Docker CLI

A Docker Command Line Interface (Docker CLI) é a via de comunicação do usuário com a Docker Engine.

A estrutura fundamental de um comando na CLI é:

```
docker {objeto} {ação}
```

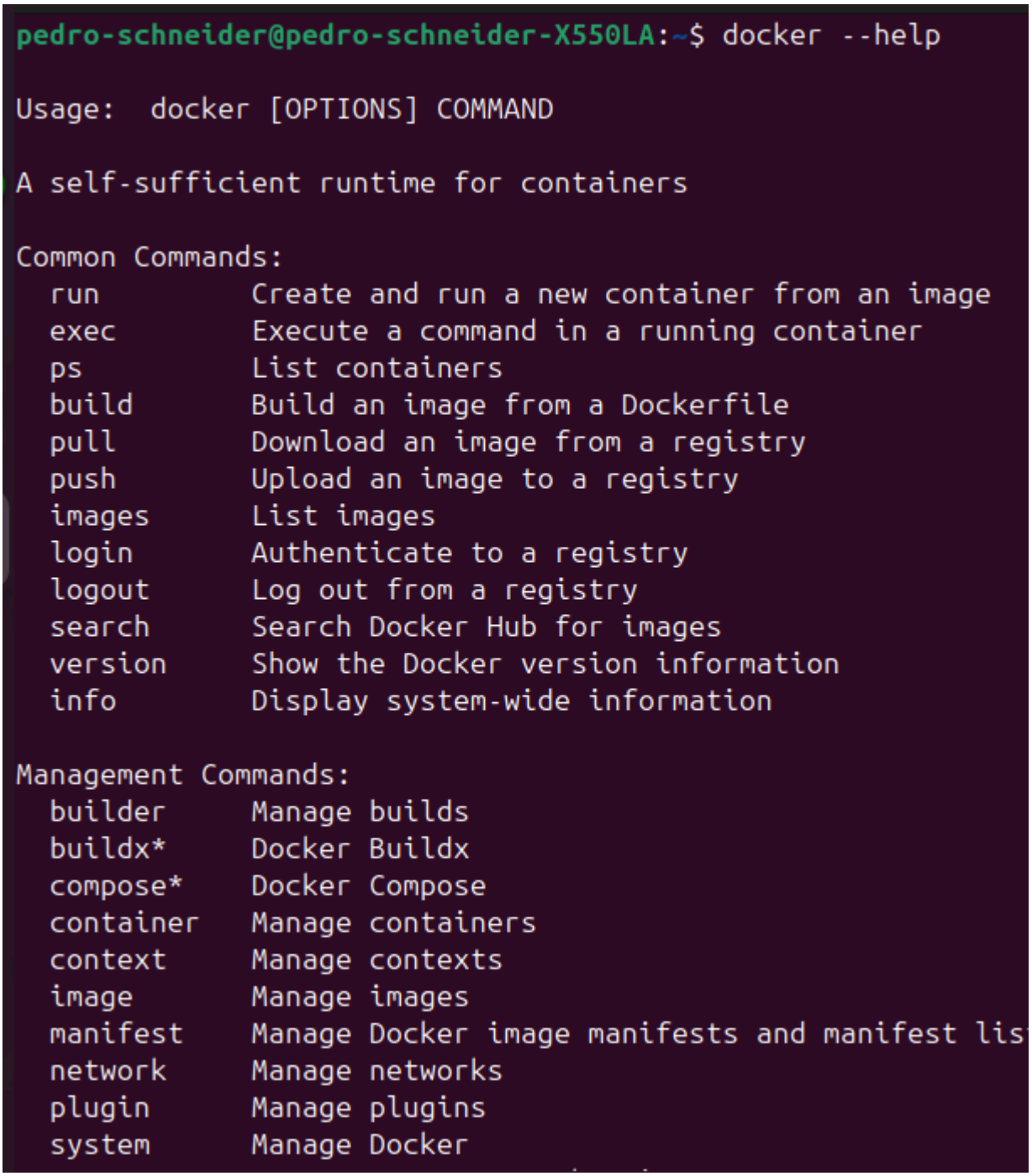
O objeto pode ser um container, imagem, rede, volume etc.

As ações vão depender do tipo de objeto com o qual se está interagindo. Para um container por exemplo, se pode listar (ls), remover (rm), criar (create) etc.

Help

É possível consultar os possíveis objetos e ações na CLI pelo comando "--help".

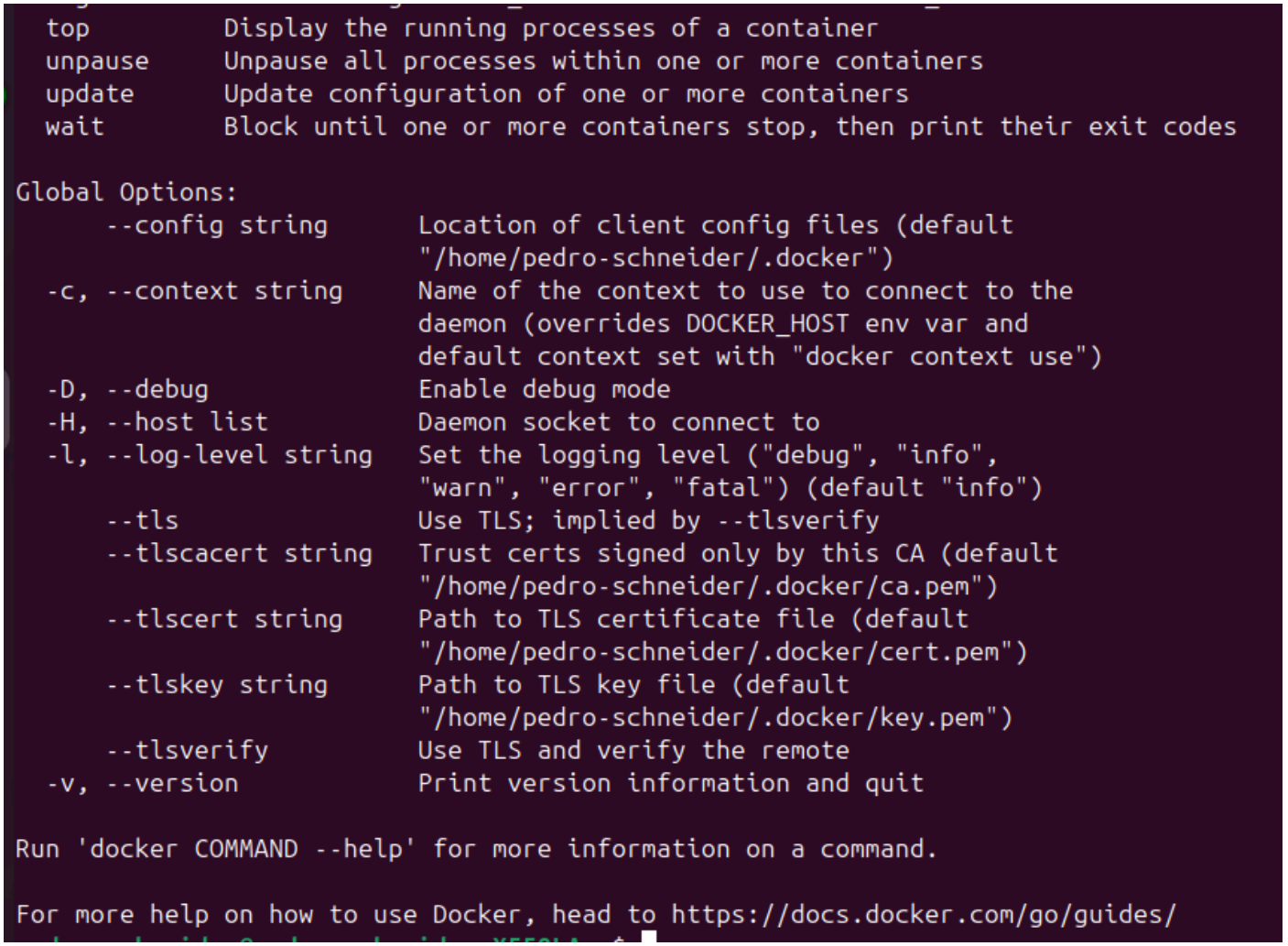
Por exemplo, "`docker --help`" retorna:



```
  trust       Manage trust on Docker images
  volume      Manage volumes

Swarm Commands:
  swarm       Manage Swarm

Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  events      Get real time events from the server
  export      Export a container's filesystem as a tar archive
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
```



Ou seja, informações sobre: comandos comuns, administrativos, gerais e globais.

Se o comando “--help” for chamado para um comando específico do docker, como, por exemplo, “docker builder --help”:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker builder --help
Extended build capabilities with BuildKit

Usage:  docker buildx [OPTIONS] COMMAND

Extended build capabilities with BuildKit

Options:
      --builder string   Override the configured builder instance (default "default")
  -D, --debug            Enable debug logging

Management Commands:
  imagetools  Commands to work on images in registry

Commands:
  bake        Build from a file
  build       Start a build
  create      Create a new builder instance
  dial-stdio  Proxy current stdio streams to builder instance
  du          Disk usage
  inspect     Inspect current builder instance
  ls          List builder instances
  prune       Remove build cache
  rm          Remove one or more builder instances
  stop        Stop builder instance
  use         Set the current builder instance
  version     Show buildx version information

Run 'docker buildx COMMAND --help' for more information on a command.

Experimental commands and flags are hidden. Set BUILDX_EXPERIMENTAL=1 to show them.
```

Informações sobre o comando em específico, com seus parâmetros e suas funções, são retornadas.

Manipulação básica de containers através da CLI

Todo container é inicializado de forma isolada um do outro, por isso, posteriormente será necessário o uso de redes para a interação entre eles.

Criando o container

O comando para criar um container, sem inicializá-lo, é:

```
docker container create --name nome_do_container -it alpine sh
```

- A opção i (interactive) garante que o container continue recebendo informações da stream stdin, assim sendo possível enviar comandos repetidamente antes do encerramento da sessão.
- A opção t (TTY) adiciona uma estrutura interativa de terminal dentro do container para a interação permitida pela opção i.
- O comando sh adiciona uma shell dentro do container, fornecendo um backend para os comandos enviado pelo TTY.
- A opção alpine indica que o novo container deve ser construído a partir da alpine, uma imagem básica baseada no Linux Alpine.

Caso o desejado fosse que o container rodasse logo após a criação bastaria alterar o comando “create” por “run”.

Startando o container

Para inicializar o container, basta utilizar o comando “start”.

```
docker container start nome_do_container
```

Interagindo com o container

Para enviar comandos e receber informações do container, deve-se conectar o terminal (o stdin, stdout e stderr) a shell do container. Isso é feito com o comando “attach”. Deve-se inicializar o container antes de conectar-se a ele.

```
docker container attach nome_do_container
```

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container attach test
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ #
```

A partir disso, abre-se um TTY para que comandos, como o ls, possam ser executados na shell desse container.

OBS: Os arquivos que aparecem na listagem são apenas os do container. Se forem corrompidos, apenas o container é corrompido. Containers são estruturas descartáveis, modificações feitos nele não afetam outros containers. São como “diapers” (fraldas).

Exclusão de um container

Basta utilizar o comando “rm”.

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container rm test
test
```

Start com attach

É possível já se conectar ao container na mesma linha da inicialização:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container start -ia test
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ #
```

As opções “ia” indicam interactive e attach.

Do create para o attach em uma linha

É possível criar, inicializar e se conectar a um container em uma só linha:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container run --name test2 -it alpine sh
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ #
```

O comando “run” cria, inicializa e conecta ao container sequencialmente.

Renomeando um container

É possível renomear containers, com o comando “raname”.

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container create -it alpine sh
78da299a3c2623d36ddba7798c9056c75f6203b0b5d5a347c1b3e52feb65b35e
pedro-schneider@pedro-schneider-X550LA:~$ docker container ls -a
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                          PORTS     NAMES
78da299a3c26   alpine        "sh"       6 seconds ago    Created                                   agitated_sinoussi
002ae1245ead   alpine        "sh"       24 minutes ago   Exited (0) About a minute ago             test2
c264e3ad7072   alpine        "sh"       38 minutes ago   Exited (0) 25 minutes ago                 test
b2595bec46ad   hello-world   "/hello"   22 hours ago     Exited (0) 22 hours ago                   thirsty_cori
c72680774a20   hello-world   "/hello"   23 hours ago     Exited (0) 23 hours ago                   laughing_blackburn
80e53ab81251   hello-world   "/hello"   23 hours ago     Exited (0) 23 hours ago                   competent_archimedes
pedro-schneider@pedro-schneider-X550LA:~$ docker container rename 78da test3
pedro-schneider@pedro-schneider-X550LA:~$ docker container ls -a
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                          PORTS     NAMES
78da299a3c26   alpine        "sh"       32 seconds ago   Created                                   test3
002ae1245ead   alpine        "sh"       25 minutes ago   Exited (0) About a minute ago             test2
c264e3ad7072   alpine        "sh"       38 minutes ago   Exited (0) 25 minutes ago                 test
b2595bec46ad   hello-world   "/hello"   23 hours ago     Exited (0) 23 hours ago                   thirsty_cori
c72680774a20   hello-world   "/hello"   23 hours ago     Exited (0) 23 hours ago                   laughing_blackburn
80e53ab81251   hello-world   "/hello"   23 hours ago     Exited (0) 23 hours ago                   competent_archimedes
```

No exemplo acima, foi criado um container sem nome especificado, que ganha um nome aleatório do Docker, e depois foi renomeado para “test3” com o comando “rename” a partir da ID.

Comandos com arquivos e logs de containers

Desconectar do container sem interromper sua execução

É possível com o atalho CTRL+P+Q:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container attach test3
You cannot attach to a stopped container, start it first
pedro-schneider@pedro-schneider-X550LA:~$ docker container start -ia test3
/ # pedro-schneider@pedro-schneider-X550LA:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS          PORTS     NAMES
78da299a3c26   alpine    "sh"      5 minutes ago   Up 43 seconds             test3
pedro-schneider@pedro-schneider-X550LA:~$
```

Executar comandos sem attach

É possível executar comandos num container sem se conectar a ele, através do comando “exec”.

```
/ # pedro-schneider@pedro-schneider-X550LA:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS          PORTS     NAMES
78da299a3c26   alpine    "sh"      5 minutes ago   Up 43 seconds             test3
pedro-schneider@pedro-schneider-X550LA:~$ top

top - 16:28:03 up  6:05,  1 user,  load average: 0.81, 0.78, 0.66
Tasks: 283 total,   1 running, 282 sleeping,   0 stopped,   0 zombie
Mem: 7501612K used, 508436K free, 660244K shrd, 152232K buff, 2769408K cached
CPU:   2% usr   0% sys   0% nic  95% idle   0% io   0% irq   0% sirq
Load average: 0.39 0.64 0.63 1/1290 18
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     S     1724   0%   3   0% sh
   13     0 root     R     1624   0%   3   0% top
^Ccontext canceled   S     1616   0%   0   0% top
pedro-schneider@pedro-schneider-X550LA:~$
```

No exemplo acima, foi executado o comando “top” no container “test3”. A linha de código necessária, que não aparece na imagem, é:

```
docker container exec nome_do_container commando
```

OBS: para um comando funcionar em um container, como “top” ou “cat”, seu binário deve estar disponível na imagem usada para a construção do container, como a alpine. Ou seja, o leque de comandos disponíveis varia com a imagem.

Cópia de pastas e arquivos de fora

Se eu tenho uma pasta ou arquivo em meu sistema e quero copiá-lo para um container, basta utilizar o comando “cp”:

```
pedro-schneider@pedro-schneider-X550LA:~$ mkdir pasta_teste
pedro-schneider@pedro-schneider-X550LA:~$ touch pasta_test/arquivo_teste
touch: cannot touch 'pasta_test/arquivo_teste': No such file or directory
pedro-schneider@pedro-schneider-X550LA:~$ touch pasta_test/arquivo_teste.txt
touch: cannot touch 'pasta_test/arquivo_teste.txt': No such file or directory
pedro-schneider@pedro-schneider-X550LA:~$ rmdir pasta_teste
pedro-schneider@pedro-schneider-X550LA:~$ mkdir pasta_teste
pedro-schneider@pedro-schneider-X550LA:~$ touch pasta_teste/arquivo_teste.txt
pedro-schneider@pedro-schneider-X550LA:~$ docker container cp pasta_teste test3:/
Successfully copied 2.05kB to test3:/
pedro-schneider@pedro-schneider-X550LA:~$ docker container exec test3 ls
bin
dev
etc
home
lib
media
mnt
opt
pasta_teste
proc
root
run
sbin
srv
sys
tmp
usr
var
pedro-schneider@pedro-schneider-X550LA:~$
```

No exemplo acima, uma pasta é criada, depois um arquivo de texto é criado dentro dessa pasta, e a pasta é copiada para dentro do container test3 com o comando “cp”. Depois, utilizando o exec, o comando de listagem (ls) é executado no container, mostrando que, de fato, a pasta “pasta_teste” foi copiada. Agora, é possível modificá-lo como quiser:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container attach test3
/ # ls
bin          etc          lib          mnt          pasta_teste  root         sbin         sys          usr
dev          home         media        opt          proc         run          srv          tmp          var
/ # cd pasta_teste
/pasta_teste # ls
arquivo_teste.txt
/pasta_teste # echo "conteudo_teste" > arquivo_teste
/pasta_teste # ls
arquivo_teste      arquivo_teste.txt
/pasta_teste # rm arquivo_teste
/pasta_teste # echo "conteudo_teste" > arquivo_teste.txt
/pasta_teste # cat arquivo_teste.txt
conteudo_teste
/pasta_teste #
```

Cópia de pastas e arquivos para fora

Para fazer o processo contrário, basta inverter fonte e destino nos parâmetros do comando “cp”:

```
pedro-schneider@pedro-schneider-X550LA:~$ mkdir bkp
pedro-schneider@pedro-schneider-X550LA:~$ cd bkp
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container cp test3:/pasta_teste/arquivo_teste.txt .
Successfully copied 2.05kB to /home/pedro-schneider/bkp/.
pedro-schneider@pedro-schneider-X550LA:~/bkp$ ls
arquivo_teste.txt
pedro-schneider@pedro-schneider-X550LA:~/bkp$ cat arquivo_teste.txt
conteudo_teste
pedro-schneider@pedro-schneider-X550LA:~/bkp$
```

Visualizando logs

Para visualizar o registro de comando dentro de um container, basta utilizar o comando “logs”:

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container logs test3
/ # ls
bin          etc          lib          mnt          pasta_teste  root         sbin         sys          usr
dev          home         media        opt          proc         run          srv          tmp          var
/ # cd pasta_teste
/pasta_teste # ls
arquivo_teste.txt
/pasta_teste # echo "conteudo_teste" > arquivo_teste
/pasta_teste # ls
arquivo_teste      arquivo_teste.txt
/pasta_teste # rm arquivo_teste
/pasta_teste # echo "conteudo_teste" > arquivo_teste.txt
/pasta_teste # cat arquivo_teste.txt
conteudo_teste
pedro-schneider@pedro-schneider-X550LA:~/bkp$ ;5R;5R;16R;16R;16R;16R;16R;16R
```

Inspeção do container

O comando “inspect” retorna um conteúdo JSON com diversas informações sobre o container, como nome, ID, variáveis de estado (status, Error, StartedAt etc), a imagem, o diretório, uso de hardware, redes conectadas etc.

O comando é:

```
docker container inspect nome_do_container
```

Mapeamento de volumes

Volumes, em Docker, é um diretório com dados persistentes de containers. Isto é, um container mapeado para um volume vai conter todos os dados do volume, e vice-versa. Os dados são persistentes pois, mesmo após a deleção do container, os dados permanecem no volume.

Mapeamento na criação do container

Para mapear um diretório como volume para um container sendo criado, basta passar a opção v, que tem como parâmetros o caminho do host (diretório do volume) e o diretório a ser mapeado para o container:

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container run -v /home/pedro-schneider/bkp/:/bkp/ --name test-vol -it alpin
e sh
/ # ls
bin   bkp   dev   etc   home  lib   media mnt   opt   proc  root  run   sbin  srv   sys   tmp   usr   var
```

A sintaxe dos parâmetros de v é: path_host:path_container.

Repare que o container já é inicializado com uma pasta bkp, que está mapeada a pasta bkp do host.

Se uma alteração no conteúdo mapeado é feita no container, ela também é feita no volume:

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container run -v /home/pedro-schneider/bkp/:/bkp/ --name test-vol -it alpin
e sh
/ # ls
bin    bkp    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # cd bkp
/bkp # ls
arquivo_teste.txt
/bkp # echo "conteudo alterado pelo container" > arquivo_test.txt
/bkp # echo "conteudo alterado pelo container" > arquivo_teste.txt
/bkp #
```

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ cat arquivo_teste.txt
conteudo alterado pelo container
pedro-schneider@pedro-schneider-X550LA:~/bkp$
```

E vice-versa:

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ echo "conteudo alterado pelo host"
 > arquivo_teste.txt
pedro-schneider@pedro-schneider-X550LA:~/bkp$
```

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container run -v /home/pedro-schneider/bkp/:/bkp/ --name test-vol -it alpin
e sh
/ # ls
bin    bkp    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # cd bkp
/bkp # ls
arquivo_teste.txt
/bkp # echo "conteudo alterado pelo container" > arquivo_test.txt
/bkp # echo "conteudo alterado pelo container" > arquivo_teste.txt
/bkp # cat arquivo_teste.txt
conteudo alterado pelo host
/bkp #
```

Cuidado a se tomar: remoção de arquivos e subdiretórios

A remoção de um arquivo ou subdiretório de um dos diretórios (volume ou container) implica na remoção no outro também.

```
/bkp # rm arquivo_teste.txt
/bkp # rm arquivo_test.txt
/bkp #
```

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ ls
pedro-schneider@pedro-schneider-X550LA:~/bkp$
```

Persistência dos dados

Caso o container seja deletado, seus dados mapeados ao volume permanecerão salvos na máquina hospedeira:

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container run -v /home/pedro-schneider/bkp/:/bkp/ --name test-vol -it alpin
e sh
/ # ls
bin    bkp    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # cd bkp
/bkp # ls
arquivo_teste.txt
/bkp # echo "conteudo alterado pelo container" > arquivo_test.txt
/bkp # echo "conteudo alterado pelo container" > arquivo_teste.txt
/bkp # cat arquivo_teste.txt
conteudo alterado pelo host
/bkp # rm arquivo_teste.txt
/bkp # rm arquivo_test.txt
/bkp # touch test
/bkp # exit
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container rm test-vol
test-vol
pedro-schneider@pedro-schneider-X550LA:~/bkp$ ls
test
pedro-schneider@pedro-schneider-X550LA:~/bkp$
```

E podem ser usados para mapear um novo volume num novo container:

```
pedro-schneider@pedro-schneider-X550LA:~/bkp$ docker container run -v /home/pedro-schneider/bkp/:/bkp/ --name test-vol-2 -it alp
ine sh
/ # ls
bin    bkp    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # cd bkp
/bkp # ls
test
/bkp #
```

Aplicações

Volumes são muito usados para fazer persistir os dados de logs, configurações, bancos de dados etc. É um backup.

Mapeamento de Portas

É possível criar containers na forma de webservers.

```
docker run -d --name my-nginx nginx
```

A linha de código acima:

- cria e inicializa um container (“docker run --name my-nginx”)
- no modo detached, para não trancar o terminal (“-d”)
- e utilizando a imagem nginx, que forma um webserver básico.

IP da bridge network

O Docker, por padrão, cria uma rede bridge conectando todos os containers criados, com cada um recebendo um IP. O host tem acesso a essa rede através de uma interface criada pelo Docker, e consegue se conectar com qualquer container pertencente a ela utilizando o IP dado a esse container.

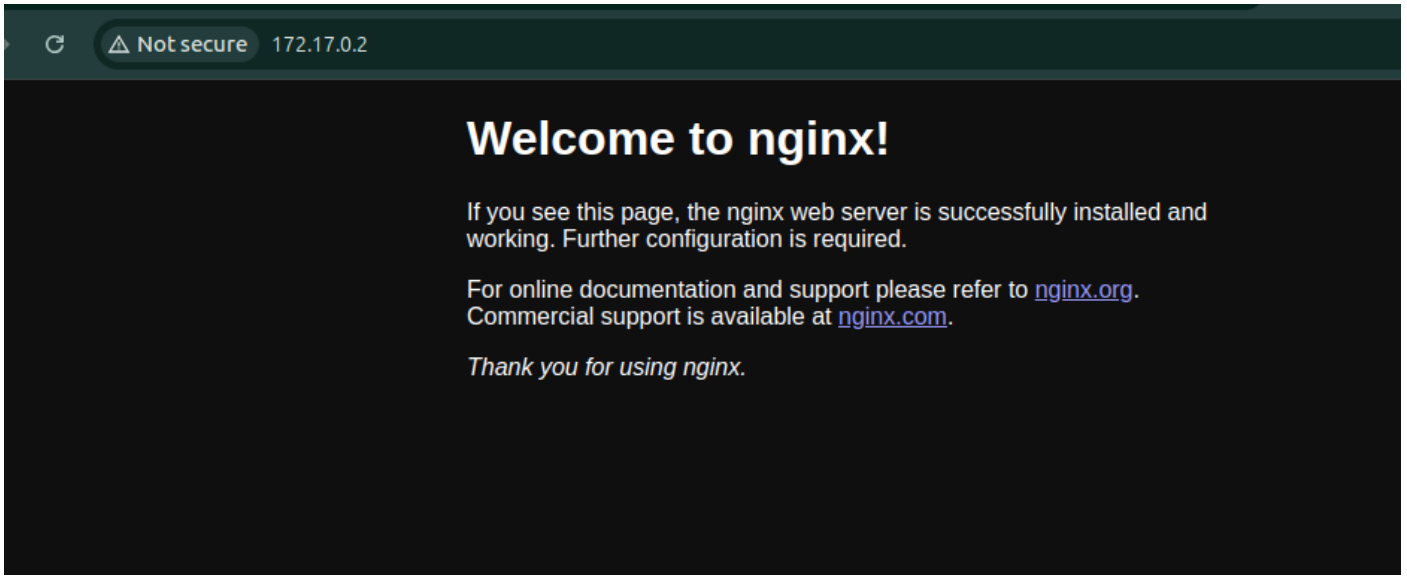
Por exemplo, se um container for criado com a imagem do nginx (ou seja, for construído como um webserver), ele ganha um IP dentro da rede bridge que pode ser visto com o comando “inspect”:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker run -d --name my-nginx nginx
2f20c88f70a9f0c817339b8d5dbfcdcf9453449f03e16c2f905c588ac7ddfa5f
pedro-schneider@pedro-schneider-X550LA:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS     NAMES
2f20c88f70a9   nginx     "/docker-entrypoint.…"   7 seconds ago   Up 7 seconds   80/tcp    my-nginx
```

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container inspect my-nginx
```

Vai estar no item“NetworkSettings”.

Se ele for entrado em um web-browser, a seguinte página é carregada:



Interface docker0

Essa conexão com o container é possível graças à interface de rede que o docker cria para o host, chamada docker0, que permite a conexão do host com a rede bridge que contém os containers. O gateway 172.17.0.1 dá rota para os containers da rede, que em IPs na forma 172.17.0.x.

Acesso externo

Porém, a conexão com o container utilizando esse IP só é possível pela máquina host, onde a engine e a docker0 estão instaladas, já que é na docker0 que está o gateway para a rede bridge onde estão os containers.

Para acessar de fora da máquina, é necessário o mapeamento de portas, para que o IP da máquina hospedeira, que está conectada à Internet, possa ser usado.

Mapeamento de Portas

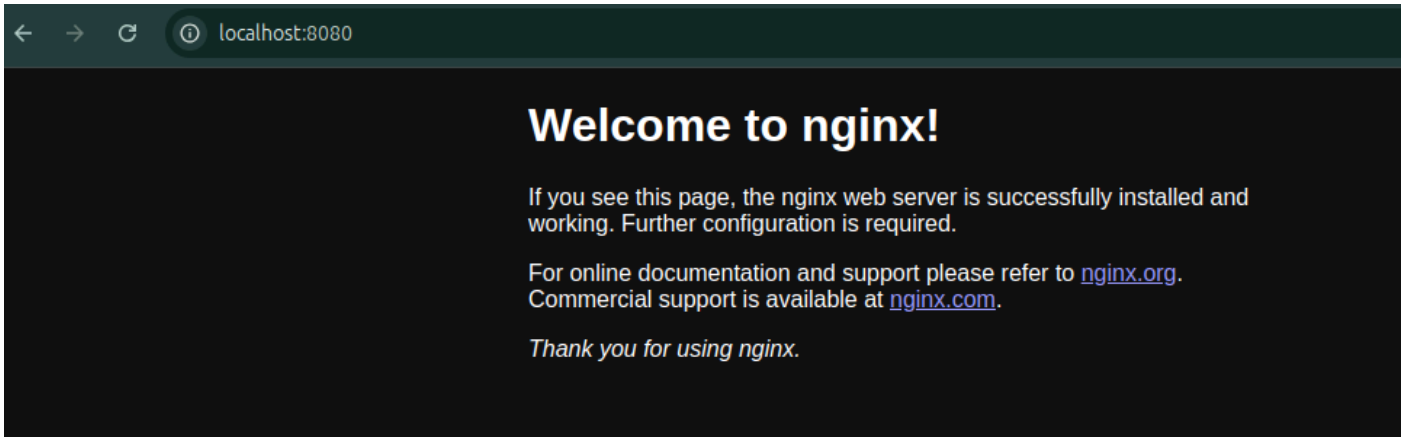
Assim como é possível mapear diretórios do host com diretórios do container para criar volumes, é possível mapear portas do host com portas do host para que máquina conectadas à mesma rede do hospedeiro possam se conectar com o container utilizando o IP desse mesmo hospedeiro:

```
docker run --name my-nginx -d -p 8080:80 nginx
```

A linha de código acima:

- cria e inicializa um container (“docker run --name my-nginx”)
- no modo detached, para não trancar o terminal (“-d”)
- mapeando a porta 8080 do host com a 80 (por ser um serviço HTTP, o nginx expõe essa porta) do container (“-p 8080:80”)
- e utilizando a imagem nginx, que forma um webserver básico.

Assim, é possível acessar o container através da porta 8080 do host:



Na coluna “PORTS” na listagem de containers, é indicado “0.0.0.0:8080->80/tcp”, o que indica que o container escuta todos os IPs disponíveis no host (0.0.0.0) através da porta 8080 do host que está mapeada a porta 80 do container, com o protocolo TCP.

Entrypoint e interatividade

Para algumas imagens, como a nginx, o container é inicializado com um entrypoint, sem shell e sem modo interativo. Portanto, para se conectar ao container e realizar comandos dentro dele, é necessário o uso do comando “exec”:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS              PORTS
     NAMES
e06518875773   nginx     "/docker-entrypoint.…"   About a minute ago   Up About a minute   0.0.0.0:8080->80/tcp, [::]:8080->80/t
cp   my-nginx-mpd
pedro-schneider@pedro-schneider-X550LA:~$ docker container exec -it my-nginx-mpd sh
# ls
bin   dev                  docker-entrypoint.sh  home  lib64  mnt  proc  run   srv  tmp  var
boot  docker-entrypoint.d  etc                   lib   media  opt  root  sbin  sys  usr
# cd etc
# ls
adduser.conf            debian_version  group-     kernel         mtab          profile.d    rmt       systemd
alternatives            default         gshadow    ld.so.cache    nginx         rc0.d        security  terminfo
apt                     deluser.conf    gshadow-   ld.so.conf     nsswitch.conf rc1.d        selinux   timezone
bash.bashrc             dpkg            gss        ld.so.conf.d   opt           rc2.d        shadow    update-motd.d
bindresvport.blacklist  e2scrub.conf    host.conf  libaudit.conf  os-release    rc3.d        shadow-   xattr.conf
ca-certificates         environment     hostname   localtime      pam.conf      rc4.d        shells
ca-certificates.conf    fonts           hosts      login.defs     pam.d         rc5.d        skel
cron.d                  fstab           init.d     logrotate.d    passwd        rc6.d        ssl
cron.daily              gai.conf        issue      mke2fs.conf    passwd-       rcS.d        subgid
debconf.conf            group           issue.net  motd           profile       resolv.conf  subuid
#
```

Imagem

Uma imagem está para a classe assim como um container está para um objeto. Isto é, a imagem é uma blueprint, um construtor que gera uma instância que pode ser posteriormente personalizada sem afetar a estrutura geradora inicial, tornando-a reutilizável.

Gerando uma imagem por commit

É possível gerar uma imagem a partir de um container com o comando “commit”.

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container run -it --name docker-test alpine sh
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # mkdir pasta_teste
/ # cd pasta_teste
/pasta_teste # touch arquivo_teste
/pasta_teste # echo "conteudo teste" > arquivo_teste
/pasta_teste # cat arquivo_teste
conteudo teste
/pasta_teste # exit
pedro-schneider@pedro-schneider-X550LA:~$ docker container commit docker-test docker-teste-img
sha256:d282161144c2f84cc077462376d98cdc298d90a935e1947740ca0f605677757f
pedro-schneider@pedro-schneider-X550LA:~$ docker images ls
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE

No images found matching "ls": did you mean "docker image ls"?
pedro-schneider@pedro-schneider-X550LA:~$ docker image ls
REPOSITORY         TAG       IMAGE ID       CREATED          SIZE
docker-teste-img   latest    d282161144c2   12 seconds ago   7.8MB
alpine             latest    91ef0af61f39   2 weeks ago      7.8MB
nginx              latest    39286ab8a5e1   6 weeks ago      188MB
hello-world        latest    d2c94e258dcb   17 months ago    13.3kB
pedro-schneider@pedro-schneider-X550LA:~$
```

A imagem “docker-teste-img” está listada como uma das imagens disponíveis na máquina host. Agora, é possível reconstruir o conteúdo do container docker-test em um novo container:

```
pedro-schneider@pedro-schneider-X550LA:~$  docker container run -it --rm --name docker-teste-2 docker-teste-img sh
/ # ls
bin          etc          lib          mnt          pasta_teste  root         sbin         sys          usr
dev          home         media        opt          proc         run          srv          tmp          var
/ # cd pasta_teste
/pasta_teste # ls
arquivo_teste
/pasta_teste # cat arquivo_teste
conteudo teste
/pasta_teste #
```

Vemos que a pasta “pasta_teste” e o arquivo “arquivo_teste” são corretamente reconstruídos.
OBS: a opção “--rm” foi adicionada para que o container fosse automaticamente removido após seu fechamento, por conveniência.

Gerando o arquivo .tar

As imagens são salvas no formato tar para facilitar a transmissão.
Com a Docker Engine, o comando para isso é o “save”:

```
pedro-schneider@pedro-schneider-X550LA:~/images$ docker image save -o saved-docker-teste-img.tar docker-teste-img
pedro-schneider@pedro-schneider-X550LA:~/images$ ls
saved-docker-teste-img.tar
```

A opção “-o” indica que o próximo parâmetro será o nome da imagem (com a extensão desejada, tar) e o último parâmetro é o nome da imagem que se quer salvar em tar.

Carregando a imagem de um tar

Para carregar a imagem de um arquivo tar, basta utilizar o comando “load”:

```
pedro-schneider@pedro-schneider-X550LA:~/images$ docker image rm docker-teste-img
Untagged: docker-teste-img:latest
Deleted: sha256:d282161144c2f84cc077462376d98cdc298d90a935e1947740ca0f605677757f
Deleted: sha256:f4bdd078d4c7b18a4538920c89e27bd158ad3fa2961182185ccae24706e33f13
pedro-schneider@pedro-schneider-X550LA:~/images$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
alpine        latest    91ef0af61f39   2 weeks ago     7.8MB
nginx         latest    39286ab8a5e1   6 weeks ago     188MB
hello-world   latest    d2c94e258dcb   17 months ago   13.3kB
pedro-schneider@pedro-schneider-X550LA:~/images$ docker image load -i saved-docker-teste-img.tar
c25182d54a77: Loading layer [==================================================>]  4.096kB/4.096kB
Loaded image: docker-teste-img:latest
pedro-schneider@pedro-schneider-X550LA:~/images$ docker image ls
REPOSITORY         TAG       IMAGE ID       CREATED          SIZE
docker-teste-img   latest    d282161144c2   17 minutes ago   7.8MB
alpine             latest    91ef0af61f39   2 weeks ago      7.8MB
nginx              latest    39286ab8a5e1   6 weeks ago      188MB
hello-world        latest    d2c94e258dcb   17 months ago    13.3kB
pedro-schneider@pedro-schneider-X550LA:~/images$
```

No exemplo acima, a imagem é removida e depois recarregada com o arquivo tar gerado anteriormente.
A opção “-i” indica que o próximo parâmetro é o nome do arquivo do qual se deseja carregar a imagem.

Consulta do histórico da imagem

É possível consultar o histórico de uma imagem. É organizado por camadas, mostrando quando e como cada uma foi criada, além do tamanho em disco.

```
pedro-schneider@pedro-schneider-X550LA:~$ docker image history docker-teste-img
IMAGE          CREATED        CREATED BY                                      SIZE      COMMENT
d282161144c2   20 hours ago   sh                                              132B
<missing>      2 weeks ago    /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      2 weeks ago    /bin/sh -c #(nop) ADD file:5758b97d8301c84a2…   7.8MB
pedro-schneider@pedro-schneider-X550LA:~$
```

Exportando o container

É possível exportar o container diretamente em forma de arquivo tar com o comando export:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container export my-container -o my-container-exp.tar
pedro-schneider@pedro-schneider-X550LA:~$ ls
anaconda3  Desktop    Downloads  Music                pasta_teste  Public  Templates
bkp        Documents  images     my-container-exp.tar  Pictures     snap    Videos
pedro-schneider@pedro-schneider-X550LA:~$ docker container rm my-container
my-container
pedro-schneider@pedro-schneider-X550LA:~$ docker image import my-container-exp.tar my-container-imp
sha256:d116f3271a67462c4c362a0ccab7c4bd8b64e3017fe50b13538c4aeefc7f005f
pedro-schneider@pedro-schneider-X550LA:~$ docker image ls
REPOSITORY         TAG       IMAGE ID       CREATED          SIZE
my-container-imp   latest    d116f3271a67   7 seconds ago    78.1MB
docker-teste-img   latest    d282161144c2   20 hours ago     7.8MB
alpine             latest    91ef0af61f39   2 weeks ago      7.8MB
ubuntu             latest    b1e9cef3f297   4 weeks ago      78.1MB
nginx              latest    39286ab8a5e1   6 weeks ago      188MB
hello-world        latest    d2c94e258dcb   17 months ago    13.3kB
pedro-schneider@pedro-schneider-X550LA:~$ docker container run -it --name my-container my-container-imp sh
# ls
bin   dev  home  lib64  mnt  proc  run   srv  test_dir  usr
boot  etc  lib   media  opt  root  sbin  sys  tmp       var
# exit
pedro-schneider@pedro-schneider-X550LA:~$
```

A importação ocorre criando-se uma imagem a partir do tar e depois utilizando essa imagem para criar um novo container.

OBS: a diferença desse método para o que exporta uma imagem em tar, com o comando “save”, é que, com o método que utiliza o comando “export”, é exportado apenas o sistema de arquivos do container em tar, sem as camadas da imagem nem metadados que se tem na geração da imagem com o comando “commit”.

Outros comandos

O comando inspect mostra diversos metadados sobre a imagem, como o sha256 das camadas, a versão da docker engine em que foi gerada etc.

```
docker image inspect nome_da_imagem
```

O comando prune exclui todas as imagens que não foram utilizadas para gerar um container existente, parado ou rodando

```
docker image prune
```

O comando pull importa a imagem do repositório do Docker Hub. Por exemplo, para importar a alpine:

```
docker image pull alpine
```

O comando push faz o upload da imagem para a conta do usuário no Docker Hub.

```
docker image push identificador_da_imagem
```

Dockerfile

Um Dockerfile é um documento de texto com especificações de camadas para a construção de uma imagem.

Para construir uma imagem a partir de um Dockerfile, utiliza-se o comando build. Essa é a maneira usual de se gerar imagens, ao invés do uso do “commit”.

Gerando o Dockerfile

Para o exemplo, um Dockerfile simples com 2 camadas será gerado: um para importar os arquivos básicos da imagem ubuntu, e outra com um comando RUN para a instalação do vim.

```
pedro-schneider@pedro-schneider-X550LA:~$ nano Dockerfile
pedro-schneider@pedro-schneider-X550LA:~$
```

```
pedro-schneider@pedro-schneider-X
  GNU nano 7.2                                  Dockerfile
FROM ubuntu
RUN apt-get update && apt-get -y install vim
```

Construindo a imagem

Para construir a imagem a partir desse Dockerfile, utiliza-se o comando “build”:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker image build -t my-ubuntu .
[+] Building 21.0s (6/6) FINISHED                                   docker:default
 => [internal] load build definition from Dockerfile                         0.0s
 => => transferring dockerfile: 94B                                          0.0s
 => [internal] load metadata for docker.io/library/ubuntu:latest             0.0s
 => [internal] load .dockerignore                                            0.0s
 => => transferring context: 2B                                              0.0s
 => CACHED [1/2] FROM docker.io/library/ubuntu:latest                        0.0s
 => [2/2] RUN apt-get update && apt-get -y install vim                      19.6s
 => exporting to image                                                       1.2s
 => => exporting layers                                                      1.2s
 => => writing image sha256:9195979a7a075a5de081e3bc8b5ffd8828f256332bf83df0789495c89c28695d  0.0s
 => => naming to docker.io/library/my-ubuntu                                 0.0s
pedro-schneider@pedro-schneider-X550LA:~$
```

A opção -t indica a determinação de um título (nome da imagem) como próximo parâmetro.
O “.” indica que o diretório de contexto (onde está o Dockerfile) é o atual.

Volumes

Como visto anteriormente, volumes são diretórios externos ao containers que mapeiam seu sistema de arquivos numa via de duas mãos.
Mas o que acontece se um volume for criado sem um diretório especificado?

Diretório padrão

Dado um container da imagem alpine cuja pasta data foi mapeada a um volume sem diretório especificado:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container run -it --name test -v /data alpine sh
/ # ls
bin    data   dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # cd data
/data # touch test_file
/data # exit
```

Um arquivo é criado dentro de data. Esse arquivo deve estar presente no volume também, mas onde exatamente?
Para isso, é preciso inspecionar o volume com o comando “inspect”. Mas, para isso, é necessário saber o nome do volume, que pode ser descoberto listando os volumes:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker volume ls
DRIVER    VOLUME NAME
local     b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538
```

Inspecionando, encontrado o ponto de montagem:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker volume inspect b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538

[
    {
        "CreatedAt": "2024-09-27T15:15:17-03:00",
        "Driver": "local",
        "Labels": {
            "com.docker.volume.anonymous": ""
        },
        "Mountpoint": "/var/lib/docker/volumes/b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538/_data",
        "Name": "b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538",
        "Options": null,
        "Scope": "local"
    }
]
```

Esse é o diretório de volumes padrão do Docker. Para acessá-lo, é necessário permissão de root:

```
root@pedro-schneider-X550LA:/home/pedro-schneider# cd /var/lib/docker/volumes/b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f
819e51a20bc538/_data
root@pedro-schneider-X550LA:/var/lib/docker/volumes/b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538/_data# ls
test_file
root@pedro-schneider-X550LA:/var/lib/docker/volumes/b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538/_data# echo
 "test content" > test_file
root@pedro-schneider-X550LA:/var/lib/docker/volumes/b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538/_data# exit
exit
```

Um novo conteúdo foi adicionado ao arquivo criado pelo container. Agora, voltando ao container:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker container start test
test
pedro-schneider@pedro-schneider-X550LA:~$ docker container attach test
/ # cd data
/data # cat test_file
test content
/data #
```

Verificamos que o conteúdo de fato está lá.

Nome do volume

É possível criar um volume com um nome personalizado antes de mapeá-lo a um container:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker volume create my-volume
my-volume
pedro-schneider@pedro-schneider-X550LA:~$ docker volume ls
DRIVER    VOLUME NAME
local     b522c8048477d50fc4b0b67c854f36910b2a28128e4c6f8c2f819e51a20bc538
local     my-volume
pedro-schneider@pedro-schneider-X550LA:~$ docker run -it --name test2 -v my-volume:/data alpine sh
/ # cd data
/data # touch test_file
```

OBS: repare que é o nome do volume que é passado, não o diretório. Se passa o diretório apenas quando o volume não está no diretório padrão do Docker.

Agora, indo ao ponto de montagem do volume:

```
pedro-schneider@pedro-schneider-X550LA:~$ docker volume inspect my-volume
[
    {
        "CreatedAt": "2024-09-27T15:26:15-03:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/my-volume/_data",
        "Name": "my-volume",
        "Options": null,
        "Scope": "local"
    }
]
pedro-schneider@pedro-schneider-X550LA:~$ sudo su
root@pedro-schneider-X550LA:/home/pedro-schneider# cd /var/lib/docker/volumes/my-volume/_data
root@pedro-schneider-X550LA:/var/lib/docker/volumes/my-volume/_data# ls
test_file
root@pedro-schneider-X550LA:/var/lib/docker/volumes/my-volume/_data#
```

Verificamos que o arquivo de fato está lá.

Outros comandos

O comando rm remove o volume passado de argumento, desde que ele não esteja mapeado com um container existente:

```
docker volume prune nome_do_vol
```

O comando prune remove todos os volumes que não estão mapeados a nenhum container existente:

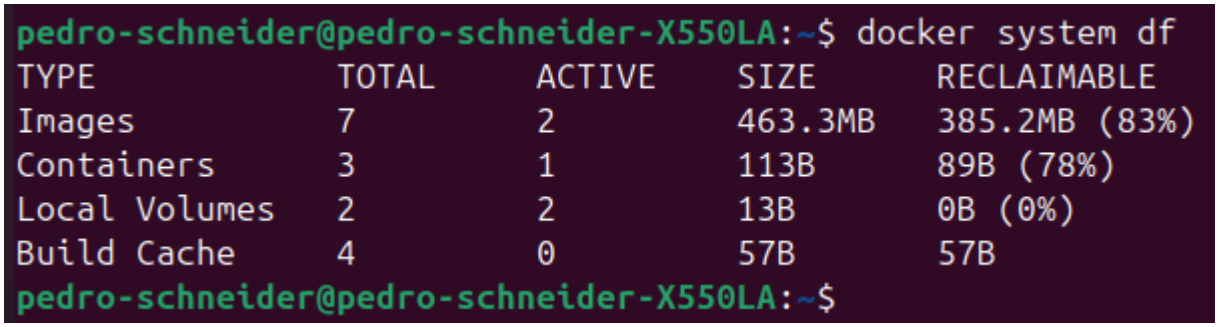
```
docker volume prune
```

Docker System

Outro objeto disponível na CLI é o system. Esse objeto organiza e retorna informações sobre o sistema do Docker como um todo.

Espaço em disco

É possível verificar a ocupação de espaço em disco pelo Docker com o comando df:



Informações gerais

O comando info retorna diversas informações sobre a Docker Engine e a máquina hospedeira. Alguns exemplos: versão da Engine, alocação de containers, plugins, SO do host, arquitetura do host etc.

```
docker system info
```

Prune

O comando prune no objeto system deleta:

- todos os containers parados
- todos os volumes não mapeados
- todos as redes não utilizadas
- todas as imagens não utilizadas

É necessário tomar cuidado ao utilizar esse comando, pelo escopo das deleções.

Uma alternativa mais segura é utilizar o comando prune objeto por objeto:

```
docker container prune
docker image prune
docker volume prune
docker network prune
```

Assim, se tem mais controle sobre o que está sendo deletado a cada linha de comando.