CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA (CEFET/RJ)

CEFET-CONNECTIONS: APLICANDO UMA ARQUITETURA DE MICROSERVIÇOS EM UMA REDE SOCIAL.

Daniel Dante dos Santos Viana

Professor orientador: Renato Mauro, M.Sc

Rio de Janeiro, RJ

Agosto / 2017

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA (CEFET/RJ)

CEFET-CONNECTIONS: APLICANDO UMA ARQUITETURA DE MICROSERVIÇOS EM UMA REDE SOCIAL.

Projeto Final II apresentado em cumprimento às normas do Departamento de Educação Superior do CEFET-RJ, como parte dos requisitos para obtenção do título de Tecnólogo em Sistemas para Internet.

Daniel Dante dos Santos Viana

Professor orientador: Renato Mauro, M.Sc

Rio de Janeiro, RJ

Agosto / 2017

Ficha catalográfica elaborada pela Biblioteca Central do CEFET/RJ

V614 Viana, Daniel Dante dos Santos
CEFET-Connections : aplicando uma arquitetura de microserviços em uma rede social / Daniel Dante dos Santos Viana.—2017.
35f. : il. (algumas color.) ; enc.

Projeto Final (Tecnólogo) Centro Federal de Educação Tecnológica Celso Suckow da Fonseca , 2017.
Bibliografia : f. 34-35
Orientador : Renato Mauro

1. Redes sociais on-line. 2. Serviços da web. 3. Arquitetura de rede. I. Mauro, Renato (Orient.). II. Título.

CDD 006.76

DEDICATÓRIA

Primeiramente agradeço a Deus.

Dedico também aos meus pais Renato e Maria de Fátima (in memória), ao meu irmão Henrique, à minha esposa Larissa e aos meus familiares pelo apoio e carinho que me deram, sem medir esforços, para que eu concluísse mais essa etapa de minha vida.

AGRADECIMENTOS

Agradeço de forma muito especial ao meu orientador Renato Mauro que nunca desistiu de mim, me incentivou e me motivou quando a esperança já estava chegando ao fim.

A todos os meus amigos da vida que torceram e torcem por mim.

A todos que passaram por minha vida nesses anos de formação, pois sempre deixaram um pedacinho de si e levaram um pedacinho de mim.

SUMÁRIO

CEFET-CONNECTIONS: APLICANDO UMA ARQUITETURA DE MICROSERVIÇOS EM UMA REDE SOCIAL.

LISTA DE FIGURAS

“Cada sonho que você deixa para trás, é um pedaço do seu futuro que deixa de existir”.

Steve Jobs

RESUMO

O objetivo deste trabalho foi implementar uma arquitetura com funcionalidades comuns em aplicações baseadas em microserviços a fim de explorar as vantagens, desvantagens e dificuldades que este modelo arquitetônico oferece.

Para aplicar a implementação da arquitetura, foi sugerida a ideia de uma rede social para os alunos do *CEFET* que foi intitulada *CEFET-Connections*. Nessa rede social é possível o aluno publicar posts, criar comentários e curtir *os posts.*

Para conclusão do projeto foi necessário o estudo de diversos padrões arquiteturais existentes, a fim de deixar o sistema escalável e com um bom desempenho. Por se tratar de uma arquitetura de microserviços, foi necessária a implementação de oito microserviços, além da utilização de um banco *NoSQL* conhecido como *MongoDB* foi necessário o uso de um cache distribuído *Redis*, um a gente de mensageria o *RabbitMQ* e uso de containers com *Docker para o isolamento dos serviços*.

ABSTRACT

The objective of this work was to implement an architecture with functionalities Common applications in micro-service applications in order to exploit the advantages, Disadvantages and difficulties that this architectural model offers.

In order to implement the architecture implementation, the idea of a social network for CEFET students that was called CEFET-Connections was suggested. In this social network it is possible for the student to publish posts, create comments and enjoy the posts.

To conclude the project it was necessary to study several existing architectural patterns in order to make the system scalable and performing well. Because it was a microservice architecture, it was necessary to implement eight microservices, in addition to the use of a NoSQL database known as MongoDB, it was necessary to use a Redis distributed cache, a message to RabbitMQ and container use with Docker For the isolation of services.

Capítulo 1

1. Introdução

Atualmente existem diversos meios de comunicação que um sistema utiliza para trocar dados com outros. Por volta de 1999, surgiu a necessidade de padronizar as comunicações entre diferentes plataformas (PC, Mainframe, Mac, Windows, Linux, entre outros) e linguagens de programação (PHP, C#, Java, etc). Diversos padrões foram propostos, DCOM e CORBA, mas nenhum obteve êxito suficiente, tanto pela independência de plataforma como pelo modelo proposto (MORAES et al.).

É nesse contexto que surgem os *Web services*, como uma alternativa que visa ajudar na comunicação entre sistemas distribuídos. Os *Web services* são componentes que permitem as aplicações enviarem e receberem dados e são na sua essência, uma solução que permite a comunicação entre aplicações diferentes (JUNIOR e CANDEIAS). Recorrendo a esta tecnologia, é possível que novas aplicações possam ser facilmente integradas com outras previamente existentes, ultrapassando os tradicionais problemas de interoperabilidade existentes quando as aplicações comunicantes encontram-se desenvolvidas para diferentes plataformas e/ou em diferentes linguagens de programação.

Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, um formato intermediário como *XML, Json, CSV, etc*. Dentre as diferentes implementações de Webservices existe uma que é abordada na pesquisa, o Representational State Transfer, mais conhecido como REST. Como uma abordagem de programação, REST é uma alternativa leve para Web Services, um estilo de arquitetura para sistemas de hipermídia distribuídos, onde aplicações conseguem se comunicar com outras aplicações utilizando o HTTP como protocolo de comunicação (HE HAO, 2013).

Com o aumento dos sistemas corporativos e a comunicação entre diferentes sistemas em diferentes plataformas, foram criadas algumas técnicas para melhorar a comunicação entre as diversas aplicações. Existem dois tipos de aplicações quando falamos de sistemas corporativos, aplicações monolíticas onde todos os componentes de negócio encontram-se dentro de uma mesma unidade de implantação e os microserviços onde é dividido o grande sistema monolítico em pequenos outros sistemas.

Em suma, o estilo arquitetônico microserviço é uma abordagem para o desenvolvimento de um pedido único, como um conjunto de pequenos serviços, cada um

executando em seu próprio processo e se comunicando com mecanismos leves, muitas vezes, uma API de recursos HTTP (NAMIOT e SNEPS-SNEPPE, 2014). Há um mínimo de gerenciamento centralizado destes serviços, que podem ser escritos em diferentes linguagens de programação e utilizam diferentes tecnologias de armazenamento de dados. Estes são ligados através de serviços, onde é abordado o REST como um serviço web para a comunicação entre sistemas, com um estilo de arquitetura construído sobre determinados princípios básicos da Internet. Contudo, existem vantagens e desvantagens nas diferentes aplicações.

Um dos principais pontos negativos nas aplicações monolíticas é que se tem um grande ponto único de falha, um exemplo seria um ERP, o sistema desenvolvido para cuidar de toda a empresa desde o financeiro, RH e pagamentos até o contato com clientes. Todas essas funcionalidades estão agrupadas em um único sistema, isso significa, por exemplo, que caso ocorra algum erro no cadastro de clientes, que é apenas uma funcionalidade do sistema, acarretará um erro em todo o sistema deixando-o fora do ar. Na aplicação com microserviços isso não ocorre, pois caso o cadastro de clientes pare, apenas esse sistema é impactado não gerando problemas em todos os outros que se comunica por serviço. Na arquitetura monolítica se usa bibliotecas ou um módulo a fim de se componentizar partes do sistema, ou seja, estão ligados por um programa e por chamadas as funções em memória (FOWLER e LEWIS, 2014). A responsabilidade de um ponto de falha está dentro do sistema que utiliza o módulo, logo uma forte responsabilidade e acoplamento são adquiridos pelo sistema, enquanto que com a utilização de microserviço, este se comunica com outros sistemas que tem uma responsabilidade única e estão fora do mesmo processo, se comunicando através de solicitações de serviços web. Uma das principais razões para a utilização de serviços como componentes (em vez de bibliotecas) é que os serviços podem ser implementados independentes. Caso exista um sistema que contenha várias bibliotecas em um único processo, uma mudança para qualquer resultado de uma única função, o que irá gerar impacto para o sistema inteiro. Mas se o sistema é decomposto em vários serviços (pequenos sistemas) pode-se esperar muitas mudanças no serviço sem impacto (FOWLER e LEWIS, 2014).

Para aplicar o conceito de microserviços foi escolhida a criação de uma rede social chamada CEFET-*Connections*, apenas para os alunos do *CEFET*, onde os mesmos possam interagir na rede social através de posts (imagens e textos) categorizados sobre o *CEFET* em uma timeline, onde cada post pode ser curtido e comentado. Os posts são categorizados por assuntos específicos referentes ao *CEFET*, são eles: disciplinas, cursos e melhorias. Na timeline podem-se filtrar os posts apenas pelos assuntos que o usuário tiver mais interesse.

No assunto disciplinas, os posts terão subcategorias que são todas as disciplinas existentes no *CEFET*, onde os alunos poderão escrever apenas posts sobre as disciplinas oferecidas, como exemplo, pode-se citar a troca de conhecimento e divulgação de exercícios das disciplinas. No assunto curso, os alunos poderão escrever apenas posts relacionados aos cursos oferecidos, como exemplo, pode-se citar o uso da rede social para os novos alunos, que muitas vezes querem saber mais sobre o curso que estão fazendo, podendo existir feedbacks com outros alunos que estão terminando, ou, até mesmo para a divulgação de palestras relacionadas ao seu curso. No assunto melhorias os alunos poderão escrever posts sobre as possíveis melhorias que podem ocorrer no *CEFET*, como exemplo, pode-se citar o uso da rede social para feedbacks dos alunos sobre a infraestrutura da instituição, desde as melhorias da cantina até as salas de aulas.

Assim, a abordagem dessa pesquisa é a aplicação de uma arquitetura de microserviços para a rede social *CEFET-Connections*, a fim de verificar a aplicabilidade desse tipo de arquitetura em um sistema de grande escala como a rede social. Além desta introdução, este trabalho está divido em capítulos conforme a estrutura a seguir: o capítulo 2 aborda os principais conceitos da arquitetura proposta, dando uma visão geral sobre cada um, tais como REST e Microserviços, bem como sobre o conceito de se ter uma arquitetura bem dividida e desacoplada onde seus deploys sejam independentes. Além disso, a utilização de virtualização baseada em containers facilita o deploy e garante que os sistemas rodem em qualquer ambiente independente da sua configuração, seja ele produção, homologação e desenvolvimento.

Capítulo 2

2. Fundamentação Teórica

Este capítulo aborda os principais conceitos do tema estudado. Serão apresentados os conceitos principais de uma arquitetura de microserviços , bem como o meio de comunicação REST , virtualização de containers e o fornecimento de uma rede social para a aplicabilidade da arquitetura proposta.

2.1. REST WS

O termo REST foi primeiramente introduzido por *Roy Fielding* para descrever a arquitetura Web em sua tese de doutorado *(Architectural Styles and the Design of Network-based Software Architectures)*, um dos principais autores da especificação do protocolo HTTP, na qual descreve REST como um estilo arquitetural onde recursos são seu foco e são utilizados em aplicações web. Segundo Fielding (2000, tradução do autor)

> Transferência de Estado Representacional (REST) estilo de arquitetura para sistemas de hipermídia distribuídos, descrevendo a engenharia de software princípios orientadores REST e as limitações de interação escolhidas para manter esses princípios, enquanto contrastando-as com as restrições de outros estilos arquitectónicos. REST é um híbrido estilo derivado de vários dos estilos arquitectónicos baseados em rede. (FIELDING, 2000, p.76)

Existem algumas restrições e características que são definidas para que uma arquitetura seja considerada REST. Ao longo desse tópico, serão apresentados seus conceitos, que são eles: Arquitetura Cliente-Servidor, estado de conversação *Stateless* com o servidor, Cliente Cacheável, sistema em camadas e Código sob demanda.

A primeira restrição é utilização de uma arquitetura Cliente/Servidor para garantir a separação dos interesses. Ao separar as preocupações de interface de usuário e armazenamento de dados, podemos deixar a aplicação portável para qualquer plataforma, melhorando e simplificando os componentes do servidor (FIELDING, 2000). A seguir a imagem de uma arquitetura Cliente/Servidor.

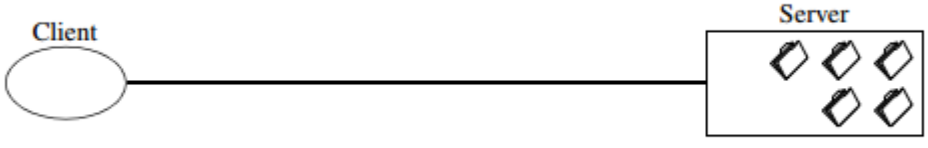

Figura 1 – Exemplo de uma arquitetura Cliente-Servidor de FIELDING (2000).

Na segunda restrição é definida para a comunicação entre o cliente e o servidor, devendo ser *stateless*, ou seja, cada requisição do cliente para o servidor deve conter todas as informações necessárias para o entendimento da requisição e não pode tomar como vantagem qualquer informação armazenada no contexto do servidor, dessa forma, mantendo o estado da sessão inteiramente do lado do cliente. Com essa restrição são geradas algumas propriedades, como visibilidade, confiabilidade e escalabilidade (FIELDING, 2000).

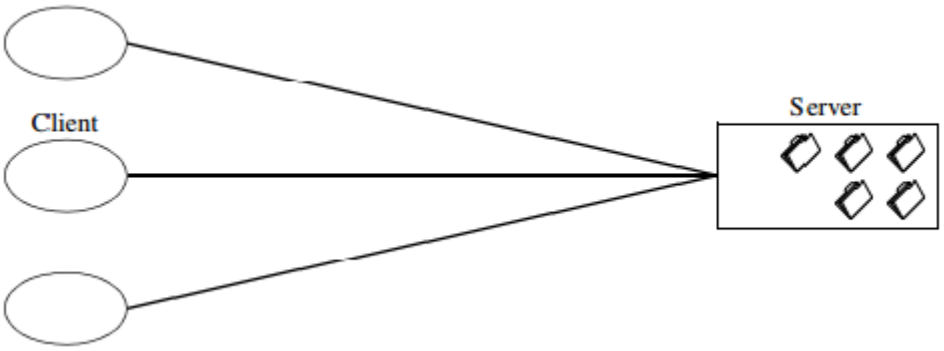

Figura 2 – Figura representando Cliente-Stateless-Server de FIELDING (2000).

A Terceira restrição adicionada a essa arquitetura é o *cache, que é* um armazenamento temporário no disco rígido de páginas web que podem ser imagens ou outros documentos e ficheiros. Utilizando técnicas de *cache* é possível reduzir o uso da largura de banda disponível, aumentar a velocidade do acesso, entre outras vantagens. Baseado nesse conceito, os navegadores ao acessarem um site, guardam uma cópia dos arquivos usados pelo mesmo no disco rígido, a cópia pode ser usada na montagem da página ao invés de trafegar todo o arquivo original pela Internet inteira até o seu computador novamente.

A restrição c*ache* requer que cada dado da resposta de uma requisição ao servidor seja explicitamente indicada como cacheável ou não-cacheável. Caso seja cacheável, o cliente tem o direito de reutilizar esse dado mais tarde (FIELDING , 2000). Com o uso do c*ache* na arquitetura, conseguimos melhorad a eficiência, a escalabilidade e a redução da latência média de uma série de interações.

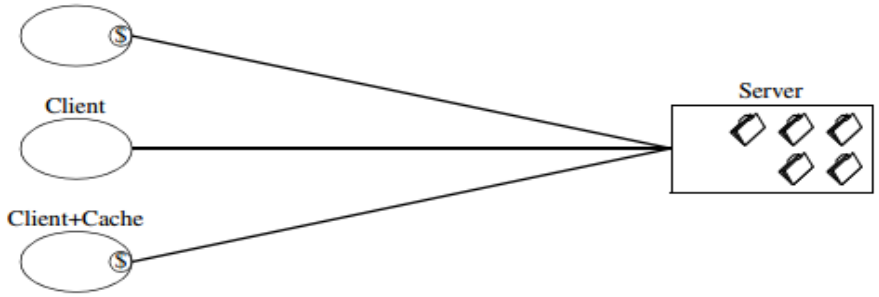

Figura 3 – Client-Cache-Stateless-Server de FIELDING (2000).

Existem algumas características que são essencias para distinguir o estilo arquitetônico REST das demais baseadas em redes, além das restrições anteriormente citadas. *Interface* uniforme é a característica central da arquitetura REST conforme mencionado por Fielding "a característica central que distingue o estilo arquitetônico REST do outros estilos com base em rede é sua ênfase em uma interface uniforme entre os componentes." (2000, p.81, tradução do autor) Assim , deve haver uma interface uniforme entre o cliente e o servidor, com o objetivo de torná-los independentes um do outro. Da mesma forma que na linguagem de programação java, temos uma interface, para que os utilizadores não saibam de sua implementação.

Para se obter uma interface uniforme, várias restrições de arquitetura precisam ser feitas e em REST é definida por quatro restrições de interface: identificação de recursos; manipulação de recursos através de representações; mensagens auto-descritivas; e hipermídia como o motor do estado do aplicativo (FIELDING , 2000).

Sistema em camadas é outra restrição identificada nessa arquitetura onde a aplicação deve ser construída em camadas. Entretanto, uma camada só pode ver a camada imediatamente abaixo (FIELDING, 2000). O objetivo principal deste pré-requisito é garantir que as aplicações sejam escaláveis. Ao restringir o conhecimento do sistema a uma única camada, colocamos um limite da complexidade geral do sistema e promovemos uma independência. A principal desvantagem dos sistemas de camadas é que adiciona uma sobrecarga e latência para o tratamento de dados reduzindo o desempenho percebido pelo usuário.

A última restrição adicionada a arquitetura é código sob demanda, o cliente deve ser capaz de executar scripts armazenados no servidor de forma a estender as funcionalidades do cliente. Um exemplo disso é a habilidade dos browsers HTTP executarem JavaScripts. Este é o único pré-requisito opcional para arquiteturas REST (FIELDING, 2000).

REST foi inicialmente descrito como uma forma de arquitetura para o protocolo HTTP, mas não está limitado a este protocolo. O padrão REST pode ser baseado em outros protocolos de aplicações desde que haja um vocabulário uniforme e rico para indicar a transferência de estado, porém há condições conceituais que devem ser seguidas para que possa maximizar a abstração, manter o alto nível de desacoplamento entre as camadas de aplicação e minimizar os round trips entre cliente/servidor. A World-Wide Web é considerada uma implementação do estilo arquitetural REST. Os browsers são os clientes que requisitam as representações de recursos, retornada nos formatos HTML, JPEG, etc. O HTML suporta hipermídia de forma que um cliente não é obrigado a conhecer o caminho de todos os recursos (páginas) da aplicação, mas estes são normalmente apresentados na sua raiz (página inicial).

A ideia da transferência representacional de estado está diretamente ligada à hipermídia retornada junto aos recursos e por isso, uma aplicação não é REST se o formato utilizado na representação do recurso não suportar hipermídia.

2.2. Arquitetura de Microserviços

Microserviço é relativamente um novo termo para padrões de arquitetura de software. Arquitetura de microserviço é um conjunto de pequenos serviços independentes. Cada um dos serviços roda em um processo próprio. Os serviços devem se comunicar utilizando uma abordagem leve, como os que rodam sobre o HTTP (NAMIOT e SNEPS-SNEPPE, 2014). Nessa abordagem recorreremos ao REST, conforme descrito no tópico anterior. Além disso, os serviços devem poder ser implantados independentes. Com essa independência podemos criar aplicações em diferentes plataformas ou linguagens, se comunicando através de serviços e usando seus próprios modelos de dados, tornando-as independentes.

Mediante ao explicado anteriormente o oposto a essa abordagem permite identificar outro tipo de arquitetura, que é conhecida como Monolítica. Nesta abordagem podemos exemplificar utilizando a linguagem Java para Web. Quando queremos fazer a implantação de uma aplicação, geramos um arquivo WAR que é um único arquivo que contém toda a aplicação compactada. Internamente esse arquivo contém todos os serviços e componentes em um único arquivo, tornando muito fácil sua implantação, sendo uma das muitas vantagens para essa abordagem. Além disso, podemos dizer que temos uma única equipe tomando conta dessa aplicação, facilitando o seu desenvolvimento e gerenciamento da equipe (NAMIOT e SNEPS-SNEPPE, 2014).

Existem algumas desvantagens na arquitetura monolítica, uma delas é a dificuldade encontrada para a compreensão e a modificação desse tipo de aplicação, especialmente se ela for uma grande aplicação (NAMIOT e SNEPS-SNEPPE, 2014). Com a aplicação crescente é difícil adicionar novos membros no time ou substituir os que deixam o time. A base do código se torna muito complicada, diminuindo a produtividade do time. Com isso, podemos observar que com o passar do tempo a qualidade pode diminuir, pois para aumentar a produtividade é preciso aumentar o número de desenvolvedores que chegam nessa base de código, ficando difícil de ser mantida e dificultando o entendimento real do problema, aumentando a falta qualidade do código, devido ao curto tempo que é estimado para que se tenha uma alta produtividade. Sendo assim, a modularidade acaba sendo destruída. Aplicações monolíticas impedem que existam desenvolvedores trabalhando de forma independente, isso torna o desenvolvimento contínuo muito difícil. Para que se possa atualizar uma aplicação nessa abordagem, mesmo que seja pequena modificação, é necessário reimplementar o aplicativo inteiro (NAMIOT e SNEPS-SNEPPE, 2014).

Outra desvantagem na arquitetura monolítica é a pilha de tecnologia, ou seja, o conjunto de tecnologias utilizadas em uma aplicação para o seu desenvolvimento. Torna-se muito difícil ou até impossível a mudança de tecnologia ou atualização, conforme descrito por Namiot (NAMIOT e SNEPS-SNEPPE, Tradução):

> Com a arquitetura monolítica, é muito difícil (ler-impossível) muda-lá. Por exemplo, não há quase nenhuma maneira de alterar a estrutura de desenvolvimento, etc. Isso pode dificultar o incremento de uma nova tecnologia. E todos os componentes com a aplicação irá aderir a tecnologia selecionada no início. (NAMIOT e SNEPS-SNEPPE, 2014, p.24)

Com isso, não é possível atualizar as tecnologias, tendo de trabalhar em uma mais antiga, não evoluindo ou adequando a aplicação, o que pode tornar a aplicação desatualizada.

A imagem a seguir mostra um exemplo das duas arquiteturas citadas:

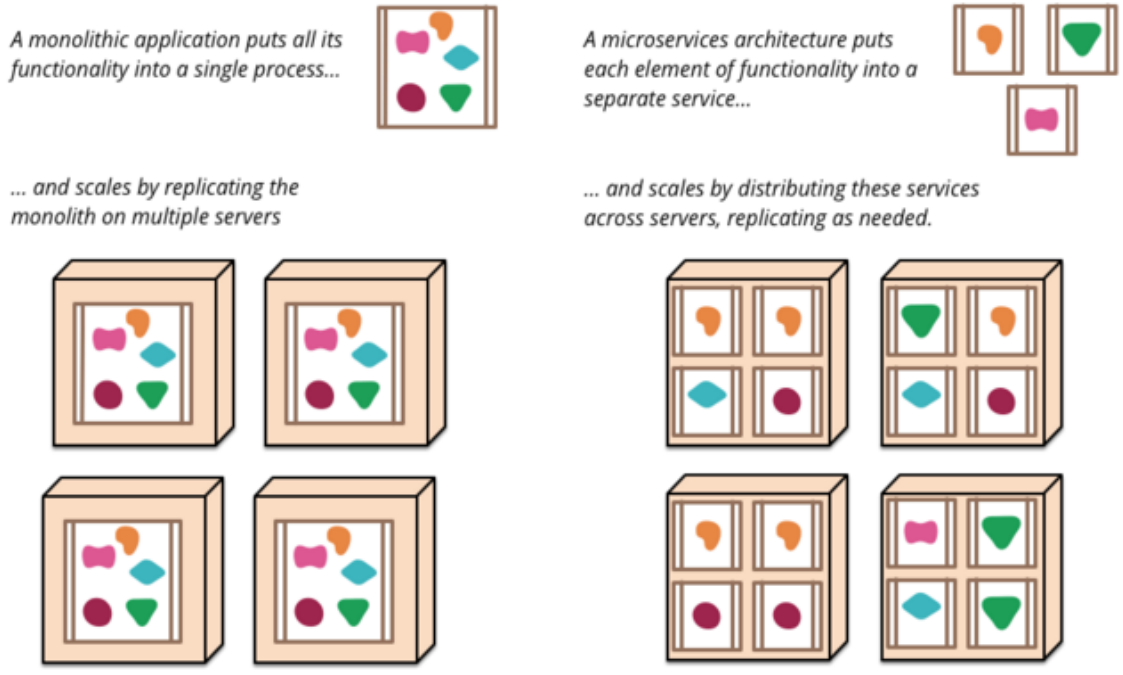

Figura 4 – Monolítica e Microserviço.

Algumas características podem ser definidas no estilo de arquitetura de microserviços, conforme descreve Fowler no texto a seguir (FOWLER e LEWIS, Tradução):

> O termo "Arquitetura de Micro-serviços (Microservice Architecture)" surgiu nos últimos anos para descrever uma maneira específica de desenvolver software como suites de serviços com deploy independente. Embora não exista uma definição precisa desse estilo de arquitetura, há certas características comuns em relação à organização à capacidade de negócios, ao deploy automatizado, à inteligência nos terminais e ao controle descentralizado de linguagens e de dados. *(FOWLER e LEWIS, 2014)*

No texto de Fowler é possível identificar as características comuns como: *organização a capacidade de negócios, deploy automatizado, à inteligência nos terminais, controle descentralizado de linguagens e de dados.* Essas características fazem com que seja possível identificar ou construir esse estilo de arquitetura.

Muitas definições são usadas nessa arquitetura, assim como componentização por serviço. Se torna difícil a definição do que faz um componente, porém Fowler define componente como uma unidade de software que é independente, substituível e atualizável. (FOWLER e LEWIS, 2014).

Na arquitetura de Microserviços, assim como na arquitetura monolítica, utilizará de bibliotecas, que são também por definição, componentes, porém com um conceito um pouco

diferente. Bibliotecas são componentes que estão ligados a um programa fazendo chamadas a funções em memória, diferentemente de serviços que são processos independentes que se comunicando através de solicitações na Web. Uma das grandes razões que se tem, ao utilizar serviços como componentes, é que serviços são de deploy independentes, ou seja, caso ocorra uma mudança em um serviço, não é necessário que toda a aplicação seja reimplantada, apenas o serviço modificado, mas se uma aplicação utiliza de múltiplas bibliotecas em um único processo, resultará em uma reimplantação de toda a aplicação.

Quando analisamos uma aplicação de grande escala, conseguimos identificar uma clara divisão feita pela gestão, gerando algumas equipes na área de tecnologia, são elas: equipe de interface de usuário (UI), equipe lógica do lado do servidor, equipe de banco de dados. Com essa divisão fica claro que a menor das mudanças pode levar um enorme tempo para execução e aprovação orçamental (FOWLER e LEWIS, 2014). Em uma abordagem com microserviços é diferente, pois é feita sobre a divisão dos serviços em torno da capacidade de negócio. Existe uma equipe para cada serviço da área de negócio, gerando uma pilha de tecnologia para cada serviço, atendendo desde interface com usuário, banco de dados e lógica do lado do servidor. Embora aplicações monolíticas também possam ser divididas dessa forma, não é comum encontrar e na maioria das vezes, difícil de aplicar, pois a modularização e os limites nessa arquitetura se torna difícil identificar. Com a separação mais explicita exigido por componentes de serviços, torna-se fácil a identificação dos limites de cada equipe (FOWLER e LEWIS, 2014).

A entrega com foco em produto e não em projeto é um dos princípios dos defensores de Microserviços, pois não é apenas a entrega de pedaços de um software a ser concluído e a equipe dissolvida. A ideia principal é seguir o modelo de produto, onde a equipe constrói o produto e fica com ele durante sua vida completa. A ideia de um produto ligado a recursos de negócios olha para o software como um conjunto de funcionalidades para ser concluída. Podendo ajudar seus usuários a aumentar a capacidade de negócio (FOWLER e LEWIS, 2014). Esse tipo de abordagem também pode ser aplicada a uma arquitetura monolítica, porém com a divisão do negócio em vários serviços, tornando fácil a ligação entre o usuário e o desenvolvedor .

Com a clara divisão da equipe em áreas de negócio e consequentemente o sistema monolítico em vários outros pequenos sistemas, formando componente de serviços, é descentralizada a necessidade de se ter uma dependência de uma única pilha de tecnologia, ou seja, conseguimos comunicar sistemas de diferentes plataformas e linguagens para compor uma única aplicação. Com isso a grande aplicação não depende de uma tecnologia especifica,

tendo vários times para cada serviços da área de negócio, independentes de tecnologia. A descentralização do gerenciamento de dados também segue a mesma ideia, pois para cada microserviço existe sua própria base de dados. Conforme a Figura 5 a seguir:

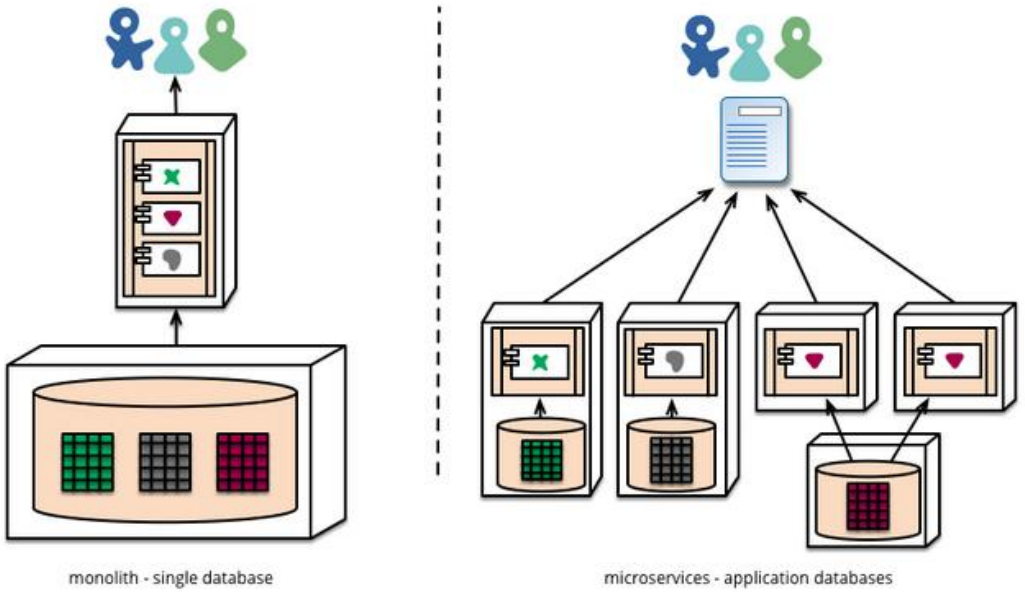

Figura 5 – Monolítica e Microserviço Banco de dados.

2.3 Containers

Os *containers* são um método de virtualização de um sistema operacional que permite executar uma aplicação e suas dependências em processos com recursos isolados. Os *containers* permitem facilmente empacotar códigos, configurações e dependências de uma aplicação, oferecendo um ambiente consistente e produtivo ao desenvolvimento e controle de versão. De acordo com AGARWAL (2015) virtualização a nível de sistemas operacionais é a existência de múltiplas instâncias de espaço de usuários isolados. Tais instâncias têm seus próprios recursos, sistemas de arquivos e são executados completamente isolados um dos outros. Com a criação de *containers* a aplicação se torna abstraída de todas as infraestruturas subjacentes, o que é essencial para o desenvolvimento isolado de serviços em uma arquitetura de microserviços.

Em comparação com as máquinas virtuais, os *containers* se tornam mais leves, pois as aplicações que rodam em *containers* utilizam chamadas ao sistema operacional host, ou seja, todos os *containers* utilizam um único sistema operacional e compartilham o kernel do sistema entre si, tornando mais leve e consumindo menos recursos que as máquinas virtuais.

Porém esta forma de virtuliazação não é tão flexível quanto as máquinas virtuais, uma vez que não se pode hospedar um sistema operacional diferente em comparação ao host (AGARWAL, 2015).

A seguir a Figura 5 consiste em demonstrar a estrutura dos *containers* .

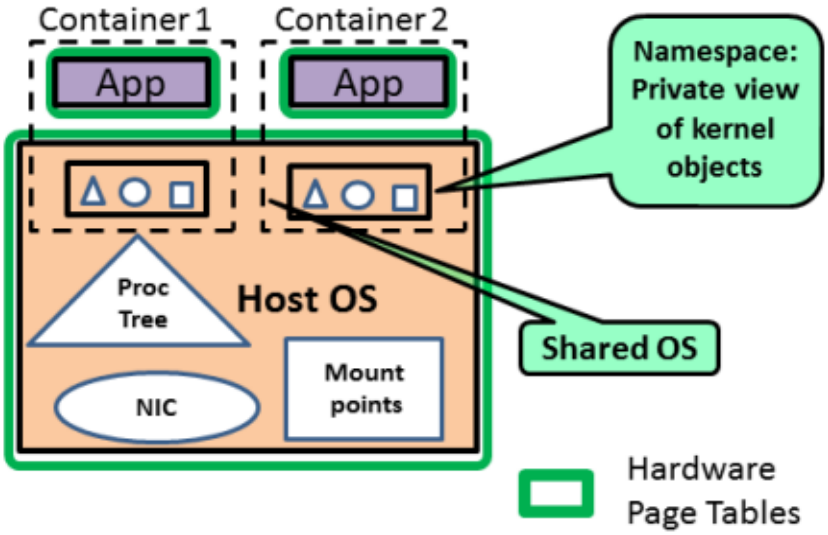

Figura 6 – Estrutura dos *containers*

As tecnologias de containers são vitais para aplicações web que utilizam microserviços, pois se tem a garantia de uma plataforma segura e portátil em nuvem. As principais tecnologias são:

- ***Solaris Containers*:** Teve sua primeira release em 2004, é a combinação de controles de recursos do sistema e a separação fornecida por zonas. As zonas atuam como servidores virtuais completamente isolados em uma única instância do sistema operacional.
- ***LXC***: Desenvolvido em 2008, é a primeira e mais completa implementação de um gerenciador de *containers linux.* O kernel do Linux fornece a funcionalidade de cgroups que permite a limitação e priorização de recursos (CPU, memória, E/S de bloco, rede, etc.) sem a necessidade de iniciar qualquer máquina virtual e também a funcionalidade de isolação do namespace que permite o isolamento completo de um aplicativo. Além disso, permite a visão do ambiente operacional, incluindo árvores de processo, rede, IDs de usuário e sistemas de arquivos montados.
- ***Docker***: é um projeto de código aberto lançado em 2013 que permite a criação de *containers* que rodam softwares de recursos compartilhados, fornecendo uma camada adicional de abstração e automação de virtualização de sistema de nível operacional em

Linux, Mac OS e Windows, sendo assim uma solução de virtualização em nível de sistema operacional.

- ***rkt Containers***: O rkt é um mecanismo de *containers* de aplicativos desenvolvido para ambientes nativos em nuvem. Possui uma abordagem pod-native, um ambiente de execução plugável e uma área de superfície bem definida que o torna ideal para integração com outros sistemas. Este foi desenvolvido em 2014.

2.3.1 Docker

O *Docker* é uma plataforma aberta para desenvolvedores e administradores de sistemas, usada para construir, executar e disponibilizar aplicações utilizando o isolamento de recursos do *kernel linux* para a criação de *containers*. O isolamento e a segurança permitem que você execute múltiplos *containers* simultaneamente em um determinado host. Os recipientes são leves, porque eles não precisam da carga extra de um hypervisor (usado em máquinas virtuais), mas são executados diretamente no kernel da máquina host. Isso significa que você pode executar mais *containers* em uma determinada combinação de hardware do que se você estivesse usando máquinas virtuais. Você pode até mesmo executar recipientes Docker dentro de máquinas host, que são máquinas virtuais. A estrutura básica da plataforma está descrita na Figura 6.

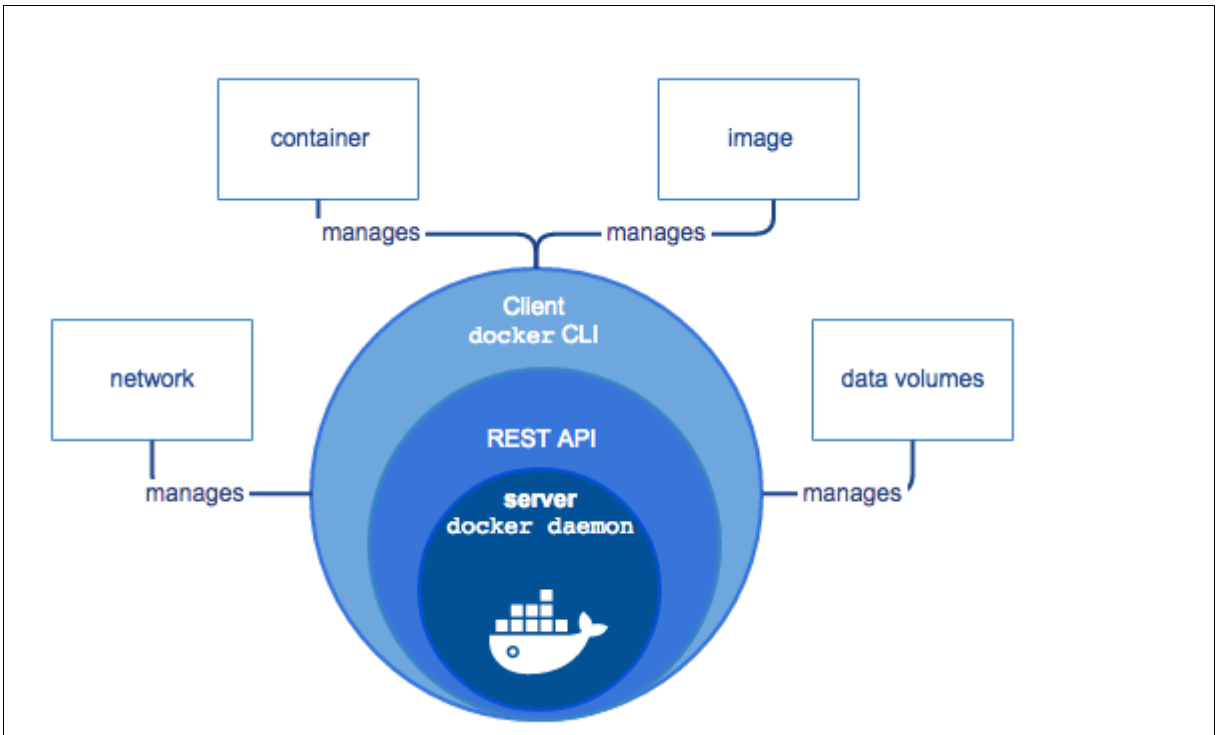

Figura 7 – Plataforma Docker

Os componentes existentes do docker conforme descrito na documentação da plataforma Docker (2017), são:

- ***Docker Engine***: Escuta as solicitações da API do Docker e administra objetos do Docker, como imagens, *containers*, redes e volumes. Um daemon também pode se comunicar com outros daemons para gerenciar os serviços do Docker.
- ***Docker Client***: É o componente que executa as solicitações para o daemon do *Docker Engine.*
- ***Dockerfile***: É um arquivo de texto onde são descritos os comandos para a criação das *imagens docker.*
- ***Docker Imagens***: A base para criação de *containers,* onde são constituídas de uma série de camadas. Cada camada representa uma instrução no *Dockerfile* na imagem.
- ***Docker Registry***: São os repositórios das imagens.
- ***Docker Containers***: São os *containers* gerados a partir de imagens.
- ***Docker Componse***: É uma ferramenta para definir e executar múltiplos containers dockers. Suas configurações são definidas em arquivo docker-compose, onde são definidos os containers. Em apenas comando é possível rodar todos os containers definidos no arquivo.

Conforme visto na seção 2.3 tanto *Docker* quanto as demais soluções de criações de *containers* existentes, possuem uma limitação de interoperabilidade quando imagens criadas a partir de um sistema operacional linux não funcionarão nativamente em um sistema operacional *windows* .

Capítulo 3

3. CEFET-Connections: a rede social em microserviços

O *CEFET-Connections* propõe uma rede social para os alunos do *CEFET*, onde é possível que os alunos façam posts, comentários e curtam os posts. Cada post publicado pelo aluno será visto por todos os outros em uma timeline que é um agrupados de posts, onde os mesmos são ordenados pela sua data de publicação. Diferentemente do *Facebook* em que podem ser feitos posts sem uma categorização, no *CEFET-Connections* os posts devem ser criados para determinados assuntos existentes na aplicação, são eles: Disciplina, Curso e Melhorias. Na timeline é possível filtrar os posts pelos assuntos, afim de melhorar a busca de posts para o aluno, mostrando apenas os relacionados de determinados assuntos. O design da aplicação se inspira no *Facebook* em que possui a timeline como centro da aplicação e a rolagem para que carregue mais posts. Sua estrutura gráfica é bem simplificada facilitando a compreensão e usabilidade pelos alunos. Este capítulo apresenta arquitetura proposta, a forma de armazenamentos e envio dos dados e a maneira encontrada para se fazer deploy utilizando containers em uma arquitetura de microserviços.

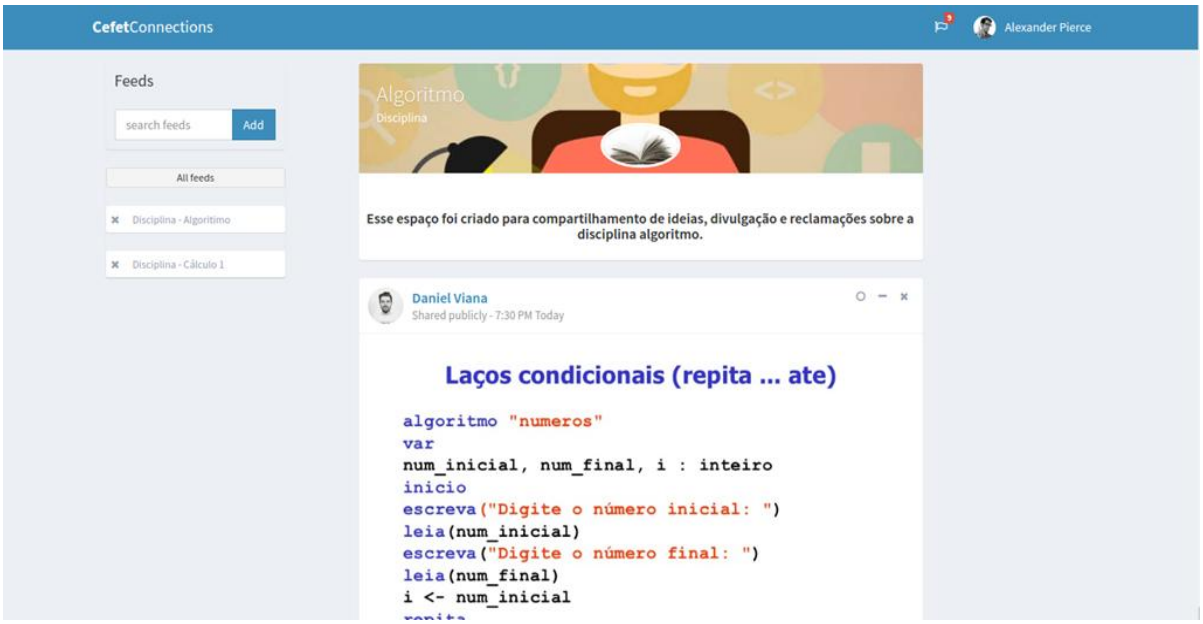

Figura 8 – Página principal do *CEFET-Connections*.

3.1 Arquitetura

A arquitetura do *CEFET-Connections* foi construída utilizando microserviços na qual foi dividida em infraestrutura e negócio. O negócio foi divido em pequenos serviços em que houve a necessidade de se definir suas granularidades conforme FOWLER (2014) define de *bounded context*, ou seja, saber onde começa e termina cada parte do negócio para então dividir em pequenos serviços. Os serviços foram divididos em apenas três: *Aluno*, *Post* e *Timeline*.

O serviço *Aluno* contempla apenas o contexto do aluno, existindo uma aplicação *REST* com o recurso aluno podendo ser criado, atualizado e removido. Para esse serviço existe uma base de dados onde é apenas acessada por serviços externos via requisições *HTTP* ao recurso *REST* Aluno. O banco de dados utilizado para todos os serviços é um *NoSQL* conhecido como *MongoDB*. A seguir a tabela representando o recurso aluno e seus respectivos *endpoints* com seus *payloads* em *json*:

Endpoints	Método *HTTP*	*Payload*
/v1/alunos	*POST*	{ "curso": "string", "email": "string", "id": "string", "nome": "string", "password": "string", "urlImagem": "string" }
/v1/alunos/{id}	*PUT*	{ "curso": "string", "email": "string", "id": "string", "nome": "string", "password": "string", "urlImagem": "string" }
/v1/alunos/{id}	*DELETE*	**Não contempla**

O serviço *Post* contempla o contexto de um post, ou seja, tanto um comentário ou curtida também fazem parte de um post, pois não é possível que os mesmos existam sem um post, por esse motivo eles estão agrupados no mesmo *bounded context* do post. Esse serviço também é uma aplicação *REST* com o recurso post podendo ser criado, atualizado, removido, curtido e seus comentários criados e removidos. A seguir a tabela representando o recurso post e seus *endpoints* com seus *payloads* em *json*:

Endpoints	Método *HTTP*	*Payload*
/v1/posts	*POST*	{ "alunoId": "string", "descricao": "string", "id": "string", "titulo": "string", "urlImagem": "string" }
/v1/posts/{id}	*PUT*	**Não contempla**
/v1/posts/{id}	*DELETE*	**Não contempla**
/v1/posts/{id}/comentários	*PUT*	{ "id": "string", "texto": "string", "usuarioId": "string", "usuarioNome": "string" }

/v1/posts/{id}/comentarios/{id}	*DELETE*	**Não contempla**
/v1/posts/{id}/curtir	*PUT*	{ "id": "string"}
/v1/posts/{id}/descurtir	DELETE	**Não contempla**

O serviço *Timeline* é diferente dos descritos anteriormente, pois ele não é acessível via *REST* e não possui uma base de dados que seja apenas dele, pois ele é apenas um componente utilizado para atualização dos dados que serão apresentados para a camada da visão. Este será mais detalhado na seção 3.1.1.

O contexto da infraestrutura foi divido em quatro serviços: Gateway, Discovery Server, Config Server e Authorization Server. A Figura 8 a seguir descreve a arquitetura da infraestrutura juntamente com a parte de negocio.

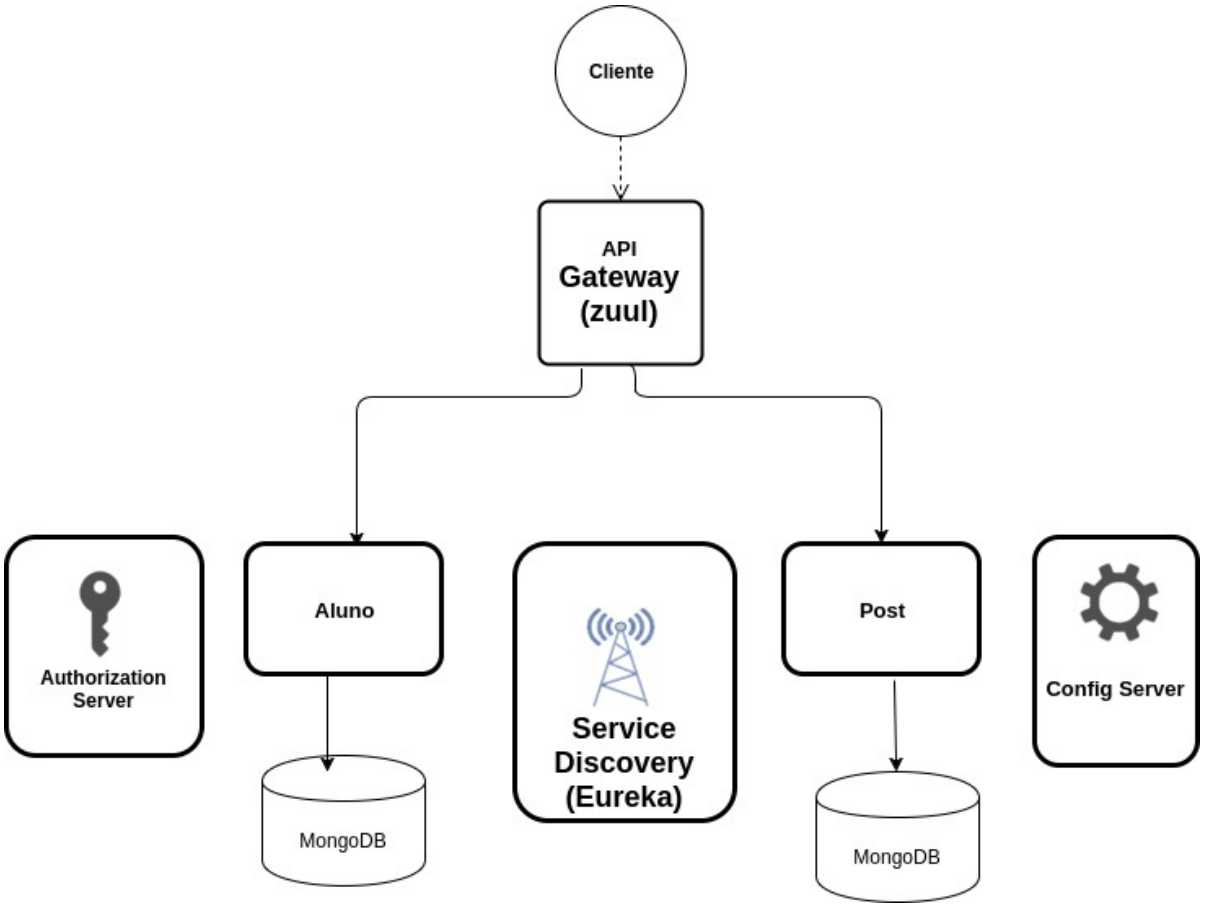

Figura 9 – Arquitetura

Os serviços de infraestrutura foram criados para facilitar o desenvolvimento de aplicações utilizando microserviços, resolvendo problemas de centralização de configuração, localização dos diferentes serviços e o balanceamento de carga para um grande número de requisições. Tanto os serviços de infraestrutura quanto os de negócios foram implementados utilizando a linguagem *Java* na versão 8 e os *frameworks Spring Cloud e Spring Boot* para

facilitar o desenvolvimento dos serviços em nuvem. Todos os serviços de infraestrutura serão descritos a seguir*:*

- ***Config Server****:* O Config Server utiliza o framework *Spring Cloud Config* que fornece suporte ao servidor e ao cliente para a configuração externalizada em um sistema distribuído. Com o Config Server você tem um lugar central para gerenciar propriedades externas para aplicativos em todos os ambientes, de modo que se encaixam muito bem com as aplicações Spring, mas podem ser usadas com qualquer aplicativo e executado em qualquer linguagem. A seguir a Figura 9 mostra a divisão das pastas no *Config Server*.

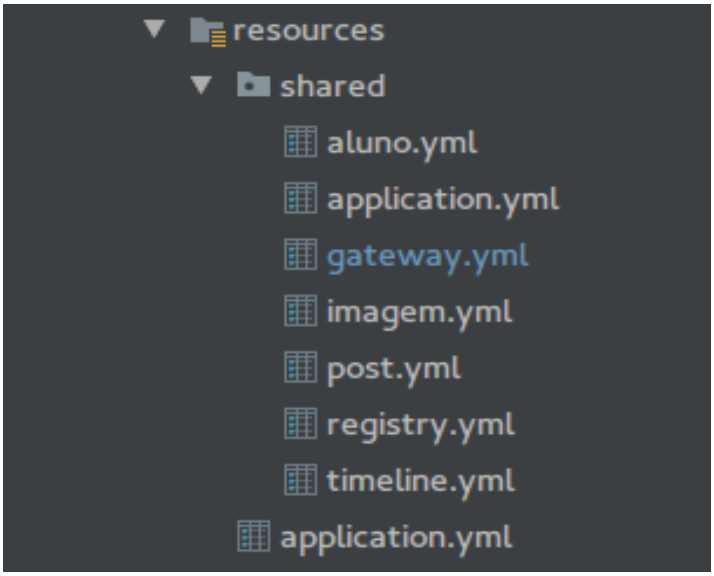

Figura 10 – Arquivos de configurações do *Config Server.*

Com essas configurações definidas, quando um determinado serviço for executado, em seu *bootstrap*, a primeira coisa é buscar as configurações de url de acesso ao banco de dados, parte do seu servidor *HTTP* no *Config Server*. Para que todo o serviço consiga ser executado, é necessário que o *Config Server* esteja rodando.

- ***Gateway (Zuul):*** Como pode ser visto na Figura 8, existem dois serviços principais que expõem *API* externa ao cliente. Com o número crescente de alunos e novas ideias para a plataforma, esse número pode crescer muito rapidamente, bem como toda a complexidade do sistema. Na verdade, centenas de serviços podem estar envolvidos na renderização de uma página web complexa. Em teoria, um cliente poderia fazer pedidos para cada um dos microserviços diretamente. Mas, obviamente, há desafios e limitações com esta opção, como a necessidade de conhecer todos os endpoints endereços, executar solicitação http para cada serviço com informação separadamente, mesclar o resultado do lado do cliente. Normalmente,

uma abordagem muito melhor é usar uma *API Gateway*. É um único ponto de entrada no sistema, usado para lidar com solicitações encaminhando-os para o serviço de backend apropriado ou invocando vários serviços de backend e agregando os resultados. A *Netflix* colocou o seu serviço de *Gateway* opensource chamado *Zuul* conforme descreve os detalhes em seu blog NETFLIX (2017). O *Zuul* foi utilizado para fazer com que os pedidos dos clientes sejam redirecionados para os serviços expostos corretamente e a autenticação dos clientes. A Figura 10 a seguir detalha o arquivo de configuração para as chamadas ao serviço aluno.

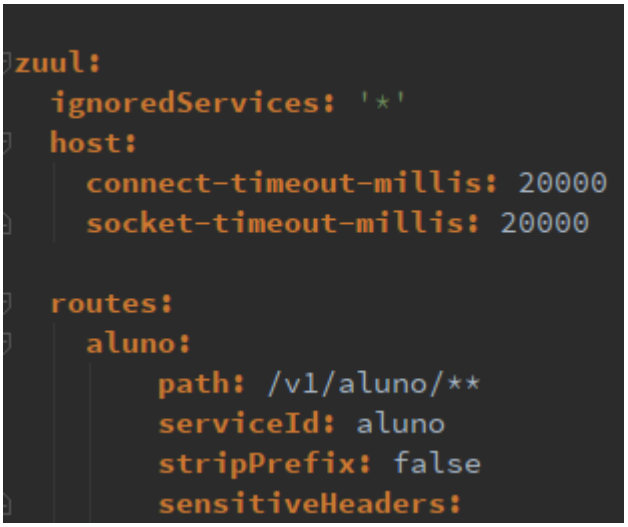

Figura 11 – Exemplo do mapeamento das rotas.

No arquivo *yaml* é possível notar que em *routes:* existe a rota aluno e o seu caminho é igual ao definido na seção 3.1. Com o uso do ' ** ' indica que qualquer requisição do cliente com o path *' /v1/aluno'* será redirecionado para o serviço aluno (*serviceId*).

- ***Service Discovery (Eureka):*** Outro padrão de arquitetura comumente conhecido é o *Service Discovery*. Ele permite a detecção automática de locais de rede para instâncias de serviço, que podem ter endereços dinâmicos atribuídos devido a escala automática, falhas e atualizações. A seguir a Figura 11 mostra o funcionamento do *Service Discovery.*

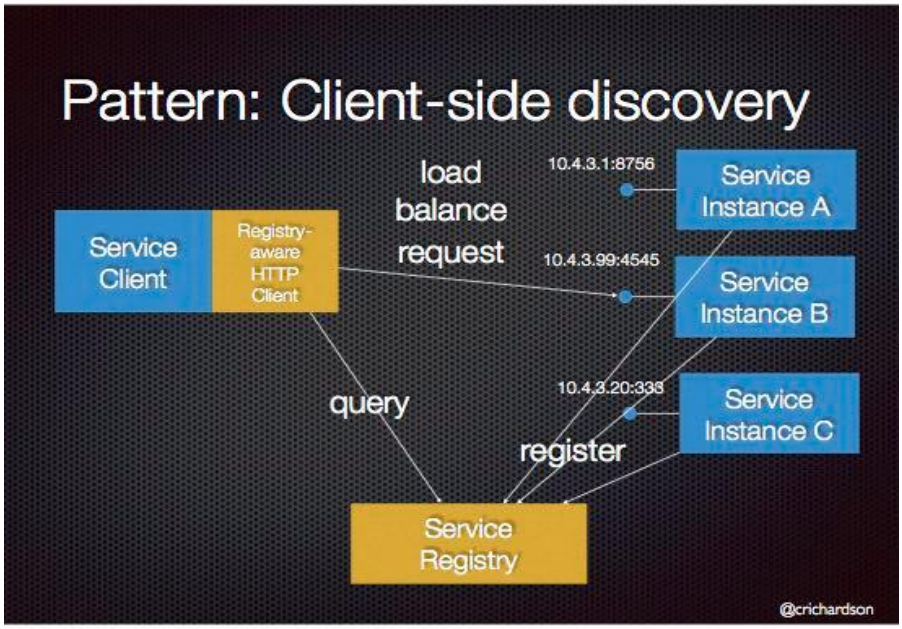

Figura 12 - Service Discovery RICHARDSON (2017)

Na arquitetura está sendo usado um *Service Discovery Eureka* desenvolvido pela *Netflix* que assim como o *Zuul* também é *opensource. Eureka* é um bom *Service Discovery* quando é necessário que o cliente saiba os locais das instâncias dos serviços, o que é necessário para o *Zuul em* seu roteamento e, além disso, ainda é feito o balanceamento de carga para os diferentes serviços.

- ***Authorization Service :*** A parte de autenticação ficou extraida nesse serviço que basicamente gerencia os tokens de acesso as aplicações. Os tokens são gerados utilizando *JWT* que é um padrão aberto para envio de informações de forma segura entre o cliente e servidor. A informação pode ser verificada e confiável porque está assinada digitalmente, usando um par de chaves públicas / privadas. Um exemplo do token a seguir: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ. O servidor consegue tanto gerar o token quanto descriptografar, onde suas informações são em formato de *JSON*, pode-se notar que o token possui três informações separadas por ponto. A seguir um exemplo do *token* descriptografado*:*

{
"alg": "HS256",
"typ": "JWT"
}

Essa primeira parte é definida o tipo de algoritmo utilizado para a criptografia.

```
{
 "id": "1234567890",
 "nome": "John Doe",
 "aluno": true
}
```

A segunda parte são os dados trafegados sobre o cliente.

```
HMACSHA256(
        base64UrlEncode(header) + "." +  base64UrlEncode(payload),
        palavra secreta
)
```

E a terceira parte a assinatura com a chave privada 'palavra secreta'.

3.1.1 Desempenho

O desempenho na *web* é crucial para a vida da aplicação e quando se cita rede social essa preocupação aumenta se analisarmos o *Facebook* ou *Twitter,* que possuem milhões e até bilhões de usuários fazendo milhões de requisições por segundo. Já se foi o tempo em que as aplicações desse porte possuíam apenas dez usuários, a maioria das aplicações hoje nascem pensando em funcionar bem com dez a dez mil usuários acessando simultaneamente a aplicação, tornando uma tarefa complexa criar um modelo que atenda a essa necessidade.

Pensando nesse cenário, para a criação do *CEFET-Connections* pensamos em como resolver esse tipo de problema. A primeira tentativa foi a mais simples que era escalar a aplicação em vários serviços, pois se o problema é ter que tratar tantas requisições, bastaria aumentar o número de serviços no servidor, porém como analisado, aumentar o número de serviços não garante a performance, pois ainda existia o problema de consumo no banco de dados que ficaria mais lenta pelo maior número de consultas. Por mais que estivéssemos usando o *Eureka* visto na seção 3.1 que possui um balanceador de carga, ainda sim existia o problema de conseguir escalar o banco de dados. Além disso, apenas alguns dados são consultados e alterados no banco de dados e o que o usuário vê na tela, provavelmente pode estar obsoleto. Seguindo essa linha de pensamento, usamos uma estratégia de arquitetura conhecida como *CQRS* descrita por Fowler no texto a seguir (FOWLER , Tradução):

> *CQRS significa Segregação de Responsabilidade de Comando. É um padrão que ouvi pela primeira vez descrito por Greg Young. No seu coração é a noção de que você pode usar um modelo diferente para atualizar informações do que o modelo usado para ler informações. Para*

> *algumas situações, esta separação pode ser valiosa, mas tenha cuidado de que, para a maioria dos sistemas, o CQRS acrescenta uma complexidade arriscada. (FOWLER, 2011)*

Por mais que *Fowler* descreva que exista uma complexidade na criação desse tipo de arquitetura, ao análisarmos o cenário da rede social , vimos que faria sentido utilizar a estratégia descrita. A seguir a Figura 12 define a arquitetura com *CQRS:*

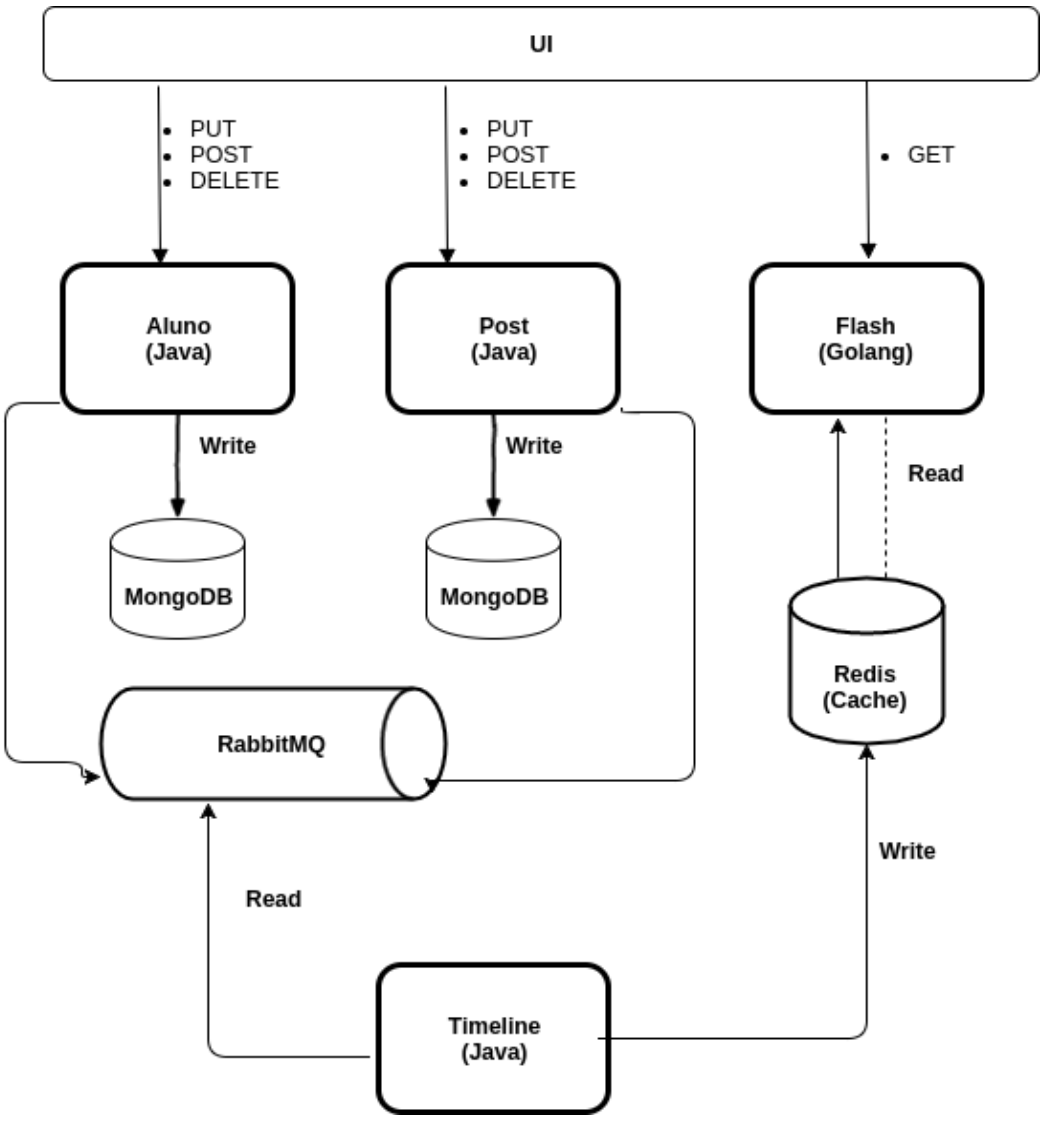

Figura 13 – Arquitetura CQRS *CEFET-Connections*

Na Figura 12 foi definida a arquitetura utilizando CQRS. Pode-se observar que basicamente dividimos o sistema em escrita(*POST, PUT, DELETE*) e leitura (*GET*), ou seja, tudo o que for alteração dos dados tanto no serviço *Aluno* quanto no serviço *Post* será escrito nas suas respectivas base de dados, que nesse caso é o *MongoDB*, e as consultas serão feitas através da aplicação *Flash* diretamente no *Redis.*

A *Timeline* é o serviço responsável por atualizar os dados da consulta, pois com a divisão de um banco de dados para escrita e outro para leitura, os dados escritos no banco de escrita devem ser replicados para o banco de leitura, que nesse caso o banco de leitura estará utilizando um cache distribuído, conhecido como *Redis*, que é um armazenador de estrutura de dados de chave-valor open source e na memória e por conta de sua velocidade e fácil uso, utilizamos para armazenar os dados de leitura.

Os serviços *Aluno* e *Post* publicam mensagens quando existem escritas nas bases para o *RabbitMQ*, que é um agente de mensagens open source. Essas mensagens são recebidas pela Timeline que atualiza o *Redis*. Toda mensagem publicada está no formato *JSON* para facilitar no momento da consulta no Redis.

Toda consulta existente no sistema, seja ela para buscar dados do aluno, post e timeline, são feitas diretamente no serviço *Flash que é uma aplicação escrita em Golang* uma linguagem *open source* criada em 2009 pelo *Google,* por conta de sua velocidade e de uma boa API de leitura para o Redis foi utilizada para aumentar o desempenho na leitura dos dados.

3.2 Armazenamentos e envio dos dados

A partir da compreensão da interface com os serviços Aluno e Post, é apresentado na tabela a seguir o modelo dos documentos que estão armazenados em um formato json, que é a estrutura utilizada pelo MongoDB.

Aluno	**Post**

alunos = [
  {
  "curso": "string",
  "email": "string",
  "id": "string",
  "nome": "string",
  "password": "string",
  "urlImagem": "string"
  }
]
	Posts =[
	  {
	  "alunoId": "string",
	  "descricao": "string",
	  "id": "string",
	  "titulo": "string",
	  "urlImagem": "string"
	  “comentarios”:[{
	  "id": "string",
	  "texto": "string",
	  "usuarioId": "string",
	  "usuarioNome": "string"
	  }],
	  “curtidas” : {
	  “alunosId”:[{“id”:”string”}]
	  }
	]

Além das estruturas anteriores, todas as alterações do banco são enviadas para a timeline através de eventos ocorridos dentro da aplicação. Foi necessário o armazenamento das mensagens enviadas a fim de conseguir identificar no sistema quais foram os eventos ocorridos para uma determinada entidade. A seguir a tabela representando a estrutura da mensagem armazenada no MongoDB:

Evento Aluno

```
eventos = [{
  id:”string”,
 “aggregateId”: “string”,
 “status”:”string”,
“aluno”:{
    "curso": "string",
    "email": "string",
    "id": "string",
    "nome": "string",
    "password": "string",
    "urlImagem": "string"
},
“registrado”: date
}]
```

Evento Post

```
eventos = [{
  id:”string”,
 “aggregateId”: “string”,
 “status”:”string”,
 “post”:{
  "alunoId": "string",
     "descricao": "string",
     "id": "string",
     "titulo": "string",
     "urlImagem": "string"
     “comentarios”:[{
          "id": "string",
          "texto": "string",
          "usuarioId": "string",
          "usuarioNome": "string"
      }],
     “curtidas” : {
        “alunosId”:[{“id”:”string”}]
     }
},
“registrado”: date
}]
```

O atributo **aggregateId** identificado da tabela acima é usado para conseguir saber toda a vida útil de um determinado objeto persistido no banco, pois o aggregateId é o dado representado pelo id da entidade agregada como exemplo, no contexto dos eventos de um post é o próprio post, ou seja, a partir do id do post eu consigo saber quais eventos ocorreram para aquele post desde sua criação até sua remoção no banco de dados. A ideia de persistir os eventos existentes em uma arquitetura utilizando *CQRS* (descrita na seção 3.1.1) foi descrita por Fowler como *Event Sourcing* no texto a seguir(FOWLER, Tradução):

> A ideia fundamental da Event Sourcing é a de garantir que todas as mudanças no estado de um aplicativo sejam capturadas em um objeto de evento e que esses objetos de evento sejam eles mesmos armazenados na sequência em que foram aplicados pela mesma vida que o próprio estado da aplicação. *(FOWLER, 2005)*

Esse tipo de abordagem é interessante, pois às vezes podemos consultar um objeto para descobrir o estado atual, e isso responde muitas perguntas. No entanto, há momentos em que não queremos apenas ver onde estamos também queremos saber como chegamos lá.

3.3 Deploy

Uma vez cumprida às tarefas para implementação das aplicações, o próximo passo foi iniciar a subida das aplicações para a nuvem. O cloud utilizado para hospedar a aplicação foi a Digital Ocean. Para facilitar os deploys de diferentes tecnologias utilizadas e diferentes aplicações, surgiu a necessidade de colocar os serviços para rodarem em containers que estão descritos no capítulo 2, seção 2.3. Essa necessidade visa uma maneira mais ágil de organizar o deploy em produção dos serviços Post e Aluno e de outros novos serviços que possam surgir.

Seguindo essa abordagem, basta fazer uma única configuração, uma única vez, para garantir o mesmo funcionamento nas máquinas de desenvolvimento, homologação e produção. Isso permite que cada time de desenvolvimento responsável por um serviço seja também responsável por decidir e gerenciar quais versões de bibliotecas e dependências seus projetos vão utilizar.

Para criação dos containers foi utilizado Docker na versão 2 que está mais detalhada no capítulo 2, seção 2.3.1. Com isso criamos um Dockerfile para cada serviço, o Dockerfile como descrito no capítulo 2, serve para criarmos uma imagem Docker que represente as configurações desejadas para o container. A seguir, na Figura 13, está representado o arquivo Dockerfile do serviço Config Server.

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD configserver-*.jar /configserver.jar
ENV JAVA_OPTS=""
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar
/configserver.jar" ]
EXPOSE 8888
```

Figura 14 – Dockerfile do Config Server

Cada linha do arquivo Dockerfile executa um comando para a construção da Imagem Docker, que é descrito a seguir:

- **FROM:** A instrução *FROM* inicializa um novo estágio de compilação e define a Imagem Base para instruções subsequentes. Como tal, um Dockerfile válido deve começar com uma instrução FROM. A imagem pode ser qualquer imagem válida - é especialmente

fácil começar puxando uma imagem dos repositórios públicos conforme foi utilizada a imagem openjdk. O repositório publico mais conhecido é o DockerHub que será detalhado o seu uso para a geração de nossas imagens.

- **VOLUME:** A instrução *VOLUME* deve ser usada para expor qualquer área de armazenamento de banco de dados, armazenamento de configuração ou arquivos / pastas criadas pelo seu container docker, com esse comando é possível compartilhar partes de seu container com o sistema operacional host.
- **ENV:** A instrução *ENV* é útil para fornecer variáveis de ambiente necessárias para seus serviços.
- **ENTRYPOINT:** O melhor uso para *ENTRYPOINT* é configurar o comando principal da imagem, permitindo que essa imagem seja executada a partir do comando definido, nesse exemplo executamos o configserver.jar.
- **ADD:** É a extração automática de arquivos para a imagem, no exemplo está sendo copiado o arquivo local configserver-*.jar para o container no caminho /configserver.jar.
- **EXPOSE:** A instrução *EXPOSE* indica as portas nas quais um container irá ouvir conexões. No caso do configserver a porta 8888.

Para gerar a imagem definida no *Dockerfile*, foi utilizado um pluggin do *maven* que abstrai a necessidade de algumas informações passadas em linha de comando para o Docker. Após a geração das imagens foi colocado no repositório público *DockerHub* as imagens dos serviços gerados para que a partir do servidor na nuvem fosse possível fazer os downloads e execuções das imagens.

Com as imagens no repositório foi possível criar o arquivo docker-compose, conforme descrito no capítulo 2, seção 3.2.1 que é possível inicializar mais de um container em um único comando. A seguir o exemplo do arquivo docker-compose:

```
version: "2"

services:
  post:
    image: danielviana/cefetconnections:post-1.0.0
    container_name: "post"
    depends_on:
      - eureka
    links:
      - mongodb
      - redis
      - config

  aluno:
    image: danielviana/cefetconnections:aluno-1.0.0
    container_name: "aluno"
    depends_on:
      - eureka
    links:
      - mongodb
      - config
```

Figura 15 – Arquivo docker-compose

Com o arquivo docker-compose gerado é possível executar em linha de comando o docker, basta executar o comando ' **docker-compose up -d** ' para inicializar todos os serviços definidos no arquivo compose.

Capítulo 4

4. Métricas retiradas das API's:

Para que fosse possível verificar os dados de desempenho das API's REST, utilizou-se repetições de chamadas HTTP para simular o acesso dos usuários no sistema. Para se gerar as repetições de chamadas, utilizou-se um framework implementado em Golang conhecido como Vegeta (https://github.com/tsenart/vegeta). A seguir está sendo gerando 10 requests por segundo durante 30 segundos para simular as consultas na timeline (API em Golang).

echo "GET http://104.236.238.182:8088/view/v1/post?since=0&&until=-1" | vegeta attack -rate=10 -duration=30s | vegeta report

Dados capturados a seguir:

Requests [total, rate] 300, 10.03

Duration [total, attack, wait] 30.024685479s, 29.899999709s, 124.68577ms

Latencies [mean, 50, 95, 99, max] 126.029058ms, 123.686679ms, 136.232619ms, 159.263198ms, 251.69922ms

Bytes In [total, mean] 85200, 284.00

Bytes Out [total, mean] 0, 0.00

Success [ratio] 100.00%

Status Codes [code:count] 200:300

A partir dos dados capturados é possível tirar algumas conclusões de desempenho da API. Nos dados é possível verifica os campos Requests [total, rate] 300, 10.03, isso significa que foram feitas trezentas chamadas a API durante dez segundos e todas retornadas com sucesso, ou seja, retornando duzentos no seu Status Codes, que indica que em uma chamada HTTP foi realizada com sucesso, esse dado pode ser verificado também em *Status Codes [code:count] 200:300* .

Além dos dados sobre o número de requests recebidos pela API também é possível tirar dados sobre sua latência que é uma medida fundamental de desempenho da rede, pois

mede a quantidade de tempo entre o início de uma ação e seu término e a avaliação de seu rendimento é feita com base no número total de ações que ocorrem em um determinado período de tempo. Para esse caso de estudo a API teve uma latência de *126.029058ms*, conforme pode se verificar nos dados gerados anteriormente.

A fim de simular um alto acesso a aplicação, foi estipulada uma tentativa de dez mil requests por segundo para simular a carga durante 30 segundos.

echo "GET http://104.236.238.182:8088/view/v1/post?since=0&&until=-1" | vegeta attack -rate=1000 -duration=30s | vegeta report

Dados capturados a seguir:

Requests [total, rate] 300000, 9998.08

Duration [total, attack, wait] 31.07210062s, 30.005773349s, 1.066327271s

Latencies [mean, 50, 95, 99, max] 48.93901ms, 128.522µs, 290.392226ms, 719.278214ms, 5.691844501s

Bytes In [total, mean] 11649396, 38.83

Bytes Out [total, mean] 0, 0.00

Success [ratio] 13.67%

Status Codes [code:count] 0:258981 200:41019

É possível analisar que mesmo com o aumentando do número de requests por segundo, a aplicação manteve-se respondendo todos os requests e sua latência em média foi reduzida para 48.93901ms.

Além do desempenho dos *requests* é possível verificar na imagem abaixo que a utilização da linguagem *Golang* também impacta diretamente o uso dos recursos no servidor:

Figura 16 - Uso de recursos usando *Golang*.

Figura 17 – Uso de recursos usando *Java*.

Foi verificado que com a linguagem Golang é utilizado 0.2% da memória do servidor e 0.0% da CPU em contra partida a linguagem Java utiliza 14.5% da memória e 8.8 % do uso da CPU. Conclui-se que o desenvolvimento com Golang é mais interessante, pois possui um menor custo para manter a aplicação no servidor.

A Netflix disponibilizou como open source o seu"Chaos Monkey", um software que intencionalmente derruba servidores como forma de testar a tolerância a falhas de um ambiente em nuvem. Porém para o nossos teste vamos fazer manualmente a parada dos serviços em produção. Deixando apenas o Redis e Flash(aplicação de consulta escrita em Golang), afim de, verificar se nossa aplicação está tolerante a falhas pois mesmo que o serviço de postagem pare é necessário que o de consulta continue a funcionar.

A seguir os containers rodando:

Figura 18 – serviços existentes.

Número de containers que ficaram rodando:

Figura 19 – serviços rodando.

Aplicando o mesmo framework para validar a consulta podemos capturar os seguintes dados:

echo "GET http://104.236.238.182:8088/view/v1/post?since=0&&until=-1" | vegeta attack -rate=10 -duration=30s | vegeta report

Requests [total, rate] 300, 10.03

Duration [total, attack, wait] 30.024685479s, 29.899999709s, 124.68577ms

Latencies [mean, 50, 95, 99, max] 126.029058ms, 123.686679ms, 136.232619ms, 159.263198ms, 251.69922ms

Bytes In [total, mean] 85200, 284.00

Bytes Out [total, mean] 0, 0.00

Success [ratio] 100.00%

Status Codes [code:count] 200:300

Como pode ser visto, a aplicação de consulta continua a funcionar do mesmo jeito dos testes de cargas que anteriormente. Com isso, podemos concluir que sua independência fica clara ao parar todos os sistemas.

Capítulo 5

5. Conclusão

O objetivo deste trabalho foi implementar uma arquitetura com funcionalidades comuns em aplicações baseadas em microserviços a fim de explorar as vantagens, desvantagens e dificuldades que este modelo arquitetônico oferece.

A ideia para implementação foi uma rede social composta por um *backend* responsável por servir as funcionalidades consumidas por uma aplicação *web*. Na totalidade, foram criados sete microserviços sendo três deles apenas necessários para a infraestrutura da arquitetura.

A fim de facilitar as configurações e padronizações nas gerações dos *deploys* dos serviços, foi utilizado um gerador de *containers* conhecido como *Docker.* Onde foram gerados *containers e imagens Dockers* para todos os serviços criados e suas *imagens* foram publicadas em um *repositório público* e remoto conhecido como *DockerHub,* para facilitar o *deploy* na *nuvem.* Tanto para o banco de dados *MongoDB* quanto para o *cache Redis,* foram utilizados *containers* originados das imagens do *repositório público DockerHub*.

O maior desafio durante o desenvolvimento foi o estudo para solucionar alguns problemas que a arquitetura com microserviços proporciona, como a necessidade do uso de um *Gateway* para fornecer um único ponto de entrada e abstrair as chamadas para os diversos serviços existentes. O uso de um *Service Discovery* para que possamos saber onde se localizam todos os serviços, pois quando estamos em um contexto na nuvem podemos ter serviços em diferentes *IPs* e *Portas,* e esse mapeamento é feito pelo *Service Discovery*, que recebe o *IP* e *Porta* de cada serviço ao ser executado e a centralização das configurações em um único serviço como o *Config Server*. No entanto, dentre todos os padrões citados a implementação de uma arquitetura com CQRS foi a parte mais trabalhosa, pois, fazer com que existam duas bases, uma para leitura e outra para escrita, torna a sincronização dos dados algo muito complexo. Contudo, essa abordagem reduz muitos problemas relacionados ao desempenho das aplicações, pois, diminuímos a concorrência no banco evitando *locks* nos dados. As consultas se tornam mais rápidas pelos dados estarem desnormalizados no banco de leitura evitando *joins* com várias tabelas. Além disso, os dados para busca estão em um *cache* de chave e valor em memória, o que torna ainda mais rápida a busca dos dados. Outra evidência que podemos destacar é característica da heterogeneidade tecnológica da arquitetura com microserviços, pois quase todos os serviços estão implementados em Java porém, para

melhorar a performance da busca, foi implementado uma aplicação usando a linguagem de programação *Golang* que é conhecida por ser leve e performática.

Conclui-se que a utilização de uma arquitetura de microserviços é uma forma viável para construção de aplicações. Ela oferece diversas vantagens em relação a outras abordagens como as aplicações monolíticas. No entanto, podemos afirmar que esse tipo de arquitetura torna o desenvolvimento das aplicações muito mais complexo.

Trabalhos futuros

A partir do trabalho produzido, podem-se destacar como possíveis trabalhos futuros:

- A criação da parte web da aplicação com as chamadas ao backend criado, para que se possa concluir, de fato, a rede social *CEFET-Connections*.
- Incluir um pipeline para facilitar o deploy, utilizando Integração continua ou Deploy Continuo.
- Adicionar técnicas para verificações de falhas entre os serviços, como *Circuit Break.*
- Utilização de um orquestrador de containers como *Docker Swarm* ou *Kurbernets*, tornando obsoleto o uso da arquitetura da *Netflix* (*Eureka* e *EdgeServer (Zuul)*) pois, os mesmos já possuí todas as facilidades implementadas nos serviços de infraestrutura.

REFERÊNCIAS BIBLIOGRÁFICAS

AGARWAL, Kavita. **A Study of Virtualization Overheads.** 2015. p. 1-45. Master of Science –Stony Brook University**.** Disponível em: http://www.oscar.cs.stonybrook.edu/papers/files/KavitaAgarwalMSThesisSubmission.pdf

DOCKER. **Docker platform.** Disponível em: https://docs.docker.com/engine/docker-overview/ Acessado em Agosto de 2017.

FIELDING,Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** 2000. p. 1-162. Doutorado- University California, Irvine.

FOWLER, Martin; LEWIS, James. **Microservices**. Disponível em: http://martinfowler.com/articles/microservices.html Acessado em Setembro de 2015.

FOWLER, Martin. **CQRS**. 14 de julho de 2011. Disponível em: https://martinfowler.com/bliki/CQRS.html Acessado em Agosto de 2017.

FOWLER, Martin. **BoundedContext**. 15 de janeiro de 2014. Disponível em https://martinfowler.com/bliki/BoundedContext.html Acessado em Agosto de 2017.

FOWLER, Martin. **Event Sourcing**. 12 de dezembro de 2005. Disponível em https://martinfowler.com/bliki/BoundedContext.html Acessado em Agosto de 2017.

HAO HE. **What is Service-Oriented Architecture**. 30 de Setembro de 2003. Disponível em: http://uic.edu.hk/~spjeong/ete/xml_what_is_service_oriented_architecture_sep2003.pdf Acessado em Agosto de 2015.

JUNIOR , Jonas B. de M; CANDEIAS, Ana Lucia B. **Interoperabilidade de SIG através de Web Services**. Disponível em: https://www.ufpe.br/cgtg/ISIMGEO/CD/html/cartografia%20e%20sig/Artigos/C24.pdf Acessado em Agosto de 2015.

MORAES, Juliano; BREDA, Marcos; GIL, Paulo; MEDAGLIA, Refael. **Web Services**. Disponível em: http://www.inf.pucrs.br/~gustavo/disciplinas/sd/material/Artigo_WebServices_Conceitual.pdf Acessado em Agosto de 2015.

NAMIOT, Dmitry; SNEPS-SNEPPE, Manfred. **On Micro-services Architecture**. International Journal of Open Information Technologies, vol. 2, no. 9, 2014.

NETFLIX. **Zuul gateway**. Disponível em : https://medium.com/netflix-techblog/announcing-zuul-edge-service-in-the-cloud-ab3af5be08ee Acessado em Agosto de 2017.

RICHARDSON, Chris. **cliente-side service discovery**. Disponível em: http://microservices.io/patterns/client-side-discovery.html Acessado em Agosto de 2017.