



ENGENHARIA DA COMPUTAÇÃO

FELIPE CARLOS RIBEIRO CARDOZO

DESENVOLVIMENTO DE UMA REDE SOCIAL UTILIZANDO A ARQUITETURA DE MICROSERVIÇOS

MACAÉ

2019



FELIPE CARLOS RIBEIRO CARDOZO

**DESENVOLVIMENTO DE UMA REDE SOCIAL UTILIZANDO A ARQUITETURA
DE MICROSERVIÇOS**

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Engenharia de Computação, da Faculdade Salesiana Maria Auxiliadora, como requisito parcial à obtenção do grau de Bacharel em Engenharia de Computação.
Orientador: Prof. D.Sc. Alan Carvalho Galante

MACAÉ

2019

FOLHA DE APROVAÇÃO

FELIPE CARLOS RIBEIRO CARDOZO

DESENVOLVIMENTO DE UMA REDE SOCIAL UTILIZANDO A ARQUITETURA DE MICROSERVIÇOS

Trabalho de conclusão de curso aprovado como requisito parcial para obtenção do grau de Engenharia Da Computação da FSMA – Faculdade Salesiana Maria Auxiliadora, pela seguinte comissão examinadora:

Prof. Alan Carvalho Galante

Prof. Rogerio de Jesus da Silva

Prof. Mauro Dias de Carvalho

MACAÉ

2019

AGRADECIMENTO

Inicialmente gostaria de agradecer ao meus queridos pais Claudia Maria e Francisco Carlos, e minha querida irmã Ana Claudia, que me proporcionaram uma excelente educação, sempre apoiaram meus estudos e minha paixão por tecnologia.

A minha querida namorada Carrollina Moraes, uma grande incentivadora nessa jornada, pois durante todo o curso estava ao meu lado me apoiando. Obrigado por ser tão atenciosa, carinhosa e por entender minha ausência em alguns momentos.

Aos meus queridos familiares, mas principalmente meus tios Cristina e Fachico que ajudaram a custear minha faculdade, e seus filhos Aparecida e Alisson que sempre foram grandes inspirações para mim.

Ao meu Orientador Alan Carvalho, que me ajudou a possibilitar esse trabalho e foi responsável por me influenciar na área de desenvolvimento ao introduzir ótimas aulas de desenvolvimento de software.

Também sou grato à empresa Sapiensia, que me concedeu a chance de fazer estágio supervisionado e assim iniciar minha carreira na área de desenvolvimento de software, obrigado por confiarem nos conhecimentos que adquiri durante minha faculdade.

RESUMO

Corporações utilizam aplicações para resolver problemas e trazer conforto para seus usuários, essas aplicações muito das vezes são a fonte primaria de uma empresa, logo é necessário que tais aplicações possuam o maior desempenho possível, facilidade de manutenção e que seja tecnologicamente heterogênea para o maior filtro de profissionais capacitados para tais tarefas. Esse projeto tem o objetivo de implantar uma arquitetura de microsserviços mostrando soluções para os problemas precitados. Tal projeto foi desenvolvido com diversas ferramentas e linguagens de programação, como C#, TypeScript no lado do servidor com NodeJS, Angular e tipos de banco de dados diferentes, como o SQL Server e o MongoDB.

Palavras Chaves: Arquitetura, Microsserviços, C#, NodeJS, Ionic.

ABSTRACT

Corporations use applications to solve problems and bring comfort to their users, these applications are often the primary source of a company, with the level of importance, it is necessary that such applications have the highest possible performance, ease of maintenance and that is technologically heterogeneous to the largest filter of professionals trained for such tasks. This project has the objective of implanting a microservices architecture showing solutions to the above problems. Such a project was developed with several tools and programming languages, such as C#, JavaScript on the server side with NodeJS, Angular and different database types such as SQL Server and MongoDB.

Keywords: Architecture, Microservices, C#, NodeJS, Ionic.

LISTA DE ILUSTRAÇÕES

Figura 1 – Modelo de Arquitetura monolítica.....	15
Figura 2 – Arquitetura de três camadas.....	16
Figura 3 – Modelo de Arquitetura com Microserviços	17
Figura 4 – Modelo de Fluxo do JWT.....	24
Figura 5 – RPC no RabbitMQ	26
Figura 6 – Diagrama de classes do Identity Service.....	30
Figura 7 – Diagrama de classes do Feed Service	31
Figura 8 – Diagrama de classes do Support Service.....	32
Figura 9 – Diagrama de casos de uso do Flue	33
Figura 10 - Ecossistema da rede social Flue	35
Figura 11 – Gerenciador do RabbitMQ	36
Figura 12 – Controller do IdentityService.....	37
Figura 13 – Método de criação de usuário	38
Figura 14 – Método de autenticação de usuário	39
Figura 15 – Serviço ouvinte do RabbitMQ	40
Figura 16 – Manipulador do Comando de Criar Pessoa	40
Figura 17 – Controller do SupportService	41
Figura 18 – Caso de uso do Registrar	42
Figura 19 – Página de Registro.....	43
Figura 20 – Caos de Uso Login	44
Figura 21 – Página de Login	44
Figura 22 – Caso de Uso Seguir Pessoa.....	45
Figura 23 – Página de Pesquisa	45
Figura 24 – Caso de Uso Manter Post e Manter Comentários	46
Figura 25 – Página Novo Post	47
Figura 26 – Página Feed.....	47
Figura 27 – Caso de Uso Editar Perfil	48
Figura 28 – Página Editar Contar	48
Figura 29 – Caso de Uso Enviar Mensagem para o Suporte.....	49
Figura 30 – Página de Suporte.....	49

LISTA DE ABREVIATURAS E SIGLAS

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CPU	Graphics Processing Unit
DB	Database
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
JSON	JavaScript Object Notation
JWT	JSON Web Token
REST	Representational State Transfer
RPC	Remote Procedure Call
SPA	Single Page Application
SQL	Structured Query Language
UML	Unified Modeling Language

SUMÁRIO

1. INTRODUÇÃO.....	10
1.1 PROBLEMA	10
1.2 HIPÓTESES	11
1.3 OBJETIVOS	11
1.3.1 Objetivo Geral	11
1.3.2 Objetivos Específicos.....	11
1.4 JUSTIFICATIVAS	11
2. METODOLOGIA	13
2.1 PERSPECTIVAS DO ESTUDO	13
2.1.1 Tipo de Pesquisa.....	13
2.1.2 Fonte de Dados	13
2.1.3 Instrumento de Coleta de Dados.....	13
2.1.4 LIMITAÇÃO DO ESTUDO.....	13
3. FUNDAMENTAÇÃO TEÓRICA.....	15
3.1 ARQUITETURA MONOLÍTICA	15
3.1.1 Arquitetura de três camadas	16
3.2 ARQUITETURA DE MICROSERVIÇOS.....	16
3.3 HARDWARE E INFRAESTRUTURA.....	17
3.3.1 Comunicação	18
3.3.2 Vantagens	19
3.3.3 Desvantagens.....	19
3.4 TÉCNICAS DE ENGENHARIA DE SOFTWARE PARA MICROSERVIÇOS	
20	
3.4.1 Elo de Microserviços a Orientação Objeto e Técnicas SOLID	20
3.4.2 Manutenibilidade	22
3.4.3 FERRAMENTAS DE TRABALHO	22
4. MINIMUNDO - REDE SOCIAL FLUE	27

4.1	INFRAESTRUTURA	27
4.2	NEGOCIO.....	27
4.3	ESTILIZAÇÃO.....	28
5.	MODELAGEM DO NEGOCIO	30
5.1	DIAGRAMA DE CLASSES.....	30
5.1.1	Identity Service.....	30
5.1.2	Feed Service	31
5.1.3	Support Service	32
5.2	DIAGRAMA DE CASOS DE USO.....	32
6.	IMPLEMENTAÇÃO.....	34
6.1	AMBIENTE E TÉCNICAS DE DESENVOLVIMENTO	34
6.2	MICROSSERVIÇOS EM AÇÃO	34
6.2.1	SERVIÇO DE BARRAMENTO RABBITMQ	35
6.2.2	IDENTITY SERVICE	36
6.2.3	FEED SERVICE.....	39
6.2.4	SUPPORT SERVICE	41
7.	VALIDAÇÕES.....	42
7.1	CASO DE USO REGISTRAR.....	42
7.2	CASO DE USO LOGIN.....	44
7.3	CASO DE USO SEGUIR PESSOA.....	45
7.4	CASO DE USO MANTER POST	46
7.5	CASO DE USO EDITAR PERFIL	48
7.6	CASO DE USO ENVIAR MENSAGEM PARA O SUPORTE	49
8.	CONCLUSÃO.....	50
8.1	TRABALHOS FUTUROS.....	50
	REFERÊNCIAS BIBLIOGRÁFICAS	52

1. INTRODUÇÃO

Segundo Blockhead (2017), desde décadas passadas, desenvolvedores estudam novos métodos de desenvolvimento de sistemas, utilizando novas tecnologias ou arquiteturas para que tornem os clientes mais felizes com o resultado de suas aplicações.

A utilização da arquitetura de microsserviços vem sendo adotada por muitas empresas nesse processo de produzir sistemas mais performáticos, disponíveis para o usuário, e permitir a manutenibilidade de grandes sistemas com muitos desenvolvedores, o uso de grandes empresas como Netflix, Uber, Facebook, Google, Microsoft ajudou a popularizar a arquitetura e provar seu desempenho pelo sucesso de seus casos de uso.

1.1 PROBLEMA

Segundo Santos (2017), a arquitetura monolítica é predominante em sistemas por ter a facilidade de ser desenvolvida e por não existir nenhuma alternativa desta arquitetura até o começo dessa década, onde nasce a ideia de arquitetura de microsserviços.

Utilizando arquiteturas monolíticas, Santos (2017) menciona alguns dos problemas que são comuns em arquiteturas monolíticas comparadas com arquiteturas de microsserviços:

- Arquiteturas monolíticas são custosas para escalar horizontalmente, um sistema em uma data específica pode ter mais requisições que o normal, então para uma aplicação com arquitetura monolítica a escalabilidade teria que ser feita em toda aplicação e não somente onde está sendo a sobrecarga de requisições.
- Monólitos possuem códigos englobados, pode ser trabalhoso para um novo desenvolvedor de um time se acostumar com uma aplicação com dezenas ou centenas de classes, isso se aplica bastante ao refatorar códigos de uma aplicação ou amadurecimento de testes automatizados.
- O processo que faz alterações entrarem em produção é demorado em grandes aplicações, que podem conter muitos testes unitários, de integração ou de aceitação fazendo que demande um enorme espaço de tempo para uma alteração do código chegar em produção para o usuário final.
- Arquiteturas monolíticas limitam escopos de tecnologia, por utilizar aplicações monolíticas pode ser mais difícil de desenvolver com mais de uma tecnologia. Tecnologias são mais performáticas ou mais confortáveis dependendo de cada caso de uso.

Com a arquitetura de microsserviços é possível resolver esses problemas?

1.2 HIPÓTESES

- Existem cenários para utilizar a arquitetura de microsserviços.
- Ao usar microsserviços é possível uma pluralidade maior de tecnologias.
- Microsserviços são independentes.
- Microsserviços tendem ser mais disponíveis.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Demonstrar as vantagens e desvantagens ao utilizar a arquitetura de Microsserviços sobre arquiteturas monolíticas, e desenvolver uma rede social para demonstrar o ecossistema de uma arquitetura de microsserviços, nesse desenvolvimento vai ser utilizado boas práticas de engenharia de software.

1.3.2 Objetivos Específicos

- Realizar estudo sobre a utilização de Microsserviços e mostrar as vantagens e desvantagens de se usar Microsserviços em relação a uma aplicação monolítica.
- Desenvolver uma aplicação utilizando a arquitetura de Microsserviços.
- Criar uma rede social, e disponibilizá-lo como um software aberto chamado de **Flue**.
- Utilizar tecnologias diferentes e mostrar que é possível com Microsserviços.
- Promover tecnologias, técnicas, ferramentas e bibliotecas de *software* atuais para o desenvolvimento do *software*.

1.4 JUSTIFICATIVAS

Segundo Susan Fowler (2017), o setor de tecnologia teve uma rápida alteração nas arquiteturas e práticas de sistemas distribuídos, o que fez grandes empresas (Netflix, Twitter, Amazon, eBay, etc..) abandonar a construção de aplicações monolíticas e adotar a arquitetura baseada em microsserviços, embora conceitos de arquitetura de microsserviços não sejam novos, o uso em massa de microsserviços é atual, e sua adoção é motivada por

dificuldades com estabilidade, falta de eficiência, desenvolvimento lento, e dificuldade de adicionar novas tecnologias a aplicações monolíticas.

2. METODOLOGIA

2.1 PERSPECTIVAS DO ESTUDO

A seguir serão destacadas noções da metodologia deste trabalho de conclusão de curso em andamento.

2.1.1 Tipo de Pesquisa

Este trabalho de conclusão de curso é uma pesquisa explicativa com estudo de caso. Verifica-se o que colaboradores da POSGRADUANDO (2012), diz a respeito da pesquisa explicativa: “A pesquisa explicativa exige maior investimento em síntese, teorização e reflexão a partir do objeto de estudo. Visa identificar os fatores que contribuem para a ocorrência dos fenômenos ou variáveis que afetam o processo”.

2.1.2 Fonte de Dados

As fontes de dados desse trabalho de conclusão foram retiradas ou elaboradas em base de referências bibliográficas de livros, artigos, palestras e vídeos.

2.1.3 Instrumento de Coleta de Dados

O instrumento de coleta de dados inicial é a observação em problemas comuns de empresas com sistemas monolíticos e a limitação de uso de tecnologias homogenias.

2.1.4 LIMITAÇÃO DO ESTUDO

A aplicação do estado da arte de microsserviço é de grande adoção em empresas de alcance global e tais empresas adotaram essa arquitetura por necessitar dos benefícios que microsserviço traz, nesse trabalho de conclusão que vai utilizar uma arquitetura tão complexa terá limitações comparado a quem está aplicando essa metodologia, demonstrar os benefícios que as empresas querem ao adotar microsserviço pode ser limitante por causa do tamanho de um trabalho de conclusão.

2.1.4.1 Poucos Microsserviços

É muito discutido que uma arquitetura com poucos serviços não pode ser chamada de microsserviço, já que os benefícios serão pequenos comparados aos ecossistemas que contenham centenas de microsserviços. Também é muito discutido se serviços no mesmo

repositório fazem sentido e se são microsserviços, E como isso se trata de uma demonstração em um trabalho de conclusão não será possível o desenvolvimento de muitos serviços.

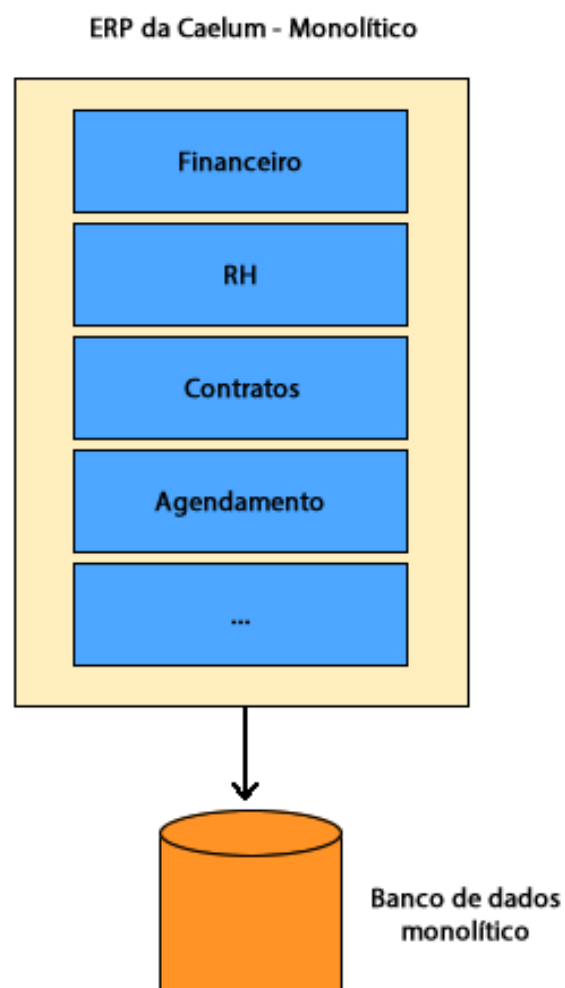
3. FUNDAMENTAÇÃO TEÓRICA

3.1 ARQUITETURA MONOLÍTICA

Segundo Caelum (2015), as arquiteturas monolíticas são usadas em aplicações com várias camadas de contexto, gerando um único programa. Essa arquitetura é a mais utilizada em *softwares*.

A Figura 1 mostra como é desenvolvido uma arquitetura monolítica:

Figura 1 – Modelo de Arquitetura monolítica



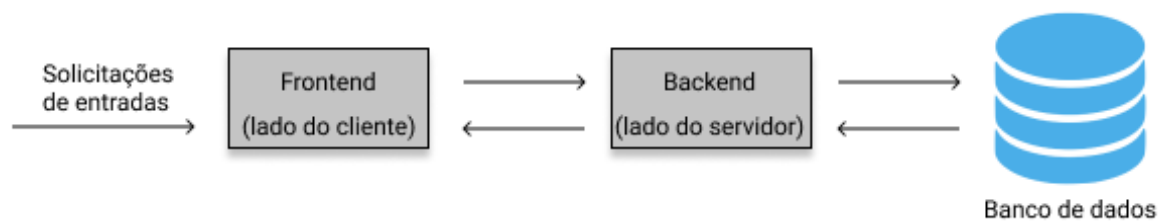
Fonte: (CAELUM, 2015)

Como é exemplificado na Figura 1, o retângulo com blocos dentro representa os serviços, logo se algum desses blocos falhar acontecerá uma inconsistência no sistema inteiro.

3.1.1 Arquitetura de três camadas

Segundo Susan Fowler (2017), aplicações de software costumam ser partidos em três elementos distintos, o lado do cliente (*frontend*), do servidor (*backend*) e o banco de dados. As requisições são solicitadas no lado do cliente, o lado do servidor faz a parte pesada com transparência ao usuário e se necessário o lado do servidor persiste dados no banco de dados, a seguir uma representação dessa arquitetura:

Figura 2 – Arquitetura de três camadas



Fonte: (Elaborada pelo autor)

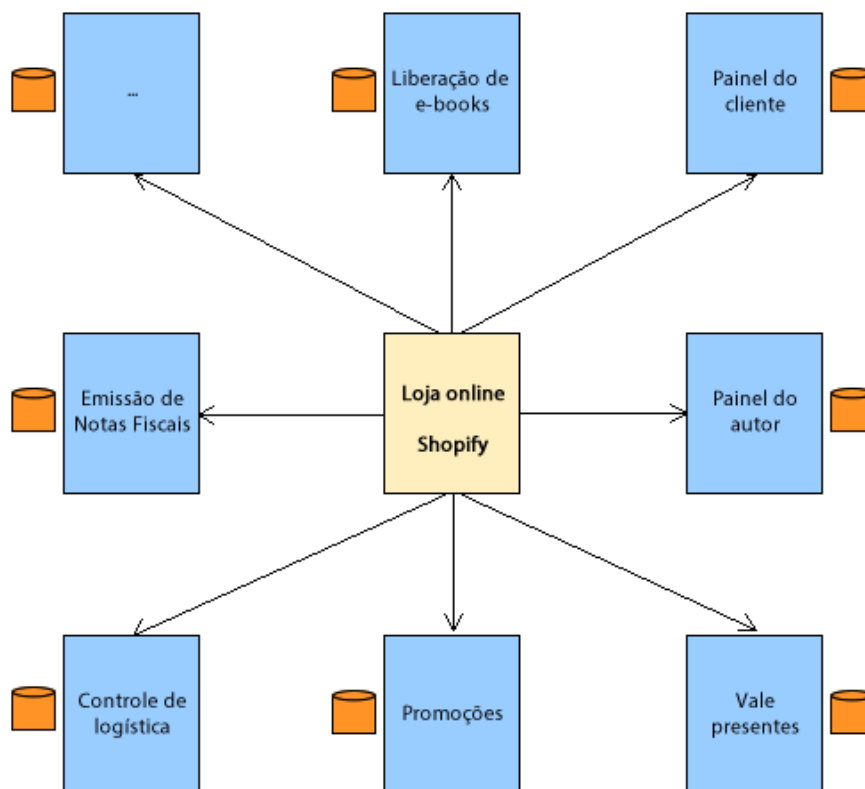
3.2 ARQUITETURA DE MICROSERVIÇOS

Segundo Baltieri (2017), microserviço é somente um serviço pequeno e na maioria das vezes com um único contexto. Uma aplicação com a arquitetura monolítica pode ser um microserviço se essa aplicação tiver um escopo pequeno, mas não é esse o foco desse trabalho de conclusão de curso, e sim o microserviços como arquitetura.

Segundo Blockhead (2017), a arquitetura de microserviços é definida por pequenos serviços que são autônomos e trabalham em conjunto. Arquitetura de microserviços surgiu para solucionar os problemas que aplicações monolíticas causavam. Segundo Fowler (2014), diferente das aplicações de arquitetura monolíticas, na arquitetura de microserviços é “dividida” as camadas de aplicações em uma suíte de pequenos serviços, assim não sendo um ponto único de problema se acontecer alguma falha. As arquiteturas de microserviços são construídas em volta da linguagem de negócio do sistema.

A Figura 3, mostra como é o desenvolvimento de uma arquitetura de microserviços:

Figura 3 – Modelo de Arquitetura com Microserviços



Fonte: (CAELUM, 2015)

Na Figura 3 é exemplificado o funcionamento de uma arquitetura de microserviços. Diferente da Figura 1 que exemplifica a arquitetura monolítica, nessa figura é demonstrado os serviços separados, e com seus próprios bancos de dados, representado na imagem por um cilindro laranja.

Para aplicar uma arquitetura de microserviços com sucesso é necessário entender diversos conceitos tecnológicos, uma arquitetura de microserviços não possui só microserviços, essa arquitetura precisa da sustentação de outras áreas da tecnologia, é de suma importância a aplicação correta de infraestrutura, hardware, software, rede e conceitos sólidos de programação para o estado da arte de um ecossistema da microserviços.

3.3 HARDWARE E INFRAESTRUTURA

Hardware é a primeira camada de um ecossistema de microserviços, em microserviços trata-se de servidores e seus facilitadores, para Susan Fowler (2017), existem muitos tipos de servidores que podem ser aplicados, existem servidores que são otimizados para banco de dados, outros para processar tarefas que exigem muita CPU, cada servidor precisa de um

sistema operacional instalado, não existe um sistema operacional perfeito para microsserviços, tudo dependerá do que suas aplicações necessitam, mas a maioria dos ecossistemas utilizam distribuições Linux pelos motivos de ser gratuito, código aberto e otimizado para servidores.

Segundo Susan Fowler (2017), cada servidor deve ser fornecido, configurado, e depois deve ser necessário instalar o sistema operacional e uma ferramenta de gerenciamento de configuração para instalar as aplicações e configura-las, também é necessário um pleno monitoramento no nível do servidor do ecossistema, para que se aconteça problemas físicos (falha no disco, falha na rede ou se o CPU está no limite de processamento), os problemas possam ser facilmente diagnosticados e tratados.

3.3.1 Comunicação

A camada de comunicação de uma arquitetura de microsserviços se espalha por todas as outras camadas, porque é nessa camada que toda a comunicação entre os serviços é feita. Segundo Susan Fowler (2017), a maioria dos microsserviços interagem uns com outros através da rede utilizando três conceitos, sejam eles comunicações por RPC, trocando mensagens por *endpoints* de uma API ou trocando mensagens para um *Message Broker* que vai rotear a mensagem para onde ela foi destinada.

De todos paradigmas de comunicação, trocar mensagens por HTTP+REST em *endpoints* de API, é o principal e mais comum caso de uso, por utilizar conceitos já adotados no desenvolvimento web, tornando mais fácil e estável e confiável para profissionais já acostumados, porém utilizar comunicação interna por HTTP+REST limita nossa arquitetura em operações síncronas (bloqueante), logo é perdido toda disponibilidade que o paradigma de operações assíncronas propõe.

O paradigma de comunicação de trocas de mensagem por um *Message Broker* é assíncrono (não bloqueante), mas sua implementação é mais complexa, a troca de mensagens pode ser exemplificado por um microsserviço enviando dados (mensagem) pela rede para um *broker* de mensagem, que irá rotear essa mensagem para outro microsserviço, mas utilizar somente *Message Broker* apresenta desvantagens, *brokers* centralizam todas as comunicações em filas, o que pode ser um problema se não tiver um escalonamento de qualidade.

Segundo Angeline e Lima (2017), diferente do paradigma de HTTP+REST uma API RPC é a principal maneira de compartilhar métodos e funções por chamadas remotas, com transparência ao usuário. RPC são extremamente performáticos por suportar formatos binários e

outros protocolos além do HTTP, logo em muitos casos de empresas que possuem o estado da arte de microsserviços utilizam RPC como comunicação principal.

3.3.2 Vantagens

A arquitetura de microsserviços tem diversas vantagens ao se comparar com arquiteturas monolíticas, segundo Santos (2017), essas vantagens têm ligação ao fato de se dividir os deveres de um *software*, se divide também as chances de um serviço está com problemas, claro, isso se aplica a microsserviços bem codificados e planejados, aliás código ruim é código ruim em qualquer lugar. Trabalhar com um código menor tende de ser mais fácil, por ser mais rápido de entender a complexidade, ajuda e incentiva desenvolvedores a melhorar esses códigos.

Segundo Moreira e Beder (2015), Heterogeneidade Tecnológica é um grande motivo para a defesa de arquiteturas de microsserviços, já que por ser uma estrutura de múltiplos serviços, traz a possibilidade de escrever esses serviços em tecnologias diferentes, utilizar bancos de dados diferentes e de diversificar ferramentas. Caso a troca de tecnologia não traga benefícios será mais fácil de reescreve-la.

Uma aplicação menor leva menos tempo para ser compilada ou interpretada, logo seu *Deploy* é rápido, Segundo Duarte (2017) a arquitetura monolítica pode frustrar desenvolvedores que implantam projetos em plataformas de serviços de nuvem, pois esses desenvolvedores produzem muito em pouco tempo, mas todo seu trabalho demora para chegar ao cliente, já que uma arquitetura monolítica também possui um *deploy* e bateria de testes monolítica.

3.3.3 Desvantagens

Entretanto há motivos para pesar no outro lado da balança, a utilização dessa arquitetura pode ocasionar problemas em times com pouca experiência e trazer complexidade onde não precisa.

Dependência de documentação. Essa arquitetura precisa de mais documentação que arquiteturas monolíticas, já que o desenvolvedor pode ter dificuldades de encontrar o que deseja, por ser mais fácil encontrar dependência de código direto do que dependências de rotas.

Santos descreve a desvantagem de desenvolver um projeto com arquitetura de microsserviços logo de início:

“Em um início de projeto, os problemas resolvidos por esta arquitetura são inexistentes. Portanto um dos grandes desafios é saber em qual parte do ciclo do projeto ela deve ser

implementada, isto demanda mais tempo de desenvolvimento” (SANTOS, 2017).

Desenvolver tal arquitetura envolve cautelas, como, não compartilhar dependências diretamente, o que pode frustrar desenvolvedores que não se adaptam a essas prevenções.

3.4 TÉCNICAS DE ENGENHARIA DE SOFTWARE PARA MICROSERVIÇOS

Conceitos e técnicas de engenharia de software são fundamentais para a arquitetura e levantamentos de sistemas, segundo Sommerville (2011) a engenharia de software tem o objetivo de apoiar o profissional de software com técnicas que apoiam especificação, projeto e evolução de programas, um software que engenheiros e arquitetos iniciaram deve ser um projeto manutenível para que outros engenheiros façam alterações e manutenções no código do projeto, olhar para um projeto a longo prazo sempre é inteligente.

3.4.1 Elo de Microsserviços a Orientação Objeto e Técnicas SOLID

Para Aniche (2015), pensar em um sistema com o paradigma de orientação a objetos é, mais do que pensar em código, é organizar e concluir como partes de um sistema serão encaixados para se tornar um só, alterando uma dessas partes mudarão as outras partes ao redor, tal partes podem ser classes.

Técnicas para deixar o código de um software mais manutenível, pode ser relacionado a técnicas de SOLID, segundo Aragão (2017) SOLID é um acrônimo que representa 5 princípios da programação orientada objeto que foram alavancados por Robert Cecil Martin ou também conhecido por Uncle Bob no início do segundo milênio, relatado por ele, são os princípios de Responsabilidade única (S), aberto e fechado (O), substituição de Liskov (L), segregação de interfaces (I), inversão de dependência (D).

Segundo Junior (2018), um microsserviço é uma aplicação que responde a uma única e exclusiva função, pautado no clássico princípio da responsabilidade única do SOLID. Conceitos de SOLID podem ser aplicados além da orientação objeto.

3.4.1.1 *Princípio da Responsabilidade Única*

Segundo Aniche (2015), o princípio de responsabilidade única é sobre a utilização de coesão, em orientação objetos uma classe coesa é uma classe com responsabilidade única, assim

não se responsabilizando por mais conceitos do sistema, se uma classe representa um conceito, ela deve representar somente o que ela foi nomeada.

Também é necessário a implementação de encapsulamento, que traz o conceito de ao mudar um único ponto, essas alterações sejam propagadas naturalmente ao redor do sistema.

3.4.1.2 Princípio do Aberto/Fechado

Segundo Aniche (2015), o princípio do aberto/fechado é utilizado para fazer com que novas regras sejam mais simples e essas alterações se espalhe em todo sistema. Esse princípio tem o conceito de que classes devem ser abertas para extensão e fechadas para modificação, abertas para extensão pode significar que estender suas funcionalidades deve ser uma tarefa fácil, mas alterar funcionalidades existentes deve ser uma ação que não ocorra o tempo todo. A solução para isso é deixar de instanciar as implementações concretas dentro da classe específica, e passar a receber essas dependências pelo construtor. Em orientação objetos o código evolui por meio de novos códigos, e não de alterações em códigos já existentes.

Um outro benefício de respeitar o princípio de Aberto/Fechado é a facilidade que ele traz em testes unitários, porque é possível usar *mocks* para suas dependências e testar somente suas funcionalidades.

3.4.1.3 Princípio da substituição de Liskov

Segundo Aniche (2015), casos de classes que necessitam de herança, são dependentes de desenvolvedores cujo tem o dever de pensar nas pré-condições e pós-condições que a classe pai possui. No início das linguagens orientadas a objetos a herança era uma funcionalidade usada para vender, porque ela era vendida como reuso de código erroneamente. É sobre isso que o Princípio de Substituição de Liskov discute, é necessário pensar antes de herdar uma classe, é preciso pensar sobre o contrato estabelecido pela classe pai para que não ocorra problemas desnecessários em outras utilizações dessas classes.

3.4.1.4 Princípio da segregação de interfaces

Segundo Aniche (2015), tudo que foi mencionado sobre coesão de classes deve ser aplicado para interfaces também, interfaces *fats* (gordas ou pesadas) devem ser evitadas, elas trazem pouco reuso ao desenvolvedor, é mais viável partir essas interfaces para que elas possuam uma única responsabilidade, interfaces de responsabilidade única são mais reutilizáveis e são mais estáveis.

3.4.1.5 Princípio da inversão de dependência

Segundo Aniche (2015), interfaces possuem muita importância em sistemas orientados a objetos, logo programar voltado para interfaces faz todo sentido, trazendo os benefícios de flexibilidade e estabilidade, e se uma interface é estável, acoplar a ela não é um problema. O princípio diz que, sempre que uma classe depender de outra, ela deve depender de outro módulo mais estável do que ela mesma. Para isso módulos de alto nível não podem depender de módulos inferiores, e detalhes devem depender de abstrações, não o inverso.

3.4.2 Manutenibilidade

Sistemas mais manuteníveis é um dos maiores objetivos desse trabalho de conclusão de curso, segundo Sommerville manutenção de software pode ser descrito por:

“A manutenção de software é o processo geral de mudança em um sistema depois que ele é liberado para uso. O termo geralmente se aplica ao software customizado em que grupos de desenvolvimento separados estão envolvidos antes e depois da liberação. As alterações feitas no software podem ser simples mudanças para correção de erros de codificação, até mudanças mais extensas para correção de erros de projeto, ou melhorias significativas para corrigir erros de especificação ou acomodar novos requisitos. As mudanças são implementadas por meio da modificação de componentes do sistema existente e, quando necessário, por meio da adição de novos componentes.” (SOMERVILLE, 2011).

Para Castro (2016), A manutenção de um software é responsável pela maioria dos custos usados para manter um software, por isso é uma atividade importante, e seu custo elevado é causado por problemas não previstos antes da implementação do software, logo se esses problemas forem tratados antes, a economia de tempo e de custos será exponencial.

3.4.3 FERRAMENTAS DE TRABALHO

3.4.3.1 DotNet Core e C#

Segundo Colaboradores da Microsoft (2018), *DotNet Core* é uma plataforma de desenvolvimento mantida pela Microsoft e a comunidade de *software* livre do Github. *DotNet Core* é multiplataforma, então oferece suporte a Windows, Linux, MacOS e Containers.

A plataforma *DotNet Core* aceita as linguagens C#, F# e Visual Basic. Nesse trabalho de conclusão será usado a linguagem C#. O *DotNet Core* será usado como uma das plataformas de *backend* para o Trabalho.

3.4.3.2 NodeJS

O *NodeJS* é uma plataforma de interpretação de código, com o foco em usar a linguagem mais usada do mundo, o Javascript, fora do ambiente natural dele de *frontend*, usando-o no *backend*. Pereira descreve o nascimento da plataforma da seguinte forma:

“Esta tecnologia possui um modelo inovador: sua arquitetura é totalmente *non-blocking thread* (não bloqueante). ” (PEREIRA, 2014).

Pereira também afirma que ao se trabalhar com arquivos ou realizar muito I/O com uma aplicação não bloqueante é possível resultar em bastante performance e otimização de consumo de memória, pois desse modo é utilizado o processador ao máximo e de forma eficiente, principalmente em altas cargas de processamento.

NodeJs será usado para provar que é possível utilizar duas tecnologias diferentes de *backend* usando microsserviços.

3.4.3.3 SQL Server

De acordo com a equipe de redação da Impacta, SQL Server é um banco relacional SQL, significando que as informações são manipuladas por campos de tabelas, além de ser um dos SGBD com grande adoção por grandes empresas, tal sistema oferece recursos para facilitar a persistência de dados e garantir sua integridade.

3.4.3.4 MongoDB

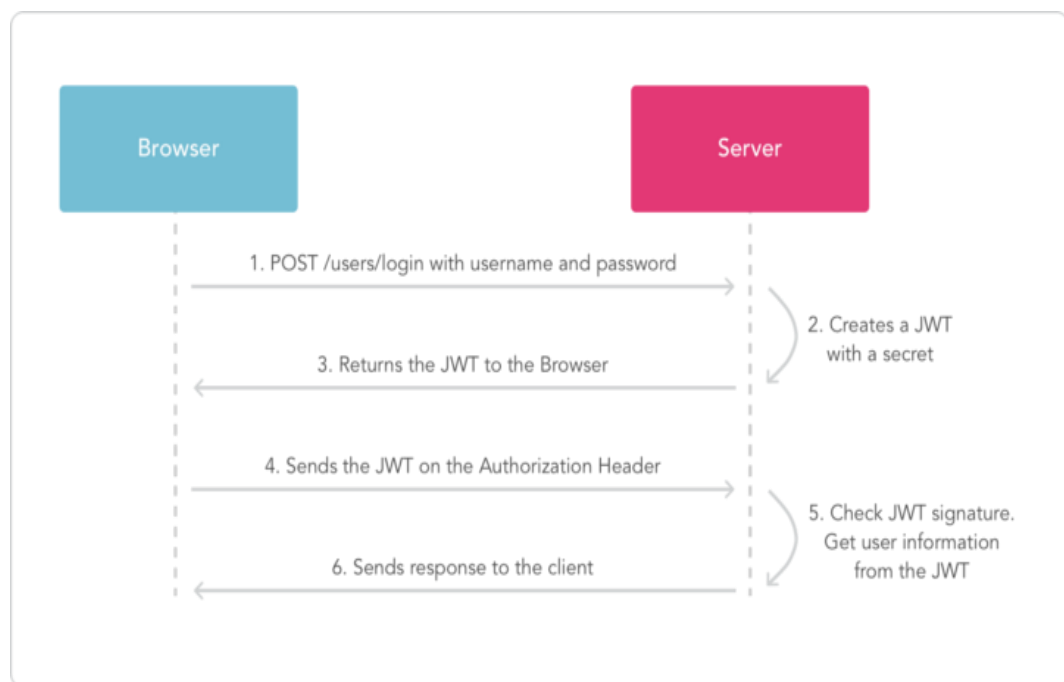
MongoDB é um banco não relacional, diferente de banco de dados SQL. Um banco de dados que foi feito para ser fácil para provas de conceitos e demonstrações, rápido para transações e possui interface gráfica para configurá-lo no navegador. Castilho descreve o banco de dados não relacional assimilando com banco relacionais da seguinte forma:

“Ele é classificado como um banco NoSQL do tipo “Document Database”, o que significa que ele armazena e recupera documentos, que podem estar em formato XML, JSON, entre outros. De uma forma ingênua, podemos dizer que um documento equivale a uma linha de uma tabela em um modelo relacional.” (CASTILHO, 2013).

3.4.3.5 JWT

Como Nascimento (2015) descreve, o JWT (JSON Web Token) é um padrão de mercado onde é definido como se transita e armazena dados por JSON de forma compacta e segura, ele exemplifica o uso de Web Tokens no seguinte diagrama de sequência:

Figura 4 – Modelo de Fluxo do JWT



Fonte: (NASCIMENTO, 2015)

Com o JWT é possível autenticar nas APIs com menos complicação, mas mantendo um nível excelente de segurança.

3.4.3.6 Ionic e Angular

De acordo com Passos (2018), o Ionic é um *framework* para o desenvolvimento de aplicações híbridas para smartphones, com essa ferramenta é possível criar aplicações que pareçam nativas, mas contendo somente uma base de código, que pode gerar aplicativos para IOS, Android, Windows e Web.

O Ionic utiliza o *framework* SPA (Single Page Application) Angular, com esse ferramental será possível desenvolver um aplicativo Android e uma aplicação Web para esse trabalho de conclusão.

3.4.3.6.1 Angular

Angular é um *framework* que facilita a vida do desenvolvedor web, utilizando tecnologias webs convencionais como HTML5 e a linguagem TypeScript da Microsoft, essa linguagem possui o diferencial de possuir tipos, por causa dos tipos, essa linguagem traz mais controle e conforto aos desenvolvedores ao refatorar seus códigos, e por causa disso ajuda a ter um desenvolvimento com menos falhas.

O Angular possui a tecnologia *two-way data binding*, que segundo Almeida (2016), o *two-way data binding* permite alterações da visualização ser refletidas na fonte de dados e vice-versa sem a necessidade de manipulação explícitas.

Esse *framework* traz simplificação na configuração de rotas da aplicação, criação de componentes reutilizáveis, estilização do conteúdo e estruturação de código. E por possuir uma comunidade grande e ativa, ajuda bastante o desenvolvimento ao encontrar problemas comuns.

3.4.3.7 RabbitMQ e RPC

De acordo com Adriano (2018), o RabbitMQ é um serviço de mensageria que utiliza o protocolo de AMQP (*Advanced Message Queuing Protocol*) como comunicação de dados, esse tipo de protelo é amplamente utilizado para o uso de mensagens assíncronas.

Utilizando o RabbitMQ é possível implementa-lo com diversas tecnologias de comunicação entre processos diferentes, alguns exemplos dessas tecnologias são *Workers*, *Pub/Sub* e *RPC* (*Remote Procedure Call*).

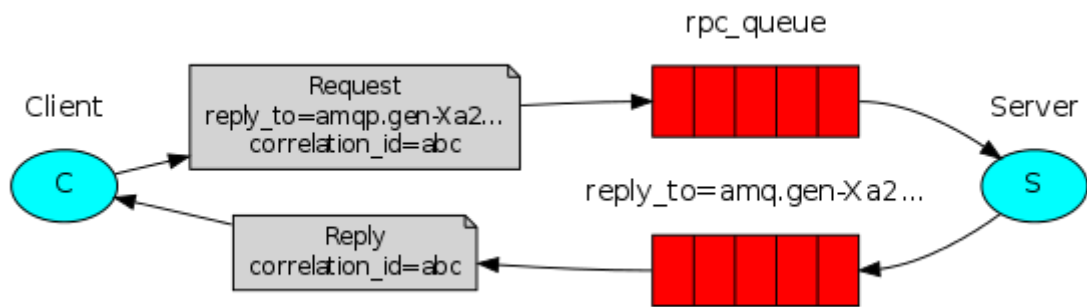
Esse trabalho de conclusão será implementado com RabbitMQ com *RPC*, segundo Adriano sobre o modelo de *RPC*:

“Esse modelo é utilizado na comunicação entre aplicações desenvolvidas em diferentes tecnologias. Imagine o seguinte cenário: Um app desenvolvido em Node.js precisa fazer requisições em uma API .NET Core, mas mesmo que o server

caia, essas requisições não podem ser perdidas.” (ADRIANO, 2018).

Utilizar RabbitMQ com RPC permite o tipo de comunicação assíncrona exemplificada a baixo:

Figura 5 – RPC no RabbitMQ



Fonte: (DOCUMENTAÇÃO OFICIAL DO RABBITMQ)

Para Adriano (2018), o fluxo básico do uso de RPC com RabbitMQ retirado da documentação oficial do RabbitMQ, também pode ser usado para exemplificar duas aplicações, uma Node.js e outra .NET Core se comunicando, para ele:

“C (client) seria a nossa aplicação Node.js, nós estamos enviando 3 informações para o exchange: `reply_to` (fila que eu irei aguardar), `correlation_id`= minha referência, como eu posso ser localizado e os dados em buffer que eu quero enviar para minha fila.

`Rpc_queue`: essa seria a fila que irá receber os dados do exchange e irá enviar para o server.

Server (seria o projeto .NET Core) irá receber os dados, realizar as regras e devolver para fila que veio no `reply_to`

`Reply_to`: irá retornar os dados processados junto com o `correlation_id`.” (ADRIANO, 2018).

3.4.3.8 Containers e Docker

Abstrações adicionais podem ser construídas acima do hardware, utilizando a tecnologia de containers é possível isolar e abstrair recursos fornecidos pelo hardware, containers foram popularizados pelo uso em massa de Docker. Segundo escritores da redação EVO (2018), containers possuem o benefício da independência entre aplicações e infraestrutura e são definidos como:

"Containers são blocos de espaços divididos pelo Docker em um servidor, possibilitando a implementação de estruturas de

Microserviços que compartilham o mesmo sistema operacional, porém de forma limitada (conforme a demanda por capacidade).” (EVO, 2018).

4. MINIMUNDO - REDE SOCIAL FLUE

4.1 INFRAESTRUTURA

Uma empresa privadas deseja iniciar um grande projeto de criar uma rede social, essa empresa possui alto porte de investimento, e gostaria que o sistema já estivesse pronto para competir com grandes redes sociais suportando altas frequências de usuários, e dispondo o máximo possível de disponibilidade e facilidade de manutenção para os desenvolvedores desde seu início, mesmo que isso signifique que o projeto demore mais para ser iniciado, logo foi decidido que vai ser adotada a arquitetura de microserviço no projeto.

Para atingir mais usuários ativos na rede social foi decidido que deve haver duas aplicações ao cliente, um aplicativo celular focando nos usuários de celular e outra aplicação web para navegadores, pois assim será possível acessar a rede social em demais dispositivos. Para tornar o desenvolvimento dessas aplicações mais simples, foi decidido utilizar o *framework* Ionic, com esse *framework* é possível desenvolver o aplicativo e a aplicação web com o mesmo código, pois ele compila a aplicação para celular e gera o código que é utilizado na web.

É necessário que a arquitetura elaborada possua o mínimo possível de dependência com serviços de nuvem ou de hospedagem, logo deve ser possível migrar de provedores de infraestrutura com o menor esforço possível caso essa empresa consiga algum contrato comercial com algum desses provedores de hospedagem, logo foi optado por utilizar *containers*, neste caso, o Docker, pois utilizando *containers* será possível migrar de provedores de infraestrutura com menos dificuldade, pois *containers* deve funcionar igualmente em qualquer hospedagem.

4.2 NEGOCIO

Na rede social deve existir uma página para cadastrar o usuário, com os dados de nome de usuário (*username*), senha, nome e e-mail. Não podem existir e-mails e nome de usuários iguais, logo deve haver aparecer um erro amigável para esse tipo de exceção. O nome de usuário e senha podem ser utilizados para o usuário se autenticar na aplicação, após se autenticar, o

aplicativo deve redirecionar para tela de Feed, logo deve ser exibido os posts dos usuários seguidos pelo usuário.

Posts serão os conteúdos principais da rede social, eles devem conter somente texto, nome de usuário e seus comentários, ao ser enviado deve estar a mostra para todos usuários interessados (usuários que seguiram o usuário criador do post), esses posts podem receber comentários de outros usuários. Posts e comentários podem não podem ser alterados, mesmo o usuário criador não poderá alterar, mas o usuário criado do post ou comentário poderá deletá-los.

Deve existir uma página de perfil, o usuário poderá ver sua própria página de perfil, essa página deve conter seus dados pessoais, como o nome, usuário, descrição, imagem pessoal do usuário e a quantidade de pessoas seguidas, seguindo e posts criados por esse usuário e deverá existir uma um botão que permita o usuário editar seus dados, somente ele poderá editar os dados exibidos, será possível alterar o nome, e-mail, descrição e a imagem pessoal do usuário. A baixo deve ser exibido um Feed que só exiba os posts criados por esse mesmo usuário. A página de perfil poderá ser acessada também ao clicar na imagem de outro usuário, mas quando o acesso da página de perfil é feito de um usuário para o outro usuário, a página terá seu funcionamento limitado para somente leitura, bloqueando as alterações de outros usuários.

A rede social deve possuir uma página de pesquisa, essa página vai permitir ao usuário pesquisar outros usuários da rede social, seguir outros usuários e nessa página deve ser possível identificar se o usuário pesquisado já foi seguido, e se for, permitir que com um clique pare de seguir esse usuário. A pesquisa deve buscar por nome e nome de usuário (*username*), a pesquisa não deve ser *case-sensitive*, isso é, não deverá diferenciar letras maiúsculas de letras minúsculas.

A rede deve conter um meio de comunicação de suporte, para envio de problemas do sistema, essas mensagens enviadas para o suporte devem conter somente texto, ao enviar a mensagem deve aparecer uma notificação ao usuário se a mensagem foi enviada com sucesso para o suporte.

4.3 ESTILIZAÇÃO

A aplicação deve possuir a função de *Dark Mode*, isso é um tema escuro para usuários que preferem fundos escuros em seus aplicativos em vez de fundos claros, tal função já é presente em diversas redes sociais, então será uma exigência de clientes necessária na Flue também, inicialmente essa função deve respeitar as configurações de preferencias de temas do

dispositivo do usuário, se a opção de *Dark Mode* estiver habilitada para o dispositivo do usuário deve ser entendido que essa é sua preferência no primeiro acesso ao aplicativo, caso ele mudar a configuração dentro do aplicativo, deverá ser armazenada essa preferência e respeitá-la nos próximos acessos.

5. MODELAGEM DO NEGOCIO

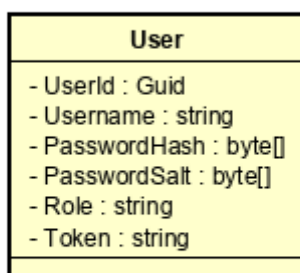
Com base do minimundo do capítulo 4, segue os diagramas de UML de cada Serviço.

5.1 DIAGRAMA DE CLASSES

De acordo com Macoratti (2010), o diagrama de classes lista todos os conceitos do domínio que serão implementados no sistema e as relações desses conceitos, diagrama de classes são de suma importância por definirem a estrutura do sistema.

5.1.1 Identity Service

Figura 6 – Diagrama de classes do Identity Service

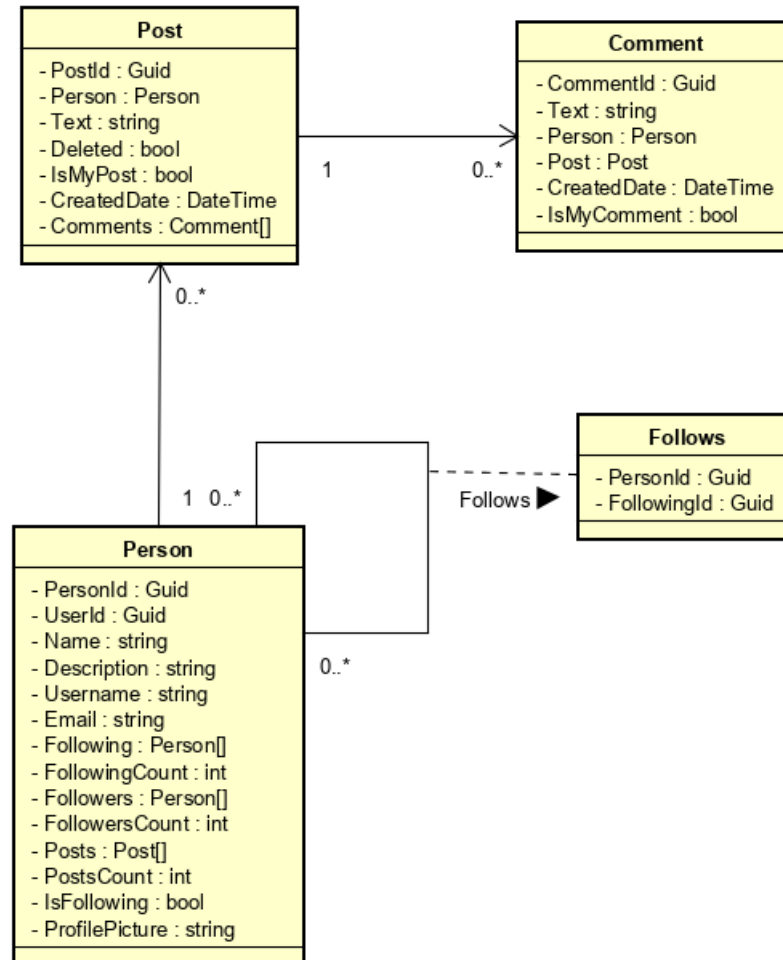


Fonte: (Elaborada pelo autor)

A figura 6 está representado o diagrama de classes do Identity Service, esse serviço só possui uma classe de domínio que é o User, essa classe contém somente propriedades que permitem o usuário a se autorizar no sistema como seu identificador (UserId) , usuário (Username), o hash e salt criptografada de sua senha (PasswordHash e PasswordSalt), sua função é utilizada para autorização (Role) e seu Token utilizado para retornar ao usuário seu JWT.

5.1.2 Feed Service

Figura 7 – Diagrama de classes do Feed Service



Fonte: (Elaborada pelo autor)

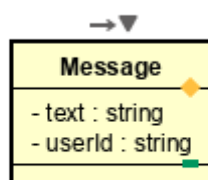
A figura 7 está representado o diagrama de classes do Feed Service, esse é o maior serviço dessa arquitetura, como pode ser observado na figura acima existe uma classe de domínio chamada **Person** que representa a pessoa usuária do Feed Service, essa classe contém identificadores de pessoa e usuário, nome da pessoa, descrição de seu perfil, usuário, e-mail, foto de perfil.

A classe de domínio **Person** também possui um relacionamento de muito para muitos consigo mesmo utilizando uma tabela de relacionamento de **Follows**, esse relacionamento serve para listar pessoas que estão seguindo essa pessoa, pessoas que estão sendo seguidas e suas quantidades.

Person também possui um relacionamento de um para muitos de Post, onde infere que pessoa pode conter muitos posts, Post é classe de domínio que representa as publicações da rede social, essa classe contém propriedades de identificador, texto da publicação, soft deleted e data de criação, além disso Post também possui um relacionamento de um para muitos com Comment, que uma classe de domínio um pouco similar a Post, já que representa os comentários do mesmo, Comment contém propriedades de identificador, texto, e da data de sua criação.

5.1.3 Support Service

Figura 8 – Diagrama de classes do Support Service



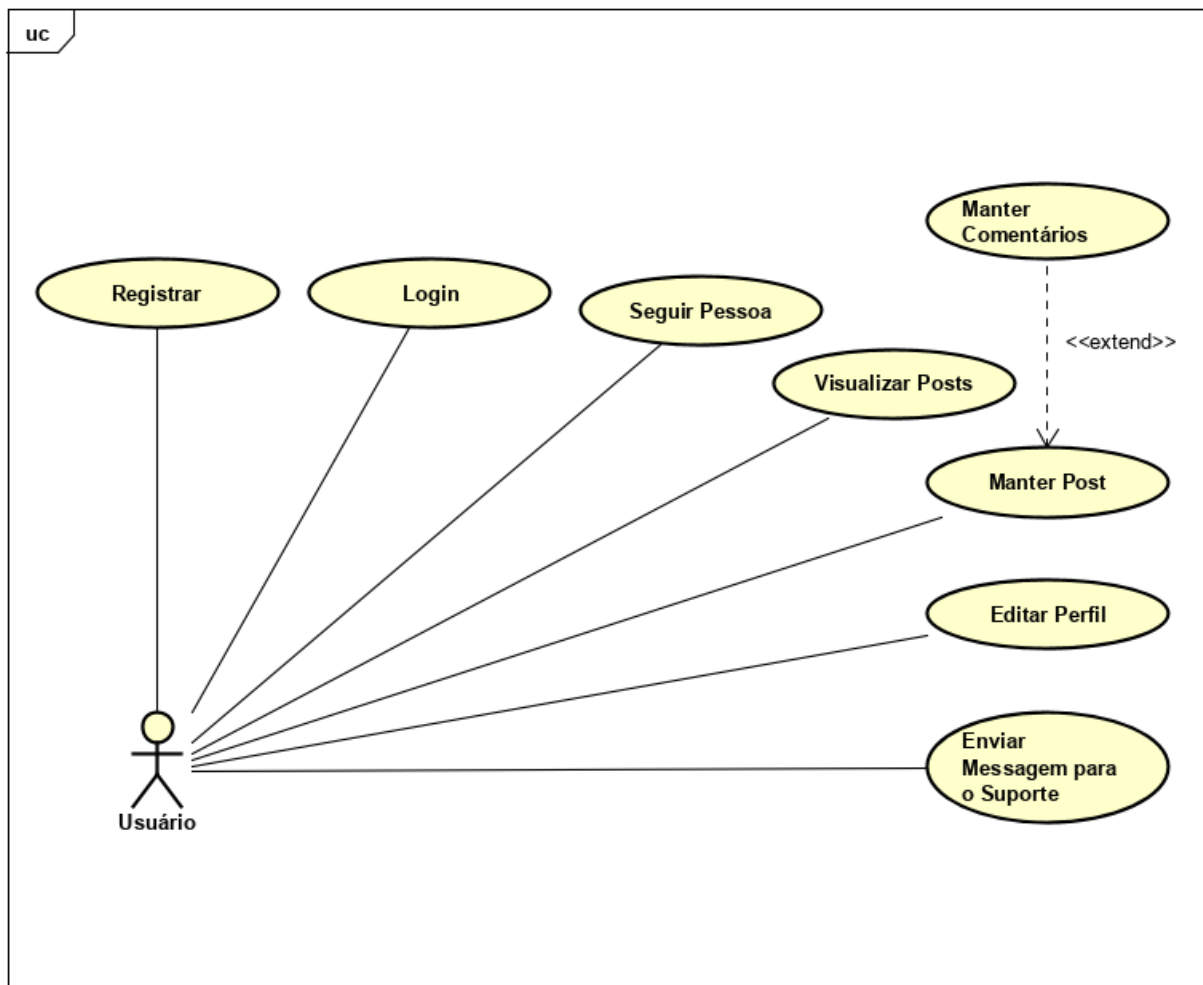
Fonte: (Elaborada pelo autor)

A figura 8 está representado o diagrama de classes do Support Service, o único serviço que utiliza TypeScript como linguagem e MongoDB como banco de dados não relacional, essa classe de domínio é bastante simples, ela tem a responsabilidade de conter somente uma propriedade de texto e o identificador do usuário.

5.2 DIAGRAMA DE CASOS DE USO

De acordo com Vieira (2015) diagramas de caso de uso podem nos auxiliarem no levantamento de requisito funcionais do sistema, apresentar as principais funções do sistema, descrevem interação entre partes do sistema, com foco nos atores. O diagrama de casos de uso utiliza atores, que serão os bonecos que representam os possíveis usuários do sistema e relacionamento desses atores as elipses, cujo estão dentro do cenário de um diagrama de casos de uso.

Figura 9 – Diagrama de casos de uso do Flue



Fonte: (Elaborada pelo autor)

Na figura 9 pode ser observado os casos de uso de toda a rede social Flue, o único ator é o usuário da rede social.

O Usuário acessa primeiramente o caso de uso Registrar para registrar na rede social, após se registrar o usuário pode utilizar o caso de uso do Login em qualquer momento para entrar na rede social, em seguida o Usuário pode utilizar o caso de uso Seguir Pessoa para adicionar seguidores, em seguida o usuário poderá utilizar o caso de uso Visualizar Posts para ver os posts de seus seguidores, o usuário também pode criar ou deletar seus posts de acordo com o caso de uso Manter Post, a figura 9 expressa uma extensão do Manter Comentários para Manter Posts, isso é, o usuário pode criar ou excluir comentários de um post também, além disso o usuário com o caso de uso Editar Perfil, pode alterar informações de seu perfil e por último o usuário pode enviar mensagens para o suporte se necessário ao utilizar o caso de uso Enviar Mensagem para o Suporte.

6. IMPLEMENTAÇÃO

Este trabalho de conclusão de curso apresentara uma aplicação da arquitetura de microsserviço em uma rede social, um tipo de aplicação que tem dificuldades de crescer e disputar funcionalidades e tempo de disponibilidade com grandes empresas, logo será um ótimo exemplo da aplicação da arquitetura de microsserviços logo de início. Dito isso não é recomendado a implementação da arquitetura de microsserviços no início de um sistema simples.

6.1 AMBIENTE E TÉCNICAS DE DESENVOLVIMENTO

É necessário mencionar as técnicas de desenvolvimento que utilizei para realizar esse trabalho, a seguir será mencionado o ambiente de desenvolvimento utilizado para o desenvolvimento do trabalho e qual tipo de ambiente pode ser usado para um resultado similar, também é mencionado o uso das tecnologias e do porquê de usar elas com base na fundamentação teórica, pesquisa.

No desenvolvimento desse trabalho foi usado o sistema operacional da Microsoft, o Windows 10, porem tudo que foi produzido por esse trabalho pode ser desenvolvido em maquinas com outros sistemas operacionais também, como distribuições Linux e o MacOS. O SQL Server ainda não possui suporte nativo ao MacOS mas pode ser usado com containers (Docker) sobre alguma distribuição Linux perfeitamente.

No final do desenvolvimento foi construído as imagens de Docker dos serviços, banco de dados e da SPA (*Single Page Application*), e essas imagens foram levadas para um servidor Ubuntu, para rodar os containers dos componentes.

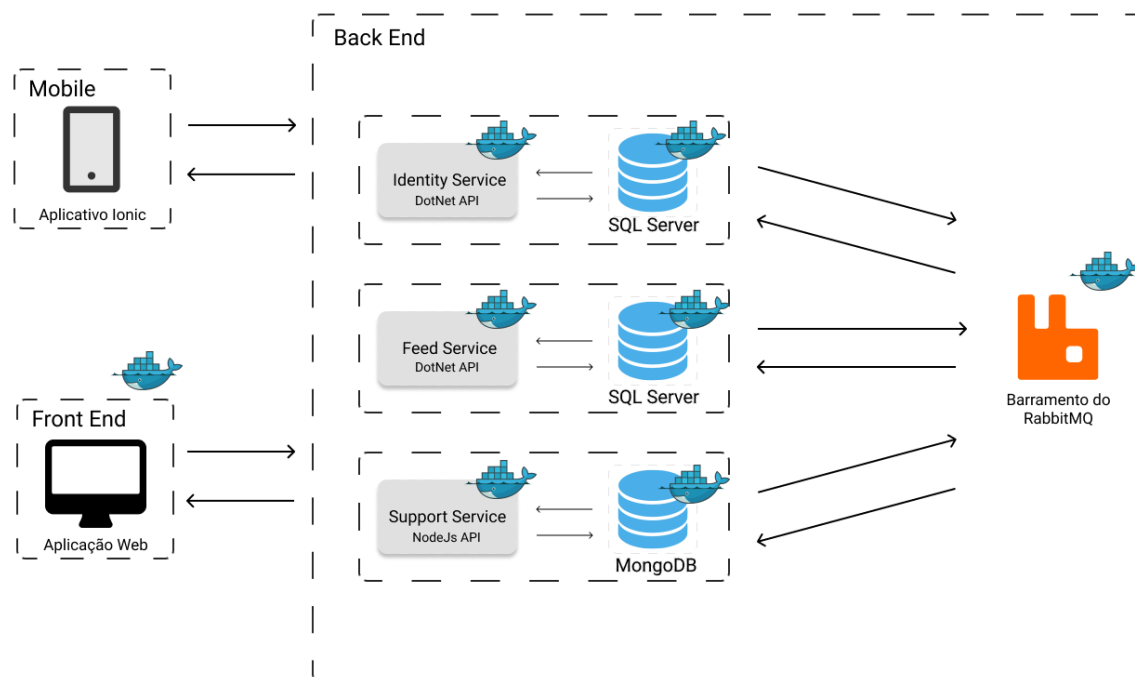
Como mencionando no capítulo 2.4 de Ferramentas de Trabalho, será usado **DotNet Core** e **NodeJS** para montar as aplicações dos serviços, **SQL Server** e **MongoDB** como banco de dados dessas aplicações, **JWT** será o mecanismo de autenticação, e para exibir uma interface gráfica no aplicativo mobile e WEB será usado o *framework* **Ionic** que utiliza também o *framework* **Angular**.

6.2 MICROSERVIÇOS EM AÇÃO

Em seguida será descrito implementações fundamentais para o funcionamento dessa arquitetura como um ecossistema de microsserviços.

Na seguinte imagem da Figura 10, será demonstrada a arquitetura utilizada para essa aplicação.

Figura 10 - Ecossistema da rede social Flue



Fonte: (Elaborada pelo autor)

A Figura 10 demonstra todo funcionamento do ecossistema implementado, são 3 microsserviços, o Identity Service e Feed Service implementados com .NET Core com a linguagem C#, utilizando banco de dados SQL Server e o Support Service, um serviço que foi implementado em NodeJS com a linguagem TypeScript, utilizando um banco de dados NoSQL, o MongoDB.

O RabbitMQ é o serviço de mensageria utilizado para os microsserviços se comunicarem pelo serviço protocolo RPC que o RabbitMQ prover.

A imagem também demonstra o lado do cliente, onde é possível acessar esses serviços por uma SPA (Single Page Application) e um aplicativo mobile construídos com o framework Ionic e Angular.

A imagem também descreve que todos componentes desse ecossistema, menos a aplicação mobile, utilizam-se de containers Docker.

6.2.1 SERVIÇO DE BARRAMENTO RABBITMQ

O RabbitMQ também é um microserviço desse sistema, esse serviço será o responsável pelo barramento de mensagens dos microserviços, os microserviços se conectaram a ele se eles necessitarem de escutar mensagens ou enviar mensagens.

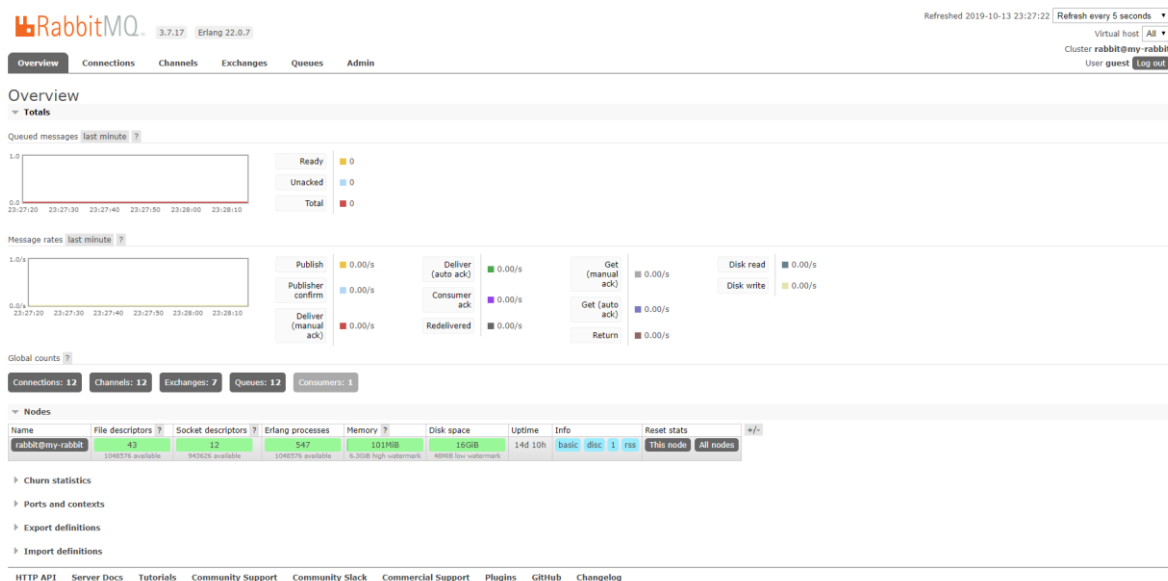
O comando abaixo foi utilizado para executar um container deste serviço com o Docker:

```
docker run -d --hostname my-rabbit --name some-rabbit -p 5672:5672 -p 5673:5673 -p 15672:15672 rabbitmq:3-management
```

Esse comando irá baixar a imagem “rabbitmq:3-management”, e em seguida vai rodar um container dessa imagem com o nome “some-rabbit”, com o nome para uso de redes “my-rabbit” e vai liberar as portas utilizadas pelo serviço e do gerenciado web.

Em seguida já será possível utilizar o RabbitMQ em outros serviços e acessar o gerenciador do RabbitMQ na porta 15672 para mais informações.

Figura 11 – Gerenciador do RabbitMQ



Fonte: (Elaborada pelo autor)

6.2.2 IDENTITY SERVICE

O Identity Service é o microserviço que se responsabiliza por autenticação e autorização, esse serviço é a parte mais dependente desse sistema, é o serviço que todos outros serviços dependem, por se tratar do serviço que registrar e autentica o usuário.

Esse microsserviço possui um *controller* com dois *endpoints* muito importantes, eles permitem que o usuário se registre ou autentique para receber seu token JWT.

Figura 12 – Controller do IdentityService

```
[AllowAnonymous]
[HttpPost( template: "authenticate")]
public IActionResult Authenticate([FromBody]LoginModel loginModel)
{
    var user = _userService.Authenticate(loginModel);
    return Ok(new { user.UserId, user.Username, user.Token });
}

[AllowAnonymous]
[HttpPost( template: "register")]
public IActionResult Register([FromBody]UserModel userModel)
{
    try
    {
        _userService.Create(userModel);
        return Ok();
    }
    catch(AppException ex)
    {
        return BadRequest( error: new { message = ex.Message });
    }
}
```

Fonte: (Elaborada pelo autor)

Como é exibido na figura 12, os dois endpoints possuem o atributo de decoração “AllowAnonymous” permitindo qualquer usuário sem autorização se registrar ou pedir seu Token, com o token JWT no cliente do usuário seja Web ou aplicativo mobile, ele pode acessar todos os outros serviços.

Após passar pelo *controller* usuário, é chamado o UserService, um serviço que tem a responsabilidade de autenticar e manter usuários.

6.2.2.1 Criar Usuário

Figura 13 – Método de criação de usuário

```
public void Create(UserModel userModel)
{
    var user = new User {Username = userModel.Username, Role = Role.User};

    if(_userRepository.GetByUsername(user.Username) != null)
        throw new AppException( message: "Username \"" + user.Username + "\" is already taken");

    CreatePasswordHash(userModel.Password, out var passwordHash, out var passwordSalt);
    user.PasswordHash = passwordHash;
    user.PasswordSalt = passwordSalt;

    _userRepository.Save(user);

    var cmd = new CreatePersonCommand()
    {
        IdentifierId = user.UserId,
        Username = userModel.Username,
        Name = userModel.Name,
        Email = userModel.Email
    };

    var wrapper = new WrapperCommand(cmd);
    var cmdJson = JsonConvert.SerializeObject(wrapper);
    _messageBroker.Call(cmdJson);
    _messageBroker.Close();
}
```

Fonte: (Elaborada pelo autor)

Na figura 13 é exibido o processo de criação de um usuário neste serviço, inicialmente o objeto é convertido para um novo modelo coerente com o serviço, em seguida é executada uma validação se o nome de usuário do usuário já está em uso, logo em seguida ocorre um processo de criptografia da senha do usuário, assim nos retornando um *hash* e *salt* para não ser preciso persistir a senha do usuário, após o usuário é persistido no banco do IdentityService e um comando de criação do Usuário é serializado e enviado para uma abstração do gerenciador do RabbitMQ, o *Message Broker*.

Agora que o comando foi enviado para RabbitMQ, outros serviços podem escuta-lo e executar a ação que foi implementada para que esse comando.

6.2.2.2 Autenticar

Figura 14 – Método de autenticação de usuário

```
public User Authenticate(LoginModel loginModel)
{
    if (string.IsNullOrEmpty(loginModel.Username) || string.IsNullOrEmpty(loginModel.Password))
        throw new AppException( message: "Username or password is incorrect");

    var user = _userRepository.GetByUsername(loginModel.Username);

    if (user == null)
        throw new AppException( message: "Username incorrect");

    if (!VerifyPasswordHash(loginModel.Password, user.PasswordHash, user.PasswordSalt))
        throw new AppException( message: "Password incorrect");

    user.Token = GenerateToken(user);
    return user;
}
```

Fonte: (Elaborada pelo autor)

O método de autenticação vai promover o usuário seu Token (JWT) de acesso ao sistema, esse método irá validar se o nome de usuário ou senha do usuário é nulo ou vazio, após o irá consultar o usuário pedido pelo o nome de usuário em seguida haverá algumas validações e se tudo ocorrer com sucesso no final o usuário irá receber um objeto User com seu Token.

6.2.3 FEED SERVICE

O Feed Service é o microsserviço responsável pela parte mais importante da rede social, seus feed de posts, esse serviço disponibiliza as funções de inserir posts, comentar esses posts, seguir e ser seguido por pessoas.

Figura 15 – Serviço ouvinte do RabbitMQ

```

consumer.Received += (model, ea) =>
{
    string response = null;

    var body = ea.Body;
    var props = ea.BasicProperties;
    var replyProps = Channel.CreateBasicProperties();
    replyProps.CorrelationId = props.CorrelationId;

    try
    {
        var message = Encoding.UTF8.GetString(body);
        var wrapper = JsonConvert.DeserializeObject<WrapperCommand>(message);
        var cmd = JsonConvert.DeserializeObject<Command>(value: JObject.Parse(message)["Command"].ToString(),
            wrapper.TypeCommand);

        using (var scope = services.CreateScope())
        {
            var handler = scope.ServiceProvider.GetRequiredService<IMediatorHandler>();
            handler.SendCommand((Command) cmd);
        }

        response = cmd.ToString();
    }
}

```

Fonte: (Elaborada pelo autor)

O FeedService possui um ouvinte do RabbitMQ, isso é um serviço que utiliza de um cliço de vida *singleton*, onde o mesmo está rodando sempre, escutando comandos em formato de mensagem do RabbitMQ e em seguida irá deserializar para um comando, após a deserialização, o comando é enviado para um manipulador de comandos, onde ele será responsável a promover uma ação para cada comando.

Após o Identity Service enviar um comando de pedido para a criação da entidade pessoa, esse serviço irá enviar o comando para seu manipulador.

Figura 16 – Manipulador do Comando de Criar Pessoa

```

public Task<Unit> Handle(CreatePersonCommand request, CancellationToken cancellationToken)
{
    var aggregate = new PersonAggregate(request);
    _personRepository.Save(aggregate);

    return Unit.Task;
}

```

Fonte: (Elaborada pelo autor)

O manipulador do comando criação de pessoa, somente irá criar um agregado de pessoa, onde irá abstrair validações e ações de pessoa, e no final será persistido a pessoa do usuário que criamos no primeiro serviço de identificação.

6.2.4 SUPPORT SERVICE

O SupportService é um simples serviço que tem a responsabilidade de persistir mensagens dos usuários.

Figura 17 – Controller do SupportService

```
router.post('/createSupportMessage', async (req, res): Promise<void> => {  
  try {  
    const { text, userId } = req.body;  
    const message = new Message();  
    message.text = text;  
    message.userId = userId;  
    message.save();  
  
    res.json({  
      message: 'Successfully saved support message',  
      status: 200,  
    });  
  } catch (error) {  
    res.status(error.status).json(error);  
  }  
});
```

Fonte: (Elaborada pelo autor)

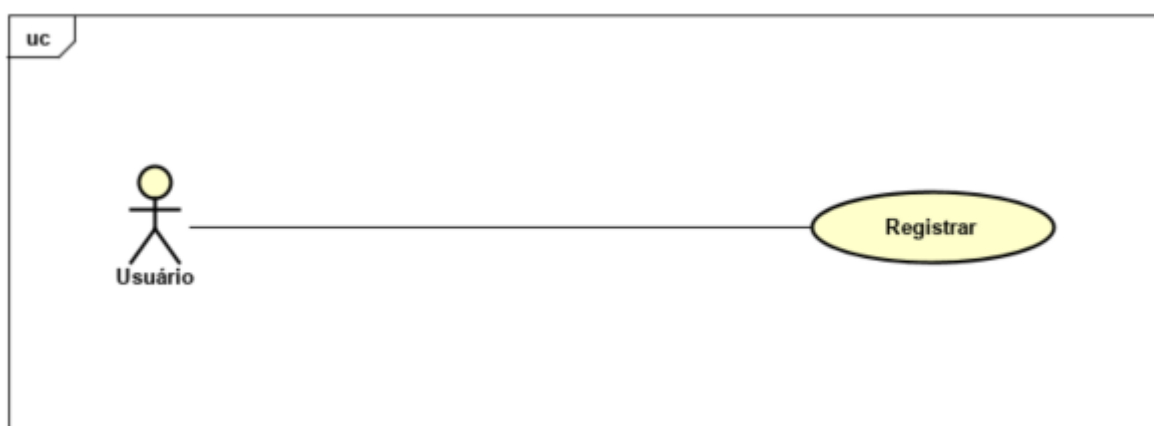
O *controller* do SupportService contém um único *endpoint* que irá receber as mensagens do usuário e persistir no seu banco não relacional.

7. VALIDAÇÕES

Após a implementação dos serviços é preciso validá-los, a validação vai ser feita por casos de uso, por fim devemos encontrar os resultados esperados em cada caso de uso.

7.1 CASO DE USO REGISTRAR

Figura 18 – Caso de uso do Registrar

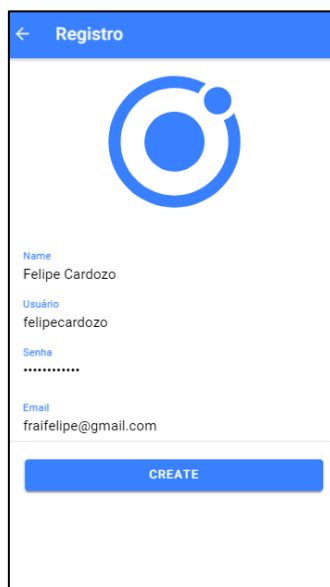


Fonte: (Elaborada pelo autor)


O caso de uso “Registrar” pode ser acessado pelo “Usuário”, para isso, será necessário utilizar o microserviço Identity Service que irá cadastrar um novo usuário e em seguida vai emitir um comando para o RabbitMQ que será recebido pelo microserviço Feed Service para a criação de uma Pessoa para esse usuário.

Ao se registrar no Flue, como é demonstrado na figura 18, o Identity Service vai registrar o usuário no seu banco de dados e a seguir irá cadastrar uma pessoa para esse usuário no Feed Service com as informações necessárias.

Figura 19 – Página de Registro



← Registro



Name
Felipe Cardozo

Usuário
felipecardozo

Senha

Email
fraifelipe@gmail.com

CREATE

Fonte: (Elaborada pelo autor)

A figura 19 exibe a tela de registro da aplicação, nela é possível colocar um nome pessoal, nome de usuário, senha e e-mail, após inserir os dados validos e apertar no botão de confirmação de registro, deve exibir uma mensagem de sucesso do registro.

7.2 CASO DE USO LOGIN

Figura 20 – Caos de Uso Login



Fonte: (Elaborada pelo autor)

Após o Identity Service registrar um Usuário em seu serviço e uma Pessoa no serviço do Feed, é possível fazer o login na aplicação, ao se registrar e acessar a página de login vai ser possível entrar no sistema.

Figura 21 – Página de Login

A imagem mostra a interface de login de uma aplicação. No topo, há um título 'Login'. Abaixo dele, há um ícone circular azul com um ponto no centro. Seguem dois campos de entrada: 'Usuário' com o valor 'felipecardozo' e 'Senha' com caracteres ocultos por pontos. Na base, há dois botões: 'Entrar' (azul) e 'Registrar' (cinza).

Fonte: (Elaborada pelo autor)

A figura 21 demonstra a página de login, nela só é necessário inserir seu usuário e senha cadastrados e apertar no botão para obter acesso, em seguida o usuário deve ser redirecionado para tela do Feed.

7.3 CASO DE USO SEGUIR PESSOA

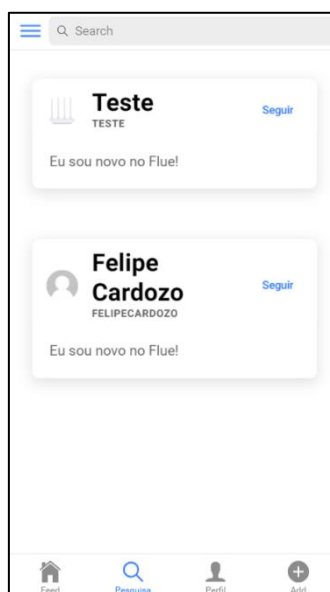
Figura 22 – Caso de Uso Seguir Pessoa



Fonte: (Elaborada pelo autor)

Para seguir outras pessoas o usuário deve se direcionar a página de pesquisa, abaixo uma imagem da tela com usuários preenchidos:

Figura 23 – Página de Pesquisa



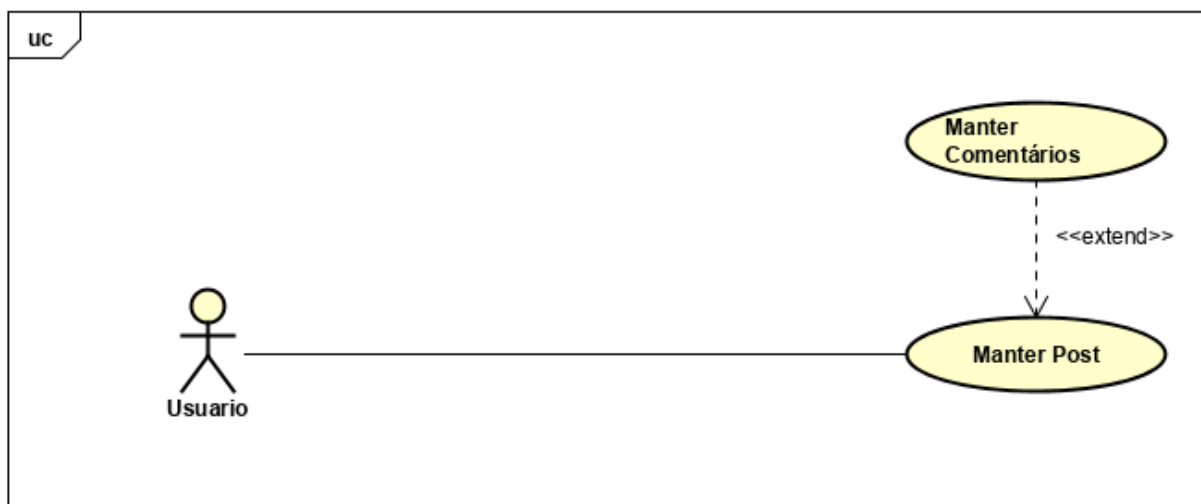
Fonte: (Elaborada pelo autor)

Na tela de pesquisa do aplicativo é possível encontrar outras pessoas para seguir. Logo a tela de pesquisa deve listar usuários de acordo com o que foi inserido na barra de pesquisa, que faz o filtro pelos nomes das pessoas.

Em cada bloco dos resultados de pesquisa deve ser exibido a foto de perfil, nome pessoal, nome de usuário e descrição, além disso deve conter um botão permitindo o usuário seguir ou parar de seguir as pessoas.

7.4 CASO DE USO MANTER POST

Figura 24 – Caso de Uso Manter Post e Manter Comentários



Fonte: (Elaborada pelo autor)

Ao acessar a página Novo Post, o usuário pode escrever um texto e ao apertar no botão de enviar, esse texto se transformará em uma nova publicação, um novo Post.

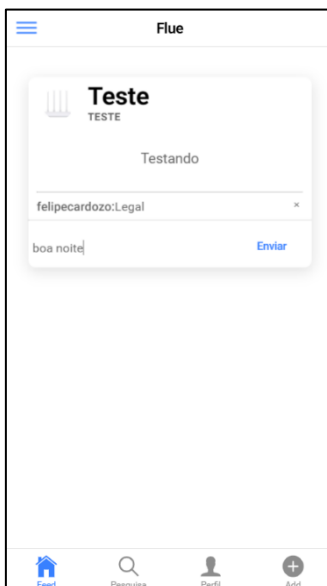
Figura 25 – Página Novo Post



Fonte: (Elaborada pelo autor)

O usuário também pode visualizar os Posts de Pessoas que ele segue, além disso, ele pode comentar nessas publicações. A Figura 26 demonstra isso:

Figura 26 – Página Feed



Fonte: (Elaborada pelo autor)

7.5 CASO DE USO EDITAR PERFIL

Figura 27 – Caso de Uso Editar Perfil



Fonte: (Elaborada pelo autor)

Ao acessar a página de Perfil e apertar na opção de Editar Conta, o usuário deve ser redirecionado para a página responsável de alteração dos dados do usuário. Os dados alteráveis devem ser nome pessoal, e-mail, descrição e sua foto de perfil. A seguir a tela:

Figura 28 – Página Editar Conta

Captura de tela da interface de usuário para a página 'Editar Conta'. No topo, há uma barra de navegação com um ícone de seta para trás e o texto 'Back' à esquerda, e o título 'Editar Conta' à direita. Abaixo, há um ícone de perfil de usuário cinza. Seguem-se os campos de formulário: 'Nome' com o valor 'Felipe Cardozo', 'Email' com o valor 'fraifelipe@gmail.com', e 'Descrição' com o valor 'Eu sou novo no Flue!'. No final, há um botão azul com o texto 'Atualizar'.

Fonte: (Elaborada pelo autor)

7.6 CASO DE USO ENVIAR MENSAGEM PARA O SUPORTE

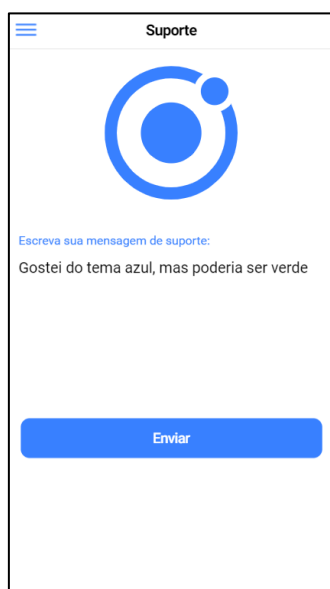
Figura 29 – Caso de Uso Enviar Mensagem para o Suporte



Fonte: (Elaborada pelo autor)

O usuário poderá acessar a página de Suporte e enviar uma mensagem, contendo somente texto. A seguir a tela:

Figura 30 – Página de Suporte



Fonte: (Elaborada pelo autor)

8. CONCLUSÃO

A arquitetura de microsserviços tende ser mais utilizada com o passar do tempo e com o aumento de necessidades de um sistema, é possível concluir que um sistema de tecnologias heterogênicas é muito favorável por permitir profissionais com habilidades diferentes trabalharem juntos e resolverem problemas distintos.

Para construir um ecossistema de microserviços é necessário unir diversos ramos da computação, para isso é fato que para a construção desses microsserviços é necessário conhecimentos e boas práticas em engenharia de software, arquitetura, infraestrutura e redes. Se esses campos de conhecimentos não estiverem amplos, o desenvolvedor deverá encontrar diversos obstáculos na implementação dessa arquitetura.

Foi constatado que implementar uma arquitetura de microserviços ainda é muito mais trabalhoso que uma aplicação com a arquitetura monolítica, realmente deve ser pensando muito se é benéfico sair da arquitetura monolítica, cujo é barata e fácil de implementar do que a de microsserviços. Mas a tendência é ficar mais fácil e acessível com o surgimento de novas ferramentas e boas práticas para o trabalhar com uma arquitetura de microsserviços.

8.1 TRABALHOS FUTUROS

Na elaboração desse trabalho foi pensado bastante no atrativo de pluralidade de tecnologias, o futuro desse trabalho de conclusão poderá focar mais nesses atrativos, talvez inserindo linguagens como Elixir para uso de concorrência, Python por causa de suas ferramentas e bibliotecas únicas de matemática e inteligência artificial e Rust ou C para tarefas pesadas que requerem processamento de linguagens baixo nível.

Alguns microserviços que podem ser implementados futuramente são os serviços de chat e de notificações para tornar o Flue uma aplicação de rede social completa. Uma boa escolha de tecnologia para o serviço de chat é o Elixir, linguagem que foi implementada do ecossistema Erlang, muito utilizado por ramos de telecomunicações por ser excepcional em termos de concorrência. E o serviço de notificação um simples serviço NodeJs será ser suficiente.

Um dos aspectos de comunicação entre microserviços que foi utilizado nesse trabalho de conclusão de curso foi o de filas de mensagens por barramentos como o RabbitMQ, esse método é excelente para chamadas assíncronas, mas para chamadas assíncronas não é a melhor opção, a nova ferramenta gRPC do Google parece estar bem avançada nesse aspecto de

comunicação síncrona entre serviços, mas como é uma tecnologia que precisa se provar ainda no mercado ficará para o futuro desse projeto.

REFERÊNCIAS BIBLIOGRÁFICAS

ADRIANO, T. Introdução ao RabbitMQ. 2018. Disponível em: <<https://medium.com/@programadriano/introdu%C3%A7%C3%A3o-ao-rabbitmq-4bfba460ad03/>>. Acesso em 16 de jun. 2019.

ANGELINI; LIMA. Microserviços e REST (sério?). 2017. Disponível em: <<https://elo7.dev/microservicos-rest/>> Acesso em 04 de abr. 2019.

ANICHE, M. Orientação a objetos e SOLID para Ninjas. Casa do Código, 2015.

ARAGÃO, T. SOLID—Princípios da Programação Orientada a Objetos. Disponível em: <<https://medium.com/thiago-aragao/solid-princ%C3%ADpios-programa%C3%A7%C3%A3o-orientada-a-objetos-ba7e31d8fb25>>. Acesso em 23 de mar. 2019.

BALTIERI, A. Microserviços - Introdução. 2017. (17m49s). Disponível em: <<https://youtu.be/2572gPN8LyI>>. Acesso em: 04 nov. 2018.

BLOCKHEAD. Como desenvolver Microserviços: Top 10 Hacks para Modelar, Integrar e Implantar Microserviços. Babelcube, 2017.

CAELUM. Arquitetura de Microserviços ou monolítica? 2015. Disponível em: <<http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica/>>. Acesso em 30 de set. 2018.

CASTILHO, R. Testando o RavenDB 2013. Disponível em: <<https://robsoncastilho.com.br/2013/05/30/testando-o-ravendb/>>. Acesso em 16 de nov. 2018.

DAGNINO, G. Faltam 100 mil profissionais para áreas de TI. 2018. Disponível em: <<https://www.dci.com.br/servicos/faltam-100-mil-profissionais-para-areas-de-ti-1.740407>>. Acesso em: 04 nov. 2018.

DUARTE, L. Introdução à Arquitetura de Microsserviços. 2017. Disponível em: <<http://www.luiztools.com.br/post/introducao-arquitetura-de-micro-servicos/>> Acesso em 02 de out. 2018.

EVO, 3 aspectos importantes sobre Containers, Microserviços e Docker 2018. Disponível em: <<https://www.eveo.com.br/blog/containers-microservicos-docker/>>. Acesso em 8 de abr. 2019.

FOWLER, M. Microservices, a definition of this new architectural term. 2014 Disponível em: <<https://martinfowler.com/articles/microservices.html/>> Acesso em 04 de nov. 2018.

FOWLER, S. Microsserviços prontos para a produção. Novatec, 2017.

IMPACTA. Entenda de uma vez por todas o banco de dados SQL Server 2017. Disponível em: <<https://www.impacta.com.br/blog/2017/12/22/entenda-de-uma-vez-por-todas-o-banco-de-dados-sql-server/>>. Acesso em 04 de maio. 2018.

MACORATTI, J. UML - Diagrama de Classes e objetos. Disponível em: <http://www.macoratti.net/net_uml1.htm>. Acesso em 13 de out. 2019.

MICROSOFT. Guia do .Net Core. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/core>>. Acesso em 29 de set. 2018.

MOREIRA; BEDER. Desenvolvimento de Aplicações e Microsserviços: Um estudo de caso. 2015. Disponível em: <<http://revistatis.dc.ufscar.br/index.php/revista/article/view/364/127/>> Acesso em 02 de out. 2018.

NASCIMENTO, W. Entendendo tokens JWT (Json Web Token), 2015. Disponível em: <<https://medium.com/tableless/entendendo-tokens-jwt-json-web-token-413c6d1397f6>> Acesso em 17 de nov 2018.

PASSOS, C. O que é o IONIC? 2018. Disponível em: <<https://medium.com/codigorefinado/oque-%C3%A9-o-ionic-4f8c7b94c51b/>>. Acesso em 04 de out. 2019.

PEREIRA. Aplicações web real-time com Node.js. Casa do Código, 2014.

POSGRADUANDO. As diferenças entre pesquisa descritiva, exploratória e explicativa. 2012. Disponível em: <<http://posgraduando.com/diferencas-pesquisa-descritiva-exploratoria-explicativa/>>. Acesso em 17 de nov. 2018.

RABBITMQ. Remote procedure call (RPC). Disponível em: <<https://www.rabbitmq.com/tutorials/tutorial-six-dotnet.html>> Acesso em 16 de jun. 2019.

SANTOS, L. Microserviços: dos grandes monólitos às pequenas rotas. 2017. Disponível em: <<https://medium.com/trainingcenter/microservi%C3%A7os-dos-grandes-mon%C3%B3litos-%C3%A0s-pequenas-rotas-adb70303b6a3/>> Acesso em 02 de out. 2018.

SOMMERVILLE, I. Engenharia de Software 9ª Edição. Pearson Prentice Hall, 2011.

VIEIRA, R. UML — Diagrama de Casos de Uso. 2015. Disponível em: <<https://medium.com/operacionalti/uml-diagrama-de-casos-de-uso-29f4358ce4d5/>> Acesso em 13 de out. 2019.