UMA ABORDAGEM DE APLICAÇÃO AO AMBIENTE DE MICROSERVICES

Raniéri Geroldi[[1]](#footnote-1)

Orientador: Luis Antonio Schneiders[[2]](#footnote-2)

**Resumo:** A utilização de softwares e serviços tem se tornado cada vez mais comum no dia a dia das pessoas. Com esse aumento na demanda de software alguns requisitos como alta disponibilidade/desempenho, facilidade de atualização em ambiente de produção, evolução de tecnologias entre outros, se tornaram indispensáveis para empresas mantenedoras de software e serviços. Muitos desses requisitos se tornam complexos de administrar em grandes ambientes utilizando a convencional arquitetura monolítica, para grandes empresas essa complexidade pode se tornar desvantagem comercial. Para suprir esses requisitos de forma simplificada, surgiu a arquitetura de microservices, a qual consiste em desenvolver pequenos serviços independentes entre si, o que permite atingir os requisitos citados anteriormente com menor complexidade, além de oferecer novas vantagens sobre o modelo convencional. A adoção da arquitetura de microservices irá levar um novo paradigma para as equipes de desenvolvimento, o que nem sempre será vantajoso em todos os projetos, para cada novo projeto devem-se verificar seus requisitos e a partir disso definir ou não a utilização de microservices.

**Palavras-chave:** Microservices. Arquitetura de Software. Aplicação Monolítica. Sistemas distribuídos. Arquitetura orientada a serviços.

1. INTRODUÇÃO

A utilização de softwares no nosso dia a dia tornou-se cada vez mais comum, tanto no âmbito profissional quanto em nossa vida pessoal. Com a propagação da internet em escala global, novos serviços e aplicativos[[3]](#footnote-3)[[4]](#footnote-4)[[5]](#footnote-5) surgem a cada dia, com mais usuários e cada vez mais conectados[[6]](#footnote-6). Estas situações exigem que as empresas que disponibilizam tais serviços passem a se preocupar com requisitos como: disponibilidade, segurança da informação, alto desempenho, distribuição de conteúdo em larga escala, entre outros aspectos.

Não só a quantidade de softwares desenvolvidos aumentou, assim como, seus requisitos para atender novas demandas (J.G. Hall, J. Grundy, I. Mistrik, P. Lago, e P. Avgeriou, 2011). Novas exigências e requisitos de qualidade, complexidade, disponibilidade, entre outros, confrontam o modelo tradicional de desenvolvimento de software embasado em uma arquitetura monolítica.

A arquitetura monolítica apresenta diversas desvantagens para ambientes complexos (Sam Newman, 2015; Chris Richardson, 2016), entre as quais se destacam:

* dificuldade de escalar a aplicação;
* complexidade nos testes;
* dificuldade em evoluir as tecnologias utilizadas sem grandes impactos na aplicação;
* impossibilidade de tirar proveito do ambiente mais apropriado para cada módulo da aplicação.

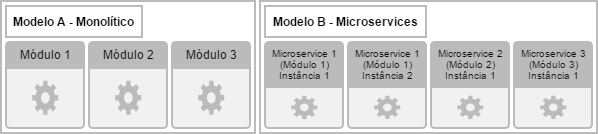
Em alguns casos, como o do Netflix[[7]](#footnote-7), utilizar uma arquitetura monolítica representaria diversos problemas ao gerenciamento do ambiente[[8]](#footnote-8). Não só a periodicidade das atualizações necessárias para manter o produto competitivo seria prejudicada, assim como a capacidade de evoluir as tecnologias utilizadas e de escalar a aplicação como fosse necessário.

Para atender às novas necessidades impostas ao desenvolvimento de software, muitas evoluções ocorreram ao passar dos anos, entre essas evoluções, está à arquitetura de microservices.

A arquitetura de microservices consiste em construir uma aplicação, como um conjunto de pequenos serviços independentes, onde cada serviço rode em seu próprio processo e efetuem comunicação utilizando protocolos WEB (LEWIS; FOWLER, 2014).

A partir das características propostas na arquitetura de microservices, os principais problemas associados às aplicações monolíticas serão resolvidos. A característica de se separar a aplicação em serviços independentes permite utilizar a tecnologia que for mais adequada a cada módulo, além de permitir escalar cada módulo independentemente dos outros. A realização de testes será facilitada, pois atualizações de código irão impactar somente ao módulo que pertencem.

Figura 1 - Aplicação monolítica x Aplicação microservices



Fonte: Do autor (2017).

Como visto no Modelo A da Figura 1, temos uma única aplicação com todos os módulos, sendo que quando é necessário escalar essa aplicação, replicamos toda ela para um novo ambiente, mesmo que um único ponto da aplicação, como por exemplo, o Módulo 1 seja o motivo do gargalo.

No Modelo B da Figura 1 utilizando a arquitetura de microservices, podemos observar que cada instância é responsável por um módulo separado de aplicação. Em caso de gargalo podemos replicar somente um serviço, como no exemplo o Microservice 1 (Módulo 1). Dessa forma tanto o processo de escalonamento da aplicação, quanto garantir redundância maior de determinadas partes, se tornam mais fáceis e exigem menos recursos de hardware, se comparado com aplicações monolíticas.

A utilização da arquitetura de microservices está se tornando cada vez mais comum no mercado, em alguns casos se tornando a arquitetura padrão para novos projetos. Apesar de sua ampla adoção, a utilização de microservices representa novos desafios, tanto para os responsáveis pelo desenvolvimento do software, quanto os responsáveis pela estrutura de rede que irá comportar essas aplicações.

1. MÉTODOS E MATERIAIS

Será desenvolvida e avaliada uma aplicação, utilizando a arquitetura de microservices e os componentes necessários para obter as seguintes características:

* alto desempenho/disponibilidade;
* autenticação centralizada;
* facilidade de testes;
* fácil evolução de tecnologias;
* possibilidade de utilização de vários stacks para desenvolvimento, conforme as necessidades de cada microservice;
* balanceamento de carga;
* facilidade de escalar a aplicação;
* conteinerização dos serviços (Docker).

Para a criação e desenvolvimento dos componentes, serão utilizados padrões e tecnologias livres adotados pelo mercado. O ambiente será construído em máquinas virtuais, utilizando a distribuição Linux CentOS[[9]](#footnote-9), contendo Docker[[10]](#footnote-10) para virtualizar os serviços necessários. Toda rede será configurada para simular uma situação real.

Será utilizado um SSO, para autenticação centralizada, serão desenvolvidos microservices e interfaces web para simulação de um sistema. Os microservices se registrarão no service registry, que irá disponibilizar as instâncias dos mesmos para o balanceamento de carga. Será implementado um gateway para requisições e respostas personalizadas de dispositivos como celulares, tablets, videogames, entre outros. Serão utilizados DNS e proxy reverso para simular condições reais.

Serão avaliadas as dificuldades de implementação da arquitetura de microservices, se comparado com a arquitetura monolítica, assim como suas vantagens e desvantagens. Entre os principais fatores a serem avaliados estão: alta disponibilidade, escalabilidade e segurança.

Por fim serão analisados cenários para implantação parcial ou total de microservices, levando em consideração integração com sistemas legados e migração de pontos de gargalo.

* 1. Tecnologias utilizadas

Para criação da arquitetura proposta neste artigo foram utilizadas as seguintes tecnologias:

* Java 1.8 (build 25.121-b13)/Spring Boot 1.5.1.RELEASE(Spring Cloud - Dalston.M1)/Netflix Eureka server/client;
* NodeJS 7.2.0/AngularJS 2.4.5/TypeScript 2.0.3/ES6;
* CentOS 7 x86\_64-Minimal-1611/Ubuntu 12.04.5;
* Docker 1.12.5(build 7392c3b)/Bind 9.9.4-RedHat-9.9.4-38.el7\_3/Nginx 1.10.2;
* MongoDB 3.0.14/MySQL 8.0/PostgreSQL 9.6.2.

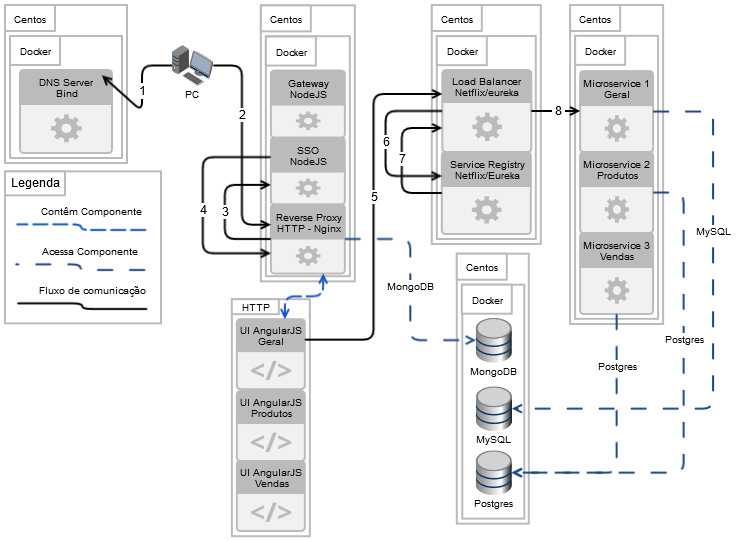
1. PROPOSTA E IMPLEMENTAÇÃO

A implementação do ambiente irá abordar em detalhes a configuração dos componentes específicos ao contexto de microservices, detalhes de configurações referentes a sistema operacional e banco de dados não serão discutidos.

A seguir será exibida a arquitetura implementada, especificando as tecnologias utilizadas em cada componente, assim como o ciclo de vida de uma requisição para um de seus serviços.

Foram implementados três microservices, cada um utilizando base de dados individuais, contendo APIS Rest, para consumo de serviços via HTTP. Para cada um dos microservices foi desenvolvida uma interface web, seguindo o modelo SPA (BIRDEAU et al., 2002).

Figura 2 – Modelo proposto



Fonte: Do autor (2017).

A numeração encontrada na Figura 2 representa em ordem crescente os passos de uma requisição efetuada por um usuário.

Como se pode observar na Figura 2, o usuário irá acessar uma aplicação web pertencente à estrutura implementada armazenada no servidor HTTP, http://geral.noface.com.br, a página web irá consumir serviços através de uma API Rest.

Após a consulta de DNS (1), o navegador redireciona para a aplicação solicitada (2), ao entrar na aplicação, o usuário será direcionado ao SSO (3), se o SSO possuir uma sessão válida o usuário será direcionado novamente para a página solicitada anteriormente, agora com um token de autorização (4). Caso o SSO não possuir uma sessão válida, o mesmo solicitara as credenciais para acesso ao sistema. O token fornecido pelo SSO será utilizado como forma de validar as credenciais do usuário ao consumir serviços das APIS Rest, o token deverá ser enviado a cada requisição de serviço.

Ao consumir serviços a aplicação web irá efetuar a requisição ao load balancer (5), o mesmo ao receber a requisição irá verificar a qual microservice a requisição pertence, após a definição do microservice, o load balancer irá consultar o service registry (6) para obter uma lista das instâncias ativas do microservice (7). A implementação do load balancer e service registry pertencem ao Netflix/eureka e foi configurada para apenas uma zona de disponibilidade.

Após receber a lista de instâncias ativas, o load balancer irá selecionar uma instância através do algoritmo de balanceamento de carga e enviar a requisição (8). Ao receber a requisição o microservice irá extrair o token de autenticação recebido e validar o token diretamente com o SSO, para evitar que tokens inválidos ou que já tenham expirado sejam utilizados.

No exemplo citado, não houve uso do gateway, o gateway será utilizado em casos em que a utilização dos serviços não ocorra por uma aplicação web, como por exemplo, apps de celular ou tablet ou em casos que o protocolo do serviço solicitado não seja web friendly. Neste caso, a única diferença no fluxo é que as requisições partirão do gateway para o load balancer.

* 1. Instalação Docker

Inicialmente será efetuada a instalação e configuração do Docker, onde todos os componentes da arquitetura serão instalados. O sistema operacional escolhido para utilização do Docker foi o Centos, serão seguidos os passos de instalação e configuração conforme site oficial.

Figura 3 - Instalação do aplicativo Docker

|  |
| --- |
| sudo yum install -y yum-utils  sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo  sudo yum makecache fast && sudo yum -y install docker-ce  sudo usermod -aG docker $USER && sudo systemctl enable docker && sudo systemctl start docker |

Fonte: Do autor (2017).

* 1. Instalação Bind e Nginx

Bind[[11]](#footnote-11) e Nginx[[12]](#footnote-12) serão instalados em containers Docker, utilizando volumes[[13]](#footnote-13) para facilitar a configuração pós-instalação. Nginx será utilizado como servidor HTTP e proxy reverso.

Figura 3 - Instalação e inicialização container BIND

|  |  |
| --- | --- |
| FROM centos  RUN yum -y install bind-utils bind && \  yum clean all  ADD container-image-root /  RUN rndc-confgen -r /dev/urandom -a -c /etc/rndc.key && \  chown named:named /etc/rndc.key && \  chmod 755 /entrypoint  EXPOSE 53/udp 53/tcp  VOLUME [ "/named" ]  ENTRYPOINT [ "/entrypoint"]  CMD [ "/usr/sbin/named" ] docker build --tag bind9 --rm . | rm -rf /Docker/volumes/bind9  mkdir -p /Docker/volumes/bind9/  chcon -Rt svirt\_sandbox\_file\_t /Docker/volumes/bind9/  docker run --detach --name bind9 \  --publish 53:53/udp \  --net=host \  --volume /Docker/volumes/bind9:/named \  bind9 |

Fonte: Do autor, adaptado de https://github.com/CentOS/CentOS-Dockerfiles/tree/master/bind/centos.

Figura 4 - Configuração BIND

|  |  |
| --- | --- |
| acl "trusted" {  192.168.241.0/24;  192.168.0.0/24;  };  options {  allow-query { localhost; trusted; };  }  include "/named/etc/named.conf.local"; | zone "noface.com.br" {  type master;  file "/named/etc/zones/db.noface.com.br";  };  zone "241.168.192.in-addr.arpa" {  type master;  file "/named/etc/zones/db.192.168.241";  }; |
| @ IN SOA ns1.noface.com.br. admin.noface.com.br. ()  IN NS ns1.noface.com.br.  154 IN PTR ns1.noface.com.br.  156 IN PTR apis.noface.com.br.  157 IN PTR sso.noface.com.br.  157 IN PTR geral.noface.com.br.  ... | @ IN SOA ns1.noface.com.br. admin.noface.com.br. ()  IN NS ns1.noface.com.br.  ns1.noface.com.br. IN A 192.168.241.154  apis.noface.com.br. IN A 192.168.241.156  sso.noface.com.br. IN A 192.168.241.157  geral.noface.com.br. IN A 192.168.241.157  ... |

Fonte: Do autor (2017).

Figura 5 - Instalação e inicialização container NGINX

|  |  |
| --- | --- |
| FROM centos  RUN yum -y update && yum -y install epel-release \  yum -y install nginx && yum clean all  COPY start.sh /start.sh  RUN mkdir /nginx && chmod 0755 /nginx && chown nginx:nginx /nginx  RUN rm /etc/nginx/nginx.conf &&  RUN rm -rf /etc/nginx/conf.d  RUN rm -rf /var/log/nginx  RUN ln -sf /nginx/nginx.conf /etc/nginx/nginx.conf  RUN ln -sf /nginx/conf.d /etc/nginx/conf.d  RUN ln -sf /nginx/logs /var/log/nginx  RUN ln -sf /nginx/www /var/www && chmod +x start.sh  VOLUME ["/nginx"]  EXPOSE 80  CMD ["/start.sh"] | docker build --tag nginx --rm .  rm -rf /Docker/volumes/nginx  mkdir -p /Docker/volumes/nginx  chcon -Rt svirt\_sandbox\_file\_t /Docker/volumes/nginx  mkdir /Docker/volumes/nginx/logs  mkdir /Docker/volumes/nginx/conf.d  mkdir /Docker/volumes/nginx/www  cp conf/nginx.conf /Docker/volumes/nginx/  cp conf/proxy.conf /Docker/volumes/nginx/conf.d/  docker run --detach --name nginx \  --publish 80:80 --net=host \  --privileged=true –volume \  /Docker/volumes/nginx:/nginx nginx |

Fonte: Do autor (2017).

Figura 6 - Configuração NGINX

|  |  |
| --- | --- |
| user nginx;  daemon off;  http {...;include /etc/nginx/conf.d/\*.conf;..;} | |
| server {  listen 80 default\_server;  server\_name sso.noface.com.br;  location / {  proxy\_set\_header Host $host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_pass http://apis.noface.com.br:7001;  }  } ... | server {  listen 80;  server\_name geral.noface.com.br;  root /var/www/test;  index index.html index.htm;  }  ... |

Fonte: Do autor (2017).

* 1. Sso – Oauth2

O SSO foi desenvolvido utilizando NodeJS[[14]](#footnote-14) e MongoDB[[15]](#footnote-15), baseado na RFC 6749. Foram desenvolvidas as seguintes funcionalidades necessárias ao ambiente:

* Cadastro de usuários;
* Cadastro de aplicações;
* Autenticação de credenciais;
* Autorização de acesso;
* Validação de token.

Toda aplicação que utilizará o SSO deverá ser cadastrada no mesmo, para ter permissão de autenticação. Ao efetuar o cadastro deverá ser informado nome da aplicação e URL de retorno (após a autenticação de credenciais do usuário, essa URL será utilizada para retornar a aplicação de origem). O sistema irá gerar automaticamente um id único e um segredo para a aplicação. Essas informações deverão ser enviadas ao SSO a cada requisição, a fim de garantir que somente aplicações autorizadas possam utilizar o serviço.

Ao acessar um recurso protegido, como por exemplo, http://geral.noface.com.br, primeiramente será efetuado redirecionamento ao SSO, para o endpoint de autorização, exemplo:

* http://sso.noface.com.br/sso/session/authorize?client\_id=XLIGwv1IxdGtFGM&response\_type=TOKEN&state=/cidade

Na URL de autorização é informado o id da aplicação cliente (client\_id), o tipo de resposta esperado (response\_type) e o estado em que a aplicação cliente se encontrava quando solicitou autorização (state).

Se o SSO verificar que não existe uma sessão válida, será efetuada a validação das credenciais usuário.

Após possuir uma sessão válida será efetuado o redirecionamento para aplicação cliente, utilizando a URL de retorno informada no cadastro da mesma. Exemplo:

* http://geral.noface.com.br/sso\_callback?token=b5bc8a40-164b-11e7-9689-45b653468645&username=someone&expires=43200&token\_ type=bearer&state=/cidade

Na URL de retorno é informado o token de autorização (token), usuário da sessão (username), tempo até expiração do token (expires), tipo do token de resposta (token\_type) e estado (state) parâmetro enviado pela própria aplicação cliente na requisição de autorização.

Após a aplicação cliente, receber a resposta do SSO, ela irá validar o token recebido para garantir que não houve nenhum tipo de adulteração. O token deverá ser enviado no corpo da requisição para o seguinte endpoint:

* http://sso.noface.com.br/sso/session/validate

Figura 7 - Exemplo do corpo da requisição para validação de sessão

|  |
| --- |
| { "token": "7608ded0-165a-11e7-9689-45b653468645" } |

Fonte: Do autor (2017).

A seguir configuração do container Docker, instalação e inicialização do SSO. A inicialização da aplicação é um script no arquivo de configuração package.json.

Figura 9 – Configuração e inicialização do SSO

|  |  |
| --- | --- |
| FROM node:latest  RUN mkdir -p /usr/src/app  WORKDIR /usr/src/app  COPY package.json /usr/src/app/  RUN npm install  COPY . /usr/src/app  EXPOSE 7001  CMD [ "npm", "start" ] | {  "name": "sso-noface",  "main": "app.js",  "scripts": {  "start": "node app.js --port=7001 --mode=PROD"  ...  }  } |

Fonte: Do autor (2017).

* 1. Aplicações WEB

As aplicações WEB foram desenvolvidas em AngularJS[[16]](#footnote-16), e servem de interface para o usuário consumir os endpoints disponibilizados pelos microservices, todas aplicações serão disponibilizadas pelo Nginx.

Para garantir que apenas usuários autorizados acessem os recursos, é verificado a cada troca de view e após consumir os endpoints se o token continua válido.

Para cada aplicação é definido um ponto de entrada para a mesma, neste caso AppComponent, nele serão monitoradas as trocas de estado das views, cada vez que houver inicio de navegação em uma nova view, será verificada a existência de uma sessão válida.

Figura 8 – Ponto de entrada aplicação AngularJS

|  |
| --- |
| @NgModule({...,bootstrap: [AppComponent]})  export class AppModule {}  export class AppComponent implements OnInit { ...  ngOnInit() {  this.router.events.pairwise().subscribe((event) => this.handleNavigation(event));  }  private handleNavigation(event) {  if ( navigationEvent instanceof NavigationStart ) {  this.session.handle(navigationEvent.url);  } }  ...; } |

Fonte: Do autor (2017).

Caso não houver uma sessão válida ou o usuário não estiver tentando acessar um recurso sem proteção, ocorre o redirecionamento ao SSO para autorização. Após a autorização do SSO, o token recebido pela aplicação cliente será validado, para garantir que não houve nenhum tipo de adulteração durante o transporte.

Figura 9 - Validação de sessão da aplicação

|  |
| --- |
| export class Session { ...;  handle(url:string):void{  if(this.isLoggedIn) return;  ...;  if(!UnsecuredUrls.contains(url)){  this.redirectToAuthorize(url);  }  }  validateToken(token:string):Promise<any> {  return this.http.post(this.ssoValidateTokenUrl,{token:token}).toPromise().then(res=>{  ...;this.isLoggedIn=true;...;  }).catch(err=>{  this.isLoggedIn=false;...; return Promise.reject(err);})}  ..; } |

Fonte: Do autor (2017).

Cada vez que é consumido um endpoint em um microservice, o token deve ser enviado para garantir que o usuário possui autorização.

Figura 10 - Envio do token de autorização ao consumir endpoint de um microservice

|  |
| --- |
| export function generateCustomRequestOptions(bodyData:any,token:String) : RequestOptions {  let headers:Headers=new Headers();  let options = {body:null,headers:headers};  headers.append('Authorization', 'Bearer '+token);  if(bodyData) options.body = bodyData;  return new RequestOptions(options);  } |

Fonte: Do autor (2017).

Caso o código de retorno de um endpoint for 403 (Forbidden), significa que o token fornecido pela aplicação foi adulterado ou foi invalidado durante a requisição. Nesse caso será efetuado logout de todas as aplicações utilizando aquele token.

* 1. Service registry – Eureka

Em uma aplicação monolítica, geralmente temos uma URL pela qual podemos utilizar todos os endpoints disponibilizados, em contrapartida em um ambiente de microservices temos uma URL para cada serviço e várias instâncias do mesmo serviço, sendo que cada qual pode ter sua própria URL.

Nesse cenário temos dois problemas principais: identificar diversas URL nas aplicações que irão utilizar os serviços e saber quais instâncias de cada serviço estão ativas.

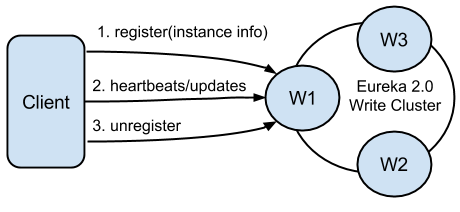
Para solucionar esses problemas, foi implementado o padrão de projeto service registry. Nele cada instância de microservice se registra em um ponto central, no momento em que o endpoint de um microservice for consumido, a aplicação irá consultar no service registry as URLs das instâncias ativas referentes ao microservice.

A implementação de service registry escolhida foi Eureka[[17]](#footnote-17), desenvolvida pelo Netflix. Um dos principais fatores a ser observado na implementação de um cluster utilizando service registry é sua abordagem perante o teorema CAP (BREWER, 1998). Observando a abordagem do Eureka, foi optado por alta disponibilidade devido às características do ambiente, dessa forma podem ocorrer casos onde os dados de cada nodo estão inconsistentes entre si. Para contornar esse problema é exigido balanceamento de carga adequado para as requisições dos usuários e mecanismos de falhas.

Eureka oferece possibilidade de diversas zonas de disponibilidade e possui um cliente para balanceamento de carga e controle de mecanismo de falhas chamado Ribbon[[18]](#footnote-18). Toda comunicação entre os componentes acontece por API Rest.

Após efetuar o registro o cliente continua enviando informações periódicas sobre o estado de sua instância, caso o cliente interrompa o envio dessas informações por determinado período de tempo, o service registry irá considerar essa instância inativa e remover de sua lista de consulta.

Figura 11 - Registro de cliente Eureka



Fonte: Netflix Eureka.

Figura 12 - Pacote de registro do cliente com o Eureka

|  |
| --- |
| POST /eureka/apps/API-GERAL-NOFACE HTTP/1.0  Host: service-registry.noface.com.br  X-Real-IP: 192.168.241.175  Connection: close  Content-Length: 975  Accept-Encoding: gzip  Content-Type: application/json  Accept: application/json  DiscoveryIdentity-Name: DefaultClient  DiscoveryIdentity-Version: 1.4  DiscoveryIdentity-Id: 192.168.241.175  User-Agent: Java-EurekaClient/v1.6.1  {"instance":{"instanceId":"192.168.241.175:api-geral-noface:7002","hostName":"apis-private.noface.com.br","app":"API-GERAL-NOFACE","ipAddr":"192.168.241.175","status":"UP","overriddenstatus":"UNKNOWN","port":{"$":7002,"@enabled":"true"},"securePort":{"$":443,"@enabled":"false"},"countryId":1,"dataCenterInfo":{"@class":"com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo","name":"MyOwn"},"leaseInfo":{"renewalIntervalInSecs":10,"durationInSecs":20,"registrationTimestam  p":0,"lastRenewalTimestamp":0,"evictionTimestamp":0,"serviceUpTimestamp":0},"metadata":{"@class":"java.util.Collections$EmptyMap"},"homePageUrl":"http://apis-private.noface.com.br:7002/","statusPageUrl":"http://apis-private.noface.com.br  :7002/info","healthCheckUrl":"http://apis-private.noface.com.br:7002/health","vipAddress":"api-geral-noface","secureVipAddress":"api-geral-noface","isCoordinatingDiscoveryServer":"false","lastUpdatedTimestamp":"1492030548028","lastDirtyTimestamp":"1492030549611"}} |

Fonte: Do autor (2017).

Figura 13 - Heartbeat cliente

|  |
| --- |
| PUT /eureka/apps/API-GERAL-NOFACE/192.168.241.175:api-geral-noface:7001?status=UP&lastDirtyTimestamp=1492030549914 HTTP/1.0  Host: service-registry.noface.com.br  X-Real-IP: 192.168.241.175  Connection: close  Content-Length: 0  DiscoveryIdentity-Name: DefaultClient  DiscoveryIdentity-Version: 1.4  DiscoveryIdentity-Id: 192.168.241.175  Accept-Encoding: gzip  User-Agent: Java-EurekaClient/v1.6.1 |

Fonte: Do autor (2017).

Eureka foi utilizado juntamente com Spring Boot[[19]](#footnote-19) e o módulo Spring Cloud[[20]](#footnote-20), para inicializar o serviço devemos indicar que se trata de uma aplicação Spring Boot e ativar o Eureka Server.

Figura 14 - Classe principal Eureka server

|  |
| --- |
| @EnableEurekaServer  @SpringBootApplication  public class ServiceRegistryApplication {  public static void main(String[] args) {  SpringApplication.run(ServiceRegistryApplication.class, args);  }  } |

Fonte: Do autor (2017).

Deve-se configurar a porta onde o serviço irá rodar, e desativar o registro com outro servidor. Outras configurações indicam a qual zona de disponibilidade o servidor pertence e o modo de execução produção/desenvolvimento.

Figura 15 - Configuração Eureka server

|  |
| --- |
| server.port=9001  eureka.datacenter=Principal  eureka.environment=prod  eureka.server.enableSelfPreservation=false  eureka.client.register-with-eureka=false  eureka.client.fetch-registry=false |

Fonte: Do autor (2017).

A instalação do serviço foi efetuada no Docker sobre uma máquina base com Java 8[[21]](#footnote-21) instalado.

Figura 16 - Configuração e inicialização Eureka com Docker

|  |  |
| --- | --- |
| FROM java8  RUN mkdir /usr/app  WORKDIR /usr/app  COPY service-registry-noface.jar /usr/app  COPY start.sh /usr/app  EXPOSE 7002  CMD [ "/usr/app/start.sh" ] | /usr/bin/java -jar service-registry-noface.jar |
| docker build -t service-registry . | docker run --name service-registry -p 7003:7002 -d ervice-registry |

Fonte: Do autor (2017).

* 1. Implementação Microservices

Os microservices foram criados utilizando Spring Boot com o servidor Undertow[[22]](#footnote-22). Todos os serviços criados são disponibilizados via API Rest. Cada microservice se comunica com uma base de dados diferente MySQL[[23]](#footnote-23) ou PostgreSQL[[24]](#footnote-24).

Os componentes foram escolhidos devido à baixa utilização de memória que o stack oferece. Em testes uma aplicação Spring Boot com mais de 3200 classes, utilizou entre memória heap e não heap 32MB[[25]](#footnote-25).

Para iniciar uma nova aplicação, basta indicar que a mesma será pertencente ao Spring Boot e configurar o servidor a ser utilizado no arquivo de dependências.

Figura 17 - Classe inicialização Spring Boot

|  |
| --- |
| @SpringBootApplication  public class ApiGeralNofaceApplication {  public static void main(String[] args) {  SpringApplication.run(ApiGeralNofaceApplication.class, args);  }  } |

Fonte: Do autor (2017).

Figura 18 - Configuração de dependências Undertow

|  |
| --- |
| buildscript { ext { springBootVersion = '1.5.1.RELEASE' } }  dependencies { compile("org.springframework.boot:spring-boot-starter-undertow:${springBootVersion}") } |

Fonte: Do autor (2017).

Para criação da API Rest, é necessário criar controllers e definir os métodos a serem expostos e seus parâmetros. No exemplo abaixo é exibido o método de deleção pertencente ao controller que gerencia as operações sobre entidade Pais, que representa a tabela Paises na base de dados.

Figura 19 - Controller Países

|  |
| --- |
| @RestController  @RequestMapping( "/paises" )  public class PaisController extends BaseController<Pais>{  @RequestMapping(method = RequestMethod.DELETE)  public T delete(@RequestBody T entity){...;}  } |

Fonte: Do autor (2017).

Existem diversas configurações de segurança envolvendo cada microservice. Inicialmente deve-se ativar a proteção contra ataques CSRF[[26]](#footnote-26) mesmo que sua aplicação for totalmente stateless, cookies ou tokens de autenticação podem ser usados para injeção de dados que comprometam a segurança. Mesmo que a aplicação use somente JSON[[27]](#footnote-27) para toda troca de dados, dependendo da forma em que a informação é interpretada pode haver risco a segurança.

Por padrão o Spring Boot não permite a requisição de domínios diferentes ao qual a aplicação está executando, devido ao fato de estarmos em um ambiente de microservices onde existem componentes executando em diversos domínios é necessário habilitar esta opção. No caso dos microservices apenas o balanceador de carga terá acesso direto aos endpoints, então o único domínio a ser liberado deve ser o dele.

Os microservices estão apenas expondo uma API Rest, e o controle de sessão fica a cargo do SSO, dessa forma deve-se configurar a política de criação de sessões no microservice como stateless. A sessão será validada no SSO a partir do token enviado pela requisição do usuário.

Figura 20 - Configuração de segurança base Spring Boot

|  |
| --- |
| @Configuration  @Order  public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  @Override  protected void configure(HttpSecurity http) throws Exception {  http.cors();  http.csrf().disable();  http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);  http.authorizeRequests().antMatchers("/\*\*").hasRole(CustomUserDetailsService.ROLE\_USER);  }  @Bean  CorsConfigurationSource corsConfigurationSource() {  CorsConfiguration configuration = new CorsConfiguration();  configuration.setAllowedOrigins(Arrays.asList("apis.noface.com.br"));  configuration.setAllowedMethods(Arrays.asList("GET","DELETE","PUT","POST"));  configuration.setAllowedHeaders(Arrays.asList("\*"));  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  source.registerCorsConfiguration("/\*\*", configuration);  return source;  } } |

Fonte: Do autor (2017).

Figura 21 - Validação do token com SSO e inserção do usuário autenticado no contexto da requisição

|  |
| --- |
| public class AuthorizationTokenFilter extends GenericFilterBean {  @Override  public void doFilter(...) {  ...  token=this.extractToken(authToken);  if(this.sessionRepository.validateToken(token)!=null){  UserDetails details = ...;  UsernamePasswordAuthenticationToken user = ...;  SecurityContextHolder.getContext().setAuthentication(user);  filterChain.doFilter(servletRequest, servletResponse);  } } } |

Fonte: Do autor (2017).

Figura 22 - Repositório responsável por validar o token com o SSO e manter lista de tokens válidos até serem invalidados pelo SSO

|  |
| --- |
| @Repository  public class SessionRepository {  public Session validateToken(String token){  if(this.isValidatedToken(token)) return this.getSession(token);  ...  Session session = this.restTemplate.postForObject(this.validateTokenUrl  ,this.generateValidationToken(token)  ,Session.class);  return session;  } } |

Fonte: Do autor (2017).

Para que a instância do microservice esteja disponível para balanceamento de carga, primeiramente é necessário efetuar o registro no service registry, as seguintes configurações são necessárias.

Figura 23 - Configuração para registro com server Eureka

|  |
| --- |
| spring.application.name=api-geral-noface  server.port=7002  eureka.instance.hostname=apis-private.noface.com.br  eureka.client.serviceUrl.defaultZone=http://service-registry.noface.com.br/eureka |

Fonte: Do autor (2017).

Microservices que possuem os mesmos serviços e devem realizar balanceamento de carga entre si devem possuir o mesmo nome, se estiverem distribuídos em máquinas diferentes podem utilizar a mesma porta, caso estiverem na mesma máquina deve ser executados em portas diferentes.

* 1. Load Balancer – Eureka/Ribbon

O load balancer foi implementado com base no cliente Ribbon para balanceamento de carga desenvolvido pelo Netflix para trabalhar em conjunto com o service registry Eureka. Além do balanceamento de carga, ele oferece funcionalidades como circuit break, retornos padrão em caso de falha, entre outras.

Toda comunicação com os microservices irá passar pelo load balancer, após a requisição chegar ao load balancer, ele irá consultar uma lista de instâncias disponíveis que podem responder a requisição solicitada. A lista de instâncias disponíveis pode estar em cache ou será feita a consulta no service registry, caso uma lista estiver em cache existem configurações para definir o tempo de validade da mesma.

Ribbon foi utilizado juntamente com Spring Boot e o módulo Spring Cloud, para inicializar o serviço devemos indicar que se trata de uma aplicação Spring Boot e habilitar a descoberta de clientes.

Figura 24 - Inicialização load balancer

|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  public class LoadBalancerApplication {  public static void main(String[] args) {  SpringApplication.run(LoadBalancerApplication.class, args);  }  } |

Fonte: Do autor (2017).

Deve-se configurar o endereço do service registry, pode se configurar várias zonas de disponibilidades, desta forma caso o service registry pare de responder o balanceador de carga poderá consultar em outro. No exemplo abaixo apenas a zona padrão foi configurada.

Figura 25 - Configuração load balancer

|  |
| --- |
| spring.application.name=load-balancer-noface  eureka.client.serviceUrl.defaultZone=http://service-registry.noface.com.br/eureka |

Fonte: Do autor (2017).

Registro do load balancer com service registry.

Figura 26 - Registro load balancer com server Eureka

|  |
| --- |
| POST /eureka/apps/LOAD-BALANCER-NOFACE HTTP/1.0  Host: service-registry.noface.com.br  X-Real-IP: 192.168.241.174  Connection: close  Content-Length: 947  Accept-Encoding: gzip  Content-Type: application/json  Accept: application/json  DiscoveryIdentity-Name: DefaultClient  DiscoveryIdentity-Version: 1.4  DiscoveryIdentity-Id: 192.168.241.174  User-Agent: Java-EurekaClient/v1.6.1  {"instance":{"instanceId":"192.168.241.174:load-balancer-noface:6002", "hostName":"192.168.241.174", "app":"LOAD-BALANCER-NOFACE", "ipAddr":"192.168.241.174", "status":"UP", "overriddenstatus":"UNKNOWN", "port":{"$":6002,"@enabled":"true"}, "securePort":{"$":443,"@enabled":"false"}, "countryId":1, "dataCenterInfo":{"@class":"com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo","name":"MyOwn"},"leaseInfo":{"renewalIntervalInSecs":30,"durationInSecs":90,"registrationTimestamp":0,"lastRenewalTimestamp":0,"evictionTimestamp":0,"serviceUpTimestamp":0},"metadata":{"@class":"java.util.Collections$EmptyMap"},"homePageUrl":"http://192.168.241.174:6002/","statusPageUrl":"http://192.168.241.174:6002/info","healthCheckUrl":"http://192.168.241.174:6002/health","vipAddress":"load-balancer-noface", "secureVipAddress": "load-balancer-noface", "isCoordinatingDiscoveryServer":"false", "lastUpdatedTimestamp":"1492033895142","lastDirtyTimestamp":"1492033896370"}} |

Fonte: Do autor (2017).

Para efetuar o balanceamento de carga é necessário indicar que todas as requisições Rest devem utilizar o algoritmo Round-robin[[28]](#footnote-28) ao escolher a lista de instâncias disponíveis.

Figura 27 - Configuração para utilização do algoritmo round-robin

|  |
| --- |
| @Configuration  public class RestTemplateConfig {  @LoadBalanced  @Bean  public RestTemplate restTemplate() {  return new RestTemplate();  }  } |

Fonte: Do autor (2017).

Exemplo de requisição efetuada por uma aplicação web.

Figura 28 – Requisição aplicação WEB para load balancer

|  |
| --- |
| POST /geral/pais/find HTTP/1.0  Host: apis.noface.com.br  X-Real-IP: 192.168.241.176  Connection: close  Content-Length: 40  User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:52.0) Gecko/20100101 Firefox/52.0  Accept: application/json, text/plain, \*/\*  Accept-Language: en-US,en;q=0.5  Accept-Encoding: gzip, deflate  Referer: http://geral.noface.com.br/pais  Content-Type: application/json  Authorization: Bearer 096c7d20-1fc6-11e7-bcdf-6d4eeec0370e  Origin: http://geral.noface.com.br  {"codigo": null,"nome": "brasil"} |

Fonte: Do autor (2017).

Figura 29 – Resposta load balancer para aplicação WEB

|  |
| --- |
| HTTP/1.0 200 OK  Expires: 0  Cache-Control: no-cache, no-store, max-age=0, must-revalidate  X-XSS-Protection: 1; mode=block  Pragma: no-cache  X-Frame-Options: DENY  Date: Wed, 12 Apr 2017 21:30:24 GMT  Connection: close  Access-Control-Allow-Origin: \*  Vary: Origin  X-Content-Type-Options: nosniff  Content-Type: application/json;charset=UTF-8  X-Application-Context: load-balancer-noface:6002  [{"id":27,"codigo":"BRA","nome":"Brasil"}] |

Fonte: Do autor (2017).

* 1. Análise e resultados do modelo proposto

A partir da implementação do modelo proposto, as desvantagens apresentadas pela arquitetura monolítica foram sanadas, além da obtenção das vantagens inerentes à arquitetura de microservices.

Devido ao fato de microservices serem processos independentes, cada serviço pode utilizar a tecnologia mais compatível com seu objetivo e ser instanciado no ambiente que se adequa a suas necessidades. Caso o serviço necessite de maior processamento (GPU), grande fluxo de IO ou alta uso de memória pode-se instanciar o mesmo no ambiente mais adequado. Existem serviços como Amazon EC2[[29]](#footnote-29), que oferecem ambientes específicos[[30]](#footnote-30) para as necessidades de cada tipo de serviço.

Outro ponto a se destacar é a possibilidade de se configurar um ambiente mais seguro, conforme modelo apresentado na Figura 2, os únicos componentes que precisam ser expostos ao acesso externo são o servidor HTTP e o load balancer.

Além de limitar a necessidade de expor componentes, cada um dos componentes pode ser configurado para responder requisições apenas para origens pré-determinadas, evitando assim que requisições não permitidas sejam executadas por terceiros mesmo que possuam um token de autenticação válido. O load balancer é configurado para aceitar requisições a partir do servidor HTTP ou microservices, por sua vez o service registry e o microservices são configurados para aceitar requisições apenas do load balancer.

1. Considerações finais

A arquitetura de microservices não só soluciona desvantagens referentes à arquitetura monolítica, assim como proporciona novas possibilidades para atingir os requisitos necessários a ambientes complexos que necessitam de alta disponibilidade/desempenho.

Microservices podem ser utilizados de forma parcial ou completa, em casos onde existem sistemas legados, novos módulos ou módulos que estejam causando gargalo na aplicação, podem ser migrados para serviços separados permitindo assim escalar os mesmos conforme a necessidade. Novas aplicações podem empregar a arquitetura desde sua concepção para facilitar desenvolvimento posterior.

Apesar de todas as vantagens apresentadas pela arquitetura de microservices, não existe bala de prata (Frederick P. Brooks, Jr., 1986) para solucionar os problemas em ambientes de alta complexidade. Com a adoção de microservices, a complexidade de configuração do ambiente aumenta com a adição de novos componentes não existentes em aplicações monolíticas.

A possibilidade de utilizar a linguagem de programação mais adequada para cada serviço acaba exigindo uma equipe mais qualificada. Em um ambiente de aplicação monolítica geralmente é utilizada apenas uma linguagem, sendo assim os desenvolvedores necessitam conhecer apenas o paradigma apresentado pela linguagem utilizada, padrões de código são mais fáceis de implantar e controlar e a contratação e treinamento de novos indivíduos para equipe é facilitada. Em um ambiente com diversas linguagens é necessários que a equipe lide com diversos paradigmas e o treinamento de novos indivíduos se torna um processo complexo, dessa forma podendo ser gerada grande dependência sobre um ou mais desenvolvedores.

Aspectos técnicos como modelagem da aplicação para que cada serviço seja coeso e transações distribuídas se tornam desafios muitas vezes não contemplados em uma aplicação monolítica.

Com novos requisitos surgindo a todo o momento para o desenvolvimento de aplicações, novas tecnologias, paradigmas e arquiteturas surgem para suprir esses requisitos, de forma que sempre se deve analisar com cuidado o que será aplicada em cada situação, buscando equilíbrio entre a solução ideal e a viabilidade de implementação.

**REFERÊNCIAS**

AVGERIOU P.; GRUNDY J.; HALL J.G.; LAGO P.; MISTRIK I. **Relating Software Requirements and Architectures.** 1. Ed. 3 agosto 2011. Springer Science & Business Media.

BOGNER, Justus; ZIMMERMANN, Alfred. **Towards integrating microservices with adaptable enterprise Architecture**. Disponível em: <<http://ieeexplore.ieee.org/document/7584392>>. Acesso em: 26 mar. 2017.

BOOTH, David et al. **Web services Architecture.** Disponível em: <<https://www.w3.org/TR/ws-arch/>>. Acesso em: 18 abr. 2017.

CHEMIN, Beatris F. **Manual da Univates para trabalhos acadêmicos:** Planejamento, elaboração e apresentação. 3. ed. Lajeado: Univates, 2015.

ERL, Thomas. **SOA: Principles of Service Design.** 1. Ed. 28 julho 2007. Prentice Hall.

FOWLER, Martin. **Microservices Resource Guide**. Disponível em: <<https://martinfowler.com/microservices>>. Acesso em: 15 mar. 2017.

GILBERT, Seth; LYNCH, Nancy. **A perspectives on the CAP theorem.** Disponível em: <<http://ieeexplore.ieee.org/document/6122006>>. Acesso em: 03 abr. 2017.

HARDT, Dick**,** Microsoft**. The OAuth 2.0 authorization framework**.Disponível em: <<https://tools.ietf.org/html/rfc6749>>. Acesso em: 29 mar. 2017.

IQBAL, M. Ashraf; SALTZ, Joel H.; BOKHARI, Shahid H. **Performance tradeoh's in static and dynamic load balancing strategies.** Disponível em: <<https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19860014876.pdf>>. Acesso em: 18 abr. 2017.

JR,. Frederick P. Brooks. **No Silver Bullet — Essence and Accident in Software Engineering.** Disponível em: <<http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf>>. Acessado em: 3 mai. 2017.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems.** 1. Ed. 20 fevereiro 2015. O'Reilly Media.

RICHARDSON, Chris. **Service Registry.** Disponível em: <<http://microservices.io/patterns/service-registry.html>>. Acesso em: 18 abr. 2017.

RICHARDSON, Chris. **Monolithic Architecture.** Disponível em: <<http://microservices.io/patterns/monolithic.html>>. Acesso em: 26 abr. 2017.

1. Bacharel em Curso de Redes de Computadores pelo Centro Universitário UNIVATES, Lajeado. [↑](#footnote-ref-1)
2. Professor do Centro Universitário UNIVATES, Lajeado. [↑](#footnote-ref-2)
3. "• Apple App Store: number of available apps 2017 | Statistic - Statista." <https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/>. Acessado em 25 abr. 2017. [↑](#footnote-ref-3)
4. "• Number of Google Play Store apps 2017 | Statistic - Statista." <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>. Acessado em 25 abr. 2017. [↑](#footnote-ref-4)
5. "25 Fastest growing cloud companies: Global 100 Software Leaders ...." <https://www.pwc.com/gx/en/industries/technology/publications/global-100-software-leaders/25-fastest-growing-cloud-companies.html>. Acessado em 25 abr. 2017. [↑](#footnote-ref-5)
6. "Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017 ...." 7 fev. 2017, <http://www.gartner.com/newsroom/id/3598917>. Acessado em 25 abr. 2017. [↑](#footnote-ref-6)
7. The Netflix Tech Blog: Netflix Conductor: A microservices orchestrator. 12 dez. 2016. <http://techblog.netflix.com/2016/12/netflix-conductor-microservices.html>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-7)
8. "Time to Move to a Four-Tier Application Architecture - NGINX." 6 fev. 2015, <https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture/>. Acessado em 26 abr. 2017. [↑](#footnote-ref-8)
9. CentOS.Disponível em: <https://www.centos.org/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-9)
10. Docker. Disponível em: <https://www.docker.com/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-10)
11. “Internet Systems Consortium." <https://www.isc.org/>. Acesso em: 13 abr. 2017. [↑](#footnote-ref-11)
12. "NGINX Wiki!." <https://www.nginx.com/resources/wiki/>. Acesso em: 13 abr. 2017. [↑](#footnote-ref-12)
13. "Manage data in containers - Docker Documentation." <https://docs.docker.com/engine/tutorials/dockervolumes/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-13)
14. "Node.js." <https://nodejs.org/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-14)
15. "MongoDB." <https://www.mongodb.com/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-15)
16. "AngularJS." <https://angularjs.org/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-16)
17. "Eureka 2.0 Architecture Overview · Netflix/eureka Wiki · GitHub." 31 dez. 2014, <https://github.com/Netflix/eureka/wiki/Eureka-2.0-Architecture-Overview>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-17)
18. "Home · Netflix/ribbon Wiki · GitHub." <https://github.com/Netflix/ribbon/wiki>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-18)
19. "Spring Boot - Projects." <https://projects.spring.io/spring-boot/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-19)
20. "Spring Cloud - Projects." 26 jan. 2016, <http://projects.spring.io/spring-cloud/spring-cloud.html>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-20)
21. "Java 8 Central - Oracle." <http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-21)
22. "Undertow." <http://undertow.io/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-22)
23. "MySQL." <https://www.mysql.com/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-23)
24. "PostgreSQL." 9 fev. 2017, <https://www.postgresql.org/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-24)
25. "Spring Boot Memory Performance spring.io." 10 dez. 2015, <https://spring.io/blog/2015/12/10/spring-boot-memory-performance>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-25)
26. "13. Cross Site Request Forgery (CSRF) - Spring." <http://docs.spring.io/autorepo/docs/spring-security/3.2.0.CI-SNAPSHOT/reference/html/csrf.html>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-26)
27. "JSON." <http://www.json.org/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-27)
28. "Getting Started · Client Side Load Balancing with Ribbon and Spring ...." 9 mar. 2016, <https://spring.io/guides/gs/client-side-load-balancing/>. Acesso em: 18 abr. 2017. [↑](#footnote-ref-28)
29. "EC2 - Amazon Web Services." <https://aws.amazon.com/pt/ec2/>. Acessado em 3 mai. 2017. [↑](#footnote-ref-29)
30. "Tipos de instância do EC2 – Amazon Web Services (AWS)." <https://aws.amazon.com/pt/ec2/instance-types/>. Acessado em 3 mai. 2017. [↑](#footnote-ref-30)