

Micro-serviços

Willian Marques Freire e Munif Gebara Junior

Resumo—Este artigo tem por objetivo a apresentação do conceito Micro-serviço, e provar que é possível desenvolver uma estrutura, que vise a alta coesão e o baixo acoplamento que é empregado quando se trata de micro-serviços. Dentre as justificativas para o desenvolvimento deste artigo, está o fato de que, o micro-serviço além de privilegiar o desenvolvimento de aplicações de acordo com o que foi citado anteriormente, o mesmo é muito utilizado quando se trata de aplicações distribuídas. Aplicações com escopos bem definidas, coreografadas, e específicas, têm surgido como o objetivo de facilitar o desenvolvimento de sistemas. Neste artigo, é desenvolvido um exemplo desta arquitetura para prova de conceito, e são feitos diversos testes de tecnologias atuais. A tecnologia principal utilizada é o Eureka da empresa Netflix. Todo este trabalho será desenvolvido durante este artigo, e ao final será apresentado os resultados do mesmo.

Palavras-chave—Microservice, Micro-serviço, Eureka.

I. INTRODUÇÃO

MICRO SERVIÇOS é um novo paradigma que influencia diretamente o modo em que são desenvolvidas, e distribuídas as aplicações. Há certas características relacionadas à sua organização, à capacidade de negócio independentes, ao *deploy* automatizado, à inteligência e controle descentralizado de liguagens e de dados [4]. Em uma publicação feita por Sampaio [3], o mesmo definiu através de estudos que Micro-serviços são componentes de alta coesão, baixo acoplamento, autônomos e independentes, que representa um contexto de negócio de uma aplicação.

Para exemplificação sobre a motivação do uso de micro-serviços, pode-se citar os sistemas ERP (*Enterprise Resource Planning* ou sistemas para Planejamentos de Recursos Empresariais), que são desenvolvidos para cuidar de setores empresariais, desde o financeiro, recursos humanos, produção, estoque, dentre outros. Em um sistema para Planejamento de Recursos Empresariais, todas as funcionalidades do mesmo são agrupadas dentro deste grande sistema, fazendo com que seja uma aplicação monolítica, ou seja, uma aplicação feita em somente uma unidade.

Por consequência, um dos principais pontos negativos, é que se houver um grande ponto de falha em algum módulo do sistema, isto poderá afetar a aplicação inteira, incluindo funcionalidades não relacionadas com o mesmo. Outro ponto negativo, é a base de código fonte, que se torna exponencialmente extensa de acordo com o tempo de desenvolvimento, tornando assim, novos membros do projeto improdutivo durante algum tempo, já que a complexidade do código é bem maior [5].

Um fato que ocorreu no ano de 2014, foi que o Docker (*container* portátil padronizado) começou a ser utilizado

amplamente pela comunidade de desenvolvedores. Uma razão importante para sua utilização generalizada que Adrian (Membro e fundados da eBays Research Labs) observa, é sua portabilidade e aumento na velocidade com *container* de aplicações, que entregava algo em minutos ou horas e passou a entregar em segundos [6]. Na figura 1 é apresentado sua utilização entre os anos 2012 e 2016.

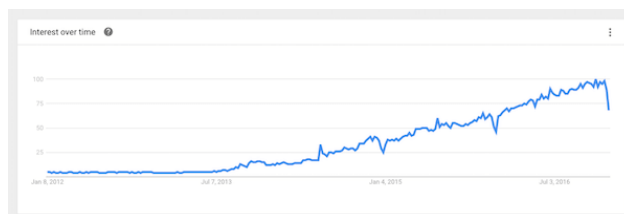


Figura 1. Gráfico de utilização do docker entre os anos 2012 e 2016.

Semelhantemente, a velocidade de desenvolvimento e implantação de um micro-serviço, permite e incentiva a implementação e estudo dos mesmos. Segundo Adrian micro-serviços possuem também outras características em comum, como implantação com pouca frequência, novas versões implantadas automaticamente, orquestração de uso geral não é necessário (uma vez que, sistemas inteiros são implantados com todas as partes ao mesmo tempo), arquiteturas utilizam centenas de micro-serviços, e cada publicação, é altamente customizada [7].

Sendo assim, implantar a Arquitetura de Micro-serviços em empresas, irá proporcionar diferentes benefícios para a estrutura de negócio como, usufruir de liberdade maior para o desenvolvimento de serviços de modo independente, implantar aplicações automaticamente através de ferramentas de integração contínua (como Hudson, Jenkins e outras), possibilitar utilização de códigos escritos em linguagens diferentes para diferentes serviços, facilitar a ampliação e integração de micro-serviços com serviços terceirizados (através de APIs), organizar o código em função de capacidades de negócio, dando assim, mais visão das ofertas e necessidades dos clientes. Dentre todos os benefícios citados, também é possível fazer o gerenciamento de falhas, visto que, o particionamento favorece uma visão mais detalhada de cada serviço, o que significa que se um serviço venha a falhar, os outros continuarão funcionando [8].

Este trabalho tem por objetivo, preencher a lacuna a respeito do desenvolvimento de uma simples estrutura para micro-serviços, fazendo com que, utilizando-se de ferramentas existentes, seja fácil a criação da mesma. Durante este trabalho, serão criados micro-serviços, que posteriormente abrirão possibilidades para integração entre eles e outros serviços existentes. O mesmo está organizado entre uma revisão bibliográfica sobre

o assunto, uma parte com desenvolvimento dos micro-serviços, e a apresentação de resultados pouco antes da conclusão.

A. Revisão bibliográfica

Segundo dados de Chris Richardson [9], diversas empresas estão utilizando micro-serviços, dentre as citadas estão: Comcast Cable, Uber, Netflix, Amazon, Ebay, SoundCloud, Karma, Groupon, Hailo, Gilt, Zalando, Lending Club, AutoScout24. Os problemas associados ao desenvolvimento de software em larga escala ocorreram em torno da década de 1960. Posteriormente, na década de 1970 viu-se um enorme aumento de interesse da comunidade de pesquisa, para o design de software em suas aplicações e no processo de desenvolvimento. Nesta década, o design foi muitas vezes considerado como uma atividade não associada com a implementação em si, e portanto requerendo um conjunto especial de notações e ferramentas. Por volta da década de 1980, a integração do design nos processos de desenvolvimento contribuiu para uma fusão parcial dessas duas atividades, tornando assim mais difícil fazer distinções puras.

Da mesma forma, as referências ao conceito de arquitetura de software também começaram a aparecer década de 1980. No entanto, uma base sólida sobre o tema foi estabelecida apenas em 1992, por Perry Wolf (autor do livro "*Foundations for the study of software architecture*"). Sua definição de arquitetura de software era distinta do design de software, e desde então tem-se gerado uma grande comunidade de pesquisadores estudando as aplicações práticas da arquitetura de software com base em micro-serviços, permitindo dddassim que os conceitos sejam amplamente adotados pela indústria e pela academia.

O advento e a difusão da orientação a objetos, a partir dos anos 80 e, em particular, a década de 1990, trouxe sua própria contribuição para o campo da Arquitetura de Software. O clássico por Gamma et al. abrange a concepção de software orientado a objetos e como traduzi-lo em código que apresenta uma coleção de soluções recorrentes, chamados padrões. Esta ideia não é nova nem exclusiva à Engenharia de Software, mas o livro é o primeiro compêndio a popularizar a ideia em grande escala. Na era pré-Gamma, os padrões para soluções orientada a objetos já estavam sendo utilizados. Um exemplo típico de um padrão de projeto arquitetônico em programação orientada a objetos, é o *Model-View-Controller (MVC)*, que tem sido um dos insights seminais no desenvolvimento precoce de interfaces gráficas de usuário [17].

Cerca de sete anos atrás a empresa Netflix (provedora global de filmes e séries de televisão via streaming - distribuição de dados, geralmente de multimídia em uma rede através de pacotes) começou a migrar suas aplicações legadas para uma arquitetura baseada em APIs, (Interface de programação de aplicativos) hospedadas na nuvem (local para armazenamento de dados online) da Amazon (empresa transnacional de comércio eletrônico dos Estados Unidos com sede em Seattle), influenciando assim, o crescimento de uma ideologia na área de desenvolvimento de softwares que foi batizada pelo nome de "micro-serviço".

Pouco antes, uma investigação realizada pela empresa Cisco (Companhia sediada em San José, Califórnia, Estados Unidos

da América) em 2016 revela que, apesar de toda a euforia sobre a Internet das Coisas, o consumo de vídeo via internet gera 63% do tráfego global. A expectativa é que essa marca chegue a 79% até 2020 e o tráfego de dados gerado por vídeos em resolução Ultra HD subirá de 1.6% para 20.7% do total em 2020. Um levantamento realizado pela Cisco VNI *Mobile* em 2016 também mostra que os dispositivos IoT mais simples, geram uma quantidade de dados equivalentes a 7 vezes o que é produzido por um celular comum (não um smartphone). Demandando pouco das redes de telecomunicações, os dispositivos IoT não representarão um grande peso para os provedores de infraestrutura na América Latina [10].

A fim de informação sobre a utilização de *internet* nos últimos anos, segundo o relatório "The State of Internet" de 2016, da Akamai (Empresa de *Internet* americana, sediada em Cambridge, Massachusetts), o país melhor colocado na faixa de redes com banda igual ou maior a 15 Mb/s é o Chile - 4,4% de seus serviços de Internet atingem essa marca. Entretanto, para chegar a essa posição, o Chile investiu pesadamente entre 2014 e 2015, conseguindo crescer 150% de um ano para outro. O Uruguai fica logo abaixo, com 4,1% de sua Internet na faixa dos 15 Mb/s. Atualmente no Brasil, somente 1,1% dos serviços atingem esta marca.

Na arquitetura de micro-serviços, quando é feito a implantação ou atualização dos mesmos, não é afetado outros serviços, e cada micro-serviço pode ser desenvolvido em uma linguagem diferente. Governança granular também é possível para cada micro-serviço, já que o mesmo não tem dependência com outros, e pode ser monitorado e coreografado separadamente. Essa arquitetura descentraliza o gerenciamento de dados, uma vez que cada micro-serviço pode armazenar seus dados de uma maneira que se adapte a ele, e são mais resistentes do que as aplicações tradicionais, devido ao fato de que uma única aplicação pode ser retirada de um monte de aplicações, considerando que são independentes uns dos outros.

1) *Tecnologias para gerenciamento de Micro-serviços:* Neste trabalho foi utilizado diversas tecnologias para desenvolvimento dos micro-serviços. A principal utilizada que é utilizado para registro centralizado das aplicações, é a tecnologia da Netflix *Service Discovery (Eureka)*. Existem outras bibliotecas que podem trabalhar em conjunto com o Eureka, e serão utilizadas neste trabalho, algumas são: Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

2) *Zuul:* Zuul é a "porta da frente" para todas as requisições de dispositivos e sites para o *back-end*. O mesmo foi construído para permitir roteamento dinâmico, monitoramento, resiliência e segurança. O Zuul foi desenvolvido pela Netflix pelo fato de que, o volume e a diversidade do tráfego da API do mesmo, resultam em problemas de produção, surgem rapidamente e sem aviso prévio, e a empresa necessitava de um sistema que permita os mesmos mudarem rapidamente o comportamento e reagir a estas situações. O Zuul utiliza diferentes tipos de filtros, que permitem aplicar rapidamente funcionalidades aos serviços de ponta.

Em suma, esses filtros ajudam a executar as seguintes funções: Autenticação e segurança, identificação de requisitos

de autenticação para cada recurso, negação de solicitações indesejadas, insights e monitoramento, rastreamento de dados significativos e estatísticas, a fim de dar uma visão precisa da produção, roteamento dinâmico, encaminhamento dinâmico solicitações para diferentes clusters de backend conforme necessário, *stress Testing*, aumento gradual de tráfego para um cluster, a fim de avaliar o desempenho, *load Shedding*, alocação de capacidade para cada tipo de solicitação e liberação de pedidos que excedem o limite, manipulação de resposta estática e construção de respostas diretamente na ponta ao invés de encaminhá-las para um cluster interno. Dentre os vários componentes que integram a biblioteca do Zuul, estão: Zuul-core que contém funcionalidades a fim de compilar e executar filtros, Zuul-simple que demonstra como construir um aplicativo com zuul-core e Zuul-netflix que adiciona componentes Netflix utilizando Ribbon para solicitações de roteamento [13].

3) *Ribbon*: Ribbon oferece suporte à comunicação entre processos na nuvem, e inclui balanceadores de carga desenvolvidos pela netflix. A tecnologia citada fornece os seguintes recursos: regras de balanceamento de carga múltiplas e conectáveis, integração com a descoberta de serviços, resiliência de falhas incorporada e clientes integrados com balanceadores de carga. O Ribbon é composto pelos seguintes projetos: *Ribbon-core* que inclui definições de interface e balanceamento de carga e cliente, implementações de balanceador de carga comuns, integração de cliente com balanceadores de carga e fábrica de clientes. *Ribbon-eureka* que inclui implementações do balanceador de carga com base no *Eureka-client* (biblioteca para registro e descoberta de serviços). *Ribbon-httpclient* que inclui a implementação de balanceamento de carga baseada em JSR-311 [12].

4) *Hystrix*: Em um ambiente distribuído, inevitavelmente algumas das muitas dependências de serviços falharão, e esta biblioteca ajuda a controlar as interações entre serviços distribuídos, adicionando tolerância de latência e lógica de tolerância a falhas. O mesmo faz isso isolando pontos de acesso entre os serviços, interrompendo falhas em cascata através deles, todas as quais melhoram a resiliência geral do sistema. Atualmente, dezenas de bilhões de threads isoladas e centenas de bilhões não isoladas, são executadas utilizando o Hystrix todos os dias na Netflix. Isso resulta em uma melhoria dramática no tempo, atividade e resiliência das aplicações. Hystrix é um projeto desenvolvido também para proteger e controlar a latência e falhas, de dependências acessadas por meio de bibliotecas de terceiros, monitoramento em tempo real, alertas e controle operacional. Quando se trata de micro-serviços, os mesmos contém dezenas de dependências com outros serviços, o que ocasiona que se um deles falhar, e o mesmo não estiver isolado destas falhas externas, corre o risco de também ser afetado. Como exemplo, um aplicativo que dependa de 40 serviços, em que cada serviço tem 99,99% de disponibilidade, pode se esperar: $99,99^{40} = 99,6\%$ de tempo de atividade, 0,4% de 1 bilhão de falhas resulta em 4 milhões de falhas. Mesmo que pequena a possibilidade de falha, se somar a quantidade de micro-serviços ao tempo de indisponibilidade que pode surgir por pequenas falhas, o problema pode ser facilmente escalável fazendo com que assim

serviços importantes fiquem até mesmos horas indisponíveis. Quando toda a aplicação está funcionando e configurada de maneira correta, o fluxo de solicitações ocorrer conforme a figura 2.

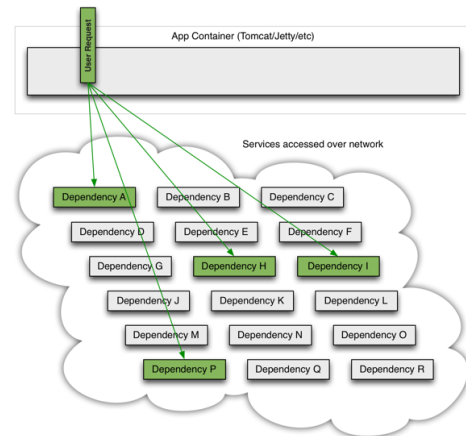


Figura 2. Wiki Hystrix (Internet Overtime) - 2015).

Quando um dos muitos serviços se torna latente, ele pode bloquear toda a solicitação do usuário, conforme apresentado na figura 3.

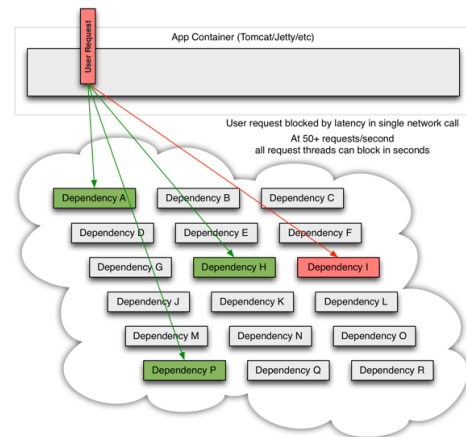


Figura 3. Wiki Hystrix (3 dimensions to scaling) - 2015).

Com tráfego de alto volume, uma única dependência com latência excessiva, pode fazer com que todos os recursos fiquem saturados em segundos. Cada ponto em um aplicativo que atinge a rede, ou em uma biblioteca cliente que pode resultar em solicitações de rede, é uma fonte de falha potencial. Esses aplicativos também podem resultar em latências entre os serviços, causando ainda mais falhas em cascata em toda a aplicação, conforme a figura 4.

A biblioteca Hystrix subjaz os seguintes princípios de design: impedir que qualquer dependência única utilize todas as threads de usuários de um container (como Tomcat) desperdiçando carga do sistema, fornecer soluções sempre que possível para proteger os usuários contra falhas, utilizar técnicas de isolamento para limitar o impacto de qualquer

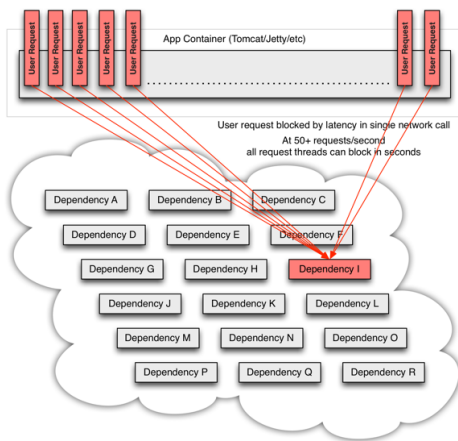


Figura 4. Wiki Hystrix (Container) - 2015).

dependência, otimizar o tempo de descoberta através de métricas, monitoramento e alertas em tempo real, otimizar o tempo de recuperação por meio de propagação de baixa latência de alterações de configuração e, oferecer suporte para alterações de propriedade dinâmicas na maioria dos aspectos do Hystrix, o que permite fazer modificações operacionais em tempo de execução com loops de realimentação de baixa latência, protegendo contra falhas em toda a execução do cliente, não apenas no tráfego da rede. (Hystrix, 2015)

5) *Eureka + Spring Cloud*: Spring Cloud fornece integrações Netflix OSS para Spring Boot por meio de auto-configuração, e vinculação ao Spring Environment e outros padrões de programação Spring. Dentre os produtos Spring Cloud encontra-se, os clientes Eureka ou Service Discovery, que é um dos princípios fundamentais de uma arquitetura baseada em micro-serviços. Configurar um micro-serviço é trabalhoso, pois envolve diversas técnicas de descoberta e registro de serviços, e com o Service Discovery da Netflix, torna-se eficiente este trabalho pois com poucas anotações Java consegue-se criar uma aplicação simples Eureka.

Eureka também vem com um componente de cliente baseado em Java, o cliente Eureka, que torna as interações com o serviço muito mais fácil. O cliente também tem um balanceador de carga incorporado, que faz balanceamento de carga *round-robin* (Algoritmos simples de agendamento e escalonamento de processos) básico. Quando um cliente se registra no Eureka, o mesmo fornece metadados como host e porta, dentre outras informações que podem ser encontradas na documentação. Se o registro falhar durante a configuração, a instância da aplicação é removida do registro. Em resumo, o Eureka é um serviço baseado em REST (Representational State Transfer), que é utilizado principalmente na AWS (*Amazon Web Services*), para localizar serviços com a finalidade de balanceamento de carga, e *failover* (tolerância a falhas) de servidores de camada intermediária.

Semelhantemente, a Amazon possui um produto chamado AWS ELB (*Amazon Web Services Elastic Load Balancer*), que é uma solução de balanceamento de carga para serviços de ponta, expostos ao tráfego web do usuário final, e a diferença

entre o mesmo e o produto da Netflix, é que o Eureka preenche a necessidade de balanceamento de carga médio. Embora teoricamente pode-se colocar serviços de nível intermediário junto com o AWS ELB, no EC2 *classic* (*Elastic Compute Cloud*), pode-se expor à rede externa, e perder toda a utilidade dos grupos de segurança AWS. O AWS ELB também possui uma solução de balanceamento de carga em proxy (servidor intermediário para requisições entre cliente e servidor final) tradicional, enquanto no Eureka, o balanceamento ocorre no nível da instância, servidor e host. As instâncias do cliente sabem todas as informações sobre quais aplicações precisam conversar.

Na Netflix, além de desempenhar um papel crítico no balanceamento de carga de nível médio, o Eureka é utilizado para os seguintes fins: implementações com Netflix Asgard, um serviço para fazer atualizações de serviços de forma rápida e segura, registro e exclusão de instâncias e transporte de metadados específicos de aplicativos adicionais sobre serviços. Dentre os motivos para utilizar o Eureka está o fato de que, o mesmo provê uma solução para balanceamento de carga *round-robin* simples, e quando não pode-se expor o tráfego das aplicações externamente com o AWS ELB, o Eureka resolve este problema.

Sendo assim, com o Eureka a comunicação é transparente, pois o mesmo fornece informações sobre os serviços desejados para comunicação, mas não impõe quaisquer restrições sobre o protocolo ou método de comunicação. Exemplificando, pode-se utilizar o Eureka para obter o endereço do servidor destino e utilizar protocolos como thrift, http(s) ou qualquer outro mecanismos RPC (*Remote Procedure Call*) que permite fazer conexões ou chamadas por espaço de endereçamento de rede.

6) *Modelo Arquitetural Eureka*: O modelo arquitetural implantado na Netflix utilizando o Eureka é descrita na figura 5. Existe um cluster por região que conhece somente instâncias de sua região. Há pelo menos um servidor Eureka por zona para lidar com falhas da mesma. Os serviços se registram e, em seguida, a cada 30 segundos enviam os chamados "batimentos cardíacos" ou requisições para renovar seus registros. Se o cliente não renovar o registro, ele é retirado do servidor em cerca de 90 segundos. As informações de registro e renovações são replicadas para todos as conexões no cluster. Os clientes de qualquer zona podem procurar as informações do registro para localizar seus serviços que podem estar em qualquer zona e fazer chamadas remotas.

Para serviços não baseados em Java, têm-se a opção de implementar a parte do cliente, utilizando o protocolo REST desenvolvido para o Eureka ou executar um "*side car*" que é uma aplicação Java com um cliente embutido Eureka, que manipula os registros e conexões. Quando se trabalha com serviços em nuvem, pensar em resiliência se torna ímprobo. Eureka se beneficia dessa experiência adquirida, e é construído para lidar com falha de um ou mais servidores do mesmo.

B. Desenvolvimento

C. Iniciando com Eureka Client

Este trabalho tem por objetivo o a pesquisa e desenvolvimento de uma estrutura de micro-serviços. Como o projeto

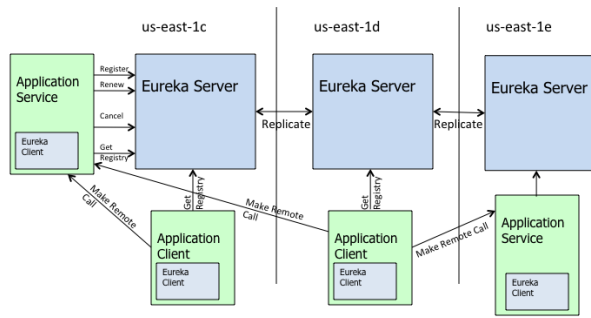


Figura 5. Wiki Eureka - 2015).

será baseado em Java, inicialmente será criado um projeto que utilizará maven, para gerenciamento de dependências, e o Eureka Client para descoberta de serviços. Após a criação de um projeto utilizando Maven, será incluso o groupId org.springframework.cloud e o artifactId spring-cloud-starter-eureka no arquivos de configuração das dependências.

Como resultado, quando um cliente se registra com o Eureka, ele fornece meta-dados sobre si, indicador de estado ou saúde, página inicial, dentre outros. O Eureka recebe mensagens heartbeat (disponibilidade) de cada instância pertencente a um serviço, e se algum heartbeat falhar, a instância é removido do registro. Para inicializar um projeto com Eureka Client, será utilizado algumas anotações Java fornecidas pelo Eureka descritas a seguir: *@Configuration*, para utilizar recursos do projeto Spring Config, para facilitar configurações de projetos Spring baseado em Java, *@ComponentScan* para buscar componentes em pacotes java, *@EnableAutoConfiguration* para ativar as configurações automáticas internas, *@EnableEurekaClient* para ativar a descoberta de serviços do Eureka, *@RestController* para criar um controlador Rest (*Representational State Transfer*), *@RequestMapping* para mapear as rotas da aplicação. As mesmas podem ser visualizadas na figura 6.

```

1 @Configuration
2 @ComponentScan
3 @EnableAutoConfiguration
4 @EnableEurekaClient
5 @RestController
6 public class Application {
7
8     @RequestMapping("/")
9     public String home() {
10         return "Hello world";
11     }
12
13     public static void main(String[] args) {
14         new SpringApplicationBuilder
15             (Application.class).web(true)
16             .run(args);
17     }
18 }

```

Figura 6. Início de utilização do Eureka

Para que possa surtir efeito na aplicação, é necessário fazer ajustes nas configurações do Eureka, dentro do diretório resources da aplicação Java. Esta configuração é feita dentro de um arquivo *Application.yml* conforme a figura 7.

```

1 Eureka
2   cliente:
3     ServiceUrl:
4       DefaultZone:
5         http://localhost:8761/eureka/

```

Figura 7. Configuração do Eureka

Neste arquivo de configuração, encontra-se uma peculiaridade. O DefaultZone é a URL do serviço Eureka para qualquer cliente. O nome do aplicativo padrão (ID de serviço), o host e a porta podem ser acessadas respectivamente pelas variáveis de ambientes: *\$spring.application.name*, *\$spring.application.name* e *\$server.port*. A anotação Java *@EnableEurekaClient* faz com que, a aplicação corrente se registre no Eureka, para que assim possa localizar outros serviços.

D. Status e Saúde do serviço

Com a página de status e os indicadores de integridade de uma instância do Eureka, é possível visualizar informações do serviço. Para acessar os indicadores de saúde, deve-se configurar as rotas padrões de acesso a mesma. Por padrão, o eureka utiliza a conexão do cliente, para determinar se está ativo. Caso não seja utilizado o Discovery Client, não será propagado o status de verificação de integridade atual do serviço. Para que os indicadores de saúde e status da aplicação funcionem corretamente, devem ser feito configurações conforme a figura 8.

```

1 eureka:
2   instance:
3     statusPageUrlPath:
4       ${management.context-path}/info
5     healthCheckUrlPath:
6       ${management.context-path}/health
7   client:
8     healthcheck:
9       enabled: true

```

Figura 8. Configuração de indicadores de Status

Para conseguir utilizar mais recursos e obter mais informações sobre o status da aplicação, a aplicação deve implementar seu próprio controle de integridade que se encontra no pacote *com.netflix.appinfo.HealthCheckHandler*

E. Alterando o ID da instância Eureka

Uma instância registrada no Eureka possui seu ID, que identifica o serviço que está no mesmo. O Spring Cloud Eureka fornece o seguinte padrão de configuração: *\${spring.cloud.client.hostname}:\${spring.application.name}:\${spring.application.instance_id:\${server.port}}*. Como exemplo a URL fica da seguinte maneira: *myhost:myapp:8080*

F. Iniciando com EurekaClient

O próximo passo para utilizar o Eureka Server para co-registrar os micro-serviços, é utilizar o Eureka Client, que pode ser utilizado para descobrir instâncias do mesmo. Para fazer isto utilizando o framework, primeiramente é necessário incluir a dependência do EurekaClient, e criar um método que busque as instâncias registradas no Eureka conforme a 9.

```

1 @Autowired
2 private EurekaClient discoveryClient;
3
4 public String serviceUrl() {
5     InstanceInfo instance =
6         discoveryClient
7             .getNextServerFromEureka
8             ("STORES", false);
9     return instance.getHomePageUrl();
10 }

```

Figura 9. Busca de instâncias

Não necessariamente é preciso utilizar o EurekaClient. Também pode-se utilizar o DiscoveryClient. A diferença entre os dois, está na maneira de como é utilizado. A mesma pode ser vista na figura 10.

```

1 @Autowired
2 private DiscoveryClient discoveryClient;
3
4 public String serviceUrl() {
5     List<ServiceInstance> list =
6         discoveryClient.getInstances
7             ("STORES");
8     if (list != null && list.size() > 0) {
9         return list.get(0).getUri();
10    }
11    return null;
12 }

```

Figura 10. Busca de instâncias com DiscoveryClient

G. Performanece de registro no Eureka

Registrar um serviço no Eureka pode ser considerado um pouco lento, pelo fato de que, ser uma instância também envolve um heartbeat periódico para o registro, com duração padrão de 30 segundos. Um serviço não estará disponível para descoberta por clientes, enquanto uma instância tenha todos os metadados em seu cache local. Para alterar o período em que isto ocorre, pode ser configurado através da propriedade *eureka.instance.leaseRenewalIntervalInSeconds*. Entretanto, em produção, não deve ser alterado este padrão pelo fato de que, existem alguns cálculos internos do Eureka, que fazem suposições de renovação de locação.

H. Zonas Eureka

Primeiramente, para se configurar uma zona Eureka, é necessário ter certeza de que existem servidores Eureka implantados em cada zona e que eles são pares uns dos outros. Em seguida, precisa-se informar em qual zona o mesmo está. Para fazer isto será utilizado a propriedade *metadataMap*. E isto pode ser feito conforme a figura 11.

```

1 eureka.instance.metadataMap.zone = zone1
2 eureka.client.preferSameZoneEureka = true

```

Figura 11. Compilação do firmware

I. Primeiros passos com Eureka Server

Como este projeto é baseado em Java no backend, e utiliza maven como gerenciador de dependências, será incluso nas configurações de dependências Maven o groupId *org.springframework.cloud* e o artifactId *spring-cloud-starter-eureka-server*. Com esta dependência adicionada, será possível utilizar o Eureka Server.

Após adicionar esta dependência, será criada a classe principal, que se encarregará de iniciar a aplicação Eureka Server. Utilizando-se da anotação *@EnableEurekaServer* fornecida pelo framework, e seguindo o padrão utilizado no Eureka Client para iniciar a aplicação, é possível ver um resultado. Um exemplo de código pode ser na figura 12.

```

1 @SpringBootApplication
2 @EnableEurekaServer
3 public class Application {
4     public static void main(String[] args) {
5         new SpringApplicationBuilder
6             (Application.class).web(true)
7             .run(args);
8     }
9 }

```

Figura 12. Código inicial Eureka Server

J. Modo Autônomo

A combinação entre o cliente e servidor Eureka, e as pulsações para verificação de disponibilidade entre os mesmos, tornam o servidor Eureka resiliente à falhas, contanto que haja algum tipo de monitoramento para mantê-lo funcionando. No modo autônomo, pode-se preferir desativar o comportamento padrão do lado do cliente, para que ele não continue tentado alcançar seus pares caso haja falha. Para isto será feito diversas configurações como pode ser visto na figura 13.

```

1 server:
2   port: 8761
3
4 eureka:
5   instance:
6     hostname: localhost
7   client:
8     registerWithEureka: false
9     fetchRegistry: false
10    serviceUrl:
11      defaultZone:
12        http://${eureka.instance.hostname}
13        :${server.port}/eureka/

```

Figura 13. Configuração Eureka Server

Acima foi apresentado as seguintes configurações: port que configura a porta em que será instalado a aplicação, hostname

para identificar o nome do host, *registerWithEureka* que indica se a própria aplicação do Eureka se registrará em si mesma, e *fetchRegistry* que diz se o Eureka buscará registros associados a eles que ainda estão executando, porém ainda não registradas no Eureka. Com o Eureka, os registros podem ser ainda mais resistentes e disponíveis, executando várias instâncias e pedindo-lhes para se registrarem uns com os outros. Tudo o que precisa para fazê-lo é configurar o *serviceUrl* dos pares, conforme a figura 14.

```

1  ———
2  spring:
3    profiles: peer1
4  eureka:
5    instance:
6      hostname: peer1
7    client:
8      serviceUrl:
9        defaultZone: http://peer2/eureka/
10 ———
11 spring:
12   profiles: peer2
13 eureka:
14   instance:
15     hostname: peer2
16   client:
17     serviceUrl:
18       defaultZone: http://peer1/eureka/

```

Figura 14. Configuração de perfis

Neste exemplo, o serviço pode ser utilizado para executar o mesmo servidor em 2 hosts (peer1 e peer2), executando-o em diferentes perfis Spring. É possível utilizar esta configuração para testar a descoberta dos pares em um único host, manipulando neste caso em servidor Linux, o arquivo hosts para resolver os nomes de host que pode ser encontrado dentro do diretório */etc/hosts*.

Em alguns casos, é preferível que o Eureka utilize os endereços IP dos serviços, ao invés do nome do host. Para isto deve ser definido a configuração *eureka.instance.preferIpAddress* como true, e quando a aplicação se registrar com o Eureka, o mesmo utilizará seu endereço IP ao invés do nome de host.

K. Clientes Hystrix

Segundo Martin Flower Et. Al [11], é comum que os sistemas de software façam chamadas remotas para software em execução em diferentes processos, provavelmente em máquinas diferentes em uma rede. Uma das grandes diferenças entre chamadas em memória e chamadas remotas é que, chamadas remotas podem falhar ou travar sem uma resposta, até que algum limite de tempo limite seja atingido. Devido a diversos problemas que podem ocorrer na arquitetura de micro-serviços, a Netflix criou a biblioteca chamada Hystrix, que implementa o padrão disjuntor que interrompe automaticamente o serviço quando ocorre falhas. Uma falha de serviço no nível inferior de serviços pode causar falha em cascata em todo o caminho até o usuário. No Hystrix o padrão de limite de falhas são 20 em 5 segundos, e quando ocorre o circuito é aberto e a chamada não é feita, isto pode ser visto na figura 15.

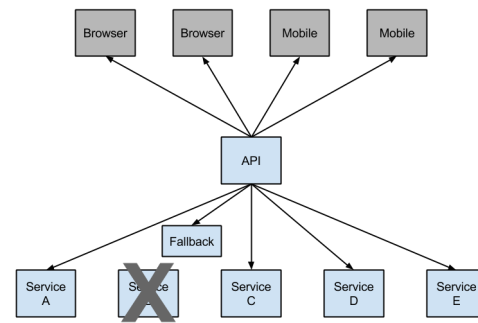


Figura 15. Fallback Hystrix em falhas em cascata.

L. Primeiros passos com Hystrix

Para utilizar o Hystrix, precisa ser incluso a dependência com o *groupId org.springframework.cloud* e o *artifactId spring-cloud-starter-Hystrix*, e a anotação *@EnableCircuitBreaker* na classe principal, conforme a figura 16.

```

1  @SpringBootApplication
2  @EnableCircuitBreaker
3  public class Application {
4
5      public static void main(String[] args) {
6          new SpringApplicationBuilder
7              (Application.class).web(true)
8              .run(args);
9      }
10 }
11
12 @Component
13 public class StoreIntegration {
14
15     @HystrixCommand
16     (fallbackMethod = "defaultStores")
17     public Object getStores
18         (Map<String, Object> parameters) {
19         //do stuff that might fail
20     }
21
22     public Object defaultStores
23         (Map<String, Object> parameters) {
24         return /* something useful */;
25     }
26 }
27 }

```

Figura 16. Hystrix

No exemplo de código da figura 16, foi implementando uma classe que se contém um método que buscará os serviços, e para isto foi utilizado a anotação *@HystrixCommand*. É possível ativar as métricas Hystrix e a central de gerenciamento do mesmo adicionando as dependências conforme a figura 17. O endpoint para acesso ao gerenciador é */hystrix.stream*.

M. Ribbon

Ribbon é um balanceador de carga do lado do cliente, que fornece controles sobre o comportamento dos clientes HTTP e TCP. Uma observação importante é que a anotação *@FeignClient* já utiliza Ribbon, fazendo com que assim seja desnecessário a utilização do Ribbon, entretanto, quando for

```

1 <dependency>
2   <groupId>
3     org.springframework.boot
4   </groupId>
5   <artifactId>
6     spring-boot-starter-actuator
7   </artifactId>
8 </dependency>
9 <dependency>
10  <groupId>
11    org.springframework.cloud
12  </groupId>
13  <artifactId>
14    spring-cloud-starter-hystrix-dashboard
15  </artifactId>
16 </dependency>

```

Figura 17. Dependências do Gerenciados Hystrix

preciso um controle mais versátil sobre a tecnologia é optável a utilização do mesmo.

Para incluir o ribbon no projeto, será utilizado o mesmo padrão de configurações feito anteriormente. O que será alterado é o artifactId da dependência maven, que agora será utilizado *spring-cloud-starter-ribbon*, e para configurar o cliente Ribbon criado uma classe de configuração e será anotado com *@RibbonClient*, conforme a figura 18.

```

1 @Configuration
2 @RibbonClient(name = "foo",
3   configuration = FooConfiguration.class)
4 public class TestConfiguration {
5 }

```

Figura 18. Ribbon

N. FeignClient

FeignClient é uma biblioteca que faz com que clientes de serviços web sejam escritos de forma mais fácil. Para utilizá-lo é preciso instalar a dependência *spring-cloud-starter-feign*, e anotar a classe principal com *@EnableFeignClients* conforme a figura 19. O mesmo provê suporte para anotações Spring MVC, e por utilizar o mesmo conversor de mensagens HTTP que o Spring Web, é integrada com Hystrix para fornecer um cliente com balanceamento de carga.

```

1 @Configuration
2 @ComponentScan
3 @EnableAutoConfiguration
4 @EnableEurekaClient
5 @EnableFeignClients
6 public class Application {
7   public static void main(String[] args) {
8     SpringApplication
9       .run(Application.class, args);
10  }
11 }

```

Figura 19. FeignClient

Ao anotar uma interface com *@FeignClient* conforme a figura 20, pode ser mapeado os métodos para que consiga

acesso a endpoints da biblioteca. O nome do método será qualificado e aplicado ao contexto da aplicação, fazendo com que assim não seja implementado corpo ao método pois será apenas repassados chamadas REST.

```

1 @FeignClient("stores")
2 public interface StoreClient {
3   @RequestMapping(method =
4     RequestMethod.GET,
5     value = "/stores")
6   List<Store> getStores();
7
8   @RequestMapping(method =
9     RequestMethod.POST,
10    value = "/stores/{storeId}",
11    consumes = "application/json")
12   Store update(
13     @PathVariable("storeId") Long storeId,
14     Store store);
15 }

```

Figura 20. Mapeamento FeignClient

Cada Cliente Feign faz parte de um conjunto de componentes que trabalham juntos, para comunicar-se via HTTP. O Spring Cloud permite com que se tenha controle total sobre clientes *Feign*, declarando uma classe de configuração que implemente determinados métodos do FeignClient conforme a figura 21. Duas das possíveis configuração, são: modificar o padrão de Contrato que o FeignClient utiliza para que assim seja personalizado o padrão de comunicação REST que o mesmo utiliza e modificar o método de autenticação do FeignClient.

```

1 @Configuration
2 public class FooConfiguration {
3   @Bean
4   public Contract feignContract() {
5     return new feign.Contract.Default();
6   }
7
8   @Bean
9   public BasicAuthRequestInterceptor
10    basicAuthRequestInterceptor() {
11     return new BasicAuthRequestInterceptor
12       ("user", "password");
13   }
14 }

```

Figura 21. Autenticação com FeignClient

O. Zuul

Até neste capítulo, foi falado de rotas e endpoints, mas ainda não foi explorado a essencialidade de utilizar rotas. Roteamento é uma parte fundamental de uma arquitetura de micro-serviços, pelo fato de que, uma uri (identificador uniforme de recursos) pode ser mapeado de acordo com sua utilidade. Como exemplo, uma uri *apiusuario* é mapeado para o serviço de usuário, *apivendas* pode ser mapeado para o serviço de vendas de uma loja. Zuul é um roteador com balanceamento de carga básico. A empresa Netflix atualmente utiliza o Zuul para os seguintes desígnios: autenticação, insights teste de

stress da api, *Canary test* que segundo Danilo Sato [15] é uma técnica para reduzir o risco de introduzir uma nova versão de software na produção lentamente lançando a mudança para um pequeno subconjunto de usuário antes de lança-la em toda a infra-estrutura e torná-la disponível para todos, roteamento dinâmico, serviço de migração, divisão de carga, segurança, manipulação de respostas e gestão de tráfego de dados.

Para incluir o Zuul em um projeto utilizando os padrões aplicados até neste capítulo, é necessário utilizar o artifactId *spring-cloud-starter-zuul*, e para habilitá-lo, a classe principal deve ser anotada com *@EnableZuulProxy*, e este encaminhará chamadas locais para o serviço adequado. Para que de acordo com a rota seja chamado o serviço desejado deve ser configurado no arquivo de configurações principais *application.yml*, e para ignorar demais serviços utiliza-se a propriedade *zuul.ignored-services*. Na figura 22 está um exemplo da utilização da mesma.

```
1 zuul:
2   ignoredServices: '*'
3   routes:
4     produtos: /meus-produtos/**
```

Figura 22. Zuul

Para obter um controle mais refinado sobre determinadas rotas, pode-se especificar o caminho e o *serviceId*, conforme a figura 23.

```
1 zuul:
2   routes:
3     produtos:
4       path: /meus-produtos/**
5       serviceId: produtos_service
```

Figura 23. Controle refinado das rotas

Isto significa que quando ocorrer uma chamada para a rota */meus-produtos*, o mesmo será encaminhado para o serviço *produtos_service*. Se for desejável especificar uma URL para uma localização física do serviço, pode ser feito conforme a figura 24.

```
1 zuul:
2   routes:
3     produtos:
4       path: /produtos/**
5       url:
6         http://exemplo.com/produtos_service
```

Figura 24. URL física do serviço

As configurações de rotas feitas até o momento, não são executadas como um *HystrixCommand* ou balanceadas com *Ribbon*. Para conseguir isto, deve-se especificar um serviço de rota e criar um cliente *Ribbon* para o *serviceId*. Na figura 25 contém um exemplo de configuração para o mesmo.

```
1 zuul:
2   routes:
3     produtos:
4       path: /meus-produtos/**
5       serviceId: produtos
6
7   ribbon:
8     eureka:
9       enabled: false
10
11  produtos:
12    ribbon:
13      listOfServers: exemplo.com,faype.com
```

Figura 25. Cliente Ribbon

P. Migração de aplicações

Um padrão comum ao migrar um serviço Web existente, é remover endpoints antigo, e lentamente substituí-los por novas implementações. O proxy Zuul é uma ferramenta extremamente útil para isto, pelo fato de que, pode-se utilizá-lo para lidar com todo o tráfego de clientes antigos, mas lidar com solicitações para novos. Na figura 26 contém um exemplo de configuração.

```
1 zuul:
2   routes:
3     primeiro:
4       path: /primeiro/**
5       url: http://primeiro.exemplo.com
6     segundo:
7       path: /segundo/**
8       url: forward:/segundo
9     terceiro:
10      path: /terceiro/**
11      url: forward:/3rd
12    legacy:
13      path: /**
14      url: http://teste.exemplo.com
```

Figura 26. Atualização de Rotas

Ao processar as solicitações de entrada, parâmetros de consulta são decodificados para que os mesmos possam estar disponíveis para possíveis modificações no filtro Zuul. Estes são recodificados ao reconstruir o pedido backend nos filtros de rota. O resultado pode ser diferente da entrada original, principalmente se o mesmo foi codificando utilizando por exemplo na linguagem Javascript a função *encodeURIComponent()*. Isto não causa problemas na maioria dos casos, entretanto, algumas aplicações web podem exigir codificação de url para consultas complexas. Para forçar a codificação original da url, é possível utilizar a configuração *zuul.forceOriginalQueryStringEncoding* definindo-a como *true*.

Q. RxJava com Spring MVC

RxJava é uma implementação Java VM de extensões reativas: uma biblioteca para compor programas assíncronos e baseados em eventos utilizando sequências observáveis [19].

Spring Cloud fornece suporte para *observables* que em RxJava é um objeto que implementa a interface *Observable*

que em seguida, este assinante reage a qualquer item ou sequência de itens que o objeto *Observable* emite. Esse padrão facilita operações simultâneas porque não precisa bloquear enquanto espera que o *Observable* emita objetos, mas ao invés disto, cria uma sentinela na forma de assinante que está pronto para reagir apropriadamente em qualquer tempo futuro que o *Observable* gere [19]. Utilizando o Spring Cloud pode-se retornar objetos *rx.Single*, *rx.Observable* e SSE (Eventos enviados pelo servidor) que é uma tecnologia pelo qual um navegador recebe atualizações de um servidor via HTTP. Na figura 27 está dois exemplos, segundo a Netflix [16] de como utilizá-los.

```

1 @RequestMapping(method =
2   RequestMethod.GET, value = "/multiple")
3   public Single<List<String>> multiple() {
4       return Observable
5         .just("multiple", "values")
6         .toList().toSingle();
7   }
8
9 @RequestMapping(
10   method = RequestMethod.GET,
11   value = "/responseWithObservable")
12   public ResponseEntity<Single<String>>
13     responseWithObservable() {
14
15     Observable<String> observable =
16     Observable.just("single value");
17     HttpHeaders headers = new HttpHeaders();
18     headers
19       .setContentType(APPLICATION_JSON_UTF8);
20     return new ResponseEntity<>(
21       observable.toSingle(),
22       headers, HttpStatus.CREATED);
23   }

```

Figura 27. Compilação do firmware

R. Motivação de uso das tecnologias apresentadas

Até o prezado momento, foram apresentados diversas tecnologias, entretanto, fica a questão sobre o por que de tantas tecnologias. e a resposta é clara e objetiva. Para que se tenham micro-serviços, é necessário a utilização de um recurso para a orquestração dos mesmos, neste caso, entra o Eureka. Quando se tem micro-serviços, é necessário a centralização dos mesmos para que não fique espalhados e perdidos em hosts ou portas diferentes, surge a necessidade de um gateway, neste caso, o Zuul. A partir do momento que são integrados muitos micro-serviços, torna-se difícil o gerenciamento da configuração dos mesmos, surgindo a necessidade de um mecanismo para fácil configuração e integração de aplicações, por isto, foi apresentado o Spring Config. Após tudo isto, surge ainda um problema, o balanceamento de carga, um problema abrangente quando se trabalha com aplicações distribuídas. Surge então, a motivação de utilizar o Ribbon para distribuição de carga. Com tudo isto configurado, ainda não se tem garantia precisa de disponibilidade da aplicação, fazendo com que, se caso ocorrer um problema em um micro-serviço, necessite de uma tecnologia que resolva esta questão, fazendo assim necessário a utilização do Hystrix. A motivação de apresentar

tecnologias Spring Cloud e Netflix, como citado na introdução deste trabalho, é o fato da experiência e sucesso por parte dos mesmos em aplicações distribuídas.

II. RESULTADOS

Desenvolver uma estrutura utilizando micro-serviços pode ser trabalhoso, quando se tem muitos serviços para gerenciar se torna um processo complexo. Entretanto, existem diversas ferramentas que auxiliam neste processo. Durante este trabalho, foram feitos diversos testes, utilizando ferramentas da empresa Netflix para auxílio no processo de desenvolvimento de micro-serviços. Foi utilizado ferramentas para registro de micro-serviços, balanceamento de carga e tolerância a falhas. Em todos foi obtido resultados e têm funcionado da forma esperada.

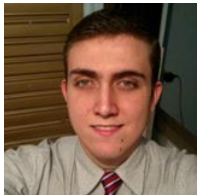
III. CONCLUSÕES E TRABALHOS FUTUROS

Conclui-se portanto, que é possível criar uma estrutura utilizando micro-serviços, e que existem dificuldades ao gerenciar este estrutura, quando existem muitos micro-serviços envolvidos. Mas utilizando-se de ferramentas para auxílio no desenvolvimento desta estrutura, torna-se possível a criação da mesma. Quando é utilizado esta estrutura, os sistemas são mais tolerantes a falhas pelo fato de que, se um serviço parar, os demais continuam. Um ponto a ser levantado, é que as aplicações possuem alta coesão e baixo acoplamento, o que facilita no gerenciamento das mesmas.

REFERÊNCIAS

- [1] Paulo Silveira. Mauricio Balboa Linhares. Tecnologias na Netflix. 2017 [Online] Disponível: http://content.blubrry.com/hipsterstech/hipsters_041_netflix.mp3. [Acesso: 25-Abr-2017]
- [2] Flavio Silveira. Microserviços - O que é? 2016 [Online] Disponível: <http://flaviosilveira.com/2016/microservicos>. [Acesso: 06-Mai-2017]
- [3] Cleuton Sampaio. Micro serviços: O que são e para que servem. 2015 [Online] Disponível: <http://www.obomprogramador.com/2015/03/micro-servicos-o-que-sao-e-para-que.html>. [Acesso: 28-Fev-2017]
- [4] James Lewis. (2016). Microserviços em poucas palavras. [Online] Disponível: <https://www.thoughtworks.com/pt/insights/blog/microservices-nutshell>. [Acesso: 20-Mar-2017]
- [5] Adriano Almeida. Arquitetura de microserviços ou monolítica. 2015 [Online] Disponível: <http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica>. [Acesso: 22-Mar-2017]
- [6] Cristiano Diedrich. O que é Docker. 2015 [Online] Disponível: <http://www.mundodocker.com.br/o-que-e-docker>. [Acesso: 01-Dez-2016]
- [7] Jan Stenberg. O estado da arte em micro serviços. 2015 [Online] Disponível: <https://www.infoq.com/br/news/2015/04/microservices-current-state>. [Acesso: 08-Fev-2017]
- [8] Ricardo Peloi. Como implantar uma verdadeira Arquitetura de Microserviços na sua empresa. 2016 [Online] Disponível: <http://sensedia.com/blog/soa/implantar-arquitetura-de-microservicos>. [Acesso: 08-Mar-2017]
- [9] Chris Richardson. Whois using microservices. 2016 [Online] Disponível: <http://microservices.io/articles/whoisusingmicroservices.html>. [Acesso: 20-Mai-2017]
- [10] IDC. Connecting the IoT: The road to success. 2016 [Online] Disponível: <http://www.idc.com/infographics/IoT>. [Acesso: 15-Fev-2017]
- [11] Martin Fowler et al. Microservices. 2014 [Online] Disponível: <https://www.martinfowler.com/articles/microservices.html>. [Acesso: 20-Mai-2017]
- [12] Netflix. Ribbon. 2016 [Online] Disponível: <https://github.com/Netflix/ribbon/wiki>. [Acesso: 06-Mai-2017]
- [13] Netflix. Zuul. 2016 [Online] Disponível: <https://github.com/Netflix/zuul/wiki>. [Acesso: 02-Mai-2017]

- [14] Martin Fowler. CircuitBreaker. 2014 [Online] Disponível: <https://martinfowler.com/bliki/CircuitBreaker.html>. [Acesso: 02-Mar-2017]
- [15] Danilo Sato. CanaryRelease. 2014 [Online] Disponível: <https://martinfowler.com/bliki/CanaryRelease.html>. [Acesso: 10-Abr-2017]
- [16] Netflix. Spring Cloud Netflix. 2016 [Online] Disponível: <http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html>. [Acesso: 18-Dez-2016]
- [17] Nicola Dragoni et al. *Microservices: yesterday, today, and tomorrow*. Technical University of Denmark, vol 1, pp. 1-3, 2016
- [18] Willian Marques Freire, Munif Gebara Junior. Artigo IoT - Internet das coisas. vol 1, pp.1-3, 2017
- [19] ReactiveX. Observable. 2015 [Online] Disponível: <http://reactivex.io/documentation/observable.html>. [Acesso: 25-Nov-2016]



Willian Marques Freire Possui Ensino Médio completo pelo Colégio Estadual Rosa Delúcia Calsavara (2013). Atualmente é Desenvolvedor de Software da Gumga Tecnologia da Informação S/A. Tem experiência na área de Ciência da Computação.



Munif Gebara Júnior Possui graduação em Ciência da Computação pela Universidade Estadual de Maringá (1997) e mestrado em Engenharia Elétrica e Informática Industrial pela Universidade Tecnológica Federal do Paraná (2001). Atualmente é professor da Fundação Faculdade de Filosofia Ciências e Letras de Mandaguari e professor de ensino superior da Faculdade de Tecnologia e Ciências do Norte do Paraná Ltda e desenvolvedor.