Balanceamento de Carga com NODE.JS/NGINX e DOCKER

Neste how-to, vamos ver como é fácil de balancear a carga *containerizada* com Docker aplicações NODE.JS com NGINX. Criaremos um aplicativo node.js simples que serve um arquivo HTML, como se fosse um web server (frontend), um container com o Docker e o seviço NGINX que usará o algoritmo round-robin para balancear entre duas instâncias em execução desse aplicativo node.js.

Docker e Containers

Como já explicamos, o Docker é uma plataforma de container de software. Os desenvolvedores usam o Docker para eliminar o problema "funciona na minha máquina" ao colaborar com colegas de trabalho. Isso é feito colocando partes de uma arquitetura de software em formato de container

Usando container, tudo o que é necessário para executar um software é empacotado em imagens isoladas. Diferentemente das Máquinas Virtuais (VMs), os containeres não agrupam um sistema operacional completo - são necessárias apenas bibliotecas e configurações necessárias para fazer o software funcionar. Isso os torna eficientes, leves, independentes e garante que o software sempre seja executado na mesma configuração, independentemente de onde foi implantado.

Instalando o Docker

Tudo o que precisamos para testar essa arquitetura é o Docker-CE, o NGINX e o NODE.JS instalados numa máquina virtual com Linux (sugiro o CentOS-7).

Para instalar o Docker já passamos as instruções em outro documento e para instalar o nginx e node.js no CentOS, basta digitar:

yum -y install nginx* node*
systemctl start nginx && systemctl enable nginx
systemctl start node && systemctl enable node

Criando o aplicativo Node.JS

Para mostrar o balanceamento de carga do NGINX em ação, criaremos um aplicativo Node.js. simples que serve um arquivo HTML estático. Depois disso, colocaremos esse aplicativo em contêiner e vamos executá-lo duas vezes. Por fim, configuraremos uma instância **NGINX** para despachar solicitações para ambas as instâncias de nosso aplicativo de forma balanceada

No final, poderemos acessar `http://localhost:8080` em nossa máquina local "aleatoriamente" obter resultados de uma ou outra instância. De fato, o resultado não será decidido aleatoriamente, configuraremos o NGINX para usar o algoritmo round-robin para decidir qual instância responderá a cada solicitação

Mas vamos dar um passo de cada vez. Para criar esse aplicativo, primeiro criaremos um diretório para o aplicativo e, em seguida, criaremos um arquivo `index.js` que responderá às solicitações HTTP

Para criar o diretório, vamos emitir o seguinte comando:
`mkdir application`
Depois disso, vamos criar o arquivo `index.js` neste diretório e colar o seguinte código-fonte:

```
var http = require('http');

var fs = require('fs');


http.createServer(function (req, res) {

  res.writeHead(200, {'Content-Type': 'text/html'});

  res.end(`<h1>${process.env.MESSAGE}</h1>`);

}).listen(8080);
```

Tudo o que esse script Node.JS faz é responder a solicitações HTTP com e uma tag HTML que contém uma mensagem definida pela variável

`MESSAGE`

Para entender melhor como isso funciona, podemos executar os seguintes comandos:

```
export MESSAGE=Servidor WEB com NodeJS

node index
```

E, em seguida, abra `http://localhost:8080` em um navegador da web. Viram? Temos uma página da web simples com a mensagem. Antes de prosseguir, paremos nosso aplicativo pressionando `CTRL+C`

Dockerizing os aplicativos Node.js.

Para *dockerizar* nosso aplicativo *Node.js* precisamos criar um arquivo chamado `Dockerfile` no diretório `application`

O conteúdo deste arquivo será:

```
FROM node

RUN mkdir -p /usr/src/app

COPY index.js /usr/src/app

EXPOSE 8080

CMD [ "node", "/usr/src/app/index" ]
```

Depois disso, precisamos criar uma imagem a partir disso `Dockerfile` o que pode ser feito através do seguinte comando:

```
docker build -t load-balanced-app .
```

E então podemos executar as duas instâncias do nosso aplicativo com os seguintes comandos:

```
docker run -e "MESSAGE=Primeira Instancia" -p 8081:8080 -d load-balanced-app
```

```
docker run -e "MESSAGE=Segunda Instancia" -p 8082:8080 -d load-balanced-app
```

Depois de executar os dois comandos, poderemos abrir as duas instâncias em um navegador da web acessando `http://localhost:8081` e `http://localhost:2`

O primeiro URL mostrará uma mensagem dizendo "Primeira instância", o segundo URL mostrará uma mensagem dizendo "Segunda instância".

Balanceamento de carga com uma instância NGINX encaixada

Agora que temos as duas instâncias de nosso aplicativo em execução em diferentes contêineres do Docker e respondendo em portas diferentes em nossa máquina host, vamos configurar uma instância do NGINX para carregar solicitações de equilíbrio entre elas. Primeiro, começaremos criando um novo diretório.

```
mkdir nginx-docker
```

Nesse diretório, criaremos um arquivo chamado `nginx.conf` com o seguinte código:

```
upstream my-app {

    server 172.17.0.1:8081 weight=1;

    server 172.17.0.1:8082 weight=1;

}

server {

    location / {

        proxy_pass http://my-app;

    }

}
```

Este arquivo será usado para configurar o NGINX. Nele, definimos um grupo de servidores (upstream) contendo os dois URLs que respondem pelas instâncias do nosso aplicativo. Ao não definir nenhum algoritmo específico para carregar solicitações de equilíbrio, estamos usando round-robin, que é o padrão no NGINX. Existem várias outras opções para carregar solicitações de equilíbrio com o NGINX, por exemplo, o menor número de conexões ativas ou o menor tempo médio de resposta .

Depois disso, definimos uma `server` propriedade que configura o NGINX para transmitir solicitações HTTP `http://my-app`, que é tratada pelo UPSTREAM definido anteriormente. Além disso, observe que codificamos `172.17.0.1` como o IP do gateway, este é o gateway padrão ao usar o Docker. Se necessário, você pode alterá-lo para atender à sua configuração local.

Agora vamos criar o `Dockerfile` que será usado para *containerizar o* NGINX com esta configuração. Este arquivo conterá o seguinte código:

```
FROM nginx

RUN rm /etc/nginx/conf.d/default.conf

COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Depois de criar os dois arquivos, agora podemos criar e executar o NGINX em contêiner no Docker. Conseguimos isso executando os seguintes comandos:

```
docker build -t load-balance-nginx .

docker run -p 8080:80 -d load-balance-nginx
```

Depois de emitir esses comandos, vamos abrir um navegador da web e acessar `http://localhost:8080`. Se tudo correu bem, veremos uma página da web com uma das duas mensagens: Primeira Instância ou Segunda Instância. Se pressionarmos recarregar em nosso navegador da Web algumas vezes, perceberemos que, de tempos em tempos, a mensagem exibida alterna entre Primeira Instância e Segunda Instância. Este é o algoritmo de balanceamento de carga round-robin em ação.

Neste exemplo subimos somente dois containers de webserver no formato node.js, porem, poderíamos ter subido muitos outros, como vimos em sala de aula. E essa é a idéia do container, possibilitar uma escalabilidade flexível tanto para aumentar como para diminuir uma topologia. Obviamente que quanto maior a topologia mais necessário algum tipo de automatização/orquestração e aí entraria o Kubernetes, OpenShift, etc.

Abaixo alguns comandos úteis ao se trabalhar com o DOCKER:

docker rm $(docker ps -aq)
docker rm <id container>
docker start <id container>
docker kill <id>
docker stop <id>
docker exec <id> -ti /bin/bash
docker ps -qa

```
docker rmi <image>
docker rmi -f $(docker images -aq)
docker run --name teste  -d  -p  8080:80  nginx
docker image list
docker image pull nginx
docker container start/stop <container>
docker container run -v /var/lib/teste1:/var ubuntu
```

Quando o contêiner for criado, podemos dizer ao Docker para montar um diretório local no host Docker para um diretório no contêiner. A imagem NGINX usa a configuração NGINX padrão, que usa **/usr/share/nginx/html** como diretório raiz do contêiner e coloca arquivos de configuração em **/etc/nginx**. Para um host Docker com conteúdo nos arquivos de diretório local **/var/www** e configuração em **/var/nginx/conf,**execute este comando (que aparece em várias linhas aqui apenas para legibilidade):

```
# docker run --name mynginx2 --mount type=bind
source=/var/www,target=/usr/share/nginx/html,readonly --mount
type=bind,source=/var/nginx/conf,target=/etc/nginx/conf,readonly -p
80:80 -d nginx

docker run --name webserver-1 --mount type=bind
source=/var/www,target=/usr/share/nginx/html,readonly --mount
type=bind,source=/var/nginx/conf,target=/etc/nginx/conf,readonly -p
8081:8080 -d load-balance-app

docker run --name webserver-2 --mount type=bind
source=/var/www,target=/usr/share/nginx/html,readonly --mount
type=bind,source=/var/nginx/conf,target=/etc/nginx/conf,readonly -p
8082:8080 -d load-balance-app

docker run --name webserver-3 --mount type=bind
source=/var/www,target=/usr/share/nginx/html,readonly --mount
type=bind,source=/var/nginx/conf,target=/etc/nginx/conf,readonly -p
8083:8080 -d load-balance-app
```

```
docker rm $(docker ps -aq)
docker rm <id container>
docker start <id container>
docker kill <id>
docker stop <id>
docker exec <id> -ti /bin/bash
docker ps -qa
docker rmi <image>
docker rmi -f $(docker images -aq)
docker run --name teste  -d  -p  8080:80  nginx
docker image list
docker image pull nginx
docker container start/stop <container>
docker container run -v /var/lib/teste1:/var ubuntu
```

service nginx start
chkconfig nginx on
nginx -v
service nginx status
pgrep nginx
ss -tlpn | grep -i :80
service iptables stop
chkconfig iptables off
hostnamectl set-hostname <new> (FOR AMI Linux 2)
vi /etc/sysconfig/network (for AMI Linux)

Conteúdo => /usr/share/nginx/html

User Guide => http://nginx.com/resources/admin-guide/load-balancer/

/etc/nginx/conf.d/loadbalancer.conf

```
server {
listen 80;
server_name lb.sublogic.net;
location / {
proxy_pass http://balanceador
 }
}
upstream balanceador {
server webserver-1.sublogic.net;
server webserver-2.sublogic.net;
}
```