BIKESHARING EM MICROSSERVIÇOS

Métodos e Técnicas de Suporte ao Desenvolvimento de Software

Grupo 10
Daniel Pinto – 8200865
Tiago Pinto – 8150459
Carlos Barbosa – 8150243

Escola Superior de Tecnologia e Gestão
Mestrado em Engenharia Informática

Janeiro de 2021

Índice

1. Introdução

Os últimos anos têm gerado um grande alvoroço à volta dos microsserviços. As aplicações, cada vez mais complexas, tornaram-se mais difíceis de gerir, de atualizar e de manter, levando a indústria a procurar novos paradigmas para o desenvolvimento de *software*.

A arquitetura de microsserviços propõe-se resolver os problemas clássicos das aplicações monolíticas, dividindo-as em blocos ou serviços independentes. Esses serviços, modulares, podem ser implementados e geridos individualmente, orientando as equipas de desenvolvimento para o *continuous delivery*.

Uma das empresas de referência na adoção e implementação de uma arquitetura de microsserviços é a Netflix, que em 2014 disponibilizou o Netflix OSS, um conjunto de *frameworks* e bibliotecas desenvolvidas para resolver alguns problemas comuns em sistemas distribuídos em escala. Atualmente, o Netflix OSS é sinónimo de desenvolvimento de microsserviços em ambiente *cloud*.

Desenvolvida e mantida pela Pivotal, Spring é uma *framework* Java criada com o objetivo de facilitar o desenvolvimento de aplicações, explorando, para isso, os conceitos de Inversão de Controlo e Injeção de Dependências. A Spring Cloud possui uma *stack* completa de microsserviços, cujo objetivo é fornecer uma integração completa entre o Spring Boot e o projeto Netflix OSS. *Service Discovery* (Eureka), Circuit Breaker (Hystrix/Turbine), *Intelligent Routing* (Zuul) e *Client Side Load Balancing* (Ribbon), são alguns dos padrões do Netflix OSS que o Spring Cloud fornece.

Considerando as premissas elencadas, e aproveitando a orientação do docente de Métodos e Técnicas de Suporte ao Desenvolvimento de Software para a elaboração do projeto descrito ao longo deste documento, elegeu-se, como *stack* tecnológica, a Spring Cloud para o desenvolvimento de uma aplicação de microsserviços para a gestão de uma plataforma de *bike sharing*.

O tema surgiu devido a alguma proximidade de um dos elementos do grupo com a implementação, em contexto profissional, de uma plataforma de *bike sharing* no Município de Lousada. Esta proximidade permitiu perceber algumas das regras do negócio e consequentemente, inferir que esta seria uma área de atuação enquadrável na arquitetura de microsserviços. A vontade de “fazer melhor” foi, também, um catalisador na escolha do tema.

2. Caso de estudo

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 carbono é um dos eixos do quadro de planos integrados de mobilidade sustentável, onde se inserem as plataformas de Bicicletas Partilhadas (*bike sharing*).

A plataforma de *bike sharing* assenta a sua operacionalidade num conjunto de bicicletas elétricas, parqueadas em estações de ancoragem para bicicletas (docas).

As docas – normalmente agrupadas em estações com capacidade para dez bicicletas –, além de servirem de ponto de carregamento, são a interface de levantamento e de entrega de cada bicicleta. Como tal, estão distribuídas por pontos estratégicos na cidade, garantindo-se que existem mais pontos de ancoragem do que bicicletas, promovendo-se, desta forma, a liberdade de mobilidade em cada aluguer.

Os utilizadores do serviço de *bike sharing* podem utilizar uma bicicleta mediante um registo numa aplicação *mobile*. Depois de registados, têm a possibilidade de consultar, de uma lista de bicicletas disponíveis, os detalhes da bicicleta (marca, modelo, nível de bateria) que querem alugar, podendo ser conduzidos até ao ponto de levantamento (doca). Esta funcionalidade implica a aceitação, por parte do utilizador, do acesso da aplicação ao GPS do dispositivo móvel.

O levantamento é feito depois de digitalizado um código QR estampado em cada bicicleta, que faz com que a doca desacople a bicicleta em questão. 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.

A devolução é efetuada numa das docas que compõem o ecossistema de *bike sharing* e, finalizado o aluguer, assim que o utilizador acople novamente a bicicleta. O tempo de utilização é calculado considerando a data e hora de levantamento e de entrega, sendo taxado da seguinte forma:

- 0,50€/minuto durante os 15 primeiros minutos;
- 1,00€/minuto, nos minutos seguintes.

Assim, no final de cada utilização, é apresentada ao utilizador a quantia a liquidar pelo aluguer e um formulário de avaliação da experiência, onde aquele pode indicar algum problema relacionado com a bicicleta utilizada ou com as docas de levantamento e/ou entrega.

Na secção seguinte apresentam-se diagramas de atividades dos processos de negócio por forma a descrever as interações do utilizador com a aplicação *mobile*.

3. Processos de negócio

Para responder ao caso de estudo descrito, foi considerado o desenvolvimento de uma aplicação android que respondesse aos requisitos aflorados. De forma muito generalista, os processos inerentes à aplicação em causa encontram-se ilustrados pela Figura 1, Figura 2, Figura 3 e Figura 4.

A primeira diz respeito ao processo de reserva de uma bicicleta e inicia-se com o login e autenticação do utilizador, que o leva, posteriormente, a uma página que lista as bicicletas disponíveis e passíveis de serem reservadas. Numa primeira fase, este processo foi pensado de modo que a reserva fosse válida durante cinco minutos, período durante o qual a bicicleta não poderia ser levantada por nenhum outro utilizador. No entanto, e com o decorrer da implementação, este mecanismo levantava algumas questões relacionadas com a fluidez do processo, com a otimização dos recursos do ponto de vista do negócio – poderiam existir reservas que não fossem efetivadas em alugueres e que alocavam uma bicicleta que nunca seria usada; as reservas poderiam ser encadeadas, o que permitiria que, no limite, um utilizador reservasse indefinidamente uma bicicleta que nunca chegava a usar – e, do ponto de vista da arquitetura, punha em causa a independência dos microsserviços envolvidos neste processo. A sua redefinição resulta no diagrama ilustrado pela Figura 2.

A Figura 3, intitulada numa fase embrionária de "utilização", pretende descrever o processo de aluguer. Este começa com a digitalização de um código QR que, depois de validado, permite libertar uma bicicleta de uma doca e, assim, se dar início ao processo de "utilização"/aluguer durante o qual são recolhidos dados de GPS relativos ao trajeto efetuado. Posteriormente, estes dados permitirão o acesso ao histórico do utilizador. O mesmo processo engloba, também, a entrega da bicicleta numa das docas do ecossistema de *bike sharing* em apreço.

A Figura 4 pretende descrever o processo de pagamento de um aluguer e caracteriza-se pela existência de um subprocesso ("Calcular valor a pagar") que implementa a lógica descrita na secção anterior, nomeadamente a que respeita à aplicação das tarifas definidas. Depois de calculado o pagamento, o utilizador é notificado do valor a pagar e, depois de saldado o valor do aluguer, convidado a deixar uma avaliação da sua experiência de utilização.

Os processos identificados foram, salvo as alterações ligeiras referenciadas no processo de reserva, implementados com base nos diagramas descritos.

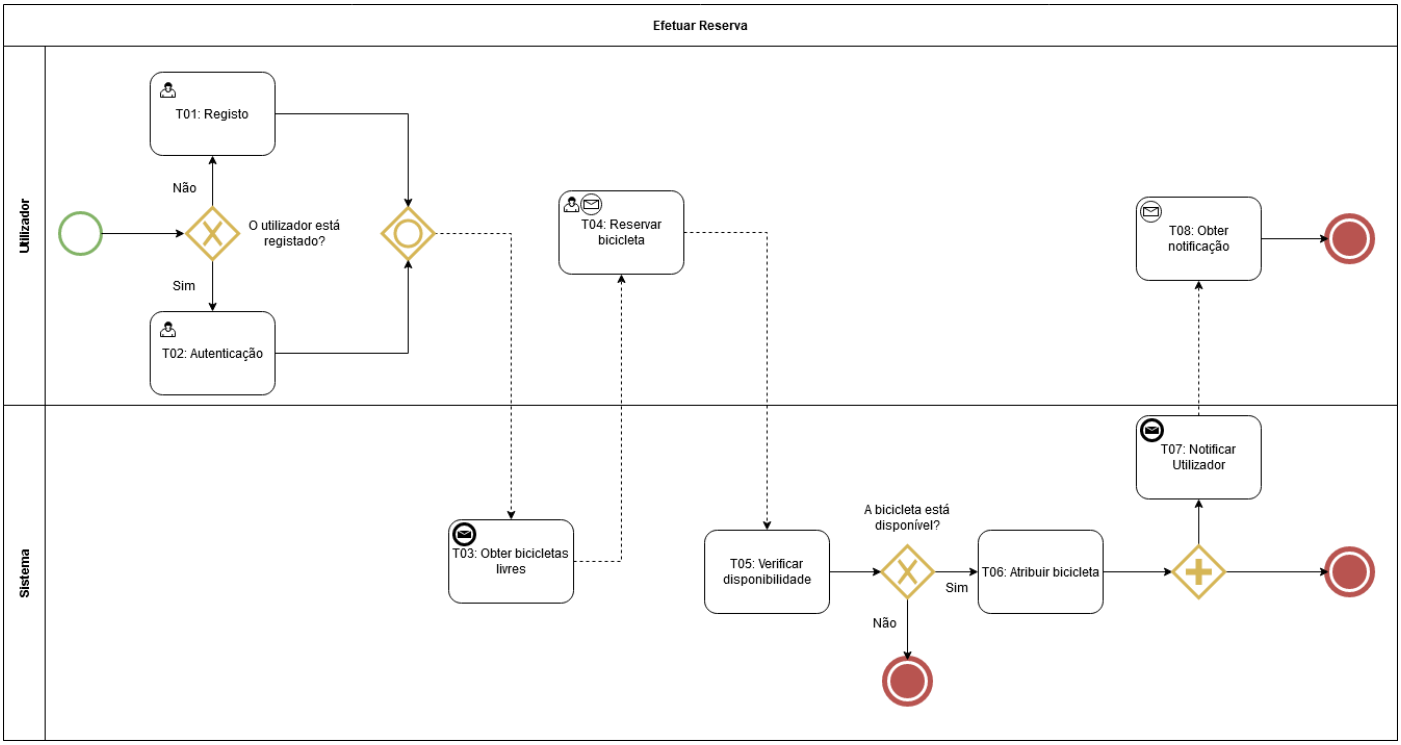

Figura 1 –Efetuar Reserva

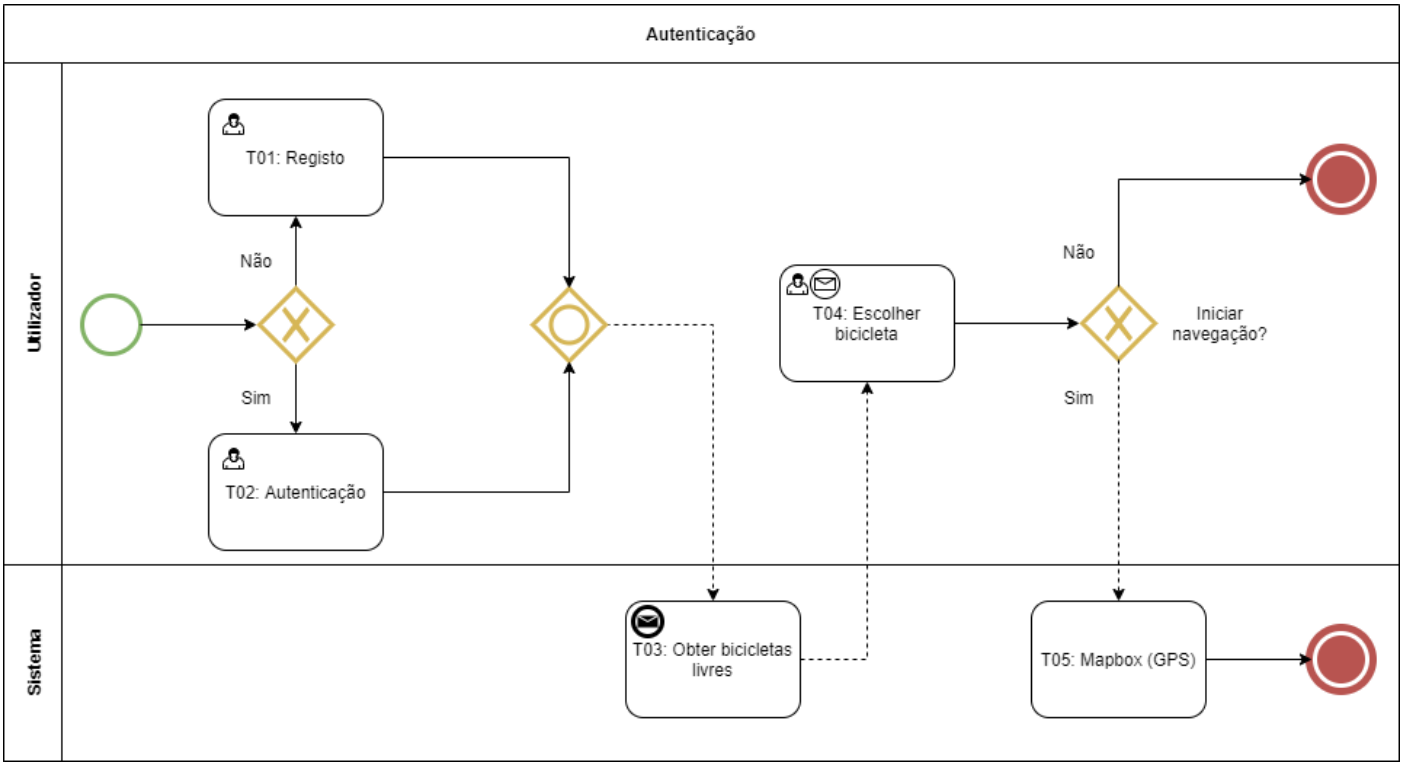

Figura 2 –Autenticação

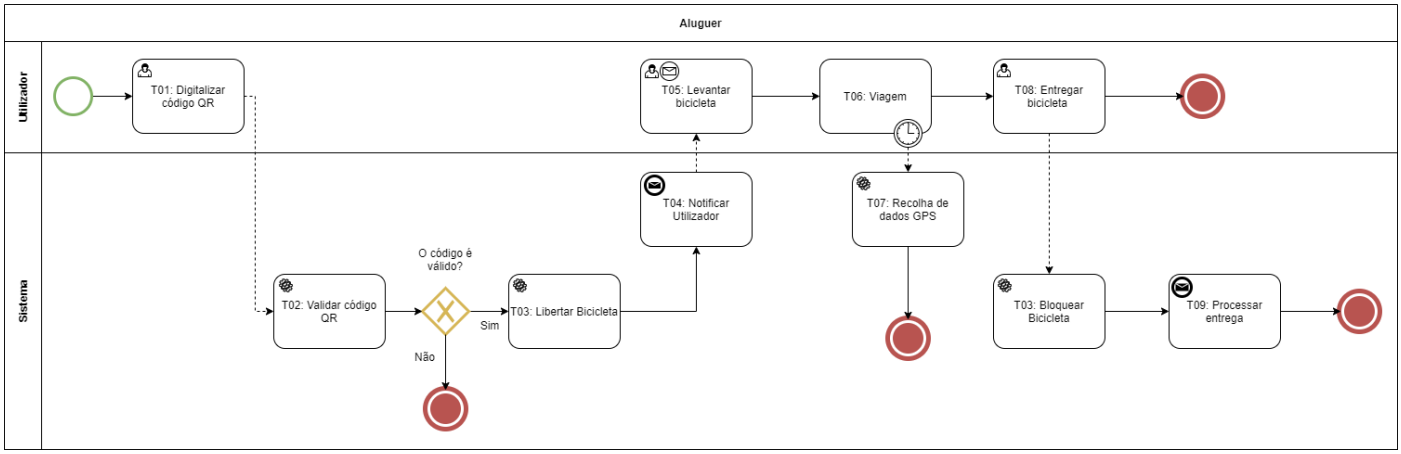

Figura 3 – Aluguer

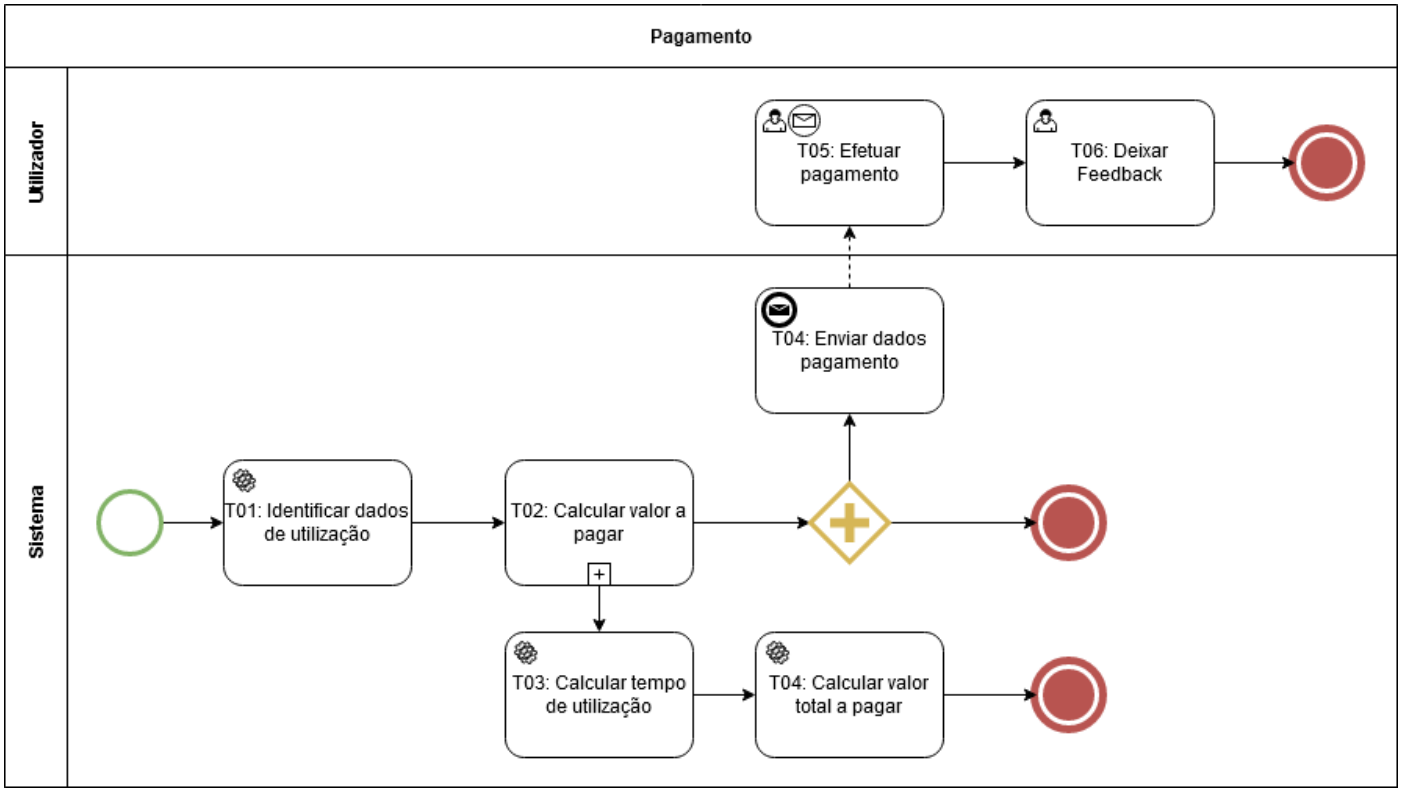

Figura 4 - Pagamento

4. 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.

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
- Auth
- Bike
- Dock
- Rental
- Payment
- Feedback
- Travel history
- Dummy
- Notification
- Token manager

Cada microsserviço 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 microsserviços 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.

5. Ambiente de desenvolvimento

O *Vagrant* é uma ferramenta para criar e gerir ambientes de máquina virtual num único fluxo de trabalho. Centrado na automação, o *Vagrant* reduz o tempo de configuração do ambiente de desenvolvimento, aumenta a paridade de produção e resolve os problemas associados à expressão "Funciona na minha máquina!".

Conscientes desta problemática, antes de se escrever qualquer linha de código, houve a preocupação de se criar um ambiente de desenvolvimento comum aos elementos do grupo. No entanto, considerando as inúmeras ferramentas a utilizar, e não obstante da existência de inúmeras *boxes* em repositórios públicos, decidiu-se pela criação de uma máquina virtual Xubuntu 20.04 que tivesse, nativamente, o conjunto de ferramentas necessárias para o desenvolvimento do projeto. Assim, a *box* utilizada – e que pode ser obtida aqui – foi configurada[1] com o seguinte software:

- Java OpenJDK (versão 8 e 11)
- OpenSSH
- Git
- IntelliJ IDEA
- Visual Studio Code
- Postman
- Docker
- Minikube
- Kubectl
- VBox Guest Tools

Depois de obtido ficheiro *package.box* referenciado anteriormente, a execução dos comandos que se seguem, num terminal na pasta onde se encontra o ficheiro *package.box*, cria uma máquina virtual *Vagrant* no *Virtual Box*:

```
vagrant box add xubuntu64 package.box

vagrant init ubuntu64

vagrant up
```

A máquina virtual em questão foi configurada com dois processadores, 4GB de RAM e um disco de 30GB. Apesar de se ter revelado de extrema utilidade durante a fase de desenvolvimento, à medida que a implementação de microsserviços no *minikube* começou a crescer, sentiu-se a necessidade de migrar os *deployments* para o ambiente *cloud*, uma vez que os recursos computacionais da *box* começaram a esgotar-se rapidamente.

Esta migração passou por dois momentos distintos, que se traduzem nas plataformas de *cloud* utilizadas. No primeiro momento, utilizou-se o *Microsoft Azure*, fazendo uso dos USD$100 de créditos concedidos para contas de estudante. No entanto, este plafond rapidamente se esgotou (cerca de duas semanas) o que conduziu à migração de todos os microsserviços para a *Google Cloud Platform* (GCP), onde, à data da redação e defesa do projeto, se encontram alojados.

Da utilização de ambos os serviços, e embora os créditos de trial concedidos pela GCP sejam o triplo dos de *Azure*, a comparação de custos entre ambas as soluções de *cloud* é favorável à GCP.

[1] https://blog.engineyard.com/building-a-vagrant-box

O acesso ao cluster de *Kubernetes* do projeto pode ser efetuado depois de instalado o *Google Cloud SDK*. A Google disponibiliza-o aqui[2].

Depois de instalado, é necessário efetuar o login – Figura 5 – com uma das contas associadas ao projeto, executando, num terminal, o comando `gcloud auth login`.

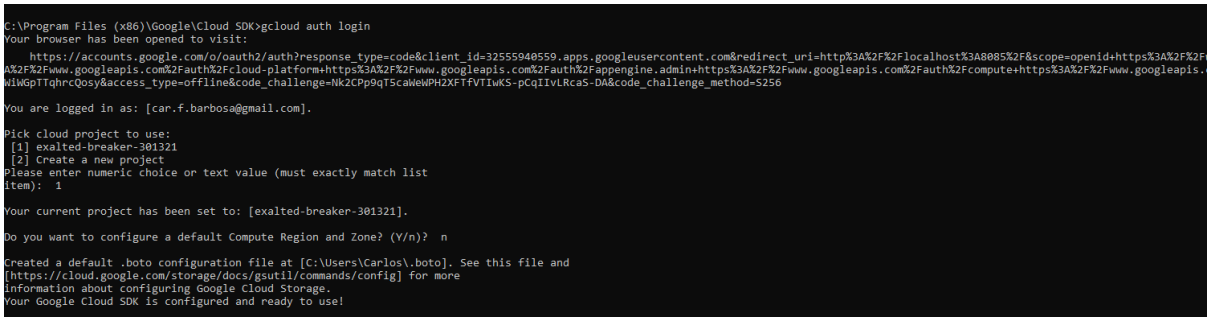

Figura 5 – Autenticação gcloud

Após a autenticação há que instalar os componentes necessários para a utilização do Kubernetes a partir da máquina local. Assim, num terminal com privilégios, executa-se o comando `gcloud components install kubectl` como ilustra a Figura 6.

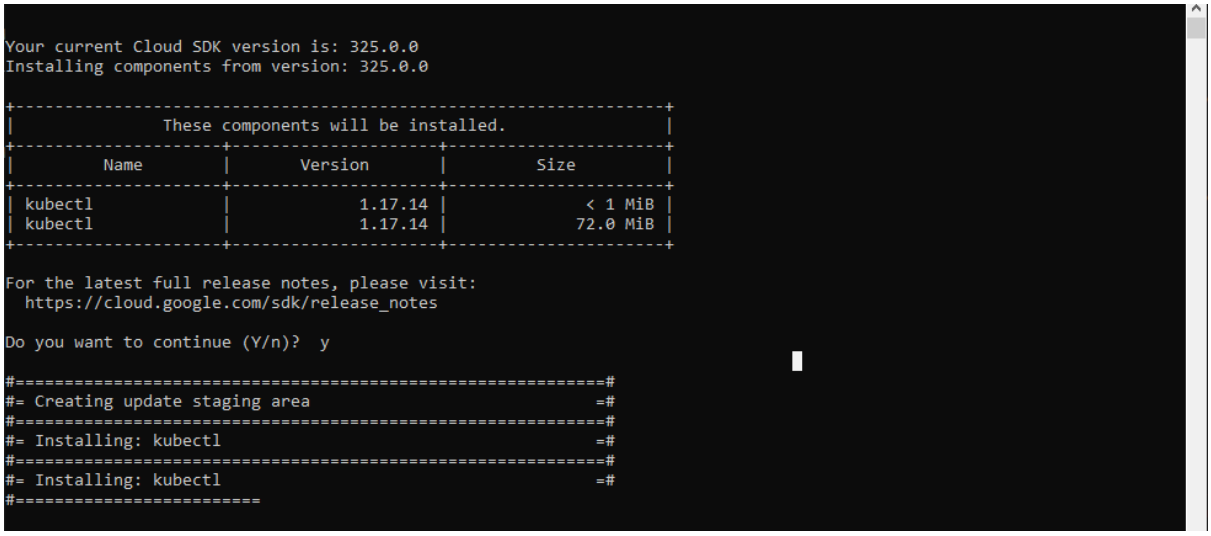

Figura 6 – Instalação kubectl

Concluída a instalação, é necessário executar o comando `gcloud container clusters get-credentials mtsds-bikeshare-0` para que seja criado o ficheiro de configuração que armazena as informações relativas à conectividade ao cluster de *Kubernetes* (nome do cluster, zona, endereço IP, certificado, *token* de autenticação, etc.). O comando `kubectl proxy` permite que o *dashboard* do *Kubernetes* possa ser acedido localmente através do endereço disponível aqui. O *token* de autenticação encontra-se disponível na raiz do repositório do *gitlab* sob o nome *k8s_token*.

[2] https://cloud.google.com/sdk/docs/downloads-interactive

6. Arquitetura de Microsserviços

A Figura 7 ilustra a arquitetura de microsserviços do projeto, especificada com maior detalhe no capítulo 7, e pretende fornecer uma visão global da implementação. De referir que para o desenvolvimento do projeto se recorreu à *framework* Spring, no que aos microsserviços diz respeito, e ao *android*, para o *frontend*.

Numa primeira fase, tinha sido pensado o desenvolvimento de uma aplicação web. No entanto, e considerando a natureza do caso de estudo e do desafio que representava, no início do projeto, o desenvolvimento de uma aplicação *android*, decidiu-se seguir este caminho.

6.1. Frontend

As preocupações subjacentes ao desenvolvimento do *frontend* relacionam-se diretamente com o tipo de negócio em causa. Assim sendo, tentou-se garantir a fluidez de interação entre o utilizador e a aplicação e a mobilidade que a plataforma *android* consegue oferecer.

O *frontend* é a interface de utilizador para o acesso aos microsserviços através de uma API *gateway* (*zuul*). A sua implementação faz uso da *Retrofit* API, desenvolvida pela *Square* para comunicações REST. Esta API fornece um padrão simples de implementação para transmissão de dados entre aplicação e servidor, usando JSON, e é uma ótima opção quando há necessidade de integração entre uma aplicação e os serviços disponibilizados no *backend*.

De referir, também, a utilização do *mapbox*, uma plataforma para tratar dados de localização em ambiente *web* e *mobile*, no mapeamento de alugueres e navegação de utilizadores, quer durante um aluguer, quer na condução do utilizador até à bicicleta que vai alugar.

6.2. Microsserviços

As subsecções seguintes descrevem, detalham e enquadram cada um dos microsserviços implementados sob a perspetiva da sua funcionalidade no ecossistema da aplicação.

6.2.1. eureka-server

As arquiteturas de microsserviços 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[3] 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*. Assim, todos os microsserviços, à exceção dos SGBDs e respetivos serviços de gestão (*pgadmin* e *mongoexpress*) implementam um *Eureka Client*.

[3] A escalabilidade horizontal é abordada com mais detalhe em secção própria.

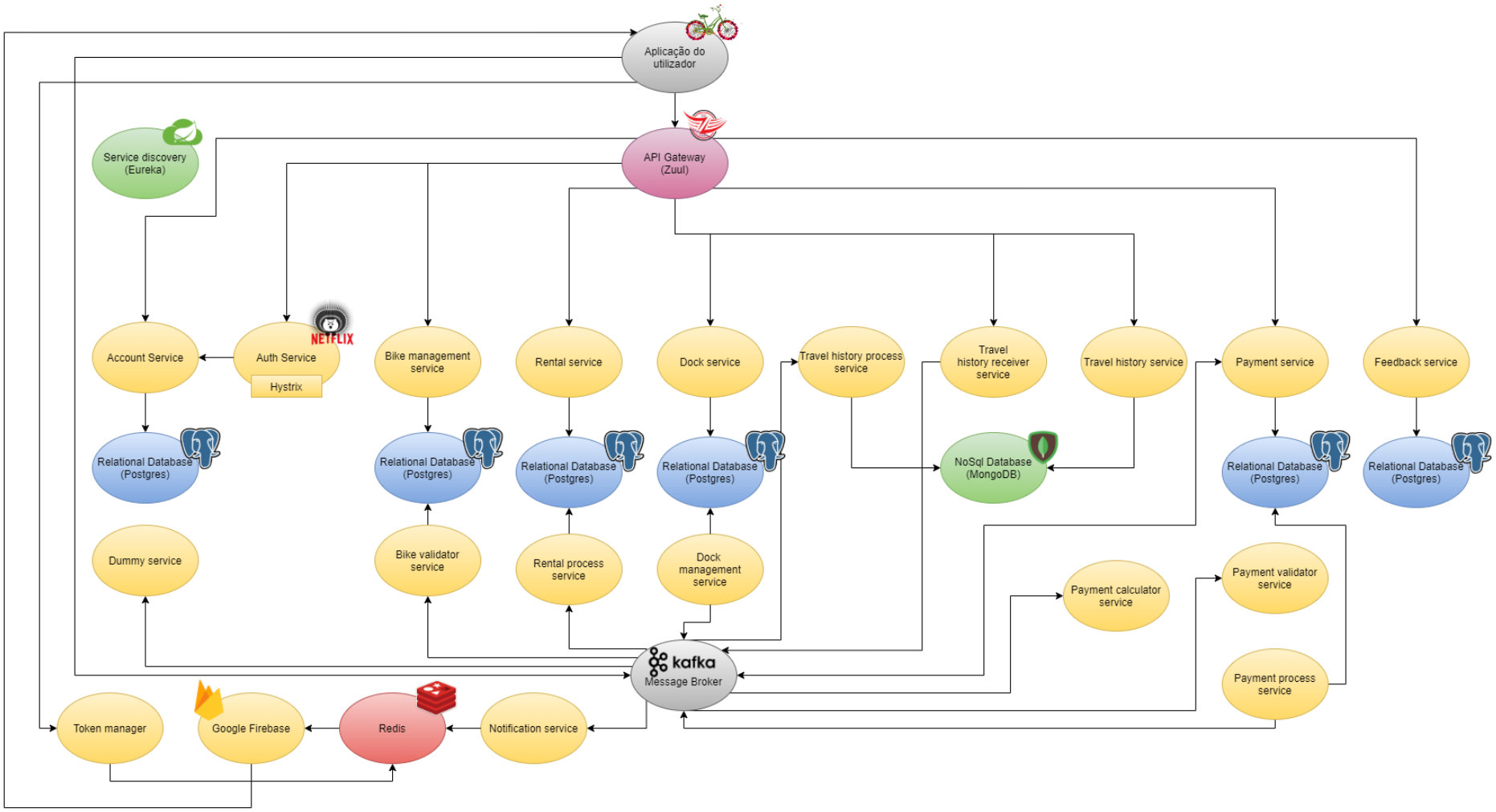

Figura 7 – Arquitetura de Microsserviços

6.2.2. zuul-server

O *Zuul*, tal como o Eureka, é um módulo do Netflix OSS que atua como uma API *gateway* ou *Edge service*, recebendo todas as solicitações provenientes da interface do utilizador (UI) e, delegando essas mesmas solicitações aos microsserviços internos. Portanto, o *Zuul* escuda todos os outros microsserviços, expondo a sua API ao cliente/UI como um proxy para os microsserviços internos, fornecendo uma "porta de entrada" unificada para o ecossistema do projeto. Na verdade, é este serviço que permite que a aplicação android consuma serviços de vários *hosts/*microsserviços.

O Zuul integra com outros componentes da *stack* Netflix OSS, como o *Hystrix* e o *Eureka*, e é usado no projeto para gestão de regras de roteamento e balanceamento de carga.

Como também é um microsserviço, o *Zuul* é escalável de forma independente como os seus pares.

6.2.3. postgres-account

Base de dados em PostgreSQL que armazena as informações relativas aos utilizadores e acedida, apenas, pelo microsserviço *account-service*.

6.2.4. pgadmin-service

O *pgadmin* é uma ferramenta de gestão com interface gráfico do PostgreSQL e foi implementada como microsserviço para facilitar a gestão das bases de dados de utilizadores, bicicletas, docas, alugueres, pagamentos e *feedback*.

De forma centralizada, a plataforma *pgadmin* permite efetuar *queries* SQL a estas bases de dados e, em contexto de desenvolvimento, revelou-se uma ferramenta de grande utilidade na análise e despiste de problemas associados à comunicação e persistência da informação, essencial para o correto funcionamento de toda a arquitetura.

6.2.5. account-service

Este microsserviço é responsável pela gestão de utilizadores e contém o CRUD de operações na base de dados PostgeSQL que os armazena, comunicando apenas por *REST* nos seguintes *endpoints*:

- POST em **/account/user** para registo do utilizador;
- PUT em **/account/user** para atualização do registo, mais concretamente o nome do utilizador;
- PUT em **/account/user/password** para atualização da *password*;
- GET em **/account/user/{email}** para obtenção das informações relativas ao utilizador.
-

6.2.6. auth-service

Este microsserviço é responsável pela autenticação e geração de *tokens* JWT, comunicando diretamente com o *account-service*, por REST, para obter informações do utilizador. Caso o login seja bem-sucedido, retorna um *token* JWT que permite fazer pedidos autenticados aos restantes serviços. Comunica apenas por *REST*.

6.2.7. postgres-bike

Base de dados em PostgreSQL que armazena as informações relativas às bicicletas e acedida pelos microsserviços *bike-management-service* e *bike-validator-service*.

6.2.8. bike-management-service

Microsserviço responsável pela gestão de bicicletas e que contém o *CRUD* de operações na base de dados PostgeSQL que as armazena. Comunica apenas por *REST*.

6.2.9. bike-validator-service

O *bike-validator-service* é utilizado para validar se o código da bicicleta corresponde ao código de registo da base de dados. Pensado para ser implementado em conjunto com a digitalização de um código QR, valida a introdução de um código de bicicleta que permite, em caso de validação bem-sucedida, ativar o mecanismo de libertação de uma bicicleta presente em cada uma das docas e simulado, no projeto, pelo *dummy-service*.

6.2.10. postgres-dock

Base de dados em PostgreSQL que armazena as informações relativas às docas e acedida pelos microsserviços *dock-management-service* e *dock-service*.

6.2.11. dock-service

É o microsserviço responsável pela listagem de docas com bicicletas acopladas ao utilizador, permitindo-lhe uma visão de todas as bicicletas em carregamento e, consequentemente passíveis de serem alugadas, num determinado momento. Comunica apenas por *REST* e liga-se diretamente à base de dados de docas.

6.2.12. dock-management-service

Este microsserviço é responsável pela gestão de docas e é utilizado para associar e dissociar uma bicicleta de uma determinada doca quando começa e/ou termina um aluguer, comunicando diretamente com a base de dados de docas.

6.2.13. dummy-service

Este microsserviço simula o mecanismo eletrónico de uma doca e é responsável por trancar e destrancar uma bicicleta, comunicando apenas por tópicos *kafka*, sendo simultaneamente *producer* e *consumer*.

6.2.14. postgres-rental

Base de dados em PostgreSQL que armazena as informações relativas aos alugueres e é acedida pelos microsserviços *rental-process-service* e *rental-service*.

6.2.15. rental-service

Este microsserviço é o *entry point* para início de um aluguer. Recebe, por REST, o pedido de aluguer de uma bicicleta, cria uma entrada na base de dados para o aluguer com o estado "validando aluguer" e envia uma mensagem para o *bike-validator-service* que irá validar se o código da bicicleta inserido pelo utilizador corresponde ao da base de dados.

6.2.16. rental-process-service

Microsserviço responsável por processar um aluguer, completando-o com a data de fim e enviando uma mensagem ao *payment-process-service*, através do *kafka,* para que se inicie o processo de pagamento.

6.2.17. mongo

Coleção de documentos em *mongoDB* que armazena as informações relativas aos dados de GPS recolhidos durante um aluguer e usada pelos microsserviços *travel-history-service* e *travel-history-process-service*. O primeiro é responsável pela recolha de dados e o segundo pela exibição do histórico de localizações na listagem de alugueres efetuados.

6.2.18. mongoexpress

Na mesma senda do *pgadmin*, o *mongoexpress* é uma ferramenta de gestão com interface gráfico, mas para o *mongoDB*. Tal como o *pgadmin*, revelou-se uma ferramenta de grande utilidade na análise e despiste de problemas associados à comunicação e persistência da informação relativa aos dados de GPS recolhidos

6.2.19. travel-history-service

É o microsserviço responsável por permitir ao utilizador obter, por *REST* os dados de GPS relativos a determinado aluguer. Acede diretamente à coleção *travel-history* em *mongoDB* e não efetua mais nenhum tipo de comunicação.

6.2.20. travel-history-process-service

Este microsserviço é responsável por armazenar os dados GPS enviados pelo utilizador. Lê os dados GPS através de um tópico *kafka* e processa-os na coleção *travel-history*.

6.2.21. travel-history-receiver-service

Microsserviço que serve como *entry point REST* para o envio de dados GPS. Perante o consentimento do utilizador, o *frontend* envia coordenadas GPS de dois em dois segundos para este serviço, que os publica num tópico *kafka*, posteriormente consumido e processado pelo *travel-history-process-service*.

6.2.22. postgres-payment

Base de dados em PostgreSQL que armazena as informações relativas aos pagamentos e é acedida pelos microsserviços *payment-service* e *payment-process-service*.

6.2.23. payment-service

É o microsserviço responsável por receber pedidos de pagamento, por *REST*, efetuados pelo utilizador, encaminhando-os, através de um tópico de *kafka*, para serem processados.

Permite também obter os pagamentos efetuados por determinado utilizador e comunica diretamente com a base de dados PostgreSQL.

6.2.24. payment-validator-service

Este microsserviço recebe mensagens de pedidos de pagamento de um tópico *kafka* e efetua a validação do pagamento. De seguida, envia uma mensagem, novamente via *kafka*, para o *payment-process-service*, que irá atualizar o estado do pagamento, caso este seja, ou não, bem-sucedido.

6.2.25. payment-calculator-service

Serviço responsável por fazer o cálculo do valor a pagar por parte do utilizador mediante a duração do aluguer de uma bicicleta. Quando o aluguer é fechado, é enviada uma mensagem por *kafka* para este serviço, que com base na duração do aluguer, calcula o valor e envia-o para o *payment-process-service* que irá gerar o pagamento com o valor calculado.

6.2.26. payment-process-service

Este serviço comunica com a base de dados PostgreSQL.de pagamentos, recebendo mensagens com o valor do pagamento e o aluguer a que pertence. É responsável por criar uma entrada de pagamento na base de dados, colocando o seu estado em “aguardando pagamento”. Envia, posteriormente, uma notificação para o utilizador, através do *notification-service*, para que este salde o pagamento.

Este serviço é também responsável por alterar o estado do pagamento.

6.2.27. redis

O *redis* (***RE****mote* ***DI****ctionary* ***S****erver*) é uma solução distribuída de armazenamento de dados muito popular nos últimos tempos. Além de ser fácil de usar, suporta vários tipos de estruturas que permitem suprir a grande maioria das necessidades de dados. Os dados são armazenados sob a forma de chave-valor, fazendo lembrar a estrutura de *Dictionary* do .net e de *Map* do Java, e podem ter diferentes formatos: *strings, hashes, lists, sets* e *sets* ordenados.

O *redis* é extremamente rápido, tanto em escrita como em leitura de dados, graças ao facto de os armazenar em memória (apesar de permitir que os dados sejam persistidos fisicamente). Por esse motivo, por ser muito utilizado como servidor de cache aplicacional e por permitir que uma chave expire após um determinado período, o *redis* foi a base de dados NOSQL escolhida para ser utilizada na gestão das sessões de utilizador do projeto.

6.2.28. token-manager

Microsserviço responsável por guardar, no dicionário redis, a associação entre o email do utilizador e o *token firebase* da sua aplicação *android*. Quando o utilizador se autentica na aplicação, é gerado um *token*, enviado para este serviço por REST, que o guarda associado ao seu email. Deste modo, sempre que é necessário notificar o utilizador na aplicação *android*, indica-se ao *notification-service* o email do utilizador a notificar, que, consultando o *redis*, obtém o *token* referente àquele email e notifica a aplicação corretamente pelo *firebase*.

6.2.29. notifications-service

Este serviço é responsável pelo envio de notificações a determinado utilizador, comunicando diretamente com o *redis*, o *firebase* e o *kafka*, de quem recebe mensagens para que proceda ao envio de notificações. Depois de consultar o dicionário *redis* à procura do *token* correspondente ao email do utilizador a notificar, envia uma mensagem ao *firebase*, que posteriormente notificará o utilizador na aplicação android. A descrição deste processo de comunicação detalha-se no capítulo 7.

6.2.30. postgres-feedback

Base de dados em PostgreSQL que armazena as informações relativas aos feedbacks deixados pelos utilizadores no final de cada aluguer e respetivo pagamento. É acedida, apenas, pelo microsserviço *feedback-service*.

6.2.31. feedback-service

Microsserviço responsável pela receção e processamento do *feedback* do utilizador. Liga-se diretamente à base de dados de *feedback* e recebe pedidos *REST*.

7. Comunicação e processos

Esta secção descreve detalhadamente mecanismos, tecnologias e processos de comunicação utilizados. No paradigma de microsserviços, 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 microsserviços, detalhada adiante neste capítulo, de onde sobressai a utilização do *Apache Kafka* e do *Firebase*.

O *Apache Kafka* é uma plataforma distribuída de transmissão de dados que é capaz de publicar, subscrever, armazenar e processar fluxos de dados provenientes de diversas fontes e entregá-los a vários clientes. Resumindo, o Kafka movimenta grandes volumes de dados não apenas do ponto A ao ponto B, mas para qualquer outro ponto, simultaneamente. Desenvolvido pela *LinkedIn* para processar 1,4 biliões de mensagens por dia, é uma solução de transmissão de dados *open source* aplicável a várias necessidades corporativas.[4]

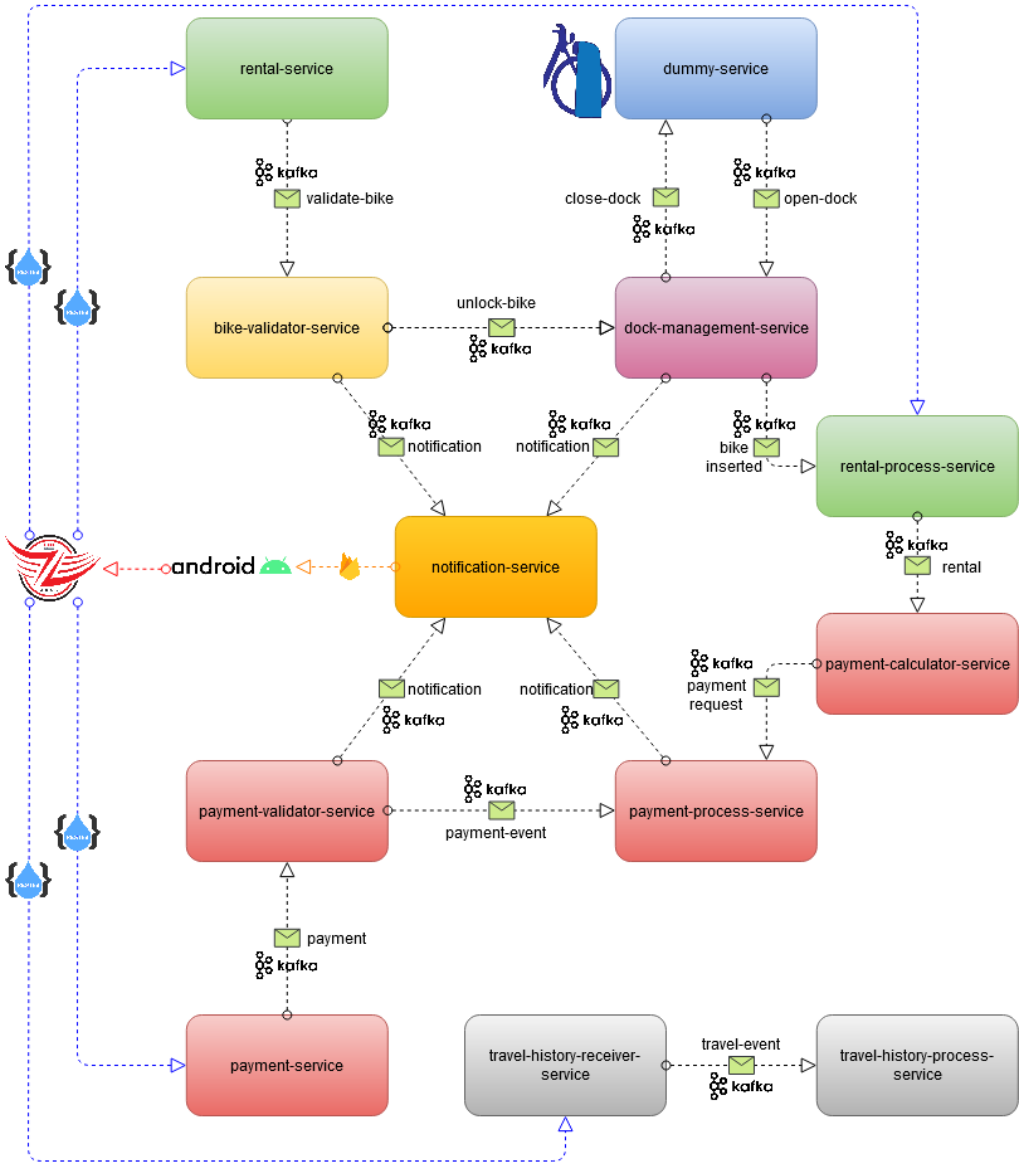

Figura 8 – Diagrama de comunicação entre microsserviços

[4] https://www.redhat.com/pt-br/topics/integration/what-is-apache-kafka

O *Apache Kafka* é habitualmente incorporado em pipelines de transmissão que partilham dados entre sistemas e/ou aplicações, bem como em sistemas e aplicações que consomem esses dados. É, como tal, compatível com vários casos de uso em que a produtividade e escalabilidade são fatores vitais. Como minimiza a necessidade de integrações *point-to-point* (P2P) para a partilha de dados em determinadas aplicações, potenciando a troca assíncrona de mensagens, o *Kafka* reduz a latência das comunicações a milésimos de segundos. Isto significa que os dados são disponibilizados mais rapidamente aos utilizadores, o que é uma vantagem para os casos de uso que exigem disponibilidade de dados em tempo real.

O *Firebase* é uma plataforma de desenvolvimento *mobile* e *web* adquirida pela Google em 2014 e é um *backend* completo que disponibiliza diversos serviços que auxiliam o desenvolvimento e a gestão de aplicações. Um dos serviços disponibilizados por esta plataforma, implementado e referenciado no diagrama da Figura 8 é o *Firebase Cloud Messaging* (FCM), uma solução para envio de mensagens entre plataformas.

O FCM permite o envio notificações *push* aos utilizadores de aplicações através de uma API. 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 do aplicativo e conecta-se ao dispositivo móvel apenas quando há novas mensagens.

A Figura 9 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. De facto, este serviço consome mensagens do tópico *notification* do *kafka*, e obtém, do dicionário *redis,* o *token* do utilizador, de que se serve para notificar a aplicação *android* fazendo uso do FCM.

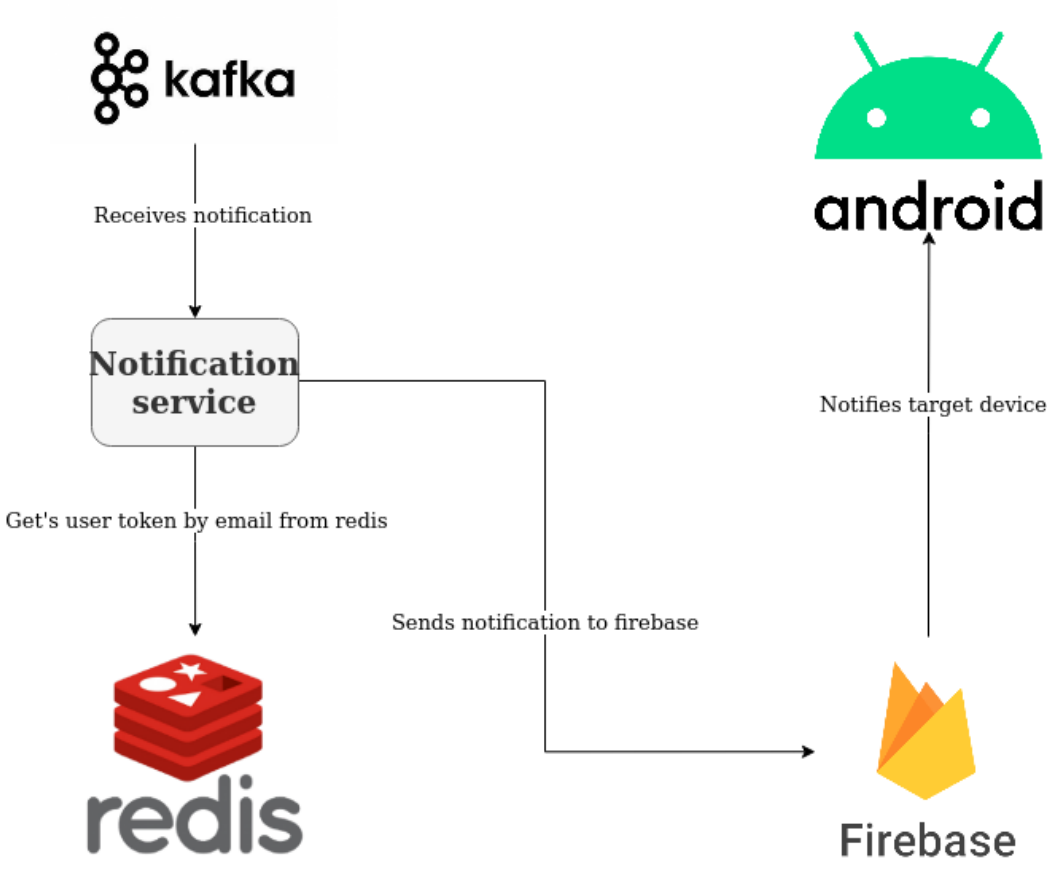

Figura 9 – Processo de notificações

A Figura 10 documenta o processo de autenticação de um utilizador. Se o utilizador não estiver registado, é chamado o *account-service*, que, depois de validados os *inputs*, adiciona o registo à base de dados *account* em PostgreSQL.

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 *openfeign*, uma biblioteca que permite fazer uma abstração das comunicações REST por meio de interfaces.

Para cada autenticação válida é gerado um *token* JWT que, por opção de implementação, nomeadamente no que respeita à articulação com a aplicação android desenvolvida, não expira.

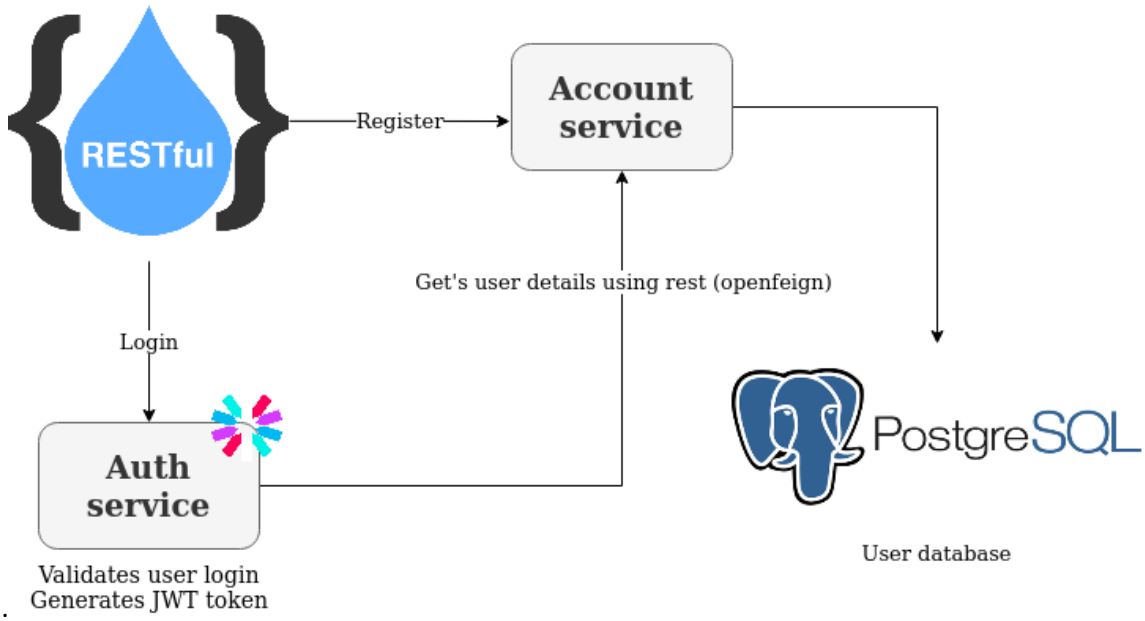

Figura 10 – Processo de autenticação

Depois de autenticado – Figura 11 –, o utilizador inicia a sua experiência de utilização sendo-lhe mostradas as bicicletas disponíveis para aluguer naquele momento. Esta listagem resulta de um pedido REST ao *dock-service* que por sua vez consulta a base de dados de docas, retornando apenas as que têm bicicletas.

Da lista resultante da consulta, o utilizador pode selecionar uma bicicleta na localização que lhe seja mais favorável e, através da introdução de um código de desbloqueio, é feito um pedido ao *rental-service* para se dar início ao aluguer de uma a bicicleta.

Por sua vez, o *rental-service* criará uma entrada de aluguer na base de dados de alugueres e publicará uma mensagem no *kafka* que será consumida pelo *bike-validator-service*. Este serviço consulta a base de dados de bicicletas e verifica se o código de desbloqueio digitalizado corresponde ao definido naquela base de dados. Caso o código de desbloqueio não tenha correspondência, o aluguer é cancelado e enviada uma notificação ao utilizador, através do *kafka*, que será lida e tratada pelo *notification-service*.

No caso de o código ser válido, este mesmo serviço notificará o *dock-management-service*, novamente servindo-se do *kafka*, para que este desassocie a bicicleta da doca e persista essa alteração na base de dados.

De seguida o *dock-management-service* publica duas mensagens: a primeira para ser consumida pelo *dummy-service*, que simulará a abertura da doca, e a segunda para ser consumida pelo *notification-service* que notificará o utilizar do início do aluguer.

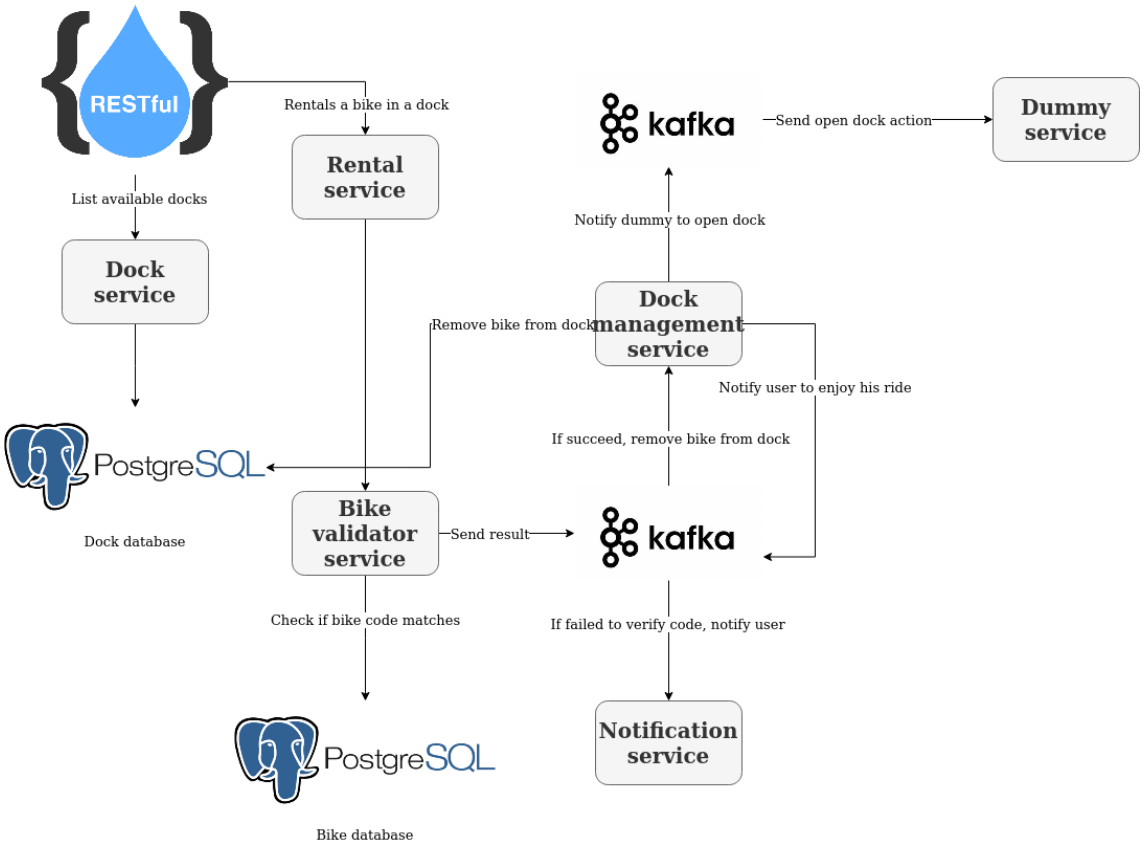

Figura 11 – Processo de aluguer

O processo de entrega e instrução de pagamento ilustrado pela Figura 12 inicia-se pela invocação efetuada pelo *dummy-service* ao *dock-management-service,* através da publicação de uma mensagem no *kafka*. A mensagem consumida por este último serviço contém a identificação quer da bicicleta quer da doca onde é feita a entrega. Com esta informação é atualizada a base de dados de docas, reassociando a bicicleta a uma doca – a de entrega – e notificado o *rental-process-service*, através de uma mensagem *kafka*, de que o aluguer foi terminado.

O *rental-process-service* é responsável por registar na base de dados de alugueres a conclusão do mesmo e por notificar o *payment-calculator-service*, novamente servindo-se do *kafka*, com os detalhes do aluguer, em específico o seu início e o seu término. Este serviço calcula o valor do aluguer de acordo com as regras definidas para o negócio e notifica o *payment-process-service* do valor a pagar pelo utilizador.

Com esta informação, o *payment-process-service* regista o pagamento na base de dados de pagamentos e informa o *notification-service*, novamente através de uma mensagem no *kafka*, que este deve notificar o utilizador com o valor calculado para aquele aluguer.

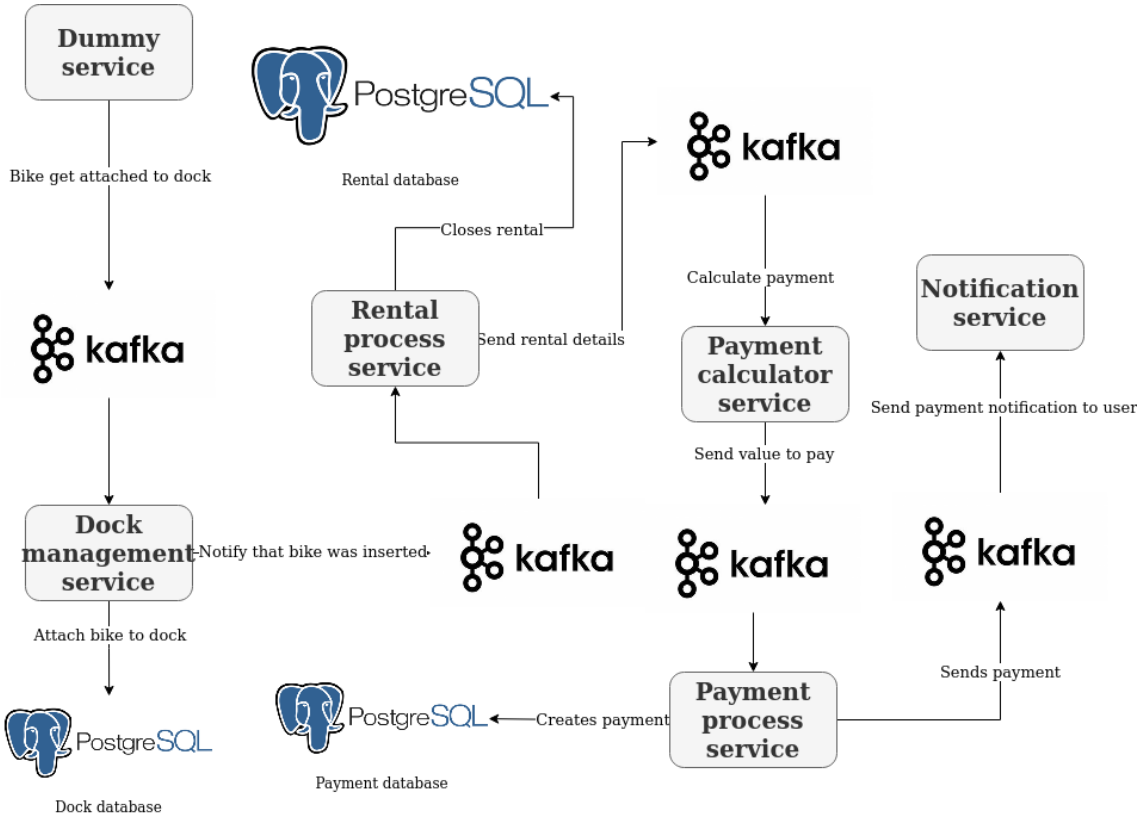

Figura 12 – Processo de entrega e instrução de pagamento

A Figura 13 demonstra o processo de pagamento de um aluguer. Os dados de pagamento são submetidos ao *payment-service*, através de um pedido REST, que os adiciona à fila de pagamentos, consumida pelo *payment-validator-service*. Este serviço simula a validação dos dados introduzidos para pagamento e poderia, em produção, ser um serviço *third-party* responsável pela transação financeira.

O *payment-validator-service* é responsável pela publicação de uma mensagem *kafka* consumida, quer pelo *notification-service*, quer pelo *payment-process-service*. O primeiro consome a mensagem se a validação do pagamento não for efetuada com sucesso e notifica o utilizador do insucesso da transação. O segundo, se a validação do pagamento for bem-sucedida, atualiza o estado do pagamento na base de dados de pagamentos e notifica, também, o utilizador de que o aluguer foi pago com sucesso.

O processo de recolha de dados GPS que ocorre durante um aluguer está documentado pela Figura 14. Este processo não deve, todavia, ser dissociado da utilização destes dados para a produção de um histórico de alugueres de um utilizador e inicia-se com chamadas REST, de 2 em dois segundos, ao *travel-history-receiver-service* contendo dados de GPS. Estes dados são enviados para uma fila do *kafka*, e processados assincronamente pelo *travel-history-process-service* que os adiciona a uma coleção do *mongoDB*.

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-history-service*, que devolve os dados GPS relativos àquele aluguer.

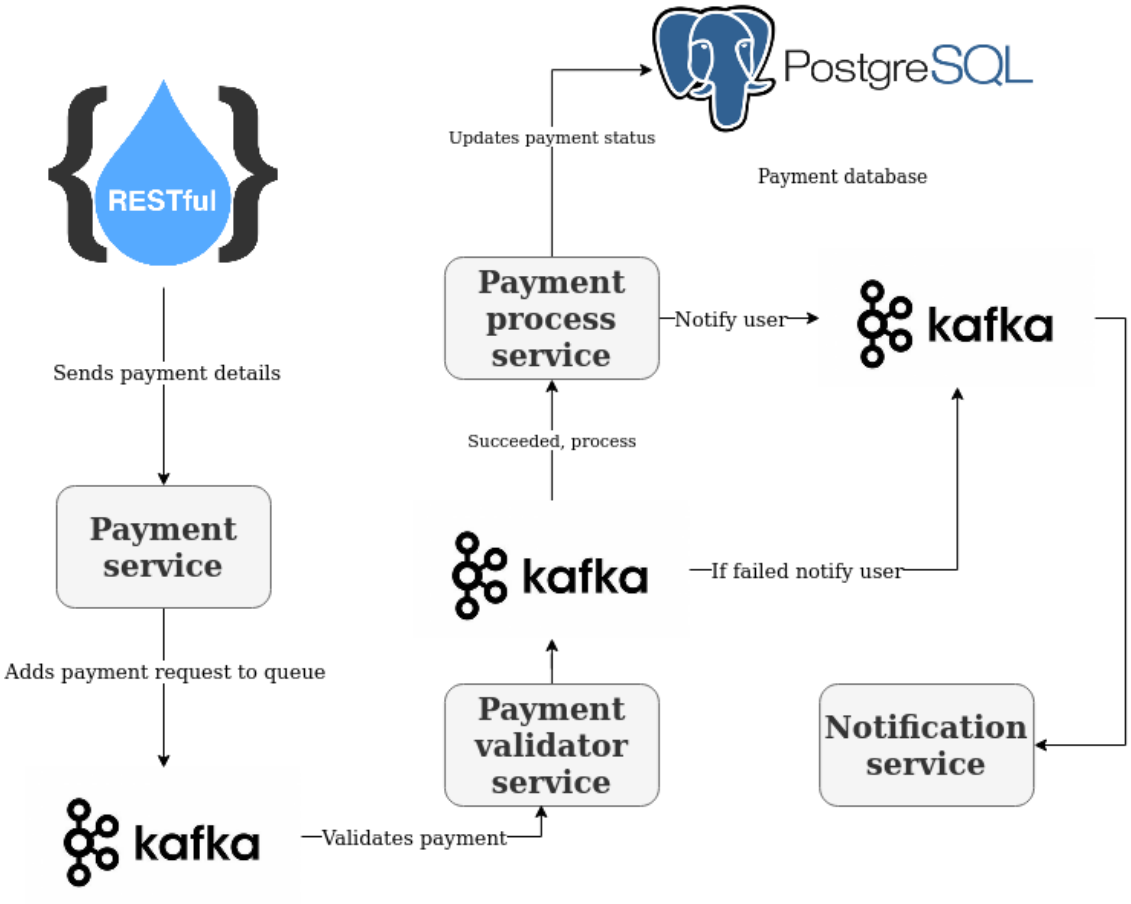

Figura 13 – Processo de pagamento

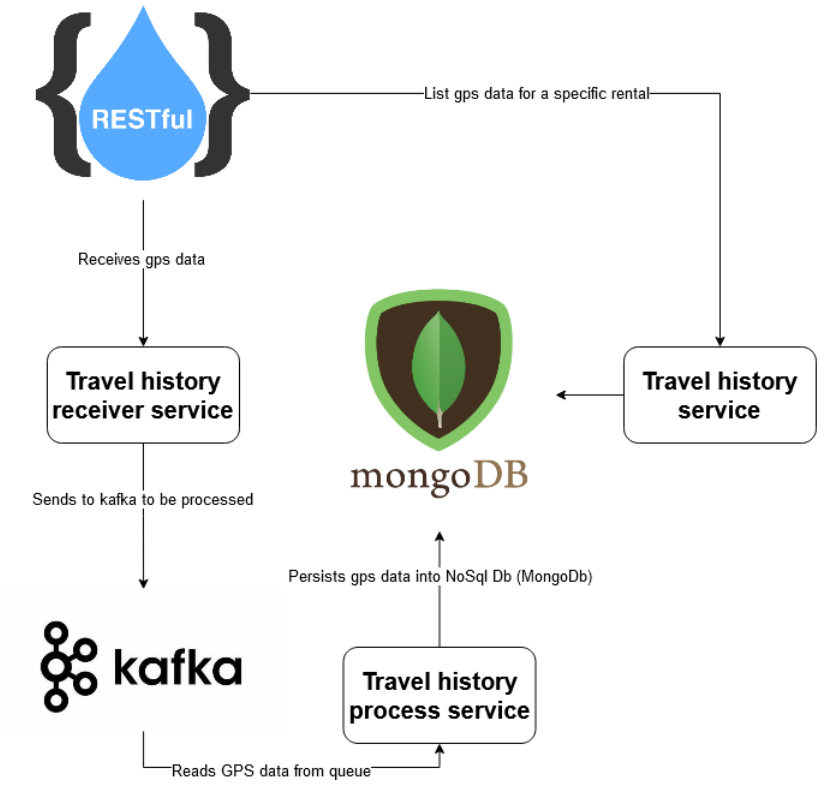

Figura 14 – Processo de histórico de dados de aluguer

8. Persistência de dados

Num ambiente baseado em *containers*, a persistência de dados é sempre uma preocupação primordial uma vez que o fim de vida de um *container* significa o mesmo destino para os dados que este armazena localmente.

Esta é uma das premissas da utilização do *Docker*: os containers são descartáveis porque foram desenvolvidos para atender um processo durante um determinado período temporal e não para serem usados como servidores virtuais. Portanto, é imperioso garantir a persistência dos dados – em volumes e/ou sistemas de armazenamento distribuídos – necessária ao funcionamento de uma aplicação, de modo que o desaparecimento de um container não arraste consigo a informação imprescindível ao seu funcionamento e, de forma mais abrangente, ao funcionamento do ecossistema do qual faz parte.

À medida que o projeto foi escalando em dimensão, foram surgindo desafios que puseram em causa a garantia da persistência de dados almejada, nomeadamente aqueles armazenados nas bases de dados.

Na primeira fase, enquanto o projeto foi desenvolvido dentro dos limites da máquina virtual *Vagrant* referenciada no capítulo 5; o orquestrador de *containers* se resumia ao *minikube* e o cluster de *kubernetes* não o era porque tinha apenas um nó, a persistência de dados foi relativamente fácil de alcançar: independentemente do número de vezes que um *pod* morresse e voltasse a ser lançado pelo *kubernetes*, como o armazenamento era local à máquina virtual do *minikube*, estava sempre disponível proporcionando uma sensação de persistência, que se alterou quando o ambiente de desenvolvimento necessitou de ser escalado para um ambiente de *cloud*.

Em qualquer dos provedores de *cloud* utilizados (*Azure* e GCP), a criação de um *cluster* de *Kubernetes* tem, por defeito, três nós. Neste cenário, a persistência de dados em armazenamento local pregou algumas partidas. Parecia funcionar numas vezes, noutras não. A explicação era afinal simples: os dados eram armazenados nos nós onde um determinado *pod* estava a ser executado e quando se dava um reinício e o nó que servia de *host* ao *pod* se alterava, o armazenamento local mudava para o mesmo nó de execução do *pod*. Este comportamento tornou-se demasiado evidente com a escalabilidade do cluster de *Kubernetes* para oito nós (Figura 15), *setup* que se mantém à data na GCP.

Com oito nós e com algumas tentativas de se criarem vários *Persistent Volume Claims* sobre um único *Persistent Volume* – esta é uma relação do tipo um para um, e não muitos para um – a solução passou por criar um volume com 10GB de capacidade por base de dados PostgeSQL e um outro para suporte ao MongoDB, conforme documenta a Figura 16. Desta forma assegurou-se a persistência de dados necessária perante a variedade de nós e a possível escalabilidade horizontal de cada um dos microsserviços, descrita no capítulo seguinte.

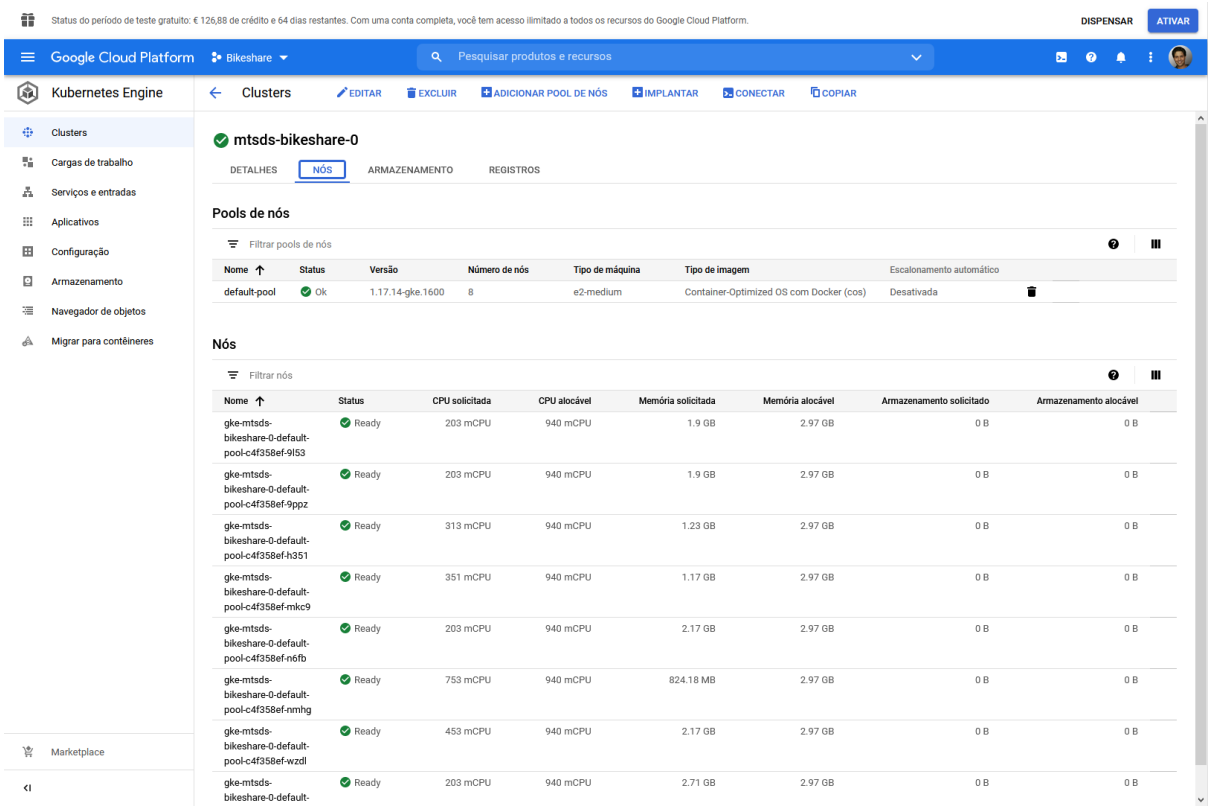

Figura 15 – Cluster de Kubernetes na GCP

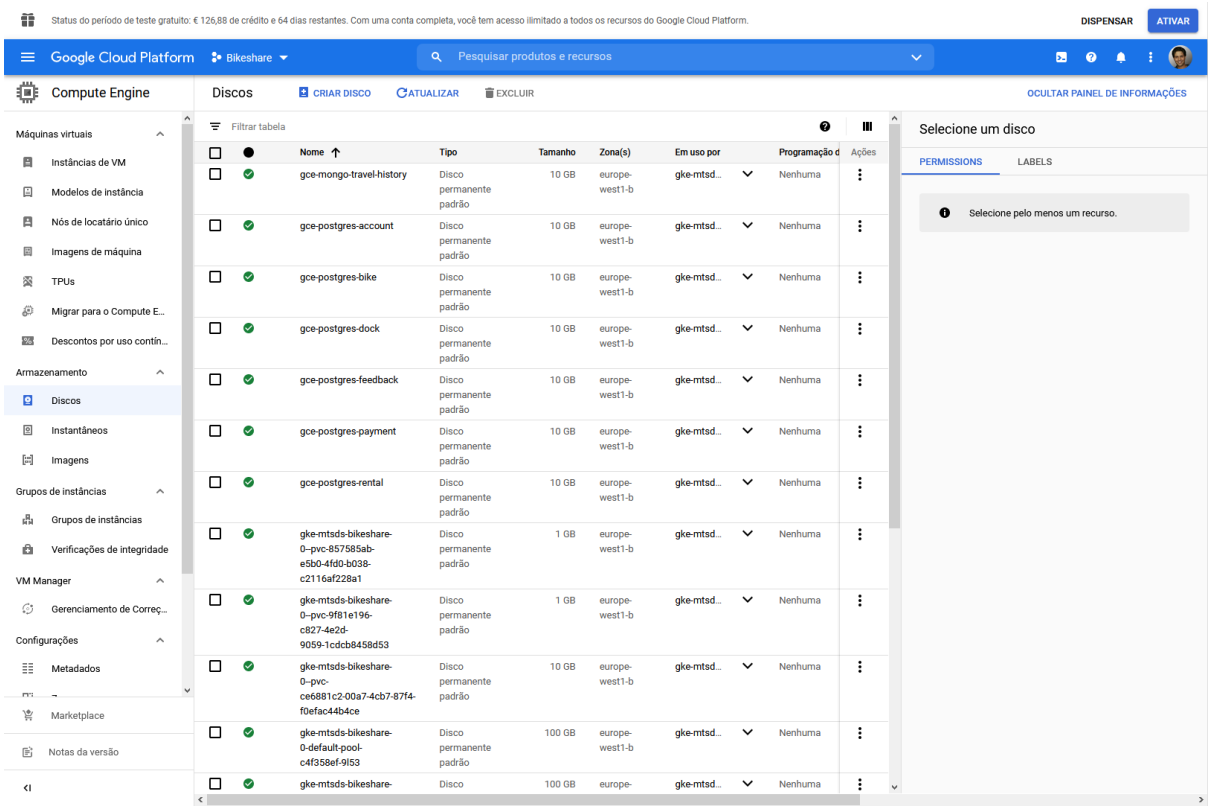

Figura 16 – Armazenamento GCP

9. Escalabilidade

Uma das grandes vantagens da arquitetura de microsserviços 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.

De forma simplista, a escalabilidade de microsserviços 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 microsserviço é independente.

A escalabilidade de serviços é feita em duas dimensões: vertical e horizontalmente, conforme ilustra a Figura 17. 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.

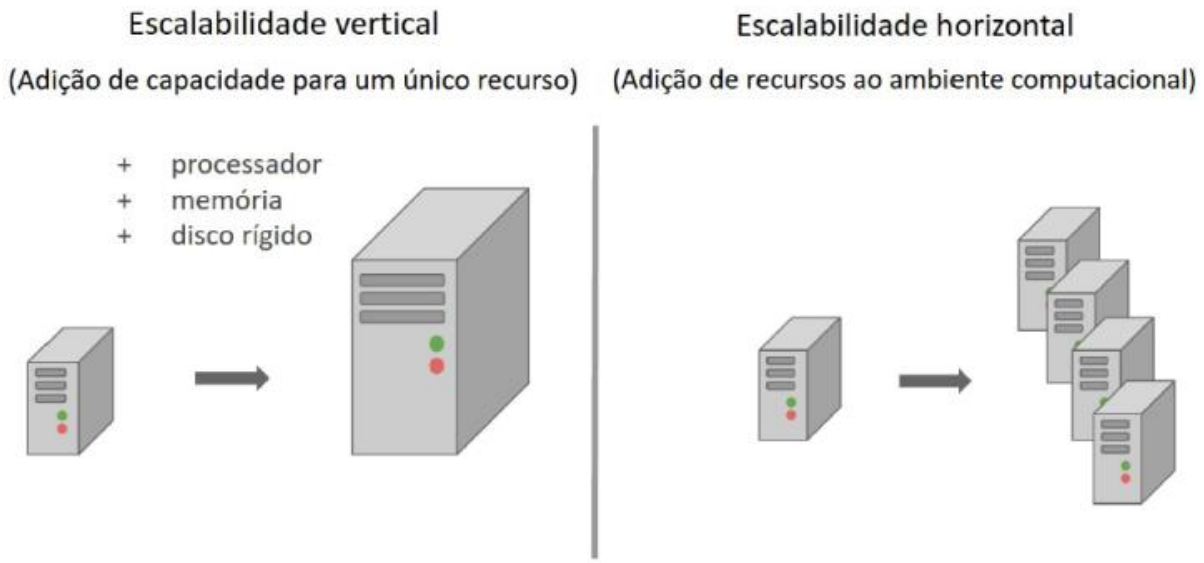

Figura 17 – Escalabilidade

Uma vez que na arquitetura de serviços do projeto são utilizados ambos os mecanismos (*Eureka* e *Zuul*, respetivamente), faz sentido que a arquitetura seja escalável horizontalmente.

Dotar o projeto deste mecanismo implicou a utilização de um servidor de métricas para o *Kubernetes* que continuamente monitoriza os recursos dos vários *pods*. Assim que são atingidos os limites (CPU e memória) definidos para que um determinado *pod* escale horizontalmente, é criada uma réplica do serviço. Em ambiente de desenvolvimento, esta funcionalidade pode ser testada recorrendo ao *Collection Runner* do *Postman*.

10. Mecanismos e recursos

Como já havia sido referenciado, houve a necessidade de recorrer aos provedores de *cloud* para implementação do projeto. Dada a sua dimensão e a necessidade de se garantir, sempre que possível, a segregação de contextos, o número de microsserviços implementados foi crescendo a um ritmo quase frenético, que não se conseguiu prever no início da implementação.

Outro dos recursos a evidenciar tem a ver com a utilização do *container registry* do *GitLab* como repositório para as imagens *Docker* de cada microsserviço. O acesso ao *registry* pelo *Kubernetes* é possível recorrendo a um *secret* que contém as credenciais do repositório.

Também foram implementados mecanismos que permitem a compilação, *upload* para o *container registry* referenciado e levantamento dos serviços através da execução de *shell scripts*, automatizando o *deployment*.

De referir que a cada microsserviço corresponde um ficheiro *deployment.yaml*, responsável pelo lançamento dos respetivos *Deployments* (ou *Stateful Sets*), *Services* e *Replica Sets*, que tem a particularidade de definir a variável de ambiente SPRING_PROFILES_ACTIVE com o valor "prod". Redefinindo-a para "dev" possibilita-se o *deployment* dos serviços num ambiente como o *minikube* em que a resolução de nomes não tem a preponderância de um ambiente "clusterizado", com vários nós, como os ambientes *cloud*, e onde os *endpoints REST* respondem por *localhost*.

Do ponto de vista funcional, foi implementada uma página web em *Spring Web* para o *dummy-service*, com o objetivo de simular o processo de entrega de uma bicicleta numa doca. Tal como o serviço que lhe está associado, a página é um *consumer* e um *producer* de tópicos *kafka*, *open-dock* e *close-dock*, respetivamente.

11. Trabalho futuro

O trabalho efetuado deixa-nos, enquanto grupo, muito confortáveis do ponto de vista da implementação. É claro que, como em todos os projetos sob o paradigma de microsserviços em que o *continuous integration* e o *continuous delivery* (CI/CD) estão sempre presentes no ciclo de vida de uma aplicação, haverá melhorias a introduzir. E algumas foram sendo identificadas ao longo da implementação:

- Implementar, no *frontend,* a funcionalidade de digitalização de um código QR para desbloquear uma bicicleta da respetiva doca, apesar de, no *backend*, existir a validação do código da mesma;
- Configurar o tempo de expiração do *token* JWT gerado na autenticação do utilizador, uma vez que, para efeitos de desenvolvimento, se tornou mais prático não o fazer;
- Implementar um servidor de configurações que permita que estas sejam geridas a partir de, por exemplo, um repositório *git*.
- Implementar serviços de monitorização e alarmística, como o *Elasticsearch Stack (Elasticsearch, Logstash, Kibana)* para *logs* e o *Prometheus* para métricas, com respetiva interface gráfica (*Kibana* e *Grafana*, no caso do *Prometheus*) que permita uma visibilidade de topo sobre o comportamento da aplicação e da infraestrutura.

12. 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 microsserviços. *Service Discovery*, *Load Balancers*, *Circuit Breakers*, *Inter Process Communication* 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 microsserviços começa a emergir como uma alternativa relevante, baseada na proposta de transformar um sistema único num conjunto de pequenos serviços, ou microsserviços, 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 na equipa – porque faz sentido chamar-lhe equipa, em detrimento de grupo – que se propôs realizar este projeto. Conscientes da complexidade de arquiteturar uma aplicação sob o paradigma de microsserviços 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.