



INSIGHT

# Introdução ao Docker



# AGENDA

1. Primeiros passos com Docker
2. O que são imagens?
3. Volumes/Armazenamento
4. Como construir imagens?
5. Comunicação entre containers
6. Docker compose

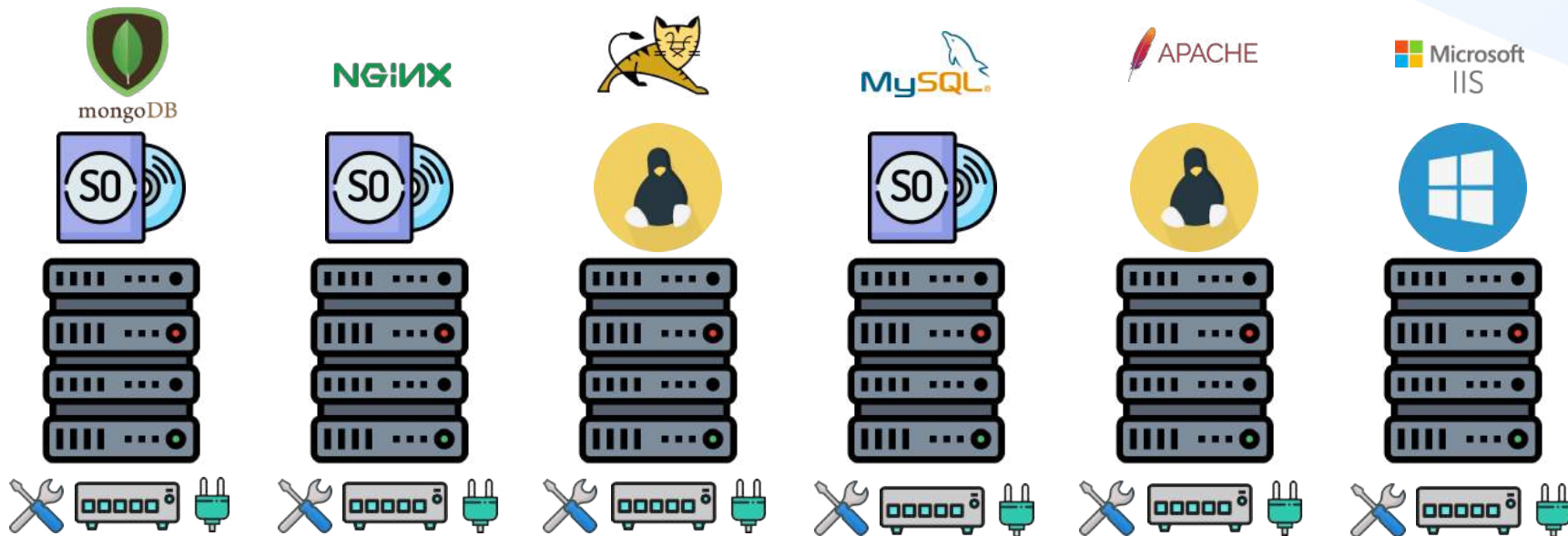
# 1. Primeiros passos com Docker

Como surgiram os containers?

## Máquinas virtuais

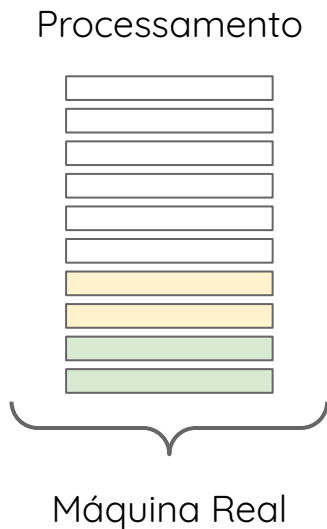
### Como hospedamos aplicações anteriormente?

- ▶ Para vários serviços, utilizamos várias máquinas



## Máquinas virtuais

### Capacidade subutilizada!

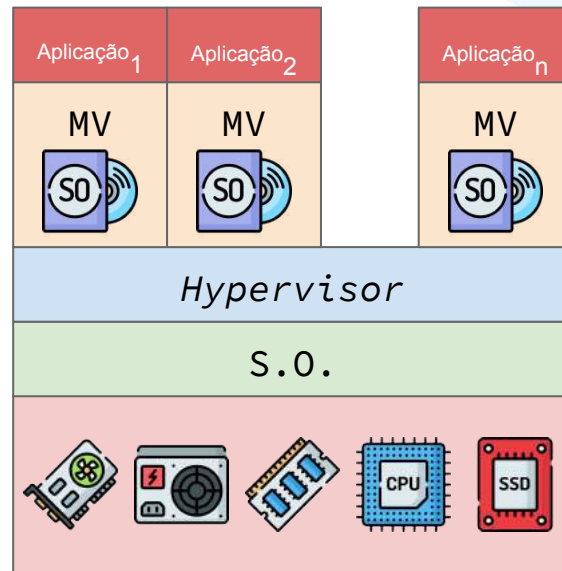


Máquinas subutilizadas na maior parte do tempo, desperdiçando poder computacional!

- ▶ Uso sazonal da capacidade máxima.

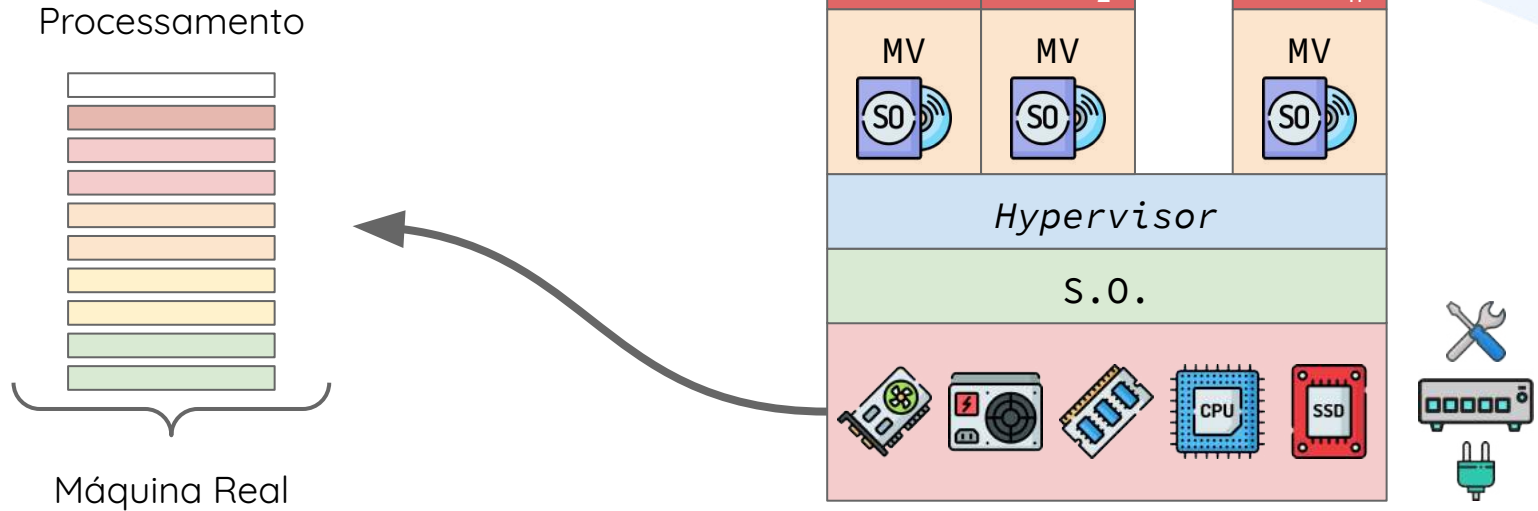
## Máquinas virtuais

**Solução:** vamos virtualizar nossos serviços!



## Máquinas virtuais

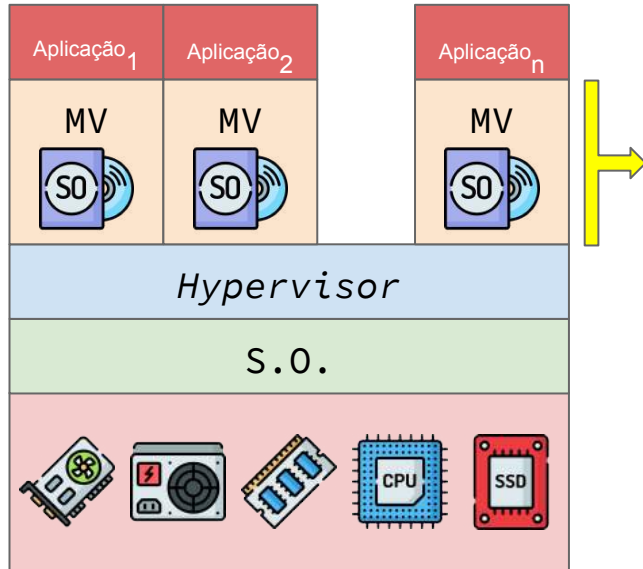
As MV otimizaram o uso da capacidade de processamento disponível!





# Máquinas virtuais

Mas quais são as desvantagens das MV?



## Custos de um Sistema Operacional

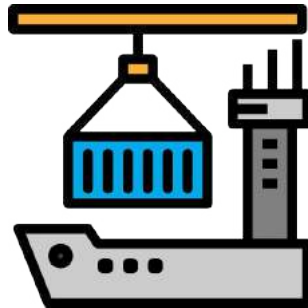
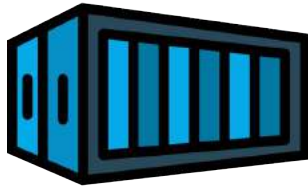
- ▶ Configuração
- ▶ Atualização
- ▶ Segurança
- ▶ Para cada máquina virtual
  - ▶ Armazenamento
  - ▶ Processamento
  - ▶ Memória

## Máquinas virtuais

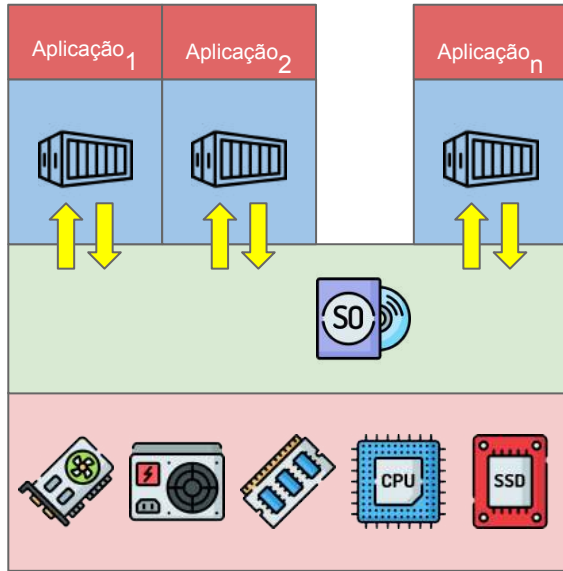
Já resolvemos os problemas das máquinas físicas com as máquinas virtuais.

- ▶ E agora, como resolver o problema das máquinas virtuais?

Utilizando...



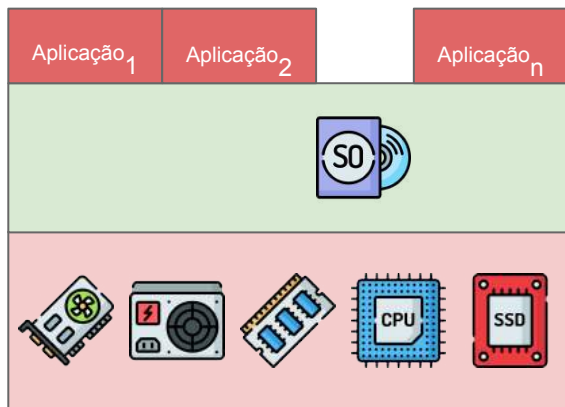
## Caracterizando um container



Um container será um ambiente de virtualização

- ▶ Com menos consumo de processamento,
- ▶ Com menor custo de manutenção/configuração,
- ▶ Com menor tempo de inicialização/finalização.

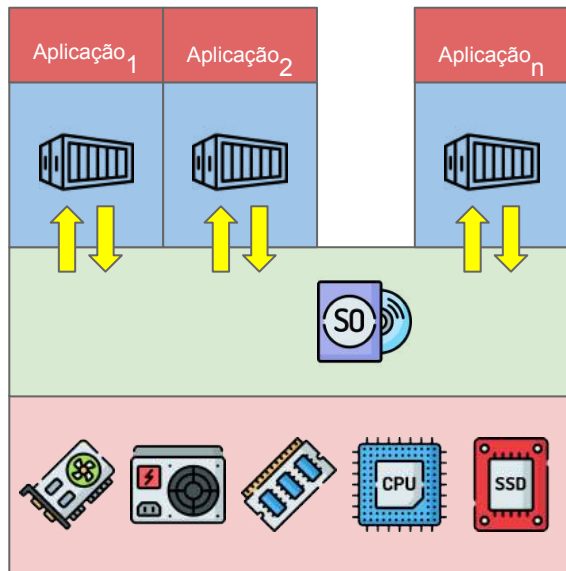
## Caracterizando um container



**Por que não utilizar as aplicações diretamente no SO principal?**

1. Concorrência entre portas,
2. Consumo unilateral da CPU,
3. Dependências entre diferentes recursos.

## Caracterizando um container

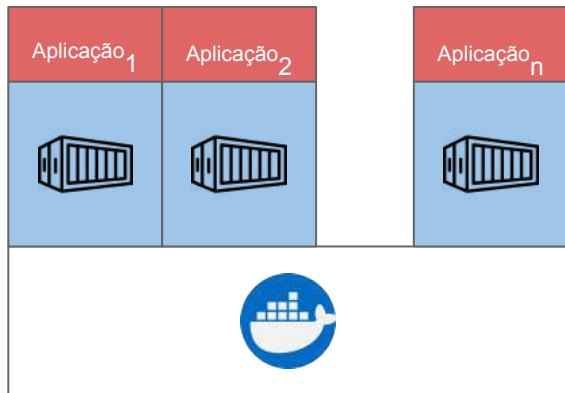


### Ganhamos:

- ▶ Controle sobre o uso de cada recurso,
- ▶ Velocidade para criação/destruição de serviços,
- ▶ Maior gerenciamento de dependências de bibliotecas,
- ▶ Menor consumo de processamento do que as MV.

# Docker

Alternativa de virtualização em que o kernel da máquina hospedeira é compartilhado com a máquina virtualizada ou o software em operação.

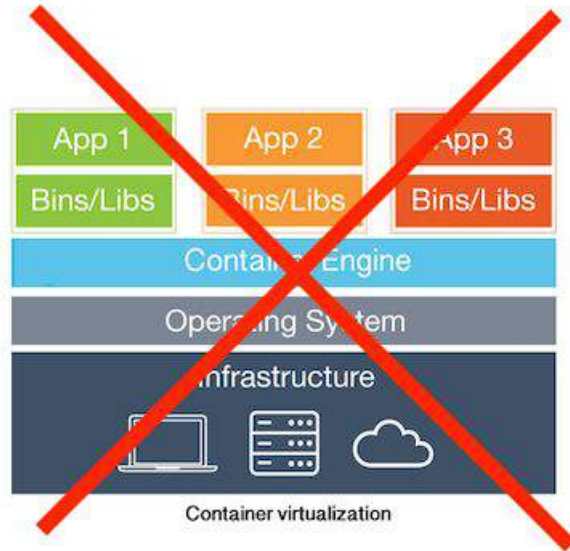
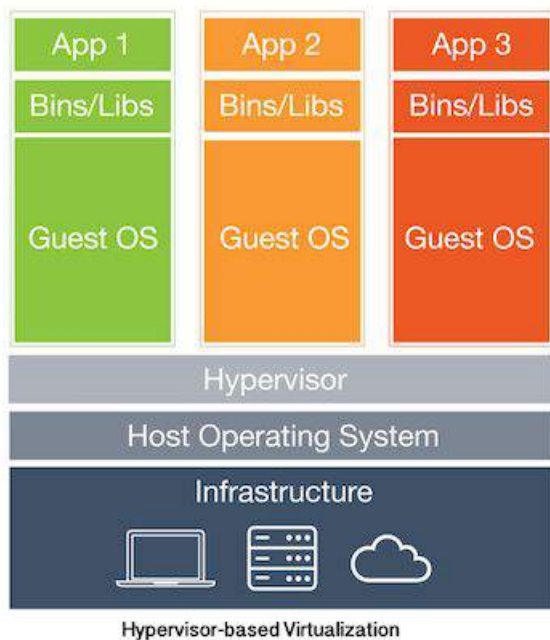


## Docker Engine

## Hypervisor x OS-level virtualization

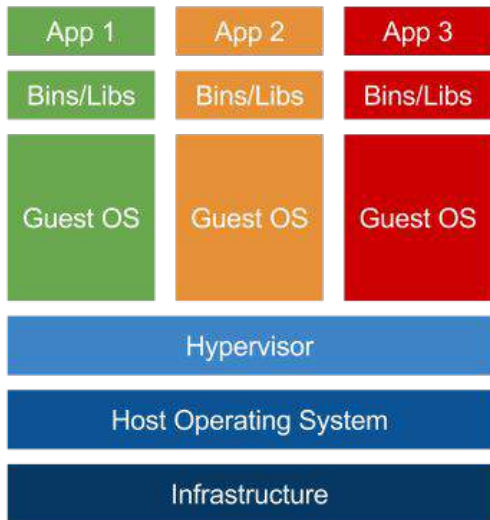
- ▶ O hypervisor apresenta aos sistemas operacionais convidados uma plataforma operacional virtual e gerencia a execução dos sistemas operacionais convidados.
- ▶ Várias instâncias de vários sistemas operacionais podem compartilhar os recursos de hardware virtualizado: por exemplo, as instâncias Linux, Windows e macOS podem ser executadas em uma única máquina x86 física.
- ▶ Isso contrasta com a virtualização no nível do sistema operacional, onde todas as instâncias (contêineres) devem compartilhar um único kernel, embora os sistemas operacionais convidados possam diferir no espaço do usuário, como distribuições Linux diferentes com o mesmo kernel.

# Docker NÃO é um Hypervisor

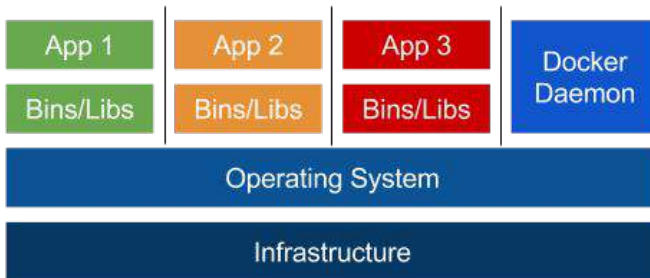




# Docker NÃO é um Hypervisor



- Docker Daemon está fora do caminho de execução;
- Apps possuem “paredes” entre elas;
- Docker Daemon é apenas mais um processo.



apps DO NOT sit “on top” of the container engine

## Docker - A(s) tecnologia(s)

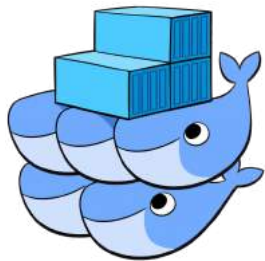


**Docker Compose:** uma ferramenta para definir e executar aplicativos Docker a partir de vários contêineres.



**Docker Hub:** Um repositório com mais de 250 mil imagens diferentes para os seus containers.

## Docker - A(s) tecnologia(s)

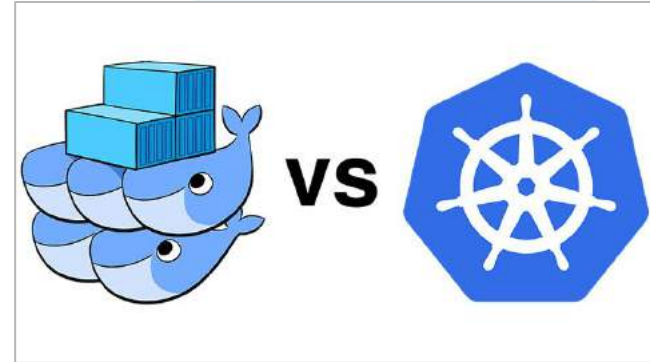
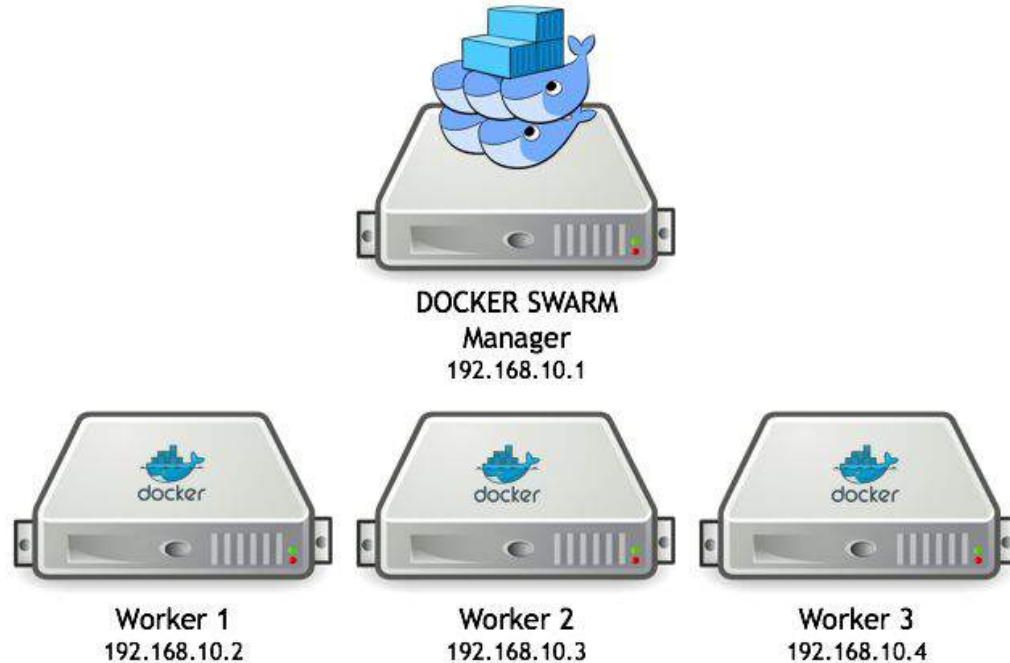


**Docker Swarm:** ferramenta nativa de orquestração do docker.

Permite que containers executem distribuídos em um cluster, controlando a quantidade de containers, registro, deploy e update de serviços.

# Docker - A(s) tecnologia(s)

## Docker Swarm



## Play with Docker

Ao longo do curso, utilizaremos a ferramenta online **Play With Docker**.

- ▶ Nele você poderá utilizar comandos e testar as diversas funcionalidades que o Docker proporciona.

Ao acessar o site, basta clicar em **+Add New Instance** e começar a utilizá-lo como se estivesse usando sua máquina normalmente.

## Exercício 1 - Hello World!

Vamos iniciar nosso estudo sobre Docker com o Hello World dos containers!

```
docker run hello-world
```

Para verificarmos a versão atual do Docker, basta executarmos

```
docker version
```

## Glossário

**Imagem:** pacote com todas as dependências e informações necessárias para criar um contêiner.

**Dockerfile:** arquivo de texto que contém instruções sobre como criar uma imagem.

**Build:** Ação de criação de uma imagem.

**Contêiner:** Instância de uma imagem do Docker.

**Volume:** Oferece um sistema de arquivos gravável que pode ser usado pelo contêiner.

- ▶ Volumes ficam no sistema de host e são gerenciados pelo Docker.

## Glossário

**Tag:** rótulo que pode ser aplicado a imagens, de modo que as diferentes imagens ou versões da mesma imagem possam ser identificadas.

**Repositório:** coleção de imagens relacionadas, rotuladas com uma tag que indica a versão da imagem.

**Registro:** Serviço que fornece acesso aos repositórios.

- ▶ O registro padrão para as imagens públicas é o Docker Hub.
- ▶ Um registro geralmente contém repositórios de várias equipes.
- ▶ As empresas geralmente têm registros privados para armazenar e gerenciar as imagens que criaram.



## Glossário

**DockerHub:** registro público para fazer upload de imagens e trabalhar com elas.

- ▶ Hospeda imagens, registros públicos ou privados, cria gatilhos e ganchos da Web e integra-se com GitHub e Bitbucket.

**Compose:** Ferramenta de linha de comando e formato de arquivo YAML para definir e executar aplicativos de vários contêineres.

**Cluster:** Coleção de hosts do Docker expostos como um único host virtual do Docker.

- ▶ O aplicativo pode ser dimensionado para várias instâncias dos serviços distribuídos em vários hosts do cluster.
- ▶ Podem ser criados com o Kubernetes, o Azure Service Fabric, o Docker Swarm, etc.

## Glossário

**Orquestrador:** Ferramenta que simplifica o gerenciamento de clusters e hosts do Docker.

- ▶ Permite gerenciar imagens, contêineres e hosts por meio de uma CLI ou GUI.
- ▶ Responsável por executar, distribuir, dimensionar e reparar de cargas de trabalho em uma coleção de nós.
- ▶ Produtos de orquestrador são os mesmos que fornecem infraestrutura de cluster, como Kubernetes e Azure Service Fabric.



## 2. O que são imagens?

Criando e executando containers

## Comandos básicos com containers

Vamos tentar entender melhor o que aconteceu quando executamos o comando `docker run hello-world`

Ao executar esse comando, o Docker:

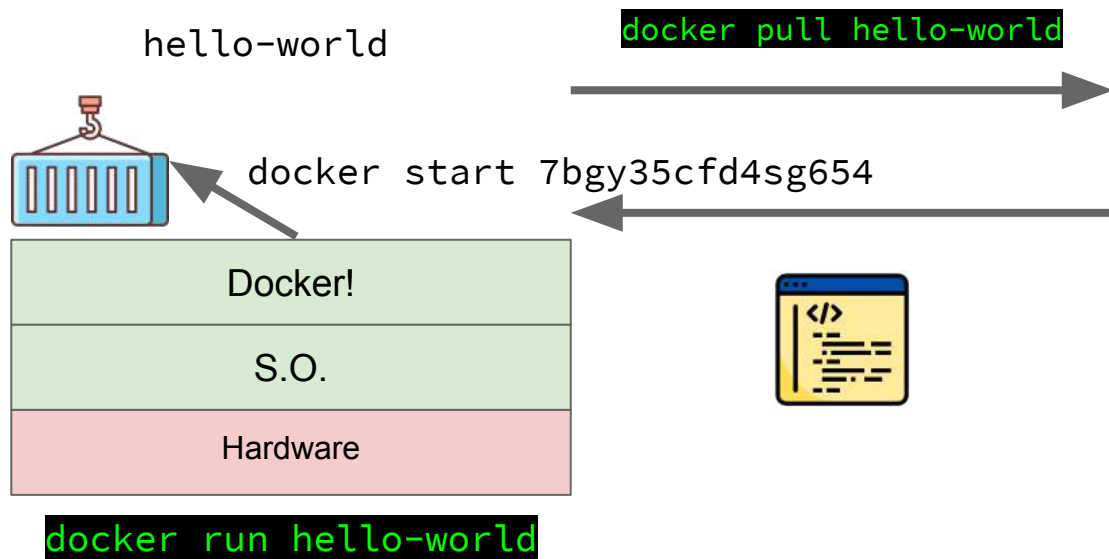
- ▶ Verifica se temos a **imagem** localmente
- ▶ Caso contrário, busca e faz o download no [Docker Store](#)!

## Comandos básicos com containers

A imagem do Docker é, basicamente, uma **série de instruções** que o Docker seguirá para criar um container.

Em seguida, com o container criado, o Docker irá **executá-lo**.

# A diferença!



Docker Hub

## Comandos básicos com containers

Vamos tentar executar uma imagem do Alpine

```
docker run alpine
```

Reparem que não é feito apenas um download. A imagem é dividida em várias *camadas*.

- ▶ Por que nada aconteceu quando o download foi finalizado?

## Algumas imagens e seus tamanhos


```
$ docker images
```


REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<b>busybox</b>	<b>latest</b>	<b>19485c79a9bb</b>	<b>6 weeks ago</b>	<b>1.22MB</b>
<b>alpine</b>	<b>latest</b>	<b>965ea09ff2eb</b>	<b>16 hours ago</b>	<b>5.55MB</b>
ubuntu	latest	cf0f3ca922e0	3 days ago	64.2MB
debian	stable-slim	eece114ab04d	5 days ago	69.2MB
debian	latest	8e9f8546050d	5 days ago	114MB
centos	latest	0f3e07c0138f	2 weeks ago	220MB





# Imagens mais populares


<https://hub.docker.com/search/?type=image>


**Oracle Java 8 SE (Server JRE)** DOCKER CERTIFIED  
By Oracle • Updated 20 days ago  
Oracle Java 8 SE (Server JRE)  
Container Docker Certified Linux x86-64 Programming Languages VERIFIED PUBLISHER


**couchbase** OFFICIAL IMAGE  
Updated 16 minutes ago **10M+** **488**  
Downloads Stars  
Couchbase Server is a NoSQL document database with a distributed architecture.  
Container Linux x86-64 Storage Application Frameworks


**postgres** OFFICIAL IMAGE  
Updated 18 minutes ago **10M+** **7.1K**  
Downloads Stars  
The PostgreSQL object-relational database system provides reliability and data integrity.  
Container Linux IBM Z ARM 64 ARM 386 x86-64 PowerPC 64 LE Databases

**traefik** OFFICIAL IMAGE  
Updated 18 minutes ago **10M+** **1.1K**  
Downloads Stars  
Traefik, The Cloud Native Edge Router  
Container Linux Windows ARM ARM 64 x86-64 Application Infrastructure

**busybox** OFFICIAL IMAGE  
Updated 18 minutes ago **10M+** **1.7K**  
Downloads Stars  
Busybox base image.  
Container Linux PowerPC 64 LE IBM Z ARM x86-64 ARM 64 386 Base Images

**alpine** OFFICIAL IMAGE  
Updated 18 minutes ago **10M+** **5.7K**  
Downloads Stars  
A minimal Docker image based on Alpine Linux with a complete package index and only 5 MB in size!  
Container Linux x86-64 ARM 64 PowerPC 64 LE 386 ARM IBM Z Featured Images Base Images Operating Systems

**ubuntu** OFFICIAL IMAGE  
Updated 18 minutes ago **10M+** **10K+**  
Downloads Stars  
Ubuntu is a Debian-based Linux operating system based on free software.  
Container Linux IBM Z 386 x86-64 ARM PowerPC 64 LE ARM 64 Base Images Operating Systems

**node** OFFICIAL IMAGE  
Updated 18 minutes ago **10M+** **8.0K**  
Downloads Stars  
Node.js is a JavaScript-based platform for server-side and networking applications.  
Container Linux x86-64 386 IBM Z PowerPC 64 LE ARM 64 ARM Application Infrastructure

## Comandos básicos com containers

Para verificarmos os containers que estão sendo executados, utilizaremos o comando

```
docker ps
```

- ▶ Quantos containers temos ativos atualmente?

Quando um container não está sendo executado, ele fica *parado* (**stopped**)!

## Comandos básicos com containers

Para verificarmos todos os containers (parados e em execução), utilizamos a *flag* -a

```
docker ps -a
```

Com esse comando, conseguimos ver algumas informações sobre os containers, como **id**, **nome**, a **imagem** baseada, **comando inicial**, etc.

## Comandos básicos com containers

```
[node1] (local) root@192.168.0.23 ~
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
89d9c30c1d48: Pull complete
Digest: sha256:c19173c5ada610a598915111163d28a67368362762534d8a8121ce95cf2bd5a
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
[node1] (local) root@192.168.0.23 ~
$ docker run alpine
[node1] (local) root@192.168.0.23 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              N
AMES
[node1] (local) root@192.168.0.23 ~
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              N
AMES
d627d71f316c        alpine              "/bin/sh"          12 seconds ago      Exited (0) 11 seconds ago
vibrant_wilbur
```

## Comandos básicos com containers

Para que o container criado realize alguma atividade, precisamos informar junto do comando de execução o que ele deverá fazer. Por exemplo

```
docker run alpine echo "Olá Mundo"
```

Mas como faremos para interagir com o container do Alpine que está sendo criado?

## Comandos básicos com containers

O comando run aceita a informação de qual versão de uma determinada imagem gostaríamos de utilizar. Para isso, basta informarmos através de uma **tag**.

Se nenhuma tag for informada, o Docker irá utilizar sempre a versão **latest**.

```
docker run ubuntu:18.04 cat /etc/issue
```

## Comandos básicos com containers

No [hub.docker.com](https://hub.docker.com) podemos encontrar mais informações sobre as tags disponíveis.



The screenshot shows the Docker Hub interface for a container image. The 'DESCRIPTION' tab is active, displaying a list of supported tags and their respective Dockerfile links. The 'REVIEWS' and 'TAGS' tabs are also visible. On the right side, there is a section for adding a product review, including a dropdown menu to select a product tier and a 'Start Your Review' button.

**DESCRIPTION**   REVIEWS   TAGS

### Supported tags and respective Dockerfile links

- 18.04, bionic-20190912.1, bionic, latest
- 18.10, cosmic-20190719, cosmic
- 19.04, disco-20190913, disco, rolling
- 19.10, eoan-20190916, eoan, devel
- 16.04, xenial-20190904, xenial

**Add Product Review**

Select a product tier

[Start Your Review](#)

## Comandos básicos com containers

Para instanciarmos um container e executarmos um comando no background, podemos utilizar a flag `-d`.

```
docker run -d alpine sleep 20
```

Por que não vai funcionar se não inserirmos o comando `sleep`?



## Comandos básicos com containers

Da mesma forma que podemos instanciar e executar um comando dentro de um container, podemos pará-lo

```
docker stop 3hg567caz645
```

## docker run

Pergunta: o comando run sempre cria novos containers?

```
docker run --rm alpine echo "Olá Mundo"
```

## Exercício - Comandos básicos com containers

1. Crie um container CentOS
2. Execute o comando sleep e suspenda o SO por 2000 segundos
3. Verifique os containers em execução
4. Pare o container que está executando o comando sleep

## Comandos básicos com containers

Caso queiramos executar comandos adicionais em um container que já está em execução, podemos utilizar o comando `exec`.

```
docker exec 3hg56 ls
```

## Exercício - Comandos básicos com containers

1. Crie um container Ubuntu no modo iterativo,
2. Crie um arquivo de texto dentro do container com o comando touch
3. Saia do container
4. Utilize o comando run novamente para tentar "acessar" o container passado
5. Busque pelo arquivo criado anteriormente com o comando ls

## Comandos básicos com containers

Faremos a ligação do terminal da nossa máquina com o terminal dentro do container adicionando a *flag* `-it`

```
docker run -it alpine
```

Perceba que o terminal automaticamente muda e estamos **dentro do container!**

- ▶ *O que acontece se, em outro terminal, executarmos o comando*

```
docker ps ?
```

## Comandos básicos com containers

Lembrem-se: as ferramentas do seu host provavelmente vão ser diferentes do seu container!

Verifiquem a versão do bash no host e no container.

1. `bash --version` (host)
2. `docker run alpine bash --version` (container)

## Comandos básicos com containers

Para inicializarmos um container novamente, basta utilizarmos o comando `docker start <id>`. Dessa forma, nosso terminal não será atrelado ao do container reexecutado.

Para isso, adicionaremos duas flags

- ▶ `-a` (*attach*) - anexa os terminais
- ▶ `-i` (*interactive*) - para interagirmos com o terminal



## Cenário 1 - Comandos básicos com containers

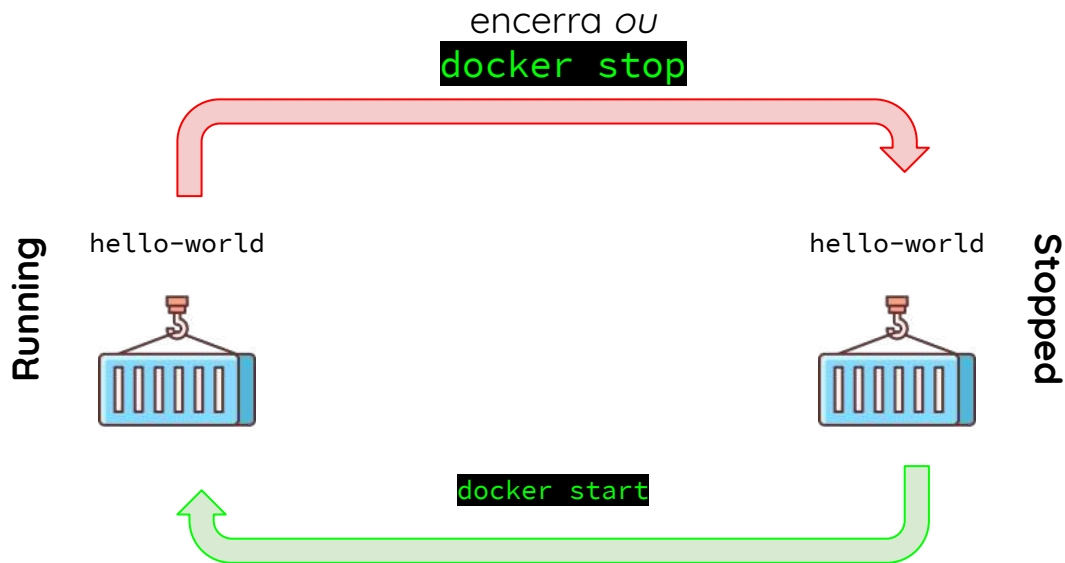
1. Verifique qual a versão do Docker Server Engine que está sendo executado na sua máquina
2. Verifique quantas imagens estão disponíveis no Docker Host
3. Execute um container utilizando uma imagem do Redis
4. Pare o container que você acabou de criar
5. Verifique quantas imagens estão sendo executadas no momento
6. Crie os seguintes containers
  - a. `alpine` com `sleep` de 1000s
  - b. `nginx:alpine` com `sleep` de 900s
  - c. `nginx:alpine` com `sleep` de 1500s
  - d. `ubuntu` com `sleep` de 1000s
  - e. `alpine`
  - f. `redis`

## Cenário 1 - Comandos básicos com containers

7. Verifique quantos containers estão sendo executados no momento
8. Qual o `id` do container instanciado a partir da imagem `alpine` que não está rodando?
9. Qual o estado do container `alpine` que está parado?
10. Delete todos os containers do Docker Host
11. Apague a imagem do `ubuntu`
12. Faça apenas o `pull` da imagem `nginx:1.14-alpine`
13. Execute o container `nginx:1.14-alpine` e nomeie-o `webapp`
14. Remova todas as imagens do Docker Host

# Layered File System

Vimos os dois estados principais de um container



## Layered File System

Infelizmente, depois de tantos testes, criamos um grande número de containers. Para **remover um container**, utilizamos o comando

```
docker rm <id>
```

ou o comando

```
docker container prune
```

para remover todos os containers **inativos (stopped)**.

## Layered File System

Da mesma forma que removemos containers, podemos remover imagens que não nos interessam mais!

O comando `docker images` lista as imagens que temos na nossa máquina e o comando `docker rmi <nome_imagem>` remove a imagem.

Atenção: imagens só serão removidas se não existirem mais containers daquela imagem!

## Layered File System

No Docker, toda imagem é composta por uma ou mais **camadas**.

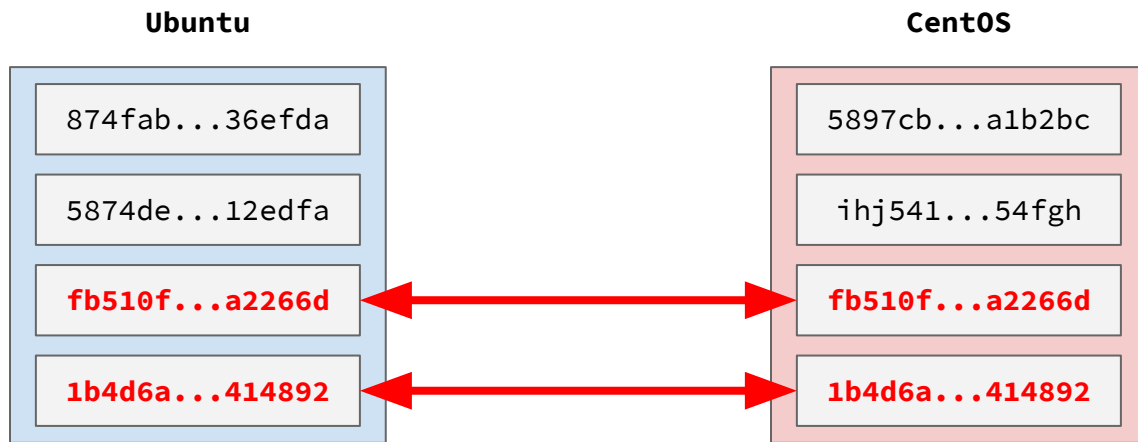
Isso ficou claro quando baixamos a imagem do Ubuntu!

```
$ docker run ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
35c102085707: Pull complete
251f5509d51d: Pull complete
8e829fe70a46: Pull complete
6001e1789921: Pull complete
Digest: sha256:d1d454df0f579c6be4d8161d227462d69e163a8ff9d20a847533989cf0c94d90
Status: Downloaded newer image for ubuntu:latest
```

Esse sistema de camadas chama-se **Layered File System**!

## Layered File System

As camadas utilizadas em uma imagem podem ser reaproveitadas em outra!



## Layered File System

As camadas de uma imagem são apenas para **leitura!**

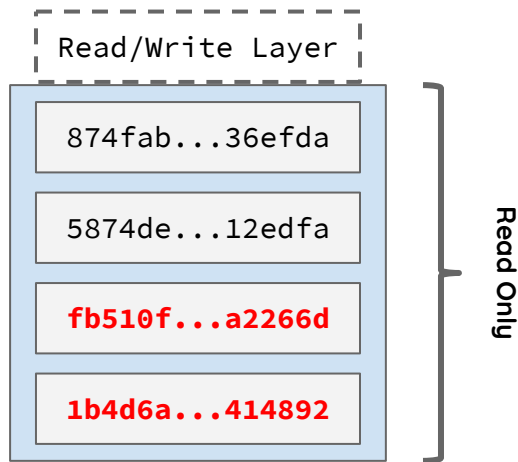
Mas como conseguimos criar arquivos anteriormente? 🤔

Não escrevemos na imagem! O Docker cria uma nova camada acima da imagem! Nessa nova camada podemos ler e escrever!

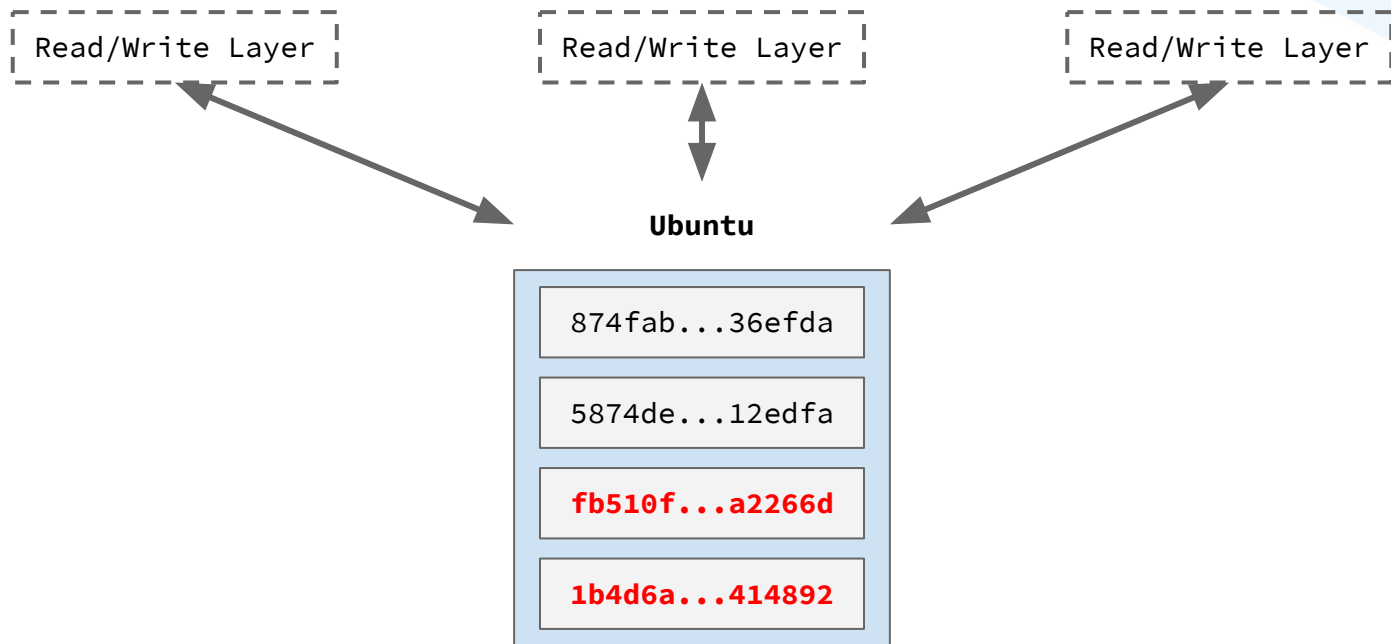


## Layered File System

Quando criamos um container, o Docker cria uma nova camada acima da imagem. Essa nova camada pode ser lida e escrita!



# Layered File System



# c

## 3. Volumes/Armazenamento

## Salvando dados com volumes

Quando removemos um container, **removemos a camada de leitura e escrita.**

E se quisermos **persistir esses dados?**

O lugar especial que usaremos para fazer essa persistência são os **volumes de dados.**

## Salvando dados com volumes

Quando criamos um volume de dados, o que estamos fazendo é apontá-lo para uma pequena pasta no Docker Host.

O container até poderá ser removido, mas a pasta no **Docker Host** ficará intacta. Para isso, utilizaremos a flag `-v`.

```
docker run -d -p 8080:80 -v "/usr/share/nginx/html" nginx
```

## Salvando dados com volumes

A pasta que acabamos de criar no container foi a `/var/www`.

Mas a qual pasta ela estará ligada em nossa máquina?

Para descobrirmos, podemos inspecionar o nosso container.

```
docker inspect 316
```

O que vai nos interessar é o **"Mounts"**.

```
docker inspect 316 | grep -i -A 5 mounts
```

## 63 Salvando dados com volumes

```
[node1] (local) root@192.168.0.8 ~
$ docker inspect 316 | grep -i -A 5 mounts
  "Mounts": [
    {
      "Type": "volume",
      "Name": "3b7d37a6b52edf91a3f9f22bf5860e195464e828bb5fff0140ca96dd8b5bd8e3",
      "Source": "/var/lib/docker/volumes/3b7d37a6b52edf91a3f9f22bf5860e195464e828bb5fff0140ca96dd8b5bd8e3/_data",
      "Destination": "/usr/share/nginx/html",
    }
  ]

```

## 64 Salvando dados com volumes

A pasta gerada pelo Docker pode ser configurada.

Vamos fazer essa configuração e executar o container no modo interativo

```
docker run -p 8080:80 -v "/root/www:/usr/share/nginx/html" nginx
```

```
docker run -p 8080:80 -v "$HOME/www:/usr/share/nginx/html" nginx
```

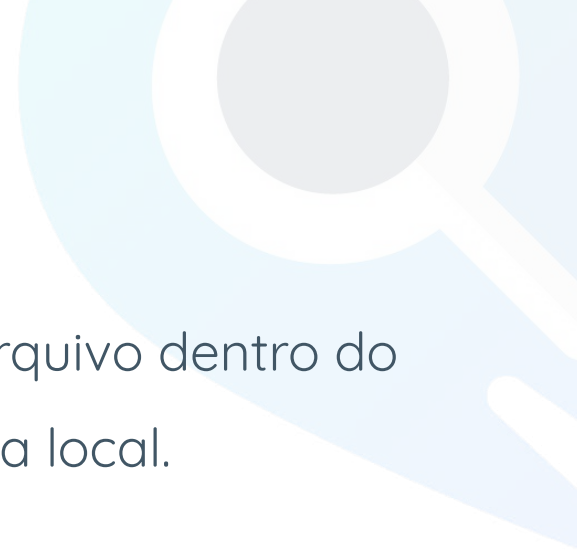
```
docker run -p 8080:80 -v ~/www:/usr/share/nginx/html nginx
```

```
docker run -p 8080:80 -v "~/www:/usr/share/nginx/html" nginx
```



## Salvando dados com volumes

Podemos fazer um simples teste criando um arquivo dentro do container e abrindo-o dentro de nossa máquina local.



## Rodando código em um container

Da mesma forma que criamos um arquivo dentro de um container e utilizamos no Docker Host, **podemos fazer o inverso.**

Com isso, podemos implementar localmente um código em uma linguagem não instalada em nossa máquina, compilá-lo e executá-lo **dentro do container!**

Ou seja, **nosso ambiente de desenvolvimento pode ser dentro de um container!**

4.

## Construindo nossas próprias imagens

## Criando um Dockerfile

Até agora utilizamos imagens prontas direto do Docker Hub, mas ainda **não criamos nossas próprias imagens** para distribuição.

Para criarmos uma imagem, precisamos criar uma *receita de bolo* para ela: o **Dockerfile**.

## Criando um Dockerfile

O Dockerfile que vamos criar nada mais é do que um arquivo de texto com a extensão **.dockerfile**.

Exemplo: `mongo.dockerfile`.

Em geral, criamos novas imagens baseados em uma primeira imagem base.

Se não especificarmos um nome, podemos escolher apenas **Dockerfile**.

## Criando um Dockerfile

Vamos utilizar um exemplo de criação de imagem com o mongodb. Vamos utilizar a imagem base do alpine com a seguinte descrição:

```
FROM alpine:3.9
```

## Criando um Dockerfile

Também podemos indicar o mantenedor da imagem a ser criada.

```
FROM alpine:3.9
```

```
MAINTAINER Insight Data Science Lab
```

## Criando um Dockerfile

Para executarmos um comando, utilizaremos a palavra chave RUN.

```
FROM alpine:3.9
```

```
MAINTAINER Insight Data Science Lab
```

```
RUN apk add --no-cache mongodb
```



## Criando um Dockerfile

```
FROM alpine:3.9
```

```
MAINTAINER Insight Data Science Lab
```

```
RUN apk add --no-cache mongodb
```

```
VOLUME /data/db
```

```
EXPOSE 27017
```

```
CMD [ "mongod", "--bind_ip", "0.0.0.0" ]
```

## Criando um Dockerfile

Para criarmos uma imagem baseada em um Dockerfile, basta utilizarmos a flag `-f` no comando `docker build`.

Além disso, indicaremos o nome da imagem com a *flag* `-t`.

Por fim, indicamos onde está o Dockerfile.

```
docker build -f Dockerfile -t insightlab/mongo:1.0 .
```

```
docker build -t insightlab/mongo:1.0 .
```

## Subindo a imagem no Docker Hub

Para disponibilizarmos uma imagem para outras pessoas, precisamos enviá-la para o Docker Hub.

Primeiramente, precisamos criar uma conta no Docker Hub.

Em seguida, executamos o comando `docker login`.

Em seguida, executamos o comando `docker push` com a imagem que queremos enviar.

```
docker push insightlab/mongo:1.0
```

## Subindo a imagem no Docker Hub

Para baixarmos uma imagem, podemos utilizar o comando `docker pull`.

```
docker pull insightlab/mongo:1.0
```

## Executando

Executando o servidor

```
docker run -d -p 27017:27017 -v $HOME/db:/data/db insightlab/mongo:1.0
```

Testando através do cliente mongo-express

```
docker run -p 8081:8081 -e ME_CONFIG_MONGODB_SERVER="172.17.0.2" mongo-express
```

## ENTRYPOINT x CMD

**ENTRYPOINT** especifica o comando que será executado quando o container inicia.

- ▶ Ponto de entrada (ENTRYPOINT) padrão: `/bin/sh -c`

**CMD** especifica argumentos de entrada do ENTRYPOINT.

- ▶ Docker não tem um comando (CMD) padrão.
- ▶ O comando é executado a partir do ponto de entrada.

Assim, ao executar:

```
docker run -it alpine bash
```

O que é executado é o seguinte:

```
/bin/sh -c bash
```

## Usando ENTRYPOINT e CMD

Dockerfile

```
FROM alpine  
ENTRYPOINT ["/bin/ping"]  
CMD ["localhost"]
```

Construindo a imagem

```
docker build -t alpine-ping .
```

Usando

```
docker run --rm alpine-ping  
docker run --rm alpine-ping 8.8.8.8
```

5.

## Comunicação entre containers

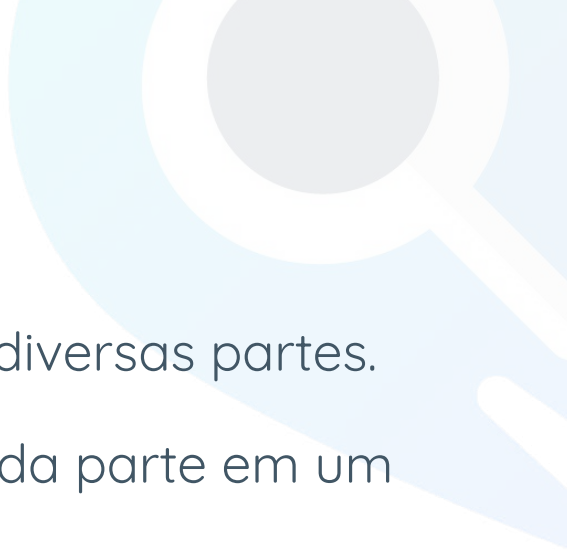


## Networking no Docker

Normalmente, uma aplicação é composta por diversas partes.

Com containers, é bem comum separarmos cada parte em um container específico.

Mas como fazer com que essas partes se comuniquem entre si?



## Networking no Docker

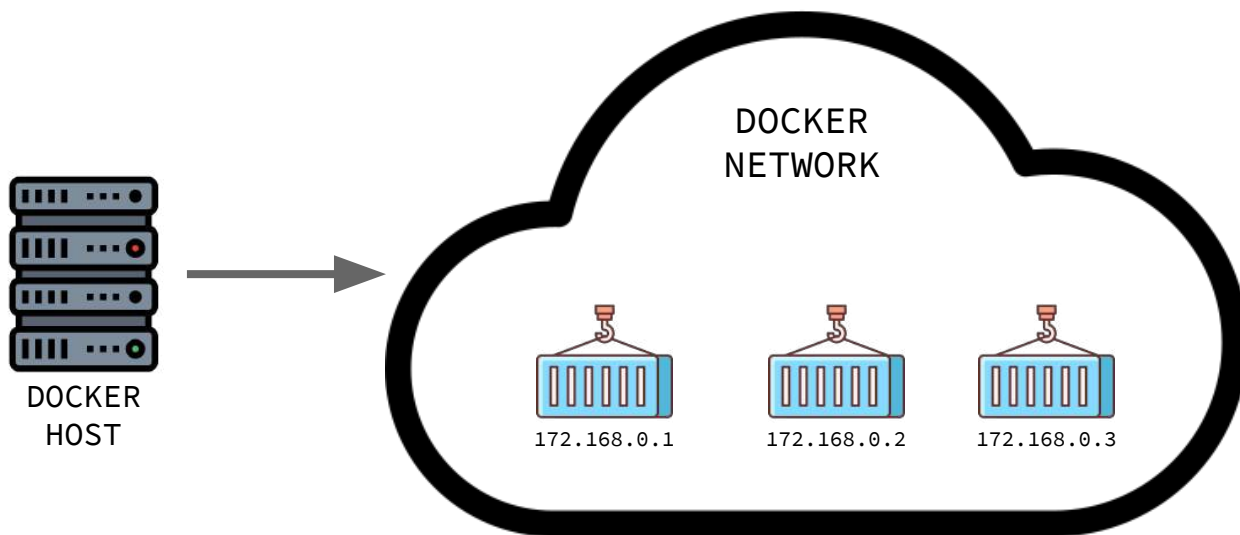
Listando as redes existentes

```
docker network ls
```



## Networking no Docker

O Docker, por padrão, oferece uma **default network**.



## Networking no Docker

Para verificar a rede criada pelo Docker, vamos instanciar um container Alpine.

Em um segundo terminal, inspecione as características do container criado com o comando `docker inspect [id]`.

No container, com o comando `hostname -i`, podemos verificar o IP atribuído àquela instância.

## Networking no Docker

Dentro dessa rede local, os containers podem se comunicar livremente!

Podemos criar outras instâncias e tentar utilizar o comando `ping` entre elas.



## Networking no Docker

Sempre que instanciamos um container, **ele irá receber um novo IP.**

Isso pode ser ruim nas situações em que queremos, por exemplo, inserir dentro de uma aplicação o endereço exato de um banco de dados.

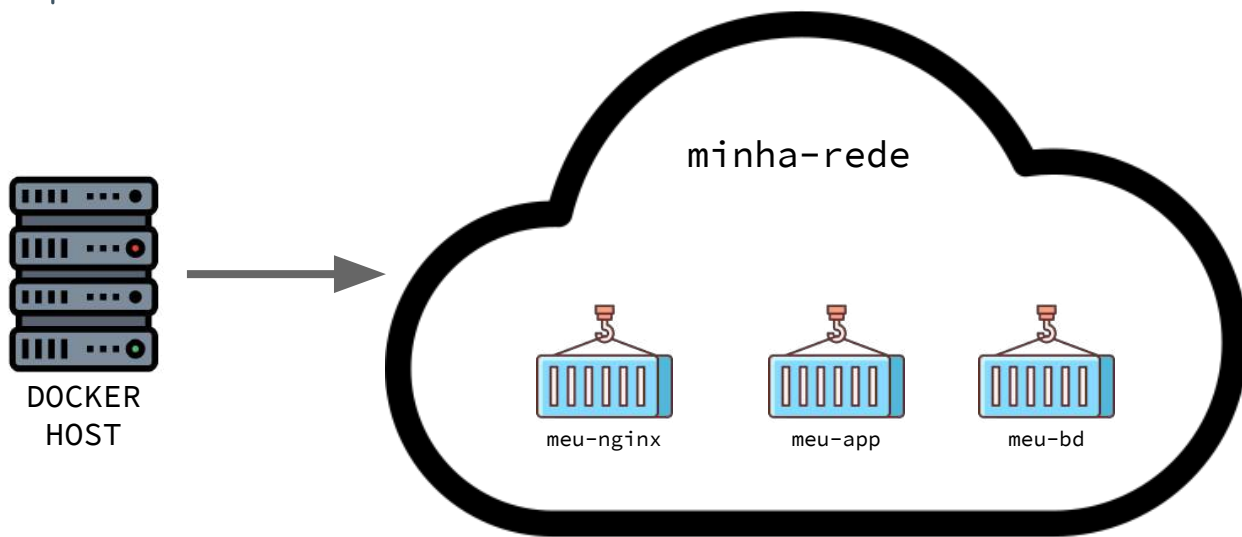
## Networking no Docker

Infelizmente, apesar de conseguirmos nomear um container, com a opção `--name`, **a rede padrão do Docker não permite atribuir um *hostname* a um container.**

Se criarmos nossa própria rede, podemos nomear nossos containers e realizar a comunicação utilizando esses nomes!

## Networking no Docker

A configuração abaixo só pode ser feita quando criamos uma rede própria.





## Networking no Docker

Utilizaremos o comando `network` e indicaremos qual o driver utilizaremos para a criação da rede.

```
docker network create --driver bridge minha-rede
```

## Networking no Docker

As próximas instâncias de containers devem ser associadas a rede `minha-rede` criada anteriormente.

```
docker run -it --name meu-alpine --network  
minha-rede alpine
```

Utilizando o comando `docker inspect`, qual a rede apresentada no container `meu-alpine`? Como podemos testar a comunicação entre containers na nova rede?

## Networking no Docker

Agora conecte mongodb e mongo-express usando a minha-rede.

### Executando o servidor mongodb

```
docker run -d -p 27018:27017 -v $HOME/db:/data/db --name=mongo  
--network=minha-rede insightlab/mongo:1.0
```

OU

```
docker run -d -v $HOME/db:/data/db --name=mongo --network=minha-rede  
insightlab/mongo:1.0
```

### Executando o cliente mongo-express

```
docker run -p 8080:8081 -e ME_CONFIG_MONGODB_SERVER="mongo"  
--network=minha-rede mongo-express
```

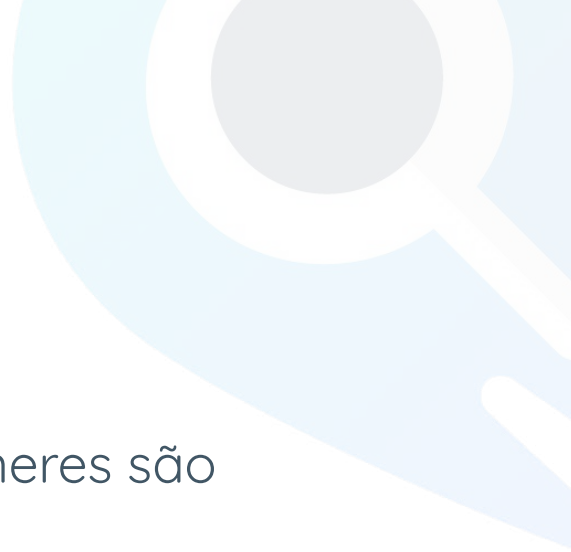
OU

```
docker run -p 8080:8081 --network=minha-rede mongo-express
```

## 6. Trabalhando com o Docker Compose

## Características do Docker Compose

- Vários ambientes isolados em um único host
- Preserva dados de volume quando os contêineres são criados
- Recria apenas contêineres que foram alterados
- Uso de variáveis e mover uma composição entre ambientes



## Entendendo o Docker Compose

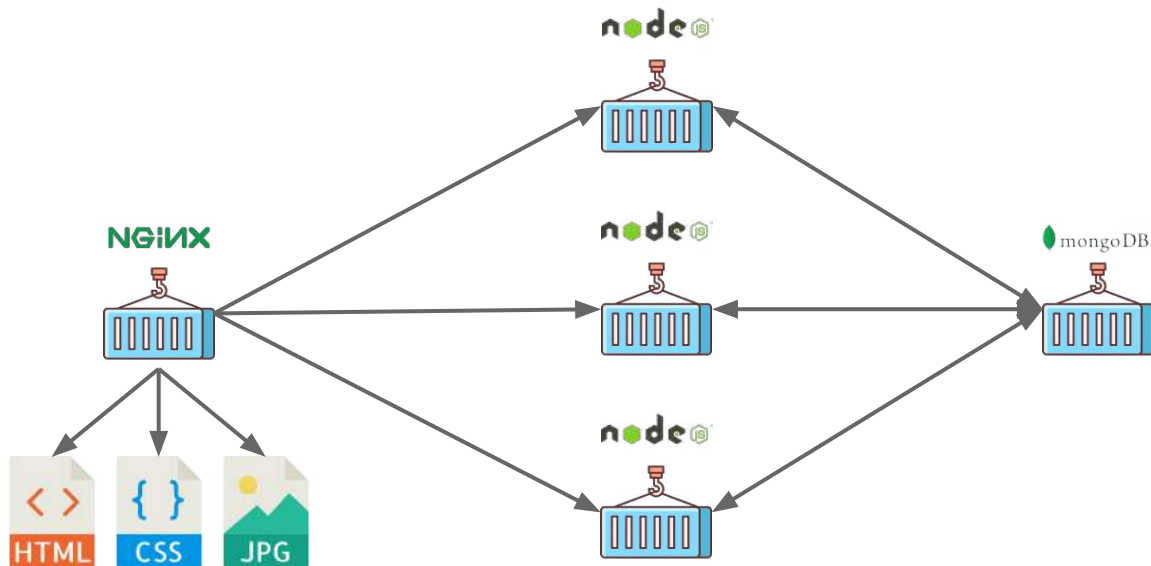
Instanciar um container "manualmente" pode ser problemático: poderíamos **esquecer de alguma flag** ou **errar alguma parte do comando**.

Quando instanciamos mais de um container ao mesmo tempo, esse problema tende a piorar.

Ou seja, essa forma de instanciar containers não é recomendada!

## Entendendo o Docker Compose

Aplicações reais, em geral, tem mais de 2 ou 3 containers.



## Entendendo o Docker Compose

Ao invés de instanciar todos os containers manualmente, podemos utilizar o **Docker Compose**!

O Docker Compose segue um arquivo YAML: escreveremos tudo que queremos que aconteça no instanciamento dos nossos containers.





## Docker Compose e ciclo de vida da aplicação

- Iniciar, parar e reconstruir serviços
- Ver o status dos serviços em execução
- Transmitir a saída de log dos serviços em execução
- Executar um comando único em um serviço



## Tarefas usuais com Docker Compose

- Divida seu aplicativo em serviços
- Pull ou construção de imagens
- Configurar variáveis de ambiente
- Configurar rede
- Configurar volumes
- Build e execução



## Passos básicos

- Defina o ambiente do seu aplicativo com um Dockerfile para que ele possa ser reproduzido em qualquer lugar.
- Defina os serviços que compõem seu aplicativo no docker-compose.yml para que possam ser executados juntos em um ambiente isolado.
- Execute o docker-compose e o Compose inicia e executa todo o aplicativo.

## Exemplos

```
docker-compose up -d  
docker-compose down  
docker-compose start  
docker-compose stop  
docker-compose build  
docker-compose logs  
docker-compose events  
docker-compose exec service command
```



## Exemplo de docker-compose.yml

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
```



## Exemplo de docker-compose.yml

```
version: '3'
services:
  mongo:
    image: mongo
  mongo-express:
    image: mongo-express
    ports:
      - "8080:8081"
    links:
      - mongo
```



## Desafio Docker

<https://github.com/InsightLab/docker-introduction-challenge>

# OBRIGADO!

## Dúvidas?

Você pode nos encontrar em

- ▶ Prof. Gustavo Coutinho
  - ▶ @gustavolgcr
  - ▶ [gustavo.coutinho@insightlab.ufc.br](mailto:gustavo.coutinho@insightlab.ufc.br)
- ▶ Prof. Regis Pires
  - ▶ @regispires
  - ▶ [regis@insightlab.ufc.br](mailto:regis@insightlab.ufc.br)
- ▶ Lucas Peres
  - ▶ @lucasp96
  - ▶ [lucasperes@insightlab.ufc.br](mailto:lucasperes@insightlab.ufc.br)