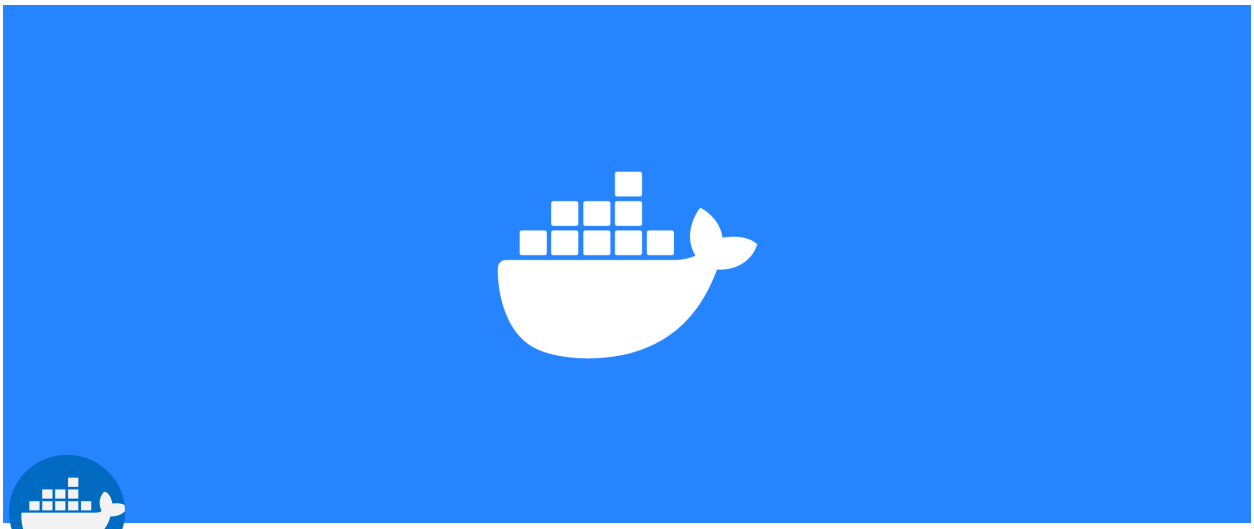


Docker

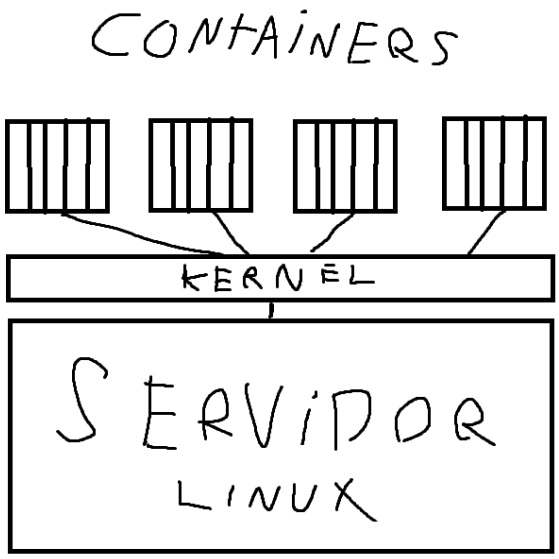
Containers

Dado um computador/infraestrutura com um sistema operacional X, **containers** são **ambientes totalmente isolados** os quais **rodam aplicações de servidor** em um mini sistema muito mais leve que VMs.

Os containers conseguem se aproveitar do kernel do sistema operacional X e utiliza-lo em cada container de forma a imitar um novo computador para rodar suas aplicações independentemente.

Os containers podem possuir sua propria distribuição do sistema, mas ele precisa compartilhar o mesmo kernel do sistema operacional do computador maior, nossa infraestrutura.

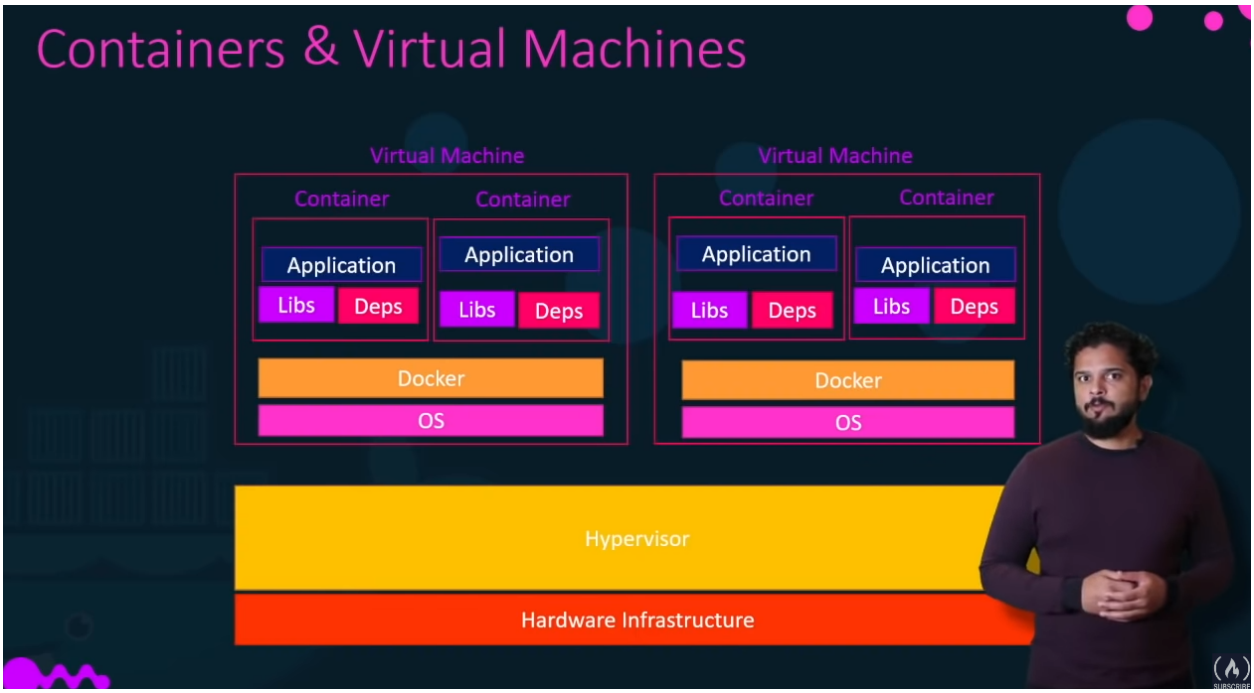
Cada container possui suas próprias dependencias e instalações de sistema, deixando-o limpo, somente com o necessário.



Containers vs Virtual Machines

VMs são totalmente isoladas, enquanto containers ainda compartilham o kernel do sistema operacional da infraestrutura.

É fato que cada um tem suas vantagens, então por que não utilizar ambos?



Uma infraestrutura pode ter VMs e estas, por sua vez, containeres.

Imagens

Um container precisa ter uma imagem especificando qual a distribuição e que o mini-sistema operacional dele deve ter para rodar a aplicação.

Isso inclui, compiladores das linguagens de programação a serem utilizadas, bibliotecas, comandos de shell especificos, etc.

Algumas imagens já estão disponíveis no **Docker Hub,** mas caso você queira criar a sua, também é possível.

Um container só se mantém ativo enquanto o processo dentro dele está ativo, se não houver aplicações rodando, não há por que da sua execução. Dessa forma, ao rodar um sistema operacional puro com o comando `docker run ubuntu`, ele roda e imediatamente para, pois nenhum processo/serviço está rodando dentro desse sistema operacional.

Comandos

docker run {imagem}

docker run | Docker Documentation

Esse comando roda uma imagem que já existe no sistema, se não existir, ela vai tentar procurar no Docker Hub.

Attached and detached mode

Ao executar um container, existem duas formas nas quais ele pode rodar.

Attached → Ao dar run em um container, dois terminais serão criados, o terminal da infraestrutura e o terminal do container.

> Em um primeiro momento, o seu terminal não está ligado ao terminal do container, mas por padrão, `docker run {imagem}` faz com que o seu terminal “se linke”a o stdout padrão do terminal do container e faz com que compartilhem dados de saída.
>
> Entretanto, você não tem os inputs nem o terminal em si ligado. Para linkar o seu terminal ao terminal do container, é necessário usa parâmetros no `docker run`

Interactive: `docker run -i {imagem}` → Agora o container lerá inputs do seu terminal.

pseudo-TTY + Interactive: `docker run -it {imagem}` → Por algum motivo, isso faz um link quase completo do seu terminal ao terminal do container.

Detached `docker run -d {imagem}` → Você não ficará preso dentro do terminal do container, nesse caso, voltará para o terminal da máquina que está rodando o docker, a infraestrutura.

Port Mapping / Port Forwarding

Antes de entender o que é port mapping, é necessário entender como os dispositivos conseguem se comunicar por meio dos seus ips, públicos e privados.

▼ Entendendo IPs

Cada servidor no mundo possui um ip, esse ip é o código que localiza o servidor na internet. Dentro desse servidor, vários serviços podem ser rodados, como um express do nodejs, mas para acessar esses diferentes serviços de um mesmo servidor pelo navegador (ou outro veículo), é necessário diferenciar eles por portas.

Portas são diferentes entradas do servidor que hospedam um serviço cada uma. Um servidor pode hospedar vários serviços como um banco de dados, um site, um gerenciador de arquivo, tudo sendo diferenciado pela porta, mas acessada pelo mesmo ip.

A conexão que vemos em muitos sites de internet é a junção de um ip com uma porta, isso é possível por conta do protocolo TCP/IP. Essa porta geralmente é a 80 ou 403.

Nesse sentido, nós só podemos acessar o servidor por meio do nosso computador pois ele possui um **ip público,** disponível para todo o mundo, que ao colocá-lo no navegador + uma porta, recebemos um site ou serviço desse servidor em questão.

IPs públicos e IPs privados.

Cada servidor possui um ip público, que permitem que nós nos conectemos a ele.

Além dos servidores, os roteadores de internet também possuem um ip público. Esse IP público é pago, e só existe um no mundo para cada servidor ou roteador.

Se o nosso roteador tiver um serviço rodando na porta 80, qualquer pessoa com o ip público do nosso roteador (meuip.com) pode entrar e acessar esse serviço.

Diferente dos roteadores e servidores, nossos dispositivos como PC, celular, notebook, não possuem um IP público, eles possuem IPs privados.

Esses IPs privados são ips da rede local, o roteador distribui um IP privado para cada dispositivo conectado na nossa wifi (ou rede) pelo protocolo DHCP.

Esses IPs não podem ser consultados por pessoas fora da rede local, se eu quiser acessar um serviço que um pc da minha casa está hospedando, eu tenho que usar o `ip privado do meu pc:porta do serviço` somente de outro dispositivo dessa mesma rede local.

localhost é o mesmo que o ip privado da própria máquina que estou usando, por isso que a maior parte das aplicações executadas nos nossos pcs são acessadas em uma porta do localhost.

Se eu tentar acessar um serviço hospedado no ip privado de algum amigo conectado a outra rede de internet, eu não vou conseguir.

Finalmente, Port Mapping

Para eu conseguir acessar o serviço que meu amigo está hosteando no ip privado dele, é necessário que ele faça um mapeamento de portas, isto é, **mapear a porta do ip privado dele a uma porta do seu ip público**.

Com isso, **o serviço também estará rodando em uma porta do ip público**, isso faz com que eu tenha acesso também.

O que isso tem a ver com o Docker?

Quando eu rodo um processo em meu container no docker, ele roda esse serviço em uma porta do seu **ip privado,** sim, o container tem seu próprio ip privado, então não podemos acessar ele pelo nosso computador. É preciso que a porta do container do docker seja mapeada para uma porta do ip privado do

nosso computador, assim, conseguimos acessar o serviço do container pelo nosso navegador.

Como mapear uma porta do container a uma porta do docker host/toggl

Usando o parâmetro -p do docker run

```
docker run -p {porta do docker host}:{porta do container} {imagem}
```

Exemplo:

```
docker run -p 5000:80 docker/getting-started
```

O port mapping é muito útil quando queremos rodar a mesmo processo em portas diferentes.

Volume Mapping

Arquivos salvos dentro do container são encontrados somente dentro dele, e ao parar um container, todos os arquivos são excluídos.

Para salvar o arquivo docker na nossa estrutura, é preciso fazer um mapping de diretorios, fazendo com que, um diretorio do container se linke a um diretório do pc.

```
docker run -v {diretorio do pc}:{diretorio do container} {imagem}
```

Assim, os arquivos também ficarão salvos no diretório do pc.

Exemplo:

```
docker run -v /desktop/mysql:/var/lib/mysql {imagem}
```

docker ps

Mostra todos os containers em execução.

`docker ps -a` → Mostra todos os containeres, executando ou não.

docker stop {nome do container/id do container}

Interrompe a execução de um container.

docker images

Lista das imagens disponíveis no sistema

docker rmi {nome da imagem}

Remove a imagem do sistema

OBS: Tenha certeza que nenhum container rodando está executando essa imagem.

docker pull {nome da imagem}

Faz o download da imagem do Docker Hub, mas não a executa.

docker push {nome da imagem}

Faz o upload da imagem no Docker Hub.

docker exec {nome do container/id do container} {comando}

Esse comando faz com que comandos shellscript sejam executados dentro do container.

docker inspect {nome do container/id do container}

Retorna detalhes minuciosos do container em um JSON

docker logs {nome do container/id do container}

Retorna um log do stdout do container.

Variáveis de ambiente

Variáveis de ambiente são variáveis que estão definidas no sistema operacional, podendo ser consultadas pelo código da aplicação usando métodos específicos.

Nesse sentido, em vez de um código definir variáveis imutáveis, podemos fazer aquela linha de código consultar uma variável do sistema (varíavel de ambiente).

Assim, um container pode ter uma série de variáveis diferentes de outro container, criando variações de execuções.

Setando uma variável de ambiente no container.

`docker run -e {VARIAVEL=VALOR} {imagem}`

Vários containers podem ser rodados com diferentes variáveis de ambiente.

Verificando variáveis de ambiente de um container.

Usamos `docker logs`

`docker logs {nome / id do container}`

Criando a própria imagem

Imagens próprias são necessárias quando você quer rodar a sua própria aplicação.

Para criar uma imagem, precisamos estabelecer o que deve conter nela, o que deve ter instalada e quais shell scripts rodar antes de rodar a aplicação.

Para isso cria-se um arquivo de instruções do Docker chamado **Dockerfile**

O Dockerfile, ao ser **lido pelo comando** `docker build`, cria a imagem que queremos usar em um container.

Dockerfile

Dentro do Dockerfile, são especificado comandos.

Toda imagem é baseada em uma imagem de SO ou uma imagem da comunidade.

Deve-se começar com a importação da imagem

`FROM Ubuntu` → Especifica que a imagem base do nosso container é o Ubuntu

A partir daí, instalamos bibliotecas, adicionamos arquivos e iniciamos o servidor.

- Cheat Sheet: Dockerfile Cheat Sheet - Kapeli
- Best Practices: Best practices for writing Dockerfiles | Docker Documentation

Exemplo de Dockerfile

```
FROM Ubuntu

RUN apt-get update && apt-get install python
# Costuma-se usar apt-get e não só apt em Dockerfiles

COPY . /opt
# Copia todos os arquivos da pasta do dockerfile para a pasta opt do container
```

```
CMD "node ./opt/src/server.js"
# Executa o comando node
```

CMD vs ENTRYPOINT

CMD e ENTRYPOINT são dois comandos Dockerfile os quais especificam comandos a serem rodados quando o container é iniciado.

CMD

`CMD “comando completo”`

Ou

`CMD [”comando”, “parametro”]` → PREFERIVEL

O CMD é um comando hard coded, não se pode mudar seu parâmetro ou o comando em si sem antes rodar o Dockerfile.

Para casos que seja necessário um dinamismo, o **ENTRYPOINT** é o ideal.

ENTRYPOINT

`ENTRYPOINT “comando”`

ou

`ENTRYPOINT [“comando1", "parametro1"]` → PREFERIVEL

O Comando especificado no entrypoint será executado como CMD, mas a diferença é que nós podemos passar parâmetros na hora do run.

Exemplo:

`docker run {imagem} “hello world”`

com `ENTRYPOINT ["echo"]`

O que acontece é que, o comando do entrypoint (echo) será jogado pra antes do seu parâmetro (hello world)

Assim, o console mostrará no stdout a mensagem que escrevemos como parâmetro.

Pode se mudar o comando do entrypoint na hora do run, usando `—entrypoint Novo comando`

E se eu quiser um valor padrão para o ENTRYPOINT?

Se quisermos um valor padrão para o parâmetro do entrypoint, colocamos um CMD com o valor do parâmetro desejado.

Exemplo

```
FROM ubuntu

ENTRYPOINT ["echo"]

CMD ["Hello World"]
```

Por padrão, o valor que aparecerá na tela é o Hello World, mas se nós o especificarmos, ele virará o especificado.

Tags

A escrever

Expose

Se sua aplicação roda em uma porta específica, essa porta precisa ser exposta no docker file

`EXPOSE 8080`

Agora, ela pode ser acessível / mapeada

Networking

Por padrão, os containeres do docker se comunicam por uma rede chamada **bridge,** é como se o docker criasse um roteador exclusivo para os containers, e desse um ip interno pra cada um.

Dessa forma, só é possível acessá-los do nosso pc por meio de um port mapping, mas os containers conseguem acessar uns os outros pelos seus ips internos.

Containers de mesma rede não podem ter as mesmas portas em uso.

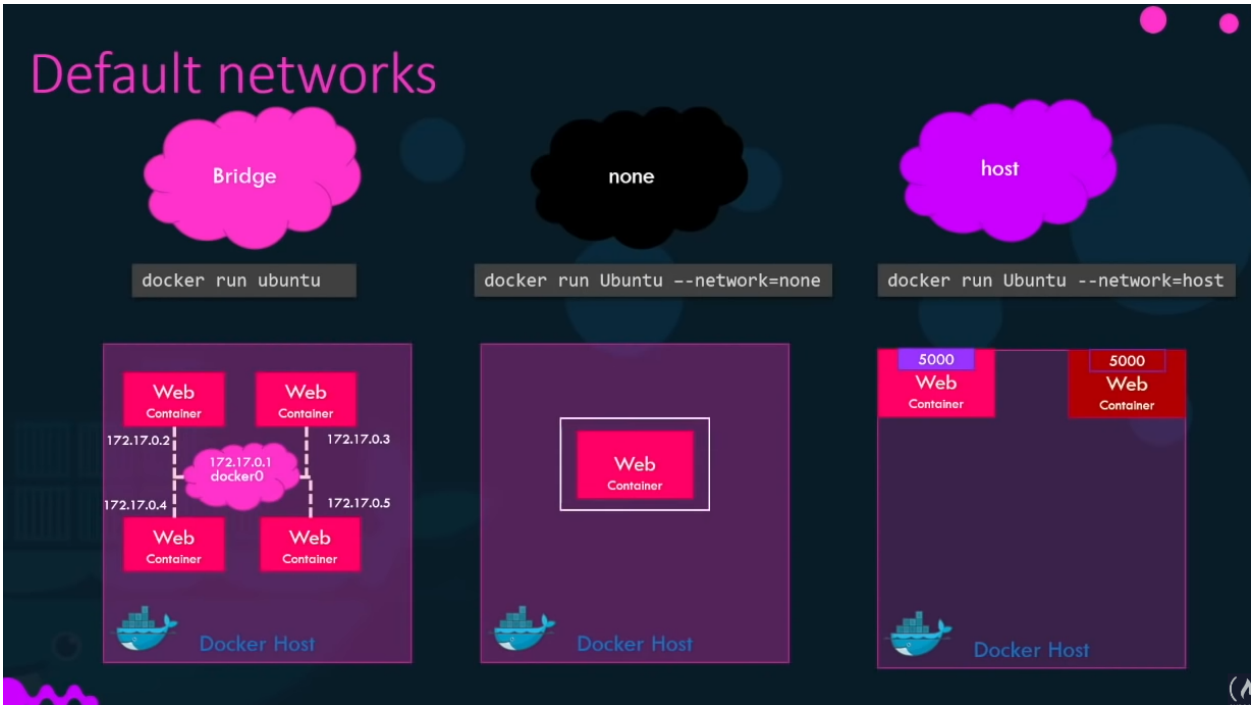
É possível ver o ipv4 (ip interno) do container com `docker inspect`

Embbeded DNS

Em vez de acessar um container ou outro dentro da rede local com o ip, os containers podem usar o nome dos outros, servindo como se fosse um domínio.

Default Networks

O Docker define 3 redes padrões.



bridge → Rede a qual os containers se conectam por padrão em

```
docker run {imagem}
```

Cria uma rede local para os containers.

none → Rede a qual os containers não têm acesso externo

```
docker run --network=none {imagem}
```

host → Rede local do nosso computador, os containers agora recebem ip interno dos nossos próprios roteadores de casa

```
docker run --network=host {imagem}
```

Comando padrão pra conectar a uma rede específica:

```
docker run --network={rede} {imagem}
```

Custom Networks

É possível criar redes isoladas para conectar apenas os containers que desejar

Comando para acessar a camada de redes:

`docker network`

Criando rede custom

`docker network create {nome-da-rede}`

Listar networks

`docker network ls`

Armazenamento / Volumes

Ao instalar o Docker em um sistema (Linux por exemplo), ele guarda suas informações no diretório `/var/lib/docker`

Dentro dessa pasta, estão armazenadas as seguintes coisas:

Quando um container é iniciado, todos os arquivos produzidos dentro do container são temporários, ou seja, um arquivo criado no container será excluído assim que ele parar.

Isso é um problema se o nosso container rodar uma aplicação de banco de dados, por exemplo, que precisa do armazenamento de arquivo.

Por isso que o Docker tem um método chamado **Mounting**

OBS: O container cria uma cópia dos arquivos das imagens, então ainda é possível alterá-los dentro do container, mas quando ele parar, são apagados, e a imagem não salva as mudanças feitas nesses aqruivos.

Mounting

Mounting é o ato de linkar o diretório de um container a um diretório do nosso computador, de modo a perpetuar a existencia dos arquivos gerados no container.

Esse diretório pode ser da pasta do docker (volume) ou de qualquer parte do computador (bind)

Nesse sentido, existem duas formas de linkar as pastas do container ao host.

Volume

Volume é uma pasta criada na própria pasta do Docker em `/var/lib/docker/volumes` o qual armazena os arquivos do container.

Criando volume

```
docker volume {create nome_do_volume}
```

Assim, uma pasta para guardar arquivos foi criada.

Rodando um container com o link de pasta / Fazendo mounting

```
docker run {imagem} -- mount type= volume ,source= nome_do_volume ,target= {pasta_do_container}
```

- Definir type=volume
- Source=Nome do Volume
- Vírgulas juntas
- Target=Pasta do container alvo

Binding

Binding é quando queremos linkar uma pasta do container a qualquer pasta do nosso computador.

Pra isso, é preciso especificar o diretório exato da pasta.

```
docker run {imagem} -- mount type= bind ,source= nome_da_pasta_do_pc ,target= {pasta_do_container}
```

- Definir type=bind
- Source=Nome da pasta do pc
- Vírgulas juntas

- Target=Pasta do container alvo

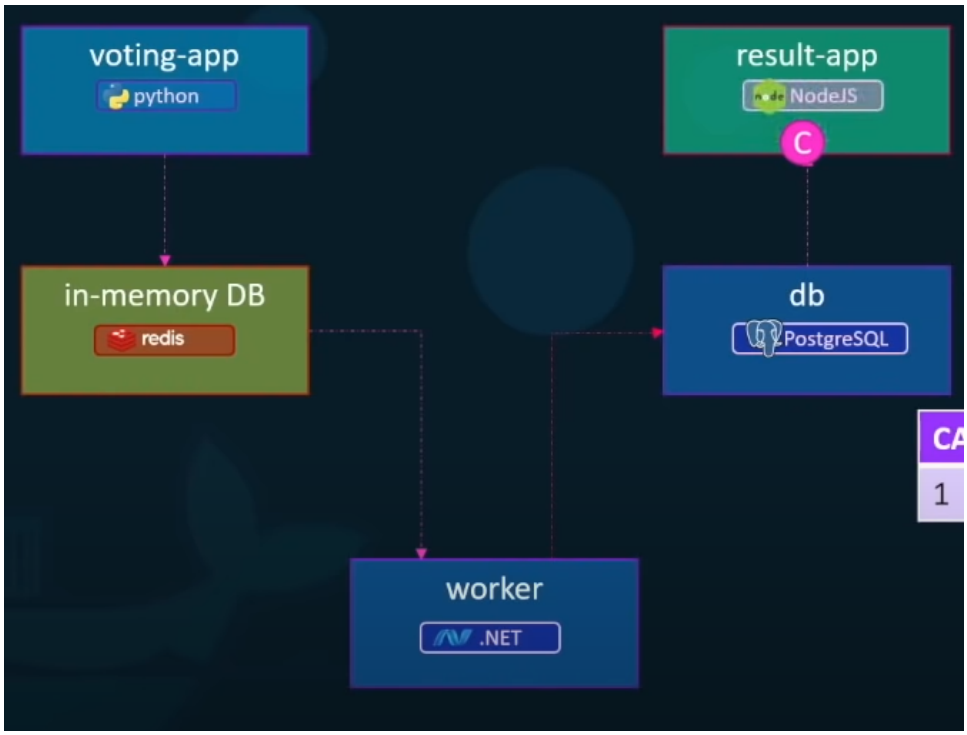
Docker Compose

Ao rodar uma aplicação Fullstack, devemos usar vários serviços interligados.

Exemplo: Uma API é consumida pelo front-end, e seus dados são armazenados em um banco de dados.

Essas três aplicações rodam em três containers diferentes, cada uma em sua respectiva porta e/ou rede.

A questão é que elas precisam rodar ao mesmo tempo e criando conexões entre si por meio dos seus ips privados.



Exemplo de aplicação onde cada um roda em um container e é necessário que haja comunicação entre si.

No caso acima, cada aplicação roda na mesma rede bridge, então cada um pode se conectar pelos seus ips privados.

Docker-compose.yaml

Esse arquivo é responsável por rodar os containeres todos de uma vez e quais são suas imagens, portas e especificações.

Imagine que é uma lista de docker runs a serem rodados.

O docker-compose é escrito em yaml - Yet Another Markdown Language, e possui sintaxe própria.

Versões

O docker-compose possui 3 versões até agora, e o ideal é usar a mais recente.

Depois da primeira versão é necessário que nós especifiquemos a versão que estamos usando do docker compose.

Exemplo de arquivo docker-compose.yaml

```
version: 3

services:
  db:
      image: postgres:9.4

  front:
      image: {imagem-do-front}
      ports:
          - 5000:80

  back:
      image: nodejs
```

Essa sintaxe roda os containers dando nome e especificação a cada um.

Como os serviços se comunicam entre si?

Normalmente, fazemos conexões por IPs privados ou domínios que são convertidos em IP pelo DNS.

Não é diferente dentro dos programas que estamos rodando nos containers, cada um precisa se conectar ao outro por meio dos seus IPs privados, mas como os pegamos?

Ao rodar uma série de containers no docker compose, **cada um deles pode ser acessado na rede pelo seu nome.**

O ip `privado:porta` de cada um pode ser acessado pelo seu nome de container por conta de alguma espécie de DNS que o docker faz.

Ou seja, dentro do seu programa em python, node ou etc, o ip privado do outro serviço vira o nome do container.

Isso pode ser armazenado em variáveis de ambiente, hardcoded, ou como você desejar.

E se um container depender de outro antes de iniciar?

Se um container depende de um outro, é necessário adicionar um parâmetro dentro da sua especificação no docker-compose

Exemplo, se eu tenho um backend que depende do banco de dados:

```
version: '3'

services:
  db:
      image: postgres:9.4
  back:
      image: nodejs
      depends_on:
          - db
```

Simples, agora o db vai rodar antes do back!

Buildando no compose

Se uma imagem precisar se buildada e usada dentro do compose, é possível fazê-lo

```
version: '3'

services:
  db:
      image: postgres:9.4
  back:
      build: {pasta do dockerfile}
```

Agora, antes do container ser executado, ele buildará a imagem a partir do dockerfile que está especificado em tal pasta.

Experiência com problemas

Deletar multiplos processos

```
docker rm $(docker ps -a -q)
```

O parâmetro `-q` faz com que somente os ids dos processos sejam mostrados.

O cifrão joga o cout do `docker ps -a -q` dentro da variavel que o `rm` espera

Problemas de conexão dentro dos processos do container

Adicionar parametros de keepalive

```
docker run --sysctl net.ipv4.tcp_keepalive_time=200 --sysctl net.ipv4.tcp_keepalive_intvl=200 --sysctl net.ipv4.tcp_keepalive_probes=5 crawler
```

Problemas com twistd PID

Não dar kill em um container rodando, vai corrompê-lo

Resetar os arquivos do docker

```
docker system prune -a
```

Ver o sistema operacional que uma imagem se baseia

```
docker run {imagem} cat /etc/*release*
```

Inspecionando variáveis de ambiente

Usando `docker inspect {container}` é possível encontrar as variáveis de ambiente setadas em `Config → Env`