**PROJETO TESTE PRÁTICO ADMINISTRADOR DE DADOS**

**AUTOR: CARLOS EDUARDO DA FONSECA MATHIAS**

**Docker Compose**

Antes de aprender sobre Docker devemos saber um pouco mais sobre sua história. No início o **Docker** era conhecido como **dotCloud**, uma empresa de **PaaS**(Platform as a Service), que tinha como função hospedar aplicações, fazendo tudo que é necessário, configurar e levantar servidor, controle de portas e etc. Hoje temos outras empresas de **PaaS** como, **Microsoft Azure**, **Heroku**, **Google Cloud Platform** e **Amazon Web Services**.

**Docker** teve seu lançamento em 20 de Março de 2013 e foi desenvolvido pela **dotCloud**. Uma plataforma Open Source escrita em **Go**,que é uma linguagem de programação desenvolvida dentro do **Google**, onde tem como função facilitar a criação e administração de ambientes isolados, os famosos **Containers**.

Dentro de cada container você pode ter uma aplicação, por exemplo, temos três containers, o primeiro tem um banco de dados em **mongoDB**, o segundo um servidor **nginx** e o terceiro container uma aplicação em **nodejs**. Porém para fazer toda a comunicação entre esses três containers temos que fazer tudo na mão, e pensando nesse problema surgiu o **Docker Compose**.

O **Docker Compose** é um orquestrador de containers. *E qual a função de um orquestrador?* Ele se comporta como uma banda ou orquestra, tocando em sincronia durante uma determinada apresentação musical.

Como já sabemos, somente com o **Docker** fica muito cansativo subir múltiplos containers, mas com o **Docker Compose** podemos subir múltiplos containers com apenas um único comando. O orquestrador depende de um arquivo de texto **YAML**(extensão **.yml**), dentro desse arquivo descrevemos tudo que queremos que aconteça para subir a nossa aplicação, como se fosse uma receita de bolo. E dessa forma não precisamos ficar subindo de um em um container, somente com um único comando subimos múltiplos containers e deixamos o orquestrador tocar a música.

Preparando os ingredientes

Primeiramente devemos criar uma conta no [**Docker hub**](https://www.docker.com/), entrando na página inicial, clique em **[Get Started](https://www.docker.com/get-started)**.

Em seguida devemos instalá-lo em nossa máquina, então:

* Para instalar o Docker no [Windows](https://docs.docker.com/docker-for-windows/install/)
* Para instalar o Docker no [MacOS](https://docs.docker.com/docker-for-mac/install/)
* Para instalar o Docker no LINUX

Em seguida instale o [**Docker Compose**](https://docs.docker.com/compose/install/) em sua máquina.

Verificando a versão do seu Docker

Depois de ter instalado o Docker dê o seguinte comando em seu terminal:

docker --version

A sáida deve ser parecida com essa:

Docker version 19.03.5, build 633a0ea838

# Verificando a versão do seu Docker Compose

Depois de ter instalado o Docker Compose dê o seguinte comando em seu terminal:

docker-compose --version

A sáida deve ser parecida com essa:

docker-compose version 1.23.2, build 1110ad0

# Fazendo login da sua máquina local com o Docker HUB

Antes de mais nada o que é Docker Hub? É um repositório do próprio Docker onde todas as imagens (imagens, como nodejs, mongodb, ubuntu e etc) estão armazenadas para utilizarmos em nossos projetos.

Para começar a brincar com o Docker precisamos realizar o login, então via terminal digite:

docker login

A saída deve ser:

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to [https://hub.docker.com](https://hub.docker.com/) to create one.  
Username: <digite-seu-username-do-docker>  
Password: <digite-sua-senha-do-docker>

Depois de inserir as informações necessárias, deverá exibir em seu terminal:

Login Succeeded

Pronto! Agora podemos começar a brincadeira :D

# Colocando a mão na massa!

## Criando nosso primeiro container

Como toda tecnologia na área de progromação, devemos dar o **Hello World.**Dizem que tem uma maldição do **hello world**, quando você não executa esse comando, nunca aprende uma determinada tecnologia… Então é melhor fazer!

Para criar nosso primeiro container devemos executar o seguinte comando:

docker run hello-world

Ao executarmos **docker run hello-world**, a primeira mensagem é:

Unable to find image 'hello-world:latest' locally latest: pulling from library/hello-world

Ou seja, o Docker não conseguiu encontar a imagem localmente, então ele foi em algum lugar (docker hub) e baixou a imagem.

No final do download da imagem, será exibido em seu terminal algo como:

Hello from Docker!This message shows that your installation appears to be working correctly.To generate this message, Docker took the following steps:  
 1. The Docker client contacted the Docker daemon.  
 2. The Docker daemon pulled the “hello-world” image from the Docker Hub.  
 (amd64)  
 3. The Docker daemon created a new container from that image which runs the  
 executable that produces the output you are currently reading.  
 4. The Docker daemon streamed that output to the Docker client, which sent it  
 to your terminal.To try something more ambitious, you can run an Ubuntu container with:  
 $ docker run -it ubuntu bashShare images, automate workflows, and more with a free Docker ID:  
 <https://hub.docker.com/>For more examples and ideas, visit:  
 <https://docs.docker.com/get-started/>

# Visualizando container ativos e inativos em sua máquina local

**Comando docker ps**: Serve para exibir todos os containers ativos em sua máquina local. Exemplo:

CONTAINER ID IMAGE COMMAND CREATED  
075ba11b95b7 mongo "docker-entrypoint.s…" 6 minutes ago  
2ddf6b997621 postgres "docker-entrypoint.s…" 5 days agoSTATUS PORTS NAMES  
Up 51 seconds 27017/tcp my-mongo  
Up 23 minutes 0.0.0.0:5432->5432/tcp my-postgres

**Comando docker ps -a**: Serve para exibir todos os containers inativos em sua máquina local. Exemplo:

CONTAINER ID IMAGE COMMAND CREATED  
285cg41j82c3 ubuntu "docker-entrypoint.s…" 13 minutes ago  
1gqf6a285289 mysql "docker-entrypoint.s…" 11 days agoSTATUS PORTS NAMES  
Up 51 seconds 5000/tcp my-ubuntu  
Up 23 minutes 3306/tcp my-mysql

# Baixando containers

**O comandodocker pull <imagem>**: Serve para baixar uma imagem do docker hub, porém sem executar.

**O comandodocker pull <username-docker>/<imagem>**: Serve para baixar uma imagem do docker hub de um usuário específico, porém sem executar.

**O comando docker run <imagem>**: Serve para baixar uma imagem do docker hub e executar em seguida.

**Comando docker run -it <imagem>**: Serve para executar e em seguida atrelar o seu terminal com o terminal do container. Logo após subir o container, seu terminal será liberado e ficará parecido com o exemplo abaixo:

root@ID\_CONTAINER:/#

# Executando containers

**O comando docker start <id>** ou **docker start <name>**: Serve para iniciar um container que está parado e baixado em sua máquina local.

**O comando docker start -a -i <id>** ou **docker start -a -i <name>**: Serve para iniciar um container que está parado e baixado em sua máquina local e atrelar em seu terminal. O parâmetro **-a**pesquisa todos os containers e integra os seus terminais, já o parâmetro **-i** habilita a interação entre os containers.

## Executando containers em segundo plano

**Os comandos docker start**ou **docker run** iniciam o container e trava o terminal e temos que abrir outro terminal para executar outros comandos. Para deixar o container executando em segundo plano e deixar o terminal livre, execute **docker start -d <id>** ou **docker run -d <id>**. O parâmetro **-d** tem como função liberar o terminal depois que o container for executado com êxito.

# Reinicializando containers

**O comando docker restart <id>** ou **docker restart <name>**: Serve para reinicializar um container que está parado e baixado em sua máquina local.

**O comando docker restart -a -i <id>** ou **docker restart -a -i <name>**: Serve para reinicializar um container que está em execução e atrelar em seu terminal.

# Parando containers

**O comando docker stop <id>** ou **docker stop <name>**: Serve para encerrar um container que está executando em sua máquina local.

## Encerrando container com tempo

**O comando docker stop -t<tempo> <id>** ou **docker stop -t<tempo> <name>**: Serve para encerrar um container que está executando em sua máquina local em de acordo com o tempo do parâmetro. Exemplo:

docker stop -t 10 66efdsg54821ss4

## Parando todos os containers

**O comando docker stop -t 0 $(docker ps -q)**: Serve para encerrar todos os containers que estão em execução na sua máquina local. O parâmetro **$(docker ps -q)**retorna todos os **IDs** e executa o comando **stop** em cada um, como se fosse um laço de repetição.

# Fazendo upload de containers

**O comando docker push <imagem>**: Serve para fazer upload de um container local para o docker hub, onde ficará em sua conta.

# Execluindo containers

**O comando docker rm <id>** ou **docker rm <name>**: Serve para excluir um container local.

**O comando docker container prune**: Serve para excluir todos os containers inativos em sua máquina local.

# Imagens locais

## Pesquisando

**O comando docker images**: Serve para todas as imagens em sua máquina local.

REPOSITORY TAG IMAGE ID CREATED SIZEbraum\_web latest eacb4b2e8a22 4 days ago 978MB  
postgres latest cf879a45faaa 7 days ago 394MB  
mongo latest 8e89dfef54ff 12 days ago 386MB  
ubuntu 18.04 ccc6e87d482b 3 weeks ago 64.2MB  
hello-world latest fce289e99eb9 13 months ago 1.84kB

## Removendo imagens

**O comando docker rmi <image\_id>**: Serve para excluir uma imagem em sua máquina local.

**OBS:** Repare que o comando tem um **rmi** que é destinado a excluir imagens, se por algum acaso você errar esse comando e colocar **rm** excluirá um container, tome cuidado.

# Atrelando o terminal com um container

Vamos utilizar como exemplo um container que tem uma imagem do ubuntu, execute:

docker start -it 66efdsg54821ss4

Logo em seguida o console vai mudar, ficando da seguinte maneira:

root@66efdsg54821ss4:/#

Atualizando o container com o console atrelado:

root@66efdsg54821ss4:/# apt-get update

Agora vamos criar um arquivo nesse container, execute:

root@66efdsg54821ss4:/# touch ola-docker.txt

Escrevendo dentro do arquivo:

root@66efdsg54821ss4:/# echo "docker é muito legal!" > ola-docker.txt

Exibindo o que tem dentro do arquivo:

root@66efdsg54821ss4:/# cat ola-docker.txt

Saída:

root@66efdsg54821ss4:/# cat ola-docker.txt  
docker é muito legal!

# Comunicação de containers

## Linkando containers automágicamente

**O comando docker run -d -P <imagem\_id>**: Serve para linkar a porta interna e a porta do container para gerar comunicação. O parâmetro **-P** atribui portas para comunicação automáticamente.

**Obs:** Com o comando **docker port <id>**ou **docker port <name>**podemos verificar quais portas o docker está usando.

## Linkando containers e passando portas como parâmetros

**O comando docker run -d -P 3333:5432 <imagem\_id>**: Serve para linkar a porta que queremos, ficando da seguinte maneira:

docker run -d -P <porta-do-pc>:<porta-do-container> <imagem\_id>

# Volumes no docker

Containers são voláteis, isto é, ao remover um, removemos os dados juntos. Para deixar os dados persistentes devemos usar **volumes**. Os volumes salvos **não ficam no container** **e sim no** **Docker Host**.

## Criando um container com caminho para volumes

**O comando docker run -v “caminho” <imagem>**: Serve para criar um volume com a imagem passado como parâmetro. Exemplo:

docker run -V "var/www" ubuntu

E podemos criar um link do container com a nossa máquina local, ficando da seguinte forma:

docker run -V "<caminho-pc>:/var/www" ubuntu

Dessa forma estamos dizendo para o container gravar informações em dois locais, primeiro será grado em “/var/www” e em seguida será copiado para o caminho"<caminho-pc>" .

# Docker inspect

Com o docker inspect conseguimos visualizar informações detalhadas do container e seu volume. Depois de ter criado o volume e copiado para algum caminho do seu PC, execute o comando a seguir:

docker inspect <id\_container>

Note que a saída do comando gera uma exibição JSON das configurações do container, imagem, volume e etc. Vá em **Mounts** e dentro das chaves verifique se está correto o **Source** e **Destination**.

**OBS:** Repare que dentro desse **JSON** é exibido várias **informações do container**, como **porta de comunicação**, **rede**, **volume**, **hosts** e etc. Todas essas informações serão úteis para você gerar comunicação entre os containers.

Uma vantagem de se usar volumes é que, imagine uma aplicação em NodeJS, mas não temos node instalado em nossa máquina. Com volumes podemos usar o Docker para executar os comandos em node, compilar e executar. Tudo isso utilizando volumes!

# Executando código e compilando via Docker

O comando para execução é **docker run -d -P 8080:3000 -V "<caminho-no-pc>/<aplicação>:/var/www" -w "/var/www" node npm start**.

Traduzindo o comando:

* O comando**docker run -d -P 8080:3000**diz: “docker, roda o container sem travar o terminal, e quero a porta **8080** do meu PC conectada com a porta **3000** do meu container”.
* Em seguida **-V "<caminho-no-pc>/<aplicação>:/var/www"**, diz: “Docker, link o volume do container **/var/www**com o volume da minha máquina local **<caminho-no-pc>/<aplicação>**”. Logo, ambos diretórios estão conectados!
* E por último é passado o comando **-w "/var/www" node npm start**, que diz: “Docker, inicie o work director **var/www**com a **imagem em node** e o comando **npm start**”. Dessa forma vai executar o comando **npm start** com a imagem em **node**, dentro de **/var/www** dentro do container.

**OBS:**O **-w**é o work director, dizemos em qual caminho executaremos tal comando.

Por fim, temos um exemplo do comando:

docker run -d -P 8080:3000 -V "C:\Users\user\Desktop\programa-teste:/var/www" -w "/var/www" node npm start

Você também pode executar o comando acima de outra forma! Vá até a pasta (via terminal) onde tem os arquivos em **Node**e execute:

**docker run -d -P 8080:3000 -V "$(pwd):/var/www" -w "/var/www" node npm start**

Com **$(pwd)** o docker vai pegar o diretório atual e linkar com **/var/www** do container. Não se esqueça, o **npm start** precisa estar dentro do **package.json***.*

# Conectando o banco de dados com a aplicação

Execute o comando:

docker run -d --name meu-mongo --network rede-prod mongo

Em seguida:

docker run -d -P 8080:3000 --name my-app --network rede-prod <img>

**Obs:** O banco de dados deve subir antes da aplicação. Para ativar o banco você deve bater pelo menos uma vez em um endpoint da sua app onde depende do banco de dados.

# Criando imagens

Para criação de imagens temos dois comandos principais:

1. **O comando docker build -f Dockerfile**: Cria a imagem apartir de um Dockerfile (vamos ver mais adiante como criar um Dockerfile).
2. **O comando docker build -f <caminho>/Dockerfile -t nome-user/nome-imagem**: Constrói e nomeia uma imagem não-oficial informando o caminho para o Dockerfile.

# Redes no docker

O Docker possuí uma **rede default** para a comunicação dos containers, porém usando vários containers pode gerar conflitos, em seguida explicarei com mais detalhes.

Imagens no Docker acessam a mesma rede, porém apenas através de IPs. Ao criar uma rede é possível realizar a comunicação entre os containers através do seu nome, além da segurança de evitar conflitos. Veremos também que durante a criação de uma rede precisamos identificar qual driver utilizaremos para comunição, geralmente, é o driver **bridge**.

## Interligando containers na prática

O docker tem uma **default network**, onde todos os containers se comunicam.

Executando o comando **inspect**do docker, podemos visualizar com mais detalhes. Um exemplo, seria:

"Networks": {  
 "bridge": {  
 "IPAMConfig": null,  
 "Links": null,  
 "Aliases": null,  
 -> "NetworkID": "7d045d9d8403105ab556d4",  
 -> "EndpointID": "4dc017b19d22d130af7bf180dd8e3662",  
 "Gateway": "172.17.0.1",  
 "IPAddress": "172.17.0.2",  
 "IPPrefixLen": 16,  
 "IPv6Gateway": "",  
 "GlobalIPv6Address": "",  
 "GlobalIPv6PrefixLen": 0,  
 "MacAddress": "02:42:ac:11:00:02",  
 "DriverOpts": null  
 }  
 }

O container está utilizando a rede default do docker!

## Testando comunicação entre containers

Para testar vamos baixar uma imagem do linux:

docker run -it ubuntu

Em seguinda execute:

root@ID\_CONTAINER# hostname -i

**OBS:** A saída desse comando será o IP desse container.

Agora levante outro container qualquer, dê um **inspect** nesse novo container, pegue o IP e **coloque no terminal do container atrelado** com o **ubuntu**. Em seguida execute o seguinte comando:

root@ID\_CONTAINER# apt-get update && apt-get install -y iputils-ping

Ele vai baixar as **ferramenstas necessárias** para o teste, no fim da instalação execute:

root@ID\_CONTAINER# ping <ip-do-outro-container>

No final o teste devera ficar parecido com:

root@ID\_CONTAINER# ping <ip-do-outro-container>PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.  
64 bytes from 172.17.0.2: icmp\_seq=1 ttl=64 time=0.125 ms  
64 bytes from 172.17.0.2: icmp\_seq=2 ttl=64 time=0.065 ms  
64 bytes from 172.17.0.2: icmp\_seq=3 ttl=64 time=0.063 ms  
64 bytes from 172.17.0.2: icmp\_seq=4 ttl=64 time=0.054 ms  
64 bytes from 172.17.0.2: icmp\_seq=5 ttl=64 time=0.060 ms  
64 bytes from 172.17.0.2: icmp\_seq=6 ttl=64 time=0.058 ms  
64 bytes from 172.17.0.2: icmp\_seq=7 ttl=64 time=0.053 ms  
64 bytes from 172.17.0.2: icmp\_seq=8 ttl=64 time=0.065 ms  
64 bytes from 172.17.0.2: icmp\_seq=9 ttl=64 time=0.057 ms  
64 bytes from 172.17.0.2: icmp\_seq=10 ttl=64 time=0.069 ms — — 172.17.0.2 ping statistics — -  
10 packets transmitted, 10 received, 0% packet loss, time 9203ms  
rtt min/avg/max/mdev = 0.053/0.066/0.125/0.022 ms

# Criando sua rede

Não é recomendável utilizar a rede default do Docker, porque pode ocorrer conflitos com o **Docker Compose**. Então, vamos criar nossa rede com o seguinte comando docker network create --driver bridge <nome-da-rede>:

docker network create --driver bridge prod

**OBS:** O parâmetro **--driver**indica qual o driver que será utilizado durante a criação da rede local. É possivel listar as redes em sua máquina local com o comando **docker network**.

## Subindo um container atrelado a nova rede criada:

docker run -it --name minha-app --network prod node

Em seguida faça um **inspect** no container **minha-app**. Veja em network que o container está atrelado a rede que criamos **prod**.

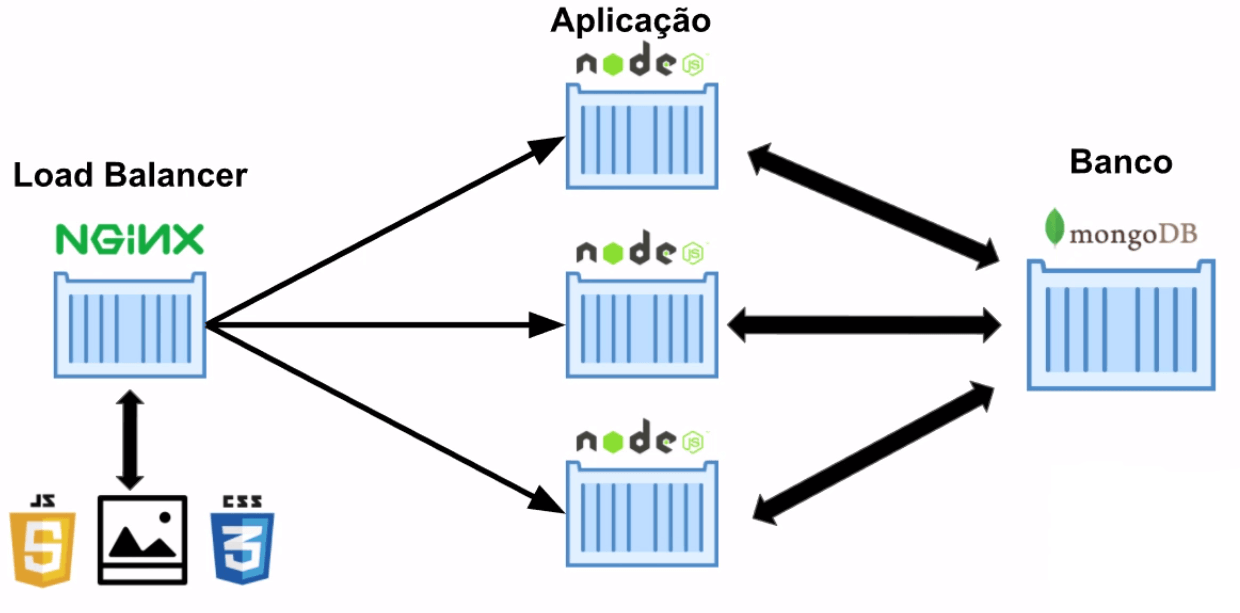
**OBS:**Por padrão os containers ficam na mesma rede com o nome **bridge**, e sua comunicação deve ser feita por IP. Na **rede default** do docker, só podemos realizar a comunicação utilizando IPs.

Imagine fazer para cada container milhares desses comandos acima? Cansativo não? Que tal chamar o orquestrador e realizar isso tudo somente com um único comando?

# Docker Compose

Agora não precisamos ficar executando muitos comandos no terminal sem necessidade, somente com um único comando e um arquivo **YAML** conseguimos subir tudo em sincronia!

Imagine que temos um servidor com três microsserviços e um banco de dados, como subir toda essa arquitetura? Como subir todos os container em sincronia?



Como podemos ver no exemplo acima, temos cinco containers, um para **Load Balancer**, três para **Aplicação** e um para o **Banco**. O container em nginx serve para distribuir arquivos estáticos, como imagens, vídeos, javascript e css. A aplicação fica responsável por toda a regra de negócio (ou lógica) e o banco para a persistência dos dados. Imagine só você ter que configurar cada um desses containers na mão? Docker compose vai nos ajudar com esse problema!

# Criando o Dockerfiles

O Dockerfile nada mais é do que **um meio que utilizamos para criar nossas próprias imagens**. Em outras palavras, ele serve para para construir um container, permitindo definir um ambiente personalizado e próprio para seu projeto pessoal ou empresarial. Exemplo Dockerfile do nginx:

FROM nginx:latest  
MAINTAINER will3g  
COPY /public /var/www/public  
COPY /docker/config/nginx.conf /etc/nginx/nginx.conf  
EXPOSE 8080 3333  
ENTRYPOINT ["nginx"]  
# Parametros extras para o entrypoint  
CMD ["-g", "daemon off;"]

* Com FROM nginx:latest dizemos para o Docker que queremos utilizar a última versão do nginx como base.
* Com o comando COPY /public /var/www/public copiamos o conteúdo da pasta public para dentro do container.
* O CMD ["-g", "daemon off;"] quer dizer para o docker executar o comando nginx passando os parâmetros **-g** e **daemon off**.

**Obs:** esse Dockerfile deve ficar dentro da pasta do nginx de sua aplicação!

# Criando um Ambiente utilizando Docker Compose!

Exemplo de arquivo **docker-compose.yml**:

version: '3'OBS: O "-" serve para dizer que isso é um array, podendo passar mais de um parâmetro.  
# Exemplo:  
#  
# ports:  
# - "3000:80"  
# - "80:80"  
#  
# Aqui estamos usando duas portas para o host, a 3000 e a 80services:  
 nginx:   
 build:   
 dockerfile: ./<caminho>/nginx.dockerfile   
 context: .   
 image: nginx   
 container\_name: nginx   
 ports:   
 - "80:80"   
 networks:   
 - production-network   
 depends\_on:   
 - "node1"   
 - "node2"   
 - "node3"mongodb:   
 image: mongo   
 networks:   
 - production-networknode1:   
 build:   
 dockerfile: ./<caminho>/arquivo.dockerfile   
 context: .   
 image: node  
 container\_name: braum  
 ports:   
 - "3000"   
 networks:   
 - production-network   
 depends\_on:   
 - "mongodb"node2:  
 build:  
 dockerfile: ./<caminho>/arquivo.dockerfile  
 context: .  
 image: node  
 container\_name: theodor  
 ports:  
 - "3001"  
 networks:  
 - production-network  
 depends\_on:  
 - "mongodb"  
   
 node3:  
 build:  
 dockerfile: ./<caminho>/arquivo.dockerfile  
 context: .  
 image: node  
 container\_name: parker  
 ports:  
 - "3002"  
 networks:  
 - production-network  
 depends\_on:  
 - "mongodb"  
networks:  
 production-network:  
 driver: bridge

## Explicando com detalhes

Primeiramente vamos começar com a primeira **chave-valor** no deparamos nesse arquivo docker-compose.

version: '3'

A cada alteração dentro desse arquivo a versão deve ser incrementada!

Logo em seguida temos os serviços, começaremos explicando o primeiro **nginx**:

**services:** # Declarando serviços  
 **nginx:** # Serviço nginx declarado  
 **build:** # Construa com base  
 **dockerfile: ./<caminho>/nginx.dockerfile** # Nessa receita  
 **context: .** # Comece a procurar apartir desse diretório  
 **image: nginx** # Execute a imagem do nginx  
 **container\_name: nginx** # O nome desse container é nginx  
 **ports:** # Execute o container nas portas  
 **- "80:80"** # 80 do meu PC e a porta 80 do container  
 **networks:** # Definindo redes  
 **- prod** # Execute na rede prod  
 **depends\_on:** # Esse serviço depende que os serviços... **- "node1"** # Esteja em execução **- "node2"** # Esteja em execução **- "node3"** # Esteja em execução

Em seguida temos o serviço do banco de dados:

**mongodb:** # Serviço mongodb declarado **image: mongo** # Execute a imagem do mongo **networks:** # Definindo redes **- prod** # Execute na rede prod

**OBS:** Repare que o serviço **mongodb** não dependede nenhum serviço, logo ele será o primeiro a subir!

Logo após, temos as aplicações em **node**:

**node1:** # Serviço node1 declarado  **build:**# Construa com base **dockerfile: ./<caminho>/arquivo.dockerfile** # Na receita **context: .** # Comece a procurar apartir desse diretório **image: node** # Execute a imagem do node **container\_name: braum** # O nome desse container é braum **ports:** # Execute o container nas portas **- "81:3000"** # 81 do meu PC e a porta 3000 do container **networks:** # Definindo redes **- prod** # Execute na rede prod **depends\_on:** # Essa aplicação depende **- "mongodb"** # Depende do database estar em execução**node2:** # Serviço node2 declarado  **build:**# Construa com base **dockerfile: ./<caminho>/arquivo.dockerfile** # Na receita **context: .** # Comece a procurar apartir desse diretório **image: node** # Execute a imagem do node **container\_name: theodor** # O nome desse container é theodor **ports:** # Execute o container nas portas **- "82:3001"** # 82 do meu PC e a porta 3000 do container **networks:** # Definindo redes **- prod** # Execute na rede prod **depends\_on:** # Essa aplicação depende **- "mongodb"** # Depende do database estar em execução **node3:** # Serviço node3 declarado  **build:**# Construa com base **dockerfile: ./<caminho>/arquivo.dockerfile** # Na receita **context: .** # Comece a procurar apartir desse diretório **image: node** # Execute a imagem do node **container\_name: parker** # O nome desse container é parker **ports:** # Execute o container nas portas **- "83:3002"** # 83 do meu PC e a porta 3000 do container **networks:** # Definindo redes **- prod** # Execute na rede prod **depends\_on:** # Essa aplicação depende **- "mongodb"** # Depende do database estar em execução

E por fim temos a criação da **network**:

**networks:** # Definindo a rede para comunicação dos containers  
 **prod:** # Aqui criamos uma nova rede chamada prod  
 **driver: bridge** # Definimos que a rede utiliza o driver bridge

# Construindo a partir da receita

Antes de executar o docker compose, precisamos “buildar” as instruções do arquivo **docker-compose.yml**.

* Entre na pasta onde o docker-compose.yml se encontra.
* Execute o comando no terminal para construir as instruções:

docker-compose build

# Comandos docker compose

* Subindo os containers com um único comando!

docker-compose up

* Parando os containers com um único comando!

docker-compose down

* Reinicializando os containers com um único comando!

docker-compose restart

* Exibindo os serviços que estão em execução com um único comando!

docker-compose ps

* Exibindo os serviços que estão parados com um único comando!

docker-compose ps -a

# Resumão dos comandos

## Docker HUB

**docker login**: Inicia o login com o docker.

**docker push <user/imagem>**: Envia a imagem para o docker hub de um usuário específico.

**docker pull <user/imagem>**: Baixa a imagem do docker hub de um usuário específico.

## Informações

**docker --version**: Exibe a versão do docker.

**docker inspect ID\_CONTAINER**: Exibe as informações do container.

**docker ps**: Exibe os containers ativos.

**docker ps -a**: Exibe os containers inativos.

## Execução

**docker run <imagem>**: Baixa, cria a imagem e executa o container com a mesma.

**docker run -it <imagem>**: Baixa, cria a imagem e executa o container com a mesma, porém executa o terminal iterativo para configurar o container.

**docker run -d -p --name <nome-container> <imagem>**: Baixa, cria a imagem e executa o container com a mesma e atribuií um nome para o container.

**docker run -d -p 1234:8080 --name <nome-container> <imagem>**: Baixa, cria a imagem, executa o container com a mesma, atribuií um nome para o container e liga a porta do PC 1234 com a porta 8080 do container.

**docker run -V "caminho-volume" <imagem>**: Baixa, cria a imagem e executa o container com a mesma e cria um volume no respectivo caminho do container.

**docker run -it --name <nome-container> --network <nome-rede> <imagem>**: Baixa, cria a imagem e executa o container com a mesma na network **<nome-rede>**e no final da execução do container o terminal hierático é liberado para a configuração do container.

## Inicialização, reinicialização e encerramento

**docker start <id\_container>**: Inicia o container.

**docker start -a -i <id\_container>**: Inicia o container e em seguida exibe o modo iterativo para a configuração do container.

**docker restart <id\_container>**: Reinicia o container.

**docker stop <id\_container>**: Encerra o container.

## Remoção

**docker rm <id\_container>**: Remove o container.

**docker rmi <id\_image>**: Remove a imagem.

**docker container prune**: Remove todos os containers inativos.

## Construção dockerfile

**docker build -f Dockerfile**: Cria uma imagem apartir de um dockerfile.

**docker build -f Dockerfile -t <user/imagem>**: Cria e nomeia uma imagem apartir de um dockerfile, não-oficial.

**docker build -f Dockerfile -t <user/imagem> “caminho/dockerfile”**: Cria e nomeia uma imagem a partir de um dockerfile, não-oficial apartir de um caminho.

## Relaciado à rede

**hostname -i**: Exibe o IP do container.

**docker network create --driver bridge <nome-rede>**: Cria uma rede especificando o driver desejado.

## Compose

**docker-compose up**: Executa os containers.

**docker-compose down**: Encerra os containers.

**docker-compose ps**: Exibe os containers em execução.

**docker-compose ps -a**: Exibe os containers inativos.

**docker exec -it <nome-container> ping <serviço-x>**: Executa o comando **ping** para testar se a comunicação entre os dois containers está funcionando.

**FIM**