

# **Manual do Docker e Container**

**Prof. Dr. Clayton Pereira**



## Preparação do Ambiente

---

Advanced Institute for Artificial Intelligence – AI2

<https://advancedinstitute.ai>

# 1. Conceitos

## 1.1. O que é Docker?

### Docker

É uma ferramenta que se apoia em recursos existentes no kernel, inicialmente Linux, para isolar a execução de processos. As ferramentas que o Docker traz são basicamente uma camada de administração de containers, baseado originalmente no LXC (Linux Containers).

Alguns isolamentos possíveis:

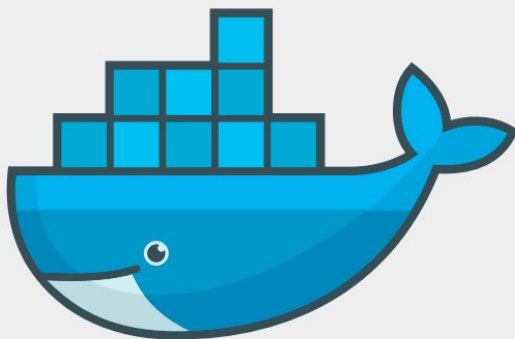
- Limites de uso de memória
- Limites de uso de CPU
- Limites de uso de I/O
- Limites de uso de rede
- Isolamento da rede (que redes e portas são acessíveis)
- Isolamento do file system
- Permissões e Políticas
- Capacidades do kernel

<https://www.docker.com/why-docker>

#### Definição oficial

*Containers* Docker empacotam componentes de *software* em um sistema de arquivos completo, que contém tudo necessário para a execução: código, *runtime*, ferramentas de sistema - qualquer coisa que possa ser instalada em um servidor.

Isto garante que o *software* sempre irá executar da mesma forma, independente do seu ambiente.



## 1.2. O que são *Containers*?

### Container

É o nome dado para a segregação de processos no mesmo *kernel*, de forma que o processo seja isolado o máximo possível de todo o resto do ambiente. Em termos práticos são *File Systems*, criados a partir de uma “imagem” e que podem possuir também algumas características próprias.

## 1.3. O que são *imagens Docker*?

### Imagens

Uma imagem Docker é a materialização de um modelo de um sistema de arquivos, modelo este produzido através de um processo conhecido como *Build*. Esta imagem é representada por um ou mais arquivos e pode ser armazenada em um repositório, como o [Docker Hub](#) que falamos na aula passada.

## 1.4. Diferença entre *Imagens e Containers*

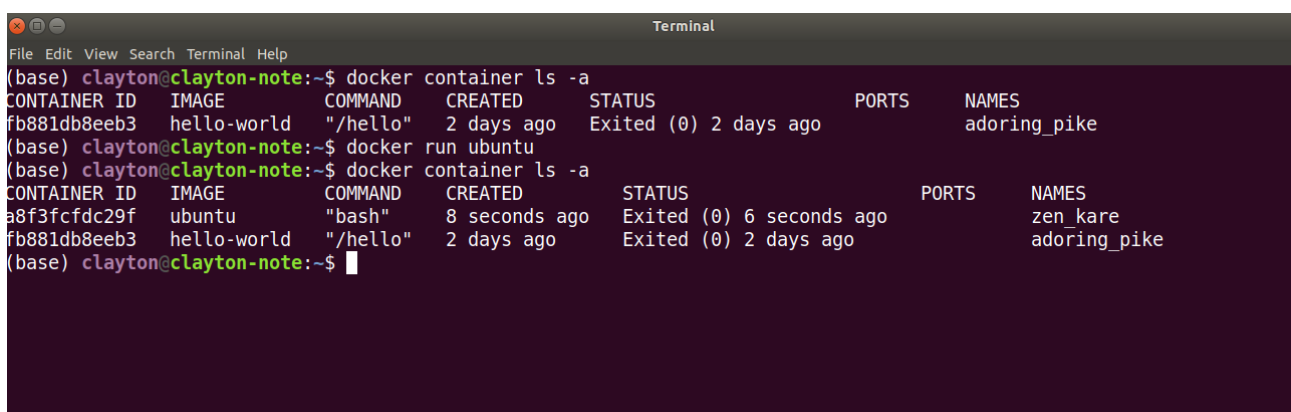
Quando executamos o comando *docker run hello-world*, a primeira coisa que o **Docker** faz é verificar se temos a imagem *hello-world* no nosso computador, caso não tenhamos, o Docker irá buscar no [Docker Hub](#) e baixar. A imagem é como se fosse uma receita de bolo, uma série de instruções que o Docker seguirá para criar um container, que irá conter as instruções da imagem, no caso, *hello-world*.

Existem milhares de imagens armazenadas no Docker Hub, todas disponíveis para aplicação, como é o caso da imagem do SO Ubuntu:

### **docker run ubuntu**

Ao executarmos o comando, o download será iniciado imediatamente. Podemos verificar que a imagem é dividida em camadas e ao termino do processo, nenhuma mensagem é exibida porque o container foi criado porém, a imagem não executa nada! Podemos verificar isso vendo os containers que estão sendo executado no momento:

### **docker ps -a**

A terminal window titled "Terminal" showing the output of Docker commands. The user is in a shell with prompt "(base) clayton@clayton-note:~\$". They run "docker container ls -a", which shows a table with columns: CONTAINER ID, IMAGE, COMMAND, CREATED, STATUS, PORTS, and NAMES. The first row shows a container with ID fb881db8eeb3, image hello-world, command "/hello", created 2 days ago, status "Exited (0) 2 days ago", and name adoring\_pike. Then they run "docker run ubuntu", which shows a new container being created. Finally, they run "docker container ls -a" again, showing the new container with ID a8f3fcfdc29f, image ubuntu, command "bash", created 8 seconds ago, status "Exited (0) 6 seconds ago", and name zen\_kare. The previous container is still listed with status "Exited (0) 2 days ago".

```
(base) clayton@clayton-note:~$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
fb881db8eeb3   hello-world  "/hello"   2 days ago   Exited (0) 2 days ago           adoring_pike
(base) clayton@clayton-note:~$ docker run ubuntu
(base) clayton@clayton-note:~$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
a8f3fcfdc29f   ubuntu    "bash"     8 seconds ago   Exited (0) 6 seconds ago       zen_kare
fb881db8eeb3   hello-world  "/hello"   2 days ago   Exited (0) 2 days ago           adoring_pike
(base) clayton@clayton-note:~$
```

Podemos verificar que após executar o comando para baixar a imagem do Ubuntu, nenhuma mensagem foi exibida porém, ao listarmos os containers ele aparece como criado. É possível verificar ainda o **id**, o **nome** dentre outras informações dos containers.

Lembram-se do comando **echo** que utilizamos nas aulas de Linux? Pois bem... podemos testá-los também no container criado através do comando:

```
docker run ubuntu echo "Aula de containers"
```

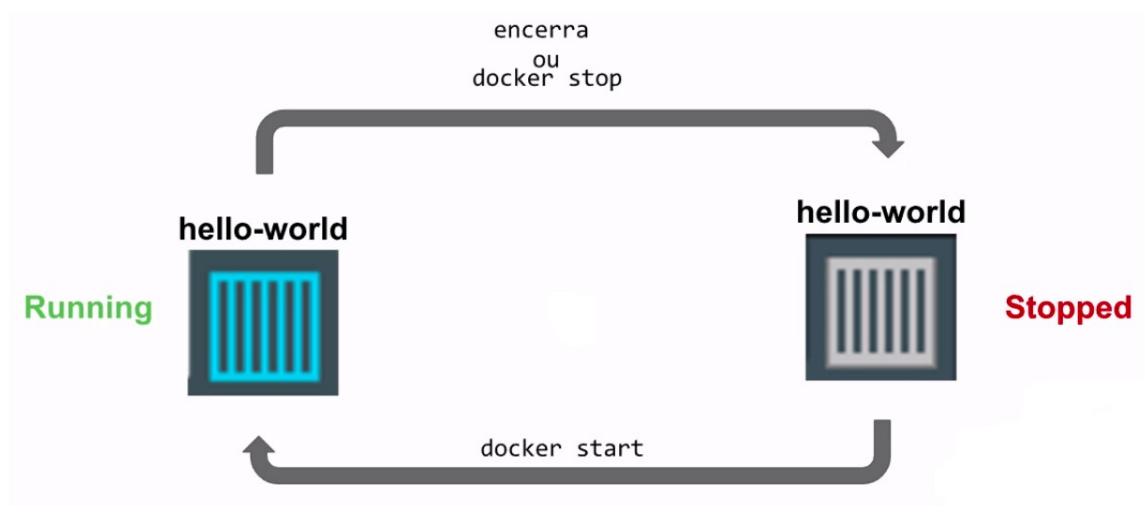
## 1.5. Trabalhando dentro de um *Container*.

Podemos fazer com que o terminal da nossa máquina seja integrado ao terminal de dentro do *container*, para ficar um terminal iterativo através da *flag -it*.

```
docker run -it ubuntu
```

Através do comando **CTRL+D** ou simplesmente um **exit**, paramos a execução da nossa máquina ubuntu e caso queira iniciá-la novamente, basta adicionar o id do container junto ao comando **start**, o mesmo caso queira parar o container, mas agora utilizando o comando **stop**.

```
docker container start -a -i (id do container)
```



1.6.

## Removendo um *Container*

Para removermos um container, é necessário também passar o seu id, como apresentado no comando abaixo:

```
docker rm (id do container)
```

Uma outra forma para removermos os containers inativos sem ter que executar essa tarefa um por um dos container, podemos utilizar o comando **prune**, que serve para limpar algo específico do Docker, como queremos remover container parados, executamos o seguinte comando:

## docker container prune

Esse comando é considerado “poderosíssimo” e também perigoso por isso, é realizada também uma confirmação antes da execução total do processo, como apresentado na imagem abaixo:

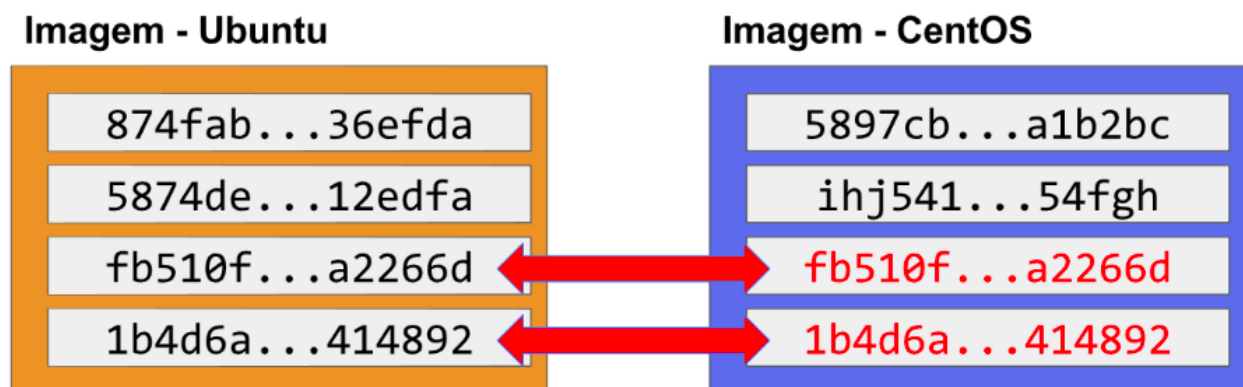
```
root@649c4737ecad: /home
File Edit View Search Terminal Help
(base) clayton@clayton-note:~$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
649c4737ecad   ubuntu    "bash"    16 minutes ago   Exited (0) 15 minutes ago   interesting_m
eninsky
0aab94fa10ef   ubuntu    "echo 'Aula de conta..." 19 minutes ago   Exited (0) 19 minutes ago   musing_beaver
a8f3fcfdc29f   ubuntu    "bash"    26 minutes ago   Exited (0) 12 minutes ago   zen_kare
fb881db8eeb3   hello-world  "/hello"   2 days ago       Exited (0) 2 days ago       adoring_pike
(base) clayton@clayton-note:~$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
649c4737ecad19ecfaad0091a991297e06a9f46d0a8b1c03c2b6609178798d4c
0aab94fa10ef0304a300a49ab9a859acae3bdc9802443d39aa912a714f01c97
a8f3fcfdc29f02d62d38294088814023558f8ce3c57dca2e51478cc860dc4472
fb881db8eeb399017147577082bc6c1a0832305d76a106dfb25ce3683bfa5f87

Total reclaimed space: 16B
(base) clayton@clayton-note:~$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
(base) clayton@clayton-note:~$
```

## 1.6. Camadas de uma imagem

Quando baixamos a imagem do Ubuntu, reparamos que ela possui algumas 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 poupamos tempo, já que precisamos de menos tempo para baixar uma imagem.

## 2. Vamos então criar um novo Container para um site Estático

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 através do seguinte comando:

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

### 2.1. Imagens e seus *usernames*

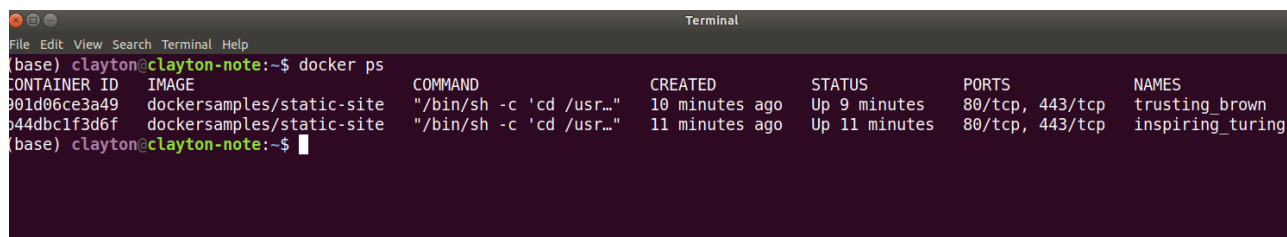
Nas imagens que vimos anteriormente, para as imagens oficiais não é necessário colocarmos um *username* na hora de baixá-las. Esse *username* representa o usuário que toma conta da imagem, quem a criou. Como a imagem que vamos utilizar foi criada por outro(s) usuário(s), precisamos especificar o seu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

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

Com isso agora temos nosso container sendo executado porém, onde está o site estático? Qual é a porta de acesso??



```
(base) clayton@clayton-note:~$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                NAMES
901d06ce3a49   dockersamples/static-site           "/bin/sh -c 'cd /usr..." 10 minutes ago Up 9 minutes   80/tcp, 443/tcp     trusting_brown
b44dbc1f3d6f   dockersamples/static-site           "/bin/sh -c 'cd /usr..." 11 minutes ago Up 11 minutes   80/tcp, 443/tcp     inspiring_turing
(base) clayton@clayton-note:~$
```

O comando `docker ps` me mostra a utilização da porta 80 e 443 mas, isso ainda não faz com que eu consiga o acesso. Vamos então parar os containers que estão em execução com o **stop** porém, vamos passar mais uma flag, agora o **-t**, que irá agilizar o processo de parada do container que por padrão, vem configurado para aguardar 10 segundos, como podemos ver na imagem abaixo:

**docker**  
**stop -t 0**  
(id do container)

```
al2_fiep@recognarasp-desktop: ~
File Edit View Search Terminal Help
(base) clayton@clayton-note:~$ docker stop --help

Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop one or more running containers

Options:
  -t, --time int    Seconds to wait for stop before killing it (default 10)
(base) clayton@clayton-note:~$
```

Agora  
então

iremos fazer a comunicação de nossa máquina com o servidor criando um *match* utilizando a flag -P, que irá atribuir uma porta aleatória nos permitindo essa comunicação.

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

Com o comando para listar os containers, podemos ver o apontamento das portas entre nossa máquina e os containers:

```
al2_fiep@recognarasp-desktop: ~
File Edit View Search Terminal Help
Options:
  -t, --time int    Seconds to wait for stop before killing it (default 10)
(base) clayton@clayton-note:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS          NAMES
b44dbc1f3d6f   dockersamples/static-site  "/bin/sh -c 'cd /usr..."  About an hour ago  Up About an hour  80/tcp, 443/tcp  inspiring_turing
(base) clayton@clayton-note:~$ docker stop -t 0 b4
b4
(base) clayton@clayton-note:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS          NAMES
02a0672d02a7   dockersamples/static-site  "/bin/sh -c 'cd /usr..."  9 seconds ago  Up 8 seconds  0.0.0.0:49154->80/tcp, :::49154->80/tcp, 0.0.0.0:49153->443/tcp, :::49153->443/tcp  vigilant_wiles
(base) clayton@clayton-note:~$
```

O endereço 0.0.0.0 da minha máquina está lincada com a porta 80 do container e o endereço 0.0.0.0:49153 está lincada com a porta 443 do container. Podemos também listar as portas que estão sendo utilizadas através do comando `docker port`:

```
(base) clayton@clayton-note:~$ docker port 02a0672d02a7
443/tcp -> 0.0.0.0:49153
443/tcp -> :::49153
80/tcp -> 0.0.0.0:49154
80/tcp -> :::49154
```

Agora que temos os endereços, basta ir até o navegador e acessar o localhost de nossa máquina passando como parâmetro o número da porta.



## Hello Docker!

This is being served from a **docker** container running Nginx.



## 2.1. Renomeando Containers

Podemos tornar mais fácil a manutenção de nossos containers renomeando de acordo com sua aplicação, no caso do container para um servidor web, poderíamos então chamá-lo de **meu-site** através da flag **--name**:

```
al2_flep@recognarasp-desktop: ~  
File Edit View Search Terminal Help  
(base) clayton@clayton-note:~$ docker ps -a  
CONTAINER ID   IMAGE          COMMAND                  CREATED    STATUS    PORTS    NAMES  
(base) clayton@clayton-note:~$ docker run -d -P --name meu-site dockersamples/static-site  
7bccf2cca894b24740e9d85b5543230eb8ba8498e10f4b48da7265e7b4ba013b  
(base) clayton@clayton-note:~$ docker ps -a  
CONTAINER ID   IMAGE          COMMAND                  CREATED    STATUS    PORTS    NAMES  
(base) clayton@clayton-note:~$ docker ps -a  
CONTAINER ID   IMAGE          COMMAND                  CREATED    STATUS    PORTS    NAMES  
7bccf2cca894  dockersamples/static-site  "/bin/sh -c 'cd /usr..."  3 seconds ago  Up 2 seconds  0.0.0.0:49158->80/tcp, :::49158->80/tcp, 0.0.0.0:49157->443/tcp, :::49157->443/tcp  meu-site  
(base) clayton@clayton-note:~$
```

Uma outra configuração muito utilizada seria indicar ao container qual porta ele poderia utilizar na sua máquina, atrelando ela junto a porta 80 através da flag **-p** (minúsculo):

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

É possível também setarmos uma variável de ambiente no momento da criação do nosso container, o que irá resultar em uma (ou mais) mensagem no nosso servidor:

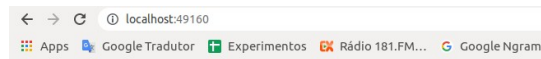


```
(base) clayton@clayton-note:~$ docker run -d -P --name autor-site -e AUTHOR="Prof. Clayton" dockersamples/static-site
44d5d1ca4072298b39f5ff7ba2c25bbd0a1fd8068effcabfaa6178a8a7fbd7ee
(base) clayton@clayton-note:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
44d5d1ca4072	dockersamples/static-site	"/bin/sh -c 'cd /usr..."	5 seconds ago	Up 4 seconds	0.0.0.0:49160->80/tcp, :::49160->80/tcp, 0.0.0.0:49159->443/tcp, :::49159->443/tcp	autor-site
7bccf2cca894	dockersamples/static-site	"/bin/sh -c 'cd /usr..."	8 minutes ago	Exited (137) About a minute ago		meu-site

```
(base) clayton@clayton-note:~$
```

Acessando agora novamente através via *browser*, podemos então verificar a mensagem no container:



## Hello Prof. Clayton!

This is being served from a **docker** container running Nginx.

Para finalizar essa nossa lição, vamos parar os containers que estão em execução, para listá-los, já sabemos como fazer mas e para parar todos de uma só vez?

```
ai2_flep@recognarasp-desktop: ~
(base) clayton@clayton-note:~$ echo "Listando os containers pelo id"
Listando os containers pelo id
(base) clayton@clayton-note:~$ docker ps -q
44d5d1ca4072
(base) clayton@clayton-note:~$ docker stop -t 0 $(docker ps -q)
44d5d1ca4072
(base) clayton@clayton-note:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
44d5d1ca4072	dockersamples/static-site	"/bin/sh -c 'cd /usr..."	6 minutes ago	Exited (137) 6 seconds ago		autor-site
7bccf2cca894	dockersamples/static-site	"/bin/sh -c 'cd /usr..."	15 minutes ago	Exited (137) 8 minutes ago		meu-site

```
(base) clayton@clayton-note:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS   NAMES
(base) clayton@clayton-note:~$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
44d5d1ca4072298b39f5ff7ba2c25bbd0a1fd8068effcabfaa6178a8a7fbd7ee
7bccf2cca894b24740e9d85b5543230eb8ba8498e10f4b48da7265e7b4ba013b

Total reclaimed space: 17.51kB
(base) clayton@clayton-note:~$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS   NAMES
```

## 2.2. Resumo 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.

### 3. Volumes de Dados

Como vimos até agora, containers são apenas uma pequena camada de leitura e escrita que funcionam sobre as imagens e que não podem ser modificadas pois estas, são apenas para leitura. A partir do momento que removemos nosso container, essa camada de leitura e escrita também é removida, o que faz com que todos os nossos dados também sejam removidos, já que podemos ter dados importantes nesses containers.

Buscando resolver essa volatilidade dos containers, já que foram desenvolvidos para serem criados e removidos rápido e facilmente, um local conhecido como **volumes de dados** pode ser desenvolvido e utilizado.

Esse processo seria como se estivéssemos criando uma nova pasta dentro do **docker host**, onde todos os dados que foram adicionados ali serão mantidos. Vamos verificar como isso é feito então no nosso terminal.

Criando um novo container com a imagem do ubuntu, vamos agora passar a flag **-v**, que será a responsável por informar o docker que no momento da criação desse container, será necessário também a criação de um diretório **var/www**:

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

Em seguida, podemos verificar se essa atividade realmente foi realizada através do inspect. Após aplicar esse comando, podemos verificar no conteúdo gerado que diversas informações são apresentadas porém, para essa aplicação iremos nos atentar ao conteúdo apresentado em **Mounts**.

```

File Edit View Search Terminal Help
ali_fisp@recognitionasp-desktop:~
(base) clayton@clayton-note:~$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
(base) clayton@clayton-note:~$ clear
(base) clayton@clayton-note:~$ docker run -v "/var/www" ubuntu
(base) clayton@clayton-note:~$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
6e1628f0e6e   ubuntu   "bash"    5 seconds ago    Exited (0) 4 seconds ago    beautiful_lehmann
(base) clayton@clayton-note:~$ docker inspect 6e
[
  {
    "Id": "6e1628f0e6e6a29d619ed65416d9e9e07b03c662409aae5483ad0e57731931",
    "Created": "2021-10-04T18:23:45.314813276Z",
    "Path": "bash",
    "Args": [],
    "State": {
      "Status": "exited",
      "Running": false,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 0,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2021-10-04T18:23:45.866012422Z",

```

## **docker inspect** (id do container)

```
"Mounts": [
  {
    "Type": "volume",
    "Name": "b7a044586a288bf55ad7b7622e0810e02d91f0c2b97549082aa938a546c0df70",
    "Source": "/var/lib/docker/volumes/b7a044586a288bf55ad7b7622e0810e02d91f0c2b97549082aa938a546c0df70/_data",
    "Destination": "/var/www",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
],
```

É possível verificarmos em **Source** e **Destination** o diretório criado e seu *path* na máquina local. Podemos verificar que esse caminho apresentado pelo docker não é dos mais simpáticos rs então, ele nos possibilita alterar isso também, passando agora o *path* desejado da seguinte forma:

```
root@4adba55117bd: /var/www
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente$ docker run -it --name souce-site -v "/home/clayton/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/:/var/www" ubuntu
root@4adba55117bd:/# ls
bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run/sbin srv sys usr var
root@4adba55117bd:/# cd /var/www/
root@4adba55117bd:/var/www# touch file_fiep01.txt
root@4adba55117bd:/var/www# echo "Arquivo criado dentro de um volume no Docker host" > file_fiep01.txt
root@4adba55117bd:/var/www#
```

Vários parâmetros e flags foram passados na execução dessa tarefa, em seguida, acessamos o diretório local através do container e criamos um novo arquivo, escrevemos algo para comprovar essa transição e salvamos no arquivo criado. Assim sendo, podemos verificar que criar um volume é associar um diretório local com nosso container, fazendo com que no momento que este container seja destruído, o seu volume com suas informações fique a salvo no diretório local através do docker host. Assim sendo, porque usamos os volumes?

Justamente pelo simples fato de estarmos constantemente criando e removendo containers após seu uso, dessa forma, os dados também seriam removidos, o que não acontece quando executamos a tarefa de criação de volumes.

## **3.1. Fazendo o caminho inverso**

Da mesma forma que podemos escrever dentro de um diretório em um container e salvar os dados no docker host na nossa máquina, podemos também fazer o caminho inverso, ou seja, criando na nossa máquina e fazendo com que o arquivo seja salvo no container. Isso é importante devido ao fato de eu precisar de alguma aplicação ou biblioteca e não ter instalado em minha máquina então, posso criar um container que possua as características necessária para execução de um código e colocar para rodar dentro dele, ou seja, o ambiente de desenvolvimento fica completamente dentro do container.

Então, se um *container* possui Node, Java, PHP, seja qual for a linguagem, não precisamos tê-los instalados na nossa máquina, nosso ambiente de desenvolvimento será dentro do próprio container.

Para realizarmos essa tarefa teremos que utilizar algumas flags como **-d** que irá evitar congelar nosso terminal, **-p** para indicarmos a porta a ser utilizada pelo localhost, **-v** que irá conter o path (mapeamento do volume) do diretório onde será salvo nosso volume e por fim **-w**, (*Working Directory*), para dizer em qual diretório o comando deve ser executado, a pasta **/var/www** e por fim o pacote **Node**, necessário para a execução de nosso código html chamando o parâmetro **npm start** para iniciar o container.

```
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo$ docker run -p 8080:3000 -v "/home/clayton/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo:/var/www" -w "/var/www" node npm start
> volume-exemplo@1.0.0 start
> node .
Server is listening on port 3000
```

O docker me responde que está ouvindo na porta 3000 então, podemos verificar nossa aplicação rodando no *browser* através do redirecionamento em: <http://localhost:8080/>. Esta então é a grande vantagem da utilização do Docker, importando uma imagem, posso eu e minha equipe trabalhar no mesmo ambiente sem a necessidade de controle de versão de bibliotecas e pacotes, estando todos trabalhando na mesma versão.

Melhorando ainda mais esse comando, podemos passar outra flag que irá reduzir o *path* através da técnica de interpolação de comandos, ou seja, passo um valor chamando outro, no caso o parâmetro será o **pwd** através da flag **-d**:

```
Terminal
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo$ docker run -d -p 8080:3000 -v "${pwd}:/var/www" -w "/var/www" node npm start
38289cb579ff952600313b1367ab72e3a4d867764685ec3a0b1968d9bf2249f9
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
38289cb579ff   node     "docker-entrypoint.s..." 6 seconds ago  Up 5 seconds  0.0.0.0:8080->3000/tcp, :::8080->3000/tcp  loving_boyd
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo$
```

Com o comando abaixo, podemos verificar também o código em html desenvolvido para a navegação e integração com o node.

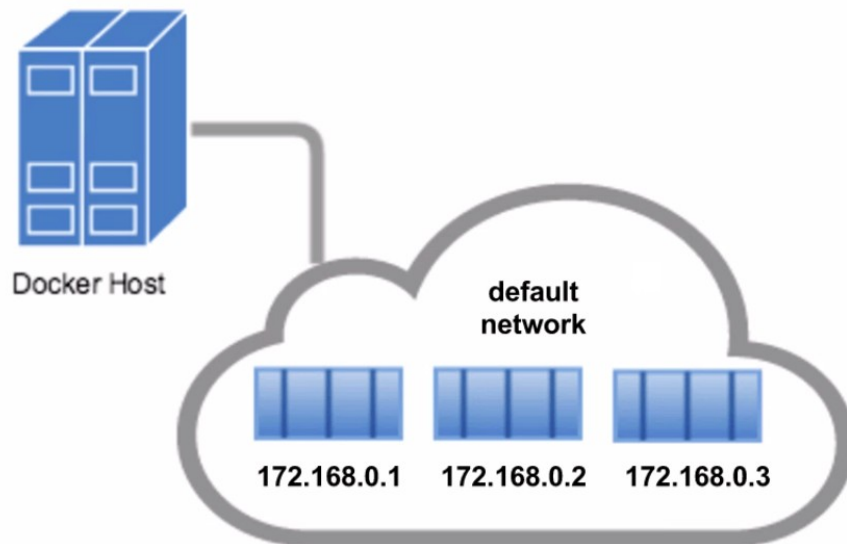
**curl http://localhost:8080**

Podemos acessar também os logs do nosso container através do comando:

**docker container logs (id do container)**

## 4. Redes com Containers

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.



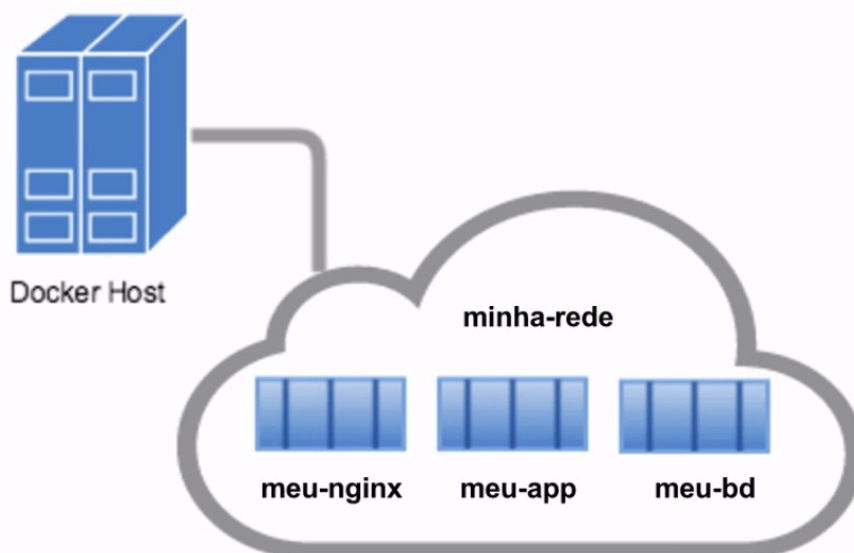
## 4.1. Comunicação entre containers utilizando os seus nomes

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.

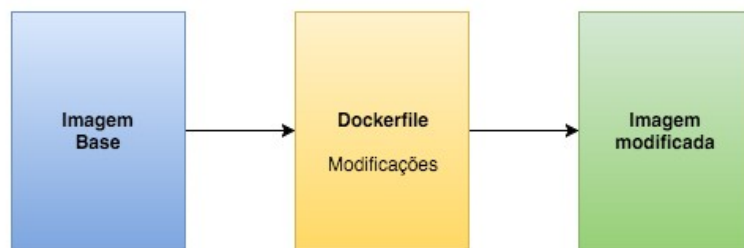
Na rede padrão do Docker, só podemos realizar a comunicação utilizando IPs, mas se criarmos a nossa própria rede, podemos "batizar" os nossos containers, e realizar a comunicação entre eles utilizando os seus nomes:



## 4.2. Criando nossas próprias imagens

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. Sabemos que uma imagem é como se fosse uma “receita de bolo”, onde temos que seguir alguns passos para obter sucesso na sua criação então, para criarmos nossa própria imagem, teremos primeiro que criar essa receita e isso é realizado através do **Dockerfile**.

Então, quando se utiliza **Dockerfile** para gerar uma imagem, basicamente, é apresentada uma lista de instruções que serão aplicadas em determinada imagem para que outra imagem seja gerada com base nas modificações.



### 4.2.1 Montando o **Dockerfile**

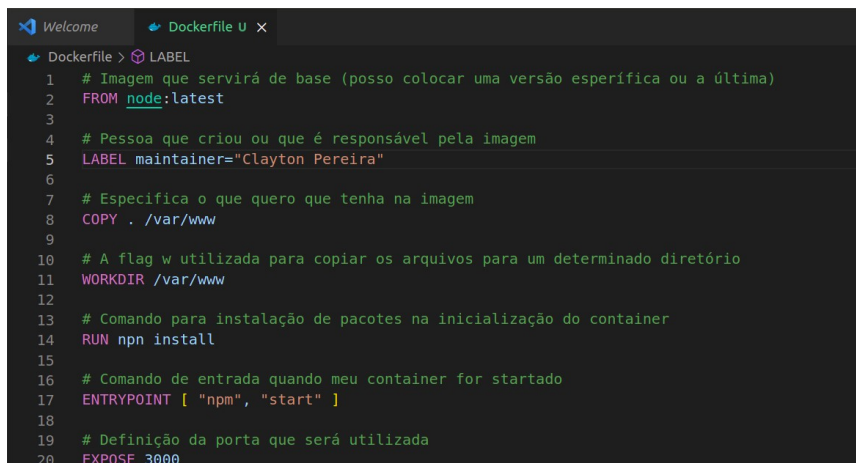
Então, no nosso projeto, devemos criar o arquivo **Dockerfile**, que nada mais é do que um arquivo de texto. Ele pode ter qualquer nome, porém nesse caso ele também deve possuir a extensão **.dockerfile**, por exemplo **node.dockerfile**, mas vamos manter o nome padrão mesmo **Dockerfile**, já que temos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

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.

Baseado no comando que utilizamos na aula que criamos o **Node**, vamos agora criar nosso **Dockerfile** seguindo o mesmo comando:

```
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo$ docker run -p 8080:3000 -v "/home/clayton/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/volume-exemplo:/var/www" -w "/var/www" node npm start
> volume-exemplo@1.0.0 start
> node .
Server is listening on port 3000
```

A imagem abaixo apresenta o código texto desenvolvido para realizar a mesma tarefa através do **Dockerfile**:



```

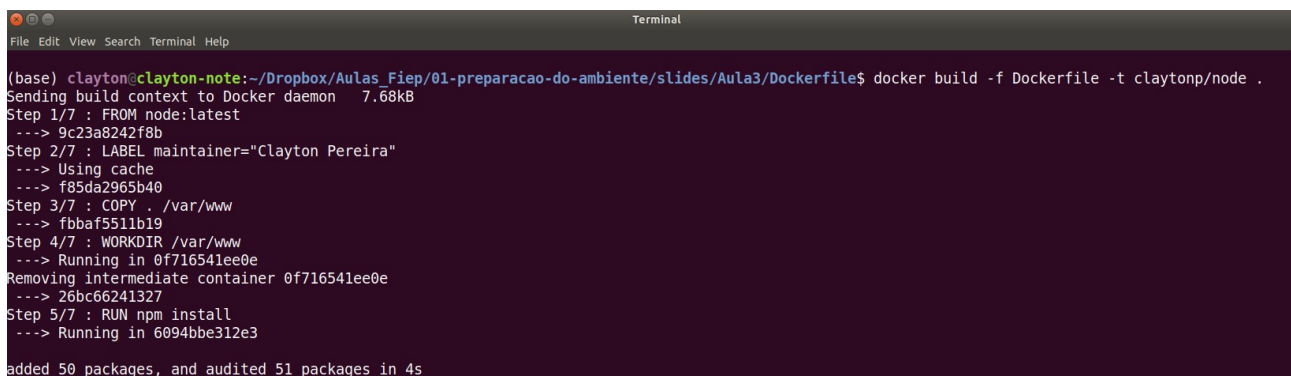
1  # Imagem que servirá de base (posso colocar uma versão específica ou a última)
2  FROM node:latest
3
4  # Pessoa que criou ou que é responsável pela imagem
5  LABEL maintainer="Clayton Pereira"
6
7  # Especifica o que quero que tenha na imagem
8  COPY . /var/www
9
10 # A flag w utilizada para copiar os arquivos para um determinado diretório
11 WORKDIR /var/www
12
13 # Comando para instalação de pacotes na inicialização do container
14 RUN npm install
15
16 # Comando de entrada quando meu container for startado
17 ENTRYPOINT [ "npm", "start" ]
18
19 # Definição da porta que será utilizada
20 EXPOSE 3000

```

Após termos realizado as alterações necessárias, hora de “*Buildar*” nossa imagem através do comando ***docker build***. Esse comando irá criar a nossa imagem através do ***Dockerfile*** que acabamos de criar.

Como o nome do nosso ***Dockerfile*** é o padrão, poderíamos omitir a *flag -f*, 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 e o seu nome através da *flag -t*. Para imagens não-oficiais, colocamos o nome no padrão **NOME\_DO\_USUARIO/NOME\_DA\_IMAGEM**, foi isso que fizemos na imagem apresentada abaixo. Por último, especificamos o diretório que se encontra nosso ***Dockerfile***, como já estamos no diretório colocamos apenas o ponto (.).



```

(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$ docker build -f Dockerfile -t claytonp/node .
Sending build context to Docker daemon  7.68kB
Step 1/7 : FROM node:latest
--> 9c23a8242f8b
Step 2/7 : LABEL maintainer="Clayton Pereira"
--> Using cache
--> f85da2965b40
Step 3/7 : COPY . /var/www
--> fbbaf5511b19
Step 4/7 : WORKDIR /var/www
--> Running in 0f716541ee0e
Removing intermediate container 0f716541ee0e
--> 26bc66241327
Step 5/7 : RUN npm install
--> Running in 6094bbe312e3
added 50 packages, and audited 51 packages in 4s

```

Ao executarmos os 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***.

Dessa forma, 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.

## 4.2.2 Criando um container a partir da imagem criada



Agora que já temos nossa imagem criada, podemos verificá-la através do comando **docker images** e então, damos início a criação do nosso container.

```
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED          STATUS          PORTS          NAMES
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$ docker run -d -p 8080:3000 claytonp/node
ac219bda877022fd1ecc6454c3ff2ab12f5bc1e8a4796cc5283b124959f1d08e
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED          STATUS          PORTS          NAMES
ac219bda8770   claytonp/node                       "npm start"             22 seconds ago   Up 21 seconds   0.0.0.0:8080->3000/tcp, :::8080->3000/tcp   relaxed_leakey
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$
```

Apenas realizamos o direcionamento da porta e passamos agora, o nome da imagem que acabamos de criar. Após isso, já será possível visualizar nossa imagem em <http://localhost:8080/> . Caso precisemos realizar qualquer tipo de alteração no código ou na nossa imagem, será necessário realizar o *build* novamente e construir um novo container através da imagem gerada do **Dockerfile**.

### 4.2.3 Disponibilizando nossa imagem no *Docker Hub*

Agora que criamos nossa imagem, podemos deixá-la publica e disponibilizar para a comunidade através do **Docker Hub**. Vamos criar uma conta no site: <https://hub.docker.com/>, após isso, efetuamos login da nossa conta no terminal através do usuário e senha cadastrado no site. Após isso, já podemos subir nossa imagem através do comando **push**.

```
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$ docker push claytondad/node
Using default tag: latest
The push refers to repository [docker.io/claytondad/node]
cf05b3896798: Pushed
74b4a4e78019: Pushed
311271778acb: Mounted from library/node
3fecbd1300ea: Mounted from library/node
443d37436905: Mounted from library/node
1d59188d1ae5: Mounted from library/node
6664df8b907f: Mounted from library/node
193c69a58521: Mounted from library/node
65bd1a7ee0f5: Mounted from library/node
e637b6ee6754: Mounted from library/node
22f8b5520ced: Mounted from library/node
latest: digest: sha256:c5614dc3774459ce927d59ef273eac168d368eb076086797f55a0f9f5e5a5495 size: 2633
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Dockerfile$
```

Efetuada o *upload* da imagem, ela agora está publica e pode ser baixada por qualquer pessoa através do comando:

**docker pull** (nome do proprietário/imagem)

Para a minha imagem o comando ficou:

**docker pull** claytondad/node

```
clayton@jesus3:~$ sudo docker pull claytondad/node:latest
latest: Pulling from claytondad/node
5e7b6b7bd506: Pull complete
fd67d668d691: Pull complete
1ae016bc2687: Pull complete
0b0af05a4d86: Pull complete
ca4689f0588c: Pull complete
8c33de21d690: Pull complete
f113b2c481db: Pull complete
0f84649efc4d: Pull complete
5990cbd9430a: Pull complete
96cd686809bb: Pull complete
8523ed2eee95: Pull complete
Digest: sha256:c5614dc3774459ce927d59ef273eac168d368eb076086797f55a0f9f5e5a5495
Status: Downloaded newer image for claytondad/node:latest
docker.io/claytondad/node:latest
```



## 4.2.4 Criando um container com Apache

Vamos agora criar uma imagem contendo uma distro Linux Debian e utilizar como um servidor Web através do Apache, para isso, temos que criar um novo **Dockerfile** contendo essas instruções.

```
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Apache$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
claytondad/server21  latest         c3f21df6f06c   13 seconds ago 252MB
claytonp/node        latest         38f58705994d   3 hours ago   913MB
claytondad/node      latest         38f58705994d   3 hours ago   913MB
node                 latest         9c23a8242f8b   10 days ago   908MB
debian               latest         a178460bae57   10 days ago   124MB
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Apache$ docker run -it claytondad/server21
root@562e04b3e62e:/# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root         1      0   0 14:41 pts/0        00:00:00 bash
root        10      1   0 14:43 pts/0        00:00:00 ps -ef
root@562e04b3e62e:/# /etc/init.d/apache2 start
```

Através de alguns comandos, podemos iniciar os processos de inicialização do apache:

- ps -ef:** Lista os serviços em execução;
- ss -s:** Apresenta as portas que estão sendo usadas;
- ss -a:** Apresenta as postas ativas e seus serviços;

Listando o endereço IP do servidor:

```
root@562e04b3e62e:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
63: eth0@if64: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
root@562e04b3e62e:/#
```

Com o telnet + endereço IP, posso verificar se tudo está funcionando corretamente:

```
(base) clayton@clayton-note:~/Dropbox/Aulas_Fiep/01-preparacao-do-ambiente/slides/Aula3/Apache$ telnet 172.17.0.3 80
Trying 172.17.0.3...
Connected to 172.17.0.3.
Escape character is '^]'.
quit
```

## 5. Trabalho Final Docker Container

- i.** Conforme mencionado em aula, encaminhe o path de sua imagem armazenada no Docker Hub (Vale ressaltar que não precisa ser exatamente a imagem da forma que fizemos em aula, quanto maior a dificuldade diferença, acredito ser melhor ainda para o aprendizado).
- ii.** Em equipe, desenvolva uma imagem que represente o aprendizado de vocês durante essa primeira etapa, essa imagem pode ser informações sobre o curso, sobre o grupo, sobre o Hub enfim, vale a imaginação. :-)
- iii.** A imagem desenvolvida pela equipe deve ser alocada no Docker Hub e encaminhada via e-mail contendo os integrantes da equipe para o endereço: [clayton.pereira@unesp.br](mailto:clayton.pereira@unesp.br).

Gostaria também de parabenizá-los pela dedicação e esforço em aprender todo esse conteúdo, pode parecer difícil agora porém, lá na frente verão o resultado e outro detalhe, se fosse fácil... qualquer um fazia :-)

Bom trabalho a todos!!!

Abraços.  
08/10/2021