Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Arquitetura baseada em microserviços para gestão de equipamentos informáticos na FGH

Abdul Sacur

Orientador(es) | Pedro Salgueiro

Vitor Beires Nogueira

Évora 2020

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Arquitetura baseada em microserviços para gestão de equipamentos informáticos na FGH

Abdul Sacur

Orientador(es) | Pedro Salgueiro

Vitor Beires Nogueira

Évora 2020

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	Paulo Miguel Quaresma (Universidade de Évora)
Vogais	|	Pedro Patinho (Universidade de Évora) (Arguente) Pedro Salgueiro (Universidade de Évora) (Orientador)

Évora 2020

A minha filha, Luna Sacur, você é a minha maior fonte de inspiração.

Conteúdo

Sumário

A utilização de arquitectura de microsserviços leva-nos a um novo paradigma no desenvolvimento de sistemas para Web num panorama diferente do que tem vindo ser feito nos últimos anos. Este trabalho consiste na criação de uma arquitectura baseada em microsserviços, com vista a resolver a insuficiência de uma ferramenta informatizada que ajude na monitorização de equipamentos informáticos na Friends in Global Health (FGH), que actualmente apresenta uma organização e processos complexos, se for considerada a mobilidade dos seus beneficiários tanto por transferência, fim de contratos, promoção ou despromoção. O departamento de Tecnologias da Informação (IT) da FGH é responsável pelas áreas de controlo de infraestruturas e gestão de equipamento e consumíveis de informáticos que pela especificidade da organização envolve, *laptops*, *tablets*, *smartphones*, servidores e outros equipamentos. Para o desenho e desenvolvimento da arquitectura foram utilizadas como principais tecnologias: GraphQL, *Representational State Transfer* (REST), Spring Boot e Node.js. Na primeira instância de aproximação e de elemento de prova de conceitos, foi escolhido três entidades estritamente relacionadas, a saber: Utilizador, Equipamento e Histórico. Com essa abordagem, foi possível analisar o comportamento dos microsserviços, a sua comunicação, bem como os benefícios deste tipo de arquitectura.

Palavras chave: Microsserviços, GraphQL, REST, Node.js, Spring Boot

Abstract

Architecture based on microservices for IT equipment management at FGH

The use of microservices architecture takes us to a new pattern in the development of systems for Web in a different vision than what has been done in the last years. This work consists in creating an architecture based on microservices, in order to solve the lack of a computerized tool that helps in the monitoring computer equipment at Friends in Global Healt (FGH), which currently presents a complex organization and procedure, if one considers the mobility of its beneficiaries is considered by transfer, end of contracts, promotion or demotion. FGH's IT department is responsible for the infrastructure control and management of computer equipment and consumables, which, due to the specific nature of the organization, involves laptops, tablets, smartphones, servers and other communication equipment. The main technologies used for the design and development of architecture were: GraphQL, REST, Spring Boot and Node.js. As a first instance and as a proof of concept, we chose the following three related entities: User, Equipment and History. With this approach, it was possible to analyze the behavior of microservices and their communication, as well as the benefits of this type of architecture.

Keywords: Microservices, GraphQL, REST, Node.js, Spring Boot

Agradecimentos

Inicialmente, agradeço a Allah por ter iluminado o meu caminho, e permitido que eu chegasse até aqui. Aos meus pais, Raquel Abdul Sacur e Ismael Abdula (in memoriam), pela educação, conselhos, confiança, atenção, paciência e incentivo durante toda a minha caminhada. A minha esposa e filha, Jéssica Amiel e Luna Sacur respectivamente, que precisaram suportar tantos momentos de ausência para que esse trabalho fosse realizado. Agradeço também, aos meus familiares e amigos, que nunca negaram palavras de força, incentivo e optimismo ao longo da jornada académica. E por fim, mas não menos importante, aos professores, Pedro Salgueiro e Vítor Nogueira, por todo o apoio e pela paciência na orientação e incentivo que tornaram possível a conclusão desta dissertação.

Lista de Acrónimos

REST *Representational State Transfer*

SOA *Service-Oriented Architecture*

AMQP *Advanced Message Queuing Protocol*

HTTP Protocolo de Transferência de Hipertexto

API Interface de Programação de Aplicações

IT Tecnologias da Informação

SIDA Síndrome de Imunodeficiência Adquirida

VIH Vírus da Imunodeficiência Humana

FGH Friends in Global Health

UE Universidade de Évora

URI *Uniform Resource Identifier*

URL *Uniform Resource Locator*

HTML *HyperText Markup Language*

JSON *JavaScript Object Notation*

XML *Extensible Markup Language*

JPEG *Joint Photographic Experts Group*

SDL *Schema Definition Language*

SQL *Structured Query Language*

CRUD *Create, Read, Update e Delete*

JWT *JSON Web Token*

JPA *Java Persistence API*

MVC *Model-View-Controller*

IDE *Integrated Development Environment*

Lista de Figuras

Lista de Tabelas

Lista de Listagens

1

Introdução

Neste capítulo introdutório é apresentada a organização Friends in Global Health (FGH) onde o trabalho foi realizado e aplicado. Em seguida a motivação e a declaração do problema são discutidas bem como os objectivos gerais e específicos do trabalho. A última Secção descreve estrutura da dissertação.

A Friends in Global Health (FGH) é uma Organização Não-Governamental sem fins lucrativos que se dedica a actividades ligadas à saúde pública, em particular na área de prevenção e combate ao Vírus da Imunodeficiência Humana (VIH) e Síndrome de Imunodeficiência Adquirida (SIDA), em cooperação com instituições do Sistema Nacional de Saúde de Moçambique, tendo como objectivo garantir a saúde e o bem-estar dos pacientes que beneficiam das suas actividades e o alcance das metas que lhe são estabelecidas pelos seus doadores e pelo Governo da República de Moçambique.

Desde sua implantação em 2009 na província da Zambézia área da sua actuação, a FGH tem uma cobertura de apoio de 144 Unidades Sanitárias onde os colaboradores têm usado como ferramentas de apoio e de trabalho *laptops*, impressoras, servidores de bases de dados e outros equipamentos informáticos que estão sob tutela do Departamento de Informática.

1.1 Motivação

A FGH até então tem mais de 900 colaboradores dispersos pelas províncias da Zambézia e Maputo com uma média de mais de 400 utilizadores que usam computadores, onde cada utilizador possui uma ficha física (em papel) do equipamento informático (computador, flash, teclado, rato, pasta, monitor, entre outros periféricos) que lhes são atribuídos no acto de processo de contratação. Esta ficha é supervisionada pelo departamento de IT. Em casos de despedimentos, perdas, trocas de reposição, a gestão deste processo físico torna-se complexo pois, pela dispersão geográfica dos visados (utilizadores e gestores dos equipamentos de informática). Isto é mais evidente quando se procura ter uma informação pontual de um determinado utilizador ou grupo de utilizadores, em situações que possa existir uma eventual reestruturação de equipa ou despedimento compulsivo.

De forma recorrente a direcção da FGH faz questões como:

- Quantos equipamentos existem por distritos?
- Quantos equipamentos estão disponíveis?
- Quantos equipamentos estão em manutenção?
- Quantos equipamentos estão fora de uso (por avaria ou descontinuação de manutenção)?

Devido às características particulares da forma como a informação é gerida é difícil fornecer relatórios actualizados e de forma fiável, pois os registos não estão informatizados. De forma a mitigar as situações anteriormente descritas, surge a necessidade de dar resposta à insuficiência através de uma ferramenta informatizada que ajude na monitorização dos equipamentos informáticos na FGH.

1.2 Objectivos

O objectivo geral do trabalho apresentado nesta dissertação é desenhar e implementar uma arquitectura baseada em microsserviços, com as seguintes funcionalidades:

- Gerir dados de cada utilizador (identificação, nome, departamento, localização);
- Gerir dados de cada equipamento Informático;
- Gerir dados dos empréstimos e devoluções dos matérias informáticos;
- Listar equipamento informático associado ao utilizador;
- Listar equipamento informático por categoria e estado (disponível, ocupado, avariado);
- Listar equipamento informático por localização;

1.3 Estrutura da dissertação

Esta dissertação está dividida em capítulos com a seguinte organização:

- **Introdução:** Neste capítulo é feita uma introdução ao trabalho são apresentadas as motivações, o objectivo e a estrutura da dissertação.

- **Estado da Arte:** Neste capítulo são apresentadas as revisões bibliográficas sobre o tema e as ferramentas propostas.
- **Arquitectura Proposta:** É feita a apresentação dos requisitos do sistema e proposta da arquitectura.
- **Implementação:** É feita a implementação da arquitectura proposta e descrição dos componentes tecnológicos que fazem parte da arquitectura.
- **Avaliação:** Neste capítulo é feita a avaliação crítica sobre o trabalho realizado descrevendo se a arquitectura e ferramentas foram as mais apropriadas para implementar o trabalho.
- **Conclusões e Trabalho Futuro:** Neste capítulo são apresentadas as conclusões sobre o projecto e são indicados pontos para possíveis extensões futuras.

2

Estado da Arte

Neste capítulo é apresentado o estado da arte relacionado com os conceitos e termos técnicos que darão base teórica ao objecto de estudo e que são necessários para que se possa perceber o trabalho descrito nesta dissertação.

O estado da arte associado a este trabalho vai falar sobre os padrões de arquitectura de software, do estilo de arquitectura REST, da linguagem de consulta GraphQL, uma comparação entre REST e GraphQL, e de *frameworks* para criação de serviços Web RESTful e GraphQL.

2.1 Padrões de Arquitectura de Software

Os padrões de arquitectura ajudam a definir as características básicas, estrutura e o comportamento do sistema. Existem vários padrões de arquitectura de software entre os quais destaco: os padrões monolítico, *Service-Oriented Architecture* (SOA) e o de microsserviços.

2.1.1 Arquitectura monolítica

Num sistema monolítico todos os processos são altamente acoplados como um único serviço. Quaisquer alterações no sistema envolvem a construção e o *deployment* de uma nova versão de todo sistema. Entretanto apesar de ser uma boa abordagem para construir software, cada vez mais os desenvolvedores começaram a notar que não é a melhor solução para fazer um *deployment* na *cloud*, pois uma pequena mudança no sistema requer que todo o monólito seja reconstruído e com isso feito novamente o deployment. Outro problema de um sistema monolítico está relacionado com a forma como o sistema pode escalar, que obriga ao redimensionamento de todo o sistema, ao invés das partes que exigem mais recursos [FL14].

2.1.2 Arquitectura SOA

A arquitectura orientada a serviços é um estilo de arquitectura de software cujo princípio fundamental defende que as funcionalidades implementadas pelas aplicações devem ser disponibilizadas na forma de serviços. Frequentemente, estes serviços são conectados através de um barramento de serviços [Wik20e]. As interfaces de serviço fornecem acoplamento flexível o que significa que podem ser chamadas com pouco ou nenhum conhecimento de como a integração é implementada. Devido a esse acoplamento flexível e à forma como os serviços são publicados, os desenvolvedores podem economizar tempo reutilizando componentes em outros sistemas [Edu20].

2.1.3 Arquitectura microsserviços

Uma arquitectura baseada em microsserviços é uma abordagem arquitectónica e organizacional de desenvolvimento de software na qual o software consiste em pequenos serviços independentes que comunicam entre si, usando uma Interface de Programação de Aplicações (API) bem definida [Ama19]. Os microsserviços são tipicamente implementados e operados por pequenas equipas com autonomia suficiente para que cada equipa possa alterar os seus detalhes internos de implementação com um impacto mínimo no resto do sistema.

2.1.4 Diferenças entre as arquitecturas SOA e de microsserviços

A principal diferença entre ambos está no contexto de utilização: a arquitectura SOA possuí um âmbito corporativo, enquanto a arquitectura de microsserviços possui um âmbito de sistema. Isto é, o SOA permite que os sistemas existentes sejam expostos por interfaces fracamente acopladas, cada uma correspondente a uma função corporativa, permitindo que sistemas numa parte de uma organização reutilizem a funcionalidade em outros sistemas, ao passo que, microsserviços permite que os componentes internos de um único sistema sejam divididos em pequenas entidades que podem ser alteradas, dimensionadas e administradas independentemente [Edu20].

2.1.5 Características de uma arquitectura baseada em microsserviços

Embora não exista uma definição formal em relação às características que uma arquitectura em microsserviços deve possuir. Os autores [FL14], definem algumas características que são observadas na maioria das arquitecturas de microsserviços já existentes, a destacar:

- **Componentes via serviços** - No processo de utilização de componentes como serviço, o desenho do sistema é centrado no uso de bibliotecas que por força da arquitectura permite fragmentar o software em serviço. Daí a existência de componentes que são manipulados pelas funções construídas no sistema. Um exemplo, é o uso do *web services* Feign (ver Secção 4.2.4) para fazer os pedidos para um outro serviço a fim de realizar uma determinada tarefa.

- **Organizado através das áreas do negócio** - O estilo de microsserviços geralmente é organizado em torno dos recursos e prioridades dos negócios. Ao contrário de uma abordagem monolítica em que equipas diferentes têm um foco específico em, digamos, bases de dados, camadas de tecnologia ou lógica do lado do servidor - a arquitectura de microsserviços utiliza equipas multifuncionais: como equipa de vendas, equipas de pesquisas, entre outros.

- **Administração descentralizada de dados** - Como o sistema é decomposto em serviços, também é evidente que os dados precisam ser descentralizados. Isso significa que cada serviço pode ter sua própria tecnologia de bases de dados favorita ou que melhor se enquadra para o produto. Essa situação é geralmente denominada como persistência poliglota, como se pode observar na Figura 2.1.

- **Resistente a falhas** - Os microsserviços são projectados para lidar com falhas, é bem possível que um serviço falhe, por um motivo ou outro, nesse caso, o cliente precisa responder a esta falha da melhor forma possível, por isso é fundamental que se configure a monitorização de todos os serviços. Isto é particularmente importante numa configuração de serviços complexa, na qual cada cliente depende do outro.

2.1.6 Comunicação entre microsserviços

Um dos aspectos mais importantes no desenvolvimento de microsserviços em vez de um sistema monolítico, é uma comunicação entre os serviços. Como num sistema monolítico os processos são altamente acoplados como um único serviço, as invocações entre componentes são realizadas ao nível de chamadas de métodos, funções ou procedimentos, ao nível da linguagem.

Um sistema baseado em microsserviços pode ser um sistema distribuído com vários processos ou serviços, executados em um ou vários servidores. Um dos grandes desafios ao mudar para uma arquitectura baseada

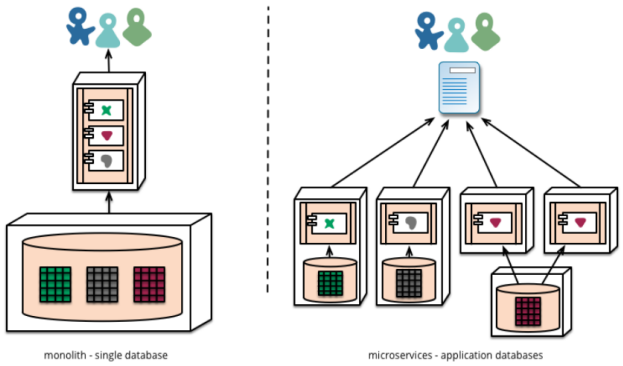

Figura 2.1: Exemplo de base de dados monolítica e de microsserviços [FL14]

em microsserviços é perceber o mecanismo de comunicação dos vários serviços que compõem o nosso sistema [Mic19].

Existem várias formas para classificar a comunicação entre microsserviços, veremos a seguir duas formas pelas quais os serviços podem comunicar entre si numa arquitectura de microsserviços:

Comunicação por HTTP

Protocolo de Transferência de Hipertexto (HTTP) é um protocolo síncrono. O cliente envia um pedido e espera uma resposta do serviço. Esse mecanismo é independente da execução do código do cliente que pode ser síncrona ou assíncrona [Mic19]. A Figura 2.2, apresenta uma arquitectura que utiliza o protocolo HTTP para comunicação entre diferentes serviços, a forma mais comum de utilização do protocolo HTTP na arquitectura baseada em microsserviços é utilizando o REST (ver Secção 2.2). Esta abordagem adopta verbos HTTP como GET, POST, PUT, DELETE, entre outros, para diferentes operações. O REST é a abordagem de comunicação de arquitectura mais comum usada na criação de serviços [Mic19].

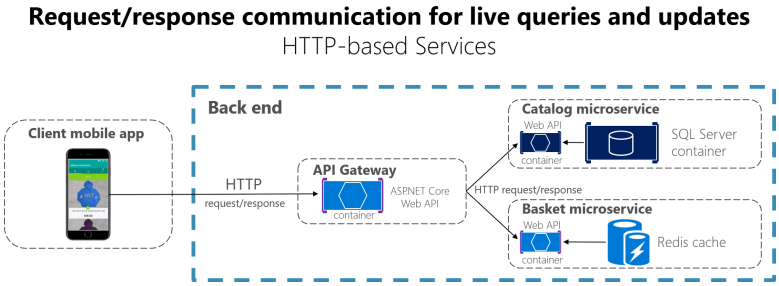

Figura 2.2: Exemplo de comunicação utilizando o HTTP [Mic19]

Comunicação por Mensagens

Diferentemente da comunicação HTTP usando comunicação por mensagens, os serviços envolvidos não se comunicam directamente, como se pode observar na Figura 2.3. Ao invés disso, os serviços enviam mensagens para um agente intermediário de mensagens que são subscritos por outros serviços [Rab19]. Uma mensagem é composta por um cabeçalho (metadados como informações de identificação ou de segurança) e um corpo. As mensagens são geralmente enviadas por meio de protocolos assíncronos, como *Advanced Message Queuing Protocol* (AMQP) [Mic19]. O remetente grava a mensagem no intermediário de mensagens e o intermediário entrega-a ao destinatário. Um benefício importante do uso de um intermediário de mensagens é que o remetente não precisa saber o local da rede do consumidor. Outro benefício é que um intermediário de mensagens armazena em *buffer* as mensagens até que o consumidor possa processá-las [Ric18]. Exemplos de intermediários de mensagens são:

- **Spring AMQP Framework**: Módulo do ecossistema Spring, que facilita o desenvolvimento de soluções de mensagens baseadas em AMQP, fornecendo um modelo de abstracção de alto nível para o envio e recepção de mensagens [Spr20b].
- **RabbitMQ**: É um *framework* leve e de fácil de configuração tanto no servidor local como na *cloud* que suporta vários protocolos de mensagens como o AMQP. Foi desenvolvido tendo em conta os requisitos de alta escalabilidade e disponibilidade [Rab20].

- **Apache Kafka**: É uma plataforma de *streaming* de evento distribuído de código aberto usada por milhares de empresas para *pipelines* de dados de alto desempenho, análise de *streaming*, integração de dados e sistemas de missão crítica [Kaf20].

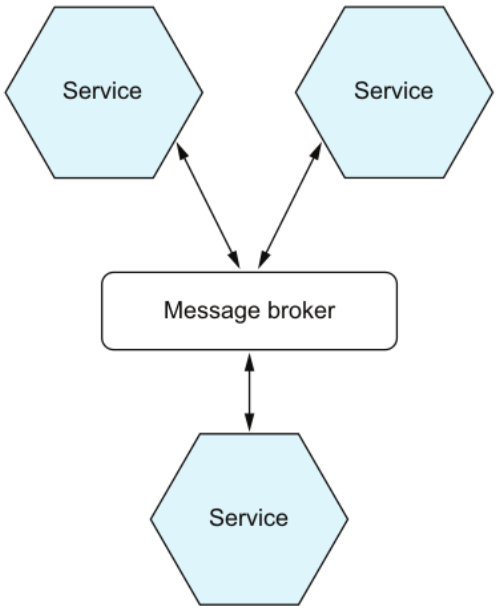

Figura 2.3: Exemplo de comunicação utilizando um agente intermediário de mensagens [Ric18]

2.1.7 Vantagens da arquitectura em microsserviços

Na essência são muitos e vários benefícios que se pode ter quando os microsserviços são implementados de forma correcta. Todos os benefícios moldam-se à disposição dos Sistemas Distribuídos [New15]:

- **Heterogeneidade**: É possível implementar a colaboração entre os serviços de forma múltipla, onde há abertura para escolha do tipo de tecnologia a usar em cada serviço. Desta forma, as equipas podem escolher diferentes linguagens de programação para cada serviço.

- **Resiliência**: Os microsserviços devem ser construídos e testados para serem resilientes de forma independente. Dessa forma, os microsserviços devem definir limites de serviço que facilitam a identificação e o isolamento do problema em caso de falhas.

- **Escalabilidade**: Permite escalar somente os microsserviços necessários ao em vez de escalar todo sistema, como acontece no monolítico.

- ***Deployment* facilitado**: Visto que, microsserviços são pequenos serviços que operam de forma independente, uma alteração necessária na funcionalidade, requer apenas a modificação e *deploy* do serviço em questão, enquanto que no sistema monolítico exige que o *deploy* seja de todo sistema, o que pode ser de alto risco e alto impacto.

2.1.8 Desafios da arquitectura em microsserviços

Apesar das muitas vantagens os microsserviços podem não ser a melhor opção para alguns projectos. Em geral, o principal ponto negativo dos microsserviços é a complexidade que qualquer sistema distribuído possui, a conhecer [BP18]:

- O âmbito e identificação de microsserviços requer conhecimento substancial do domínio.
- Os limites e os contratos entre os serviços são difíceis de identificar e, uma vez estabelecidos, e se for necessário fazer alterações, pode-se levar muito tempo para fazer tal mudança.
- Os microsserviços podem ser sistemas distribuídos e, portanto, exigem diferentes premissas a serem feitas sobre estado, consistência e confiabilidade da rede.
- Ao distribuir componentes do sistema em redes e aumentar a heterogeneidade técnica, os microsserviços introduzem novos modos de falha.

2.2 REST

O conceito *Representational State Transfer* (REST) foi introduzido em 2000 na tese de doutoramento de Roy Fielding, um dos principais autores das especificações do protocolo HTTP [FT00]. Roy Fielding, descreve o REST como sendo um estilo de arquitectura de software que define um conjunto de restrições a serem usados para a criação de serviços Web. Os serviços Web que estão em conformidade com o estilo arquitectural REST, são chamados serviços Web RESTful [Wik19c]. É importante realçar que o REST não é um protocolo mas sim um estilo de arquitectura.

2.2.1 Princípios e Restrições de REST

A seguir serão listados os fundamentais princípios e restrições de REST:

Client-Server

Cada mensagem HTTP contém toda a informação necessária para compreender o pedido [Wik19c]. Isso ajuda a estabelecer uma arquitectura distribuída, dando suporte à evolução independente da lógica do lado do cliente e do servidor [FT00].

Stateless

Cada pedido do cliente ao servidor deve conter todas as informações necessárias para perceber o pedido, e não pode tirar proveito de nenhum contexto armazenado no servidor. Por outras palavras, cada pedido ao servidor deve ser entendido como o primeira e não deve ter relação com o anterior. Desta forma é muito mais fácil escalar o sistema [FT00].

Cache

As restrições de cache exigem que a resposta de um pedido sejam implicitamente ou explicitamente rotulados como armazenáveis em cache ou não. Se uma resposta for armazenável em cache, a cache do cliente poderá reutilizar os dados da resposta para pedidos posteriores equivalentes [FT00].

Desenvolvimento em camadas

A arquitetura REST permite que o sistema seja desenvolvido em camadas, por exemplo, um sistema pode implementar as APIs no servidor A, armazena dados no servidor B e autentica pedidos no servidor C. Um cliente normalmente não precisa conhecer se está conectado directamente ao servidor final ou a um intermediário ao longo do caminho [FT00].

Uniform Interface

O princípio que caracteriza um sistema REST em particular é ter uma interface uniforme (*Uniform Interface*), que sustenta que os métodos de interacção entre os componentes do sistema devem ser regulados por convenções uniformes. De fato, Fielding indica quatro restrições que devem ser atendidas para satisfazer esse princípio[Avr17]:

- **Identificação de Recursos** - Um recurso é um objecto ou a representação de algo significativo no domínio do sistema. A interacção com estes recursos é feita através das APIs. Exemplos de entidades: um carro, um equipamento, um utilizador e qualquer outra entidade que possa ser abstraída de um determinado contexto. O conceito de recurso é, portanto, muito semelhante ao de um objecto no mundo da programação orientada a objectos. A identificação do recurso deve ser feita utilizando-se o conceito de *Uniform Resource Identifier* (URI), que é um dos padrões utilizados pela Web. Alguns exemplos de URIs:
  - https://mozapi/utilizadores;
  - https://mozapi/utilizadores/5;
  - https://mozapi/equipamentos.

- **Manipulação de Recursos** - Um recurso pode ser representado de várias formas como por exemplo *HyperText Markup Language* (HTML), *Extensible Markup Language* (XML), *JavaScript Object Notation* (JSON) ou mesmo como um arquivo *Joint Photographic Experts Group* (JPEG). Isto significa que os clientes interagem com os recursos por meio das suas representações, que é uma formas poderosa de manter os conceitos de recursos abstraídos das suas interacções. Na Listagem 2.1 é apresentado um exemplo de JSON para representar um recurso.

```
{
    "assetName": "Dell Laptop E5580",
    "assetTag": "A0050",
    "available": true,
    "category": {
        "categoryId": 1,
        "name": "Laptop"
    },
    "purchaseAt": "2019-09-10",
    "purchaseCost": 18760.00,
```

```
    "serial": "XXTGHKO",
    "notes": "Novo"
}
```

Listagem 2.1: Exemplo de representação de um recurso em JSON

- **Mensagem Auto-descritiva** - Cada pedido à API deve conter todas as informações que o serviço precisa para executar o pedido e cada resposta que o serviço retorna contém todas as informações que o cliente precisa para entender a resposta. Os sistemas REST também operam com a noção de um conjunto restrito de tipos de mensagens que são totalmente compreendidos pelo cliente e pelo servidor. O documento que define o HTTP descreve oito tipos de mensagens (Métodos HTTP) que podem ser enviados aos servidores HTTP. Seis deles são amplamente utilizados hoje. São eles: GET, HEAD, OPTIONS, PUT, POST e DELETE. Os três primeiros são mensagens somente leitura. Os três últimos são mensagens de actualização. Existem regras bem definidas de como os clientes e servidores se devem comportar ao usar estas mensagens. Os nomes e significados dos elementos de metadados das mensagens (cabeçalhos HTTP) também estão bem definidos. Os sistemas REST entendem e seguem essas regras com muito cuidado [Amu08]. A Tabela 2.1, mostra a relação entre operações *Create, Read, Update e Delete* (CRUD) e HTTP, estas operações CRUD são executadas pelo REST, suportando os métodos HTTP.

Operação	**Método HTTP**
Create	POST
Read	GET
Update	PUT, PATCH
Delete	DELETE

Tabela 2.1: Operações CRUD e seus respectivos métodos HTTP

2.2.2 Vantagens da arquitectura REST

Como se pode observar a arquitectura REST tem inúmeras vantagens tais como, permitir a evolução independente dos diferentes componentes do desenvolvimento de um sistema: se, por exemplo, hoje o *frontend* de um sistema REST é escrito em Angular [Ang20] e posteriormente a equipa de desenvolvimento decidir trocar por uma outra é possível substituir o componente do *frontend* sem modificar uma linha de código de *backend* (vice-versa também é possível). A separação entre cliente e servidor tem uma vantagem óbvia: o sistema pode escalar sem muitos problemas. Independência em relação a linguagem de programação e tecnologias específicas: um sistema REST é sempre independente do tipo de plataforma ou programação: adapta-se ao tipo de linguagem ou plataformas usadas, o que oferece liberdade considerável na escolha da linguagem ou estrutura para a implementação de um componente.

2.3 GraphQL

GraphQL é uma linguagem de consulta criada pelo Facebook em 2012 e lançada publicamente em 2015. É considerada uma alternativa para arquitecturas REST, além de oferecer um serviço *runtime* para executar comandos e utilizar uma API [Wik19b]. O GraphQL não está vinculado a nenhuma base de dados ou mecanismo de armazenamento específico e, em vez disso, é apoiado pelo seu código e dados existentes [Gra19].

Pode-se afirmar que o GraphQL é uma linguagem de consulta e manipulação de APIs, dando assim a possibilidade do cliente informar o servidor quais os dados que ele realmente precisa a partir de um único Uniform Resource Locator *(URL)* disponibilizado pelo servidor.

2.3.1 Conceitos Básicos

O GraphQL funciona com base em esquemas fortemente tipados, composto por tipos de objectos. Do ponto de vista do cliente, as operações mais comuns do GraphQL são consultas e mutações. Se consideramos o ponto de vista do modelo CRUD, uma consulta seria equivalente à operação de leitura e mutações seriam equivalentes às operações idempotentes (criar, actualizar e excluir) [tG19].

Schema Definition Language

O GraphQL possui o seu próprio sistema de tipos que é usado para definir o esquema de uma API. Este esquema é a representação de um objecto, que espelha a estrutura dos dados a serem consultados. A Listagem 2.2, ilustra o *Schema Definition Language* (SDL) que contém o esquema do tipo *Car*, onde são declarados os atributos que compõem o tipo carro, o ponto de exclamação (!) significa que o atributo é obrigatório, e por último, o atributo (*owner*) irá retornar objecto do tipo *Owner*.

```
type Car {
  id: ID!
  plate: String!
  seat: Int!
  model: String!
  color: String
  owner: Owner!
}
```

Listagem 2.2: Exemplo de SDL

Consultas com GraphQL

Como foi explicado anteriormente, o GraphQL trabalha com um único URL. Isto funciona porque a estrutura dos dados retornados não é fixa. Em vez disso, é totalmente flexível e permite que o cliente decida quais os dados que são realmente necessários. Por exemplo na Listagem 2.3, é apresentado um exemplo de procura de todos os carros, cujo o SDL foi apresentado na Listagem 2.2, mas na resposta queremos somente as matrículas, como se pode constatar na Listagem 2.4, onde na resposta temos somente as matriculas de todos os caros registados no sistema.

```
query {
  allCarrs {
    plate
  }
}
```

Listagem 2.3: Exemplo de uma consulta GraphQL

```
{
    " allCarrs": [{
            "plate": "AZZ 099 SF"
```

```
        },
        {
            "plate": "EBB 9 99 ZB"
        },
        {
            "plate": "ASA 0 92 MP"
        }
    ]
}
```

Listagem 2.4: Resposta da consulta feita na Listagem 2.3

Mutations

O GraphQL permite criar, alterar e apagar dados. Como estas três operações causam modificações, em GraphQL são chamadas de *mutations*. A sintaxe para criar, alterar e apagar dados não difere muito da sintaxe das consultas, mas devem sempre começar com a palavra chave *mutation*. A Listagem 2.5 ilustra a mutação para a criação do carro, onde o nome da mutação é *createCar*. Devemos passar obrigatoriamente para mutação como entrada os atributos *plate*, o *seat*, o *model* e o *owner* e na resposta será devolvido um objecto do tipo *Car*.

```
type Mutation {
  createCar(
    plate: String!
    seat: Int!
    model: String!
    color: String
    owner: Ownwer!
  ): Car
}
```

Listagem 2.5: Exemplo de mutação GraphQL

2.3.2 Vantagem da linguagem de consulta GraphQL em relação a arquitectura REST

Os serviços RESTful tem sido uma forma popular de expor dados de um sistema. Quando o conceito de REST foi desenvolvido, os sistemas clientes eram relativamente simples e o ritmo de desenvolvimento era inferior ao de hoje [tG19]. O REST foi, portanto, uma boa solução para muitos sistemas, no entanto, o cenário da API mudou radicalmente nos últimos dois anos [tG19]. Em particular, há três factores que desafiam a forma como as APIs são projectadas [tG19]:

- **Carregamento eficiente dos dados**: O aumento do uso de dispositivos móveis, dispositivos com poucos recursos e fraca qualidade de rede foram os motivos iniciais pelos quais o Facebook desenvolveu o GraphQL. O GraphQL minimiza a quantidade de dados que precisam ser transferidos pela rede e, portanto, aprimora principalmente os sistemas que operam sob essas condições.

- **Resolve os problemas de *Over Fetch* ou *Under Fetching***: O *over fetching* acontece quando a API retorna mais dados do que o cliente precisa num pedido. O *under fetching* acontece quando precisamos de recorrer a vários pedidos em múltiplos URLs para ter a informação necessária. Com o GraphQL o cliente recebe o que precisa com um único *URL*.

- **Desenvolvimento rápido**: Com as APIs RESTFul, a forma como os dados são expostos pelo servidor geralmente precisa ser modificada para satisfazer os requisitos específicos e alterações de desenho no lado do cliente. Isso dificulta práticas de desenvolvimento rápido e iterações de produtos. O GraphQL resolve esse problema como vimos anteriormente, o cliente irá receber somente aquilo que precisa.

2.3.3 Desvantagem da linguagem GraphQL em relação à arquitectura REST

Embora o GraphQL tenha muitas vantagens sobre as APIs RESTFul tradicionais, também existem desvantagens. Entre elas, encontram-se as seguintes:

- **Optimização e complexidade de consultas**: Como os clientes têm a possibilidade de criar consultas muito complexas, os serviços devem estar preparados para lidar com elas adequadamente, sendo necessário limitar a quantidade de níveis que podem ser acedidos, pois, se isso não acontecer é possível que o cliente comprometa o serviço [tG19].
- **Avaliação de limites**: Nos casos em que é necessário disponibilizar a API de um serviço a terceiros, e de algum modo limitar a quantidade de pedidos e apenas permitir mais operações mediante o pagamento do serviço, esse tipo de limitação é muito mais difícil de se conseguir com o GraphQL do que com REST [Han19].
- **Cache**: A implementação de um processo de armazenamento em cache é significativamente mais complexa com o GraphQL em comparação com as consultas de uma API REST, pois estas podem ser guardadas usando métodos de armazenamento em cache do protocolo HTTP. Enquanto que com GraphQL, isso se torna complexo porque cada consulta pode ser diferente, mesmo que opere sobre a mesma entidade. É possível pedir somente o nome de um autor numa consulta, mas desejar saber o endereço de email na próxima. É aí que se torna necessário implementar uma cache com um detalhe mais fino, que pode ser difícil de implementar [Wie19].
- **Monitorização de erros**: As APIs baseadas em REST tiram proveito dos códigos de status HTTP para informarem ao cliente dos erros. Isso torna a monitorização das APIs muito fácil e sem esforço para o desenvolvedor. Já os serviços GraphQL retornam sempre um código de status HTTP 200, independentemente da consulta ter ou não sido bem-sucedida. Isto pode dificultar a monitorização de erros e pode levar a uma complexidade adicional para a monitorização [Smi20].

2.4 *Frameworks* para criação de serviços Web RESTful e GraphQL

Nesta Secção serão apresentadas as ferramentas, bibliotecas ou *frameworks* que ajudam na criação de serviços Web usando o REST e GraphQL.

2.4.1 Criação de Serviços REST

Actualmente existem várias ferramentas ou *frameworks* disponíveis para a criação de serviços Web RESTful, nesta Secção serão apresentadas brevemente um *framework* e uma especificação baseadas na linguagem Java [Jav20a], a destacar:

- Spring *Model-View-Controller* (MVC)

- JAX-RS

Spring MVC

O Spring (ver Secção 4.1.1) é um dos *frameworks* mais usados, para a linguagem de programação Java [Sus20]. A sua popularidade deve-se ao facto de se apresentar como um framework leve, que é utilizado para o desenvolvimento de diversos tipos de sistemas e também continuar a evoluir activamente desde a sua criação. O Spring MVC faz parte do ecossistema do Spring, que segue o padrão MVC. O MVC é um padrão de software, cujo o foco está na reutilização do código e a separação de conceitos em três camadas interconectadas, onde a apresentação dos dados e interação dos utilizadores (*front-end*) são separados dos métodos que interagem com o banco de dados (*back-end*) [Wik20c].

O componente responsável por orquestrar o funcionamento do Spring MVC é o `DispatcherServlet`. O `DispatcherServlet` segue o padrão *front controller,* cujo objectivo deste é fornecer um ponto de entrada central para todos os pedidos e de seguida, mapear para o recurso certo, como controladores, modelos e visualizações [Wei14].

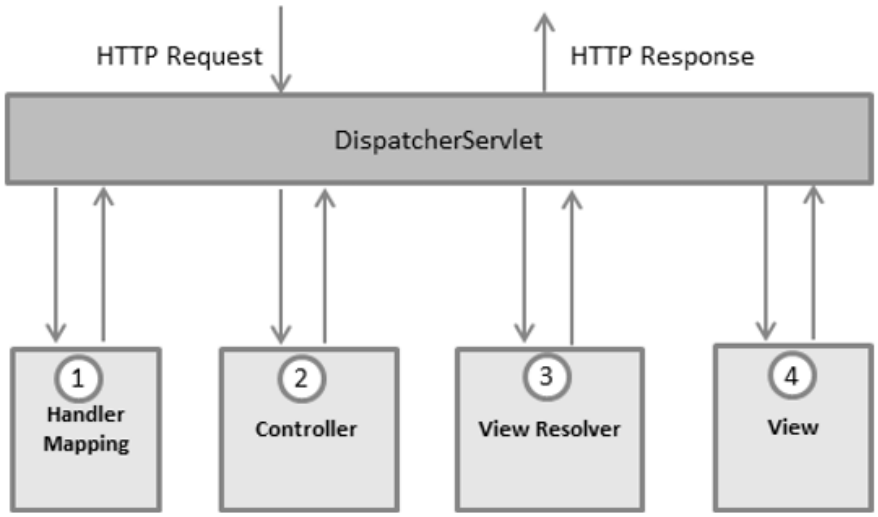

Figura 2.4: Ciclo de vida de um pedido [Jav20b].

Todo o pedido ao serviço inicia-se sob a forma de um pedido e termina com uma resposta enviada ao cliente. Durante esse caminho ocorrem vários eventos, representados na Figura 2.4. Primeiramente, tudo começa quando o pedido enviado pelo cliente chega ao `DispatcherServlet`. Este, a partir da URL, descobre qual controlador deverá ser chamado para atender o pedido, para tal, um `HandlerMapping` é usado. Uma vez obtido o controlador alvo, este é executado, recuperando-se depois os resultados, que os retorna ao *servlet*, encapsulado num objecto do tipo modelo. Além disso, o nome lógico da visualização a ser exibida será retornado. Por fim, o *front controller* redirecciona a resposta do pedido para a exibição [Wei14].

As vantagens deste *framework* são inúmeras, dos quais podemos destacar as seguintes [Jav20b]:

- **Responsabilidades separadas**: O Spring MVC separa responsabilidades onde o objecto modelo, controlador, *view resolver*, `DispatcherServlet`, entre outros, podem ser executado por um objecto especializado.
- **Leve**: Utiliza um *container* de *servlet* leve para desenvolver e fazer *deploy* do sistema.

- **Configuração robusta**: Oferece uma configuração robusta tanto para a estrutura como para as classes do sistema que inclui referências fáceis através de contextos, tais como controladores web e validações.
- **Desenvolvimento rápido**: O Spring MVC através do Spring Boot (ver Secção 4.1.1), facilita o desenvolvimento rápido dos sistemas, pois o programador não precisa de ser preocupar com configurações mas sim com as regras do negócio.
- **Código de negócio reutilizável**: Em vez de criar novos objectos, permite-nos utilizar os objectos existentes.

JAX-RS

Diferente do Spring o JAX-RS é uma especificação que faz parte da plataforma Java EE [Cor20] e foi projectado para ser uma solução padrão e portátil. Actualmente, existem muitas implementações de referência disponíveis para o JAX-RS, sendo a mais utilizada o Jersey [Jer20]. O Jersey inicialmente foi desenvolvido pela Sun, de seguida adquirida pela Oracle *Corporation* e actualmente é suportada pela fundação Eclipse.

Embora o JAX-RS e o Spring *framework* possam ser usados para criar um serviço Web RESTful em Java, existem várias diferenças entre ambos, como podemos observar na Tabela 2.2. Contudo, a programação em ambos se baseia em anotações. As diferenças entre anotações encontram-se na Tabela 2.3.

JAX-RS	**Spring MVC**
JAX-RS é a especificação padrão do Java para serviços RESTful.	O Spring MVC é uma forma alternativa de escrever serviços RESTful em Java, ele não implementa o JAX-RS de forma nativa.
O JAX-RS é apenas uma especificação, é preciso incluir uma implementação como Jersey.	O Spring MVC é a implementação completa dos serviços RESTful da Spring.
O JAX-RS usa anotações que fazem parte do Java EE.	O Spring MVC usa suas próprias anotações personalizadas para o criar serviços RESTful

Tabela 2.2: Comparação entre JAX-RS e Spring MVC [LEA20]

JAX-RS	**Spring MVC**	**Função**
@Get	@GetMapping	Buscar recursos.
@Post	@PostMapping	Criar um recurso
@Put	@PutMapping	Actualizar um recurso
@Delete	@DeleteMapping	Apagar recurso
@Path	@RequestMapping	Especifica o nome da URI
@QueryParam	@RequestParam	Usado para especificar um parâmetro de consulta
@PathParam	@PathVariable	Usado para especificar um parâmetro de consulta

Tabela 2.3: Comparação entre anotações mais utilizadas do JAX-RS e Spring MVC

2.4.2 Criação de Serviços GraphQL

O GraphQL pode ser criado usando várias linguagens de programação como Java, JavaScript [Wik20a], PHP [Wik20d], entre outros. A seguir serão apresentados duas ferramentas que facilitam na criação do servidor GraphQL:

- Apollo Server
- GraphQL-Yoga Server

Apollo Server

O Apollo Server é uma biblioteca que facilita a criação de servidores GraphQL em JavaScript, o seu código fonte é *open source* e é compatível com especificações do GraphQL [Wie19]. Algumas vantagens de usar o Apollo Server são [Wie19]:

- **Excelente Ecossistema**: Enquanto o GraphQL está em seus estágios iniciais, o ecossistema Apollo oferece soluções para muitos de seus desafios, como por exemplo, a compatibilidade com arquitectura REST.
- **Documentação**: Enquanto o Apollo continua a evoluir, a equipa de desenvolvimento e a comunidade por trás desta mantêm a documentação actualizada e têm muitas informações sobre como usar o Apollo.
- **Ferramentas**: O Apollo possuí diversos recursos internos para extrair toda a complexidade na criação de sistemas Web, pois além do Apollo Server, ela possui ferramentas como: Apollo Client, que é utilizado para criação de clientes GraphQL.

GraphQL-Yoga

O Yoga é um servidor GraphQL completo com foco na fácil configuração, bom desempenho e óptima experiência de desenvolvimento [Pri20]. É uma ferramenta feita em Node.js (ver Secção 4.1.2). O Yoga foi construído sobre uma variedade de outras ferramentas, tais como, Express e Apollo Server. Cada um destes fornece uma certa funcionalidade necessária para a construção de um servidor GraphQL. A utilização individual destes pacotes implica uma sobrecarga no processo de configuração. O Yoga abstrai a complexidade inicial e permite que o desenvolvedor se foque na construção do sistema [Pri20].

2.5 Conclusão

Este capítulo abordou os conceitos de microsserviços, sua arquitectura, seus benefícios, limitações e tipos de comunicações entre microsserviços. Em seguida, o REST, os seus princípios e restrições, suas vantagens, conceitos do GraphQL, suas vantagens e desvantagens em relação ao REST. E por último, abordou os *frameworks* para criação de serviços em REST e GraphQL.

3

Arquitectura Proposta

Este capítulo descreve a análise do sistema, requisitos funcionais e não funcionais. Os requisitos funcionais estão relacionados no controlo comportamental do sistema sobre o que deve fazer e o que não deve fazer na resposta daquilo que lhe for solicitado quando estiver a operar, enquanto que, os requisitos não funcionais são apresentados todos requisitos que não afectam na funcionalidade básica do sistema. E por último, é apresentada a arquitectura do sistema, sem ter em conta as tecnologias que a compõe.

3.1 Análise de Requisitos

Com o objectivo de responder à insuficiência de uma ferramenta informatizada que ajude na monitorização dos equipamentos informáticos na Friends in Global Health (FGH), os diferentes requisitos foram identificados através do método de observação coadjuvada com o envolvimento directo do autor na tramitação de processos envolvendo as partes beneficiárias desta solução. São eles:

3.1.1 Utilizadores e Características

Nesta Secção serão descritos descritos e arrolados os papeis e características de cada utilizador observando a sua categoria ou grupo. O Utilizador Genérico e o Administrador são as duas partes que interagem com o sistema, e estes possuem níveis de permissões diferentes, a saber:

Administrador:

- Autenticar-se no sistema.
- Registar novos utilizadores.
- Adicionar e corrigir informação de um equipamento informático.
- Actualizar o estado de um equipamento informático.
- Associar um equipamento informático a um utilizador.
- Ver equipamento pela localização.
- Marcar se o utilizador poderá ou não autenticar-se no sistema.

Utilizador Genérico:

- Pode autenticar-se no sistema.
- Ver o histórico do utilizador.

3.1.2 Requisitos de Interface

Existem vários tipos de interfaces suportadas pelo sistema, dos quais, iremos destacar o de hardware e de comunicação.

Hardware:

- O hardware deverá ser composto por vários servidores. Os clientes podem ser qualquer dispositivo que tenha ligação à Internet ou acesso aos servidores;

Comunicação:

- Este sistema irá comunicar com a base de dados que contém todas as informações pretendidas pelo cliente. Os clientes podem entrar em contacto com o servidor através do protocolo HTTP por meio de uma API REST ou GraphQL. Esta função permite que o sistema use os dados recuperados pelo servidor para atender o pedido do cliente.

3.1.3 Requisitos Funcionais

São declarações de serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como o sistema se deve comportar em situações específicas [Som15]. Os requisitos descrevem as funcionalidades que o sistema oferece aos utilizadores.

As vantagens em descrever correctamente os requisitos funcionais são inúmeras, a destacar [Gur20]:

- Ajuda a verificar se o sistema está fornecendo todas as funcionalidades que foram mencionadas no requisito funcional.
- Um documento de requisito funcional ajuda a definir a funcionalidade de um sistema ou um de seus subsistemas.
- Ajudam a definir claramente o serviço e o comportamento esperados do sistema.
- Os erros detectados no estágio de colecta de requisitos funcionais são menos custosos de corrigir.
- Ofereça suporte às metas, tarefas ou actividades do utilizador.

A seguir serão listadas os diferentes requisitos funcionais do sistema, para isso será utilizada a sigla RF para identificar o requisito funcional.

Equipamento Informático:

Para a FGH é considerado material informático todo aquele equipamento computadorizado que inclui *smartphones*, *tablets*, *laptops*, disco duro externo e outros periféricos imprescindíveis para uso e funcionamento normal no ambiente informatizado. Assim, o sistema deverá permitir que o administrador:

- [RF1] Registe equipamento informático.
- [RF2] Actualize o equipamento informático.
- [RF3] Pesquise o equipamento por um registo específico.
- [RF4] Visualize lista do equipamento informático por estado (disponível, ocupado, avariado), e por localização.

Empréstimo:

É considerado empréstimo o acto de entrega de material informático a um colaborador da organização, para o tempo em que o seu contrato com a organização estiver em vigor. A ser assim, o acto de retorno do equipamento por parte do colaborador para a organização é considerada devolução, um dos sub-itens do empréstimo. O sistema deverá permitir:

- [RF5] Que o administrador faça empréstimo de equipamento informático aos utilizadores.
- [RF6] Que o administrador faça devolução de equipamento informático outrora emprestado.
- [RF7] Aos utilizadores visualizarem os detalhes do empréstimo e da devolução.

- [RF8] Que o administrador registe o estado do equipamento no acto da devolução.
- [RF9] Fazer seguimento de equipamentos emprestados, o que envolverá saber o tempo de permanência e data de devolução.
- [RF10] Que o utilizador visualize a lista do equipamento emprestado.

***Login*:**

O Login é acto de autenticação e consequente autorização indispensável para legitimar o utilizador ao Sistema. Neste caso o sistema deverá permitir:

- [RF11] Que o utilizador efectue *login* com as suas credenciais.
- [RF12]Que o administrador crie e actualize novo utilizador.
- [RF13] Que o utilizador altere a sua senha.
- [RF14] Discriminar os utilizadores por e níveis de permissão.
- [RF15] Que o utilizador efectue o *logout*.

Distrito:

A FGH como organização tem o campo de actuação configurado em Distritos. É nos distritos onde a organização endereça os seus apoios e acompanhamento tanto à comunidade como aos seu colaboradores. O sistema deverá permitir que o administrador:

- [RF16] Crie novo distrito.
- [RF17] Actualize um distrito.
- [RF18] Associar utilizador a um distrito.

3.1.4 Requisitos Não Funcionais

São restrições nos serviços ou funções oferecidas pelo sistema. Eles incluem restrições de tempo, restrições no processo de desenvolvimento. Os requisitos não funcionais geralmente aplicam-se ao sistema como um todo, e não aos recursos individuais do sistema ou serviços [Som15]. Os requisitos não funcionais olham o sistema com base na capacidade de resposta, usabilidade, segurança, portabilidade e outros padrões não funcionais que são críticos para o sucesso do sistema de software [Gur20]. O não cumprimento de qualquer um dos requisitos pode resultar em sistemas que não atendem às necessidades internas do negócio, do utilizador ou do mercado, ou que não atendam aos requisitos obrigatórios impostos por agências regulatórias ou de padrões. Em alguns casos, a não conformidade pode causar problemas jurídicos significativos [Agi20].

Os benefícios no uso correcto dos requisitos funcionais são [Gur20]:

- Os requisitos não funcionais garantem que o sistema segue as regras legais e de conformidade.
- Eles garantem a confiabilidade, disponibilidade e desempenho do sistema.

- Garantem uma boa experiência do utilizador e facilidade de operação do sistema.
- Ajudam na formulação da política de segurança do sistema.

Usabilidade

O sistema fornecerá um menu de ajuda e suporte em todas as interfaces para o utilizador interagir com o sistema. O utilizador pode usar o sistema lendo ajuda e suporte.

Segurança

Para impedir o acesso não autorizado ao sistema o utilizador deverá fornecer suas credenciais. A senha deve ter pelo menos oito caracteres, e deve conter pelo menos um dígito, caracteres maiúsculos como minúsculos, bem como caracteres de pontuação. O sistema deverá fornecer um alto nível de segurança e integridade, somente utilizador com senha e nome de utilizador válidos podem fazer *login* para visualizar a página do utilizador.

Disponibilidade

O sistema deve estar sempre disponível para acesso 24 horas por dia, 7 dias por semana. Também na ocorrência de qualquer falha no sistema principal, o sistema deve estar disponível o mais breve possível, para que o processo de negócios não seja severamente afectado.

Manipulação de Erros

O erro deve ser consideravelmente minimizado e uma mensagem de erro apropriada que orienta o utilizador a se recuperar de um erro deve ser fornecida. A validação da entrada do utilizador é altamente essencial.

3.2 Arquitectura do Sistema

Arquitectura baseada em microsserviços consiste em serviços diferentes independentes que operam de forma autónoma. A arquitectura do sistema está representada na Figura 3.1, que consiste nos seguintes diferentes serviços:

- **Utilizador**: conterá módulos que fornecem dados do utilizador e sua validação para acesso ao sistema. Este microsserviço relaciona-se directamente com o microsserviço Distrito, para atribuir uma localização ao Utilizador, e está localização será posteriormente utilizada pelos microsserviços Histórico e Equipamentos.
- **Distrito**: será responsável pela administração dos locais. Na essência o microsserviço Distrito será utilizado pelo microsserviço Utilizador para gestão do mesmo. O que significa que se por ventura houver necessidade de transferência de Utilizador o microsserviço utilizador irá retirar o seu utilizador do suposto distrito corrente o para o suposto distrito de transferência. Havendo necessidade de se fazer uma listagem de equipamentos de um certo distrito, irá recorrer-se ao microsserviço Utilizador para listar todos utilizadores pertencentes àquele distrito, o que consequentemente irá configurar o número de equipamentos existentes no tal distrito a partir da lista dos utilizadores.

- **Equipamento**: o seu foco é o CRUD sobre todo equipamento informático em circulação na FGH. Este microsserviço cria registo, remove, actualiza e faz leitura do equipamento informático relacionado (emprestado) ou não aos colaboradores da FGH. Este microsserviço é actualizado pelo microsserviço Histórico no momento de devolução do equipamento para disponível ou não.

- **Histórico**: este relaciona-se directamente ao microsserviço Equipamento e Utilizador. O microsserviço Histórico é responsável pela administração do empréstimo e devolução do material informático, sendo a actualização dessa informação gerida por este microsserviço.

É possível observar na Figura 3.1 que as bases de dados estão descentralizadas. Conforme mencionado antes (ver Secção 2.1.5) cada microsserviço pode ter sua própria base de dados. Obviamente podem ser implementados com uma base de dados centralizada mas com isso perde-se um dos pilares fundamentais dos microsserviços, a independência entre eles, uma vez que, ao partilhar a base de dados, se algum deles for modificado acabará afectando o resto da base de dado. Um outro aspecto negativo comparando com a metodologia de uma base de dados por serviço é que está abordagem de base de dados centralizada acaba ficando refém de uma única tecnologia de sistema de gestão de bases de dados. Por exemplo, se decidirmos utilizar um sistema de base de dados não relacional para o um determinado microsserviço, isso não seria possível, visto que estão fortemente acoplados .

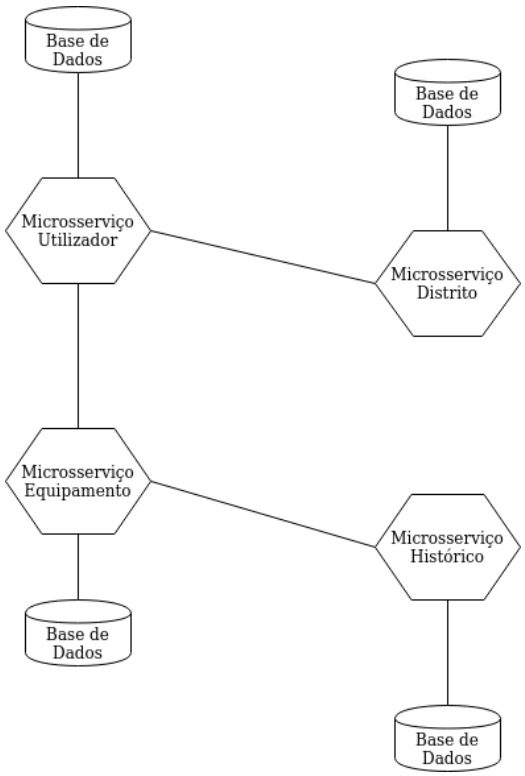

Figura 3.1: Arquitectura baseada em microsserviços

3.3 Operações por Microsserviço

Nesta Secção serão listadas as diferentes operações por microsserviço, para tal será utilizada a sigla OP para identificar o número da operação. Os microsserviços Utilizador, Histórico e Equipamento serão listados também a respectiva especificação.

3.3.1 Operações Microsserviço Utilizador

O microsserviço utilizador tem a tarefa de gestão de utilizadores, bem como a autenticação e privilégios de acesso em relação aos outros microsserviços, tais como:

- [OP1] Criar utilizador.
  - URL: api/users
  - Método: POST
  - Headers: Content-Type: application/json
  - Descrição: Cria um novo utilizador
  - Corpo Obrigatório:
    - name: String
    - username: String
    - password: String
    - roles: {id: Long, name: String}
  - Corpo Opcional:
    - phoneNumber: String
- [OP2] Actualizar utilizador.
  - URL: api/users/{ID}
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Actualiza o utilizador
  - Corpo Obrigatório:
    - name: String
    - username: String
    - roles: {id: Long, name: String}
    - password: String
  - Corpo Opcional:
    - phoneNumber: String
    - enable: Boolean
  - Parâmetro Obrigatório:
    - ID: Long
- [OP3] Apagar utilizador.
  - URL: api/users/{ID}
  - Método: DELETE
  - Headers: Content-Type: application/json
  - Descrição: Apaga o utilizador
  - Parâmetros Obrigatórios:
    - ID: Long

- [OP4] Desabilitar e habilitar utilizador.
  - URL: api/users/{ID}/ativo
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Desabilita o utilizador ou habilita o utilizador.
  - Corpo Obrigatório:
    - ativo: Boolean
  - Parâmetros Obrigatórios:
    - ID: Long
- [OP5] Pesquisar utilizadores.
  - URL: api/users/
  - Método: GET
  - Headers: Content-Type: application/json
  - Descrição: Lista todos Utilizadores
- [OP6] Autenticar utilizador.
  - URL: api/security/oauth/token
  - Método: POST
  - Headers: Content-Type: application/json
  - Descrição: Token de autenticação do Utilizador
  - Corpo Obrigatório:
    - password: String
    - username: String

3.3.2 Operações Microsserviço Distrito

Este microsserviço será utilizado pelo microsserviço Utilizador para fornecer a localização a que o utilizador pertence.

- [OP7] Criar distrito.
- [OP8] Actualizar distrito.
- [OP9] Apagar distrito.
- [OP10] Desabilitar distrito.
- [OP11] Pesquisar distritos.

3.3.3 Operações Microsserviço Equipamento

O microsserviço equipamento tem a tarefa de gestão de equipamentos, e fornecer dados aos microsserviços histórico e manutenção.

- [OP12] Criar equipamento.
  - URL: api/equipment
  - Método: POST
  - Headers: Content-Type: application/json
  - Descrição: Cria um novo equipamento
  - Corpo Obrigatório:
    - assetName: String
    - assetTag: String
    - available: Boolean
    - category: {categoryId: Long , name: String}
  - Corpo Opcional:
    - purchaseAt: String (date-time)
    - purchaseCost: number
    - serial: String
    - notes: String
- [OP13] Actualizar equipamento.
  - URL: api/equipment/{ID}
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Actualiza o equipamento
  - Corpo Obrigatório:
    - assetName: String
    - assetTag: String
    - available: Boolean
    - category: String
  - Corpo Opcional:
    - purchaseAt: String (date-time)
    - purchaseCost: number
    - serial: String
    - notes: String
  - Parâmetro Obrigatório:
    - assetId: Long
- [OP14] Apagar equipamento.
  - URL: api/equipment/{ID}
  - Método: DELETE

  - Headers: Content-Type: application/json
  - Descrição: Apaga o equipamento
  - Parâmetros Obrigatórios:
    * ID: Long

- [OP15] Desabilitar equipamento.
  - URL: api/equipment/ativo/{ID}
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Desabilita ou habilita o equipamento, está operação faz negação do que estava previamente gravado no sistema.
  - Parâmetros Obrigatórios:
    * ID: Long

- [OP16] Pesquisar equipamento.
  - URL: api/equipment/
  - Método: GET
  - Headers: Content-Type: application/json
  - Descrição: Lista todos Equipamentos

- [OP17] Listar todos os equipamentos disponíveis.
  - URL: api/equipment/available
  - Método: GET
  - Headers: Content-Type: application/json
  - Descrição: Lista todos Equipamentos disponíveis.

- [OP18] Cria nova categoria.
  - URL: api/equipment/categories/
  - Método: POST
  - Headers: Content-Type: application/json
  - Descrição: Cria uma nova categoria do equipamento.
  - Corpo Obrigatório:
    * name: String

- [OP19] Actualizar categorias.
  - URL: api/equipment/categories/{ID}
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Actualiza categoria do equipamento.
  - Corpo Obrigatório:
    * name: String

  - Parâmetro Obrigatório:
    * categoryId: Long

- [OP20] Listar categorias.
  - URL: api/equipment/categories/
  - Método: GET
  - Headers: Content-Type: application/json
  - Descrição: Lista categorias do equipamento.

3.3.4 Operações Microsserviço Histórico

Este microsserviço é responsável pela gestão de empréstimos dos equipamentos, utiliza o microsserviço equipamento para obter a lista de equipamentos que podem ser emprestados.

- [OP21] Criar empréstimo do equipamento ao utilizador.
  - URL: api/historic
  - Método: POST
  - Headers: Content-Type: application/json
  - Descrição: Cria histórico, cria um empréstimo do equipamento ao utilizador.
  - Corpo Obrigatório:
    * userId: Long
    * equipmentId: Long
    * dateCheckout: String(date-time)
  - Corpo Opcional:
    * notes: String

- [OP22] Actualizar empréstimo do equipamento ao utilizador.
  - URL: api/historic/{ID}
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Actualiza o empréstimo do equipamento.
  - Corpo Obrigatório:
    * userId: Long
    * equipmentId: Long
    * dateCheckout: String(date-time)
  - Corpo Opcional:
    * notes: String
    * dateCheckin : String(date-time)
    * checkinStatus: String(date-time)
  - Parâmetro Obrigatório:
    * historicId: Long

- [OP23] Apagar empréstimo do equipamento ao utilizador.
  - URL: api/historic/{ID}
  - Método: DELETE
  - Headers: Content-Type: application/json
  - Descrição: Apaga o empréstimo do equipamento ao utilizador
  - Parâmetro Obrigatório:
    - ID: Long
- [OP24] Criar devolução do equipamento ao utilizador.
  - URL: api/historic/{ID}
  - Método: PUT
  - Headers: Content-Type: application/json
  - Descrição: Devolução do equipamento outrora emprestado.
  - Corpo Obrigatório:
    - userId: Long
    - equipmentId: Long
    - dateCheckout: String(date-time)
    - dateCheckin : String(date-time)
    - checkinStatus: Enum - String [FUNCIONAL, AVARIADO, ROUBADO]
  - Corpo Opcional:
    - notes: String
  - Parâmetro Obrigatório:
    - historicId: Long
- [OP25] Lista do Histórico por equipamento.
  - URL: api/equipment/{ID}
  - Método: GET
  - Headers: Content-Type: application/json
  - Descrição: Lista o Histórico por equipamento
  - Parâmetro Obrigatório:
    - equipmentId: Long
- [OP26] Lista do Histórico por utilizador.
  - URL: api/users/{ID}
  - Método: GET
  - Headers: Content-Type: application/json
  - Descrição: Lista o Histórico por utilizador
  - Parâmetro Obrigatório:
    - userId: Long

3.4 Conclusão

Este capítulo descreve os requisitos de software, faz a identificação dos tipos de utilizadores e suas permissões na interacção com o sistema. É de crucial importância a análise correta dos requisitos de software, pois estes forneçam não só uma visão do objectivo final a ser alcançado, como também definem prioridades no tempo de desenvolvimento do sistema. E por fim, fez-se uma radiografia da arquitectura proposta, onde foram listados todos os microsserviços que compõem a arquitectura suas principais operações por microsserviço e sua especificação. não só uma visão do objectivo final a ser alcançado

4

Implementação

Neste capitulo é descrita a implementação de um protótipo do sistema por forma a validar a sua aplicabilidade. Neste protótipo são implementados três entidades relacionadas entre si: o utilizador, equipamentos e históricos, afim de estudar o seu comportamento e suas relações. Na Secção 4.1 são apresentadas as principais tecnologias que foram utilizadas para a implementação do sistema baseado em microsserviços. A Secção 4.2 descreve a arquitectura, onde será possível observar os vários componentes tecnológicos escolhidos para sua criação, e de seguida é feita uma breve descrição de cada ferramenta escolhida e sua relação com outros componentes. Na última Secção, conclui o capítulo com um resumo.

4.1 Tecnologias

Nesta Secção serão apresentadas as ferramentas tecnológicas utilizadas no processo do desenvolvimento da arquitectura. Por forma a implementar o sistema da forma mais eficiente, rápida e segura, usando as ferramentas que definem o estado da arte. O protótipo utiliza uma variedade de tecnologias, que incluem:

- Spring Framework;

- Node.js;
- GraphQL;
- MySQL;
- OAuth2;
- Eureka;
- Swagger
- Zuul, entre outros.

4.1.1 Spring Framework

Spring é um *framework* de código aberto que facilita o desenvolvimento de sistemas através da linguagem Java usando os conceitos de inversão de controle e injecção de dependências. Sua primeira versão foi desenvolvida por Rod Johnson em Outubro de 2002 e lançada sob a licença Apache 2.0 em Junho de 2003, a versão mais recente é a 5 lançada em 2017 [Wik20f]. O *framework* oferece vários módulos que podem ser utilizados de acordo com as necessidades do projecto. Para o desenho dos microsserviços serão utilizados os seguintes módulos: Spring Boot, Data e Cloud.

Spring Boot

O Spring Boot é o modulo que faz parte do ecossistema Spring, cujo objectivo é de facilitar a criação de sistemas em Spring.

O Spring Boot trabalha seguindo convenções e configurações padrão para abstrair o máximo possível das configurações necessárias de um sistema Spring, bastando definir qual tipo de sistema que se deseja criar, escolher o *starter* apropriado, e o Spring se encarregará de criar as configurações básicas necessárias do sistema que escolheu. O *starter* escolhido contém todas as dependências que o sistema necessita para funcionar, assim sendo, não é necessário preocupar com as dependências do projecto [Spr20d]. Alem disso, o Spring Boot oferece servidores embutidos o que facilita ainda mais o desenvolvimento e testes dos sistemas a desenvolver. Na Figura 4.1, é apresentado o *starter* do microsserviço Utilizador, que foi construído utilizando a *Integrated Development Environment* (IDE) do Spring, o STS Tools [Spr20a] e está por sua vez foi desenvolvida a partir do Eclipse [Ecl20], é possível observar ainda na Figura 4.1 as dependências que foram utilizadas para desenvolver o microsserviço.

Spring Cloud

O Spring Cloud é o modulo que facilita na criação e gestão dos microsserviços, pois ela fornece ferramentas para desenvolver rapidamente alguns dos padrões comuns em sistemas distribuídos (por exemplo, gestão de configuração, *service discovery*, *circuit breaker*, *proxy*, e muito mais) [Spr20c].

Spring Data

O Spring Data *Java Persistence API* (JPA), facilita a implementação de repositórios baseados em JPA. Este módulo lida com suporte aprimorado para camadas de acesso a dados baseadas em JPA. Isso facilita a

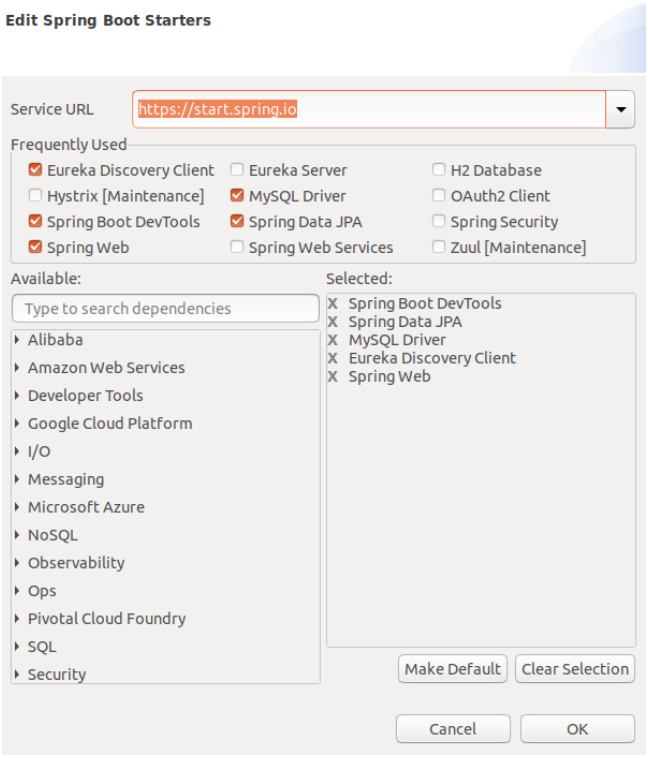

Figura 4.1: *Starter* do Spring Boot

criação de sistemas com tecnologia Spring que usam tecnologias de acesso a dados [Spr20e]. Com o Spring Data não é necessário criar a base de dados manualmente, ela faz um mapeamento directo dos objectos para tabelas da base de dados, que permite actualizar a base de dados de acordo com as alterações dos objectos ou classes, assim ganhando uma maior produtividade no momento de desenvolvimento do sistema.

Existem três recursos principais do Spring Data JPA [Jan20]:

- **Repositório sem código**: O *repository pattern* é um dos mais populares padrões relacionados com a persistência. Esconde os detalhes específicos de implementação do armazenamento de dados e permite-lhe implementar o seu código de negócio a um nível de abstracção mais elevado.
- **Redução do código**: O Spring Data JPA fornece uma implementação por padrão para cada método definido por uma das suas interfaces de repositório. Isto significa que não é necessário implementar as operações básicas de leitura ou escrita.
- **Consultas geradas**: Outro recurso do Spring Data JPA é a geração de consultas a bases de dados com base no nome do método. Se a consulta não for muito complexa, precisamos definir um método em nossa interface de repositório com o nome que começa com *findBy*. Após definir o método, o Spring analisa o nome do método e cria uma consulta de forma dinâmica.

4.1.2 Node.js

O Node.js é um interpretador de JavaScript assíncrono com código aberto orientado a eventos, criado por Ryan Dahl em 2009, focado em migrar a programação do Javascript do cliente (*frontend*) para os servidores, criando aplicações de alta escalabilidade (como um servidor Web), permitindo a manipulação

de milhares de conexões/eventos simultâneas em tempo real numa única máquina física [Wik20b]. O *gateway* implementado em GraphQL foi feito utilizando o Node.js, o mesmo será utilizado pelos clientes para se comunicar com os microsserviços REST através de pedidos HTTP.

O Node.js traz muitos benefícios, dos quais destaco os seguintes [Poi20]:

- ***Single-thread***: Cada sistema terá instância de um único processo com um *event looping* [JS20]. O mecanismo de eventos ajuda o servidor a responder de forma não obstrutiva e torna o servidor altamente escalável.
- **Assíncrono e orientada a eventos**: O Node.js possuí bibliotecas assíncronas, ou seja, sem bloqueio. Isto significa que um servidor baseado em Node.js normalmente não espera o retorno dos dados. O servidor passa para a próxima instrução, e um mecanismo de notificação de eventos do Node.js ajuda o servidor a obter uma resposta do pedido anterior assim que estiver disponível.
- **Muito rápido**: Sendo criada no mecanismo JavaScript V8 do Google Chrome, a biblioteca Node.js. é muito rápida na execução do código.

4.1.3 MySQL

O MySQL é um sistema grátis e *open source* de gestão de base de dados relacional actualmente pertencente a Oracle *Corporation*. Utiliza a linguagem *Structured Query Language* (SQL) e o armazenamento de informações são feitas em tabelas conectadas com chaves e índices de forma relacional.

4.2 Protótipo

O protótipo é mostrado na Figura 4.2, como se pode observar o protótipo contém:

- **GraphQL *Gateway***: Aplicação implementada em GraphQL, com finalidade de ser o único ponto de entrada para o sistema e encapsular a arquitectura interna do sistema.
- **Zuul Proxy e Resource Server**: Este recebe a solicitação do *gateway*, de seguida verifica se utilizador tem autorização, e se tiver, encaminha para o respectivo microsserviço.
- **OAuth Service**: Tem como objectivo autenticar o utilizador, e com o *token* obtido da autenticação, o utilizador poderá aceder o recurso pretendido desde que tenha autorização.
- **Eureka Discovery**: Este serviço comporta-se como um servidor, centralizando as informações dos serviços disponíveis, permitindo que sejam localizados e disponibilizados seus terminais quando necessário e também fornecerá o balanceamento de cargas.
- **Microsserviços Utilizador, Equipamentos e Histórico**: Aplicações implementadas em Spring Boot, com finalidade de fornecer as informações necessárias ao cliente, expondo a base de dados através de Operações CRUD.
- **Cliente Feign**: O Feign é uma biblioteca que ajuda na chamada entre os microsserviços, fornecendo uma forma declarativa de chamar o microsserviço pretendido.

A componente cliente comunica-se com o *gateway* e este por sua vez com o *proxy* que é também um servidor de recurso. O servidor de recurso se comunica com o servidor de OAuth com a finalidade de

autenticar o utilizador. Após a autenticação o utilizador poderá consultar o microsserviço pretendido, para mais detalhes ver a Secção 4.2.10.

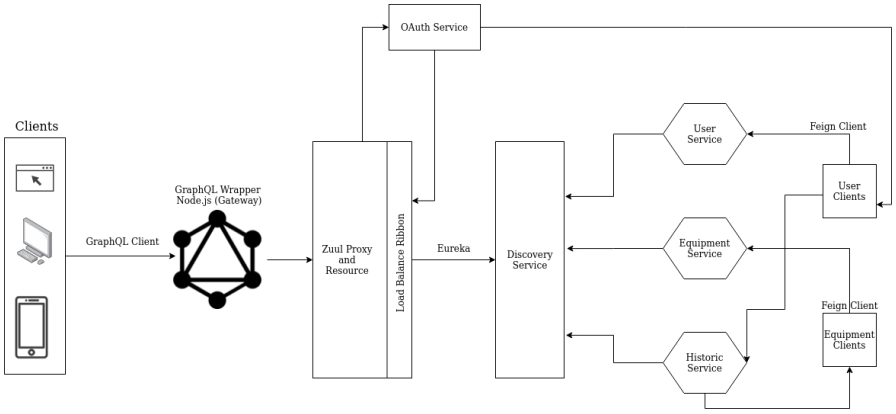

Figura 4.2: Protótipo em microsserviços

4.2.1 *Gateway* API GraphQL

O *gateway* API é um serviço sobre microsserviços que expõe e combina a funcionalidade dos microsserviços para uso dos sistemas clientes. É responsável pelo encaminhamento de pedidos, composição da API. Na Figura 4.2, é possível observar que a camada GraphQL, que foi implementada utilizando o Yoga (ver Secção 2.4.2), está irá encapsular a arquitectura interna do sistema e fornecer uma API personalizada para os clientes, passando assim, a ser o único ponto de entrada para consumir os microsserviços, esta camada, será responsável por buscar dados dos microsserviços subjacentes usando o protocolo HTTP. Os clientes farão pedidos somente dos dados que precisam para a API GraphQL uma vez que, diferentes clientes precisam de dados ligeiramente diferentes.

```
type Equipment{
    assetId: ID
    ...
}

type User {
    userId : ID
    ...
}
type Category{
    ...
}

type Historic{
    historicId: ID
    user: User
    equipmentId: Int
    ...
}

enum CheckinStatus{
    FUNCIONAL,
    AVARIADO,
    ROUBADO
}
```

```
type Query {
    user(id: ID): User
    equipment(id: ID): Equipment
    ...
}
...
```

Listagem 4.1: Parte do código fonte do esquema da arquitectura em GraphQL

A Listagem 4.1, mostra parte do esquema do *gateway* da API baseado em GraphQL. Onde são definidos vários tipos de objectos (User, Historic, Equipment, ...). Os tipos de objectos corresponde às entidades dos microsserviços Utilizador, Histórico, Equipamento. A Listagem possui também um tipo de objecto *query* que define os pedidos que o esquema poderá realizar. Quando o servidor GraphQL executa um pedido, ele deve recuperar os dados solicitados em uma ou mais bases de dados.

4.2.2 *Service discovery* - Eureka

O Eureka *service discovery* é um registo que permite que uma instância de um microsserviço se registe quando inicia. Dessa forma, o cliente pode pedir o registo para descobrir as instâncias disponíveis do serviço antes de chamá-lo. Cada microsserviço regista-se no *service discovery* de forma dinâmica, podendo ter mais de uma instância por microsserviço.

O Eureka é o *service discovery* do Netflix que é usado como servidor e cliente. O servidor Eureka contém as informações sobre todos os clientes que se registam no servidor. Todos microsserviços serão registados no servidor Eureka e o mesmo tem conhecimento de todos os microsserviços em execução em cada porta e endereço IP. O servidor Eureka recebe mensagens de *heartbeat* de cada instância, e, se não receber um *heartbeat* em 30 segundos a instância será removida do registo.

Para exemplificar a sua funcionalidade, imaginemos que o microsserviço Histórico precisa obter os dados de um utilizador. O microsserviço Histórico precisa saber qual é o endereço e a porta para enviar o pedido. Para que o microsserviço Utilizador seja descoberto pelo Histórico, é necessário que ambos estejam registados no servidor de Eureka como clientes de Eureka. Assim, antes de enviar o pedido o microsserviço Histórico irá descobrir o microsserviço Utilizador informando apenas o nome, e está funcionalidade é independente do número de instâncias do microsserviço Utilizador.

Para criar um servidor Eureka, primeiro é adicionada a biblioteca `spring-cloud-starter-netflix-eureka-server` nas dependências do projecto Spring Boot. De seguida é activado o servidor Eureka com as seguintes anotações `@SpringBootApplication` e `@EnableEurekaServer`, como podemos ver na Listagem 4.2. Depois de anotar a classe principal, o projecto é configurado com as propriedades do servidor Eureka, no ficheiro `application.properties`, como mostrado na Listagem 4.3.

```
@EnableEurekaServer
@SpringBootApplication
public class ServicoEurekaServerApplication {

  public static void main(String[] args) {
    SpringApplication.run(ServicoEurekaServerApplication.class, args);
  }

}
```

Listagem 4.2: Classe principal do servidor Eureka

```
spring.application.name= servico-eureka-server
server.port= 8761
eureka.client.register-with-eureka= false
eureka.client.fetch-registry= false
```

Listagem 4.3: Configurações do ficheiro *application.properties* do servidor Eureka

4.2.3 Zuul *Proxy*

Este *proxy* tem como objectivo facilitar a invocação dos microsserviços, fornecendo um encaminhamento dinâmico, isto é, em vez de ter várias URLs para serem invocadas pelo *gateway* do GraphQL, apenas será disponibilizada um URL, como também irá facilitar a adição da camada que fará a gestão da autenticação e autorização.

O Zuul é implementado adicionando as seguintes dependências: `spring-cloud-starter-netflix-zuul` e `spring-cloud-starter-netflix-eureka-client` ao projecto. Depois, a classe principal do projecto é anotada com as anotações `@EnableZuulProxy` e `@EnableEurekaClient`, permitindo assim que o serviço actue como um *proxy* dinâmico e como um cliente Eureka.

Por fim, o projecto é configurado modificando o ficheiro `application.properties`, para definir o nome do microsserviço e a porta que irá se registar no Eureka. De seguida, são configurados os mapeamentos dos microsserviços e suas URLs, por exemplo para o microsserviço Equipamento, a sintaxe `zuul.routes.equipment.service-id`, informa o nome do sistema registada no servidor Eureka, este valor é atribuído em cada sistema na propriedade do `application.properties`. E por fim, a sintaxe `zuul.routes.equipment.path`, define a URL para acessar os dados dos equipamentos, no caso `/api/equipment/`, como mostrado na Listagem 4.4.

Assim, com as URLs e a porta 8090 definida, estes serão utilizados no *gateway* para fazer pedidos aos respectivos microsserviços, por exemplo se precisarmos recuperar os detalhes de um dado equipamento a URL a se utilizar poderia ser, `http://localhost:8090/api/equipment/5`, onde o número 5 é o código do equipamento que estamos recuperando os detalhes.

```
spring.application.name=sga-gateway
server.port=8090

zuul.routes.user.service-id=user-microservice
zuul.routes.user.path=/api/users/**

zuul.routes.historic.service-id=delivery-reception-service
zuul.routes.historic.path=/api/historic/**

zuul.routes.equipment.service-id=it-equipment-microservice
zuul.routes.equipment.path=/api/equipment/**
```

Listagem 4.4: Configurações do ficheiro *application.properties* do servidor Zuul

4.2.4 Feign

O Feign fornece uma abstracção sobre os pedidos em REST por meio de anotações, através da qual os microsserviços podem usar para comunicar entre si, sem precisar escrever código detalhado do cliente REST. Primeiramente definimos uma interface Java, de seguida anotamos essa interface com `FeignClient( name ="nome do microsserviço")` para mapear o nome do microsserviço que pretendemos comunicar com ele,

referir que este nome deve ser o mesmo com a qual o microsserviço foi configurado no arquivo `application.properties`. A estrutura Spring Cloud irá gerar dinamicamente uma classe *proxy* que será utilizada para invocar o serviço REST. Para activar o cliente Feign é necessário adicionar a anotação `@EnableFeignClients` à classe principal do projecto.

4.2.5 Microsserviço Utilizador

Este microsserviço é responsável pelo CRUD do utilizador e suas permissões, alem disso, tem a finalidade de auxiliar na autenticação de utilizadores e na associação de um recurso a um utilizador. Por exemplo o microsserviço Histórico, utiliza o cliente Feign utilizador para fazer pedidos de forma a associar o equipamento ao utilizador no acto de empréstimo ou devolução e para autenticação o servidor OAuth, faz pedidos REST a fim de ter os dados do utilizador. O microsserviço foi construído utilizando o Spring Boot com JPA, tem como dependência o cliente Eureka e este foi configurado para que porta do servidor e a instância Eureka sejam atribuídos de forma dinâmica (ver Listagem 4.5), podendo assim ter mais de uma instância do microsserviço Utilizador, se necessário. A base de dados do microsserviço é composta por três tabelas, mostrada na Figura 4.3, as mesmas foram criadas automaticamente pelo Spring Data a partir das classes de modelo.

```
spring.application.name=user-microservice
server.port=${PORT:0}
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance_id:$
    {random.value}}
...
```

Listagem 4.5: Configurações do ficheiro *application.properties* do microsserviço Utilizador

- *user*: tabela que contem os dados do utilizador.
- *role*: tabela com as permissões do utilizador.
- *user_role*: associação muitos para muitos entre user e role.

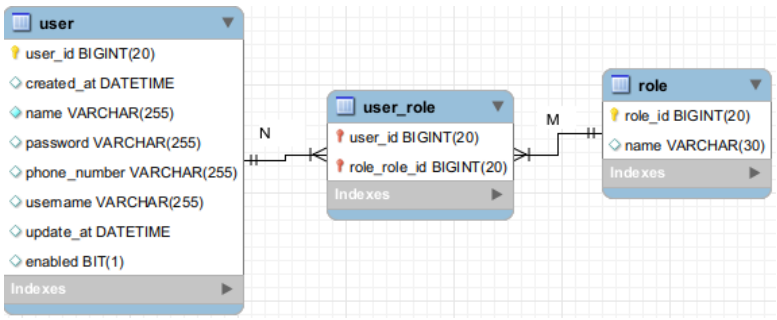

Figura 4.3: Base de dados microsserviço Utilizador

4.2.6 Microsserviço Equipamentos e Histórico

Os microsserviços Equipamentos e Histórico foram construídos usando o Spring Boot com JPA, o microsserviço Equipamento é responsável pelo CRUD dos equipamentos e o microsserviço Histórico pelo CRUD

do histórico (empréstimo ou devolução). O microsserviço Histórico faz pedidos ao microsserviço equipamentos com auxilio do Feign, a fim de criar empréstimo ou devolução de equipamentos. No acto do empréstimo o microsserviço Histórico, faz pedidos ao microsserviço Equipamento para obter a lista dos equipamentos possíveis de serem emprestados, os não ocupados, e no processo de devolução é actualizado o equipamento no Microsserviço equipamento como disponível ou não, o equipamento fica disponível se na devolução estiver em estado funcional, caso contrario o equipamento continua indisponível no microsserviço Equipamento.

4.2.7 Documentação da API

Para documentar a API dos microsserviços Utilizador, Equipamento e Histórico apresentada na Secção 3.3 foi utilizado o *framework* Swagger. O Swagger é um conjunto de regras (em outras palavras, uma especificação) para um formato que descreve APIs REST. O formato é tanto legível por máquina quanto por humanos. Como resultado, ele pode ser usado para compartilhar documentação entre os *stakeholders*, e a equipa de desenvolvimento, mas também pode ser usado por várias ferramentas para automatizar processos relacionados à API [Rat20]. A especificação com Swagger pode incluir informações como [Swa20]:

- Quais são todas as operações que a API suporta?
- Quais são os parâmetros da API e o que retorna?
- Se a API precisa de autorização.
- Pode também conter informações como: termos, contacto e licença para o uso da API.

A especificação da API usando o Swagger pode ser escrita manualmente ou automaticamente através de anotações feitas no código fonte, sendo que está última opção é a que foi utilizada. Portanto para criação da especificação dos microsserviços acima mencionados, foi necessário adicionar em cada microsserviço as dependências do *framework* Swagger (ver Figura 4.4).

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

Figura 4.4: Dependências do *framework* Swagger

De seguida, é criada uma classe de configuração `SwaggerConfig` (ver Listagem 4.6) em cada microsserviço, onde conterá todas as configurações, tais como: pacote a ser percorrido pelo Swagger de modo a gerar automaticamente a especificação, *metadados*, autenticação, entre outros.

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

  @Bean
  public Docket productApi() {
    return new Docket(DocumentationType.SWAGGER_2)
```

```
        .select().
          apis(RequestHandlerSelectors.basePackage("mz.org.fgh.sga.app.historic"))
        .paths(PathSelectors.any()).build().apiInfo(metaData())
        .securitySchemes(Arrays.asList(apiKey()));
}

private ApiInfo metaData() {
  return new ApiInfoBuilder().title("SGA HISTORIC MICROSERVICE API").description("\"
  Description of USER MICROSERVICE API\"")
}

private ApiKey apiKey() {
    return new ApiKey("jwtToken", "Authorization", "header");
}
```

Listagem 4.6: Parte do código fonte que contém a configuração do Swagger no microsserviço Histórico

Após as configurações em cada microsserviço, a documentação das APIs podem ser encontradas nos seguintes endereços:

- Microsserviço Utilizador: http://endereço-do-servidor:8090/api/users/swagger-ui.html
- Microsserviço Equipamento: http://endereço-do-servidor:8090/api/equipment/swagger-ui.html
- Microsserviço Histórico: http://endereço-do-servidor:8090/api/historic/swagger-ui.html

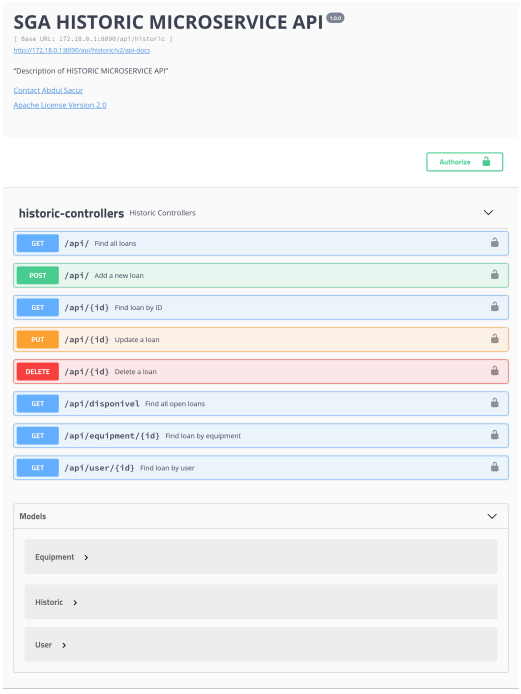

Figura 4.5: Operações do microsserviço Histórico

A Figura 4.5, lista todas as operações por microsserviço, é possível ainda encontrar a especificação por cada operação, para tal é necessário clicar na operação que se pretende conhecer a especificação. A Figura 4.6 mostra a especificação da operação *Find loan by user*.

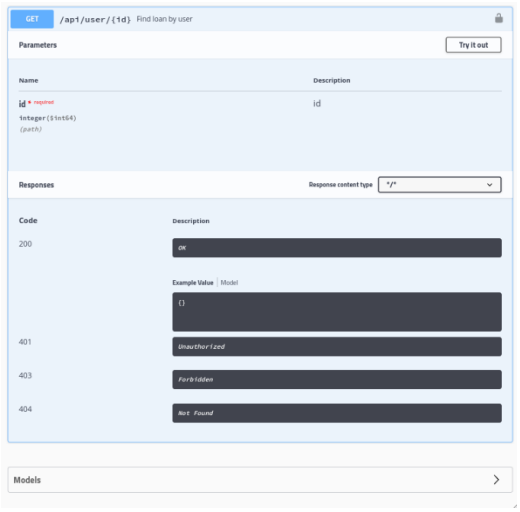

Figura 4.6: Parte da especificação do microsserviço Histórico

4.2.8 OAuth Service

O OAuth2 é uma estrutura de autorização e autenticação de segurança baseada em *token* que divide a segurança em quatro componentes [Car17].

- *Resource Server* - é o recurso que pretendemos proteger e assegurar que somente os utilizadores autenticados tenham o devido acesso, exemplo: microsserviço Equipamento.
- *Resource Owner* - define quais os sistemas que podem pedir seu serviço e o que podem fazer com o mesmo (leitura e escrita). Neste processo regista-se o nome do sistema juntamente com uma chave secreta. A combinação do nome do sistema e da chave secreta faz parte das credenciais transmitidas ao autenticar um *token* OAuth2, como pode ser visto na Listagem 4.7.

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
  clients.inMemory().withClient("graphql").secret(passwordEncoder.encode("
  palavrasecreta")).scopes("read", "write")
      .authorizedGrantTypes("password", "refresh_token").
  accessTokenValiditySeconds(3600)
      .refresh\textit{token} ValiditySeconds(3600);
}
```

Listagem 4.7: Parte do código fonte que contém a configuração do *Resource Owner*

- *Client* - este é o sistema que fará o pedido ao serviço em nome do utilizador, por exemplo o *gateway* Node.js.
- *Authentication Server* - responsável por autenticar o utilizador e fornecer um *token* de autorização, que será utilizado quando fizer um pedido ao *resource server*.

Os quatro componentes interagem entre si para autenticar o utilizador (ver Secção 4.2.10) . O utilizador apenas precisa apresentar suas credenciais, se a autenticação acontecer com sucesso um *token* será fornecido. Desta forma, com o mesmo *token* ou *refresh token* o utilizador terá acessos aos vários microsserviços.

Para criar o *Authentication Server* cuja a responsabilidade é de autenticar o utilizador, primeiro são adicionadas as dependências `spring-cloud-starter-oauth2` e `spring-cloud-starter-netflix-eureka-client`. De

seguida, na classe principal são colocadas as anotações `@EnableFeignClients` e `@EnableEurekaClient`, desta forma o microsserviço será registado no servidor Eureka. E com o suporte da biblioteca Feign (ver Listagem 4.8), é feita o pedido ao microsserviço Utilizador, usando o método que foi declarado na interface `UserFeignClient`. O retorno do pedido é passado para uma classe especial, `UserDetailsService`, que faz parte da estrutura do Spring Security, usada para recuperar as informações de autenticação e autorização do utilizador (ver Listagem 4.9).

```
@FeignClient(name ="user-microservice")
public interface UserFeignClient {

  @GetMapping("/user/{username}")
  public SystemUser findByUsername(@PathVariable String username);
}
```

Listagem 4.8: Interface Feign utilizada para se comunicar com o microsserviço Utilizador

```
@Service
public class UserService implements UserDetailsService {

  @Autowired
  private UserFeignClient client;

  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
     {
    SystemUser user = client.findByUsername(username);
      ....
    return new User(user.getUsername(), user.getPassword(), user.getEnabled(), true, true
    , true, authorities);
  }

}
```

Listagem 4.9: Classe UserService do microsserviço *OAuth*

Na Listagem 4.9, o `UserDetailsService` é utilizado na classe de configuração `SpringSecurityConfig` que estende a classe `WebSecurityConfigurerAdapter` usada pelo Spring Security para manipular a autenticação do utilizador usando a encriptação *Bcrypt* [Wik19a]. Por fim, é criada a classe de configuração `AuthorizationServerConfig`, que é anotada com as anotações `@Configuration` e `@EnableAuthorizationServer`, este último, activa o `AuthorizationServer` disponibilizando um `TokenEndpoint(/oauth/token)`. O URL será utilizado para obter o *JSON Web Token* (JWT), este é um padrão que define uma forma compacta e independente de transmitir objectos JSON com segurança entre diferentes sistemas.

4.2.9 Resource Service

O Zuul tem a responsabilidade de encaminhar as solicitações provenientes do *gateway* GraphQL para os diferentes microsserviços, tornando assim o local perfeito para adicionar a camada de servidor de recurso. Assim sendo, para cada URL que o utilizador precisar de consultar, o mesmo deverá enviar um *token* contendo a autorização obtida no microsserviço *OAuth*. Se não tiver permissão para efectuar a operação será apresentado um erro.

No microsserviço Zuul, foi acrescentada a dependência do OAuth2 e activado o servidor de recurso com anotação `@EnableResourceServer`, na classe `ResourceServerConfig`. A classe `ResourceServerConfig` é responsável pela verificação da autorização das solicitações recebidas no Zuul. O método `configure` tem como objectivo configurar as políticas de acesso dos URLs (ver Listagem 4.10). Em particular, é feito o seguinte:

1. Dar acesso a todos na URL, */api/security/oauth/***;
2. Todos os outros URLs precisam de autorização;
3. ADMIN e USER tem permissão de leitura;
4. ADMIN tem permissão de escrita.

```
@Override
public void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests().antMatchers("/api/security/oauth/**").permitAll()

.antMatchers(HttpMethod.GET, "/api/users/**", "/api/historic/**", "/api/equipment/**")
.hasAnyRole("ADMIN", "USER").antMatchers("/api/users/**", "/api/equipment/**", "/api/
  historic/**")
.hasRole("ADMIN").anyRequest().authenticated();
}
```

Listagem 4.10: Parte da classe de configuração responsável pela autorização aos microsserviços

4.2.10 Fluxo de Comunicação

O presente fluxo da Figura 4.7 apresenta a relação entre os diferentes microsserviços. A seguir, é descrita a sequência de comunicação, onde podemos destacar o seguinte:

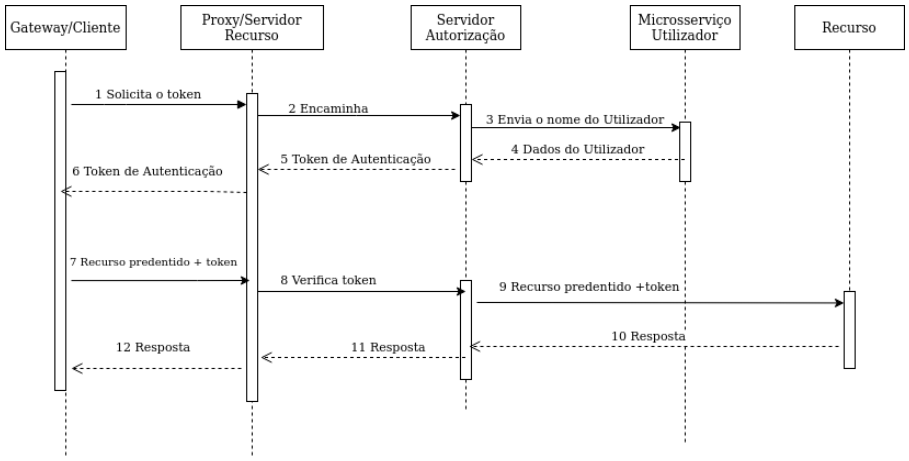

Figura 4.7: Diagrama de Sequência de comunicação entre os microsserviços

- Cliente solicita um *token* de acesso.
- O *gateway* encaminha para o *proxy*, e este por sua vez encaminha para o servidor de autenticação.
- O servidor de autenticação faz pedido ao microsserviço de Utilizador a fim de obter os dados do utilizador que está solicitando o *token* de autenticação.
- O servidor de autenticação redireciona para o *proxy* se a solicitação foi autenticada ou não.
- Se não estiver autenticado, o *proxy* redireccionará para o cliente com uma resposta não autenticada.
- Se autenticado, o *proxy* envia para o cliente o *token* de autenticação.

- Com o token, o cliente faz pedidos aos microsserviços, e este pode ser autorizado ou não.
- Se não estiver autorizado, o *proxy* redireccionará para o cliente com uma resposta não autorizado.
- Se autorizado, o *proxy* envia o cliente a resposta do seu pedido.

4.2.11 Conclusão

Para construção da arquitectura foram utilizadas tecnologias como o Spring Boot, pois facilita o processo de configuração e *deploy* do sistema. Foi também usado o Spring Data e Cloud, onde o Spring Data foi responsável por tratar o acesso aos dados, enquanto que o Spring Cloud visava facilitar o desenvolvimento e gestão dos microsserviços seguindo alguns padrões do sistemas distribuído, tais como: o *service discovery* e *proxy*. De seguida, foi configurada a camada de segurança que utiliza o padrão OAuth2, e foi explicada o fluxo de comunicação da camada com outros elementos do protótipo. Portanto, ficou implementado um protótipo funcional que demonstra a "viabilidade" e utilidade do conceito.

5

Avaliação

Neste capitulo é feita uma avaliação crítica sobre o trabalho realizado, descrevendo se a arquitectura e as ferramentas utilizadas são as mais apropriadas para implementar o trabalho.

Por ser um tema sugestivo para o panorama do caso da Friends in Global Health (FGH), foi assumido o desafio de trazer uma nova abordagem na materialização da solução proposta neste trabalho, de modo a que se estivesse alinhado com as novas tendências do desenho e implementação de soluções informatizadas.

A presente dissertação descreve a análise e implementação da prova de conceito para um sistema baseado em microsserviços, cujo foco visa responder aos desafios da insuficiência de uma ferramenta informatizada que ajude na monitorização dos equipamentos informáticos na FGH.

A arquitectura baseada em microsserviços foi usada no desenvolvimento do sistema, permitindo um desenvolvimento rápido, facilidade de escalar, pois os seus serviços estão isolados e oferece também uma liberdade no uso de tecnologias durante o processo de desenvolvimento.

Sistemas que obedecem a arquitectura baseadas em microsserviços estão actualmente sendo usados por muitas organizações como: Amazon, Netflix, Uber e Spotify, o que mostra a sua relevância.

Fazendo uma avaliação global sobre o trabalho descrito nesta dissertação, pode-se avaliar o grau da sua pertinência aos olhos da FGH nos seguintes aspectos:

- **Flexibilidade**: O sistema apresenta uma arquitectura de fácil implementação e manutenção, o que torna este um sistema muito leve em termos de complexidade na comunicação e transacção de informação entre os microsserviços associados ao utilizador final.
- **Confiabilidade**: Tendo sido desenhado especificamente para um cenário particular, o presente sistema responde sobremaneira aos grandes desafios a que a equipa da logística da FGH há tanto tempo vem enfrentando, apresentando assim a resposta às necessidade em tempo útil que consequentemente ajudarão na tomada de decisão a vários níveis da organização relativas à logística de equipamentos de tecnologia de informação sob alçada da equipa IT. Este aspecto concorre para a redução significativa de processo burocráticos que comprometem o cumprimento efectivo de certas actividades e necessidades relacionadas com a logística de equipamentos da organização. Pode-se considerar que a implementação deste sistema vai catapultar àquelas que são as aspirações de melhor prestar serviços aos utilizadores de equipamentos de tecnologia de informação destinadas às áreas de actuação da organização .
- **Escalabilidade**: A arquitectura do presente sistema mostra-se estar preparada para se adequar às realidades a que lhes forem impostas, tanto no aspecto de infraestruturas física (servidores e outro *hardware*) como também da parte lógica do negócio da organização.

As ferramentas usadas ofereceram algum conforto no desenvolvimento do sistema, primeiro por terem muito suporte na comunidade de desenvolvedores que têm usado estas ferramentas, e também por serem de fácil compreensão. Com excepção da ferramenta do *gateway* do GraphQL, que utilizou o Yoga para o seu desenvolvimento por este oferecer menor curva de aprendizagem e configuração em relação ao Apollo Server.

5.1 Conclusão

É evidente que a implementação deste sistema oferecerá um valor acrescentando no cumprimento das metas e responsabilidade da organização no campo em que ela actua. Não descurando o mérito que a mesma já têm em África na resposta de assistência a pacientes vivendo com HIV e SIDA. São várias as inovações que a organização tem implementado para melhorar a sua prestação no País, e esta será uma delas no âmbito de transparência e boa governação Institucional.

6

Conclusões e Trabalho Futuro

Neste capitulo são apresentadas as principais conclusões relativamente ao trabalho no âmbito desta dissertação, o trabalho futuro e algumas considerações finais.

Após a conclusão do trabalho, pode-se concluir que os objectivos estabelecidos no início do mesmo foram alcançados.

No estado de arte foram identificadas as diferenças entre a arquitectura monolítica e de microsserviços. Comparativamente a arquitectura de microsserviços, oferece a vantagem de ser menos complexa para sua manutenção, pois é composta por serviços isolados. Outra vantagem dos microsserviços é a capacidade de serem escalados facilmente. Por exemplo, o microsserviço responsável pela gestão dos utilizadores pode exigir mais de uma instância para melhor desempenho do sistema como um todo. Também, foram estudadas as características e comunicação da arquitectura em microsserviço, similarmente foram mostradas os possíveis desafios que a arquitectura apresenta em relação a monolítica.

A arquitectura proposta é baseada no modelo de negócios e atende aos requisitos que foram definidos nos estágios iniciais. Uma vez que o actual cenário passa pelo processo de preenchimento manual, baseado

em arquivos no formato em papel, algo que com a implementação da presente solução transita para um paradigma automatizado.

Foi desenvolvido um protótipo como prova de conceito para validar a arquitectura. A sua implementação foi feita escolhendo três microsserviços da estrutura proposta, que representam uma utilização real do sistema. A proposta final, foi uma arquitectura inovadora que combina o GraphQL e o REST. O GraphQL na arquitectura tem a função de *gateway*, que apresenta os seguintes benefícios: disponibiliza um único URI para todos os pedidos, carregamento eficiente dos dados, pois dá aos clientes o controlo sobre os dados que são retornados do servidor. Outro benefício é que, embora a API seja muito mais flexível, esta abordagem reduz significativamente o esforço de desenvolvimento. Isso porque escrevemos o código do lado do servidor usando uma estrutura de execução de consulta que é projectada para suportar a composição e projecções da API. E o REST foi utilizado para o desenvolvimento e comunicação dos microsserviços como também também na comunicação entre o *gateway* e o *proxy* Zuul.

O sistema vai de encontro com o que tinha sido planeado inicialmente, responde às necessidades da FGH e é um ponto de partida para um sistema que permite fazer toda a gestão dos equipamentos, algo essencial na FGH.

6.1 Trabalho Futuro

Neste trabalho como mencionado nas secções anteriores, foi estudado e implementado um modelo de arquitectura inovador para o desenvolvimento de serviços Web. No entanto, muitas das suas características, ou vantagens ainda estão em debate. Nesta Secção são apresentados alguns possíveis pontos de trabalho futuro:

- Primeiramente criar métodos de validação dos dados em todos os microsserviços, e de seguida terminar o desenvolvimento dos restantes microsserviços;

- Adicionar o microsserviço de auditoria, para ser possível saber que utilizador criou ou modificou um item, bem como saber saber a data e hora que a transacção ocorreu.

- Adicionar a camada de *circuit breaker*: com este padrão, se a chamada para o microsserviço exceder o limite de tentativas configuradas, ele passará do estado fechado para aberto. Assim, a próxima falha fará com que o circuito se abra e não responda por um tempo. Uma vez no estado aberto, em vez de tentar enviar pedidos ao microsserviço que está a falhar, os pedidos entram em curto-circuito. Desde modo, sempre que possível, será executado um *fallback* apropriado, encaminhando para um serviço diferente ou retornar uma resposta em cache. Após esse tempo, entra num estado semiaberto e, se não falhar, o circuito é fechado novamente; em caso de falha, retorna ao estado aberto [BP18].

- Adicionar o Spring Cloud Configuration [Nas20], que é um padrão de configuração centralizada em que a gestão de configuração de todos os serviços é feita num repositório central, em vez de ser distribuída por microsserviços individuais. Cada microsserviço extrai a sua configuração do repositório central durante inicialização [Han20].

- Criar uma biblioteca comum em Java, para evitar a duplicação de código entre os microsserviços.

- Incorporar a arquitectura o Zipkin, que é um sistema de rastreio distribuído que ajuda a recolher dados de tempo de resposta necessários para solucionar problemas de latência nas arquitecturas de microsserviços. As suas características incluem tanto a recolha como a pesquisa destes dados [Zip20].

6.2 Considerações Finais

Pessoalmente estou muito satisfeito com a escolha e a conclusão deste trabalho, pois ele me permitiu adquirir alguns conhecimentos que não obtive durante o mestrado, além de fortalecer as bases em certos aspectos sobre os quais eu já possuía. Especificamente, ele permitiu-me conhecer um modelo de arquitectura que me era desconhecido até o inicio do trabalho. Acredito e espero que este projecto de dissertação seja muito útil para todos os interessados na implementação deste modelo de arquitectura.

Bibliografia

[Agi20] Scale Agile. Nonfunctional requirements. https://www.scaledagileframework.com/nonfunctional-requirements/, 2020. [Online; accessed 06-Setembro-2020].

[Ama19] Amazon. Microsserviços. https://aws.amazon.com/pt/microservices/, 2019. [Online; accessed 06-Outubro-2019].

[Amu08] Mike Amundsen. Rest - the short version. http://exyus.com/articles/rest-the-short-version/, 2008. [Online; accessed 28-Setembro-2019].

[Ang20] Angular. Angular. https://angular.io/, 2020. [Online; accessed 19-Julho-2020].

[Ani20] Mitchell Anicas. Uma introdução ao oauth 2. https://www.digitalocean.com/community/tutorials/uma-introducao-ao-oauth-2-pt, 2020. [Online; accessed 20-Agosto-2020].

[Avr17] Shif Ben Avraham. What is rest — a simple explanation for beginners, part 2: Rest constraints. https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-2-rest-constraints-129a4b69a582, 2017. [Online; accessed 25-Setembro-2019].

[BP18] M. Bruce and P.A. Pereira. *Microservices in Action*. Manning Publications, 2018.

[Car17] John Carnell. *Spring Microservices in Action*. Manning Publications Co., USA, 1st edition, 2017.

[Cor20] Oracle Corporation. Java platform, enterprise edition. https://www.oracle.com/java/technologies/java-ee-glance.html, 2020. [Online; accessed 22-Julho-2020].

[Ecl20] Eclipse. Eclipse. https://www.eclipse.org/, 2020. [Online; accessed 18-Agosto-2020].

[Edu20] IBM Cloud Education. Soa (service-oriented architecture). https://www.ibm.com/cloud/learn/soa, 2020. [Online; accessed 18-Julho-2020].

[Exp20] Express. Express. https://expressjs.com/, 2020. [Online; accessed 21-Julho-2020].

[FL14] Martin Fowler and James Lewis. Microservices. https://martinfowler.com/articles/microservices.html, 2014. [Online; accessed 06-Outubro-2019].

[FT00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[Gra19] Graphql. Graphql. https://graphql.org/learn/, 2019. [Online; accessed 01-Outubro-2019].

[Gur20] Guru. What is a functional requirement? specification, types, examples. https://www.guru99.com/functional-requirement-specification-example.html, 2020. [Online; accessed 28-Agosto-2020].

[Han19] Akira Hanashiro. *GraphQL A revolucionária linguagem de consulta e manipulação de dados para APIs*. Caso do Código, 2019.

[Han20] Brian Hannaway. Microservice configuration: Spring cloud config server tutorial. https://dzone.com/articles/configuring-micro-services-spring-cloud-config-ser, 2020. [Online; accessed 11-Julho-2020].

[Jan20] Thorben Janssen. What is spring data jpa? and why should you use it? https://thorben-janssen.com/what-is-spring-data-jpa-and-why-should-you-use-it/, 2020. [Online; accessed 24-Julho-2020].

[Jav20a] Java. Java. https://www.java.com/, 2020. [Online; accessed 20-Julho-2020].

[Jav20b] Javatpoint. Jax-rs vs spring rest. https://www.javatpoint.com/spring-mvc-tutorial, 2020. [Online; accessed 20-Julho-2020].

[Jer20] Jersey. About jersey. https://github.com/eclipse-ee4j/jersey/, 2020. [Online; accessed 21-Setembro-2020].

[JS20] Node JS. Node js. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/, 2020. [Online; accessed 18-Agosto-2020].

[Kaf20] Apache Kafka. Apache kafka. https://kafka.apache.org/, 2020. [Online; accessed 19-Julho-2020].

[LEA20] LEARNJAVA. Spring mvc tutorial. https://learnjava.co.in/jax-rs-vs-spring-rest/, 2020. [Online; accessed 21-Julho-2020].

[Mic19] Microsoft. Comunicação em uma arquitetura de microsserviço. https://docs.microsoft.com/pt-br/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture, 2019. [Online; accessed 12-Outubro-2019].

[Nas20] Wladimilson Nascimento. Criando uma api rest com o spring boot. https://www.treinaweb.com.br/blog/criando-uma-api-rest-com-o-spring-boot/, 2020. [Online; accessed 23-Julho-2020].

[New15] S. Newman. *Building Microservices*. O'Reilly Media, 2015.

[Poi20] Tutorials Point. Node.js tutorial. https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm, 2020. [Online; accessed 24-Julho-2020].

[Pri20] Prisma. graphql-yoga. https://github.com/prisma-labs/graphql-yoga, 2020. [Online; accessed 22-Julho-2020].

[Rab19] Eduardo Rabelo. Comunicação em uma arquitetura de microsserviço. https://medium.com/@oieduardorabelo/micro-servi2019. [Online; accessed 12-Outubro-2019].

[Rab20] RabbitMQ. Rabbitmq. https://www.rabbitmq.com/, 2020. [Online; accessed 19-Julho-2020].

[Rat20] Ron Ratovsky. Getting started with swagger [i] - what is swagger? https://swagger.io/blog/api-development/getting-started-with-swagger-i-what-is-swagger/, 2020. [Online; accessed 28-Agosto-2020].

[Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018.

[Smi20] Cameron Smith. Advantages and disadvantages of graphql. https://stablekernel.com/article/advantages-and-disadvantages-of-graphql/, 2020. [Online; accessed 04-Outubro-2019].

[Som15] I. Sommerville. *Software Engineering*. Pearson Education, 2015.

[Spr20a] Spring. Spring. https://spring.io/tools, 2020. [Online; accessed 18-Agosto-2020].

[Spr20b] Spring. Spring amqp. https://spring.io/projects/spring-amqp, 2020. [Online; accessed 19-Julho-2020].

[Spr20c] Spring. Spring cloud. https://spring.io/projects/spring-cloud, 2020. [Online; accessed 24-Julho-2020].

[Spr20d] Spring. Spring cloud config. https://cloud.spring.io/spring-cloud-config/reference/html/, 2020. [Online; accessed 22-Julho-2020].

[Spr20e] Spring. Spring data jpa. https://spring.io/projects/spring-data-jpa, 2020. [Online; accessed 24-Julho-2020].

[Sus20] Andrej Suschevich. 10 popular java frameworks. https://medium.com/javarevisited/10-popular-java-frameworks-for-web-applications-691c28f6c182, 2020. [Online; accessed 21-Setembro-2020].

[Swa20] Swagger. What is swagger? https://swagger.io/docs/specification/2-0/what-is-swagger/, 2020. [Online; accessed 28-Agosto-2020].

[tG19] How to GraphQL. Basics tutorial - introduction. https://www.howtographql.com/basics/0-introduction/, 2019. [Online; accessed 04-Outubro-2019].

[Wei14] H.L. Weissmann. *Vire o jogo com Spring Framework*. Casa do Código, 2014.

[Wie19] R. Wieruch. *The Road to Graphql: Your Journey to Master Pragmatic Graphql in JavaScript with React.Js and Node.Js*. Leanpub, 2019.

[Wik19a] Wikipédia. Bcrypt. https://en.wikipedia.org/wiki/Bcrypt, 2019. [Online; accessed 23-Agosto-2020].

[Wik19b] Wikipédia. Graphql. https://pt.wikipedia.org/wiki/GraphQL, 2019. [Online; accessed 01-Outubro-2019].

[Wik19c] Wikipédia. Rest. https://pt.wikipedia.org/wiki/REST, 2019. [Online; accessed 23-Setembro-2019].

[Wik20a] Wikipédia. Javascript. https://pt.wikipedia.org/wiki/JavaScript, 2020. [Online; accessed 21-Julho-2020].

[Wik20b] Wikipédia. Node.js — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.php?title=Node.js&oldid=58175648, 2020. [Online; accessed 11-Junho-2020].

[Wik20c] Wikipédia. Php. https://pt.wikipedia.org/wiki/MVC, 2020. [Online; accessed 21-Setembro-2020].

[Wik20d] Wikipédia. Php. https://pt.wikipedia.org/wiki/PHP, 2020. [Online; accessed 21-Julho-2020].

[Wik20e] Wikipédia. Service-oriented architecture — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.php?title=Service-oriented_architecture&oldid=56319603, 2020. [Online; accessed 18-Julho-2020].

[Wik20f] Wikipédia. Spring framework — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.php?title=Spring_Framework&oldid=49759404, 2020. [Online; accessed 09-Junho-2020].

[Zip20] Zipkin. Zipkin. https://zipkin.io/, 2020. [Online; accessed 11-Junho-2020].