

## Transcrição

Antes de vermos sobre o Docker, precisamos saber mais como hospedamos aplicações e como surgiram os *containers*.

## A evolução do host de aplicações

Antigamente, quando queríamos montar o nosso sistema, com vários serviços (banco de dados, proxy, etc) e aplicações, acabávamos tendo vários servidores físicos, cada um com um serviço ou aplicação do nosso sistema, por exemplo:



E claro, não conseguimos instalar os serviços diretamente no hardware do servidor, ou seja, precisamos de um intermediário entre as aplicações e o hardware, que é o **sistema operacional**. Ou seja, devemos instalar sistemas

operacionais em cada servidor, e os sistemas poderiam ser diferentes:



E se quisermos que uma aplicação se comunique com outra ou faça qualquer comunicação externa, devemos conectar os servidores à rede. Além disso, para eles funcionarem, precisamos ligá-los à eletricidade. Logo, havia diversos custos de eletricidade, rede e configuração dos servidores.

Além disso, o processo era lento, já que a cada nova aplicação, deveríamos comprar/montar o servidor físico, instalar o sistema operacional, configurá-lo e subir a aplicação.

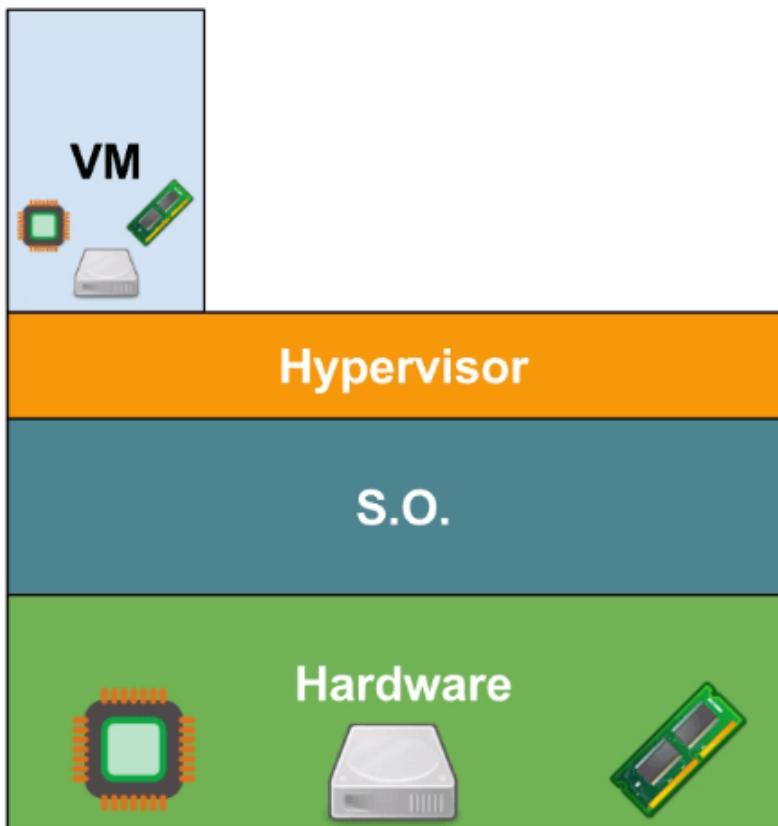
## Capacidade pouco aproveitada

O que foi falado anteriormente não era o único problema desse tipo de arquitetura. Era muito comum termos servidores parrudos, com uma única aplicação sendo executada, para normalmente ficarem funcionando abaixo da sua capacidade, para quando for necessário, o servidor aguentar uma grande quantidade de acessos. Isso resultava em muita capacidade ociosa nos servidores, com muitos recursos desperdiçados.

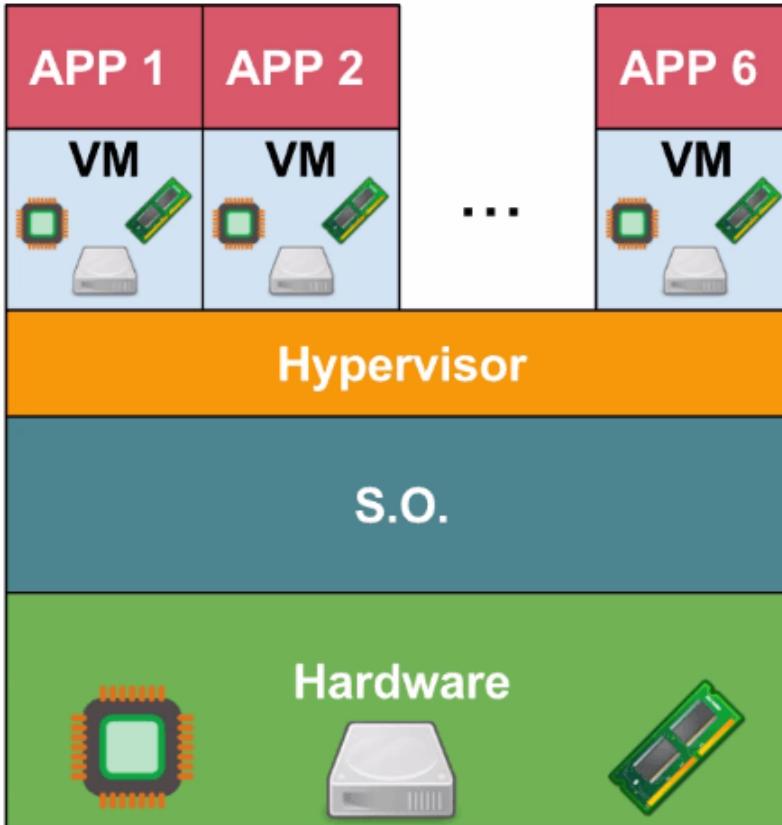
## Virtualização, uma solução?

Para fugir desses problemas de servidores ociosos e alto tempo e custo de subir e manter aplicações em servidores físicos, surgiu como solução a **virtualização**, surgindo assim as **máquinas virtuais**.

As máquinas virtuais foram possíveis de ser criadas graças a uma tecnologia chamada **Hypervisor**, que funciona em cima do sistema operacional, permitindo a virtualização dos recursos físicos do nosso sistema. Assim, criamos uma máquina virtual que tem acesso a uma parte do nosso HD, memória RAM e CPU, criando um computador dentro de outro:



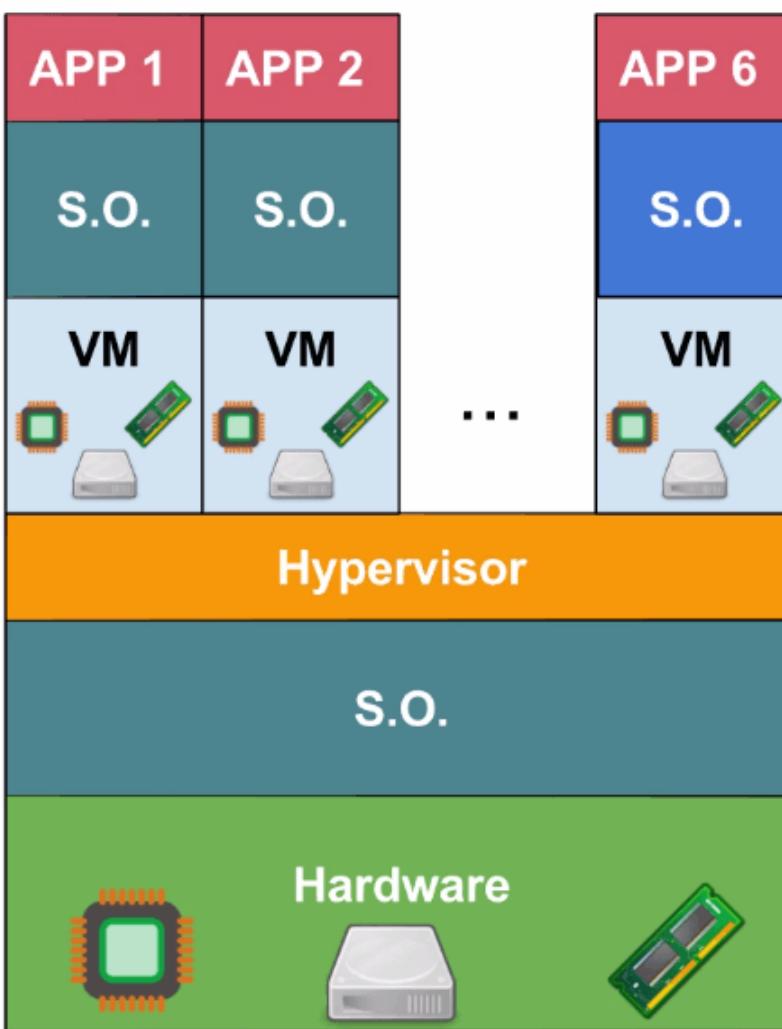
E se temos uma máquina virtual que está acessando uma parte do nosso hardware como um todo, conseguimos colocar dentro dela uma das nossas aplicações. E replicar isso, criando mais máquinas virtuais com outras aplicações:



Assim, reduzimos a quantidade de servidores e consequentemente os custos com luz e rede. Além disso, dividimos melhor o nosso hardware, aproveitando melhor os seus recursos e diminuindo a ociosidade.

## Problemas das máquinas virtuais

Apesar de resolver os problemas do uso de vários servidores físicos, as máquinas virtuais também possuem problemas. Para podermos hospedar a nossa aplicação em uma máquina virtual, também precisamos instalar um sistema operacional nela:



Cada aplicação necessita de um sistema operacional para poder ser executada, e esses sistemas podem ser diferentes. E apesar dos sistemas operacionais serem um software, eles possuem um custo de memória RAM, disco e processamento. Ou seja, precisamos de uma capacidade mínima para manter as funcionalidades de um sistema operacional, aumentando o seu custo de manutenção a cada sistema que tivermos.

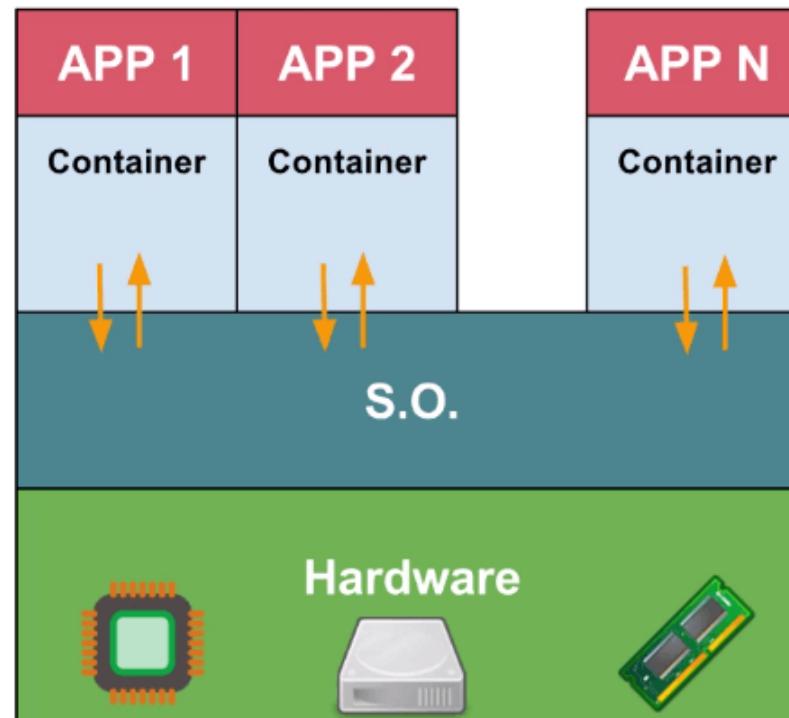
Além disso, há um custo de configuração, isto é, liberar portas, instalar alguma biblioteca específica, etc, toda uma configuração que um sistema operacional pede. Também devemos sempre estar atentos à sua segurança, mantendo o sistema de cada máquina virtual sempre atualizado.

Muitas vezes, o tempo voltado para a manutenção das máquinas virtuais era o mesmo tempo voltado para a nossa aplicação em si. Ou seja, acabávamos dividindo o valor da nossa empresa, ao invés de focar somente nas aplicações, dividíamos o trabalho com a manutenção dos sistemas operacionais.

Então, como melhorar essa situação? Daí surgiram ***containers***, que serão vistos no próximo vídeo.

## Transcrição

Um *container* funcionará junto do nosso sistema operacional base, e conterá a nossa aplicação, ou seja, a aplicação será executada dentro dele. Criamos um *container* para cada aplicação, e esses *containers* vão dividir as funcionalidades do sistema operacional:



Não temos mais um sistema operacional para cada aplicação, já que agora as aplicações estão dividindo o mesmo sistema operacional, que está em cima do nosso hardware. Os próprios *containers* terão a lógica que se encarregará dessa divisão.

Assim, com somente um sistema operacional, reduzimos os custos de manutenção e de infraestrutura como um todo.

## Vantagens de um container

Por não possuir um sistema operacional, o *container* é muito mais leve e não possui o custo de manter múltiplos sistemas operacionais, já que só teremos um sistema operacional, que será dividido entre os *containers*.

Além disso, por ser mais leve, o *container* é muito rápido de subir, subindo em questão de segundos. Logo, o *container* é uma solução para suprir o problema de múltiplas máquinas virtuais em um hardware físico, já que com o *container*, nós dividimos o sistema operacional entre as nossas aplicações.

## Necessidade do uso de containers

Mas por que precisamos dos *containers*, não podemos simplesmente instalar as aplicações no nosso próprio sistema operacional? Até por que já fazemos isso, já que no nosso sistema operacional temos um editor de texto, terminal, navegador, etc.

No caso das nossas aplicações, essa abordagem pode ter alguns problemas. Por exemplo, se dois aplicativos precisarem utilizar a mesma porta de rede? Precisaremos de algo para isolar uma aplicação da outra. E se uma aplicação consumir toda a CPU, a ponto de prejudicar o funcionamento das outras aplicações? Isso acontece se não limitarmos a aplicação. Outro problema que pode ocorrer é cada aplicação precisar de uma versão específica de uma linguagem, por exemplo, uma aplicação precisa do Java 7 e outra do Java 8. Além disso, uma aplicação pode acabar congelando todo o sistema. Por isso é bom ter essa separação das aplicações, isolar uma da outra, e isso pode ser feito com os *containers*.

Com os *containers*, conseguimos limitar o consumo de CPU das aplicações, melhorando o controle sobre o uso de cada recurso do nosso sistema (CPU, rede, etc). Também temos uma facilidade maior em trabalhar com versões específicas de

linguagens/bibliotecas, além de ter uma agilidade maior na hora de criar e subir *containers*, já que eles são mais leves que as máquinas virtuais.



## Transcrição

Agora que já vimos a diferença entre máquinas virtuais e *containers*, chegou a hora de introduzirmos o **Docker** nesse contexto, que se divide entre duas coisas: a **Docker, Inc.**, empresa que toma conta do Docker, e a tecnologia dos *containers*.

## Docker, Inc.

Primeiramente, devemos falar sobre a **Docker, Inc.**, que no início era chamada de **dotCloud**. A **dotCloud** era uma empresa de **PaaS** (*Platform as a Service*), sendo responsável pela hospedagem da nossa aplicação, levantando o servidor, configurando-o, liberando portas, etc, fazendo tudo o que é necessário para subir a nossa aplicação. Outros exemplos de empresas de **PaaS** são o **Heroku**, **Microsoft Azure** e **Google Cloud Platform**.

Inicialmente, para prover a parte de infraestrutura, a **dotCloud** utilizava o **Amazon Web Services** (AWS), serviço que nos disponibiliza máquinas virtuais e físicas para trabalharmos. E para hospedar uma aplicação, sabemos que precisamos do sistema operacional, mas a **dotCloud** introduziu o conceito de *containers* na hora de subir uma aplicação, dando origem ao **Docker**, tecnologia utilizada para baratear o custo de hospedar várias aplicações em uma mesma máquina.

Ou seja, quando a **dotCloud** criou o **Docker**, sua intenção era economizar os gastos da empresa, subindo várias aplicações em *containers*, em um mesmo hardware do AWS, e com o passar do tempo a empresa percebeu que tinham muitos desenvolvedores interessados na tecnologia que ela havia criado, a tecnologia que permite a criação de *containers*, que faz o intermédio entre eles e o sistema operacional, o **Docker**.

## As tecnologias do Docker

O **Docker** nada mais é do que uma coleção de tecnologias para facilitar o *deploy* e a execução das nossas aplicações. A sua principal tecnologia é a **Docker Engine**, a plataforma que segura os *containers*, fazendo o intermédio entre o sistema operacional.

Outras tecnologias do Docker que facilitam a nossa vida e que veremos neste curso são o **Docker Compose**, um jeito fácil de definir e orquestrar múltiplos *containers*; o **Docker Swarm**, uma ferramenta para colocar múltiplos *docker engines* para trabalharem juntos em um *cluster*; o **Docker Hub**, um repositório com mais de 250 mil imagens diferentes para os nossos *containers*; e a **Docker Machine**, uma ferramenta que nos permite gerenciar o Docker em um *host* virtual.

## Open Source

Quando a empresa **dotCloud** tornou-se a **Docker, Inc.**, focada em manter o **Docker**, ela o abriu para o mundo *open source*, tudo disponibilizado no seu [GitHub](https://github.com/docker) (<https://github.com/docker>), inclusive com várias empresas contribuindo para o desenvolvimento dessa tecnologia.

Apesar de haver alguns serviços pagos, em sua grande parte a tecnologia do Docker é uma tecnologia *open source*, utilizada por várias empresas. Então, vamos colocar as mãos na massa e aprender a instalar o Docker nas próximas aulas.



## Transcrição

Como o Docker é uma aplicação que se liga fortemente ao sistema operacional e é dependente de várias de suas funcionalidades, a instalação para cada um dos sistemas operacionais é diferente, e vamos abordar o caso do Windows neste vídeo.

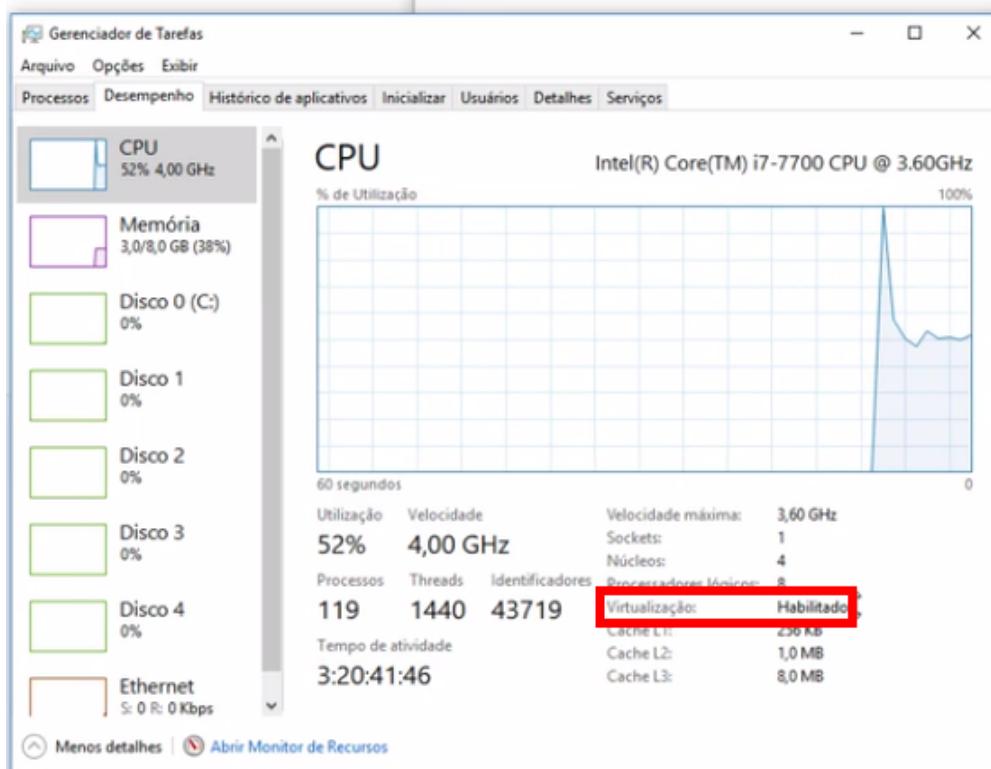
## Instalação principal no Windows

Existem duas possibilidades para instalar o Docker no Windows. Temos a principal, utilizando o [Docker for Windows](https://store.docker.com/editions/community/docker-ce-desktop-windows) (<https://store.docker.com/editions/community/docker-ce-desktop-windows>), no qual podemos baixar o instalador clicando [aqui](https://download.docker.com/win/stable/InstallDocker.msi) (<https://download.docker.com/win/stable/InstallDocker.msi>) e a alternativa, utilizando o Docker Toolbox, que veremos daqui a pouco.

Primeiramente, devemos nos atentar aos requisitos do uso do **Docker for Windows**, ou seja, devemos possuir um Windows com:

- Arquitetura 64 bits
- Versão **Pro, Enterprise ou Education**.
- Virtualização habilitada

Em relação a este último ponto, o Windows **por padrão** já deixa a virtualização habilitada, podemos conferir acessando o **Gerenciador de Tarefas**, e indo na aba *Performance*:



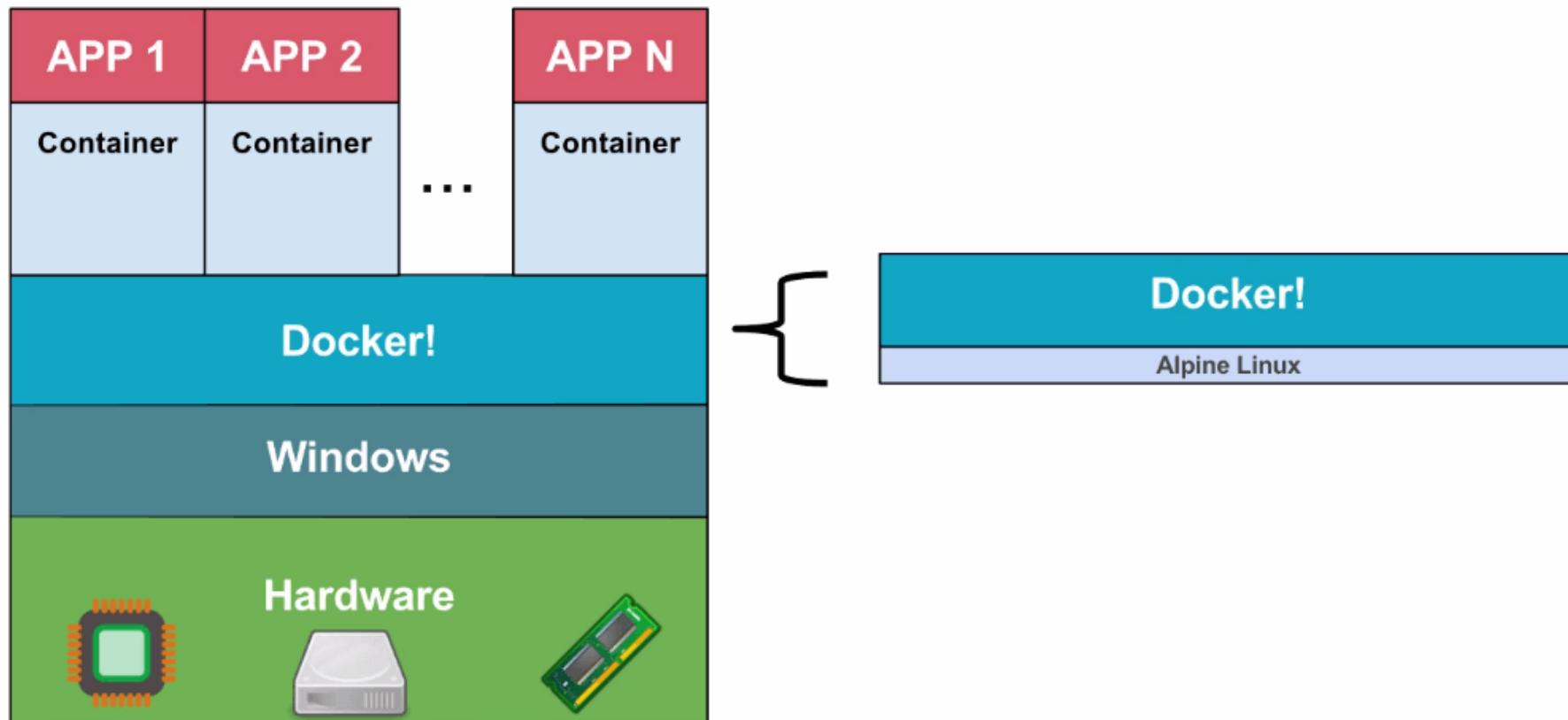
*Se no seu caso a virtualização não estiver habilitada, por favor poste no fórum do curso a versão do seu Windows e modelo do seu computador para tentarmos ajudá-lo pois cada fabricante de Hardware configura isto de modos diferentes.*

Seguimos o passo a passo do instalador para aceitar a licença, autorizamos o instalador e seguimos com a instalação. Ao clicar em **Finish**, precisamos encerrar a sessão do Windows e iniciá-la novamente. Ao fazer o login, precisamos habilitar o *Hyper-V*, clicando em **Ok**, para que o computador será reiniciado.

Quando o computador terminar a reinicialização, irá aparecer um ícone do Docker na barra inferior, à direita, ao lado do relógio. O Docker pode demorar um pouco para inicializar, mas quando a mensagem *Docker is running* for exibida, significa que ele foi instalado com sucesso e já podemos utilizá-lo.

# Funcionamento do Docker no Windows

O Docker é executado em cima de uma *micromáquina virtual*, chamada *Alpine Linux*, onde será executada a sua **Docker Engine**:



Mas para criar máquinas virtuais, o Docker precisa utilizar uma tecnologia chamada de *Hyper-V*, que é um *Hypervisor*. O problema disto é que o *Hyper-V* só está presente nas versões **Professional**, **Education** e **Enterprise**, ou seja, a maioria dos usuários comuns, que utilizam a versão **Home Edition**, não poderão instalar o Docker pelo modo tradicional, e terá que utilizar o **Docker Toolbox**.

Além disto, a principal ferramenta de instalação , o **Docker for Windows** necessita que você esteja utilizando um **Windows 10 - 64 bits**, com uma das versões citadas acima.

Vamos agora então detalhar o processo de instalação para cada esse caso.

## Instalação alternativa no Windows

Para instalar o **Docker Toolbox**, primeiramente devemos baixá-lo [aqui](#)

(<https://download.docker.com/win/stable/DockerToolbox.exe>). Ainda assim, precisamos garantir que o nosso Windows seja **64bits** e que ele tenha a virtualização habilitada.

O **Docker Toolbox** vai instalar tudo que é necessário para que trabalhemos com o Docker em nosso computador, pois ele irá instalar também a **Oracle VirtualBox**, a máquina virtual da Oracle que vai permitir executarmos o Docker sem maiores problemas.

A diferença é que, quando trabalhamos com o **Docker for Windows**, podemos utilizar o terminal nativo do Windows, já no **Docker Toolbox**, ele instalará o **Docker Machine**, que deverá ser utilizado no lugar do terminal nativo do Windows.



## Transcrição

Para instalar o Docker no MacOS, utilizamos o [Docker for Mac](https://store.docker.com/editions/community/docker-ce-desktop-mac) (<https://store.docker.com/editions/community/docker-ce-desktop-mac>), no qual podemos baixar o instalador clicando [aqui](https://download.docker.com/mac/stable/Docker.dmg) (<https://download.docker.com/mac/stable/Docker.dmg>).

Primeiramente, devemos nos atentar aos requisitos do uso do **Docker for Mac**, ou seja, devemos possuir um MacOS:

- Modelo 2010 ou mais recente;
- Versão OS X El Capitan 10.11 ou mais recente;
- Com no mínimo 4GB de memória RAM;
- Sem VirtualBox instalada na versão 4.3.30 ou anterior, pois causa incompatibilidade com o Docker.

A página dos requisitos pode ser acessada [aqui](https://docs.docker.com/docker-for-mac/install/#what-to-know-before-you-install) (<https://docs.docker.com/docker-for-mac/install/#what-to-know-before-you-install>).

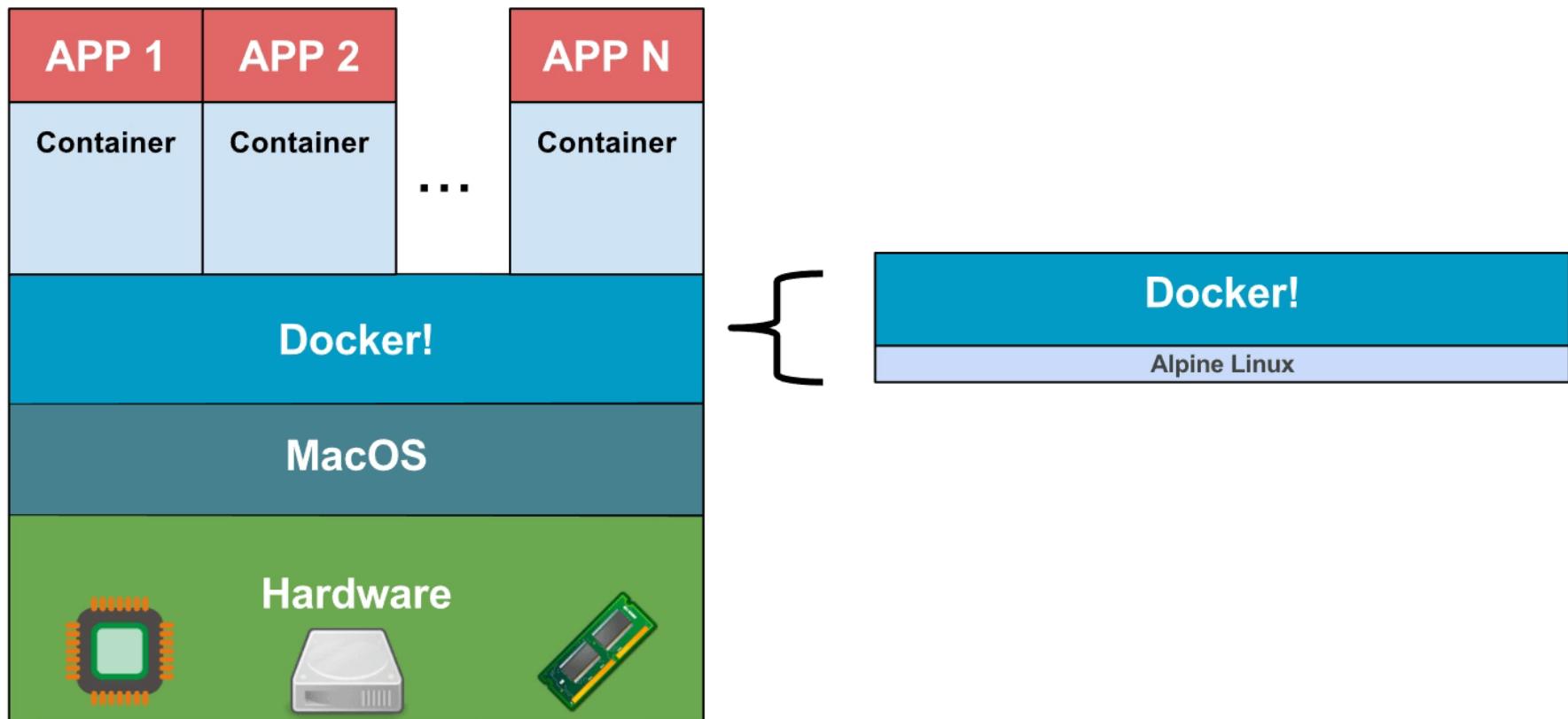
*Caso você não atinja algum desses requisitos, há uma instalação alternativa, o **Docker Toolbox**, que será visto mais adiante, então não se preocupe.*

Com isso, podemos instalar o Docker, clicando no **.dmg** baixado anteriormente e arrastando o Docker para as nossas aplicações. Após isso, já podemos pesquisar pelo Docker, confirmar que queremos utilizá-lo e damos acesso privilegiado a ele, clicando em **OK** e digitando a senha de administrador em seguida.

No menu superior do MacOS, à direita, o ícone do Docker aparecerá. Ele pode demorar um pouco para inicializar, mas quando a mensagem ***Docker is now up and running!*** for exibida, significa que ele já pode ser utilizado.

## Funcionamento do Docker no MacOS

O Docker é executado em cima de uma *micro máquina virtual*, chamada ***Alpine Linux***, onde será executada a sua **Docker Engine**:



Mas para criar máquinas virtuais, o Docker precisa utilizar uma tecnologia chamada de ***HyperKit***, que é um ***Hypervisor***. O problema disto é que o *HyperKit* só está presente na versão OS X El Capitan 10.11 ou mais recente. Mas uma alternativa é instalar o **Docker Toolbox**.

Vamos agora então detalhar o processo de instalação para cada esse caso.

## Instalação alternativa no MacOS

Para instalar o **Docker Toolbox**, primeiramente devemos baixá-lo [aqui](#) (<https://download.docker.com/mac/stable/DockerToolbox.pkg>).

O **Docker Toolbox** vai instalar tudo que é necessário para que trabalhemos com o Docker em nosso computador, pois ele irá instalar também a **Oracle VirtualBox**, a máquina virtual da Oracle que vai permitir executarmos o Docker sem maiores problemas.

A diferença é que, quando trabalhamos com o **Docker for Mac**, podemos utilizar o terminal nativo do MacOS, pois o terminal está sendo executado dentro do próprio sistema operacional. Já no **Docker Toolbox**, ele instalará o **Docker Machine**, que deverá ser utilizado no lugar do terminal nativo do MacOS.

## Preparando o ambiente: Instalando Docker no Ubuntu

Neste passo-a-passo, será visto como instalar o Docker no Ubuntu 64 bits. Todos os comandos listados devem ser executados no seu terminal.

Antes de mais nada, remova possíveis versões antigas do Docker:

```
sudo apt-get remove docker docker-engine docker.io
```

[COPIAR CÓDIGO](#)

Depois, atualize o banco de dados de pacotes:

```
sudo apt-get update
```

[COPIAR CÓDIGO](#)

Agora, adicione ao sistema a chave GPG oficial do repositório do Docker:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

[COPIAR CÓDIGO](#)

Adicione o repositório do Docker às fontes do APT:

```
sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
```

[COPIAR CÓDIGO](#)

Atualize o banco de dados de pacotes, para ter acesso aos pacotes do Docker a partir do novo repositório adicionado:

```
sudo apt-get update
```

[COPIAR CÓDIGO](#)

Por fim, instale o pacote **docker-ce**:

```
sudo apt-get install docker-ce
```

[COPIAR CÓDIGO](#)

Caso você queira, você pode verificar se o Docker foi instalado corretamente verificando a sua versão:

```
sudo docker version
```

[COPIAR CÓDIGO](#)

E para executar o Docker sem precisar de **sudo** , adicione o seu usuário ao grupo **docker** :

```
sudo usermod -aG docker $(whoami)
```

[COPIAR CÓDIGO](#)



## Transcrição

Com o Docker instalado no nosso sistema operacional (seja ele qual for), já podemos testá-lo para ver o seu funcionamento.

Se o nosso Docker foi instalado pelo **Docker for Mac** ou **Docker for Windows**, conseguimos executar os seus comandos através do terminal nativo do Mac ou do Windows (Prompt de Comando). Mas se o nosso Docker foi instalado pelo **Docker Toolbox**, devemos executar os seus comandos através do **Docker Quickstart Terminal**, terminal que foi instalado pelo próprio **Docker Toolbox**.

Então, vamos abrir um terminal que consiga se comunicar com o nosso Docker, e executar o seguinte comando para verificar a sua versão:

```
docker version
```

[COPIAR CÓDIGO](#)

Também podemos executar o clássico *Hello World*:

```
docker run hello-world
```

[COPIAR CÓDIGO](#)

Ao executar o comando, a primeira mensagem impressa é:

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

[COPIAR CÓDIGO](#)

Ou seja, o Docker não conseguiu achar a imagem localmente, e ele foi em algum lugar e a baixou. Como assim? Quando executamos o comando `docker run hello-world`, estamos dizendo para o Docker criar um *container* com a imagem do **hello-world**. Como não possuímos essa imagem localmente, ele foi buscá-la no **Docker Hub**, repositório do próprio Docker com várias imagens para utilizarmos em nossos projetos.

Baixada a imagem, ela é executada, exibindo a seguinte mensagem:

```
Hello from Docker!
```

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

To generate [this](#) message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "[hello-world](#)" image [from](#) the Docker Hub.
3. The Docker daemon created a [new](#) container [from](#) that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To [try](#) something more ambitious, you can run an Ubuntu container [with](#):

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, [and](#) more [with](#) a free Docker ID:

```
https://cloud.docker.com/
```

For more examples [and](#) ideas, visit:  
<https://docs.docker.com/engine/userguide/>

[COPIAR CÓDIGO](#)

Na mensagem, é detalhado o que foi feito para a execução da imagem. O nosso Docker local entrou em contato com a **Docker Engine**, que por sua vez baixou a imagem **hello-world** do **Docker Hub**, criou um *container* com ela e a executou. Após isso, a saída é impressa para nós e a imagem é encerrada.

Esses passos descritos, imagem, *container*, seus ciclos de vida, tudo isso veremos ao longo dos próximos capítulos.

## Uma alternativa online

### Utilizando o Play With Docker

Não se desespere caso você tenha tido algum problema em instalar o Docker em sua máquina, você também tem a opção de utilizar o [Play With Docker](https://labs.play-with-docker.com/) (<https://labs.play-with-docker.com/>). Nele você poderá utilizar os comandos vistos daqui em diante e testar as diversas funcionalidades que o Docker proporciona. Ao acessar o site basta clicar em +Add New Instance e começar a utilizá-lo como estivesse usando sua máquina normalmente.



## Transcrição

Nesta aula, vamos entender melhor o processo de criação de *containers* e execução de imagens.

Primeiramente, devemos verificar se o Docker está rodando. Se sim, no Linux ou no macOS, vamos executar os comandos do Docker através do próprio **terminal nativo do sistema operacional**. No caso do Windows, o próprio Docker recomenda que os seus comandos sejam executados através do **Windows PowerShell**.

*Lembrando que se o seu Docker foi instalado pelo Docker Toolbox, você deve executar os seus comandos através do Docker Quickstart Terminal, terminal que foi instalado pelo próprio Docker Toolbox.*

## A diferença entre imagens e containers

Na aula anterior, para executar a imagem **hello-world**, executamos o comando `docker run hello-world`. Quando executado esse comando, a primeira coisa que o Docker faz é verificar se temos a imagem **hello-world** no nosso computador, e caso não tenhamos, o Docker buscará e baixará essa imagem no [Docker Hub](https://hub.docker.com/) (<https://hub.docker.com/>)/[Docker Store](https://store.docker.com/) (<https://store.docker.com/>).

A imagem é como se fosse uma receita de bolo, uma série de instruções que o Docker seguirá para criar um *container*, que irá conter as instruções da imagem, do *hello-world*. Criado o *container*, o Docker executa-o. Então, tudo isso é feito quando executamos o `docker run hello-world`.

Há milhares de imagens na **Docker Store** disponíveis para executarmos a nossa aplicação. Por exemplo, temos a imagem do Ubuntu:

```
docker run ubuntu
```

[COPIAR CÓDIGO](#)

Ao executar o comando, o download começará. Podemos ver que não é feito somente um download, pois a imagem é dividida em **camadas**, que veremos mais à frente. Terminados os downloads, nenhuma mensagem é exibida, então significa que o *container* não foi criado? Na verdade o *container* foi criado, o que acontece que é a imagem do Ubuntu não executa nada, por isso nenhuma mensagem foi exibida.

Podemos verificar isso vendo os *containers* que estão sendo executados no momento, executando o seguinte comando:

```
docker ps
```

[COPIAR CÓDIGO](#)

Ao executar esse comando, vemos que não há nenhum *container* ativo, pois quando não há nada para o *container* executar, eles ficam **parados**. Para ver **todos** *containers*, inclusive os parados, adicionamos a *flag* `-a` ao comando acima:

```
docker ps -a
```

[COPIAR CÓDIGO](#)

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NA
4139842e283a	ubuntu	"/bin/bash"	3 minutes ago	Exited (0) 3 minutes ago		el
c1a155091114	hello-world	"/hello"	4 days ago	Exited (0) 4 days ago		ni

[COPIAR CÓDIGO](#)

Com esse comando, conseguimos ver o **id** e o **nome** do *container*, valores que são criados pelo próprio Docker. Além disso, temos a outras informações dos *containers*, a **imagem** em que eles são baseados, o **comando inicial** que roda quando ele é executado, quando ele foi criado e qual o seu status.

Então, o *container* do Ubuntu foi executado, mas ele não fez nada pois não pedimos para o *container* executar algo que funcione dentro do Ubuntu. Então, quando executamos o *container* do Ubuntu, precisamos passar para ele um comando que rode dentro dele, por exemplo:

```
docker run ubuntu echo "Ola Mundo"
```

[COPIAR CÓDIGO](#)

Com isso, o Docker irá executar um *container* com Ubuntu, executar o comando `echo "Ola Mundo"` dentro dele e nos retornar a saída:

```
alura@alura-estudio-03:~$ docker run ubuntu echo "Ola Mundo"  
Ola Mundo
```

[COPIAR CÓDIGO](#)

Só que sabemos que o Ubuntu é um sistema operacional completo, então não queremos ficar somente executando um comando por comando dentro dele, sempre criando um novo *container*. Então, como fazemos para criar um *container* e **interagir** com ele mais do que com um único comando?

## Trabalhando dentro de um container

Podemos fazer com que o terminal da nossa máquina seja integrado ao terminal de dentro do *container*, para ficar um terminal interativo. Podemos fazer isso adicionando a *flag* `-it` ao comando, atrelando assim o terminal que estamos utilizando ao terminal do *container*:

```
docker run -it ubuntu
```

[COPIAR CÓDIGO](#)

Assim que executamos o comando, já podemos perceber que o terminal muda:

```
alura@alura-estudio-03:~$ docker run -it ubuntu  
root@05025384675e:/#
```

[COPIAR CÓDIGO](#)

Com isso, estamos trabalhando **dentro do container**. E dentro dele, podemos trabalhar como se estivéssemos trabalhando dentro do terminal de um Ubuntu, executando comandos como `ls` , `cat` , etc.

Agora, se abrirmos outro terminal e executar o comando `docker ps` , veremos o *container* que estamos executando. Podemos parar a sua execução, digitando no *container* o comando `exit` ou através do atalho `CTRL + D` .

## Executando novamente um container

Paramos a execução do *container*, tanto que o comando `docker ps` não nos retorna mais nada. E se listarmos todos os *containers*, através do comando `docker ps -a` , vemos que ele está lá, parado. Mas agora, para não criar novamente um novo *container*, queremos executá-lo novamente.

Fazemos isso pegando **id** do *container* a ser iniciado, e passando-o ao comando `docker start`

```
docker start 05025384675e
```

[COPIAR CÓDIGO](#)

Esse comando roda um *container* já criado, mas não atrela o nosso terminal ao terminal dele. Para atrelar os terminais, primeiramente devemos parar o *container*, com o comando `docker stop` mais o seu id:

```
docker stop 05025384675e
```

[COPIAR CÓDIGO](#)

E rodamos novamente o *container*, mas passando duas *flags*: `-a`, de *attach*, para integrar os terminais, e `-i`, de *interactive*, para interagirmos com o terminal, para podermos escrever nele:

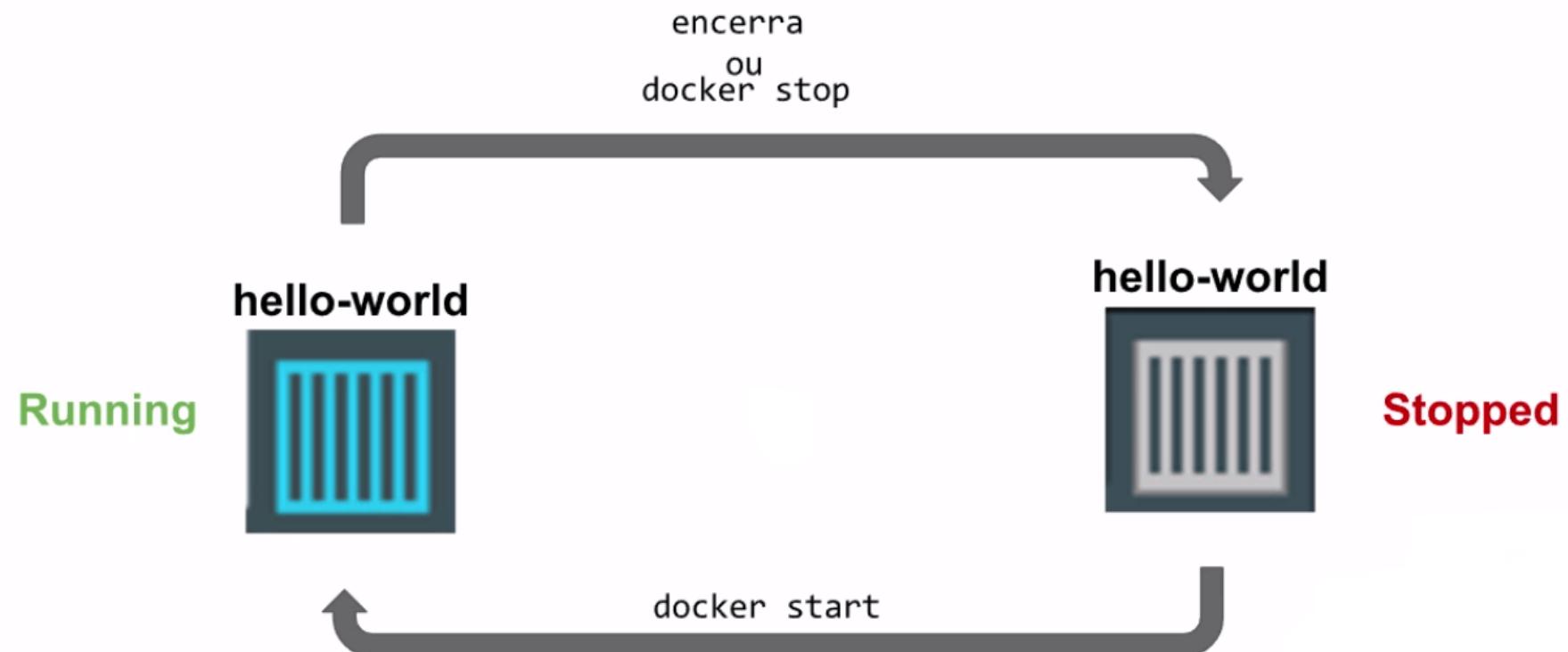
```
alura@alura-estudio-03:~$ docker start -a -i 05025384675e
root@05025384675e:/#
```

[COPIAR CÓDIGO](#)

Com isso, conseguimos ver um pouco de como subir um *container*, pará-lo e executá-lo novamente, além de trabalhar dentro dele.

## Transcrição

Vimos no vídeo anterior os dois principais estados de um *container*, quando criamos um ou iniciamos, ele fica no estado de *running*, e quanto a sua execução encerra ou paramos, ele fica no estado de *stopped*:



## Removendo containers

Só que com os testes que fizemos até agora, acabamos criando vários *containers* (lembrando que podemos ver todos os *containers* criados executando o comando `docker ps -a`) e nunca removemos algum deles, já que os comandos acima só mudam os seus estados. Para **remover** um *container*, executamos o comando `docker rm`, passando para ele o **id** do *container* a ser removido, por exemplo:

```
docker rm c9f83bfb82a8
```

[COPIAR CÓDIGO](#)

Mas para limpar todos os *containers* inativos, devemos remover um por um? Não, pois há um novo comando do Docker, o `prune`, que serve para limparmos algo específico do Docker. Como queremos remover os *containers* parados, executamos o seguinte comando:

```
docker container prune
```

[COPIAR CÓDIGO](#)

O comando é tão poderoso que ele pede para confirmarmos se é isso mesmo que queremos fazer.

## Listando e removendo imagens

E do mesmo jeito que temos o comando `docker container` para mexermos com o *container*, temos o comando `docker images`, que nos exibe as imagens que temos na nossa máquina. Para remover uma imagem, utilizarmos o comando `docker rmi`, passando para ele o nome da imagem a ser removida, por exemplo:

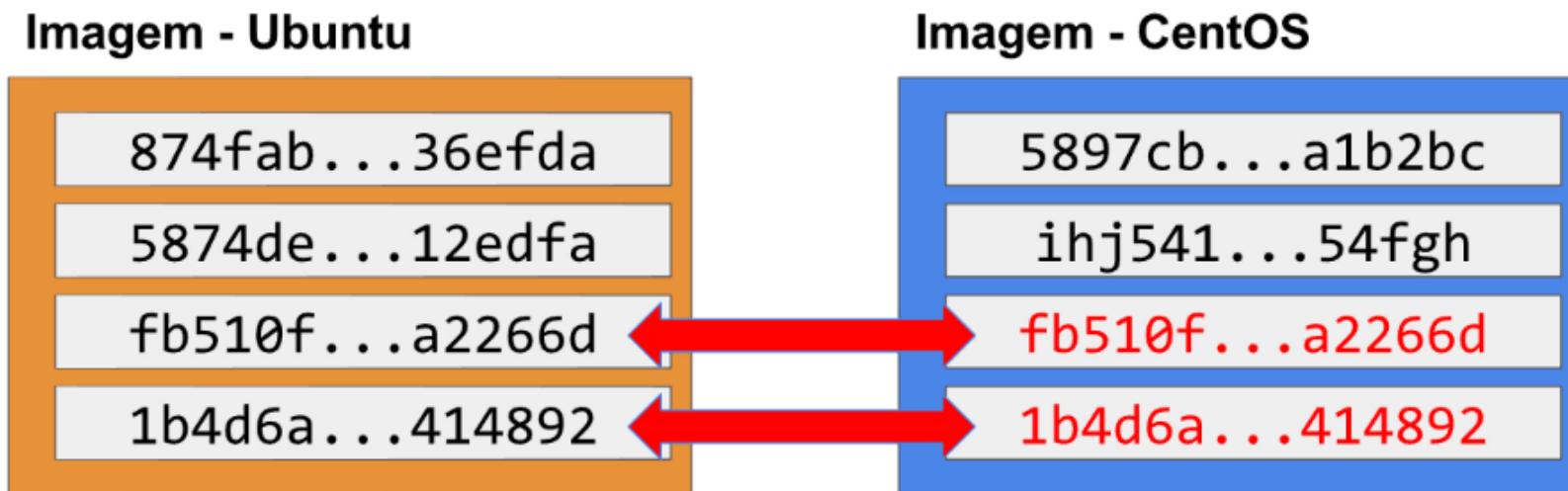
```
docker rmi hello-world
```

[COPIAR CÓDIGO](#)

## Camadas de uma imagem

Na aula anterior, quando baixamos a imagem do Ubuntu, reparamos que ela possui **camadas**, mas como elas funcionam? Toda imagem que baixamos é composta de uma ou mais camadas, e esse sistema tem o nome de *Layered File System*.

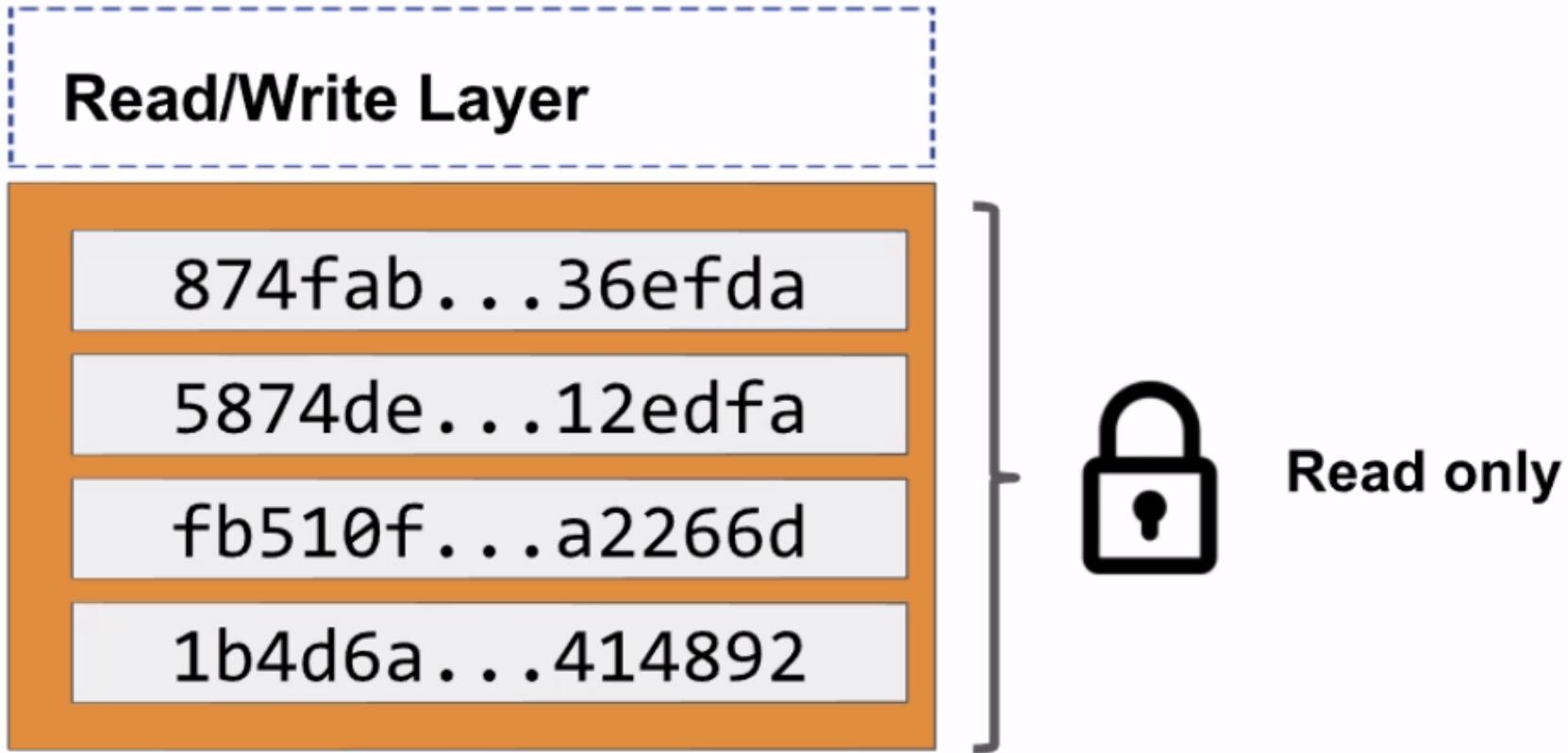
Essas camadas podem ser **reaproveitadas** em outras imagens. Por exemplo, já temos a imagem do Ubuntu, isso inclui as suas camadas, e agora queremos baixar a imagem do CentOS. Se o CentOS compartilha alguma camada que já tem na imagem do Ubuntu, o Docker é inteligente e só baixará as camadas diferentes, e não baixará novamente as camadas que já temos no nosso computador:



No caso da imagem acima, o Docker só baixará as duas primeiras camadas da imagem do CentOS, já que as duas últimas são as mesmas da imagem do Ubuntu, que já temos na nossa máquina. Assim pouparamos tempo, já que precisamos de menos tempo para baixar uma imagem.

Uma outra vantagem é que as camadas de uma imagem são **somente para leitura**. Mas como então conseguimos criar arquivos na aula anterior? O que acontece é que não escrevemos na imagem, já que quando criamos um *container*, ele cria uma nova camada acima da imagem, e nessa camada podemos ler e escrever:

# Container



Então, quando criamos um *container*, ele é criado em cima de uma imagem já existente e nele nós conseguimos escrever. E com uma imagem base, podemos reutilizá-la para diversos *containers*:

## Container

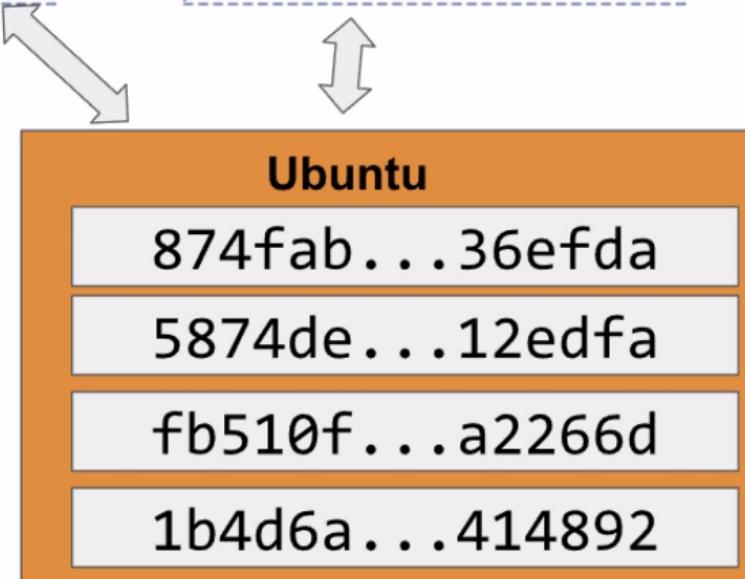
## Container

## Container

Read/Write Layer

Read/Write Layer

Read/Write Layer



Isso nos traz economia de espaço, já que não precisamos ter uma imagem por *container*.



## Transcrição

Agora que já conhecemos mais sobre *containers*, imagens e a diferença entre eles, já podemos fazer um *container* mais interessante, um pouco mais complexo. Então, vamos criar um *container* que segurará um site estático, para entendermos também como funciona a parte de redes do Docker. Para tal, vamos baixar a imagem `dockersamples/static-site`:

```
docker run dockersamples/static-site
```

[COPIAR CÓDIGO](#)

Nas imagens que vimos anteriormente, as imagens *oficiais*, não precisamos colocar um *username* na hora de baixá-las. Esse *username* representa o usuário que toma conta da imagem, quem a criou. Como a imagem que vamos utilizar foi criada por outro(s) usuário(s), precisamos especificar o seu *username* para baixá-la.

Terminado o download da imagem, o *container* é executado, pois sabemos que os *containers* ficam no estado de *running* quando são criados. No caso dessa imagem, o *container* está executando um processo de um servidor web, que está disponibilizando o site estático para nós, então esse processo trava o terminal. Mas como evitamos que esse travamento aconteça?

Para tal, paramos o *container* que acabamos de criar e para impedir o travamento, nós executamos-o sem atrelar o nosso terminal ao terminal do *container*, fazendo isso através da flag `-d` :

```
docker run -d dockersamples/static-site
```

[COPIAR CÓDIGO](#)

Assim, o *container* fica executando em segundo plano. Podemos verificar que o *container* realmente está rodando executando o comando `docker ps`:

```
alura@alura-estudio-03:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND
a6f2fab332db      dockersamples/static-site   "/bin/sh -c 'cd /u..."
```

[COPIAR CÓDIGO](#)

Mas como fazemos para acessar o site estático?

## Acessando o site

Em nenhum momento dizemos onde está o site estático. Qual porta que utilizamos para acessá-lo? A **80**, conforme está na saída do `docker ps`? Essa é a porta interna que o *container* está utilizando. Então, o que precisamos fazer é *linkar* essa porta interna do *container* a uma porta do nosso computador. Para fazer isso, precisamos adicionar mais uma *flag*, a `-P`, que fará com que o Docker atribua uma porta aleatória do mundo externo, que no caso é a nossa máquina, para poder se comunicar com o que está dentro do *container*.

```
docker run -d -P dockersamples/static-site
```

[COPIAR CÓDIGO](#)

Agora, ao executar novamente o comando `docker ps`, na coluna **PORTS**, vemos algo como:

## POROS

0.0.0.0:9001->80/tcp, 0.0.0.0:9000->443/tcp

[COPIAR CÓDIGO](#)

No caso do mapeamento acima, vemos que a porta **9001** da nossa máquina faz referência à porta **80** do *container*, e a porta **9000** da nossa máquina faz referência à porta **443** do *container*. Uma outra maneira de ver as portas é utilizar o comando `docker port`, passando para ele o **id** do *container*:

```
alura@alura-estudio-03:~$ sudo docker port 989e4d7d3638  
443/tcp -> 0.0.0.0:9000  
80/tcp -> 0.0.0.0:9001
```

[COPIAR CÓDIGO](#)

Então, se quisermos acessar a porta **80**, que é onde está o site estático, na nossa máquina, como o endereço **0.0.0.0** representa a nossa máquina local, podemos acessar o endereço <http://localhost:9001/> no navegador.

*Caso você esteja utilizando o Docker Toolbox, como ele está rodando em cima de uma máquina virtual, o endereço http://localhost:9001/ não funcionará, pois você deve acessar a porta através do IP da máquina virtual. Para descobrir o IP dessa máquina virtual, basta executar o comando docker-machine ip . Com o IP em mãos, basta acessá-lo no navegador, utilizando a porta que o Docker atribuiu, por exemplo http://192.168.0.38:9001/ .*

## Nomeando um container

Uma outra coisa interessante que é possível fazer quando estamos criando um *container* é que podemos dar um **nome** para o *container*, assim não ficamos dependendo os ids aleatórios que o Docker atribui, tornando mais fácil na hora de parar e remover o *container*, por exemplo. Para dar um nome para o *container*, utilizamos a *flag* `--name` :

```
docker run -d -P --name meu-site dockersamples/static-site
```

[COPIAR CÓDIGO](#)

Assim o nome do nosso *container* será **meu-site**. Agora, para pará-lo, basta passar o seu nome para o comando `docker stop`:

```
docker stop meu-site
```

[COPIAR CÓDIGO](#)

A mesma coisa seria para rodar o *container* novamente, ou para removê-lo, bastando apenas nós utilizarmos o seu nome.

## Definindo uma porta específica

Uma outra coisa interessante para vermos é que, quando estamos criando um *container* e queremos *linkar* uma porta interna sua a uma porta do nosso computador, utilizamos a *flag* `-P`, para o Docker atribuir uma porta aleatória da nossa máquina, assim podemos nos comunicar com o que está dentro do *container*. Mas podemos definir essa porta, utilizando a *flag* `-p`, nesse modelo: `-p PORTA-MUNDO-EXTERNO:PORTA-CONTAINER`, por exemplo:

```
docker run -d -p 12345:80 dockersamples/static-site
```

[COPIAR CÓDIGO](#)

Nesse exemplo, através da porta **12345** do nosso computador podemos acessar a porta **80** do *container*.

## Atribuindo uma variável de ambiente

Além disso, podemos atribuir uma variável de ambiente no *container*. Por exemplo, a página do site estático pega o valor da variável de ambiente `AUTHOR` e o exibe junto à mensagem de *Hello*, então podemos modificar o valor dessa variável, através da flag `-e`:

```
docker run -d -P -e AUTHOR="Douglas Q" dockersamples/static-site
```

[COPIAR CÓDIGO](#)

Quando abrirmos o site, a mensagem que será exibida é *Hello Douglas Q*.

## Parando todos os containers de uma só vez

Por último, podemos ver apenas os ids dos *containers* que estão rodando, executando o comando `docker ps -q`. E com esse comando, podemos parar todos os *containers* de uma só vez. Para isso, podemos utilizar a interpolação de comandos, no padrão `$(comando)`, que executa o comando, capture sua saída e insere isso na linha de comando:

```
docker stop $(docker ps -q)
```

[COPIAR CÓDIGO](#)

Então, o comando `docker ps -q` será executado e a sua saída, os ids dos *containers* que estão rodando, será inserida no comando `docker stop`, parando assim todos os *containers*.

Além disso, o comando `docker stop` demora um pouco para ser executado pois ele espera 10 segundos para parar o *container*. Podemos diminuir esse tempo através da flag `-t`, passando o tempo a ser aguardado, por exemplo:

```
docker stop -t 0 $(docker ps -q)
```

[COPIAR CÓDIGO](#)

## O que aprendemos?



41%

ATIVIDADES  
9 de 9FÓRUM DO  
CURSOVOLTAR  
PARA  
DASHBOARDMODO  
NOTURNOABRIR  
CADERNO

90.6k xp



Aprendemos neste capítulo:

- Comandos básicos do Docker para podermos baixar imagens e interagir com o container.
- Imagens do Docker possuem um sistema de arquivos em camadas (Layered File System) e os benefícios dessa abordagem principalmente para o download de novas imagens.
- Imagens são *Read-Only* sempre (apenas para leitura)
- Containers representam uma instância de uma imagem
- Como imagens são *Read-Only* os containers criam nova camada (layer) para guardar as alterações
- O comando Docker `run` e as possibilidades de execução de um container

Segue também uma breve lista dos comandos utilizados:

- `docker ps` - exibe todos os containers em execução no momento.
- `docker ps -a` - exibe todos os containers, independentemente de estarem em execução ou não.
- `docker run -it NOME_DA_IMAGEM` - conecta o terminal que estamos utilizando com o do container.
- `docker start ID_CONTAINER` - inicia o container com id em questão.
- `docker stop ID_CONTAINER` - interrompe o container com id em questão.

- docker start -a -i ID\_CONTAINER - inicia o container com id em questão e integra os terminais, além de permitir interação entre ambos.
- docker rm ID\_CONTAINER - remove o container com id em questão.
- docker container prune - remove todos os containers que estão parados.
- docker rmi NOME\_DA\_IMAGEM - remove a imagem passada como parâmetro.
- docker run -d -P --name NOME dockersamples/static-site - ao executar, dá um nome ao container.
- docker run -d -p 12345:80 dockersamples/static-site - define uma porta específica para ser atribuída à porta 80 do container, neste caso 12345.
- docker run -d -P -e AUTHOR="Fulano" dockersamples/static-site - define uma variável de ambiente AUTHOR com o valor *Fulano* no container criado.



41%

ATIVIDADES  
9 de 9

FÓRUM DO  
CURSO

VOLTAR  
PARA  
DASHBOARD

MODO  
NOTURNO

ABRIR  
CADERNO



90.6k xp

a



## Transcrição

Nesta aula, começaremos a falar sobre os **volumes**, mas antes vamos relembrar o que vimos na aula anterior.

## Recapitulando...

Na aula anterior, vimos que os *containers* nada mais são do que uma pequena camada de leitura e escrita, que funcionam em cima das imagens, que não podem ser modificadas, pois são somente para leitura.

Quando removemos um *container* (comando `docker rm`), a camada de leitura e escrita também é removida, o que faz com que os nossos dados também sejam removidos, o que é muito ruim, já que esses dados podem ser importantes, como por exemplo um banco de dados, então toda vez que o *container* for removido, tudo o que escrevemos nele será jogado fora? Não é isso que queremos, então temos que ver um jeito de **persistir esses dados**, mas também trabalhando com *containers*.

É da natureza dos *containers* a volatilidade, isto é, eles são criados e removidos rapidamente e facilmente, mas devemos ter um lugar para salvar os dados, e esse lugar são os **volumes**.

## O que são os volumes?

Quando escrevemos em um *container*, assim que ele for removido, os dados também serão. Mas podemos criar um local especial dentro dele, e especificamos que esse local será o nosso **volume de dados**.

Quando criamos um volume de dados, o que estamos fazendo é apontá-lo para uma pequena pasta no **Docker Host**. Então, quando criamos um volume, criamos uma pasta dentro do *container*, e o que escrevermos dentro dessa pasta na verdade estaremos escrevendo do Docker Host.

Isso faz com que não perdemos os nossos dados, pois o *container* até pode ser removido, mas a pasta no **Docker Host** ficará intacta.

## Trabalhando com volumes

Sabendo disso, vamos ver como trabalhar com o **Docker Host**. No Terminal ou PowerShell (ou Docker Quickstart Terminal), criamos um *container* com o `docker run`, mas dessa vez utilizando a *flag* `-v` para criar um volume, seguido do nome do mesmo:

```
docker run -v "/var/www" ubuntu
```

[COPIAR CÓDIGO](#)

No exemplo acima, criamos o volume `/var/www`, mas a que pasta no **Docker Host** ele faz referência? Para descobrir, podemos inspecionar o *container*, executando o comando `docker inspect`, passando o seu **id** para o mesmo:

```
docker inspect 8cf7b40ce226
```

[COPIAR CÓDIGO](#)

Temos uma saída com diversas informações, mas a que nos interessa é o **"Mounts"**:

```
"Mounts": [  
{  
  "Type": "volume",
```

```
        "Name": "5e1cbfd48d07284680552e56087c9d5196659600ccd6874bfa3831b51ddd0576",
        "Source": "/var/lib/docker/volumes/5e1cbfd48d07284680552e56087c9d5196659600ccd6874bfa3831b5
        "Destination": "/var/www",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
    }
]
```

[COPIAR CÓDIGO](#)

Nele, podemos ver que o `/var/www` será escrito na nossa máquina no diretório `/var/lib/docker/volumes/5e1cbfd48d07284680552e56087c9d5196659600ccd6874bfa3831b51ddd0576/_data`, endereço que foi gerado automaticamente pelo Docker. Ou seja, tudo que escrevermos na pasta `/var/www` do *container*, na verdade estaremos escrevendo na pasta `/var/lib/docker/volumes/5e1cbfd48d07284680552e56087c9d5196659600ccd6874bfa3831b51ddd0576/_data` da nossa máquina.

E ao remover o *container*, a pasta continuará na nossa máquina. Essa pasta gerada pelo Docker pode ser configurada, podemos dizer a pasta que será referenciada pela pasta `/var/www` do *container*. Por exemplo, se quisermos escrever dentro do Desktop da nossa máquina, devemos passá-lo antes do volume, separando-os com dois pontos. Além disso, vamos executar o *container* no modo interativo:

```
docker run -it -v "C:\Users\Alura\Desktop:/var/www" ubuntu
root@abd0286c0083:/#
```

[COPIAR CÓDIGO](#)

Ou seja, quando escrevermos na pasta `/var/www` do *container*, estaremos escrevendo no Desktop da nossa máquina. Para provar isso, na pasta `/var/www`, vamos criar um arquivo e escrever nele uma mensagem:

```
root@abd0286c0083:/# cd /var/www/  
root@abd0286c0083:/var/www# touch novo-arquivo.txt  
root@abd0286c0083:/var/www# echo "Este arquivo foi criado dentro de um volume" > novo-arquivo.txt
```

[COPIAR CÓDIGO](#)

Ao acessarmos o nosso Desktop, o arquivo estará lá, também com a mensagem escrita. E ao remover o *container*, a sua camada de escrita é removida, mas os arquivos continuam no nosso Desktop.

Então, o uso de volumes é importante para salvarmos os nossos dados fora do *container*, e esses volumes sempre estarão atrelados ao **Docker Host**. No caso acima, atrelamos o volume com o Desktop, mas podemos atrelar com um lugar mais seguro, salvando os dados do banco de dados nele, logs, e até mesmo o código fonte, coisa que faremos no próximo vídeo.



## Transcrição

Nessa aula vamos usar um exemplo escrito Node.js. O código desse projeto pode ser baixado [aqui](https://s3.amazonaws.com/caelum-online-public/646-docker/03/projetos/volume-exemplo.zip) (<https://s3.amazonaws.com/caelum-online-public/646-docker/03/projetos/volume-exemplo.zip>).

Já vimos que o que escrevemos no volume (pasta `/var/www` do *container*) aparece na pasta configurada da nossa máquina local, que no vídeo anterior foi o Desktop. Mas podemos pensar o contrário, ou seja, tudo o que escrevemos no Desktop será acessível na pasta `/var/www` do *container*.

Isso nos dá a possibilidade de implementar localmente um código de uma linguagem que não está instalada na nossa máquina, e colocá-lo para compilar e rodar dentro do *container*. Se o *container* possui Node, Java, PHP, seja qual for a linguagem, não precisamos tê-los instalados na nossa máquina, **nosso ambiente de desenvolvimento pode ser dentro do *container*.**

É isso que faremos, pegaremos um código nosso, que está na nossa máquina, e colocaremos para rodar dentro do *container*, utilizando essa técnica com volumes.

## Rodando código em um container

Para isso, vamos usar um exemplo escrito Node.js, que pode ser baixado [aqui](https://s3.amazonaws.com/caelum-online-public/646-docker/03/projetos/volume-exemplo.zip) (<https://s3.amazonaws.com/caelum-online-public/646-docker/03/projetos/volume-exemplo.zip>). Até podemos executar esse código na nossa máquina, mas temos que instalar o Node na versão certa em que o desenvolvedor implementou o código.

Agora, como fazemos para criar um *container*, que irá pegar e rodar esse código Node que está na nossa máquina? Vamos utilizar os volumes. Então, vamos começar a montar o comando.

Primeiramente, como vamos rodar um código em Node.js, precisamos utilizar a sua imagem:

```
docker run node
```

[COPIAR CÓDIGO](#)

Além disso, precisamos criar um volume, que faça referência à pasta do código no nosso Desktop:

```
docker run -v "C:\Users\Alura\Desktop\volume-exemplo:/var/www" node
```

[COPIAR CÓDIGO](#)

Agora, para iniciar o seu servidor, executamos o comando `npm start`. Para executar um comando dentro do *container*, podemos iniciá-lo no modo interativo ou passá-lo no final do `docker run`:

```
docker run -v "C:\Users\Alura\Desktop\volume-exemplo:/var/www" node npm start
```

[COPIAR CÓDIGO](#)

Por fim, esse servidor roda na porta **3000**, então precisamos *linkar* essa porta a uma porta do nosso computador, no caso a **8080**. O comando ficará assim:

```
docker run -p 8080:3000 -v "C:\Users\Alura\Desktop\volume-exemplo:/var/www" node npm start
```

[COPIAR CÓDIGO](#)

Executado o comando, recebemos um erro. Nele podemos verificar a seguinte linha:

```
npm ERR! enoent ENOENT: no such file or directory, open '/package.json'
```

[COPIAR CÓDIGO](#)

Isto é, o `package.json` não foi encontrado, mas ele está dentro da pasta do código. O que acontece é que o *container* não inicia já dentro da pasta `/var/www`, e sim em uma pasta determinada pelo próprio *container*. Por exemplo, se a imagem é baseada no Ubuntu, o *container* iniciar no `root`.

Então devemos especificar que o comando `npm start` deve ser executado dentro da pasta `/var/www`. Para isso, vamos passar a flag `-w` (*Working Directory*), para dizer em qual diretório o comando deve ser executado, a pasta `/var/www`:

```
docker run -p 8080:3000 -v "C:\Users\Alura\Desktop\volume-exemplo:/var/www" -w "/var/www" node npm
```

[COPIAR CÓDIGO](#)

Agora, ao acessar a porta `8080` no navegador, vemos uma página exibindo a mensagem **Eu amo Docker!**. E para testar que está mesmo funcionando, podemos editar o arquivo `index.html` localmente, salvá-lo e ao recarregar a página no navegador, a nova mensagem é exibida! Ou seja, podemos criar um ambiente de desenvolvimento todo baseado em *containers*, o que ainda facilita o trabalho da nossa equipe, já que se todos utilizarem o *container*, todos terão o mesmo ambiente de desenvolvimento.

## Melhorando o comando

Por fim, sabemos que podemos executar o Docker de qualquer local da nossa máquina, então podemos executar o comando que fizemos dentro da pasta do nosso projeto. Fazendo isso, podemos melhorar esse comando com o auxílio da interpolação

de comandos, já que o comando `pwd` retorna o nosso diretório atual:

```
alura@alura-estudio-03:~$ cd Desktop/volume-exemplo/  
alura@alura-estudio-03:~/Desktop/volume-exemplo$ pwd  
/home/alura/Desktop/volume-exemplo
```

[COPIAR CÓDIGO](#)

Assim, ao invés de passar o diretório físico para dentro do comando `docker run`, podemos utilizar a interpolação de comandos, e interpolar o comando `pwd`, assim a sua saída será capturada e inserida dentro do `docker run`:

```
alura@alura-estudio-03:~$ cd Desktop/volume-exemplo/  
alura@alura-estudio-03:~/Desktop/volume-exemplo$ pwd  
/home/alura/Desktop/volume-exemplo  
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker run -p 8080:3000 -v "$(pwd):/var/www" -w "/
```

[COPIAR CÓDIGO](#)

Assim, vimos como rodar um código local, que está na nossa máquina, dentro de um *container*, utilizando a tecnologia dos volumes, *linkando* a nossa pasta local com uma pasta do *container*, criando assim um ambiente de desenvolvimento todo baseado em *containers*.



## Transcrição

Já trabalhamos com a imagem do **ubuntu**, **hello-world**, **dockersamples/static-site** e por fim do **node**, mas até agora não criamos a nossa própria imagem, para podermos distribuir para as outras pessoas. Então é isso que faremos nesta aula.

No começo do treinamento, foi comentado que a imagem é como se fosse uma *receita de bolo*. Então, para criarmos a nossa própria imagem, temos que criar a nossa *receita de bolo*, o **Dockerfile**, ensinando o Docker a criar uma imagem a partir da nossa aplicação, para que ela seja utilizada em outros locais.

## Montando o Dockerfile

Então, no nosso projeto, devemos criar o arquivo **Dockerfile**, que nada mais é do que um arquivo de texto. Ele pode ter qualquer nome, porém nesse caso ele também deve possuir a extensão **.dockerfile**, por exemplo **node.dockerfile**, mas vamos manter o nome padrão mesmo.

Geralmente, montamos as nossas imagens a partir de uma imagem já existente. Nós podemos criar uma imagem do zero, mas a prática de utilizar uma imagem como base e adicionar nela o que quisermos é mais comum. Para dizer a imagem-base que queremos, utilizamos a palavra **FROM** mais o nome da imagem.

Como o nosso projeto precisa do Node.js, vamos utilizar a sua imagem:

```
FROM node
```

[COPIAR CÓDIGO](#)

Além disso, podemos indicar a versão da imagem que queremos, ou utilizar o `latest`, que faz referência à versão mais recente da imagem. Se não passarmos versão nenhuma, o Docker irá assumir que queremos o `latest`, mas vamos deixar isso explícito:

```
FROM node:latest
```

[COPIAR CÓDIGO](#)

Outra instrução que é comum colocarmos é quem cuida, quem criou a imagem, através do `MAINTAINER`:

```
FROM node:latest  
MAINTAINER Douglas Quintanilha
```

[COPIAR CÓDIGO](#)

Agora, especificamos o que queremos na imagem. No caso, queremos colocar o nosso código dentro da imagem, então utilizarmos o `COPY`. Como queremos copiar tudo o que está dentro da pasta, vamos utilizar o `.` para copiar tudo que está na pasta do arquivo `Dockerfile`, e vamos copiar para `/var/www`, do exemplo da aula anterior:

```
FROM node:latest  
MAINTAINER Douglas Quintanilha  
COPY . /var/www
```

[COPIAR CÓDIGO](#)

No projeto, já temos as suas dependências dentro da pasta `node_modules`, mas não queremos copiar essa pasta para dentro do *container*, pois elas podem estar desatualizadas, quebradas, então queremos que a própria imagem instale as

dependências para nós, rodando o comando `npm install`. Para executar um comando, utilizamos o `RUN`:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
RUN npm install
```

[COPIAR CÓDIGO](#)

Agora, **deletamos a pasta `node_modules`**, para ela não ser copiada para o *container*. Além disso, toda imagem possui um comando que é executado quando a mesma inicia, e o comando que utilizamos na aula anterior foi o `npm start`. Para isso, utilizamos o `ENTRYPOINT`, que executará o comando que quisermos assim que o *container* for carregado:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
RUN npm install
ENTRYPOINT npm start
```

[COPIAR CÓDIGO](#)

Também podemos passar o comando como se fosse em um array, por exemplo `["npm", "start"]`, ambos funcionam.

Falta colocarmos a porta em que a aplicação executará, a porta em que ela ficará exposta. Para isso, utilizamos o `EXPOSE`:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
RUN npm install
```

```
ENTRYPOINT ["npm", "start"]
EXPOSE 3000
```

[COPIAR CÓDIGO](#)

Por fim, falta dizermos onde os comandos rodarão, pois eles devem ser executados dentro da pasta `/var/www`. Então, através do `WORKDIR`, assim que copiarmos o projeto, dizemos em qual diretório iremos trabalhar;

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
WORKDIR /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE 3000
```

[COPIAR CÓDIGO](#)

Com isso, finalizamos o **Dockerfile**, baseado no comando que fizemos na aula anterior:

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker run -p 8080:3000 -v "$(pwd):/var/www" -w "/
```

[COPIAR CÓDIGO](#)

Resta agora criar a imagem.

## Criando a imagem

Para criar a imagem, precisamos fazer o seu *build* através do comando `docker build`, comando utilizado para *buildar* uma imagem a partir de um **Dockerfile**. Para configurar esse comando, passamos o nome do **Dockerfile** através da *flag* `-f`:

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker build -f Dockerfile
```

[COPIAR CÓDIGO](#)

Como o nome do nosso **Dockerfile** é o padrão, poderíamos omitir esse parâmetro, mas se o nome for diferente, por exemplo `node.dockerfile`, é preciso especificar, mas vamos deixar especificado para detalharmos melhor o comando.

Além disso, passamos a *tag* da imagem, o seu nome, através da *flag* `-t`. Já vimos que para imagens não-oficiais, colocamos o nome no padrão **NOME\_DO\_USUÁRIO/NOME\_DA\_IMAGEM**, então é isso que faremos, por exemplo:

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker build -f Dockerfile -t douglasq/node
```

[COPIAR CÓDIGO](#)

E agora dizemos onde está o **Dockerfile**. Como já estamos rodando o comando dentro da pasta **volume-exemplo**, vamos utilizar o ponto (`.`);

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker build -f Dockerfile -t douglasq/node .
```

[COPIAR CÓDIGO](#)

Ao executar o comando, podemos perceber que cada instrução executada do nosso **Dockerfile** possui um **id**. Isso por que para cada passo o Docker cria um *container* intermediário, para se aproveitar do seu sistema de camadas. Ou seja, cada instrução gera uma nova camada, que fará parte da imagem final, que nada mais é do que a imagem-base com vários *containers* intermediários em cima, sendo que cada um desses *containers* representa um comando do **Dockerfile**.

Assim, se um dia a imagem precisar ser alterada, somente o *container* referente à instrução modificada será alterado, com as outras partes intermediárias da imagem já prontas.

## Criando um container a partir da nossa imagem

Agora que já temos a imagem criada, podemos criar um *container* a partir dela:

```
docker run -d -p 8080:3000 douglasq/node
```

[COPIAR CÓDIGO](#)

Ao acessar o endereço da porta no navegador, vemos a página da nossa aplicação. No **Dockerfile**, também podemos criar variáveis de ambiente, utilizando o `ENV`. Por exemplo, para criar a variável `PORT`, para dizer em que porta a nossa aplicação irá rodar, fazemos:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
ENV PORT=3000
COPY . /var/www
WORKDIR /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE 3000
```

[COPIAR CÓDIGO](#)

E no próprio **Dockerfile**, podemos utilizar essa variável:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
ENV PORT=3000
COPY . /var/www
WORKDIR /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE $PORT
```

[COPIAR CÓDIGO](#)

E como modificamos o **Dockerfile**, precisamos construir a nossa imagem novamente e podemos perceber que dessa vez o comando é bem mais rápido, já que quase todas as camadas estão *cacheadas* pelo Docker.

Agora que criamos a imagem, vamos disponibilizá-la para outras pessoas. E é isso que veremos no próximo vídeo.



## Transcrição

Já criamos a imagem, mas por enquanto ela só está na nossa máquina local. Para disponibilizar a imagem para outras pessoas, precisamos enviá-la para o [Docker Hub](https://hub.docker.com/) (<https://hub.docker.com/>).

O primeiro passo é criar a nossa conta. Com ela criada, no terminal nós executamos o comando `docker login` e digitamos o nosso login e senha que acabamos de criar.

Após isso, basta executar o comando `docker push`, passando para ele a imagem que queremos subir, por exemplo:

```
docker push douglasq/node
```

[COPIAR CÓDIGO](#)

Esse comando pode demorar um pouco, mas terminada a sua execução, podemos ver que várias mensagens *Mounted from library/node*, ou seja, o Docker já sabe que essas camadas podem ser reaproveitadas da imagem do `node`, então não tem o porquê dessas camadas subirem também, então só as camadas diferentes são enviadas para o Docker Hub.

Mais uma vantagem em se trabalhar com camadas, o *Layered File System*, pois até na hora de fazer o upload, só é feito das camadas diferentes, as outras são referenciadas da imagem-base que estamos utilizando, no caso a do `node`.

Por fim, ao acessar a nossa conta do Docker Hub, podemos ver que a imagem está lá. Para baixá-la, podemos utilizar o comando `docker pull`:

```
docker pull douglasq/node
```

[COPIAR CÓDIGO](#)

Esse comando somente baixa a imagem, sem criar nenhum *container* acima dela.

Então, esse é um jeito de simples de compartilharmos uma imagem com outras pessoas, através do **Docker Hub**. A imagem é disponibilizada em um repositório público, mas também podemos disponibilizar em repositórios privados, que no momento da criação do curso, cada usuário pode criar um repositório privado gratuitamente.

## O que aprendemos?

Aprendemos neste capítulo:

- A entender o papel do arquivo DockerFile para criar imagens.
  - O Dockerfile define os comandos para executar instalações complexas e com características específicas.
- Vimos os principais comandos como `FROM` , `MAINTAINER` , `COPY` , `WORKDIR` , `RUN` , `EXPOSE` e `ENTRYPOINT`
- A subir uma imagem criada através de um Dockerfile para o Docker Hub e disponibilizar para os desenvolvedores

Lembrando também:

- as imagens são *read-only* sempre
- um container é uma instância de uma imagem
- para guardar as alterações a *docker engine* cria uma nova layer em cima da última layer da imagem

Segue também uma breve lista dos comandos utilizados:

- `docker build -f Dockerfile` - cria uma imagem a partir de um Dockerfile.
- `docker build -f CAMINHO_DOCKERFILE/Dockerfile -t NOME_USUARIO/NOME_IMAGEM` - constrói e nomeia uma imagem não-oficial informando o caminho para o Dockerfile.
- `docker login` - inicia o processo de login no Docker Hub.
- `docker push NOME_USUARIO/NOME_IMAGEM` - envia a imagem criada para o Docker Hub.
- `docker pull NOME_USUARIO/NOME_IMAGEM` - baixa a imagem desejada do Docker Hub.



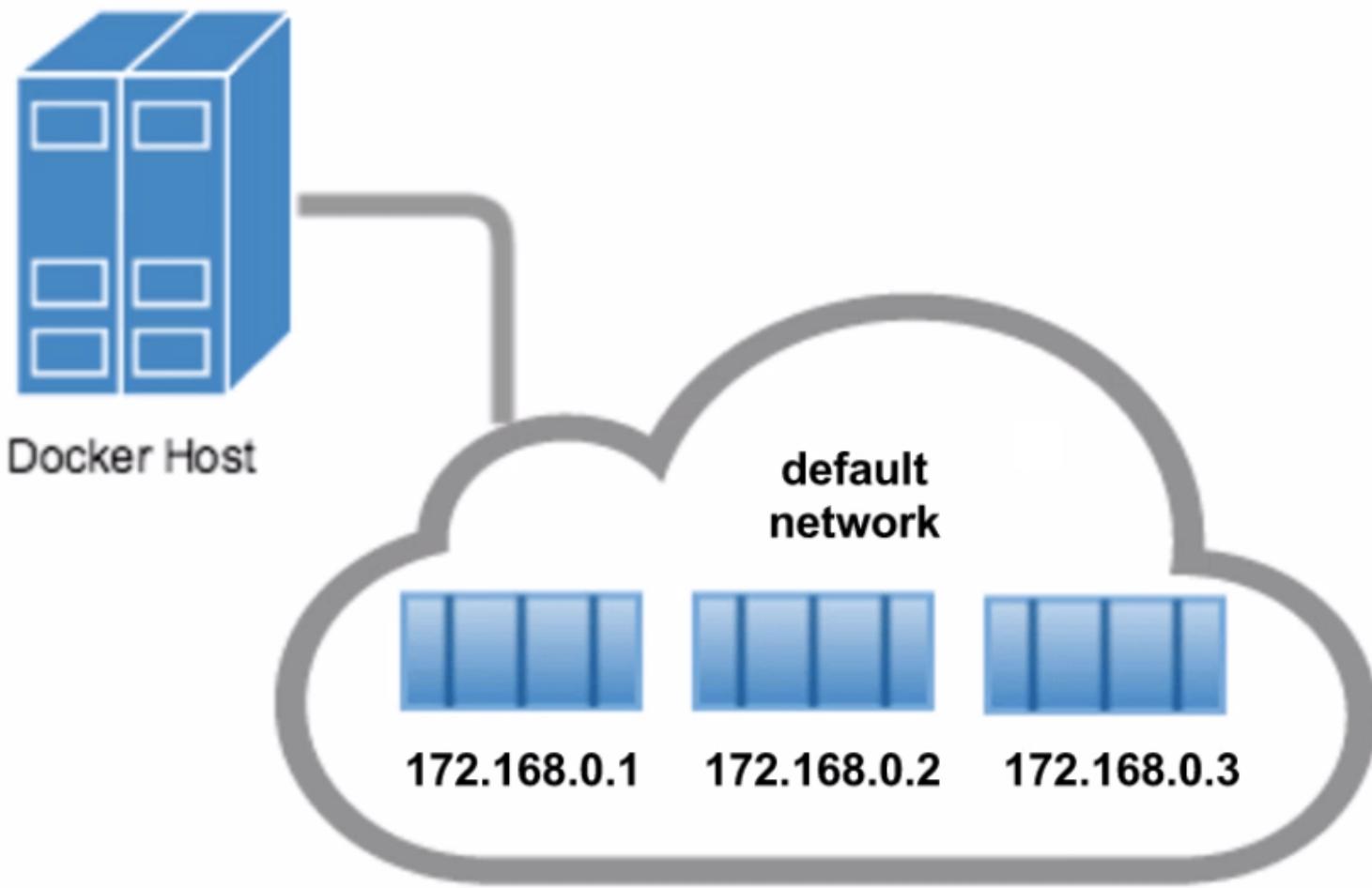
## Transcrição

Neste capítulo, veremos como funciona a rede, e como fazemos para interligar diversos *containers* no Docker. Normalmente uma aplicação é composta por diversas partes, sejam elas o *load balancer/proxy*, a aplicação em si, um banco de dados, etc. Quando estamos trabalhando com *containers*, é bem comum separarmos cada uma dessas partes em um *container* específico, para cada *container* ficar com somente uma única responsabilidade.

Mas se temos uma parte da nossa aplicação em cada *container*, como podemos fazer para essas partes falarem entre elas? Pois para a nossa aplicação funcionar como um todo, os *containers* precisam trocar dados entre eles.

## Redes com Docker

A boa notícia é que no Docker, por padrão, já existe uma ***default network***. Isso significa que, quando criamos os nossos *containers*, por padrão eles funcionam na mesma rede:



Para verificar isso, vamos subir um *container* com Ubuntu:

```
docker run -it ubuntu
```

[COPIAR CÓDIGO](#)

Em outro terminal, vamos verificar o **id** desse *container* através do comando `docker ps`, e com ele em mãos, vamos passá-lo para o comando `docker inspect`. Na saída desse comando, em *NetworkSettings*, vemos que o *container* está na rede padrão *bridge*, rede em que ficam todos os *containers* que criamos.

Voltando ao terminal do *container*, se executarmos o comando `hostname -i` vemos o IP atribuído a ele pela rede local do Docker:

```
root@973feeeeb1df:/# hostname -i  
172.17.0.2
```

[COPIAR CÓDIGO](#)

Então, dentro dessa rede local, os *containers* podem se comunicar através desses IPs. Para comprovar isso, vamos deixar esse *container* rodando e criar um novo:

```
docker run -it ubuntu
```

[COPIAR CÓDIGO](#)

E vamos verificar o seu IP:

```
root@dd316a9f585f:/# hostname -i  
172.17.0.3
```

[COPIAR CÓDIGO](#)

Agora, no primeiro *container*, vamos instalar o pacote `iutils-ping` para podermos executar o comando `ping` para verificar a comunicação entre os *containers*:

```
root@973feeeeb1df:/# apt-get update && apt-get install iutils-ping
```

[COPIAR CÓDIGO](#)

Após o término da instalação, executamos o comando `ping`, passando para ele o IP do segundo *container*. Para interromper o comando, utilizamos o atalho **CTRL + C**:

```
root@973feeeeb1df:/# ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.180 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.133 ms
64 bytes from 172.17.0.3: icmp_seq=3 ttl=64 time=0.148 ms
^C
--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.133/0.153/0.180/0.024 ms
```

[COPIAR CÓDIGO](#)

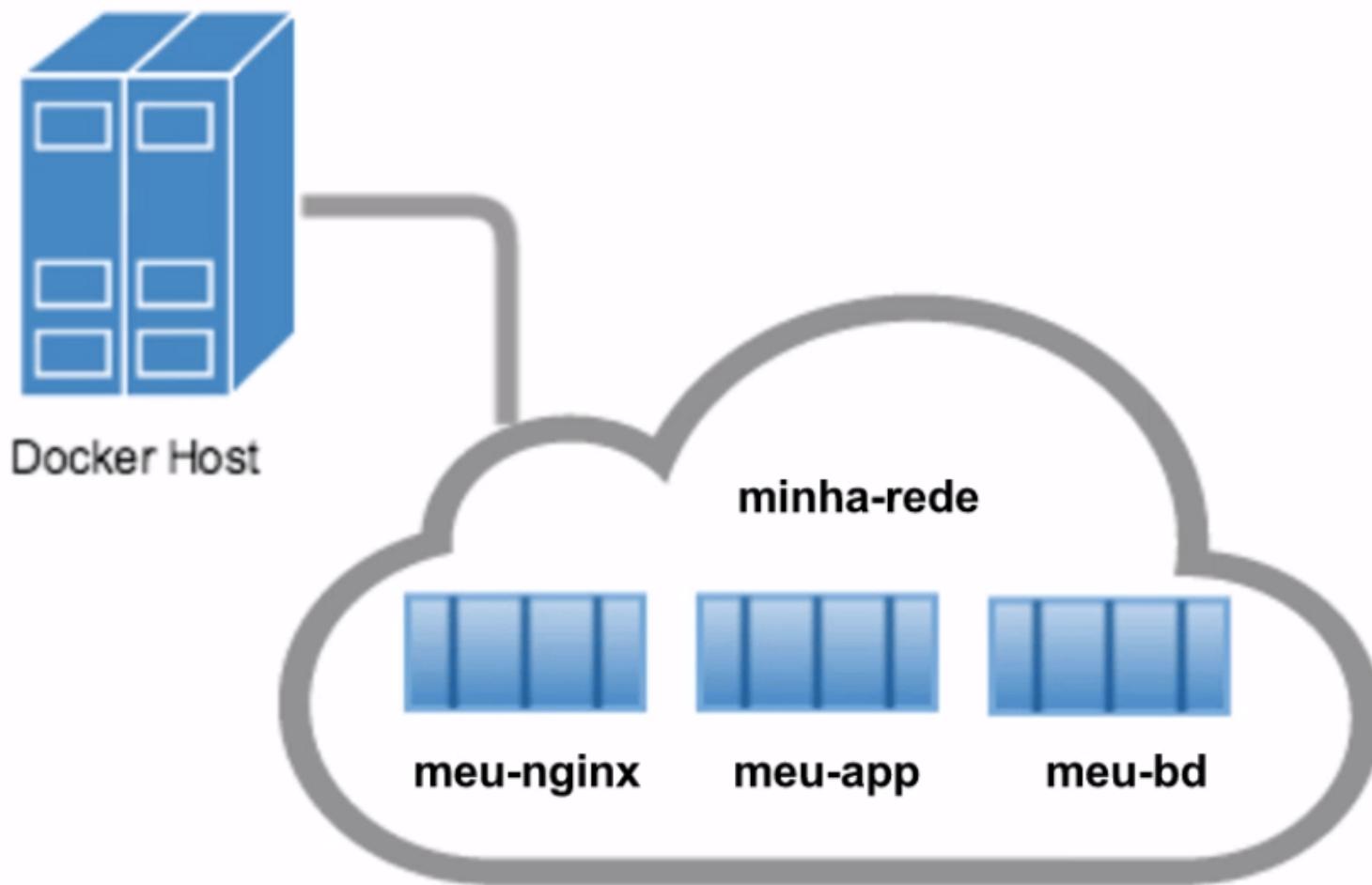
Assim, podemos ver que os *containers* estão conseguindo se comunicar entre eles.

## Comunicação entre containers utilizando os seus nomes

Então, o Docker criar uma rede virtual, em que todos os *containers* fazem parte dela, com os IPs automaticamente atribuídos. Mas quando os IPs são atribuídos, cada hora em que subirmos um *container*, ele irá receber um IP novo, que será determinado pelo Docker. Logo, se não sabemos qual o IP que será atribuído, isso não é muito útil quando queremos fazer a comunicação entre os *containers*. Por exemplo, podemos querer colocar dentro do aplicativo o endereço exato do banco de dados, e para saber exatamente o endereço do banco de dados, devemos configurar um nome para aquele *container*.

Mas nomear um *container* nós já sabemos, basta adicionar o `--name`, passando o nome que queremos na hora da criação do *container*, certo? Apesar de conseguirmos dar um nome a um *container*, a rede do Docker não permite com que atribuamos um **hostname** a um *container*, diferentemente de quando criamos a nossa própria rede.

Na rede padrão do Docker, só podemos realizar a comunicação utilizando IPs, mas se criarmos a nossa própria rede, podemos "batizar" os nossos *containers*, e realizar a comunicação entre eles utilizando os seus nomes:



Isso não pode ser feito na rede padrão do Docker, somente quando criamos a nossa própria rede.

## Criando a nossa própria rede do Docker

Então, vamos criar a nossa própria rede, através do comando `docker network create`, mas não é só isso, para esse comando também precisamos dizer qual *driver* vamos utilizar. Para o padrão que vimos, de ter uma nuvem e os *containers* compartilhando a rede, devemos utilizar o *driver* de ***bridge***.

Especificamos o driver através do `--driver` e após isso nós dizemos o nome da rede. Um exemplo do comando é o seguinte:

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

[COPIAR CÓDIGO](#)

Agora, quando criamos um *container*, ao invés de deixarmos ele ser associado à rede padrão do Docker, atrelamos à rede que acabamos de criar, através da flag `--network`. Vamos aproveitar e nomear o *container*:

```
docker run -it --name meu-container-de-ubuntu --network minha-rede ubuntu
```

[COPIAR CÓDIGO](#)

Agora, se executarmos o comando `docker inspect meu-container-de-ubuntu`, podemos ver em *NetworkSettings* o *container* está na rede ***minha-rede***. E para testar a comunicação entre os *containers* na nossa rede, vamos abrir outro terminal e criar um segundo *container*:

```
docker run -it --name segundo-ubuntu --network minha-rede ubuntu
```

[COPIAR CÓDIGO](#)

Agora, no ***segundo-ubuntu***, instalamos o `ping` e testamos a comunicação com o ***meu-container-de-ubuntu***:

```
root@00f93075d079:/# ping meu-container-de-ubuntu  
PING meu-container-de-ubuntu (172.18.0.2) 56(84) bytes of data.
```

```
64 bytes from meu-container-de-ubuntu.minha-rede (172.18.0.2): icmp_seq=1 ttl=64 time=0.210 ms
64 bytes from meu-container-de-ubuntu.minha-rede (172.18.0.2): icmp_seq=2 ttl=64 time=0.148 ms
64 bytes from meu-container-de-ubuntu.minha-rede (172.18.0.2): icmp_seq=3 ttl=64 time=0.138 ms
^C
--- meu-container-de-ubuntu ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.138/0.165/0.210/0.033 ms
```

[COPIAR CÓDIGO](#)

Conseguimos realizar a comunicação entre os *containers* utilizando somente os seus nomes. É como se o **Docker Host**, o ambiente que está rodando os *containers*, criasse uma rede local chamada **minha-rede**, e o nome do *container* será utilizado como se fosse um *hostname*.

Mas lembrando que só conseguimos fazer isso em redes próprias, redes que criamos, isso não é possível na rede padrão dos *containers*.



## Transcrição

Para praticar o que vimos sobre redes no Docker, vamos criar uma pequena aplicação que se conectará ao banco de dados, utilizando tudo o que vimos no vídeo anterior.

O que vamos fazer é utilizar a aplicação **alura-books**, que irá pegar os dados de um banco de dados de livros e exibi-los em uma página web. É uma aplicação feita em Node.js e o banco de dados é o MongoDB.

## Pegando dados de um banco em um outro container

Então, primeiramente vamos baixar essas duas imagens, a imagem **douglasq/alura-books** na versão **cap05** e a imagem **mongo**:

```
docker pull douglasq/alura-books:cap05  
docker pull mongo
```

[COPIAR CÓDIGO](#)

Na imagem **douglasq/alura-books**, não há muito mistério. Ela possui o arquivo **server.js**, que carrega algumas dependências e módulos que são instalados no momento em que rodamos a imagem. Esse arquivo carrega também as configurações do banco, que diz onde o banco de dados estará em execução, no caso o seu *host* será **meu-mongo**, e o *database*, com nome de **alura-books**. Então, quando formos rodar o *container* de MongoDB, seu nome deverá ser **meu-mongo**. Além disso, o arquivo realiza a conexão com o banco, configura a porta que será utilizada (**3000**) e levanta o servidor .

No **Dockerfile** da imagem, também não há mistério, é basicamente o que vimos no vídeo anterior. Por fim, temos as **rotas**, que são duas: a rota `/`, que carrega os livros e os exibe na página, e a rota `/seed`, que salva os livros no banco de dados.

Caso queira, você pode baixar [aqui](https://s3.amazonaws.com/caelum-online-public/646-docker/05/projetos/alura-docker-cap05.zip) (<https://s3.amazonaws.com/caelum-online-public/646-docker/05/projetos/alura-docker-cap05.zip>) o código da versão **cap05** da imagem **alura-books**.

Visto isso, já podemos subir a imagem:

```
docker run -d -p 8080:3000 douglasq/alura-books:cap05
```

[COPIAR CÓDIGO](#)

Ao acessar a página <http://localhost:8080/> (<http://localhost:8080/>), nenhum livro nos é exibido, pois além de não termos levantado o banco de dados, nós não salvamos nenhum dado nele. Então, vamos excluir esse *container* e subir o *container* do MongoDB, lembrando que o seu nome deve ser **meu-mongo**, e vamos colocá-lo na rede que criamos no vídeo anterior:

```
docker run -d --name meu-mongo --network minha-rede mongo
```

[COPIAR CÓDIGO](#)

Com o banco de dados rodando, podemos subir a aplicação do mesmo jeito que fizemos anteriormente, mas **não podemos nos esquecer que ele deve estar na mesma rede do banco de dados**, logo vamos configurar isso também:

```
docker run --network minha-rede -d -p 8080:3000 douglasq/alura-books:cap05
```

[COPIAR CÓDIGO](#)

Agora, acessamos a página <http://localhost:8080/seed/> (<http://localhost:8080/seed/>) para salvar os livros no banco de dados. Após isso, acessamos a página <http://localhost:8080/> (<http://localhost:8080/>) e vemos os dados livros são extraídos do banco e

são exibidos na página. Para provar isso, podemos parar a execução do `meu-mongo` e atualizar a página, veremos que nenhum livro mais será exibido.

Então, esse foi um exemplo para praticar a comunicação entre *containers*, sempre lembrando que devemos colocá-los na mesma rede. Na próxima aula, veremos um jeito de orquestrar melhor diversos *containers* e automatizar esse processo de levantá-los e configurá-los, ao invés de fazer tudo na mão.

## O que aprendemos?



78%

Neste capítulo aprendemos:

- Que imagens criadas pelo Docker acessam a mesma rede, porém apenas através de IP.
- Ao criar uma rede é possível realizar a comunicação entre os containers através do seu nome.
- Que durante a criação de uma rede precisamos indicar qual driver utilizaremos, geralmente, o driver `bridge`.

ATIVIDADES  
8 de 8FÓRUM DO  
CURSOVOLTAR  
PARA  
DASHBOARDMODO  
NOTURNOABRIR  
CADERNO

92.0k xp

a



## Transcrição

Nesta aula, estudaremos uma tecnologia chamada **Docker Compose**, que nos auxiliará a lidar com múltiplos *containers* simultaneamente.

Na aula anterior, para subir a aplicação **alura-books**, foi necessário subirmos dois *containers*, executando os seguintes comandos:

```
docker run -d --name meu-mongo --network minha-rede mongo  
docker run --network minha-rede -d -p 8080:3000 douglasq/alura-books:cap05
```

[COPIAR CÓDIGO](#)

Isso tudo depois de termos construído pelo menos a imagem **douglasq/alura-books**

## O problema

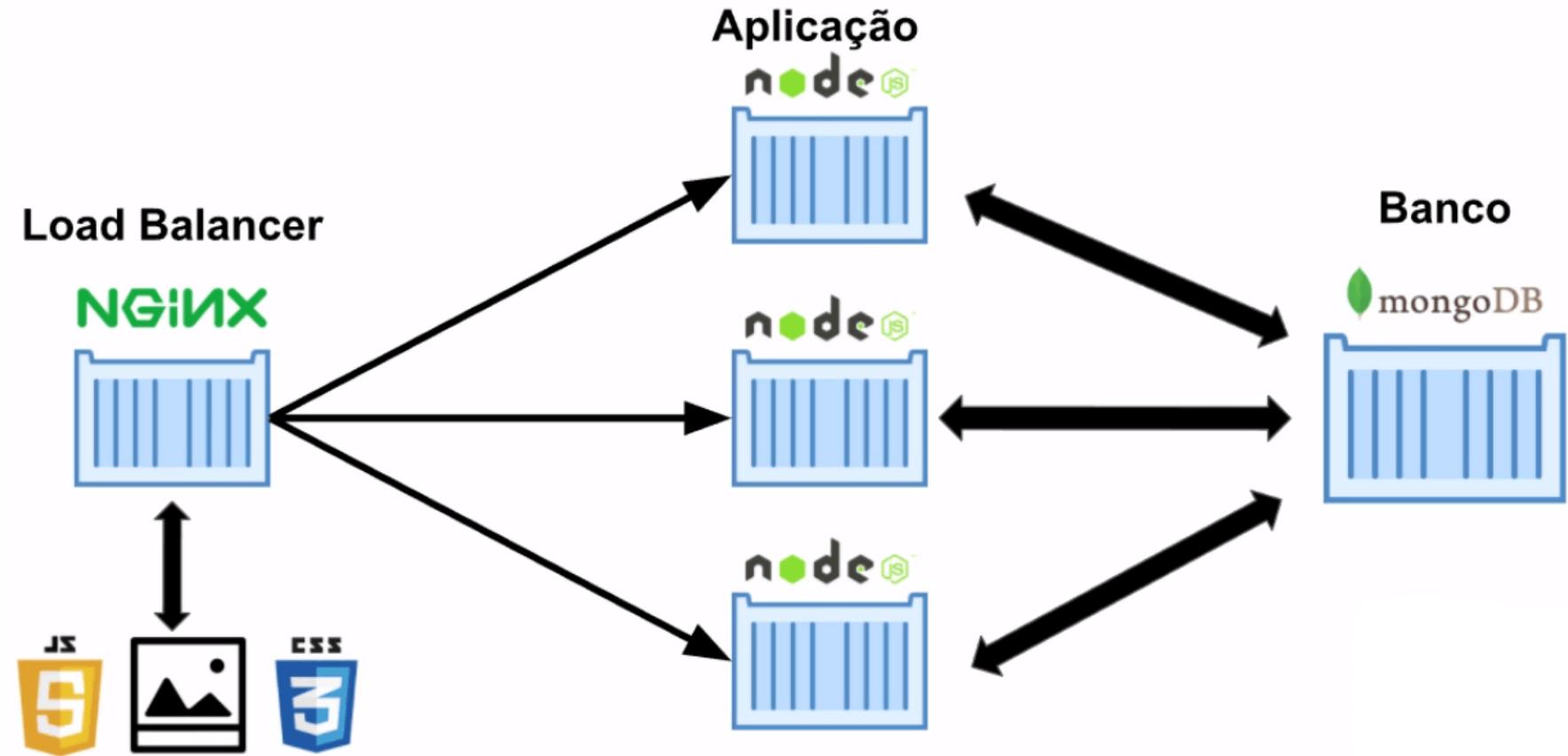
Esses dois comandos criam dois *containers*, mas subindo eles desse jeito manual, é muito comum esquecermos de passar alguma *flag*, ou subir o *container* na ordem errada, sem a devida rede, ou seja, é um trabalho muito manual e facilmente suscetível a erros, isso com somente dois *containers*.

Esse modo de subir os *containers* na mão é bom se quisermos criar um ambiente rapidamente, ou são poucos *containers*, mas quando a aplicação começa a crescer, temos que digitar muitos comandos.

## Funcionamento das aplicações em geral

Na vida real, sabemos que a aplicação é maior que somente dois *containers*, geralmente temos dois, três ou mais *containers* para segurar o tráfego da aplicação, distribuindo a carga. Além disso, temos que colocar todos esses *containers* para se comunicar com o banco de dados em um outro *container*, mas quanto maior a aplicação, devemos ter mais de um *container* para o banco também.

E claro, se temos três aplicações rodando, não podemos ter três endereços diferentes, então nesses casos utilizamos um *Load Balancer* em um outro *container*, para fazer a distribuição de carga quando tivermos muitos acessos. Ele recebe as requisições e distribui para uma das aplicações, e ele também é muito utilizado para servir os arquivos estáticos, como imagens, arquivos CSS e JavaScript. Assim, a nossa aplicação controla somente a lógica, as regras de negócio, com os arquivos estáticos ficando a cargo do *Load Balancer*.



Se formos seguir esse diagrama, teríamos que criar cinco *containers* na mão, e claro, cada *container* com configurações e *flags* diferentes, além de termos que nos preocupar com a ordem em que vamos subi-los.

## Docker Compose

Ao invés de subir todos esses *containers* na mão, o que vamos fazer é utilizar uma tecnologia aliada do Docker, chamada **Docker Compose**, feito para nos auxiliar a orquestrar melhor múltiplos *containers*. Ele funciona seguindo um arquivo de texto **YAML** (extensão **.yml**), e nele nós descrevemos tudo o que queremos que aconteça para subir a nossa aplicação, todo o nosso processo de *build*, isto é, subir o banco, os *containers* das aplicações, etc.

Assim, não precisamos ficar executando muitos comandos no terminal sem necessidade. E esse será o foco desta aula, montar uma aplicação na estrutura descrita anteriormente na imagem, que é uma situação comum no nosso dia-a-dia.



## Transcrição

O código do projeto pode ser baixado [aqui](https://s3.amazonaws.com/caelum-online-public/646-docker/06/projetos/alura-docker-cap06.zip) (<https://s3.amazonaws.com/caelum-online-public/646-docker/06/projetos/alura-docker-cap06.zip>).

Para começarmos a entender como funciona o **Docker Compose**, primeiramente vamos entender como funciona a aplicação que utilizaremos como base. É uma aplicação bem semelhante à utilizada na aula anterior, com o mesmo servidor, rotas e banco de dados. De novidade, é que agora precisamos criar o **NGINX**, que é mais um *container* que devemos subir.

Então, ou utilizamos a imagem **nginx**, ou criamos a nossa própria. Como vamos configurar o **NGINX** para algumas coisas específicas, como lidar com os arquivos estáticos, vamos criar a nossa própria imagem, por isso que na aplicação há o **nginx.dockerfile**:

```
FROM nginx:latest
MAINTAINER Douglas Quintanilha
COPY /public /var/www/public
COPY /docker/config/nginx.conf /etc/nginx/nginx.conf
EXPOSE 80 443
ENTRYPOINT ["nginx"]
```

```
# Parâmetros extras para o entrypoint  
CMD ["-g", "daemon off;"]
```

[COPIAR CÓDIGO](#)

Nesse arquivo, nós utilizamos a última versão disponível da imagem do **nginx** como base, e copiamos o conteúdo da pasta **public**, que contém os arquivos estáticos, e um arquivo de configuração do NGINX para dentro do *container*. Além disso, abrimos as portas **80** e **443** e executa o NGINX através do comando **nginx**, passando os parâmetros extras **-g** e **daemon off**.

Por fim, vamos ver um pouco sobre o arquivo de configuração do NGINX, para entendermos um pouco como o *load balancer* está funcionando.

No arquivo **nginx.conf**, dentro **server**, está a parte que trata de servir os arquivos estáticos. Na porta **80**, no *localhost*, em **/var/www/public**, ele será responsável por servir as pastas **css**, **img** e **js**. E todo resto, que não for esses três locais, ele irá jogar para o **node\_upstream**.

No **node\_upstream**, é onde ficam as configurações para o NGINX redirecionar as conexões que ele receber para um dos três *containers* da nossa aplicação. O redirecionamento acontecerá de forma circular, ou seja, a primeira conexão irá para o primeiro *container*, a segunda irá para o segundo *container*, a terceira irá para o terceiro *container*, na quarta, começa tudo de novo, e ela vai para o primeiro *container* e assim por diante.

Isso já está tudo pronto, basta baixarmos o código da imagem e da aplicação [aqui \(<https://s3.amazonaws.com/caelum-online-public/646-docker/06/projetos/alura-docker-cap06.zip>\)](https://s3.amazonaws.com/caelum-online-public/646-docker/06/projetos/alura-docker-cap06.zip).

Agora, no próximo vídeo, escreveremos o responsável por orquestrar a subida de cada uma dessas partes da nossa aplicação, o **docker-compose.yml**.



## Transcrição

Para utilizar o **Docker Compose**, devemos criar o seu arquivo de configuração, o **docker-compose.yml**, na raiz do projeto. Em todo arquivo de **Docker Compose**, que é uma espécie de receita de bolo para construirmos as diferentes partes da nossa aplicação, a primeira coisa que colocamos nele é a versão do Docker Compose que estamos utilizando:

```
version: '3'
```

[COPIAR CÓDIGO](#)

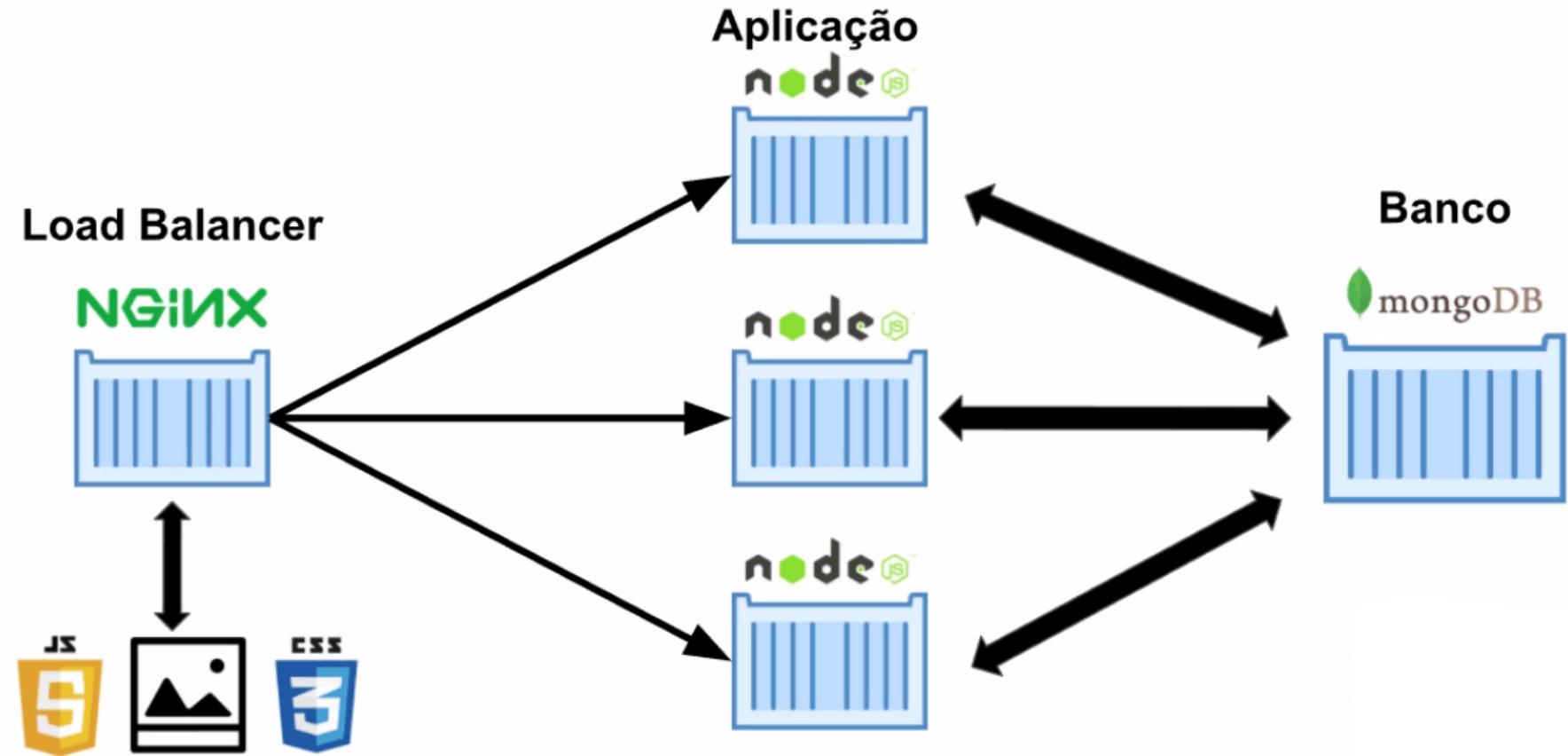
Estamos utilizando a versão 3 pois é a versão mais recente no momento da criação do treinamento. O YAML lembra um pouco o JSON, mas ao invés de utilizar as chaves para indentar o código, ele utiliza **espaços**.

Agora, começamos a descrever os nossos serviços, os nossos **services** :

```
version: '3'  
services:
```

[COPIAR CÓDIGO](#)

Um serviço é uma parte da nossa aplicação. Lembrando do nosso diagrama:



Temos NGINX, três Node, e o MongoDB como serviços. Logo, se queremos construir cinco *containers*, vamos construir cinco serviços, cada um deles com um nome específico.

Então, vamos começar construindo o **NGINX**, que terá o nome `nginx` :

```

version: '3'
services:
  nginx:

```

[COPIAR CÓDIGO](#)

Em cada serviço, devemos dizer como devemos construí-lo, como devemos fazer o seu `build` :

```
version: '3'  
services:  
  nginx:  
    build:
```

[COPIAR CÓDIGO](#)

O serviço será construído através de um **Dockerfile**, então devemos passá-lo onde ele está. E também devemos passar um **contexto**, para dizermos a partir de onde o **Dockerfile** deve ser buscado. Como ele será buscado a partir da pasta atual, vamos utilizar o ponto:

```
version: '3'  
services:  
  nginx:  
    build:  
      dockerfile: ./docker/nginx.dockerfile  
      context: .
```

[COPIAR CÓDIGO](#)

Construída a imagem, devemos dar um nome para ela, por exemplo **douglasq/nginx**:

```
version: '3'  
services:  
  nginx:  
    build:  
      dockerfile: ./docker/nginx.dockerfile  
      context: .  
    image: douglasq/nginx
```

[COPIAR CÓDIGO](#)

E quando o Docker Compose criar um *container* a partir dessa imagem, vamos dizer que o seu nome será **nginx**:

```
version: '3'  
services:  
  nginx:  
    build:  
      dockerfile: ./docker/nginx.dockerfile  
      context: .  
    image: douglasq/nginx  
    container_name: nginx
```

[COPIAR CÓDIGO](#)

Sabemos também que o NGINX trabalha com duas portas, a **80** e a **443**. Como não estamos trabalhando com HTTPS, vamos utilizar somente a porta **80**, e no próprio arquivo, podemos dizer para qual porta da nossa máquina queremos mapear a porta **80** do *container*. Vamos mapear para a porta de mesmo número da nossa máquina:

```
version: '3'  
services:  
  nginx:  
    build:  
      dockerfile: ./docker/nginx.dockerfile  
      context: .  
    image: douglasq/nginx  
    container_name: nginx  
    ports:  
      - "80:80"
```

[COPIAR CÓDIGO](#)

No YAML, toda vez que colocamos um traço, significa que a propriedade pode receber mais de um item. Agora, para os *containers* conseguirem se comunicar, eles devem estar na mesma rede, então vamos configurar isso também. Primeiramente, devemos criar a rede, que não é um serviço, então vamos escrever do começo do arquivo, sem as tabulações:

```
version: '3'  
services:  
    nginx:  
        build:  
            dockerfile: ./docker/nginx.dockerfile  
            context: .  
        image: douglasq/nginx  
        container_name: nginx  
        ports:  
            - "80:80"  
  
networks:
```

[COPIAR CÓDIGO](#)

O nome da rede será **production-network** e utilizará o *driver bridge*:

```
version: '3'  
services:  
    nginx:  
        build:  
            dockerfile: ./docker/nginx.dockerfile  
            context: .  
        image: douglasq/nginx  
        container_name: nginx  
        ports:  
            - "80:80"
```

- "80:80"

```
networks:  
  production-network:  
    driver: bridge
```

[COPIAR CÓDIGO](#)

Com a rede criada, vamos utilizá-la no serviço:

```
version: '3'  
services:  
  nginx:  
    build:  
      dockerfile: ./docker/nginx.dockerfile  
      context: .  
    image: douglasq/nginx  
    container_name: nginx  
    ports:  
      - "80:80"  
    networks:  
      - production-network
```

```
networks:  
  production-network:  
    driver: bridge
```

[COPIAR CÓDIGO](#)

Isso é para construir o serviço do NGINX, agora vamos construir o serviço do MongoDB, com o nome **mongodb**. Como ele será construído a partir da imagem **mongo**, não vamos utilizar nenhum Dockerfile, logo não utilizamos a propriedade **build**. Além disso, não podemos nos esquecer de colocá-lo na rede que criamos:

```
version: '3'
services:
  nginx:
    build:
      dockerfile: ./docker/nginx.dockerfile
      context: .
    image: douglasq/nginx
    container_name: nginx
    ports:
      - "80:80"
    networks:
      - production-network

  mongodb:
    image: mongo
    networks:
      - production-network

networks:
  production-network:
    driver: bridge
```

[COPIAR CÓDIGO](#)

Falta agora criarmos os três serviços em que ficará a nossa aplicação, **node1**, **node2** e **node3**. Para eles, será semelhante ao NGINX, com Dockerfile **alura-books.dockerfile**, contexto, rede **production-network** e porta **3000**:

```
version: '3'
services:
  nginx:
    build:
```

```
dockerfile: ./docker/nginx.dockerfile
context: .
image: douglasq/nginx
container_name: nginx
ports:
- "80:80"
networks:
- production-network

mongodb:
image: mongo
networks:
- production-network

node1:
build:
dockerfile: ./docker/alura-books.dockerfile
context: .
image: douglasq/alura-books
container_name: alura-books-1
ports:
- "3000"
networks:
- production-network

node2:
build:
dockerfile: ./docker/alura-books.dockerfile
context: .
image: douglasq/alura-books
container_name: alura-books-2
ports:
```

```
- "3000"
networks:
  - production-network

node3:
  build:
    dockerfile: ./docker/alura-books.dockerfile
    context: .
  image: douglasq/alura-books
  container_name: alura-books-3
  ports:
    - "3000"
  networks:
    - production-network

networks:
  production-network:
    driver: bridge
```

[COPIAR CÓDIGO](#)

Com isso, a construção dos nossos serviços está finalizada.

## Ordem dos serviços

Por último, quando subimos os *containers* na mão, temos uma ordem, primeiro devemos subir o **mongodb**, depois a nossa aplicação, ou seja, **node1**, **node2** e **node3** e após tudo isso subimos o **nginx**. Mas como que fazemos isso no **docker-compose.yml**?

Nós podemos dizer que os serviços da nossa aplicação **dependem** que um serviço suba antes deles, o serviço do **mongodb**:

```
version: '3'
services:
  nginx:
    build:
      dockerfile: ./docker/nginx.dockerfile
      context: .
    image: douglasq/nginx
    container_name: nginx
    ports:
      - "80:80"
    networks:
      - production-network

  mongodb:
    image: mongo
    networks:
      - production-network

  node1:
    build:
      dockerfile: ./docker/alura-books.dockerfile
      context: .
    image: douglasq/alura-books
    container_name: alura-books-1
    ports:
      - "3000"
    networks:
      - production-network
    depends_on:
      - "mongodb"
```

```
node2:
  build:
    dockerfile: ./docker/alura-books.dockerfile
    context: .
  image: douglasq/alura-books
  container_name: alura-books-2
  ports:
    - "3000"
  networks:
    - production-network
  depends_on:
    - "mongodb"

node3:
  build:
    dockerfile: ./docker/alura-books.dockerfile
    context: .
  image: douglasq/alura-books
  container_name: alura-books-3
  ports:
    - "3000"
  networks:
    - production-network
  depends_on:
    - "mongodb"

networks:
  production-network:
    driver: bridge
```

[COPIAR CÓDIGO](#)

Da mesma forma, dizemos que o serviço do **nginx** depende dos serviços **node1**, **node2** e **node3**:

```
version: '3'
services:
  nginx:
    build:
      dockerfile: ./docker/nginx.dockerfile
      context: .
    image: douglasq/nginx
    container_name: nginx
    ports:
      - "80:80"
    networks:
      - production-network
    depends_on:
      - "node1"
      - "node2"
      - "node3"

  mongodb:
    image: mongo
    networks:
      - production-network

  node1:
    build:
      dockerfile: ./docker/alura-books.dockerfile
      context: .
    image: douglasq/alura-books
    container_name: alura-books-1
    ports:
      - "3000"
    networks:
```

```
- production-network
depends_on:
  - "mongodb"

node2:
  build:
    dockerfile: ./docker/alura-books.dockerfile
    context: .
  image: douglasq/alura-books
  container_name: alura-books-2
  ports:
    - "3000"
  networks:
    - production-network
  depends_on:
    - "mongodb"

node3:
  build:
    dockerfile: ./docker/alura-books.dockerfile
    context: .
  image: douglasq/alura-books
  container_name: alura-books-3
  ports:
    - "3000"
  networks:
    - production-network
  depends_on:
    - "mongodb"

networks:
```

```
production-network:  
  driver: bridge
```

[COPIAR CÓDIGO](#)

Assim, encerramos a configuração do `docker-compose.yml`. Vamos ver como subir a aplicação a partir desse arquivo no próximo vídeo.

## Instalando o Docker Compose no Linux

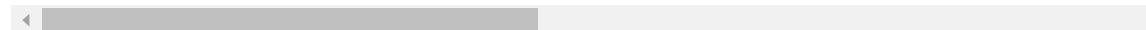


88%

ATIVIDADES  
6 de 13FÓRUM DO  
CURSOVOLTAR  
PARA  
DASHBOARDMODO  
NOTURNOABRIR  
CADERNO

O **Docker Compose** não é instalado por padrão no Linux, então você deve instalá-lo por fora. Para tal, baixe-o na sua versão mais atual, que pode ser visualizada no seu [GitHub \(https://github.com/docker/compose/releases\)](https://github.com/docker/compose/releases), executando o comando abaixo:

```
sudo curl -L https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)
```

COPIAR CÓDIGO

Após isso, dê permissão de execução para o **docker-compose**:

```
sudo chmod +x /usr/local/bin/docker-compose
```

COPIAR CÓDIGO

Pronto, o **Docker Compose** já está instalado no seu Linux!



92.1k xp

a



## Transcrição

Se você utiliza Linux, provavelmente não conseguirá utilizar o **Docker Compose**, pois o mesmo não é instalado na instalação padrão. Para instalá-lo, primeiramente faça [este exercício](https://cursos.alura.com.br/course/docker-e-docker-compose/task/29559) (<https://cursos.alura.com.br/course/docker-e-docker-compose/task/29559>) e prossiga com o treinamento.

Com o `docker-compose.yml` pronto, podemos subir os serviços, mas antes devemos garantir que temos todas as imagens envolvidas neste arquivo na nossa máquina. Para isso, dentro da pasta do nosso projeto, executamos o seguinte comando:

```
$ cd Desktop/alura-docker-cap06/  
~/Desktop/alura-docker-cap06$ sudo docker-compose build
```

[COPIAR CÓDIGO](#)

Com os serviços criados, podemos subi-los através do comando `docker-compose up`. Esse comando irá seguir o que escrevemos no `docker-compose.yml`, ou seja, cria a rede, o *container* do MongoDB, os três *containers* da aplicação e o *container* do NGINX. Depois, são exibidos alguns *logs*, sendo que cada um dos *containers* fica com uma cor diferente, para podermos distinguir melhor.

Se tudo funcionou, quando acessarmos o NGINX, seremos redirecionados para algum dos *containers* da nossa aplicação. Configuramos a porta 80 para acessar o NGINX, então vamos acessar no navegador a página <http://localhost:80/> (<http://localhost:80/>). Ao acessar a página, vemos a seguinte mensagem no *log*:

Ou seja, fomos redirecionados para o primeiro *container*. Ao atualizar a página, vemos no *log* que fomos redirecionados para o segundo *container*, ou seja, o *load balancer* está funcionando.

Vamos então alimentar o banco de dados acessando a página <http://localhost:80/seed> (<http://localhost:80/seed>), e novamente acessar a página inicial, a <http://localhost:80/> (<http://localhost:80/>), assim veremos os livros sendo exibidos.

Então, com um único comando, levantamos todos os *containers*, montamos o *load balancer*, servimos os arquivos estáticos, criamos o banco e colocamos todos eles na mesma rede, bem mais prático do que estávamos fazendo anteriormente.

Para parar a execução, utilizamos o atalho **CTRL + C**. E não somos obrigados a ficar vendo esses *logs*, podemos utilizar a já conhecida *flag* **-d**:

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

COPIAR CÓDIGO

E com o comando **docker-compose ps**, podemos ter uma visualização simples dos serviços que estão rodando:

Name	Command	State	Ports
<hr/>			
alura-books-1	npm start	Up	0.0.0.0:9022->3000/tcp
alura-books-2	npm start	Up	0.0.0.0:9021->3000/tcp
alura-books-3	npm start	Up	0.0.0.0:9023->3000/tcp
aluradockercap06_mongodb_1	docker-entrypoint.sh mongod	Up	27017/tcp
nginx	nginx -g daemon off;	Up	443/tcp, 0.0.0.0:88->80/tcp

[COPIAR CÓDIGO](#)

Agora que não estamos mais vendo os *logs*, como paramos os serviços? Para isso, utilizamos o comando `docker-compose down`. Esse comando para os *containers* e os remove.

E não é por que eles são serviços, que eles não tem um *container* por debaixo dos panos, então nós conseguimos interagir com os *containers* utilizando todos os comandos que já vimos no treinamento, por exemplo para testar a comunicação entre eles:

```
docker exec -it alura-books-1 ping alura-books-2
```

[COPIAR CÓDIGO](#)

Mas também podemos utilizar o nome do **serviço**, não precisamos necessariamente utilizar o nome do *container*:

```
docker exec -it alura-books-1 ping node2
```

[COPIAR CÓDIGO](#)

Assim conseguimos fazer a comunicação tanto com o nome do *container* quanto com o nome do serviço, pois os dois estão atrelados ao mesmo IP.

## Para saber mais: Docker e Microsserviços

### Sobre Microsserviços

Já ouviu falar de Microsserviços? Se já ouviu, pode pular a introdução abaixo e ir diretamente para a parte "Docker e Microsserviços", senão continue comigo.

Uma forma de desenvolver uma aplicação é colocar todas as funcionalidades em um único "lugar". Ou seja, a aplicação roda em uma única instância (ou servidor) que possui todas as funcionalidades. Isso talvez seja a forma mais simples de criar uma aplicação (também a mais natural), mas quando a base de código cresce, alguns problemas podem aparecer.

Por exemplo, qualquer atualização ou *bug fix* necessita parar todo o sistema, buildar o sistema todo e subir novamente. Isso pode ficar demorado e lento. Em geral, quanto maior a base de código mais difícil será manter ela mesmo com uma boa cobertura de testes e as desvantagens não param por ai. Outro problema é se alguma funcionalidade possuir um gargalo no desempenho o sistema todo será afetado. Não é raro de ver sistemas onde relatórios só devem ser gerados à noite para não afetar o desempenho de outras funcionalidades. Outro problema comum é com os ciclos de testes e *build* demorados (falta de agilidade no desenvolvimento), problemas no monitoramento da aplicação ou falta de escalabilidade. Enfim, o sistema se torna um legado pesado, onde nenhum desenvolvedor gostaria de colocar a mão no fogo.

### Arquitetura de Microsserviços

Então a ideia é fugir desse tipo de arquitetura monolítica monstruosa e dividir ela em pequenos pedaços. Cada pedaço possui uma funcionalidade bem definida e roda como se fosse um "mini sistema" isolado. Ou seja, em vez de termos uma única aplicação enorme, teremos várias instâncias menores que dividem e coordenam o trabalho. Essas instâncias são chamadas de Microsserviços.

Agora fica mais fácil monitorar cada serviço específico, atualizá-lo ou escalá-lo pois a base de código é muito menor, e assim o *deploy* e o teste serão mais rápidos. Podemos agora achar soluções específicas para esse serviço sem precisar alterar os demais. Outra vantagem é que um desenvolvedor novo não precisa conhecer o sistema todo para alterar uma funcionalidade, basta ele focar na funcionalidade desse microsserviço.

Importante também é que um microsserviço seja acessível remotamente, normalmente usando o protocolo HTTP trocando mensagens JSON ou XML, mas nada impede que outro protocolo seja usado. Um microsserviço pode usar outros serviços para coordenar o trabalho.

Repare que isso é uma outra abordagem arquitetural bem diferente do monolítico e por isso também é chamado de *arquitetura de microsserviços*.

Por fim, uma arquitetura de Microsserviços tem um grau de complexidade muito alta se comparada com uma arquitetura monolítica. Aliás, há aqueles profissionais que indicam partir para uma [arquitetura monolítica primeiro e mudar para uma baseada em microsserviços depois](http://sdtimes.com/martin-fowler-monolithic-apps-first-microservices-later/) (<http://sdtimes.com/martin-fowler-monolithic-apps-first-microservices-later/>), quando estritamente necessário.

## Docker e Microsserviços

Trabalhar com uma arquitetura de microsserviços gera a necessidade de publicar o serviço de maneira rápida, leve, isolada e vimos que o **Docker** possui exatamente essas características! Com **Docker** e **Docker Compose** podemos criar um ambiente ideal para a publicação destes serviços.

O **Docker** é uma ótima opção para rodar os microsserviços pelo fato de isolar os *containers*. Essa utilização de *containers* para serviços individuais faz com que seja muito simples gerenciar e atualizar esses serviços, de maneira automatizada e rápida.

## Docker Swarm

Ok, tudo bem até aqui. Agora vou ter vários serviços rodando usando o **Docker**. E para facilitar a criação desses containers já aprendemos usar o **Docker Compose** que sabe

subir vários *containers*. O **Docker Compose** é a ferramenta ideal para coordenar a criação dos *containers*, no entanto para melhorar a escalabilidade e desempenho pode ser necessário criar muito mais *containers* para um serviço específico. Em outras palavras, agora gostaríamos de criar *muitos containers* aproveitando *várias máquinas* (virtuais ou físicas)! Ou seja, pode ser que um microsserviço fique rodando em 20 *containers* usando três máquinas físicas diferentes. Como podemos facilmente subir e parar esses *containers*? Repare que o **Docker Compose** não é para isso e por isso existe uma outra ferramenta que se chama **Docker Swarm** (que não faz parte do escopo desse curso).

**Docker Swarm** facilita a criação e administração de um *cluster* de *containers*.

## Docker Cheat Sheet - Os Comandos Utilizados

Segue a lista com os principais comandos utilizados durante o curso:

- Comandos relacionados às informações
  - docker version - exibe a versão do docker que está instalada.
  - docker inspect ID\_CONTAINER - retorna diversas informações sobre o container.
  - docker ps - exibe todos os containers em execução no momento.
  - docker ps -a - exibe todos os containers, independentemente de estarem em execução ou não.
- Comandos relacionados à execução
  - docker run NOME\_DA\_IMAGEM - cria um container com a respectiva imagem passada como parâmetro.
  - docker run -it NOME\_DA\_IMAGEM - conecta o terminal que estamos utilizando com o do container.
  - docker run -d -P --name NOME dockersamples/static-site - ao executar, dá um nome ao container e define uma porta aleatória.
  - docker run -d -p 12345:80 dockersamples/static-site - define uma porta específica para ser atribuída à porta 80 do container, neste caso 12345.
  - docker run -v "CAMINHO\_VOLUME" NOME\_DA\_IMAGEM - cria um volume no respectivo caminho do container.
  - docker run -it --name NOME\_CONTAINER --network NOME\_DA\_REDE NOME\_IMAGEM - cria um container especificando seu nome e qual rede deverá ser usada.
- Comandos relacionados à inicialização/interrupção
  - docker start ID\_CONTAINER - inicia o container com id em questão.
  - docker start -a -i ID\_CONTAINER - inicia o container com id em questão e integra os terminais, além de permitir interação entre ambos.
  - docker stop ID\_CONTAINER - interrompe o container com id em questão.

- Comandos relacionados à remoção
  - docker rm ID\_CONTAINER - remove o container com id em questão.
  - docker container prune - remove todos os containers que estão parados.
  - docker rmi NOME\_DA\_IMAGEM - remove a imagem passada como parâmetro.
- Comandos relacionados à construção de Dockerfile
  - docker build -f Dockerfile - cria uma imagem a partir de um Dockerfile.
  - docker build -f Dockerfile -t NOME\_USUARIO/NOME\_IMAGEM - constrói e nomeia uma imagem não-oficial.
  - docker build -f Dockerfile -t NOME\_USUARIO/NOME\_IMAGEM CAMINHO\_DOCKERFILE - constrói e nomeia uma imagem não-oficial informando o caminho para o Dockerfile.
- Comandos relacionados ao Docker Hub
  - docker login - inicia o processo de login no Docker Hub.
  - docker push NOME\_USUARIO/NOME\_IMAGEM - envia a imagem criada para o Docker Hub.
  - docker pull NOME\_USUARIO/NOME\_IMAGEM - baixa a imagem desejada do Docker Hub.
- Comandos relacionados à rede
  - hostname -i - mostra o ip atribuído ao container pelo docker (funciona apenas dentro do container).
  - docker network create --driver bridge NOME\_DA\_REDE - cria uma rede especificando o driver desejado.
- Comandos relacionados ao docker-compose
  - docker-compose build - Realiza o build dos serviços relacionados ao arquivo docker-compose.yml, assim como verifica a sua sintaxe.
  - docker-compose up - Sobe todos os containers relacionados ao docker-compose, desde que o build já tenha sido executado.
  - docker-compose down - Para todos os serviços em execução que estejam relacionados ao arquivo docker-compose.yml.

