# O que é Docker e como funciona

O Docker é uma plataforma que implementa virtualização de software e utiliza a tecnologia de contêineres para facilitar a implantação e execução de aplicações. Podemos entender *contêiner* como uma unidade leve e portátil que irá conter todo o necessário para executar um pedaço de software, incluindo o código, runtime, bibliotecas e dependências. Neste modelo de virtualização temos diversos benefícios,com destaque para: isolamento de contextos e o versionamento de aplicações.

**Isolamento de Contextos**

Como cada contêiner possui seu próprio sistema de arquivos, processo, espaço de rede e recursos, garantindo que a aplicação dentro do contêiner não interfira em outras aplicações ou no sistema hospedeiro, isso proporciorna um alto grau de independência e isolamento.

**Versionamento de Aplicações**

No docker as aplicações são encapsuladas em imagens, que são versões imutáveis e autossuficientes. Uma imagem docker é composta por camadas, permitindo versionamento eficiente e a reutilização de partes comuns entre diferentes aplicações. Utilizando um arquivo conhecido como Dockerfile, o versionamento é facilitado , um arquivo de configuração que descreve os passos para criar uma imagem. As alterações no Dockerfile resultam em novas versões da imagem, garantindo consistência na implantação.

O Docker proporciona uma abordagem eficiente para o desenvolvimento, empacotamento e execução de aplicações, trazendo benefícios como isolamento de contextos, consistência entre ambientes e versionamento controlado. Essas características tornam o Docker uma ferramenta poderosa para equipes de desenvolvimento e operações que buscam eficiência e confiabilidade em todo o ciclo de vida de uma aplicação.

# 

# O que são e como Containers funcionam

O container funciona da seguinte maneira: dentro de um sistema operacional, temos vários containers, isto é, diversas aplicações sendo executadas. No entanto, eles funcionarão diretamente como **processos** dentro do nosso sistema.

Enquanto uma máquina virtual terá toda aquela etapa de **virtualização** dos sistemas operacionais dentro do sistema original, os containers funcionarão diretamente como processos dentro do sistema.

Portanto, no que diz respeito ao **consumo**, podemos visualizar que ele será um pouco menor. O consumo de recursos, a carga para que ele possa ser executado, é um pouco menor, porque eles são processos, e não uma virtualização completa.

As próximas perguntas são as seguintes:

* Como um processo garantirá isolamento ?
* Como ele funcionará sem instalar um sistema operacional ?
* Como vamos conseguir resolver e responder essas perguntas ?

A questão é que, quando containers estiverem em execução no nosso sistema operacional, para garantir o isolamento entre cada um deles e o sistema operacional original, existe um conceito chamado ***namespaces***, que garantirá que cada um desses containers tenha capacidade de se isolar em determinados **níveis**

# O que são Namespaces

Teremos os principais namespaces: primeiramente, o **PID**, que garante o isolamento a nível de processo dentro de cada um dos containers. Portanto, um processo dentro de um container, que, consequentemente, é um processo dentro do sistema operacional, estará isolado de todos os outros do nosso *host*, isto é, da nossa máquina original.

Teremos também o **NET**, o namespace de **rede**, que garantirá o isolamento entre uma interface de rede de cada um dos containers e também do nosso sistema operacional original.

O namespace **IPC** será de intercomunicação entre cada um dos processos da nossa máquina. O **MNT**, que é a parte de *file system* (sistema de arquivos), montagem, volumes e afins, também estará devidamente isolado. Por fim, o **UTS** faz um compartilhamento e um isolamento ao mesmo tempo do host, isto é, do *kernel*, da máquina que executa o container.

Este último namespace específico, UTS, ajuda a responder à próxima pergunta:

* Como o conteiner dentro do sistema operacional funcoinará sem um sistema operacional ?

Na verdade, graças ao namespace UTS, se executarmos nossos containers em uma máquina com kernel *Linux*, cada um dos containers, a princípio, também terá um pedaço desse kernel, mas devidamente isolado.

Assim, conseguimos responder à pergunta: não precisamos necessariamente instalar um sistema operacional dentro de um container, porque ele já terá, graças ao namespace UTS, acesso ao kernel do sistema original, mas isoladamente.

**Cgroups**

Por fim, na parte de **gerenciamento de recursos**, suponha que queiramos definir, conforme levantado em um problema anteriormente, o consumo máximo de memória de CPU e afins para cada um dos containers.

Existe outro conceito chamado ***Cgroups***, que garantirá que podemos definir, tanto de maneira automática, quanto de maneira manual, como os consumos serão feitos para cada um desses processos, isto é, para cada um desses containers dentro do sistema operacional.

De volta às perguntas originais, graças aos namespaces e aos Cgroups, conseguimos garantir o isolamento, garantir que o nosso container funciona sem instalar um sistema operacional dentro dele, e também conseguir ter um controle de gerenciamento de recursos como **memória** e **CPU**.

Quanto à questão de por que eles são mais leves, entendemos que eles funcionarão como processos diretamente do sistema operacional, mas ao longo deste curso, entenderemos ainda mais por que eles conseguem ser tão mais práticos em relação a uma máquina virtual em termos de consumo de recursos e de tamanho de armazenamento também no nosso sistema operacional.

# Instalando Docker

Documentação oficial: <https://docs.docker.com/engine/install/ubuntu/>

Execute os comandos:

sudo apt-get update

sudo apt-get install \

ca-certificates \

curl \

gnupg \

lsb-release

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

definido a versão do docker:

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

Atualize os pacotes novamente com o comando:

sudo apt-get update

Vamos executar o comando de instalação do *Docker Community Edition* (docker-ce), do CLI do Docker (docker-ce-cli), e do containerd.io, que será responsável pela parte do funcionamento dos containers no nosso sistema.

sudo apt-get install docker-ce docker-ce-cli containerd.io

Para testar a ferramenta usar o comando : docker run hello-world – (talvez precise do sudo)

# Criando um grupo para não precisar usar o docker como sudo

sudo usermod -aG docker $USER

comando explicado:

Com o comando usermod, utilizando o sudo na frente, podemos passar o parâmetro -aG seguido de docker, que é o nome do grupo ao qual queremos adicionar o usuário, e por fim, $USER, que será o nosso próprio usuário.Obs: é necessário reiniciar a máquina após essa criação de usuário e grupo.

**Etapas do comando docker run:** Procura a imagem localmente -> Baixa a imagem caso não encontre localmente -> Valida o hash da imagem -> Executa o container.

**Diferença entre docker run e docker exec:** O docker run cria um novo container e o executa. O docker exec permite executar um comando em um container que já está em execução.

anotações gerais:

* O Docker Hub é um grande repositório de imagens que podemos utilizar;
* A base dos containers são as imagens;
* Como utilizar comandos acerca do ciclo de vida dos containers, como: docker start, para iniciar um container que esteja parado; docker stop, para parar um que esteja rodando; docker pause, para pausar um container e docker unpause para iniciar um container pausado;
* Conseguimos mapear portas de um container com as flags -p e -P.

# Construindo uma imagem

Para construir uma imagem fazemos um arquivo dockerfile, nele passamos os parâmetros que iremos utilizar para criar a imagem e consequentemente o contêiner:

**FROM** node:14

WORKDIR /app-node

**COPY** . .

RUN npm install

ENTRYPOINT npm start

**Explicação do arquivo:**

FROM node:14 **:** Para definir que queremos pegar a partir da imagem node disponível no docker hub e após os dois pontos é a versão que queremos.

WORKDIR **:** define o nosso diretório padrão por exemplo, queremos que todos esses comandos sejam executados no diretório que estamos atualmente por padrão.

**COPY** . . **:** copia o conteúdo do nosso host para nossa imagem. Podemos fazer um COPY de ponto, ou seja, esse ponto é o diretório atual dentro do nosso host, para ponto também, que será o nosso diretório atual dentro da nossa imagem.

ENTRYPOINT : Por fim, queremos que o ponto de entrada (ENTRYPOINT) do nosso container ao executar essa imagem e começar a ter seu container devidamente em execução, seja iniciar a aplicação. Para isso, usamos npm start.

Após a configuração abrir o terminal e dar o comando docker build, passamos o -t, para criar um nome, ou seja, etiquetar a nossa imagem. Nesse caso, vamos colocar sanarafelicipo/app-node, por exemplo, na versão 1. Com os dois pontos (:), podemos explicitar qual é a versão que estamos criando. ele fica assim:

**docker build -t sanarafelicio/app-node:1.0 .**

**executando container especificando as portas a serem utilizadas:**

Mais uma vez, vamos dar um docker run no terminal. Também vamos colocar o -p 8080:3000. Sabemos que a porta 300 é onde a nossa aplicação está executando. A imagem será do sanarafelicio/app-node na versão 1.0. Também adicionaremos a flag -d para rodar em modo *detached*. **Comando: docker run -d 8080:3000 danielartine/app-node:1.0**

**Parando ou removendo todos os conteiners de uma vez:** docker stop $(docker container ls -q) ou docker stop $(docker container ls -aq)

para remover no lugar do stop usar rmi

**Comando explicado:** docker container ls para dizer que queremos parar todos os que estiverem em execução como entrada para o stop por isso é preciso colocar esse segundo comando entre cifrão e parênteses ($()), Além disso, precisamos dizer que queremos pegar apenas o ID. Entre os parênteses, colocamos a flag -q, de *quiet* (silencioso), assim, ele vai pegar apenas o ID.

o a é pra pegar todos os containers inclusive os que estão parados.

Esse comando de remoção também pode ser usado para remover imagens:

docker rmi $(docker image ls -aq)

às vezes será necessário forçar a remoção para isso: –force no final do comando

## **Lendo uma variável de ambiente**

Basicamente, apenas atribuímos que queremos ler uma variável de ambiente chamada PORT desta maneira: process.env.PORT, no arquivo index.js:

app.listen(**process.env.PORT,** ()=>{ console.log("Server is listening on port 3000")})

definir na nossa imagem que vamos querer receber esse parâmetro, definindo que vamos ter um argumento com a instrução ARG. Esse argumento será usado para definir essa variável de ambiente dentro do nosso container posteriormente, que vai ser a porta que vamos querer utilizar. Então colocamos o EXPOSE com essa porta, e para obter o valor dessa variável que estamos utilizando dentro da criação da nossa imagem usamos $PORT. Exemplo do arquivo dockerfile:

**FROM** node:14

WORKDIR /app-node

ARG PORT=6000

EXPOSE $PORT

**COPY** . .

RUN npm install

ENTRYPOINT npm start

**Diferença entre ENV e ARG:**

A instrução ARG carrega variáveis apenas no momento de build da imagem, enquanto a instrução ENV carrega variáveis que serão utilizadas no container.

**Subindo a imagem para o Docker hub**

Após criar uma conta no [dockerhub](https://hub.docker.com/repository/docker/sanara2707/app-node/general) e fazer o seu login, vamos no terminal fazer o login para poder fazermos um push no dockerhub.

Para isso no terminal coloque o comando: **docker login -u sanarafm27@gmail.com**

Após executamos o comando docker tag com a imagem que vai ser copiada e a imagem que queremos gerar. Nesse caso, queremos gerar sanarafelicio/app-node:1.0 . Após geramos é só fazer o push para enviar a imagem:

**docker tag sanarafelicio/app-node:1.0 sanara2707/app-node:1.0**

**docker push sanara2707/app-node:1.2**

**Anotações:**

* Imagens são imutáveis, ou seja, depois de baixadas, múltiplos containers conseguirão reutilizar a mesma imagem.
* Imagens são compostas por uma ou mais camadas. Dessa forma, diferentes imagens são capazes de reutilizar uma ou mais camadas em comum entre si.

## Como persistir informações entre containers

Há 3 formas de fazer isso:

A primeira é o ***bind-mount***, que é uma maneira de fazer um vínculo (*bind*) entre o sistema de arquivos do nosso sistema operacional e o sistema de arquivos do nosso container. Assim, teremos uma ponte entre esses dois que persistirá essa informação no nosso host.

Temos o ***volume***, efetivamente, que será gerenciado pelo Docker, o que vamos entender em mais detalhes adiante.

Por fim, temos o ***tmpfs mount***, que é temporário, cuja utilidade também vamos entender mais adiante.

**Persistindo dados com Bind-mount**

Basicamente, essa solução faz uma ligação entre um ponto de montagem do nosso sistema operacional e algum diretório dentro do container.

No entanto, a ideia agora é definir que, quando esse container for criado e executado, as informações persistidas em determinado diretório dentro dele sejam persistidas em algum diretório na nossa máquina, localmente, no nosso *host.*

*Para isso, além de definir os comandos básicos e a flag -it, podemos colocar uma flag -v.* ficando assim o comando:

***docker run -it -v ubuntu***

*Com essa flag, podemos dizer que queremos que o diretório no nosso host corresponda a um determinado caminho dentro do nosso container.*

**docker run -it -v /home/sanara/Documentos/Cursos/Docker/volume-docker:/app ubuntu bash**

Assim, tudo o que for gravado dentro desse diretório /app será persistido no diretório /home/nome-de-usuario/volume\_docker do nosso *host* (o nome de usuário deve ser o da sua máquina).

Podemos substituir o comando -v por uma sintaxe mais verbos, porém mais intuitiva ficando assim:

**docker run -it --mount type=bind,source=/home/sanara/Documentos/Cursos/Docker/volume-docker,target=/app ubuntu bash**

No comando acima, dissemos que queremos fazer um *mount* do tipo *bind*, e que o diretório da nossa máquina, nossa *source*, vai ser /home/nome-de-usuario/volume-docker. Já o nosso *target* vai ser /app também, dentro desse container.

## Persistindo dados com Volumes

A persistência com volumes é a mais recomendada pelo docker, os volumes são uma **área gerenciada pelo Docker** dentro do sistema de arquivos. Então, mesmo que as nossas informações continuem no nosso *host* original para serem persistidas, nós teremos uma área que o Docker vai gerenciar, o que é muito mais seguro em termos de alguém fazer alterações indesejadas ou causar algum problema. Para criar um volume usamos o comando: **docker volume create meu-volume**

Mas aonde o docker armazena na nossa máquina ? Existe um diretório na nossa máquina onde o Docker está realmente; ou melhor, onde estão as diversas informações que o Docker armazena na nossa máquina. O caminho para ela é /var/lib/docker, este é um caminho completamente gerenciado pelo docker para acessa-lo basta usar o comando: **cd /var/lib/docker**

## Gerenciamento de Volume

* create: para criar volumes
* inspect: para inspecioná-los
* ls: para lista-los
* prune: para remover os volumes que não estãos endo usados
* rm : para remover qualquer volume sendo usado ou não.

Ou seja, conseguimos gerenciar esses volumes por meio da interface do Docker e não ficamos 100% dependentes do sistema de arquivos do nosso sistema operacional, pois o Docker é quem vai gerenciar isso para nós agora. Isso é muito interessante, porque agora não dependemos diretamente de uma estrutura de pastas específica do nosso sistema operacional - ele vai estar sempre nesse diretório de volumes.

## Criando volume com –mount

Também temos a possibilidade de criar um volume com a flag --mount. Essa maneira é até mais fácil, porque por padrão ele assume que o tipo que vamos criar é um volume, então não precisamos colocar o tipo, como fizemos com o *bind*. Comando:

**docker run -it --mount source=meu-volume,target=/app ubuntu bash**

## Persistindo dados com tmpfs

A primeira é que tmpfs só funcionará no *host* Linux.

Criando um novo conteiner com tmpfs : docker run -it --tmpfs=/app ubuntu bash

Vamos dar um ls para conferir seu conteúdo. Ele criou a pasta "app" mas, diferente das outras vezes em que criamos essa pasta, agora ela está destacadaemverde no retorno do ls, assim como a pasta "tmp".

Isso significa que a pasta "app" é temporária. Ela está sendo escrita na memória do nosso *host*. Ou seja, no momento em que esse container parar de funcionar, os arquivos da pasta "app" serão perdidos.

A ideia do *tmpfs* é, basicamente, persistir dados na memória do seu *host*, mas esses dados não estão sendo escritos na camada de *read-write*. Eles são escritos diretamente na memória do *host*.

Isso é importante quando temos algum dado sensível que não queremos persistir na camada de *read-write*, por questões de segurança, mas queremos tê-los de alguma maneira.

## TMPFS com a flag –mount

**docker run -it --mount type=tmpfs,destination=/app ubuntu bash**

Se executarmos ls dentro do container, mais uma vez as pastas "app" e "tmp" estarão destacadas em verde, como pastas temporárias.

Resumo persistência de dados com docker:

Conhecemos três possibilidades de persistência de dados: o **tmpfs**; os ***bind-mounts***, que fazem uma ligação direta entre o sistema de pastas do nosso host e do nosso container; e os **volumes**, que são a solução recomendada, pois são gerenciados pelo Docker e permitem um controle maior, sem depender, diretamente, da estrutura de pastas do nosso *host*.

## Comunicação através de redes

Criando uma rede Bridge

Podemos criar um container com o nome e posteriormente utilizar o nome para fazer a comunicação de redes do nosso container :

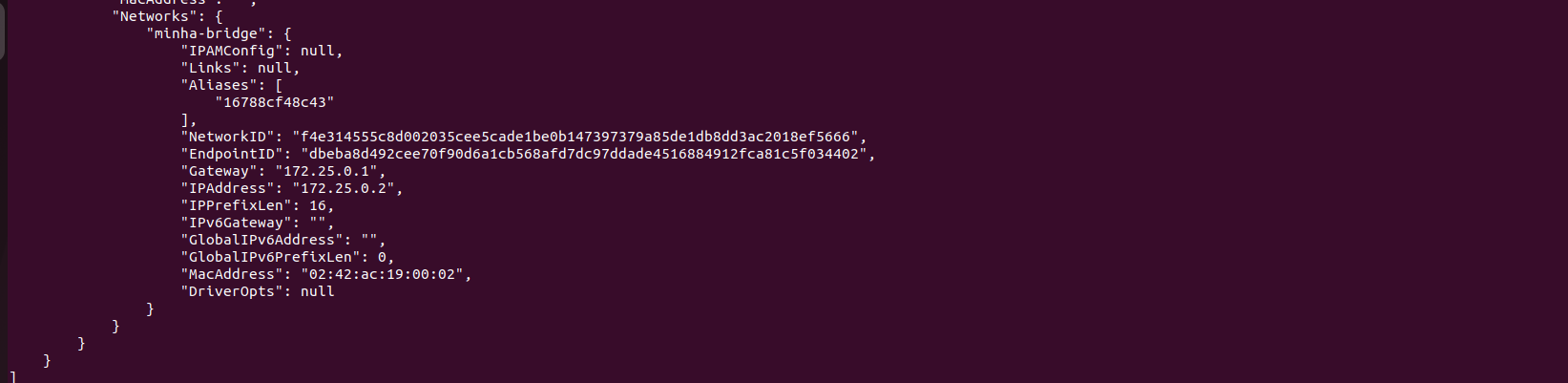
docker run -it --name ubuntu1 ubuntu bash

Para criar a nossa própria rede usamos o comando: docker network create --driver bridge minha-bridge

No momento em que vamos criar o nosso contêiner, além de definir o nome dele, também definiremos a rede através do --network minha-bridge:

docker run -it --name ubuntu1 --network minha-bridge ubuntu bash

se criarmos 2 containers na mesma rede criada podemos fazer a comunicação entre eles via *hostname* com a *user-defined bridge*, Ele mostra que as redes *user-defined bridge*, ou seja, as redes que são criadas por usuários de *bridge*, provêem essa resolução automática de DNS entre contêineres, que é basicamente o que estamos fazendo aqui agora.



Rede None e Host:

Começaremos pela rede none, que utiliza o driver null. Vamos executar um container com a rede none:

**docker run -d --network none ubuntu sleep 1d**

Na prática, quando utilizamos o driver none, estamos simplesmente indicando que esse contêiner não terá qualquer interface de rede vinculada a ele. Com isso, o contêiner fica completamente isolado em termos de rede, não conseguindo realizar nenhuma operação que envolva a rede do contêiner, porque seu driver é **none**, ou seja, utiliza o driver null.

Agora, se quisermos fazer o contrário, ou seja, que o nosso contêiner tenha uma interface de rede, já vimos como fazer com a **bridge**. Mas em alguns casos, queremos que a interface de rede seja mais vinculada ao nosso **host**.

Vamos criar um container com a rede **host**:

**docker run -d --network host aluradocker/app-node:1.0**

Isso significa que agora, se abrirmos uma nova aba do navegador e tentarmos acessar essa aplicação, conseguiremos fazê-lo mesmo sem ter feito o mapeamento de portas. Isso ocorre porque, ao utilizar o driver host, removemos quaisquer isolamentos que existiam entre a interface de rede do contêiner e do host.Estamos utilizando a mesma rede, usamos a mesma interface do host que está hospedando esse contêiner. Portanto, se houvesse alguma outra aplicação na porta 3000 com o host em execução, não conseguiríamos utilizar o contêiner dessa maneira, pois haveria um conflito de portas, já que a interface seria a mesma.

## Comunicação aplicação e Banco

baixar a imagem do mongo na máquina: **docker pull mongo:4.4.6**

baixar a imagem do alura books na máquina: **docker pull aluradocker/alura-books:1.0**

Em seguida, iremos executar o container responsável pelo banco de dados. Para isso, execute o comando **docker run -d --network minha-bridge --name meu-mongo mongo:4.4.6.**

Precisamos agora executar o container responsável pela aplicação alura books que irá se comunicar com o banco de dados. Para isso, execute o comando **docker run -d --network minha-bridge --name alurabooks -p 3000:3000 aluradocker/alura-books:1.0.**

utilizamos a flag **-p** para em seguida validar o funcionamento da aplicação através de nosso host

Agora, estamos fazendo um docker run com o contêiner nessa rede. O nosso contêiner de alura-books vai se comunicar com o banco de dados mongo, e como eles estão na mesma rede bridge criada manualmente, a comunicação pode ser feita via *hostname*.

Resumo:

* O docker dispõe por padrão de três redes: bridge, host e none;
* A rede bridge é usada para comunicar containers em um mesmo host
* Redes bridges criadas manualmente permitem comunicação via hostname
* A rede host remove o isolamento de rede entre o container e o host
* A rede none remove a interface de rede do container
* Podemos criar redes com o comando docker network create

## Conhecendo o Docker Compose

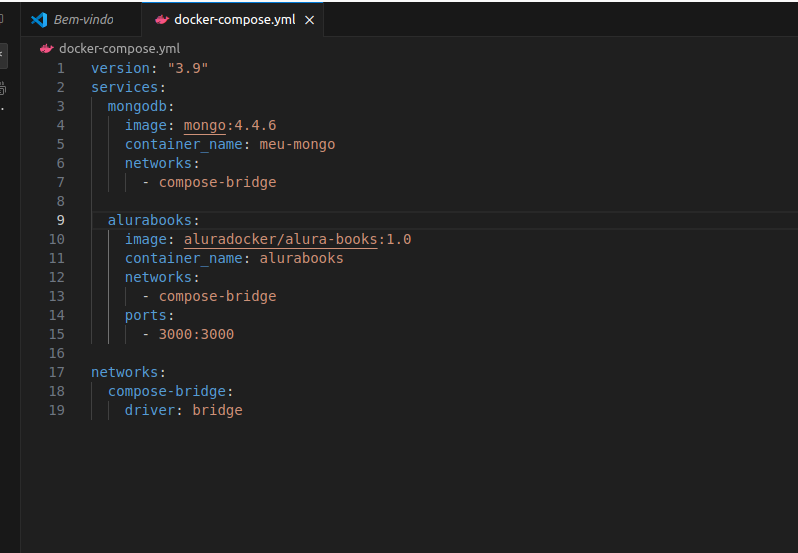
Anteriormente, executamos o container *Alura Docker* da imagem *Alura Books* na versão 1.0 juntamente com o *Mongo* na versão 4.4.6.

Porém, se nossa aplicação crescer, teremos que subir vários containers manualmente. Nesse caso, sempre que quisermos parar um container, teremos que usar o comando docker stop ou docker rm para remover um a um. Para cada container será necessário executar um comando, além da pilha de execução com todos esses containers.

Existe uma solução do próprio *docker* que nos ajudará a resolver esse tipo de situação, que é o *Docker Compose*.

O *Docker Compose* é uma ferramenta de coordenação de containers, lembrando que é diferente de orquestração. Ele nos auxilia a compor vários containers em um mesmo ambiente por meio de um único arquivo. Assim, podemos compor uma aplicação maior com nossos containers usando-o.Faremos isso por meio da definição de um arquivo *YAML*, um tipo de estrutura específica baseada na indentação.

Arquivo docker-compose.yml

****

version: "3.9" : versão que utilizaremos, nesse caso será a 3.9.

services: definir os serviços que serão utilizados por esse container, nesse teremos um serviço chamado mongodb e outro alura books.

image: mongo:4.4.6: a imagem que esse serviço vai utilizar.

container\_name: meu -mongo: nome do container .

networks: Configura a rede que o container usar. nesse exemplo utilizaremos a rede compose-bridge.

Agora, precisamos fazer o mesmo para o Alura Books.

ports: as portas que o container será executado na nossa máquina 3000:3000.

networks: deve ser alinhado com a parte de services, definimos a rede compose-bridge.

driver: para definir o tipo de rede que iremos utilizar: driver bridge

Após isso é só entrar na pasta aonde está seu arquivo docker-compose.yml pelo terminal e dar o comando: **docker compose up**

adicionando a anotação depends\_on:

Uma delas é a depends\_on, que expressa **dependência entre serviços**.Quando definimos uma dependência de um serviço para outro, ele inicia o serviço nessa **ordem específica**. Porém, existe um detalhe a ser lembrado, o depends\_on **não espera** necessariamente que a aplicação dentro do container esteja pronta para receber as requisições. Ele aguarda apenas que o **container esteja pronto**, o que não significa que a aplicação dentro do container esteja preparada.

