

Conceitos de docker

OBS: Todo o ambiente já havia sido configurado anteriormente, realizado a instalação do docker e docker-compose.

OBS 2: Nos testes de ping dentro dos containers foi realizado a instalação da ferramenta ping.

*apt install iputils-ping -y

Namespace é o responsável por garantir a separação dos containers dentro como processos. Os principais namespaces que podemos citar são esses:

- PID : Provê isolamento dos processos rodando dentro do container
- NET : Provê isolamento das interfaces de rede
- IPC : Provê isolamento da comunicação entre processos e memória compartilhada
- MNT : Provê isolamento do sistema de arquivos / Ponto de montagem
- UTS : Provê isolamento do kernel. Age como se o container fosse outro host.

Para realizar o gerenciamento de consumo de cada container utilizamos o CGROUPS, esse serviço pode definir o consumo de processamento, memória e armazenamento de cada container, sendo de forma manual ou automática

Comandos iniciais do docker

Docker run → Executa um container

Inicialmente o comando run procura a imagem localmente, caso a imagem não exista no localhost, é baixado no dockerhub, é feito a validação da imagem por hash, e após isso a imagem é executada.

docker ps ou docker container ls → Verifica os containers em execução

```
viniciusfigueiraspop-os:~$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
0e2f7a8a4580   dockersamples/static-site          "/bin/sh -c 'cd /usr..." 7 minutes ago  Up 7 minutes  443/tcp, 0.0.0.0:8080->80/tcp, :::8080->80/tcp  nervous_ishizaka
viniciusfigueiraspop-os:~$
```

docker stop "id_container" – Para o container

docker start "id_container" – Inicia o container novamente

docker exec -it "id_container" bash – Acessa o container e consegue executar comandos no container

docker pause "id_container" – Pausa o container

docker unpause "id_container" – Despausa o container

docker rm "id_container" → Realiza a remoção do container, se realizar a inicialização da mesma imagem, irá subir outro container tem ter os dados do container que foi removido

docker port "id_container" → Mostra as portas do container e a porta de saída da aplicação

docker run -p 8080:80: o parâmetro -p [MINUSCULO] indica a porta de saída da aplicação do nosso local hosto com a porta de saída do container. Neste exemplo a nossa porta de acesso para aplicação é a porta 8080, e a do container está na 80.

```
viniciusfigueira@pop-os:~$ docker run -d -p 8080:80 dockersamples/static-site
0e2f7a8a458047da18d86bf67644cc126fa4dbd8cd37edafce6bb5907a0927a5
viniciusfigueira@pop-os:~$
```

docker run -P: o parâmetro -P [MAIÚSCULO } define uma porta de saída “default”, que conseguimos visualizar através do comando “docker port “id_container”.

```
viniciusfigueira@pop-os:~$ docker run -d -P dockersamples/static-site
a834bb3adb8a44200d714e89b4fcdc601ed1a1b26fe64c9a3747f1dcab9d29cd
viniciusfigueira@pop-os:~$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED
a834bb3adb8a   dockersamples/static-site          "/bin/sh -c 'cd /usr... 8 seconds ago
l_vaghan
viniciusfigueira@pop-os:~$ docker port a834bb3adb8a
443/tcp -> 0.0.0.0:49153
443/tcp -> :::49153
80/tcp -> 0.0.0.0:49154
80/tcp -> :::49154
```

*Basicamente o parâmetro -p / -P realiza o mapeamento de portas entre o host e o container.

Docker login -u “usuario” → Realiza o login no docker hub

```
viniciusfigueira@pop-os:~/Desktop/Imagem-Docker$ docker login -u viniciusfigueira
Password:
WARNING! Your password will be stored unencrypted in /home/viniciusfigueira/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

.docker push “nome imagem” → Sobe a imagem para o dockerhub

```
viniciusfigueira@pop-os:~/Desktop/Imagem-Docker$ docker push viniciusfigueira/app-node:1.2
The push refers to repository [docker.io/viniciusfigueira/app-node]
f3e055d4fdb: Pushed
406fad3729aa: Pushed
6583fac223f4: Pushed
fd74f45fc6f0: Mounted from library/node
52fd0dede527: Mounted from library/node
ec7d61405005: Mounted from library/node
95904c181913: Mounted from library/node
df69bfa94785: Mounted from library/node
f35deb8d96fc: Mounted from library/node
f6c2459e2059: Mounted from library/node
f8323fb3a55c: Mounted from library/node
2f4dc9775f33: Mounted from library/node
1.2: digest: sha256:5c0181fbb117027418f35a51d94b2aedd7943550f3891064fbbf8b82eed9ada3 size: 2839
viniciusfigueira@pop-os:~/Desktop/Imagem-Docker$
```

docker stop \$(docker container ls -q) → Para todos os containers em execução

docker rm \$(docker container ls -aq) → Remove todos os containers, inclusive os parados

```
viniciusfigueira@pop-os:~/Desktop/Imagem-Docker$ docker rm $(docker container ls -aq)
f282b88ed027
c60064de0874
0e2f7a8a4580
d81314111c4f
16e704e60b0a
f599bd6f2437
```

docker rmi \$(docker image ls -aq) –force → Realiza a exclusão de todas imagens.

```

viniciusfigueira@pop-os:~$ docker rmi $(docker image ls -qa) --force
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:9101220a875cee98b016668342c489ff0674f247f6ca20dfc91b91c0f28581ae
Deleted: sha256:825d55fb6340083b06e69e02e823a02918f3ffb575ed2a87026d4645a7fd9e1b
Deleted: sha256:c5ec52c98b3193052e15d783aca2bef10d8d829fa0d58fedfede511920b8f997
Untagged: mongo:4.4.6
Untagged: mongo@sha256:6efa052039903e731e4a5550c68a13c4869ddc93742c716332883fd9c77eb79b
Deleted: sha256:61ea24dc52c6396df1a77f7e1a00c1019b4dd7873228f1916de40628677e8824
Deleted: sha256:d15f19a1c02bc207931e16bc7b1eea7e6fa8b3455b97baa9b644cdd0810eb0d9
Deleted: sha256:3eb55acaa76ab743fef1e51751ffadb309833bc14ba8f797a99f6cb8053a470d
Deleted: sha256:8a12f2aa23bd64f99b0c85fdf892559efec71cb8e3f9413cdd1ee9b7538d7620
Deleted: sha256:8af39b762ba05b5e27e50b2ea833b15222ca13dc4fcb49db1ef75c17f160ab2a
Deleted: sha256:681411fe07a6756775291433d433106ea249f4a7fe843fcbc764b1d314de0fce
Deleted: sha256:410101431e8ceb9d83396c898cd06a01956cc05e9fb850da6239aa367b432f1a
Deleted: sha256:1003a8d51d08121d9f5878d38e2017afc81e934cc435b02e6e8fcf755070ada9
Deleted: sha256:6268c0171db866453faff20b5ed263bb0355eb02dbc4aa7591f50280c484274c
Deleted: sha256:04e7cd90709321f7df0acc2128629778e6905b0be1c26110ae546283aadb3ab1
Deleted: sha256:878dab86cf0f5ad62033738e93d064580c63a6a2140badccddc8e7bddd45fd94
Untagged: aluradocker/alura-books:1.0
Untagged: aluradocker/alura-books@sha256:ed754b9994539e1b9114ecd8686e2e09fdcf5e6f1aefd71c6e7cdc63afeb08
Deleted: sha256:ebff169e5013a6457e39386df10afeb86d2af6608f47f66df194fe4659171a80
Deleted: sha256:5b9ceff42bb90583ec9ba4807a3bd710bdb1ebd278d13da9d12f2a6253f9840e
Deleted: sha256:02e6ffa8f55930fb17d576f35f946210e14e7a4b4488c4f2ace2aa9ccbdb0c72
Deleted: sha256:d2155e693dfe8acf50232721bd3605dd9a8acc03212897d5b3be12b137755d6e
Deleted: sha256:8bdb068caeadd20faeb6b8dc363e50ca6ba944a7130880f15fed03e278399c10
Deleted: sha256:84e040a3b39dd8533febf65709350ef282b5f8edb3a2d3e20fe41f626841166e
Deleted: sha256:48aede9d5b2dd3ed037ce61b66e64bb398b69a342aaa191ad02cd00204f97281
Deleted: sha256:fd842df589c9dce7e7dfc5d45daf7ea907367642ddb8ca9eaa4d3c729fa9e912
Deleted: sha256:63102ff996737f31e062250371c0dd77387910c5c146496600b4cfb20db5a90d
Deleted: sha256:5d6bba18f7b25c9b93d3cc0d93a4cff54eb88b0ba22ed867633a21fc3ded5f57
Deleted: sha256:2c40c66f7667aefbb18f7070cf52fae7abbe9b66e49b4e1fd740544e7ceae8dc
viniciusfigueira@pop-os:~$ docker images
REPOSITORY      TAG                IMAGE ID           CREATED            SIZE

```

Nos 3 comandos acima os parâmetros significam

-q : Selecionar todos os id

-a: Selecionar os containers e imagens também parados

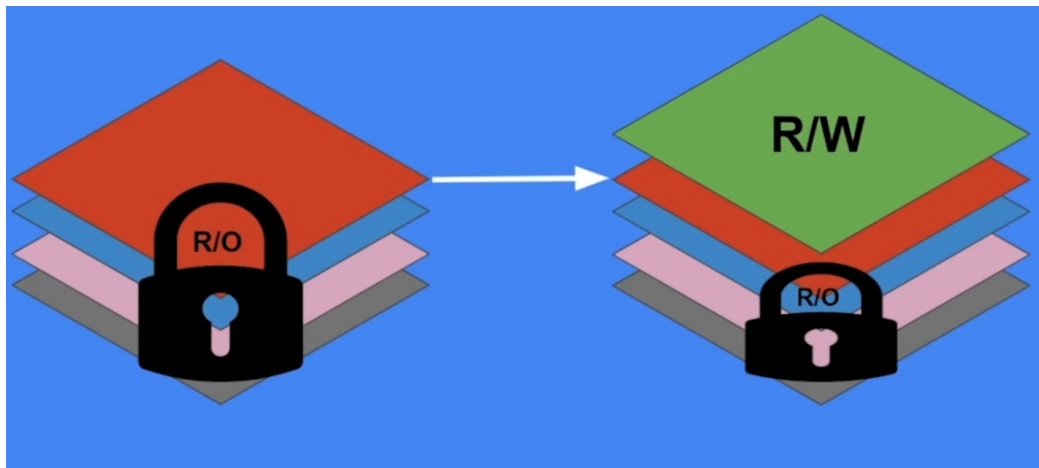
--force : Forçar a exclusão dos arquivos em caso de conflito

Imagens de containers

Imagem é um conjunto de camadas, que quando são juntadas dão origem a uma imagem.

Lembrando que cada camada tem um identificador. Então quando vamos baixar uma imagem no docker hub, realizamos o download das camadas que formam a image. Se por acaso, tivermos alguma imagem que já tenha alguma dessas camadas, é feito o download apenas das camadas restantes, evitando dados duplicados no nosso host.

Após realizar a criação de uma imagem ela sempre será R/O(Read Only) não sendo possível alterar os dados dessas imagens. Após a execução dessa imagem todos os dados criados são salvos em uma camada RW que não faz parte da imagem original, esses dados podem ser armazenados em um local reservado, ou até mesmo de forma temporária, com a morte desse container, esses dados são perdidos.



Como podemos observar a primeira representação é uma imagem concluída, totalmente read-only. A segunda representação mostra um container em execução, com a imagem ainda em R/O e uma camada adicional R/W onde permite a adição de dados sem interferir na imagem.

`docker images` ou `docker image ls` → Conseguimos visualizar as imagens que temos baixadas.

```
viniciusfigueira@pop-os:~$ docker images
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
ubuntu              latest      825d55fb6340  2 weeks ago  72.8MB
hello-world         latest      feb5d9fea6a5  6 months ago  13.3kB
dockersamples/static-site latest      f589ccde7957  6 years ago  191MB
viniciusfigueira@pop-os:~$
```

`docker inspect "image_id"` → Conseguimos pegar informações detalhadas sobre a imagem

`docker history "image_id"` → Conseguimos visualizar as camadas dessa imagem, que quando são unidas da forma a imagem completa.

```
viniciusfigueira@pop-os:~$ docker history f589ccde7957
IMAGE          CREATED          CREATED BY                                      SIZE
f589ccde7957   6 years ago     /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "cd /u... 0B
<missing>      6 years ago     /bin/sh -c #(nop) WORKDIR /usr/share/nginx/h... 0B
<missing>      6 years ago     /bin/sh -c #(nop) COPY file:c8203f6bfe2ff6ba... 8.75kB
<missing>      6 years ago     /bin/sh -c mkdir -p /usr/share/nginx/html      0B
<missing>      6 years ago     /bin/sh -c #(nop) ENV AUTHOR=Docker            0B
<missing>      6 years ago     /bin/sh -c #(nop) CMD ["nginx" "-g" "daemon ... 0B
<missing>      6 years ago     /bin/sh -c #(nop) EXPOSE 443/tcp 80/tcp        0B
<missing>      6 years ago     /bin/sh -c ln -sf /dev/stdout /var/log/nginx... 22B
<missing>      6 years ago     /bin/sh -c apt-key adv --keyserver hkp://pgp... 65.4MB
<missing>      6 years ago     /bin/sh -c #(nop) ENV NGINX_VERSION=1.9.12-1... 0B
<missing>      6 years ago     /bin/sh -c #(nop) MAINTAINER NGINX Docker Ma... 0B
<missing>      6 years ago     /bin/sh -c #(nop) CMD ["/bin/bash"]           0B
<missing>      6 years ago     /bin/sh -c #(nop) ADD file:b5391cb13172fb513... 125MB
viniciusfigueira@pop-os:~$
```

Para realizar a criação da imagem utilizamos o comando

`docker build t "nome imagem": "versão" "diretório"`

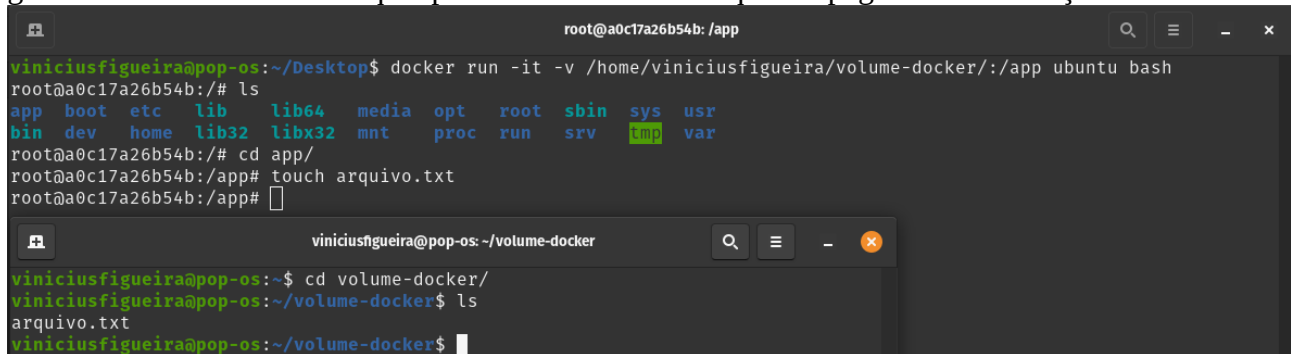
ex: `docker build t viniciusfigueira/appnode .`

Onde -t significa o tagueamento de nome e o "." significa o diretório local

Persistindo dados

Para realizar a persistência de dados gravados em um container, são utilizados 3 formatos, sendo eles: bind mounts, volumes e tmpfs mount.

Utilizando o bind mount, podemos definir o diretório de gravação com o parâmetro `-v` na execução do container, seguindo o exemplo abaixo passamos o parâmetro, definimos o diretório onde ficará gravado em nosso host e de qual pasta do container tem que ser pego essa informação.



The image shows two terminal windows. The first window, titled 'root@a0c17a26b54b: /app', shows the execution of 'docker run -it -v /home/viniciusfigueira/volume-docker/:/app ubuntu bash'. Inside the container, the user runs 'ls' showing a directory listing, then 'cd app/' and 'touch arquivo.txt'. The second window, titled 'viniciusfigueira@pop-os: ~/volume-docker', shows the user running 'cd volume-docker/' and 'ls', which displays 'arquivo.txt'.

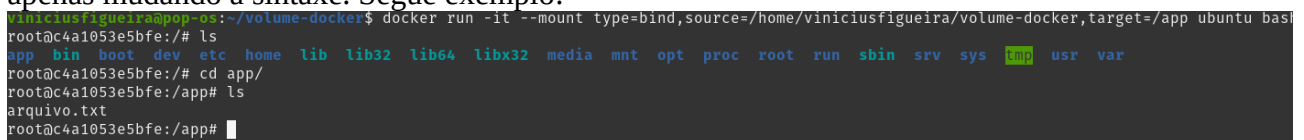
```
root@a0c17a26b54b: /app
viniciusfigueira@pop-os:~/Desktop$ docker run -it -v /home/viniciusfigueira/volume-docker/:/app ubuntu bash
root@a0c17a26b54b:/# ls
app  boot  etc  lib  lib64  media  opt  root  sbin  sys  usr
bin  dev  home  lib32  libx32  mnt  proc  run  srv  tmp  var
root@a0c17a26b54b:/# cd app/
root@a0c17a26b54b:/app# touch arquivo.txt
root@a0c17a26b54b:/app#

viniciusfigueira@pop-os: ~/volume-docker
viniciusfigueira@pop-os:~$ cd volume-docker/
viniciusfigueira@pop-os:~/volume-docker$ ls
arquivo.txt
viniciusfigueira@pop-os:~/volume-docker$
```

No primeiro terminal estamos acessando o container e criando um arquivo no diretório `/app`. Esses dados estão sendo salvos no diretório `home/viniciusfigueira/volume-docker` de nosso host. Como mostra o segundo terminal.

Neste caso se pararmos esse container e executarmos outro container com o mesmo parametro e diretório iremos continuar acessando os dados pelo segundo container.

Temos outra opção de fazer, utilizando o parâmetro `--mount`, inclusive recomendado nas documentações do docker por ser mais semântico. Funciona da mesma forma que o parâmetro `-v` apenas mudando a sintaxe. Segue exemplo:



The image shows a terminal window titled 'root@c4a1053e5bfe: /app'. The command 'docker run -it --mount type=bind,source=/home/viniciusfigueira/volume-docker,target=/app ubuntu bash' is executed. Inside the container, the user runs 'ls', 'cd app/', and 'ls', which displays 'arquivo.txt'.

```
viniciusfigueira@pop-os:~/volume-docker$ docker run -it --mount type=bind,source=/home/viniciusfigueira/volume-docker,target=/app ubuntu bash
root@c4a1053e5bfe:/# ls
app  bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@c4a1053e5bfe:/# cd app/
root@c4a1053e5bfe:/app# ls
arquivo.txt
root@c4a1053e5bfe:/app#
```

Agora partimos para a segunda opção, que seria utilizando volumes, que é a alternativa mais recomendado pelo docker para utilização em ambientes produtivos, pois se trata de um local de armazenamento gerenciado pelo docker no próprio host. Sendo assim um jeito mais seguro de armazenar os dados e evitar possíveis alterações nesses diretórios.

Para realizarmos a criação de um volume realizamos o seguinte comando:

`docker volume create "nome-do-volume"`

`docker volume ls` → Conseguimos consultar nossos volumes criados.

```

viniciusfigueira@pop-os:~/volume-docker$ docker volume ls
DRIVER      VOLUME NAME
viniciusfigueira@pop-os:~/volume-docker$ docker volume create meu-volume
meu-volume
viniciusfigueira@pop-os:~/volume-docker$ docker volume ls
DRIVER      VOLUME NAME
local       meu-volume

```

Para executarmos um container neste volume, utilizamos o parâmetro -v apontando para o volume criado ao invés de algum diretório. Lembrando que por ser um ponto de armazenamento diferente não irá ter os arquivos criados anteriormente no diretório informado manualmente

`docker run -it -v meu-volume:/app ubuntu bash`

```

viniciusfigueira@pop-os:~$ docker run -it -v meu-volume:/app ubuntu bash
root@250c1e159ab7:/# ls
app  boot  etc  lib  lib64  media  opt  root  sbin  sys  usr
bin  dev  home  lib32  libx32  mnt  proc  run  srv  tmp  var
root@250c1e159ab7:/# cd app
root@250c1e159ab7:/app# ls
arquivo.txt
root@250c1e159ab7:/app# rm arquivo.txt
root@250c1e159ab7:/app# touch arquivo-meu-volume
root@250c1e159ab7:/app# ls
arquivo-meu-volume
root@250c1e159ab7:/app#

```

A partir de agora qualquer arquivo gravado no dir /app do container permanecerá persistente no nosso volume, e qualquer container que iniciar com esse ponto de montagem terá acesso aos arquivos também. Criamos o arquivo para podermos localizarmos em nosso host

Esse volume fica arquivado por padrão no seguinte diretório, esse diretório armazena diversas informações referente ao docker

`var/lib/docker` : Dentro desse diretório temos várias informações, mas no momento estamos buscando nosso volume.

```

root@pop-os:/var/lib/docker# ls
buildkit  image  nuke-graph-directory.sh  plugins  swarm  trust
containers  network  overlay2  runtimes  tmp  volumes
root@pop-os:/var/lib/docker# cd volumes/
root@pop-os:/var/lib/docker/volumes# ls
backingFsBlockDev  metadata.db  meu-volume
root@pop-os:/var/lib/docker/volumes# cd meu-volume/
root@pop-os:/var/lib/docker/volumes/meu-volume# ls
_data
root@pop-os:/var/lib/docker/volumes/meu-volume# cd _data/
root@pop-os:/var/lib/docker/volumes/meu-volume/_data# ls
arquivo-meu-volume

```

Nessa imagem conseguimos visualizar nossos volumes disponíveis e os dados dentro deles.

Temos a opção de executar o container também com o parâmetro `--mount`, neste caso não é necessário especificar o `type`, como foi feito no caso do `bind`, apenas informamos o volume que desejamos armazenar. Ficando da seguinte forma:

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

```
viniciusfigueira@pop-os:~$ docker run -it --mount source=meu-volume,target=/app ubuntu bash
root@2a48ad5f11b0:/# ls
app bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var
root@2a48ad5f11b0:/# cd app/
root@2a48ad5f11b0:/app# ls
arquivo-meu-volume
root@2a48ad5f11b0:/app#
```

E desta forma utilizando o `--mount` não é necessário criar um volume prévio, se passarmos o nome de um volume não existente no `source=` o próprio docker identifica e realiza a criação de um novo volume.

```
viniciusfigueira@pop-os:~$ docker run -it --mount source=meu-novo-volume,target=/app ubuntu bash
root@148c2a9b1fee:/# exit
viniciusfigueira@pop-os:~$ docker volume ls
DRIVER      VOLUME NAME
local       meu-novo-volume
local       meu-volume
viniciusfigueira@pop-os:~$
```

Temos a terceira forma que seria `tmpfs`. Nesta forma os dados armazenados no container permanecem de forma temporária enquanto o container está em execução, após a parada deste container os dados são perdidos. Essa forma de armazenamento pode ser utilizado em situações que não desejamos que alguns dados permanecem gravados na camada de RW do container após a parada do mesmo, por exemplo com dados sensíveis. Lembrando que essa forma de armazenamento até o momento só funciona em ambientes linux.

Para executar um container com esse tipo de armazenamento rodamos o seguinte comando:

```
docker run -it --tmpfs=/app ubuntu bash
```

```
viniciusfigueira@pop-os:~$ docker run -it --tmpfs=/app ubuntu bash
root@5c67866f8926:/# ls
app bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var
root@5c67866f8926:/# touch app/arquivotemp
root@5c67866f8926:/# ls app/
arquivotemp
root@5c67866f8926:/# exit
viniciusfigueira@pop-os:~$ docker run -it --tmpfs=/app ubuntu bash
root@b3c6c8cde75d:/# ls app/
root@b3c6c8cde75d:/#
```

Podemos observar que o `dir /app` fica com o fundo verde, que indica ser um diretório temporário, ao criarmos um arquivo nesse diretório ele permanece lá até o status de container `running`, se criarmos outro container passando o mesmo parâmetro ele não localizará o arquivo criado.

E da mesma forma de criação com outros volumes, com esse formato também podemos utilizar a tag `--mount`, ficando da seguinte forma:

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

```
viniciusfigueira@pop-os:~$ docker run -it --mount type=tmpfs,destination=/app ubuntu bash
root@3a597f7e56fb:/# ls
app bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var
root@3a597f7e56fb:/#
```

Comunicação entre containers

Por padrão os containers criados no nosso host ficam em uma rede bridge, onde podem se comunicar entre eles, conseguimos ver as informações de um container, inclusive informação de rede com o seguinte comando

`docker inspect "id_container"`

```
"Networks": {
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "cde602bac3cb116ffa8c155213126652a4882b26eddefab87ab36f52c05c2a5b",
    "EndpointID": "91efb14361f9545e884207ea2b5531bd0b3c06675698482b81165cfa14bd91f4",
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:02",
    "DriverOpts": null
  }
}
```

Como dito anteriormente retornará várias informações, mas ao final podemos ver essas informações, identificando o endereço do container e informações de rede do mesmo.

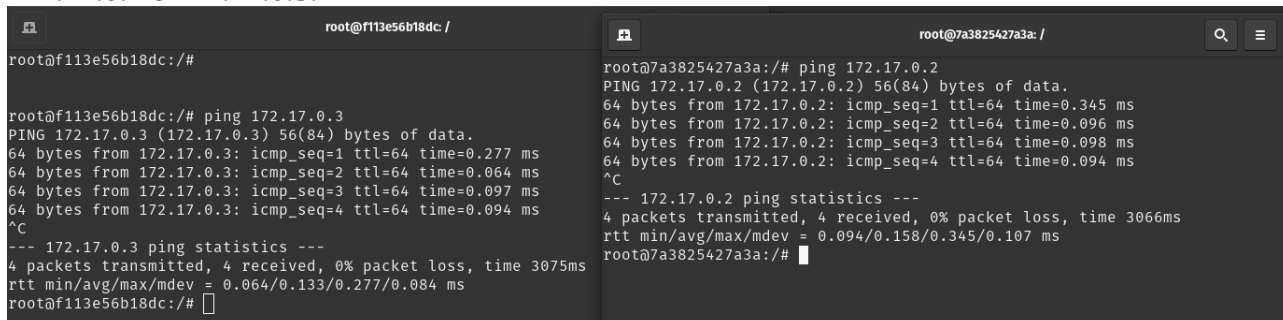
Para verificarmos todas nossas configurações de rede utilizamos o seguinte comando:

`docker network ls`

```
viniciusfigueira@pop-os:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cde602bac3cb        bridge             bridge              local
92245e9095ff        host               host                local
9d3fd208b6ee        none              null                local
```

e temos a seguinte saída com as informações de rede.

Dito isso iremos subir dois containers e testar de fato a conexão entre eles, passando o comando `docker inspect` conseguimos descobrir o ip de cada container sendo eles respectivamente: 172.17.0.2 e 172.17.0.3.



The image shows two terminal windows side-by-side. The left window is titled 'root@f113e56b18dc: /' and shows a ping command being executed from the container with IP 172.17.0.3 to the container with IP 172.17.0.2. The output shows 4 successful pings with varying response times. The right window is titled 'root@7a3825427a3a: /' and shows a ping command being executed from the container with IP 172.17.0.2 to the container with IP 172.17.0.3. The output shows 4 successful pings with varying response times. Both windows show the standard ping output including packet loss statistics.

Porém a comunicação via IP nem sempre pode ser funcional por ter a criação de vários containers e a troca de Ips, então podemos realizar essa conexão via hostnames. Por padrão o docker cria os containers com nomes aleatórios, mas podemos alterar isso usando o parâmetro `--name` e definindo o nome do container. Além disso podemos criar nossa própria rede bridge para comunicação, então faremos isso nos próximos passos:

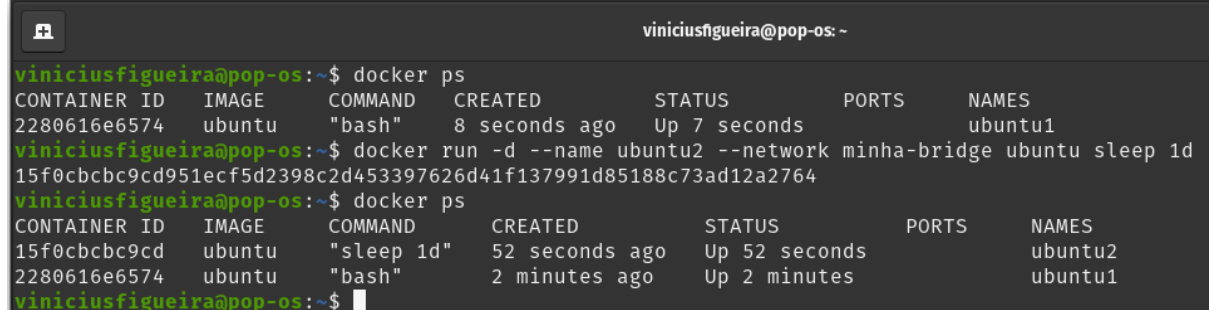
Criando uma rede:

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

```
viniciusfigueira@pop-os:~$ docker network create --driver bridge minha-bridge
9650cf7996b2ecc7d11b092b3776cd17522c39bedcf433938e8b5128a464aef7
viniciusfigueira@pop-os:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cde602bac3cb        bridge              bridge              local
92245e9095ff        host                host                local
9650cf7996b2        minha-bridge        bridge              local
9d3fd208b6ee        none                null                local
viniciusfigueira@pop-os:~$
```

No comando acima definimos o tipo de rede como bridge e colocamos o nome da rede, agora criaremos nossos containers com o nome que desejarmos e dentro da rede que criamos.

```
viniciusfigueira@pop-os:~$ docker run -it --name ubuntu1 --network minha-bridge ubuntu bash
root@2280616e6574:/#
```



```
viniciusfigueira@pop-os:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
2280616e6574   ubuntu   "bash"    8 seconds ago    Up 7 seconds           ubuntu1
viniciusfigueira@pop-os:~$ docker run -d --name ubuntu2 --network minha-bridge ubuntu sleep 1d
15f0cbcb9cd951ecf5d2398c2d453397626d41f137991d85188c73ad12a2764
viniciusfigueira@pop-os:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
15f0cbcb9cd   ubuntu   "sleep 1d"    52 seconds ago    Up 52 seconds           ubuntu2
2280616e6574   ubuntu   "bash"       2 minutes ago     Up 2 minutes            ubuntu1
viniciusfigueira@pop-os:~$
```

Criamos dois containers na rede que criamos e iremos testar a conectividade pelo hostname que podemos observar ser respectivamente(ubuntu1 , ubuntu2)

```
root@2280616e6574:/# ping ubuntu2
PING ubuntu2 (172.18.0.3) 56(84) bytes of data.
64 bytes from ubuntu2.minha-bridge (172.18.0.3): icmp_seq=1 ttl=64 time=0.186 ms
64 bytes from ubuntu2.minha-bridge (172.18.0.3): icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from ubuntu2.minha-bridge (172.18.0.3): icmp_seq=3 ttl=64 time=0.119 ms
64 bytes from ubuntu2.minha-bridge (172.18.0.3): icmp_seq=4 ttl=64 time=0.106 ms
^C
--- ubuntu2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3072ms
rtt min/avg/max/mdev = 0.106/0.129/0.186/0.033 ms
root@2280616e6574:/#
```

Como podemos observar, funcionou o ping pelo hostname e ele informa a rede na qual está comunicando.

Explicando as interfaces padrão none e host

none: Quando criamos um container e setamos a interface none, informamos que queremos um container sem interface de rede pré definida, ou seja, este container não vai ter nenhuma comunicação por padrão.

Host: Ao contrário da rede none, quando setamos a interface host, tiramos o isolamento de rede entre o container e o host, ou seja, o container roda da mesma interface do host. Desta forma se tivermos alguma aplicação rodando no container não é necessário especificar ou dar expose de portas, conseguimos uma comunicação direta entre host e container.

Após mostrar esses conceitos e comandos iremos criar um ambiente de teste baseado nos arquivos disponibilizados pela plataforma alura.

Primeiro iremos baixar as imagens do docker hub, sendo a imagem de banco e aplicação

Iremos utilizar o banco mongo em uma versão específica

Para baixar essa imagem utilizamos o comando docker pull

```
docker pull mongo:4.4.6
```

```
docker pull aluradocker/alura-books:1.0
```

```
viniciusfigueira@pop-os:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	825d55fb6340	2 weeks ago	72.8MB
mongo	4.4.6	61ea24dc52c6	9 months ago	423MB
aluradocker/alura-books	1.0	ebff169e5013	4 years ago	691MB

Podemos ver que as imagens foram baixadas com sucesso, então iremos executar setando nossa rede e o nome. No banco iremos utilizar um nome já definido pela aplicação para poder realizar a comunicação por hostname. Ficando assim os comandos.

```
docker run -d --network minha-bridge --name meu-mongo mongo:4.4.6
```

```
viniciusfigueira@pop-os:~$ docker run -d --network minha-bridge --name meu-mongo mongo:4.4.6  
f9f532152a4d5fce626fbd167fe7ca3897d180f4d153e668283f9747531c0bce
```

```
docker run -d --network minha-bridge --name alurabooks -p 3000:3000 aluradocker/alura-books:1.0
```

```
viniciusfigueira@pop-os:~$ docker run -d --network minha-bridge --name alurabooks -p 3000:3000 aluradocker/alura-books:1.0  
2c5a6642393dabd346ce5e63bd55b42dd8745c400e16bb12e492ba55a99bbdc7  
viniciusfigueira@pop-os:~$
```

Após isso podemos acessar nosso ambiente pelo browse. Acessando o seguinte ponto ponto para popularmos nossa dash inicial.

localhost:3000/seed

Posteriormente podemos acessar normalmente nossa dash e a aplicação estará em execução.

Utilizando Docker Compose

Docker Compose trata-se de uma ferramenta de coordenação de containers. Através de um arquivo podemos provisionar mais de um recurso por vez, facilitando o processo de ciclo de vida dos containers.

Neste caso criamos um arquivo YML para realizar a criação desses containers apenas com um comando.

```
docker-compose.yml
1  version: '3.9' #Versão YML
2  services: #Serviços que serão executados
3    mongodb: #Primeiro container
4      image: mongo:4.4.6 #Imagem base do container
5      container_name: meu-mongo #Nome do container
6      networks: # Interface de rede do container
7        - compose-bridge #Nome da interface de rede
8
9    alurabooks: #Segundo container
10     image: aluradocker/alura-books:1.0 #Imagem base do container
11     container_name: alura-books #Nome do container
12     networks: # Interface de rede do container
13       - compose-bridge #Nome da interface de rede
14     ports: #Configuração de portas para acesso a aplicação
15       - 3000:3000 #Portas selecionadas, sendo a primeira do host, e segunda a do container
16     depends_on: #Este comando serve para indicar que esse serviço depende do serviço mongodb para funcionar
17       - mongodb
18
19
20 networks: #Configurando a interface de rede mencionada a cima
21   compose-bridge:
22     driver: bridge #Modo em que a placa vai ser configurada
```

Após criar o arquivo YML e salvar em um diretório definido, iremos rodar o comando para criar nosso ambiente

`docker-compose up`

Lembrando de utilizar o comando no diretório em que o arquivo foi criado.

```
iniciastiguel@pop-os: ~/Desktop/yml$ docker-compose up
Creating alura-books ... done
Creating meu-mongo ... done
Attaching to alura-books, meu-mongo
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.699+00:00"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx":"main","msg":"Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --
meu-mongo | ssldisabledprotocols: 'none'"}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.701+00:00"},"s":"W",  "c":"ASIO",    "id":22601,   "ctx":"main","msg":"No TransportLayer configured during NetworkInterface startup"}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.701+00:00"},"s":"I",  "c":"NETWORK",  "id":4648601, "ctx":"main","msg":"Implicit TCP FastOpen unavailable. If TCP FastOpen is required, set
meu-mongo | tcpFastOpenServer, tcpFastOpenClient, and tcpFastOpenQueueSize."}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.701+00:00"},"s":"I",  "c":"STORAGE",  "id":4615611, "ctx":"initandlisten","msg":"MongoDB starting","attr":{"pid":1,"port":27017,"dbPath":"/
meu-mongo | data/db","architecture":"64-bit","host":"677d207414ad"}}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.701+00:00"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx":"initandlisten","msg":"Build Info","attr":{"buildInfo":{"version":"4.4.6","gitVer
meu-mongo | sion":"72e6621c2c9eb709358d5e78ad7f5c1d0d0d7","opensslVersion":"OpenSSL 1.1.1  11 Sep
meu-mongo | target_arch":"x86_64"}}}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.701+00:00"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx":"initandlisten","msg":"Operating System","attr":{"os":{"name":"Ubuntu","version":
meu-mongo | "18.04"}}}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.701+00:00"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx":"initandlisten","msg":"Options set by command line","attr":{"options":{"net":{"bi
meu-mongo | ndIp":"*"}}}}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.702+00:00"},"s":"I",  "c":"STORAGE",  "id":22297,   "ctx":"initandlisten","msg":"Using the XFS filesystem is strongly recommended with the
meu-mongo | WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem","tags":{"startupWarnings"}}
meu-mongo | {"t":{"$date":"2022-04-21T23:56:23.702+00:00"},"s":"I",  "c":"STORAGE",  "id":22315,   "ctx":"initandlisten","msg":"Opening WiredTiger","attr":{"config":{"create,cache_size=74
meu-mongo | 07M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=
meu-mongo | 100000,close_scan_interval=10,close_handle_minimum=250),statistics_log=(wait=0),verbose=[recovery_progress,checkpoint_progress,compact_progress],"}}
alura-books | npm info it worked if it ends with ok
alura-books | npm info using npm@5.3.0
alura-books | npm info using node@v8.2.1
alura-books | npm info lifecycle alura-docker@1.0.0-prestart: alura-docker@1.0.0
alura-books | npm info lifecycle alura-docker@1.0.0-start: alura-docker@1.0.0
alura-books |
alura-books | > alura-docker@1.0.0 start /var/www
```

Ao executar o comando sem utilizar o parâmetro `-d` ao fecharmos o terminal, os serviços são encerrados, para evitar isso utilizamos o seguinte comando

`docker-compose up -d`

Para encerrarmos o serviço utilizamos o seguinte comando:

docker-compose down

```
viniciusfigueira@pop-os:~/Desktop/ymls$ docker-compose up -d
Creating network "ymls_compose-bridge" with driver "bridge"
Creating meu-mongo ... done
Creating alura-books ... done
viniciusfigueira@pop-os:~/Desktop/ymls$ docker-compose ps
  Name                       Command                                State              Ports
-----
alura-books    npm start                                Up                 0.0.0.0:3000->3000/tcp,:::3000->3000/tcp
meu-mongo      docker-entrypoint.sh mongod             Exit 14
viniciusfigueira@pop-os:~/Desktop/ymls$ docker-compose down
Stopping alura-books ... done
Removing alura-books ... done
Removing meu-mongo ... done
Removing network ymls_compose-bridge
viniciusfigueira@pop-os:~/Desktop/ymls$
```