Fundação Educacional do Município de Assis
IMESA - Instituto Municipal de Ensino Superior de Assis

ANDRÉ VIEIRA DA SILVA

Desenvolvimento de aplicações com base na arquitetura de micro serviços

2016

ASSIS-SP

Fundação Educacional do Município de Assis
IMESA - Instituto Municipal de Ensino Superior de Assis

ANDRÉ VIEIRA DA SILVA

Desenvolvimento de aplicações com base na arquitetura de micro serviços

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito de Curso de Graduação

Aluno: André Vieira da Silva

Orientador: Dr. Almir Rogério Camolesi

Linha de Pesquisa: Informática

2016

ASSIS-SP

AGRADECIMENTOS

Agradeço a Deus por fazer tudo dar certo desde de sempre que mesmo nas dificuldades sempre iluminou meus pensamentos e fez que conseguisse chegar até esta etapa.

Ao Drº Almir Rogério Camolesi, meu orientador que me dizia para não desanimar quando informava que estava com dificuldades, onde muitas vezes perdi meu foco devido à complexidade do trabalho, ele sempre acreditou que era capaz e me fez enxergar que tudo que precisava era acreditar que era capaz.

Agradecer a minha mãe que sempre me via estudando e correndo atrás do meu sonho que era fazer uma faculdade de tecnologia. Ela sempre rezou por mim e sempre, sempre deixava tudo pronto para eu ir para faculdade e estudar. E sempre me fala para não desistir e não desanimar.

RESUMO

O desenvolvimento deste projeto mostrará como funciona na teoria e pratica a arquitetura de micro serviços e o porquê de grandes empresas estarem adotando este tipo de arquitetura.

Mostrando as tecnologias envolvidas e como elas se relacionam para atingir um objetivo.

Palavras chaves: Arquitetura, serviço, micro, monolítico

ABSTRACT

The development of this project will show how it works in theory and practice microservice architecture and why large companies are adopting this type of architecture.

Showing the technologies involved and how they relate to an end.

Keywords: Architecture, service, micro, monolithic

LISTA DE FIGURAS

Sumário

1) INTRODUÇÃO

(FOWLER, 2014) Diz que, a arquitetura de micro serviço é uma maneira de projetar as aplicações de software como suítes de serviços de forma independente e implementável.

(FOWLER, 2014) Descreve a arquitetura de micro serviço como um conjunto de pequenos serviços, sendo que, cada um possui seu próprio processo e se comunicando através de mecanismos leves utilizando recursos HTTP. Seus *deploy*[1] são independentes e automatizados e seus serviços podem ser desenvolvidos em diferentes linguagens de programação e diferentes tecnologias de armazenamento de dados.

(MARTINS, 2015) Destaca uma das grandes vantagens do micro serviço está na sua independência e nas suas linhas de código reduzida por apenas deixar disponível um serviço para uma causa maior, facilitando o monitoramento para desempenho e caso o serviço esteja defeituoso sua substituição não afeta outros blocos da solução, assim como acontece em sistema monolítico.

Computação em nuvem (Cloud Computing) é fundamental para uma grande empresa que tem o objetivo de reduzir os custos operacionais e se desvincular de data center locais, permitindo utilizar diversas aplicações via internet em qualquer lugar do mundo e independente de qualquer plataforma. Sendo assim, deixando a velha estrutura de sistema local, dando segurança e qualidade na prestação desse serviço principalmente quando esta se tratando em banco de dados. As vantagens são significativas quando se pensa em segurança, qualidade e escalabilidade. (YOSHIDA, 2015)

As principais empresas que disponibilizam o serviço cloud são Windows Azure da MicroSoft, AWS da Amazon, Gloogle Cloud Plataform da empresa Google e SoftLayer da IBM.

---

[1] Processo de instalar aplicação no servidor deixando disponível para o uso do usuário

1.1) OBJETIVO

Esse trabalho tem como objetivo principal abordar o conceito de arquitetura de micro serviços e suas influencias nas grandes empresas, visando suas vantagens e desvantagens. Assim como, mostrar as tecnologias envolvidas para o funcionamento e como é feito a programação utilizando esta arquitetura.

1.2) JUSTIFICATIVA

Segundo (RICHARDSON, 2014), para grandes aplicações faz mais sentido usar uma arquitetura de micro serviços, que divide a aplicação em um conjunto de serviços individuais ficando mais fácil de entender e ser desenvolvido, assim como implantados de forma independente. Possibilitando adotar novas tecnologias e frameworks[2]. Ao desenvolver micro serviços é necessário lidar com alguns problemas complexos de gerenciamento de dados distribuídos, apesar dos obstáculos essa arquitetura é adequada para aplicações complexas e de grande porte que estão evoluindo rapidamente, especialmente para aplicações do tipo SaaS[3].

Dessa forma, é de se concluir que esse novo paradigma possui sua complexidade, mas seus benefícios são muito significativos para grandes empresas e as tecnologias envolvidas para a integração desses serviços como REST[4], Cloud Computing faz ter boas expectativas sobre esse novo paradigma.

---

[2] Uma solução para uma família de problemas semelhantes que usa um conjunto de classes e interfaces para decompor o problema.

[3] Software consumido como serviço sem necessidade de instalação ou qualquer tipo de configuração. O fornecedor de serviço cobra uma taxa para disponibilizar o serviço e dar suporte sendo utilizado 100% através da internet

[4] Utilização de JSON ou XML para representar os dados associados a um recurso

1.3) MOTIVAÇÃO

(FOWLER, 2014) Descreve arquitetura de micro serviço como um novo paradigma de programação. O conceito de pensar que os componentes são serviços implementável independentes entre os blocos envolvidos na solução.

(RICHARDSON, 2014) Diz que este tipo de arquitetura é adotado quando o sistema monolítico atingi suas limitações, onde sua deficiência gera grandes prejuízos para grandes empresas. Empresas como Amazon, Ebay e Groupon que adotaram este tipo de arquitetura devido ao grande fluxo de acesso e a grande solução foi a utilização de micro serviços com a cloud computing para aumentar sua estrutura e escalabilidade.

1.4) ESTRUTURA DO TRABALHO

Este trabalho está organizado em quatro capítulos, expondo os pré-requisitos de forma estruturada para melhor compreensão da leitura.

No primeiro capítulo é apresentado a introdução do trabalho, onde é levantado de forma superficial o contexto da pesquisa.

Em seguida o capitulo dois Fundamentos Teóricos é responsável de abordar o conteúdo do trabalho de forma objetiva. Sua estrutura é composta de pré-requisitos para ter uma melhor entendimento do tema Arquitetura de micro serviços. Neste capitulo é exposto a arquitetura de micro serviços em um sistema e toda a influência nela contida, e qual requisitos são necessários abordar para se ter sucesso neste tipo de arquitetura.

No capitulo três Desenvolvimento do Projeto, será mostrado uma solução exemplo de como será trabalhado a arquitetura de micro serviço em um projeto real, após ter adquirido os conceitos no capítulo anterior.

No capitulo quatro Conclusão Final mostrará de uma forma clara e objetiva o ponto de vista de usar esta arquitetura nos dias de hoje.

2) FUNDAMENTAÇÕES TEORICAS

Neste capitulo será apresentado os conceitos desta pesquisa abordando os fundamentos envolvidos para o entendimento da arquitetura de micro serviços, assim como, as opções que ficam disponíveis ao utilizar este tipo de arquitetura, mostrando as vantagens e desvantagens e como obter um melhor aproveitamento de recurso utilizando micro serviços.

2.1) ARQUITETURA MONOLÍTICA

Um sistema na arquitetura monólito possui suas funcionalidades em um único sistema. (ALMEIDA, 2015) Essas funcionalidades dependem uma da outra para seu funcionamento, ou seja, todas as funções do sistema monolítico são fortemente acopladas umas às outras e compartilham a mesma base de dados. Como mostra a Figura 1 abaixo.

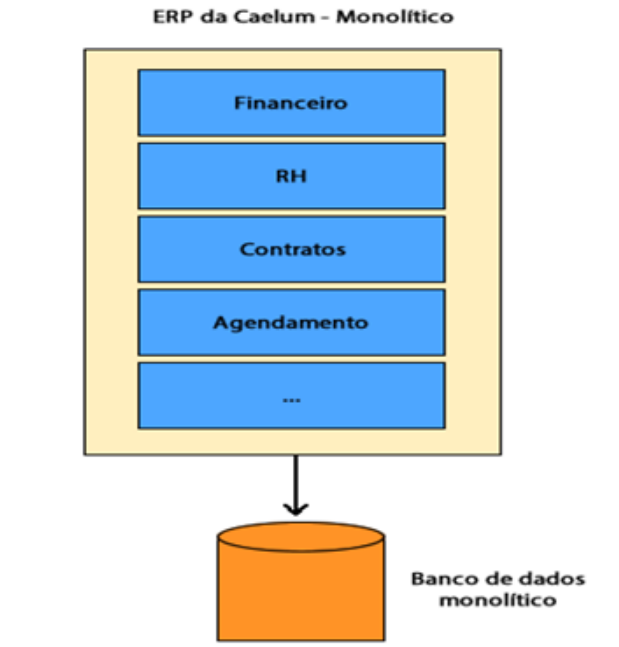


Figura 1 - Arquitetura Monolítica ERP Caelum

(ALMEIDA,2014)

Na arquitetura monolítica a alteração de algum componente ou erro acaba impactando em todo restante fazendo que todo o sistema pare de funcionar. (MARTINS, 2015)

Segundo (MACEDO, 2015) os desenvolvedores de um sistema monólito precisam conhecer o sistema como um todo para que as alterações definidas para o contexto não “quebrem” as linhas de código. Ou seja, com as alterações feitas é necessário testar o sistema como um todo para garantir a integridade do sistema.

A arquitetura monolítica é uma arquitetura clássica utilizada por muitos anos, os desenvolvedores utilizam um código fonte comum. Esta arquitetura possui código fonte muito extenso e complexo e entender a sua complexidade é a grande dificuldade para adicionar desenvolvedores no projeto que segue a arquitetura monolítica. (NOBRE, 2015).

2.2) ARQUITETURA DE MICRO SERVIÇO

“A arquitetura de micro serviços é fundamentada na criação de um conjunto de APIs[5] e componentes muito pequenos, com baixa capacidade funcional” (NOBRE, 2015).

(MAURO, 2015) Desenvolver na arquitetura de micro serviço é ter um contexto delimitado e autossuficiente para fins de desenvolvimento de software. É entender e atualizar o código de um micro serviço sem ter nenhuma informação do interior do micro serviço correspondente, pois a interação entre eles é via APIs não compartilhando estrutura de dados, esquemas de banco de dados, ou outras representações internas do objeto.

[5] API é um conjunto de rotinas e padrões de programação para acesso a um aplicativo de software ou plataforma baseado na Web.

O assunto de micro serviço começou a se popularizar no final de 2004 e desde de 2006 veio ganhando força como mostra o resultado do Google Trends. Ilustrada na figura a baixo.

Figura 2 - Interesses no assunto Micro Serviço

Segundo (LITTLE, 2014) o tema ganhou destaque na QCon San Francisco 2012, ThoughtWorks através de James Lewis e com a publicação do artigo de Martin Fowler.

(FOWLER, 2014) Tem como objetivo em seu artigo explicar as principais ideias e princípios sobre micro serviço. Sendo que, o estilo arquitetônico de micro serviço é uma ideia importante para aplicações empresariais. Empresas como Netflix, The Guardian, o Governo do Reino Unido Serviço Digital entre outras utilizam esta arquitetura e vê com a experiência adquirida como positiva em comparação com aplicações monolíticas.

Segundo (NEWMAN, 2015) utilizar micro serviço aumenta significativamente a liberdade para reagir e tomar decisões diferentes na solução, possibilitando responder de forma mais rápida as mudanças inevitáveis que causam impactos no dia a dia. Um dos princípios de micro serviço é reunir as coisas que mudam pela mesma razão, e separar as coisas que mudam por razões diferentes.

(FOWLER, 2014) Defini a arquitetura de micro serviços como uma aplicação de um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicam utilizando recursos HTTP. Esses serviços podem ser escritos em diferentes linguagens de programação e usar diferente tecnologias de armazenamento de dados como mostra a Figura 2.

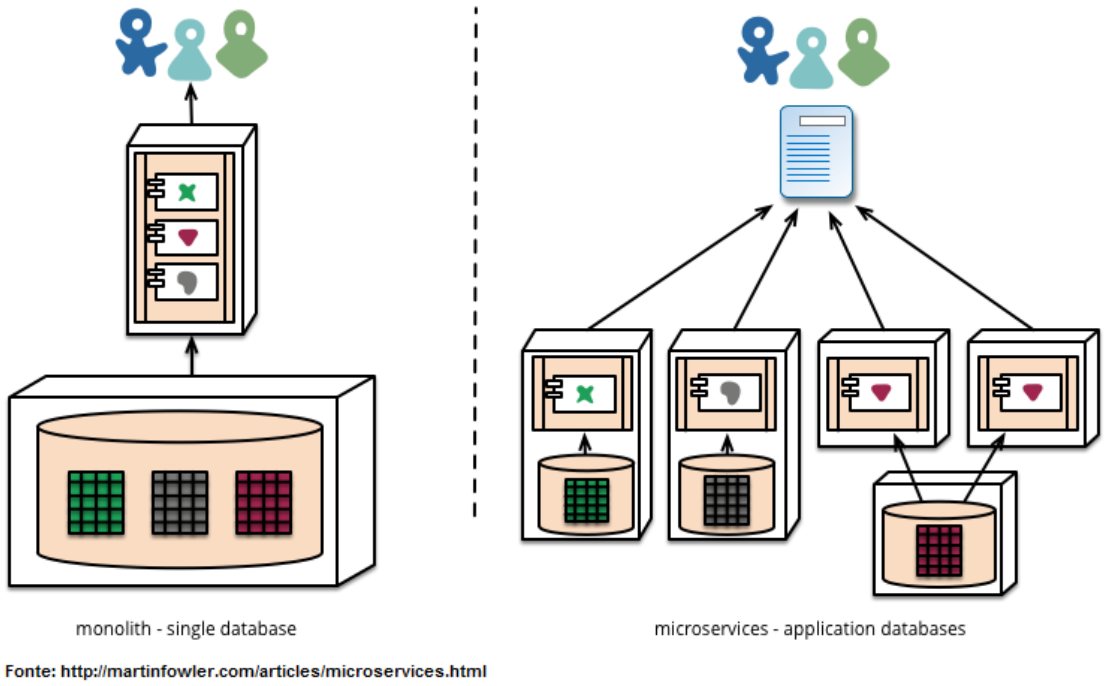


Figura 3 - Estrutura de arquitetura monolítico e micro serviços

Segundo (MAURO, 2015) na existência de vários pequenos serviços específicos, a necessidade de atualizar todos os serviços para o funcionamento da solução, não os defini como micro serviços, porque estão fracamente acoplados. Os serviços não podem se comunicar com o mesmo banco de dados é necessário dividir o banco de dados deixando-os independentes.

(NEWMAN, 2015) Micro serviço permite adotar a tecnologias mais rapidamente, e compreender como novos avanços podem melhorar. Uma das maiores barreiras para experimento de novas tecnologias são os riscos associados. Em uma aplicação monolítica ao abordar uma nova linguagem de programação, banco de dados, ou estrutura, qualquer mudança terá um grande impacto no

sistema. Já com um sistema constituído por múltiplos serviços, existe vários novos lugares para experimentar novas tecnologias podendo escolher qual serviço ocasionara menos impacto.

Com autonomia do serviço qualquer mudança necessária não irá comprometer o sistema como um todo, ou seja, a solução continuará em processo mesmo que algum serviço não esteja disponível.

As características de uma arquitetura de micro serviço não possui uma definição formal, porem se espera que na utilização de micro serviços algumas características sejam comuns na estruturação do seu desenvolvimento. (FOWLER, 2014)

Uma das características de micro serviço é a componentização através de serviços.

Um componente pode ser definido como uma unidade de software representada independentemente, substituível e atualizável. Em micro serviço o componente é quebrado em pequenos serviços que se comunica com solicitação de serviço web ou remoto. Sua implantação e processo são independentes do software e autônomo. Com essa autonomia trabalhar com micro serviços acaba sendo vantajoso, pois, a substituição do componente pode ser feita sem comprometer toda aplicação como acontece na arquitetura monolítica, ou seja, minimizar o impacto para as mudanças.

(MARTINS, 2015) Diz que na utilização de arquitetura de micro serviços os códigos são extremamente reduzidos diferente do monólito. Isso ocorre porque na arquitetura de micro serviço não é necessário entender todo o contexto da solução, mas sim uma pequena parte dela, a parte que faz sentido para o contexto. Devido à redução de código a entrega acaba se tornando continua ocorrendo mudanças constantes no serviço.

(MACEDO, 2015) Com a utilização de micro serviço dentro de uma empresa a divisão de times se torna possível devido a autonomia e independência entre o micro serviço, algo que seria impossível no monólito deixando várias equipes responsáveis pela mesma base de código sem que quebre suas linhas de programação. A mudança de uma arquitetura monolítica para micro serviço

acaba facilitando para adicionar pessoas ao projeto, devido a separação de contexto. Com contexto reduzido entender o problema acaba sendo facilitado, algo que seria oposto na arquitetura monolítica que é necessário entender toda a solução para garantir que o contexto não influencie todo o restante, sendo assim, adicionar desenvolvedores ao time de programação acaba se tornando uma vantagem em micro serviço. Como mostra a figura a baixo:

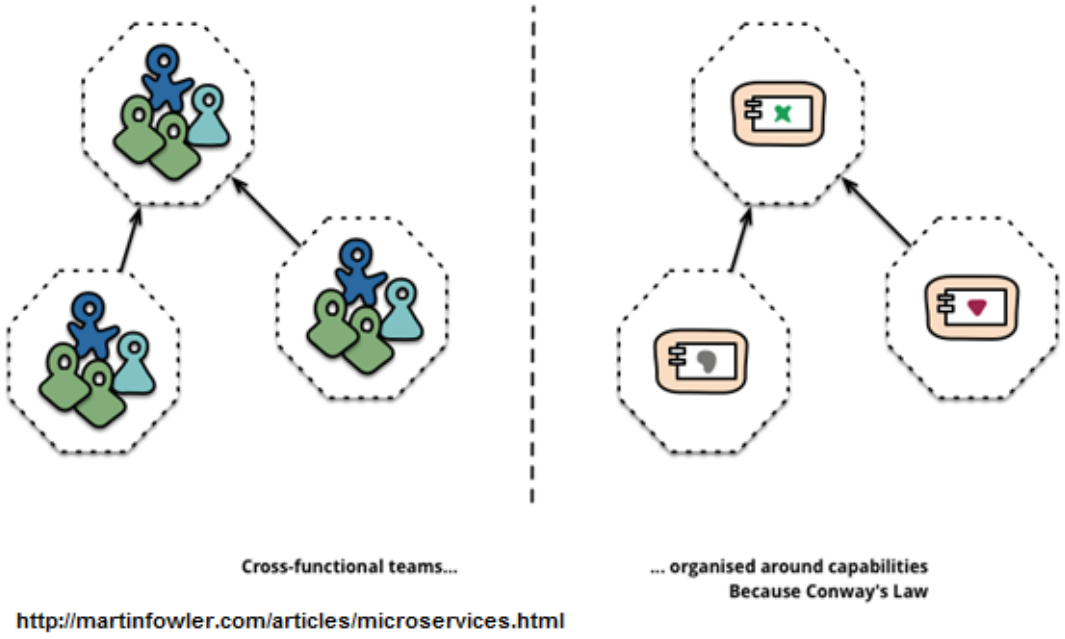


Figura 4 - Equipes para contextos específicos

Em micro serviço cada equipe é responsável pela construção e sua operação. (FOWLER, 2014)

(SAKURAI, 2015) Deu como exemplo a empresa Gilt que utiliza essa arquitetura de micro serviços para sua regra de negócio de “vendas relâmpagos” onde oferece descontos a partir de uma hora especifica. A empresa Gilt em 2009 teve um sobrecarga devido ao aumento considerável em seu trafego. Como utilizavam um sistema com arquitetura monolítica os desenvolvedores trabalhavam na mesma base de código e a publicação de atualizações poderia demorar mais de duas semanas para testar todas as integrações e se caso

desse errado tinha dificuldades para localizar a raiz do problema. A solução foi migrar a arquitetura monolítica para uma arquitetura macro/micro serviço.

Um macro serviço trata de um domínio em especifico e um micro serviço é o resultado que se obtém em quebrar um macro serviço, dividindo em serviços menores.

(SAKURAI, 2015) Fala que na utilização desse processo migratório de arquitetura a empresa Gilt criou 10 macros serviços para o núcleo de negócio. Com a construção desses serviços foi adicionado uma base de dados dedicada para cada serviço, utilizando base de dados especificas para cada necessidade. A utilização dessa arquitetura resolveu 99% dos problemas de escalabilidade. Porem esses serviços eram semi monolíticos e existia uma carência no responsável pelo código. A solução foi quebrar esses macros serviços em micro serviços dando responsabilidade para cada equipe de desenvolvimento testa-los publica-los e monitora-los. Sendo assim, cada equipe se torna dona do seu serviço e não uma pessoa responsável pelo código.

Segundo (MAURO, 2015) na utilização de micro serviço não há tempo para cometer erros, devido as mudanças necessárias e constantes, o ritmo de produção é definido como “ofegante” para proporcionar experiências inovadoras para os clientes. Por fim defende que arquitetura de micro serviço é o futuro.

2.2.1) RESILIENCIA NA ARQUITETURA DE MICRO SERVIÇO

A resiliência é um fator importante quando se trata de arquitetura de micro serviço. Segundo (MARTINS, 2015) a utilização da arquitetura facilita na estratégia de resiliência da aplicação devido os serviços trabalharem isoladamente permitindo que o sistema como um todo continue operacional.

Segundo (BIRD, 2015) a resiliência tem que ser concebida e construída sobre cada serviço. Um serviço pode consumir outro(s) serviços, no entanto, não podem ficar refém deles, ou seja, antecipar a falhas de outros serviços, definir uma estratégia em caso de falha e adicionar uma forma defensiva para garantir

sua integridade. Criar um serviço para minimizar o impacto caso aconteça a falha de outros serviços, para tornar mais fácil e mais rápido a recuperação do serviço ou reiniciar caso seja necessário.

(MARTINS, 2015) Diz que a empresa Netflix revolucionou no requisito resiliência quando se trata de micro serviços. As falhas acontecem, e a eficiência de reagir a ela sendo eficaz é o que determina o sucesso da solução.

Segundo (HEMEL, 2013) A empresa NetFlix criou uma ferramenta chamada Chaos Monkey que derruba servidores intencionalmente para testar a tolerância a falhas no ambiente em nuvem e monitora a recuperação.

2.2.2) GOVERNANCIA DESCENTRALIZADA

Uma das características da arquitetura de micro serviços é a governança descentralizada. Segundo (FOWLER, 2014) com a governança descentralizada o desenvolvedor tem como opção utilizar tecnologias heterogenias, independentes para cada micro serviço. Essa escolha pode ser de linguagens específicas para determinado contexto, assim como um banco de dados apropriado, ou até mesmo escolha do sistema operacional.

Na arquitetura de micro serviço, cada serviço possui seu próprio banco de dados, ou seja, existe uma descentralização na gestão de dados. Como ilustrado na Figura 2.

Segundo (FOWLER, 2014) tratando de uma forma mais abstrata a descentralização de gestão de dados, o modelo conceitual dos dados será tratado de forma diferente por cada serviço. Por exemplo no dado de cliente para o setor de vendas existe um contexto, porem analisando mesmo dado no setor de SAC (Serviço de atendimento ao cliente) pode existir um outro contexto. Ainda segundo (FOWLER, 2014) uma maneira sutil para fazer a divisão e mapeamento de contexto é utilizar o DDD (Domain-Driver Design) no qual divide um domínio complexo em múltiplos contexto delimitando e mapeando as relações entre eles. Este processo é útil para arquitetura monolítica e de micro serviço.

A arquitetura de micro serviço permite que cada serviço gere sua própria base de dados, seja da diferentes instancias da mesma tecnologia da base dados ou sistemas de banco de dados completamente diferentes conhecida como **Polyglot Persistence.**[6]

Segundo (NOBRE, 2015) uma das grandes complexidades de utilizar a arquitetura de micro serviço é a duplicação de dados e a consistência dele entre os serviços. Esse é um grande desafio para empresas de como tratar a consistência de dados.

2.2.3) AUTOMATIZAÇÃO DE PRODUÇÃO

A utilização de micro serviço faz com que o código fique significativamente reduzido. Com essa redução entender o contexto se torna menos complexo e fazer alterações de um serviço acaba se tornando frequente.

Com o processo de deploy frequente é necessário testar o serviço antes de subir para o usuário e como suas entregas são continua, fazer o processo de teste manualmente acaba se tornando uma contradição na arquitetura de micro serviço.

(MACEDO, 2015) Desenvolvedor da InfoGlobo mostrou os processos de evolução da empresa Rede Globo no requisito automação de produção. Pois, devido a entrega continua de projeto a empresa tinha um desempenho insatisfatório que comprometia toda a eficiência da equipe de desenvolvimento por causa de formulários que eram preenchidos manualmente justificando as alterações de serviço. Uma das soluções para agilizar o processo foi a criação do deploy blue green.

O projeto deploy blue green são ambientes de produção idênticos, existindo dois servidores que alternam entre si, enquanto um está operacional e todo processo

[6] Capacidade de utilizar diferentes tecnologias de persistência de banco, onde existe um entendimento comum.

de trafego é direcionada para ele o outro fica ocioso. Quando o software atinge a fase final de teste o servidor ocioso recebe essas atualizações é trocado o link simbólico e toda aplicação é direcionada para o servidor atualizado.

(MARTINS, 2015) Diz que a arquitetura de micro serviço não é recomendado para qualquer empresa, devido a maturidade de teste de produção automatizado. Uma dica dada pelos especialistas para quem quer utilizar arquitetura de micro serviço é fazer a aplicação monolítica para conhecer bem suas regras de negócio e assim conhecer as fronteiras para criação de micro serviço. Como mostra a figura a baixo:

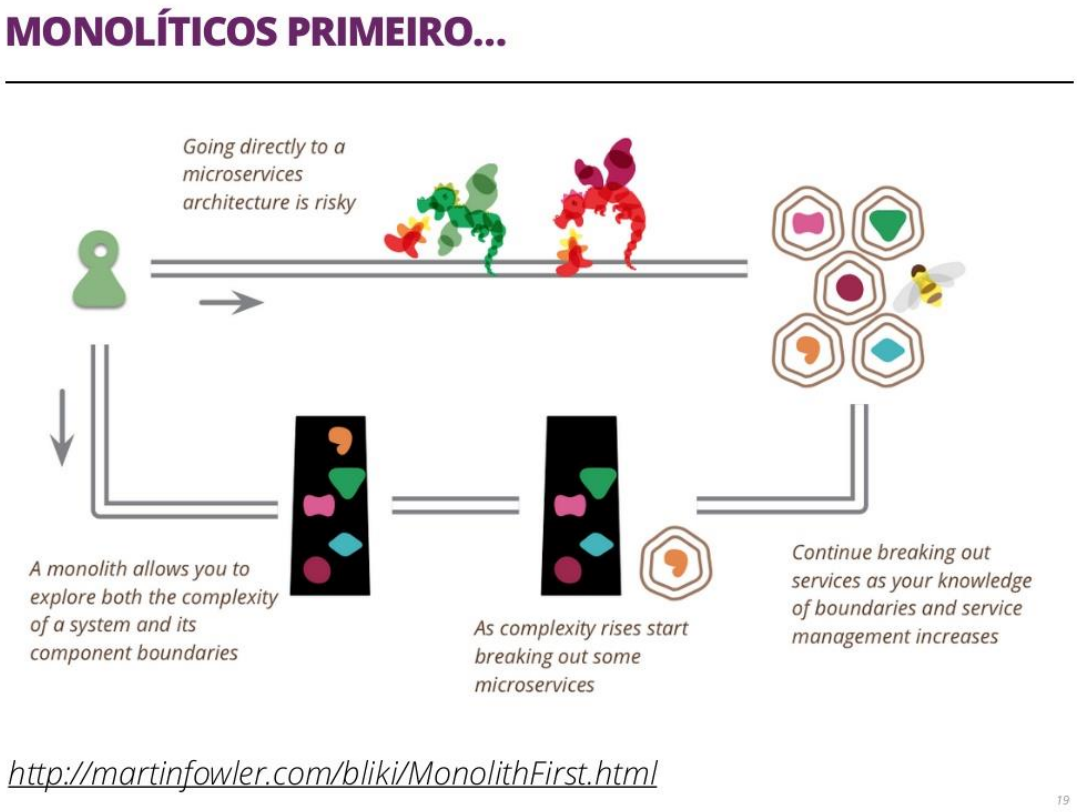


Figura 5 - Evolução monolítica para micro serviço

(FOWLER, 2014) Da o exemplo do site Guardian onde sua estrutura foi projetada e construída como monolítico, porem veio evoluindo na direção de micro serviços. O núcleo do site ainda é monólito, no entanto, eles adicionam novos recursos através da construção de micro serviço que utilizam API do monólito.

2.2.4) MONITORAMENTO DE UM MICRO SERVIÇO

O objetivo da arquitetura de micro serviço é fazer que todos os serviços funcionem de forma harmoniosa para alcançar um proposito. Porem nem tudo sai como planejado e as coisas podem falhar, traçar uma estratégia de resiliência é muito importante, no entanto, é necessário saber onde está a raiz do problema para poder ser eficiente e eficaz.

(FOWLER, 2014) Diz que é importante ser capaz de detectar as falhas rapidamente e restabelecer automaticamente o serviço. Monitorar em tempo real uma aplicação de micro serviço em diversos elementos da arquitetura (como pedidos por segundo), métricas relevantes (encomendas por minuto que são recebidas) e monitoramento de semântica onde se caso algo der errado o sistema aciona as equipes de desenvolvimento para acompanhar e investigar. Tudo isso é necessário pois existe uma colaboração entre os serviços e corrigi-los é emergencial. Seria de se esperar um monitoramento com registros sofisticados que trate individualmente cada serviço, como painéis mostrando os status, métricas operacionais, rendimentos e latência.

(SAKURAI, 2015) Afirma que a empresa Gilt usa diversas ferramentas OpenSource[7] para monitorar seus micros serviço como: New Relic, Boundary e Graphite e também desenvolveu o seu próprio sistema de monitoramento chamado Cave onde é configurado as regras e cria alertas para seus desenvolvedores.

(BIRD, 2015) O monitoramento é uma ferramenta importante dos testes realizados em uma implantação continua. Principalmente utilizados em testes ao vivo utilizando técnicas como Canary Release (onde uma pequena parcela dos usuários é direcionada para a nova versão do serviço).

[7] Código fonte de licenciamento livre e que pode ser adaptado para diferentes fins.

2.2.5) SOA (ARQUITETURA ORIENTADA A SERVIÇO) X MICRO SERVIÇO

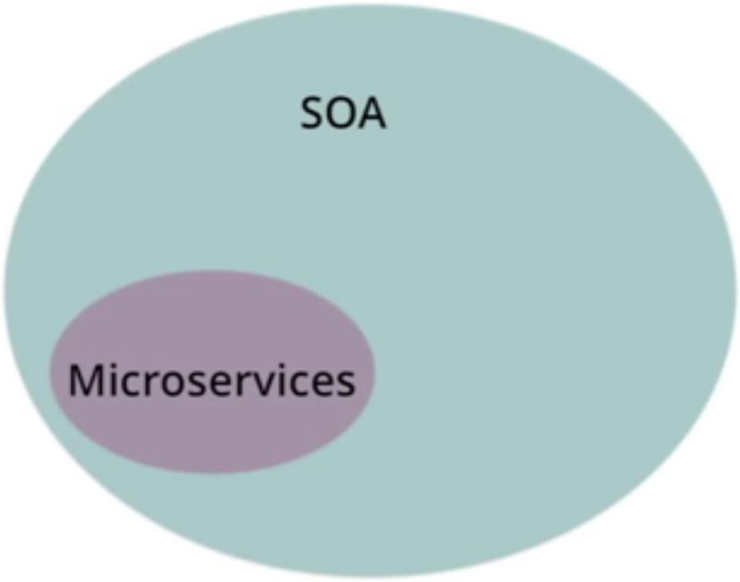
Segundo (CORDEIRO, 2012) SOA surgiu em 1996 no artigo “Service Oriented Architectures” feito pelos pesquisadores Roy Schulte e Yefim Natis do Gartner Group.

(LASKEY, 2006) Defini SOA como um paradigma para organizar e utilizar capacidade distribuídas que podem estar em diferentes domínios.

Segundo (STENBERG, 2014) os softwares eram construídos de forma monolítica, logo após migraram para a estratégia SOA, onde encontraram problemas e na atualidade os softwares estão migrando para arquitetura de micro serviços.

(LITTLE, 2014) Diz em seu artigo que ao longo dos últimos anos, o assunto sobre micro serviço aumentou. Depois da apresentação na QCon San Francisco 2012, ThoughtWorks com James Lewis e o co-autor Martin Fowler sobre o tema micro serviço. No entanto, Steve Jovens argumentou que micro serviço não é algo inovador é apenas um conceito novo para SOA.

(WEISSMANN, 2014) Afirma que micro serviço não é algo novo e que a arquitetura de micro serviço não passa de uma subclassificação de SOA. Micro serviço é uma forma mais “light” de implementar SOA com menos elementos removendo o ESB. Como apresentado na Figura 6:



Fonte: http://stackoverflow.com/questions/25501098/difference-between-microservices-architecture-and-soa

Figura 6 - Relação SOA com micro serviço

(RHUBART, 2015) Demonstra em seu artigo a abordagem de micro serviços e SOA como formas diferentes para resolver problemas. SOA é uma estratégia que muda a toda estrutura de TI da empresa, separando em diversos serviços tornando-a flexível que se expande em diferentes aplicações e sistemas envolvendo diferentes unidades organizacionais. Porém, micro serviço é estruturar uma aplicação envolvendo apenas a equipe responsável por seu desenvolvimento. Micro serviço deve ser independentemente destacável, enquanto SOA é implementado muitas vezes em implantações monolíticas.

Segundo (RHUBART, 2015) pessoas que utilizam SOA ganham uma nova liberdade através da adaptação das ideias de micro serviço. Em micro serviço existe a independência de tecnologia, possibilitando substituição de tecnologias antigas por mais novas, sem ter que substituir toda aplicação. O SOA clássico é mais influenciado em plataforma, de modo que micro serviço pode oferecer mais opções em todas as dimensões. Micro serviço acabou se tornando uma forma de restaurar a flexibilidade que pode ter sido perdida pelo SOA que se tornou demasiadamente rígido e monolítico.

Segundo (FOWLER, 2014) o assunto micro serviço comparado com SOA é algo comum porque são estilos semelhantes. Uma das grandes diferenças entre

micro serviço em relação a SOA é o foco em ESBs usados para interagir em aplicações monolíticas utilizados por SOA. Como mostra a Figura 7:

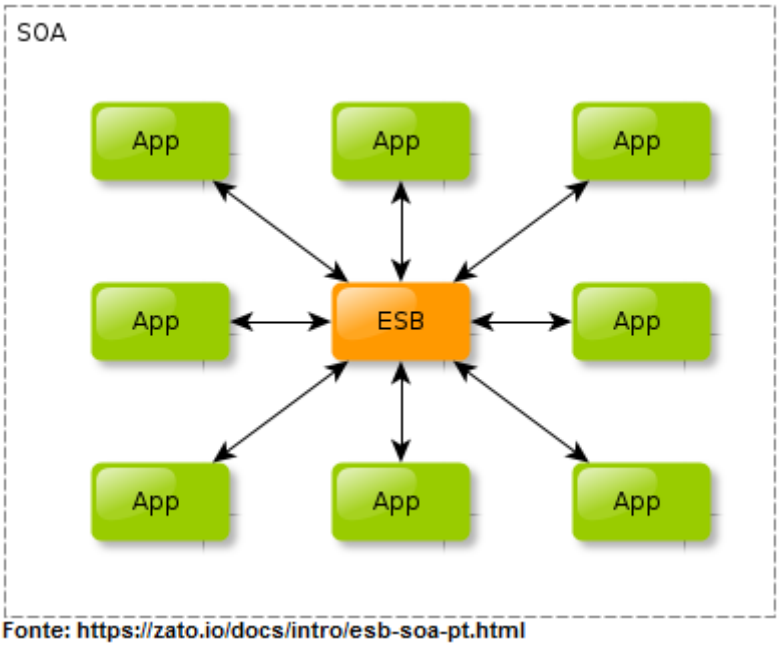


Figura 7 - Arquitetura SOA com ESB

(STENBERG, 2014) Utiliza a definição de Russ Mile comparando software monolítico como pedregulhos difíceis de quebrar em porções menores que são difíceis de mudar ou mover. Essas mudanças são relacionadas a melhorias continuas. SOA como rochas, que ainda pelo seu tamanho é difícil de mudar e não dá o retorno desejado. E micro serviços são como cristais, muito fácil de mudar. Com a união de coisas pequenas podem fazer um trabalho maior, onde cada porção tem um proposito. Ou seja, componentes atômicos e mais simples focando na evolução e mudanças no componente.

(MARTINS, 2015) Defini a anatomia de um micro serviço entre domínio da fronteira de negócio que o micro serviço é responsável, exposição de recursos como protocolos de comunicação, camada de persistência de dados e camada externa para comunicação com outros serviços. Anatomia é ilustrada na Figura 8:

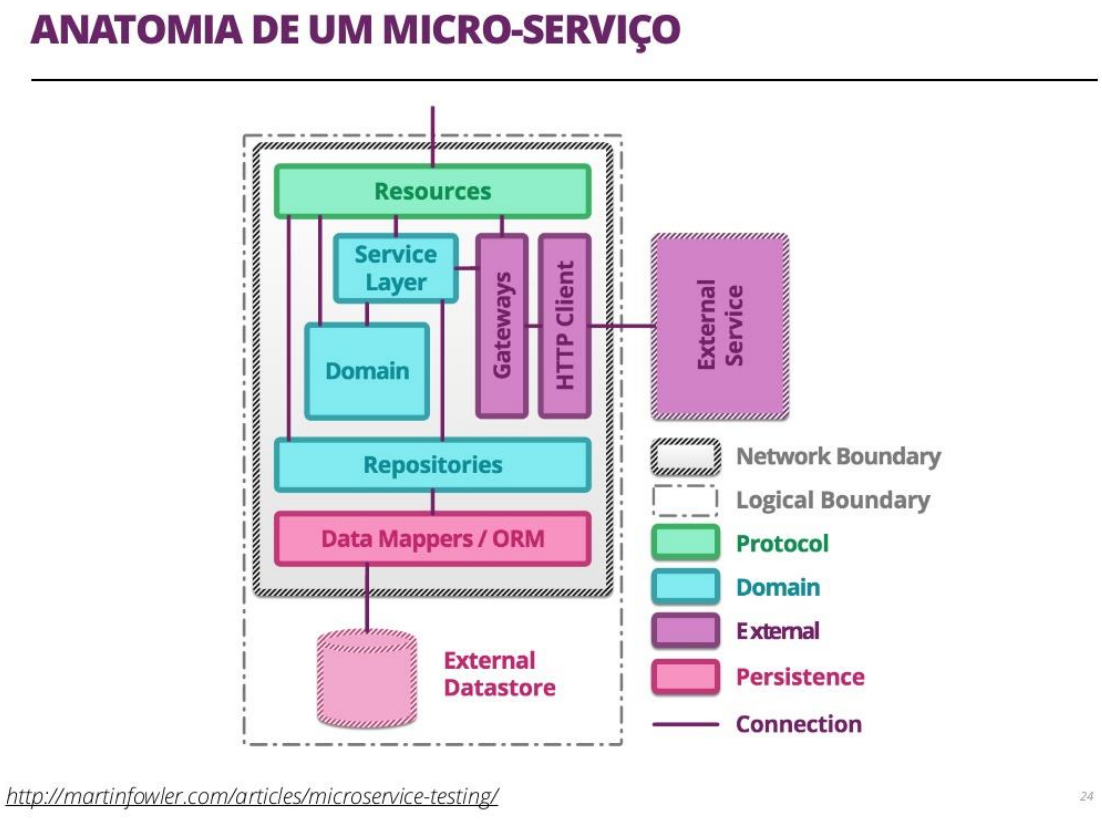


Figura 8 - Anatomia de um micro serviço

(MARTINS, 2015) Micro serviço são extremamente pequenos e focados em fazer apenas uma coisa bem-feita. Para diminuir o impacto no código ao redor. Como micro serviço são extremamente pequenos escalar os serviços necessários acaba se tornando algo vantajoso.

(SAKURAI e CASCARROLHO, 2015) Diz que na Arquitetura SOA uma grande parte do sistema é responsável por diversas funcionalidades e responsável por vários domínios. Na arquitetura de micro serviço é uma forma segmentada desse cenário. Ou seja, SOA seria composto de vários micros serviços.

Segundo (SAKURAI e CASCARROLHO, 2015) utilizando arquitetura de micro serviço, processos que precisam de recursos computacionais específicos podem ser escalados individualmente em servidores específicos para otimizar a necessidade da solução, diferente da arquitetura monolítica que precisava subir a aplicação por inteira sendo necessário mais recurso computacional.

2.2.6) A INFLUENCIA DA COMPUTAÇÃO EM NUVEM

(ALECRIM, 2008) Diz que a expressão cloud computing (Computação em nuvem) começou em 2008, porem as ideias da denominação já existem há muito tempo. A computação em nuvem permite utilizar em qualquer lugar e independente de qualquer plataforma, as diversas aplicações por meio da internet com as mesmas facilidades em aplicações instaladas em computadores locais.

Segundo (VELTE, VELTE e ELSENPETER, 2012) a função da computação em nuvem é cortar custos operacionais permitindo ao departamento de TI (Tecnologia da Informação) se concentrarem nas estratégias de projetos ao invés de manter data center operacionais.

(FERNANDES, 2012) Defini a computação em nuvem como um serviço ao invés de produto, onde recursos compartilhados, software e informações são fornecidas, onde o acesso pode ser feito por diferentes plataformas como tablet, celulares e computadores.

(LAURO, 2013) Diz que o foco principal da computação em nuvem é sua elasticidade, onde pode ser moldada conforme a necessidade. Antigamente empresas que precisavam de mais recursos computacionais perdiam horas com seus técnicos para montar a estrutura e subir os sistemas operacionais e deixar tudo operacional, hoje com a computação em nuvem com poucos comandos é possível aumentar seus recursos de estrutura e subir suas aplicações.

A infraestrutura da nuvem é conceituada em SaaS (Software como serviço), PaaS (Plataforma como serviço) e Iaas (Infraestrutura como serviço). Como mostrada na Figura 9:

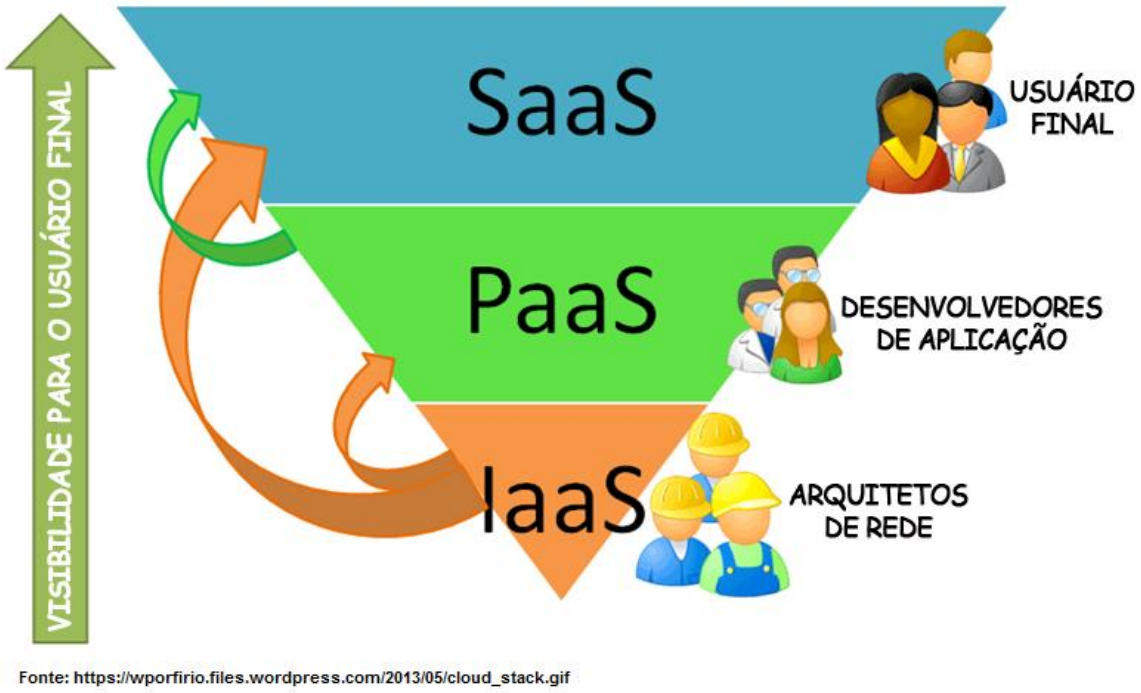


Figura 9 - Camadas da Computação em Nuvem

(ALECRIM, 2008) Defini “Software as a Service” (SaaS) como um software que é contratado por um tempo periódico sem a necessidade de uma licença para instalação. Ele é pago somente pelos recursos utilizados e pelo tempo de uso. “Platform as a Service” (PaaS) é voltada para desenvolvedores incluindo os recursos necessários à operação, como armazenamento, banco de dados, escalabilidade (aumento automático da capacidade de armazenamento ou processamento), suporte a linguagens de programação, segurança entre outros. “Infrastructure as a Service” (IaaS) seu foco é na estrutura de hardware ou de maquinas virtuais, com o usuário podendo acessar a recursos do sistema operacional. “Database as a Service (DaaS)” é direcionada ao fornecimento de serviços para armazenamento e acesso de volume de dados. Onde a vantagem é a flexibilidade para expandir o banco de dados, compartilhar as informações com outros sistemas, facilitando o acesso remoto para usuários autorizados. “Testing as a Service” (TaaS) oferece um ambiente para testar aplicações de maneira remota, simulando o comportamento em nível de execução.

Segundo (RICHARDSON, 2014) para utilizar de forma eficaz, a arquitetura de micro serviço exige um nível de automação, como um PaaS. E para aplicações complexas e de grande porte que estão evoluindo rapidamente que utilizam SaaS, micro serviço é a arquitetura mais adequada para este tipo de aplicação.

(SAKURAI e CASCARROLHO, 2015) Diz que a necessidade da migração de uma arquitetura monolítica para micro serviço é benéfico devido ao isolamento dos problemas encontrado em sistemas monolíticos e fazendo que cada serviço migrado para arquitetura de micro serviço utilize recurso da computação em nuvem conforme a necessidade, seja ela processamento ou flexibilidade de dados para que o micro serviço faça da melhor forma possível sua função escalonando apenas o que for necessário e não aplicação como um todo.

(NEWMAN, 2015) A empresa Gilt utilizava uma aplicação monolítica até 2009 e existia uma sobre carga de trafego em seu sistema. A Gilt conseguiu lidar com os picos de trafego quebrando seu sistema em mais de 450 micro serviços, onde cada um é executada em várias maquinas separadas utilizando recursos de nuvem da Amazon, onde o escalonamento é feito por demanda fazendo que o custo seja controlado de forma mais eficaz.

Segundo (RICHARDSON, 2014) cada serviço pode ser escalado de forma independente de outros serviços através da duplicação e particionamento, possibilitando que cada serviço seja implementado no hardware mais adequado conforme a necessidade de recurso.

2.2.7) BENEFÍCIOS DA ARQUITETURA DE MICRO SERVIÇO

O uso da arquitetura de micro serviço aumenta as possibilidades de otimização para grandes sistemas.

Segundo (NEWMAN, 2015) os benefícios são muitos, todos que a computação distribuída pode oferecer, porem em um grau maior devido o quão longe eles podem levar os conceitos da computação distribuída e a arquitetura orientada a serviço. Como a heterogeneidade de tecnologia, onde um sistema composto de

vários serviços, colaborando, podendo decidir usar diferentes tecnologias dentro de cada um.

(RICHARDSON, 2014) Vê na utilização da arquitetura de micro serviço vantagens como a redução de contexto que acaba limitando as linhas de código, tornando o micro serviço relativamente pequeno não tornando a IDE[8] lenta. O código acaba se tornando facilmente compreendido pelo desenvolvedor tornando–os mais produtivos. Por ser relativamente pequenos sua inicialização é muito mais rápida do que uma grande aplicação monolítica.

(ALMEIDA, 2015) Arquitetura de micro serviço minimiza os pontos de falha da aplicação (Single point of failure) vistos em arquitetura monolítica, devido a autonomia de cada micro serviço que são independentes e autônomos.

(MACEDO, 2015) Diz que um dos benefícios da arquitetura de micro serviço é a adição de desenvolvedores no projeto. Cada micro serviço possui um contexto limitado facilitando seu entendimento.

Segundo (RODRIGUES, GROFFE e SIMAS, 2015) o ganho da divisão de times na arquitetura de micro serviço, cada equipe é responsável por seu serviço e selecionar desenvolvedores que são familiarizados com a regra de negócio envolvida.

(SAKURAI e CASCARROLHO, 2015) A vantagem da autonomia em micro serviço facilita a manutenção da solução, pois cada problema é tratado de forma isolada.

(MARTINS, 2015) Os impactos das mudanças são minimizados devido a autonomia dos micros serviços. Uma outra vantagem é utilização da ferramenta correta para cada problema como escolha de linguagem, banco de dados e sistema operacional. Na utilização de micro serviço o isolamento de cada serviço facilita na estratégia de resiliência.

[8] Ambiente de desenvolvimento integrado

2.2.8) DESVANTAGENS DA ARQUITETURA DE MICRO SERVIÇO

Assim como qualquer outra tecnologia, arquitetura de micro serviço possui desvantagens significativas que devem ser consideradas para quem for utilizar no desenvolvimento da solução.

Segundo (RICHARDSON, 2014) uma das desvantagens na arquitetura de micro serviço é a complexidade da computação distribuída para n serviços e realização de testes automatizados para vários serviços. Sua complexidade operacional devido ao grande número de instancias para gerenciar. Para se obter sucesso em arquitetura de micro serviço o nível de automação necessita ser significativamente alto.

(NOBRE, 2015) Com a independência das equipes gera duplicidade de código e podendo gerar duplicidade de dados. E garantir a persistência desses dados é complexo.

Segundo (MARTINS, 2015) a cultura devops[9] precisa ser muito madura para suportar a demanda continua de micro serviço, devido uma das grandes vantagens de micro serviço é a capacidade de mudança continua.

(BIRD, 2015) Micro serviço gera uma perda de latência devido à sobre carga de chamadas remotas. Dando como exemplo a empresa LinkedIn, onde uma única solicitação de usuário pode acionar uma cadeia de até 70 serviços.

[9] Capacidade de pôr o software e serviços de forma rápida em produção para aumento de produtividade e redução de custos.

3) DESENVOLVIMENTO DO PROJETO

O sistema proposto foi uma agenda de tarefas, onde seu desenvolvimento seguiu os conceitos da arquitetura de micro serviços.

O funcionamento da agenda inicia-se com o login de usuário, após a inserção dos dados ele passará por um teste de validação. Se a validação estiver correta o acesso a tarefas será permitido, caso contrário será retornado um erro de login onde uma das sugestões será a confirmação dos dados ou efetuar um cadastro valido.

Efetuando o acesso a tarefa, o usuário poderá criar, atualizar, pesquisar e deletar tarefas existentes.

Para mostrar a comparação de sua estrutura foi projetado um caso de uso monolítico para ilustrar o funcionamento do sistema. Como mostra a Figura 10:

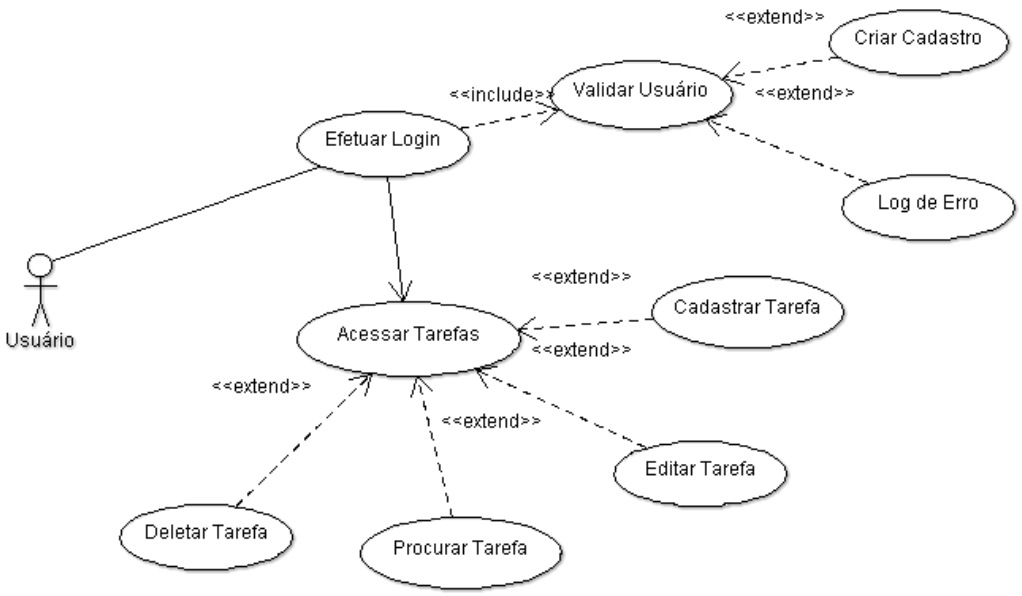


Figura 10 - Estrutura monolítica da solução

A solução foi pensado de forma monolítica, pois segundo (MARTINS, 2015) para começar a construção de uma solução no conceito de arquitetura de micro serviço é necessário conhecer muito bem as fronteiras de sua solução para que

facilite a separação dos serviços que façam sentido. E de acordo com os especialistas da área uma boa forma de começar é pensando monolítico.

A transformação da arquitetura da aplicação monolítica para arquitetura de micro serviço ficou projetada da seguinte forma:

- Micro serviço de Usuário: Este serviço é responsável de receber uma solicitação de login, validar os usuários, criar cadastro de usuário.
- Micro serviço de Tarefa: Serviço responsável pelo controle de manipulação de dados de tarefas. Sua responsabilidade é armazenar os dados no seu banco de dados e permitir toda a dinâmica de um CRUD (Create, Read, Update e Delete na linguagem Inglesa).
- API Cliente Web: Essa API é responsável de orquestrar os outros micro serviços, pois através dele que será feita todas as solicitações para os micro serviços.

Então, olhando o caso de uso novamente, pode-se ter a visão da arquitetura de micro serviço, onde um caso de uso acabou se fragmentando em pedaços menores. Como pode ser observado na Figura 11.

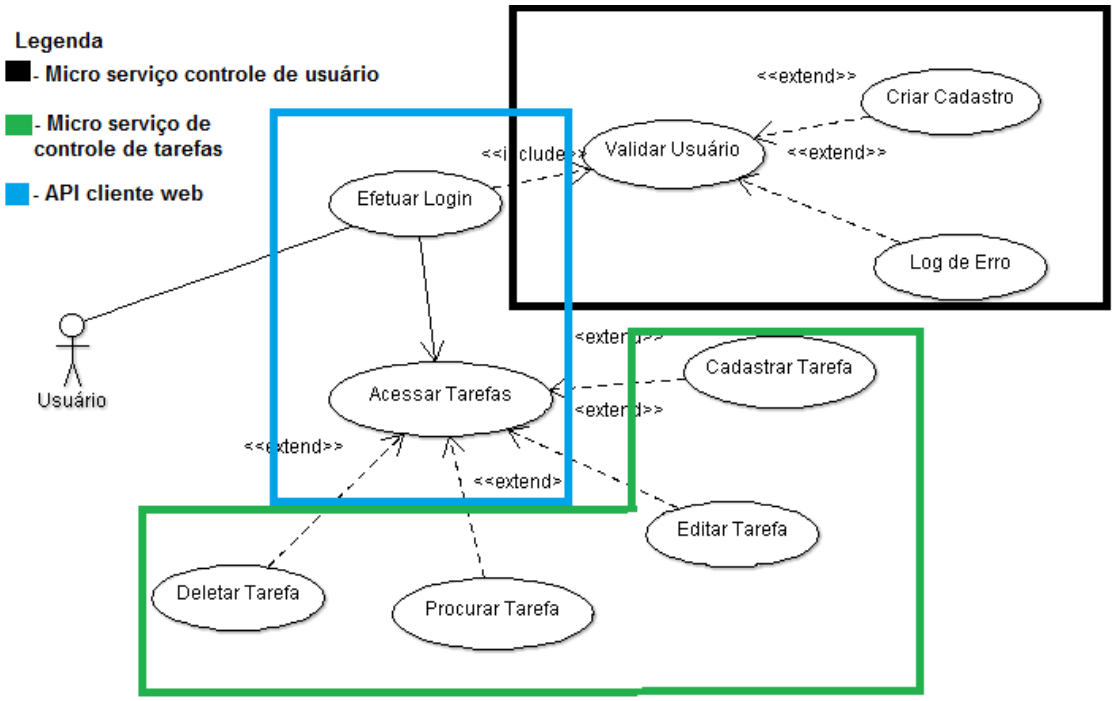


Figura 11 - Visão de arquitetura de micro serviço

Após a separação de contexto acabou gerando os seguintes casos de uso:

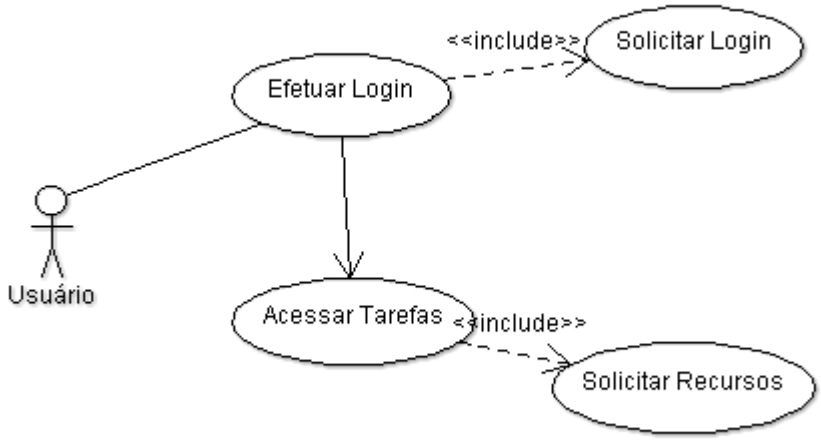


Figura 12 - API Cliente Web

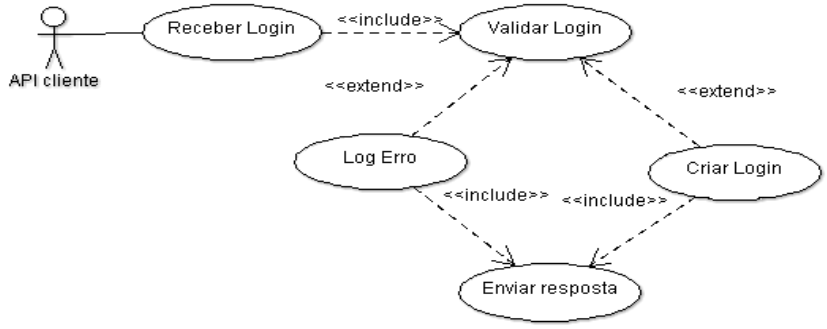


Figura 13 - Micro serviço de Usuário

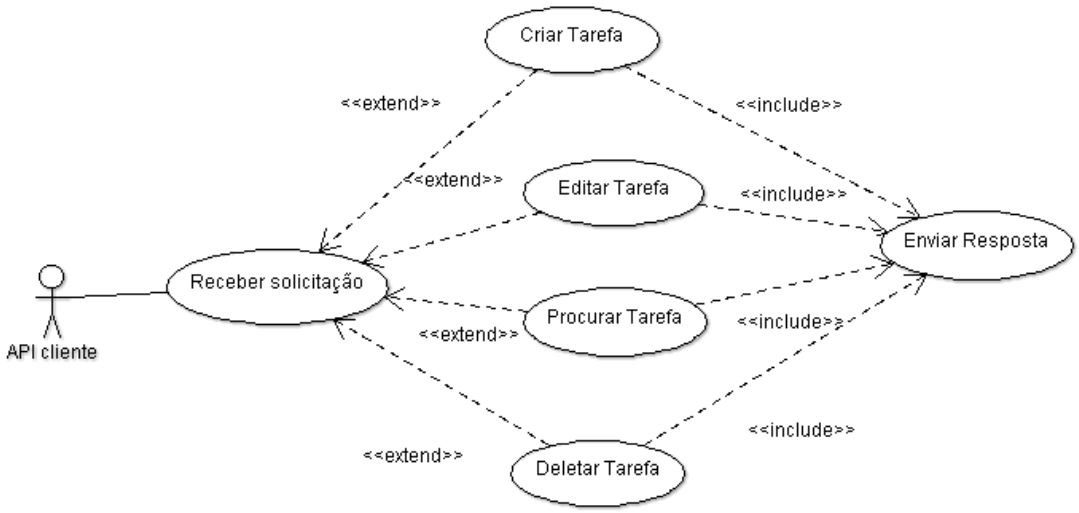


Figura 14 - Micro Serviço de Tarefa

3.1) ARQUITETURA DA SOLUÇÃO

Para o desenvolvimento da aplicação foi utilizado a seguinte arquitetura ilustrada na Figura 15:

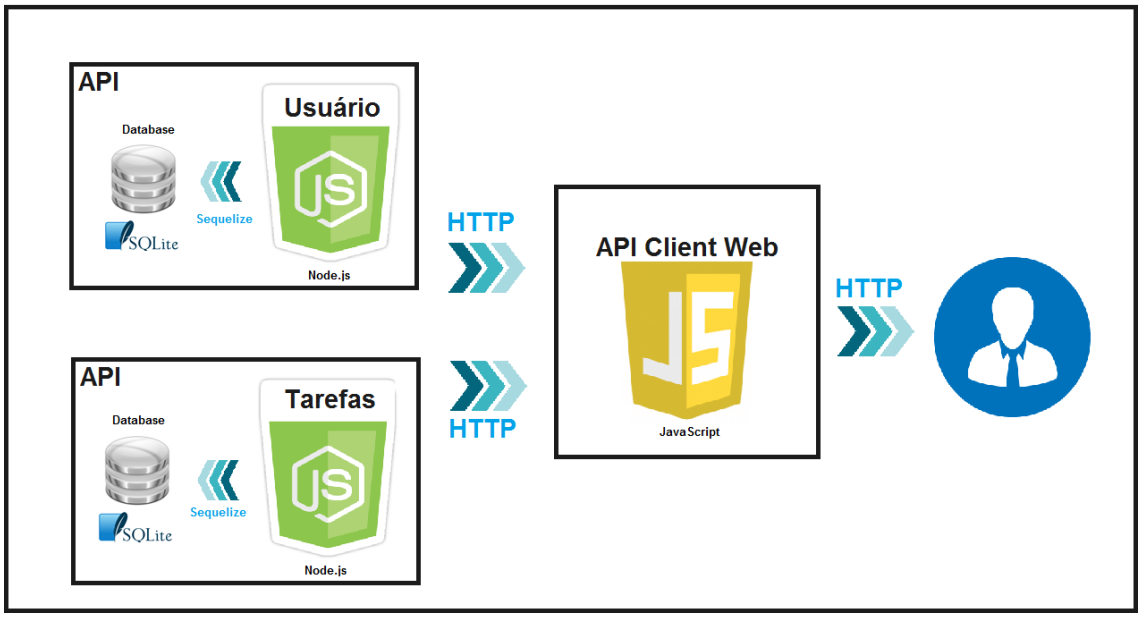


Figura 15 - Arquitetura do projeto

A arquitetura mostra as tecnologias utilizadas para a construção dos micro serviços, assim como, a utilização de seus recursos HTTP para uma aplicação final para o usuário.

A plataforma de desenvolvimento utilizada será o Node.js

Node.js é uma plataforma para construção de redes rápidas e escaláveis utilizando o motor JavaScript V8 do Google Chrome do lado servidor ao invés do tradicional JavaScript do lado cliente. Orientada a eventos, é muito eficiente para aplicações em tempo real e dispositivos distribuídos. (MOREIRA, 2013).

3.2) BASE DE DADOS

Para a construção de cada micro serviço foi utilizado o banco de dados relacional SQLite3

Segundo (BRAGA, 2008) o SQLite é um SGBD (Sistema de gerenciamento de banco de dados) que possui algumas peculiaridades:

- auto-suficiente: precisando de poucas bibliotecas externas;

- independente de uma estrutura cliente-server: ou seja, não existe um processo servidor intermediário, ele lê e escreve os dados diretamente no arquivo armazenado no disco;

- configuração de base de dados zero: não precisa ser instalado para uso, não existindo a necessidade de setup ou processo servidor para inicializar ou configurar.

- Estrutura transacional: todas as alterações e consultas são atômica (“tudo ou nada” se caso uma parte falhar a transação toda não será efetuada e o arquivo se mantem inalterado), consistente (garanti a qualidade da transação usando as regras definidas como restrições, cascatas e gatilhos), isolada (evento que executa a transação de forma concorrente em serie ou seja uma após o outro) e

durável (garante a integridade dos dados após a conclusão da transação) (ACID).

Para efetuar a persistência no banco de dados SQLite3 utilizando Node.js foi necessário trabalhar com uma framework Node.js chamada Sequelize. Suas funções adotam o padrão Promises que é uma promessa assumindo três estados; Pendente (estado inicial, ainda não concluído ou rejeitado), Realizado (sucesso na operação) ou Rejeitado (falha na operação). (SEQUELIZE)

3.3) CONSTRUINDO MICRO SERVIÇO DE TAREFA

Para o início da construção da aplicação em Node.js foi gerado o arquivo package.json, ele é o arquivo responsável de armazenar todas dependências dos módulos necessários para a construção do micro serviço de tarefas

Os módulos necessários para a construção está ilustrado na Figura 16:

```
{
  "name": "tarefa",
  "version": "1.0.0",
  "description": "API de tarefas",
  "main": "index.js",
  "scripts": {
    "start": "npm run apidoc && npm run clusters",
    "clusters": "babel-node clusters.js",
    "apidoc": "apidoc -i routes/ -o public/apidoc",
    "test": "NODE_ENV=test mocha test/**/*.js"
  },
  "author": "Andre Vieira",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0",
    "body-parser": "^1.15.0",
    "compression": "^1.6.1",
    "consign": "^0.1.2",
    "cors": "^2.7.1",
    "express": "^4.13.4",
    "helmet": "^1.1.0",
    "morgan": "^1.6.1",
    "sequelize": "^3.19.2",
    "sqlite3": "^3.1.1",
    "winston": "^2.1.1"
```

```
  },
  "devDependencies": {
    "apidoc": "^0.15.1",
    "babel-register": "^6.5.2",
    "chai": "^3.5.0",
    "mocha": "^2.4.5",
    "supertest": "^1.2.0"
  }
}
```

Figura 16 - Package.json de Tarefa

A definição de alguns módulos são:

- babel-cli e babel-preset-es2015: ele permite que node utilize códigos ES6/7;

- compression: compacta as respostas JSON e todos os arquivos estáticos das requisições para o formato GZIP;

- consign: permite carregar as dependências de forma mais simples;

- cors: um dos módulos mais importante da aplicação, ele é responsável de permitir ou barrar as requisições que serão utilizadas por outros domínios;

- express: modulo responsável de manipular os views, routes, controllers e models;

A estruturação de diretórios segue o padrão MVC (model, view, controller), assim como mostra a Figura 17:

Figura 17 - Estrutura de Diretório

O diretório /libs é responsável de armazenar os arquivos de configuração do servidor de micro serviço de tarefas.

O arquivo libs/boot.js como pode ser observado na Figura 18, possui uma credencial, ela é necessária quando se usa o protocolo HTTPS (Hyper Text Transfer Protocol Secure). Está credencial tem como parâmetros um certificado e uma chave para o acesso, esses dois arquivos foram gerados no site http://www.selfsignedcertificate.com que cria certificados fictícios não validos para produção, após gerar o certificado e a chave, foi adicionado na raiz do projeto para ser utilizado pela solução através do modulo *FILE SYSTEM* com a função **.readFileSync()**. Após a validação do certificado o arquivo boot.js tem a função de iniciar o servidor e fazer a sincronia do *SEQUELIZE* com o banco de dados, utilizando a função **app.db.sequelize.sync()** e cria o servidor utilizando as credenciais, que ficará ouvindo na porta 3000 através da função **.listen()**. O endereço da porta está configurada no arquivo middlewares.js

```
import https from "https";
import fs from "fs";
module.exports = app => {
	if(process.env.NODE_ENV !== "test"){
		const credentials = {
			key: fs.readFileSync("tarefas.key", "utf8"),
			cert: fs.readFileSync("tarefas.cert", "utf8")
		}
		app.db.sequelize.sync().done(() => {
			https.createServer(credentials, app)
				.listen(app.get("port"), () => {
					console.log(`Tarefas API - porta ${app.get("port")}`);
				});
		});
	}
};
```

Figura 18 - Arquivo boots.js

O arquivo libs/middlewares.js representado na Figura 19, é responsável de orquestrar os recursos da aplicação como conversor dos arquivos JSON através do *modulo BODY-PARSER*, o *modulo HELMET* que possui vários tipos de segurança contra invasões que irá melhorar a segurança da aplicação, definição

da porta de comunicação do servidor e o domínio que usará os recursos, assim como, quais recursos serão compartilhados.

Toda essa função será externalizada na aplicação pelo **.exports** através da variável app. Como pode ser visto na Figura 19:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import compression from "compression";
import helmet from "helmet";
import logger from "./logger.js";
module.exports = app => {
	app.set("port",3000);
	app.set("json spaces",4);
	app.use(morgan("common",{
		stream: {
			write: (message) => {
				logger.info(message);
			}
		}
	}));
	app.use(helmet());
	app.use(cors({
		origin: ["http://localhost:3002"],
		methods: ["GET" ,"POST" ,"PUT" ,"DELETE"],
		allowedHeaders: ["Content-Type", "Authorization"]
	}));
	app.use(compression());
	app.use(bodyParser.json());
	app.use((req, res, next) => {
		delete req.body.id;
		next();
	});
	app.use(express.static("public"));
};
```

Figura 19 - Arquivo middlewares.js

Para a estruturação do carregamento dos módulos será criado o arquivo na raiz do projeto chamado index.js com a seguinte configuração mostrado na Figura 20:

```
import express from "express";
import consign from "consign";

const app = express();

consign({verbose:false})
        .include("libs/config.js")
        .then("db.js")
        .then("libs/middlewares.js")
        .then("routes")
        .then("libs/boot.js")
        .into(app);

module.exports = app;
```

Figura 20 - Arquivo index.js

Ele é responsável de importar o recurso do modulo *EXPRESS* e *CONSIGN*.

O modulo *CONSIGN* irá estruturar o carregamento do código.

O parâmetro passado no *CONSIGN* “verbose” está com status “false” para que nos teste unitários com mocha[10] não gere alguns arquivos de logs deixando mais limpo.

O arquivo db.js possui os seguintes parâmetros no modulo *SEQUELIZE*:

- **database**: define o nome da base de dados;

- **username**: informa o nome de usuário de acesso;

- **password**: - informa a senha do usuário;

- **params.dialect**: informa qual banco de dados será usado;

- **params.storage**: informa o diretório que será gravado o base de dados;

- **params.define.underscored**: padroniza o nome dos campos da tabela em minúsculo;

[10] Mocha é uma framework de teste JavaScript para uso com Node.js para teste simples unitários

O código no arquivo db.js mostrado na Figura 21:

```
import fs from "fs";
import path from "path";
import Sequelize from "sequelize";

let db = null;

module.exports = app => {
	if(!db) {
		const config = app.libs.config;
		const sequelize = new Sequelize(
			config.database,
			config.username,
			config.password,
			config.params
		);
		db = {
			sequelize,
			Sequelize,
			models: {}
		};
		const dir = path.join(__dirname,"models");
		fs.readdirSync(dir).forEach(file => {
			const modelDir = path.join(dir, file);
			const model = sequelize.import(modelDir);
			db.models[model.name] = model;
		});
	}
	return db;
};
```

Figura 21 - Arquivo db.js

A configuração desses parâmetros foi declarado no arquivo config.development.js e sua representação pode ser vista na Figura 22.

```
import logger from "./logger.js";
module.exports = {
	database: "tarefas",
	username: "",
	password: "",
	params: {
		dialect: "sqlite",
		storage: "tarefas.sqlite",
		logging: (sql) => {
			logger.info(`[${new Date()} ${sql}]`);
		},
```

```
                define: {
                        underscored: true
                }
        }
};
```

Figura 22 - Arquivo config.development.js

O diretório model irá armazenar todos os modelos da aplicação, no entanto, o único modelo necessário é o modelo de tarefa, onde foi definido os campos para ser armazenado no banco de dados configurado no arquivo db.js

A codificação no model/tarefas.js pode ser analisada na Figura 23.

```
module.exports = (sequelize, DataType) => {
        const Tarefas = sequelize.define("Tarefas", {
                id: {
                        type: DataType.INTEGER,
                        primaryKey: true,
                        autoIncrement: true
                },
                title: {
                        type: DataType.STRING,
                        allowNull: false,
                        validate: {
                                notEmpty: true
                        }
                },
                done: {
                        type: DataType.BOOLEAN,
                        allowNull: false,
                        defaultValue: false
                }
        });
        return Tarefas;
};
```

Figura 23 - Arquivo model/tarefas.js

No arquivo tarefa.js tem como parâmetro um *sequelize* que irá fazer a sincronização com o banco de dados e *datatype* que irá representar o tipo de dado.

A função **sequelize.define()** cria ou alterar a tabela do banco de dados que no caso, será a tabela “tarefas” possuindo o seguintes campos: id (Integer), title (String), done (boolean).

O diretório routes tem como funcionalidade de gerenciar todas as rotas da aplicação. Este diretório possui dois arquivos chamados index.js e tarefas.js.

O index.js é responsável de carregar a rota principal “/” da aplicação. No arquivo tarefas.js tem como funcionalidade direcionar a aplicação para as rotas que serão compartilhados via HTTP e que são externalizados para outras aplicações web através do modulo *CORS* utilizado no arquivo de middlewares.js. Uma das rotas existentes é “/tarefas” que utiliza o método *GET* para localizar todas as tarefas, método *POST* que é encarregado de criar as tarefas. O endereço “/tarefas/:id” tem como função manipular as tarefas através do id passado no parâmetro da url e utiliza os seguintes métodos, *GET* que retorna a resposta da tarefa que possuir o id informado, *PUT* atualiza os campos da tarefa identificada e *DELETE* que remove a tarefa cadastrada com o id informado na url.

O código responsável por toda essa manipulação ficou da seguinte forma como mostra a Figura 24:

```
module.exports = app => {
	const Tarefas = app.db.models.Tarefas;
	app.route("/tarefas"
		.get((req,res) => {
			Tarefas.findAll({})
				.then(result => res.json(result))
				.catch(error => {
					res.status(412).json({msg: error.message});
				});
		});
	app.route("/tarefas/:id"
		.get((req,res) => {
			Tarefas.findOne({where: req.params})
				.then(result => {
					if(result) {
						res.json(result);
					} else {
						res.sendStatus(404);
					}
				})
				.catch(error => {
					res.status(412).json({msg: error.message});
				});
		})
```

```
                .put((req,res) => {
                        Tarefas.update(req.body, {where: req.params})
                                .then(result => res.sendStatus(204))
                                .catch(error => {
                                        res.status(412).json({msg: error.message});
                                });
                })
                .delete((req,res) => {
                        Tarefas.destroy({where: req.params})
                                .then(result => res.sendStatus(204))
                                .catch(error  => {
                                        res.status(412).json({msg: error.message});
                                });
                });
};
```

Figura 24 - Arquivo de rota de tarefas

3.4) CONSTRUINDO MICRO SERVIÇO DE USUÁRIO

Este serviço é responsável de manipular os dados de usuário e faz a validação do usuário cadastrado.

Para a construção desse micro serviço foi necessário criar o arquivo package.json, assim como foi criado no micro serviço de tarefas.

O arquivo package.json pode ser observado na Figura 25.

```
{
  "name": "user-api",
  "version": "1.0.0",
  "description": "API de gestao de usuario",
  "main": "index.js",
  "scripts": {
    "start": "npm run apidoc && npm run clusters",
    "clusters": "babel-node clusters.js",
    "apidoc": "apidoc -i routes/ -o public/apidoc",
    "test": "NODE_ENV=test mocha test/**/*.js"
  },
```

```
  "apidoc": {
    "name": "Documentacao - Node Usuario API",
    "template": {
      "forceLanguage": "pt_br"
    }
  },
  "author": "Andre Vieira",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0",
    "bcrypt": "^0.8.5",
    "body-parser": "^1.15.0",
    "compression": "^1.6.1",
    "consign": "^0.1.2",
    "cors": "^2.7.1",
    "express": "^4.13.4",
    "helmet": "^1.1.0",
    "jwt-simple": "^0.4.1",
    "morgan": "^1.6.1",
    "passport": "^0.3.2",
    "passport-jwt": "^2.0.0",
    "sequelize": "^3.19.2",
    "sqlite3": "^3.1.1",
    "winston": "^2.1.1"
  },
  "devDependencies": {
    "apidoc": "^0.15.1",
    "babel-register": "^6.5.2",
    "chai": "^3.5.0",
    "mocha": "^2.4.5",
    "supertest": "^1.2.0"
  }
}
```

Figura 25 - Arquivo package.json de Usuário

Este serviço utiliza módulos semelhantes ao serviço de tarefas, porem, possui algumas peculiaridades distintas como os módulos:

- bcrypt: responsável de fazer a criptografia da senha do usuário;
- passport: recurso para estratégia de autenticação;
- passport-jwt: escolha do recurso a ser utilizado pelo passport para autenticação;
- jwt-simple: biblioteca para encodificação/decodificação de tokens do JWT;

O diretório foi estruturado com o padrão MVC muito semelhante ao serviço de tarefas, no entanto, existe diferenças significativas entre eles como arquivo de autenticação, geração de tokens para acesso e criptografia de dados.

Para a construção do modelo de usuário foi necessário fazer o import do modulo *BCRYPT* que é responsável de fazer a criptografia da senha do usuário, assim como utilizar os parâmetros de *hooks* que é uma função que executa antes ou depois de uma operação no banco de dados. A função **bcrypt.genSaltSync()** gera a criptografia e a função **bcrypt.hashSync()** que utiliza o password informado pelo usuário aplicando a criptografia gerada pela **genSaltSync().**

O parâmetro de *classMethods* compara a senha informada com a senha criptografada. A codificação ficou da seguinte forma, como mostra a Figura 26.

```
import bcrypt from "bcrypt";
module.exports = (sequelize, DataType) => {
	const Users = sequelize.define("Users", {
		id: {
			type: DataType.INTEGER,
			primaryKey: true,
			autoIncrement: true
		},
		name: {
			type: DataType.STRING,
			allowNull: false,
			validate: {
				notEmpty: true
			}
		},
		password: {
			type: DataType.STRING,
			allowNull: false,
			validate: {
				notEmpty: true
			}
		},
		email: {
			type: DataType.STRING,
			unique: true,
			allowNull: false,
			validate: {
				notEmpty: true
			}
		}
	}, {
		hooks: {
			beforeCreate: user => {
				const salt = bcrypt.genSaltSync();
```

```
                                        user.password = bcrypt.hashSync(user.password, salt);
                                }
                        },
                        classMethods: {
                                isPassword: (encodedPassword, password) => {
                                        return bcrypt.compareSync(password, encodedPassword);
                                }
                        }
                });
                return Users;
};
```

Figura 26 - Arquivo Model User.js

No diretório routes possui os seguintes arquivos, index.js responsável pela rota principal “/”, token.js onde sua função é validar os dados informados pelo usuário e gerar um token de acesso. Enquanto o arquivo users.js controla as rotas para *GET, POST* e *DELETE.*

Para acesso as rotas da aplicação é necessário fazer uma autenticação e o modulo *PASSPORT* é o responsável por essa ação. Através dele pode-se adotar diversas estratégias de autenticação, utilizando serviços externos como Facebook, Twitter e entre outros. O meio de autenticação utilizada foi Json Web Tokens (JWT).

Para a codificação do arquivo routes/token.js foi necessário utilizar o import do modulo *JWT-SIMPLE* para fazer a codificação e descodificação dos dados do usuário e após a validação, é gerado um token de acesso.

A codificação pode ser observada na Figura 27:

```
import jwt from "jwt-simple";
module.exports = app => {
        const cfg = app.libs.config;
        const Users = app.db.models.Users;
        app.post("/token", (req,res) => {
                if(req.body.email && req.body.password) {
                        const email = req.body.email;
                        const password = req.body.password;
                        Users.findOne({where: {email: email}})
```

```
                                .then(user => {
                                        if(Users.isPassword(user.password,password)) {
                                                const payload = {id: user.id};
                                                res.json({
                                                        token: jwt.encode(payload, cfg.jwtSecret)
                                                });
                                        } else {
                                                res.sendStatus(401);
                                        }
                                })
                                .catch(error => res.sendStatus(401));
                } else {
                        res.sendStatus(401);
                }
        });
};
```

Figura 27 - Arquivo Token.js

A comparação da senha do usuário com a senha que está no bando é feita através da função .**isPassword()** criada no modelo de usuário, onde os parâmetros são a senha codificada e a senha informada

A variável payload irá armazenar o id do usuário para geração do token de acesso para esse id. O token é criado através do método *jwt.encode* que usa uma chave secreta para cria-lo. Esta chave secreta foi definida no arquivo libs/config.development.js, através desta chave que é formado todos os tokens de acesso como mostra a Figura 28:

```
import logger from "./logger.js";
module.exports = {
        database: "user",
        username: "",
        password: "",
        params: {
                dialect: "sqlite",
                storage: "user.sqlite",
                logging: (sql) => {
                        logger.info(`[${new Date()}] ${sql}`);
                },
                define: {
                        underscored: true
                }
        },
        jwtSecret: "Us3r2",
        jwtSession: {session: false}
};
```

Figura 28 - Arquivo config.development

O parâmetro jwtSecret é responsável pela palavra secreta para encode/decode dos tokens. O campo jwtSession está informando para o Passport que a autenticação não terá sessão de usuário.

Para usar a autenticação nas rotas de usuário foi criado o arquivo auth.js na raiz do projeto, sua função é aplicar as regras de autenticação usando o passport. Na codificação do arquivo auth.js temos alguns parâmetros importantes para efetuar a validação como a variável params. Esta variável possui os seguintes atributos *secretOrKey* , ela irá carregar a palavra secreta determinada no arquivo config.development, O campo *jwtFromRequest* irá aceitar a requisição e retornará o token.

A variável strategy cria uma estancia do objeto *Strategy* passando como parâmetro a variável params que contém as configurações de token junto com a palavra secreta e como segundo parâmetro um payload, nele contém um id que foi codificado em forma de token através da função **jwt.encode()** existente no arquivo routes/token.js.

O retorno do arquivo auth.js é a inicialização da função do passport através do *initialize* e a autenticação utilizando a estratégia jwt com a palavra secreta. Sua codificação pode ser vista na Figura 29.

```
import passport from "passport";
import {Strategy, ExtractJwt} from "passport-jwt";

module.exports = app => {
	const Users = app.db.models.Users;
	const cfg = app.libs.config;
	const params = {
		secretOrKey: cfg.jwtSecret,
		jwtFromRequest: ExtractJwt.fromAuthHeader()
	};
	const strategy = new Strategy(params, (payload, done) => {
		Users.findById(payload.id)
			.then(user => {
				if(user) {
					return done(null, {
						id: user.id,
						email: user.email
					});
				}
				return done(null, false);
			})
			.catch(error => done(error, null));
```

```
        });
        passport.use(strategy);
        return {
                initialize: () => {
                        return passport.initialize();
                },
                authenticate: () => {
                        return passport.authenticate("jwt", cfg.jwtSession);
                }
        };
};
```

Figura 29 - Arquivo auth.js

Após a configuração das regras de autenticação foi aplicado nas seguintes rotas de usuário no arquivo routes/users.js representado na Figura 30. O método *.ALL* que é responsável de inicializar o carregamento das rotas com os métodos *GET* e *DELETE* essa autenticação é feita através da função **.authenticate()** do arquivo auth.js. O método de *POST* não necessita de validação pois ele é responsável pela criação de usuário para uma futura autenticação.

```
module.exports = app => {
        const Users = app.db.models.Users;

        app.route("/user")
                .all(app.auth.authenticate())

                .get((req, res) => {
                        Users.findById(req.user.id, {
                                attributes: ["id", "name", "email"]
                        })
                        .then(result => res.json(result))
                        .catch(error => {
                                res.status(412).json({msg: error.message});
                        });
                })

           .delete((req, res) => {
                        Users.destroy({where: {id: req.user.id} })
                                .then(result => res.sendStatus(204))
                                .catch(error => {
                                        res.status(412).json({msg: error.message});
                                });
                });
        app.post("/users", (req, res) => {
                Users.create(req.body)
                        .then(result => res.json(result))
```

```
                .catch(error => {
                        res.status(412).json({msg: error.message});
                });
        });
};
```

Figura 30 - Arquivo routes/users.js

3.5) CONSTRUINDO APLICAÇÃO CLIENTE WEB

Após a construção dos serviços RestFull de tarefas e usuários, foi criado uma aplicação que consumi esses serviços através do protocolo HTTP.

Para a construção dessa aplicação foi utilizado o JavaScript EcmaScript 6 criando uma página Single Page Application (SPA) ou seja, aplicações de uma única página. O arquivo package.json da aplicação está ilustrado na Figura 31.

```
{
  "name": "ntask-web",
  "version": "1.0.0",
  "description": "Versão web do gerenciador de tarefas",
  "scripts": {
    "start": "npm run build && npm run server",
    "server": "http-server public -p 3002",
    "build": "npm run browserify && npm run uglify",
    "browserify": "browserify src -t babelify -o public/js/app.js",
    "uglify": "uglify -s public/js/app.js -o public/js/app.min.js"
  },
  "author": "Andre Vieira da Silva",
  "dependencies": {
    "babelify": "^6.3.0",
    "browser-request": "^0.3.3",
    "browserify": "^11.2.0",
    "http-server": "^0.8.4",
    "tiny-emitter": "^1.0.0",
    "uglify": "^0.1.5"
  },
  "devDependencies": {
    "grunt": "^1.0.1",
    "grunt-cli": "^1.2.0",
    "grunt-contrib-clean": "^1.0.0",
```

```
    "grunt-contrib-concat": "^1.0.1",
    "grunt-contrib-copy": "^1.0.0",
    "grunt-contrib-uglify": "^1.0.1"
  }
}
```

Figura 31 - Arquivo package.json Cliente Web

A definições dos módulos utilizados são:

- http-server: cliente de servidor HTTP para arquivos estáticos;
- browserify: compilador JavaScript que permite utilizar módulos do NPM;
- babelify: plugin para front-end para o browserify para códigos EcmaScript 6;
- babel-preset-es2015: para reconhecer códigos ES6;
- tiny-emitter: modulo responsável para trabalhar de forma orientada a eventos;
- browser-request: uma versão do modulo request focado para browsers;

Para a construção do layout será usado arquivo CSS da framework Ionic onde o seu download é encontrado nos seguintes links:

- CSS do Ionic: http://code.ionicframework.com/1.0.0/css/ionic.min.css
- CSS do Ionic icons:

  http://code.ionicframework.com/ionicons/2.0.0/css/ionicons.min.css

- Fontes do Ionic Icons:

  http://code.ionicframework.com/1.0.0/fonts/ionicons.eot

  http://code.ionicframework.com/1.0.0/fonts/ionicons.svg

  http://code.ionicframework.com/1.0.0/fonts/ionicons.ttf

  http://code.ionicframework.com/1.0.0/fonts/ionicons.woff

A estrutura de diretórios está ilustrado na figura 32.

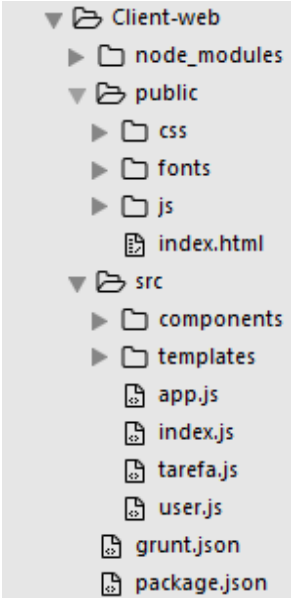


Figura 32 - Estrutura de diretórios Cliente Web

O HTML principal da aplicação está localizada no diretório public/index.html.

Para o consumo dos micro serviços tarefas e usuário, foi criado os modelos dentro do diretório src/tarefa.js representado na Figura 33 e src/user.js que pode ser observado na Figura 34.

```
import TinyEmitter from "tiny-emitter";
import Request from "browser-request";

class Tarefa extends TinyEmitter {
	constructor() {
		super();
		this.request = Request;
		this.URL = "https://localhost:3000";
	}
}
module.exports = Tarefa;
```

Figura 33 - Arquivo src/tarefa.js

```
import TinyEmitter from "tiny-emitter";
import Request from "browser-request";

class User extends TinyEmitter {
        constructor() {
                super();
                this.request = Request;
                this.URL = "https://localhost:3001";
        }
}
module.exports = User;
```

Figura 34 - Arquivo src/user.js

Os arquivos src/tarefa.js e src/user.js utilizam os módulos *TINYEMITTER* para permitir escutar os eventos e *BROWSER-REQUEST* que irá carregar a URL da API.

Após a criação da classe User que irá carregar a url do serviço de usuário no endereço https://localhost:3001.

A classe foi utilizada dentro do diretório src/componentes/user.js. Essa nova classe de usuário utiliza a herança da classe pai do diretório src/user.js junto com os métodos. O código é apresentado na Figura 35.

```
import Users from "../user.js";
import Template from "../templates/user.js";
import Loading from "../templates/loading.js";

class User extends Users {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.renderUserData()
  }
  addEventListener() {
    this.userCancelClick();
  }
  renderUserData(){
    const opts = {
      method: "GET",
      url: `${this.URL}/user`,
      json: true,
      headers: {
```

```
        authorization: localStorage.getItem("token")
      }
    };
    this.body.innerHTML = Loading.render();
    this.request(opts, (err, resp, data) => {
      if (err || resp.status === 412) {
        this.emit("error", err);
      } else {
        this.body.innerHTML = Template.render(data);
        this.addEventListener();
      }
    });
  }
  userCancelClick() {
    const button = this.body.querySelector("[data-remove-account]");
    button.addEventListener("click", (e) => {
      e.preventDefault();
      if (confirm("Tem certeza que deseja excluir sua conta?")) {
        const opts = {
          method: "DELETE",
          url: `${this.URL}/user`,
          headers: {
            authorization: localStorage.getItem("token")
          }
        };
        this.request(opts, (err, resp, data) => {
          if (err || resp.status === 412) {
            this.emit("remove-error", err);
          } else {
            this.emit("remove-account");
          }
        });
      }
    });
  }
}
```

Figura 35 - Arquivo src/components/user.js

A função **renderUserData()** utiliza o método *GET* herdado da classe user.js (src/user.js) onde é passado a url de acesso e passando o token de autorização. No caso de não encontrar nenhuma irregularidade ele ficara ouvindo os eventos através do **addEventListener().**

A função **userCancelClick()** é responsável de deletar o usuário logado sua composição é semelhante a função **renderUserData().**

O arquivo src/tarefa.js carrega os métodos do serviço de tarefa através da url https://localhost:3000 e é utilizado como classe pai dentro da classe tarefas.js localizada dentro do diretorio src/componentes/tarefas.js. Esta classe herdar os métodos *GET, POST, PUT, DELETE*.

A função **renderTarefaList()** carrega o método *GET* do serviço de tarefas e caso se nenhum erro for encontrado ele ficara ouvindo os eventos através da função **addEventListener().** O evento é responsável de ficar checando se terá alguma alteração no status da tarefa cadastrada como concluída ou não, no seu campo de checkbox através do evento **tarefaDoneCheckBox()** que usará o método de *PUT* para fazer a operação. Outro evento presente em **addEventListener()** é o evento de remover uma tarefa, este evento está identificado como **tarefaRemoveClick()** ele usará o método *DELETE* do serviço de tarefas.

As regras de aplicação pode ser observado na Figura 36:

```
import Tarefa from "../tarefa.js";
import Template from "../templates/tarefas.js";
import Loading from "../templates/loading.js";

class Tarefas extends Tarefa {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.renderTarefaList()
  }
  addEventListener() {
    this.tarefaDoneCheckbox();
    this.tarefaRemoveClick();
  }
  renderTarefaList(){
    const opts = {
      method: "GET",
      url: `${this.URL}/tarefas`,
      json: true,
      headers: {
        authorization: localStorage.getItem("token")
      }
    };
    this.body.innerHTML = Loading.render();
    this.request(opts, (err, resp, data) => {
      if (err) {
        this.emit("error", err);
      } else {
        this.body.innerHTML = Template.render(data);
        this.addEventListener();
      }
    });
  }
  tarefaDoneCheckbox() {
    const dones = this.body.querySelectorAll("[data-done]");
```

```
    for(let i = 0, max = dones.length; i < max; i++) {
      dones[i].addEventListener("click", (e) => {
        e.preventDefault();
        const id = e.target.getAttribute("data-tarefa-id");
        const done = e.target.getAttribute("data-tarefa-done");
        const opts = {
          method: "PUT",
          url: `${this.URL}/tarefas/${id}`,
          headers: {
            authorization: localStorage.getItem("token"),
            "Content-Type": "application/json"
          },
          body: JSON.stringify({
            done: !done
          })
        };
        this.request(opts, (err, resp, data) => {
          if (err || resp.status === 412) {
            this.emit("update-error", err);
          } else {
            this.emit("update");
          }
        });
      });
    }
  }
  tarefaRemoveClick() {
    const removes = this.body.querySelectorAll("[data-remove]");
    for(let i = 0, max = removes.length; i < max; i++) {
      removes[i].addEventListener("click", (e) => {
        e.preventDefault();
        if (confirm("Deseja excluir esta tarefa?")) {
          const id = e.target.getAttribute("data-tarefa-id");
          const opts = {
            method: "DELETE",
            url: `${this.URL}/tarefas/${id}`,
            headers: {
              authorization: localStorage.getItem("token")
            }
          };
          this.request(opts, (err, resp, data) => {
            if (err || resp.status === 412) {
              this.emit("remove-error", err);
            } else {
              this.emit("remove");
            }
          });
        }
      });
    }
  }
}

module.exports = Tarefas;
```

Figura 36 - Arquivo src/componentes/tarefas.js

O método de *POST* para a criação de tarefas é aplicada na classe TarefaForm, esta classe também usará em forma de herança as características do serviço de tarefa para poder realizar evento de cadastro de tarefa. A função responsável pelo cadastro chama-se **formSubmit()** seu objetivo é utilizar os dados o formulário de cadastro e enviar para o serviço de tarefas utilizando o método *POST.* Como pode ser observa na Figura 37.

```
import Tarefa from "../tarefa.js";
import Template from "../templates/tarefaForm.js";

class TarefaForm extends Tarefa {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-tarefa]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const tarefa = e.target.querySelector("[data-tarefa]");
      const opts = {
        method: "POST",
        url: `${this.URL}/tarefas`,
        json: true,
        body: {
          title: tarefa.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 412) {
          this.emit("error");
        } else {
          this.emit("submit");
        }
      });
    });
  }
}

module.exports = TarefaForm;
```

Figura 37 - Arquivo src/components/tarefaForm.js

A aplicação web ficou com suas responsabilidades dividas entre os serviços de usuário que é responsável pela manipulação das informações de usuário através da url do serviço https://localhost:3001 , e o serviço de tarefas que tem como responsabilidade manipular as tarefas que utiliza a url https://localhost:3000.

4) CONCLUSÃO FINAL

Este trabalho de conclusão de curso foi uma forma de apresentar um conceito recente de como estruturar uma solução de forma escalável, onde o ganho computacional e a economia de recurso, acaba se tornando uma grande vantagem nos dias de hoje, ou seja, produzir mais, gastando menos.

Um motivo importante sobre a escolha da utilização da arquitetura de micro serviços é que grandes empresas de sucesso como Netflix, Amazon, InfoGlobo, Ubber entre outros utilizam esta arquitetura em suas soluções e entender toda a dinâmica e requisitos dessa estrutura é de fato importante em um mercado tão competitivo na área de tecnologia e informação.

Trabalhar com micro serviços não é algo simples, pois são vários serviços talvez dezenas ou centenas, dependendo do escopo da empresa e monitora-los é uma grande responsabilidade. Mas existe grandes vantagens como pode ser observado como a resiliência dos projetos, as entregas continuas por serem códigos reduzidos e com menos responsabilidades, escolha de ferramentas corretas para o problema proposto, melhora para agregar desenvolvedores no projeto entre outras vantagens.

Por mais que grande parte dos sistemas de hoje não adotem a arquitetura de micro serviço, acredito que seja uma questão de tempo para que migrem, as mudanças são difíceis de se fazer, porém no mundo competitivo ter capacidade de se adequar é o elemento chave para adotar tecnologias com melhores desempenhos e se pudesse resumir arquitetura de micro serviço em uma frase seria “Arquitetura de micro serviço é sinônimo de se mudar constantemente para se fazer algo ainda melhor”.

CRONOGRAMA

	2015				2016								
Atividades	**Set**	**Out**	**Nov**	**Dez**	**Jan**	**Fev**	**Mar**	**Abr**	**Mai**	**Jun**	**Jul**	**Ago**	**Set**
Leitura e levantamento das referências bibliográficas.													
Elaboração e entrega do pré-projeto.													
Estudo sobre a Arquitetura de micro serviço e Cloud Computing.													
Estudo das ferramentas utilizadas para o desenvolvimento do trabalho.													
Escrita para o Exame de Qualificação.													
Apresentação do Exame de Qualificação.													
Desenvolvimento da Aplicação de micro serviço													
Escrita final do trabalho.													
Defesa do trabalho.													
Entrega do TCC no núcleo de Monografia.													

Legenda: - Etapas concluídas , - Etapas a serem concluídas

REFERENCIAS BIBLIOGRAFICAS

ALECRIM, E. O que é cloud computer (Computação nas nuvens) ? **Info Wester**, 2008. Disponivel em: <http://www.infowester.com/cloudcomputing.php>. Acesso em: 28 Fevereiro 2016.

ALMEIDA, A. Arquitetura de microserviços ou monolítica? **Caelum**, 8 Fevereiro 2015. Disponivel em: <http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica/>. Acesso em: 8 Fevereiro 2016.

BIRD, J. Arquitetura de software em DevOps. **Imasters**, 2015. Disponivel em: <http://imasters.com.br/desenvolvimento/arquitetura-de-software-em-devops/?trace=1519021197>. Acesso em: 29 Fevereiro 2016.

BRAGA, W. R. SQlite um banco de dados pequeno e eficiente. **Blog Welington R. Braga**, 2008. Disponivel em: <http://blog.welrbraga.eti.br/?p=81>. Acesso em: 26 Julho 2016.

CORDEIRO, E. B. SOA - Arquitetura Orientada a Serviços. **Compartilhando conhecimento em BPM, SOA e ECM/GED**, 8 Outubro 2012. Disponivel em: <http://blog.iprocess.com.br/2012/10/soa-arquitetura-orientada-a-servicos/>. Acesso em: 23 Fevereiro 2016.

FERNANDES, C. O que é coud computing? **Techtudo**, 2012. Disponivel em: <http://www.techtudo.com.br/artigos/noticia/2012/03/o-que-e-cloud-computing.html>. Acesso em: 28 Fevereiro 2016.

FOWLER, M. Microservices. **Martin Folwer**, 08 Fevereiro 2014. Disponivel em: <http://martinfowler.com/articles/microservices.html>. Acesso em: 8 Fevereiro 2016.

HEMEL, Z. O modelo de implantação da Netflix: DevOps na veia e resiliência extrema. **Infoq**, 2013. Disponivel em: <http://www.infoq.com/br/news/2013/07/modelo-netflix-devops>. Acesso em: 10 Fevereiro 2016.

LASKEY, K. Reference Model for Service Oriented. **OASIS Advancing open standards for the information society**, 2006. Disponivel em: <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf>. Acesso em: 23 Fevereiro 2016.

LAURO, L. D. e-Talks | Cloud Computing e Software como Service. **Youtube**, 2013. Disponivel em: <https://www.youtube.com/watch?v=b7hjYnWGurg>. Acesso em: 28 Fevereiro 2016.

LITTLE, M. Microservices and SOA. **InfoQ**, 2014. Disponivel em: <http://www.infoq.com/news/2014/03/microservices-soa>. Acesso em: 23 Fevereiro 2016.

MACEDO, E. De monolito web para arquitetura de microservices: o caso GloboTV / GlobosatPlay. **Infoq**, 2015. Disponivel em: <http://www.infoq.com/br/presentations/de-monolito-web-para-arquitetura-de-microservices>. Acesso em: 9 Feveiro 2016.

MARTINS, R. MTC 2015 Testes em uma arquitetura de micro-serviços. **Youtube**, 2015. Disponivel em: <https://www.youtube.com/watch?v=17oDD9qQgOY>. Acesso em: 8 Fevereiro 2016.

MAURO, T. Adopting Microservices at Netflix: Lessons for Architectural Design. **NGINX**, 2015. Disponivel em: <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>. Acesso em: 18 Fevereiro 2016.

MORAES, W. B. **Construindo Aplicaçoes com Node.js**. [S.l.]: Novatec, 2015.

MOREIRA, R. H. O que é Node.js? **NodeBr**, 2013. Disponivel em: <http://nodebr.com/o-que-e-node-js/>. Acesso em: 26 Julho 2016.

NEWMAN, S. **Building Microservices**. [S.l.]: [s.n.], 2015.

NOBRE, A. Arquitetura microservices, segurança e gerenciamento de APIs. **Youtube**, 2015. Disponivel em: <https://www.youtube.com/watch?v=F7rmSEqp4A4>. Acesso em: 9 Fevereiro 2016.

PEREIRA, C. R. **Cosntruindo API Rest com Node.js**. [S.l.]: Casa do código, 2016.

RHUBART, B. Microservices and SOA. **Oracle Technology Network**, 2015. Disponivel em: <http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html>. Acesso em: 25 Fevereiro 2016.

RICHARDSON, C. Microservices: Decomposição de Aplicações para Implantação e Escalabilidade. **Infoq**, 8 Fevereiro 2014. Disponivel em: <http://www.infoq.com/br/articles/microservices-intro>. Acesso em: 8 Fevereiro 2016.

RODRIGUES, J.; GROFFE, R.; SIMAS, L. G. Microservices. **Youtube**, 2015. Disponivel em: <https://www.youtube.com/watch?v=wvkRm5hsw1E>. Acesso em: 29 Fevereiro 2016.

SAKURAI, R. Construindo um arquitetura moderna de microservices na empresa Gilt. **InfoQ**, 2015. Disponivel em: <http://www.infoq.com/br/articles/microservices-gilt>. Acesso em: 19 Fevereiro 2016.

SAKURAI, R.; CASCARROLHO, R. Arquitetura de Microsserviço. **Youtube**, 2015. Disponivel em: <https://www.youtube.com/watch?v=kRJZOdA14TM>. Acesso em: 25 Fevereiro 2016.

SEQUELIZE, D. Docs. **Sequelize**. Disponivel em: <http://docs.sequelizejs.com/en/latest/>. Acesso em: 26 Julho 2016.

STENBERG, J. Uma introdução a arquitetura de Microservices. **InfoQ**, 2014. Disponivel em: <http://www.infoq.com/br/news/2014/07/introducing-microservices>. Acesso em: 23 Fevereiro 2016.

VELTE, A. T.; VELTE, T. J.; ELSENPETER, R. **Cloud Computing:** Computação em Nuvem Uma Abordagem Prática. [S.l.]: Alta Books, 2012.

WEISSMANN, H. L. Repensando micro serviços (microservices). **Dev/kiko Experiências em desenvolvimento de software**, 2014. Disponivel em: <http://www.itexto.net/devkico/?p=1768>. Acesso em: 25 Fevereiro 2016.

YOSHIDA, R. Introdução a AWS EC2 e aos serviços de computação na Nuvem. **Youtube**, 2015. Disponivel em: <https://www.youtube.com/watch?v=Ky48XGoWexk>. Acesso em: 8 Fevereiro 2016.