INTERFACE TECNOLÓGICA

UM ESTUDO SOBRE O CONCEITO E A APLICAÇÃO DA ARQUITETURA DE MICROSERVIÇOS

A STUDY ON THE CONCEPT AND THE APPLICATION OF MICROSERVICE ARCHITECTURE

Leticia Miranda Malipense – leticia.malipense@gmail.com
Jederson Donizete Zuchi – jederson.zuchi@fatectq.edu.br
Faculdade de Tecnologia de Taquaritinga (FATEC) – SP – Brasil

DOI: 10.31510/infa.v15i1.324

RESUMO

Este artigo tem como objetivo apresentar o conceito e a aplicação da arquitetura de microserviços, demonstrando suas vantagens e riscos dentro do universo de desenvolvimento de software, frente ao modelo monolítico tradicional. Para isso foi realizada uma pesquisa bibliográfica com uso de livros e artigos científicos, e desenvolvido uma prova de conceito utilizando este modelo arquitetural. Contudo conclui-se que a arquitetura de microserviços representa um grande avanço para o futuro do desenvolvimento de software, uma vez que esta possibilita a escalabilidade de recursos, desacoplamento entre componentes, além de uma melhor organização de código, o que pode gerar um ganho econômico considerável para as empresas, eliminando assim as barreiras impostas pelo modelo monolítico.

Palavras-chave: Desenvolvimento**.** Arquitetura. Microserviços.

ABSTRACT

This article has the objective to show the concept and application of the microservice architecture, demonstrating its advantages and risks within the universe of software development, against the traditional monolithic model. In order to validate this approach, a bibliographic research was carried out using books and scientific articles, and a prof a concept was developed using this architectural model. It is concluded that the microservice architecture represents a big advance for the future of software development, since it allows the scalability of resources, decoupling between components, and a better organization of code, which can generate economic gain for the companies, eliminating the barriers imposed by the monolithic model.

Keywords: Development**.** Architecture. Microservices.

1 INTRODUÇÃO

A tecnologia da informação (TI) se torna cada dia mais presente nos ambientes coorporativos. Com o avanço da globalização o mundo competitivo vem cada vez mais

tomando espaço, esse ambiente turbulento, que se transforma a todo instante, exige das empresas um sistema de informação ágil que acompanhe o ritmo das transformações (PILATI, 2013).

A medida que os processos se tornam cada vez mais automatizados dentro do ambiente coorporativo, surge a necessidade de evoluir os sistemas para que os mesmos possam atender a essa demanda emergente. Dentro desse contexto surgiram novos desafios para o universo do desenvolvimento de software como o crescente aumento da complexidade de código. Segundo Newman (2015) as bases de códigos crescem à medida que se escreve código para adicionar novos recursos. Ao longo do tempo, pode ser difícil saber onde uma mudança precisa ser feita, pois, a base do código é muito grande.

Contudo a necessidade de acoplar novas funcionalidades ou recursos aos sistemas pode levar a outro ponto de atenção para as instituições, que é o aumento do consumo de *hardware*, uma vez que as aplicações ganham maior complexidade e responsabilidade é inevitável que elas passem a utilizar mais recursos físicos para funcionarem corretamente. De acordo com Newman (2015) em uma aplicação gigante monolítica, tem-se que lidar e saber dimensionar tudo como uma peça. Com os serviços menores, se pode apenas escalar os serviços que precisam ser dimensionados, permitindo executar outras partes do sistema em *hardware* menor e menos poderoso. A incapacidade de orquestrar recursos pode representar um alto custo para manutenção do software.

Nesse sentido este trabalho tem a finalidade de mostrar no que consiste a arquitetura de microserviços e como esta pode contribuir para enfrentar problemas acarretados pela evolução dos sistemas, quais os ganhos e os riscos que acometem a implementação da mesma.

2 TECNOLOGIAS E FERRAMENTAS UTILIZADAS

O trabalho visa demostrar o conceito e a aplicação da arquitetura de microserviços, suas vantagens e desvantagens, e como este se distingue do modelo monolito tradicional. Além disso esse capítulo apresenta as ferramentas utilizadas no desenvolvimento do projeto.

2.1 O que é a arquitetura de microserviços

De acordo com Lewis e Fowler (2014) o termo "*Microservice Architecture*" surgiu nos últimos anos para descrever uma maneira particular de conceber aplicações de software como suítes de serviços independentemente implementáveis.

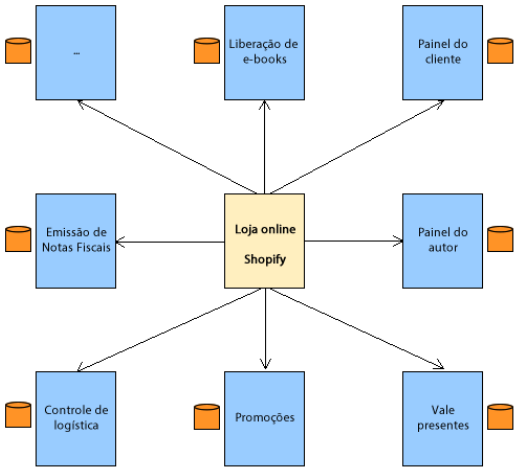
Segundo Gambarra (2017) embora não haja uma definição formal de microserviços, existem certas características que norteiam esse padrão arquitetônico. A principal ideia é construir serviços que sejam pequenos, independentes e focados na resolução de um único problema dentro de um ecossistema de uma aplicação.

Em resumo a arquitetura de microserviços é um modelo arquitetônico, que aborda o desenvolvimento de uma aplicação composta por vários serviços, onde cada serviço é autônomo, ou seja, independente dos demais, e responsável por um conjunto finito de funcionalidade, se comunicando muitas vezes através de APIs(*application programming interface*) de recursos HTTP (*HyperText Transfer Protoco*). Newman (2015) define que microserviços são pequenos serviços autônomos que trabalham juntos, cada qual como uma entidade separada, que pode ser implementado como um serviço isolado em uma plataforma como serviço (PAAS).

A autonomia e isolamento existente entre os serviços permitem que os mesmos sejam desenvolvidos utilizando diferentes linguagens de programação e tecnologias, possibilita que o processo de implantação de cada um deles seja totalmente isolado, sendo assim as aplicações podem ser instaladas em diferentes máquinas.

Na Figura 1 é demonstrado um exemplo visual da uma aplicação utilizando microserviços.

Figura 1 - Arquitetura de microserviços



Fonte: ALMEIDA (2015)

A arquitetura de microserviços surgiu como alternativa frente a alguns problemas da então padrão e famosamente difundida arquitetura monolítica.

2.2 Arquitetura Monolítica

Em uma infraestrutura monolítica os serviços que compõem um sistema são organizados logicamente no mesmo código fonte e unidade de instalação. Isso permite a dependência entre os serviços que serão gerenciados dentro do mesmo ambiente de execução e também significa que modelos e recursos comuns podem ser compartilhados entre os componentes do sistema (WOODS, 2015).

De acordo com Gambarra (2017) no passado, muitas aplicações eram construídas de forma Monolítica. Todo o negócio estava embutido em um único programa e, às vezes, até distribuídos em camadas. Porém, sempre com um alto acoplamento entre esses componentes.

As aplicações empresariais são muitas vezes construídas em três partes principais: uma interface de usuário do lado do cliente (consistindo de páginas HTML (*HyperText Transfer Protocol*) e javascript executadas em um navegador na máquina do usuário) um banco de dados (consistindo de várias tabelas inseridas em um gerenciamento de banco de dados comum e geralmente relacional) e uma aplicação do lado do servidor. O aplicativo do lado do servidor tratará pedidos HTTP, executará lógica de domínio, recuperará e atualizará dados do banco de dados e selecionará e preencherá as visualizações HTML para serem enviadas para o navegador.

Esta aplicação do lado do servidor é um monolito - um único executável lógico (LEWIS; FOWLER, 2014).

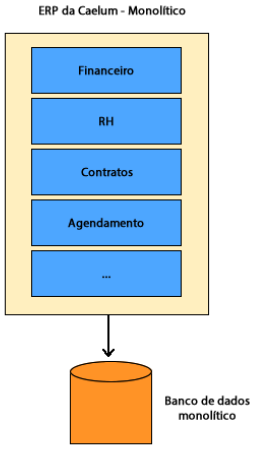
Sendo assim mesmo que a aplicação tenha um *design* modular no final o compilado gerará um único pacote a ser executado e instalado no servidor de aplicação.

> Um dos principais pontos negativos é que você tem um grande ponto único de falha, que significa que se houver algum erro em algum cadastro, isso levará junto todo o sistema, incluindo funcionalidades que não possuem nenhuma relação com essa. Outro ponto negativo é a base de código, que se torna muito extensa, podendo deixar novos membros do projeto menos produtivos por algum tempo, já que a complexidade do código é bem maior (ALMEIDA, 2015).
>
> Por outro lado, temos um sistema cujo o *deploy* é fácil de ser feito, já que o banco de dados facilmente evoluirá junto para todas as funcionalidades e há apenas um ponto onde o *deploy* precisa ser feito. Além disso, não há duplicidade de código e classes necessárias entre os diferentes módulos, já que todas elas fazem parte da mesma unidade (ALMEIDA, 2015).

A arquitetura monolítica pode ainda ser extremamente atrativa para aplicações de baixa complexidade, tornando o processo de *deploy* mais simples, à medida que as aplicações se tornam cada vez mais complexas, a mesma pode onerar a produtividade de novos membros, dificultar a integração com novas ferramentas e ainda prejudicar a manutenabilidade e qualidade do código.

Segue na Figura 2 um exemplo visual de uma aplicação monolítica.

Figura 2 - Arquitetura monolítica



Fonte: ALMEIDA (2015)

2.3 Vantagens da Arquitetura de Microserviços

Segundo Newman (2015) umas das vantagens de se utilizar da arquitetura de microserviços está na possibilidade de se usar tecnologias diferentes dentro de cada serviço. Isso permite escolher a ferramenta certa para cada trabalho. Em vez de ter que selecionar uma abordagem mais padronizada de tamanho único. Se uma parte do sistema precisa melhorar seu desempenho, há a possibilidade de decidir usar uma tecnologia diferente que é mais capaz de atingir os níveis de desempenho requeridos.

A arquitetura de microserviços prove também a diminuição da complexidade dentro das aplicações. De acordo com Almeida (2015) a base de código menor, facilita a barreira inicial na compreensão do projeto para um novo membro do time, outro fator importante é o isolamento das funcionalidades do sistema, sendo assim, esse novo modelo arquitetural também aparta as falhas que possam ocorrer em uma determinada funcionalidade, permitindo que o restante das aplicações permaneça funcional.

Neste contexto Woods (2015) categoriza microserviços como sistemas distribuídos sendo que, um microserviço é responsável por gerenciar um único domínio e as funcionalidades que manipulam esse domínio. Um sistema distribuído decompõe os componentes de um sistema monolítico em unidades individuais de distribuição, capazes de evoluir com as suas próprias exigências de escalabilidade e independente dos outros subsistemas. Isso significa que o impacto de um recurso no sistema como um todo pode ser gerenciado de forma mais eficiente e a conexão entre componentes pode compartilhar um contrato menos rígido, pois a dependência não é mais gerenciada através de um ambiente de execução (WOODS, 2015).

O desacoplamento entre os serviços, facilita o escalonamento de recursos de máquina, sendo possível que para um determinado serviço que necessite de uma capacidade maior de processamento seja escalonado um *hardware* mais poderoso, enquanto que para outros, onde não existe essa necessidade seja atribuído uma quantidade menor de recursos.

Com o uso de microserviços, também se faz capaz a adoção de novas tecnologias mais rapidamente, uma das maiores barreiras para experimentação e a adoção de novas tecnologias são os riscos associados a esse processo. Com uma aplicação monolítica, caso haja a experimentação de uma nova linguagem de programação, banco de dados ou estrutura qualquer, a mudança afetará uma grande quantidade do sistema (NEWMAN, 2015).

Com um sistema que consiste em múltiplos serviços, existem vários lugares para experimentar uma nova tecnologia, sendo possível escolher um serviço que talvez seja o de

menor risco para uso da tecnologia, sabendo ser possível limitar qualquer potencial impacto negativo (NEWMAN, 2015).

Dessa forma a arquitetura de microserviços contribui para a inovação e exploração de novas abordagens, enquanto que o monolítico pelo risco que apresenta tende a permanecer arcaico

2.4 Desvantagens da Arquitetura de Microserviços

Abordando a arquitetura de microserviços não se pode deixar de abranger todos os pontos, e nem admitir que a mesma é a ideal em todas as situações, para isso nesta seção serão abordados alguns pontos onde os microserviços assumem uma posição de desvantagens, Segundo Fowler (2014) são os seguintes:

- **Distribuição:** os sistemas distribuídos são mais difíceis de programar, uma vez que as chamadas remotas são lentas e correm risco de falha.
- **Consistência eventual:** manter uma consistência forte é extremamente difícil para um sistema distribuído, o que significa que todos devem gerenciar a consistência eventual.
- **Complexidade operacional**: é necessária uma equipe de operações maduras para gerenciar muitos serviços, que estão sendo redistribuídos regularmente.

2.5 Arquitetura RESTful

> REST é um termo definido por Roy Fielding em sua tese de mestrado no qual ele descreve sobre um estilo de arquitetura de software sobre um sistema operado em rede. REST é um acrônimo para "Transferência de Estado Representacional" (*Representational State Transfer*) (RONDON, 2010).

RESTful nada mais é do que a aplicabilidade da arquitetura. A arquitetura REST pode ser entendida como uma abstração da arquitetura WEB (*World Wide Web*), ou seja, o REST consiste em princípios/regras/*constraints* que quando seguidas, permitem a criação de um projeto com interfaces bem definidas. Desta forma, permitindo, por exemplo, que aplicações se comuniquem (PIRES, 2017).

Segundo Ferreira (2017) REST na verdade pode ser considerado como um conjunto de princípios, que quando aplicados de maneira correta em uma aplicação, a beneficia com a arquitetura e padrões da própria web.

2.6 Spring Framework

Spring é um framework de código aberto (*open source*), criado por Rod Johnson, em meados de 2002, e apresentado no seu livro *Expert One-on-One*: JEE *Design and Development*. Foi criado com o intuito simplificar a programação em Java (GENTIL, 2012).

Segundo Santana (2014) o Spring é um dos frameworks Java mais conhecido e utilizado. Esse framework implementa um grande número de funções, como injeção de dependência, persistência de dados e uma implementação para o padrão MVC (*Model View Controller*) para a criação de aplicações WEB.

Spring atualmente possui diversos módulos mas o principal (core) pode ser utilizado em qualquer aplicação Java, as principais funcionalidades são a injeção de dependência e a programação orientada a aspectos, cabe ao desenvolvedor dizer ao Spring o que quer usar. O que faz dele uma poderosa ferramenta, pois não existe a necessidade de se arrastar todas as ferramentas do framework para criar uma aplicação simples (GENTIL, 2012).

2.7 Spring Boot

O Spring Boot é um framework com base no Spring Framework, com isso ele usufrui de toda a maturidade do Spring Framework, e ainda abstrai a sua complexidade de configuração, o que facilita no processo de desenvolvimento. Grande parte do Spring Boot foi construído pensado na produtividade do desenvolvedor, tornando conceitos como RESTful HTTP e ambientes de execução de aplicações Web embarcados fáceis de conectar e usar (WOODS, 2015).

Lobo (2017) define o Spring Boot como mais um framework, mas talvez a melhor denominação seja micro framework. Seu objetivo não é trazer novas soluções para problemas que já foram resolvidos, mas sim reaproveitar estas tecnologias e aumentar a produtividade do desenvolvedor.

De acordo com Afonso (2017) o Spring Boot veio para facilitar o processo de configuração e publicação de nossas aplicações. A intenção é ter o seu projeto rodando o mais rápido possível e sem complicação. Ele consegue isso favorecendo a convenção sobre a configuração. Basta que informe a ele quais módulos deseja utilizar.

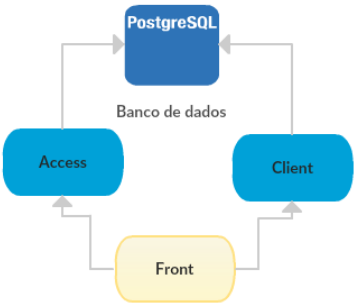
3 CRIANDO UM MICROSERVIÇO

Com o intuito de exemplificar de forma prática a aplicação da arquitetura de microserviços, foi desenvolvido uma API responsável por efetuar o cadastro de clientes, onde os administradores autenticados podem realizar o cadastro. O projeto foi desenvolvido utilizando a linguagem Java e o framework SpringBoot já mencionado, para a base de dados foi utilizado o PostgreSQL. Não foi abordado detalhadamente o *frontend* do projeto, pois a finalidade é demonstrar a implementação do modelo arquitetural no *backend*.

Para isso foram criados dois microserviços o primeiro chamado *Access* responsável por realizar a autenticação e cadastros dos usuários administradores e outro chamado *Client*, que possui a finalidade de executar todo o gerenciamento dos clientes, como cadastro, alteração, exclusão e listagens. Ambos os serviços são APIs que utilizam o padrão REST, portanto toda a comunicação com estas é feita utilizando este conceito, além disso os dois se comunicam com a mesma base de dados.

Segue na Figura 3 o diagrama que representa o modelo desenvolvido.

Figura 3 – Diagrama da arquitetura desenvolvida para o projeto



Fonte: Elaborada pelo autor

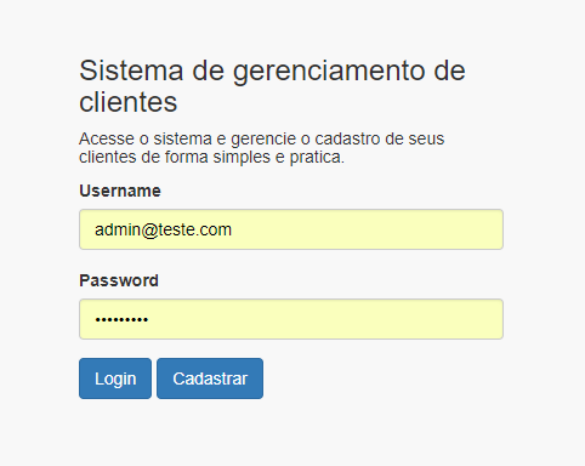
Os códigos fontes do projeto encontram-se disponíveis para consulta no GitHub nos links:

- **Frontend:** https://github.com/leticiaMalipense/Frontend.git
- **Access:** https://github.com/leticiaMalipense/Access.git.
- **Client:** https://github.com/leticiaMalipense/Client.git.

Na tela inicial da aplicação é possível realizar o *login* ou efetuar o cadastro, caso ainda não tenha realizado, informando apenas email e senha. Para o fluxo onde usuário esteja se cadastrando no sistema, o *front* encaminha para o serviço *Access* as informações digitadas no

formulário, esta aplicação fica responsável por receber esses dados em um *endpoint*, executa uma validação para garantir que o email informado ainda não pertença a nenhum outro usuário e persiste esses dados na base, com a senha criptografada para garantir a segurança dessa informação. Na Figura 4 pode se-visualizar a tela de *login*:

Figura 4 – Tela de login.



Fonte: Elaborada pelo autor

No entanto caso seja efetuado o fluxo de *login*, o *front* encaminha os dados do usuário para o *endpoint* de *login* ainda do serviço *Access*, porem desta vez utilizando o SpringSecurity as informações do usuário são autenticadas na base de dados, para quando está autenticação for positiva é gerado um *token* por meio da tecnologia Json Web Token (JWT) e retornado para o *front*, este *token* é utilizado em todas as demais requisições aos serviços para garantir a autenticidade dos acessos.

Após o usuário efetuar o *login* na aplicação é apresentado inicialmente a *home*, onde consta o CRUD (acrónimo de *Create*, *Read*, *Update* e *Delete* na língua Inglesa) de clientes, ou seja, nela é apresentada uma *grid* para visualização das informações dos clientes com as opções editar e deletar, e o botão novo que redireciona para a tela de cadastro de cliente, conforme Figura 5:

Figura 5 – Home

Fonte: Elaborada pelo autor

Todas as ações referentes a clientes são realizadas pelo serviço *Client*, responsável por gerenciar as informações deste, portanto a lista de clientes e as ações de exclusão e edição, são feitas por ele, o fluxo de cadastro de clientes também é realização pelo mesmo serviço, onde é feita a requisição ao *endpoint* de cliente e submetido as informações imputadas no formulário, nesse momento é realizada uma validação com relação ao CPF informado, para que não seja possível ser cadastrado mais de um cliente com esta mesma informação, para quando essa validação é bem sucedida o cliente é persistido na base, caso contrário é retornado um erro pelo serviço e o *front* trata essa exceção apresentando uma mensagem para o usuário informando o ocorrido.

Quando se faz uso da arquitetura de microserviços dentro desta aplicação se pode usufruir das vantagens que esta implica, portando como se tem dois serviços desacoplados caso ocorra uma falha no serviço *Client* a aplicação ainda estaria com suas demais funcionalidades trabalhando, porem se tratando de um exemplo meramente ilustrativo está aplicação possui apenas outro serviço que se manteria funcional, e caso existissem outros estes também não seriam impactados. Além disso se para um dos serviços se fizer necessário mais processamento é possível sem grandes problemas migra-lo para uma máquina com maior capacidade, o que significa que se pode escalar os nossos *hardwares* de acordo com a necessidade, sem ter que obrigatoriamente adquirir uma máquina extremamente robusta para comportar toda a aplicação.

4 CONSIDERAÇÕES FINAIS

Com base no estudo realizado conclui-se que a arquitetura de microserviços representa um grande avanço para o futuro do desenvolvimento de software, uma vez que esta possibilita a escalabilidade de recursos, desacoplamento entre componentes, além de uma melhor

organização de código, o que pode gerar um ganho econômico considerável para as empresas, eliminando assim as barreiras impostas pelo modelo monolítico.

No entanto apesar de suas vantagens a arquitetura de microserviços não é a ideal para todas as aplicações, em casos de aplicações menores esta pode aumentar a complexidade do projeto e do processo de *deploy*, além disso o fato de se trabalhar com vários serviços agrega ao projeto um novo desafio, que é a integração dessas várias aplicações distribuídas, sendo assim pode atribuir uma complexidade desnecessária, gerando um custo maior para o desenvolvimento da mesma. Dessa forma caso utilizada para os devidos fins a arquitetura de microserviços podem ser a resolução para vários problemas enfrentados hoje em dia.

Por meio da aplicação desenvolvida neste artigo utilizando a arquitetura de microserviços, foi possível compreender de maneira ilustrativa como funciona uma aplicação trabalhando com este conceito.

REFERÊNCIAS

AFONSO, Alexandre. **O que é Spring Boot?**. Disponível em <http://blog.algaworks.com/spring-boot/>. Acesso em: 11 abr. 2018.

ALMEIDA, Adriano. **Arquitetura de microserviços ou monolítica?**. Disponível em <http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica/>. Acesso em: 11 abr. 2017.

FERREIRA, Rodrigo. **REST: Princípios e boas práticas**. Disponível em < http://blog.caelum.com.br/rest-principios-e-boas-praticas/>. Acesso em: 11 abr. 2018.

FOWLER, Martin. **Microservice Trade-Offs**. Disponível em < https://martinfowler.com/articles/microservice-trade-offs.html />. Acesso em: 17 jun. 2017.

GAMBARRA, Thiago. **Alta escalabilidade com microserviços**. Disponível em < https://www.mundipagg.com/blog/alta-escalabilidade-com-microservicos/>. Acesso em: 04 abr. 2017.

GENTIL, Efraim.**Introdução ao Spring Framework**. Disponível em < https://www.devmedia.com.br/introducao-ao-spring-framework/26212>. Acesso em: 11 abr. 2018.

LEWIS, james; FOWLER, Martin. **Microservices**. Disponível em < https://martinfowler.com/articles/microservices.html/>. Acesso em: 17 jun. 2017.

LOBO, Henrique. **Spring Boot: simplificando o Spring**. Disponível em < https://www.devmedia.com.br/spring-boot-simplificando-o-spring/31979>. Acesso em: 12 abr. 2018.

NEWMAN, Sam. **Building Microservices**: Designing fine-grained systems. 1. ed. Califórnia: O'Reilly Media, 2015.

PILATI, Joel. **Impacto da TI nas organizações**. Disponível em <http://www.administradores.com.br/producao-academica/impacto-da-ti-nas-organizacoes/5568/>. Acesso em: 10 abr. 2018.

PIRES, Jackson. **O que é API? REST e RESTful?** Conheça as definições e diferenças!. Disponível em < https://becode.com.br/o-que-e-api-rest-e-restful/>. Acesso em: 18 jun. 2017.

RONDON, Thiago. **Arquitetura REST e o serviço web 'RESTful'**. Disponível em < http://sao-paulo.pm.org/pub/arquitetura-rest-e-o-servico-web-restful-/>. Acesso em: 18 jun. 2017.

SANTANA, Eduardo. **Java Spring MVC:** Criando Aplicações Web em Java. Disponível em <https://www.devmedia.com.br/java-spring-mvc-criando-aplicacoes-web-em-java/31521>. Acesso em: 12 abr. 2018.

WOODS, Dan. **Building Microservices with Spring Boot**. Disponível em < https://www.infoq.com/articles/boot-microservices?utm_source=infoq&utm_campaign=user_page&utm_medium=link>. Acesso em: 10 abr. 2017.