



Leonardo Lima

Bacharel Sistemas de Informação

Especialização em Desenv. Web e Apps

Pesquisador tecnológico - CIAg

Docker e Kubernetes

Instalação do docker:

- Windows Pro / Mac: Docker Desktop
<https://www.docker.com/products/docker-desktop>
- Windows Home: Docker toolbox
<https://github.com/docker/toolbox/releases>
- Linux: `curl -fsSL https://get.docker.com -o get-docker.sh && sh get-docker.sh`

Ferramentas extras que vamos utilizar:

- Node: <https://nodejs.org/>
- Editor de texto: [VSCode](#)
 - Extensão Docker

Entendendo o Docker

Qual problema o Docker resolve?

Apresentação feita em 2013 por Solomon Hykes, fundador da Docker

Nos primórdios do Docker, as pessoas perguntavam muito duas coisas:

- O que é esse tal de docker?
- O que é esse tal de container?

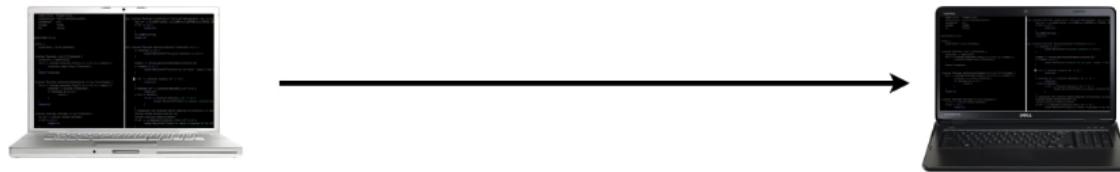
Para respondê-las, Hykes achava mais simples fazer uma demo utilizando o Docker no terminal.

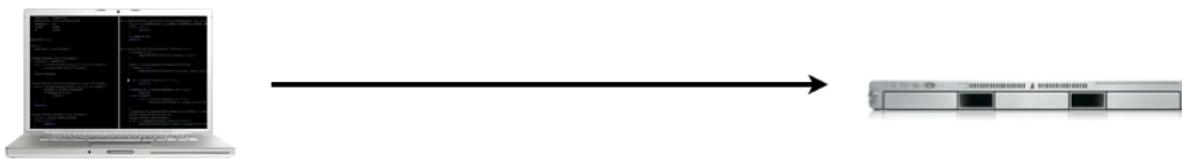
Com o tempo ele decidiu explicar o **porquê** do Docker.

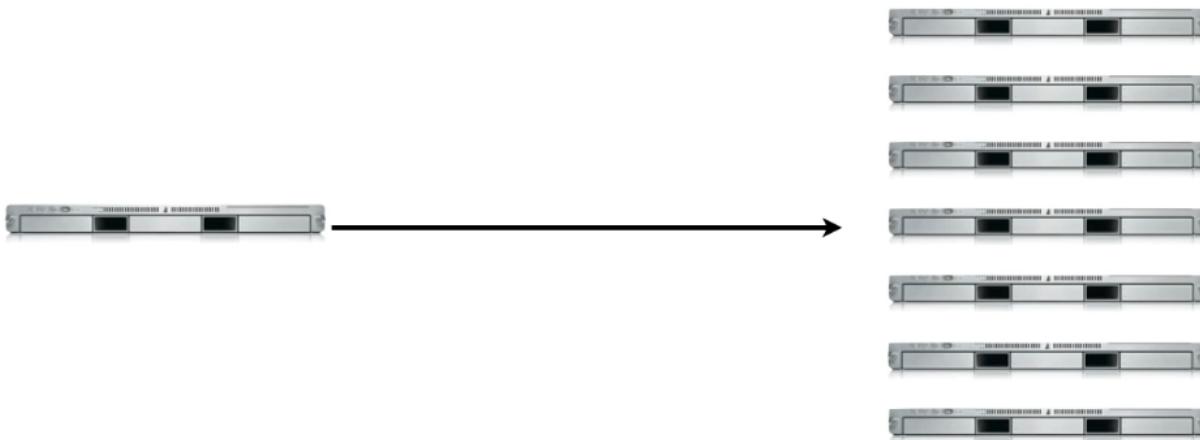
Na época ele acreditava que essa tecnologia afetaria nossa indústria, mudando a forma como criamos e distribuímos software.

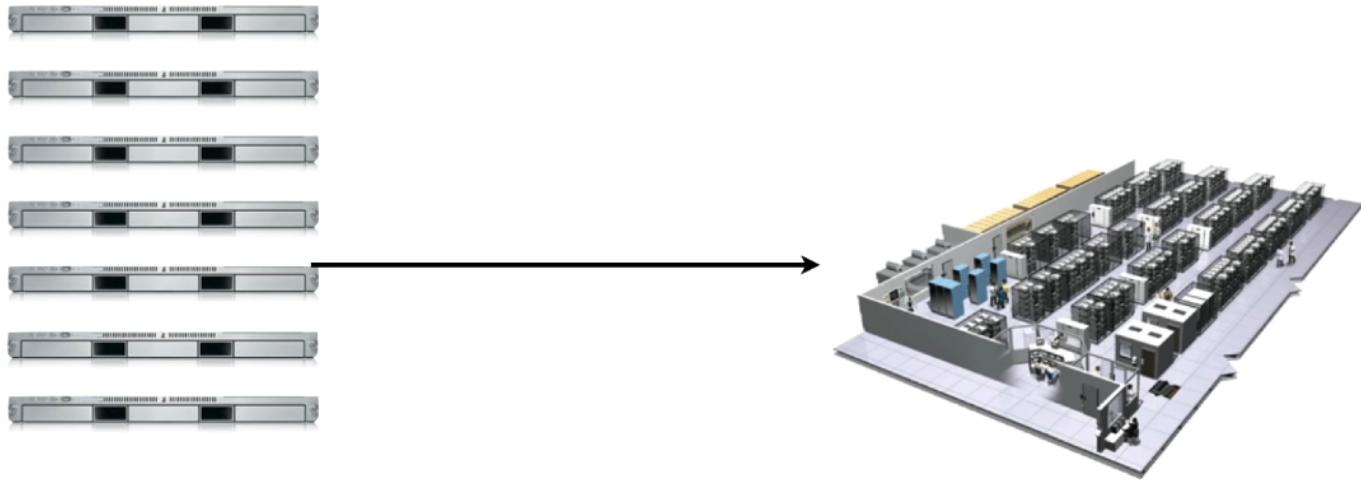
Qual problema o Docker resolve?

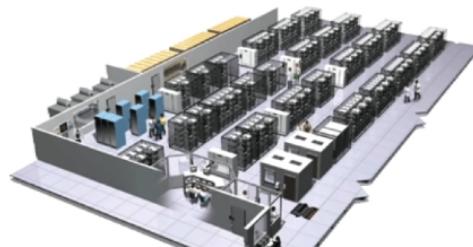
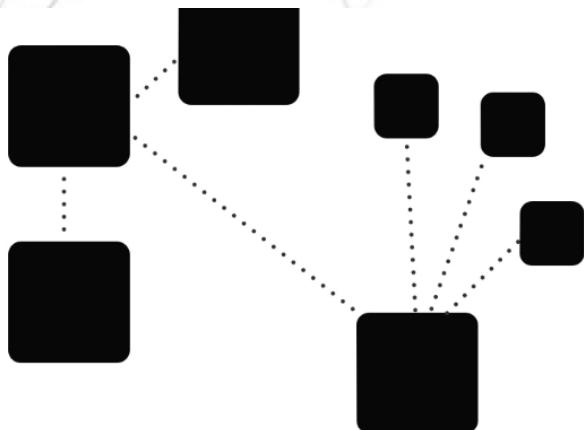
Enviar software da A para B de forma confiável e automática.











Arquiteturas estarem ficando mais complexas foi um dos fatores que complicou essa tarefa.

A popularização da arquitetura orientada à serviços (SOA) e os microserviços criaram um cenário onde temos diversas linguagens trabalhando juntas, utilizando diversos tipos de banco de dados e dependendo de outros diversos componentes.

Os desenvolvedores começam a ter que lidar com essa stack de software cada vez mais complexa.

É necessário executá-la na máquina de desenvolvimento, subir para o ambiente de homologação e depois subir para produção em uma cloud.

Uma stack que já estava ficando complexa agora tem que ser entregue em diferentes ambientes e com uma frequência maior.

Static website

nginx 1.5 + modsecurity + openssl +
bootstrap 2

Background workers

Python 3.0 + celery + pyredis + libcurl + ffmpeg
+ libopencv + nodejs + phantomjs

User DB

postgresql + pgv8 + v8

Analytics DB

hadoop + hive + thrift + OpenJDK

Queue

Redis + redis-sentinel

API endpoint

Python 2.7 + Flask + pyredis + celery + psycopg2
+ postgresql-client

Development VM

QA server

Customer appliances

Production cloud

Disaster recovery

Contributor's laptop

Cada componente da stack tem suas peculiaridades.

Temos linguagens diferentes, versões de linguagens diferentes, utilização de libs específicas...

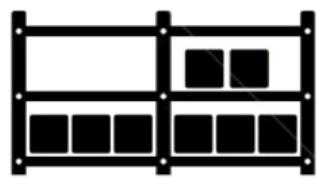
E tudo isso é responsabilidade da equipe.

Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?
	Development VM	Customer appliances	QA server	Disaster recovery	API endpoint	Contributor's laptop	Production cloud

Enviar software da A para B: uma tarefa frágil e trabalhosa.

Quem já falou ou já ouviu falar a frase: "Na minha máquina funciona"?

Começaram a procurar exemplos de outros ramos que resolveram problemas similares.





?

?

?

?

?

?



?

?

?

?

?

?



?

?

?

?

?

?



?

?

?

?

?

?



?

?

?

?

?

?



?

?

?

?

?

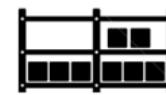
?



Durante muito tempo a indústria de entregas teve este problema.

Na década de 50, várias empresas se juntaram e concordaram em um padrão.

Uma "caixa" onde as dimensões, peso e funcionamento eram padronizados.



O container mudou a indústria de entregas.

Agora, existe uma separação de responsabilidades:

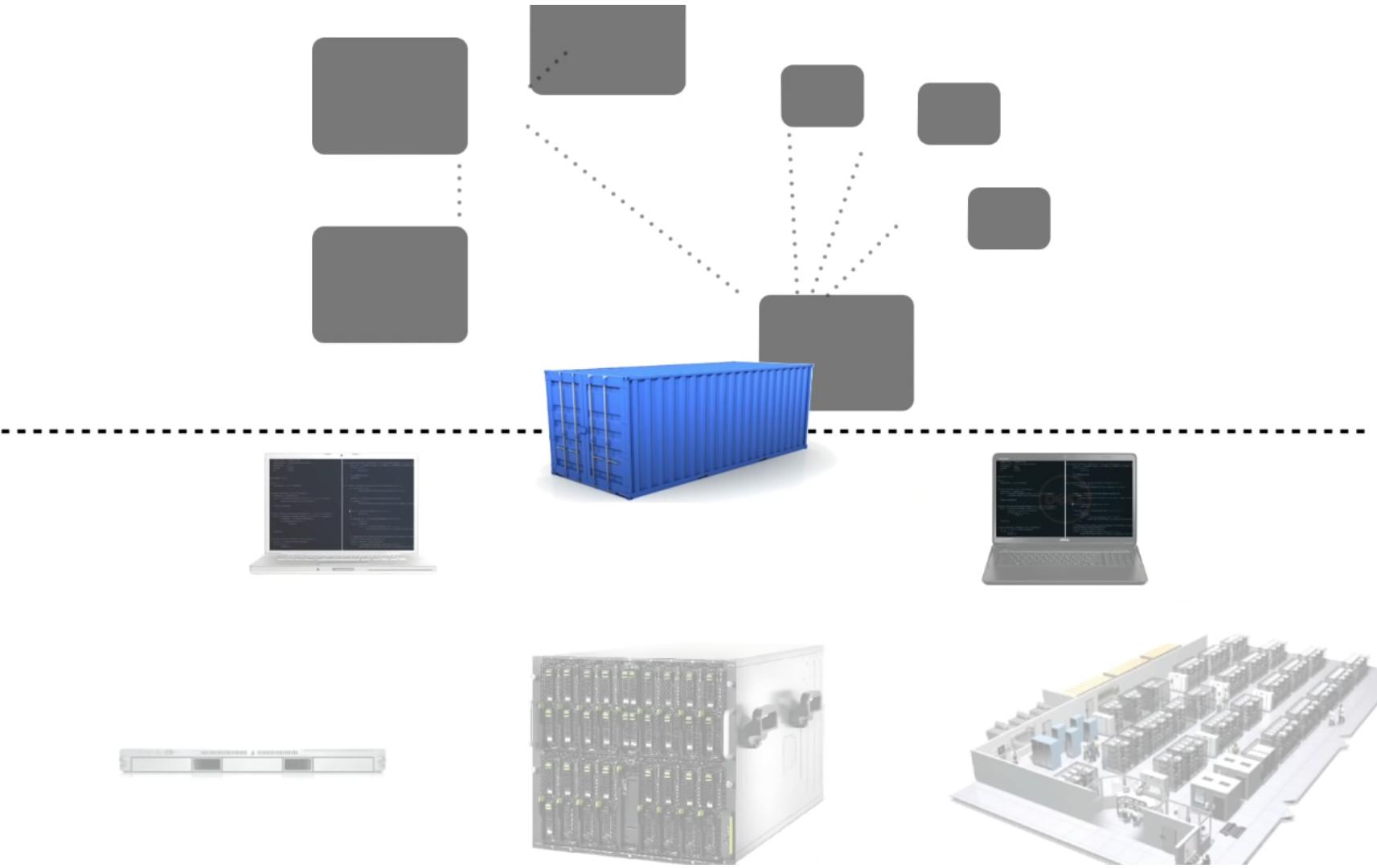
- Se eu quiser enviar meu produto para o outro lado do mundo, basta colocá-lo dentro de um container.
- As empresas de transporte agora não precisam mais se preocupar com as peculiaridades de cada bem entregue. Elas só precisam saber transportar containers.

A automação para transportar café pelo mundo é melhor e mais confiável que as ferramentas que nós usamos para transportar software de um computador para outro

Hykes, 2013

A indústria de software poderia copiar essa ideia.

Desenvolvedores também deveriam ter uma solução semelhante, onde ele coloca seu software em uma "caixa".



Mas por que isso não foi feito antes?

Algumas linguagens já tinham algumas tecnologias bem semelhantes a containers:

- Java com o .jar
- Python com o virtualenv

De certa forma, estas soluções eram incompletas.

Quando trabalhamos com jars ou virtualenvs, nós colocamos todas as dependências do nosso código em um container.

Mas, ainda assim, faltam as dependências do sistema.

E as máquinas virtuais?

VMs permitem compartilhar uma máquina inteira...

Mas, o problema é que elas juntam muita coisa. Você precisa definir o armazenamento, a memória, a rede.

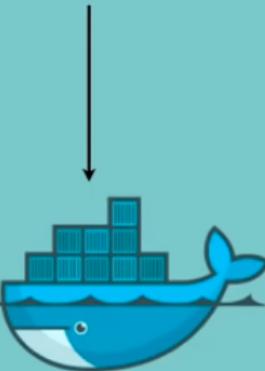
E muitas das vezes o desenvolvedor não precisa definir todos esses parâmetros.

VMs acabam sendo pesadas, principalmente para o ambiente de desenvolvimento.

A ideia do Docker era criar algo "no meio": entregar todas as dependências do software e do sistema, sem todo o peso de uma VM.

Eles conseguiram isso graças a uma funcionalidade do kernel do linux: ela permite criar um processo completamente isolado de outros processos.

Developer builds containers



Infrastructure deploys containers

Containers não são VMs

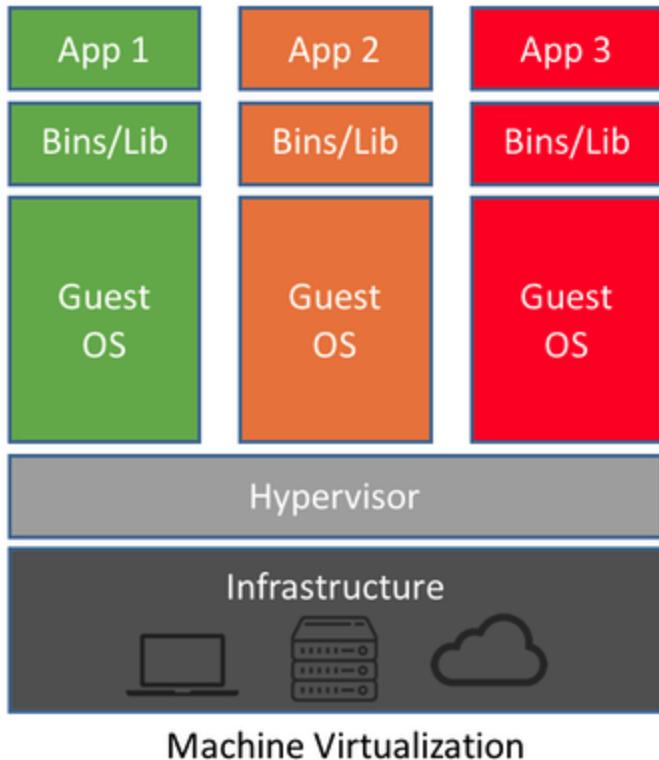
Containers não são VMs

É comum fazer a associação entre containers e VMs, mas eles tem diferenças fundamentais.

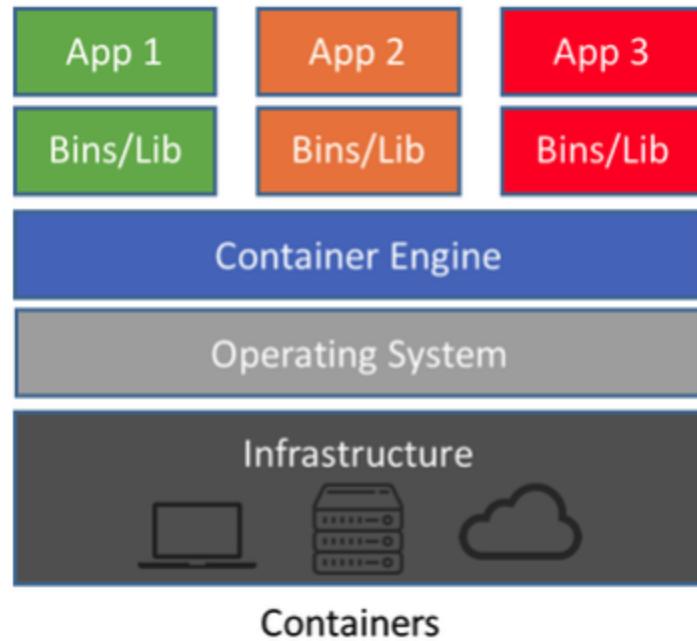
Mike Coleman utiliza a analogia entre casas (VMs) e apartamentos (containers) pra entender essas diferenças:

- Casas são auto-contidas. Possuem infra-estrutura própria, como encanamentos, eletricidade, controle de temperatura. Casas geralmente são completas: possuem banheiro, quarto, sala, cozinha, lavanderia...
- Apartamentos possuem infra-estrutura compartilhada. O prédio compartilha a eletricidade, encanamentos, ar-condicionado central. Apartamentos podem ser bastante enxutos, como no caso dos studios.

Arquitetura VM



Arquitetura Container



Executando containers

Executando containers

Sintaxe básica:

```
docker run <params> <image-name>
```

Hello, world!

```
docker run hello-world
```

Saída:

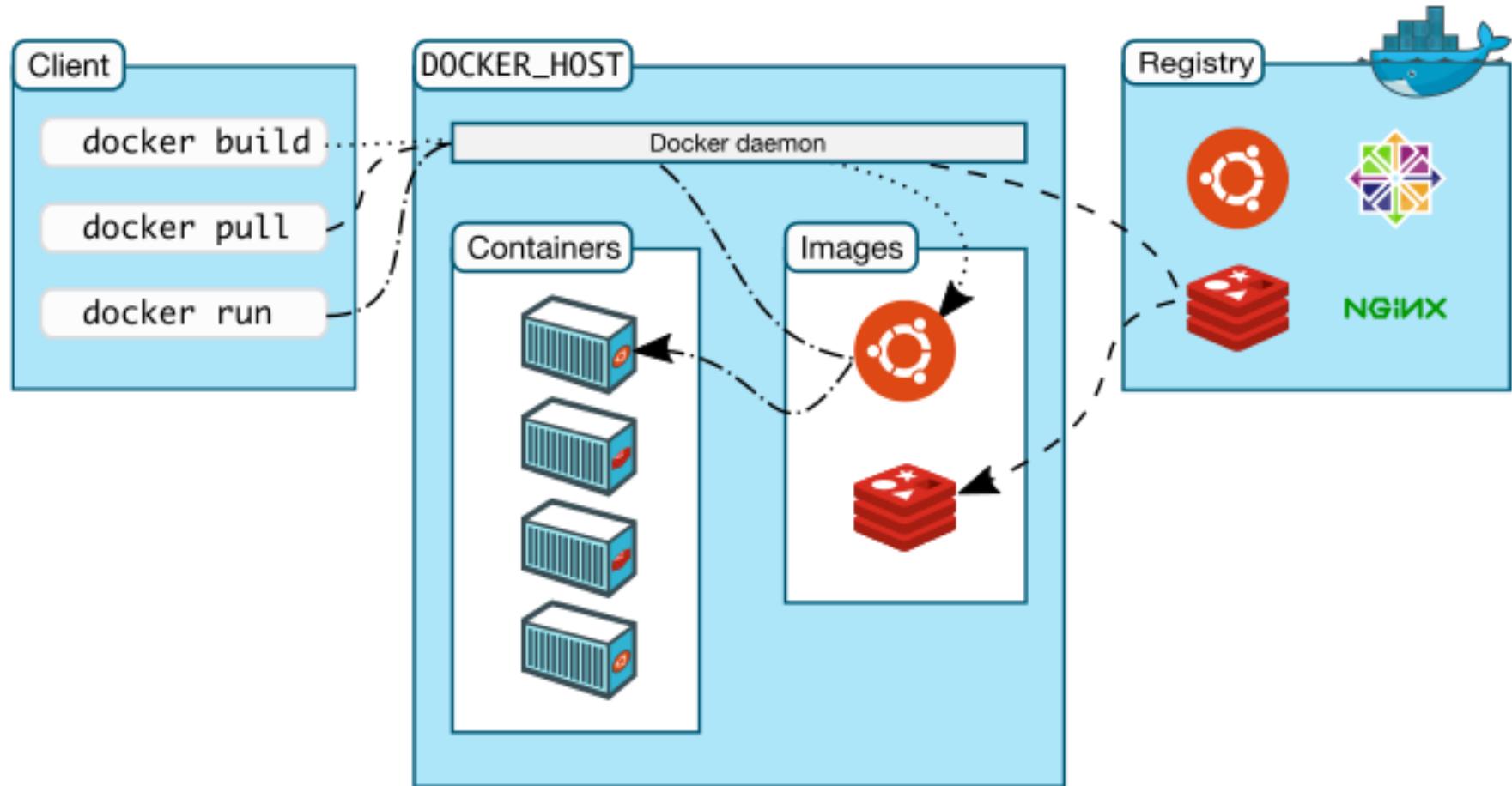
```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:5122f6204b6a3596e048758cabba3c46b1c937a46b5be6225b835d091b90e46c
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

...

...



Container

O container é uma instância em execução de uma imagem

Um container é composto por:

- Uma imagem
- Um ambiente de execução
- Um conjunto padrão de instruções

É um conceito emprestado do mundo da logística, onde foi criado como um padrão de distribuição de bens mundialmente.

O Docker define um padrão para distribuição de software.

Imagen

Ao utilizar o comando `docker run hello-world`, estamos executando um container a partir da imagem `hello-world`.

Vamos analisar a saída do comando `run`:

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

Podemos perceber que primeiro o Docker tenta encontrar uma imagem localmente. Quando ela não é encontrada, ele automaticamente baixa ela de outro local.

Imagen x container / Classe x objeto

Fazendo uma comparação com a programação orientada a objetos, a imagem seria equivalente a uma classe e o container o equivalente a um objeto.

Registro

Registro é um serviço hospedado que contém **repositórios de imagens**.

O registro padrão é chamado **Docker Hub**, e pode ser acessado através do endereço hub.docker.com ou pelo comando `docker search`.

Repositório

Um repositório é um conjunto de imagens. Um repositório pode ser compartilhado através de um servidor de registro.

As imagens no repositório podem ser nomeadas através de tags.

Tag

É um rótulo aplicado a um imagem em um repositório. É através das tags que várias imagens são diferenciadas umas das outras em um repositório.

Tag

Vamos executar um container do `nginx`:

```
docker run nginx:stable
```

Estamos indicando explicitamente a tag `stable`

Quando a tag é omitida (como no nosso caso do `hello-world`), a tag `latest` é utilizada

Registro, repositório e tag

Vamos visitar o hub.docker.com e ver estes termos na prática.

Listando containers

Em um novo terminal, executar o comando:

```
docker ps
```

Serão exibidos os containers em execução

Listando containers

Para exibir **todos** os containers:

```
docker ps -a
```

Removendo containers

Para remover os containers que criamos:

```
docker rm <NAME | ID>
```

Parâmetros do run

**--rm: remove o container automaticamente
após finalizar o container**

```
docker run --rm hello-world
```

--name: atribui um nome ao container

```
docker run --name nginx nginx:stable
```

Parâmetros do run

-d, --detach: roda o container em background

```
docker run -d nginx:stable
```

Interromper container:

```
docker stop <NAME|ID>
```

Reiniciar container

```
docker start <NAME|ID>
```

Parâmetros do run

Combinando parâmetros:

```
docker run -d --rm --name nginx nginx:stable
```

Parâmetros do run

--publish , -p: publica uma porta do container para o host

```
docker run --rm -p 9001:80 nginx:stable
```

Parâmetros do run

-e, --env: configura variáveis de ambiente:

```
docker run --rm -e MONGO_INITDB_ROOT_USERNAME=root --env MONGO_INITDB_ROOT_PASSWORD=root mongo
```

Exportando valores do ambiente local:

```
export MONGO_INITDB_ROOT_USERNAME=root  
export MONGO_INITDB_ROOT_PASSWORD=root  
docker run --rm -e MONGO_INITDB_ROOT_USERNAME --env MONGO_INITDB_ROOT_PASSWORD mongo
```

Parâmetros do run

Comando

Para determinados tipos de aplicação, faz sentido a utilização de comandos

Utilitários como bash, aws-cli, etc.

```
docker run --rm bash <comando>
```

Comando

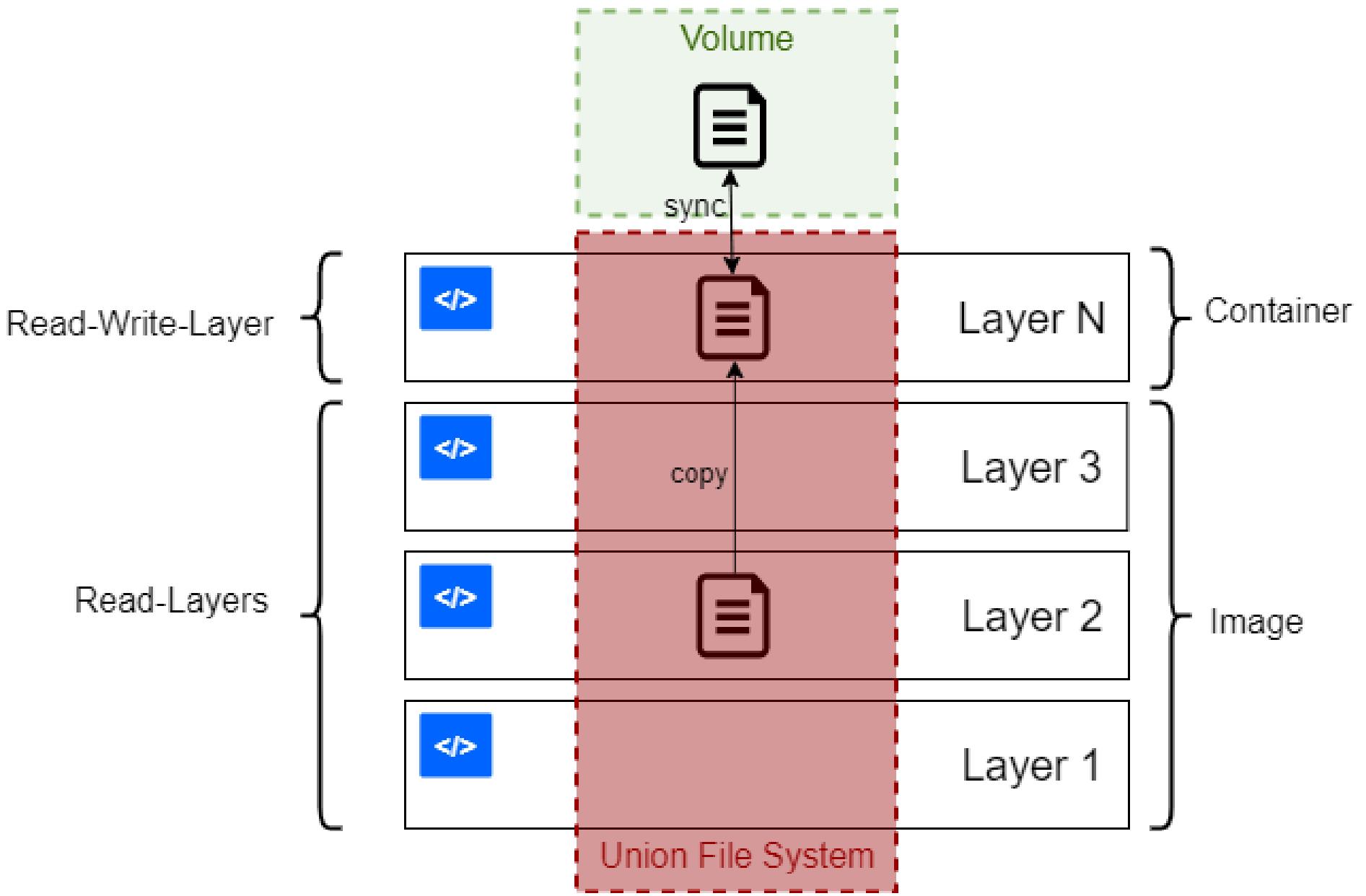
```
docker run --rm bash:latest bash -c "echo 'Hello, world!'"
```

Composição da imagem Docker

Uma imagem Docker é composta por diversas camadas.

Ao executar um container, o Docker faz o download das camadas base da imagem, que são *read-only* e adiciona uma camada a mais que é *read/write*.

```
lts: Pulling from library/sonarqube
339de151aab4: Pull complete
afbaf837951d: Pull complete
3bac5265cfae: Pull complete
c11a6d5161f0: Downloading [=====>] 120.5MB/286MB
```



Composição da imagem Docker

Quando um container é removido, a camada read/write que foi adicionada é perdida.

```
docker run bash:latest bash -c "echo 'Hello, world!' > file.txt && cat file.txt"
```

```
docker run bash:latest bash -c "cat file.txt"
```

Gerenciando dados da aplicação

Gerenciando dados da aplicação

Algumas aplicações precisam armazenar dados e que eles sejam, de fato persistentes.

Outras aplicações trabalham com entrada de arquivos.

O Docker oferece alguns mecanismos para trabalhar com armazenamento persistente, com características específicas para diferentes casos de uso.

- **Volumes**: são armazenados em uma parte do *filesystem* do *host* que é gerenciada pelo Docker (`/var/lib/docker/volumes/` no Linux). Utilizar volumes é a melhor maneira de persistir dados com o Docker. Outros processos não devem modificar os volumes.
- **Bind mounts**: pode ser armazenado em qualquer local do *filesystem* do host. Podem ser alterados por qualquer processo do host ou por containers.
- **tmpfs** são armazenados somente na memória do host, e nunca são persistidos em seu *filesystem*.

Host



bind
mount



volume



tmpfs
mount

Bind mounts

Trabalhando com bind mounts

Quando utilizamos um bind mount, um arquivo ou diretório é montado no container. Ele deve ser referenciado por seu caminho completo no host. Não precisa existir de antemão, pois será criado sob demanda caso não exista.

De acordo com a documentação oficial, são bem performáticos, mas dependem que o filesystem do host tenha disponível uma estrutura de diretório específica.

Trabalhando com bind mounts

Compartilhando um arquivo entre o host e o container

```
docker run -v $(pwd)/file.txt:/var/opt/file.txt bash:latest bash -c "cat /var/opt/file.txt"
```

Trabalhando com bind mounts

Compartilhando um diretório entre o host e o container

```
docker run -v $(pwd):/var/opt/compile -w /var/opt/compile gcc:latest gcc -o hello hello.c
```

Flag --workdir, -w: Muda de diretório dentro do container

Volumes

Trabalhando com volumes

Volumes são criados e gerenciados pelo Docker. Podem ser criados com o comando `docker volume create` ou durante a criação de um container ou serviço.

O volume é armazenado em um diretório no host. Quando você monta um volume, o diretório que é montado no container. É similar ao que acontece com bind mounts, mas, neste caso, os volumes são gerenciados pelo Docker e isolados da máquina host.

Trabalhando com volumes

Volumes podem ser montados em diferentes containers simultaneamente. Mesmo que um volume não esteja em uso por um container, ele ainda estará disponível para o Docker e **não é removido automaticamente**.

Volumes que não estão sendo usados podem ser removidos com o comando `docker volume prune`.

Trabalhando com volumes

Volumes podem ser nomeados ou anônimos. Quando um nome não é atribuído manualmente, o Docker atribui um nome aleatório único. Com exceção do nome, volumes nomeados e anônimos funcionam exatamente da mesma maneira.

Trabalhando com volumes

Criando volumes:

```
docker volume create univem # Volume nomeado  
docker volume create # Volume anônimo  
docker volume ls # Lista volumes
```

Trabalhando com volumes

Ao tentar listar os arquivos do diretório `/var/opt/univem`, nada será exibido, pois acabamos de criar o volume e ele está vazio:

```
docker run --rm -v univem:/var/opt/univem bash:latest bash -c "ls /var/opt/univem"
```

Trabalhando com volumes

Mas se utilizarmos esse mesmo volume em outro container que persiste dados no volume:

```
docker run --rm -v univem:/var/opt/univem bash:latest bash -c "echo 'Persistindo dados' > /var/opt/univem/file.txt"
```

Agora, o mesmo comando que usamos antes irá listar arquivos no volume:

```
docker run --rm -v univem:/var/opt/univem bash:latest bash -c "ls /var/opt/univem"
```

Trabalhando com volumes

Volumes com o MinIO

```
docker volume create meu_volume
docker run -p 9000:9000 -p 9001:9001 -v meu_volume:/data minio/minio server /data --console-address ":9001"
```

Credenciais: minioadmin/minioadmin

Trabalhando com volumes

Se um volume não existir, ele é criado no momento da execução do container:

```
docker run -p 9000:9000 -p 9001:9001 -v meu_volume2:/data minio/minio server /data --console-address ":9001"
```

tmpfs

Trabalhando com tmpfs

Diferente de volumes e bind mounts, o `tmpfs` é persistido somente na memória do host.

Quando um container é parado, o mount `tmpfs` é removido e nenhum dado é persistido.

Trabalhando com tmpfs

```
docker run -p 9000:9000 -p 9001:9001 --tmpfs /data minio/minio server /data --console-address ":9001"
```

Casos de uso

Casos de uso para volumes

- Compartilhar dados entre vários containers. Vários containers podem compartilhar um volume simultaneamente (read/write ou read-only). Devem ser removidos explicitamente.
- Quando não existe a garantia de que o host vai possuir uma estrutura de diretórios ou arquivos. Volumes desacoplam a configuração do host e a configuração do container.
- Quando é necessário armazenar dados em um host remoto.

Casos de uso para volumes

- Facilitam o backup e a migração entre hosts de docker. É possível interromper os containers e fazer backup do diretório dos volumes (`var/lib/docker/volumes/<volume>`).
- Quando a aplicação precisa de I/O de alta performance no Docker Desktop. Os volumes são armazenados na VM Linux, e por isso a leitura e escrita tem menor latência a alto *throughput*.

Casos de uso para volumes

- Quando a aplicação precisa do comportamento 100% nativo do filesystem no Docker Desktop. Bancos de dados, por exemplo, precisam de controle sobre o disco para garantir as transações. Como os volumes são armazenados na VM do Linux, essas garantias existem. Com bind mounts, os filesystems do macOS ou Windows se comportam ligeiramente diferente.

Casos de uso para bind mounts

- Compartilhar arquivos de configuração entre o host e os containers. Como um exemplo, o Docker usa este mecanismo para prover resolução de DNS padrão nos containers ao montar o `/etc/resolv.conf` do host em cada container.

Casos de uso para bind mounts

- Compartilhar código ou artefatos de build entre um ambiente de desenvolvimento no host Docker e um container.
- Quando o arquivo ou estrutura de diretórios do host tem a garantia de ser consistente com os requisitos do container.

Casos de uso para tmpfs

- Indicado para casos onde os dados não devem ser armazenados, seja por questão de segurança ou de performance.

Construindo imagens

Construindo imagens

O Docker pode construir imagens ao ler instruções contidas em um arquivo `Dockerfile`.

Um `Dockerfile` é um documento de texto que contém todos os comandos que um usuário executaria na linha de comando para rodar uma aplicação.

Ao utilizar arquivos `Dockerfile`, conseguimos criar uma build automatizada que executa várias instruções em sequência.

Construindo imagens

O Dockerfile pode ser composto de diversas instruções:

FROM , RUN , CMD , COPY , ADD , ENTRYPOINT . WORKDIR ...

Utilizando o comando build

Para construir imagens, utilizaremos o comando:

```
docker build <CONTEXTO>
```

É importante notar que a build é executada pelo *daemon* do Docker, e não pelo CLI.

A primeira ação da build é enviar todo o contexto, de forma recursiva, para o *daemon*. Mais adiante veremos como a utilização do `.dockerignore` otimiza o processo de build.

Utilizando o comando build

Tradicionalmente o Dockerfile se chama Dockerfile e está localizado na raiz do contexto:

```
docker build .
```

Utilizando o comando build

Nosso `Dockerfile` é composto por duas instruções:

- `FROM` : inicializa uma nova build e configura o `ubuntu:20.04` como a imagem pai.
- `RUN` : roda o comando `echo 'Hello, world!' > /var/opt/file.txt` em uma nova camada da imagem.

Utilizando o comando build

Para visualizar a imagem construída:

```
$ docker images
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	944bd7613173	10 seconds ago	72.8MB

Para executar a imagem que acabamos de construir:

```
docker run --rm <id imagem>
```

Utilizando o comando build

Ao construir imagens, é interessante a utilização de nomes e tags. Se essas imagens forem enviadas para um registro, as tags se tornam indispensáveis

Para adicionar uma tag no momento da construção das imagens, basta adicionar a flag `-t <nome:tag>`

```
docker build .
docker build -t univem .
docker build -t univem:latest .
```

Adicionando tags a uma imagem

O comando `docker tag` permite adicionar tags depois que a imagem foi construída:

```
docker tag c29284518f49 univem
```

Adicionando tags a uma imagem

Formato da tag:

```
[REGISTRO/] [REPOSITÓRIO/] imagem[:TAG]
```

```
docker tag univem univem:1.0.0
docker tag univem:1.0.0 123456789.dkr.ecr.region.amazonaws.com/univem:1.0.0
```

Quando omitido, o comando usa o registro público padrão:

```
registry-1.docker.io
```

Utilizando o comando build

Ao construir uma imagem, também é possível indicar manualmente um `Dockerfile` com a flag `-f <arquivo>`:

```
docker build -f Dockerfile2 .
```

Inspecionando a imagem criada

Vamos utilizar a ferramenta <https://github.com/wagoodman/dive> para inspecionar as imagens construídas.

Compatível com Windows, Linux e macOS.

Inspecionando a imagem criada

```
docker images  
dive <IMAGEM>
```

Inspecionando a imagem criada

Executando o dive por meio de um container:

```
docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock wagoodman/dive:latest <argumentos>
```

Inspecionando a imagem criada

Ao inspecionar a imagem construída, vemos que existem duas camadas:

- A camada do Ubuntu, gerada pela instrução `FROM ubuntu:20.04`
- A camada do comando `RUN /bin/bash -c "echo 'Hello, world!' > /var/opt/file.txt"`

Quando os comandos `RUN`, `COPY`, `ADD` são executados, novas camadas são adicionadas na imagem final.

Diferença entre instruções RUN e CMD

- RUN: Executa comandos em uma nova camada e cria uma nova imagem. Geralmente é utilizado para atualizar e instalar pacotes, baixar arquivos, entre outras tarefas.
- CMD: Configura o comando padrão e/ou parâmetros. Pode ser sobreescrito no momento de executar o container.

Instrução ADD e COPY

Os comandos `ADD` e `COPY` são bem semelhantes. Ambos tem como objetivo adicionar arquivos na imagem.

A maior diferença entre eles é que o comando `ADD` aceita URLs, além de extrair arquivos comprimidos (gzip, bzip2 e xz).

A documentação oficial recomenda utilizar `COPY` na maioria dos casos, e mudar para `ADD` quando suas features são necessárias.

Instrução ENTRYPOINT

Utilizando quando queremos uma imagem que será utilizada como "executável".

Bem semelhante ao comando `CMD`, com a diferença que este não é ignorando quando passamos parametros ao executar o container.

shell form e exec form

Os comandos `RUN` , `CMD` e `ENTRYPOINT` podem ser escritos de duas maneiras:

- `RUN <comando>` : *shell form*. o comando é executado em um shell (por padrão /bin/sh -c no Linux)
- `RUN ["executavel", "param1", "param2"]` : *exec form*

Shell form

Quando uma instrução é invocada com *shell form*, ela é executada com `/bin/sh -c <comando>` implicitamente e ocorre o processamento do shell

Alguns exemplos da *shell form*:

```
RUN apt-get install libssl-dev
```

```
CMD echo "Hello, world!"
```

```
ENTRYPOINT echo "Hello, world!"
```

Shell form

Ao construir e executar a imagem no diretório `09-docker-build-shell-form`, teremos como saída:

`Hello, Maria!`

Shell form

Também é possível indicar explicitamente qual shell utilizar:

```
ENTRYPOINT /bin/bash -c 'echo "Hello, $name!"'
```

Exec form

Quando uma instrução é invocada com *exec form*, ela é executada diretamente e não ocorre nenhum processamento do shell.

Alguns exemplos de *exec form*:

```
RUN ["apt-get", "install", "libssl-dev"]
```

```
CMD ["/bin/echo", "Hello, world!"]
```

```
ENTRYPOINT ["/bin/echo", "Hello, world!"]
```

Exec form

Geralmente ao utilizar `CMD` e `ENTRYPOINT`, se utiliza a exec `form`

Combinando ENTRYPOINT E CMD

É possível combinar o `ENTRYPOINT` com o `CMD` para combinar parâmetros de um executável. Os parâmetros do `ENTRYPOINT` sempre serão utilizados, enquanto os parâmetros do `CMD` poderão ser sobreescritos.

```
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

A saída será diferente ao executar com ou sem comando:

```
docker run <imagem>
docker run <imagem> Maria
```

Quando utilizar RUN, EXEC e ENTRYPOINT

- **RUN:** utilizado para executar instruções que vão adicionar camadas na imagem.

Quando utilizar RUN, EXEC e ENTRYPOINT

- **ENTRYPOINT**: utilizado quando você quer que um comando seja sempre executado. Pode ser combinado com o `CMD` para adicionar parâmetros extra que podem ser sobreescritos na execução do container.

Quando utilizar RUN, EXEC e ENTRYPOINT

- **CMD:** utilizado para executar um comando padrão que poderá ser sobreescrito na execução do container.

Criando uma imagem de uma aplicação

No diretório `12-ubuntu-node-image`, temos uma aplicação feita em node + express.

Ao analisar o `Dockerfile` deste diretório, podemos ter uma ideia de como várias instruções são executadas para construir a imagem de uma aplicação, partindo de um SO vazio.

Criando uma imagem de uma aplicação

A primeira instrução do nosso `Dockerfile` vai rodar uma sequência de comandos para atualizar o ubuntu, instalar alguns pacotes, baixar o node e extraí-lo e movê-lo para um diretório específico:

```
RUN \
apt-get update && \
apt-get --no-install-recommends install -y \
ca-certificates \
...
```

Instrução WORKDIR

Muda o diretório da próxima instrução. Pode utilizar um diretório absoluto ou um diretório relativo.

No caso da nossa imagem, alteramos o `WORKDIR` e em seguida utilizamos a instrução `COPY`, ou seja, o `COPY` será feito no diretório `/usr/src/app`:

```
WORKDIR /usr/src/app  
COPY package*.json ./
```

Instrução EXPOSE

Informa ao Docker que o container ouve na porta especificada. Pode ser configurada como TCP ou UDP, e usa TCP se o protocolo não for especificado.

É importante salientar que o `EXPOSE` não publica a porta. A instrução serve como uma forma de documentação entre quem construiu a imagem a quem vai rodar o container, para indicar quais portas tem a intenção de ser publicadas.

Instrução EXPOSE

Para de fato publicar e mapear as portas ao executar um container, podemos usar o parâmetro `-p porta:porta`, como já vimos.

Também é possível usar o parâmetro `-P` para publicar automaticamente todas as portas marcadas com a instrução `EXPOSE` e mapeá-las para portas maiores.

Criando uma imagem de uma aplicação

Vamos construir uma imagem da nossa aplicação e executá-la:

```
docker build -t univem:12 .
docker run --rm -p 8080:8080 univem:12
docker run --rm -P univem:12
```

Criando uma imagem de uma aplicação, tirando proveito de outras imagens

Uma grande vantagem ao construir uma imagem, é que podemos utilizar qualquer imagem mais especializada como pai da nossa imagem.

Criando uma imagem de uma aplicação, tirando proveito de outras imagens

O `Dockerfile` do diretório `13-node-image` é reduzido porque usa como base uma imagem que já inclui uma instalação do nodejs:

```
FROM node:14
```

Build multistage

Build multistage

É um mecanismo que foi introduzido para manter o tamanho das imagens pequeno, ao mesmo tempo que o `Dockerfile` se mantém legível.

Cada instrução no `Dockerfile` adiciona uma camada e é necessário lembrar de fazer a limpeza de artefatos não utilizados antes de mover pra próxima camada.

Antes da introdução do multistage, era comum a existência de diversos `Dockerfiles` para diferentes estágios de build:
`Dockerfile.build` , `Dockerfile.test` , `Dockerfile.dev` ...

Build multistage

Como o próprio nome sugere, um único `Dockerfile` pode ser dividido em vários estágios.

As imagens base de cada estágio podem ser dependentes entre si ou completamente independentes.

Também é possível compartilhar artefatos entre estágios.

Build multistage

Antes de ver uma build multistage, vamos conhecer o padrão builder, que era usado e que se tornou obsoleto com a introdução da build multistage.

Padrão builder (obsoleto)

No diretório `14-builder-pattern`, temos uma aplicação Angular que vamos construir e disponibilizar em uma imagem Docker.

Os principais artefatos do processo de build de uma aplicação Angular são o `index.html`, os arquivos javascript (`.js`), arquivos de estilo (`.css`) e outros recursos como imagens e fontes da aplicação.

Após construída, uma aplicação pode ser disponibilizada por qualquer server HTTP, como Apache ou nginx.

Padrão builder (obsoleto)

Neste diretório (14-builder-pattern) temos 3 arquivos chave:

- `Dockerfile.build` : descreve como construir a aplicação Angular, gerando os artefatos
- `Dockerfile.main` : descreve como executar a aplicação (usando o nginx) a partir de seus artefatos
- `build.sh` : script que faz a "cola" entre as etapas acima

Build multistage

No diretório `15-multistage-build`, temos o mesmo projeto Angular, mas agora somente um `Dockerfile`

Agora, conseguimos descrever todo o processo de build em um único arquivo, facilitando a manutenção do processo e o tornando mais legível

Utilizando repositórios

Utilizando repositórios

Após construída a imagem, podemos enviá-la a um servidor compartilhado, para que ela possa ser usada por outras pessoas, servidores de build, ferramentas de validação e etc.

O Docker oferece um registro open source de imagens, que pode ser hospedado por qualquer um:

```
docker run -d -p 5000:5000 --name registry registry:2
```

Utilizando repositórios

Agora podemos tagear nossas imagens para enviá-las para o registro/Formato:

```
[REGISTRO/] [REPOSITÓRIO/] imagem[:TAG]
```

```
docker tag univem:latest localhost:5000/univem:latest  
docker push localhost:5000/univem:latest  
docker pull localhost:5000/univem:latest
```

Subindo para o docker hub

Eu:

```
docker info # Mostra as configurações atuais do Docker  
docker tag univem:latest leonardortlima/univem:latest  
docker tag univem:latest leonardortlima/univem:v1  
docker tag leonardortlima/univem:latest leonardortlima/univem:v1  
docker push leonardortlima/univem:v1  
docker push leonardortlima/univem:latest
```

Vocês:

```
docker run --rm -p 3000:3000 leonardortlima/univem:latest
```

Observações importantes:

- Novas versões da imagem podem ser adicionadas por meio de novas tags
- Uma tag pode ser alterada para apontar para uma imagem mais recente

Subindo para o docker hub

Vamos editar nossa aplicação e fazer o deploy da nova versão por meio de uma tag nova:

```
docker build -t leonardortlima/univem:v2 .
docker tag leonardortlima/univem:v2 leonardortlima/univem:latest
docker push leonardortlima/univem:v2
docker push leonardortlima/univem:latest
```

Dicas gerais ao construir imagens

Crie containers efêmeros

Crie imagens que permitam que você crie, destrua e substitua containers com o menor trabalho possível.

Minimize o número de camadas

Lembre que os comandos `RUN`, `COPY` e `ADD` criam camadas adicionais.

Utilize builds multistage sempre que possível.

Entenda e tire proveito do build cache

Entender como funciona o cache acelera o desenvolvimento e evita problemas. Por padrão o cache vem habilitado, mas é possível desativá-lo durante a build com a flag `--no-cache=true`.

Também é possível limpar o todo o cache com o comando `docker builder prune`.

Entenda e tire proveito do build cache

O cache de build funciona da seguinte maneira:

- Partindo de uma imagem pai, a próxima instrução é comparada com todas as imagens filho para ver se alguma delas possui exatamente a mesma instrução. Se não possuir, o cache é invalidado.

Entenda e tire proveito do build cache

- Para as instruções `ADD` e `COPY`, o conteúdo dos arquivos na imagem são examinados e um checksum é calculado para cada arquivo. A data de alteração dos arquivos não é levada em consideração. Durante a busca no cache, o checksum é comparado com o checksum das imagens existentes. Se o conteúdo dos arquivos mudou, o cache é invalidado.

Entenda e tire proveito do build cache

- Com exceção do `ADD` e `COPY`, o mecanismo de cache não verifica os arquivos na imagem para determinar um match. Quando o comando `RUN apt-get -y update` é examinado, por exemplo, os arquivos adicionados no container não são verificados. Nesse caso, somente a string do comando é utilizada.

Crie imagens pequenas, focadas e seguindo as melhores práticas para o seu contexto

Imagens pequenas são mais rápidas pra baixar.

Quanto menos coisa existir dentro da imagem, menor será a necessidade de manutenção e a superfície para possíveis problemas.

Crie imagens pequenas, focadas e seguindo as melhores práticas para o seu contexto

Em algumas situações, tamanho é, literalmente, dinheiro.

Os principais serviços de cloud como a [AWS](#), [GCP](#) e [Azure](#) cobram por tamanho da imagem nos registros e/ou tráfego de download das imagens.

Crie imagens pequenas

Leve em consideração a imagem base que será utilizada.

A Canonical, por exemplo, vem diminuindo o tamanho da imagem base do Ubuntu constantemente. Em julho de 2015 a imagem `ubuntu:latest` tinha 188mb. Hoje, a `latest` tem 72.8mb.

Considere utilizar distros alternativas, como [Alpine Linux](#), que é uma distribuição com grande foco no tamanho, tendo apenas 5mb.

Crie imagens pequenas

Algumas imagens base, como a do [Python](#), oferecem tags alternativas:

```
docker pull python:3.8
docker pull python:3.8.3
docker pull python:3.8.3-slim
docker pull python:3.8.3-alpine
docker images
```

Dependa de tags fixas

Ao construir suas imagens, evite depender de tags como `latest` e major versions como `python:3`.

Este tipo de tag pode ser atualizado com frequência (`latest`) e/ou introduzir problemas de compatibilidade (major versions, como `python:3`).

Tags fixas, como `python:3.9.6`, ou minor versions, como `python:3.9`, apresentam menor chance de introduzir problemas de compatibilidade.

Entenda o build context

Entender e otimizar o build context faz as builds serem mais rápidas, economizando tempo de desenvolvimento.

Entenda e utilize o `.dockerignore`.

Acessando um container

Container em execução:

```
docker exec -it <ID|NOME> bash
```

Novo container:

```
docker run -it --entrypoint /bin/bash <image>
```

Limitando memória e CPU

Por padrão os containers rodam sem limites de recurso no Docker. Em alguns cenários é interessante limitar recursos dos containers.

Comando	Descrição
<code>-m or --memory=</code>	Limite máximo de memória que um container pode usar
<code>--memory-reservation</code>	Limite flexível. Não garante que o limite não será excedido.
<code>--cpus=</code>	Limite de CPUs que um container pode usar. Se o host tem duas CPUs e o container for configurado com <code>--cpus="1.5"</code> , o container vai usar no máximo uma CPU e meia

GPU

Também é possível compartilhar GPUs:

```
docker run -it --rm --gpus all ubuntu nvidia-smi
```

Docker e OCI

A Open Container Initiative (OCI) é um projeto da Linux Foundation, que tem como missão promover um conjunto de padrões e especificações sobre tecnologias de containers.

Junto com a criação do Docker, surgiu um grande interesse e uso de soluções baseadas em containers. Diversos fornecedores e provedores de cloud começaram a anunciar soluções na área.

A popularização da tecnologia foi benéfica, mas também destacou a importância da criação de certos padrões para que a portabilidade (pilar central na utilização de containers) fosse mantida.

Apesar do Docker ter sido o projeto com maior crescimento, cujo formato de imagem se tornou o padrão, surgiu o interesse de criar uma especificação única e aberta, que;

- Não depende de componentes de alto nível, como um client específico ou stack de orquestração
- Não é acoplada a um fornecedor ou projeto comercial, e que seja portável entre diferentes SOs, hardwares, arquitetura de processador, clouds, etc

Benefício para os usuários

O surgimento do OCI significa que nós, usuários, podemos adotar tecnologias de container, sem precisar se preocupar em ficar travador em um fornecedor de infra, cloud, ferramenta específica, entre outros.

Podemos focar em escolher as melhores ferramentas que melhor atende nossas aplicações.

Especificações do OCI

Os papéis específicos do OCI são:

- Criar uma especificação formal de formatos de imagens de containers e runtime, que permita que um container complacente seja portável entre os maiores SOs e plataformas sem barreiras. Também entrarão de acordo em um conjunto de ações que podem ser feitas em um container (start, exec, pause).
- Harmonizar o padrão acima com outros padrões propostos



End users create and run containers with the **docker** command.

containerd pulls images, manages networking & storage, and uses runc to run containers.

runc does the low-level 'stuff' to create and run containerised processes.

- **docker-cli**: é a ferramenta de linha de comando com a qual interagimos
- **containerd**: um processo daemon que gerencia e executa containers. Baixa e envia imagens, gerencia armazenamento e supervisiona containers em execução
- **runc**: é o runtime de container de baixo nível. É ele que de fato cria e roda os containers

Ferramentas relacionadas

- [Podman](#): pode ser utilizado como um substituto do Docker.
- [Buildah](#): ferramenta para construção de imagens OCI

Docker compose

Docker compose

Muitas vezes é necessário executar outros serviços além da aplicação.

É comum que aplicações tenham que acessar um banco de dados, armazenar arquivos em um servidor externo, utilizar mecanismo de fila, etc.

Docker compose

O Docker Compose é uma ferramenta para definir e executar aplicações que dependem de vários containers.

Com ele é possível definir todos os serviços de uma aplicação em um arquivo YAML, e com um único comando criar e executar esses serviços.

O arquivo docker-compose.yml

Primeiro, vamos construir a imagem do diretório `16-compose-init` com a tag `univem:latest`:

```
docker build -t univem:latest .
```

Por se tratar de uma aplicação web, para executar e testar o container, seria ideal expor uma porta para a API:

```
docker run -p 8080:8080 univem
```

O arquivo docker-compose.yml

Agora, vamos declarar o mesmo comportamento com o `docker-compose.yml`:

```
version: "3.9"
services:
  web:
    image: univem:latest
    ports: "8080:8080"
```

E executar o docker compose:

```
docker-compose up
```

Comandos importantes do Docker Compose

up: constrói, (re)cria, inicializa e anexa os containers de um serviço

```
docker-compose up
```

down: pára e remove containers, redes, volumes e imagens criadas pelo `up`

```
docker-compose down
```

Comandos importantes do Docker Compose

start: inicializa os containers que já existem de um serviço

```
docker-compose start
```

stop: pára os containers sem removê-los

```
docker-compose stop
```

Flags importantes do Docker Compose

-f, --file: especifica um arquivo alternativo (padrão: docker-compose.yml)

```
docker-compose -f /path/to/some/docker-compose.yml up
```

-d, --detach: modo detached

```
docker-compose up -d
```

Compose com serviços

Com o docker compose é possível declarar vários serviços que vão trabalhar em conjunto:

```
version: "3.9"
services:
  mongo:
    ...
  mongo-express:
    ...
  web:
    ...
```

Compose com serviços

Podemos observar que ao executar o comando `up`, o Docker interpreta as dependências entre serviços por meio da instrução `depends_on`, e só inicializa um container quando sua dependência está disponível.

Outro detalhe é o endereço do banco utilizado no serviço `web` e no `mongo-express`:

```
DB_URI: mongodb://root2:root2@mongo:27017  
ME_CONFIG_MONGODB_SERVER: mongo
```

Compose com build

Outra funcionalidade útil do docker-compose é indicar que um dos serviços precisa ser construído dinamicamente:

```
web:  
  build:  
    context: .
```

E na hora de executar o docker compose:

```
docker-compose up --build
```

Compose... composto

Também é possível extender serviços declarados em outros arquivos compose:

```
services:  
  mongo:  
    extends:  
      file: docker/mongo.yml  
      service: mongo  
  mongo-express:  
    extends:  
      file: docker/mongo-express.yml  
      service: mongo-express
```

Kubernetes

Kubernetes - K8s

Ferramenta open source para orquestração de containers.

Desenvolvida originalmente pelo Google.

Ajuda a gerenciar aplicações "containerizadas" em diferentes ambientes.

Qual problema o Kubernetes resolve

Gerenciar um container é fácil

O que fazer quando temos dezenas, centenas ou milhares de containers?

Qual problema o Kubernetes resolve

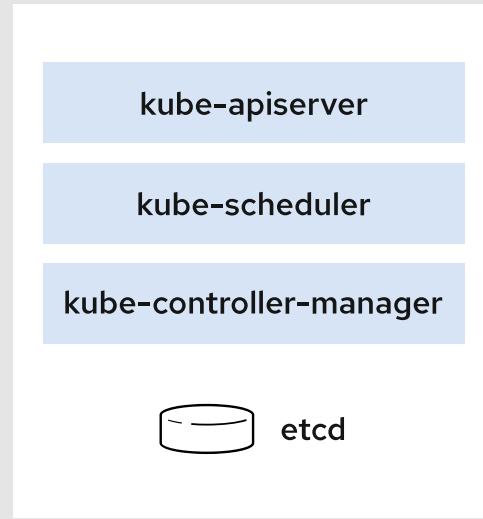
As principais vantagens ao utilizar Kubernetes são:

- Alta disponibilidade
- Escalabilidade
- Disaster recovery

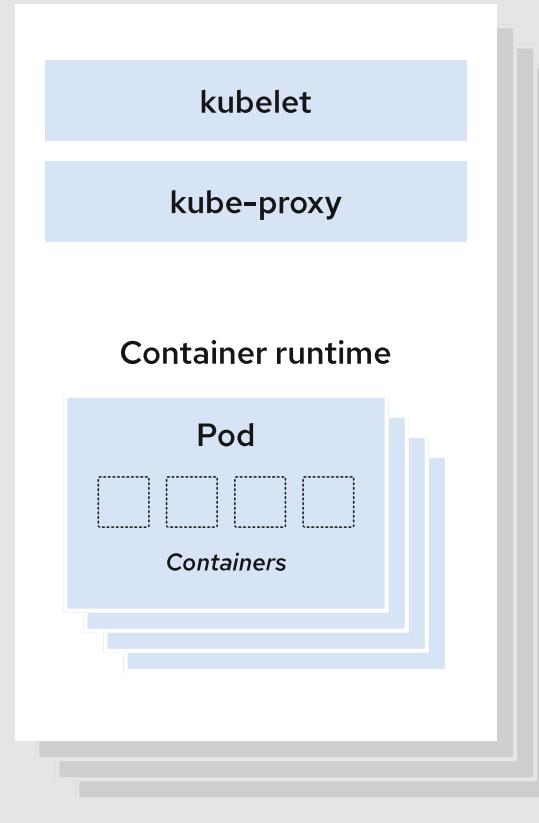
Arquitetura do Kubernetes

Kubernetes cluster

Control plane



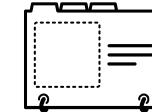
Compute machines



Persistent storage



Container registry



Underlying infrastructure



Physical



Virtual



Private



Public



Hybrid

Control plane

Control plane

É nele que estão os componentes do Kubernetes que controlam o cluster e os dados sobre o estado e a configuração do cluster.

Desempenham um papel importante: eles asseguram que os containers sejam executados na quantidade suficiente e com os recursos necessários.

Control plane

Está sempre se comunicando com as *compute machines*. Quando você configura a execução do cluster, o plano de controle faz com que isso aconteça de acordo com o planejado.

kube-apiserver

A API do Kubernetes é o front-end do plano de controle. Ela processa solicitações internas e externas.

É possível acessar a API por meio de chamadas REST, da interface de linha de comando kubectl ou de outras ferramentas relacionadas, como a kubeadm.

kube-scheduler

Monitora a saúde do cluster.

Determina onde novos containers serão executados:

Ele analisa quais os recursos de que um pod necessita, como CPU e memória. E também avalia integridade do cluster. Depois, ele programa o pod no nó de computação apropriado.

kube-controller-manager

São os controllers que de se encarregam de executar o cluster, e o kube-controller-manager tem várias funções de controller em um só.

Há um controlador que consulta o programador para se certificar de que o número correto de pods esteja em execução.

Se algum pod ficar inativo, outro controlador perceberá e corrigirá esse problema. Há um controlador conecta os serviços aos pods para que as solicitações cheguem aos endpoints corretos. E outros que criam tokens de acesso a APIs e contas.

etcd

O etcd é um banco de dados de armazenamento de chave/valor que contém as informações e dados de configuração sobre o estado do cluster. Ele é distribuído, tolerante a falhas e foi projetado para ser a principal fonte de informações sobre o cluster.

Nodes

Nodes

O cluster Kubernetes precisa de pelo menos um nó de computação, mas ele geralmente tem vários.

Os pods são programados e orquestrados para serem executados nos nós. Precisa aumentar a capacidade do cluster?
Basta adicionar mais nós.

Pods

O pod é a menor e mais simples unidade do modelo de objetos do Kubernetes. Ele representa uma instância de uma aplicação. Cada pod é formado por um container ou por vários deles com forte acoplamento, além de opções que determinam como esses containers são executados.

É possível conectar o armazenamento persistente aos pods para executar aplicações stateful.

Container runtime engine

Para executar os containers, cada nó de computação conta com um mecanismo de ambiente de execução.

Um exemplo é o Docker, mas o Kubernetes é compatível com outras que estejam em conformidade com a Open Container Initiative, como o rkt e CRI-O.

kubelet

Todos os nós de computação incluem um kubelet, uma aplicação muito pequena que faz a comunicação com o plano de controle.

O kubelet assegura que os containers estejam em execução em um pod.

Quando o plano de controle precisa que algo aconteça em um nó, o kubelet realiza a ação.

kube-proxy

O kube-proxy também está incluído em todos os nós de computação. Ele é um proxy que facilita os serviços de rede do Kubernetes.

O kube-proxy gerencia as comunicações de rede dentro e fora do cluster, recorrendo à camada de filtragem de pacotes do sistema operacional ou encaminhando o tráfego por conta própria.

Componentes extras

Armazenamento persistente

Além de gerenciar os containers que executam a aplicação, o Kubernetes também controlam os dados da aplicação anexados ao cluster.

Com o Kubernetes, os usuários podem solicitar recursos de armazenamento sem precisar conhecer os detalhes da infraestrutura subjacente. Os volumes persistentes podem ter um ciclo de vida maior do que os pods, já que são específicos para os clusters.

Registro de containers

As imagens de container usadas pelo Kubernetes ficam armazenadas em um registro, que pode ser configurado por você ou fornecido por terceiros.

Infraestrutura subjacente

Você escolhe o ambiente de execução do Kubernetes.

Ele pode ser servidores bare-metal, máquinas virtuais, nuvens privadas, públicas e híbridas. Uma das principais vantagens do Kubernetes é que ele funciona em vários tipos de infraestrutura.

Hands on no Linode:

<https://www.youtube.com/watch?v=7bA0gTroJjw>

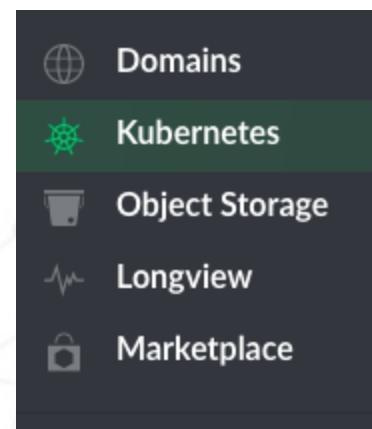
Crédito de 100 dólares válidos por 60 dias.

Precisa colocar cartão de crédito.

https://bit.ly/nc_linode

Disclaimer: não estou ganhando nada com isso!

Criar cluster



Criar cluster



Kubernetes

Need help getting started?

[Learn more about getting started with LKE.](#)

[Create Cluster](#)

Criar cluster

Kubernetes / Create Cluster

Cluster Label

univem

Region

You can use [our speedtest page](#) to find the best region for your current location.

 Dallas, TX**Kubernetes Version**

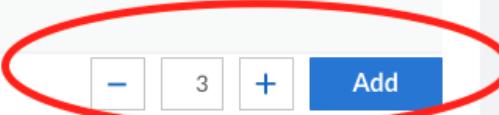
1.21

Add Node Pools

Add groups of Linodes to your cluster with a chosen size.

[Shared CPU](#)[Dedicated CPU](#)[High Memory](#)

Shared CPU instances are good for medium-duty workloads and are a good mix of performance, resources, and price.

Plan	Monthly	Hourly	RAM	CPUs	Storage	
Linode 2 GB	\$10	\$0.015	2 GB	1	50 GB	

Master node

A Linode "não cobra" pelo master node!

Ao criarmos o cluster, ele já vem incluso e configurado.

Comunicação com o master

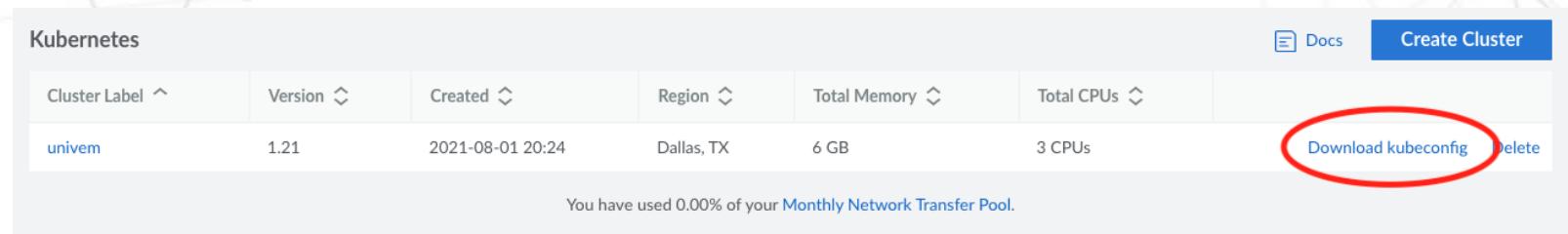
utilizaremos o `kubectl` para enviar comandos ao master do nosso cluster.

Instruções de instalação para diversas plataformas:

<https://kubernetes.io/docs/tasks/tools/>

Apontando o kubectl para o cluster:

Acessar o dashboard do cluster e baixar o kubeconfig:



The screenshot shows the Kubernetes dashboard interface. At the top, there's a navigation bar with "Docs" and a "Create Cluster" button. Below it is a table with the following data:

Cluster Label ^	Version ^	Created ^	Region ^	Total Memory ^	Total CPUs ^	
univem	1.21	2021-08-01 20:24	Dallas, TX	6 GB	3 CPUs	Download kubeconfig Delete

Below the table, a message states: "You have used 0.00% of your [Monthly Network Transfer Pool](#)."

A red oval highlights the "Download kubeconfig" link next to the cluster entry.

Apontando o kubectl para o cluster:

Criar variável de ambiente `KUBECONFIG`, apontando para o arquivo:

```
export KUBECONFIG=/Users/leonardolima/Downloads/univem-kubeconfig.yaml
```

Listando os nodes do cluster:

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
lke33119-50759-610702eff499	Ready	<none>	3m22s	v1.21.1
lke33119-50759-610702effc38	Ready	<none>	3m53s	v1.21.1
lke33119-50759-610702f0042f	Ready	<none>	3m41s	v1.21.1

Listando informações do cluster:

```
kubectl cluster-info
```

```
Kubernetes control plane is running at https://xyz.cpc2-us-central.linodelke.net:443
```

```
KubeDNS is running at https://xyz.cpc2-us-central.linodelke.net:443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

```
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Fazendo o deploy de um pod

Um pod é composto por um ou mais containers.

```
kubectl run univem-pos --image=leonardortlima/univem:latest --port=80
```

Verificando o status dos pods

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
univem-pos	1/1	Running	0	59s

Verificando detalhes dos pods

```
kubectl describe pods
```

Removendo pods

```
kubectl delete pods univem-pos
```

```
pod "univem-pos" deleted
```

Fazendo um Deployment

Agora vamos criar um `Deployment`, que executará vários pods simultaneamente.

O `Deployment` que vamos executar está descrito no arquivo
`20-kubernetes-linode/univem_deployment.yaml`

Criando um Deployment

Vamos fazer Deployment descrito no arquivo:

```
kubectl apply -f univem_deployment.yaml  
kubectl get pods
```

O master interpretará o arquivo e executará as ações necessárias

Editando um Deployment

É possível editar um `Deployment` após ele ter sido criado.

Vamos subir o número de replicas de 3 para 10:

```
kubectl edit deployment univem-pos-deployment
```

```
deployment.apps/univem-pos-deployment edited
```

Editando um Deployment

Se consultarmos os pods ativos novamente, veremos que o master está fazendo o deploy de mais 7:

```
kubectl get pods
```

Distribuição de carga

Quando criamos `pods` e `deployments`, o `master` automaticamente distribui a carga entre os nodes do clusters.

Ele também garante que se um node falhar, a carga será redistribuída entre os outros nodes.

```
kubectl get pods -o wide
```

Distribuição de carga

Também é possível configurar métricas de autoscaling.

O master se encarregará de monitorar a utilização dos recursos de um deployment e fazer o scale up ou scale down conforme necessário.

Expondo pods através de um service

Para expor nossos pods, vamos criar um service que fará o LoadBalacing da nossa aplicação.

No arquivo `20-kubernetes-linode/univem_service.yaml`, temos a configuração do nosso service.

Expondo pods através de um service

Vamos criar o `Service` descrito no arquivo:

```
kubectl apply -f univem_service.yaml
```

```
service/univem-service created
```

Descrevendo os services

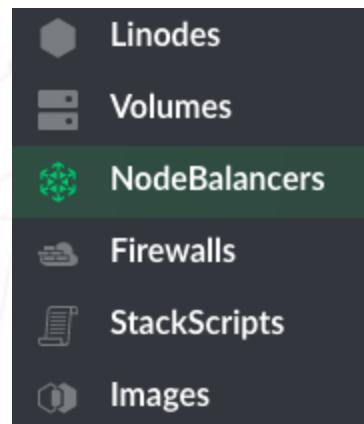
Para ver os services em execução:

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.128.0.1	<none>	443/TCP	63m
univem-service	LoadBalancer	10.128.18.93	104.200.23.49	80:31567/TCP	46s

Integração entre o cloud provider e o kubernetes

Ao executar esse comando, automaticamente foi criado o chamado `NodeBalancer` no linode:



Integração entre o cloud provider e o kubernetes

NodeBalancers							Docs	Create NodeBalancer	
Name ^	Backend Status	Transferred	Ports	IP Address	Region		Configurations	Settings	Delete
ccm-4bee1f18c65d	3 up - 0 down	9.11 KB	80	104.200.23.49	Dallas, TX		Configurations	Settings	Delete
You have used 0.00% of your Monthly Network Transfer Pool .									

Atualizando nosso site

Vamos editar nosso deployment:

```
kubectl edit deployment univem-pos-deployment
```

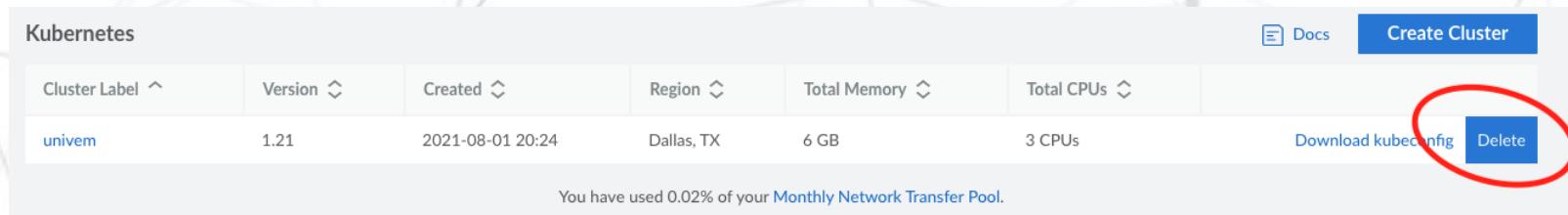
E alterar a imagem utilizada:

```
image: leonardortlima/univem:v2
```

Atualizando nosso site

O master se encarregará de destruir os containers antigos e criar novos, utilizando a nova imagem indicada

Ao terminar, não se esqueçam de remover os nodes...



Kubernetes

Cluster Label ^	Version ^	Created ^	Region ^	Total Memory ^	Total CPUs ^	
univem	1.21	2021-08-01 20:24	Dallas, TX	6 GB	3 CPUs	Download kubeconfig Delete

You have used 0.02% of your [Monthly Network Transfer Pool](#).

Delete Cluster univem

This cluster contains **1 node pool** with a total of **3 Linodes** that will be deleted along with the cluster. Deleting a cluster is permanent and can't be undone.

To confirm deletion, type the name of the cluster (**univem**) in the field below:

Cluster Name

univem

[Cancel](#)

[Delete Cluster](#)

... e também os node balancers

Name ^	Backend Status	Transferred ▾	Ports ▾	IP Address	Region ▾	Actions
ccm-4bee1f18c65d	0 up - 0 down	573.30 KB	80	104.200.23.49	Dallas, TX	Configurations Settings Delete

You have used 7% of your [Monthly Network Transfer Pool](#).

Delete NodeBalancer ccm-4bee1f18c65d?

Are you sure you want to delete this NodeBalancer?

[Cancel](#)

[Delete NodeBalancer](#)

Observação sobre deploy de containers

Observação sobre deploy de containers

Quando se trata de fazer o deploy de containers, temos algumas alternativas.

Container bare metal:

Fazer deploy e gerenciar uma imagem diretamente em uma máquina física, máquina virtual, EC2, etc.

Container as a Service:

Serviços que "escondem" a complexidade da utilização de orquestradores de containers.

Temos como exemplo o AWS Fargate, Google Cloud Run e Azure Container Instances. Eles permitem gerenciar containers em escala, sem se preocupar com a infra por baixo.

Kubernetes bare metal:

Fazer todo o gerenciamento de clusters Kubernetes, on premises, em um data center ou na cloud.

Requer conhecimento avançado Kubernetes e esforço de manutenção.

Managed Kubernetes Services

Serviços como o **Google Kubernetes Engine**, **Amazon Elastic Kubernetes Service**, **Azure Kubernetes Service** e **Linode Kubernetes Engine** (que acabamos de utilizar) removem grande parte da complexidade de fazer deploy, upgrade e manutenção geral do Kubernetes, mas ainda assim requer expertise no assunto.

Orquestradores leves

Alternativas como o **Docker Swarm** e o **Nomad** também oferecem orquestração, e são menos complexos de utilizar e manter em comparação ao Kubernetes

Qual utilizar?

Depende!

Não faz sentido usar um Kubernetes pra fazer o deploy de uma aplicação pequena, com pouca necessidade de escalonamento. Um Container as a Service pode atender.

Da mesma forma que às vezes é necessário ter controle total sobre toda a orquestração e infra. Neste cenário, um kubernetes bare metal vai permitir personalização total.

Tarefa

Tarefa

Criar um **Dockerfile** para "containerizar" uma aplicação sua.

A sua aplicação e o Dockerfile deverão ser disponibilizados para que eu também consiga reproduzir a build e executar a imagem.

[Opcional] Criar uma conta e um repositório público no Docker Hub, e fazer o push da imagem.

Prazo de entrega: 15 dias