

# Notas sobre o Docker

Lucas Caramês

April 16, 2022

## 1 Containers

### 1.1 Rodando um container no Docker

- Utilizar o comando  
`docker run`
- Neste comando podemos passar diversos parâmetros;
- Neste exemplo, vamos passar apenas o nome da imagem que é  
`docker/whalesay`
- Um comando chamado `cowsay` e uma mensagem;

### 1.2 Sobre Containers

- Def. Um pacote de código que pode executar uma ação;
  - Ex. Rodar uma aplicação em node.js, PHP, Python e etc.
- Nossos projetos serão executado dentro dos containers que criamos/utilizamos;
- Containers utilizam imagens para serem executados;
- Múltiplos containers podem rodar juntos;
- Imagem e container são recursos fundamentais do docker;
- Imagem é o “projeto” que será executado pelo container, todas as instruções estarão declaradas nela;
- Container é o docker rodando alguma imagem, consequentemente, executando algum código proposto por ela;
- O fluxo é: programamos uma imagem e executamos por meio de um container;

### 1.3 Onde encontrar as imagens?

- Vamos encontrar as imagens no repositório do Docker;
- Neste site podemos verificar quais imagens existem da tecnologia que estamos procurando, por exemplo, node.js e também aprender como utilizá-las;
- Vamos executar uma imagem em um container com o comando  
`docker run <imagem>`

## 1.4 Verificar containers executados

- O comando  
`docker ps`  
  
ou  
`docker container ls`  
  
exibe quais containers estão sendo executados no momento;
- Utilizando a flag  
`-a`  
  
, temos também todos os containers já executados na máquina;
- Este comando é útil para entender o que está sendo executado e acontece no nosso ambiente;

## 1.5 Executar container com iteração

- Podemos executar um container e deixá-lo executando no terminal;
- Para isso, basta utilizarmos a flag  
`it`  
  
;
- Desta maneira, podemos executar comandos disponíveis no container que estamos utilizando o comando `run`;
- Podemos utilizar a imagem do ubuntu para isso, como exemplo;

## 1.6 Container x Virtual Machine

- Container é uma aplicação que serve para um determinado fim, não possui sistema operacional, seu tamanho é de alguns MBs;
- VM possui sistema operacional próprio, tamanho de GBs, pode executar diversas funções ao mesmo tempo;
- Containers acabam gastando menos recursos para serem executados, por causa de seu uso específico;
- VMs gastam mais recursos, porém podem exercer mais funções;

## 1.7 Executar container em background;

- Quando iniciamos um container que persiste, ele fica ocupando o terminal;
- Podemos executar um container em background, para não precisar ficar com diversas abas do terminal aberto, utilizamos a flag  
`-d`  
  
(detached);
- Verificamos containers em background com

```
docker ps
```

também;

- Podemos utilizar o nginx para este exemplo;
- `docker run nginx`
- Para parar um container docker, executamos o código abaixo no terminal  
`docker stop <nome-do-container>`

## 1.8 Expondo porta de container

- Os containers de docker não possuem conexão com nada de fora deles;
- Por isso, precisamos expor portas, a flag é a

```
-p
```

e podemos fazer assim:

```
-p 80:80
```

```
;
```

- Desta maneira o container será acessível na porta 80;
- Podemos testar este exemplo com o nginx;

-p a:b

onde a-> é a porta que estou expondo no pc

b-> é a porta que desejo receber do container

## 1.9 Reiniciando um container

- Aprendemos a parar um container usando o

```
docker stop
```

, para voltar a rodar um container podemos usar o comando

```
docker start <id>
```

- Lembre-se que o run sempre cria um novo container;
- Então, caso seja necessário aproveitar um antigo, opte pelo start;

## 1.10 Definindo nome do container

- Podemos definir um nome do container com a flag

```
—name
```

- Se não colocamos, recebemos um nome aleatório, que pode ser um problema para uma aplicação profissional;
- A flag é inserida junto do comando run;

— ex.

```
docker run —name nome_container
```

### 1.11 Verificando os logs

- Podemos verificar o que aconteceu em um container com o comando logs;
- Utilizamos da seguinte maneira:  

```
docker logs <id>
```
- As últimas ações realizadas no container serão exibidas no terminal;
- A flag  

```
-f
```

  
exibe continuamente os logs do container. Aperte Ctrl+C para parar de seguir;

### 1.12 Removendo containers

- Podemos remover um container de uma máquina que estamos executando o Docker;
- O comando é  

```
docker rm <id>
```
- Se o container estiver rodando ainda, podemos utilizar a flag  

```
-f
```

  
(force);
- O container removido não é mais listado em  

```
docker ps -a
```

## 2 Imagens

### 2.1 O que são imagens?

- Imagens são originadas de arquivos que programamos para que o Docker crie uma estrutura que execute determinadas ações em containers;
- Elas contém informações como: imagens base, diretório base, comando a serem executados, porta da aplicação, etc;
- Ao rodar um container baseado na imagem, as instruções serão executadas em camadas;

### 2.2 Como escolher uma boa imagem?

- Podemos fazer download das imagens;
- Porém qualquer um pode fazer upload de uma imagem, e isso pode ser um problema;
- Devemos então nos atentar às imagens oficiais;
- Outro parâmetro interessante é a quantidade de downloads e a quantidade de stars;

## 2.3 Criando uma imagem

- Para criar uma imagem vamos precisar de um arquivo: Dockerfile em uma pasta que ficará o projeto;
- Este arquivo vai precisar de algumas instruções para poder ser executado;
- FROM: imagem base;
- WORKDIR: diretório de aplicação;
- EXPOSE: Porta da aplicação;
- COPY: Quais arquivos precisam ser copiados;

## 2.4 Executando uma imagem

- Para executar a imagem, primeiramente, vamos precisar fazer o build;
- O comando é  
`docker build <diretório-da-imagem>`
- Depois vamos utilizar o  
`docker run <imagem>`  
para executá-la;
- Cria uma cache p/ cada camada;
- Tem de especificar a porta que será acessada, para ter acesso ao container;

## 2.5 Alterando uma imagem

- Sempre que alteramos o código de uma imagem vamos precisar fazer o build novamente;
- Para o Docker é como se fosse uma imagem completamente nova;
- Após fazer o build, vamos executá-la pelo o outro id único criada com o  
`docker run`

## 2.6 Camadas das imagens

- As imagens são divididas em camadas (layers);
- Cada instrução no Dockerfile representa um layer;
- Quando algo é atualizado, apenas as layers depois da linha atualizada são refeitas;
- O resto permanece em cache, tornando o build mais rápido;

## 2.7 Download de imagens

- Podemos fazer o download de imagem do hub e deixá-la disponível em nosso ambiente ;
- Vamos utilizar o comando  
`docker pull <imagem>`
- Desta maneira, caso use em outro container, a imagem já estará pronta para ser utilizada;

## 2.8 Aprender mais sobre os comandos

- Todo comando no Docker tem acesso a uma flag  
—help
- Utilizando desta maneira, podemos ver todas as opções disponíveis nos comandos;
- Para lembrar algo ou executar uma tarefa diferente com o mesmo;
  - Ex.  
`docker run —help`

## 2.9 Múltiplas aplicações, mesmo container

- Podemos inicializar vários containers com a mesma imagem;
- As aplicações funcionarão em paralelo;
- Para testar isso, podemos determinar uma porta diferente para cada uma, e rodar no modo detached;

## 2.10 Alterando o nome da imagem e a tag

- Podemos nomear a imagem que criamos;
- Vamos utilizar o comando  
`docker tag <nome>`  
para isso;
- Também podemos modificar a tag, que seria como uma versão da imagem, semelhante ao git;
- Para inserir a tag, utilizamos:  
`docker tag <nome>:<tag>`

## 2.11 Iniciando imagem com um nome

- Podemos nomear a imagem já na sua criação;
- Vamos usar a flag  
—t
- É possível inserir o nome e a tag na sintaxe: nome:tag;
- Isso torna o processo de nomeação mais simples;

## 2.12 Comando start interativo

- A flag  
`-it`

pode ser utilizada com o comando `start` também;

- Ou seja, não precisamos criar um novo container para utilizá-lo no terminal;
- O comando é:  
`docker start -it <container>`

## 2.13 Removendo imagens

- Assim como nos containers, podemos remover imagens com um comando;
- Ele é o  
`docker rmi <imagem>`
- Imagens que estão sendo utilizadas por um container, apresentarão um erro no terminal;
- Podemos usar a flag  
`-f`  
para forçar a remoção;

## 2.14 Removendo imagens e containers

- Com o comando  
`docker system prune`
- Podemos remover imagens, container e networks não utilizados;
- O sistema irá exigir uma confirmação para realizar a remoção;

## 2.15 Removendo container após utilizar

- Um container pode ser automaticamente deletado após sua utilização;
- Para isso, vamos utilizar a flag  
`—rm`
- O comando seria:  
`docker run —rm <container>`
- Desta maneira economizamos espaço no computador e deixamos o ambiente mais organizado;

## 2.16 Copiando arquivos entre containers

- Para cópia de arquivos entre containers utilizamos o comando:  
`docker cp`
- Pode ser utilizado para copiar um arquivo de um diretório para um container;
- Ou de um container para um diretório determinado;

## 2.17 Verificar informações de processamento

- Para verificar dados de execução de um container utilizamos:  
`docker top <container>`
- Desta maneira temos acesso a quando ele foi iniciado, id do processo, descrição do comando CMD;

## 2.18 Verificar dados de um container

- Para verificar diversas informações como: id, data de criação, imagem e muito mais;
- Utilizamos o comando  
`docker inspect <container>`
- Desta maneira conseguimos entender como o container está configurado;

## 2.19 Verificar Processamento

- Para verificar os processos que estão sendo executados em um container, utilizamos o comando:  
`docker stats`
- Desta maneira temos acesso ao andamento do processamento e memória gasta pelo mesmo;

## 2.20 Autenticação no Docker Hub

- Para concluir esta aula, vamos precisar criar uma conta no Docker Hub;
- Para autenticar-se pelo terminal, vamos utilizar o comando  
`docker login`
- E então inserir o usuário e senha;
- Agora podemos enviar nossas próprias imagens para o HUB;

## 2.21 Logout no Docker HUB

- Para remover a conexão entre nossa máquina e o docker HUB, vamos utilizar o comando  
`docker logout`
- Agora não podemos mais enviar imagens, pois não estamos autenticados;



## 2.22 Enviando imagem para o Docker HUB

- Para enviar uma imagem nossa ao Docker HUB utilizamos o comando  
`docker push <imagem>`
- Porém, antes vamos precisar criar um repositório para a mesma no site do HUB;
- Também será necessário estar autenticado;

## 2.23 Enviando atualização de imagem

- Para enviar uma atualização vamos primeiramente fazer o build;
- Trocando a tag da imagem para a versão atualizada;
- Depois vamos fazer o push novamente para o repositório;
- Assim, todas as versões estarão disponíveis para serem usadas;

## 2.24 Baixando e utilizando a imagem

- Para baixar a imagem podemos utilizar o comando  
`docker pull <imagem>`
- E depois criar um novo container com  
`docker run <imagem>`
- E pronto! Estaremos utilizando a nossa imagem com um container;

# 3 Volumes

## 3.1 O que são volumes?

- Uma forma prática de persistir dados em aplicações e não depender de containers para isso;
- Todo dado criado por um container é salvo nele, quando o container é removido, perdemos os dados;
- Então precisamos dos volumes para gerenciar os dados e também conseguir fazer backups de forma mais simples;

## 3.2 Tipos de volumes

- Anônimos (anonymous):
  - diretórios criados pela flag -v, porém com um nome aleatório;
- Nomeados (named):
  - São volumes com nomes, podemos nos referir a estes facilmente e saber para que são utilizados no nosso ambiente;
  - Bind Mounts: Uma forma de salvar dados na nossa máquina, sem o gerenciamento do docker, informamos um diretório para este fim;

### 3.3 O problema da persistência

- Se criamos um container com alguma imagem, todos os arquivos que geramos dentro dele serão do container;
- Quando o container for removido, perderemos estes arquivos;
- Por isso, precisamos dos volumes;

### 3.4 Volumes anônimos

- Podemos criar um volume anônimo (anonymous) da seguinte maneira:

```
docker run -v /data
```

- Onde /data será o diretório que contém o volume anônimo;
- E este container estará atrelado ao volume anônimo;
- Com o comando

```
docker volume ls
```

podemos ver todos os volumes do nosso ambiente;

### 3.5 Volumes nomeados

- Podemos criar um volume nomeado (named) da seguinte maneira:

```
docker run -v nomedovolume:/data
```

- Agora o volume tem um nome e pode ser facilmente referenciado;
- Em

```
docker volume ls
```

podemos verificar o container nomeado criado;

- Da mesma maneira que o anônimo, este volume tem como função armazenar arquivos;

### 3.6 Bind mounts

- Bind mount também é um volume, porém ele fica em diretório que nós especificamos;
- Então não criamos um volume e sim, apontamos um diretório;
- O comando para criar um bind mount é:  

```
docker run /dir/data:/data
```
- Desta maneira o diretório /dir/data no nosso computador, será o volume deste container;

### 3.7 Atualização de projeto com bind mount

- Bind mount não serve apenas para volumes;
- Podemos utilizar esta técnica para atualização em tempo real do projeto;
- Sem ter que refazer o build a cada atualização do mesmo;

### 3.8 Criar um volume

- Podemos criar volumes manualmente também;
- Utilizamos o comando:  
`docker volume create <nome>`
- Desta maneira temos um named volume criado, podemos atrelar a algum container na execução do mesmo;

### 3.9 Listando todos os volumes

- Com o comando:  
`docker volume ls`  
listamos todos os volumes;
- Desta maneira temos acesso aos anonymous e os named volumes;
- Interessante para saber quais volumes estão criados no nosso ambiente;

### 3.10 Checar um volume

- Podemos verificar os detalhes de um volume em específico com o comando:  
`docker volume inspect nome`
- Desta forma temos acesso ao local em que o volume guarda dados, nome, escopo e muito mais;
- O docker, os dados dos volumes em algum diretório do nosso computador, desta forma podemos saber qual é;

### 3.11 Removendo volumes não utilizados

- Podemos remover todos os volumes que não estão sendo utilizados com apenas um comando;
- O comando é:  
`docker volume prune`
- Semelhante ao prune que remove imagens e containers, visto anteriormente;

### 3.12 Volume apenas de leitura

- Podemos criar um volume que tem apenas permissão de leitura, isso é útil em algumas aplicações;
- Para realizar esta configuração devemos utilizar o comando:  
`docker run -v volume:/data:ro`
- Este :ro é a abreviação de read only;

## 4 Networks

### 4.1 O que são Networks no Docker?

- Uma forma de gerenciar a conexão do Docker com outras plataformas ou até mesmo entre containers;
- As redes ou networks são criadas separadas do containers, como os volumes;
- Além disso existem alguns drivers de rede, que veremos em seguida;
- Uma rede deixa muito simples a comunicação entre containers;

### 4.2 Tipos de conexão

- Os containers costumam ter três principais tipos de comunicação:
  - **Externa:** conexão com uma API de um servidor remoto;
  - **Com o host:** comunicação com a máquina que está executando o Docker;
  - **Entre containers:** comunicação que utiliza o driver bridge e permite a comunicação entre dois ou mais containers;

### 4.3 Tipos de rede (drivers)

- **Bridge:** o mais comum e default do Docker, utilizado quando containers precisam se conectar (na maioria das vezes optamos por este driver);
- **host:** permite a conexão entre um container a máquina que está hosteando o Docker;
- **macvlan:** permite a conexão a um container por um MAC address;
- **none:** remove todas as conexões de rede de um container;
- **plugins:** permite extensões de terceiros para criar outras redes;

### 4.4 Listando redes

- Podemos verificar todas as redes do nosso ambiente com:  

```
docker network ls
```
- Algumas redes já estão criadas, estas fazem parte da configuração inicial do docker;

### 4.5 Criando redes

- Para criar uma rede vamos utilizar o comando:  

```
docker network create <nome>
```
- Esta rede será do tipo bridge, que é o mais utilizado;
- Podemos criar diversas redes;
- Para criar uma rede com driver diferente, você pode usar o comando  

```
docker network -d <nome-do-driver> <nome-da-rede>
```

## 4.6 Removendo redes

- Podemos remover redes de forma simples também:  
`docker network rm <nome>`
- Assim a rede não estará mais disponível para utilizarmos;
- Devemos tomar cuidado com containers já conectados;

## 4.7 Removendo redes em massa

- Podemos remover redes de forma simples também:  
`docker network prune`
- Assim todas as redes não utilizadas no momento serão removidas;
- Receberemos uma mensagem de confirmação do Docker antes da ação a ser executada;

## 4.8 Instalação do Postman

- Vamos criar uma API para testar a conexão entre containers;
- Para isso, vamos utilizar o software Postman, que é o mais utilizado do mercado para desenvolvimento de APIs;
- Link: <https://www.postman.com>

## 4.9 Conexão externa

- Os containers podem se conectar livremente ao mundo externo;
- Um caso seria: uma API de código aberto;
- Podemos acessá-la livremente e utilizar seus dados;

## 4.10 Conexão com o host

- Podemos também conectar um container com o host do Docker;
- Host é a máquina que está executando o Docker;
- Como ip de host utilizamos: `host.docker.internal`

## 4.11 Conexão entre containers

- Podemos também estabelecer uma conexão entre containers;
- Duas imagens distintas rodando em containers separados que precisam se conectar para inserir um dado no banco, por exemplo;
- Vamos precisar de uma rede bridge, para fazer esta conexão;
- Agora nosso container de flask vai inserir dados em um MySQL que roda pelo Docker também;

## 4.12 Conectar container

- Podemos conectar um container a uma rede;
- Vamos utilizar o comando:  
`docker network connect <rede> <container>`
- Após o comando o container estará dentro da rede;

## 4.13 Desconectar container

- Podemos desconectar um container a uma rede também;
- Vamos utilizar o comando  
`docker network disconnect <rede> <container>`
- Após o comando o container estará fora da rede;

## 4.14 Inspeccionando redes

- Podemos analisar os detalhes de uma rede com o comando:  
`docker network inspect <nome>`
- Vamos receber informações como: data de criação, driver, nome e muito mais;

# 5 YAML

## 5.1 O que é YAML?

- Uma linguagem de serialização, seu nome é YAML ain't Markup Language (YAML não é uma linguagem de marcação);
- Usada geralmente para arquivos de configuração, inclusive do Docker, para configurar o Docker Compose;
- É de fácil leitura para nós humanos;
- A extensão dos arquivos é yml ou yaml;

## 5.2 Vamos criar nosso arquivo YAML

- O arquivo .yaml geralmente possui chaves e valores;
- Que é de onde vamos retirar as configurações do nosso sistema;
- Para definir uma chave apenas inserimos o nome dela, em seguida colocamos dois pontos e depois o valor;

## 5.3 Espaçamento e indentação

- O fim de uma linha indica o fim de uma instrução, não há ponto e vírgula;
- A indentação deve conter um ou mais espaços, e não devemos utilizar tab;
- E cada uma define um novo bloco;
- O espaço é obrigatório após a declaração da chave;

## 5.4 Comentários

- Podemos escrever comentários em YAML também utilizando o símbolo #;
- O processador de YAML ignora comentários;
- Eles são úteis para escrever como o arquivo funciona/foi configurado;

## 5.5 Dados numéricos

- Em YAML podemos escrever dados numéricos com:
- Inteiros=12;
- Floats = 15.8;

## 5.6 Strings

- Em YAML podemos escrever textos de duas formas:
- Sem aspas: este é um texto válido;
- Com aspas: “e este também”

## 5.7 Dados nulos

- Em YAML podemos definir um dado como nulo de duas formas:
- ~ ou null
- Os dois vão resultar em None, após a interpretação

## 5.8 Booleanos

- Podemos inserir booleanos em YAML da seguinte forma:
- True e On = Verdadeiro;
- False e Off = Falso;

## 5.9 Arrays

- Os arrays, tipos de dados para listas, possuem duas sintaxes:
- Primeira: [1,2,3,4,5]
- Segunda

Items:

- 1
- 2
- 3

## 5.10 Dicionários

- Os dicionários, tipo de dados para objetos ou listas com chaves e valores, podem ser escritos assim:
- obj: {a: 1, b: 2, c: 3}
- E também com o nesting:

objeto:

chave: 1  
chave: 2

## 6 Docker Compose

### 6.1 O que é o Docker Compose?

- O Docker Compose é uma ferramenta para rodar múltiplos containers;
- Teremos apenas um arquivo de configuração, que orquestra totalmente esta situação;
- É uma forma de rodar múltiplos builds e runs com um comando;
- Em projetos maiores é essencial o uso do Compose;

### 6.2 Criando nosso primeiro Compose

- Primeiramente vamos criar um arquivo chamado docker-compose.yml na raiz do projeto;
- Este arquivo vai coordenar os containers e imagens e possui algumas chaves muito utilizadas;
- version: versão do Compose;
- services: Containers/serviços que vão rodar nessa aplicação;
- volumes: Possível adição de volumes;

### 6.3 Rodando o compose

- Para rodar nossa estrutura em Compose vamos utilizar o comando:  
`docker-compose up`
- Isso fará com que as instruções no arquivo seja executadas;
- Da mesma forma que realizamos os builds e também os runs;
- Podemos parar o Compose com `ctrl+c` no terminal;

### 6.4 Compose em background

- O Compose também pode ser executado em modo detached;
- Para isso, vamos utilizar a flag `-d` no comando;
- E então os containers estarão rodando em background;
- Podemos ver sua execução com  
`docker ps`



## 6.5 Parando o compose

- Podemos parar o compose que roda em background com:  
`docker-compose down`
- Desta maneira o serviço para e temos os containers adicionados no `docker ps -a`;

## 6.6 Variáveis de ambiente

- Podemos definir variáveis de ambiente para o Docker Compose;
- Para isso vamos definir um arquivo base em `env_file`;
- As variáveis podem ser chamadas pela sintaxe: `${VARIABLE}`;
- Esta técnica é útil quando o dado a ser inserido é sensível/não pode ser compartilhado, como uma senha;

## 6.7 Redes no compose

- O Compose cria uma rede básica Bridge entre os containers da aplicação;
- Porém podemos isolar as redes com a chave `networks`;
- Desta maneira podemos conectar apenas os containers que optarmos;
- E podemos definir drivers diferentes também;

# 7 Docker Swarm

O Docker Swarm é uma forma de orquestrar containers, até agora falamos apenas de processos isolados. Porém nem sempre é assim, na realidade precisaremos lidar com aplicações que terão milhares de acesso, então o docker swarm será responsável por fazer isso.

## 7.1 O que é orquestração de containers?

- Orquestração é o ato de conseguir gerenciar e escalar os containers da nossa aplicação;
- Temos um serviço que rege sobre outros serviços, verificando se os mesmos estão funcionando como deveriam;
- Desta forma, conseguimos garantir uma aplicação saudável e também que esteja sempre disponível;
- Alguns serviços: Docker Swarm, Kubernetes e Apache Mesos;

## 7.2 O que é Docker Swarm?

- Uma ferramenta do Docker para orquestrar containers;
- Podendo escalar horizontalmente nossos projetos de maneira simples;
- O famoso cluster;
- A facilidade do Swarm para outros orquestradores é que todos os comando são muito semelhantes ao do Docker;
- Toda instalação do Docker já vem com Swarm, porém desabilitado;

### 7.3 Conceitos fundamentais

- **Nodes:** é uma instância (máquina) que participa do Swarm;
- **Manager Node:** Node que gerencia os demais Nodes;
- **Worker Node:** Nodes que trabalham em função do Manager;
- **Service:** um conjunto de Tasks que o Manager Node manda o Work Node executar;
- **Task:** comandos que são executados nos Nodes;

### 7.4 Maneira de executar o Swarm

- Para exemplificar corretamente o Swarm vamos precisar de Nodes, ou seja, mais máquinas;
- Então temos duas soluções:
  - AWS: criar a conta e rodar alguns servidores (precisa de cartão de crédito, mas é gratuito);
  - Docker Labs: gratuito também, roda no navegador, porém expira a cada 4 horas;

### 7.5 Iniciando o Swarm

- Podemos iniciar o Swarm com o comando:  
`docker swarm init`
- Em alguns casos, quando trabalhamos com aplicativos web, precisaremos declarar o IP do servidor com a flag:  
`—advertise—addr`
- Isso fará com que a instância/máquina vire um Node;
- E também transforma o Node em um Manager;

### 7.6 Listando Nodes Ativos

- Podemos verificar quais Nodes estão ativos com:  
`docker node ls`
- Desta forma os serviços serão exibidos no terminal;
- Podemos assim monitorar o que o Swarm está orquestrando;
- Este comando será de grande utilidade a medida que formos adicionando serviços no Swarm;

### 7.7 Adicionando Novos Nodes

- Podemos adicionar um novo serviço com o comando:  
`docker swarm join —token <TOKEN> <IP>:<PORTA>`
- Desta forma duas máquinas estarão conectadas;
- Esta nova máquina entra na hierarquia como Worker;
- Todas as ações (Tasks) utilizadas na Manager serão replicadas em Nodes que foram adicionado com join;

## 7.8 Subindo um Novo Serviço

- Podemos iniciar um serviço com o comando:  
`docker service create --name <nome> <imagem>`
- Desta forma teremos um container novo sendo adicionado ao nosso Manager;
- E este serviço estará sendo gerenciado pelo Swarm;
- Podemos testar com o nginx, liberando a porta 80 o container já pode ser acessado;

## 7.9 Listando Serviços

- Podemos listar os serviços que estão rodando com:  
`docker service ls`
- Desta maneira todos os serviços que iniciamos serão exibidos;
- Algumas informações importantes sobre eles estão na tabela: nome, replicas, imagem, porta;

## 7.10 Removendo Serviços

- Podemos remover um serviço com:  
`docker service rm <nome>`
- Desta maneira o serviço para de rodar;
- Isso pode significar parar um container que está rodando e outras consequências devido a parada do mesmo;
- Checamos a remoção com:  
`docker service ls`

## 7.11 Aumentando o Número de Réplicas

- Podemos criar um serviço com um número maior de réplicas:  
`docker service create --name <NOME> --replicas <NUMERO> <IMAGEM>`
- Desta maneira uma task será emitida, replicando este serviço nos Workers;
- Agora iniciamos de fato a orquestração;
- Podemos checar o status com:  
`docker service ls`

## 7.12 Verificando a Orquestração

- Vamos remover um container de um Node Worker;
- Isso fará com que o Swarm reinicie este container novamente;
- Pois o serviço ainda está rodando no Manager, e isto é uma de suas atribuições: garantir que os serviços estejam sempre disponíveis;
- Obs: precisamos utilizar a flag force (-f);

### 7.13 Checando Token do Swarm

- As vezes vamos precisar checar o token do Swarm, para dar join em alguma outra instância futuramente;
- Então temos o comando:  
`docker swarm join --token manager`
- Desta forma recebemos o token pelo terminal;

### 7.14 Checando o Swarm

- Podemos verificar detalhes do Swarm que o Docker está utilizando;
- Utilizamos o comando:  
`docker info`
- Desta forma recebemos informações como: ID do Node, número de Nodes, número de managers e muito mais.

### 7.15 Removendo Instância do Swarm

- Podemos parar de executar o Swarm em uma determinada instância também;
- Vamos utilizar o comando:  
`docker swarm leave`
- A partir deste momento, a instância não é contada mais como um Node para o Swarm;

### 7.16 Removendo um Node

- Podemos também remover um Node do nosso ecossistema do Swarm;
- Vamos utilizar o comando:  
`docker node rm <ID>`
- Desta forma a instância não será considerada mais um Node, saindo do Swarm;
- O container continuará rodando na instância;
- Precisamos utilizar a flag -f;

### 7.17 Inspeccionando Serviços

- Podemos ver e m mais detalhes o que um serviço possui;
- O comando é:  
`docker service inspect <ID>`
- Vamos receber informações como: nome, data de criação, portas e muito mais.

### 7.18 Verificar Containers Ativados pelo Service

- Podemos ver quais containers um serviço já rodou;
- O comando é:  
`docker service ps <ID>`
- Receberemos uma lista de containers que estão rodando e também dos que já receberam baixa;
- Este comando é semelhante ao  
`docker ps -a`

### 7.19 Rodando Compose com Swarm

- Para rodar o Compose com o Swarm vamos utilizar os comandos de stack;
- O comando é:  
`docker stack deploy -c <ARQUIVO.yaml> <NOME>`
- Teremos então o arquivo compose sendo executado;
- Porém agora estamos em modo Swarm e podemos utilizar os Nodes como réplicas;

### 7.20 Aumentando réplicas do Stack

- Podemos criar novas réplicas nos Worker Nodes;
- Vamos utilizar o comando:  
`docker service scale <NOME>=<REPLICAS>`
- Desta forma as outras máquinas receberão as tasks a serem executadas;

### 7.21 Fazer Serviço não Receber mais Tasks

- Podemos fazer com que um serviço não receba mais 'ordens' do Manager;
- Para isso vamos utilizar o comando:  
`docker node update --availability drain <ID>`
- O status de drain é o que não recebe tasks;
- Podemos voltar para active, e ele volta ao normal.

### 7.22 Atualizar parâmetro

- Podemos atualizar as configurações dos nosso nodes;
- Vamos utilizar o comando:  
`docker service update --image <IMAGEM> <SERVICO>`
- Desta forma apenas os nodes que estão com o status active receberão atualizações;

### 7.23 Criando Rede para Swarm

- A conexão entre instâncias usa um driver diferente, o overlay;
- Podemos criar primeiramente a rede com  
`docker network create`
- E depois ao criar um service, adicionar a flag  
`—network <REDE>`  
para inserir as instâncias na nossa nova rede;

### 7.24 Conectar Serviço a uma Rede

- Podemos também conectar serviços que já estão em execução em uma rede;
- Vamos utilizar o comando de update:  
`docker service update —network <REDE> <NOME>`
- Depois checamos o resultado com inspect;