

Docker e Kubernetes

Rodrigo Moura J. Ayres

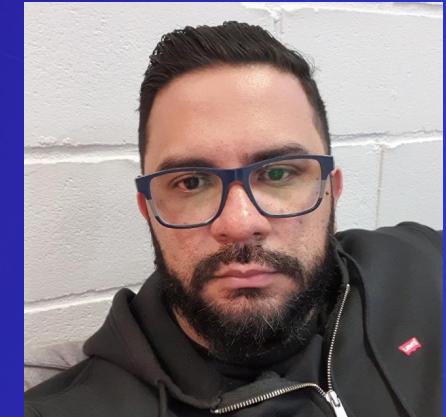
Bacharel em Ciência da Computação @ Univem

Mestre em Ciência da Computação @ UFSCar

Doutorando em Ciência da Computação @ UNESP

Professor de Ensino Superior @ Fatec Ourinhos

Professor de Ensino Superior @ Unifio



rmayres@gmail.com

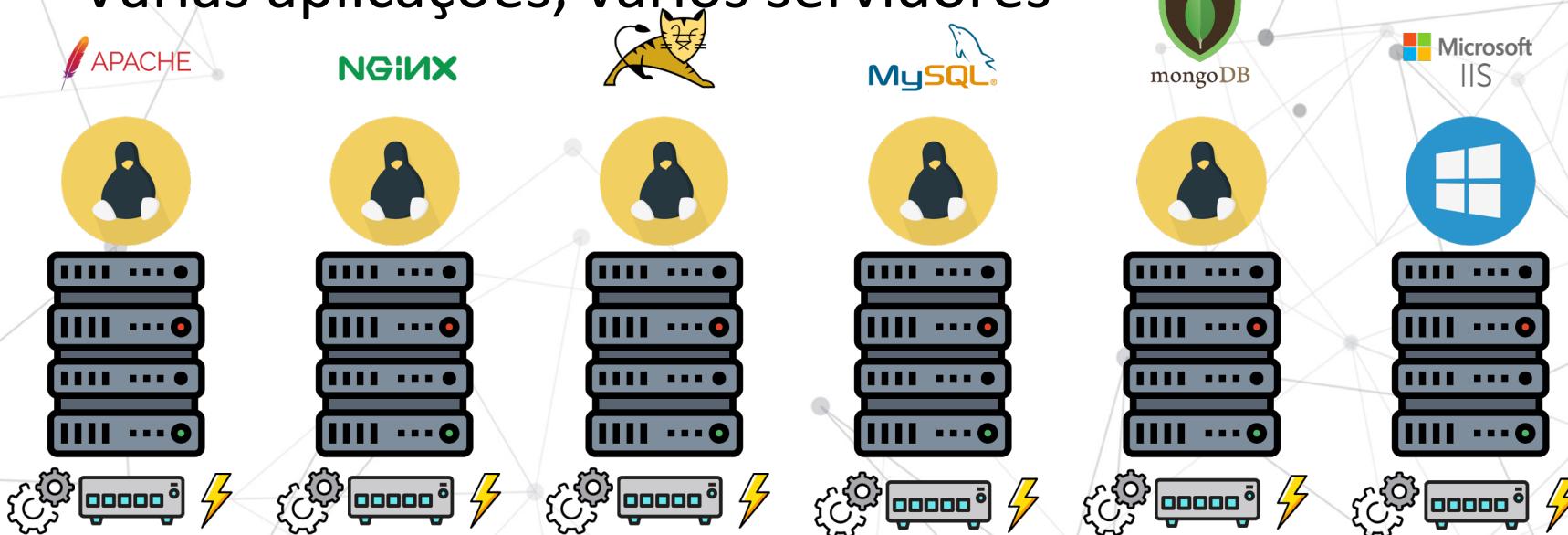
Era da Implantação Tradicional

- No início, as organizações executavam aplicações em servidores físicos.
- Não havia como definir limites de recursos para aplicações em um mesmo servidor físico, e isso causava problemas de alocação de recursos.
- Poderia haver situações em que uma aplicação ocupasse a maior parte dos recursos e, como resultado, o desempenho das outras aplicações seria inferior.

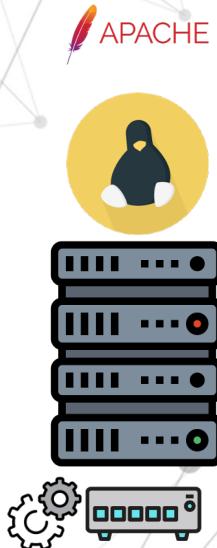
O problema das máquinas virtuais

Como hospedamos aplicações antigamente?

- ▶ Várias aplicações, vários servidores



Capacidade subutilizada!



Carga

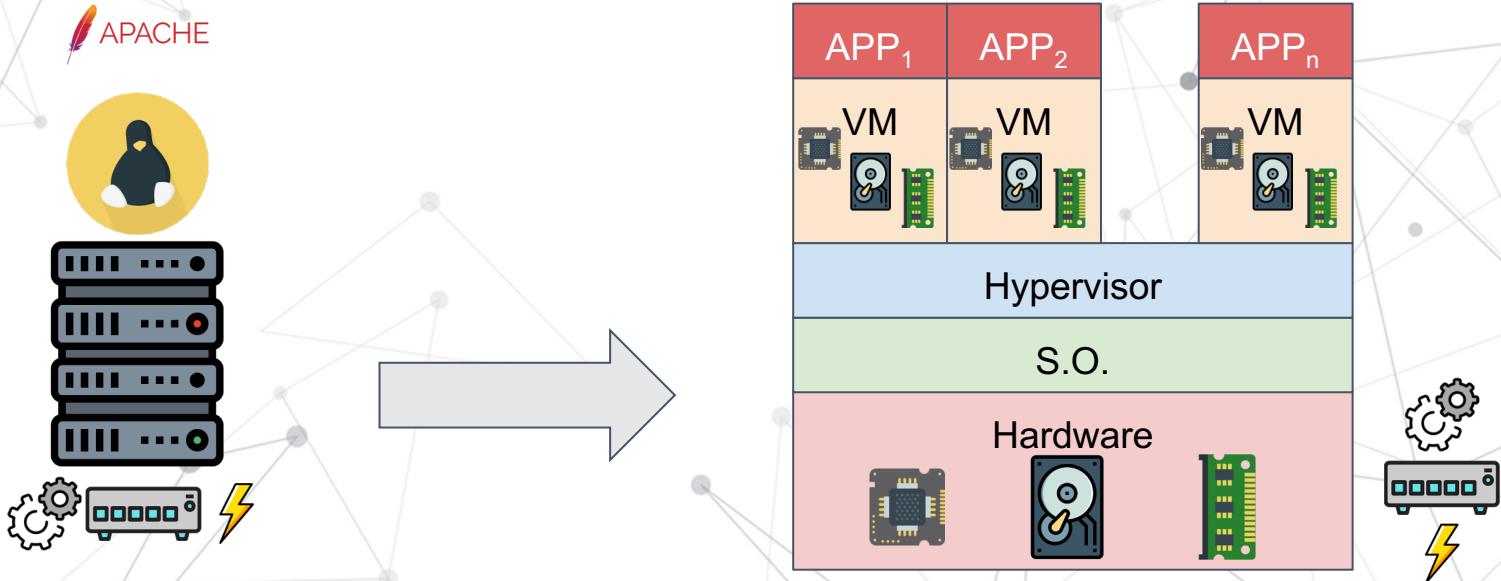
~15%

- **Muito tempo ocioso**
- ▷ **Muitos recursos desperdiçados**

Era da Implantação virtualizada

- Como solução, a virtualização foi introduzida
- Esse modelo permite que você execute várias máquinas virtuais (VMs) em uma única CPU de um servidor físico.
- A virtualização permite que as aplicações sejam isoladas entre as VMs
- ainda fornece um nível de segurança, pois as informações de uma aplicação não podem ser acessadas livremente por outras aplicações.

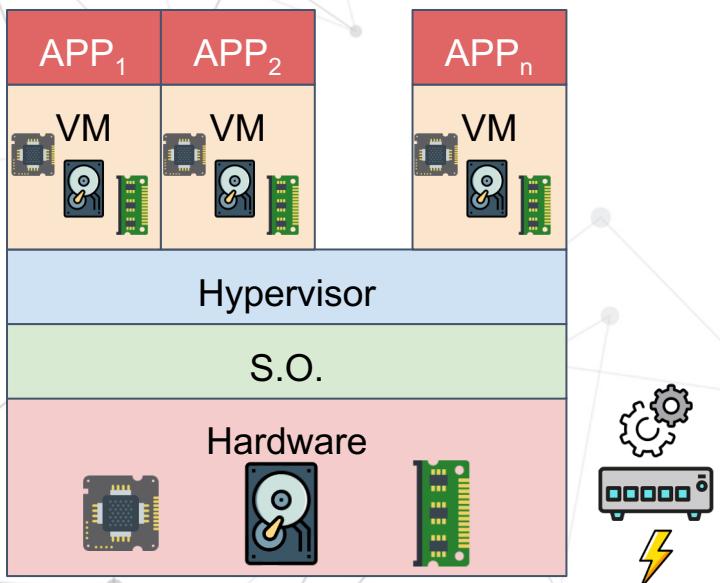
Melhorando a situação: virtualização



Era da Implantação virtualizada

- A virtualização permite melhor utilização de recursos em um servidor físico
- permite melhor escalabilidade porque uma aplicação pode ser adicionada ou atualizada facilmente
- reduz os custos de hardware e muito mais.
- Cada VM é uma máquina completa que executa todos os componentes, incluindo seu próprio sistema operacional, além do hardware virtualizado.

Melhoria no uso dos recursos de infraestrutura

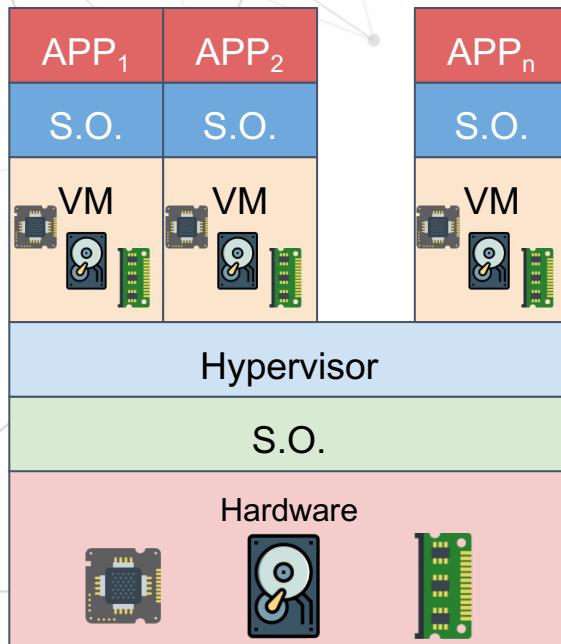


Carga



O problema das máquinas virtuais

Nem tudo são flores... O problema das VM's



6GB de RAM

60GB de Armazenamento

++% de processamento

1GB de RAM

10GB de Armazenamento

% de processamento

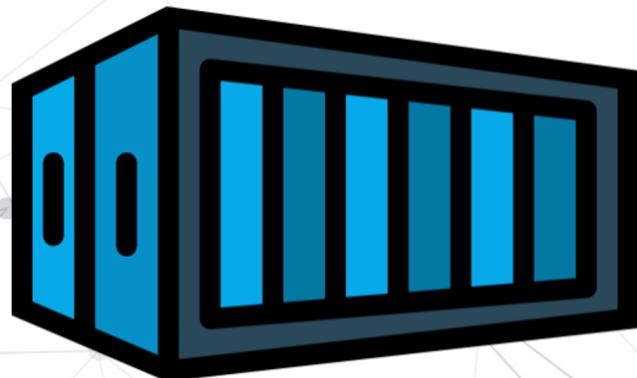
Outros custos de um S.O.

- ▶ Configuração
- ▶ Atualização
- ▶ Segurança

O problema das máquinas virtuais

Como melhorar agora?

A era dos **containers!**



Era da implantação em Contêiner

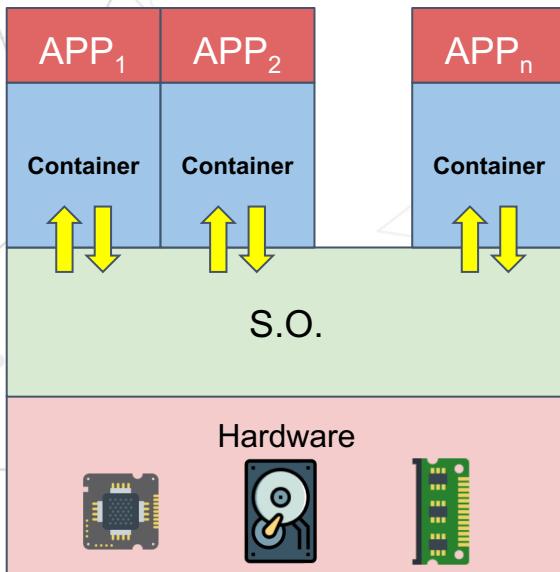
- Contêineres são semelhantes às VMs
- Têm propriedades de isolamento flexibilizados para compartilhar o sistema operacional (SO) entre as aplicações.
- são considerados leves

O que é um Container?

- Trata-se de um ambiente isolado
- Permite encapsular todas as dependências necessárias para rodá-lo, como bibliotecas e o código da aplicação.
- portanto, você não precisa depender do que está atualmente instalado no host.

O que é um container?

- ▷ **Mais leve**
- ▷ **Baixo custo na manutenção de múltiplos S.O.'s**
- ▷ **Mais rápido de subir.**



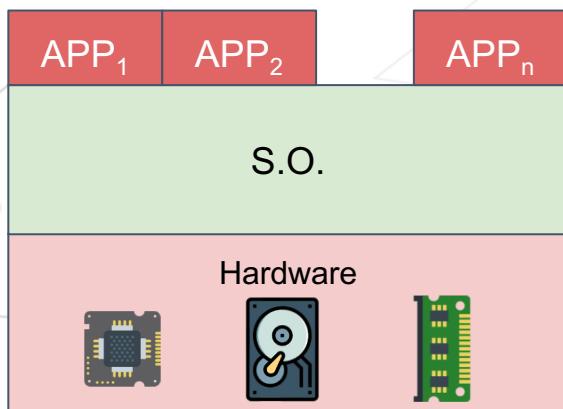
O que é um Container?

- O isolamento e a segurança permitem que você execute vários contêineres simultaneamente em um determinado host.
- Você pode compartilhar contêineres facilmente;

Mas por que precisamos deles?

Os problemas dessa abordagem

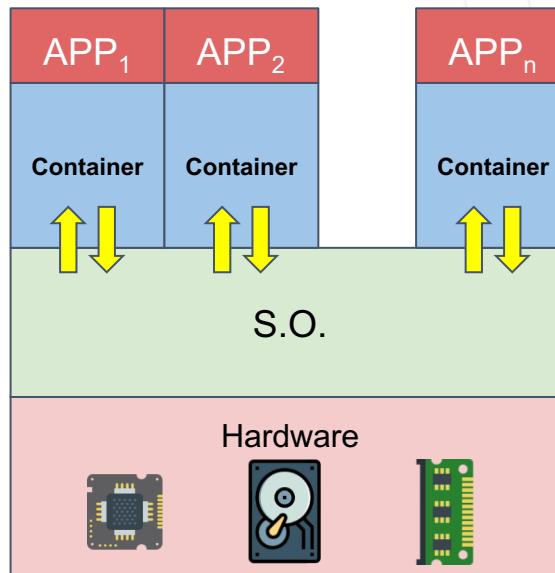
- ▶ Dois apps utilizando a mesma porta de rede?
- ▶ E se um app começar a consumir muito de um recurso, como a CPU?
- ▶ E se cada app precisar de uma versão específica de uma linguagem?
- ▶ E se um app congelar todo o sistema?



Utilizando containers

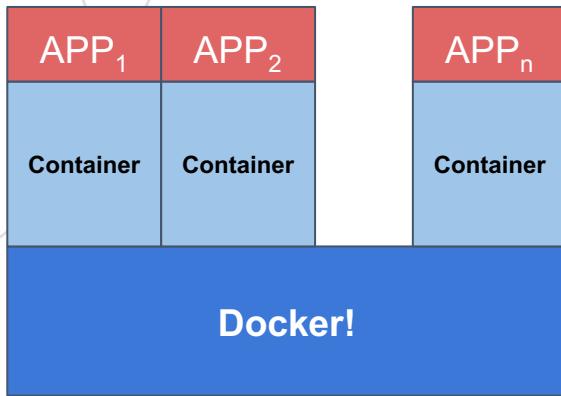
Ganhamos:

- ▷ Melhor controle sobre o uso de cada recurso (CPU, Disco, Rede...)
- ▷ Agilidade na hora de criar ou remover um container
- ▷ Maior facilidade na hora de trabalhar com diferentes versões de linguagens e bibliotecas
- ▷ Mais leves que as VM's



Docker

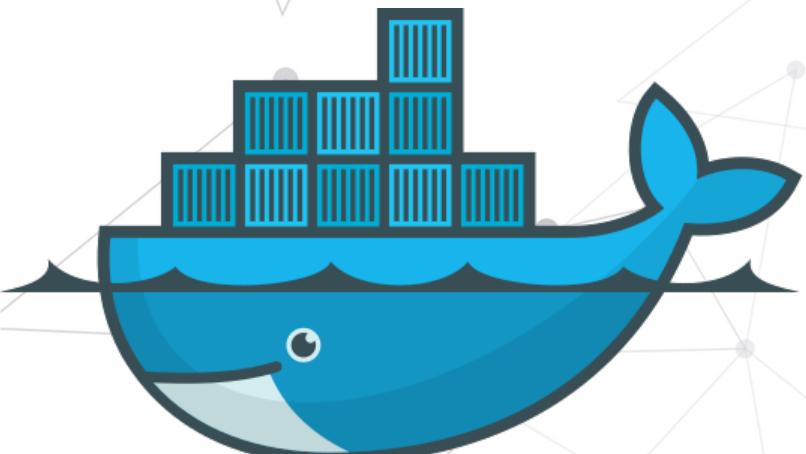
Tecnologias de containers para prover ferramentas modernas para lançar e executar aplicações



Docker Engine

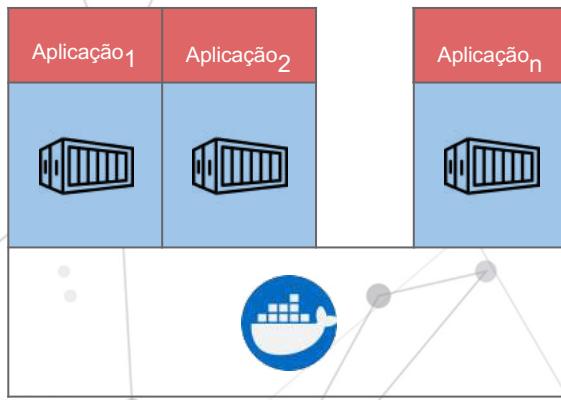
O que é Docker?

O Docker é um software para deploy/implementar uma aplicação usando contêineres.



docker

O que é Docker?

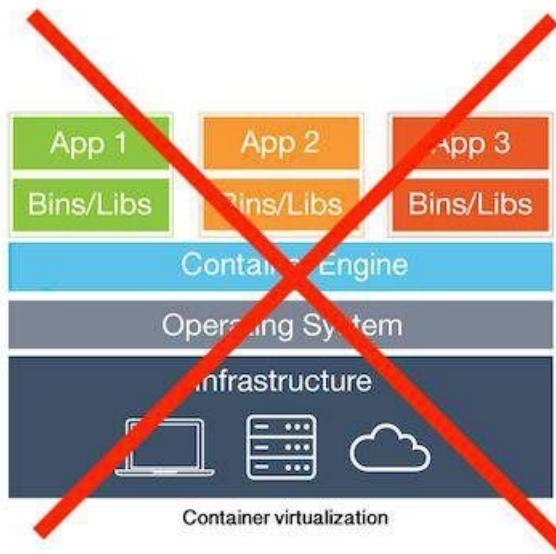
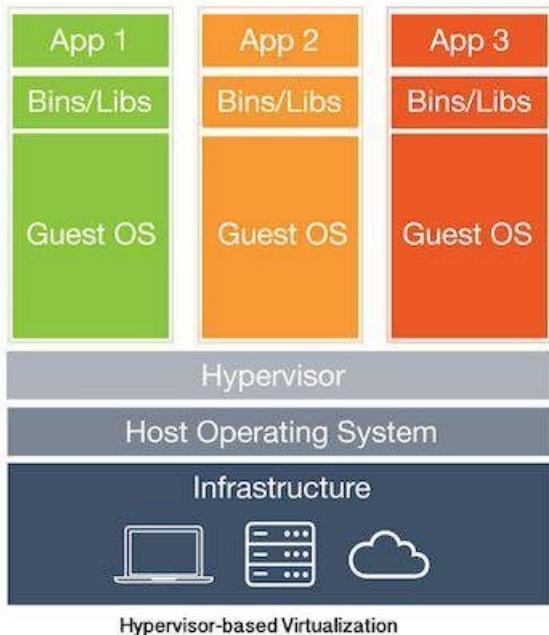


Fonte: [https://pt.wikipedia.org/wiki/Docker_\(software\)](https://pt.wikipedia.org/wiki/Docker_(software))



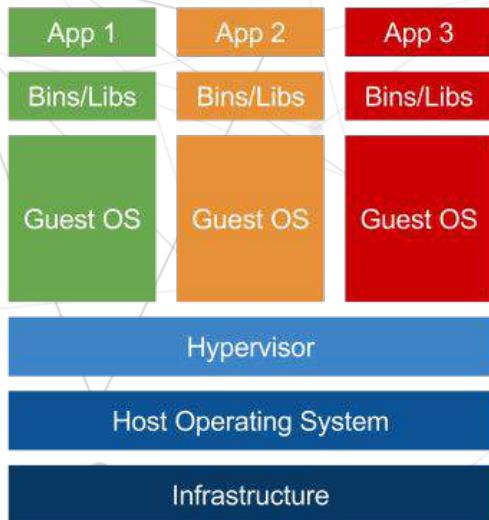
Docker Engine

Docker NÃO é um Hypervisor



<https://blog.mikesir87.io/2017/05/docker-is-not-a-hypervisor/>

Docker NÃO é um Hypervisor



- Apps possuem “paredes” entre elas;
- Docker Daemon é apenas mais um processo.

<https://blog.mikesir87.io/2017/05/docker-is-not-a-hypervisor/>

Visão Geral do Docker

- O Docker permite que você separe seus aplicativos de sua infraestrutura para que você possa entregar software rapidamente.
- Você pode gerenciar sua infraestrutura da mesma forma que gerencia seus aplicativos.
- É possível enviar, testar e implantar código rapidamente, reduzindo o atraso entre escrever código e executá-lo em produção.

A plataforma Docker

- Fornece ferramentas e uma plataforma para gerenciar o ciclo de vida de seus contêineres:
 - Desenvolva seu aplicativo e seus componentes de suporte usando contêineres.
- O contêiner se torna a unidade para distribuir e testar seu aplicativo.
- Quando estiver pronto, implante seu aplicativo em seu ambiente de produção, como um contêiner.

Por que usar o Docker?

- **Entrega rápida e consistente de seus aplicativos**
- Ele simplifica o ciclo de vida do desenvolvimento.
- Permitindo que os desenvolvedores trabalhem em ambientes padronizados.
- Os contêineres são ótimos para fluxos de trabalho de integração contínua e entrega contínua (CI/CD).

Por que usar o Docker?

- Considere o seguinte cenário de exemplo:
 - Os Devs escrevem código localmente e o compartilham usando contêineres Docker.
 - Eles usam o Docker para enviar seus aplicativos para um ambiente de teste e executar testes automatizados e manuais.

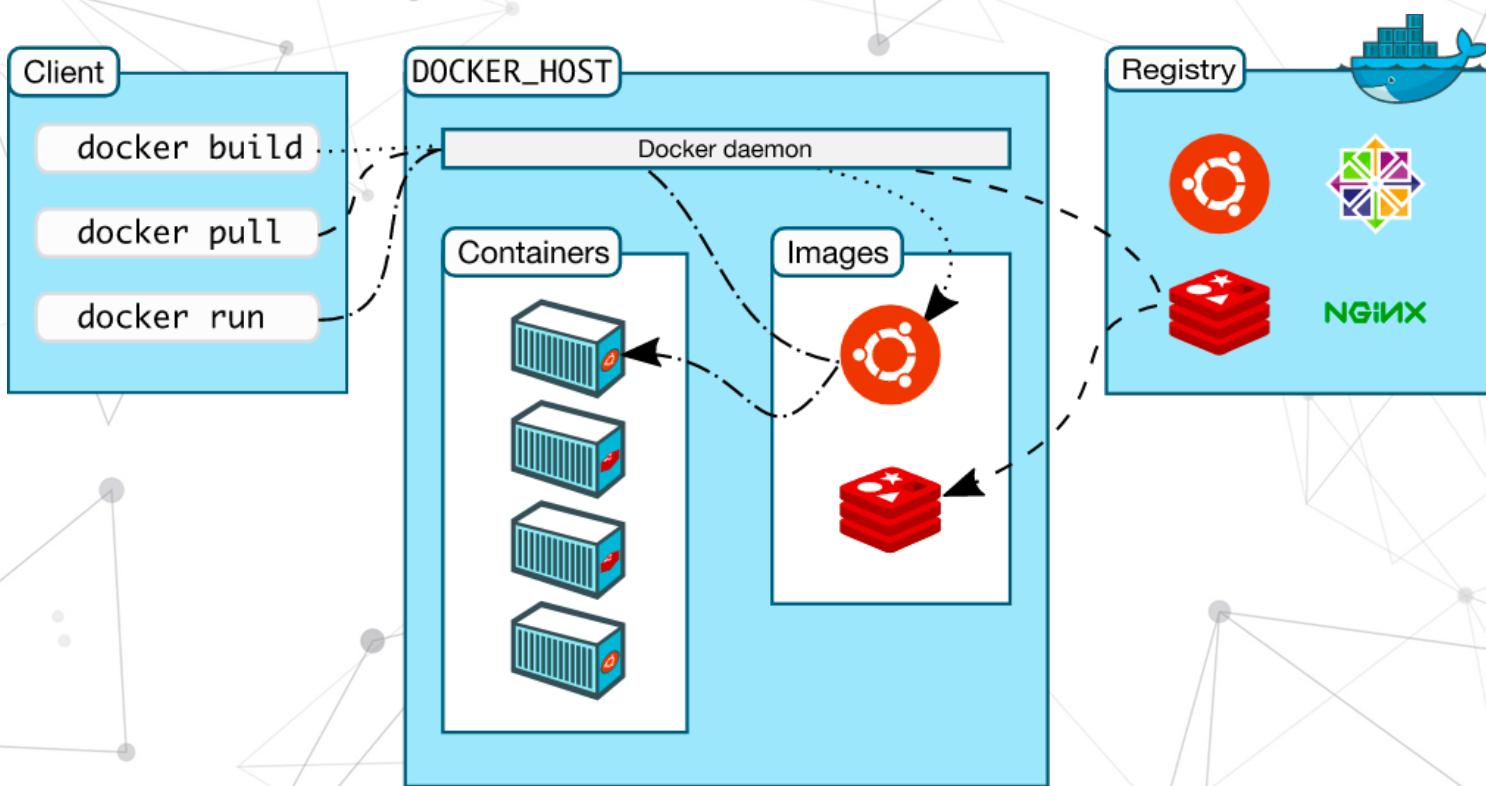
Por que usar o Docker?

- Ao encontrar bugs, eles podem corrigi-los no ambiente de desenvolvimento e reimplantá-los no ambiente de teste para teste e validação.
- Quando o teste é concluído, para obter a correção para o cliente basta enviar a imagem atualizada para o ambiente de produção.

Arquitetura Docker

- Docker usa uma arquitetura cliente-servidor.
- O cliente Docker conversa com o daemon Docker , que faz o trabalho pesado de construir, executar e distribuir seus contêineres Docker.

Arquitetura Docker



Arquitetura Docker

- O cliente Docker e o daemon podem ser executados no mesmo sistema ou você pode conectar um cliente Docker a um daemon Docker remoto.
- O cliente Docker e o daemon se comunicam usando uma API REST, por soquetes UNIX ou uma interface de rede

O daemon do Docker

- O daemon do Docker (`dockerd`) escuta as solicitações da API do Docker e gerencia objetos do Docker, como imagens, contêineres, redes e volumes.
- Um daemon também pode se comunicar com outros daemons para gerenciar os serviços do Docker.

O cliente Docker

- O cliente Docker (docker) é a principal forma que muitos usuários do Docker interagem com o Docker.
- Ao usar comandos como docker run, o cliente envia esses comandos para dockerd, que os executa.
- O cliente Docker pode se comunicar com mais de um daemon.

Docker Desktop

- É um aplicativo fácil de instalar para seu ambiente Mac, Windows ou Linux que permite criar e compartilhar aplicativos e microserviços em contêineres
- Ele inclui o daemon Docker (dockerd), o cliente Docker (docker), Docker Compose, Docker Content Trust, Kubernetes e Credential Helper.

Registros do Docker

- Um registro do Docker armazena imagens do Docker.
- O Docker Hub é um registro público que qualquer pessoa pode usar, e o Docker é configurado para procurar imagens no Docker Hub por padrão.
- Você pode até executar seu próprio registro privado.

Registros do Docker

- Quando você usa os comandos docker pull ou docker run, as imagens necessárias são extraídas do registro configurado.
- Quando você usa o docker push, sua imagem é enviada para o registro configurado.

Objetos Docker

- Ao usar o Docker, você está criando e usando imagens, contêineres, redes, volumes, plug-ins e outros objetos.

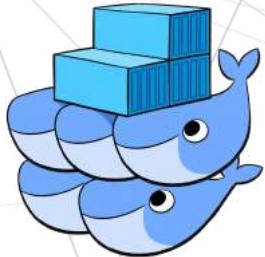
Tecnologias



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 contêineres.

Tecnologias



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.

Tecnologias

Docker Swarm



DOCKER SWARM
Manager
192.168.10.1



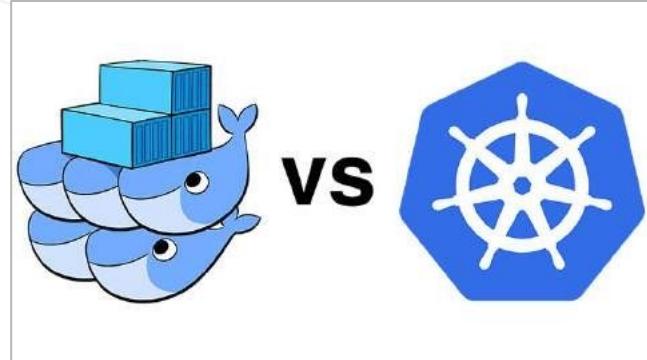
Worker 1
192.168.10.2



Worker 2
192.168.10.3



Worker 3
192.168.10.4



Play with Docker

- Ferramenta online para testes
- Nela é possível testar as diversas funcionalidades que o docker proporciona.
- Ao acessar o site, basta clicar em **+Add New Instance**

Hello World

Vamos iniciar com o Hello World dos containers!

```
docker run hello-world
```

Para verificarmos a versão atual do Docker, basta executarmos

```
docker version
```

Glossário

Imagen: 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

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.

O que são Imagens?

Imagens

- Uma imagem é um modelo somente leitura com instruções para criar um contêiner Docker.
- Por exemplo, você pode criar uma imagem ubuntu, instalar o servidor web Apache e seu aplicativo, bem como os detalhes de configuração necessários para executar seu aplicativo.

Imagens

- Você pode criar suas próprias imagens ou usar outras criadas e publicadas em um registro.
- Para construir sua própria imagem, você cria um Dockerfile com uma sintaxe simples para definir as etapas necessárias para criar a imagem e executá-la.

Imagens

- Cada instrução em um Dockerfile cria uma camada na imagem.
- Quando você altera o Dockerfile e reconstrói a imagem, apenas as camadas que foram alteradas são reconstruídas.
- Isso faz parte do que torna as imagens tão leves, pequenas e rápidas.

Contêiner vs Imagens

- Um contêiner é uma instância executável de uma imagem. Você pode criar, iniciar, parar, mover ou excluir um contêiner usando a API ou CLI do Docker.
- Você pode conectar um contêiner a uma ou mais redes, anexar armazenamento a ele ou até mesmo criar uma nova imagem com base em seu estado atual.

Contêiner vs Imagens

- Por padrão, um contêiner é relativamente bem isolado de outros contêineres e de sua máquina host.
- Um contêiner é definido por sua imagem, com base nas opções de configuração fornecidas a ele ao criá-lo ou iniciá-lo.

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:

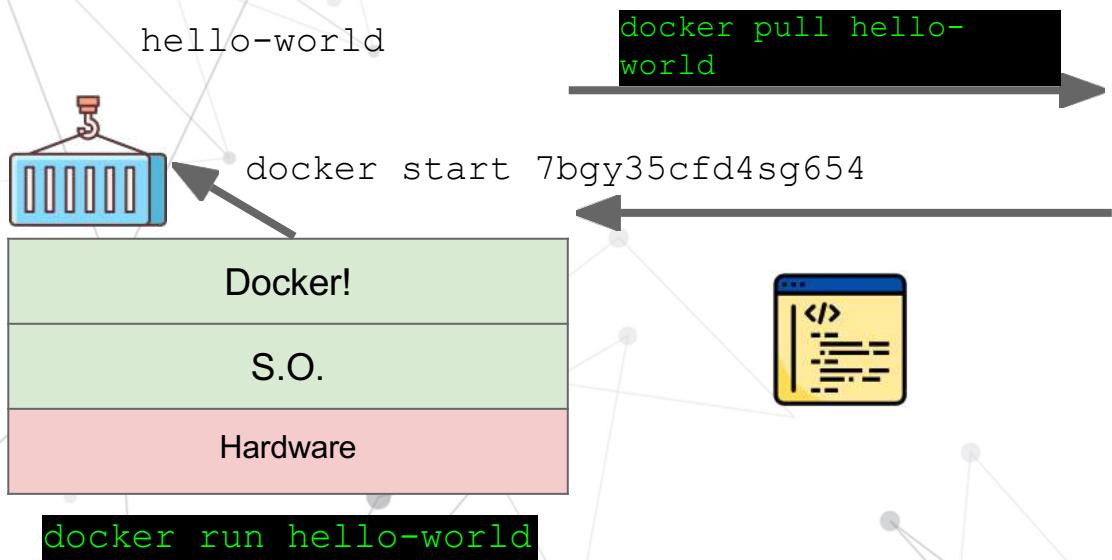
- ▶ Verifica se temos a **imagem** localmente
- ▶ Caso contrário, busca e faz o download no [Docker Store!](#)

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!



Comandos básicos com containers

Vamos tentar executar uma imagem do Alpine

```
docker run alpine
```

Reparam 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
	busybox	latest	19485c79a9bb	6 weeks ago	1.22MB
	alpine	latest	965ea09ff2eb	16 hours ago	5.55MB
	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)  **VERIFIED PUBLISHER**

By Oracle • Updated 20 days ago

Oracle Java 8 SE (Server JRE)

Container Docker Certified Linux x86-64 Programming Languages

couchbase  **OFFICIAL IMAGE**

Updated 16 minutes ago

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

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

Traefik, The Cloud Native Edge Router

Container Linux Windows ARM ARM 64 x86-64 Application Infrastructure

busybox  **OFFICIAL IMAGE**

Updated 18 minutes ago

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

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

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

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:c19173c5ada610a5989151111163d28a67368362762534d8a8121ce95cf2bd5a
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
AMES
[node1] (local) root@192.168.0.23 ~
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
d627d71f316c        alpine              "/bin/sh"          12 seconds ago    Exited (0) 11 seconds ago
vibrant_wilbur
[node1] (local) root@192.168.0.23 ~
$
```

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
```

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
```

Comandos básicos com containers

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

```
docker stop 3hg567caz645
```

Comandos básicos com containers

Pergunta: o comando run sempre cria novos containers?

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

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

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

Exercícios- Comandos básicos

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

Exercícios- Comandos básicos

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

Vimos os dois estados principais de um container

encerra ou
`docker stop`

Running



`docker start`

Stopped



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**.

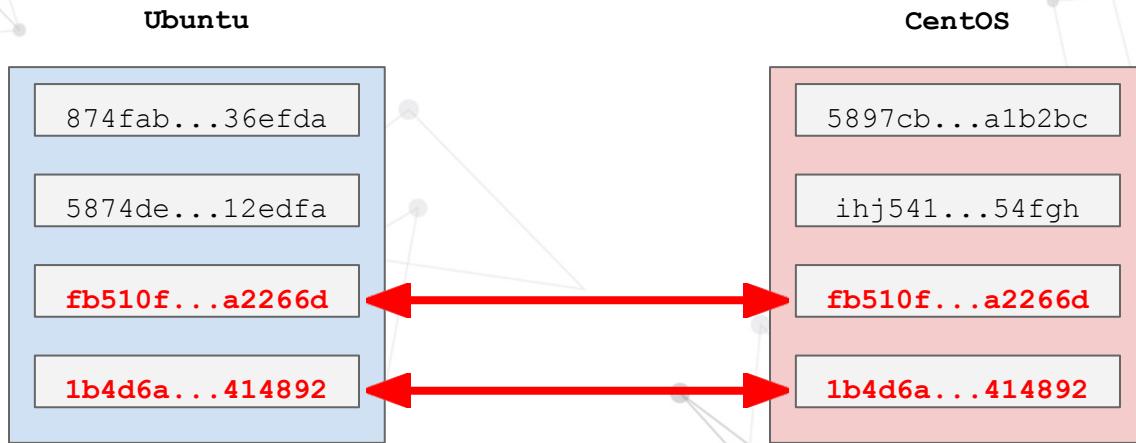
Isso ficou claro quando baixamos a imagem do Ubuntu!

```
$ 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

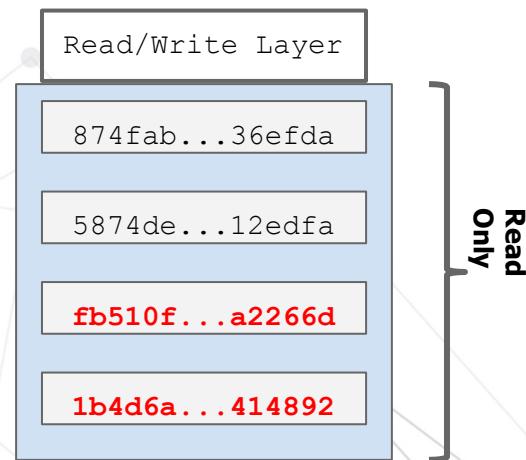
As camadas de uma imagem são apenas para **leitura**!

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!



Read/Write Layer

Read/Write Layer

Read/Write Layer

Ubuntu

874fab...36efda

5874de...12edfa

fb510f...a2266d

1b4d6a...414892

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.

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

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

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
```

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",
    }
  ]
[node1] (local) root@192.168.0.8 ~
$ █
```

Rodando código em um container

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!**

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 univem
```

Criando um Dockerfile

Para executarmos um comando, utilizaremos a palavra chave
RUN.

```
FROM alpine:3.9
```

```
MAINTAINER univem
```

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

Criando um Dockerfile

```
FROM alpine:3.9
```

```
MAINTAINER rmayres
```

```
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`.
`docker build -f Dockerfile -t rmayres/mongo:1.0 .`
- Além disso, indicaremos o nome da imagem com a flag `-t`. Por fim, indicamos onde está o Dockerfile.
`docker build -t rmayres/mongo:1.0 .`

Criando um Dockerfile

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 rmayres/mongo:1.0
```

Subindo a imagem no Docker Hub

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

```
docker pull rmayres/mongo:1.0
```

Executando

Executando o servidor

```
docker run -d -p 27017:27017 -v $HOME/db:/data/db univem/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
```

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?

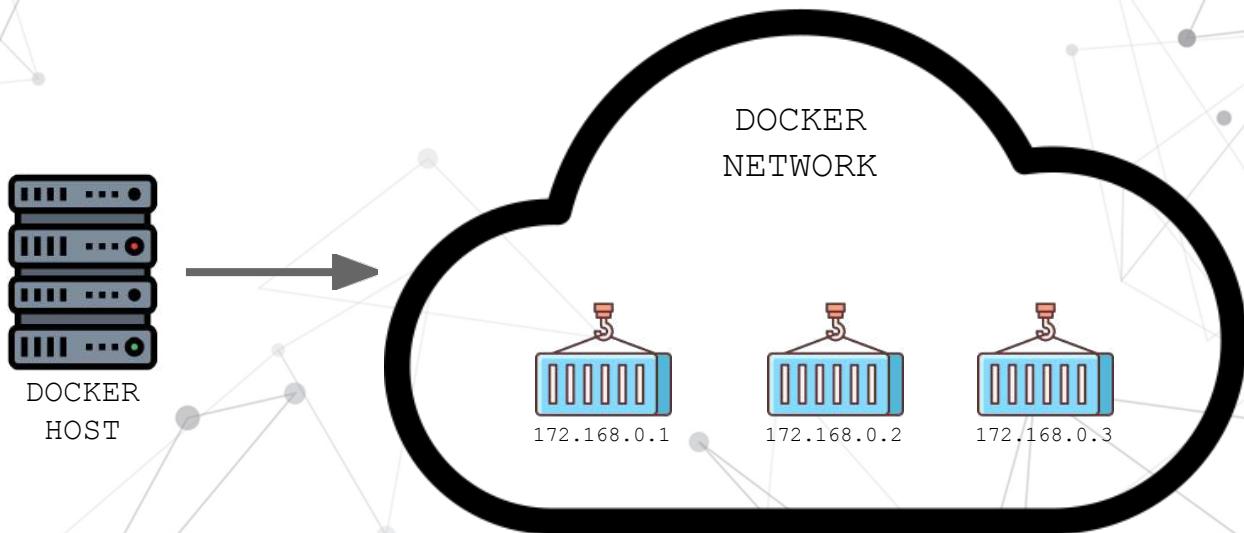
Networking no Docker

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.

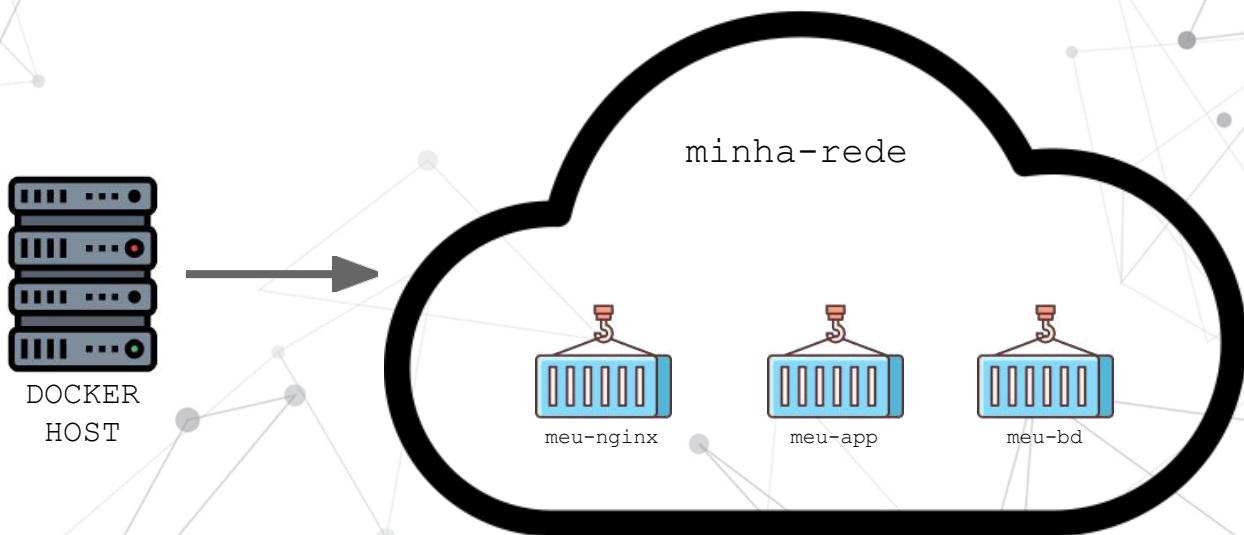
Networking no Docker

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 rmayres/mongo:1.0
```

ou

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



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

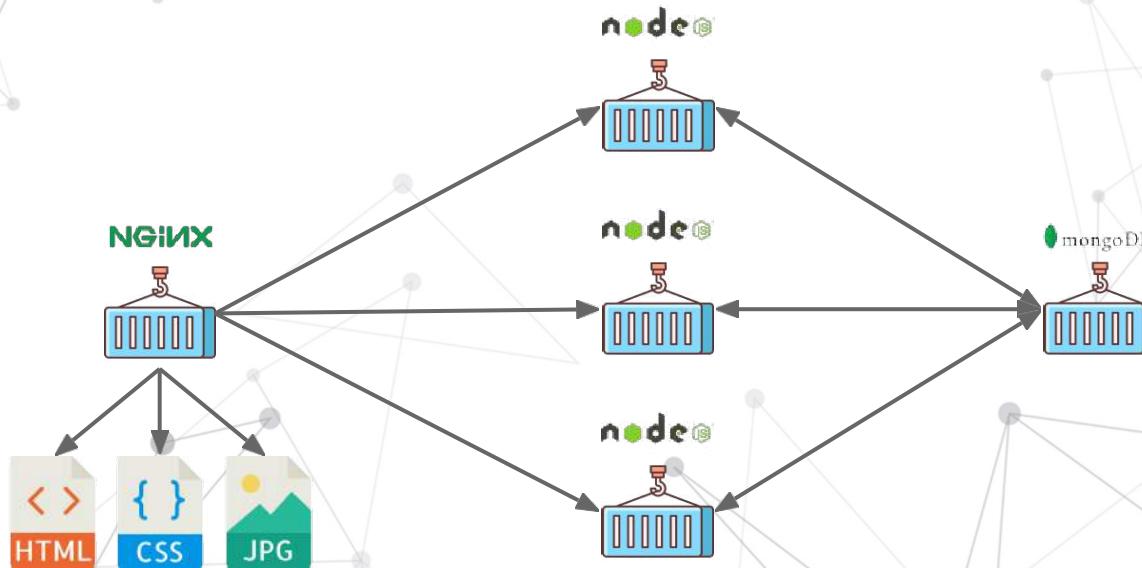
Instanciar um container "manualmente" pode ser problemático: poderíamos **esquecer de alguma flag ou errar alguma parte do comando.**

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

- 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

- Divilde 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
```

O que é Kubernetes?

- O Docker revolucionou a forma como distribuir e rodar a aplicação.
- Porém, uma vez tendo um container rodando, há ainda outras questões a resolver.
- Por exemplo, se um container falha na execução, como garantir que ele sobe novamente de maneira automática (alta disponibilidade)?

O que é Kubernetes?

- E se precisarmos de 4 containers, como automatizar e garantir que sempre teremos os 4 rodando (escalabilidade)?
- E mais, se quisermos algo elástico, como por exemplo ter no mínimo 1 container mas podendo crescer até 5 em execução, como fazer?
- Isso são só algumas questões levantadas, existem vários outras (rede, volumes, monitoramento, atualização etc) que o Docker sozinho não resolve.....

O que é Kubernetes?

- A preocupação ainda cresce se pensarmos em um arquitetura baseada em Microservices.
- Ou seja, a aplicação que estava contida em um único container, agora é dividida em vários outros que precisam interagir entre si de maneira confiável passando por vários hosts.

O que é Kubernetes?

- Reparem aqui a necessidade de ter um "gerenciador de container",
- alguém que "fica de olho" nos containers em execução, garantindo que o sistema como um todo continua rodando como foi planejado.

Kubernetes

- Criado pelo Google com grandes contribuições da Red Hat e hoje é um dos projeto open source mais populares.



Kubernetes

- O nome **Kubernetes** tem origem no Grego, significando *timoneiro* ou *piloto*.
- **K8s** é a abreviação derivada pela troca das oito letras "ubernete" por "8", se tornado *K"8"s*.

Kubernetes

- É ele quem gerencia os contêineres em execução e por isso ele também é chamado de Orquestrador de Containers.
- Através dele podemos definir o estado de um sistema completo, por exemplo baseado em Microservices,
- Permitindo balanceamento de carga, alta disponibilidade, atualizações em lote e rollbacks e muito mais.

Kubernetes

- Vale ressaltar que o Kubernetes não é único Orquestrador de Containers no mercado.
- Há uma solução da própria empresa Docker, chamada de Docker Swarm com o mesmo propósito
- Gerenciar e cuidar os container em execução.

Kubernetes (K8s)

- Ferramenta de código aberto para automatizar implantação, gerenciamento e escalonamento de aplicações containerizadas (i.e., orquestração de containers).
- Objetivo: Automatizar a infraestrutura de aplicações e facilitar seu gerenciamento.

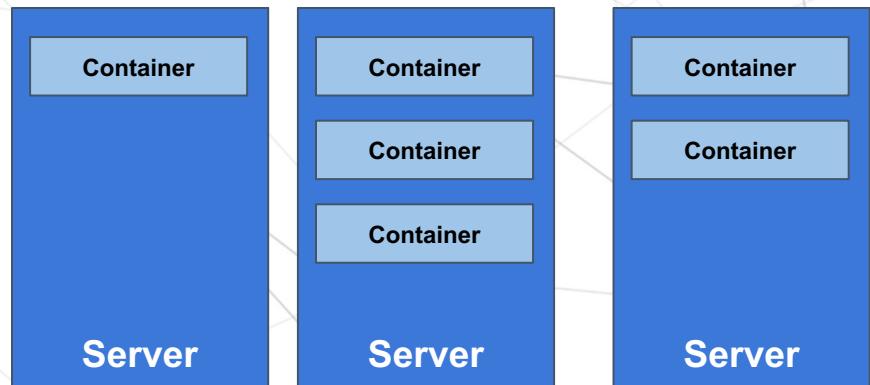


kubernetes

Orquestração

Já vimos o que são containers. Com eles podemos executar uma variedade de componentes de software em um cluster de servidores genéricos, o que ajuda a

- ▶ Prover alta disponibilidade, e
- ▶ Escalar os recursos.



Orquestração

Perguntas:

- ▶ Como garantir que instâncias de um mesmo software sejam espalhadas entre diferentes servidores para atingir alta disponibilidade?
- ▶ Como implantar uma nova versão do software e garantir que ele será atualizado em todo o cluster?
- ▶ Como escalar o software para lidar com aumento da demanda?

Orquestração

Perguntas:

- ▶ Como garantir que instâncias de um mesmo software sejam espalhadas entre diferentes servidores para atingir alta disponibilidade?
- ▶ Como implantar uma nova versão do software e garantir que ele será atualizado em todo o cluster?
- ▶ Como escalar o software para lidar com aumento da demanda?

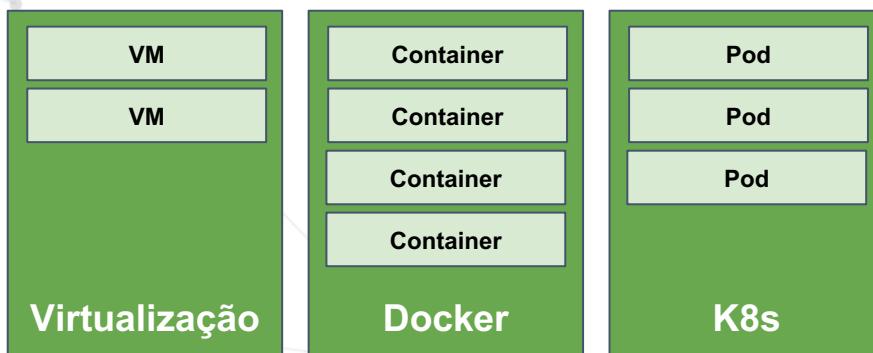
Resposta:

Orquestrar tarefas de gerenciamento! É isso que o K8s faz!



Pod

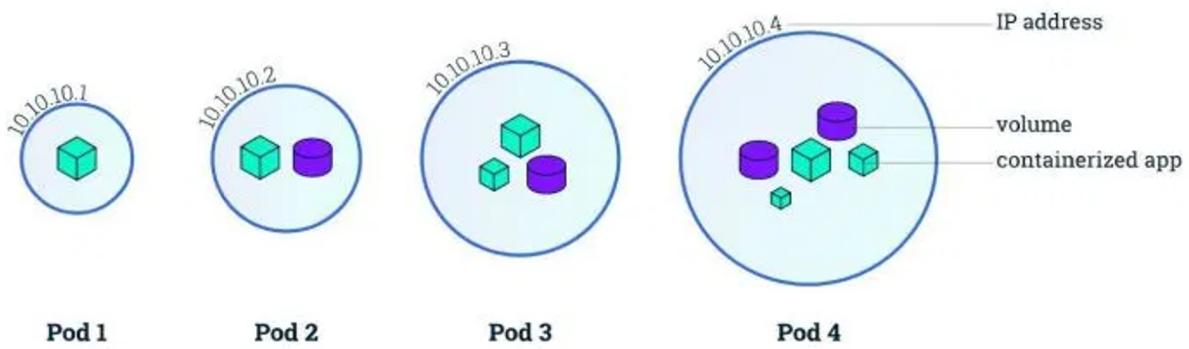
Antes de tudo...



Pod é a menor e mais básica estrutura do K8s

Pod

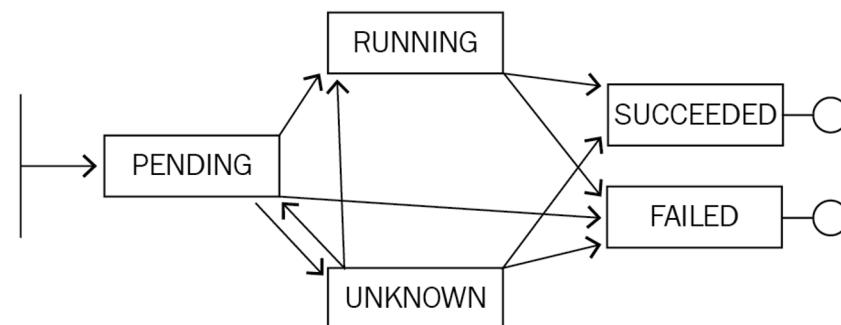
É a menor e mais básica estrutura do K8s, que consiste de um ou mais containers, recursos de armazenamento. e um único endereço IP na rede do cluster K



Ciclo de vida de um Pod

Pending: O Pod foi aceito pelo K8s, mas um ou mais containers ainda não foram criados. Isso inclui tempo de escalonamento e tempo de download da imagem.

Running: O Pod foi alocado em um node e todos os containers foram iniciados. Ele deve estar no status **Running**.
(re)inicializaç

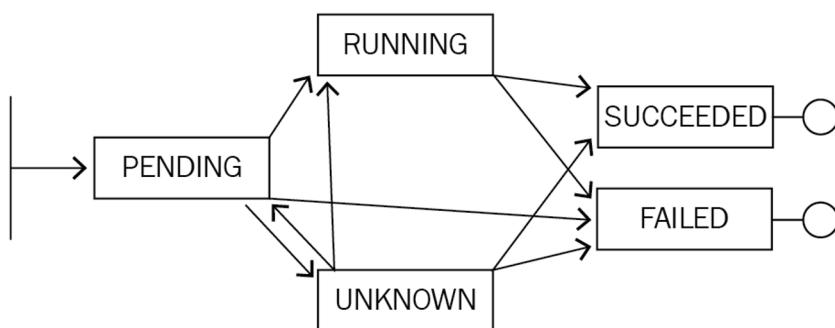


Ciclo de vida de um Pod

Succeeded: Os containers do Pod terminaram com sucesso.

Failed: Os Containers do Pod terminaram, e pelo menos um deles com falha (status != zero ou foi terminado pelo sistema)

Unknown: Pode ser obtido de comunicação



do Pod não
de erro de

Arquitetura do K8s

O K8s pode ser instalado em modo single node ou cluster

O cluster é formado por nós master e worker

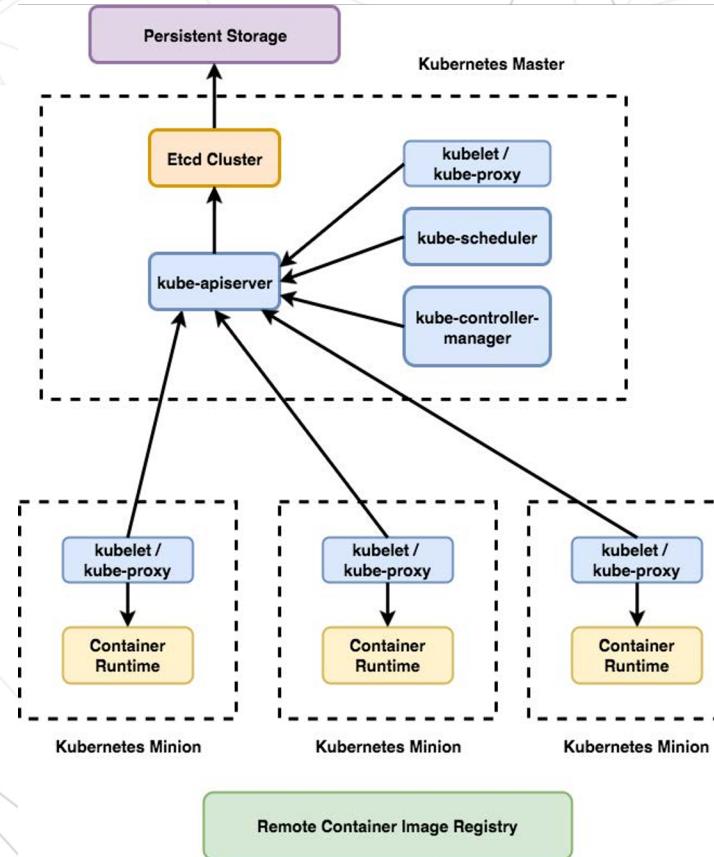
- ▶ O master executa os componentes do plano de controle
- ▶ O worker é quem geralmente executa os containers

Arquitetura do K8s

- **kubeadm**: ferramenta que automatiza grande parte do processo de criação do cluster.
- **kubelet**: componente essencial do K8s que lida com a execução de pods. Atua como um agente em cada node, intermediando as trocas de mensagens entre API server e Docker runtime.
- **kubectl**: CLI de interação com o K8s cluster.

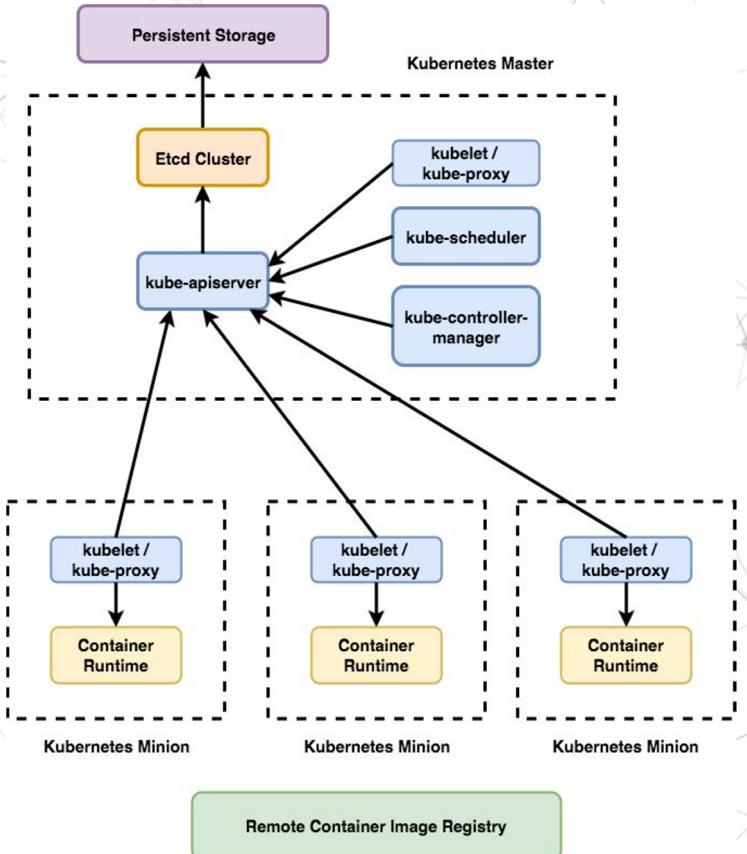
Arquitetura do K8s

- ▶ **etcd:** provê um sistema distribuído e compartilhado para armazenar o estado do cluster
 - ▶ Chave-valor
- ▶ **kube-apiserver:** serve a API do K8s
 - ▶ Baseada em REST
- ▶ **kube-controller-manager:** pacote com diversos componentes de controle



Arquitetura do K8s

- ▶ **kube-scheduler:** escalona os Pods para serem executados nos nodes
- ▶ **kube-proxy:** trata da comunicação entre nodes, adicionando regras ao firewall



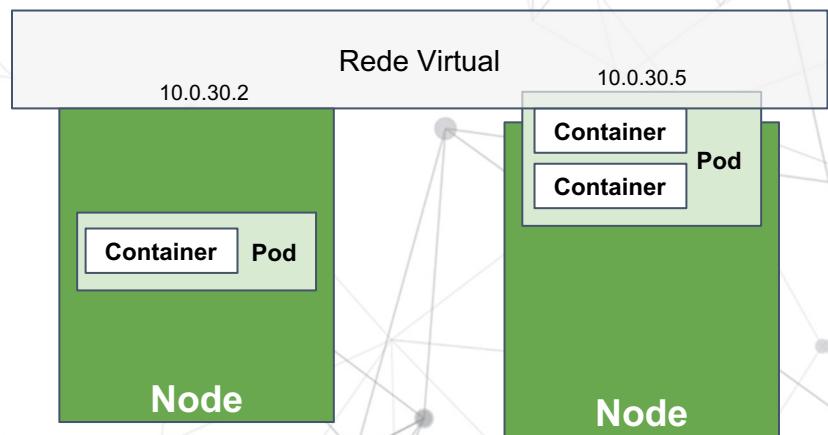
Rede no K8s

O modelo de redes do K8s envolve a criação de redes virtuais no cluster.

Cada Pod do cluster tem um endereço IP único e pode se comunicar com qualquer outro Pod do cluster, mesmo aqueles que são executados em outros nós.

Plugins:

- Flannel
- Weave



Conceitos importantes do K8s

Arquivo de manifesto/especificação

Manifestos podem ser escritos usando YAML ou JSON, mas YAML é mais utilizado porque é mais fácil de ser lido por humanos, além da possibilidade de adicionar comentários.

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuard-demo/kuard-amd64:blue
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Além de Pods

Diversos recursos podem ser manipulados:

- ▶ Pods
- ▶ Nodes
- ▶ Deployment
- ▶ ReplicaSet
- ▶ Service

Eles são considerados objetos no K8s

Kubectl e as maneiras de interação

Há duas maneiras básicas de interagir com o Kubernetes:

- ▶ Imperativa: através de diversos parâmetros do *kubectl*
 - ▶ Diz ao K8S o que fazer
 - ▶ Boa para usar quando se está aprendendo, para fazer experimentos interativos ou debugar serviços em produção.
- ▶ Declarativa: escrevendo manifestos e os usando com o comando *kubectl apply*.
 - ▶ Diz ao K8s o que você quer
 - ▶ Melhor para implantar serviços de maneira a facilitar a reprodutibilidade.
 - ▶ Recomendada para gerenciar aplicações K8s em produção

Kubectl e as maneiras de interação

- Abordagem Imperativa:
- Kubectl Get, Describe, Delete podem ser usados com quaisquer recursos:
 - Pods
 - Nodes
 - Deployment
 - ReplicaSet
 - Service

kubectl get pods
kubectl get nodes
kubectl get services

kubectl describe pod <Nome do Pod>
kubectl describe service <Nome do Service>

kubectl delete pod <Nome do Pod>
kubectl delete deployment <Nome do Deployment>

Kubectl e as maneiras de interação

Abordagem Imperativa:

- ▷ Kubectl Create
- ▷ Kubectl Run
- ▷ Kubectl Scale
- ▷ Kubectl Expose
- ▷ Kubectl Exec
- ▷ Kubectl Copy
- ▷ Kubectl Logs

Kubectl e as maneiras de interação

Abordagem Declarativa:

- ▷ `kubectl apply -f <arquivo.yaml>`
- ▷ `kubectl apply -f <arquivo1.yaml> <arquivo2.yaml>`
- ▷ `kubectl apply -f <folder>/`

Em caso de mudanças no arquivo, *kubectl apply* atualiza os recursos

Criando um Pod

```
kubectl run nginx --generator=run-pod/v1 --  
image=nginx
```

```
cat << EOF | kubectl create -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx-pod  
spec:  
  containers:  
    - name: nginx-container  
      image: nginx  
EOF
```

Exportando manifesto

1. Salva manifesto de um Pod

```
kubectl get pod my-pod -o yaml > my-pod.yaml
```

2. Salva manifesto sem informações específicas do cluster

```
kubectl get pod my-pod -o yaml --export > my-pod.yaml
```

Namespace

K8s usa namespaces para organizar objetos no cluster através de uma divisão lógica (como se fosse uma pasta).

Por padrão, kubectl interage com o namespace padrão (default).

Para usar um namespace específico (diferente do padrão), pode-se usar a flag `--namespace=<nome>`, ou ainda `-n <nome>`.

Para interagir com todos os namespaces, pode-se passar a flag `--all-namespaces` para o comando.

Namespace

1. Criar namespace

```
kubectl create namespace dev
```

```
kubectl create namespace prod
```

1. Listar namespaces

```
kubectl get namespaces
```

1. Remover namespace

```
kubectl delete namespace dev
```

Namespace

4. Filtrar Pods por namespace (opção 1)

```
kubectl get pods --namespace=teste
```

4. Filtrar Pods por namespace (opção 2)

```
kubectl get pods -n teste
```

4. Listar Pods de todos os namespaces

```
kubectl get pods --all-namespaces
```

Labels

Um Label é um par chave-valor do tipo string.

Todos os recursos/objetos K8s podem ser rotulados.

1. Equality-based requirement

environment = production

tier != frontend

1. Set-based requirement

environment in (production, qa)

tier notin (frontend, backend)

Labels

Um Label é um par chave-valor do tipo string.

Todos os recursos/objetos K8s podem ser rotulados.

1. Mostrar labels dos recursos:

```
kubectl get pods --show-labels
```

2. Deletar Pods que têm label run=myapp

```
kubectl delete pods -l environment=production,tier=frontend
```

```
kubectl get pods -l 'environment in (production),tier in (frontend)'
```

2. Atribuir label

```
kubectl label deployment nginx-deployment tier=dev
```

ATIVIDADE

- Realize os testes feitos em aula, documente-os em um arquivo e gere o pdf com os prints das evidências de execução.

Obrigado!