INSTITUTO FEDERAL GOIANO – CAMPUS MORRINHOS
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

MARCO AURÉLIO CABRAL

ANÁLISE DA ARQUITETURA DE MICROSERVIÇOS

MORRINHOS – GO
2017

MARCO AURÉLO CABRAL

ANÁLISE DA ARQUITETURA DE MICROSERVIÇOS

Monografia apresentada ao Curso Superior de Tecnologia de Sistemas para Internet do Instituto Federal Goiano – Campus Morrinhos, como requisito parcial para obtenção de título de Tecnólogo em Sistemas para Internet.
Área de concentração: Engenharia de Software/Arquitetura de Software.
Orientador: Marcel da Silva Melo.
Co-orientador: Rodrigo Elias Francisco

MORRINHOS – GO
2017

Dados Internacionais de Catalogação na Publicação (CIP)

Sistema Integrado de Bibliotecas – SIBI/IF Goiano Campus Morrinhos

C117a Cabral, Marco Aurélio.

Análise da arquitetura de microserviços. / Marco Aurélio Cabral. – Morrinhos, GO: IF Goiano, 2017.
52 f. : il. color.
Orientador: Me. Marcel da Silva Melo.
Coorientador: Me. Rodrigo Elias Francisco.

Trabalho de conclusão de curso (graduação) – Instituto Federal Goiano Campus Morrinhos, Tecnologia em Sistemas para Internet, 2017.
1. Arquitetura. 2. Recursos Computacionais. 3. Microserviço. 4. WEB. I. Melo, Marcel da Silva. II. Instituto Federal Goiano. Tecnologia em Sistemas para Internet. III. Título

CDU 004.234

RESUMO

A evolução da rede mundial de computadores fez com que mais pessoas pudessem ter acesso a recursos antes não tão requisitados. Empresas do mundo inteiro hoje podem disponibilizar serviços na web para o acesso livre o que leva a uma alta taxa de recursos computacionais gastos. Acompanhando essa evolução, surge uma necessidade de economia para que mais requisições possam ser atendidas com menores custos computacionais requeridos. Também é necessário considerar a especificidade de cada tecnologia e a qual tipo de problema ela é mais direcionada a resolver. A arquitetura de microserviços propõe uma solução possível para um direcionamento de recursos objetivo para cada parte de um sistema podendo gerar resultados mais específicos e refinados. Este trabalho tem como objetivo apresentar os principais pontos dessa arquitetura através de um estudo de caso implementado utilizando 3 linguagens de programação diferentes, cada uma cuidando de uma responsabilidade específica no sistema. Ao final é realizada uma análise da arquitetura de microserviços, apresentando pontos positivos e negativos de sua utilização

Palavras-chave: Arquitetura – Recursos Computacionais – Microserviço – Web

ABSTRACT

The evolution of the World Wide Web makes that more people have access to resources not so requested before. Companies from the entire world can provide services on the web for free access what takes a high rate of computational resources. Following this evolution, a necessity of saving up those resources rises to make more requests answered with less computational costs needed. In addition, it is necessary to consider the specificity of each technology and what kind of problem it solves. The microservices architecture proposes a possible solution for guiding the resources for each part of the system what may cause a higher quality of results. This paper objectives present the main points of this architecture through a case study implemented using 3 different programming languages, each one taking care of a specific responsibility on the system. At the end, an analysis of the microservices architecture is performed showing pros and cons of using it.
Keywords: Atchitecture – Computational Resources – Microservices – Web

SUMÁRIO

1. INTRODUÇÃO

A arquitetura de software é um dos recursos no desenvolvimento de sistemas que melhor devem ser analisados antes da execução do projeto. De Sordi *et al* (2006), afirma que a arquitetura de software é um dos principais fatores capazes de facilitar o desenvolvimento, a manutenção e a evolução de sistemas e, concomitantemente, quando mal projetada pode comprometer a organização e gestão futura do produto. Diante do exposto, conclui-se que a forma com que as arquiteturas são projetadas também evoluem buscando atender novas demandas e satisfazer às novas necessidades de mercado.

O modelo mais utilizado de arquitetura agrega todas as funcionalidades empacotando-as e disponibilizando-as simultaneamente (DE SORDI *et al*, 2006). De acordo com Fowler (2014), esse modelo, também conhecido como arquitetura monolítica, cria uma série de problemas quando se trata de escalabilidade, evolução e manutenção. Nota-se que com o tempo, essa arquitetura passa a tornar seu repositório de códigos muito extenso e acoplado, fazendo assim com que alterações no código passem a ser extremamente trabalhosas e perigosas (THONES, 2015; NEWMAN, 2015; DE SORDI *et al,* 2006). Fowler (2014) ainda explica que isso acontece pois existe uma complexidade natural evidente em manter o sistema monolítico de forma modular, complicando os processos de reimplantação.

De acordo com Humble e Farley (2010), o processo de reimplantação tanto para correção de erros quanto para disponibilidade de novas funcionalidade e melhorias deve ser constante. Esse processo, chamado de entrega contínua, deve ocorrer periodicamente, logo, com a arquitetura monolítica, será necessário reconstruir toda a aplicação realizando também as reconstruções de partes desnecessárias, ou seja, que não sofreram alterações.

A disponibilização de todos os serviços agregados em um único lugar traz também complicações à disponibilidade. Essa abordagem no desenvolvimento cria o chamado *Single Point of Failure* (Ponto único de falha), ou seja, uma falha em uma das funcionalidades ou até mesmo na infraestrutura da aplicação fará com que esta fique sujeita a uma parada na sua execução, podendo se tornar ainda mais massiva em ambientes escalados (VILLAMIZAR, 2015).

Na medida em que uma aplicação cresce necessitando atender cada vez mais requisições, esta deverá passar por um processo de melhoria em sua infraestrutura. Esse processo é chamado de escala. Uma das formas gráficas de se representar os pontos de escala é o chamado *AFK Scale Cube*. Este cubo, apresentado na Figura 1, demonstra três pontos principais, seus eixos X, Y e Z, para o escalonamento de uma aplicação (ABBOTT E FISHER, 2009).

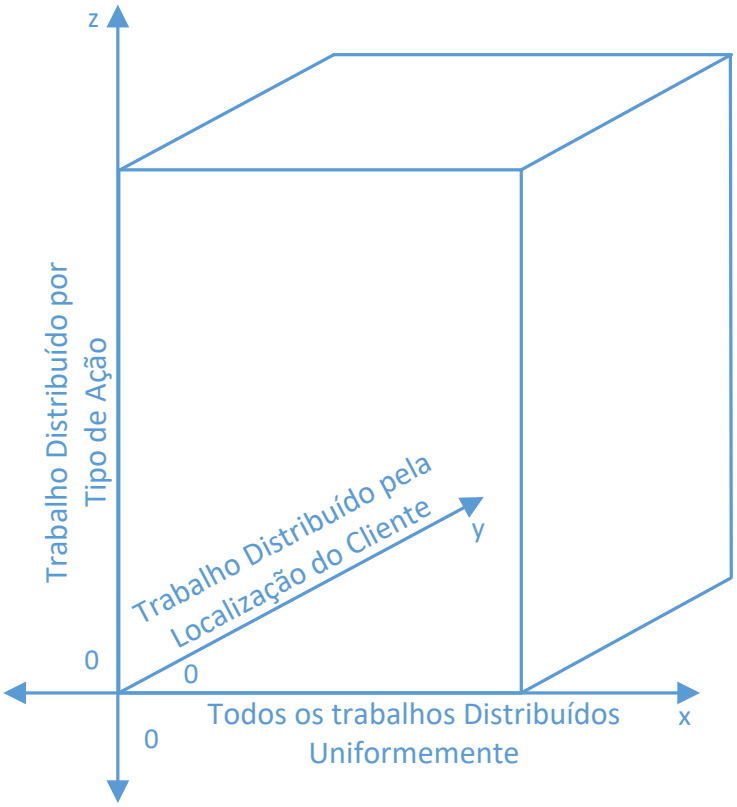

Figura 1 - AFK Scale Cube. Fonte: Abbott e Fisher (2009). Nota: Adaptado Pelo Autor

Durante o processo de escalabilidade no eixo X, referente à replicação de instâncias da mesma aplicação, esses problemas tendem a se multiplicar. Para cada novo local em que o software esteja sendo executado, terá de ser feita a reimplantação total da aplicação para efetuar qualquer tipo de alterações ou correções. Além disso nota-se mais um problema, a escalabilidade é feita com a intenção de melhorar os recursos computacionais disponíveis ao sistema. Ao se colocar todas as funcionalidades agregadas de um sistema monolítico sendo executadas no mesmo ambiente, em boa parte dos casos, funcionalidades que precisam de pouco recurso computacional estarão consumindo processamento desnecessário (FOWLER, 2014).

Baseado nesses problemas, a arquitetura de microserviços surgiu como proposta de solução. Esse modelo visa a divisão de funcionalidades dos sistemas em pequenos módulos que possuem apenas uma responsabilidade para que possam ser desenvolvidos, mantidos e escalados separadamente sendo que cada um desses serviços deve funcionar de forma independente, permitindo uma entrega contínua produtiva. Isso também permite a divisão dos possíveis pontos de falha em diversos lugares e um tratamento das funcionalidades mais fino, sendo que os microserviços podem ser desenvolvidos em tecnologias diferentes, como a linguagem de programação e o banco de dados (FOWLER, 2014; NEWMAN, 2015; THONES, 2015).

Este trabalho tem como objetivo principal apresentar a forma com que essa arquitetura é estruturada. Serão exibidos estudos feitos sobre seus principais conceitos como o protocolo de comunicação utilizado nesse trabalho, noções de arquitetura de software, uma comparação com o modelo de arquitetura monolítica (principal modelo paralelo à esta) e por fim será apresentado um estudo de caso feito com três microserviços desenvolvidos em linguagens diferentes.

1.1. Estrutura do Trabalho

O trabalho está estruturado da seguinte forma. No capítulo 2 é apresentado o levantamento teórico sobre alguns dos principais assuntos presentes nesse trabalho: arquitetura de software, arquitetura monolítica, *Service Oriented Architecture* (Arquitetura orientada a serviços ou *SOA*), as linguagens utilizadas no estudo de caso, *Respresentational State Transfer* (transferência em estado representacional ou REST), API *gateway* e o protocolo de comunicação utilizado, o *Hyper Text Transfer Protocol* (Protocolo de transferência de hipertexto ou HTTP).

O capítulo 3 mostra as características que podem ser observadas nos microserviços. Dentro desse capítulo estão presentes os aspectos a serem considerados ao se tratar de microserviços juntamente com os princípios a serem seguidos. Além disso, durante o desenvolvimento do capítulo, é traçado um paralelo de comparação com a arquitetura monolítica.

No capítulo 4 é apresentado o desenvolvimento do estudo de caso. São apresentados os três microserviços representando três diferentes funcionalidades de um sistema enquanto no capítulo 5, é apresentado a análise sobre o desenvolvimento. Nesse capítulo são apresentados pontos fortes e fracos observados e dificuldades encontradas na composição dos microserviços.

No capítulo 6 é apresentado a conclusão do trabalho. Nesta sessão são apresentadas as considerações finais sobre como a arquitetura de microserviços pôde ser aplicada em ambiente de desenvolvimento e quais os principais resultados obtidos com este estudo.

2. REFERENCIAL TEÓRICO

2.1. Arquitetura de Software

A arquitetura de software é uma ferramenta que proporciona ampla ajuda durante o desenvolvimento de sistemas. Uma definição mais clássica é apresentada por Shaw e Garlan (1996). Segundo estes autores, a arquitetura de software define o formato de cada componente do sistema e como é feito o diálogo entre os mesmos.

Segundo De Sordi *et al* (2006), ela dispõe de recursos que permitem ganhos durante a manutenção e evolução de sistemas, fator que é de grande valia em ambientes de disputa comercial. Chen (2002) ainda afirma que seu préstimo mais evidente é o de permitir descrever, observar e justificar o comportamento de um sistema e também auxiliar na produção de implementações de componentes reusáveis.

Um dos principais pontos da arquitetura de software é o de desenvolver tais componentes com alta coesão e baixo acoplamento. A alta coesão permite atribuir simplicidade e um sentido claro às funcionalidades desenvolvidas. Apresentar clareza na aplicação facilita durante o desenvolvimento quando alguma parte do código escrito necessita ser reutilizada. Já o baixo acoplamento auxilia no relacionamento mais simples entre componentes, não gerando uma dependência direta entre cada parte. Ainda, contribui para minimizar o efeito dominó de possíveis problemas se propagarem através da aplicação, já que centralização de uma determinada funcionalidade permite que apenas o componente com problema precise ser alterado (OLIVA, 1998; SHAW E GARLAN 1996).

Garlan e Perry (1995) ainda afirmam que a expressão “Arquitetura de Software” pode possuir diversas formas de ser compreendida. A mais aceita dentro da engenharia de software é a definição dos componentes que constituem um determinado sistema. Fowler (2002) ratifica essa explicação acrescentando que a divisão desses componentes deve apresentar uma separação em seu nível mais alto. Outra declaração sobre o significado de arquitetura de software que Fowler (2002) traz é que ela corresponde às decisões difíceis de serem mudadas.

Em suma, compreende-se que a arquitetura de software dita os padrões a serem obedecidos durante o desenvolvimento de um sistema. Considerando o que foi dito por Fowler (2002), conclui-se que o custo de se mudar um padrão é alto. Logo, é

de alta relevância que no início do projeto seja dedicado um tempo voltado para a criação da arquitetura.

Esses padrões correspondem à forma com que componentes e camadas do software se comunicam entre si, por exemplo, como são instanciadas classes ou como cada camada acessa outra. Ao se ter o conhecimento de como cada componente do sistema trabalha e como ele se dispõe, obtém-se agilidade durante qualquer tipo de alteração, manutenção ou remoção de recursos.

2.2. Arquitetura Monolítica

De acordo com Kurhinen (2014) e Fowler (2014), durante a evolução das metodologias de desenvolvimento de software, a arquitetura monolítica foi adotada como a mais tradicional. Ainda segundo Fowler (2014), a maioria dos projetos construídos dessa forma utilizam uma camada responsável pela interação com o usuário (normalmente constituídas por páginas web ou formulários *desktop*), o banco de dados onde estão persistidas as informações tratadas e uma aplicação sendo executada em um servidor, a qual recebe, processa e devolve conteúdo aos clientes. E também, segundo Kurhinen (2014), estas aplicações apresentam uma estrutura semelhante em gerenciamento de banco de dados, com grandes bases de informações centradas.

As aplicações são comumente constituídas com um padrão de projetos chamado MVC (*Model, View e Controller* ou Modelo, Visão e Controle), assim como é mostrado na Figura 2.

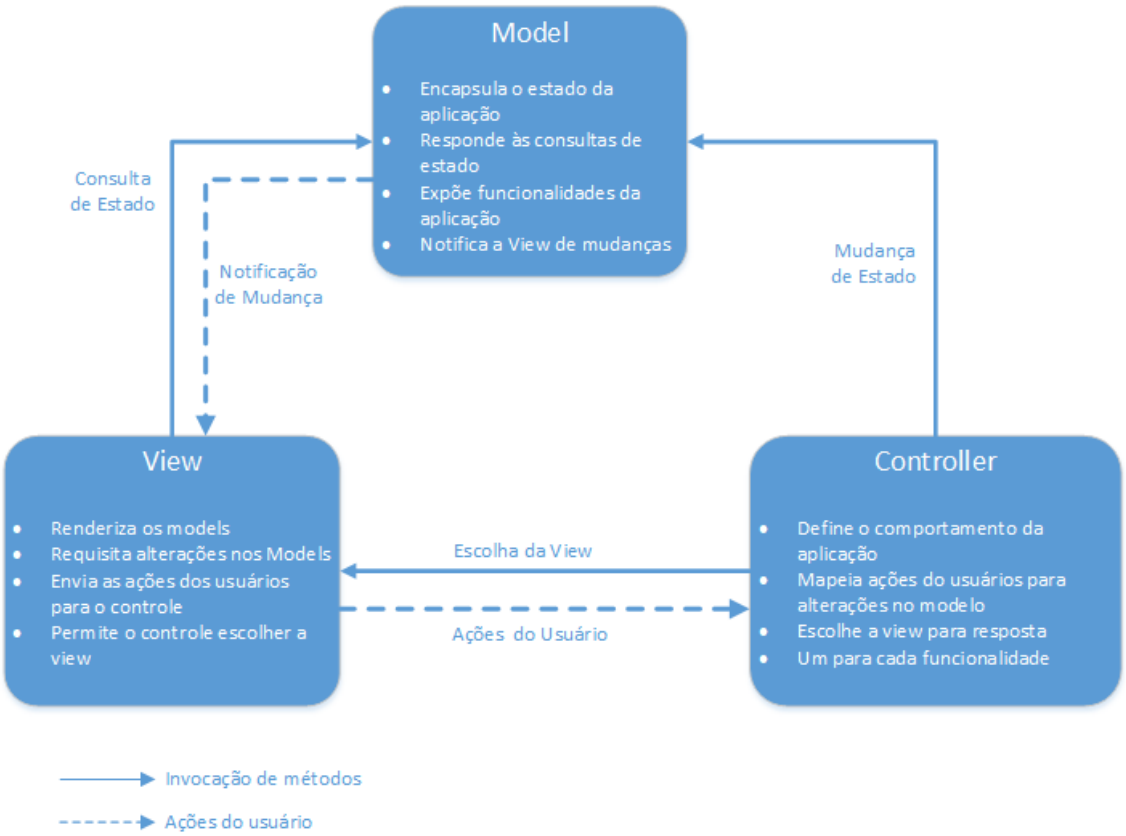

Figura 2 - Padrão MVC. Fonte: Bass (2007). Nota: Adaptado Pelo Autor

Sharan (2015) afirma que essas camadas da aplicação seguem um fluxo ordenado para receber informações, processar e obter resultados como mensagens de respostas ou a persistência dos dados, cada camada tendo conhecimento em níveis distintos das outras camadas.

A camada de *view* (Visão) é a responsável pela exibição e entrada de dados, sendo o ponto de comunicação do sistema com o usuário final. Através dessa camada o sistema recebe requisições contendo solicitações ou envios de informações para dentro do sistema. A camada de *controller* (Controle) é responsável por receber todas as requisições da camada de *view*. O *controller* também é considerado como a camada de ligação entre a *view* e *model* (Modelo), possuindo autonomia para alterar as informações exibidas na *view* e ainda manipular as informações contidas nos elementos da camada de *model*. Por fim a camada de *model* é onde estão presentes os componentes que descrevem as entidades que o domínio do sistema trata. A separação dessas camadas reitera a agregação de flexibilidade para mudanças necessárias, já que o tratamento de responsabilidades específicas permite alterações em partes específicas dentro do sistema (SHARAN, 2015; GAMMA, 2009).

Nota-se nessa disposição de componentes algumas vantagens. Primeiramente, a designação de responsabilidades específicas para cada camada permite a centralização das responsabilidades, assim colaborando de forma ágil para qualquer alteração. Além disso, responsabilidades como cálculos e funcionalidades como a leitura de arquivos ou a comunicação com serviços externos podem ser reaproveitadas e serem disponibilizadas para outras plataformas de maneira uniforme.

Com a arquitetura definida dessa forma, nota-se que todas as chamadas de funcionalidades são feitas dentro do mesmo processo, o que por sua vez, são menos custosas. Não existindo obrigatoriamente uma camada de rede separando as chamadas de processos, problemas como latência de rede, necessidade de banda, conexão com a internet e gargalos não se tornam desafios obrigatórios a serem superados. Isso facilita até mesmo a junção de novas funcionalidades aos recursos disponíveis para os desenvolvedores (RICHARDSON, 2014, FOWLER, 2014).

As ferramentas existentes para desenvolvimento de software, como as IDE's (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado), são projetadas para auxiliar o programador durante os ciclos de criação de sistemas. Assim como as aplicações monolíticas evoluíram inicialmente como uma arquitetura de software padrão, as IDE's também seguiram se adequando à essa forma de desenvolvimento. Logo, estas possuem uma série de funcionalidade que ajudam e facilitam a abordagem da arquitetura monolítica (RICHARDSON, 2014). Assim, com todas as suas funcionalidades agrupadas, os processos de desenvolvimento de testes e depuração do código podem ser feitos com mais facilidade. As IDE's desenvolvem um papel fundamental nessa etapa (FOWLER, 2014).

Assim, observa-se que o início de um projeto sem a preocupação extra presente em sistemas distribuídos pode facilitar a consolidação de um produto estável em menor tempo. Como ponto de partida, a arquitetura monolítica proporciona o conforto de não se trabalhar de forma distribuída obrigatoriamente, mas é durante sua evolução que os problemas surgem. Segundo Fowler (2015), o gráfico apresentado na Figura 3 pode demonstrar os índices de rendimentos de cada tipo de arquitetura através do tempo.

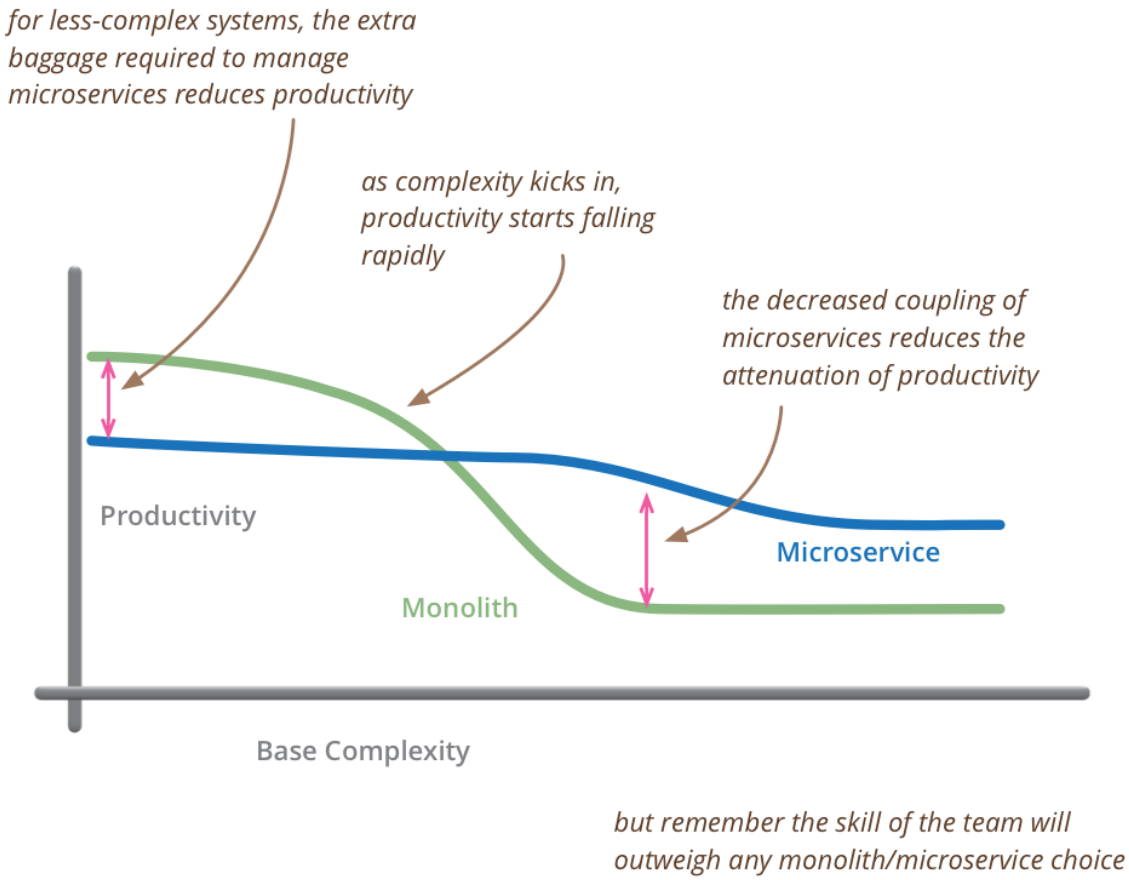

Figura 3 - Produtividade de cada Arquitetura através do tempo. Fonte: Fowler (2015)

Como apresentado no gráfico na Figura 3, Fowler (2015) afirma que o crescimento da complexidade e extensão do projeto faz com que a produção caia com o passar do tempo, seja na abordagem monolítica ou de microserviços. Também se nota que no desenvolvimento inicial do projeto, a arquitetura de microserviços perde em questão de produtividade, mas se mostra mais rentável já que sua linha de resultados através do tempo não apresenta uma queda tão drástica em relação à linha referente aos monólitos.

Newman (2015) ainda completa dizendo que essa dificuldade aumenta quando é realizada a adição de novas funcionalidades e que mesmo sistemas desenvolvidos com o foco em modularização, tendem a apresentar uma complexidade e a ter suas separações quebradas ao passar do tempo.

Ao se ver um cenário descrito dessa forma, é perceptível uma maior dificuldade durante qualquer tipo de manutenção de código, mesmo com a busca constante pela alta coesão. Fowler (2014) explica que esse é um problema comum em aplicações que vem crescendo por muito tempo. Seus repositórios de código se

tornam extensos e pedaços de código passam a ser replicados e cada vez menos reutilizados.

Ainda, a aglomeração de utilidades diversas dentro do mesmo local pode comprometer o desempenho. O monolito deverá ser implantado compactando tudo no mesmo lugar, logo, no mesmo ambiente, com os mesmos recursos disponíveis para todas as partes do sistema.

Durante o processo de implantação será necessário um esforço de coordenação de todas as equipes responsáveis por qualquer tipo de mudança ou nova funcionalidade no sistema. Por terem todos os procedimentos entrelaçados um problema em uma parte pode comprometer a operação e o fluxo natural de todo ambiente. As complicações crescem à medida em que novas implantações precisam ser feitas, já que a organização e testes deverão ser refeitos, custando mais tempo e produtividade de todos os times de desenvolvimento envolvidos (NAMIOT E SNEPS-SPEPPE, 2014; FOWLER, 2014; THONES, 2015).

Ainda tendo em mente a aglomeração das funcionalidades no monolito, um bom exemplo de desperdício de recurso é um software que pode necessitar de uma alta capacidade de processamento para uma determinada funcionalidade e um acesso à disco em outra. Nesse cenário, a otimização do ambiente que o executa fará necessário que existam tecnologias especializadas em ambos os casos. Essas necessidades ainda podem aumentar pois existem cenários em que uma determinada parte do sistema pode precisar de uma maior largura de banda ou uma maior alocação de memória (FOWLER, 2014; NEWMAN, 2015; THONES, 2015).

Os problemas citados anteriormente podem ser contidos de forma parcial respeitando o princípio da responsabilidade única não somente em suas classes, mas no sistema como um todo. Esse princípio descreve inicialmente que classes devem ser criadas com somente um motivo para serem mudadas e, que caso seja encontrado outro motivo para mudança, a classe deverá ser subdividida. Por se tratar de uma das principais boas práticas de programação, esse princípio incentivou a busca por uma nova abordagem na forma com que sistemas são desenvolvidos (NEWMAN, 2015; FOWLER, 2015; MARTIN, 1996).

2.3. Arquitetura Orientada a Serviços (SOA)

A SOA traz uma abordagem não convencional aos projetos de software. De acordo com Morgado (2013) e Khadka *et al* (2013), esta teve seus princípios na

orientação a objetos surgindo assim com a intenção de produzir soluções modularizadas, com códigos reutilizáveis e com a capacidade de captação de serviços. Ainda segundo Khadka *et al* (2013) e Papazoglou (2007), esse paradigma permite uma grande flexibilidade para composição de serviços agregando as regras de negócio do domínio do sistema e mantendo um baixo acoplamento.

Além disso, a capacidade de pluralidades dessa arquitetura em determinados aspectos é um ponto considerável. Esse paradigma permite organizar propensões de diferentes partes de um sistema de forma distribuída, mesmo pertencendo a lógicas de domínios diferentes ou com implementações em ambientes e linguagens diferentes. Logo, um leque de possibilidades é aberto para o desenvolvimento, permitindo tratamentos detalhados para cada necessidade (NIKUL, 2007; KHADKA *et al* 2013; ERL, 2009).

Segundo Khadka *et al* (2013) e Zhang (2010), sistemas empresariais devem ser projetados pensando-se na evolução para que possam responder às novas oportunidades de negócio e também para que possam sofrer manutenções facilmente. SOA consegue tratar esse problema por conta da sua modularização natural capaz de tratamentos de negócios separados e, segundo Erl (2009), feitos por grupos de serviços agregados. Assim nota-se uma tendência de migração para arquitetura, principalmente em grandes ambientes empresariais.

Essa tendência é notada pela afirmação de Khadka *et al* (2013) reiterando que problemas com manutenção e evolução são recorrentes quando se trata de grandes sistemas legado. A falta de mão de obra qualificada (por se tratar na maioria dos casos de tecnologias obsoletas), pouca documentação e arquitetura inflexível são somente alguns dos incidentes mais comuns. Porém Khadka *et al* (2013) e Zhang (2010) ainda afirmam que apesar de apresentarem fatores que colaboram para o engessamento do sistema, o software em si ainda encapsula regras de negócio que são de grande valia para a empresa. Então, mesmo possuindo dificuldades de modificações em um software herdado, SOA pode ser utilizada para reaproveitar esses recursos já existentes deixando que novos processos e novas funcionalidades sejam agregadas ao sistema.

Mesmo sendo uma boa opção para melhoria de sistemas antigos, SOA pode apresentar algumas dificuldades em sua implementação. O processo de migração é um investimento a longo prazo onde deve ser aplicado uma grande quantidade de recursos que durante esse processo, normalmente são localizados

vários tipos de problemas, não só computacionais, referentes à estrutura do software, mas também problemas quanto aos processos de negócio. Ainda assim, tudo isso é considerado viável para atingir sistemas de larga escala e alta capacidade de comunicação com outros ambientes ou outras aplicações (KHADKA *et al,* 2013; ERL, 2009).

Para construir uma estrutura desse porte, uma malha de comunicação altamente distribuível é necessária. Essa espinha dorsal tem em uma de suas funcionalidades endereçar os serviços tornando-os disponíveis para outros serviços ou outras aplicações. A E*nterprise Service Bus* (ESB) assume essa responsabilidade, servindo como *middleware* para distribuição das requisições para cada ponta do sistema e também cuidando de serializações sobre demanda de cada serviço. Através desse meio a comunicação, é comumente estabelecida usando-se *webservices* com a arquitetura REST, permitindo um caminho comunicativo facilmente estabelecido e capaz de suportar grandes tráfegos de dados (ERL, 2009; MORGADO, 2013; PAPAZOGLOU *et al*, 2007).

A Figura 4 exibe o ESB exercendo o papel de barramento de serviços.

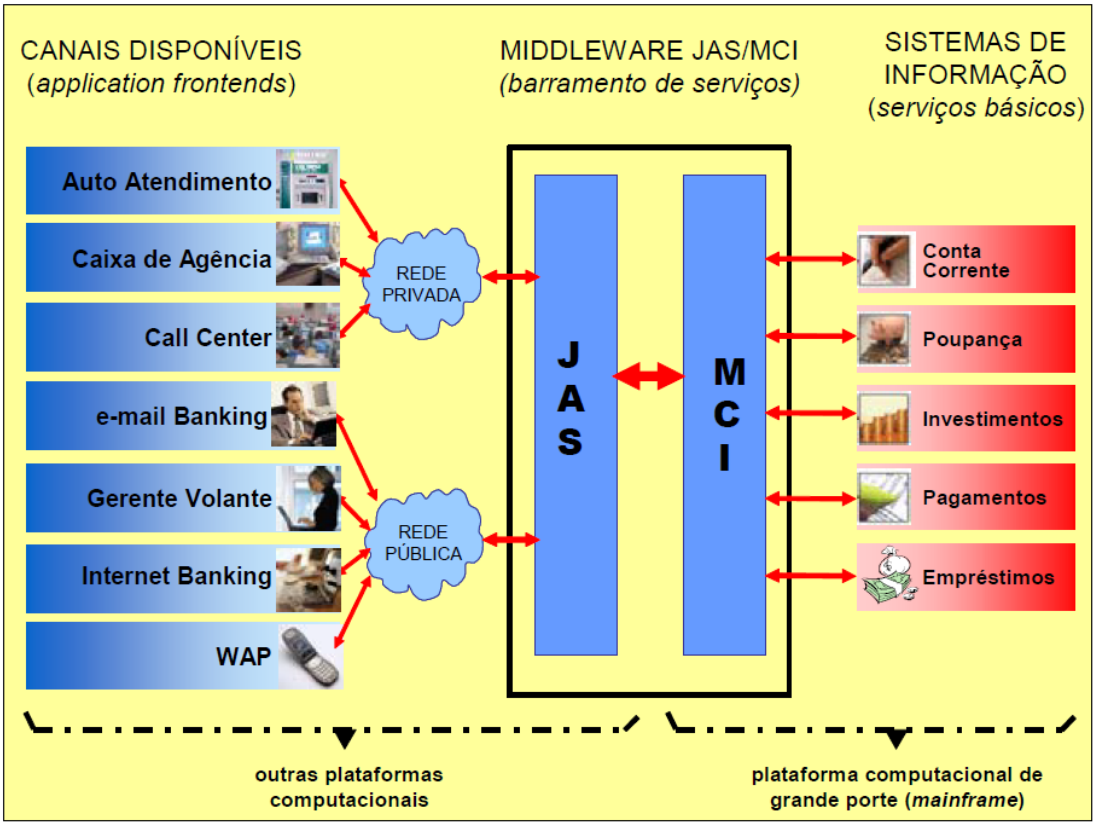

Figura 4 - Comunicação dos serviços através do ESB JAS/MCI. Fonte: De Sordi et al (2006)

A forma com que os sistemas corporativos se tornam mais distribuídos veio a ser uma proposta chamativa para cada vez mais empresas adotarem esse paradigma. SOA permite que softwares sejam desenvolvidos pensando na capacidade individual de cada serviço e também que softwares legados sejam reaproveitados de forma incremental. Tais serviços, que podem ser evoluídos de forma individual, atendendo melhor a cada nova necessidade (ERL, 2009; KHADKA *et al*, 2013; ZHANG 2010).

2.4. REST

Em ambas as arquiteturas tratadas neste trabalho, microserviços e SOA, sendo os dois sistemas distribuídos, um padrão de comunicação se faz necessário para que haja a troca de informações entre as partes do ambiente do sistema. O padrão de comunicação REST pode ser utilizado para tal tarefa em ambos os serviços, principalmente por sua simplicidade de estrutura e facilidade de compreensão (FOWLER, 2014; BARRY, 2010; RYAN, 2012; RICHADSON, 2014).

Esse modelo possui algumas restrições formais e princípios a serem seguidos. Cada uma das restrições foi pensada com a finalidade de prover um serviço de distribuição de hipermídia capaz de suprir as necessidades da crescente rede mundial de computadores. Assim, itens como escalabilidade, eficiência de rede e confiabilidade foram os principais intuitos que regraram a construção dessa arquitetura (FIELDING, 2000; HAUPT, 2014).

A primeira característica apresentada é da arquitetura cliente-servidor. Esse estilo permite a separação de responsabilidades entre as interfaces acessadas pelo usuário e o processamento dos dados juntamente com sua persistência. Ao separar tais tarefas é possível obter ganhos em portabilidade, onde somente as interfaces dos usuários deverão ser reescritas.

Logo em seguida é apresentado o conceito de conexão naturalmente sem estado. Essa restrição, primeiramente, obriga que cada requisição deve conter todos os dados necessários para sua compreensão. Tal ressalva permite a melhora de três elementos: visibilidade, confiabilidade e escalabilidade.

A visibilidade é melhorada em questões de monitoramento já que cada requisição contém as suas informações de origem/destino além da própria informação que transporta. Assim uma única requisição é necessária para se encontrar sentido

em uma transmissão. Já a confiabilidade é melhorada através da facilidade de recuperação de falhas parciais por não ser necessário recriar os dados encapsulados na requisição.

Subsequentemente, a escalabilidade é apurada pois preliminarmente o servidor não precisa guardar informações entre uma requisição e outra, permitindo assim a liberação rápida do seu processamento, logo, também não é necessário o gerenciamento de tais recursos entre os envios.

Ainda assim, tais benefícios vêm juntos a um custo. O servidor que provém o serviço perde parte do controle sobre a consistência, já que a persistência do estado por parte das aplicações clientes dependem de uma implementação correta e semântica. Uma conexão pode reduzir a performance de transmissão através da rede por conta das informações repetitivas sendo constantemente enviadas. Mesmo assim, o problema de tráfego excessivo pode ser auxiliado por mais uma restrição apresentada para essa arquitetura.

Para o melhoramento do desempenho de rede, essa arquitetura permite a utilização do recurso de cache. A implementação desse processo requer que os dados enviados sejam implicitamente ou explicitamente rotulados como “cacheáveis” ou não, dando assim orientação para o cliente sobre quais informações podem ser reusadas.

Até então foram apresentadas restrições somente quanto à estrutura e transmissão dos dados. Quanto à forma com que os dados são acessados é utilizado o conceito de interface uniforme. Cada entidade deve ser representada por uma única interface capaz de identificar e manipular cada recurso de forma semântica junto à especificação do protocolo de comunicação a ser usado (FIELDING, 2000; FERNÁNDEZ *et al,* 2010).

O protocolo de comunicação utilizado nesse trabalho será o HTTP. Este é tido como padrão para a *World Wide Web* (WWW), e também um dos protocolos mais conhecidos e utilizados. Através dele, é especificado o formato de mensagens trocadas entre servidores e clientes para se estabelecer comunicação e realizar a troca de mensagens de hipertexto. (TANENBAUM, 2003).

O protocolo HTTP dispõe de alguns métodos específicos para seu funcionamento. Os 4 métodos mais utilizados são: POST, GET, PUT e DELETE. Cada um correspondente à uma operação executada pelo sistema, onde todos os métodos podem ser acessados através de uma URI (BERNERS-LEE *et al*, 1996; NETWORK WORKING GROUP *et al*, 1999; FIELDING, 2000; KUROSE *et al,* 2010).

Tal restrição é ideal para sistemas de granularidade alta, onde o detalhamento da informação é menor. Fielding (2000) e Fernández *et al* (2010) especificam que interfaces uniformes simplificam a arquitetura de comunicação do sistema, assim como também melhoram a visibilidade das interações que movimentam grandes blocos de informações. Apesar disso, em casos de granularidade fina, onde o conteúdo é mais específico e detalhado, essa implementação se torna exaustiva e degradante quanto à eficiência tendo que ser desenvolvido diversos pontos de acesso com informações semelhantes.

3. MICROSERVIÇOS

Conforme já apresentado anteriormente, as tecnologias e suas estratégias adotadas tendem a acompanhar a evolução das novas necessidades. Essa evolução se prova eficaz em alguns cenários.

É comum que sistemas sejam desenvolvidos por grandes equipes durante vários anos. Esses sistemas, mesmo produzidos de forma arquitetada, se tornam extensos, com código rígido e atado a tecnologias defasadas. Essas são características comuns encontradas em sistemas legado baseados na arquitetura monolítica (FOWLER, 2014; NAMIOT e SNEPS-SNEPPE, 2014).

Tendo em vista os problemas encontrados em sistemas monolíticos, a arquitetura de microserviços traz uma nova abordagem. Baseada na ideia de *Single Responsability Principle,* esta arquitetura segue alguns princípios para que o sistema como um todo seja separado em pequenas unidades de trabalhos responsáveis por um único tipo de tarefa, sendo mantidas e escaladas separadamente por equipes distintas e autossuficientes (FOWLER, 2014; MARTIN, 1996; NEWMAN, 2015; THONES 2015).

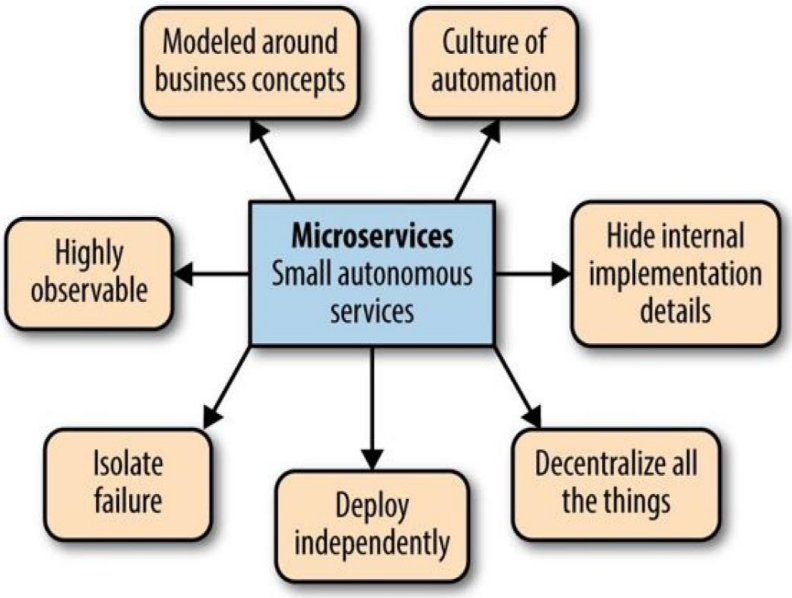

Figura 5 - Princípios dos Microserviços. Fonte: Newman (2015)

Conforme apresentado na Figura 5, Newman (2015) mostra tais princípios como caminhos a serem seguidos para alcançar microserviços autônomos e ao mesmo tempo capazes de trabalharem juntos, descrevendo cada um da seguinte forma:

- *Modeled Around Business Concepts* (Modelado em volta de conceitos de negócio): As interfaces de comunicação dos serviços, se ligadas através do contexto de negócio, serão mais estáveis que aquelas ligadas por conceitos técnicos. Assim, criando uma modelagem baseada no domínio, os serviços ficarão mais capazes de suportar mudanças em processos já existentes ou adaptação para novas regras de negócio. Sedo assim, os serviços devem ser criados baseando-se na auto capacidade de execução de negócio implementando todos os seus passos técnicos como a leitura, validação e persistência dos dados. Um erro evidente seria a criação de outros serviços relativos aos passos supracitados.
- *Culture of Automation* (Cultura de automação): A arquitetura de microserviços adiciona muita complexidade ao ecossistema de execução. Processos repetitivos, como o de construção e implantação, podem e devem ser automatizados evitando assim falhas humanas. Da mesma forma, testes automatizados auxiliam com a verificação de problemas técnicos e também servem como validadores das regras de negócio.
- *Hide Internal Implementation Details* (Esconda detalhes internos de implementação): É imprescindível que microserviços não dependam de detalhes da elaboração dos outros microserviços e de seus respectivos bancos de dados. Isso remete ao fato de que, por natureza, estes devem ser evoluídos independentemente uns dos outros, o mais desacoplado possível. Desta forma, mudanças em tecnologias e detalhes de execução devem ser irrelevantes às comunicações externas já que requisições remotas se comunicarão unicamente com uma interface, como REST. Tendo isso em mente, os microserviços devem evitar a todo custo a exibição de exceções padrões de uma determinada linguagem ou a publicação da pilha de chamadas para elementos externos. Uma implementação mais propícia deve se resguardar em códigos de erros como os do protocolo HTTP.
- *Decentralize All the Things* (Descentralize todas as coisas): Esse princípio é focado na organização dos times de desenvolvimento. É importante manter a atenção para chances de delegar capacidade de decisão às equipes. O ideal é que sejam independentes e auto capazes de cuidar dos estágios de desenvolvimento, testes, implantação e operação de seus serviços. Uma equipe autossuficiente deve conter tantos os membros do time de programação quanto

analistas de negócio e analistas de operações, como os administradores de redes e servidores. Alinhar os times desta forma permite que seus membros se tornem especialistas no seu domínio como um todo, fazendo com que a lei de Conway possa funcionar dentro do sistema. Segundo Kwan *et al* (2015), essa lei prevê que sistemas tendem a serem cópias das estruturas do domínio em que são baseados, logo, equipes separadas tendem a desenvolver sistemas de forma desacopladas.

- *Independently Deployable* (Implantação independente): Os microserviços devem ter como regra, e não como exceção, a capacidade de serem implantados sem a necessidade de organizar um processo que envolva necessariamente outros serviços. Isso permite um aumento da independência de cada time de desenvolvimento, não precisando se restringir ou orquestrar um processo de implantação extenso, assim facilitando a evolução da aplicação. A única restrição deve ser a respeito de mudanças com capacidade para quebrar a interface oferecida pelo serviço. Nesse caso, é recomendável que a versão anterior ainda permaneça disponível durante um período, permitindo que quem a utiliza possa mudar gradativamente para a nova versão.
- *Isolate Failure* (Falha Isolada): A arquitetura de microserviços pode possuir uma resistência às falhas maior que a monolítica. Para que isso seja verdade é preciso que haja planejamento para qualquer falha que possa acontecer, e elas vão acontecer. A ausência desse tipo de planejamento pode resultar em uma estrutura ainda mais frágil que a monolítica. Cada serviço deve ser reativo a falha de outros serviços que ele dependa continuando operacional. Isso quer dizer que um determinado serviço deve saber como se portar perante a erros como o de comunicação ou instabilidade de outros serviços. O ideal é que o sistema consiga produzir um resultado informativo se utilizando de informações presentes em cache ou consiga comunicar sua incapacidade de produção de uma resposta.
- *Highly Observable* (Altamente Observável): Por se tratar de um sistema distribuído, não é possível confiar na monitoria e *status* de somente uma máquina ou serviço. Neste quesito, a monitoria semântica, que permite monitorar os fluxos completos de negócios, exerce um papel mais abrangente e confiável quanto ao estado do sistema como um todo. Esse tipo de monitoria junto com a agregação dos *logs* permitem a rastreabilidade confiável de problemas e erros.

Além das informações de negócio, é de grande valia implementar recursos para saber o estado da infraestrutura em tempo real para tomada de decisões como a alocação de maiores recursos computacionais ou a liberação dos mesmos.

Em suma, esses princípios permitem que os microserviços se tornem especialistas no que fazem de forma individual. Ao mesmo tempo, trabalham juntos para um objetivo em comum, o sistema como um todo, e também, devem ser monitorados e acompanhados de perto. Dito os princípios, serão apresentadas as principais vantagens desta arquitetura.

As novas funcionalidades são desenvolvidas de acordo com novos requisitos emergentes do mercado no qual o domínio abrange. Atualmente, entidades como as grandes corporações enfrentam um ambiente cada vez mais exigente quanto às respostas de novas tendências. Dessa forma, conclui-se que novas funcionalidades dentro de um sistema são a representação de conformidade para tais exigências, que em suma, agregam mais valor ao produto (KHADKA *et al*, 2013; RICHARDSON, 2017; THÖNES, 2015).

Mesmo com o desenvolvimento de novas funcionalidades em tempo hábil, um problema recorrente é o de colocá-las em produção. O processo de implantação de aplicações monolíticas gera facilmente impasses. Primeiramente, este deve ser coordenado através da organização dos times de desenvolvimento já que todos operam simultaneamente, criando e agrupando funcionalidades em um ponto único. Com o software implantado, uma falha em qualquer parte do sistema pode se propagar para todo o resto, prejudicando mesmo os recursos que estão fora do contexto da falha. Consequentemente para a correção de uma falha todo o sistema deverá ser reimplantado (DAYA *et al,* 2016; FOWLER, 2015; THÖNES, 2015).

Já nos microserviços, dentro de seus princípios, existe o isolamento. Segundo Fowler (2014), os serviços não são executados no mesmo processo e são implantados separadamente, sendo que os times que os desenvolvem também agem de forma isolada, possuindo a total autonomia sobre qual tecnologia utilizam, como é estruturado o projeto, quais alterações são necessárias e também, quando e como será a implantação. Fowler (2014), assim como Newman (2015), afirma que as equipes devem ser modeladas baseadas nas funções exercidas por cada integrante voltado para o contexto de negócio, afim de tornar a equipe autossuficiente. Assim, podem estar presentes não só membros com conhecimentos tecnológicos, mas também analistas do negócio, como é mostrado na Figura 6.

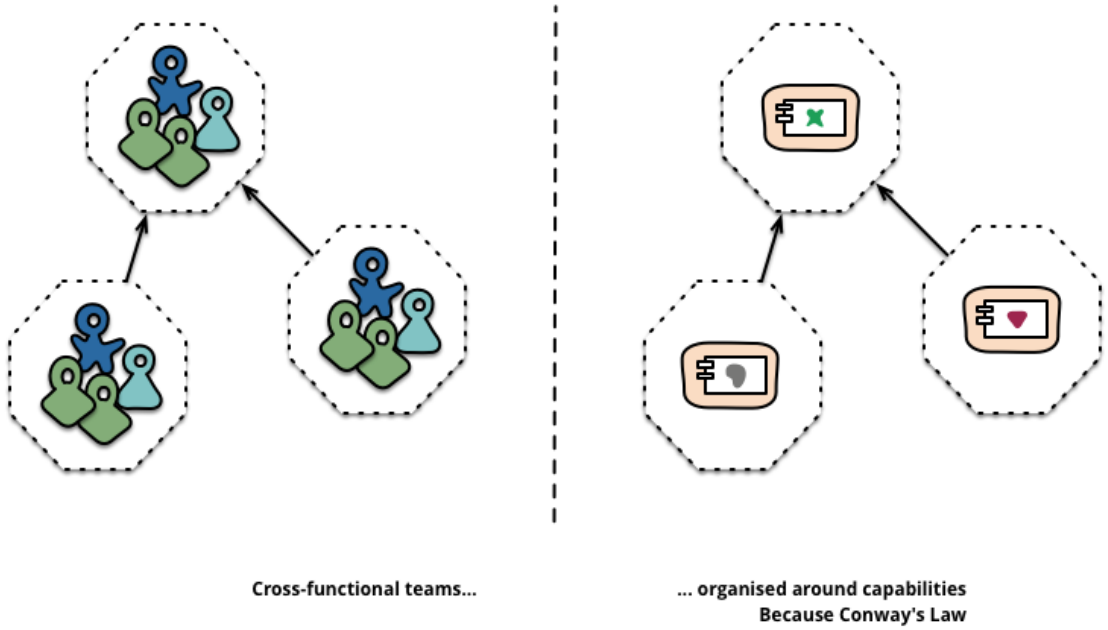

Figura 6 - Organização dos times de Microserviços. Fonte: Fowler (2014)

Mesmo com o isolamento de falhas, um problema ainda é notado: o que acontece quando um serviço falha e outros serviços dependem dele? Os microserviços devem ser desenvolvidos com o pensamento de arquitetura desenvolvida para falhas. Dessa forma, Fowler (2014) e Kurhiken (2014) afirmam que o ideal é que cada serviço também possua um banco de dados isolado, no qual a estratégia possível é manter informações de outros serviços em forma de cache. A Figura 7 demonstra a possível forma de divisão das informações em bancos de dados distintos.

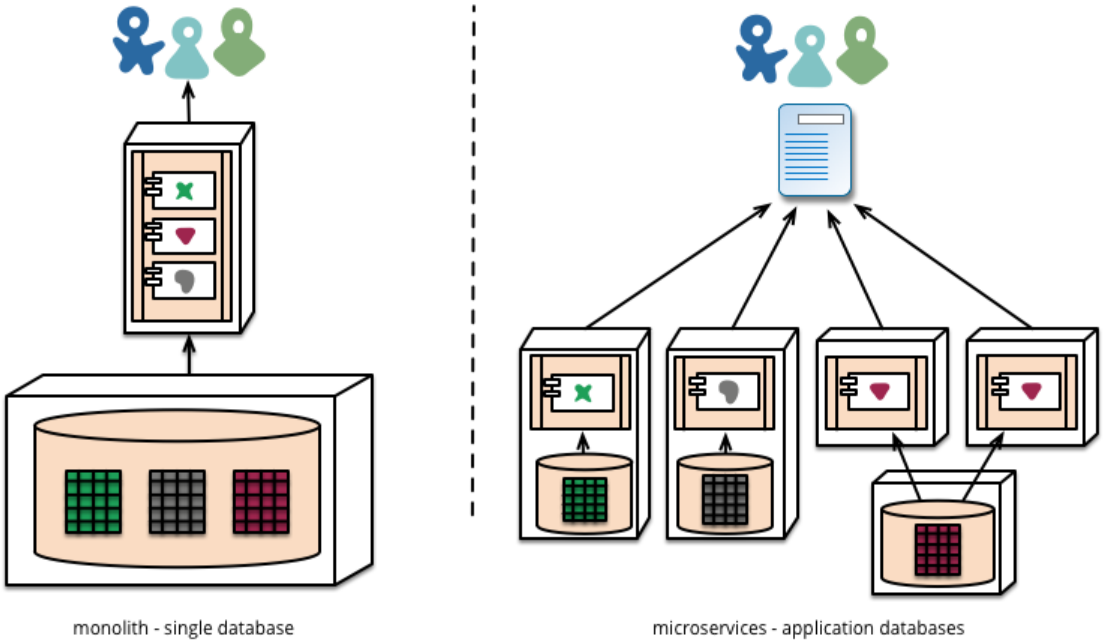

Figura 7 - Bancos de dados distribuídos e centralizados. Fonte: Fowler (2014)

Mesmo com a replicação dos dados em forma de cache, ainda é possível vivenciar algumas contravenções. Segundo Newman (2015), sistemas distribuídos tendem a se encaixarem no chamado teorema de CAP que implica em algumas restrições.

O teorema CAP, desenvolvido por Eric Brewer em 2000, e comprovado por Seth Gilbert e Nancy Lynch em 2002 (HEWITT, 2010), apresenta três propriedades presentes em sistemas: disponibilidade, tolerância à particionamento de rede e consistência. Dentre as três, somente duas podem ser escolhidas e estabelecidas como preferenciais, como apresentado na Figura 8. (HEWITT, 2010; NEWMAN, 2015)

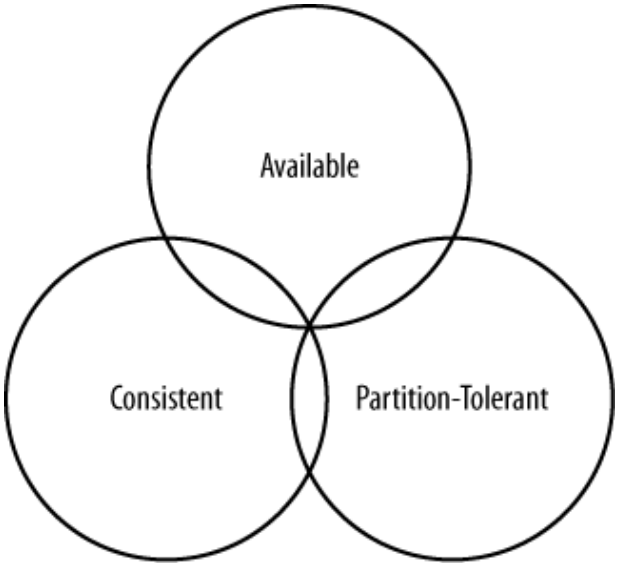

Figura 8 - As 3 vertentes do Teorema CAP. Fonte: Hewitt (2010)

Tratando-se de sistemas distribuídos, o particionamento de rede é indispensável. Como a própria nomenclatura enuncia, os sistemas distribuídos têm a características de não serem centralizados, logo, necessitando de comunicação voltada ao seu particionamento, então a rede passa a ser o principal meio de comunicação (NEWMAN, 2015).

Dessa forma, na maioria dos casos, restarão somente duas propriedades para serem focadas. A vertente da disponibilidade prevê a capacidade de que qualquer parte do sistema esteja operacional, ou seja, independente das outras partes para executar seu propósito. Em um cenário dos microserviços, fica garantido que um determinado serviço pode se utilizar de informações de consistência temporária, como as armazenadas em cache, para conseguir entregar resultados e continuar operante. Tal escolha traz consigo o sacrifício da consistência da informação, já que ela pode pertencer à um contexto onde somente um microserviço, neste caso incapaz de fornecê-la, tem acesso (HEWITT, 2010; NEWMAN, 2015).

Em ambientes onde informações sensíveis à volubilidade são tratadas, o enfoque deve ser mudado. Neste caso, deve ser tratado como prioridade a consistência, que permite que um ponto único seja capaz de garantir a veracidade de um determinado dado a todos os seus consumidores. Ainda deve ser reiterado que tais informações não são suscetíveis à cache, tendo em vista que a replicação do dado geraria a necessidade de sincronia que pode não acontecer caso o serviço retentor da informação não esteja indisponível (HEWITT, 2010; NEWMAN, 2015).

Da mesma forma, ao se optar pela consistência, a disponibilidade se torna o problema. Onde um serviço depende de outro para produzir um resultado e o segundo serviço se encontra indisponível, não existe uma consistência absoluta da informação, a qual só pode ser provida pelo microserviço indisponível. Assim o primeiro microserviço se encontrará incapacitado de produzir um resultado. (HEWITT, 2010; NEWMAN, 2015).

Conclui-se assim que o tipo de informação a ser tratada será o principal influenciador das decisões durante a abordagem da arquitetura do sistema. Em casos de informações sensíveis, como transações bancárias, fica mais evidente a prioridade retida ao particionamento de rede e à consistência. Concomitantemente, em casos onde se é preferível que exista um resultado não necessariamente consistente, como uma lista de produtos à venda, o foco deve ser mudado para disponibilidade.

O particionamento de rede se faz presente por conta dos microserviços serem, em sua essência, sistemas distribuídos. Esse fator mostra a necessidade da prioridade que o elemento de comunicação possui dentro do universo de microserviços.

Para que haja um resultado satisfatório, a comunicação é um mecanismo essencial. Segundo Namiot e Sneps-Sneppe (2014), o ideal é que tais serviços se utilizem de mecanismos simples para conversação entre processos, sejam estes voltados para requisições síncronas ou assíncronas. Ainda, com o intuito de se construir um sistema resiliente, Richardson (2017) enfatiza a necessidade de padrões de comunicação diversos para cada elemento dentro do ecossistema como a comunicação entre cada microserviço, a comunicação entre clientes externos e os serviços ou como cada serviço é descoberto, assim como exposto na Figura 9.

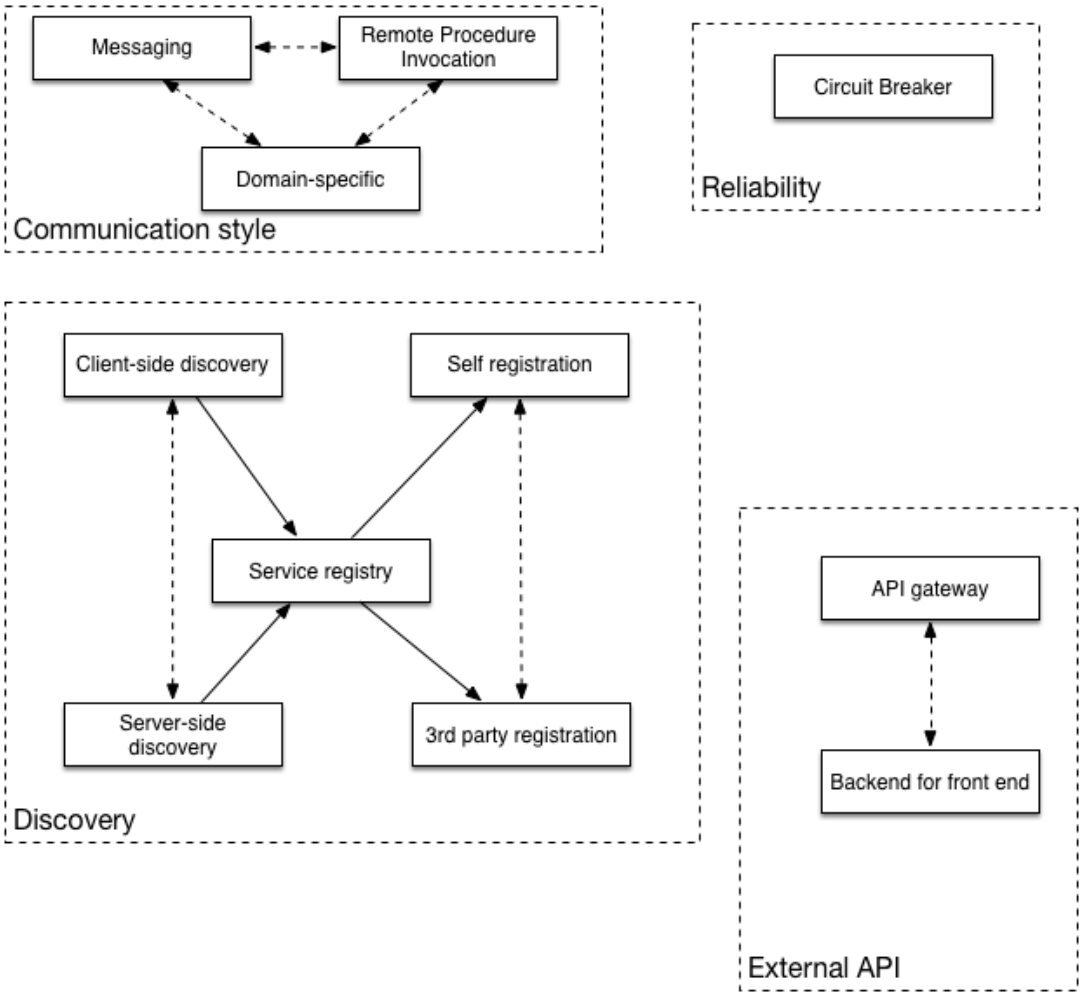

Figura 9 - Possíveis padrões de comunicação para microserviços. Fonte: Newman (2017)

Assim, notam-se algumas diferenças básicas entre os modelos de comunicação da arquitetura monolítica e os microserviços. Enquanto uma aplicação monolítica pode receber requisições diretamente de um cliente, como um navegador web, para que possa processar, executar a lógica de domínio e devolver um resultado, os microserviços poderão ter um elemento centralizador das requisições (NEWMAN, 2015; RICHARDSON, 2014).

A chamada API *Gateway* é responsável por receber as requisições e delegar para cada serviço mapeado. Além disso, alguns benefícios, segundo Daya *et al* (2016) e Newman (2015), que esse padrão oferece são:

- Capacidade de *logging* de todas as transações, por se tratar de um ponto centralizador.
- Controle da granularidade das informações disponíveis dependendo de cada cliente consumidor.
- Agregação de chamadas a serem feitas para os serviços presentes no *backend.*

A Figura 10 demonstra uma visão abrangente de como a comunicação pode ser estabelecida.

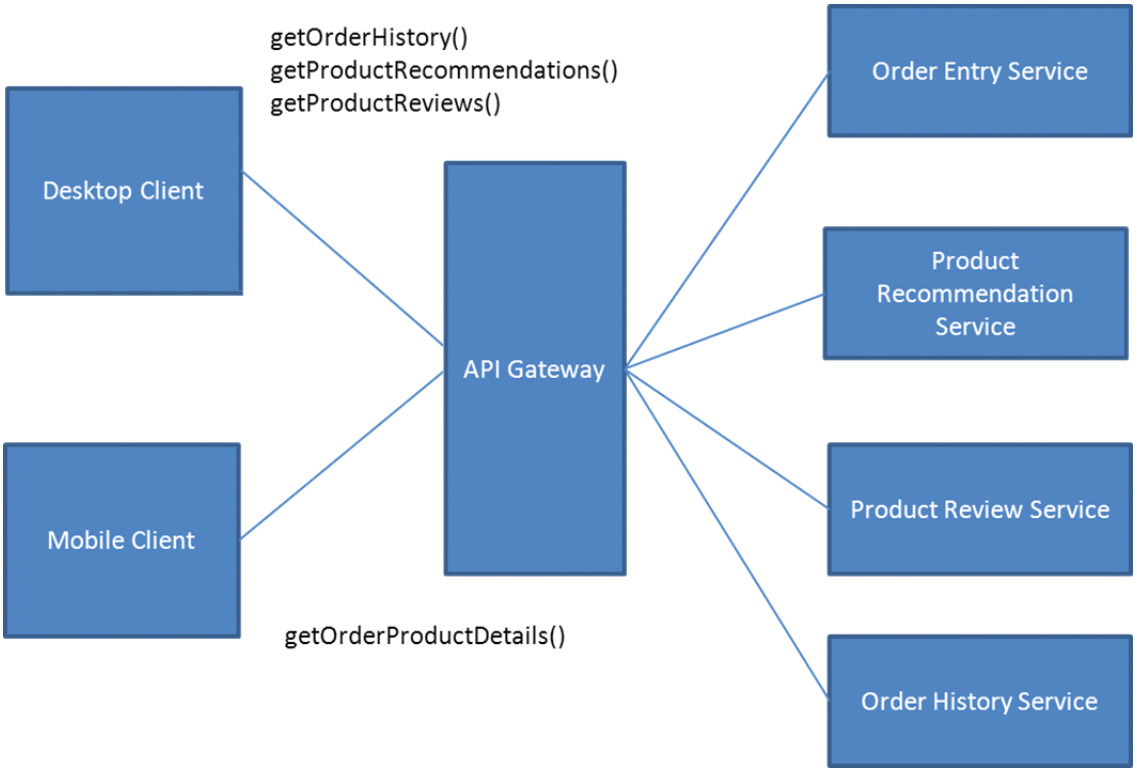

Figura 10 - API Gateway. Fonte: Daya *et al* (2016)

Mesmo que uma API *Gateway* consiga tratar as requisições provenientes de um cliente como um dispositivo móvel ou um navegador de internet, a comunicação entre um serviço e outro depende de outros padrões. De acordo com Newman (2015) e Richardson (2017), a escolha pela comunicação síncrona ou assíncrona nos guia na implementação do projeto.

Ao se decidir por comunicação síncrona, temos um padrão mais simples de ser implementado e mais familiar aos programadores. Esta pode ser implementada utilizando de padrões como REST ou *websockets*. Com essa abordagem temos mais facilidade em obter uma resposta da conclusão do procedimento solicitado por conta da característica bloqueante presente, iniciando uma requisição que segura o processo até que seja obtida uma resposta, estabelecendo o modelo *request/response* (DAYA, 2016; NEWMAN, 2015).

Mesmo assim, a comunicação síncrona deixa a desejar em alguns casos. A latência pode se tornar um problema quando existem processos demorados para serem executados. Neste cenário, bloqueando a execução do serviço, a requisição o

deixa incapacitado de processar novas solicitações até que seja concluída. Além disso, a conexão entre o cliente e o servidor deverá permanecer aberta, podendo gerar um *overhead* na rede (NEWMAN, 2015).

Já a comunicação assíncrona possibilita uma resolução para tal impasse. Em termos programáticos, o cliente realizará uma requisição ao servidor criando uma função de *callback* que será executada quando o servidor produzir um resultado e devolver a resposta. Assim, é possível utilizar o modelo *request/response*, mas sem a necessidade de espera (DAYA, 2016).

Baseado na assincronia, existe também o modelo de comunicação chamado *event-based* (baseado em eventos). Os serviços implementados nesse padrão, durante sua execução, geram eventos contendo informações capazes de descrever o que foi feito e logo em seguida o dispara, esperando que outros serviços que estejam atrelados a tal evento, sejam capazes de saber o que fazer a partir da informação recebida. Isso significa que um evento não precisa conhecer o outro para que procedimentos sejam encadeados (DAYA, 2016; NEWMAN 2015).

Desta forma, nota-se um ganho significante em desacoplamento e escalabilidade. Pode-se observar que os serviços que se encaixam nesse padrão precisam somente conhecer os eventos que lhes interessam, gerando uma ação baseada nas informações obtidas, sendo irrelevante quem as gerou. Também se nota que novos serviços podem ser diretamente anexados aos eventos já existentes. Isso é prático pois não é necessário nenhum tipo de alteração em serviços já em execução ou nos próprios eventos. Logo, o serviço novo é autossuficiente e capaz de executar suas ações sem que outros serviços tenham conhecimento.

Outro ganho que deve ser notado é quanto à sincronia das informações. Por conta da capacidade de gerar eventos, quando uma alteração é feita em um banco de dados de um determinado serviço, um gatilho pode ser disparado, invalidando dados antigos e comunicando aos outros serviços sobre uma atualização naquela informação.

Para implementação da topologia *event-based* é necessário um elemento centralizador responsável por coletar, enfileirar e distribuir cada evento gerado. O *message broker* assume esse papel permitindo que sejam criados tópicos onde determinados serviços passam a publicar os eventos. Outros serviços podem ser anexados a tais tópicos recebendo uma notificação do *message broker* sempre que seus tópicos assinados forem alimentados. É importante notar aqui que o serviço a

ser notificado não precisa estar acessível no momento em que as publicações são enviadas. Uma característica dos *message brokers* é a capacidade de armazenar e redistribuir os eventos assim que seus serviços inscritos estejam disponíveis, mantendo salvo um histórico de eventos a serem tratados, assim como é demonstrado na Figura 11 (DAYA, 2016).

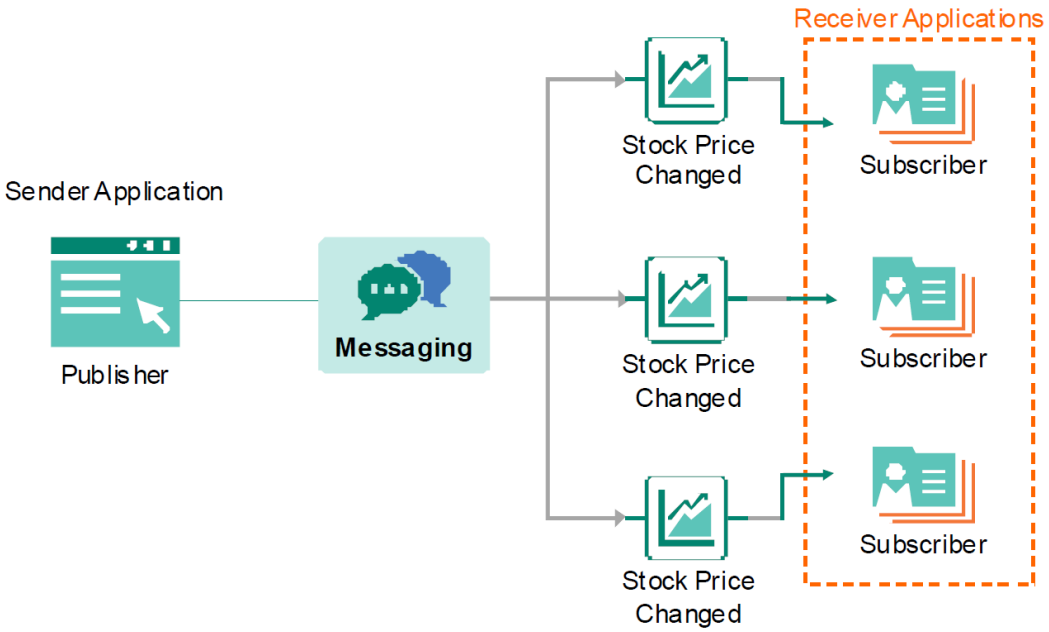

Figura 11 - Message Broker. Fonte: Daya et al (2016)

4. ESTUDO DE CASO

Durante o desenvolvimento do trabalho foi criado um estudo de caso com o objetivo de apresentar as principais características da arquitetura de microserviços. Trata-se de um mecanismo de blog onde *posts* com conteúdo de texto e imagem são criados e publicados. A Figura 12 exibe a interface web desenvolvida para o blog.

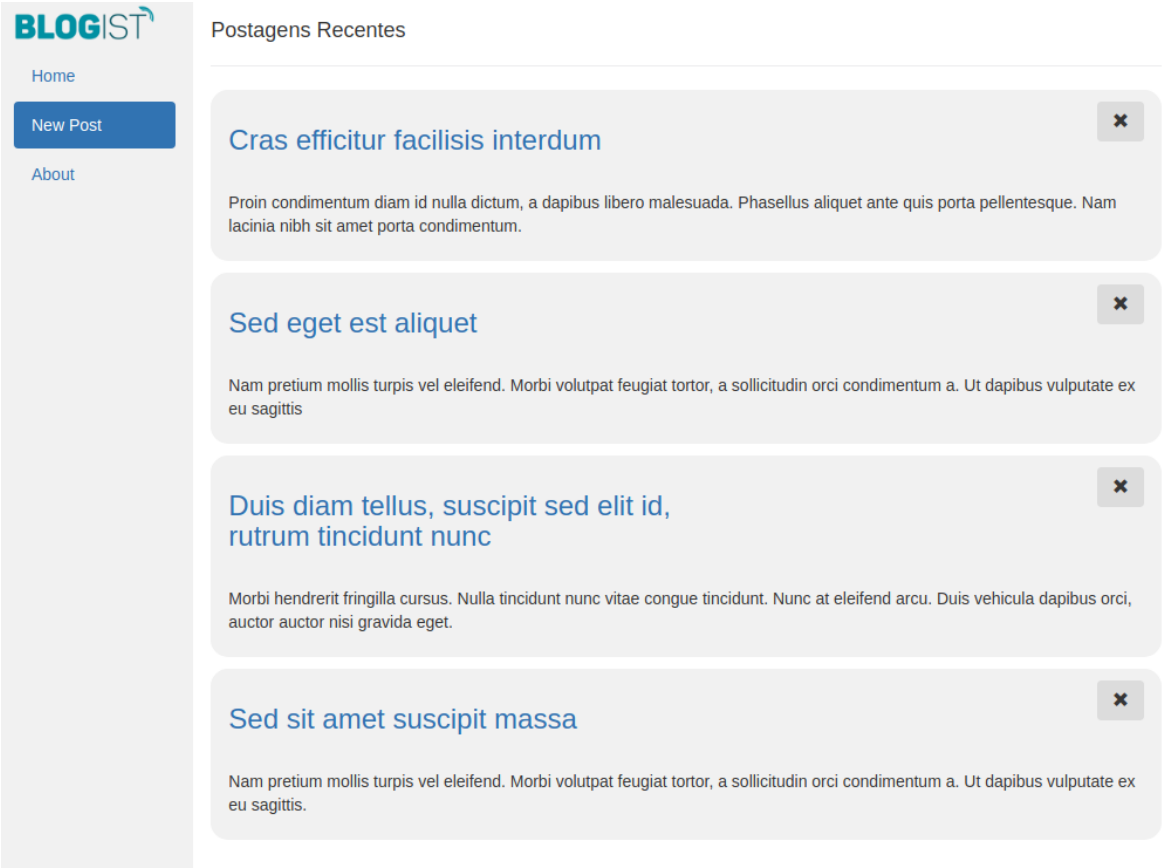

Figura 12 - Página Inicial do Blog desenvolvido

O sistema foi desenvolvido em três microserviços, cada um especializado em uma responsabilidade. Também foram utilizadas tecnologias diferentes no desenvolvimento de cada uma e é importante ressaltar que nenhuma foi escolhida propositalmente para suas determinadas funcionalidades.

A seguir, são apresentados os fluxos e processos que foram utilizados no desenvolvimento e organização das aplicações assim como suas respectivas tecnologias.

4.1. Infraestrutura

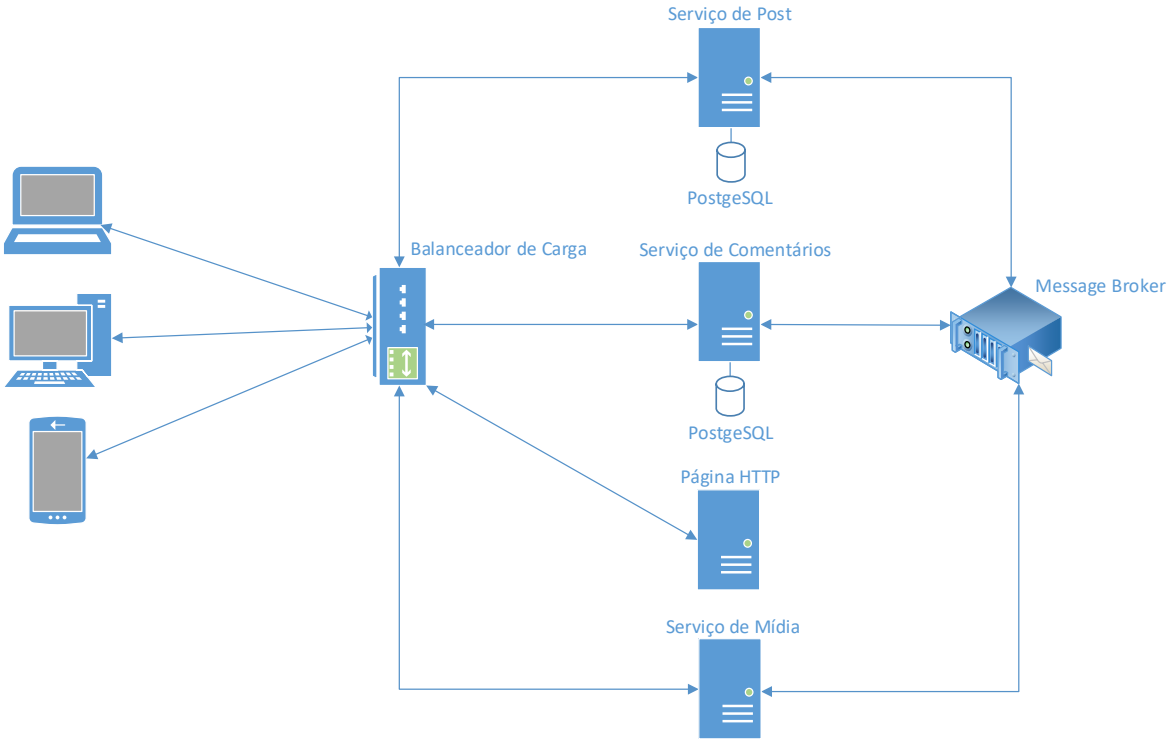

Figura 13 - Diagrama da Infraestrutura

Assim como demonstrado na Figura 13, todos os serviços desenvolvidos estão atrelados a uma infraestrutura que complementa o ambiente do sistema. Essa parte refere-se às tecnologias e recursos utilizados para comunicação entre serviços, armazenamento de dados e a comunicação entre os serviços e a interface do usuário. É de importância levar em consideração que essa infraestrutura foi desenvolvida voltada para alguns princípios dos microserviços, como o baixo acoplamento e a implantação independente.

O primeiro elemento relevante da infraestrutura é o banco de dados. Foi utilizado o PostgreSQL 9.4 para os serviços de *post* e comentários, cada um presente em uma instância diferente, sem um relacionamento direto entre si. O PostgreSQL é de código aberto e toda a documentação e especificidades da ferramenta são disponibilizados pelo PostgresSQL *Global Development Group* (POTGRESQL GLOBAL DEVELOPMENT GROUP, 2014).

O segundo elemento é o *message broker*, utilizado para a comunicação entre um serviço e outro. Para essa tarefa foi utilizado o Apache Kafka 0.9. Segundo sua documentação disponibilizada pela *Apache Software Foundation*, sua principal funcionalidade é a distribuição de fluxo de dados em tempo real (APACHE

SOFTWARE FOUNDATION, 2016). A comunicação, captura e publicação de informações, é feita através de bibliotecas desenvolvidas em cada linguagem. Como padrão de comunicação, foi configurado somente um tópico chamado “*delete-post*” onde são publicadas notificações referentes à exclusão do *post*.

Juntamente com a notificação é enviado uma cadeia de caracteres formando a representação do *post* convertida para o formato *JSON* (*JavaScript Object Notation* ou Notação de Objeto JavaScript). Essa informação é capturada pelos serviços inscritos nesse mesmo tópico, os serviços de comentários e mídia. Os dois reagem excluindo as informações referentes àquele *post* que foi apagado.

Por fim, o último elemento trata da exposição desses serviços para serem consumidos externamente. Nesse ponto se encaixa o balanceador de carga onde através dele passam todas as requisições para os serviços, sendo elas provenientes de qualquer consumidor externo.

No balanceador de carga são configurados os endereços dos serviços e as rotas para suas localizações. Vale lembrar que mais de uma instância de cada serviço pode ser configurada para uma rota, sendo possível realizar uma escala horizontal de maneira prática.

Para assumir esse papel na infraestrutura foi escolhido o NGINX. Suas funcionalidades incluem servidor HTTP, proxy reverso, servidor de e-mail, e proxy TCP/UDP genérico. Todas as suas funcionalidades e documentação são mantidas pela NGINX Inc (NGINX INC, 2017). A Figura 14 exibe o arquivo de configuração utilizado para o mapeamento das rotas.

```
server{
        listen 80;
        server_name localhost;
        location / {
                proxy_pass http://home;
        }


        location /blogist/api/media {
                proxy_pass http://media;

        }

        location /blogist/api/posts {
                proxy_pass http://posts;

        }

        location /blogist/api/comments {
                proxy_pass http://comments;
        }

}

upstream media {
        server 127.0.0.1:1000;
}

upstream comments {
        server 127.0.0.1:3000;
}

upstream posts {
        server 127.0.0.1:4000;
}


upstream home {
        server 127.0.0.1:8080;
}
```

Figura 14 – Arquivo de configuração do NGINX

4.2. Interface do Usuário

Como ponto de acesso ao usuário, foi desenvolvida uma interface web. Assim, essa interface pode ser acessada por qualquer navegador de internet. Através dela são disparadas ações do usuário que são redirecionadas para cada um dos respectivos serviços.

Além de disparar as ações, ela tem por responsabilidade a exibição dos dados. Isso exige a agregação das informações que acontece através de requisições direcionadas aos serviços. Assim, a semântica das informações existe somente neste ponto.

Para o desenvolvimento da interface foram usadas tecnologias específicas para criação de páginas com uma aparência dinâmica. As escolhas foram: Javascript, HTML e Angular JS.

Javascript é uma linguagem de programação interpretada criada para ser utilizada como elemento auxiliar no desenvolvimento de páginas web dinâmicas. Atualmente encontra-se difundida e considerada a linguagem mais ubíqua da história, sendo que praticamente todos os dispositivos como tablets, smartphones e computadores possuem um interpretador da linguagem (FLANAGAN *et al*, 2011). Mesmo sendo criada com o intuito de servir como auxiliadora no desenvolvimento de páginas web, sua utilização transpassou essa limitação hoje sendo utilizada amplamente em aplicativos *server-side*, como por exemplo na plataforma NodeJs que será citada adiante.

O HTML, também chamado *hypertext markup language* ou linguagem de marcação de hipertexto, é uma linguagem de marcação voltada para criação de páginas web que são interpretadas e exibidas pelos navegadores web (RAGGETT *et al*, 1998).

Por fim, o Angular JS é um *framework* voltado para o desenvolvimento do *front-end* de aplicações web. Dentre suas funcionalidades está a manipulação dos elementos HTML facilitando iterações para a exibição de informações. Além disso através dele são feitas requisições *AJAX* (*Assynchronous Javascript and XML* ou Javascript e XML Assíncronos), o que permite o carregamento e submissão parciais de dados aos serviços. Sua documentação oficial e sua evolução é feita pela empresa Google (GOOGLE, 2015).

4.3. Serviço de Posts

Esse serviço é responsável pelas operações voltadas aos *posts*. Suas funções incluem o acesso ao banco de dados para executar a persistência, consulta, alteração e exclusão dos dados. Nele estão presentes três classes como mostrado na Figura 15.

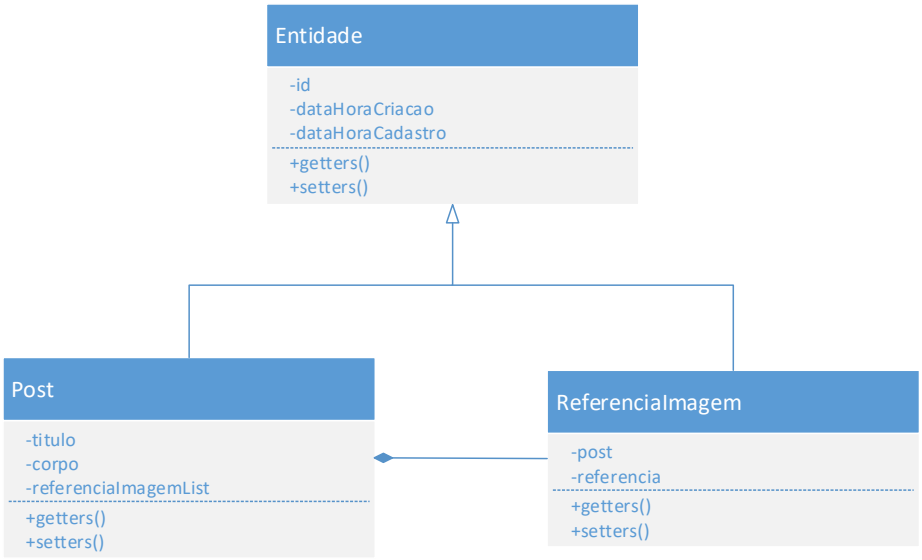

Figura 15 - Diagrama de classes do serviço de *Posts*

Seu processo depende inicialmente da iteração do usuário com a tela. Ao abrir a tela inicial é encaminhada uma requisição até o serviço de *posts* que por sua vez realiza uma consulta no banco de dados buscando todos os *posts* e logo em seguida o resultado é devolvido à tela para ser exibido assim como mostra o fluxo no diagrama da Figura 16.

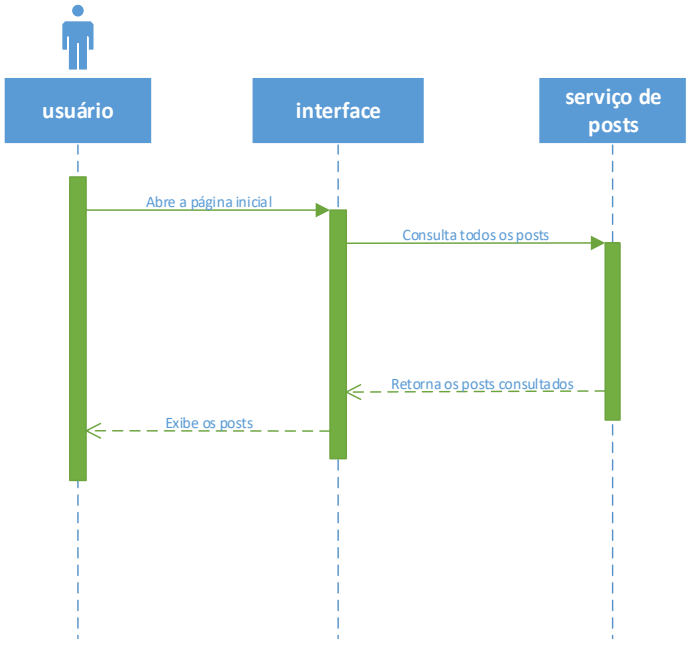

Figura 16 - Abertura da Página Inicial

Um outro exemplo de fluxo é a criação de um novo *post*. O usuário deve clicar no link de novo *post* para que a interface exiba a tela de criação, como

demonstrado na Figura 17. Logo em seguida, o usuário deve preencher as informações que serão publicadas e ao enviá-las é feita uma requisição ao serviço de *posts* e logo após o redirecionamento para a tela inicial exibido no fluxo da Figura 18.

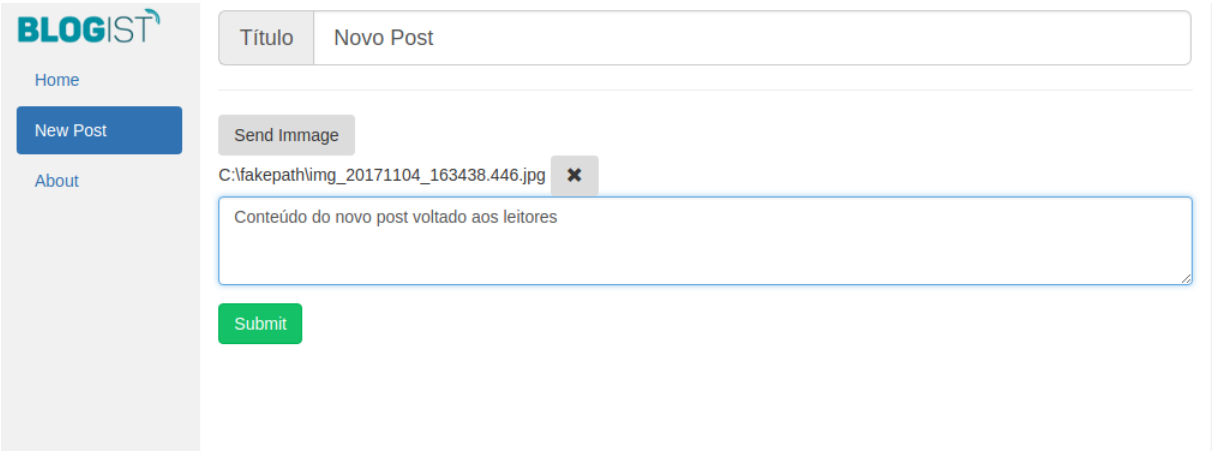

Figura 17 - Interface para cadastro de *posts*

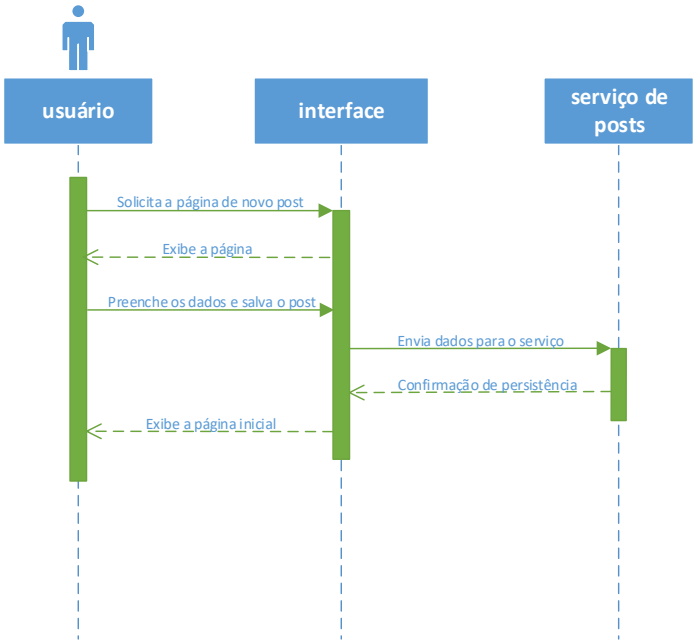

Figura 18 - Persistência de um novo *post*

As principais tecnologias utilizadas no desenvolvimento dessa etapa foram Java, Hibernate JPA e o Jersey.

Java é uma Linguagem de programação desenvolvida desde 1991 pelo *Green Team*, pequeno grupo de engenheiros da empresa Sun e atualmente a linguagem é mantida pela empresa Oracle. A linguagem Java foi desenvolvida inicialmente oferecendo suporte à programação orientada a objetos, mas a partir de

sua versão 1.8, oferece suporte à programação funcional. Vale também ressaltar que seu código é compilado e executado na *Java Virtual Machine* (Máquina Virtual Java), segundo sua documentação oficial que é mantida pela Oracle Corporation (ORACLE CORPORATION, 2015).

O Hibernate JPA se caracteriza como um *framework* ORM (Object/Relational Mapping ou Mapeamento Objeto/Relacional) escrito na linguagem de programação Java. Esse tipo de ferramenta visa facilitar os processos de banco de dados como consultas, persistência e criação de tabelas. A versão utilizada no desenvolvimento do projeto foi a 4.3 que possui sua documentação e especificações disponibilizados pela empresa Red Hat (RED HAT, 2015).

Por fim o recurso utilizado para criação dos pontos de conexão desse microserviço foi o Jersey. Essa ferramenta existe como proposta para simplificar o desenvolvimento de *webservices RESTful* encapsulando os detalhes de baixo nível da comunicação HTTP. Dentro do projeto foi utilizada a versão 2.8 mantida e disponibilizada pela Oracle Corporation (ORACLE CORPORATION, 2014).

4.4. Comentários

Este serviço expõe os recursos necessários para o tratamento da entidade de comentários. Nesses recursos estão inclusas as operações de banco de dados assim como no serviço de *posts*. A entidade de comentários ainda mantém um campo específico que mantém uma referência ao identificador único do *post*, que está localizado em outro banco de dados e é tratado por outro serviço. A Figura 19 exibe o diagrama de classe que modela a entidade desse microserviço.

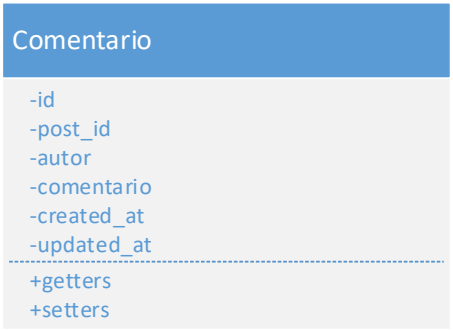

Figura 19 - Diagrama de Classes do Serviço de Comentários

Assim com o serviço de *posts*, o serviço de comentários tem suas ações também disparadas a partir da tela. A principal diferença é que ele também responde

às mensagens recebidas do *message broker*, para realizar a exclusão de comentários que tenham o seu *post* excluído.

A Figura 20 exibe o fluxo da comunicação realizada com o microserviço de comentários ao se abrir um *post* já existente. Primeiramente o usuário, através da interface, abre um *post*. Logo então, essa ação faz com que a interface requisite todos os comentários existentes para aquele determinado *post* para serem exibidos pela interface.

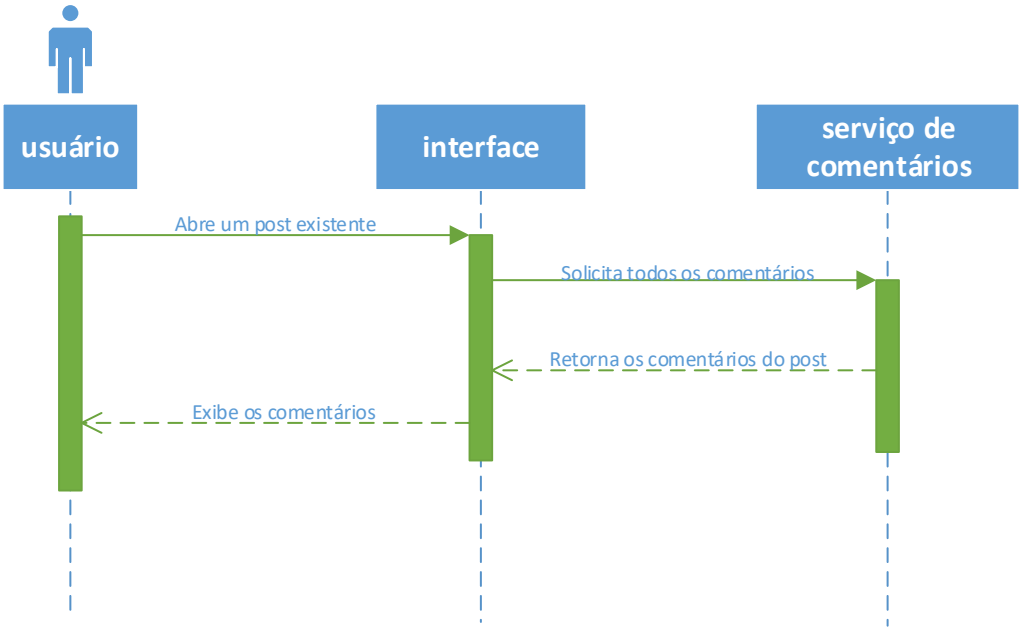

Figura 20 - Visualização de um *Post* Existente

É importante ressaltar que nesse caso, existe a participação do *message broker*. Para a exclusão do *post*, o usuário inicia ação na interface que por sua vez faz uma requisição ao recurso disponibilizado no microserviço de *posts.* Através de uma chamada interna o *post* é excluído e em seguida é enviada uma mensagem com os respectivos dados ao *message broker*. Essa mensagem contém informações como o identificador único do post e seu conteúdo para que seja possível tomar decisões baseadas na ação de exclusão. Logo em seguida o *message broker* propaga mensagem aos serviços inscritos no tópico "*delete-post*".

Um destes inscritos é o serviço de comentários. Com os dados do *post* recebidos, todos os comentários pertencentes a ele são excluídos através de uma chamada interna. Tal fluxo é ilustrado na Figura 21.

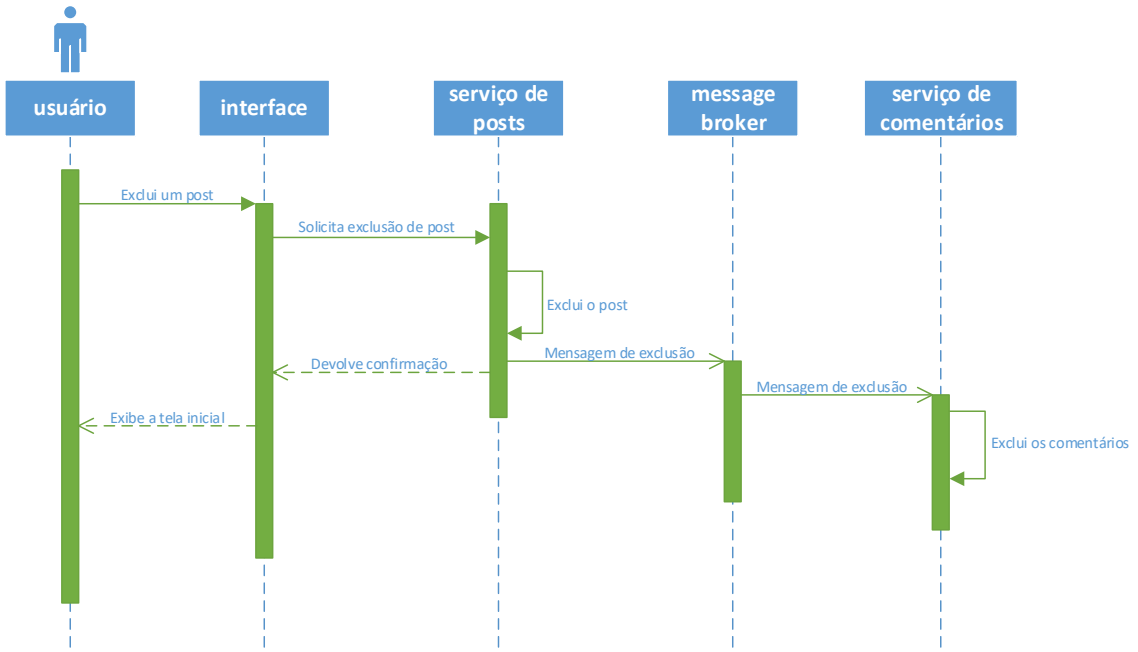

Figura 21 - Exclusão dos comentários por exclusão de *posts*

Dentro desse cenário existe uma pendência quanto às possiblidades de falha. Podemos notar que a ação de exclusão do *post* pode ter sido efetuada, mas o serviço pode não conseguir repassar os dados para o *message broker*. Tendo em vista o design para falhas, o microserviço de post possui a capacidade reenviar essas informações caso aconteça algum erro, por exemplo, um erro de comunicação. Isso garante novas tentativas de envio e a segurança extra de entrega ao *message broker*.

Ainda assim, pode ocorrer um erro de processamento dentro dos outros serviços que capturaram tal evento. No caso do serviço de comentários, uma rotina foi implementada para também executar tentativas consecutivas da exclusão dos comentários no caso de falha. É de relevância ressaltar que tais rotinas foram implementadas tendo em vista que alguns erros imprevistos podem não estar tratados e não serem resolvidos por notas tentativas de execução, logo, para não criar tentativas infinitas, existe um limite de execuções.

Em última instância, caso o limite de tentativas tenha sido excedido e ainda assim a execução não tenha obtido sucesso, os *logs* assumem o papel de evidência. A partir das informações obtidas pelos *logs*, ações podem ser tomadas de forma manual, e a partir dos novos erros encontrados, gera-se um novo tratamento de código, assim evoluindo o serviço para novos níveis de maturidade.

A composição desse serviço foi feita utilizando a linguagem de programação Ruby e o *framework* Rails. Ruby tem em suas principais propostas o

desenvolvimento ágil, dinâmico e simplificado. A versão utilizada foi a 2.4. Dentro da linguagem encontram-se os paradigmas de orientação a objeto, programação funcional e programação imperativa. A documentação e manutenção da linguagem são feitas pela Ruby Organization (RUBY ORGANIZATION, 2017).

O Rails é um *framework* desenvolvido na linguagem Ruby que engloba funcionalidades voltadas para diversas finalidades. Dentre elas estão a comunicação com banco de dados, disponibilização de *webservices* e componentes prontos para páginas web. A versão utilizada foi a 5.1 e os detalhes de sua implementação e seus recursos são encontrados na sua documentação oficial que é disponibilizada pela Rails Comunity (RAILS COMUNITY, 2017).

4.5. Mídia

Este serviço tem por responsabilidade a controle das imagens/figuras que fazem parte de cada *post*. Através dele, as imagens são enviadas, disponibilizadas ou excluídas sob demanda.

Seu fluxo é feito através da interface de criação de *posts.* O usuário abre a tela de seleção de imagens e ao confirmar a seleção da imagem, a interface envia uma requisição até o microserviço de mídia para que seja persistida. Por sua vez, o serviço de mídia devolve, como confirmação, o nome criado para aquela imagem e que deverá ser mantido como referência para o acesso desse item. Esse fluxo é demonstrado na Figura 22.

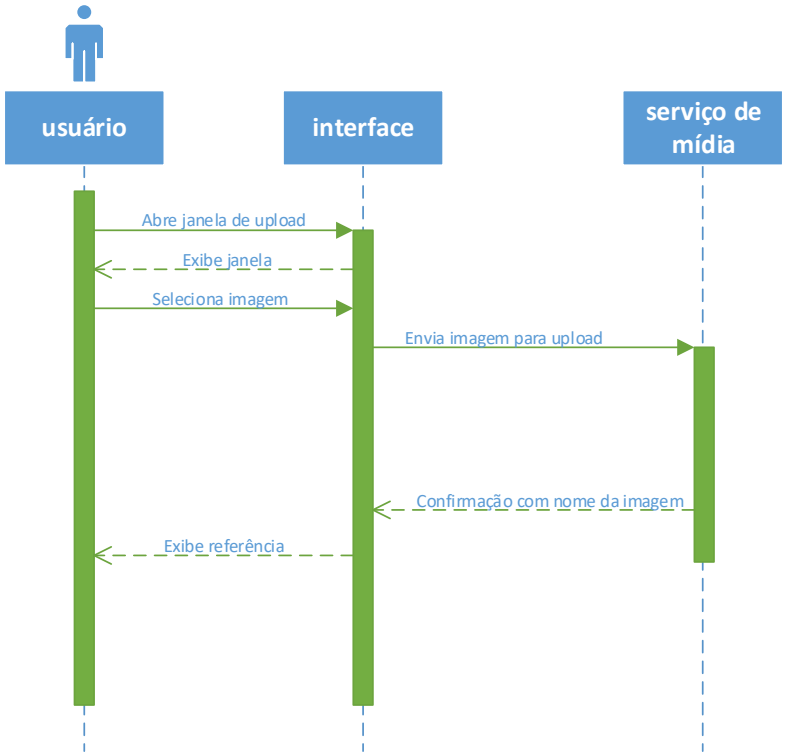

Figura 22 - Upload de Imagem

Seu desenvolvimento foi feito com o NodeJS. Esse *framework,* desde seu início, propõe a criação de aplicações utilizando a linguagem de programação JavaScript, tendo seus códigos executados não no navegador, mas diretamente no servidor. Sua arquitetura é baseada em eventos com um modelo de I/O não bloqueante. A versão utilizada foi a 6.11. Seu desenvolvimento e documentação são mantidos pela NodeJS Foundation (NODEJS FOUNDATION, 2017).

5. ANÁLISE

Tendo tecnologias direcionadas em cada serviço, é possível um tratamento mais fino, voltado a resolver um tipo específico de problema. Não somente isso, mas também podem ser isolados para ambientes de execução específicos como servidores com mais capacidade de processamento ou um melhor desempenho do disco de armazenamento.

Em contrapartida, essa separação necessita de uma análise de domínio mais minuciosa. Um dos desafios durante o desenvolvimento do projeto foi a identificação dos pontos onde haveria o relacionamento entre cada serviço e quais tipos de referências seriam consideradas para a criação desses relacionamentos.

Esse fator foi encarado, por exemplo, no desenvolvimento do serviço de *post*. Uma das dependências desse serviço é o serviço de mídia, responsável pelo upload das imagens. Por haver essa dependência o serviço de *post* foi criado possuindo uma entidade específica responsável pela referência dos identificadores únicos de cada imagem. Os identificadores são transferidos como lista até a interface do usuário juntamente com a entidade de *post* e então passados como parâmetros para o serviço de mídia para se obter acesso ao conteúdo.

Seguindo o princípio de interfaces de comunicação modeladas em volta do negócio, a criação de um modelo de negócio teria auxiliado durante a execução dessa tarefa. O modelo exibe mais facilmente os possíveis pontos de separação ou junção de cada serviço dividindo responsabilidades e gerando maior visibilidade do projeto.

Ainda dentro do contexto de junção dos serviços, há o relacionamento entre os *posts* e os comentários. As duas entidades são armazenadas em bancos de dados distintos. Logo, não existe a presença de referências diretas ou restrições de chave estrangeira pela base de dados. Havendo a dependência do serviço de *posts* pelo serviço de comentários, os identificadores únicos de cada *post* são referenciados na tabela do banco de dados dos comentários. Os comentários são consultados somente ao se informar o identificador único de seu *post*, através da sua requisição ao *webservice*.

Por conta da disponibilização de tais *webservices*, podemos notar aqui uma certa redundância de funcionalidades. Ao compor serviços em linguagens diferentes, o código desenvolvido não é reaproveitado, logo nota-se a necessidade da criação de códigos com funcionalidades semelhantes ou iguais como, por exemplo, para conexão

com o banco de dados, exibidos nas Figuras 23 e 24, ou a criação dos *webservices* de cada serviço, mostrados nas Figuras 25 e 26.

```xml
<!--suppress JpaConfigDomFacetInspection -->
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="blogist_postservice">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>br.com.kbral.postservice.model.Post</class>
        <class>br.com.kbral.postservice.model.ReferenciaImagem</class>

        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/blogist_postservice"/>
            <property name="javax.persistence.jdbc.user" value="blogist_postservice"/>
            <property name="javax.persistence.jdbc.password" value="asdf321"/>
            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>

            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQL9Dialect"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="format_sql" value="true"/>
        </properties>

    </persistence-unit>
</persistence>
```

Figura 23 - Propriedades para conexão com o banco de dados pelo serviço de *posts*

```yaml
default: &default
  adapter: postgresql
  encoding: unicode
  user: blogist_commentservice
  password: asdf321
  host: localhost
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>

```

Figura 24 - Propriedades para conexão com o banco de dados pelo serviço de comentários

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});

app.post('/blogist/api/media', (req, res) => {
  console.log('Recebendo nova imagem: '  + req.files);
        let retorno;

        if (req.files){
                retorno = 'Mandou Arquivos';

                let imagem = req.files.imagem;

                retorno = salvarArquivo(imagem,res);
                res.send(retorno);
        } else {
                res.status(200).send('Não mandou Arquivos');
        }
});

app.get('/blogist/api/media/:nomeArquivo', (req, res) => {
  console.log('Recebendo requisição para o arquivo: ' + req.params.nomeArquivo);
        res.sendFile(__dirname + '/upload/' + req.params.nomeArquivo);
});

app.delete('/blogist/api/media/:nomeArquivo', (req, res) => {
  console.log('Recebendo requisição para o arquivo: ' + req.params.nomeArquivo);
        deleteImage(req.params.nomeArquivo,function (err) {
                if(err){
                        res.status(505).send(err);
                        return;
                }
        });
        res.status(200).send("Imagem " + req.params.nomeArquivo + " removida com sucesso");
```

Figura 25 - Mapeamento do webservice de mídia

```
Rails.application.routes.draw do
  # For details on the DSL available within this file, see http://guides.rubyonrails.org/routing.html
        namespace :blogist do
                namespace :api do
                        defaults format: :json do
                                resources :comments, only: [:create, :update, :destroy]

                                get 'comments/count/:post_id', to: 'comments#count_by_post_id'
                                get 'comments/:post_id', to: 'comments#show'
                        end
                end
        end
end
```

Figura 26 - Mapeamento do webservice de comentários

Após a implementação e estabilização do ambiente, a infraestrutura se mostrou como parte mais complexa da criação de microserviços. Nela foi gasto a

maior parte do tempo de desenvolvimento do projeto. O principal fator é o de ser necessário uma padronização abstraída o suficiente para que todos os serviços consigam se aderir. É através da infraestrutura que os padrões de comunicação serviço/serviço e serviço/*front-end* foram estabelecidos e quais as interfaces de comunicação a serem usadas.

Um dos principais agentes da versatilidade apresentada foi o *message broker*. Por conta do isolamento que é provido a cada um dos serviços, novos componentes podem ser vinculados ao ambiente sem que os outros serviços já existentes sejam necessariamente alterados.

Juntamente com o *message broker*, o balanceador de carga assume um papel de importância. Este, sendo responsável pelo agrupamento dos recursos, provém transparência para qualquer agente externo, que esteja disposto a consumir os recursos do sistema.

Ainda por conta do isolamento, nota-se a independência almejada nos princípios dos microserviços. Os serviços desenvolvidos podem evoluir tendo a mínima necessidade de obedecer ao contrato das interfaces de comunicação, ou seja, o formato em que suas entidades são trafegadas, já que serão consumidas por outros serviços ou terceiros.

Essa evolução ainda conta com uma escalabilidade simples. Por se tratarem de serviços relativamente pequenos, a escalabilidade tanto vertical, referente à capacidade processamento de uma máquina individual, quanto horizontal, referente à quantidade de máquinas executando o mesmo serviço, podem ser direcionadas atingindo cada serviço separadamente. Isso remete à uma economia de recursos computacionais, já que não é necessária uma máquina com diversas especificidades de *hardware* simultâneas, ser alocada e disponibilizada para todas as funcionalidades do sistema. Sendo assim, a otimização dos ambientes pode ser totalmente direcionada a cada contexto de microserviço separadamente.

6. CONCLUSÃO E TRABALHOS FUTUROS

O aumento de usuários de dispositivos computacionais e as novas exigências em questão de performance e tempo de resposta quanto ao mercado levam cada vez mais à evolução de tecnologias e estratégias para sistemas. Sendo assim, essa evolução é natural e necessária, melhorando cada vez mais como tais recursos podem ser aplicados de maneira dinâmica e eficiente.

Os microserviços surgiram visando o direcionamento objetivo dos recursos computacionais disponíveis para tais sistemas. Adicionando o particionamento presente nos sistemas distribuídos, cada pequena parte, representando uma funcionalidade de um ambiente maior, é implantada, mantida e evoluída de forma isolada.

Tal isolamento traz o baixo acoplamento e a independência de cada funcionalidade. Porém, também é adicionado a complexidade natural de todo sistema distribuído. Por existir a execução de um processo dividido em etapas presentes em mais de um lugar, é evidente que é mais complexo manter o controle de uma determinada informação e como ela é processada em cada estágio. Ainda, os testes dessas funcionalidades se tornam mais trabalhosos de serem feitos, já que um determinado serviço pode precisar de outro para execução de suas ações, fazendo com que seja mais difícil a criação de procedimentos de testes capazes de simular tal comportamento.

A distribuição de funcionalidades mesmo específicas ainda gera redundâncias por parte do código. Procedimentos como os de bancos de dados, comunicação com o *message broker* e disponibilização de recursos em forma de *webservices* devem ser recriados para cada um dos serviços. A exceção dessa regra se trata do desenvolvimento de microserviços que utilizem a mesma linguagem de programação.

Mesmo com algumas desvantagens, um dos pontos positivos evidentes é a alta coesão. Durante a criação de um microserviço, por se tratar de uma funcionalidade específica, o contexto de desenvolvimento é menor. Isso reflete na forma com que o desenvolvedor compreende a aplicação, transformando o processo em código de maneira sensata e compreensível.

Tendo a capacidade de evolução em mente, ainda existem algumas lacunas pertencentes aos princípios dos microserviços a serem preenchidas. Um dos trabalhos futuros deve tratar da resiliência. Todos os microserviços ainda podem ser melhorados para atuarem em ambientes caóticos com falhas do ambiente operacional e/ou falhas dos outros microserviços.

Além disso, os serviços ainda não são totalmente propensos à implantação independente. A criação de um processo de implantação e o desenvolvimento de estratégias para que cada serviço consiga continuar operante, mesmo executando versões de construções diferentes, é um trabalho a ser explorado.

Para tal tarefa também se faz necessário a construção de um processo de monitoria dos serviços. A checagem de status de cada um dos serviços previne o lapso de informações em tempo real. A compreensão de como está a execução real de cada serviço é parte importante para a saúde de um ambiente altamente distribuído.

Por fim, mesmo com as complexidades enfrentadas, a arquitetura de microserviços provou seu valor. Ao término do desenvolvimento do estudo de caso, foi criada uma estrutura desacoplada, versátil e dinâmica, capaz de responder facilmente a nível programático a novas exigências e evoluir sob demanda de novos requisitos.

REFERÊNCIAS

ABBOTT, M. L.; FISHER, M. T. **The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise**. Pearson Education, 2009. 592 p.

Google, 2015. AngularJS. Disponível em <https://docs.angularjs.org/api>. Acessado e 10 de out. 2017

Apache Software Foundation. Kafka 0.11 Documentation, 2016. Disponível em <https://kafka.apache.org/documentation/>. Acessado em 01 de out. 2017.

BARRY, Douglas K. **Web services, service-oriented architectures, and cloud computing.** 2. ed. Morgan Kaufmann Publishers, 2013. 248 p.

BASS, L. **Software architecture in practice**. Addison-Wesley Professional, 2007. 640 p.

BERNERS-LEE, T.; FIELDING, R.; FRYSTYK, H. **RFC 1945: Hypertext Transfer Protocol—HTTP/1.0**. Disponível em: <https://www.rfc-editor.org/rfc/rfc1945.txt>. Acessado em: 03 de mai. 2017

CHEN, F. et al. **An architecture-based approach for component-oriented development**. In: Computer Software and Applications Conference. Oxford, COMPSAC 2002. 26th Annual International. IEEE, 2002. p. 450-455.

DAYA, S. et al. **Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach**. IBM Redbooks, 2016. 176 p.

DE SORDI, J. O.; MARINHO, B. L.; NAGY, M. **Benefícios da arquitetura de software orientada a serviços para as empresas: análise da experiência do ABN AMRO Brasil**. JISTEM J.Inf.Syst. Technol. Manag. São Paulo, v. 3, n. 1, p. 19-34, 2006. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1807-17752006000100003&lng=en&nrm=iso>. Acessado em: 03 mai. 2017.

ERL, T. **Origins and Influences of Service-Orientation (Part I)**. 2009. Disponível em <http://serviceorientation.com/serviceorientation/origins_and_influences_of_service_orientation>. Acessado em 28 de mai. 2016

FERNÁNDEZ VILLAMOR, J. I.; IGLESIAS FERNANDEZ, C. A.; GARIJO AYESTARAN, M. **Microservices: Lightweight service descriptions for REST architectural style**. In: Second International Conference on Agents and Artificial Intelligence, v. 1. 2010. p. 576 - 579.

FIELDING, R. **Architectural Styles and the Design of Network-based Software Architectures**.Tese(Doutorado em Ciência da Informação e Computação). Donald Bren School of Information & Computer Sciences, 2000, 180p.

FLANAGAN, D. **JavaScript: The Definitive Guide.** 6. ed. O'Reilly Media, 2011. 1098 p.

FOWLER, M. **MicroservicePremium.** 25 de mar. 2014. Disponível em: <https://www.martinfowler.com/bliki/MicroservicePremium.html>. Acesso em 08 de mar. 2017.

_____. **Microservices.** 13 de mai. 2015. Disponível em: <http://martinfowler.com/articles/microservices.html>. Acesso em 28 de mai. 2016

_____. **Patterns of enterprise application architecture**. Addison-Wesley Professional, 2002, 560 p.

GAMMA, Erich. **Padrões de Projetos: Soluções Reutilizáveis**. Porto Alegre. Bookman editora. 2009. p 19-22.

GARLAN, D.; PERRY, D. E. **Introduction to the special issue on software architecture**. IEEE Trans. Software Eng., v. 21, n. 4, p. 269-274.

HAUPT, F. et al. **A model-driven approach for REST compliant services**. In: Web Services (ICWS), 2014 IEEE International Conference on. IEEE, 2014. p. 129-136.

HEWITT, E**. Cassandra: the definitive guide**. O'Reilly Media, 2010. 330p.

HibernateJPA. Red Hat, 2015. Disponível em: <http://hibernate.org>. Acesso em 10 de out. 2017.

HUMBLE, J.; FARLEY, D. **Continuous delivery: reliable software releases through build, test, and deployment automation**. Pearson Education, 2010, 512 p.

KHADKA, R. et al. **Migrating a large scale legacy application to SOA: Challenges and lessons learned**. In: 2013 20th Working Conference on Reverse Engineering (WCRE). IEEE, 2013. p. 425-432.

KURHINEN, Heikki. **Developing microservice-based distributed workflow engine**. Tese (Bacharelado em Tecnologia da Informação). Mikkelin ammattikorkeakoulu, 2014, 36 p.

KUROSE, J. F.; ROSS, K. W.; ZUCCHI, W. L. **Redes de Computadores e a Internet: uma abordagem top-down**. 5 ed. São Paulo: Pearson Education, 2010, 618 p.

KWAN, I.; CATALDO, M.; DAMIAN, D. **Conway's law revisited: The evidence for a task-based perspective**. IEEE software, v. 29, n. 1, p. 90-93, 2012.

MARTIN, R. C. **SRP: The Single Responsibility Principle**. 1996. Disponível em <https://web.archive.org/web/20150202200348/http://www.objectmentor.com/resources/articles/srp.pdf>. Acesso em 12 jun de 2016.

MORGADO, P. M. B. **Proposta de processo de migração para um estilo SOA**. Dissertação (Mestrado em Engenharia Informática e Computação). Faculdade de Engenharia da Universidade do Porto, 2013, 101p.

NAMIOT, D.; SNEPS-SNEPPE, M. **On micro-services architecture. International Journal of Open Information Technologies.** International Journal of Open Information Technologies. v. 2, n. 9, 2014. Disponível em <http://cyberleninka.ru/article/n/on-micro-services-architecture>. Acesso em: 12 jun. 2016.

NETWORK WORKING GROUP et al. **RFC 2616: Hypertext Transfer Protocol--HTTP/1.1**. Disponível em: < https://www.rfc-editor.org/rfc/rfc2616.txt>. Acessado em: 03 de mai. 2017

NEWMAN, S. **Building Microservices**. O'Reilly Media, 2015. 280p.

NGINX Inc. NGINX Documentation, 2017. Disponível em: <https://nginx.org/en/docs/>. Acessado em 01 de out. 2017

NodeJS Foundation. NodeJS Documentation, 2017. Disponível em: <https://nodejs.org/en/docs/>. Acesso em 10 out. 2017.

OLIVA, A. **Guarana: uma arquitetura de software para reflexão computacional implementada em Java tm**. Dissertação (Mestrado em Ciência da Computação) Instituto de Computação da Universidade Estadual de Campinas, 1998 113p.

Oracle Corporation. Java Platform Standard Edition 8, 2015. Disponível em: <https://docs.oracle.com/javase/8/>. Acesso em 10 de out. 2017

_____. Jersey Documentation, 2014. Disponível em: <https://jersey.github.io/>. Acesso em 10 de out. 2017.

PAPAZOGLOU, M, P.; VAN DEN HEUVEL, W. **Service oriented architectures: approaches, technologies and research issues**. The VLDB journal, v. 16, n. 3, p. 389-415, 2007. Disponível em <https://link.springer.com/article/10.1007%2Fs00778-007-0044-3>. Acesso em 18 nov. 2016

The PostgresSQL Global Develpment Group. PostgreSQL 9.4.14 Documentation, 2014. Disponível em <https://www.postgresql.org/docs/9.4/static/index.html>. Acesso em 01 out. 2017.

RAGGETT, D. *et al*. **Raggett on HTML 4**. Addison-Wesley Longman, 1998. 464p

Rails Comunity. Rails Documentation, 2017. Disponível em: <http://guides.rubyonrails.org/index.html>. Acesso em 10 de out. 2017.

RICHARDSON, C. 25 de mai. 2014. **Microservices: Decomposing applications for deployability and scalability**. Disponível em: <https://www.infoq.com/articles/microservices-intro>. Acesso em 02 jun. de 2016.

_____. **Microservice Patterns**. Manning Publications, 2017, 375 p.

Ruby Organization. Ruby Documentation, 2017. Disponível em <http://ruby-doc.org/core-2.4.1/>. Acesso em 10 de out. 2017

RYAN, Jaime. **Rethinking the ESB: building a secure bus with an SOA gateway**. Network Security, v. 2012, n. 1, p. 14-17, 2012. Disponível <https://link.springer.com/article/10.1007%2Fs00778-007-0044-3>. Acessado em 06 de mai. 2017.

SHARAN, K. **Learn JavaFX 8: Building User Experience and Interfaces with Java 8**. Apress, 2015, p 419-434.

SHAW, M.; GARLAN, D. **Software architecture: perspectives on an emerging discipline**. Englewood Cliffs: Prentice Hall, 1996. 242p.

TANENBAUM, A. S. **Redes de Computadores**. 4. ed. Editora Campus, 2003. p. 493-496.

THÖNES, J. (2015). **Microservices**. *IEEE Software*, v. *32*, 2015. Disponível em <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7030212>. Acesso em 06 de mai. 2017.

VILLAMIZAR, M. et al. **Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud**. In: Computing Colombian Conference (10CCC), 2015 10th. IEEE, 2015. p. 583-590.

ZHANG, Z. et al. **A soa based approach to user-oriented system migration**. In: Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on. IEEE, 2010. p. 1486-1491.