Rodolfo Gomes Peixoto

Arquiteturas de Microsserviços e Camadas para Softwares Modernos: Uma Análise Comparativa

Campos dos Goytacazes, RJ

16 de dezembro de 2019

Rodolfo Gomes Peixoto

Arquiteturas de Microsserviços e Camadas para Softwares Modernos: Uma Análise Comparativa

Trabalho de Monografia apresentado ao Curso de Graduação em Ciência da Computação da Universidade Estadual do Norte Fluminense Darcy Ribeiro como requisito para a obtenção do título de Bacharel em Ciência da Computação, sob orientação da Profª. Annabell Del Real Tamariz.

Universidade Estadual do Norte Fluminense Darcy Ribeyro – UENF

Centro de Ciência e Tecnologia – CCT

Laboratório de Ciências Matemáticas – LCMAT

Curso de Ciência da Computação

Orientador: Profª. Annabell Del Real Tamariz

Campos dos Goytacazes, RJ

16 de dezembro de 2019

Rodolfo Gomes Peixoto

Arquiteturas de Microsserviços e Camadas para Softwares Modernos: Uma Análise Comparativa/ Rodolfo Gomes Peixoto. – Campos dos Goytacazes, RJ, 16 de dezembro de 2019-

51 p. : il. (algumas color.) ; 30 cm.

Orientador: Profª. Annabell Del Real Tamariz

Monografia (Bacharelado) – UENF-CCT-LCMAT-Ciência da Computação, 16 de dezembro de 2019.

1. Engenharia de Requisitos. 2. Orientação a Aspectos. 3. Engenharia da Segurança. 4. Confidencialidade. 5. Domínios de Requisitos. 6. Metodologia

CDU 004.41 : 004.4'2 :

Rodolfo Gomes Peixoto

Arquiteturas de Microsserviços e Camadas para Softwares Modernos: Uma Análise Comparativa

Trabalho de Monografia apresentado ao Curso de Graduação em Ciência da Computação da Universidade Estadual do Norte Fluminense Darcy Ribeiro como requisito para a obtenção do título de Bacharel em Ciência da Computação, sob orientação da Profª. Annabell Del Real Tamariz.

Trabalho Aprovado.

Profª. Annabell Del Real Tamariz
Orientadora

Prof. Dr. Leonard Barreto Moreira
UFF

Prof. Msc. Rodrigo Manhães
DIC-UENF

Campos dos Goytacazes, RJ 16 de dezembro de 2019

Dedico este trabalho a minha mãe Simone Aparecida Gomes e ao meu pai Antônio Francisco Peixoto que mesmo não tendo frequentado um nível superior e no caso do meu pai ter completado só a terceira série, ambos pensaram em educação. Dedico também a Carla Bravo que quando cheguei em Campos dos Goytacazes com um sentimento de precisar concluir a Universidade, mas sem dinheiro muitas das vezes para comer, me estendeu o braço e me ajudou, não só financeiramente, mas psicologicamente. Dedico aos amigos e professores que desejaram e de certa forma contribuíram em transformar um aluno totalmente ignorante pela falta de conhecimento, em um apaixonado por toda forma de informação.

Agradecimentos

Agradeço a minha família, por todo empenho em ajudar no que podia. Em especial a minha mãe, Simone Aparecida Gomes que apertou tudo para me ajudar, meu pai Antônio Franscico Peixoto que morreu trabalhando para levar aos seus filhos uma educação melhor.

Aos meus amigos Pedro Rodrigues e Amanda Gregório que foram um suporte durante o curso, passamos por momentos muito bons e ruins juntos e foi crucial para chegarmos os três nesse momento de apresentação de monografia. Um agradecimento forte ao Pedro Rodrigues e a Luana Rodrigues que me receberam em sua casa, quando não tinha dinheiro para alugar uma casa e ambos me receberam super bem para que eu pudesse fazer as matérias daqueles dois períodos.

Agradeço a todos os mestres que foram especiais em minha vida como a Prof. Dra. Annabell Del Real Tamariz que foi crucial para que eu tenha chegado até aqui, um ser humano ímpar e que não media esforços para ajudar os alunos, eu tenho uma gratidão eterna e que jamais desejo perder contato. Agradecimento eterno ao Prof. Dr. Geraldo de Oliveira Filho que foi incrível, aprendi a gostar de matemática e isso foi fundamental para o fim dessa caminhada. Ao Mestre Rodrigo Manhães pelos momentos de aprendizado, quando ia sanar alguma dúvida e aprendia algum conceito novo. Ao Prof. Dr. Leonard Barreto que a todo o momento que nos via, incentivava a concluir e não desistir. Há muitos mestres que foram importantes e que ficaram marcados para sempre em minha vida e os agradeço por cada conselho.

Agradecimento a minha esposa Carla Bravo por todo suporte que meu deu durante essa caminhada, sem você ter aparecido em minha vida e todo seu incentivo, não teria concluído.

Agradecimento a todos que fizeram parte da Soul Code, pois foi uma fase importante para mim em diversos aspectos, pois aprendi muito com cada membro daquela gestão e a empresa me abriu diversas portas e a chance de ir morar em diversos lugares do mundo.

E por fim, agradeço a todos aqueles que contribuíram indiretamente para a conclusão deste trabalho.

"Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes".

(Marthin Luther King)

Resumo

Com o surgimento de sistemas cada vez mais complexos a área da engenharia de software iniciou um debate mais aflorado sobre arquiteturas de software, onde a arquitetura baseada em microsserviços e a em camadas começaram a ser o foco de diversas discussões sobre suas vantagens e desvantagens. Há engenheiros que são enfáticos em dizer que todos os sistemas deveriam utilizar a arquitetura baseada em microsserviços, outros que a arquitetura monolítica é mais eficiente, então o objetivo deste trabalho é comparar de forma empírica dois sistemas, cada um deles desenvolvido seguindo os conceitos das arquiteturas, tanto em camadas, quanto a baseada em microsserviços. O método que será utilizado para a realização deste trabalho é o qualitativo, com a finalidade de implementar e analisar as arquiteturas, possibilitando assim a realização de um estudo descritivo sobre ambas. Com essa análise empírica sobre as arquiteturas, pode-se verificar as vantagens e desvantagens de cada arquitetura, evidenciado aspectos suma importância para engenheiros de softwares ou projetistas de software. Os resultados revelaram que a implementação baseada em microsserviços se mostrou mais complexa que a em camadas. Em termos de escalabilidade, ambas arquiteturas se comportaram bem, com destaque para a arquitetura baseada em microsserviços dada a facilidade de escalar um determinado contexto que tenha se tornado um gargalo no projeto. Já do ponto de vista da manutenção, a arquite-tura monolítica mostrou-se ser um pouco mais complexa, visto que muitas das vezes se faz necessário entender algumas regras mais abrangentes. Em relação ao deploy a arquitetura baseada em microsserviço foi mais rápida, porém mais complexa, pois são diversos sistemas a serem enviados. Um grande problema da arquitetura baseada em microsserviços são os logs descentralizados e precisa-se recorrer a um software de terceiros. Sobre os processos de autorização e autenticação, a arquitetura monolítica mostrou-se simples e de implementação mais rápida. Em termos de falha em uma parte do sistema, a arquitetura baseada em microsserviços comportou-se melhor, uma vez que seus sistemas são isolados. De forma geral, é importante salientar que não existe “a melhor” arquitetura, mas sim aquela que mais se adequá a um determinado projeto ou equipe.

Palavras-chaves: engenharia de software; arquiteturas; microsserviços; camadas; monolítico;.

Abstract

Modern systems are becoming increasingly complex, and software engineering has started a more intense debate about software architectures, where microservice-based and layered architecture have become the focus of many debates about their advantages and disadvantages. There are engineers who are emphatic in saying that all systems should use microservice based architecture, others that monolithic architecture is more efficient, so the aim of this paper is to empirically compare two systems, each developed following the concepts of architectures. , both layered and microservice based. The method that will be used for this work is the qualitative one, with the purpose of implementing and analyzing the architectures, so there will be the possibility to make a descriptive study about both. With this empirical analysis of architectures, we can see that each architecture has its advantages and disadvantages, and software engineers or designers should consider these factors. The results revealed that the microservice-based implementation was more complex and the layered implementation simpler in terms of scalability, both architectures behave well, however in a microservice-based architecture you have the ability to scale a particular context that has become a bottleneck in the design, from a maintenance standpoint, the monolithic architecture proved to be a bit more complex, as it is often necessary to understand some broader rules, compared to deploying the microservice based architecture was more fast but more complex because there are so many systems to send, a big problem of the microservice based architecture is the decentralized logs and we need to use third party software, regarding authorization and authentication, the monolithic architecture is simpler and faster implementation in terms of failure in one part of the air system microservice-based architecture behaves better because its systems are isolated. In general and to conclude this work, it is important to understand that there is no better architecture, but the one that best fits that project and team.

Key-words: engenharia de software. microsserviços. monolítico. arquitetura em camadas.

Lista de ilustrações

-

Lista de abreviaturas e siglas

PaaS	Plataform as a Service
SaaS	Software as a Service
IaaS	Infrastructure as a Service
SOA	Service-Oriented Architecture
BaaS	Backend as a Service

Sumário

1 Introdução

As possibilidades que sugiram com as tecnologias oriundas na era da Web 2.0, como por exemplo as diversas formas de criar, compartilhar e interagir entre os usuários. Bennett Andrea Bishop (2011) destaca que as chamadas mídias sociais se tornaram uma ferramenta onde cada usuário pode coletivamente ou individualmente publicar e compartilhar imagens, arquivos em diversos formatos, como áudio, vídeo e texto.

As possibilidades são inúmeras com a Web 2.0, que traz consigo padrões comuns de arquitetura. Há muitas informações sobre essas arquiteturas no artigo de Governor e Nickull (2009) que aborda diversos conceitos e modelos que utilizamos hoje e muitas das vezes não sabemos os nomes, como por exemplo: **Service-Oriented Architecture (SOA)**, **Software as a Service (SaaS)**, **Participation-Collaboration**, **Asynchronous Particle Update**, **Mashup**, **Rich User Experience (RUE)**, **The Synchronized Web**, **Collaborative Tagging, Declarative Living and Tag Gardening**, **Semantic Web Grounding**, **Persistent Rights Management** e **both adopters of this pattern**. Nesse momento não é necessário saber as definições, mas buscar compreender que existiu uma mudança muito significativa na transição da Web 1.0 para a Web 2.0 e essa mudança nos trouxe termos novos e outros que já não eram mais utilizados.

Os padrões de iteração, compartilhamento, leitura e postagem em massa, trouxe uma outra perspectiva de como projetar software. Por tempos a área da engenharia de software foi espelho das engenharias tradicionais. Entretanto o desenvolvimento de software se difere tanto nos processo, quanto nos projetos arquitetônicos de software. Durante essa revolução de técnicas e tecnologias os antigos conceitos já bem definidos pelo meio cientifíco como o de sistemas distribuídos, voltaram a reaparecer de forma repaginada como Richards (2015).

É comum os desenvolvedores iniciarem seus projetos sem pensar na arquitetura que será aplicada a aquele projeto. É natural desenvolver um sistema utilizando a arquitetura em camadas, comumente chamada também de arquitetura monolítica. Porém segundo Richards (2015) não se deve pensar que qualquer problema poderá ou deve ser resolvido com apenas uma arquitetura e para saber a melhor arquitetura para aquele projeto, é necessário conhecer diversos exemplos e quando utilizar.

No capítulo 2 serão destacadas algumas informações sobre as arquiteturas de software como Tanenbaum (1944), Richards (2015) e Chiaradia (2018) descrevem em seus livros. Mas é importante estar ciente que o escopo deste trabalho será apresentar duas arquiteturas, são elas a arquitetura baseada em **microsserviços** e arquitetura em **camadas/monolítica**, pois são as mais populares e a que desejamos buscar mais informações.

1.1 Problema

Schmidt e MacDonell (2018) disserta que os softwares modernos que consiste diversas **features**(funcionalidades). A cada nova funcionalidade é natural que o grau de complexidade do software aumente, não bastando a complexidade aumentar, há outros fatores como: a falta de organização da arquitetura de pastas, códigos difíceis e altamente acoplados entre si e, complexidade para dar manutenção e **refatoração**. Diante de todas as características é importante entender, segundo Schmidt e MacDonell (2018) que cria-se uma dificuldade para que outros engenheiros de software ou programadores consigam entender o design, a estrutura e as dependências que fazem parte daquela arquitetura.

Richards (2015) cita que o padrão arquitetural natural de desenvolvimento de software e o que a maioria das organizações tradicionais aplicam para todos os projetos e problemas, também tem suas desvantagens e uma delas é ser difícil se manter, refatorar ou até modificar o código. Adicionar novas funcionalidades a um sistema já existente sem ponderar as variáveis e a melhor arquitetura para aquele projeto resultará em um colapso, segundo Schmidt e MacDonell (2018).

Uma arquitetura baseado em camadas como na 1.1 tem camadas extremamente acopladas, onde cada parte é responsável por uma determinada ação na arquitetura. Como Richards (2015) explana, a arquitetura em camadas tende a se tornar complexa, difícil de manter a longo prazo, deploys lentos, testes automatizados lentos e um alto custo para escalar em alguns cenários a aplicação.

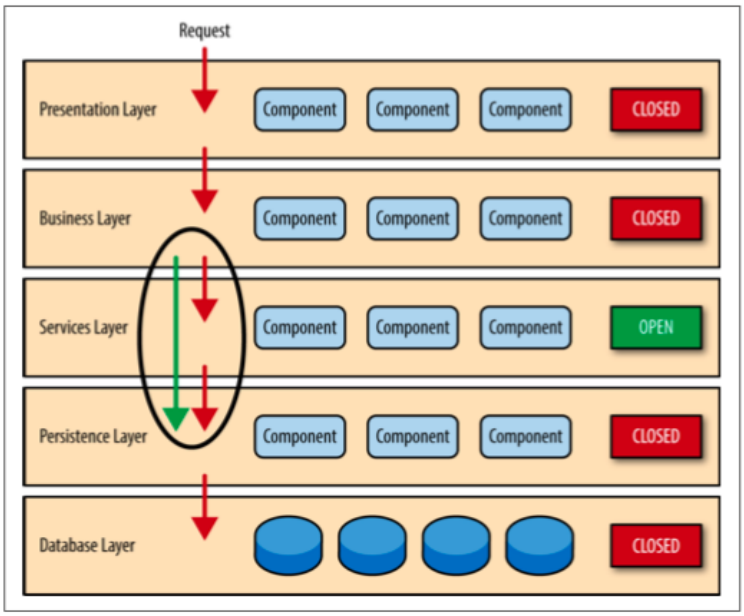
Com os sistemas monolíticos se tornando cada vez mais complexos e os problemas a serem resolvidos também, qual a melhor forma de-se desenvolver software com qualidade e manutenível a longo prazo?

1.2 Hipótese

Em diversos debates no atual cenário a melhor forma para resolver os problemas de complexidade do monolítico e de manutenibilidade dos softwares ao longo prazo é utilizar a arquitetura baseada em microsserviços, pois a mesma irá:

- Implementação arquitetural rápida;
- Escalar a aplicação de forma fácil e com; diminuição de custos;
- Manutenível a longo prazo;
- Inserção facilitada de novas linguagens e tecnologias;
- Deploy simples e rápido;

Figura 1 – Arquitetura em camadas e o seu fluxo



Richards (2015)

.

- Logs difíceis de manter;
- Simples Implementação de autorização e autenticação.

1.3 Objetivo Geral

O objetivo do trabalho é implementar e verificar o quão complexo é implementar as arquiteturas em camadas e a baseada em microsserviços. Propõe-se analisar a escalabilidade, manutenibilidade, o uso de multi tecnologias, a facilidade do deploys, a verificação dos logs, implementação e manutenção da autorização e autenticação nas duas arquiteturas e fazer uma comparação empírica entre ambas.

1.4 Justificativa

Com o avanço das tecnologias os engenheiros precisam estar preparados para lidar com problemas arquiteturais complexos, haja visto a complexidade dos sistemas que estamos desenvolvendo hoje em dia. Schmidt e MacDonell (2018) explanam o quanto precisamos entender melhor sobre as arquiteturas, tanto as vantagens, quanto as desvantagens. Com o desenvolvimento dos softwares modernos e com softwares cadas vez mais distribuídos em conceitos como **cloud**, infra-estrutura e plataforma como serviço, precisa-se de mais estudos científicos sobre as arquiteturas e mais especificamente a baseada em

microsserviços, visto que não é uma arquitetura nova, porém é uma evolução de modelos antigos para os sistemas de hoje.

Há inconvenientes quando se implementa uma arquitetura em camadas sem se planejar, como já dito suas vantagens e desvantagens devem ser levadas em consideração e também o projeto e equipe. Schmidt e MacDonell (2018) propõe que os arquitetos e os engenheiros devem entender todas as características das diversas arquiteturas que já foram catalogadas. Há poucos artigos cientifico sobre arquiteturas de modo geral e também especificamente sobre a arquitetura baseada em microsserviços.

O aprendizado em arquitetura e a experiência sobre softwares mal planejados fazem com que esse trabalho seja importante para entender melhor as arquiteturas propostas nesse projeto. Esse trabalho é importante não só para a formação, mas pela identificação com essa temática da engenharia de software.

1.5 Método

Para o propósito deste trabalho será estabelecido o estudo da arquitetura de microsserviços, analisando todo o fluxo entre as partes que compõe as arquiteturas.A implementação é uma parte fundamental desse trabalho, haja visto que com as implementações feitas, poderemos obter dados empírico e também dados analíticos.

Dito o resumo acima o método utilizado será:

- Levantamento bibliográfico sobre os tópicos abordados neste trabalho como: Arquiteturas, Microsserviços, Sistemas distribuídos e outros;
- Pesquisar sobre a Arquitetura de Microsserviços e sua implementação;
- Definição das ferramentas para a construção da arquitetura de Microsserviços;
- Escolha das tecnologias;
- Modelagem da arquitetura pensando nas ferramentas e nas vantagens da ferramenta;
- Implementação da arquitetura com a utilização das tecnologias Open Source;
- Com um navegador analisar o tempo de requisição e resposta. Relatar o esforço para a implementação de ambas as arquiteturas. Efetuar testes no desligamento do servidor e analisar como se comporta o sistemas.
- Testar e documentar os resultados

2 Arquitetura de software

A arquitetura de software de modo geral tem por objetivo apresentar a comunicação entre cada camada do sistema, bem como, suas limitações, características e interfaces como bem definido por Sizo Adriano Del Pino Lino (2010). O trabalho tem por objetivo descrever as arquiteturas baseadas em camadas e microsserviços.

Neste capítulo serão apresentadas diversas tecnologias, conceitos e definições que compõe as duas arquiteturas. Quando tratarmos de ferramentas iremos deixar claro que o intuito não é a ferramenta utilizada, mas os conceitos utilizados por elas, entretanto, deixaremos sempre em aberto a melhor tecnologia a ser aplicada, porém vamos utilizar algumas tecnologias como Node.js (http://nodejs.org), Firebase (http://firebase.google.com) e outras para conseguirmos chegar a conclusões concretas.

2.1 Arquitetura

2.1.1 Decisões de Arquitetura

Os desenvolvedores normalmente iniciam os projetos sem um planejamento de arquitetural adequada. Quando não se pensa no projeto como um todo e se desenvolve esse novo sistema de maneira desorganizada, a aplicação poderá sofrer grandes problemas conforme for crescendo, devido a falta de padrão. Naturalmente como já descreve Richards (2015) os projetos são desenvolvidos utilizando a arquitetura em camadas ou também chamada de arquitetura monolítica. Com uma arquitetura monolítica desorganizada, teremos módulos ou classes desorganizados e muitas responsabilidades em lugares inapropriados. Os engenheiros de softwares precisam entender e sempre justificar a decisão do uso daquela arquitetura.

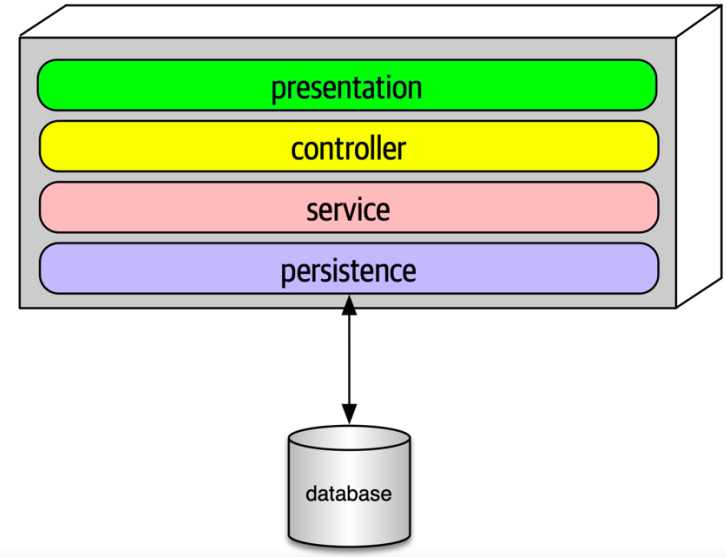
Engenheiros de software e programadores implementam, corrigem, melhoraram, então repetem o ciclo de desenvolvimento. Após todas essas etapas, que são feitas por diversas vezes, a equipe inicia a fase de pensar em reconhecer ou melhorar a arquitetura que atende aquele sistema. Não é um processo fácil, a busca pela identificação da arquitetura ideal leva-se tempo, mas se faz necessário partir de um ponto inicial, Richards (2015) descreve bem sobre essas etapas.

2.1.1.1 Arquitetura em camadas ou monolítica

A arquitetura em camadas/monolítica organizada em camadas horizontais, cada camada tem funções específica dentro do aplicativo (por exemplo, lógica de negócio, apresentação, persistência e banco de dados). No há uma quantidade ou tipo de camadas na

arquitetura monolítica, mas naturalmente utiliza-se quatro camadas, são elas: lógica de negócio, apresentação, persistência e banco de dados como dito por Richards (2015) como podemos verificar na figura 2.1.1.1 . Em algumas descrições desse tipo de arquitetura pode se encontrar também três camadas como apresentação, camada de negócio e banco de dados. Aplicações menores normalmente seguem as três camadas, porém quanto maior e mais complexo mais camadas são acrescentada.

Figura 2 – Arquitetura em camadas



Ford (2020)

.

As camadas na arquitetura tem função e responsabilidade específicas na aplicação. Podemos especificar algumas, como por exemplo a camada de apresentação que é responsável por lidar com toda a comunicação entre o sistema e o usuário que utiliza a aplicação, enquanto uma camada de negócios é responsável por executar regras de negócios específicas associadas à solicitação. Importante destacar que as camadas são abstrações e ela não precisa saber do sistema como um todo, ela só receber as informações que precisa para ser mostrada. Por exemplo, a camada de apresentação não precisa saber a regra de negócio, só exibir os dados para o usuário. Richards (2015) explica muito bem as camadas, mas para um melhor entendimento, o cálculo é feito na camada de negócio, por exemplo 5 + 4, e a camada de apresentação só irá mostrar o valor final 9.

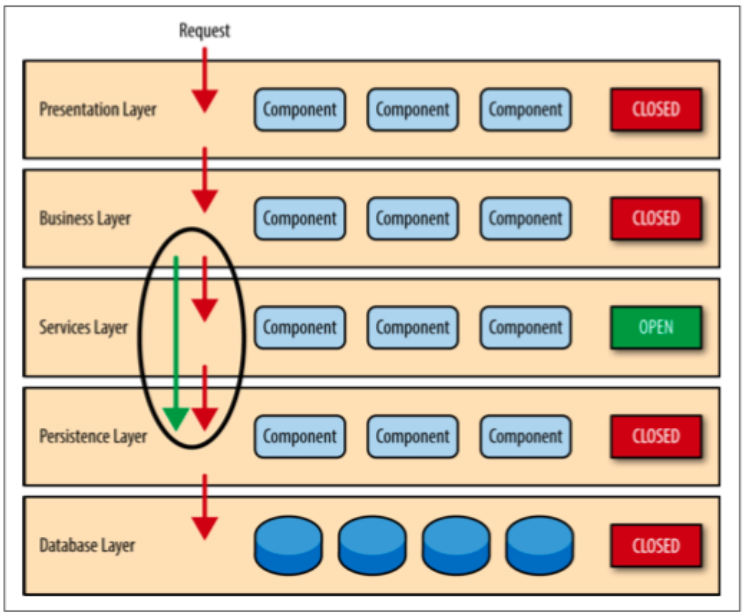
A arquitetura em camadas ou também conhecida como monolítica tem sido a mais utilizada por anos, haja visto que é uma das arquiteturas que mais temos conhecimento quanto a vantagens e desvantagens, pois há diversos artigos e também há uma experiência devido o tempo que trabalhamos com ela segundo Batista (2018).

A arquitetura monolítica pode ser descrita como um sistema centralizado, onde todas as responsabilidades e funcionalidades estão em um mesmo sistema, se por algum motivo um servidor pausar/falhar a aplicação inteira é afetada e nenhum usuário poderá utilizar a aplicação. Nesse tipo de arquitetura temos as algumas camadas que são elas: negócio, apresentação e dados, dependendo do modelo de arquitetura em camadas podemos ter outras como mostrado na figura 2.1.1.1.

Podemos descrever algumas vantagens dessa tipo de arquitetura, por exemplo as camadas seguem uma hierarquia, as dependências são centralizadas, os códigos reutilizáveis e as regras de negócio também. Aplicações monolíticas são fáceis de serem desenvolvidas, afinal são ferramentas especializadas em uma única aplicação.

Como qualquer coisa específica, tem também suas desvantagens como descreve Batista (2018), a escalabilidade, agregação de novas tecnologias e a curva de aprendizado pode-se tornar alta e dependendo da regra de negócio, pode também ficar mais complexo, pois há uma grande base de código.

Figura 3 – Arquitetura baseada em microsserviços



Richards (2015)

.

2.1.1.2 Arquitetura de Microsserviços

A arquitetura baseada em microsserviços ganhou muito destaque nos últimos anos por causa das grandes corporações que precisava escalar as suas aplicações de forma mais rápida e menos custosa e também para que suas equipem possam ter mais produtividade. Tal fato deve-se ao entendimento que é melhor um colaborador trabalhar em um pequeno

sistema, do que trabalhar com diversos sistemas complexos e que poderia gerar grandes prejuízos as empresas caso tenha alguma falha. Richards (2015) explica que ainda há muitas dúvidas sobre o padrão baseado em microsserviços e a melhor forma de projetar esse tipo de arquitetura.

A implementação desse tipo de arquitetura será realizada ao longo deste trabalho, porém é importante entender que há várias formas de se implementar a mesma arquitetura. Há conceitos que são pré requisitos para que a arquitetura seja considerada baseada em microsserviços, entretanto há formas mais elaboradas ou menos elaboradas. Cada micro serviço segue o princípio de responsabilidade única e tem como princípio que cada componente da arquitetura o isolamento desse sistema, com banco de dados próprio e comunicação via mensageria ou JavaScript Object Notation (JSON) por exemplo. Suas vantagens são perspetiveis em seus conceitos, onde podemos analisar brevemente que um sistema menor é mais simples de dar manutenção do que um sistema mais complexo, porém se faz necessário uma análise mais ampla as arquiteturas como Richards (2015) expõe.

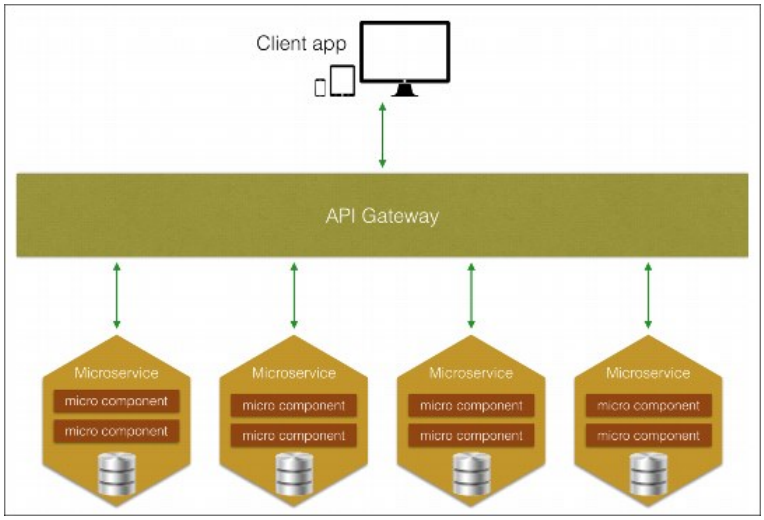
Com essa arquitetura é necessário modificar a forma como desenvolve-se software de modo geral, como por exemplo pensar em componentes de serviços, no caso desse tipo de arquitetura a cada componente serviço pensado, teremos um sistema, por exemplo para o contexto de efetuar pagamento, teríamos um gerador de boleto e uma de finalização de pagamento para cartões, seria necessário um sistema para o boleto e outro para a finalização de pagamentos via cartões. A arquitetura pode conter um ou mais módulos (por exemplo, classes em Ruby ou funções em Elixir) que representam um contexto de negócio único (por exemplo, fornecer boletos para o usuário) ou mais ainda um aplicativo de negócios de grande porte (por exemplo, financeiro de uma empresa). Richards (2015) explica que projetar o nível de granularidade dos componentes de serviço é um dos maiores desafios dos microsserviços.

Alguns conceitos chaves descritos por Richards (2015) necessitam serem compreendidos, por exemplo os conceitos de sistema distribuídos. A arquitetura baseada em microsserviços segue os conceitos de sistemas distribuídos como já dito, isso significa que todos os componentes da arquitetura são desacoplados e acessados por meio de algum tipo de comunicação de acesso por exemplo, Advanced Message Queuing Protocol (AMQP), Representational state transfer (REST), Simple Object Access Protocol (SOAP), Remote Method Invocation (RMI), etc. É por isso que ela alcança as características de escalabilidade e implantação de forma superior a outras arquiteturas.

Como já dito acima, não é pré-requisito a utilização de todos os componentens para se ter uma arquitetura baseada em microsserviços, porém para esse trabalho iremos utilizar um microserviço chamado API Gateway. O Nguyen (2016) explica o quão benéfico é ter a API Gateway para lidar com diferentes solicitações de API, roteando-as para os

microsserviços apropriados. Se você usa o API Gateway, projete e implemente-o com cuidado para evitar o acoplamento entre serviços. Cada microsserviço pode ter vários microcomponentes que podem ser separados em diferentes camadas. A 2.1.1.2 mostra esse tipo de microsserviço.

Figura 4 – Arquitetura baseada em microsserviços



Nguyen (2016)

.

Sistemas baseados puramente na arquitetura de microsserviços seguem algumas características que Cerny (2017) explora em seu artigo, são elas:

- O programa deve ter uma só tarefa e executar bem. Exemplo: gerador de PDF, ele só vai gerar PDF e fará muito bem essa função.
- A aplicação deve ser fácil de ser integrada a outros softwares. Qualquer aplicação que queira gerar PDF, poderá gerar facilmente.
- O gerador de PDF deve utilizar uma interface única com os outros sistemas, tudo de forma transparente para o usuário ou sistemas que irão utilizar.

Batista (2018) define a arquitetura de Microsserviços como uma estrutura descentralizada, onde todas as camadas da aplicação ficam em servidores próprios e ambos se comunicam através das APIs. A forma de comunicação torna-os totalmente independentes e autônomos.

Uma descrição interessante é a dos autores do artigo Zhu Len Bass (2016) onde eles definem os microsserviços como sistemas que cada serviço é pequeno (daí o "micro") e que todos os desenvolvedores desses serviços entendam que estão trabalhando no mesmo sistema.

Esse tipo de arquitetura tem diversas vantagens como Batista (2018) cita:

- Utiliza-se de comunicação leve e simples atráves do protocolo HTTP e com o HTTP version 2 a comunicação se tornou mais rápida.
- A utilização de qualquer linguagem de programação que o desenvolvedor queira utilizar.
- Permite experimentar e testar
- Bancos de dados diferentes

Com esse tipo de arquitetura há um ganho na diminuição de falhas, haja visto que são independentes, caso um sistema de pagamentos caia, o gerador de boletos, por exemplo continua funcionando. Entretanto esses benefícios não param por aí, como diz KILLALEA (2016) há diversos benefícios escondidos, ele dá alguns tópicos como: Inovação sem permissão, permissão para falhar, Interromper com confiança, você constrói, você mantém, acelera as descontinuações, testa de forma diferente, finaliza metadados centralizados, concentra a dor, caso tenha se interessado e deseja mais informações é sugerido que leia o artigo completo.

Uma das desvantagens que devemos nos preocupar quando estamos utilizando esse tipo de arquitetura é no contexto específico da aplicação e a complexidade entre a comunicação dos sistemas.

REST significa REpresentational State Transfer, que é um estilo de arquitetura, e não um protocolo como definido no artigo de Harber (2019).

Harber (2019) descreve que a API é uma abreviação de interface de programa de aplicativo, que permite que os aplicativos se comuniquem. No caso da web, uma API é normalmente um conjunto de URLs que respondem com dados quando chamados da maneira correta e com as informações corretas.

Uma API REST é uma interface sem estado para seu aplicativo. No caso da pilha MEAN (Mongo, Express, Angular e NodeJS), a API REST é usada para criar uma interface sem estado para seu banco de dados, permitindo que outros aplicativos, como um SPA Angular, trabalhem com os dados. Em outras palavras, você cria uma coleção de URLs estruturadas que retornam dados específicos quando chamados, essa definição também é descrita por Harber (2019).

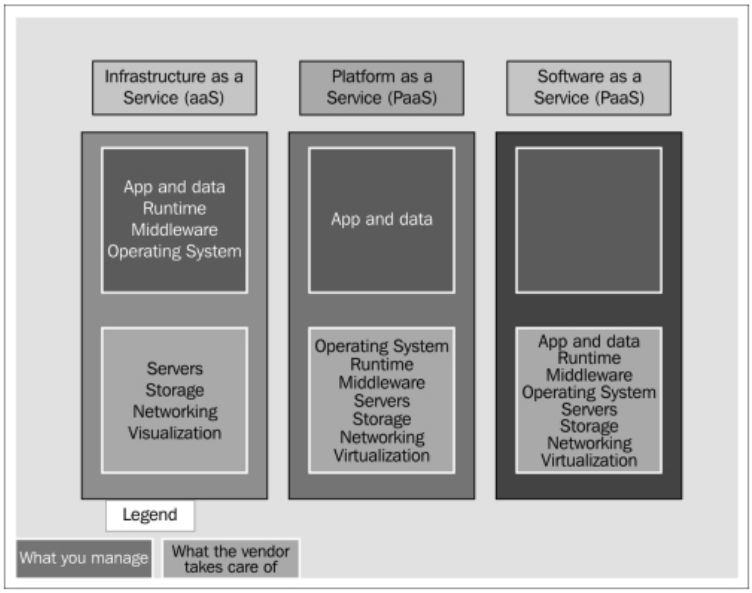
2.1.2 Computação em nuvem

A Computação em nuvem de forma simplista é levar ao usuário final, seja ele empresas ou pessoas físicas uma infraestrutura, plataforma ou software complexo, através

de serviços simples e acessível de qualquer computador com um navegador instalado. Como Sousa (2010) referencia no seu artigo, computação em nuvem é uma tendencia para os dias atuais e tem o objetivo entregar serviços sobre demanda, com pagamento baseado na utilização. Caso se interesse no assunto, o artigo de Sousa (2010) é interessante para se aprofundar nesse conceito.

Hanjura (2014) descreve que os serviços oferecidos pela computação em nuvem são divididos em diferentes modelos de serviço - infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e software como serviço (SaaS) como podemos verificar na figura 2.1.2 . Essa classificação é feita para separar os diferentes tipos de serviços que um usuário pode adquirir para atender às necessidades de negócios em um ambiente de computação em nuvem.

Figura 5 – Alguns serviços suportados pela computação em nuvem



Hanjura (2014)

.

A nuvem é classificada em alguns modelos de serviços. Cada modelo fornece um serviço distinto. Os modelos são IaSS, BaSS, PaSS e SaSS. Como Evangelista Wellington Galdino (2015) cita em seu artigo a sigla em inglês significa Infrastructure as a Service (IaaS), Backend as a Service, Plataform as a Service (PaaS) e Software as a Service (SaaS).

Infrastructure as a Service (IaaS) é o tipo de modelo de serviço que permite ao usuário final utilizar capacidade de processamento, redes de dados, memória ram, SSD, sistema operacional e diversos componentes de hardware e rede. O IaSS permite gerenciar todos os recursos através de interfaces gráficas. Exemplo de IaSS: AWS, Google Cloud e Digital Ocean.

A definição de IaaS feita por Hanjura (2014) é que o modelo de serviço em nuvem que permite ao usuário fornecer recursos de hardware virtualizado sob demanda. Fisicamente, esses recursos podem se espalhar por vários data centers, mantidos pelo provedor de serviços. Esses recursos incluem armazenamento virtual, conexões de rede e balanceadores de carga para o recurso de hardware provisionado. O usuário pode usar o recurso sob demanda e pagar por uso. Se o usuário precisar de mais recursos, o provedor poderá escalar automaticamente o hardware de acordo com a necessidade e vice-versa. Um bom exemplo de um provedor de IaaS é o Amazon Web Services (AWS - http: //aws.amazon.com). É o provedor de IaaS mais popular na nuvem. Rackspace (http://www.rackspace.com) é outro exemplo.

Plataform as a Service (PaaS) é o modelo de serviço que levanta um servidor de diversas linguagens sem muito esforço. O usuário não tem acesso a infraestrutura, rede, dependências do sistema operacional e não precisa se preocupar com a atualização da mesma. Contrata só as configurações da plataforma.

A definição de PaaS feita por Hanjura (2014) é que o modelo de serviço em nuvem que fornece as ferramentas para criar aplicativos de software na nuvem. Uma analogia próxima seria considerar o PaaS como um sistema operacional e um middleware do ambiente em nuvem. O PaaS fornece aos desenvolvedores a plataforma subjacente a ser usada para desenvolver seus aplicativos. Ele cuida do suporte a um idioma ou tecnologia específica que os desenvolvedores da pilha desejam usar. Muitos provedores de PaaS também permitem o dimensionamento sob demanda dos recursos subjacentes de computador e armazenamento, automaticamente, para liberar o usuário da nuvem da tarefa de alocar recursos manualmente. No PaaS, o consumidor do serviço controla a implantação e a configuração. O provedor PaaS provisiona os servidores, a rede e as necessidades computacionais do aplicativo de software. O modelo PaaS também permite uma arquitetura multitenant (definir isso) para que vários usuários possam usar o aplicativo da Web de maneira segura, escalável, simultânea e à prova de falhas. As sofisticadas soluções PaaS também fornecem um ambiente integrado de desenvolvimento de aplicativos da web, o que facilita a codificação colaborativa, o controle da fonte e a implantação. Heroku (http://www.heroku.com) e Google App Engine (http://cloud.google.com/AppEngine) são dois exemplos de plataformas PaaS de sucesso.

Hanjura (2014) descreve que embora o PaaS seja inerentemente mais estável em comparação com o componente SaaS. PaaS evoluiu enormemente nos últimos anos e forneceu à comunidade de desenvolvedores ferramentas incríveis (menos adjetivo!) para trabalhar e implantar aplicativos distribuídos praticamente em pouco tempo.

Hanjura (2014) descreve o Heroku (http://www.heroku.com) como um dos principais fornecedores de PaaS no negócio de software em nuvem, provando ser a solução PaaS líder em pequenas e grandes empresas. Com melhorias consistentes e a filosofia

de "conveniência"sobre "configuração", o Heroku se tornou a principal plataforma de desenvolvimento de aplicativos em nuvem para desenvolvedores. A filosofia da Heroku é permitir que os desenvolvedores se concentrem apenas em escrever aplicativos da Web e esquecer os servidores. O Heroku cuida magicamente da criação, implantação, execução e dimensionamento do aplicativo para o desenvolvedor sob demanda.

Hanjura (2014) descreve o Heroku como uma plataforma de aplicativos em nuvem poliglota que oferece uma enorme flexibilidade na escolha de uma linguagem de programação apropriada para o desenvolvimento de aplicativos da web. O Heroku fornece suporte de plataforma para Ruby, Ruby on Rails, Java, Node.js, Clojure, Scala, Python e PHP a partir do início de 2013, caso tenha interesse em saber mais sobre essa plataforma consultar Hanjura (2014).

Software as a Service (SaaS) é o modelo em que o usuário tem a aplicação disponível em tempo real através dos smartfones ou dos navegadores. O usuário não tem acesso a infraestrutura ou a plataforma, tudo é transparente para o usuário através de uma interface simples. O usuário só poderá configurar aspectos restritos da própria aplicação.

A definição de SaaS feita por Hanjura (2014) é que o modelo SaaS da nuvem fornece software que você pode consumir do alcance do seu navegador da web. Não há necessidade de instalações complexas e demoradas. Abra um navegador, aponte para um URL e use o aplicativo apontado pelo URL. O que acontece nos bastidores está oculto ao usuário. O SaaS evoluiu consideravelmente na última década. Muitos provedores de SaaS tornaram obsoleto o software para desktop ou hospedado localmente. Tudo que você precisa é de um navegador e está pronto para usar qualquer aplicativo para fazer qualquer coisa. Sem dores de cabeça de atualizações de software, incompatibilidade de versão ou portabilidade de software. O serviço Gmail (http://gmail.com) do Google é uma das implementações de SaaS mais bem-sucedidas e conhecidas. O componente SaaS cresceu exponencialmente com empresas aproveitando a infraestrutura e a plataforma subjacentes para criar versões em nuvem da maioria de suas ofertas de produtos de software. Em 2013, quase todas as empresas que valem a pena ter uma versão SaaS de seus aplicativos populares de software disponíveis para clientes on-line.

2.1.3 DevOps e suas tecnologias

O DevOps é um conjunto de práticas ou uma cultura com o propósito de diminuir o tempo entre uma mudança feita e submetida em produção de forma a garantir a qualidade, seja em código ou seja em testes. O Zhu Len Bass (2016) ainda descreve que toda mudança gera impactos nos processos, produtos, tecnologias utilizadas e estrutura organizacional, haja visto que estamos lidando com uma nova abordagem. A implantação dessa cultura não é fácil, muitas das vezes haverá tensões na organização e no negócio, pois terá que ser modificada a forma de trabalho diário como descreve Zhu Len Bass (2016).

Zhu Len Bass (2016) descreve sobre os efeitos que o DevOps têm sobre os desenvolvedores ao longo do processo de desenvolvimento de software, ele lista alguns itens que valem a pena serem citados. São eles:

- Integração contínua é uma prática que deve ser utilizada e ela é o ato de submeter um código para um repositório central e o software executar os testes automatizados e subir para homologação ou produção sem que haja uma pessoa que cuide desse processo.
- Os sistemas são monitorados e após uma implementação, caso o haja alguma falha é possível reverter as alterações.

A utilização dessa cultura depende muito de diversas ferramentas, são elas: gerenciamento de contêineres, integração contínua, orquestração, monitoramento, implantação e teste. O Zhu Len Bass (2016) cita que cada vez mais, os engenheiros de software mantêm e configuram essas ferramentas.

Docker é uma aplicação que virtualiza os servidores, essa ferramenta foi desenvolvida para fornecer, via script em YAML (YAML é um formato de arquivo para codificação de dados legíveis para humano), um ambiente de forma rápida, seja para desenvolvimento ou produção. Todas às vezes ao inicializar um servidor necessitamos de um processo manual que se repete diversas vezes. Tal processo pode serautomatizado de maneira simples com o uso do docker.

O Docker é considerada uma plataforma de virtualização em contêineres, ajuda os desenvolvedores a inicializar o ambiente com o mínimo esforço. Para tal, basta ter a plataforma instalada localmente para, em poucos minutos, tu ter um ambiente totalmente configurado. O docker é uma tecnologia de código aberto, como Ouverney (2017) descreve em seu artigo, durante o build é criado um contêiner com todas dependências instaladas e a aplicação pronta para uso. Essa plataforma ajuda na implementação da arquitetura de microsserviços, pois cada aplicação fica isolada em um contêiner, não afetando o funcionamento dos outros serviços.

Durante o desenvolvimento de uma determinado sistema nos deparamos com diversos cenários de repetição de tarefas e, quando uma tarefa é repetitiva e pode ser automatizada, o melhor é permitir que a máquina efetue esse teste, poupando esforços da equipe para a resolução de alguma outra tarefa importante. Os computadores fazem tarefas repetitivas melhor do que os humanos, o Kon (2008) descreve em seu artigo, precisamos pensar em qualidade e nas inúmeras dificuldades quando se trata de questões humanas, então, porque não automatizar os testes, pois há processos repetidos que faríamos e poderemos falhar por falta de atenção em um dado momento.

A cultura DevOps já define em seus processos a automatização dos testes de software, Kon (2008) define que os testes automatizados é uma técnica voltada a qualidade de software, onde uma alteração que possa quebrar o sistema, facilmente será percebida pelos desenvolvedores em poucos minutos. Existem alguns exemplos de testes, mas não se torna necessário descrever cada um deles, caso sinta-se curioso para saber mais sobre o artigo do Kon (2008) lhe trará uma visão mais ampla sobre os tipos de teste.

O Git é um sistema de controle de versão como explica Beer (2018). Esse tipo de software é projetado para acompanhar as alterações feitas nos arquivos ao longo do tempo. Ele explica ainda que o git é um sistema de controle de versão distribuído, o que significa que todos os que trabalham com um projeto no Git tem uma cópia do histórico completo do projeto, não apenas do estado atual dos arquivos.

O Github é uma plataforma onde se pode fazer upload de uma cópia do seu repositório Git e de forma fácil toda sua equipe poderá fazer um clone do projeto e trabalhar simultaneamente no mesmo projeto como dito por Beer (2018). Ele ainda explica que o Github faz isso fornecendo um local centralizado para compartilhar o repositório, uma interface baseada na Web para visualizá-lo e recursos como *fork, pull request, wiki* do Github que permite especificar, discutir e revisar alterações com a sua equipe de forma mais eficaz. Caso queira saber mais informações basta acessar o site http://github.com.

O Gitlab é uma plataforma de código aberto para armazenar uma cópia do seu repositório git, igualmente ao Github, podemos ter mais informações sobre o mesmo no Baarsen (2018) ou no próprio site http://gitlab.com.

2.1.4 Linguagem de Programação e Framework

Bojinov (2018) descreve o Node.js como uma nova e empolgante plataforma para desenvolvimento de aplicativos web e servidores de aplicativos. Essa tecnologia é detalhada como um projeto que trás extrema escalabilidade em aplicativos e em rede através de uma combinação engenhosa de JavaScript do lado servidor, I/O assíncrona e programação assíncrona. Ele é construído em torno de funções anônimas JavaScript e uma arquitetura orientada a eventos de encadeamento de execução única.

Sobre a multithreading, Bojinov (2018) afirma que o Node.js evita os threads por causa de sua complexidade. É dito que, com arquiteturas orientadas a eventos de thread única, o espaço ocupado na memória é baixo, o rendimento é alto, o perfil de latência de carga é melhor e o modelo de programação é mais simples. A plataforma Node.js está em uma fase de rápido crescimento, e muitos a veem como uma alternativa atraente às arquiteturas de aplicativos da web tradicionais usando Java, PHP, Python ou Ruby on Rails.

React Native é uma estrutura de desenvolvimento de aplicativos na qual você usa

tecnologias padrão da Web (ou, em alguns casos, algo semelhante às tecnologias padrão da Web) para criar seu aplicativo. Isso significa algo parecido com o HTML, JavaScript e CSS. O React Native é baseado na estrutura React do Facebook, uma estrutura popular de desenvolvimento da web. A diferença crítica entre os dois é que o React tem como alvo os navegadores da Web, enquanto o React Native tem como alvo os smartphones e seus sistemas operacionais como descrito por Zammetti (2018).

2.1.5 Banco de dados

MongoDB é um banco de dados NoSQL para a Web moderna. NoSQL significa *Not only* SQL, é um termo cunhado por Carlo Strozzi em 1998, por seu banco de dados de código aberto que não seguia o padrão SQL, mas ainda era relacional.Para mais detalhres sobre o MongoDB, consultar https://docs.mongodb.com/manual/.

Juba (2018) descreve o PostgreSQL como um sistema de gerenciamento de banco de dados relacional de objeto de código aberto. É executado nos sistemas operacionais mais modernos, incluindo Windows, MacOS e Linux. Está em conformidade com o SQL.

2.1.6 Firebase

O Yahiaoui (2017) descreve o firebase como um BaaS, um back-end como serviço, com ele evitamos instalações e configurações que não iria agregar na solução do nosso problema para o momento.

O referido autor cita ainda que o Firebase é um BaaS com muitos recursos que facilita a criação dos projetos e elimina muitas tarefas tediosas. Ademais, fornece uma plataforma segura e bem construída que oferece simplicidade e escalabilidade de forma rápida.

2.1.7 Firebase Realtime Database

Yahiaoui (2017) cita que o Firebase Realtime Database é o produto mais usado pelos desenvolvedores em toda a pilha de produtos disponíveis no Firebase. Ele oferece diversas funções como atualizações, inserções e exclusões de dados em tempo real.

O mesmo autor ainda detalha que a funcionalidade de transmissão integrada e uma API extremamente simples de usar torna o Realtime Database um recurso atraente. O interessante a se destaca que a APIs oferecem aos desenvolvedores a oportunidade de explorar as APIs em uma escala maior, independentemente do ecossistema e ambiente que eles estão usando. Além disso, o Firebase Realtime Database vem com suporte offline. Esse recurso entra em ação quando o aplicativo está em um estado de rede não tão confiável.

2.1.8 Firebase Authentication

Essa funcionalidade é uma das partes centrais de qualquer sistema e o Yahiaoui (2017) descreve que a criação de um sistema de autenticação seguro acaba sendo um trabalho muito tedioso, mesmo que seja apenas e-mails e senhas.

Os aplicativos tendem a utilizar login com as redes sociais também e é um dos recurso mais utilizados e solicitados no cenário atual. É uma maneira rápida de fazer com que o usuário acesse o sistema como é descrito por Yahiaoui (2017).

O Firebase fornece métodos de autenticação múltipla, por exemplo email e senha, login social com as principais fornecedores Google, Facebook e etc Yahiaoui (2017).

2.2 Trabalhos relacionados

Nesta seção são mostrados alguns trabalhos relacionados sobre arquiteturas. Os trabalhos ajudaram na elaboração desse trabalho.

O autor Moreira (2015) destaca as vantagens dos microsserviços e desvantagens dessa arquitetura, mostra na prática como criar uma arquitetura de microsserviço com Java Spring Boot. De modo geral o autor destaca que há diversas arquiteturas e a de microsserviços se encaixa em contextos específicos.

O trabalho de Chiaradia (2018) aborda conceitos das áreas de gestão de relacionamento com o cliente, CRM Social e Web 2.0, enumerando as características e os benefícios. É proposta uma arquitetura de microsserviços para o sistema de CRM Social na qual o autor apresenta os fluxos de comunicação e se propõe a criação de um modelo para servir de base para futuras implementações.

O objetivo do artigo de Cerny (2017) é conceituar as arquiteturas SOA (Service Oriented Architecture) quanto Microsserviços e mostrar as funcionalidades de cada arquitetura. O autor analisa a arquitetura quanto a performance, complexidade, tanto de implementação, quanto a uso nos dias atuais.

Como Pahl e Jamshidi (2016) em seu artigo com o estudo de mapeamento dos métodos, técnicas e melhores práticas em arquitetura, com uma atenção particular à aplicação na nuvem. A revisão revela que a Microsserviços ainda está em um estágio formativo. É necessária uma avaliação mais experimental e empírica dos benefícios. Este estudo também demonstrou que a automatização e facilidade da arquitetura de Microsserviços em um cloud se torna necessário para extrair as vantagens que ela pode proporcionar.

O artigo KILLALEA (2016) descreve os diversos benefícios escondidos que a arquitetura de Microsserviço proporciona aos adeptos, no caso do artigo as empresas. Há algumas vantagens que não são claras no dia a dia, entretanto o artigo esclarece 8 pontos

importantes que os administradores de modo geral poderão visualizar.

A arquitetura de Microsserviços precisa ser monitorada, devido os micro sistemas estarem totalmente desacoplados, é necessário centralizar todas as informações e aí que o artigo de GHIROTTI e RENTZ (2018) nos ajuda a entender a importância dessa arquitetura.

O trabalho conduzido por Raji Alok Hota e Huang (2015) é importante, devido o mapeamento de visualização na área científica com a utilização da arquitetura de micro serviços, utilizando todos os conceitos da web 2.0. Um exemplo desse uso, em que o artigo retrata e o aumento da mobilidade, onde o mesmo pode ser utilizando em qualquer dispositivo e independente do sistema operacional, atingindo um público mais amplo e buscando ter um ecossistema mais rico, criando ferramentas e sites interativos.

3 Metodologia

O propósito deste trabalho é realizar um estudo descritivo sobre as arquiteturas em camadas e microsserviços. A abordagem da pesquisa será qualitativa de caráter exploratório, pois iremos focar no caráter subjetivo do objeto analisado.

O método que será utilizado para a realização deste trabalho é o qualitativo, com a finalidade de implementar e analisar as arquiteturas, assim poderemos fazer um estudo descritivo sobre as duas arquiteturas ilustradas nesta pesquisa. Em relação ao caráter exploratório serão aplicados alguns problemas na arquitetura e analisaremos o seu comportamento.

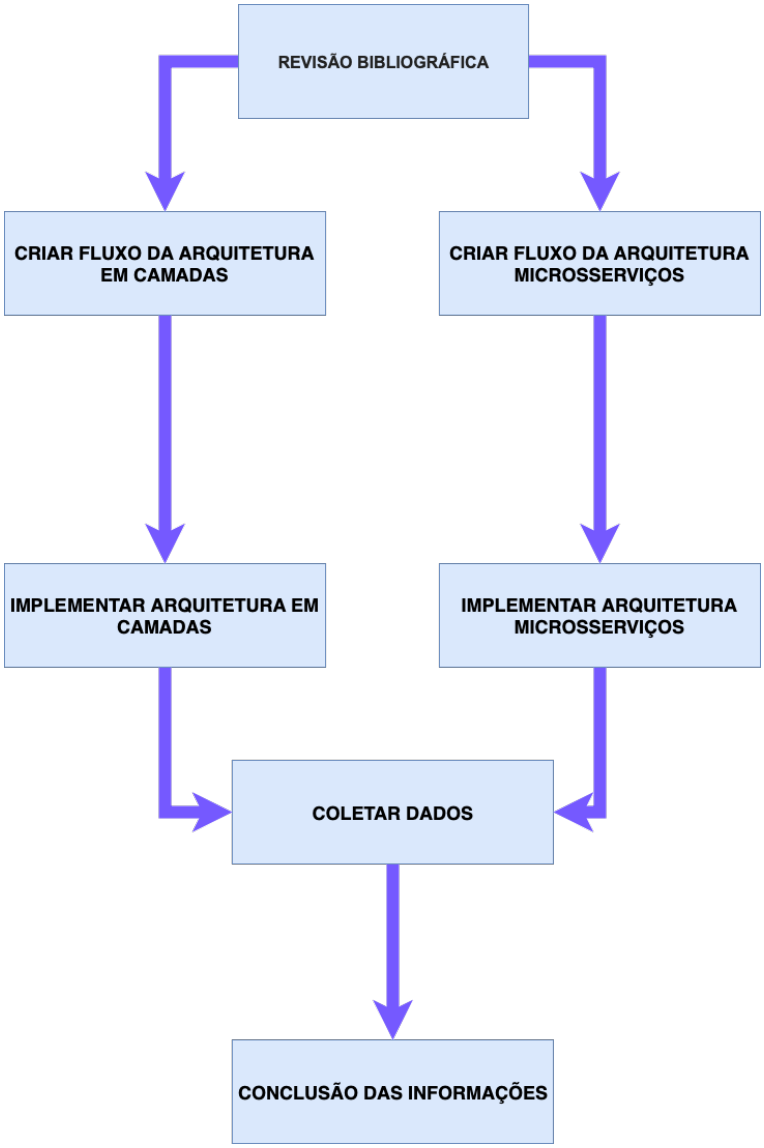
Serão obtidos os dados necessários para essa pesquisa através das pesquisas bibliográficas, da implementação e da observação comportamental do software diante de alguns problemas que serão colocados em um ambiente controlado. Com toda essa abordagem, poderemos descrever os resultados obtidos.

A pesquisa seguirá algumas fases, que serão resumidas na sequência:

1. Criação do fluxo das duas arquiteturas estudadas.
2. Implementação da arquitetura em camadas:
    - Coleta de dados durante a implementação
    - Realização de experimentos
    - Recuperação de informação
3. Implementação da arquitetura baseada em microsserviços:
    - Coleta de dados durante a implementação
    - Realização de experimentos
    - Recuperação de informação
4. Conclusão das informações em relação à:
    - Escalabilidade,
    - Manutenibilidade,
    - Implementação computacional, assim como
    - Vantagens e Desvantagens.

Na Figura 6 é possível observar o fluxograma da metodologia adotado no presente trabalho.

Figura 6 – Fluxograma da metodologia utilizada neste trabalho



Autoria própria(2019)

3.1 Fase 1 - Criação do fluxo das arquiteturas

O fluxo deve ser feito descrevendo a comunicação entre cada sistema ou tecnologia que será utilizado na arquitetura proposta.

Durante essa fase será realizado o levantamento de requisitos para as tecnologias que melhor se adéquem a arquitetura em camadas e a expertise do desenvolvedor. Com a definição dessa tecnologia, pode-se criar um fluxograma representando a comunicação entre as camadas, permitindo, de forma clara e simples, uma fácil visualização de todo processo para os interessados.

Com todas as ferramentas e tecnologias definidas, poderemos desenhar o fluxo da arquitetura em camadas e o fluxo da arquitetura baseada em Microsserviços.

3.2 Fase 2 - Implementação da arquitetura em camadas e arquitetura baseada em Microsserviços

Durante a fase de implementação, o objetivo é realmente desenvolver um sistema seguindo o fluxograma da fase 2. A implementação será feita pensando em um ambiente real, assim poderemos simular um ambiente de produção, com as tecnologias e ferramentas do dia a dia de trabalho de uma empresa.

3.3 Fase 3 - Coleta de dados

Durante a fase de implementação e também após a mesma, será feita a coleta de dados empírica como esforço para implementar as arquiteturas, manter, facilidade de extrair informações dos logs, tolerância a falhas dos softwares a problemas em determinadas partes do mesmo. E assim será feito os devidos experimentos e transformando os dados em informações para a geração de uma conclusão sobre cada arquitetura.

3.4 Fase 5 - Conclusão das informações

Com os dados já transformados em informações, o trabalho poderá ter uma conclusão real sobre a escalabilidade, manutenibilidade, a implementação de cada arquitetura e suas vantagens e desvantagens e outros insights.

4 Arquitetura em camadas

4.1 Criação do fluxo das arquiteturas

Como todo projeto a ser desenvolvido, precisamos planejar e analisar todas as tecnologias, fluxos e informações sobre o que desejamos fazer. No desenvolvimento de software não é diferente, precisamos criar um fluxograma da arquitetura.

4.1.0.1 Fluxograma da arquitetura em camadas

Como já descrito no capítulo 2 sobre a arquitetura em camadas, onde serão usadas as camadas de **Presentation Layer** , **Business Layer**, **Service Layer**, **Persistence Layer** e **Database Layer**, teremos em nosso projeto essas mesmas partes da arquitetura, porém chamaremos aqui de Visualização, Controle, Serviço, Modelo e banco de dados.

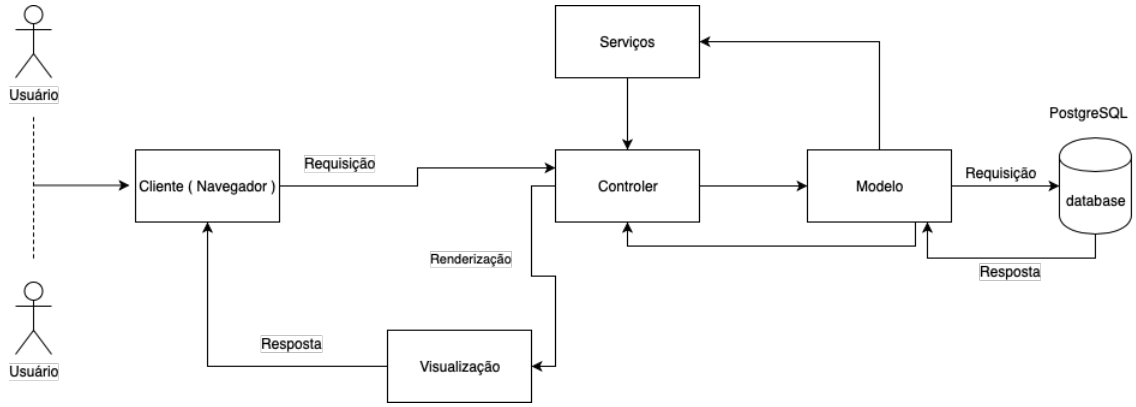
Será apresentada uma breve descrição do projeto desenvolvido. O sistema foi construído para uma Universidade, onde a quantidade de usuários é limitada a um público específico de alunos. Para poder participar do sistema ele terá que seguir alguns requisitos como ter coeficiente de rendimento 7.0 por exemplo. Em uma Universidade com 6000 alunos e 1000 professores, podemos analisar que também é um sistema sazonal. O sistema terá uma carga um pouco maior em tempos de edital aberto, logo o pico de acesso será maior durante períodos específicos e nesse caso a arquitetura em camadas soa ideal, não que para grandes acessos ela não seja uma boa escolha também, porém nesse caso em específico ela poderá ser mais fácil de se desenvolver. Nesse contexto é interessante deixar claro que não será um sistema de código aberto, haja visto que o sistema é de propriedade da Universidade e coordenado pela Doutora Annabell.

O projeto utilizará a linguagem de programação **Ruby** com o **framework Ruby on Rails**, e o banco de dados **PostgreSQL**, para armazenar nosso código, utilizaremos um repositório open source chamado Gitlab que nos ajudará a fazer um sistema automatizado com suporte a **continuous integration**, **continous deployment** e **continous delivery**. Utilizaremos o Heroku um famoso PaaS para não nos preocuparmos nesse momento com a infraestrutura.

4.1.1 Cliente (Navegador)

Toda interface que se pode visualizar os dados. Porém para a nossa aplicação utilizaremos os navegadores, mais especificamente o Chrome.

Figura 7 – UML mostrando o fluxo da arquitetura baseada em camadas, pensando que é uma única base de código



Autoria própria(2019)

4.1.2 Controller

O *controller* é a parte da aplicação que intermediará o modelo e a visualização.

4.1.3 Serviços

A camada de serviço é um objeto simples Ruby que encapsula um conjunto de lógica de negócios, movida do *controller* ou até mesmo do modelo para ser ter configurações e regras mais focadas.

4.1.4 Modelo

A camada de modelo nos permite escrever todas as regras de negócio de uma aplicação. É também a camada que permite se comunicar com o banco de dados, executando assim os famosos CRUDs.

4.1.5 Banco de dados

A camada de modelo tem acesso ao banco de dados através da ORM. O ORM *(Object Relational Mapper)* é um mapeador relacional de objetos. Isso significa que você não precisa chamar manualmente o banco de dados, o ORM lida com isso para você.

4.1.6 Visualização

A camada de visualização é responsável por pegar as informações que o controller e representar essas informações. É importante informar que a camada de visualização pode representar os dados em diversos formatos, alguns: XML, CSV, JSON e HTML.

4.2 Implementação da arquitetura em camadas

Como a arquitetura em camadas tende a utilizar todas as partes do software em um único projeto, temos aqui nas pastas do Ruby on Rails pastas bem definidas para essas camadas. Caso queira ter acesso ao código fonte o mesmo está no link: https://gitlab.com/rdxinho/pibict é necessário a solicitação de acesso.

4.2.1 Cliente (Navegador)

Como interface serão utilizados os navegadores para interagir com o sistema. O navegador faz uma requisição e recebe uma resposta e isso pode ficar mais claro utilizando o DevTools do Chrome. Na aba networking visualizamos diversas informações falaremos aqui algumas brevemente, pois não é o ponto focal do projeto. Nas figuras 9 8 10

Figura 8 – Lista de todos os arquivos carregados para renderizar a página

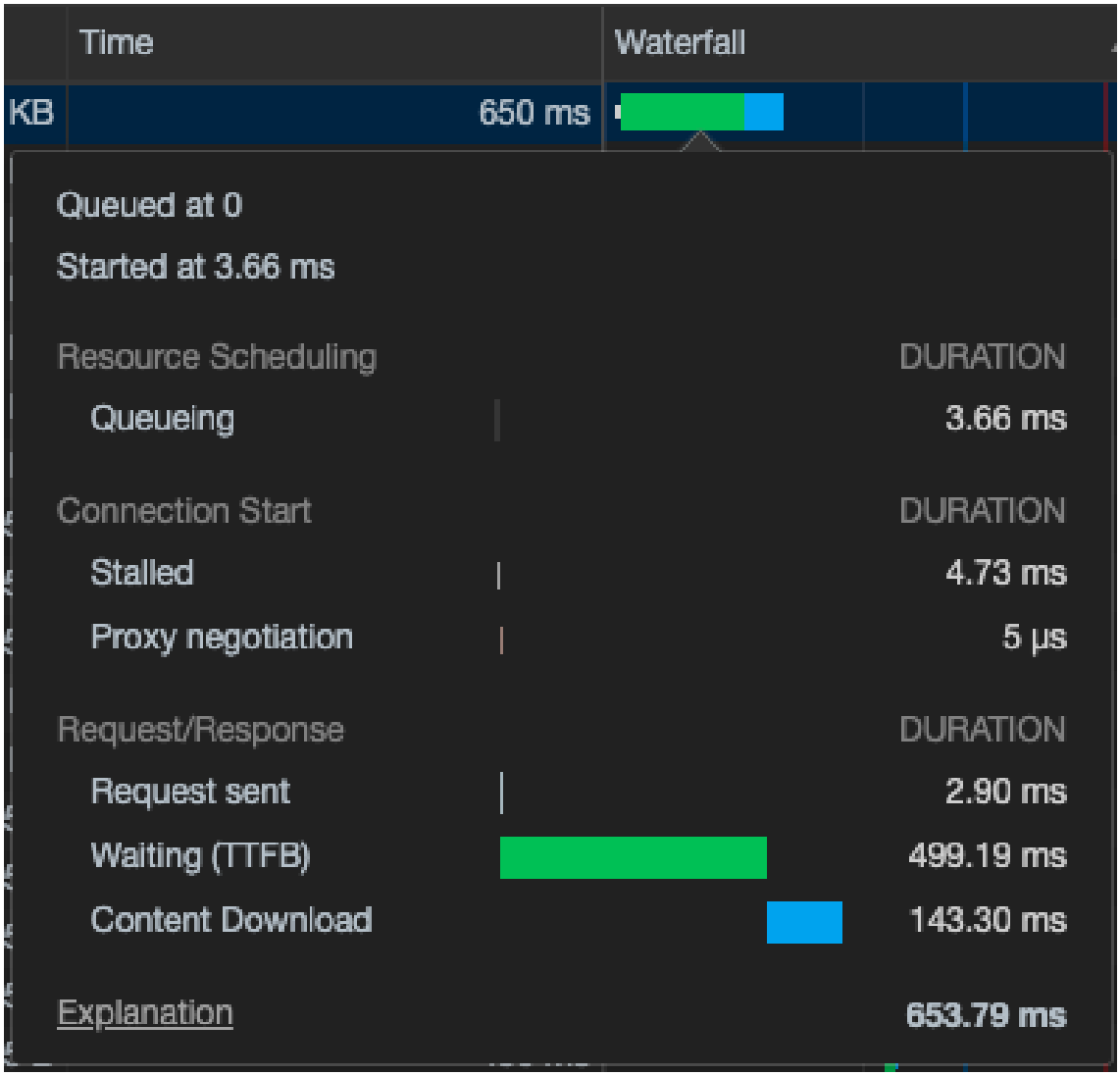
Name	Status	Type
sign_in	200	document
bootstrap.self-ec2bb108e35bf7fe2681f60d59b244bebf0c9ab672726c1324050e6257f3069e.css?body=1	200	stylesheet
rails.self-2f99d3db25a9c9a4f47ebc906cf41b64428530941a41fabf4af2de1bf9045780.css?body=1	200	stylesheet
select2.self-c6ac55e050e3e6db253b3e8c575de97675e07fade527056269d9fffdc88a988b.css?body=1	200	stylesheet
daterangepicker.self-fd69c2a932067e5697ca16542107bf3b883068c44db4ddfdac730a21e29b3777.css?body=1	200	stylesheet
style.self-2a73bf114de4d6f721c9b1e69a9a95aed750528b3addcaf0c8699806b990ae56.css?body=1	200	stylesheet
components.self-9790bb1d6a540e1115b03febedf0e978853159b4260e1df80d4b0a67ff032cbf.css?body=1	200	stylesheet
buildings.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
courses.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
laboratories.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
loading.self-46af2c633c150e56b3da52714623b4b4e45f44d6343ea4bfbd6cf57a8660f42c.css?body=1	200	stylesheet
printing.self-4c8a72ff3c4a00ae11f2b8147a143d3f063b7e6e51dfa5ad29ecfc1561deae78.css?body=1	200	stylesheet
project_generals.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
projects.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
public_notices.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
scholarships.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
student_profiles.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
subscriptions.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
teacher_profiles.self-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css?body=1	200	stylesheet
application.self-0a5bef07b742e5d7c9908cc875170c3c16e7b397dcaf504e73c07ff1cf2711c8.css?body=1	200	stylesheet
css?family=Nunito:400,600,700,800	307	
css?family=Nunito:400,600,700,800	200	stylesheet
XRXV3I6Li01BKofINeaBTMnFcQ.woff2	200	font
XRXW3I6Li01BKofAjsOUYevIWzgPDA.woff2	200	font
XRXW3I6Li01BKofA6sKUYevIWzgPDA.woff2	200	font
favicon.ico	200	vnd.microsoft.icon

Autoria própria(2019)

4.2.2 Controller

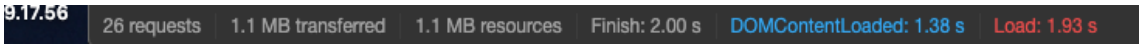
O controller são responsáveis por orquestrar o modelo e a visualização. No framework Rails utilizamos uma pasta em **app/controllers** para adicionar todos os 'orquestradores' da nossa aplicação. É a responsável por receber um request e responder a um request.

Figura 9 – Gráfico de carregamento dos arquivos



Autoria própria(2019)

Figura 10 – Barra de status com algumas informações



Autoria própria(2019)

Figura 11 – Um código simples para ilustrar o código de um controller

```
class SubscriptionsController < ApplicationController
  before_action :set_subscription, only: %i[show edit update destroy]

  def index
    @search = Subscription.reverse_chronologically.ransack(params[:q])

    respond_to do |format|
      format.any(:html, :json) {      You, a few seconds ago • Uncommitted ch
        @subscriptions = set_page_and_extract_portion_from @search.result
      }
      format.csv { render csv: @search.result }
    end
  end
```

Autoria própria(2019)

4.2.3 Serviços

Como a arquitetura baseada no modelo Model, View e Controller(MVC) enfrenta alguns problemas, adicionamos algumas camadas extras como por exemplo a camada de Serviços. Podemos criar diversos serviços com regras definidas baseada em uma determinada lógica de domínio. O controller tem uma responsabilidade muito bem definida, que é receber a requisição, verificar o que será feito, receber aquele dado e responder ao cliente. Quando o controller começa a ter lógica, já começamos a ver um espaço para a utilização de uma camada extra, que é a camada de serviço. Podemos verificar um código de serviço na figura 12.

4.2.4 Modelo

O Ruby on Rails também nos dá uma estrutura de pasta pronta para receber os modelos. Essa estrutura fica em **app/models**.

4.2.5 Banco de dados

O projeto utilizará o banco de dados PostgreSQL.

4.2.6 Visualização

Os dados serão representados utilizando o HTML, CSS e Javascript. O Ruby on Rails nos dá também o ERB para dinamicamente podermos fazer essas junção dos dados retornados pelo controller.

Figura 12 – Um código simples para ilustrar o código de um serviço

```ruby
# app/services/tweet_creator.rb
class TweetCreator
  def initialize(message)
    @message = message
  end

  def send_tweet
    client = Twitter::REST::Client.new do |config|
      config.consumer_key        = ENV['TWITTER_CONSUMER_KEY']
      config.consumer_secret     = ENV['TWITTER_CONSUMER_SECRET']
      config.access_token        = ENV['TWITTER_ACCESS_TOKEN']
      config.access_token_secret = ENV['TWITTER_ACCESS_SECRET']
    end
    client.update(@message)
  end
end
```

Fonte: Imagem retirada do link https://www.toptal.com/ruby-on-rails/rails-service-objects-tutorial

Figura 13 – Um código simples para ilustrar o código do modelo

app > models > subscription.rb

You, 12 days ago | 1 author (You)

```ruby
# frozen_string_literal: true

class Subscription < ApplicationRecord
  belongs_to :public_notice
  belongs_to :teacher_profile
  belongs_to :student_profile
  belongs_to :project

  mount_uploader :asset_registration_form, RegistrationFormUploader # ficha cadastral
  mount_uploader :asset_statement, StatementUploader # declaração de vínculo
end
```

Fonte: Imagem retirada do link https://www.toptal.com/ruby-on-rails/rails-service-objects-tutorial

5 Arquitetura microsserviços

5.1 Criação do fluxo das arquiteturas

Como todo projeto a ser desenvolvido, necessita-se planejar e analisar todas as tecnologias, fluxos e informações sobre o que desejamos fazer. No desenvolvimento de software não é diferente, precisa-se criar um fluxograma da arquitetura.

O planejamento da arquitetura baseada em microsserviços seguirá os mesmos passos já feito para a arquitetura em camadas. O fluxograma tem uma visão macro da comunicação entre os sistemas e na fase de implementação seguiremos o fluxograma e modificaremos ao longo do processo.

5.1.0.1 Fluxograma da arquitetura baseada em microsserviços

O projeto baseado nesse tipo de arquitetura segue a utilização de diversos serviços distintos que se comunicam entre si como pode ser verificar no fluxograma da **??**.

O projeto chamado aqui tem assedio nos dará dados estatísticos sobre o assédio moral ou sexual nas Universidades, e também será uma aplicação colaborativa, onde não se sabe a quantidade de usuários atendidos e quantos desenvolvedores irão colaborar com esse projeto. Então, uma arquitetura baseada em microsserviços poderá ser uma escolha ótima escolha, haja visto que poderemos contar com diversas comunidades, seja em termos de desenvolvimento de novas funcionalidades com as funcionalidades que poderão ser feitas em qualquer tecnologias e também em termos de acesso por diversos departamentos ou instâncias da sociedade. Os departamento de segurança, Universidades, empresas de seguro com o intuito de analisar os riscos sobre uma determinada apólice ou mesmo a população, tendo em vista que com informações reais a cobrança será muito mais bem fundamentada para cobrar os governantes.

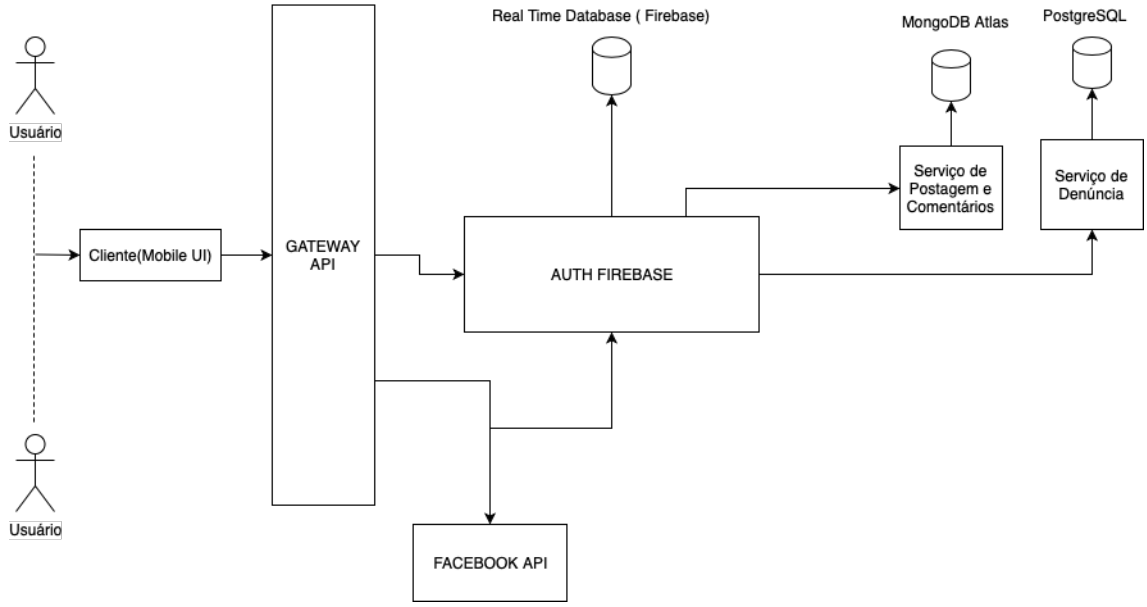
5.1.1 Cliente (Mobile UI)

A arquitetura proposta precisa de uma interface de entrada de dados para iteração dos usuários. O usuário executa uma ação no cliente que no nosso caso é um aplicativo mobile e inicia o fluxo que vai para a parte do API Gateway.

5.1.2 API Gateway

Quando se desenvolve um sistema baseado na arquitetura de microsserviços alguns problemas aparecem, como por exemplo, precisa-se ter um cliente, que poderia ser um

Figura 14 – UML mostrando o fluxo da arquitetura baseada em microsserviços



Autoria própria(2019)

outro sistema, aplicativo, página web ou qualquer outra forma de consumir uma API conhecer diversos endpoints e versões das APIs e não seria uma forma transparente como um monolítico é, então criamos um microsserviços para cuidar dessa parte de redirecionar, verificar a autorização e a permissão do usuário. Imagino um cenário de uma arquitetura baseada em microsserviços com uns 20 sistemas, como você faria para fazer a autorização e a autenticação dos seus usuários? Seria bem complexo, correto? Porém com um único ponto de entrada você torna esse processo mais simples.

5.1.3 Serviço de Postagem e Comentários

Os microsserviços são totalmente isolados e por isso cada parte tem o seu próprio banco de dados. Como podemos analisar na figura **??**. O serviço de postagem e comentários tem o contexto de cadastrar, excluir, deletar ou atualizar um comentário ou uma postagem. Como a postagem está totalmente acoplada aos comentários que foram feitas nele. Por exemplo, uma postagem que vamos dar o nome de **postagem1** está ligado aos comentários que vamos chamar de **comentario**, **comentario2** e **comentario**. Nesse contexto não faria sentido termos dois serviços um postagem e o outro para comentários, visto que eles fazem parte de um contexto definido.

5.1.4 Serviço de Denúncia

O serviço para denúncias também faz parte de um contexto próprio com seu banco de dados também próprio. As informações de cadastro e acompanhamento da denúncia será feita totalmente por esse serviço. O usuário poderá em um passo a passo fazer a denúncia do ocorrido e a plataforma se encarregará de enviar essas informações para um órgão competente, por exemplo nas Universidades.

5.2 Implementação da Arquitetura

O código da implementação dos microsserviços seguem abaixo:

1. API Users - https://github.com/rodolfopeixoto/obsidium-users-api
2. API Gateway - https://github.com/rodolfopeixoto/obsidium-gateway-api
3. Interface Mobile - https://github.com/amog-oliveira/obsidium/tree/develop
4. PostIt - https://github.com/rodolfopeixoto/obsidium-postit

5.2.1 Cliente (MOBILE UI)

O cliente foi desenvolvido em React Native como já informado no capítulo 2, é uma tecnologia muito utilizada para se criar aplicativos nativos com uma maior produtividade de reutilização de código para duas ou até três plataformas distintas, como por exemplo Web, IOS e Android.

5.2.2 API Gateway

A API Gateway foi desenvolvida em NodeJS com Express. A API Gateway é responsável por fazer o redirecionamento paras as rotas corretas dos microsserviços, entretanto, antes da comunicação entre ela e outros serviços, é feita uma verificação de autenticação e a autorização, onde há uma comunicação com um outro serviço especializado em usuários e autenticação e autorização de forma transparente.

5.2.2.1 Serviço de usuário (Firebase e API Facebook)

O Firebase tem um importante papel no nosso sistema, pois ele nos da um sistema de autorização e autenticação pronto e um banco de dados para armazenar essas informações. Com ele podemos verificar se usuário através de sua interface pode acessar determinado recurso ou não.

A API do Facebook tem o papel de facilitar a vida dos usuários que querem acessar o sistema rapidamente, sem perder tempo com cadastros longos ou com senhas que se perdem ao longo do tempo.

5.2.3 Serviço de Postagem e Comentários

O serviço de postagem e comentários foi desenvolvido também em NodeJS e Express, porém o banco de dados utilizado é o banco MongoDB Atlas. O sistema utiliza um banco de dados NoSQL e remoto, pois o banco de dados Atlas é um cloud com uma interface de fácil configuração e implementação, com o banco de dados remoto, não precisamos nos preocupar com a infra-estrutura mantida.

5.2.4 Serviço de Denúncia

O sistema de serviço de denúncias foi desenvolvido em Ruby on Rails com PostgreSQl e ambos containerizados e hospedados no heroku. É uma API relativamente simples que cadastra as denúncias e retorna as mesmas.

Conclusão

Durante o desenvolvimento deste trabalho de Monografia, a maior dificuldade talvez tenha sido na escrita e no entendimento do como mostrar o problema e a hipótese que se queria trabalhar. O objetivo deste trabalho de monografia foi atingido a partir da implementação computacional das duas arquiteturas apresentadas, tendo uma experiência prática para gerar os resultados em quanto a escalabilidade, manutenibilidade e uso de multi-tecnologias. Em relação a uma melhoras deste trabalho, acredita-se necessário evoluir a arquitetura e testar outras formas como por exemplo *performance*, comunicação via mensagens, assim como outras abordagens da arquitetura baseada em microsserviços.

Com a experiência do desenvolvimento da arquitetura em camadas e também da baseada em microsserviços podemos listar alguns pontos sobre ambas as arquiteturas. A implementação baseada em microsserviços se mostrou bem mais complexa e demorada, haja visto que você precisa construir diversas camadas, duplica código e sua implementação exige cautela. A monolítica é muito mais simples, pois tudo que você precisa está disponível em um só lugar.

Em termos de escalabilidade ambas arquiteturas se comportam bem, porém com uma arquitetura baseada em microsserviços você tem a facilidade de escalar um determinado contexto que tenha se tornado um gargalo, o que gera também diminuição de custos. Uma arquitetura monolítica, você terá a necessidade de levantar a base toda do código, muitas vezes o gargalo é em uma determinada funcionalidade e mesmo assim, teríamos que levantar toda a base de código.

Já do ponto de vista da manutenção, a arquitetura monolítica mostrou-se ser um pouco mais complexa, visto que muitas vezes se faz necessário entender algumas regras mais abrangentes. A arquitetura de microsserviços se mostrou mais simples de evoluir e manter, haja vista que são contextos menores e se há uma falha naquele contexto, vamos direto na parte que está com *bug*.

Há complexidade em adicionar novas linguagens de programação ou frameworks em uma arquitetura monolítica, haja visto que esse tipo de arquitetura não foi projetada com esse intuito, porém é possível. Com a arquitetura baseada em microsserviços ficou claro que é mais fácil de se desenvolver contextos em tecnologias que favoreçam a resolução daquele contexto, pois haverá pequenos projetos com complexidades menores para se manter. Por exemplo, na abordagem criada em microsserviços, a textitAPI Gateway foi implementada para um contexto específico, ser a porta de entrada das requisições e o mesmo ser o único sistema a se comunicar com os outros sistemas internos. Para a resolução desse problema o NodeJS foi a escolha, porém poderiamos utilizar o framework Ruby on Rails.

Já em relação ao deploy também na arquitetura baseada em microsserviço foi mais rápido, porém mais complexa, pois são diversos sistemas. A vantagem da arquitetura em camadas é que, a mesma, é enviada uma única vez; como o sistema exemplo utilizado neste trabalho ainda é pequeno o deploy mesmo sendo mais demorado que a baseada em microsserviços, ainda sim é muito rápido.

Um grande problema da arquitetura baseada em microsserviços são os logs descentralizados e precisamos aqui recorrer a um software de terceiros como elastic(ELK) ou fluentd, mais um software que demanda manutenção. Com o monolítico não é necessário, visto que os logs já estão centralizados na aplicação.

Em relação à autorização e autenticação, a arquitetura monolítica é mais simples e rápida a implementação, pois no exemplo deste trabalho utilizamos uma gem chamada *devise* a configuramos facilmente. Já no microsserviços há todo um contexto para se preocupar, mesmo com o Firebase nos facilitando de desenvolver a parte de autenticação.

A arquitetura monolítica se for bem projetada há possibilidade de projetar uma arquitetura dessas para ser anti-falhas em termos de código e de forma mais manual. Mas de modo geral, caso deixe de funcionar uma parte da aplicação, normalmente será um efeito colateral, por exemplo o banco de dados por algum motivo ter um problema, a aplicação inteira sofre esse efeito.

A arquitetura baseada em microsserviços consegue lidar muito bem com esse problema, caso não seja a parte central como a de usuário, Autenticação, autorização ou a API Gateway no caso desse projeto. Como são partes centrais, elas sofrem efeitos em cadeia e você não terá acesso ao sistema. Mas caso seja um problema na parte de comentários e posts, não teríamos problema para acessar outras partes do software, pois iria carregar normalmente em nossa aplicação mobile.

Para essa comparação podemos analisar que depende muito do que estaríamos descrevendo sobre tolerância a falhas.

De forma geral e para concluir este trabalho, é importante entender que não há melhor arquitetura, mas sim a que mais vai se adequar a aquele projeto e equipe. Diante de tantas informações que foi possível extrair desse projeto, acredito que o melhor é obter a compreensão que as arquiteturas quando bem estudadas e aplicadas de forma correta poderá diminuir e aumentar a produtividade da equipe em resolver problemas.

5.3 Trabalhos Futuros

Este trabalho buscou comparar as arquitetura monolíticas e microsserviços, porém há diversas outras que poderiam ser exploradas. Entretanto há métricas que poderão serem utilizadas, como a performance entre as arquiteturas em termos de request e response ou

na utilização de testes automatizados, com a analize voltada a perfomance dos testes entre cada arquitetura. O desenvolvimento da arquitetura microsserviços com formas de comunicação diferente, por exemplo com a utilização do HTTP e a utilização da Advanced Message Queuing Protocol (AMQP) por exemplo.

Referências

BAARSEN, J. van. GitLab Cookbook. [S.l.]: Packt Publishing, 2018. v. 1. Citado na página 28.

BATISTA, F. de A. Arquitetura de microserviços: Uma solução leve para grandes sistemas no futuro. Anais do Encontro Nacional de Pós-Graduação - VII, v. 2, n. 2, 2018. Citado 4 vezes nas páginas 19, 20, 22 e 23.

BEER, B. Introducing GitHub. 2. ed. [S.l.]: O'REILLY, 2018. v. 1. Citado na página 28.

BENNETT ANDREA BISHOP, B. D. J. W. G. K. S. Implementing web 2.0 technologies in higher education: A collective case study. ELSEVIER, v. 1, 2011. Citado na página 14.

BOJINOV, D. H. D. R. V. Node.js Complete Reference Guide. [S.l.]: Packt Publishing, 2018. v. 1. Citado na página 28.

CERNY, M. J. D. e. M. T. T. Contextual understanding of microservice architecture: Current and future directions. APPLIED COMPUTING REVIEW, n. 4, 2017. Citado 2 vezes nas páginas 22 e 30.

CHIARADIA, L. F. C. Uma proposta de arquitetura de microsserviços aplicada em um sistema de crm social. COMMUNICATIONS OF THE ACM, v. 23, n. 53, 2018. Citado 2 vezes nas páginas 14 e 30.

EVANGELISTA WELLINGTON GALDINO, S. N. J. Modelo de avaliação da capacidade das organizações da administração pública federal para a adoção de software as a service (saas) público. Revista do Serviço Público, v. 67, n. 2, 2015. Citado na página 24.

FORD, M. R. N. Fundamentals of Software Architecture. [S.l.]: O'Reilly Media, Inc., 2020. v. 1. Citado na página 19.

GHIROTTI, T. R. S. E.; RENTZ, A. Tracking and controlling microservice dependencies. COMMUNICATIONS OF THE ACM, v. 61, n. 11, 2018. Citado na página 31.

GOVERNOR, D. H. J.; NICKULL, D. Web 2.0 Architectures. [S.l.]: O'REILLY, 2009. Citado na página 14.

HANJURA, A. Heroku Cloud Application Development. [S.l.]: Packt Publishing, 2014. v. 1. Citado 3 vezes nas páginas 24, 25 e 26.

HARBER, S. H. C. Getting MEAN with Mongo, Express, Angular, and Node. [S.l.]: Manning Publications, 2019. v. 1. Citado na página 23.

JUBA, A. V. S. Learning PostgreSQL 10. 2. ed. [S.l.]: Packt Publishing, 2018. v. 1. Citado na página 29.

KILLALEA, T. The hidden dividends of microservices. COMMUNICATIONS OF THE ACM, v. 59, n. 8, 2016. Citado 2 vezes nas páginas 23 e 30.

KON, P. C. B. e F. A importância dos testes automatizados. Engenharia de Software Magazine, v. 1, n. 3, 2008. Citado 2 vezes nas páginas 27 e 28.

MOREIRA, D. M. B. P. F. M. Desenvolvimento de aplicações e micro serviços: Um estudo de caso. Tecnologias, Infraestrutura e Software, v. 4, n. 1, 2015. Citado na página 30.

NGUYEN, L. F. E. R. S. D. S. D. V. Evolve the Monolith to Microservices with Java and Node. [S.l.]: IBM Redbooks, 2016. v. 1. Citado 2 vezes nas páginas 21 e 22.

OUVERNEY, D. M. K. G. Automação do processo de implantação de software usando docker e microserviços - um estudo de caso. OPEN JOURNAL SYSTEMS, v. 3, n. 1, 2017. Citado na página 27.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. SCITEPRESS – Science and Technology Publications, v. 1, 2016. Citado na página 30.

RAJI ALOK HOTA, T. H. M.; HUANG, J. Scientific visualization as a microservice. JOURNAL OF LATEX CLASS FILES, v. 14, 2015. Citado na página 31.

RICHARDS, M. Software Architecture Patterns Understanding Common Architecture Patterns and When to Use Them. [S.l.]: O'REILLY, 2015. v. 1. Citado 7 vezes nas páginas 14, 15, 16, 18, 19, 20 e 21.

SCHMIDT, F.; MACDONELL, A. M. C. S. Multi-objective reconstruction of software architecture. International Journal of Software Engineering and Knowledge Engineering, v. 28, n. 6, 2018. Citado 3 vezes nas páginas 15, 16 e 17.

SIZO ADRIANO DEL PINO LINO, E. L. F. A. M. Uma proposta de arquitetura de software para construção e integração de ambientes virtuais de aprendizagem. RISTI, v. 1, n. 6, 2010. Citado na página 18.

SOUSA, L. O. M. e. J. C. M. F. R. C. Computação em núvem conceitos, tecnologias e aplicações. [S.l.]: ERCEMAPI, 2010. Citado na página 24.

TANENBAUM, A. S. W. A. S. Sistemas Operacionais: Projetjos e Implementação. [S.l.]: Bookman, 1944. Citado na página 14.

YAHIAOUI, H. Firebase Cookbook. [S.l.]: O'REILLY, 2017. v. 1. Citado 2 vezes nas páginas 29 e 30.

ZAMMETTI, F. Practical React Native: Build Two Full Projects and One Full Game using React Native. [S.l.]: Apress, 2018. v. 1. Citado na página 29.

ZHU LEN BASS, G. C.-S. L. Devops and its practices. IEEE Software, v. 33, n. 3, 2016. Citado 3 vezes nas páginas 22, 26 e 27.

Apêndices

APÊNDICE A – Areas de Conhecimento CNPq

Tabela Completa em :
<http://www.cnpq.br/documents/10157/186158/TabeladeAreasdoConhecimento.pdf>

Código	Área de Conhecimento
10000003	CIÊNCIAS EXATAS E DA TERRA
1.03.00.00-7	**Ciência da Computação**
1.03.01.00-3	*Teoria da Computação*
1.03.01.01-1	Computabilidade e Modelos de Computação
1.03.01.02-0	Linguagem Formais e Automatos
1.03.01.03-8	Análise de Algoritmos e Complexidade de Computação
1.03.01.04-6	Lógicas e Semântica de Programas
1.03.02.00-0	*Matemática da Computação*
1.03.02.01-8	Matemática Simbólica
1.03.02.02-6	Modelos Analíticos e de Simulação
1.03.03.00-6	*Metodologia e Técnicas da Computação*
1.03.03.01-4	Linguagens de Programação
1.03.03.02-2	Engenharia de Software
1.03.03.03-0	Banco de Dados
1.03.03.04-9	Sistemas de Informação
1.03.03.05-7	Processamento Gráfico (Graphics)
1.03.04.00-2	*Sistemas de Computação*
1.03.04.01-0	Hardware
1.03.04.02-9	Arquitetura de Sistemas de Computação
1.03.04.03-7	Software Básico
1.03.04.04-5	Teleinformática