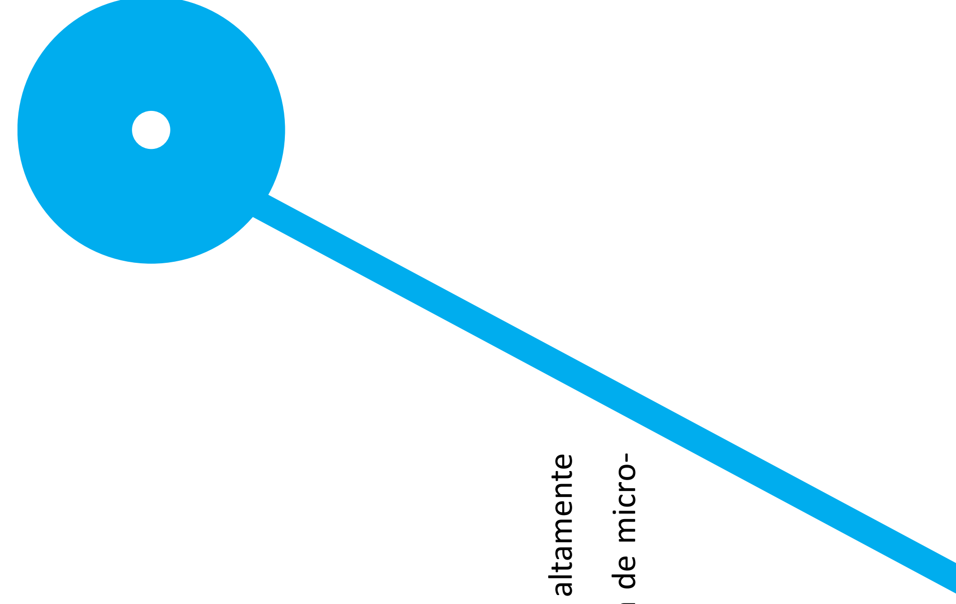
ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO



Plataforma de partilha de bicicletas altamente escalável através de uma arquitetura de micro-serviços
Daniel Fernando Ferraz Pinto
M MESTRADO Engenharia Informática

Daniel Fernando Ferraz Pinto. Plataforma de partilha de bicicletas altamente escalável através de uma arquitetura de micro-serviços
10/2022

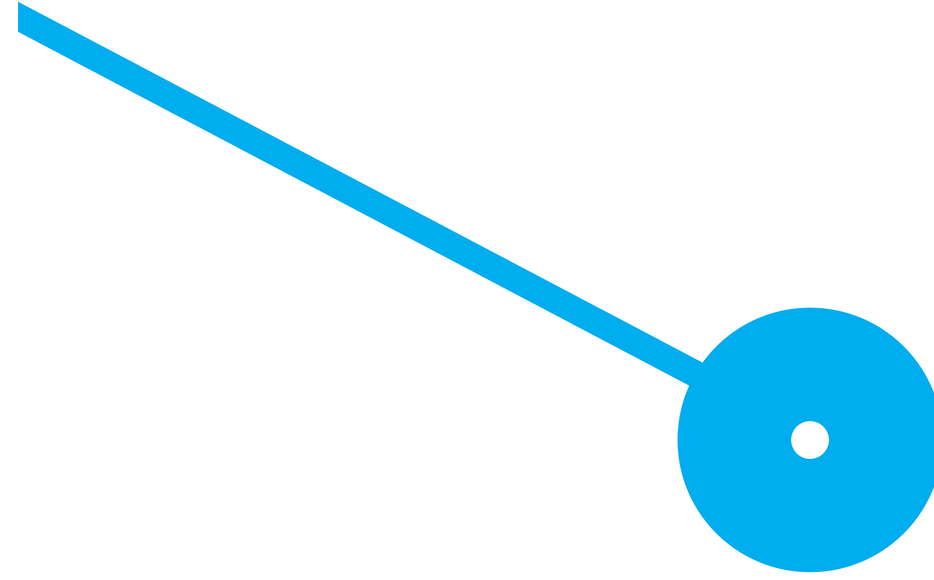
ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

M MESTRADO
Engenharia Informática

# Plataforma de partilha de bicicletas altamente escalável através de uma arquitetura de micro-serviços

Daniel Fernando Ferraz Pinto

10/2022



P.PORTO


ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

MESTRADO
Engenharia Informática

# Plataforma de partilha de bicicletas altamente escalável através de uma arquitetura de micro-serviços


Daniel Fernando Ferraz Pinto

Ricardo Jorge da Silva Santos

# Agradecimentos

Em primeiro lugar gostaria de agradecer à Escola Superior de Tecnologia e Gestão | Politécnico do Porto, e a todos os professores que me lecionaram, por todos os conhecimentos transmitidos, e pela formação académica que me proporcionaram, em especial ao Professor Ricardo Santos, o meu orientador, pela sua disponibilidade no auxílio da realização deste documento.

Gostaria de mencionar também, a atual empresa onde laboro, a LetsGetChecked, e outras por onde passei, Critical Techworks e Stratio Automotive, assim como todos os colegas de trabalho, que tiveram um papel muito importante no meu desenvolvimento pessoal e profissional, e me permitiram desenvolver conhecimentos nomeadamente na área dos micro-serviços.

Agradeço também aos meus colegas de mestrado, pela partilha de conhecimento e espírito de camaradagem.

Gostaria também de deixar um agradecimento muito especial aos meus pais, irmã e namorada, que, apesar de não estarem diretamente vinculados a este documento tiveram a maior importância em todo o meu percurso escolar.

Esta página foi deixada em branco propositadamente

# Resumo

Este documento tem como objetivo descrever o estudo e implementação, de uma de solução para um sistema de bicicletas partilhadas, que permita obter a localização, disponibilidade de bicicletas, efetuar alugueres, pagamentos, dar avaliações e obter informações em tempo real, através de uma abordagem arquitetural de micro-serviços. O principal objetivo é de criar um sistema distribuído altamente escalável e rápido que permita atender uma alta procura de bicicletas em horas de ponta em meio urbano. É abordado o atual estado da arte em sistemas de partilha de bicicletas e feita uma avaliação do que seria útil para este sistema. Para sustentar a proposta são apresentados diagramas do modo de funcionamento e arquitetura da solução.

A nível de implementação, é descrito todo o processo de desenvolvimento, assim como arquiteturas e tecnologias utlizadas, para o desenvolvimento técnico da solução proposta.

A solução foi desenvolvida com recurso a micro-serviços numa instância de Kubernetes preparada para auto escalabilidade dos serviços. Estes comunicam-se maioritariamente assincronamente, agindo de forma desacoplada e escalável, seguindo as boas práticas em arquiteturas micro-serviços.

Visualmente foi desenvolvida uma aplicação móvel multiplataforma, que comunica com os serviços e permite que o utilizador tire o máximo partido de toda esta arquitetura, oferecendo uma experiência fluida resiliente a falhas e períodos de alta demanda.

**Palavras-chave**: Bicicletas partilhadas, mobilidade sustentável, micro-serviços.

Esta página foi deixada em branco propositadamente

# Abstract

This document aims to describe the study and implementation of a solution for a shared bicycle system, which allows obtaining the location, availability of bicycles, making rentals, payments, giving assessments and obtaining information in real time, through an approach micro-services architecture. The main objective is to create a highly scalable and fast distributed system that can meet a high demand for bicycles at peak times in urban environments. The current state of the art in bicycle sharing systems is addressed and an assessment is made of what would be useful for this system. To support the proposal, diagrams of the solution's operating mode and architecture are presented.

At the implementation level, the entire development process is described, as well as the architectures and technologies used for the technical development of the proposed solution.

The solution was developed using micro-services in a Kubernetes instance prepared for self-scaling of services. These communicate mostly asynchronously, acting in a decoupled and scalable way, following best practices in micro-services architectures.

Visually, a cross-platform mobile application was developed, which communicates with the services and allows the user to take full advantage of this entire architecture, offering a fluid experience that is resilient to failures and periods of high demand.

**Keywords**: Bicycle sharing, sustainable mobility, micro-services.

Esta página foi deixada em branco propositadamente

# Conteúdo


1 Introdução ... 1

1.1 Apresentação e Oportunidade do Tema ... 1

1.2 Objetivos principais ... 2

1.3 Contributos inovadores ... 2

2 Revisão da literatura ... 3

2.1 Micro-serviços ou monolítico ... 5

3 Solução proposta ... 11

4 Desenvolvimento ... 17

4.1 Tecnologias utilizadas ... 17

4.1.1 Framework .NET ... 18

4.1.2 MassTransit ... 18

4.1.3 RabbitMQ ... 18

4.1.4 MongoDB ... 19

4.1.5 Eureka ... 19

4.1.6 Kubernetes ... 19

4.1.7 Etcd ... 20

4.1.8 Flutter ... 21

4.1.9 Google Firebase ... 21

4.1.10 Ocelot ... 21

4.1.11 MQTT ... 22

4.1.12 Mosquitto ... 22

4.1.13 Polly ... 22

4.2 Padrões arquiteturais implementados ... 22

4.2.1 Saga ... 23

4.2.2 Event sourcing ... 24

4.2.3 Database per Service ... 25

4.2.4 API gateway ... 26

4.2.5 Circuit breaker ..... 27
4.2.6 Service instance per container ..... 28
4.2.7 Service Registry ..... 28
4.2.8 Access token ..... 29
4.2.9 Health check API ..... 29
4.3 Contextos limitados ..... 29
4.4 Arquitetura ..... 30
4.5 Comunicação e processos ..... 33
4.5.1 Processo de notificação ..... 33
4.5.2 Processo de registo e autenticação ..... 35
4.5.3 Processo de processamento de dados de viagem ..... 36
4.5.4 Processo de aluguer ..... 37
4.5.5 Processo de pagamento ..... 39
4.6 Escalabilidade ..... 40
4.7 Persistência de dados ..... 41
4.8 Comunicação assíncrona ..... 41
4.9 Infraestrutura ..... 42
4.10 Dispositivo IoT ..... 43
4.11 Aplicação móvel ..... 45
4.12 Testes ..... 51
4.12.1 Testes unitários ..... 52
4.12.2 Testes de integração ..... 53
4.12.3 Continuous integration ..... 55
5 Conclusão ..... 57
6 Referências bibliográficas ..... 59

# Lista de tabelas

Tabela 1 Principais funcionalidades de monolíticos e micro-serviços ..................................... 8
Tabela 2 - Dominios e serviços propostos ................................................................... 13

Esta página foi deixada em branco propositadamente

# Lista de figuras

Figura 1 Comparação entre arquitetura monolítica e micro-serviços ..... 6
Figura 2 Fluxo do processo de registo ..... 12
Figura 3 Fluxo do processo de aluguer de bicicleta ..... 13
Figura 4 Arquitetura proposta para o serviço de partilha de bicicletas ..... 14
Figura 5 Diagrama de funcionamento do kubernetes ..... 20
Figura 6 Saga baseada em coreografia ..... 23
Figura 7 Saga baseada em orquestração ..... 24
Figura 8 - Arquitetura event sourcing ..... 25
Figura 9 Exemplo da arquitetura database per service ..... 26
Figura 10 - Arquitetura API gateway ..... 27
Figura 11 - Arquitetura da plataforma de partilha de bicicletas ..... 31
Figura 12 Processo de notificação ..... 34
Figura 13 Processo de registo e autenticação ..... 35
Figura 14 Processo de processamento de dados da viagem ..... 36
Figura 15 Processo de aluger de bicicleta ..... 37
Figura 16 Sequência de eventos do processo de alugar ..... 39
Figura 17 Processo de pagamento ..... 39
Figura 18 Escalabilidade vertical e horizontal ..... 40
Figura 19 - Dashboard do kubernetes com os nodes utilizados ..... 43
Figura 19 - Arquitetura do dispositivo IoT de gestão de docas ..... 44
Figura 20 - Dispositivo Iot desenvolvido ..... 45
Figura 20 - Registo e autenticação na aplicação móvel ..... 46
Figura 21 - Bicicletas proximas e detalhes na aplicação móvel ..... 47
Figura 22 - Detalhes do utilizador e gestão de cartões bancários na aplicação móvel ..... 48
Figura 23 - Scan de código QR e viagem na aplicação móvel ..... 49
Figura 24 - Feedback da viagem na aplicação móvel ..... 50
Figura 25 - Listagem e detalhes de aluguer na aplicação móvel ..... 51
Figura 26 - Pirâmide de testes ..... 52
Figura 27 - Pipeline de CI em GitHub Actions ..... 56

Esta página foi deixada em branco propositadamente

# Nomenclatura

## Abreviaturas, Siglas e Acrónimos

FCM – Firebase Cloud Messaging

API – Aplication programming interface (Interface de Programação de Aplicações)

IoT – Internet of things (Internet das coisas)

JWT – Json web token

FIFO – First in first out

DDD – Domain driven design

SOA – Service oriented architecture

Bloc - Business Logic Object Components

NFC - Near Field Communication (Comunicação de Campo Próximo)

CI – Continuous integration

Esta página foi deixada em branco propositadamente

# 1 Introdução

Pretende-se com este capítulo apresentar a proposta de uma plataforma de partilha de bicicletas com recurso a micro-serviços, assegurando a alta escalabilidade e performance do sistema.

Inicialmente é efetuada uma introdução do tema, seguido de uma revisão da literatura, onde são abordados sistemas existes, e estudos relevantes, de seguida é proposta uma solução para o problema apresentado seguido de algumas considerações finais.

## 1.1 Apresentação e Oportunidade do Tema

A União Europeia tem vindo a orientar as políticas públicas urbanas para o objetivo da mobilidade sustentável, protegendo o espaço público e a saúde e bem-estar dos cidadãos. A promoção da mobilidade urbana com estratégias de baixa emissão de CO2 é um dos eixos do quadro de planos integrados de mobilidade sustentável, onde se inserem as plataformas de bicicletas partilhadas [1].

A plataforma de partilha de bicicletas assenta a sua operacionalidade num conjunto de bicicletas elétricas, que, estão distribuídas pela cidade, promovendo-se, desta forma, a liberdade de mobilidade em cada aluguer. Os utilizadores do serviço de partilha de bicicletas podem utilizar uma bicicleta mediante um registo numa aplicação mobile. Depois de registados, têm a possibilidade de consultar, de uma lista de bicicletas disponíveis, a sua localização, os detalhes da bicicleta que podem alugar (marca, modelo, nível de bateria), podendo ser conduzidos até à localização da bicicleta. O levantamento é feito depois de digitalizado um código QR estampado em cada bicicleta, que faz com que esta seja desbloqueada pela doca (local onde a bicicleta é colocada e bloqueada), e permita o seu manuseamento. O utilizador, pode então, começar a utilização, sabendo-se que, durante a mesma, são recolhidos dados de GPS que permitirão a reconstrução de percursos e histórico de utilização.

O modelo de negócio é muito similar ao que vemos nos táxis, o utilizador paga com base no tempo que utilizou, sendo que a tarifa pode variar mediante intervalos de tempo. Assim, no final de cada utilização, é apresentada ao utilizador a quantia a liquidar pelo aluguer e um formulário de avaliação da experiência, onde o mesmo pode indicar algum problema relacionado com a bicicleta utilizada ou com as docas de levantamento e/ou entrega.

Esta proposta consiste em explorar uma abordagem baseada em micro-serviços. Tradicionalmente o desenvolvimento deste tipo de plataformas seguia uma abordagem

monolítica, que carece da flexibilidade necessária para lidar com incerteza assim como com dispositivos heterogéneos de forma eficiente [2]. As arquiteturas micro-serviços seguem uma abordagem de implementação bastante diferente que consiste na implementação de serviços mais pequenos circunscritos a um processo, sendo independentes, mais facilmente escaláveis e desacoplados, não implicando assim, em caso de falha, uma falha de todo o sistema [3].

Pretende-se a criação de um conjunto de pequenos serviços com recurso a sistemas e frameworks que garantam a escalabilidade e comunicação assíncrona entre eles, assim como, a construção de uma aplicação para dispositivos móveis, capaz de comunicar com estes serviços e disponibilizar toda a plataforma de partilha de bicicletas ao utilizador final.

## 1.2 Objetivos principais

O principal objetivo passa pelo desenvolvimento de uma solução baseada numa arquitetura orientada a micro-serviços que venha dar resposta ao problema da partilha de bicicletas nas cidades [4]. Como o problema tem uma variabilidade muito grande de necessidades/acessos pretende-se que a solução seja um sistema escalável *on-demand*, garantindo que cada serviço utiliza apenas os recursos necessários.

## 1.3 Contributos inovadores

O objetivo desta proposta é criar um sistema de partilha de bicicletas sob uma arquitetura de micro-serviços, que ofereça flexibilidade no tipo de bicicletas usadas, assim como na implantação do mesmo, sendo que este apenas necessita da informatização das docas, operando todo o restante sistema na *cloud* e com recurso a uma aplicação móvel que permite simplificar registos, pagamentos, otimizar viagens, fornecer informações em tempo real, e recolher opinião das viagens, com o objetivo de tornar mais simples e tentador o acesso a este tipo de serviço. A utilização de micro-serviços vai permitir uma plataforma altamente escalável e flexível com um consumo de recursos otimizado garantindo performance ao menor custo.

Este sistema poderá ser implementado nas cidades, com a colocação das docas em vários pontos estratégicos, sendo que este seria o único desafio logístico. Esta proposta permitiria às câmaras, juntas de freguesia ou organizações, implementar um serviço de partilha de bicicletas numa determinada área com relativa facilidade a um custo reduzido.

# 2 Revisão da literatura

No que diz respeito a serviços de partilha de bicicletas, existem atualmente inúmeros a atuar nas grandes cidades, alguns baseados em modelos tradicionais, outros mais modernos e tecnológicos.

Entre os serviços mais tradicionais, temos aqueles que são baseados no princípio de as bicicletas serem colocadas em docas, e o seu levantamento e devolução ser feito numa doca do provedor.

Alguns destes serviços funcionam com subscrições, sejam diárias, anuais ou mensais, em que, o utilizador associa o seu cartão bancário à plataforma e é-lhe cobrado um valor correspondente à sua subscrição. Outros são gratuitos e financiados pelos municípios, e existem ainda os que são cobrados com base no período de utilização da bicicleta [2].

Geralmente o utilizador pode levantar a sua bicicleta em doca através de um cartão RFID ou aplicação móvel estando as estações de docas espalhadas pela cidade em zonas estratégicas.

Estes sistemas foram evoluindo, e muitos deles já oferecem uma aplicação móvel onde é possível efetuar pagamentos, encontrar docas, verificar a disponibilidade das bicicletas, e avaliar o serviço prestado. Entre estes modelos, temos por exemplo o *GIRA*, presente na cidade de Lisboa, o *CitiBike* em Nova Iorque ou o *Vélib* em Paris [5].

Surgiram há poucos anos, serviços de partilha de bicicletas ou trotinetas mais modernos e tecnológicos, o objetivo é o mesmo, no entanto o modelo de negócio e modo de funcionamento difere dos tradicionais.

Estes serviços geralmente não possuem docas, o desbloqueio é efetuado diretamente na própria bicicleta, e estas são exclusivamente elétricas. O utilizador necessita ter a aplicação do serviço instalada com o seu cartão bancário associado e, pode efetuar o desbloqueio da bicicleta ou trotineta através do scan de um código presente na mesma, esta desbloquear-se-á e o utilizador poderá utilizá-la durante o tempo que pretender. Através da aplicação do serviço é possível verificar quais as bicicletas disponíveis, assim como a sua localização e nível de carga. Como não requerem doca, estas bicicletas são geralmente estacionadas na cidade arbitrariamente pelos utilizadores. O valor a pagar é gerado com base no período de utilização e é efetuado automaticamente através do cartão bancário associado previamente na aplicação [6]. Podemos encontrar este tipo de serviço em sistemas como o *Lime*, o *Circ* ou o *Bird* [7].

Já vários estudos foram feitos acerca de mobilidade urbana sustentável e de que modo, a tecnologia pode facilitar toda esta implementação [8][9], [10]. Num sistema de partilha de transporte, em que idealmente temos que ter comunicação em tempo real com toda a rede de bicicletas de modo a obter atualizações do seu estado, localização entre outras métricas, os micro-serviços podem efetivamente facilitar toda esta implementação, permitindo a um grande sistema distribuído ser dividido em pequenos pedaços, mais simples e modulares, facilitando a sua escalabilidade, fazendo com que cada serviço se comporte como um componente independente que pode ser escalado ou não independentemente dos restantes. Os micro-serviços oferecem também maior flexibilidade a nível tecnológico, permitindo que diferentes serviços usem diferentes linguagens, frameworks ou ferramentas. Por outro lado, a implementação de uma arquitetura de micro-serviços é mais complexa que uma monolítica, e num início de projeto traz mais desafios e exige mais esforço de desenvolvimento [11].

Num estudo realizado na Vilnius Gediminas Technical University [12], foi abordado o tema da migração de um sistema monolítico para micro-serviços, o artigo apresenta as vantagens e desvantagens de cada uma, assim como os desafios técnicos e arquiteturais. A complexidade na utilização de micro-serviços foi o principal ponto negativo apresentado, sendo que o autor considera que deve ser muito bem avaliado o problema e perceber se este custo se justifica. A flexibilidade e escalabilidade foi um dos principais pontos positivos apontados, pelo que, é importante perceber se o modelo de negócio ao qual se vai aplicar uma arquitetura de micro-serviços traz desafios a esse nível. Sendo um sistema de partilha de bicicletas amplamente utilizado nas grandes cidades, sobretudo em horas de ponta, é importante haver uma infraestrutura capaz de suportar aquele elevado volume de pedidos a determinada hora, com a possibilidade de que, na hora seguinte este número possa reduzir drasticamente. É também espectável que determinadas partes da aplicação sejam mais utilizadas, como por exemplo a parte dos alugueres. Deste modo, os micro-serviços tornam-se uma ótima solução para este tipo de problema, porque estando o software dividido em pequenas partes, poderemos escalar com facilidade apenas as partes com maior carga de trabalho, a fim de manter o sistema responsivo. Assumindo uma arquitetura de pequenos serviços completamente desacoplados que comuniquem entre si assincronamente, torna-se também mais fácil escalar o sistema horizontalmente [13], permitindo correr o mesmo serviço em várias máquinas a fim de aumentar o poder de processamento ao invés da abordagem clássica de escalabilidade vertical que consiste em adicionar mais capacidade de hardware a apenas uma máquina [14].

Num projeto desenvolvido e documentado por um estudante da universidade de Coimbra [15] durante o seu estágio curricular, este abordou o desenvolvimento de um serviço de reposicionamento estratégico das bicicletas, integrante num sistema de partilha de bicicletas. Recorrendo a algoritmos preditivos, que permitam perceber em que localização a bicicleta é

mais necessária num determinado período do dia, o sistema pretende, através da análise de utilização e localização das bicicletas e docas, otimizar o reposicionamento das bicicletas em docas. Diminuirá assim o número de bicicletas não utilizadas e a escassez em determinadas docas. Este projeto teve como objetivos, a previsão do uso das docas, a distribuição dinâmica de bicicletas, o diagnóstico de estações mais problemáticas em termos de excesso de bicicletas, tendo em consideração durante toda esta análise, a meteorologia atual assim como a topologia da doca. Com recurso a filas de mensagens, *APIs REST* e diferentes serviços, este projeto resulta num sistema escalável exposto aos utilizadores através de uma plataforma web, e aplicação móvel. Este tipo de abordagem para reposicionamento das bicicletas torna-se bastante relevante no modelo de negócio, e a sua implementação com recurso a micro-serviços permite que esta seja uma funcionalidade interessante, totalmente desacoplada dos restantes serviços.

Pretende-se na plataforma proposta, recolher o máximo de informação relevante possível, acerca das viagens, docas e disponibilidade a fim de futuramente, como possível incremento, fazer uma implementação de um serviço de reposicionamento de bicicletas, e dado que haja já previamente um set de dados com a informação necessária, esta implementação torna-se bastante mais simples.

## 2.1 Micro-serviços ou monolítico

Numa descrição simplificada, a arquitetura monolítica é um sistema único, não dividido. É uma aplicação de software em que diferentes componentes estão ligados a um único programa dentro de uma plataforma. Nela, existe uma dependência entre os serviços de uma mesma aplicação. Já os micro-serviços, são um tipo de arquitetura de software que desmembra aplicações em serviços independentes que comunicam entre si através de APIs bem definidas. Podemos então dizer que é uma aplicação mais leve que a criada num único bloco ou monólito [16].

Aplicações monolíticas são a abordagem padrão para o desenvolvimento de software. Apesar dessa tendência, a sua popularidade tem vindo a diminuir, pois são notoriamente difíceis de construir devido a vários problemas, como lidar com uma enorme base de código, implementar novas tecnologias, escalar e fazer *deploy*.

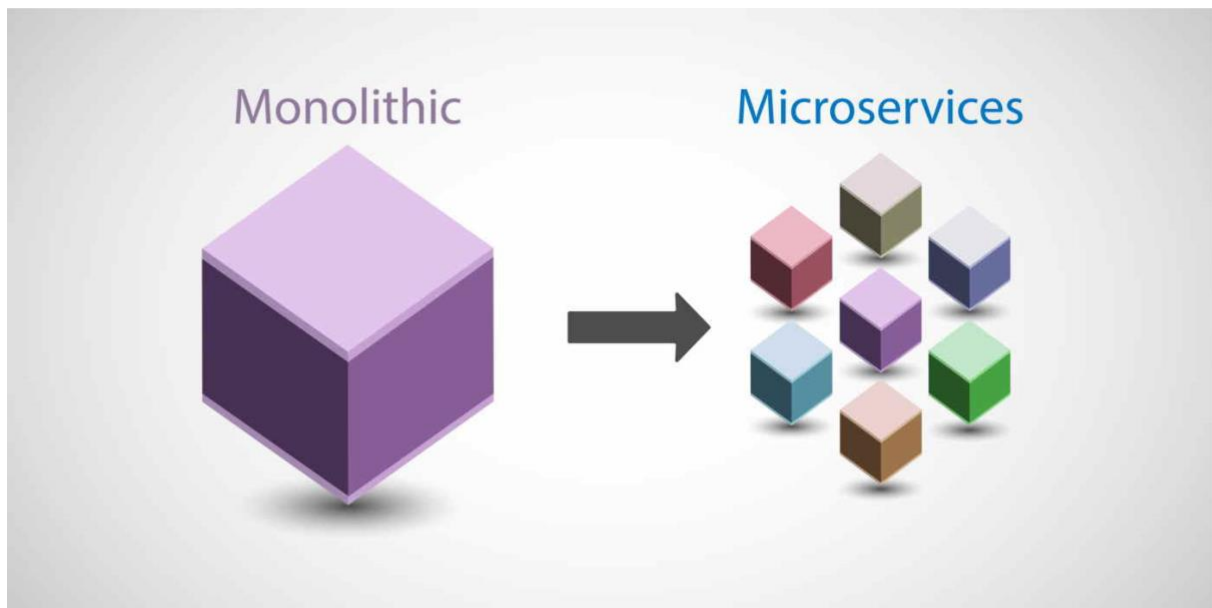


*Figura 1 Comparação entre arquitetura monolítica e micro-serviços*

Dependendo do contexto, arquiteturas de micro-serviços podem não ser a solução ideal, apesar de serem cada vez mais usadas em grandes empresas, pois ambas arquiteturas apresentam vantagens e desvantagens [17].

Os monólitos têm as seguintes vantagens:

- O desenvolvimento é mais fácil, o seu nível de complexidade é baixo, por isso são mais rápidos de desenvolver. Além disso, é possível começar com uma aplicação básica e depois adicionar recursos à medida que se desenvolve.
- Facilmente se inicia o desenvolvimento de um monólito, em parte, deve-se a um menor número de problemas transversais, como o registo ou detecção de erros. Ter todas essas preocupações transversais num só lugar permite que sejam tratadas com mais facilidade.
- Testar monólitos é fácil, podem ser facilmente executados e testados repetidamente porque são compostos por apenas um corpo.

Quanto às desvantagens, os monólitos têm os seguintes contras:

- É difícil compreender os monólitos. Dimensionar uma aplicação e adicionar mais funções pode parecer uma boa ideia no início, mas depois torna-se mais complexo de gerir à medida que o projeto vai crescendo.
- A combinação de monólitos e novas tecnologias é difícil. A arquitetura monolítica é uma unidade única, portanto, a introdução de novas tecnologias exigirá a reescrita de toda a aplicação.

- A natureza dos monólitos torna difícil alterá-los. Quando se começa a fazer alterações num sistema complexo, as chances de haver falhas na aplicação aumentam. Também pode levar algum tempo para que o desenvolvimento seja concluído.
- Monolíticos podem ser difíceis de manter, especialmente se forem mal projetados. Os sistemas monolíticos são conhecidos por terem processos fortemente acoplados, portanto, mesmo uma pequena alteração pode causar vários problemas relacionados à base de código. Uma única alteração pode resultar no não funcionamento de um programa inteiro.

Já no caso da arquitetura de micro-serviços tem as seguintes vantagens:

- **Componentes independentes** - é possível fazer deploy e atualizar todos os serviços de forma independente, oferecendo mais flexibilidade. Em segundo lugar, um bug num micro-serviço não afeta toda a aplicação, apenas esse serviço. Em termos de arquitetura monolítica versus micro-serviços, as aplicações de micro-serviços são muito mais fáceis de desenvolver.
- **Fácil compreensão** - um micro-serviço pode ser facilmente entendido e gerido porque é dividido em componentes menores e mais simples. O desenvolvedor apenas interage com um serviço específico relacionado aos seus objetivos de negócio.
- **A Escalabilidade é melhorada** - entre as vantagens da abordagem de micro-serviços está a capacidade de dimensionar cada elemento de forma independente. Isso é mais eficiente em termos de custo e tempo do que dimensionar aplicações monolíticas, mesmo quando não é necessário. À medida que uma empresa angaria mais clientes, o seu sistema monólito enfrentará cada vez mais problemas em termos de escalabilidade. Portanto, muitas empresas são obrigadas a reconstruir a sua arquitetura monolítica.
- **A tecnologia pode ser escolhida com flexibilidade** - a tecnologia não é um fator limitante para as equipas de engenharia, os micro-serviços podem ser construídos com uma variedade de tecnologias e estruturas, independentes uns dos outros.
- **Altos níveis de agilidade** – quando há uma falha num micro-serviço, isso afeta apenas esse serviço específico e não todo o sistema. Consequentemente, todas as mudanças e experimentos são realizados com menos erros e riscos reduzidos.

Apesar das vantagens já elencadas existem também as seguintes desvantagens:

- **A complexidade adicionada** - numa arquitetura de micro-serviços, os módulos e bases de dados precisam estar conectados entre si. Além disso, os serviços independentes em tal serviço devem ser *deployed* de forma independente.
- **Distribuição do sistema** - numa arquitetura de micro-serviços, várias bases de dados e módulos interagem, portanto, todas as conexões devem ser geridas com cuidado.
- **Preocupações transversais** - Existem várias preocupações transversais a serem consideradas ao projetar uma arquitetura de micro-serviços. Entre eles estão a externalização de configuração, métricas, registo e verificações de integridade.
- **Teste** - É muito difícil testar uma solução baseada em micro-serviços, pois há muitos componentes que são *deployed* de forma independente.

Em suma, podemos considerar as principais funcionalidades de ambos na Tabela 1.

*Tabela 1 Principais funcionalidades de monolíticos e micro-serviços*

| Monolíticos | Micro-serviços |
|---|---|
| Deploy simples | Escalabilidade |
| Agilidade limitada | Isolamento de erros |
| Baixa complexidade | Alta complexidade |
| Limitação de tecnologias | Flexibilidade |
| Testagem simples | Sistema fracamente acoplado |

De seguida Analisaremos a complexidade, confiabilidade, latência e escalabilidade da arquitetura monolítica versus micro-serviços para entender melhor as diferenças.

**Escalabilidade**

Não são apenas os micro-serviços que são escaláveis, um monólito também o pode ser. No entanto, monolíticos podem ser escalados numa dimensão executando várias cópias. Com um volume de dados crescente, a escalabilidade é impossibilitada. Um micro-serviço pode, portanto, ser dimensionado com menos recursos, o que é uma vantagem absoluta dos micro-serviços.

**Complexidade**

Os micro-serviços envolvem uma infinidade de códigos-fonte, estruturas e tecnologias, dependendo da complexidade da aplicação, vários servidores podem alojar os serviços, que se comunicam por meio de APIs entre eles.

A arquitetura desse tipo requer uma metodologia de desenvolvimento diferente e requer um nível mais alto de coordenação, conjunto de habilidades e compreensão da arquitetura geral.

**Latência**

Um micro-serviço envia ou recebe dados pela rede ao comunicar com outro serviço, bytes tornam-se sinais elétricos, que se tornam bytes novamente.

Os monólitos, por outro lado, não sofrem latência de rede, pois todos os serviços estão localizados no mesmo fluxo. Devido a esses motivos, os micro-serviços são mais lentos que os monólitos.

**Confiabilidade**

Um monólito consiste em apenas um servidor onde ocorrem todas as chamadas e processos. Se a rede falhar, todo o serviço ficará inativo. Por outro lado, as chamadas de rede de micro-serviços são 99,9% confiáveis. Quando um dos micro-serviços falha, o isolamento de erros permite manter o serviço.

Surge então a questão, de quando devemos usar cada um deles,

Uma das principais vantagens/razões que levam à escolha do monolítico é o tempo de desenvolvimento, assim esta escolha pode justificar-se quando:

- Se pretende o desenvolvimento de uma pequena aplicação.
- A empresa não planeia crescer, não exigindo a conceção e gestão de um sistema complexo.
- A equipa está na fase de prova de conceito, é provável que o produto cresça com o tempo, a iteração rápida é possível com uma arquitetura monolítica.
- Construção de um MVP. Monolíticos são a maneira mais rápida de obter feedback dos primeiros utilizadores neste estágio.

No caso dos micro-serviços, após um aumento significativo na demanda dos clientes, muitas empresas migram para a arquitetura de micro-serviços (p.ex: Amazon, PayPal e Spotify), quando objetivo é construir uma solução em grande escala:

- A arquitetura de micro-serviços requer um planeamento cuidadoso.
- À medida que o projeto cresce, a escalabilidade torna-se mais crítica.
- É importante usar linguagens diferentes para escrever o back-end e o front-end, como C++ para o back-end e Rails para o front-end.

- Devem existir diferentes equipas, independentes, trabalhando nas diferentes funções da sua solução.

Podemos concluir que, quando pretendemos um sistema moderno, escalável, que possa crescer de forma mais consistente, os micro-serviços tornam-se a melhor solução, e isto aplica-se a este projeto.

# 3 Solução proposta

A solução proposta consiste num modelo de negócio baseado nos apresentados anteriormente, mas com uma arquitetura de micro-serviços altamente escalável. O modelo de negócio passa por um serviço de partilha de bicicletas baseado em docas, em que estas são equipadas com dispositivos *IoT* que permitem o desbloqueio remoto das bicicletas. Para interagir com toda a plataforma, é disponibilizada uma aplicação para dispositivos móveis Android e IOS. Através da aplicação o utilizador poderá gerir todo o processo de aluguer, desde consultar as bicicletas disponíveis mais próximas de si, obter detalhes da mesma ou proceder ao *scan* da bicicleta para aluguer. Durante o aluguer, o utilizador enviará dados da viagem para registo. Poderá também consultar o histórico de alugueres, onde encontrará informações acerca percurso percorrido, duração, data de início, valor cobrado e avaliação dada. Além de funcionalidades relativas à gestão de alugueres, o utilizador poderá também fazer a gestão de cartões bancários para pagamento. Alem destas, terá também acesso a uma página de registo e login.

De modo a sustentar esta proposta, é apresentado de seguida, um conjunto de fluxos do funcionamento pretendido para a solução assim como arquitetura correspondente.

Através de uma aplicação móvel o utilizador poderá registar-se e autenticar-se. Após a autenticação, poderá listar as bicicletas disponíveis próximas de si. Para proceder ao aluguer este terá que associar um cartão bancário, caso já possua um cartão associado, este poderá avançar para o aluguer tal como apresentado na Figura 2.

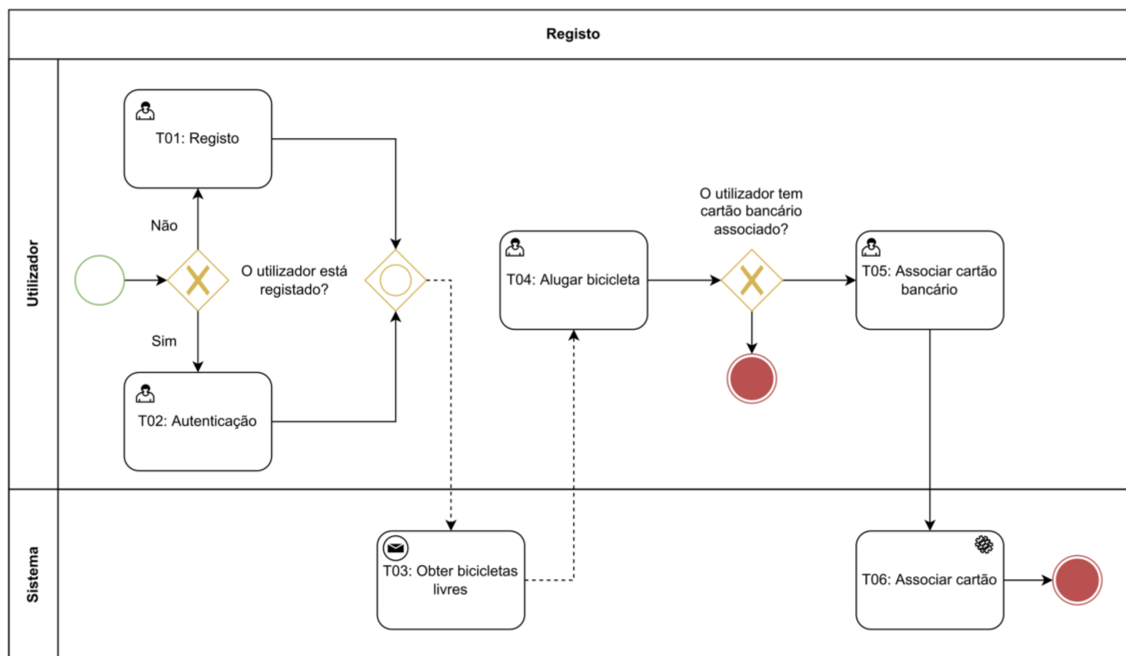



*Figura 2 Fluxo do processo de registo*

O aluguer começa com o desbloqueio da bicicleta através do código QR presente na mesma. Após validação a bicicleta será desbloqueada pela doca, e o utilizador será notificado. Durante a utilização serão recolhidos dados de localização do utilizador, de modo a ter o registo completo de percurso percorrido. A viagem termina quando o utilizador entrega a bicicleta em determinada doca, e será automaticamente gerado e processado o pagamento com base no período de utilização. No final da viagem o utilizador poderá dar uma avaliação, classificando a sua experiência.

A interação com a aplicação no que diz respeito a alugueres, verificação de disponibilidade e pagamentos é feita em tempo real. O utilizador poderá também receber notificações relativas ao estado do pagamento e disponibilidade da bicicleta tal como descrito na Figura 3. [18]

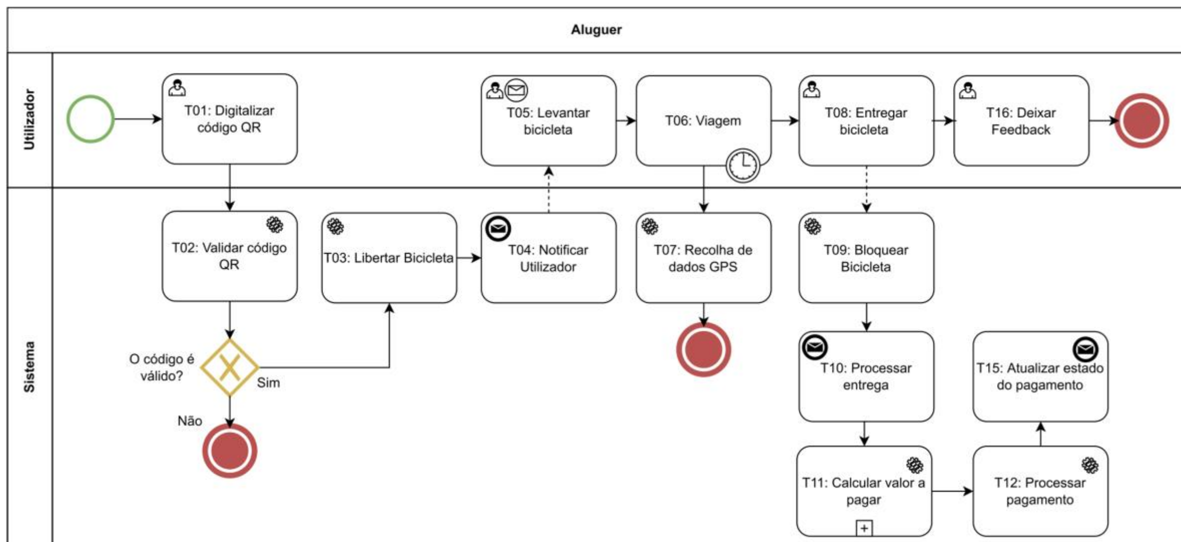



*Figura 3 Fluxo do processo de aluguer de bicicleta*

De modo a sustentar este modelo de negócio com uma gestão de recursos inteligente é proposta uma arquitetura de micro-serviços escaláveis em que o sistema é dividido por domínio [18], e cada domínio pode conter um ou mais serviços, assim como bases de dados exclusivas, esta abordagem, denominada por *domain-driven-design*, é uma abordagem de desenvolvimento de software em que o design é orientado pelo domínio, ou seja, pela área de conhecimento à qual o software se aplica.

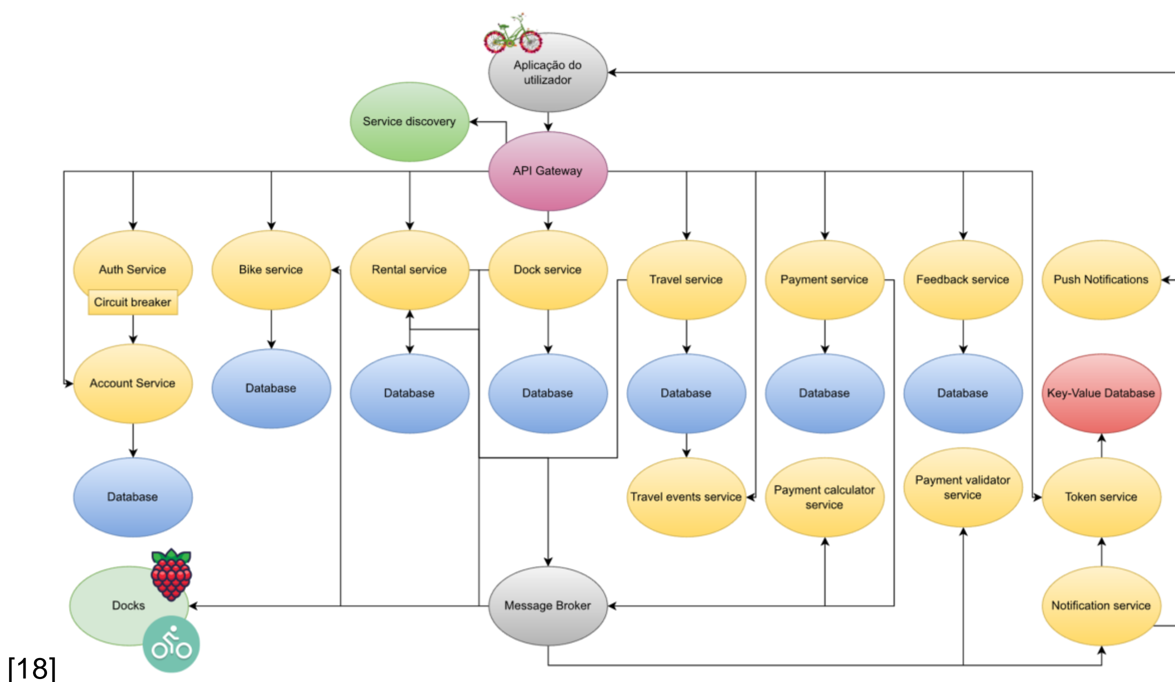
Deste modo, foram identificados os seguintes domínios e correspondentes micro-serviços:

*Tabela 2 - Dominios e serviços propostos*

| **Dominio** | **Micro-serviços** |
|---|---|
| Account | `account-service` |
| Auth | `auth-service` |
| Bike | `bike-service` |
| Dock | `dock-service, dock-internal-service` |
| Rental | `rental-service` |
| Payment | `payment-service, payment-calculator-service, payment-validator-service` |
| Feedback | `feedback-service` |
| Travel | `travel-service, travel-events-service` |

| Notification | `notification-service` |
|---|---|

Na Figura 4 está representada a proposta arquitetural para o sistema pretendido. Através da aplicação móvel o utilizador poderá interagir com toda a plataforma, a aplicação comunica com o *backend* através de um `api-gateway` que ficará responsável por filtrar os pedidos e encaminhá-los para o serviço correto. Existe a necessidade de utilizar também um *service discovery*, e que este funcione em paralelo com o `api-gateway` de forma a mapear todos os serviços e as suas replicas na plataforma. O princípio de armazenamento é que cada serviço terá a sua base de dados e esta não será partilhada com mais serviços.



[18]

*Figura 4 Arquitetura proposta para o serviço de partilha de bicicletas*

O serviço `account-service` contém a gestão de contas de utilizadores recorrendo a uma base de dados relacional, enquanto, o `auth-service` é responsável pela autenticação e gestão de *tokens*, comunicando diretamente com o `account-service`, por REST, para obter informações do utilizador. Caso a autenticação seja bem-sucedida, é retornado um *token JWT* que permite fazer pedidos autenticados aos restantes serviços. O *token* gerado por este serviço é necessário para comunicar com todos os serviços que não sejam `account-service` e `auth-service`. Os *tokens* inválidos serão barrados pelo `api-gateway` e não será possível fazer pedidos a outros serviços. Este serviço usa um *circuit breaker*, devido a comunicar diretamente com outro por REST, então no caso de haver alguma

falha no outro serviço, o *circuit breaker* tem a responsabilidade de enviar uma resposta predefinida.

Na gestão de bicicletas temos o `bike-service`, que contém toda a informação sobre as mesmas. Comunica assincronamente com o *message broker*, onde recebe pedidos de validação de bicicletas.

O `rental-service` é o ponto de entrada para um aluguer. Além disso, recebe os pedidos de aluguer, armazena o seu histórico numa base de dados, e comunica assincronamente, através do *message broker* com outros serviços a fim de manter atualizado em tempo real o estado do aluguer.

O `dock-service` é responsável pela gestão de docas, contém a informação relativa às docas existentes assim como às bicicletas a elas conectadas, sendo essa informação guardada numa base de dados. Comunica assincronamente com o *message broker* a fim de atualizar em tempo real o estado das docas.

No que diz respeito às viagens, o `travel-service` permite aceder aos eventos de uma viagem associada a um determinado aluguer, este serviço acede, apenas com permissões de leitura à base de dados utilizada pelo `travel-events-service`. Recebe os eventos de determinada viagem, através da aplicação do utilizador, e tem a responsabilidade de os colocar no *message broker* para posteriormente serem processados. O `travel-events-service` tem a responsabilidade de processar os eventos de determinada viagem, colocados no *message broker* pelo `travel-service`, como o *message broker* tem o formato FIFO, os eventos serão processados por ordem de chegada numa base de dados *NoSql*.

O `payment-service` contém as operações de gestão de pagamentos, e utiliza uma base de dados para armazenar toda a informação, comunica através do *message broker* com outros serviços de pagamento a fim de validar e processar o pagamento. O `payment-calculator-service` comunica exclusivamente através do *message broker* e tem como responsabilidade calcular o valor a pagar pelo utilizador, após este terminar a sua viagem, com base no tempo decorrido. Tendo o pagamento sido calculado e efetuado pelo utilizador, é encaminhado um pedido para o `payment-validator-service`, que funciona como uma emulação do que seria um serviço *3rd-party*, que teria a responsabilidade de validar o pagamento e retirar a quantia a pagar do cartão bancário associado. Este comunica exclusivamente através do *message broker* informando o resultado da operação de pagamento.

O `feedback-service` tem como responsabilidade a gestão de avaliações por parte dos utilizadores, armazenando todas as avaliações numa base de dados.

O `token-service` armazena os *tokens* associados ao utilizador que são necessários para comunicação com o mesmo, através da aplicação móvel. Para tal usa uma base de dados chave-valor *NoSql*, este é relevante para o `notification-service` que permite aos restantes serviços da plataforma enviarem notificações ao utilizador, subscrevendo um tópico no *message broker* onde os outros serviços enviarão as mensagens que pretendem que sejam notificadas ao utilizador. O `notification-service` tem a responsabilidade de receber essas mensagens, obter os *tokens* necessários e enviar através dum serviço *3rd-party* estas notificações diretamente para a aplicação móvel do utilizador.

O *message broker* consiste num serviço de terceiros que age como uma fila de mensagens e permite publicar e subscrever mensagens em vários tópicos, permitindo assim, que todos os serviços comuniquem através dele em diferentes *endpoints*.

# 4 Desenvolvimento

Para o desenvolvimento do projeto proposto, foram utilizadas as mais variadas tecnologias e padrões de software, seguindo as boas práticas de desenvolvimento e arquitetura. Nos subcapítulos que se seguem, são abordadas as tecnologias utilizadas para o desenvolvimento da plataforma, assim como os padrões e arquiteturas de software implementados. São também abordados os contextos limitados identificados na arquitetura, e é apresentado um diagrama arquitetural do sistema desenvolvido como um todo. Segue-se uma explicação mais elaborada dos processos e comunicação dos micro-serviços, abordando a forma como estes serviços se comportam e interagem com outros serviços ou micro-serviços, procurando detalhar as funcionalidades de cada micro-serviço. Surge também um subcapítulo onde é abordada o tipo de escalabilidade implementada, com uma breve explicação do seu modo de funcionamento e vantagens. O modelo de persistência de dados escolhido, assim como as ferramentas utilizadas, é abordado no subcapítulo seguinte, justificando o porquê da escolha de uma base de dados NoSQL para este projeto. Segue-se uma descrição da comunicação assíncrona implementada, explicando a escolha do *RabbitMQ* e o uso funcionalidades do mesmo. É também dada uma breve explicação acerca da infraestrutura projetada, e sobre que bases esta assenta, abordando as ferramentas e hardware utilizado. Tendo sido construído um dispositivo IoT para emular o funcionamento das docas, é também apresentada a sua arquitetura, hardware e modo de funcionamento, explicando como funciona a interação entre a plataforma e este dispositivo. É apresentada a aplicação móvel desenvolvida, mostrando em detalhe, todas as funcionalidades que esta possui, e como interage com os micro-serviços, explicando também o tipo de tecnologias e padrões utilizados no desenvolvimento da mesma, e as suas vantagens. Por fim é apresentado um subcapítulo sobre testes, onde são abordados todos os testes implementados neste projeto, nomeadamente unitários e de integração, assim como a automatização de *pipelines* de *continuous integration*, justificando que tipo de testes foram feitos e que desafios apresentam numa arquitetura de micro-serviços.

## 4.1 Tecnologias utilizadas

As tecnologias utilizadas têm especial importância no desenvolvimento de uma arquitetura de micro-serviços, isto porque, quando se pretende construir uma plataforma desacoplada, escalável, tolerante a falhas, também as tecnologias associadas o devem ser, permitindo assim projetar um sistema mais robusto.

Neste subcapítulo é dada uma breve explicação das tecnologias utilizadas, permitindo assim contextualizar e justificar o uso das mesmas nos subcapítulos que se seguem.

### 4.1.1 Framework .NET

O .NET é uma plataforma de desenvolvimento *open-source*, criada pela Microsoft, que é composta por uma biblioteca padrão, um compilador e uma máquina virtual. Permite a construção e a execução de aplicações para Windows e Web, incluindo Web Services.

Podemos descrever o .Net também como uma infraestrutura de programação disponibilizada para vários sistemas operativos. Inclui uma grande biblioteca de soluções pré-codificadas para lidar com problemas de programação comuns e uma "máquina virtual" que gere a execução dos programas escritos para a infraestrutura. A biblioteca de soluções contém uma série de funcionalidades padrão, permitindo que o programador se concentre no objetivo final do seu programa. A máquina virtual torna o programa independente do computador onde será executado, permitindo que seja usado em diferentes máquinas sem necessidade de alterações.

Os componentes da plataforma possibilitam a criação de códigos em algumas linguagens, como C#, VB.NET e F#, a plataforma foi pensada para utilizar as linguagens e tecnologias da Microsoft, sem precisar usar bibliotecas diferentes para cada uma delas.

### 4.1.2 MassTransit

O MassTransit é uma estrutura de aplicação distribuída de código aberto para .NET que facilita a criação de aplicações e serviços que potencializam a comunicação assíncrona, baseada em mensagens e fracamente acoplada para maior disponibilidade, confiabilidade e escalabilidade.

### 4.1.3 RabbitMQ

O RabbitMQ é um servidor de mensagens que utiliza um protocolo denominado *Advanced Message Queuing Protocol* (AMQP). Permite lidar com o tráfego de mensagens de forma rápida e confiável, além de ser compatível com diversas linguagens de programação, possuir interface de administração nativa e ser multiplataforma.

Dentre as aplicabilidades do RabbitMQ estão a garantia de comunicação assíncrona entre aplicações, diminuir o acoplamento entre aplicações, distribuir alertas e controlar a fila de trabalhos em *background*.

### 4.1.4 MongoDB

O MongoDB é uma base de dados NoSQL orientada a documentos no formato JSON, este não possui como restrição a necessidade de ter as tabelas e colunas criadas previamente, permitindo que um documento represente toda a informação necessária, com todos os dados pretendidos, no formato de um JSON.

Num documento do MongoDB podem existir valores simples, como números, *strings* e datas, assim como também podem existir listas de valores e listas de objetos.

Os documentos são agrupados em coleções. E um conjunto de coleções forma um base de dados. O MongoDB permite que a sua base de dados seja replicada para outros servidores, aumentando assim a disponibilidade das suas informações, sendo esse recurso conhecido por *replica set*. Dessa forma, cada servidor terá uma cópia dos dados.

### 4.1.5 Eureka

As arquiteturas de micro-serviços levantam alguns desafios que devem ser considerados. Por exemplo, se um determinado serviço depende da chamada de outro, basta utilizar um RestTemplate, passando por parâmetros o *host* e a porta e enviar uma requisição via REST. No entanto, se o serviço que recebe a chamada tiver sido escalado horizontalmente com várias instâncias, cada uma com sua porta e *host* específicos seria muito difícil manter os endereços individuais de cada um. Para resolver esse problema, em casos de aplicações escaláveis e distribuídas, podem utilizar-se recursos que registam e descobrem serviços, de modo a controlar toda a dinâmica da escalabilidade.

O Netflix Eureka é um módulo do Netflix OSS, que permite que os serviços sejam registados através do Eureka Server e descobertos através do *Eureka client.*

### 4.1.6 Kubernetes

O Kubernetes(k8s) é uma plataforma open source utilizada para orquestrar e gerir clusters de containers. Permite eliminar a maior parte dos processos manuais necessários para fazer *deploy* e escalar aplicações ou serviços em containers, além de fazer a gestão desses clusters com facilidade. O Kubernetes permite a comunicação tanto de elementos físicos como elementos virtuais de forma transparente. Cada um destes grupos compostos por elementos físicos e virtuais é chamado de cluster. Diferentes clusters podem comunicar-se através de uma rede criada pelo Kubernetes para essa finalidade. Os componentes dentro de um cluster podem assumir uma de duas responsabilidades possíveis: podem fazer o papel de master ou o papel de node. Os componentes que assumem o papel de server são responsáveis por estabelecer a comunicação com outros clusters, expor as APIs do Kubernetes com relação

ao cluster em questão, realizar verificações de estabilidade dos serviços expostos e realizar operações conhecidas como *scheduling*, que é basicamente uma operação de atribuição de serviços aos recursos disponíveis dentro do cluster. Os nodes ficam responsáveis por realizar os trabalhos designados pelas aplicações que neles são executadas, podem utilizar recursos dentro do próprio cluster ou presentes noutros clusters, já que cada cluster se pode comunicar com outros pela rede que é criada pelo Kubernetes e gerida pelos componentes *master*. O Kubernetes faz a gestão dos nodes através de containers, como o Docker. O master, dentro de cada cluster, fica responsável por repassar as instruções a serem executadas nos nodes, que podem criar e destruir containers.

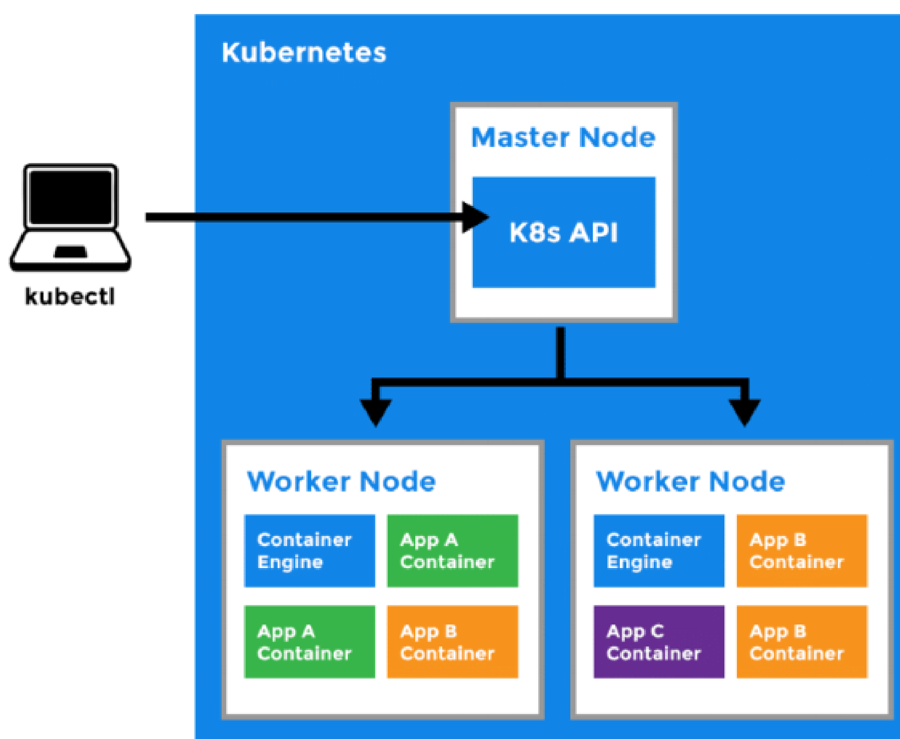



*Figura 5 Diagrama de funcionamento do kubernetes*

Na Figura 5 podemos ver um diagrama que representa um funcionamento básico do Kubernetes, em que através da ferramenta kubectl (ferramenta disponibilizada pelo Kubernetes para fazer a gestão da sua API através da linha de comandos), podemos utilizar a sua API, para criar nodes, e dentro desses nodes, arrancar containers com os mais variados serviços, sendo estes partilhados através da rede interna do Kubernetes.

### 4.1.7 Etcd

Etcd é uma base de dados *NoSQL* de *key-value* distribuída *open-source*. Serve como a espinha dorsal de muitos sistemas distribuídos, fornecendo uma maneira confiável de

armazenar dados. Ao contrário de algumas bases de dados *key-value*, o *etcd* garante a persistência dos dados, não funcionando apenas como um mecanismo de cache.

### 4.1.8 Flutter

O flutter é um kit de ferramentas da Google para construir aplicações nativamente compiladas para mobile, web ou desktop a partir de uma única base de código.

De forma resumida, o flutter permite escrever o código apenas 1 vez e executá-lo em até 4 plataformas diferentes.

### 4.1.9 Google Firebase

*Firebase* é uma plataforma de desenvolvimento mobile (e web) adquirida pela Google em 2014. Com foco de ser um *backend* completo e de fácil usabilidade, esta ferramenta disponibiliza diversos serviços diferentes que auxiliam no desenvolvimento e gestão de aplicações.

Entre as muitas funcionalidades oferecidas por esta plataforma, é de destacar o sistema de notificações *push*, que permite integrar o *firebase* com a aplicação mobile e enviar notificações para a mesma em tempo real.

### 4.1.10 Ocelot

O API Gateway funciona como uma porta de entrada para os clientes dos micro-serviços do sistema. Em vez de invocar diretamente os micro-serviços, os clientes chamam a API Gateway, que redireciona o pedido para o micro-serviço apropriado e redireciona de volta a resposta para o cliente.

Ou seja, o *gateway* funciona como uma camada intermediária entre os clientes e os micro-serviços.

O Ocelot é uma biblioteca que permite criar um API Gateway com o ASP.NET. Possuindo uma grande gama de funcionalidades, como agregação de pedidos, roteamento, autenticação, cache, *load balancing*, *logs*, entre outros.

Apesar de ter sido concebido para aplicações .NET que implementam uma arquitetura de micro-serviços, o Ocelot pode ser utilizado como API Gateway de qualquer tipo de sistema que implemente esta arquitetura.

### 4.1.11 MQTT

O MQTT é um protocolo de comunicação com baixos requisitos ao nível da largura de banda e também ao nível de hardware, sendo extremamente simples e leve. Este protocolo foi desenvolvido pela IBM e pela Eurotech e tem como finalidade comunicar através de redes com pouca largura de banda, com muita latência e, nesse sentido, pouco confiáveis.

Para isto o protocolo foi desenvolvido com recurso a vários conceitos que garantem uma elevada taxa de entrega de mensagens.

### 4.1.12 Mosquitto

O Mosquitto é um *broker MQTT open source*, que pode ser utilizado desde computadores de placa única até servidores.

O Mosquitto implementa o modelo *publilsher/subscriber* e pode ser utilizado em aplicações de IoT, as quais fazem uso de sensores de baixa potência, atuadores, dispositivos móveis, microcontroladores e outros dispositivos programáveis.

O Mosquitto oferece comandos de linha como *mosquitto_pub* e *mosquitto_sub* para publicar e subscrever no *broker*, respetivamente, além de bibliotecas em C para implementação de cliente *MQTT*.

### 4.1.13 Polly

O *Polly* é uma biblioteca *dotnet*, de resiliência e tratamento de falhas transitórias que permite aos desenvolvedores implementar políticas como *retry*, *circuit breaker*, *timeout*, *bulkhead isolation*, *rate-limiting* e *fallback* de uma maneira fluente e *thread-safe.*

Neste projeto, o *Polly* foi usado para *circuit breaker*, permitindo que os serviços sejam tolerantes a falhas.

## 4.2 Padrões arquiteturais implementados

Para produzir software de qualidade, é fundamental seguir boas práticas, é neste ponto que entram os padrões de software, que não são mais que a resolução de determinado problema que ocorre com frequência num determinado contexto no desenvolvimento de software. Há medida que a tecnologia evolui, vão surgindo novos padrões que resolvem novos problemas, e numa arquitetura de micro-serviços, existem também padrões de software que resolvem problemas inerente à implementação deste tipo de arquitetura.

Na implementação deste projeto, foram seguidos certos padrões e arquiteturas de software, este capítulo visa dar uma breve explicação acerca dos mesmos, para que posteriormente fique claro o porquê da sua escolha na implementação. São abordados padrões de comunicação entre serviços, padrões de desenvolvimento e padrões de micro-serviços.

### 4.2.1 Saga

Numa arquitetura de micro-serviços encontramos diversos desafios, como lidar com transações distribuídas.

Num ambiente isolado em que pretendemos que vários serviços comunicam entre si, num fluxo de entrega, precisamos de garantir a atomicidade e consistência de dados dos diversos repositórios.

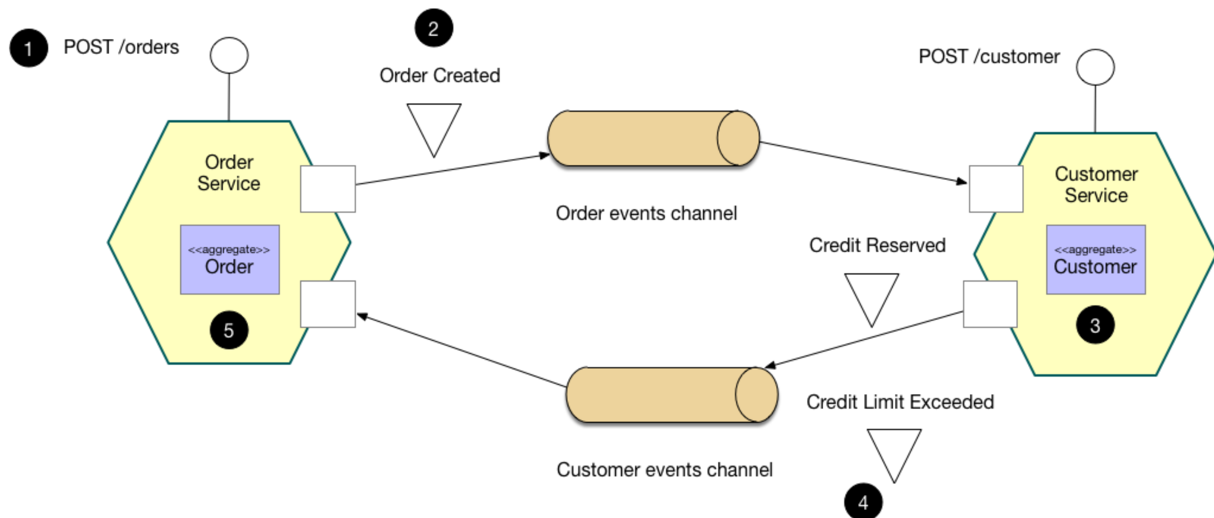



*Figura 6 Saga baseada em coreografia*

O padrão SAGA permite-nos resolver este problema, conta com duas técnicas de utilização: Coreografia e Orquestração. Na coreografia, cada nó do Saga sabe qual o próximo nó a ser chamado, seja seguindo na ação ou na compensação. Na Figura 6 está representado um

exemplo de SAGA baseado em coreografia, através da comunicação entre dois serviços num processo de compra [19].

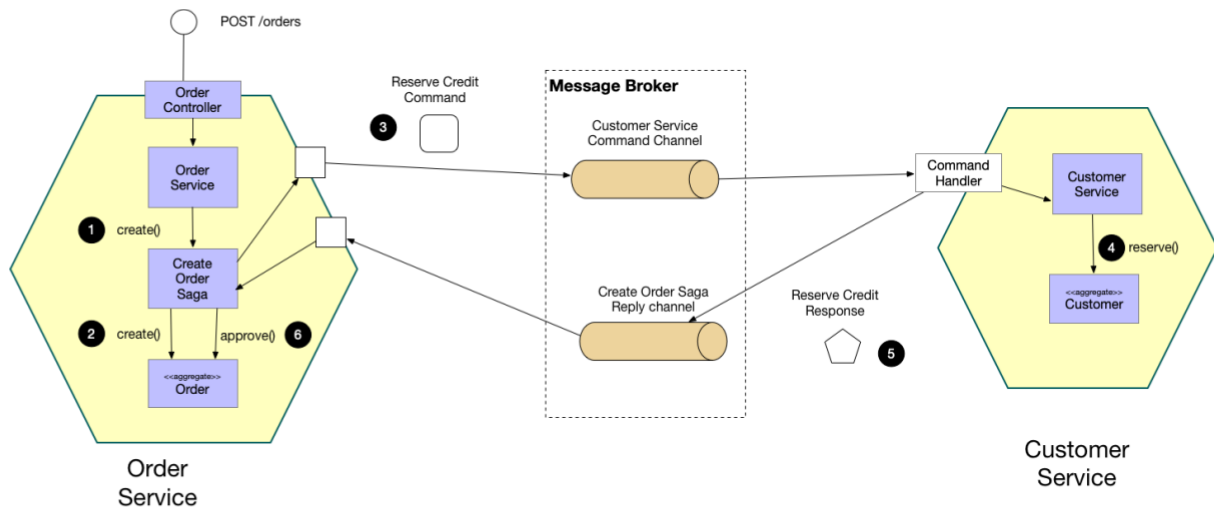



*Figura 7 Saga baseada em orquestração*

Já na orquestração, tal como podemos verificar na Figura 7, existe uma entidade, o Orquestrador, que é responsável por saber qual o próximo passo a ser executado.

Cada uma das duas abordagens tem suas vantagens e desvantagens. A cedência passa por substituir um ponto único de falha (orquestrador) por uma infraestrutura mais pesada no caso da coreografia.

Neste projeto, foi usado o SAGA baseado em orquestração, pois este permite uma infraestrutura menos dispendiosa.

### 4.2.2 Event sourcing

Um serviço normalmente precisa de atualizar a base de dados e enviar mensagens/eventos. Por exemplo, um serviço que participa numa SAGA precisa de atualizar atomicamente a base de dados e enviar mensagens/eventos. Da mesma forma, um serviço que publica um evento de domínio deve atualizar atomicamente um agregado e publicar um evento.

Surge então a questão de como atualizar de forma confiável/atômica a base de dados e enviar mensagens/eventos. Uma boa solução para esse problema é usar *event sourcing*, o fornecimento de eventos mantém o estado de uma entidade, como uma sequência de eventos de mudança de estado. Sempre que o estado de uma entidade muda, um novo evento é anexado à lista de eventos, funcionando assim de forma atómica, ou seja, o estado do evento anterior não se altera, é adicionado um novo evento à lista. O serviço reconstrói o estado atual de uma entidade reproduzindo os eventos.

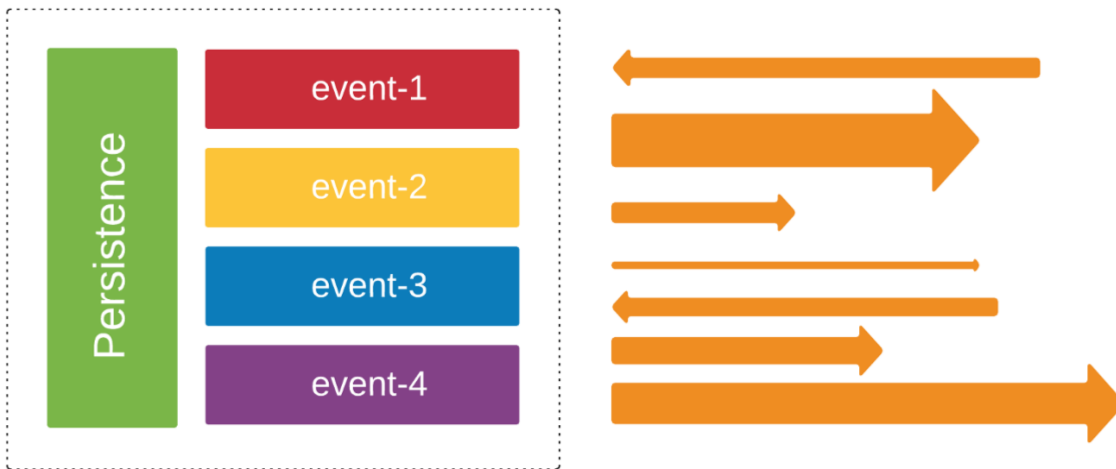



*Figura 8 - Arquitetura event sourcing*

Na Figura 8 está representada a arquitetura de *event sourcing*, os serviços persistem eventos num *event store*, que é uma base de dados de eventos. O *event store* também se comporta como um agente de mensagens, fornece uma API que permite que os serviços subscrevam eventos. Quando um serviço guarda um evento no *event store*, este é entregue a todos os subscritores [20].

### 4.2.3 Database per Service

Ao implementar uma arquitetura de micro-serviços, surge a questão, acerca de qual arquitetura de base de dados utilizar, isto porque se todos os serviços utilizarem a mesma base de dados, estes perdem muitas das suas vantagens de desacoplamento, *deploy* ou escalabilidade.

Existe também a questão de que os serviços têm diferentes requisitos de armazenamento, em alguns é mais vantajoso usar uma base de dados relacional, noutros é mais simples usar

uma base de dados não relacional, e outros até podem necessitar de bases de dados *key-value* ou orientadas a grafos.

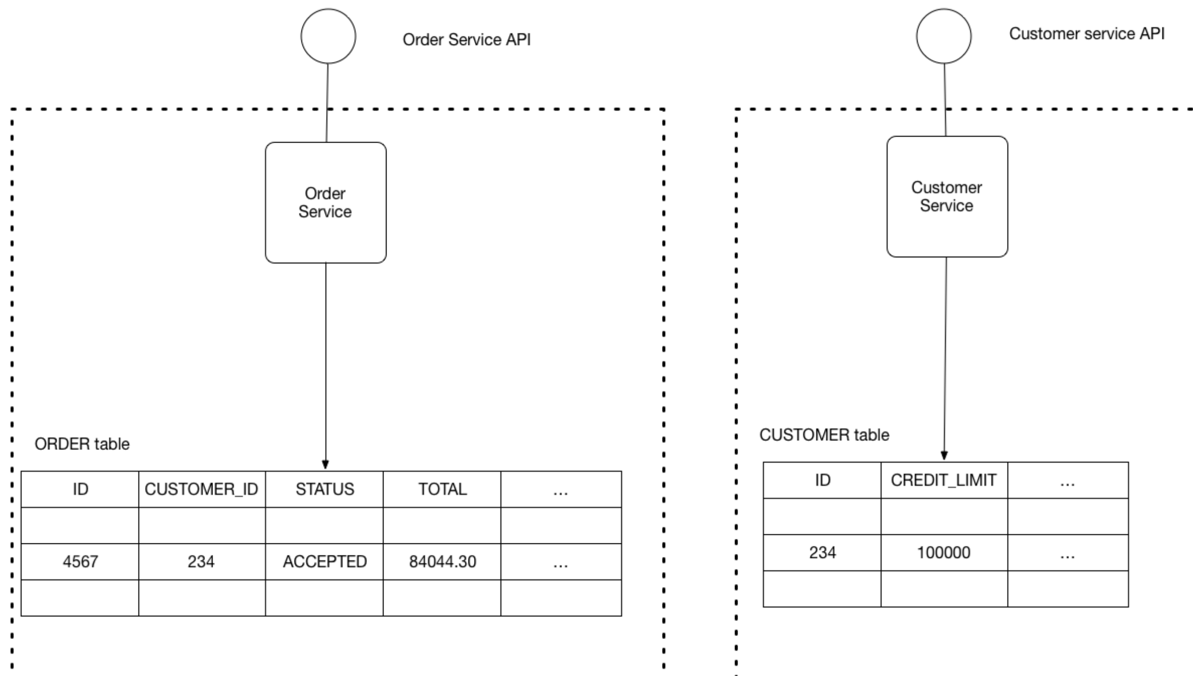



*Figura 9 Exemplo da arquitetura database per service*

Para resolver este problema, poderemos utilizar o padrão *database per service* que consiste em cada serviço ter a sua base de dados (Figura 9), que permite que cada serviço controle totalmente os seus dados, mantendo-os privados, podendo permitir o seu acesso apenas por *APIs* [21].

## 4.2.4 API gateway

Quando implementamos arquiteturas micro-serviços, deparamo-nos com o problema de como os clientes de uma aplicação comunicam com cada micro-serviço.

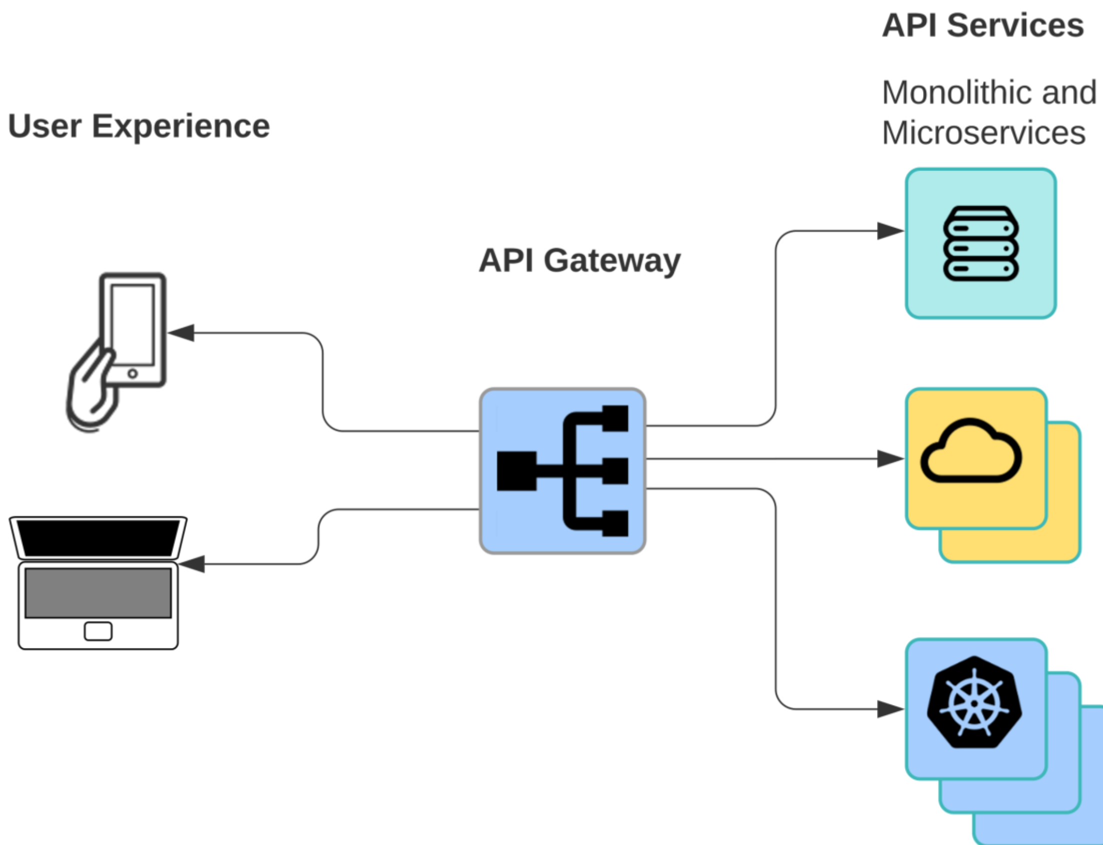



*Figura 10 - Arquitetura API gateway*

O *API gateway* surge então como uma solução para este problema, funcionando como único ponto de entrada para todos os clientes, redirecionando os pedidos para um ou mais serviços correspondentes (Figura 10).

Em vez de fornecer uma única API, o *API gateway* pode expor uma API diferente para cada cliente. O *API gateway* também pode adicionar camadas de segurança, por exemplo, verificar se o cliente está autorizado a realizar o pedido [22].

### 4.2.5 Circuit breaker

O *circuit breaker* permite evitar que uma falha de rede ou serviço se espalhe para outros serviços. Um serviço deve invocar outro serviço remoto por meio de um proxy que funcione de maneira semelhante a um disjuntor elétrico. Quando o número de falhas consecutivas ultrapassa um limite, o disjuntor desarma e, durante um período de tempo limite, todas as tentativas de chamar o serviço remoto falharão imediatamente, após o tempo limite expirar, o disjuntor permite a passagem de um número limitado de pedidos de teste, se estes pedidos forem bem-sucedidos, o disjuntor retomará a operação normal, caso contrário, se houver uma falha, o período de tempo limite será reiniciado [22].

A principal vantagem de usar um *circuit breaker* consiste em o serviço lidar com a falha dos serviços que ele invoca. No entanto, tem como desvantagem, o desafio de escolher valores de tempo limite sem criar falsos positivos ou introduzir latência excessiva.

### 4.2.6 Service instance per container

O padrão *service intance per container* consiste em fazer *deploy* de cada serviço, empacotando-o numa imagem de container (ex: Docker).

Este padrão apresenta vantagens tais como:

- Simplicidade e facilidade de aumentar ou reduzir o número de instâncias de determinado serviço, aumentando e reduzindo o seu número de containers.
- Encapsular os detalhes da tecnologia usada para construir o serviço, permitindo que todos os serviços sejam, por exemplo, iniciados e interrompidos exatamente da mesma maneira.
- Um container permite impor limites de CPU e memória consumida por uma instância de serviço.
- Os *containers* são extremamente rápidos para construir e iniciar.

De entre as desvantagens, destaca-se o facto de a infraestrutura para *deploy* de containers não ser tão rica quanto a infraestrutura para *deploy* de máquinas virtuais.

### 4.2.7 Service Registry

Numa arquitetura de micro-serviços, surge a necessidade de identificar quais são as instâncias de determinado serviço disponíveis e a sua localização, para tal, existe o *service registry*. Este consiste numa base de dados de serviços, das suas instâncias e localizações. As instâncias do serviço são registradas com o registro de serviço na inicialização e canceladas quando desligadas, permitindo que outros serviços e clientes consultem o registro do serviço para encontrar as instâncias disponíveis de determinado serviço [23].

#### *4.2.7.1 Self registration*

Surge então a questão, de como as instancias de um serviço se registam ou desligam do *service registry. Self registration* é um dos padrões usados para resolver este problema, consiste em atribuir a responsabilidade de registo e término a cada instância de serviço. Na inicialização, a instância de serviço regista-se (host e endereço IP) no registro de serviço e torna-se disponível para descoberta. O cliente *service registry* normalmente deve renovar o seu registro periodicamente para que o registro saiba que ainda está ativo. Ao desligar, a instância de serviço cancela o registro do registro de serviço.

### 4.2.8 Access token

Numa arquitetura de micro-serviços, para verificar a identidade de um cliente que efetua um pedido a um serviço, pode usar-se um *access token*. O API Gateway autêntica o pedido e passa um token de acesso (por exemplo, JWT) que identifica com segurança quem efetuou o pedido em cada chamada aos serviços. Um serviço pode incluir o token de acesso nos pedidos feitos a outros serviços [24].

O uso de um *access token* permite que a identidade do autor do pedido seja transmitida com segurança pelo sistema e que os serviços possam verificar se o autor está autorizado a realizar uma operação.

### 4.2.9 Health check API

O *health check API* surge para detectar se uma instância de um serviço em execução consegue lidar com pedidos.

Para implementar este padrão, um serviço deve conter *endpoints* na sua API de verificação de integridade (por exemplo, HTTP /health) que retorna informação relativa à integridade do serviço. Desta forma, podem ser efetuadas várias verificações, por exemplo pelo *service registry*, para auferir o status das instâncias dos serviços que se encontram registados [25].

Um cliente de *health checks*, um serviço de monitorização, *service registry* ou *load balancer,* invoca periodicamente a API de *health check* dos serviços para verificar a integridade das suas instâncias.

## 4.3 Contextos limitados

O objetivo de qualquer software passa por ajudar os seus utilizadores a resolverem os problemas relacionados com um determinado domínio. Os Contextos Limitados são um padrão central em *Domain Driven Design* (DDD) e o foco do design estratégico perante modelos e equipas com uma dimensão considerável.

O desenvolvimento de uma nova aplicação sob o paradigma de DDD implica a definição do seu próprio modelo e contexto. O contexto de um modelo é um conjunto de condições que necessitam de ser aplicadas para garantir que os termos usados na definição do modelo tenham um significado específico; sejam ubíquos. A ideia principal é definir o seu escopo, traçando os limites do seu contexto, fazendo os possíveis para manter o modelo unificado [18].

Um contexto limitado fornece o quadro lógico dentro do qual o modelo evolui. Quando há vários modelos, é necessário definir as fronteiras e as relações entre eles. No entanto, cada um tem o seu próprio contexto limitado.

Não é, porém, suficiente ter modelos unificados separados. Estes devem ser integrados, porque a funcionalidade de cada um é apenas uma parte de todo o sistema. No final, as peças devem ser montadas juntas, e todo o sistema deve funcionar corretamente.

Cada contexto limitado deve ter um nome que deve fazer parte de uma linguagem ubíqua. Todos devem conhecer os limites de cada contexto e o mapeamento entre contextos e código. Uma prática comum é definir os contextos, criar os módulos para cada contexto e usar uma convenção de nomenclatura para indicar o contexto ao qual cada módulo pertence.

Assim, foram identificados os seguintes contextos limitados para o caso em estudo:

- Account – Engloba a gestão de utilizadores.
- Auth –Autenticação de utilizadores e serviços.
- Bike – Engloba toda a gestão de bicicletas, assim como o seu estado.
- Dock – Gestão de docas.
- Rental – Gestão e processamento de alugueres.
- Payment – Gestão e processamento de pagamentos.
- Feedback – Avaliações por parte dos utilizadores.
- Travel – Gestão de viagens.
- Notification – Processamento e envio de notificações.

Cada micro-serviço implementa o contexto limitado com que se relaciona e é totalmente independente, sendo também capaz de evoluir o seu domínio, introduzindo novos conceitos ou comportamentos de negócio relacionados com o contexto em que está inserido.

Para possíveis novos requisitos de negócio, cada contexto limitado pode ter mais micro-serviços implementados. O desenvolvimento desses serviços em cada contexto limitado pode ser avaliado tendo em consideração os conceitos e regras de domínio que são necessários para atingir os requisitos a serem implementados.

## 4.4 Arquitetura

Este capítulo apresenta de forma geral a arquitetura, como os serviços se interligam e as relações entre eles.

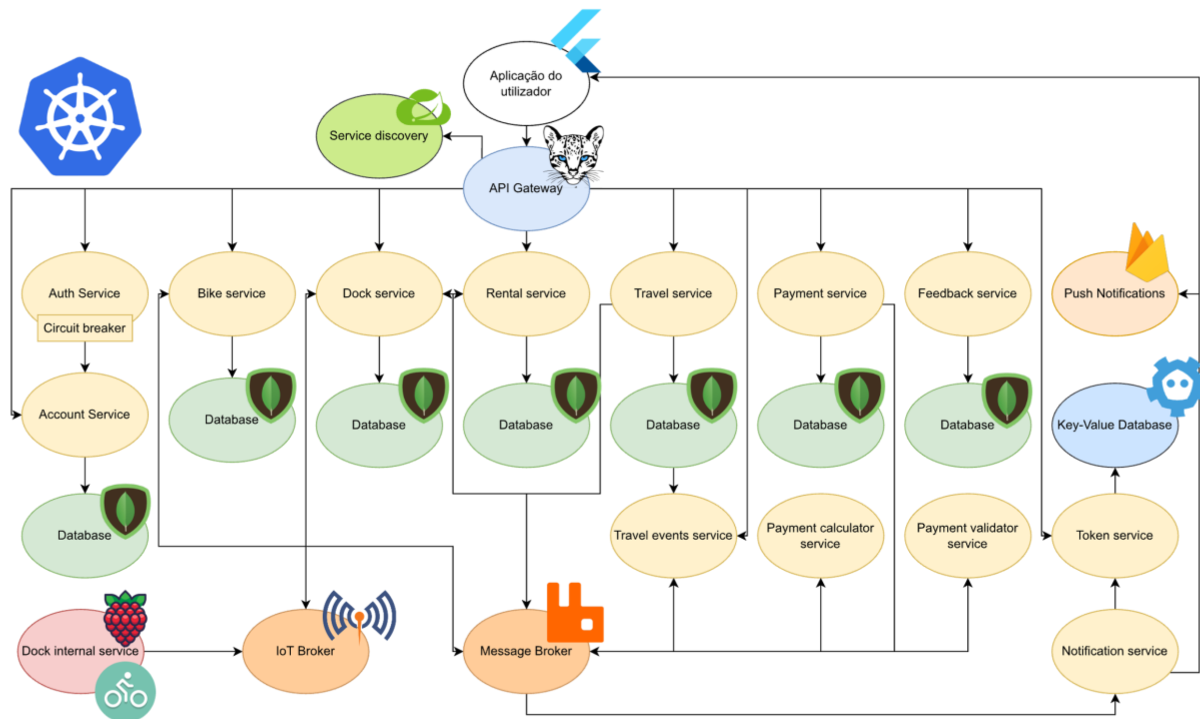



*Figura 11 - Arquitetura da plataforma de partilha de bicicletas*

A Figura 11 expõe a arquitetura implementada com todos os serviços e tecnologias associadas. A aplicação móvel, contem todo o *frontend* desta arquitetura, pelo que todos os pedidos são efetuados através dela, trata-se de uma aplicação desenvolvida em *flutter* compatível com dispositivos android e IOS.

O ponto de entrada da comunicação *backend* fica no `api-gateway`, a ferramenta utilizada foi o `ocelot`. Este foi configurado com todos os *endpoints* que estão expostos para o exterior, ou seja, cada pedido REST efetuado pela aplicação, é feito ao `api-gateway` que se encarregará de encaminhar o serviço para o respetivo serviço. O `api-gateway` permite configurar todos os pedidos permitidos, ou seja, embora os micro-serviços possam expor várias *APIs* entre eles, serviços externos, tais como a aplicação móvel, só terão acesso a *endpoints* definidos pelo `api-gateway`. O api-`gateway` efetua também a validação dos *tokens* para pedidos autenticados, ou seja, quando o utilizador se autêntica na aplicação, através do `auth-service`, é gerado um *token*, que possui uma chave partilhada com o `api-gateway`, após estar autenticado, sempre que o utilizador fizer um pedido terá de enviar o *token* que recebeu, e o `api-gateway` irá verificar se este é valido, para que possa posteriormente encaminhar o seu pedido para o serviço. A validação de tokens é concretizada diretamente no `api-gateway`, os serviços não têm a necessidade de fazer essa verificação, até porque não estão expostos para o exterior, e todos os pedidos externos lhe chegam através do próprio `api-gateway`.

Para auferir qual o serviço para o qual o `api-gateway` deve redirecionar o pedido, este precisa de conhecer o seu endereço, e como os serviços podem escalar verticalmente, utilizar endereços estáticos não permitira o correto funcionamento do sistema, para tal usa-se um `service-discovery`, onde os serviços se inscrevem quando iniciam, a partir desse momento o `service-discovery` sabe quais são os serviços em funcionamento e o seu endereço, assim, a fim de saber para onde redirecionar o pedido, o *gateway* comunica diretamente com o `service-discovery`, que lhe indica para onde deve fazer esse redireccionamento. A ferramenta usada para este propósito é o Eureka.

Na generalidade todos os micro-serviços comunicam entre si de forma assíncrona através do `message-broker` tendo a ferramenta escolhida sido o *RabbitMQ*, pela sua flexibilidade e pelo facto de oferecer uma variedade de funcionalidades. A comunicação entre o serviço interno da doca e o micro-serviço que faz a gestão das docas (`dock-service`) é feita assincronamente com recurso ao *Mosquitto*, um *broker* mais simples, pensado para dispositivos IoT. Existem 2 casos em que os serviços comunicam assincronamente através da arquitetura REST, sendo estes os serviços de notificação e autenticação. No caso do `auth-service`, este comunica sincronamente com o `user-service` a fim de obter os detalhes do utilizador para validação do login, já o `notification-service`, invoca sincronamente o `token-service`, para obter o *token* relativo à aplicação do utilizador.

A nível de armazenamento, a base de dados de eleição foi o *MongoDB*, uma base de dados *NoSQL* orientada a documentos. Devido aos serviços estarem distribuídos por contextos, este projeto não gerou grandes relações entre tabelas, acabando por não se justificar o uso de bases de dados relacionais. Assim foi escolhido o *MongoDB* por ser uma base de dados simples de utilizar e manipular, com funcionalidades interessantes, nomeadamente a pesquisa geo espacial, usada para obter as docas mais próximas de certas coordenadas GPS num determinado raio. Num projeto deste calibre, com uma quantidade considerável de serviços, o *MongoDB*, sendo uma base de dados *NoSQL* torna-se também muito vantajoso no que diz respeito à velocidade de desenvolvimento, oferecendo flexibilidade nos dados armazenamos, permite que o esquema de dados se vá alterando sem que os dados tenham que ser reconfigurados.

Foi necessária a utilização de uma base de dados *key-value*, que oferecesse uma persistência de dados eficaz, para armazenamento dos *tokens* da aplicação dos utilizadores, para que estes possam receber notificações, neste caso o *MongoDB* não se torna vantajoso, sendo que um dicionário representa a arquitetura de dados ideal para este problema. Foi então escolhido o *etcd*, uma base de dados persistente NoSQL, que funciona por *key-value*,

basicamente um dicionário com persistência, oferecendo inúmeras *APIs* para as mais variadas linguagens.

Espera-se que as docas sejam geridas por um pequeno dispositivo *IoT*, que fará a gestão de todas as docas presentes em determinada localização. Este dispositivo é um *Raspberry Pi*, que tem a responsabilidade de bloquear e desbloquear as docas, comunicando com os serviços do sistema de forma assíncrona, assim as instruções de desbloqueio são recebidas em filas FIFO, e os clientes são atendidos por ordem de chegada. Para a identificação da bicicleta foi utilizado o protocolo NFC.

A nível de notificações, a aplicação móvel está integrada com o *google firebase*, que permite uma gestão eficiente das notificações, permitindo enviar notificações para a aplicação, mesmo que esta esteja fechada. Assim os serviços apenas terão de saber o que precisam de notificar, toda a integração de notificações *push* com a aplicação é efetuada pelo *firebase*.

Por fim, todos os serviços e ferramentas correm em imagens *docker* num cluster de *kubernetes*, podendo cada serviço ou ferramenta ser escalado *on-demand*, mediante um aumento de carga do mesmo. Toda a escalabilidade é horizontal, ou seja, sempre que um serviço perceba que não consegue responder a todos os clientes de forma rápida, este replica-se, entrando em cena outro serviço, para fazer o mesmo trabalho. Por sua vez, este vai-se replicando sucessivamente, mediante necessidade, voltando a reduzir as suas instâncias quando não haja necessidade de processamento por parte dos mesmos. O *kubernetes* com as suas *APIs* permite a gestão deste processo, podendo avaliar a necessidade de um serviço se replicar através de determinadas métricas, nomeadamente o uso de memoria RAM e CPU. O *kubernetes* oferece também uma *dashboard* que permite verificar todos os serviços em execução e as suas replicas.

## 4.5 Comunicação e processos

Esta secção descreve detalhadamente mecanismos, tecnologias e processos de comunicação utilizados. No paradigma de micro-serviços, a comunicação desempenha um papel fulcral e a sua arquitetura é a chave do sucesso de uma implementação. A Figura 8 permite inferir a comunicação entre micro-serviços, detalhada adiante neste capítulo, de onde sobressai a utilização do *RabbitMQ* e do *Google firebase*.

### 4.5.1 Processo de notificação

O FCM (*Firebase Cloud Messaging*) permite o envio notificações *push* aos utilizadores de aplicações através de uma API do Google firebase. As notificações *push* são populares em dispositivos móveis porque são eficientes na gestão da bateria, ao contrário das notificações

pull, que ficam continuamente à escuta de mensagens. Com as notificações *push*, o serviço de *cloud* atua em nome da aplicação e conecta-se ao dispositivo móvel apenas quando há novas mensagens.

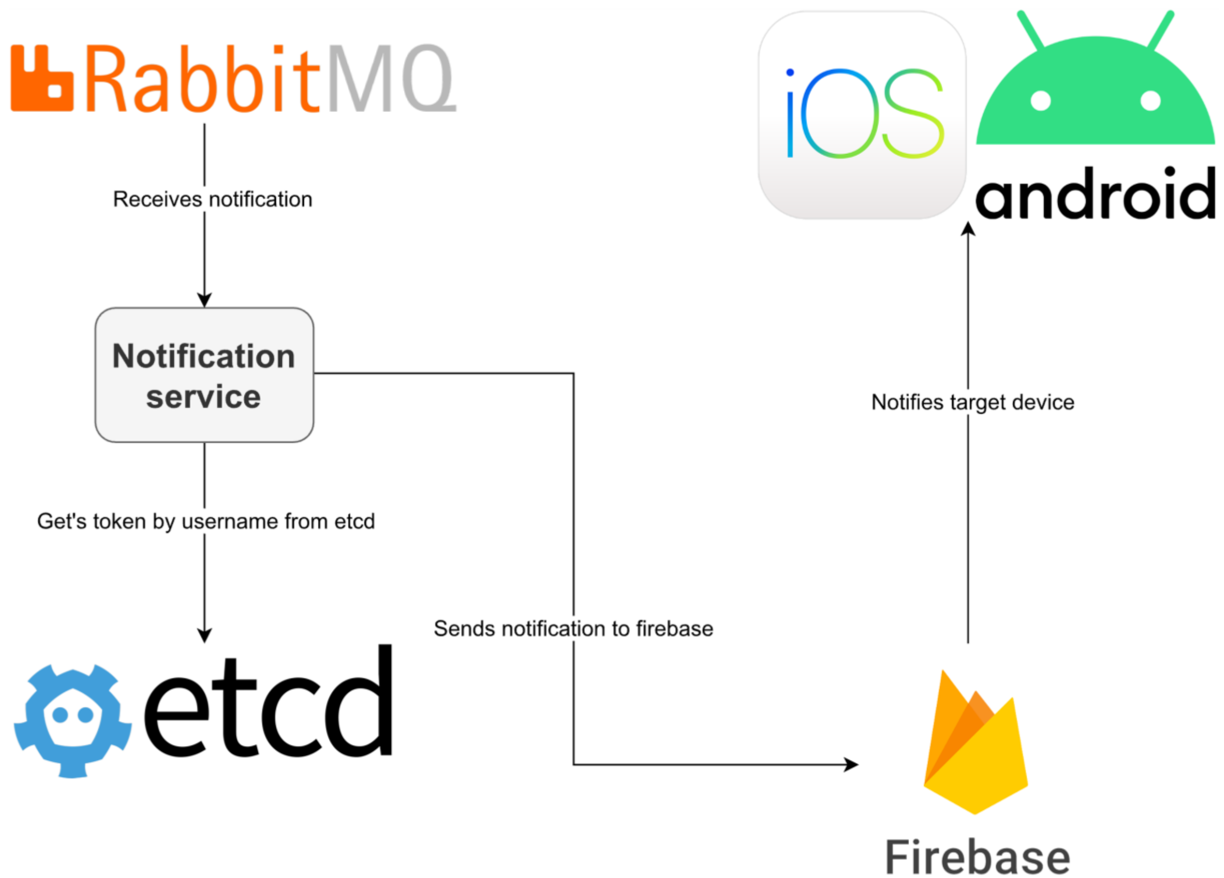



*Figura 12 Processo de notificação*

A Figura 12 ilustra o mecanismo de notificações implementado e a ter em consideração de cada vez que o `notification-service` é invocado na descrição dos processos subsequentes.

Sempre que cada serviço tem a intenção de notificar um ou mais utilizadores, este deve colocar uma mensagem no tópico notification do *RabbitMQ*, esta mensagem será posteriormente processada pelo `notification-service`, que consultará a base de dados etcd a fim de obter o *token* que permitirá, através do *firebase* notificar um utilizador em específico. Após ter o *token* do utilizador e a mensagem o serviço está em condições de invocar a API do FCM que enviará notificações push para a aplicação móvel.

### 4.5.2 Processo de registo e autenticação

A Figura 13 documenta o processo de registo e autenticação de um utilizador. Se o utilizador não estiver registado, é chamado o `account-service`, que, depois de validados os dados, adiciona o registo à base de dados dos utilizadores.

Caso o utilizador já tenha efetuado o seu registo, é feita uma chamada REST ao `auth-service` que invoca o `account-service` para recolher as informações do utilizador, proceder à sua validação e, se se verificar, à posterior autenticação. A comunicação REST entre os dois serviços utiliza o Refit Client, uma biblioteca que permite fazer uma abstração das comunicações REST por meio de interfaces.

Para cada autenticação válida é gerado um *token* JWT assim como um *token* de regeneração para que quando o *token* JWT expirar, possa ser gerado um novo.

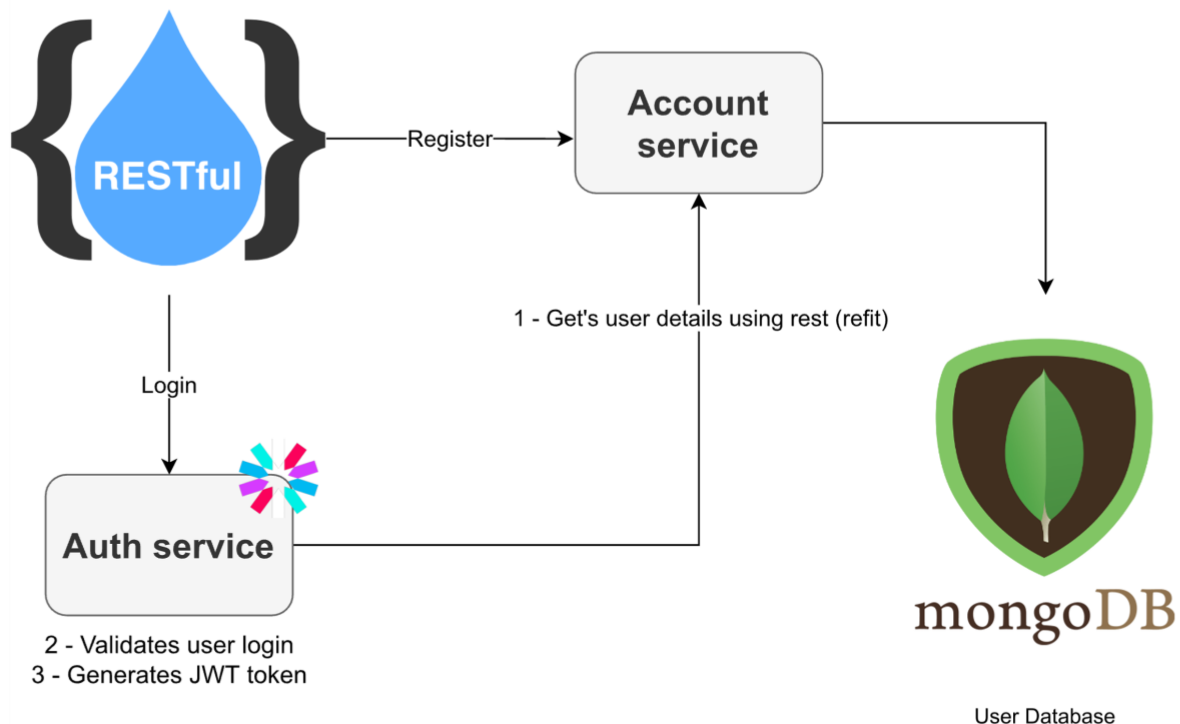



*Figura 13 Processo de registo e autenticação*

### 4.5.3 Processo de processamento de dados de viagem

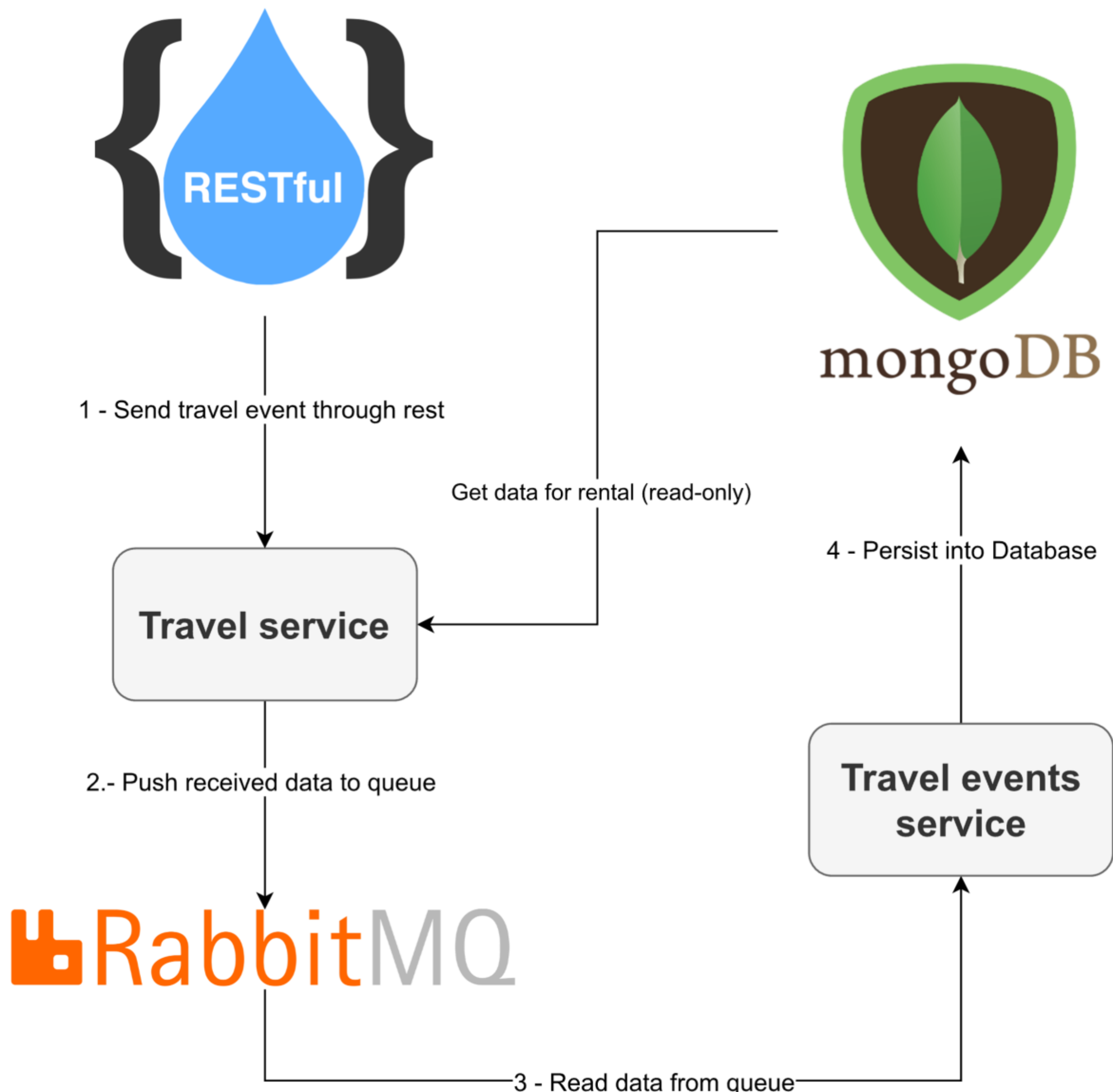



*Figura 14 Processo de processamento de dados da viagem*

O processo de processamento de dados, nomeadamente coordanadas GPS, que ocorre durante um aluguer está documentado pela Figura 16. Este processo inicia-se com chamadas REST a partir da aplicação móvel, de dois em dois segundos, ao `travel-service`. Estes dados são enviados para uma fila do *RabbitMQ*, e processados assincronamente pelo `travel-events-service` que os adiciona a uma coleção do *MongoDB* de forma sequencial.

O acesso ao histórico de um aluguer, nomeadamente no que respeita ao percurso efetuado durante o mesmo, é possível através de um pedido REST ao `travel-service`, que devolve os dados GPS relativos àquele aluguer.

### 4.5.4 Processo de aluguer

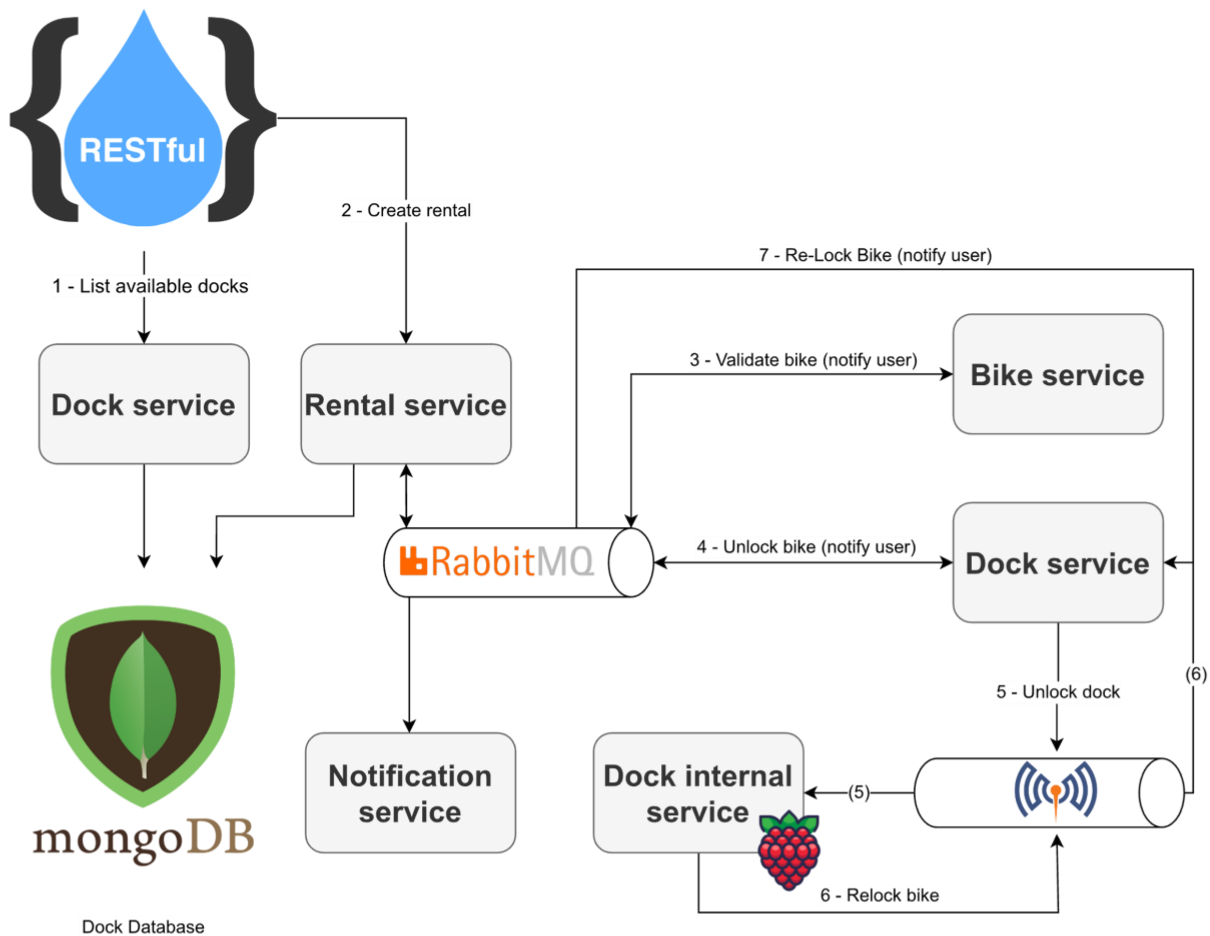



*Figura 15 Processo de aluger de bicicleta*

O processo de aluguer é composto por uma *SAGA* que através de orquestração *event sourcing* comunica com vários serviços atuando como uma máquina de estados para todo o processo de aluguer de bicicletas (Figura 15).

O ponto de entrada é o *dock service,* onde o utilizador, através da aplicação móvel requisita as bicicletas disponíveis, isto é, as docas que têm bicicletas conectadas, de seguida, este pode obter detalhes da doca e da bicicleta conectada, tais como localização da doca, modelo e marca da bicicleta. Assim que o utilizador tenha escolhido a bicicleta que pretende alugar, este pode fazer o *scan* do *QR code* presente na bicicleta.

Ao fazer scan do *QR code*, que contêm informações de identificação da bicicleta, é efetuado um pedido via REST ao `rental-service`, que cria o aluguer na base de dados *MongoDB*, e a partir desse momento, inicia a SAGA do fluxo de aluguer. A *SAGA* inicia-se no estado *submited* representando a submissão da requisição de aluguer por parte do utilizador, e começa por enviar um comando de validação da bicicleta ao `bike-service`, de forma

totalmente assíncrona, através do *RabbitMQ*. O estado do aluguer é então atualizado para *validating* e o `bike-service` processa o comando de validação da bicicleta, verificando se esta existe e está em condições de aluguer, e responde ao comando recebido com outro comando, indicando se a validação foi bem-sucedida ou se aconteceu algum erro. O comando de resposta é então recebido pelo `rental-service` que notificará o utilizador acerca do estado atual do seu aluguer e caso tenha obtido resposta positiva atualizará também o estado da *SAGA* para *reserving*, e enviará um novo comando para o `dock-service`, que tratará do desbloqueio da bicicleta. Neste ponto, o `dock-service`, trata de desassociar a bicicleta da doca na base de dados e comunicar com o serviço interno das docas, para que a doca que contem a bicicleta desbloqueie. Esta comunicação é feita assincronamente com uma mensagem por *MQTT*. Assim que o serviço interno das docas (`dock-internal-service`) receber a mensagem *MQTT* a doca é desbloqueada. Entretanto o `dock-service` devolve também a resposta ao `rental-service`, notificando o utilizador de que pode iniciar a sua viagem e começar também a enviar os eventos da mesma. A partir deste momento o estado do aluguer está em *in use*, e, só será alterado quando o utilizador devolver a bicicleta numa doca.

Quando o utilizador terminar a sua viagem, este pode simplesmente, encaixar a bicicleta numa doca que se encontre livre, e automaticamente, através da tecnologia NFC presente na doca, o `dock-internal-service`, identifica a bicicleta e envia uma mensagem assíncrona por *MQTT* para o `dock-service`, indicando que a bicicleta foi colocada na doca. O `dock-service` de seguida identifica o aluguer associado à bicicleta colocada, associa na base de dados a bicicleta à doca onde esta se encontra, e notifica o `rental-service`. Neste ponto, o `rental-service` termina o processo de aluguer, finalizando a SAGA, e fazendo um pedido assíncrono ao `payment-service` com as informações do aluguer relevantes para o processamento do pagamento.

Em cada estado desta *SAGA*, sempre que exista uma falha com algum do serviço, este pode enviar um evento de falha para o `rental-service` que irá tratar do erro e concluir a *SAGA*. O `rental-service` além de ter a responsabilidade de notificar o utilizador acerca das mudanças de estado, também atualiza o estado do aluguer na base de dados.

Na Figura 16 está representado a sequência de eventos englobados no processo de aluguer.

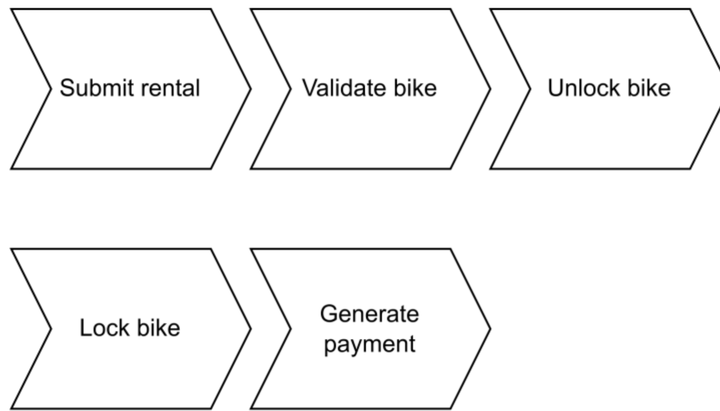



*Figura 16 Sequência de eventos do processo de alugar*

### 4.5.5 Processo de pagamento

O processo de pagamento utiliza também uma *SAGA*, isto porque este requer comunicação entre vários serviços, e depende dos mesmos para o seu resultado.

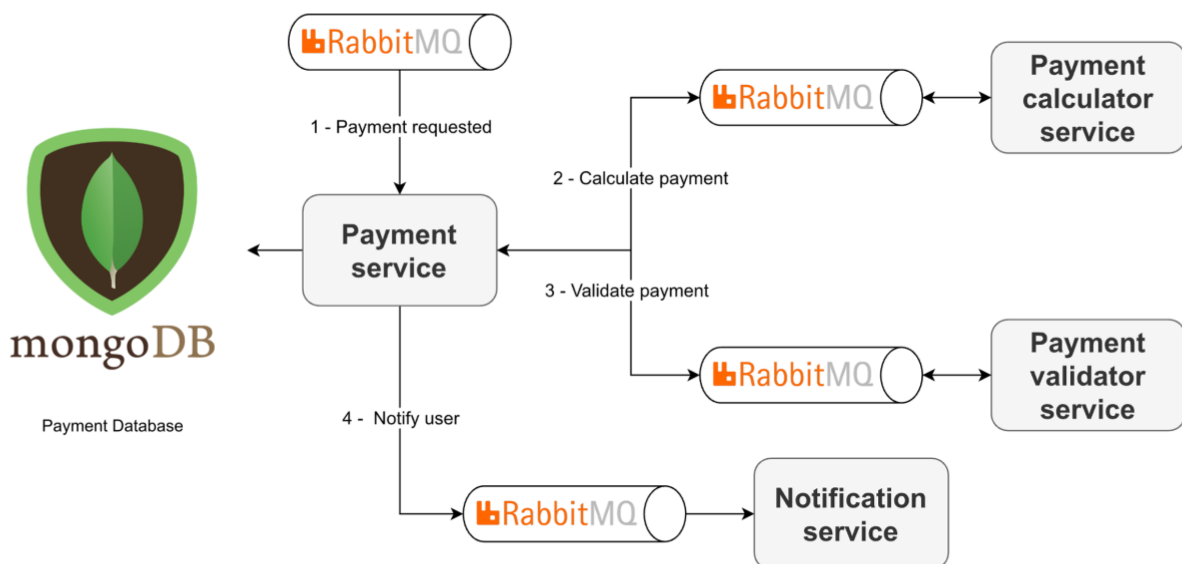



*Figura 17 Processo de pagamento*

Este processo, tal como descrito na Figura 17, inicia-se por um comando de requisição de pagamento enviado assincronamente, através de *RabbitMQ* para o `payment-service`, que tratará de criar o pagamento na base de dados *MongoDB* e iniciar a *SAGA*. O estado inicial, tal como no aluguer, é *submitted*, que é alterado para *calculating payment*, apos ser enviado um comando para o `payment-calculator-service` com os detalhes do aluguer, nomeadamente a data de inicio e fim, para que este calcule o valor a pagar pelo utilizador, e de seguida, devolva a resposta ao `payment-service` com este valor, que atualizará o pagamento na base de dados com este valor, o estado da *SAGA* é então alterado para *validating*, após o envio de um comando de validação de pagamento ao `payment-service`,

que emulará a cobrança do pagamento no cartão do utilizador, este responde ao *payment service* com um comando de sucesso ou falha na realização da cobrança e, o *payment service* envia uma notificação para o utilizador, através do *notification service*, indicando o resultado do pagamento.

Todo este processo é completamente assíncrono, a comunicação é efetuada via comandos e eventos por *RabbitMQ*.

## 4.6 Escalabilidade

Uma das grandes vantagens da arquitetura de micro-serviços é a sua escalabilidade. Esta característica traduz-se num escalonamento independente dos nós de um cluster *Kubernetes* e/ou *pods*, para atender à necessidade de recursos de um serviço, permitindo um dimensionamento correto das necessidades de infraestrutura. Este objetivo só é alcançável através da medição precisa dos recursos e garante a disponibilidade de um serviço perante um pico de procura.

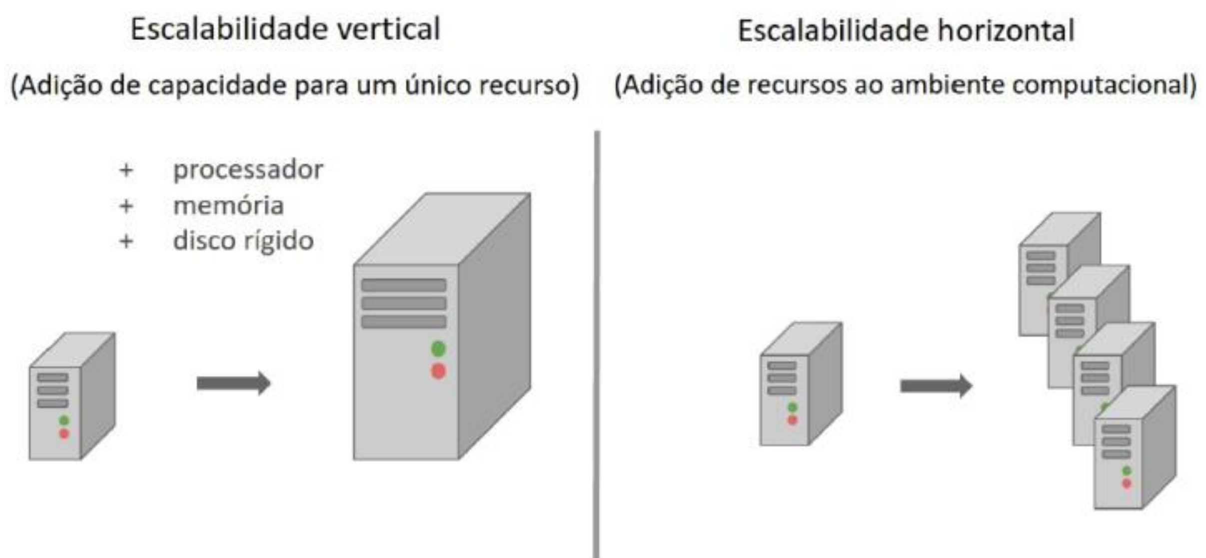



*Figura 18 Escalabilidade vertical e horizontal*

De forma simplista, a escalabilidade de micro-serviços consiste em adicionar poder computacional a determinado serviço para tratar uma maior afluência de tráfego. Esta granularidade fomenta a otimização dos recursos alocados uma vez que a necessidade de recursos de um micro-serviço é independente.

A escalabilidade de serviços é feita em duas dimensões: vertical e horizontalmente, conforme ilustra a Figura 18. Escalar verticalmente um serviço consiste em adicionar-lhe, em tempo real, recursos computacionais de acordo com os pedidos a que tem de atender. O

escalonamento horizontal, por seu turno, mantém os recursos computacionais de um serviço, mas, à medida que vão escasseando, são lançadas réplicas desse mesmo serviço.

Esta abordagem só é possível de concretizar perante a implementação de mecanismos de `service-discovery` e balanceadores de carga.

Neste projeto foi implementada escalabilidade horizontal, com recurso ao *metrics server* do *Kubertenes*, este funciona como um agregador de dados de uso de recursos em todo o cluster, permitindo assim, implementar regras de escalabilidade com base em métricas de determinada instância. Neste caso, as métricas utilizadas foram memória RAM e CPU, ou seja, sempre que uma instância de determinado serviço ultrapassa um certo limite de CPU ou RAM, este é automaticamente escalado horizontalmente (uma nova réplica do mesmo serviço é adicionada) para ajudar no processamento de carga.

## 4.7 Persistência de dados

Como referido previamente, a persistência de dados foi assegurada sobretudo por bases de dados NoSQL tais como o MongoDB ou o etcd.

O etcd foi usado sobretudo para dados chave-valor enquanto o MongoDB alojou as restantes estruturas de dados. O armazenamento de dados em MongoDB seguiu a arquitetura *database per service*, em que cada serviço possui a sua base de dados, por uma questão de gestão de recursos, foi implementada apenas uma instância de MongoDB com várias bases de dados, permitindo assim uma redução de custos face a uma instância por serviço.

O modelo de negócio não exigiu grandes relações a nível da estrutura de dados de um serviço, o que tornou o NoSQL ainda mais atraente para esta implementação, os recursos de pesquisa geo espacial do MongoDB permitiram também fazer pesquisas por coordenadas GPS num determinado raio de operação, de forma simples e bastante otimizada. Para armazenamento dos eventos da viagem, que surgem numa cadência elevada, o MongoDB mostrou-se também uma ótima escolha, possuindo uma elevada performance de escrita de dados, otimizando assim o processo. De modo geral o MongoDB cumpriu todos os desafios arquiteturais de armazenamento de dados, não havendo necessidade de usar bases de dados relacionais.

## 4.8 Comunicação assíncrona

Numa arquitetura de micro-serviços moderna, a comunicação assíncrona é fundamental, para que os serviços possam comunicar entre si sem comprometer a sua escalabilidade, fiabilidade e consistência, deste modo, o broker de mensagens, torna-se parte chave em toda uma

arquitetura deste calibre. O RabbitMQ provou ser uma excelente ferramenta para este tipo de desafio, permitindo a sua configuração e utilização de forma relativamente simples.

Padrões e arquiteturas utilizados neste projeto, tais como SAGA, têm como base de comunicação um broker de mensagens assíncronas, que tem a capacidade de processar mensagens de forma ordenada garantindo a ordenação dos dados.

Como estrutura de dados de comunicação, foram utilizados essencialmente dois tipos de abordagens, comandos e mensagens, sendo que os comandos são mensagens únicas, que após serem lidas a primeira vez desaparecem, tornando-se vantajosos por exemplo numa SAGA, em que determinado serviço quer enviar uma instrução ao outro. Por outro lado, as mensagens, após serem publicadas no RabbitMQ podem ser lidas pelos vários consumidores que subscrevem o tópico onde a mensagem havia sido publicada, esta abordagem torna-se interessante por exemplo na partilha de dados de viagem, em que outros serviços podem subscrever o tópico dos dados da viagem, e fazer o tratamento desses dados, sem depender de determinado serviço que os esteja a consumir, podendo consumi-los em simultâneo.

## 4.9 Infraestrutura

Relativamente à infraestrutura, todos os serviços foram projetados para correrem em containers Docker, seguindo o padrão *service instance per container*, em que cada instância do serviço irá correr num container Docker. Todos os containers correm em cima de Kubernetes, que funciona como orquestrador. O Kubernetes aloja também os serviços de base de dados e broker de mensagens. Toda a infraestrutura pode ser lançada com recurso a um script *shell* invocando ficheiros de configuração *yaml* correspondentes a cada serviço.

Toda esta plataforma e serviços foram desenvolvidos sobre um cluster, constituído por dois Raspberry pi 4, sendo um de 4GB de RAM e outro de 2GB, este pequeno dispositivo, apesar de limitado, foi suficiente para suportar todos os serviços, base de dados e broker de mensagens, e garantir a escalabilidade dos mesmos. O Raspberry pi de 4GB atua como *master*, ou seja, além de fazer a gestão dos outros *nodes* também permite a execução e gestão de serviços. O Raspberry pi de 2GB atua como *node*, e lança ou escala determinado serviço, sempre que o *master* o requisitar, Figura 20.

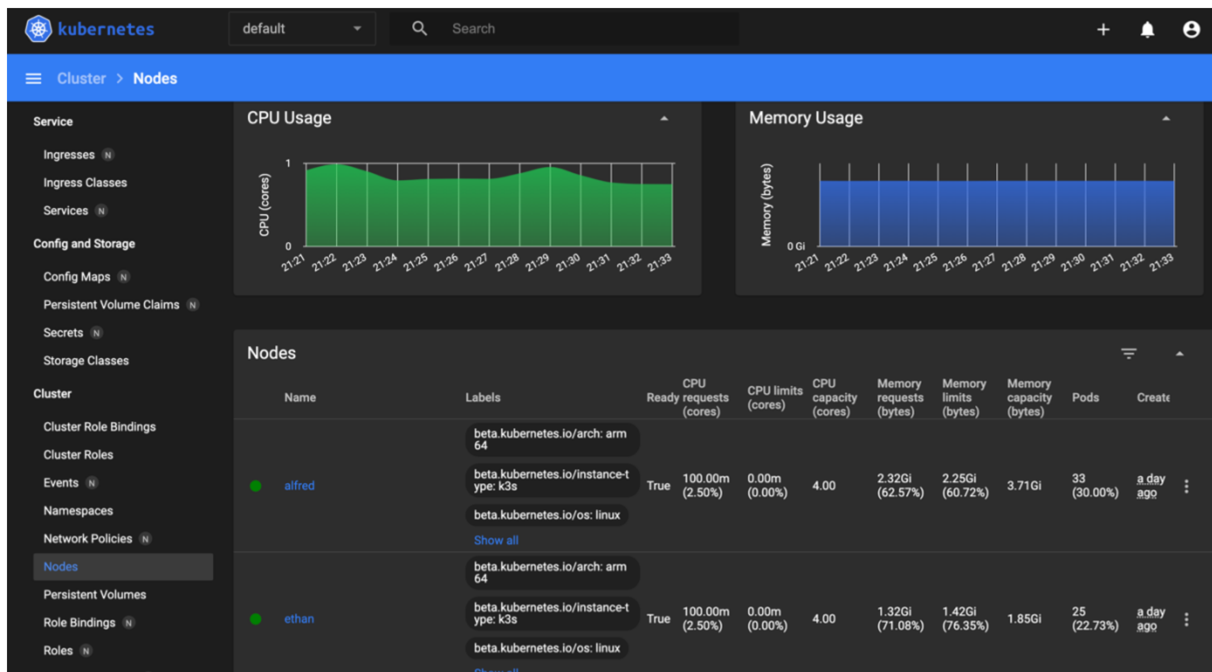



*Figura 19 - Dashboard do kubernetes com os nodes utilizados*

Para garantir a persistência dos dados e o seu acesso de forma rápida, os Raspberry Pi foram equipados com discos SSD, e as bases de dados foram configuradas no Kubernetes com volumes persistentes, assegurando a persistência dos dados em caso de paragem da infraestrutura.

## 4.10 Dispositivo IoT

Para emular o funcionamento de um conjunto de docas num determinado local, foi criado como prova de conceito um serviço *python* que corre num dispositivo IoT. Este dispositivo trata-se de um *Raspberry Pi zero w*, trata-se de um microcomputador completo, com seus componentes numa única placa lógica. Possui um processador com a arquitetura ARM de 1 core, com um clock de 1GHZ, com uma memória RAM de 512mb. Apesar de ser um microcomputador, é mais que suficiente para gerir bloqueios e desbloqueios de um conjunto de docas, permitindo ser conectado à internet por wireless.

A nível de hardware, foram instalados 2 leitores que cartões NCF MFRC522 que têm a função de identificar a bicicleta que se encontra na doca, assim como obter detalhes da mesma. Foi instalada também uma relê de 4 portas, com duas lâmpadas conectadas, emulando assim o estado da doca, ou seja, se a bicicleta estiver bloqueada na doca, a lâmpada associada à doca está acesa, caso contrário esta está apagada, num cenário real, associado a esta relê, estaria o mecanismo de bloqueio de desbloqueio da doca. Tanto os leitores NFC como as lâmpadas da relê são totalmente controlados pelos *Raspberry Pi*. As bicicletas são equipadas com uma tag NFC contendo a sua identificação.

Cada *Raspberry Pi* gere um determinado número de docas, sendo que para cada porta da relê, deverá haver um leitor NFC, dependendo assim o número de docas suportadas, do número de portas da relê.

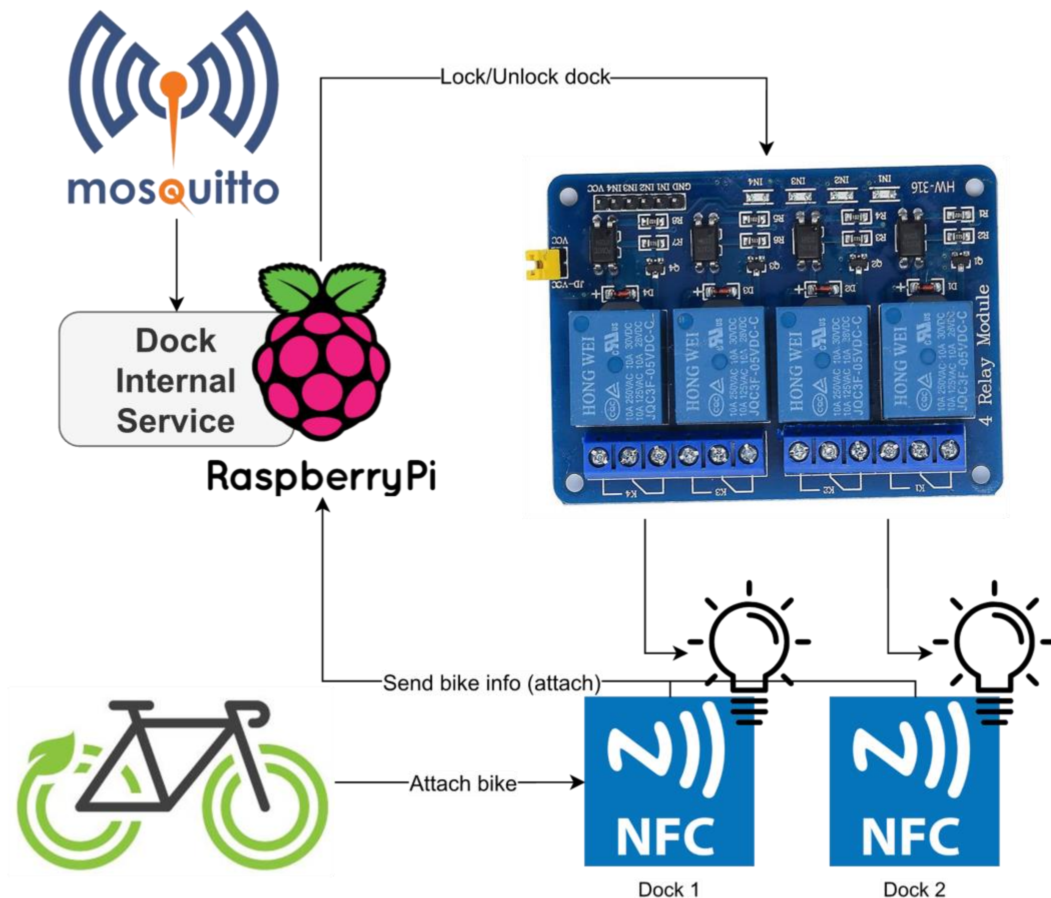



*Figura 20 - Arquitetura do dispositivo IoT de gestão de docas*

Na Figura 20 encontra-se representada a arquitetura implementada para a gestão física das docas. O serviço *python* (`dock-internal-service`) corre diretamente no *Raspberry Pi*, recebendo instruções de desbloqueio de determinada doca assincronamente, por *MQTT*. Sempre que é detetada uma bicicleta num leitor NFC, é enviada, uma mensagem assíncrona, por *MQTT* ao `dock-service` indicando que determinada bicicleta foi anexada a determinada doca, este posteriormente tratará do restante processo de bloqueio. Para efetuar o desbloqueio, o `dock-service` envia assincronamente uma mensagem para desbloqueio de determinada doca ao serviço interno que corre na doca para desbloqueio. Assim a porta da relê será alterada, apagando a lâmpada correspondente à doca, permitindo assim que outras bicicletas se conectem àquela doca, dado esta ter passado ao estado livre.

Durante o processo de bloqueio e desbloqueio das docas, é armazenado localmente no *Raspberry Pi*, o estado atual da doca (bloqueada ou desbloqueada), para evitar que caso surja algum problema, a doca não assuma um estado errado. Assim sempre que o `dock-`

`internal-service` é iniciado, este coloca as docas no seu estado previamente armazenado.

A Figura 21 apresenta duas fotos do dispositivo desenvolvido, com dois cartões NFC que representam duas bicicletas, e os leitores da doca. Assim que o estado da respetiva doca altere, a luz associada a essa doca também altera, fica ligada quando desbloqueada e desligada quando bloqueada.

*Figura 21 - Dispositivo Iot desenvolvido*

## 4.11 Aplicação móvel

A aplicação móvel, desenvolvida em flutter, está concebida para funcionar em dispositivos android e IOS. A nível de arquitetura a aplicação foi desenvolvida com recurso ao Bloc, uma arquitetura que permite a separação das regras de negócio da interface.

A aplicação foi também integrada com o firebase, permitindo assim notificações *push* com o smartphone bloqueado, mas também mensagens assíncronas durante a execução da aplicação.

Na primeira utilização, a página de registo é apresentada, após feito o registo, o utilizador deverá efetuar o login, tal como apresentado na Figura 22. O processo despoletado nesta interação encontra-se descrito neste documento, como Processo de registo e autenticação.

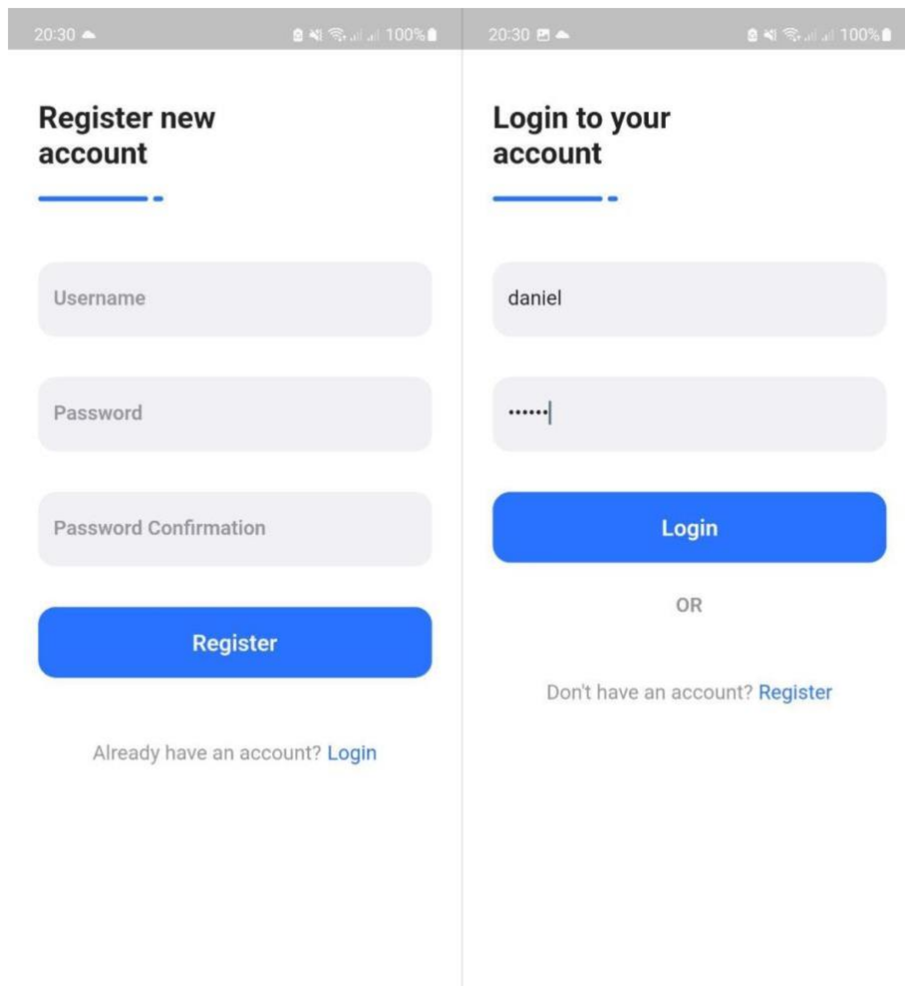



*Figura 22 - Registo e autenticação na aplicação móvel*

Estando o utilizador autenticado, a aplicação será redirecionada para o ecrã inicial, que contém um mapa com as docas próximas que possuem bicicletas disponíveis. Nesse ecrã o utilizador poderá selecionar uma doca e ver os detalhes da mesma, nomeadamente a morada, e detalhes da bicicleta conectada, Figura 23. Esta interação faz parte do Processo de aluguer, nomeadamente a parte inicial de listagem de docks disponíveis.

Através do ecrã inicial é também possível aceder a outros ecrãs, tais como scan do QR code de bicicleta, histórico de alugueres e detalhes do utilizador.

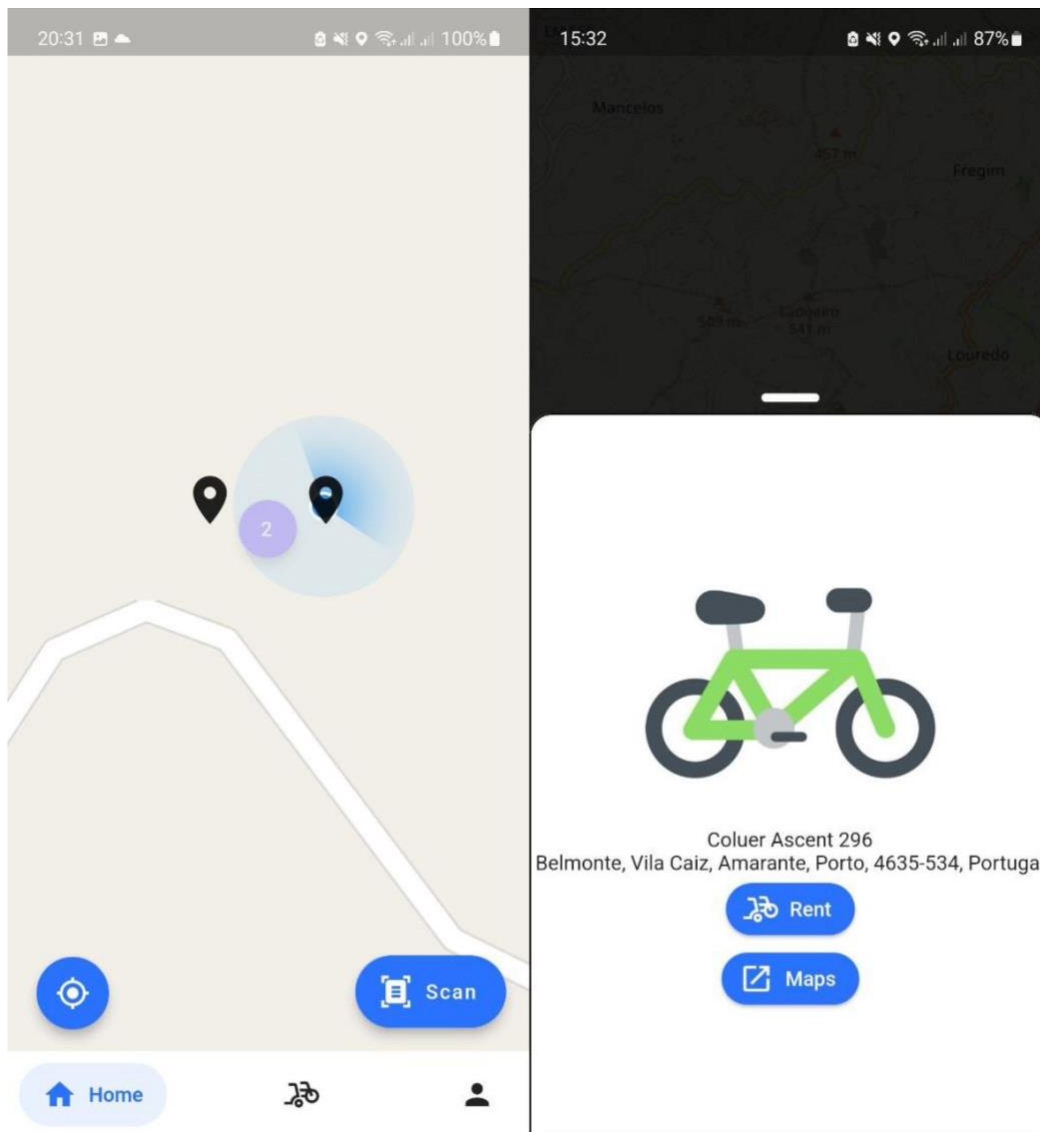



*Figura 23 - Bicicletas proximas e detalhes na aplicação móvel*

Para que o utilizador possa efetuar o aluguer de uma bicicleta é mandatório que este possua um cartão bancário associado à sua conta, para posteriormente lhe ser cobrada a viagem. A associação poderá ser efetuada através da página de detalhes do utilizador, ou então, assim que este tente fazer o scan de um código QR de uma bicicleta, caso não tenha cartão associado, será automaticamente redirecionado para a página de associação de cartão. As páginas mencionadas, estão representadas na Figura 24.

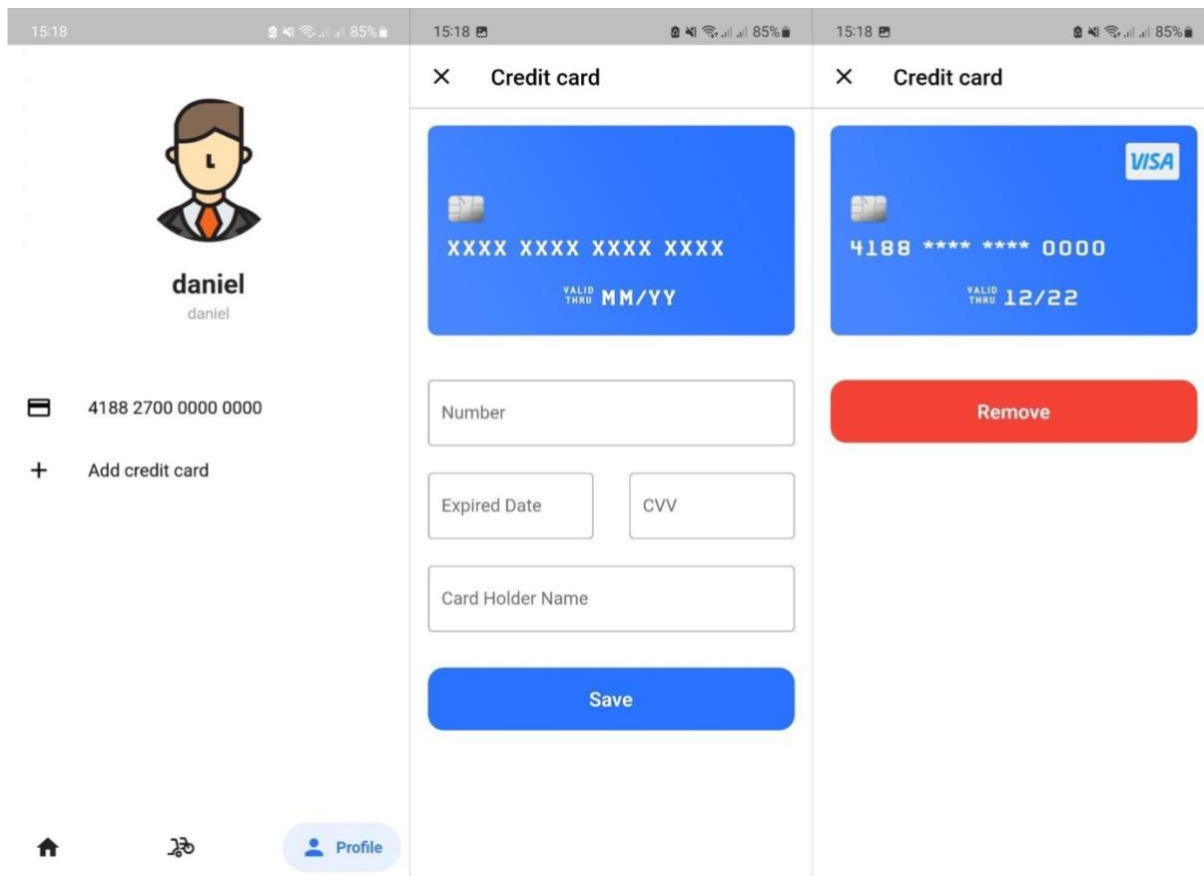



*Figura 24 - Detalhes do utilizador e gestão de cartões bancários na aplicação móvel*

Tendo um cartão bancário associado à sua conta, o utilizador está em condições de alugar uma bicicleta. Para tal, deverá dirigir-se para junto da mesma, poderá selecionar a bicicleta pretendida no mapa e selecionar a opção “Maps” que lhe irá indicar o caminho até à bicicleta, e assim que esteja junto dela, poderá fazer o scan do código QR presente na mesma, através do botão “Scan” da aplicação. Após ter feito o scan, o processo de aluguer tem continuidade, validando a bicicleta, reservando-a e desbloqueando-a, caso todos estes passos tenham sucesso, a viagem inicia-se e é apresentada ao utilizador uma página com um mapa da sua localização e o tempo de viagem decorrido, Figura 25.

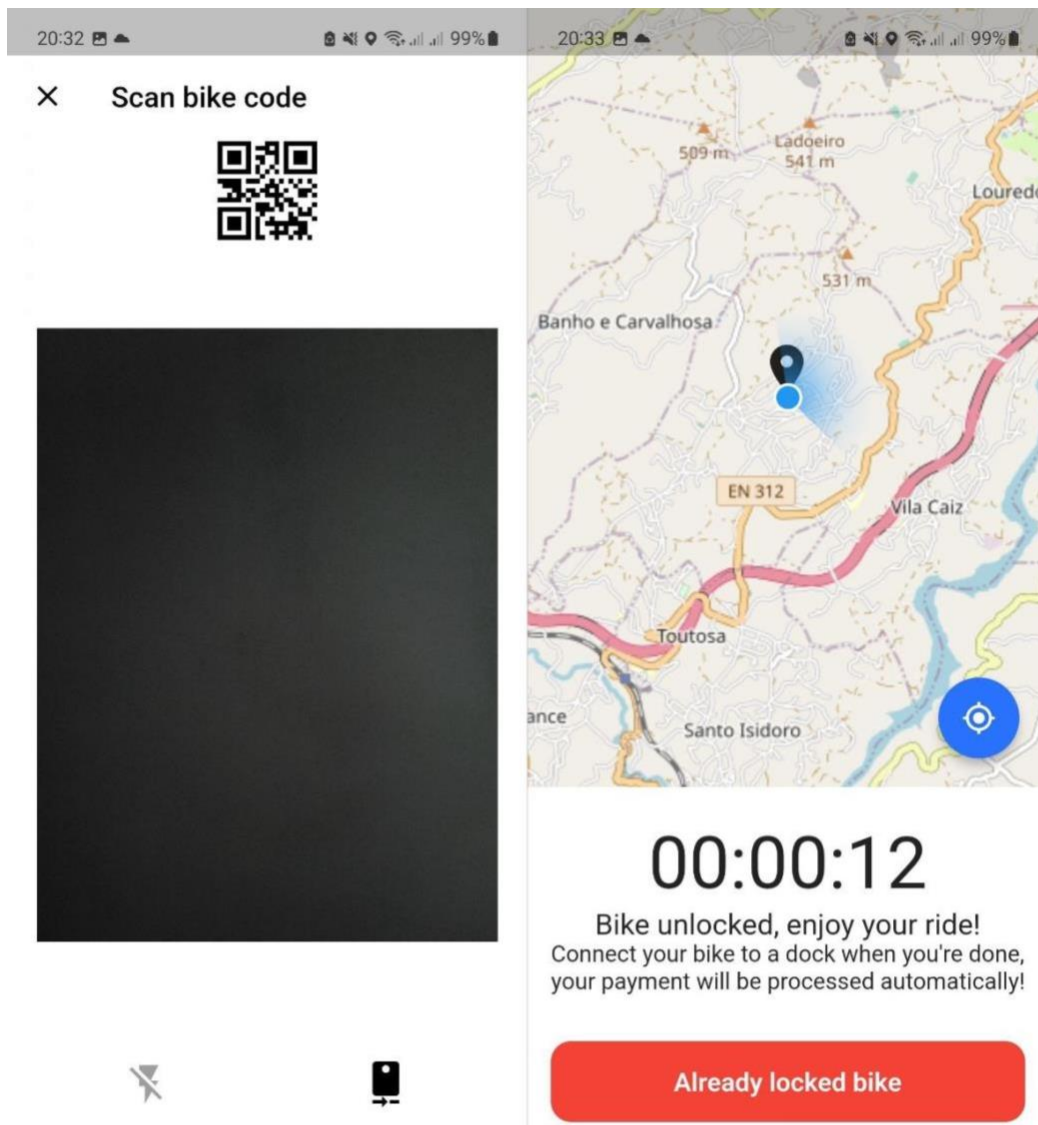



*Figura 25 - Scan de código QR e viagem na aplicação móvel*

Durante a viagem, a aplicação recolhe automáticamente dados relativos à localização do utilizador, enviado para o serviço em forma de eventos da viagem, tal como descrito no Processo de processamento de dados de viagem.

Assim que o utilizador termine a viagem, e entrega a bicicleta em determinada doca, é enviada uma notificação para a aplicação, que apresentará uma página ao utilizador para que este possa avaliar a viagem, colocando uma cotação em estrelas de 0 a 5 e um comentário opcional, Figura 26.

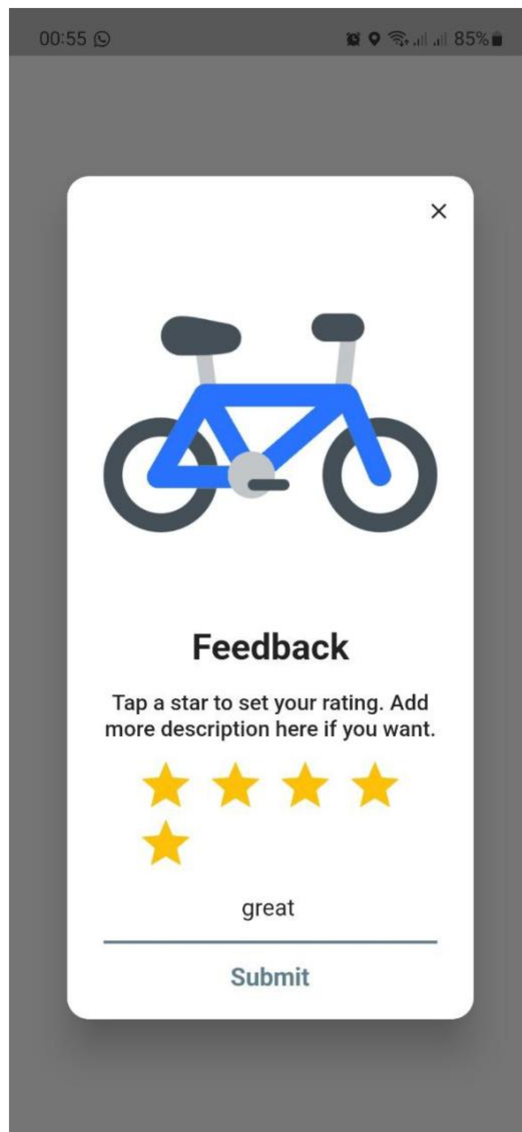



*Figura 26 - Feedback da viagem na aplicação móvel*

Por fim o pagamento da viagem é processado automaticamente, através do Processo de pagamento, terminando assim o aluguer.

O utilizador, através da aba “Rentals” tem acesso ao histórico de alugueres, que lhe permite obter em detalhe o percurso percorrido, o tempo decorrido desde a data que iniciou e o valor que lhe foi cobrado em determinado aluguer, Figura 27.

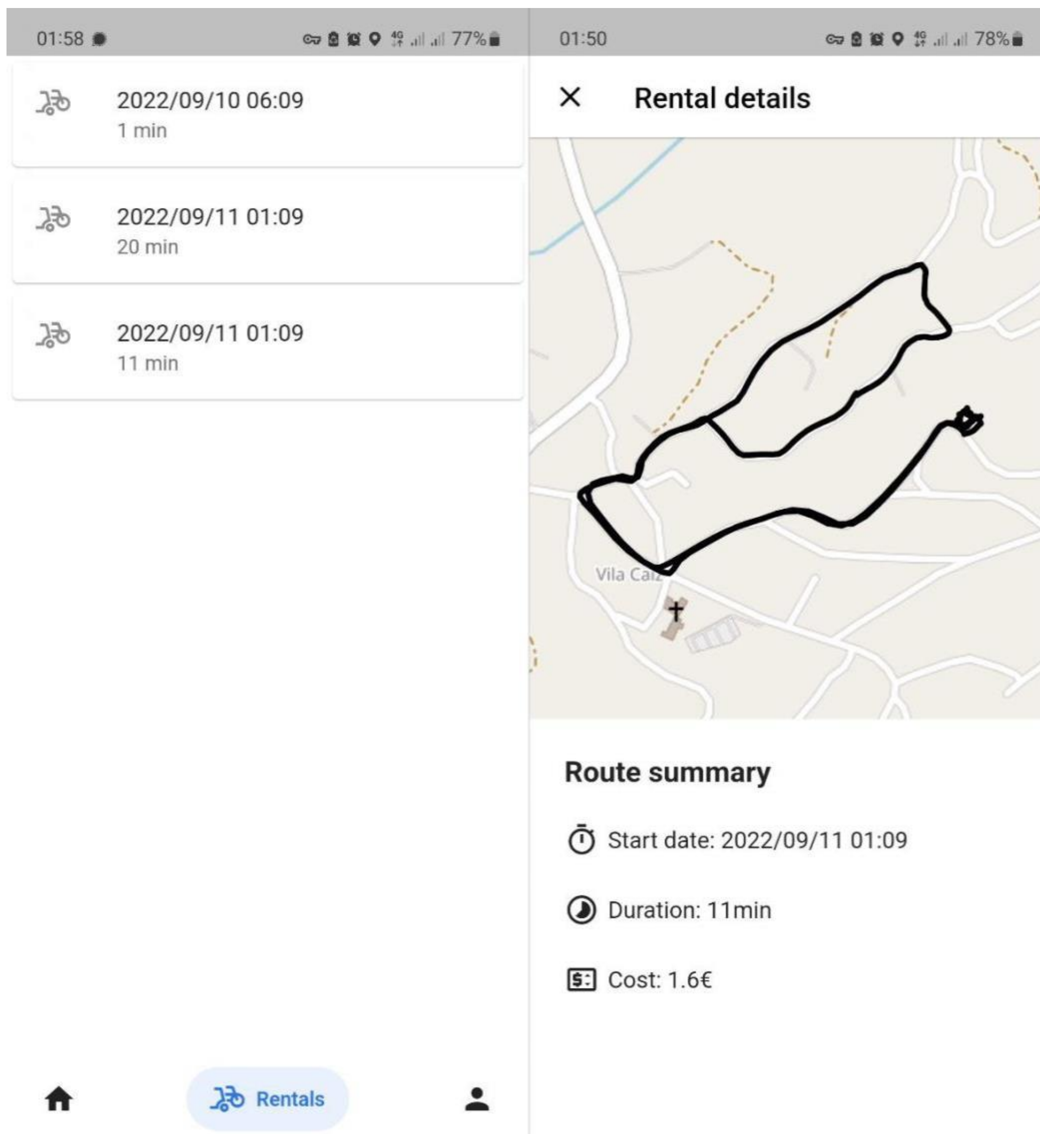



*Figura 27 - Listagem e detalhes de aluguer na aplicação móvel*

Toda a comunicação da aplicação com o *backend* de micro-serviços é feita através de pedidos REST ao `api-gateway`, ou através de notificações *push* do *firebase*. Pedidos tais como histórico de alugueres, gestão de cartões, detalhes do utilizador comunicam sincronamente utilizando o protocolo REST, enquanto que, a comunicação quando determinado serviço deseja notificar a aplicação é assíncrona, com recurso a *push notifications* do *firebase*.

## 4.12 Testes

De modo a garantir a qualidade e bom funcionamento da plataforma, é fundamental que existam testes que validem as funcionalidades do software desenvolvido.

Foi assim, desenvolvida uma bateria de testes unitários e de integração, que visam assegurar o bom funcionamento de cada micro-serviço, com recurso a uma pipeline de *continuous*

*integration*, esta bateria de testes é executada a cada *commit*, procurando assim, evitar que alterações no código possam quebrar funcionalidades existentes.

De acordo com a filosofia da pirâmide de testes, Figura 28, que define níveis de testes e a quantidade de testes proporcional a cada nível, os testes unitários são os mais importantes, estes encontram-se na base da pirâmide e devem ser realizados em maior quantidade, seguidos dos testes de integração, terminando com os testes *end-to-end* (e2e) que são deverão existir em menor quantidade. Deste modo, foram implementados em maior quantidade testes unitários, validando todas as funcionalidades principais da plataforma, seguido de alguns testes de integração que validam a integração de determinado serviço com outros serviços ou ferramentas.

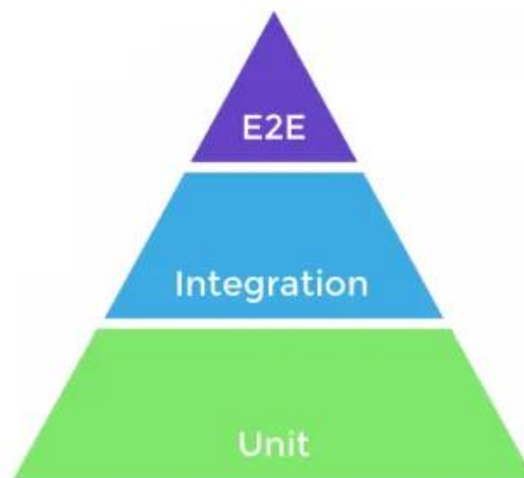



*Figura 28 - Pirâmide de testes*

### 4.12.1 Testes unitários

O teste unitário é uma verificação feita numa pequena porção de código. No teste unitário, cada parte do sistema ganha uma atenção devida e detalhada, de modo a otimizar o processo de identificação de erros. O objetivo é ajudar a identificar os bugs e impedir que estes voltem após serem efetuadas alterações no sistema.

Foram implementados testes unitários para validar as principais funcionalidades da plataforma.

Em serviços como o `rental-service`, que possui uma lógica de funcionamento relativamente complexa, contendo uma Saga que faz uso duma máquina de estados, é fundamental validar a transição de estados dentro da saga. Para tal foram efetuados diversos testes unitários para validar cada transição de estado da saga, e o comportamento associado

a determinado estado, inclusive foram testados casos de falha na saga, garantindo que esta tem o comportamento adequado no tratamento de erros. Tal como o `rental-service`, o `payment-service` exigiu um maior esforço de testes unitários, dado possuir também uma saga, recorrendo a uma máquina de estados.

Em serviços mais simples, que expõem *endpoints* REST com operações básicas à base de dados, tal como o `account-service` ou o `feedback-service`, foram efetuados testes unitários aos seus controladores, a fim de garantir que estes respondem corretamente aos pedidos REST. Serviços que comunicam assincronamente através de *producers* e *consumers*, tais como o `bike-service`, `dock-service`, `notification-service`, `payment-calculator-service`, `payment-validator-service` e `travel-event-service`, foram também testados unitariamente, validando que efetuam o tratamento correto a uma mensagem recebida, e, caso aconteça, que a resposta publicada corresponda ao esperado. Em serviços que possuem lógica de calculo, transformação de dados, ou regras de validação específicas, foram testadas as funcionalidades que incidem sobre esses pontos, tais como o cálculo do pagamento no `payment-calculator-service` ou a validação da password no `auth-service`.

Foram realizados vários testes unitários, que validam as principais e mais complexas funcionalidades dos micro-serviços desenvolvidos. Todo o código poderia ser coberto por testes unitários, havendo assim um grau de segurança maior, no entanto, devido a restrições de tempo para a implementação, optou-se por testes as funcionalidades mais relevantes e complexas, que maiores probabilidades teriam de apresentar algum tipo de *bug*.

### 4.12.2 Testes de integração

Os testes de integração, são testes entre diferentes módulos de um sistema, como por exemplo o teste de pedidos HTTP, o mais comum para APIs. Nos testes de integração, é verificado o resultado de um pedido completo, analisando o formato de resposta, o código de status na resposta HTTP, o formato de dados e validações. Este tipo de teste é muito importante, pois facilita bastante, em futuras manutenções, onde a criação de novas funcionalidades ou alterações em antigas funcionalidades podem modificar o comportamento de outras partes do sistema.

Dado a arquitetura de micro-serviços ser algo complexa, realizar testes de integração torna-se um desafio, pois é necessário criar um ambiente de teste, com as dependências necessárias que, de preferência seja descartável no fim da execução dos testes. Dada esta necessidade, surge então, a biblioteca *Testcontainers* como solução para o problema. Trata-se de uma biblioteca em *dotnet* que dá suporte a testes com instâncias descartáveis de

*containers docker*. Permitindo assim, criar e levantar automaticamente uma imagem *Docker* configurável no início da execução de uma bateria de testes, e descartá-la no final. O *TestContainers* oferece suporte para as mais variadas bases de dados e *message brokers*, nomeadamente *MongoDB* e *RabbitMQ* que foram os dois serviços utilizados nesta ferramenta.

A execução de um teste de integração, passa por um *setup* inicial, onde é criado o ambiente necessário a execução do micro-serviço, nomeadamente a criação dos containers de *MongoDB*, *RabbitMQ* ou outro serviço do qual o micro-serviço dependa. É criada também uma instância do micro-serviço, com as configurações necessárias para este usar o ambiente de testes construído no passo anterior. De seguida são executados os testes inerentes aquele micro-serviço, efetuado geralmente pedidos HTTP, validando os *status codes* e resposta aos diversos pedidos.

Em micro-serviços que comunicam com outros serviços ou micro-serviços por pedidos HTTP, foi utilizada também uma ferramenta que permite emular as respostas HTTP. Esta ferramenta, denominada por *LightHTTP*, trata-se de uma biblioteca para *dotnet* que permite criar um serviço HTTP, e configurar *endpoints* e respostas a pedidos efetuados a esses *endpoints,* a configuração desta ferramenta é também efetuada no *setup* inicial dos testes.

Usando exemplos concretos de testes de integração realizados, temos por exemplo, os testes efetuados ao `user-service`. Dado este micro-serviço depender da base de dados *MongoDB*, no *setup* inicial é automaticamente criada e instanciada uma imagem *Docker* do *MongoDB*, com as configurações necessárias aquele micro-serviço. De seguida, é criada uma instância do `user-service` com a configuração para a base de dados de teste e procede a execução dos testes. Nos testes são efetuados pedidos HTTP aos *endpoints* que este serviço oferece, validando por exemplo, que a criação de um utilizador é bem-sucedida e este fica disponível para consulta num pedido seguinte.

Num outro exemplo, temos o `auth-service`, que apesar de não ter qualquer tipo de comunicação com bases de dados, contém o `user-service` como dependência, usando-o, para obter os detalhes do utilizador durante o processo de autenticação, através de um pedido HTTP. De forma a simular e controlar a resposta por parte do `user-service` durante a execução dos testes do micro-serviço, foi utilizado o *LightHTTP* no *setup* inicial, configurando uma resposta para *endpoint* de obtenção de detalhes do utilizador. Feito o *setup*, os testes correm validando o processo de autenticação do utilizador, verificando também por exemplo, se mediante uma resposta não positiva por parte do `user-service`, se o `auth-service` responde com o esperado.

Tendo em conta a flexibilidade das ferramentas utilizadas para criação do ambiente de testes de integração para os micro-serviços, a execução destes testes não requer qualquer tipo de ambiente pré-configurado. Requer apenas que a máquina que executa os testes possua o *Docker*, permitindo assim, executar testes de integração de forma totalmente automatizada e desacoplada numa pipeline de CI.

Apesar deste tipo de testes por norma, serem feitos em menor quantidade que os testes unitários, haveria mais testes de integração para fazer, dada a limitação em termos de tempo, foi optado por testar os pontos mais importantes, e com maior probabilidade de falha na plataforma.

### 4.12.3 Continuous integration

*Continuous integration* (CI) é uma prática de desenvolvimento em que os programadores integram com frequência o código num repositório partilhado.

Cada *commit* efetuado para um repositório é verificado por uma compilação e execução de testes automatizados, permitindo detetar problemas de forma antecipada, diminuir o tempo necessário para os corrigir e assim melhorar a qualidade do software.

Para alojamento de código num repositório partilhado, foi escolhido o GitHub, deste modo, tirou-se partido das funcionalidades desta plataforma para CI/CD, o *Github Actions*. Este trata-se de uma plataforma de *continuous integration* e *continuous delivery* (CI/CD) que permite automatizar a compilação, testagem e *deploy* do código. Podem ser criados fluxos que compilem e testem cada *commit* ou *pull request*.

Foi então criado um pipeline de CI, que compila e executa os testes de todos os micro-serviços desenvolvidos, a cada *commit* e *pull-request*, forçando a que todos os testes passem, e todos os serviços compilem, para que o programador possa efetuar o *merge* do *pull request,* Figura 29.

*Figura 29 - Pipeline de CI em GitHub Actions*

# 5 Conclusão

O advento das plataformas dinâmicas para a provisão de serviços mudou drasticamente a forma como as aplicações são estruturadas. A procura por uma infraestrutura cada vez mais focada numa eficiente troca de dados e na distribuição de serviços fomentou o paradigma da Computação em Nuvem e promoveu o provisionamento de infraestrutura de forma distribuída. Aliada à abstração dos recursos físicos, através da virtualização, estava potenciado o aparecimento das primeiras arquiteturas orientadas a serviços.

Contudo, uma infraestrutura elástica, adaptável dinamicamente às necessidades do serviço e da aplicação, ainda encontra algum atrito junto das metodologias de desenvolvimento vinculadas às soluções monolíticas: o esforço arquitetural para as orientar ao serviço é gigantesco e implica uma nova relação com conceitos e mecanismos próprios do paradigma de micro-serviços. *Service discovery, load balancers, circuit breakers* ou *message brokers* fazem parte do léxico de um conjunto de padrões que evoluiu no sentido de ampliar o conceito de *service oriented architecture* (*SOA*) para uma abordagem mais alinhada com as necessidades hodiernas, sobretudo com uma modelação mais focada em módulos de serviço do que em eventos.

De uma forma quase natural, os novos conceitos foram-se expandindo para uma nova abordagem, que extrapolou os limites de um bloco de serviço, tornando o próprio serviço, também ele, numa entidade modular e que pode ser estruturada em partes menores. Juntamente com outras frentes da evolução metodológica e arquitetural, o paradigma de micro-serviços começa a emergir como uma alternativa relevante, baseada na proposta de transformar um sistema único num conjunto de pequenos serviços, ou micro-serviços, que cooperam e podem ser desenvolvidos, implementados e geridos de forma independente.

É esta a visão e, consequentemente, o grande contributo do trabalho desenvolvido e descrito ao longo deste documento. Consciente da complexidade arquitetural de uma aplicação sob o paradigma de micro-serviços e em que a comunicação interna assenta em filas assíncronas de mensagens, talvez, num futuro próximo, seja possível minimizar essa complexidade com a adoção de um padrão arquitetural comum, encapsulado numa *framework* que potencie o desenvolvimento aplicacional à luz desta filosofia.

Esta página foi deixada em branco propositadamente

# 6 Referências bibliográficas


[1] Sofi A Becker *et al.*, “European Mobility Atlas,” *Heinrich-Böll-Stiftung European Union, Brussels, Belgium*, 2021, Accessed: Jan. 27, 2022. [Online]. Available: https://eu.boell.org/en/European-Mobility-Atlas-2021-PDF

[2] P. DeMaio, “Bike-sharing: History, Impacts, Models of Provision, and Future,” *J Public Trans*, vol. 12, no. 4, pp. 41–56, Dec. 2009, doi: 10.5038/2375-0901.12.4.3.

[3] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, “Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages,” *ACM International Conference Proceeding Series*, vol. Part F129907, May 2017, doi: 10.1145/3120459.3120483.

[4] C. Bullock, F. Brereton, and S. Bailey, “The economic contribution of public bike-share to the sustainability and efficient functioning of cities,” *Sustain Cities Soc*, vol. 28, pp. 76–87, Jan. 2017, doi: 10.1016/J.SCS.2016.08.024.

[5] E. Fishman, S. Washington, and N. Haworth, “Transport Reviews A Transnational Transdisciplinary Journal Bike Share: A Synthesis of the Literature,” 2013, doi: 10.1080/01441647.2013.775612.

[6] S. Shaheen, S. Guzman, and H. Zhang, “Bikesharing in Europe, the Americas, and Asia: Past, Present, and Future,” *https://doi.org/10.3141/2143-20*, no. 2143, pp. 159–167, Jan. 2010, doi: 10.3141/2143-20.

[7] W. Qiu and H. Chang, “The interplay between dockless bikeshare and bus for small-size cities in the US: A case study of Ithaca,” *J Transp Geogr*, vol. 96, p. 103175, Oct. 2021, doi: 10.1016/J.JTRANGEO.2021.103175.

[8] A. Orlowski and P. Romanowska, “Smart Cities Concept: Smart Mobility Indicator,” *https://doi.org/10.1080/01969722.2019.1565120*, vol. 50, no. 2, pp. 118–131, Feb. 2019, doi: 10.1080/01969722.2019.1565120.

[9] C. Benevolo, R. P. Dameri, and B. D’Auria, “Smart Mobility in Smart City,” *Lecture Notes in Information Systems and Organisation*, vol. 11, pp. 13–28, Jan. 2016, doi: 10.1007/978-3-319-23784-8_2.

[10] A. Freitas, L. Brito, K. Baras, and J. Silva, “Smart mobility: A survey,” *Internet of Things for the Global Community, IoTGC 2017 - Proceedings*, Aug. 2017, doi: 10.1109/IOTGC.2017.8008972.

[11] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, “Microservices in Practice: A Survey Study”.

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

[13] M. Ramírez López and J. Spillner, “Towards Quantifiable Bound-aries for Elastic Horizontal Scaling of Microservices,” *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017, doi: 10.1145/3147234.

[14] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning,” *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2019-July, pp. 329–338, Jul. 2019, doi: 10.1109/CLOUD.2019.00061.

[15] D. Manuel Matos Braga Lopes, C. Bento, and E. Ricardo Machado Ubiwhere, “An Intelligent Bike-Sharing Rebalancing System Masters in Informatics Engineering Advisors,” 2015.

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

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

[18] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2003.

[19] Chris Richardson, *Microservices Patterns*.

[20] Ethan Garofolo, *Practical microservices: build event-driven architectures with event sourcing and CQRS*. The Pragmatic Bookshelf, 2020.

[21] Sam Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O’Reilly Media, Inc., 2020.

[22] F. Montesi and J. Weber, “Circuit Breakers, Discovery, and API Gateways in Microservices,” 2016.

[23] Q. A. Liang, J. Y. Chung, S. Miller, and Y. Ouyang, “Service pattern discovery of web service mining in Web service registry-repository,” *Proceedings - IEEE International*

*Conference on e-Business Engineering, ICEBE 2006*, pp. 286–293, 2006, doi: 10.1109/ICEBE.2006.90.

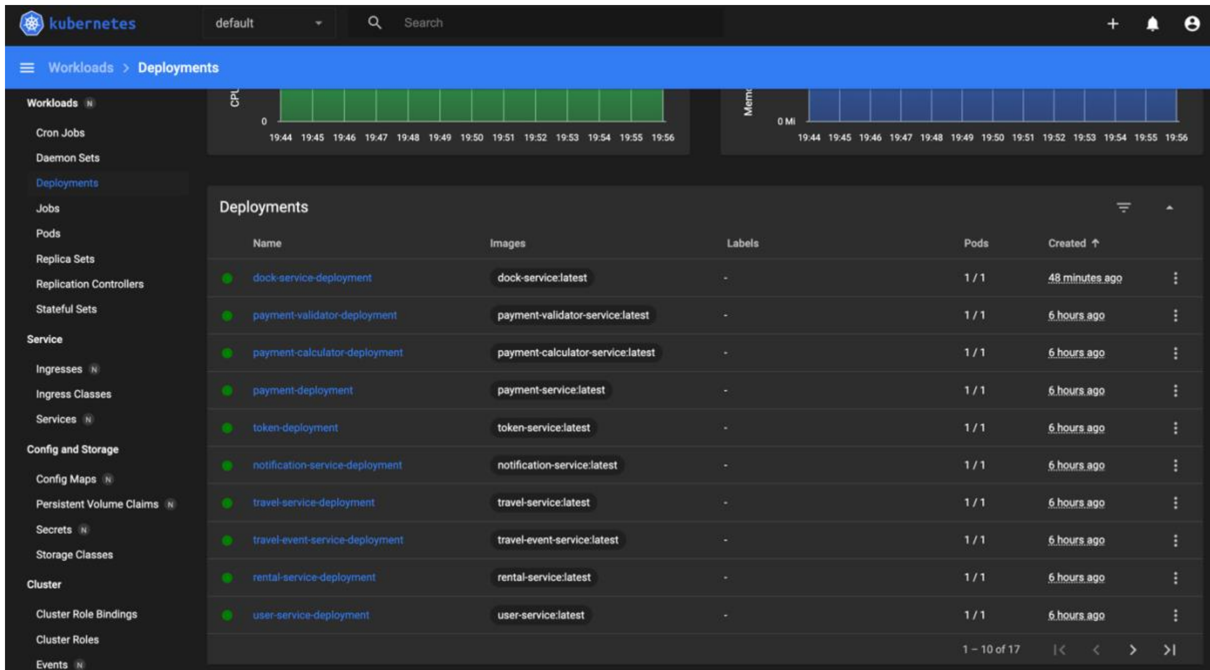
[24] M. Jones, B. Campbell, and C. Mortimore, “JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants,” May 2015, doi: 10.17487/RFC7523.

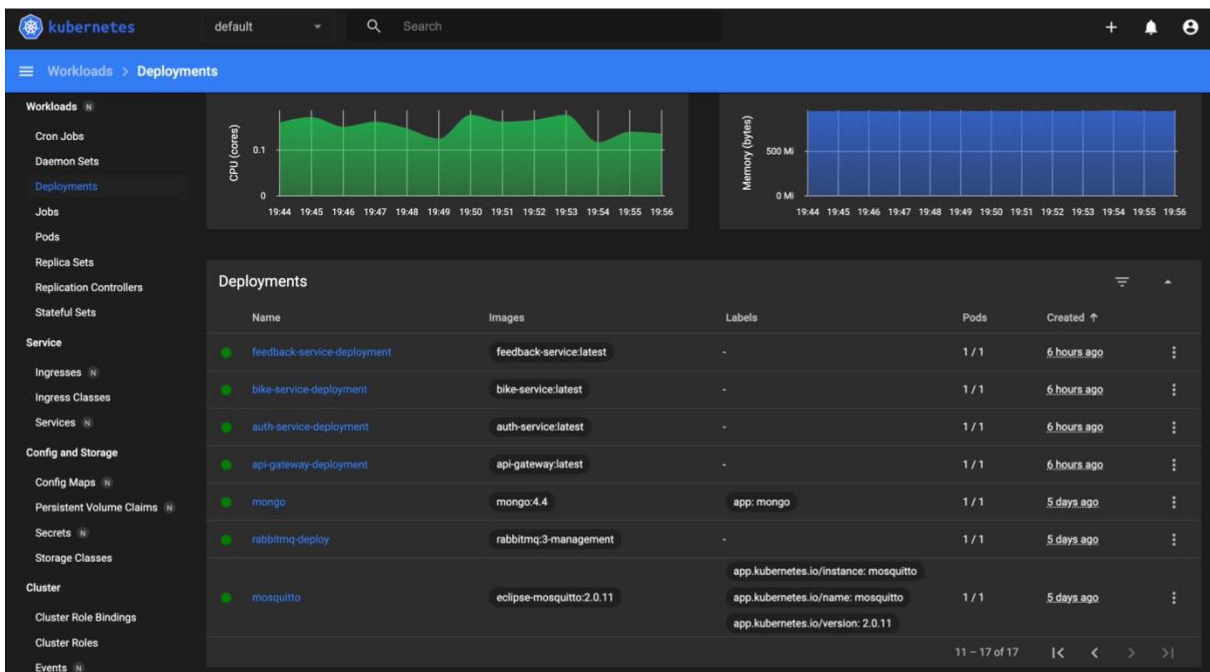
[25] J. Minkkilä, “Health Check API and Service Discovery in Microservice Environment”.

Esta página foi deixada em branco propositadamente

# Anexos

## Anexo I – Serviços a correr no *Kubernetes*



# Anexo II – Serviços registados no *service discovery*, (Eureka)

**Instances currently registered with Eureka**

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| AUTH-SERVICE | n/a (1) | (1) | UP (1) - auth-service-deployment-78b475bcc4-2bmjx:auth-service:5010 |
| BIKE-SERVICE | n/a (1) | (1) | UP (1) - bike-service-deployment-68bd6f867f-cchps:bike-service:5020 |
| DOCK-SERVICE | n/a (1) | (1) | UP (1) - dock-service-deployment-58cb8ffb6f-nwdt2:dock-service:6010 |
| FEEDBACK-SERVICE | n/a (1) | (1) | UP (1) - feedback-service-deployment-f8bff7cc-kzsts:feedback-service:5030 |
| NOTIFICATION-SERVICE | n/a (1) | (1) | UP (1) - notification-service-deployment-7d885bb467-tnhg4:notification-service:5080 |
| PAYMENT-CALCULATOR-SERVICE | n/a (1) | (1) | UP (1) - payment-calculator-deployment-7c859846c-6g7ml:payment-calculator-service:6040 |
| PAYMENT-SERVICE | n/a (1) | (1) | UP (1) - payment-deployment-74d545fb9b-bm5mg:payment-service:6030 |
| PAYMENT-VALIDATOR-SERVICE | n/a (1) | (1) | UP (1) - payment-validator-deployment-5fd75cc7c9-nqmmx:payment-validator-service:6020 |
| RENTAL-SERVICE | n/a (1) | (1) | UP (1) - rental-service-deployment-6c97f944bb-gjfxb:rental-service:5050 |
| TOKEN-SERVICE | n/a (1) | (1) | UP (1) - token-deployment-67c9787575-4rt82:token-service:5090 |
| TRAVEL-EVENT-SERVICE | n/a (1) | (1) | UP (1) - travel-event-service-deployment-6444cc57f-cj57t:travel-event-service:6050 |
| TRAVEL-SERVICE | n/a (1) | (1) | UP (1) - travel-service-deployment-7d4b867f5b-92bcc:travel-service:5060 |
| USER-SERVICE | n/a (1) | (1) | UP (1) - user-service-deployment-9dbc98f69-7lv78:user-service:5040 |

**General Info**

| Name | Value |
|---|---|
| total-avail-memory | 52mb |
| num-of-cpus | 1 |
| current-memory-usage | 24mb (46%) |
| server-uptime | 16:04 |

# Anexo III – Endpoints REST de cada serviço

Swagger
Select a definition AuthService v1

## AuthService v1 OAS3

/swagger/v1/swagger.json

### Auth

POST /api/Auth

Swagger
Select a definition BikeService v1

## BikeService v1 OAS3

/swagger/v1/swagger.json

### Bikes

GET /api/Bikes

POST /api/Bikes

GET /api/Bikes/{id}

PUT /api/Bikes/{id}

Swagger
Select a definition DockService v1

## BikeService v1 OAS3

/swagger/v1/swagger.json

### Docks

GET /api/Docks/nearby

GET /api/Docks

POST /api/Docks

GET /api/Docks/{id}

PUT /api/Docks/{id}

DELETE /api/Docks/{id}

GET /api/Docks/bike/{bikeId}

Swagger
Select a definition FeedbackService v1

## FeedbackService v1 OAS3

/swagger/v1/swagger.json

### Feedbacks

POST /api/Feedbacks

GET /api/Feedbacks

Swagger
Supported by SMARTBEAR

Select a definition PaymentService v1

# PaymentService v1 OAS3

/swagger/v1/swagger.json

## Payments

GET /api/Payments/rental/{rentalId}

GET /api/Payments/{id}

PUT /api/Payments/{id}

POST /api/Payments

Swagger
Supported by SMARTBEAR

Select a definition RentalService v1

# RentalService v1 OAS3

/swagger/v1/swagger.json

## Rentals

GET /api/Rentals/{id}

PUT /api/Rentals/{id}

GET /api/Rentals

POST /api/Rentals

Swagger
Supported by SMARTBEAR

Select a definition TravelService v1

# TravelService v1 OAS3

/swagger/v1/swagger.json

## Travel

POST /api/Travel

GET /api/Travel/rental/{rentalId}

Swagger
Supported by SMARTBEAR

Select a definition UserService v1

# UserService v1 OAS3

/swagger/v1/swagger.json

## Users

GET /api/Users/me

PUT /api/Users/me

POST /api/Users

POST /api/Users/me/credit-cards

GET /api/Users/me/credit-cards

DELETE /api/Users/me/credit-cards/{creditCardNumber}

# Anexo IV – Pipeline de CI no Github

Search or jump to... Pull requests Issues Marketplace Explore

danielpinto8zz6 / BikeShare Private

Unwatch 1 Fork 0 Star 0

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

Workflows New workflow

All workflows

.NET Core

## All workflows

Showing runs from all workflows

Filter workflow runs

47 workflow runs Event Status Branch Actor

| Run | Branch | Time |
|---|---|---|
| Fixes<br>.NET Core #47: Commit 2d5618a pushed by danielpinto8zz6 | master | 23 hours ago<br>1m 58s |
| update deployment configurations<br>.NET Core #46: Commit 796b751 pushed by danielpinto8zz6 | master | 2 days ago<br>2m 7s |
| fix configs<br>.NET Core #45: Commit b79fc7e pushed by danielpinto8zz6 | master | 2 days ago<br>2m 19s |
| Fix unit tests<br>.NET Core #44: Commit cc07f04 pushed by danielpinto8zz6 | master | 7 days ago<br>1m 59s |
| mqtt config update<br>.NET Core #43: Commit 8f8d4ae pushed by danielpinto8zz6 | master | 10 days ago<br>2m 17s |
| fix<br>.NET Core #42: Commit 0db6ecc pushed by danielpinto8zz6 | master | 10 days ago<br>1m 51s |