeto, continuará em constantes alterações e atualizações, de forma a atender às necessidades do cliente. Neste momento, foram definidas alterações nos modelos de dados, para estarem de acordo com os padrões do *TMForum*, visto que, a equipa de desenvolvimento do cliente não seguiu totalmente as definições dos modelos de dados, através dos padrões do *TMForum*.

Por fim, se o estagiário se mantiver neste projeto e na empresa continuará a desempenhar as funções descritas no decorrer deste relatório. Estas funções incluem a participação nas reuniões de diárias e definição ou criação de tarefas, tanto como na participação e resolução de novas funcionalidades.

Bibliografia

[1] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 000 149–000 154. xiii, 5, 7, 8

[2] M. Baboi, A. Iftene, and D. Gîfu, "Dynamic microservices to create scalable and fault tolerance architecture," *Procedia Computer Science*, vol. 159, pp. 1035–1044, 2019, knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705091931467X xiii, 5, 9

[3] Pmadmin. (2023) What is scrum? | the agile journey with pm-partners. [Online]. Available: https://www.pm-partners.com.au/the-agile-journey-a-scrum-overview/ xiii, 13, 14

[4] Jason John. (2023) Use o postman com a api do microsoft graph - microsoft graph. [Online]. Available: https://learn.microsoft.com/pt-br/graph/use-postman xiii, 17

[5] Benedict Quinn. (2023) Performance testing with jmeter. [Online]. Available: https://blog.scottlogic.com/2021/12/09/Performance-Testing-with-JMeter.html xiii, 18

[6] SmartBear Software. (2023) Service mocking overview | rest mocking. [Online]. Available: https://www.soapui.org/docs/rest-testing-mocking/service-mocking-overview/ xiii, 19

[7] Portableapps. (2023) Draw.io portable 20.8.16 (diagramming) released | portableapps.com. [Online]. Available: https://portableapps.com/news/2023-02-06--draw.io-portable-20.8.16-released xiii, 20

[8] Atlassian. (2023) Ferramentas ágeis de gestão de projetos para equipas de software. [Online]. Available: https://www.atlassian.com/br/software/jira/agile xiii, 19, 20

[9] K15t. (2023) How to make beautiful pages in confluence | confluence best practices (2023). [Online]. Available: https://www.k15t.com/rock-the-docs/misc/how-to-make-beautiful-pages-in-confluence xiii, 21

[10] Srebalaji Thirumalai. (2023) How to work in multiple git branches simultaneously. [Online]. Available: https://gitbetter.substack.com/p/how-to-work-in-multiple-git-branches xiii, 22

[11] JetBrains. (2023) Themes in intellij-based ides | the jetbrains platform blog. [Online]. Available: https://blog.jetbrains.com/platform/2021/10/themes-in-intellij-based-ides/ xiii, 23

[12] Amazon Studio. (2023) Amazon rds for postgresql. [Online]. Available: https://catalog.us-east-1.prod.workshops.aws/workshops/2a5fc82d-2b5f-4105-83c2-91a1b4d7abfe/en-US/2-foundation/lab9-postgres-introduction/task1 xiii, 24

[13] Steph Rifai. (2023) New extensions and container interface enhancements in docker desktop 4.9 | docker. [Online]. Available: https://www.docker.com/blog/new-extensions-and-container-interface-enhancements-in-docker-desktop-4-9/ xiii, 25

[14] Gitesh Dhore. (2023) Apache kafka architecture and use cases explained - analytics vidhya. [Online]. Available: https://www.analyticsvidhya.com/blog/2022/07/apache-kafka-architecture-and-use-cases-explained/ xiii, 26

[15] LOVISA JOHANSSON. (2023) Part 1: Rabbitmq for beginners - what is rabbitmq? - cloudamqp. [Online]. Available: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html xiii, 26, 27

[16] mongo-express. (2023) Github - mongo-express/mongo-express: Web-based mongodb admin interface, written with node.js and express. [Online]. Available: https://github.com/mongo-express/mongo-express xiii, 31

[17] Wiktor Dyngosz. (2023) How to handle sonarqube setup in java microservice project | wiktor dyngosz. [Online]. Available: https://www.wiktordyngosz.pl/14-02-2019-sonar/ xiii, 33

[18] GeeksforGeeks. (2023) Introduction to spring boot - geeksforgeeks. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-spring-boot/ xiii, 34

[19] tutorialspoint. (2023) Soa - enterprise service bus. [Online]. Available: https://www.tutorialspoint.com/soa/soa_enterprise_service_bus.htm xiii, 40, 43

[20] M. Söylemez, B. Tekinerdogan, and A. Kolukısa Tarhan, "Challenges and solution directions of microservice architectures: A systematic literature review," *Applied Sciences*, vol. 12, no. 11, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/11/5507 5

[21] T. Cerny, M. J. Donahoo, and J. Pechanec, "Disambiguation and comparison of soa, microservices and self-contained systems," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, ser. RACS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 228–235. [Online]. Available: https://doi.org/10.1145/3129676.3129682 5

[22] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," *CoRR*, vol. abs/1901.04946, 2019. [Online]. Available: http://arxiv.org/abs/1901.04946 5

[23] Postman, Inc. (2023) Postman. [Online]. Available: https://www.postman.com/ 16

[24] ApacheJMeter Foundation. (2023) Apache jmeter - apache jmeter&trade. [Online]. Available: https://jmeter.apache.org/ 17

[25] Draw.io. (2023) draw.io – diagrams for confluence and jira. [Online]. Available: https://drawio-app.com/ 18

[26] Atlassian. (2023) Confluence: A brief overview | atlassian. [Online]. Available: https://www.atlassian.com/software/confluence/guides/get-started/confluence-overview#about-confluence 20

[27] GitLab B.V. (2023) The devsecops platform | gitlab. [Online]. Available: https://about.gitlab.com/ 21

[28] JetBrains. (2023) Intellij idea – the leading java and kotlin ide. [Online]. Available: https://www.jetbrains.com/idea/ 22

[29] Ivan de Souza Cardoso. (2023) Postgresql: saiba o que é, para que serve e como instalar. [Online]. Available: https://rockcontent.com/br/blog/postgresql/ 23

[30] Docker. (2023) Docker: Accelerated, containerized application development. [Online]. Available: https://www.docker.com/ 24

[31] Apache Software Foundation. (2023) Apache kafka. [Online]. Available: https://kafka.apache.org/ 25

[32] The Apache Software Foundation. (2023) Maven – introduction. [Online]. Available: https://maven.apache.org/what-is-maven.html 28

[33] goelshubhangi3118. (2023) Introduction to java - geeksforgeeks. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-java/ 29

[34] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt e Christian Stein. (2023) Junit 5 user guide. [Online]. Available: https://junit.org/junit5/docs/snapshot/user-guide/ 30

[35] MongoDB, Inc. (2023) Mongodb: The developer data platform | mongodb. [Online]. Available: https://www.mongodb.com/ 30

[36] Redis Ltd. (2023) Redis. [Online]. Available: https://redis.io/ 31

[37] Sonar. (2023) Self-managed | sonarqube | sonar. [Online]. Available: https://www.sonarsource.com/products/sonarqube/ 33

[38] Spring Boot. (2023) Spring boot reference documentation. [Online]. Available: https://spring.io/why-spring 34