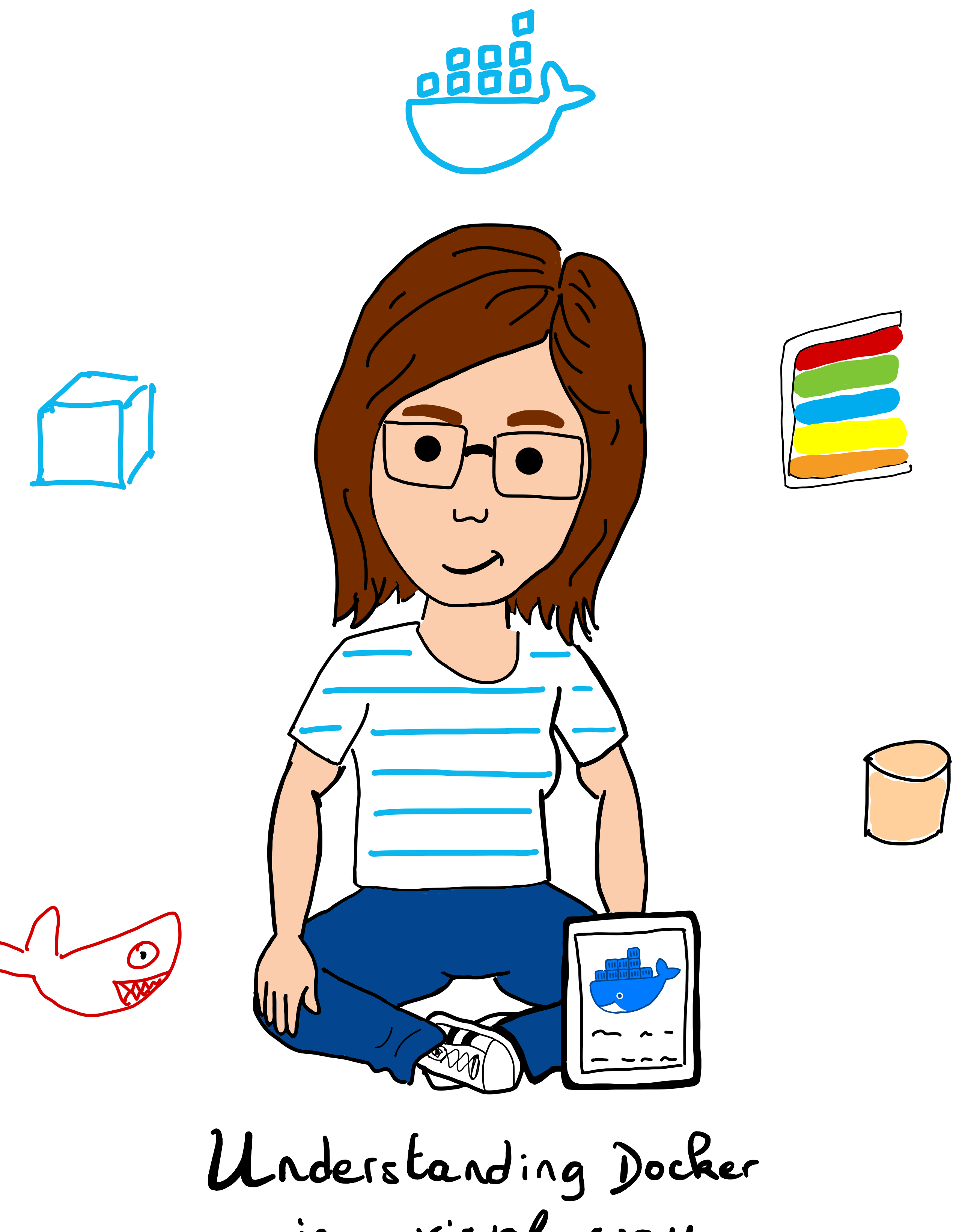
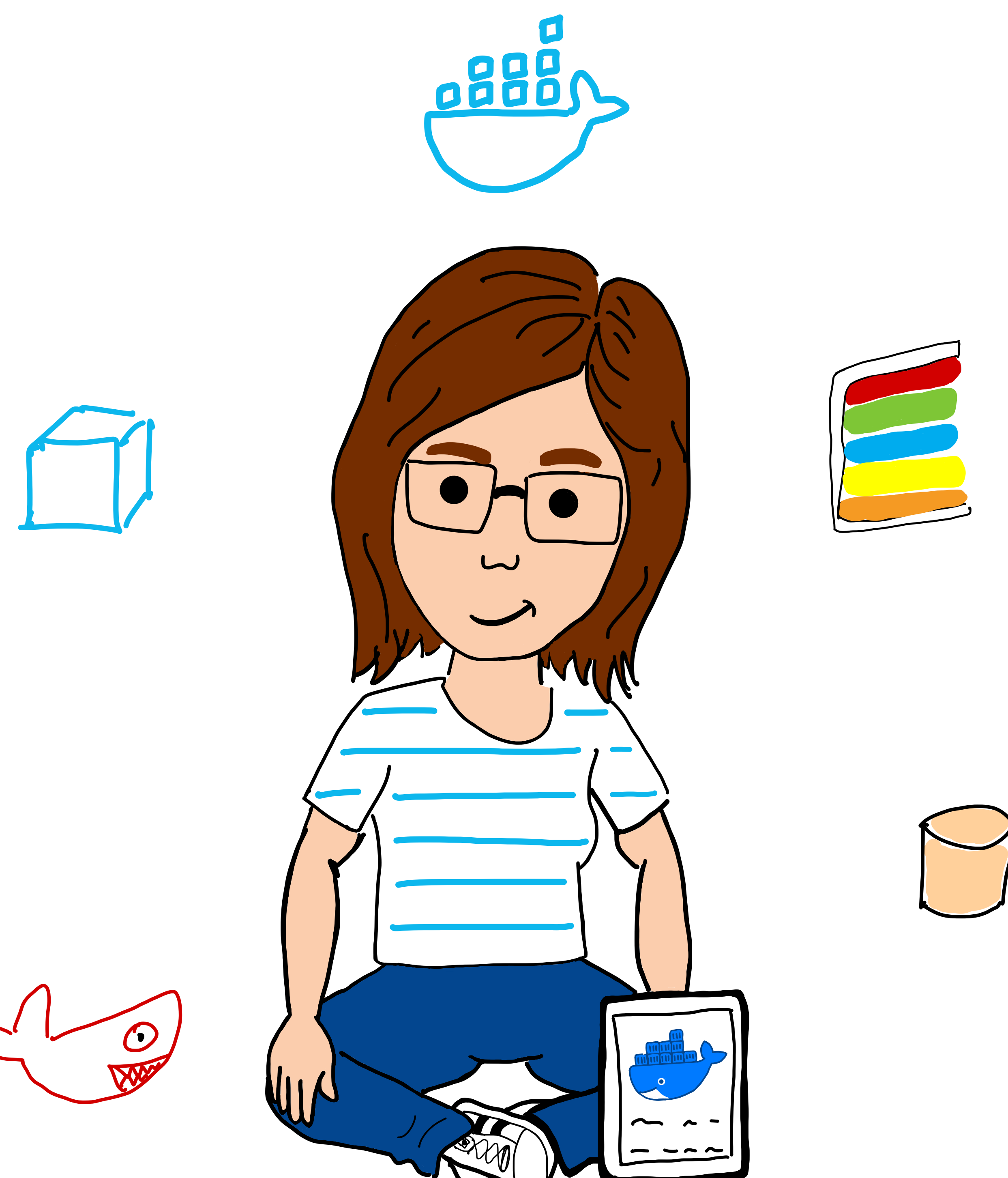


Understanding Docker in a visual way

Learn & discover Docker in sketchnotes
- with some tips included -

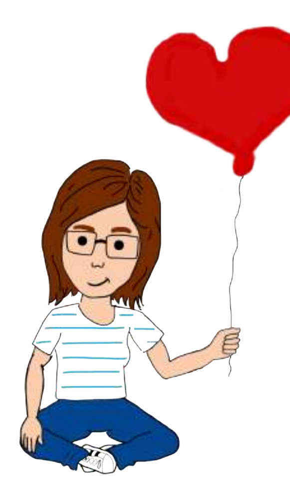
Aurélie Vache



Understanding Docker in a visual way

Aurélie Vache

Special thanks to:



Laurent, my husband, Alexandre, my son, and all the caring people who believe in me everyday :)

Thank you so much !!!

Reviewer:

Thanks to Gaelle Acas, Stéphane Philippart & Alexandre Touret who took the time to review this book.

Changelog:

Release Date	:	31/01/2021
Updated Date	:	31/01/2023
Release Version	:	2.0.0
Changelog	:	Re(writ|draw)ing
Length	:	237 pages
Docker	:	20.10.21

And ...

Thanks to my cat, "Sushi", who did a lot "Purr sketchnoting" with me ❤

Licence :

Creative Commons BY-NC-ND

http://creativecommons.org/licenses/by-nc-nd/3.0/

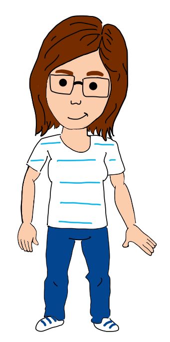
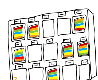
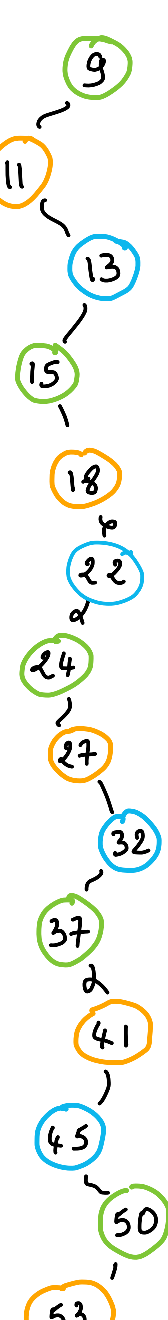


Table of contents

- Dockerize everything? — 9
- What is Docker? — 11
- Docker Engine components — 13
- Image > — 15
  - Dangling images — 18
  - Build images — 22
  - Push images — 24
  - Pull & retrieve images — 27
- Layers — 32
-  Registry — 37
- Container > — 41
  - Restart policies — 45
  - Exit status — 50
  - Run a container — 53
  - Copy to / from a container — 60



- Exec in a container 62
- Stop & restart a container 64
- Pause & unpause a container 66

Volume 69

Event 74

Search 77

Security 79

- Scan vulnerabilities 79
- SBOM 81
- Content Trust 84
- Run a container with privileged mode 89
- Run as non-root user 97

Stats 105

Clean & Purge 107

Memory & CPU constraints 112

Context 121

Network 125

System 131

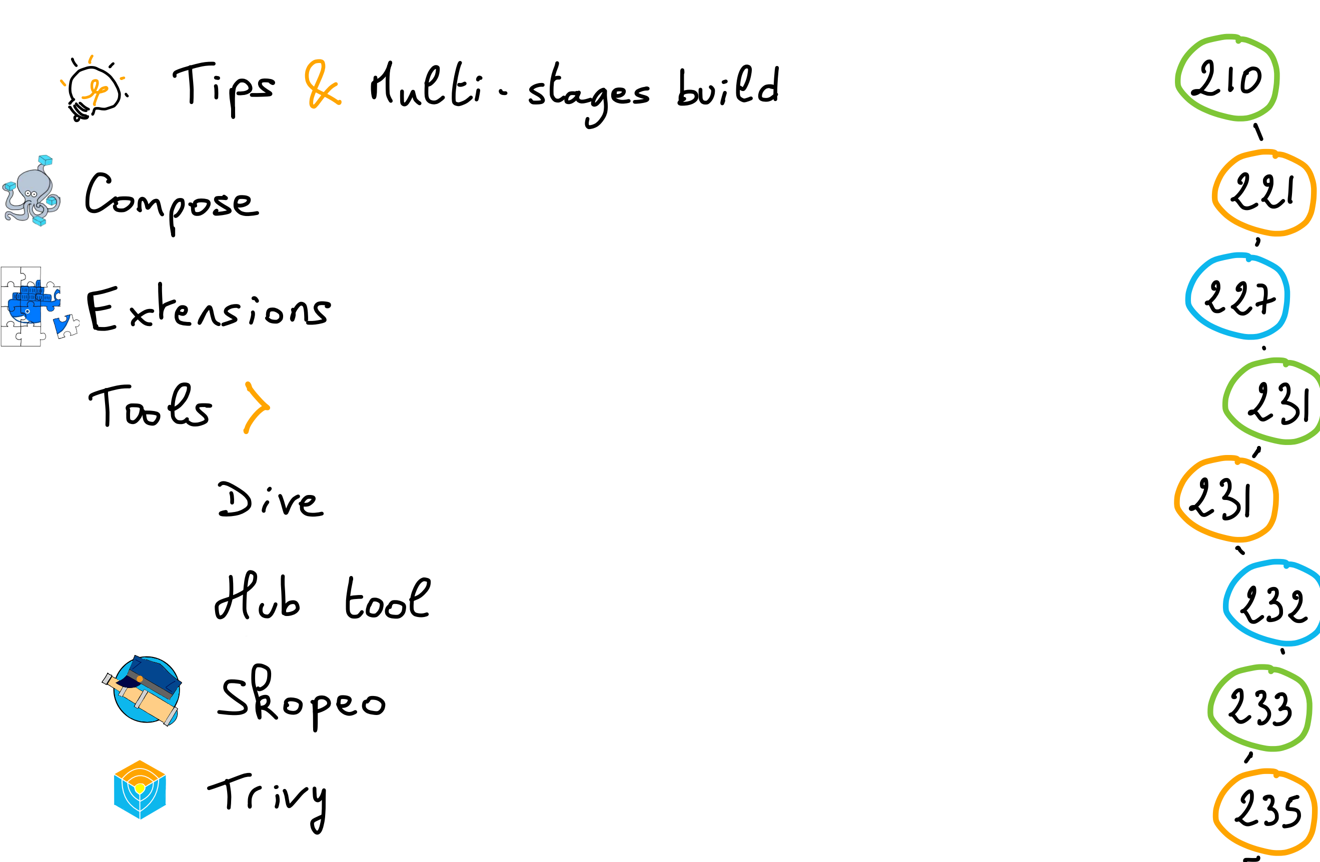
Manifest 135

Debugging / Troubleshooting 139

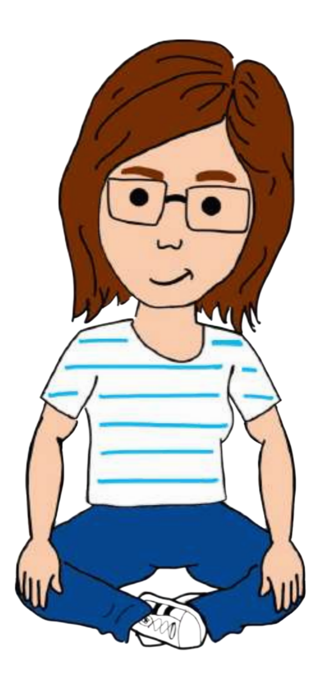
Dockerignore 145

DockerFile > 153

- FROM 155
- LABEL 157
- ARG 160
- ENV 164
- ADD & COPY 169
- RUN 173
- CMD : String vs JSON syntax 175
- CMD & ENTRYPOINT 178
- EXPOSE 183
- VOLUME 186
- USER 190
- WORKDIR 194
- ONBUILD 197
- HEALTHCHECK 200
- STOPSIGNAL 204
- SHELL 208


Tips & Multi-stages build 210
Compose 221
Extensions 227
Tools > 231
Dive 231
Hub tool 232
Skopeo 233
Trivy 235
In the same collection :) 237

Disclaimer

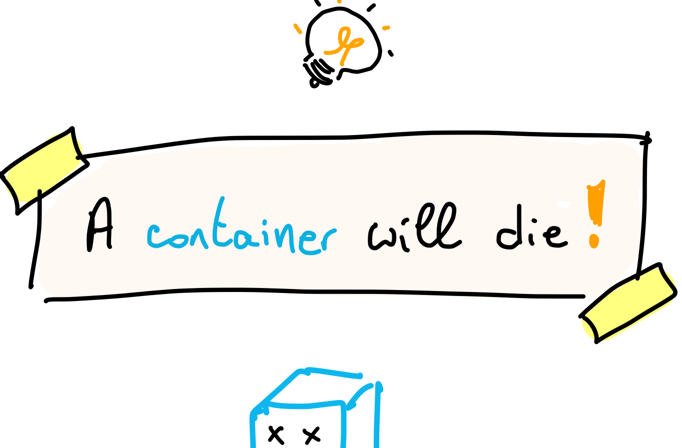


I'm just a Cloud & technologies lover who loves helping people & sharing to others what I know and understand.
I love trying explaining complex technical things in a visual way, in a "sketchnotes" way.

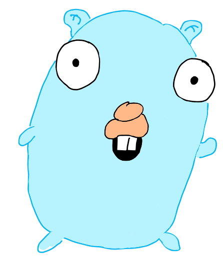
Hope you'll enjoy this book :)

If I want to run an app in a container, it's easy. I only need to find the good base image!

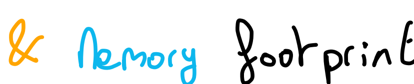
Well, let's remind things that matter & good practices ...

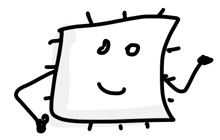


- Scalability
- Ephemerality
- Stateless
- Being resilient
- Keeping images size as small as possible



→ With a small CPU

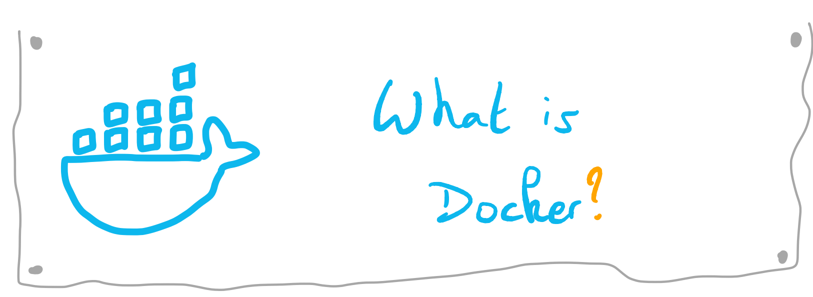




→ With a fast startup

→ With clean shutdown support

→ That allows you to create: static executable binaries



create containers & container-based apps

Open-Source

Docker

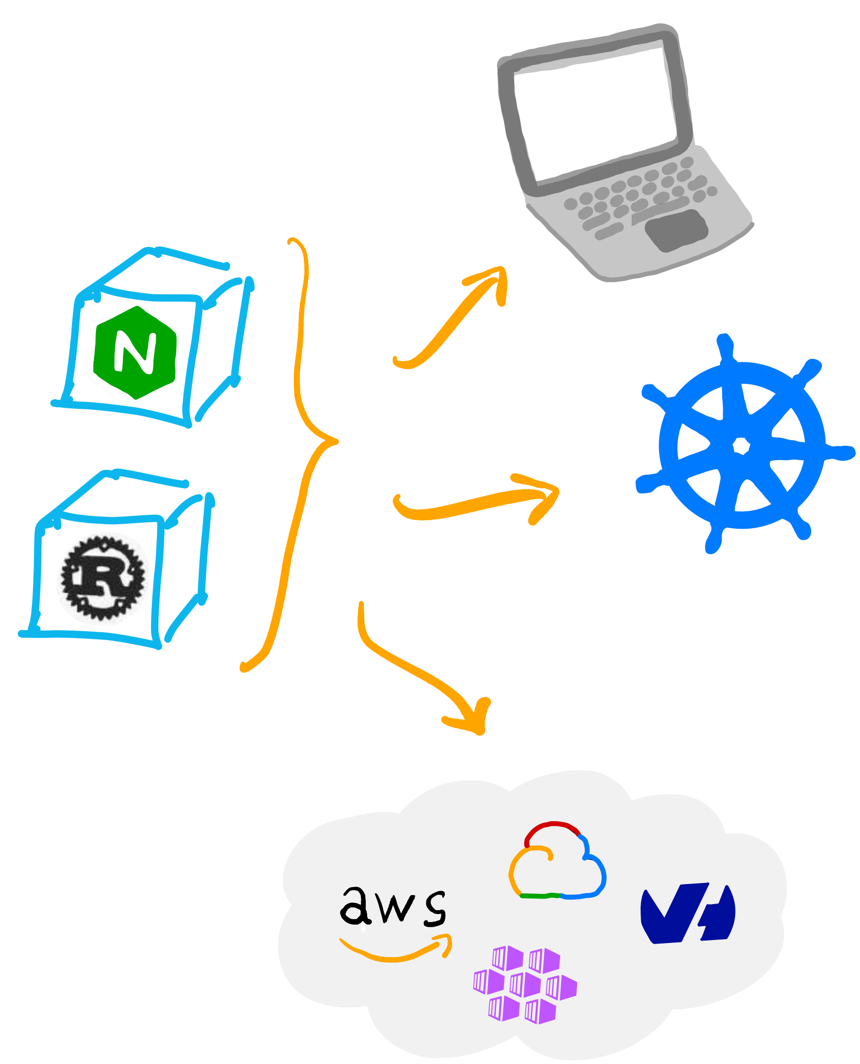
Enable portability

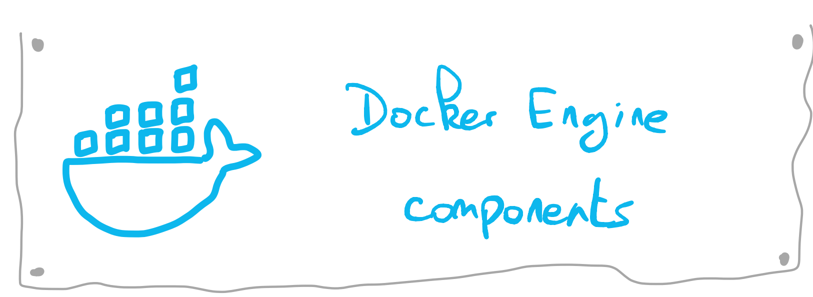
Platform for developping, shipping & running apps

Build / package one-time

&

then run whenever you want:





$ docker

CLI

REST API

$ dockerd

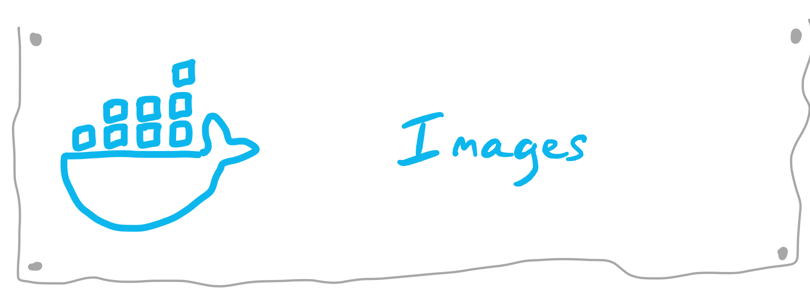
manages

Dockerd

- Docker daemon listens for API requests & creates / edits / deletes objects
- Can communicate with other daemons

Client

- Primary way to interact with Docker Engine
- docker CLI send requests (run, remove ...) to API
- Can communicate with more than one daemon



→ Images are identified by a name & a tag

repository : tag

Imagine images like a cake with layers

layer 5a13fe8
layer 8b2ac42
layer 0c41bd7
layer 4612c2f
layer 7c13f42

→ An image consists of everything needed to run an application : code, binaries, tools, runtimes, dependencies ...

→ Layers can be reused by images

Should I need to download all layers each time I pull an image?

No! Layers already available locally won't be downloaded again

How To ?

- List all images available (pulled or created myself)

```
$ docker images
```

- List all Alpine images

```
$ docker images alpine
```

- List all images with a name starting with "busy" & finishing by "libc"

```
$ docker images --filter=reference='busy*:*libc'
```

- List all unused & tagged images (dangling images)

```
$ docker images -f dangling=true
```

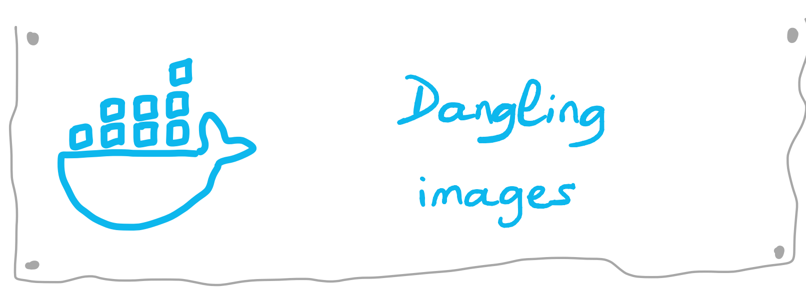
-f / --filter

- Remove ubuntu:latest image

```
$ docker rmi ubuntu:latest
```

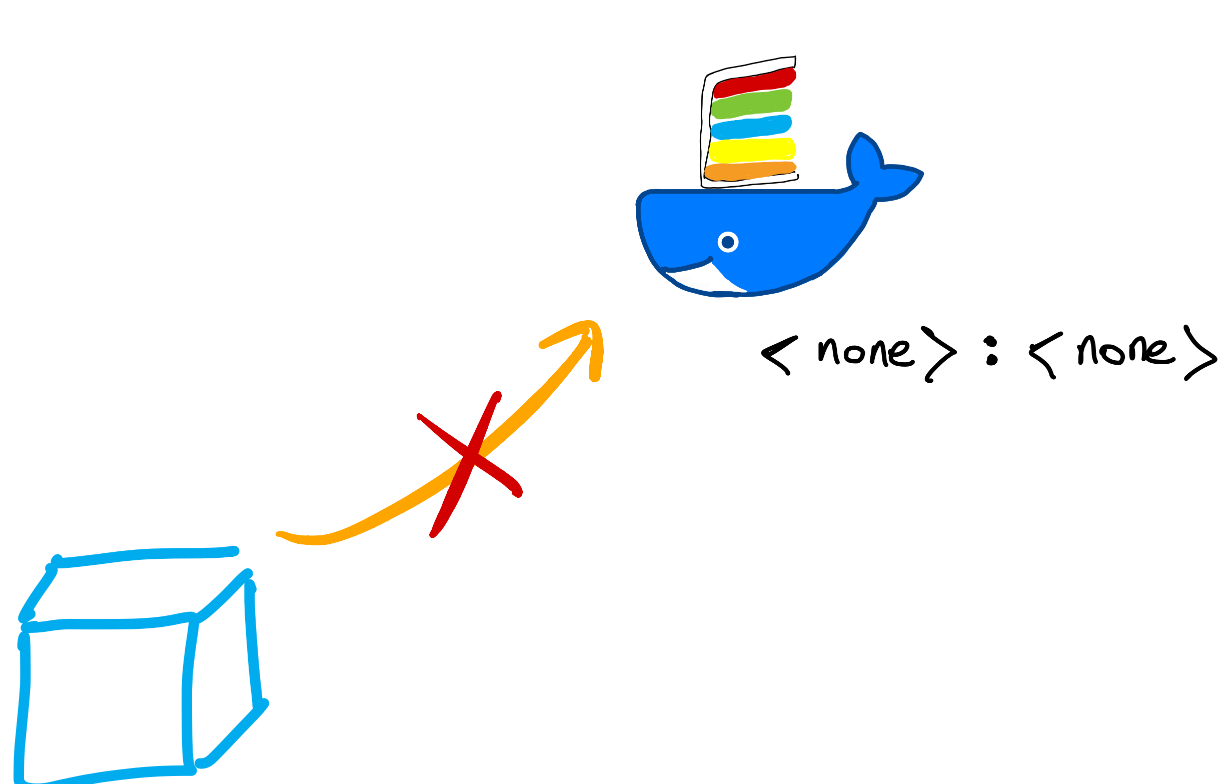
- Remove ubuntu:latest image (even if a running container exists)

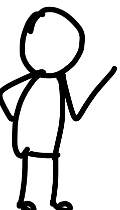
```
$ docker rmi ubuntu:latest --force
```



→ Dangling images are not tagged

& not used by any container





Dangling images are also known as "<none>:<none>" images

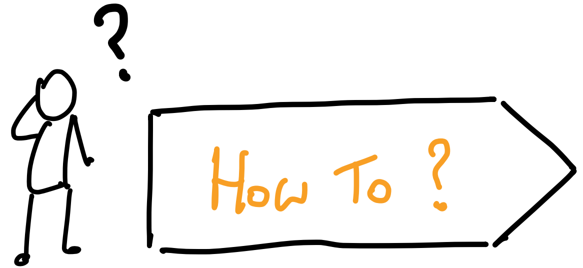
Because their repository & tag are empty. They are not referenced by any images or containers

Why?

```
REPOSITORY TAG IMAGE ID CREATED SIZE
<none> <none> 1234a56b78e 2 days ago 90MB
```

Theses images take some place in disk space.

You can remove them with docker system prune command.



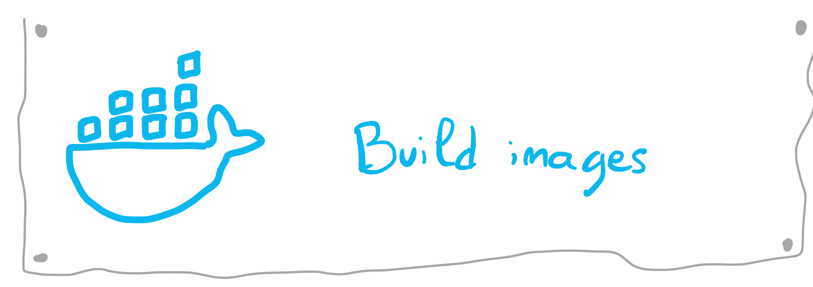
- List only dangling images

```
$ docker images -f dangling=true
```

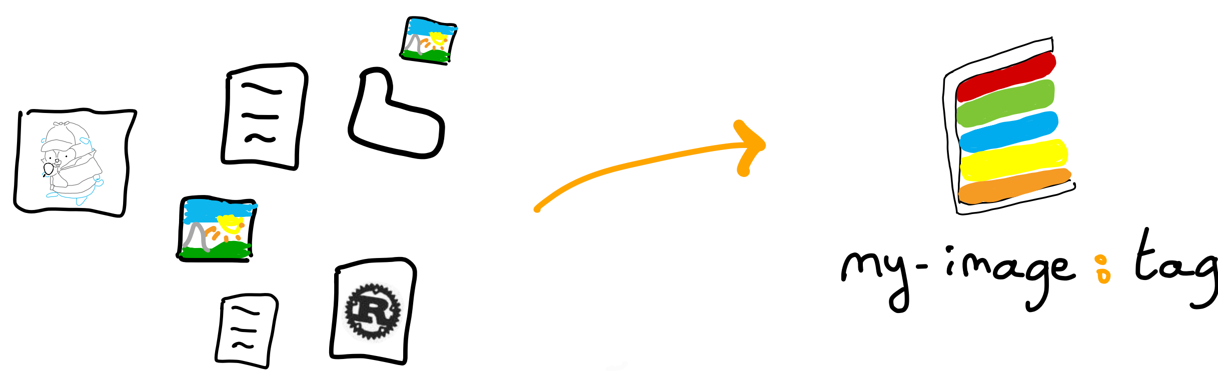
- Remove all dangling images

```
$ docker image prune
```

⚠ Without accepting the prompt, you can also list/display them



→ In order to run an app in a container, we need to build / package it in an image



→ docker build command builds images from a DockerFile & a context

A context can be a set of files in a path or in an URL

→ By default, docker build command search a DockerFile at the root of your path (case sensitive :))

- Build an image

```
$ docker build .
```

- Build an image & tagged it with my-awesome-image and 1.0.1 tag

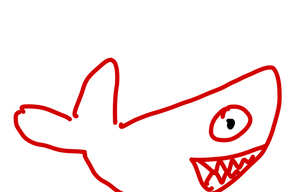
```
$ docker build -t my-awesome-image:1.0.1 .
```

- Build an image according to a specified DockerFile

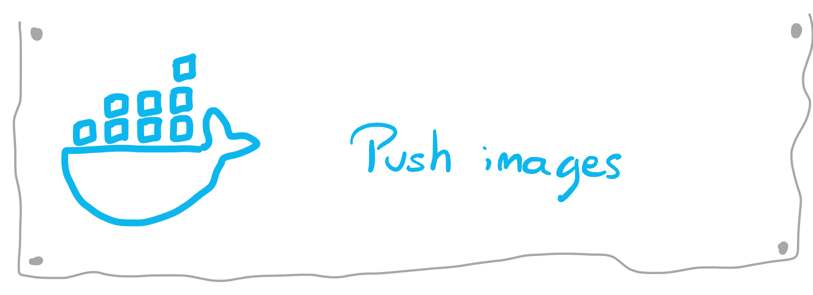
```
$ docker build -f build/DockerFile -t my-image:tag .
```

- Build an image from a Git repository (main branch, helloworld/Dockerfile file path)

```
$ docker build https://github.com/scraly/gcp-cloud-run
-demo.git#main -f helloworld/Dockerfile
```

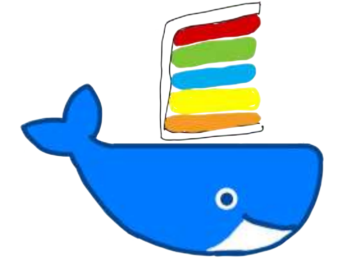


Ok, let's git clone this repository, locate the main branch & the DockerFile in the specified path



→ In order to access images through DockerHub or a private registry, you need to push them

2 push to the registry



1 tag

By default, Docker daemon pushes 5 layers of an image at once.

Set it through --max-concurrent-uploads

How To ?

- Tag an image that will be hosted in a registry :

```
$ docker tag my-img:my-tag my-registry/my-img:my-tag
```

- Push an image to a registry :

```
$ docker push my-registry/my-img:my-tag
```

→ Images are stored in a Docker registry

→ A registry can contains several images & an image can have several tags

→ By default, Docker pulls images from Docker Hub

I want to retrieve / pull the latest image of Ubuntu

```
$ docker pull my-repo/ubuntu:latest
```

name tag

Ok... but I see you are unauthenticated to my repo

No problem, authenticate me in your repository

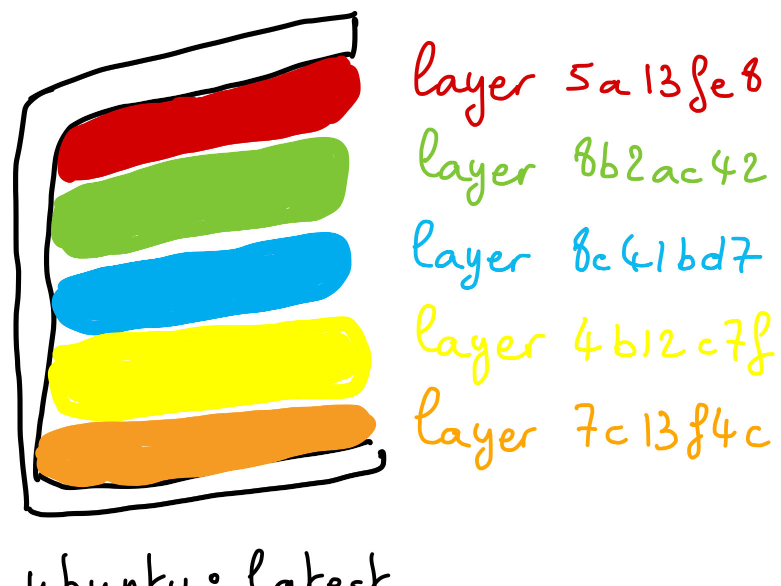
```
$ docker login
```

Perfect, you are authenticated to this registry

Now, can I have the latest image of Ubuntu?

```
$ docker pull my-repo/ubuntu:latest
```

Sure!
Here is the image with its layers



How To ?

> Pull last Ubuntu tagged image

```
$ docker pull ubuntu
```

If tag is not defined, default tag is latest

> Pull all tagged images from a repository/name

```
$ docker pull --all-tags ubuntu
```

> Pull an image from a private registry

```
$ docker pull myprivateregistry:8080/user/image-name:tag
```

> List images available locally

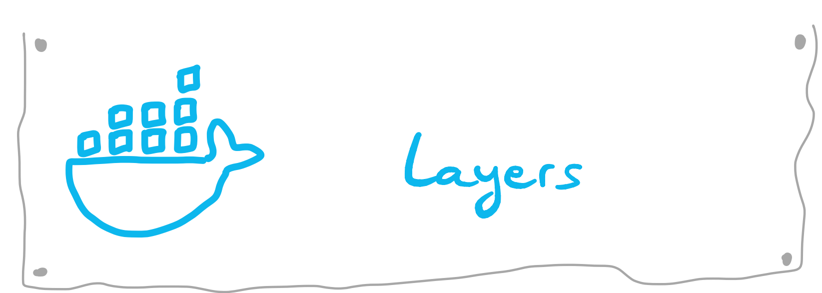
```
$ docker images
```

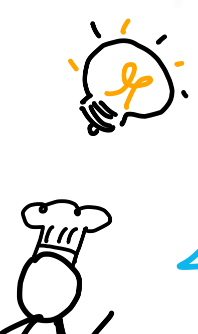
By default, private registries are reached through HTTPS.

You can define them as "insecure" registries.

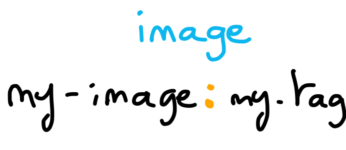
~/.docker/daemon.json

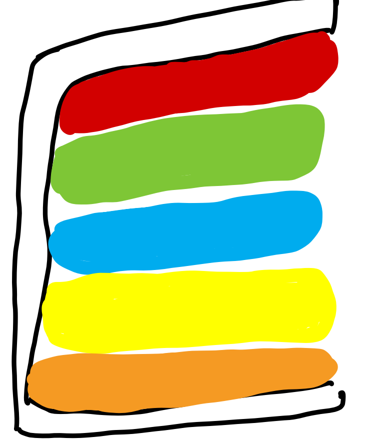
```
{
  "debug" : true,
  "insecure-registries" : ["myregistry.com:5000"],
  "experimental" : true
}
```





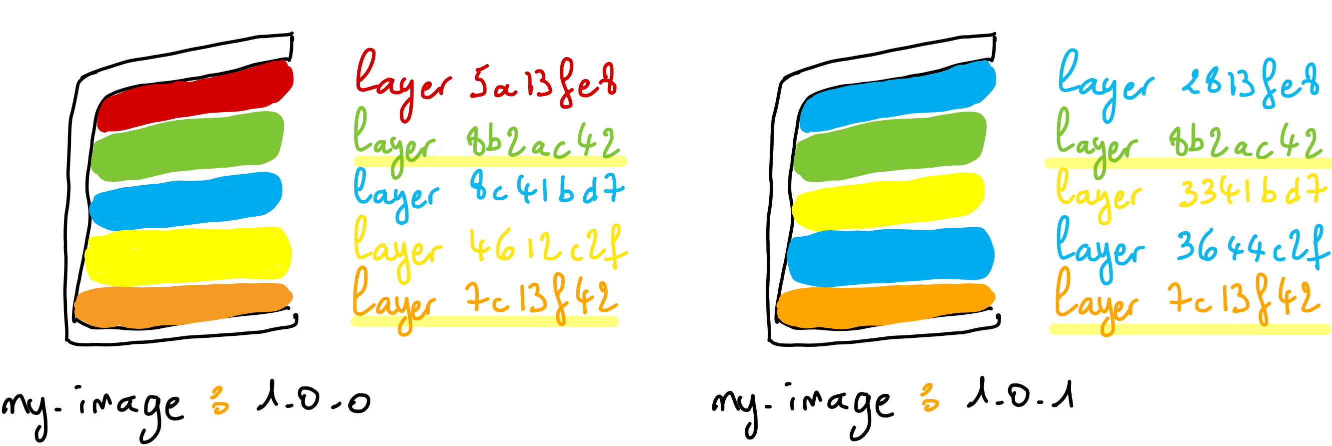
Imagine layers like a part/layer of a cake





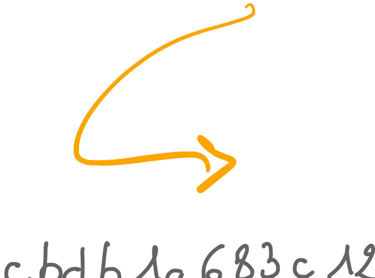
layer 5a13fe8
layer 8b2ac42
layer 8c41bd7
layer 4612c2f
layer 7c13f42

→ Layers can be reused by images



→ Layers are "read-only" / immutables

→ Layers are identified by a sha256 unique hash



cbdb1a683c12

DockerFile steps don't necessary create layers.
Others are :

RUN

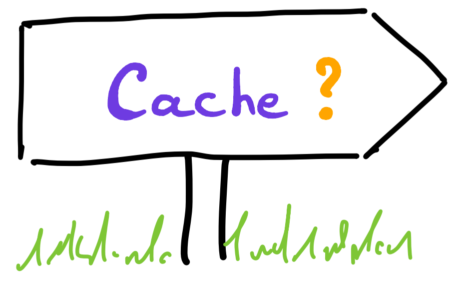
ADD

COPY

Other are intermediate layers.

→ Intermediate layers are deleted during docker build

→ Intermediate layers have a ∅ B size

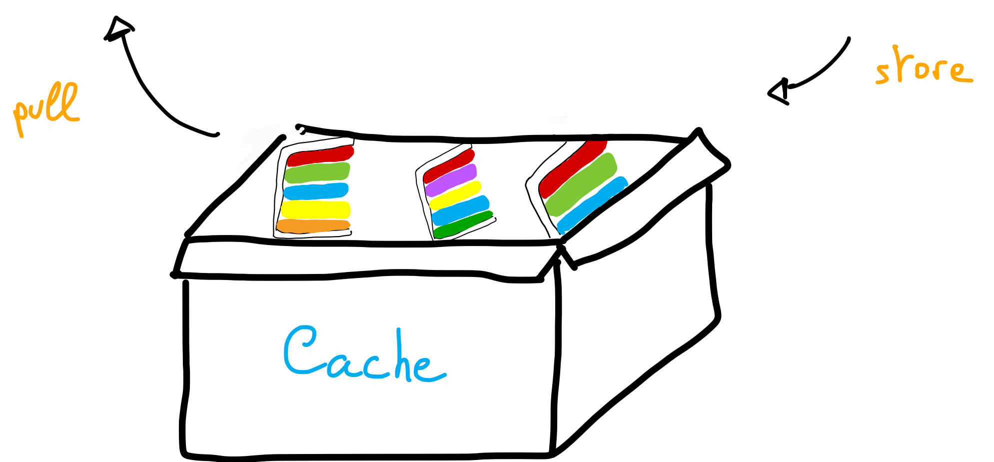


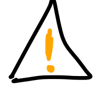
I want to build my image from a DockerFile

```
$ docker build . -t my-image
```

1. layer already exists in local cache

2. layers doesn't exist, let's build it !





FROM instruction doesn't use the cache

How To ?

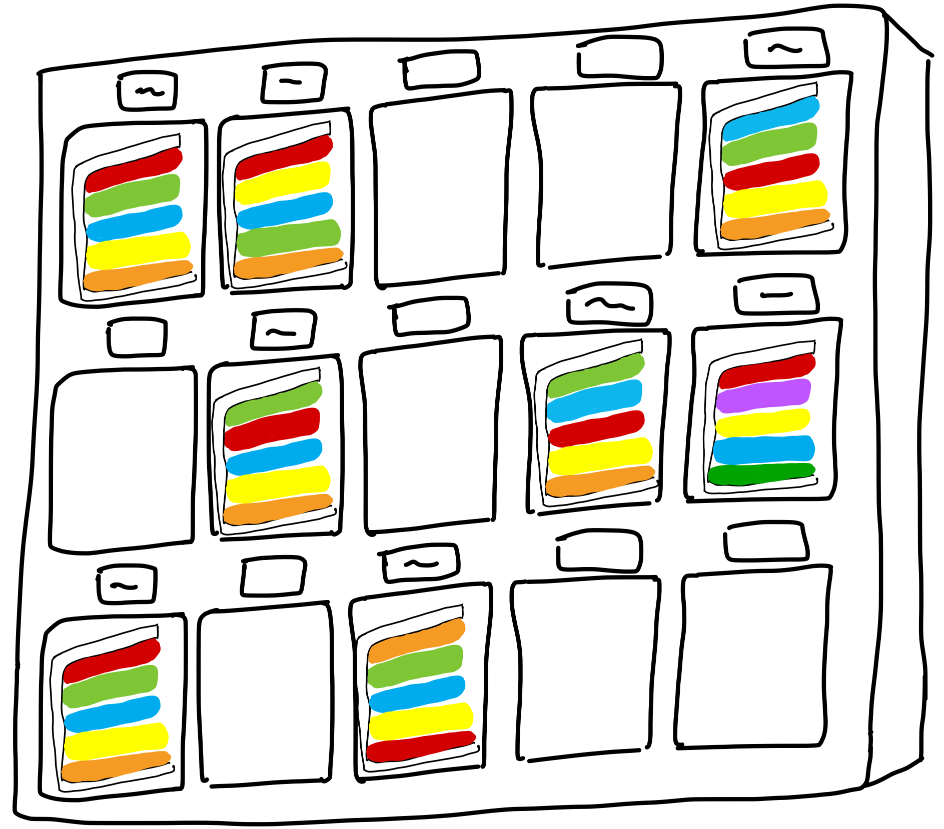
- List all layers for a given image (and their size)

```
$ docker history my-image --no-trunc
```

- Display layers digest for an image

```
$ docker inspect -f '{{json .RootFS.Layers}}'
scraly/simpleapp | jq .
```

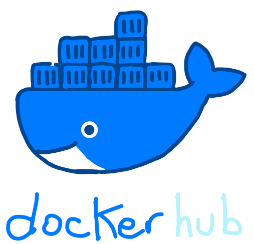
→ A registry is a storage for images



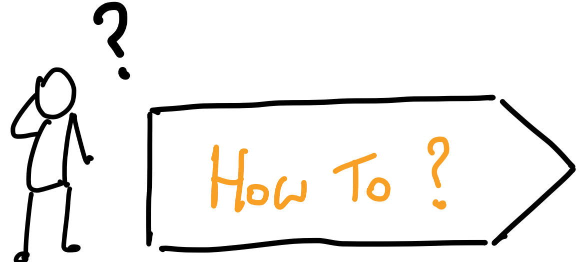
→ A registry can contain several images

& an image can have several tags

→ You can run your own registries or use existing ones like DockerHub



By default, Docker pulls images from DockerHub registry



› Pull an image from a registry

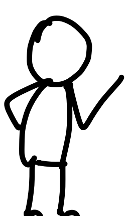
```
$ docker pull my-registry/my-image:my-tag
```

› Tag an image that will be hosted in a registry

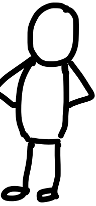
```
$ docker tag my-img:my-tag my-registry/my-img:my-tag
```

› Push an image to a registry

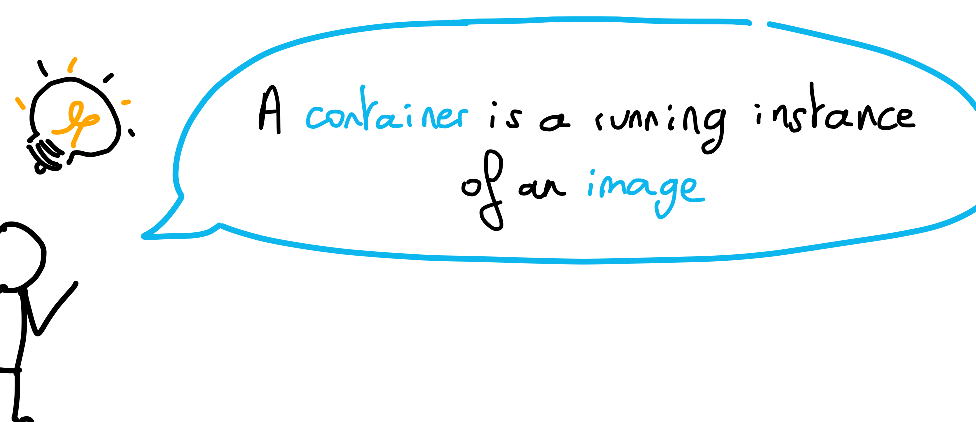
```
$ docker push my-registry/my-img:my-tag
```

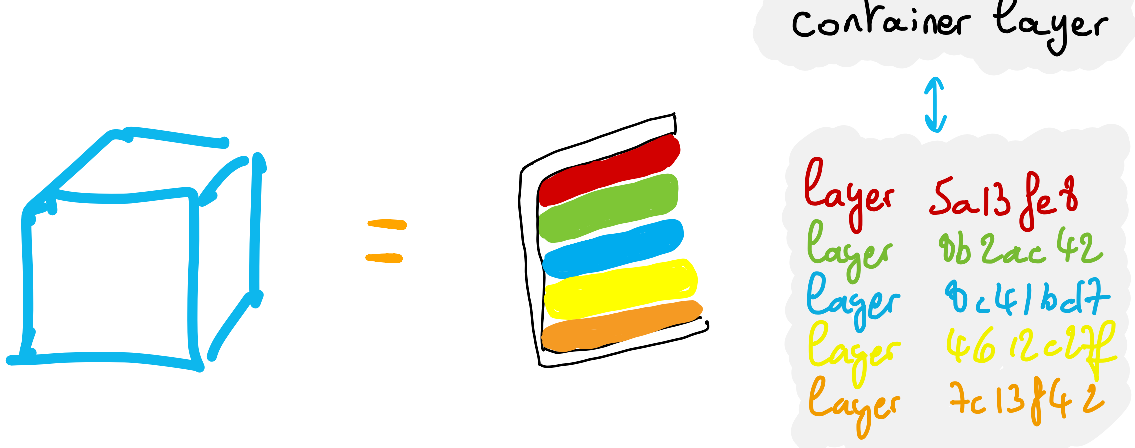



```
$ docker image remove my-registry.com/my-image:my-tag
```

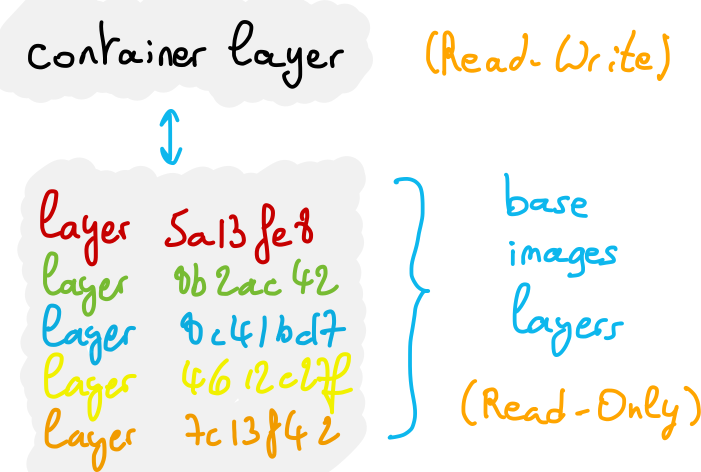



Container


A container is a running instance of an image

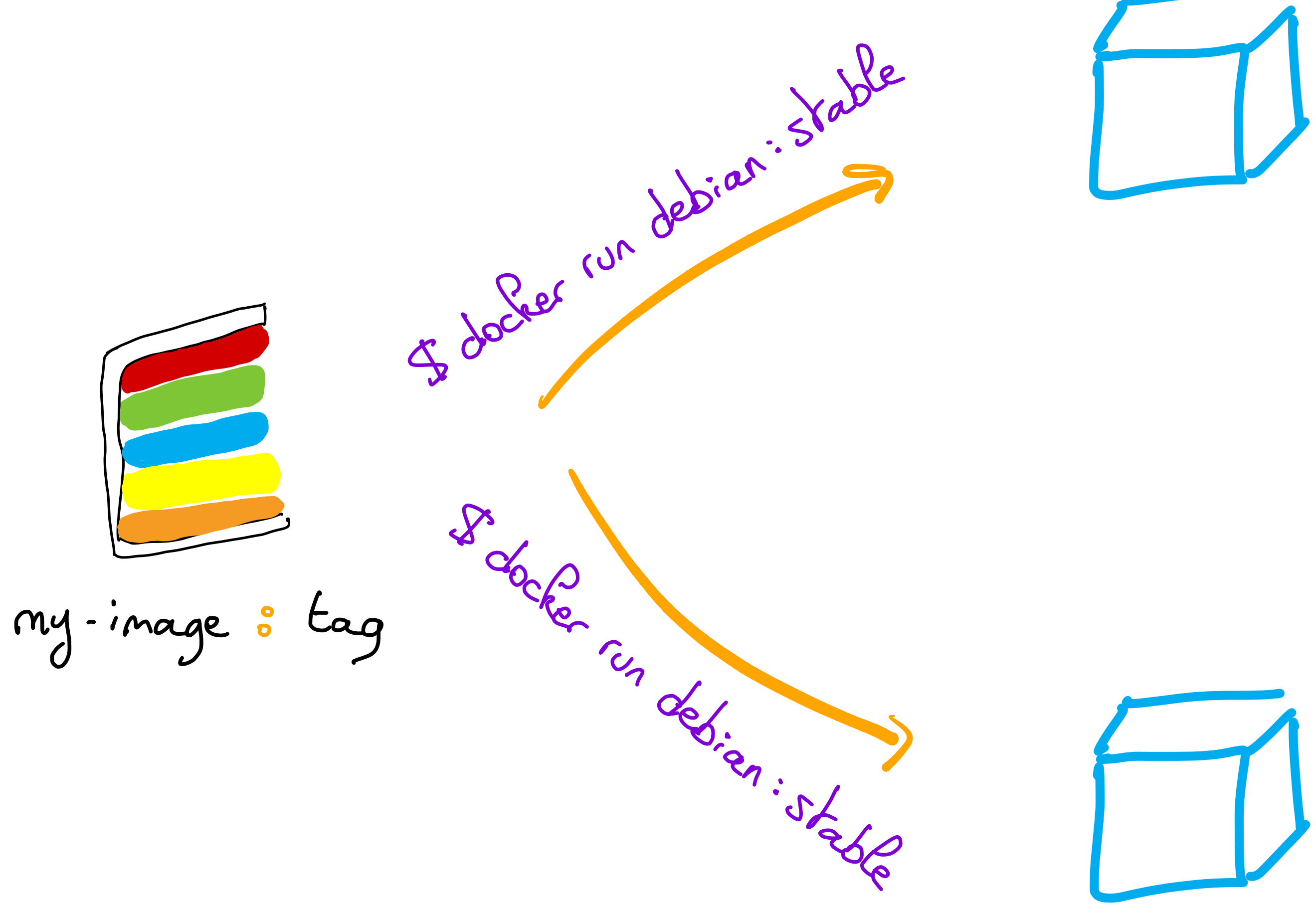

container layer
layer 5a13fe8
layer 0b2ac42
layer 8c41bd7
layer 4612c2f
layer 7c13f42

→ An application running in a container can only modify the container layer



Content stored in this container layer is lost when container is no longer running.

→ 2 docker run will create 2 distinct containers



How To ?

- Create a container with a random name

```
$ docker container create my-image
```

- Create a container

```
$ docker container create --name my-container my-image
```

docker container create command have a shortcut docker create.

- Rename a container

```
$ docker container rename my-container my-awesome-container
```

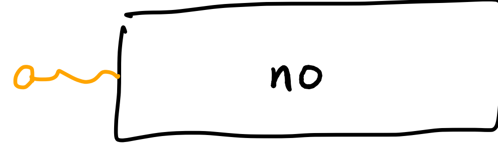
- List running containers

```
$ docker container ls
```

- List all the containers

```
$ docker container ls --all
```

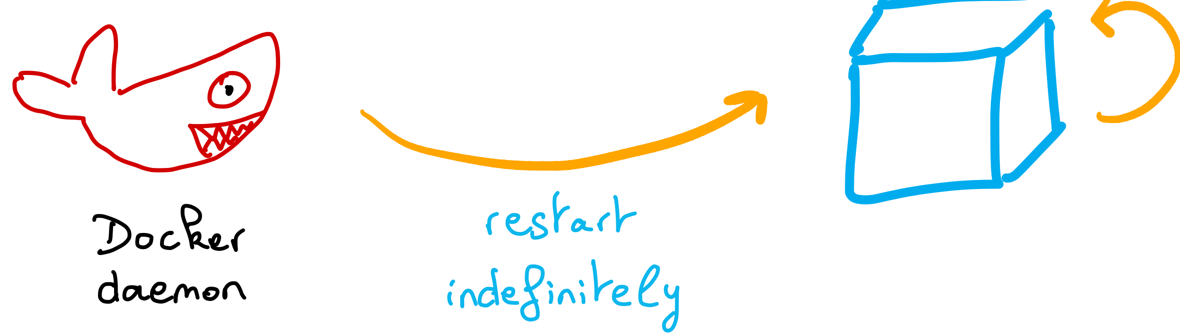
→ Several restart policies exist with --restart option that defines how a container should or not be restarted on exit:



→ Default restart policy

→ Never restart automatically a container

→ Always restart a container regardless of the exit status




```
$ docker run --restart=always my-container
```

on-failure

→ Restart a container only if it exits due to an error

```
$ docker run --restart=on-failure my-container
```

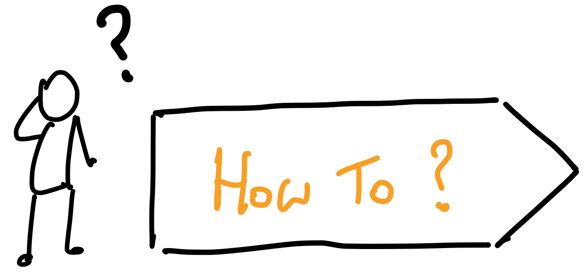
You can limit the number of restart retries the Docker daemon attempts:

```
$ docker run --restart=on-failure:5 my-container
```

unless-stopped

→ Restart a container unless it was stopped before the Docker daemon was restarted

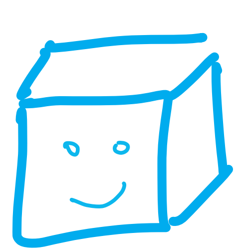
```
$ docker run --restart=unless-stopped my-container
```



> Change restart policy for a running container

```
$ docker update --restart=unless-stopped my-container
```

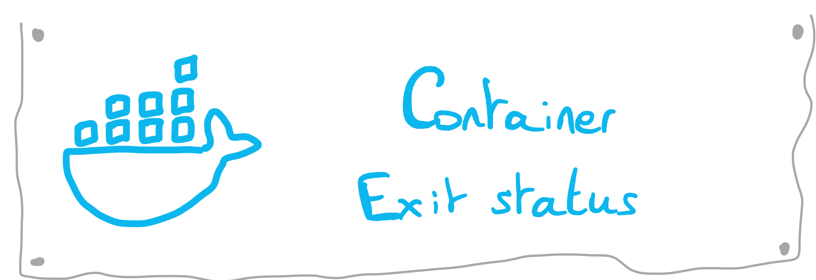
Restart policy only takes effect after a container starts successfully



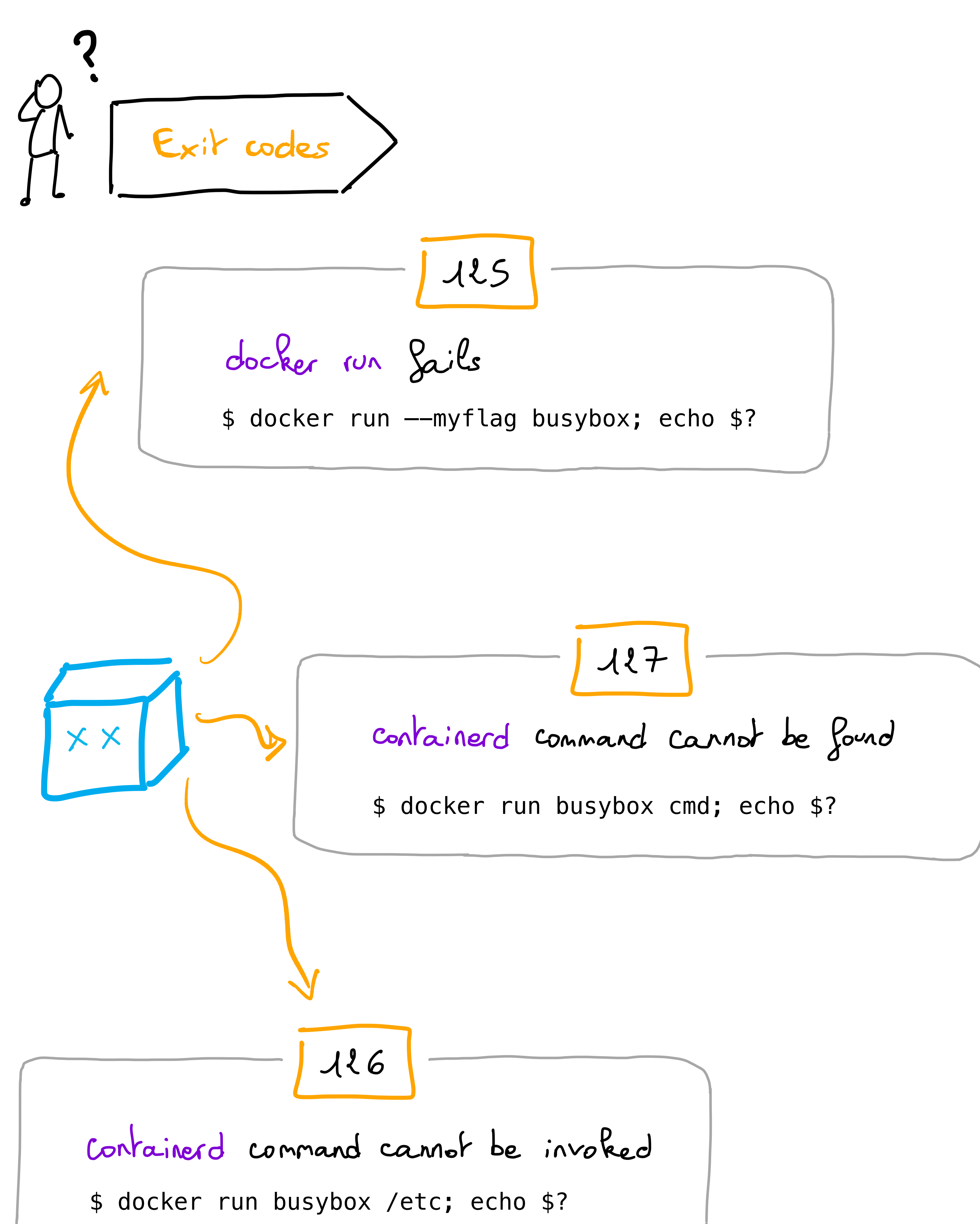
I started successfully if I'm up for at least 10 seconds

> Display the number of restarts for my-container container

```
$ docker inspect -f "{{ .RestartCount }}" my-container
```



- Exit status code gives information about why a container failed
- Exit code follows the chroot standard


Exit codes
125
docker run fails
$ docker run --myflag busybox; echo $?
127
containerd command cannot be found
$ docker run busybox cmd; echo $?
126
containerd command cannot be invoked
$ docker run busybox /etc; echo $?

128 + n → Fatal error signal

130 (128 + 2)

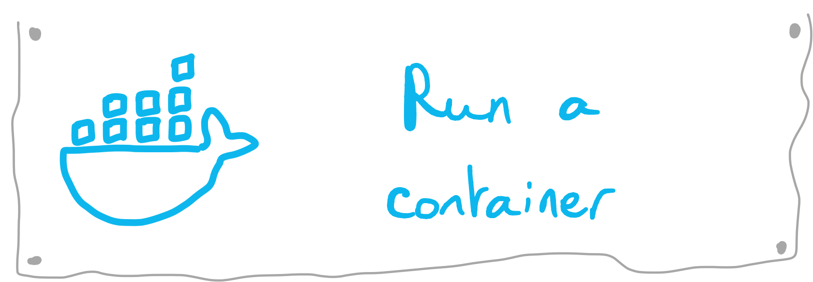
container terminated by **CONTROL + C**

137 (128 + 9)

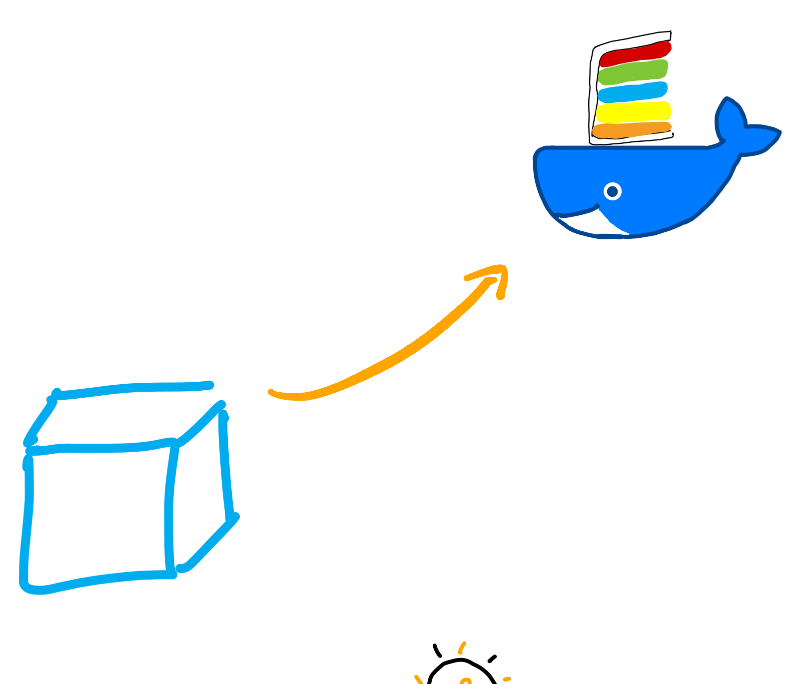
container received a **SIGKILL**

143 (128 + 15)

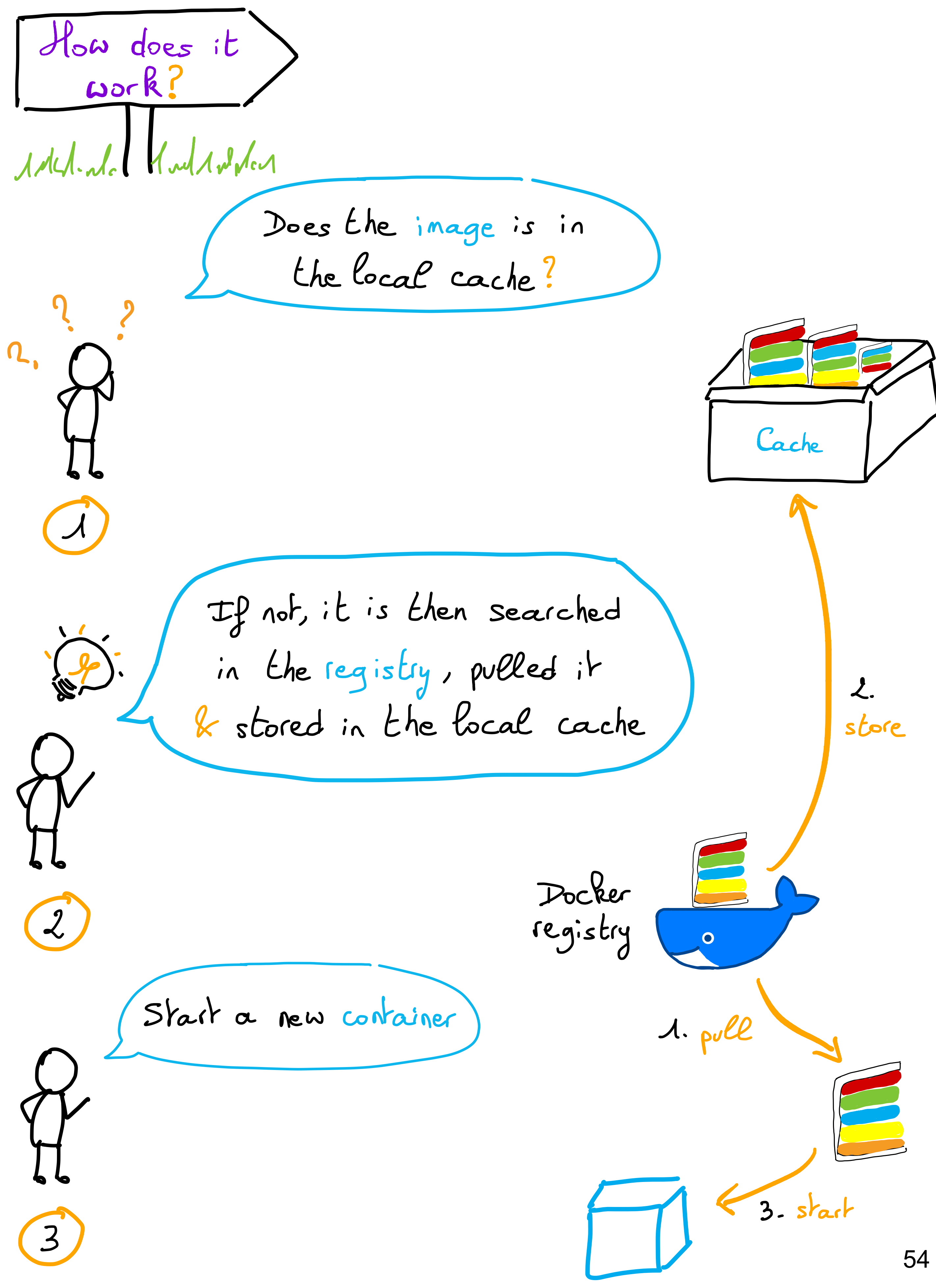
container received a **SIGTERM**

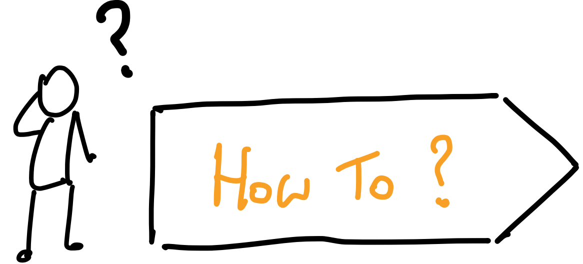


- A container is based on a static image
- docker run command allows you running a container from an image
- docker start command starts a container & all its previous changes



If you don't specify its name,
Docker generates a random two words name


How does it work?
Does the image is in the local cache?
Cache
1
If not, it is then searched in the registry, pulled it & stored in the local cache
2. store
2
Docker registry
1. pull
Start a new container
3
3. start



Create a container

```
$ docker create -it --name busybox busybox
```

Start a stopped container

```
$ docker start my-container
```

Run a container from debian image

```
$ docker run debian:stable
```

Run = { ① docker create ② docker start

› Run a container, name it & execute echo "hello sketchnotes lovers" command

```
$ docker run busybox --name busybox echo "hello sketchnotes lovers"
```

Container will execute echo command & then it will be terminated

› Run a container & open an interactive terminal into it

```
$ docker run -it busybox /bin/sh
```

-i / --interactive : allows you to interact / run commands in the container

-t / --tty : attach remote console to our locale console

⚠ With this option, container will not be stopped

Run a temporary container

```
$ docker run --rm -it my-img:my-tag /bin/bash
```

--rm : container will be removed when its exists

Run a container with detached mode & exposes ports

```
$ docker run -d -P --name my-container my-image
```

-d / -d = true : detached mode

-P : publish all exposed ports

Detached mode : local terminal is not attached to the running container

› Run a container & specify custom ports

```
$ docker run -p 8080:80 my-image:my-tag
```

-p : map local port to remote port

http://localhost:8080

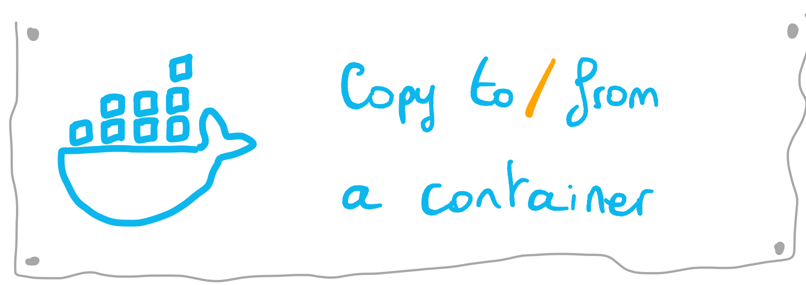
› Run a container & write the container ID in a file

```
$ docker run --cidfile /tmp/cid.txt ubuntu echo "hi!"
```

> Run a container & pass environment variables into it

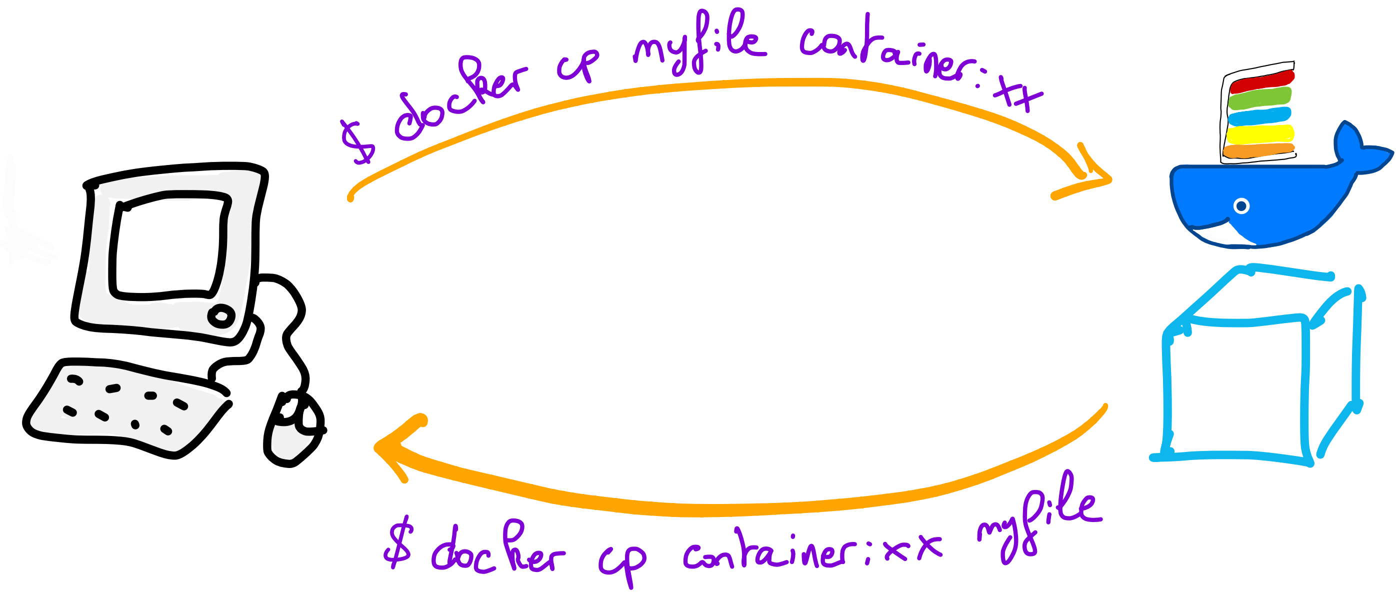
```
$ docker run -it --rm -p 5432:5432 \
-e POSTGRES_USER=my-user \
-e POSTGRES_PASSWORD=my-password \
postgres:9.6.10-alpine
```

-e / --env : set or override environment variables



→ Docker allows you copying files & folders

- from your machine to a container
- from a container to your local machine



⚠

It is not possible to copy specific system files such as resources under /proc, /sys, /dev, tmpfs ...

How To ?

- Copy "report.xml" file from a container to my local machine

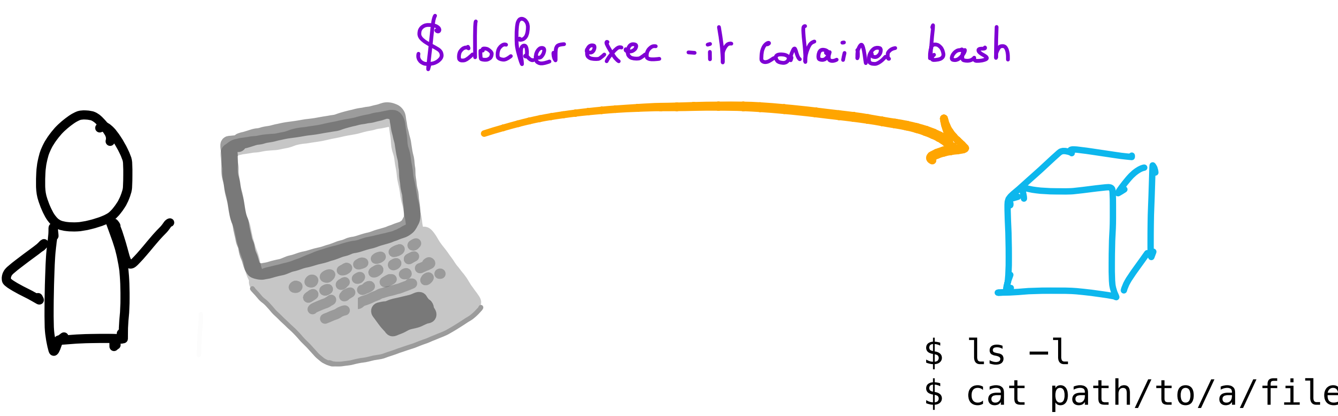
```
$ docker cp my-container:afolder/report.xml .
```

- Copy "init.conf" file inside a container

```
$ docker cp init.conf my-container:conf/env/init.conf
```

- docker exec command allows you running commands in a container

- Useful to debug in your container



How To ?

Connect to a container & open an interactive shell

```
$ docker exec -it my-container bash
```

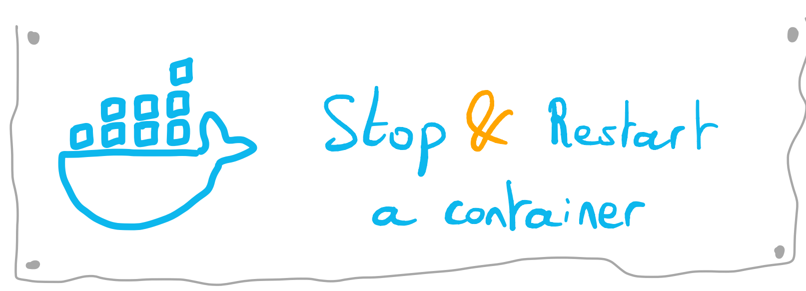
-it / --interactive --tty : interactive shell

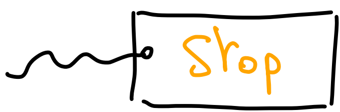
```
> ls -l
> tail -f /var/log/debug.log
```

Execute a command in a container in background

```
$ docker exec -d my-container touch path/my/file
```

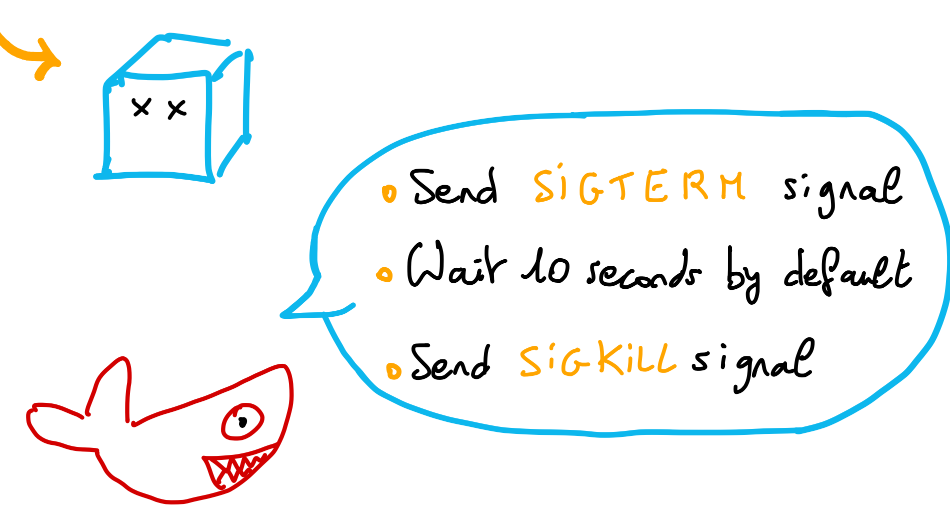
-d / --detach : detached mode

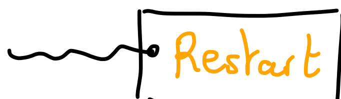




→ docker stop command stops any running containers

```
$ docker stop my-container
```





→ docker restart command restarts any running containers

You can stop and restart a container specified by its name or container ID

How To ?

› Stop container my-container

```
$ docker stop my-container
```

› Stop container my-container which have restart = yes configuration

```
$ docker update --restart=no my-container
$ docker stop my-container
```

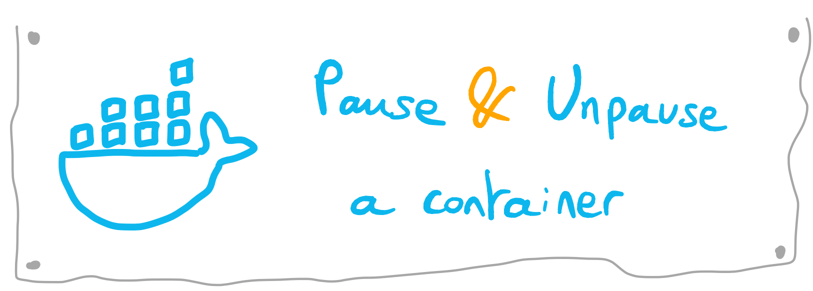
› Restart container my-container

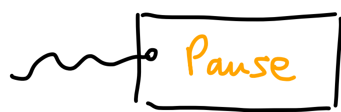
```
$ docker restart my-container
```

› Restart container my-container but wait 15 seconds before killing it

```
$ docker restart -t 15 my-container
```

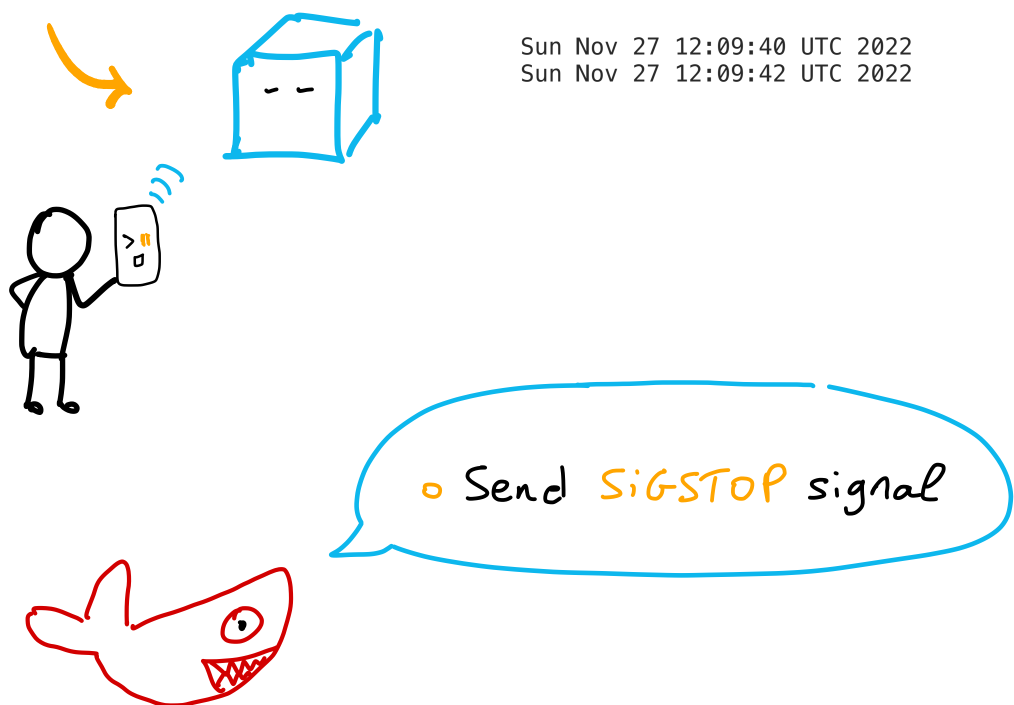
-t / --time

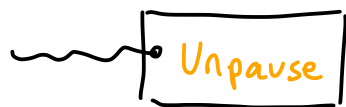




→ docker pause command pauses all of the processes inside one or several containers

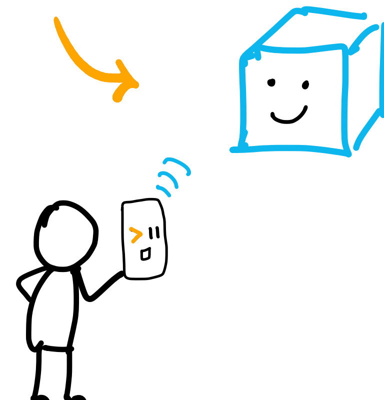
```
$ docker pause my-container
```





→ docker unpause command unpauses all the processes

```
$ docker unpause my-container
```



```
Sun Nov 27 12:09:40 UTC 2022
Sun Nov 27 12:09:42 UTC 2022
Sun Nov 27 14:01:12 UTC 2022
Sun Nov 27 14:01:14 UTC 2022
```

How To ?

› Run a container that display the date every 2 seconds

```
$ docker run --name my-container alpine /bin/sh
-c "while true; do sleep 2; date; done"
```

And run another container

```
$ docker run --name my-container2 alpine /bin/sh
-c "while true; do sleep 5; date; done"
```

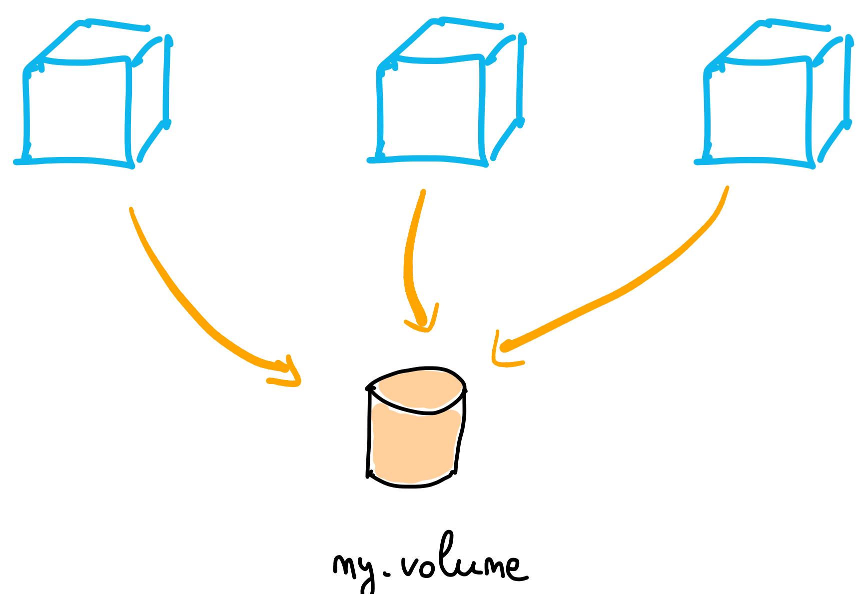
› Pause container my-container & my-container2

```
$ docker pause my-container my-container2
```

› Unpause container my-container

```
$ docker unpause my-container
```

- Volumes enable persisting data
- You can mount local folder inside a container using a volume
- They can be shared among several containers

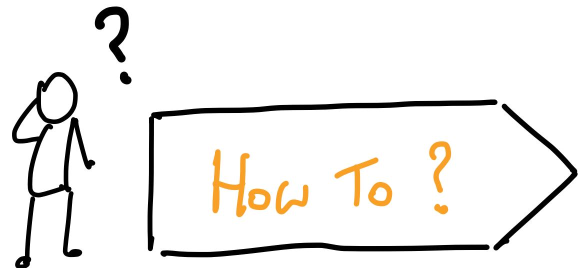


- Volume don't increase the size of a container

Volume lifecycle ≠ Container lifecycle

Volumes are saved in the host filesystem

`/var/lib/docker/volumes/`



Create a volume

```
$ docker volume create my-volume
```

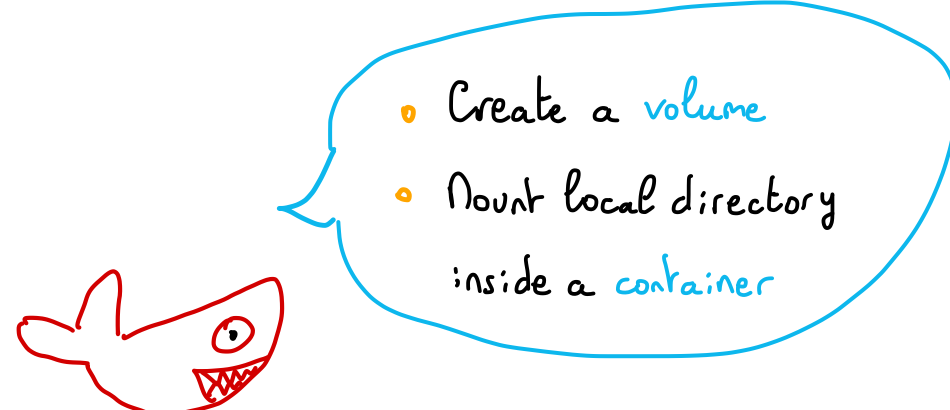
List all volumes

```
$ docker volume ls
```

Run a container & create a volume

```
$ docker run -it -v /etc:/my-etc my-container bin/bash
```

host dir / folder inside container



Run a container with a volume named my-vol

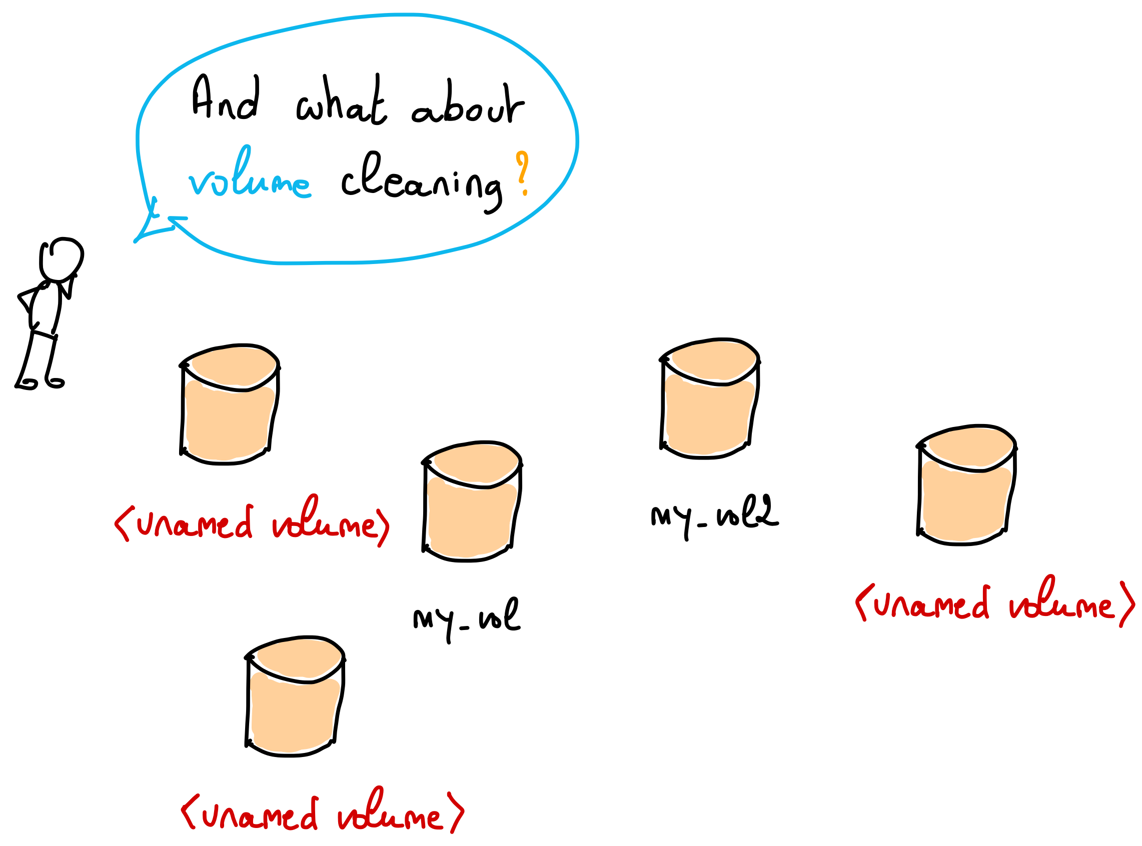
```
$ docker run -it -v /data --name my-vol my-container /bin/bash
```

Attach a running container's volume to another container

```
$ docker run -it --volumes-from my-vol my-container2 /bin/bash
```

Run a nginx container & mount a volume with read-only access

```
$ docker run -d -v nginx-vol:/usr/share/nginx/html:ro
--name nginx-vol nginx:latest
```

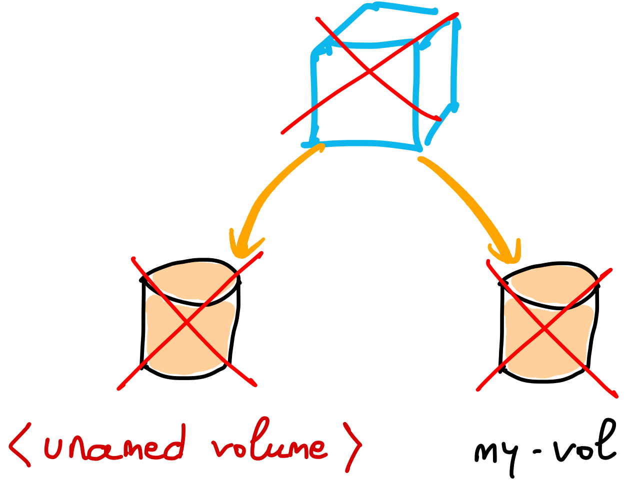


Delete a volume named my-vol

```
$ docker volume rm my-volume
```

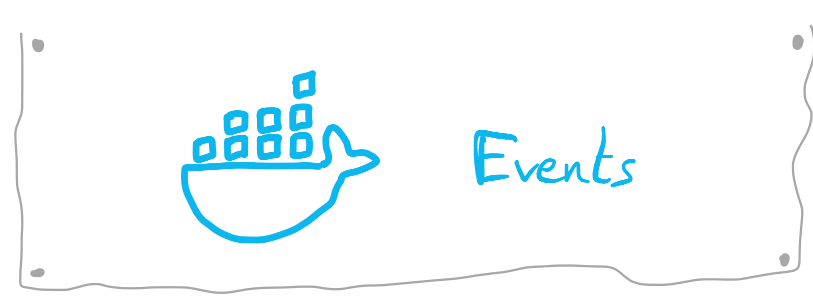
Run a container, create two volumes & remove them when top command has ended

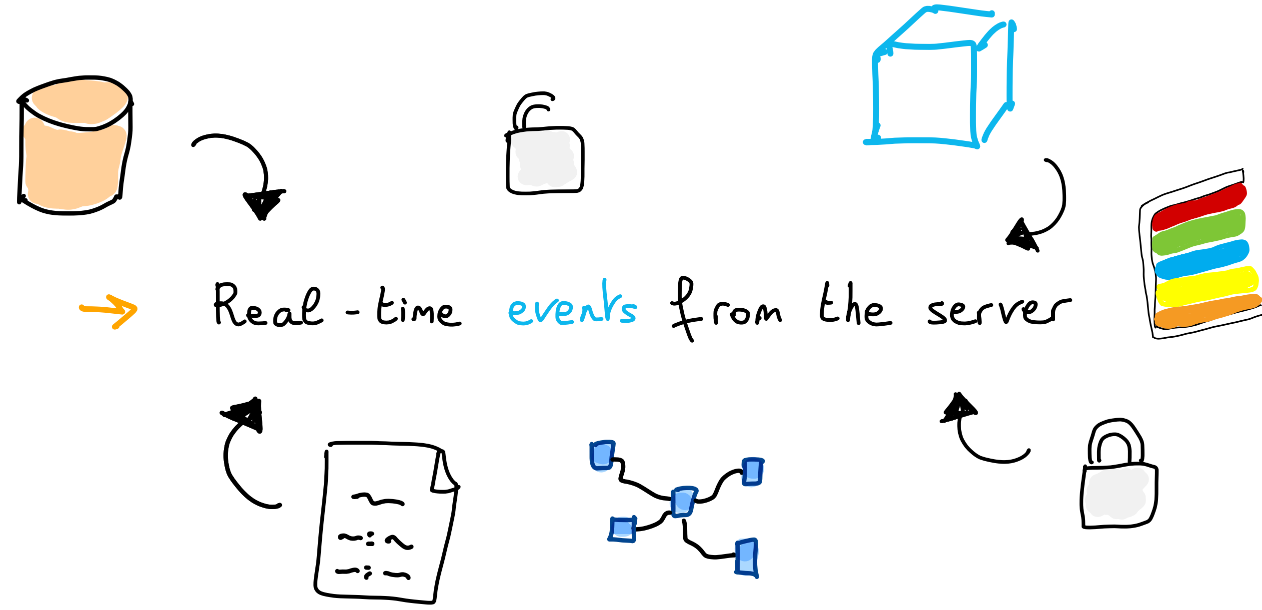
```
$ docker run --rm -v /my-folder -v my-vol:folder busybox top
```



Remove all volumes

```
$ docker volume prune
```





→ By default, last 1000 log events are displayed

→ It's possible to display events since :

- timestamp
- relative time
- date

How To ?

Display events since 5 minutes

```
$ docker events --since '5m'
```

Run container & display in real-time related container's events

```
$ docker run img & docker events
--filter 'container=$(docker ps -lq)'
```

-l : last
-q : quiet

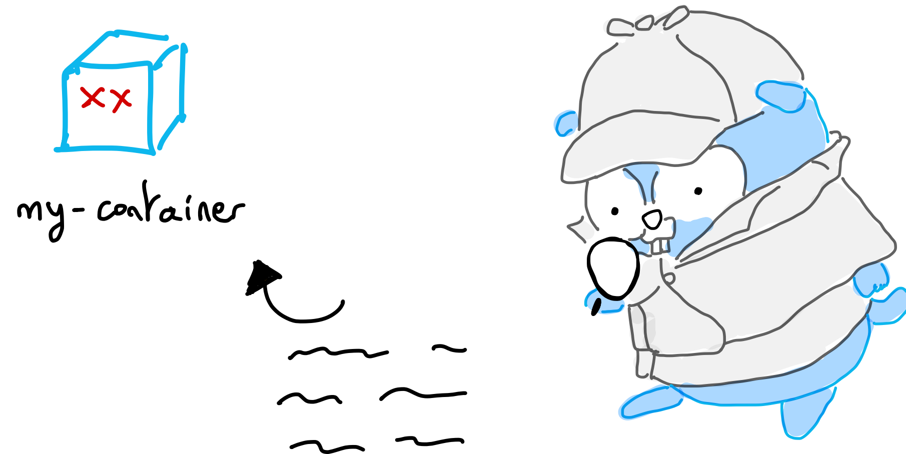
Display events in JSON

```
$ docker events --format '{{ json . }}'
```

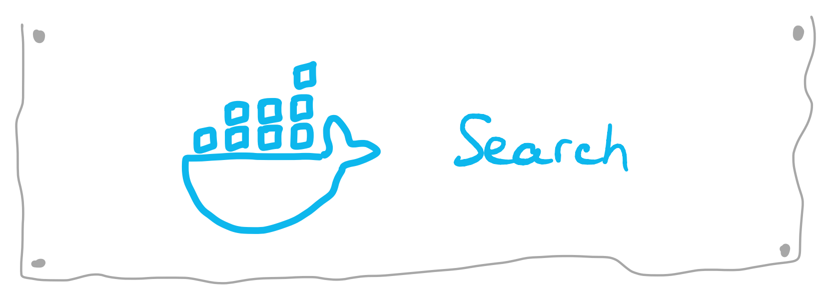
Display Alpine image's events

```
$ docker events --filter 'image=alpine'
```

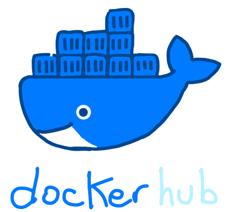
> Display stop events on a test container




```
$ docker events --filter 'container='my-container'
                --filter 'event=stop'
```



→ When you want to find an image in a registry, first habit is to search in UI ...



...

→ But, do you know you can search directly through Docker CLI? :)

→ By default, docker search command search in Docker Hub

How To ?

List Ubuntu image

```
$ docker search ubuntu
```

Search in a private registry my-image image

```
$ docker search <registry_host>:<registry_port>/my-image
```

List Debian image that have more than 10 stars

```
$ docker search --filter stars=10 debian
```

List official Alpine image with up to 5 results

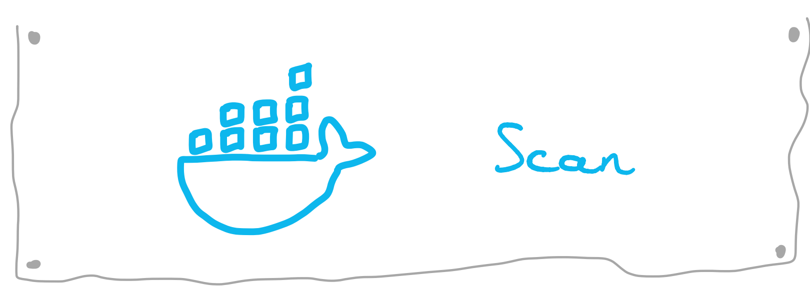
```
$ docker search -f is-official=true --limit=5 alpine
```

List busybox images without any truncated description

```
$ docker search --no-trunc busybox
```

List nginx images & format the response

```
$ docker search
--format "{{.Name}}\t{{.StarCount}}\t{{.IsOfficial}}" nginx
```



→ Scan vulnerabilities in your local images & DockerFile

⚠

Beta feature

Need to install Docker in Edge version.

Don't work for Alpine distributions yet.

How To ?

Scan a local image

```
$ docker scan my-image:my-tag
```

Scan image with a detailed analysis

```
$ docker scan my-image:my-tag --file Dockerfile
```

Display the dependencies

```
$ docker scan my-image:my-tag --dependency-tree
```

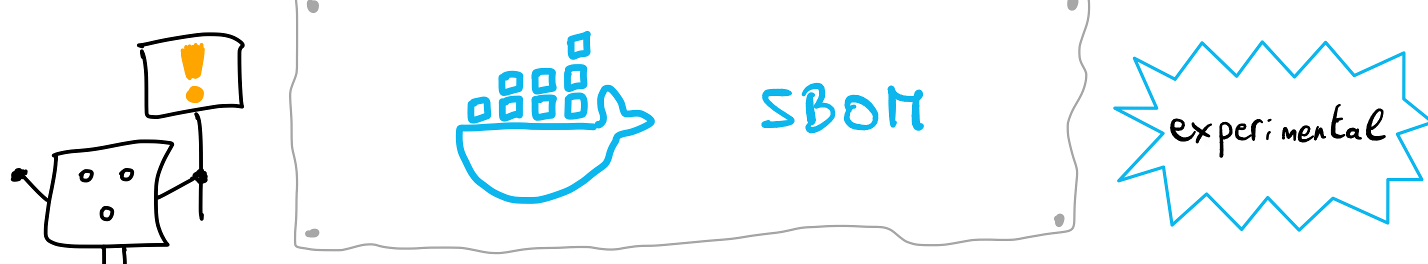
Display only vulnerabilities in your code (excluding base image)

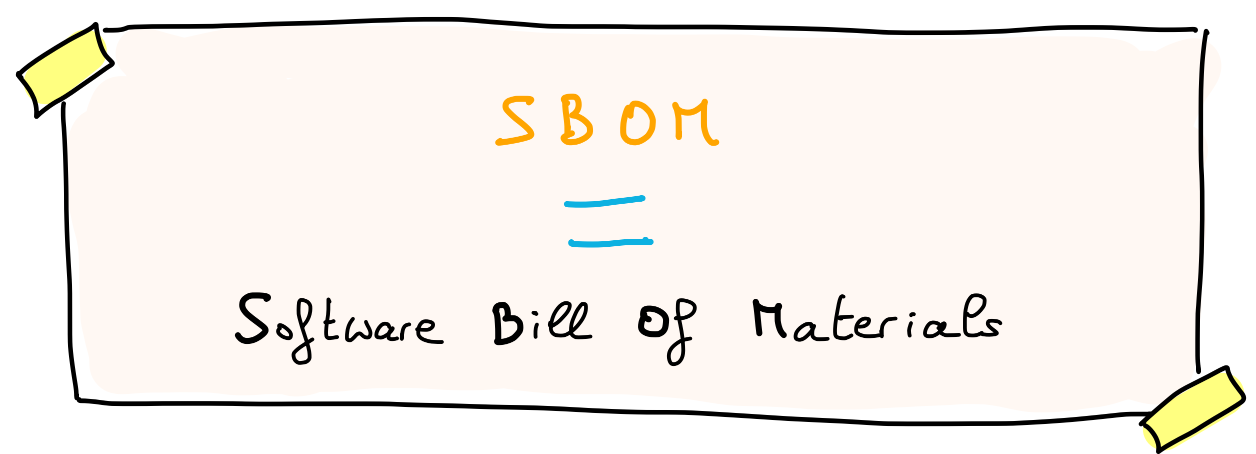
```
$ docker scan my-image:my-tag --file Dockerfile
--exclude-base
```

Display only vulnerabilities with high severity

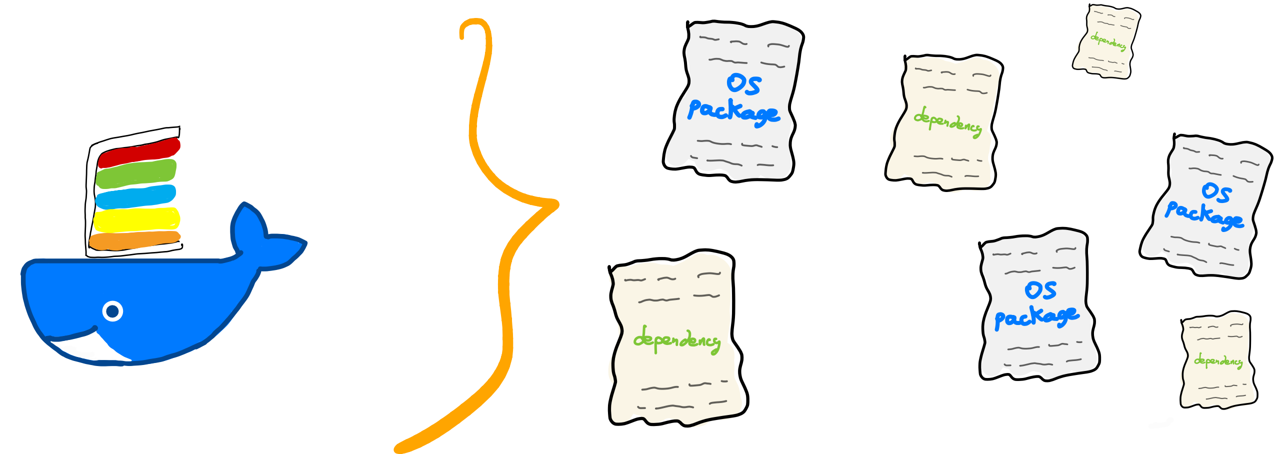
```
$ docker scan my-image:my-tag --file Dockerfile
--json |
jq '[.vulnerabilities[] | select(.severity=="high")]'
```

10 tests per month are allowed. To increase this quota, you need to sign-up for a free Snyk account.





→ An inventory of all the components & all the software dependencies in a container image



Why ?

→ To find packages that can contain vulnerabilities

I want to search if my image contains Log4J ?

Yes, thanks to docker sbom you can ◡̈

docker sbom is using Syft service, but it may change in the future

How To ?

- Generate a SBOM for an image

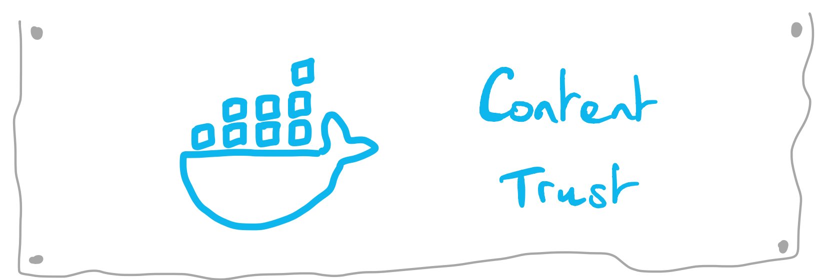
```
$ docker sbom my-image:tag
```

- Generate a SBOM in an output file

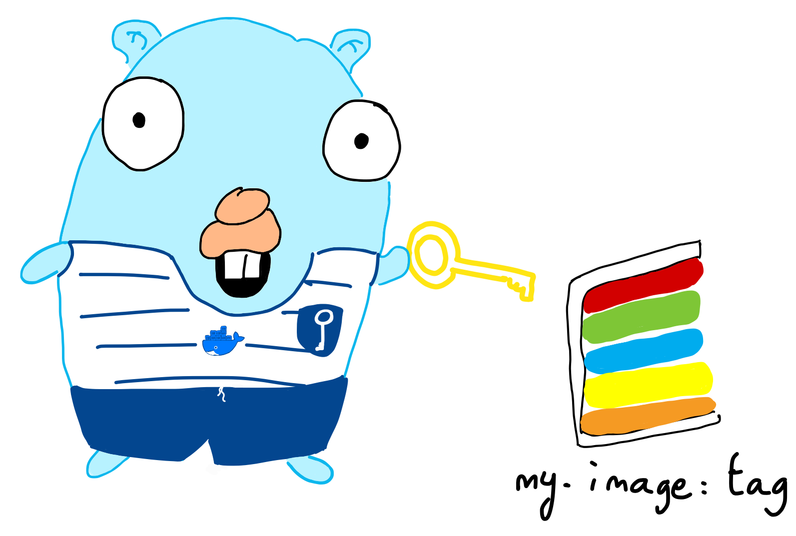
```
$ docker sbom ubuntu --output sbom.txt
```

- Generate a SBOM in JSON format

```
$ docker sbom ubuntu --format syft-json
```



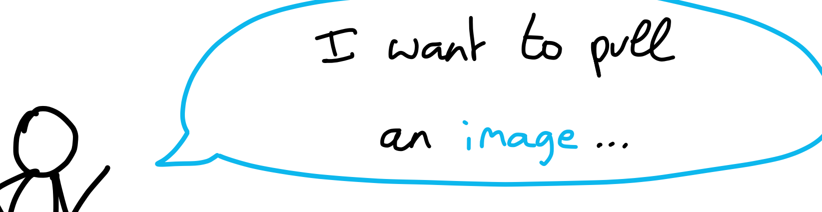
- Most of Docker images are unsafe, even the ones you can find in Docker Hub
- Increase the trust in images thanks to docker trust commands.
- A tagged image can have a signature. It attests you created & published the tag.



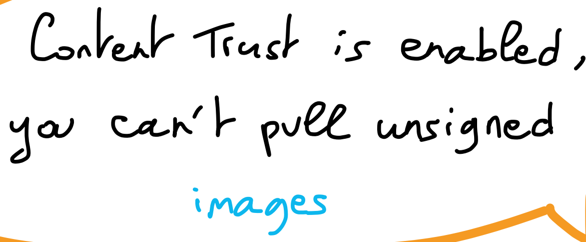
Official images are signed

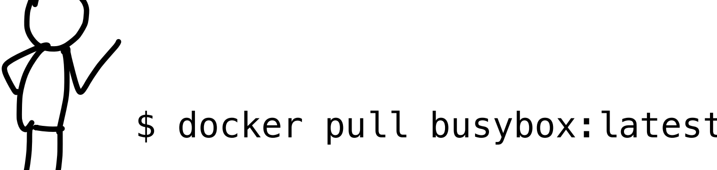
→ You can enable Docker Content Trust & protect your host for unsigned images

```
$ export DOCKER_CONTENT_TRUST=1
```

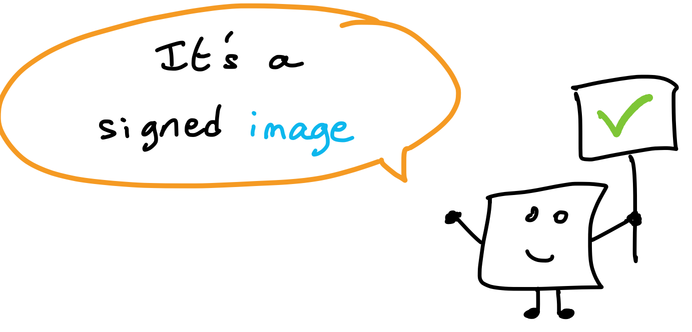



```
$ docker pull scraly/what-is-my-pod:1.0.1
```





```
$ docker pull busybox:latest
```



Once Content Trust is enabled,
docker push command will push
& sign the image!

→ Every time you sign a tagged image
or revoke a signature,
you must enter a passphrase

How To ?

Enable Docker Content Trust

```
$ export DOCKER_CONTENT_TRUST=1
```

Disable Docker Content Trust

```
$ unset DOCKER_CONTENT_TRUST
```

Create a new tag

```
$ docker tag scraly/what-is-my-pod:1.0.1
scraly/what-is-my-pod-signed:1.0.1
```

Push & sign the image

```
$ docker push scraly/what-is-my-pod-signed:1.0.1
```

Display information about key & signature in JSON format

```
$ docker trust inspect scraly/what-is-my-pod-signed:1.0.1
```

Display information about key & signature in a more readable way

```
$ docker trust inspect scraly/what-is-my-pod-signed:1.0.1
--pretty
```

Sign the 1.0.2 tag

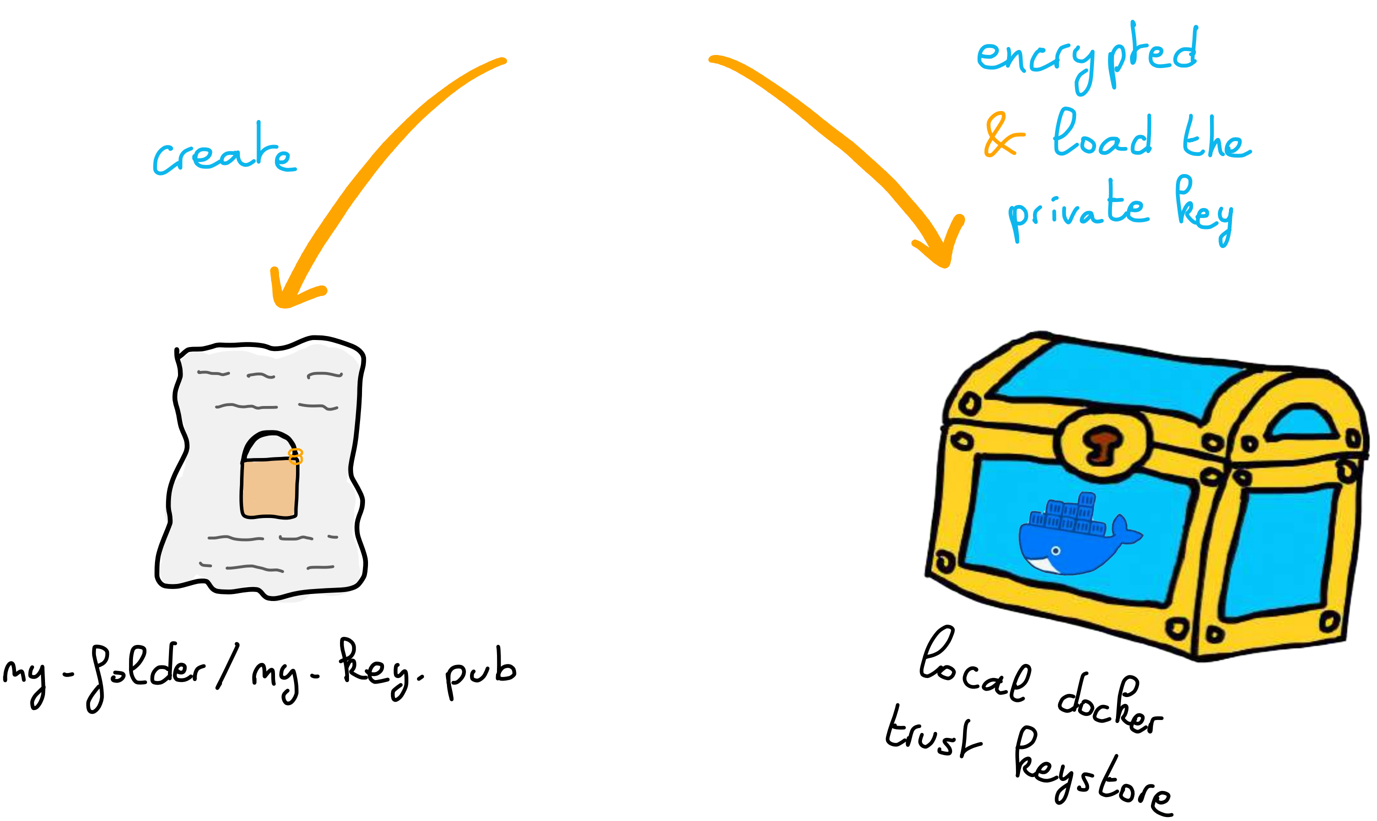
```
$ docker trust sign scraly/what-is-my-pod-signed:1.0.2
```

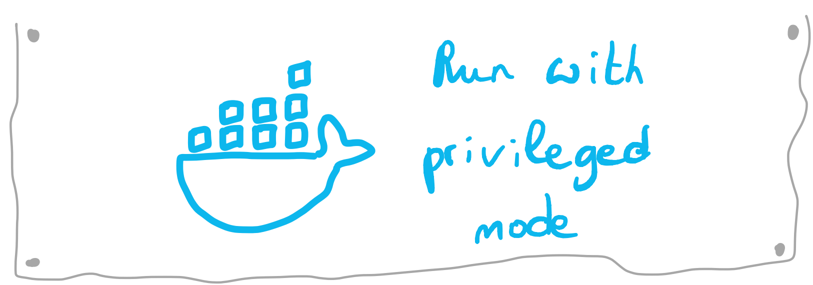
Delete signature for 1.0.2 tag

```
$ docker trust revoke scraly/what-is-my-pod-signed:1.0.2
```

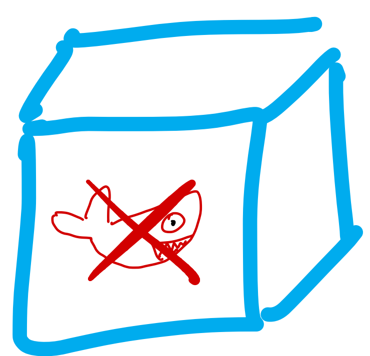
Generate a signing key pair

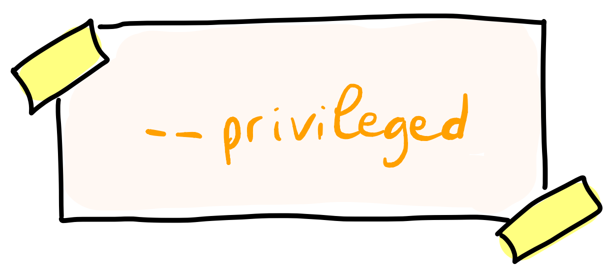
```
$ docker trust key generate my-key --dir my-folder
```



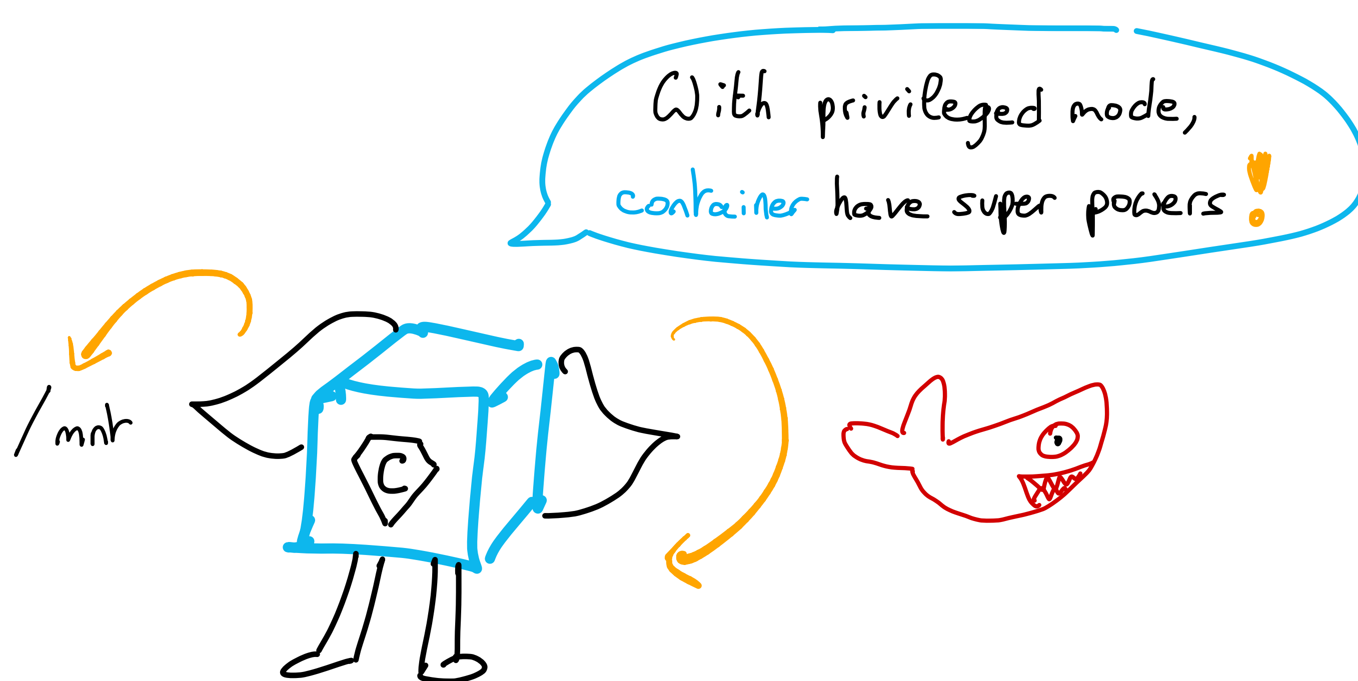


By default, containers are unprivileged





- Grants to a container root capabilities to all devices on the host
- Thanks to this mode, you can run Docker daemon in a container



It is not recommended to run privileged containers in production environment.
Be careful of this usage.

How To ?

- Run a container with privileged mode so it can access to all devices in the host

```
$ docker run --privileged my_image:tag
```

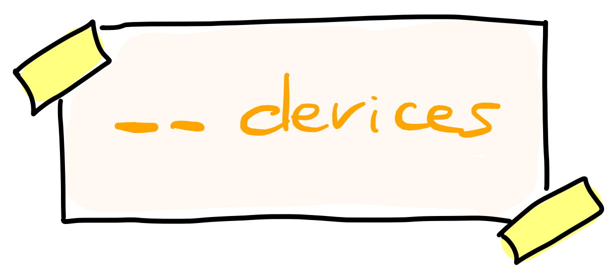
- Run a container, then create & mount a temporary file system

```
$ docker run -it --privileged ubuntu mount -t tmpfs none /mnt
```

- Check if a container is in privileged mode

```
$ docker inspect
--format '{{.HostConfig.Privileged}}' my-container
```

true / false



→ Allows to reduce risks with privileged mode

By default, containers can read, write & mknod on devices but you can customize these rights

How To ?

- Run a container which can only read partition table

```
$ docker run --device /dev/sda:/dev/xvdc:r --rm -it ubuntu
```

r = read

- Run a container which can only read & write partition table

```
$ docker run --device /dev/sda:/dev/xvdc:rw --rm -it ubuntu
```

rw = read & write

- Run a container which can only allow mknod permission

```
$ docker run --device /dev/sda:/dev/xvdc:m --rm -it ubuntu
```

m = mknod

--cap-add
/
--cap-drop

→ Allows adding & drop Linux capabilities

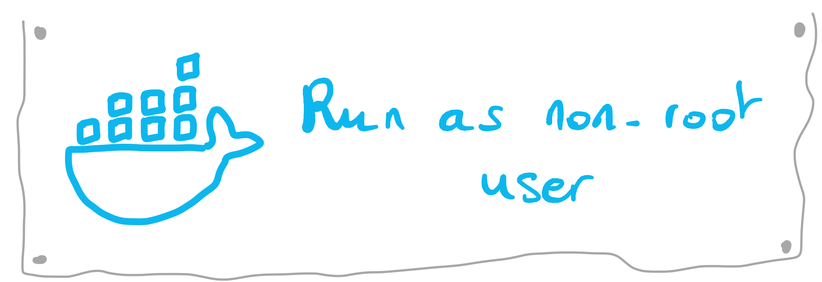
How To ?

- Run a container which use all capabilities except for chown

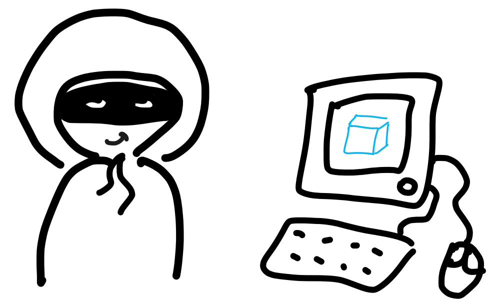
```
$ docker run --cap-add ALL --cap-drop CHOWN ubuntu
```

- Run a container which mount a FUSE based filesystem

```
$ docker run --rm -it --cap-add SYS_ADMIN
--device /dev/fuse sshf
```



→ By default, Docker runs containers as root user, with root privileges



- Running as root can cause security issues
- Anyone can start undesirable processes in the container
- Malicious user can change the UID & GID when starting the container

Best practices

- [x] Give the minimum amount of privileges necessary to run a process
- [x] Run containers as non-root user

How To ?

› Run a container & check the container is running as root

```
$ docker run --rm busybox id
```

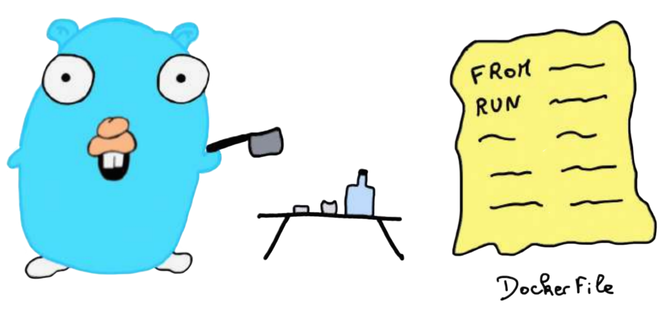
Unable to find image 'busybox:latest' locally
...
Status: Downloaded newer image for busybox:latest
uid=0(root) gid=0(root) groups=10(wheel)

› Run a container & force the container to start with a given UID & GID

```
$ docker run --rm --user 1000:1000 busybox id
```

uid=1000 gid=1000

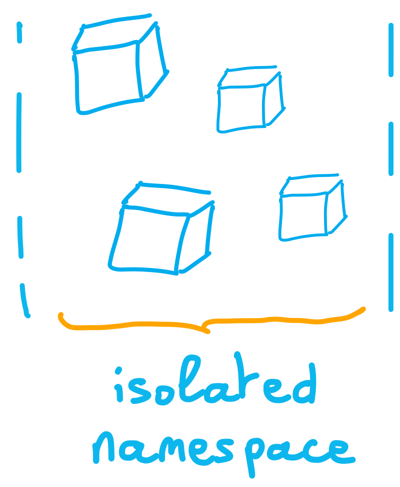
Create a user "myuser" with UID 666, give the rights to /app folder & use it

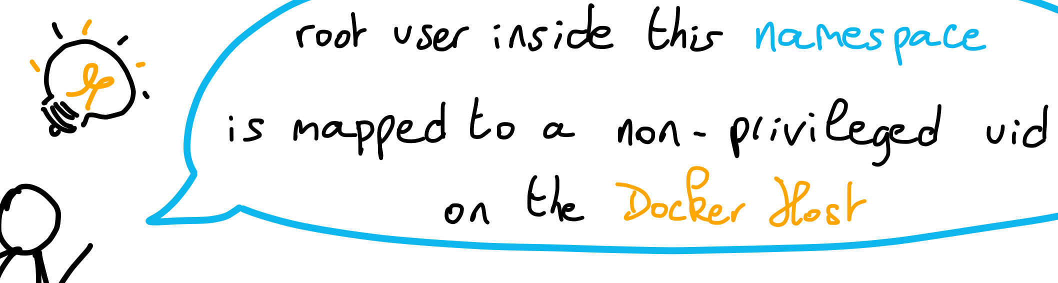



```
FROM ubuntu:latest
RUN useradd -u 666 myuser
WORKDIR /app
COPY . /app
RUN chown -R myuser /app
USER myuser
CMD ls -alrt /app
```

User Namespaces

- A concept of Linux kernel
- Isolated namespace simulating a root namespace





→ Containers can have root privileges in the user namespace without having any privileges in the Docker Host

How To ?

1. Enable userns-remap on the Daemon & start the default user namespace with the dockermap user & group mapped to non-privileged UID and GID

~/.docker/daemon.json

```
{
  "userns-remap" : "default"
}
```

Or

```
$ dockerd --userns-remap=default &
```

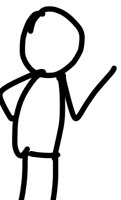
2. Verif Docker have created the dockermap user for you

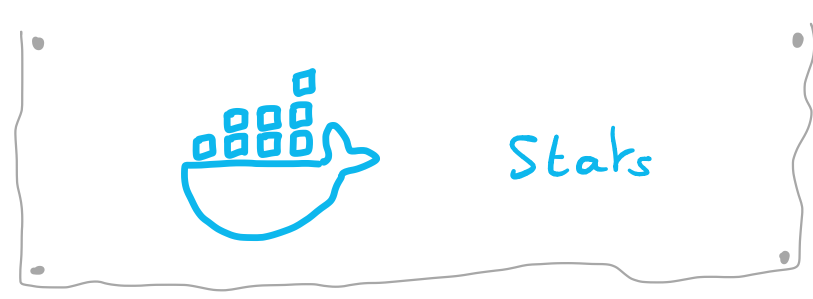
```
$ id dockermap
```

③ Now you can run a container, isolated for the host

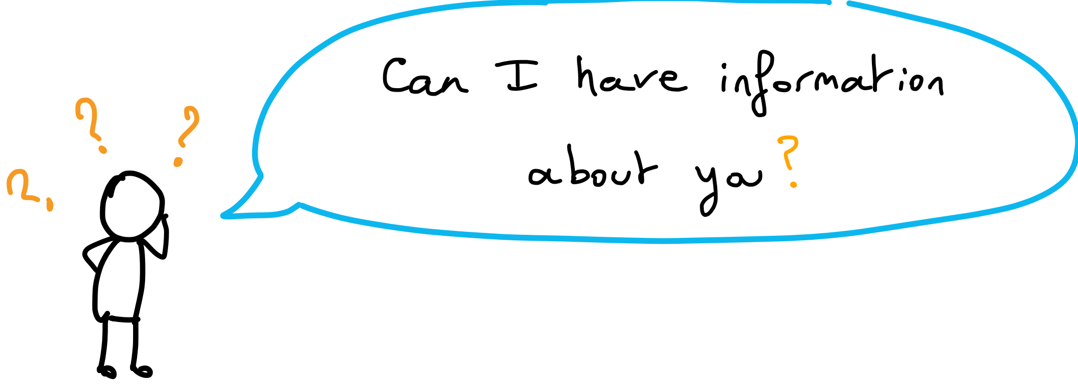
```
$ docker run -it -v /bin:/host/bin busybox /bin/sh
```

In a user namespace you can't have an impact to the host

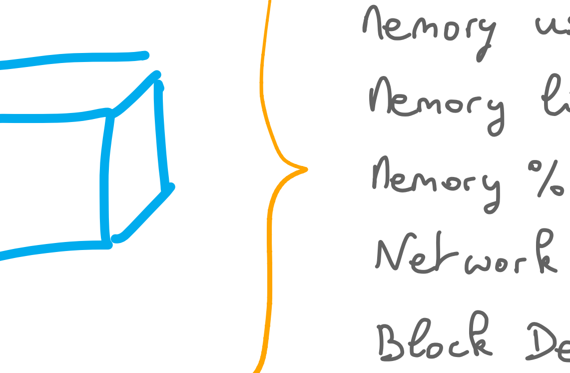




→ Display running containers streamed usage information



Sure, I'm running ◡̈



CPU %
Memory usage
Memory limit
Memory %
Network I/O
Block Devices I/O

⚠ It's possible to query stats about stopped containers but information will be empty.

How To ?

› Display information for all running containers

```
$ docker stats
```

› Display information for my-container & container 2

```
$ docker stats my-container <container_2_ID>
```

› Display non streamed stats for my-container

```
$ docker stats my-container --no-stream
```

› Display stats in a customized format response for all containers

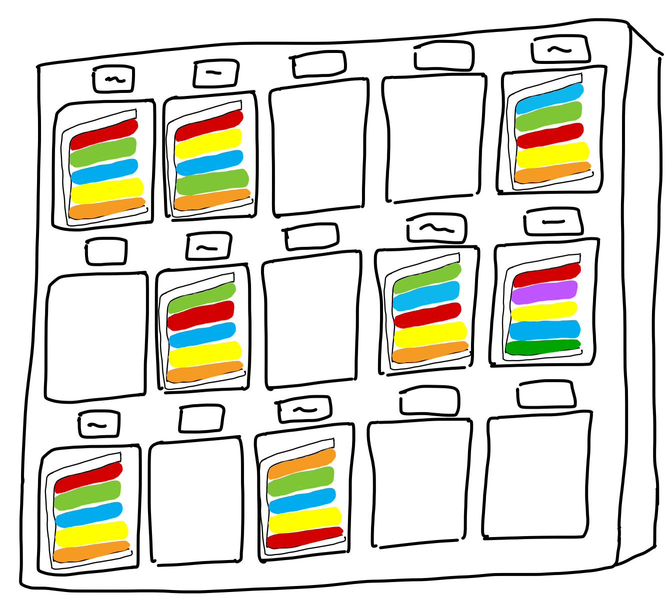
```
$ docker stats -a
--format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"
```

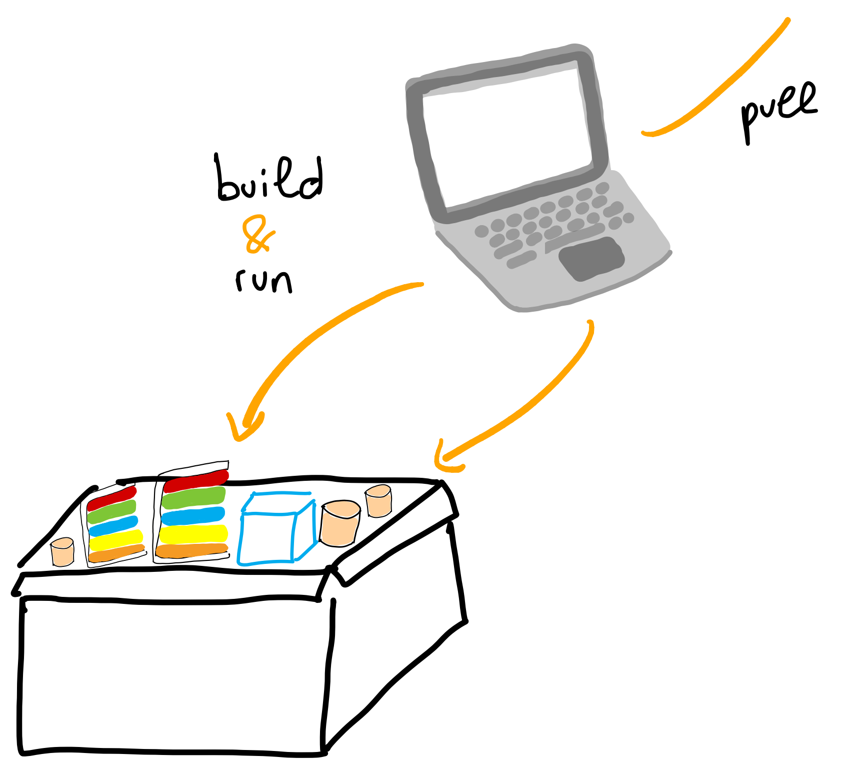
› Display stats for busybox in JSON format

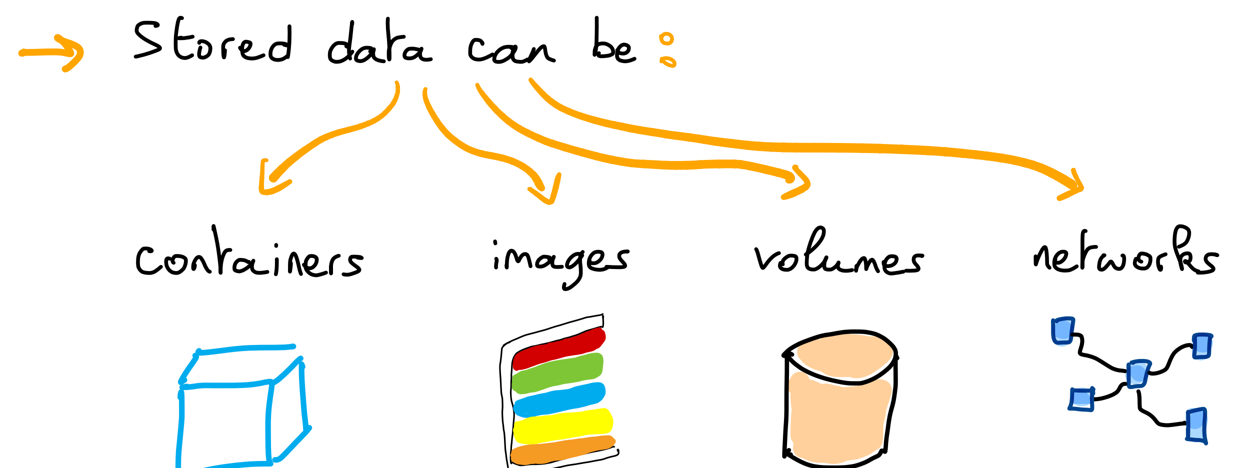
```
$ docker stats busybox --no-stream --format "{{ json . }}"
```

Clean & Purge

→ Each time you {pull / build} images & run containers, data are stored in your machine.





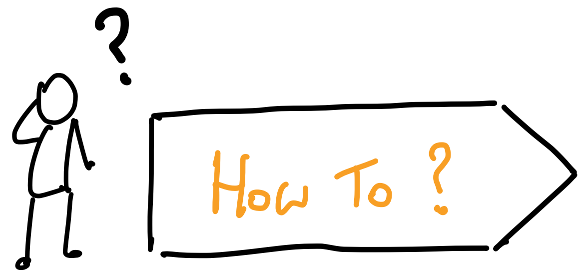


→ All theses data can be removed through CLI ☺

EX-TER-MI-NATE

⚠

When a container is removed, a volume is not automatically removed.



Remove a container

```
$ docker container rm <containerID>
```

Remove all exited containers

```
$ docker rm $(docker ps -a -q -f status=exited)
```

Remove all stopped containers

```
$ docker container prune
```

Remove an image (or several)

```
$ docker rmi <imageID1> <imageID2>
```

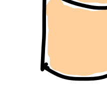
Remove dangling images

```
$ docker images purge
```

Remove all images

```
$ docker rmi $(docker images -a -q)
```

-a all images
-q option return unique IDs



Remove a volume

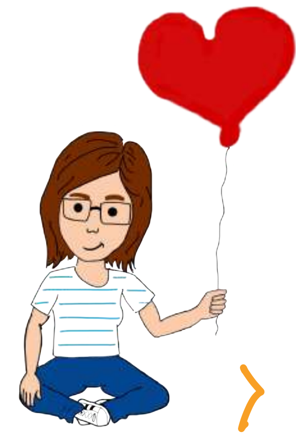
```
$ docker volume rm <volumeID>
```

Remove a container & its volume (even if it's an unamed volume)

```
$ docker rm -v <containerID>
```

my-container

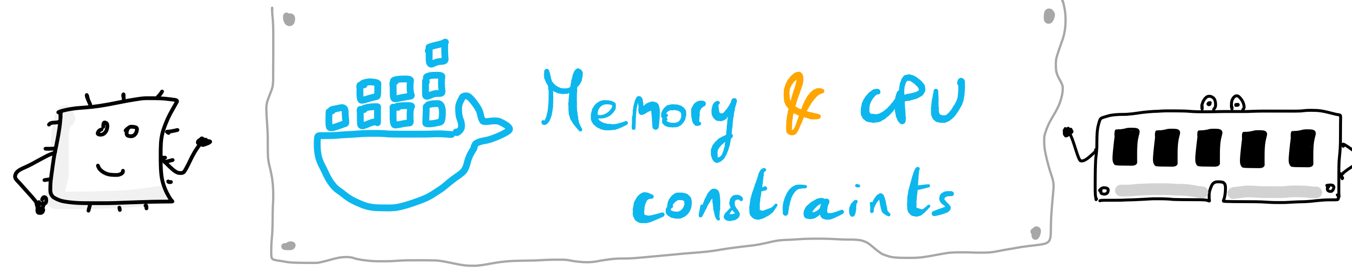
?



And ... My favorite purge command

Remove images, containers, volumes, networks active + stopped

```
$ docker system prune -a --volumes
```

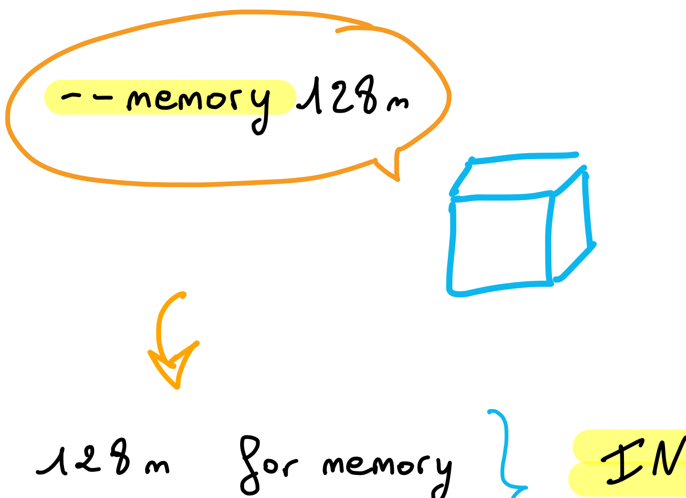


Memory limitations

→ By default, Docker containers don't have memory limitations

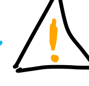
→ Containers can use all available memory they want on the host

→ When you define memory, container can swap the same amount



128 m for memory
128 m for SWAP

IN REALITY 256 m !



Be careful, using SWAP can slow down performances. Indeed, system will write to the disk.

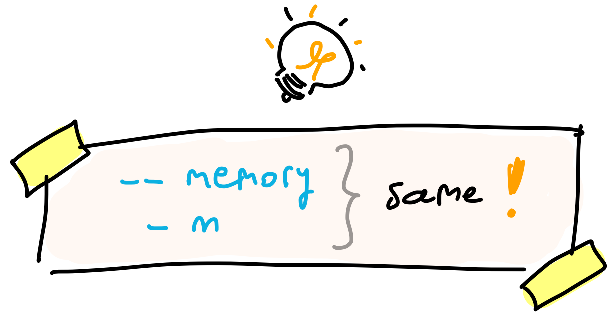
How To ?

> Run a container & set the max memory usage to 60 mb

```
$ docker run --memory 60m my-image:tag
```

> Run a container & use unlimited amount of SWAP

```
$ docker run -m 128m --memory-swap -1 my-image:tag
```

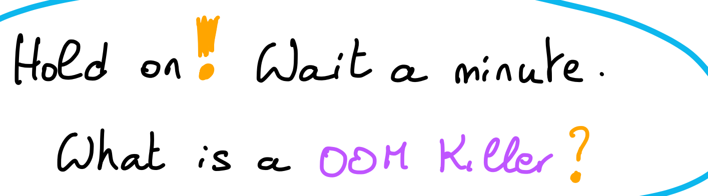


> Run a container & deactivate OOM Killer

```
$ docker run -m 128m --oom-kill-disable my-image:tag
```

-- memory limits the memory usage inside the containers ...

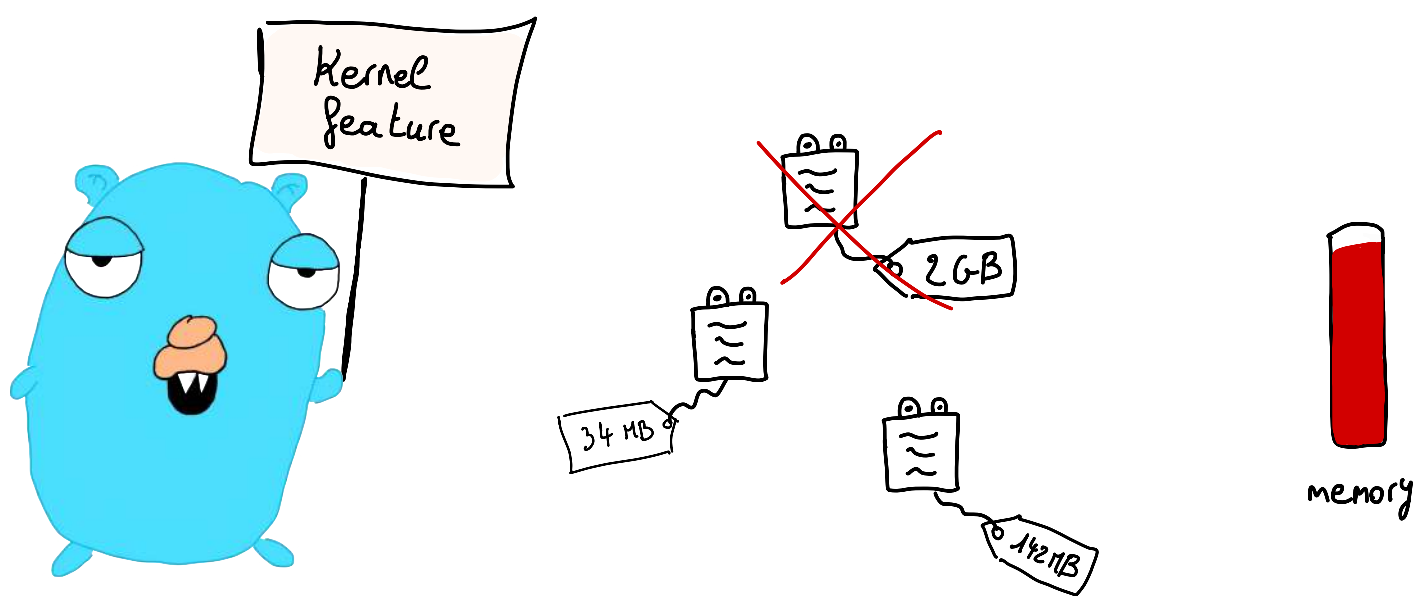
But it's not equals to the system available memory that you can see with free command inside a container.



OOM Killer = Out-Of-Memory Killer

→ Wakes up when there is no enough memory on the system

→ Kills the greediest processes until the system recovers enough resources to keep them running



- Set a max memory
  & a SWAP limit to really limit to 60 mb

```
$ docker run --memory 60m --memory-swap 60m my-image:tag
```

numbers are equals → container will not SWAP

CPU limitations

→ By default, containers don't have CPU limitation

It's not possible to define a limit of the available number of CPUs on the host

How To ?

- Run a container & limit its CPU utilization to 1 CPU

```
$ docker run --cpus 1 my-image:tag
```

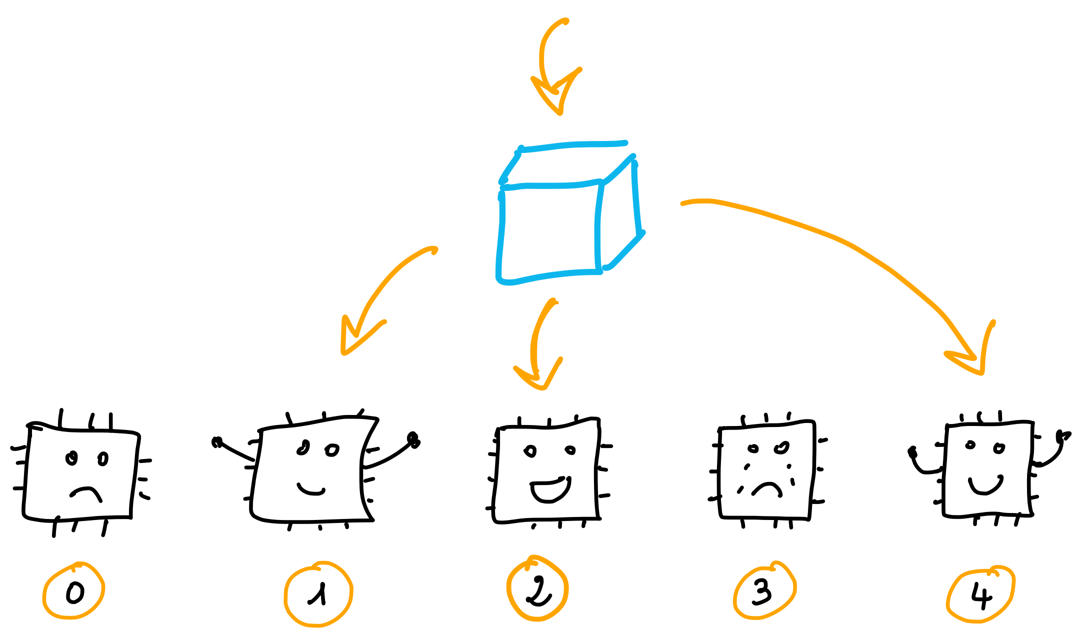
- Run a container & set it can use only 50% of available CPU

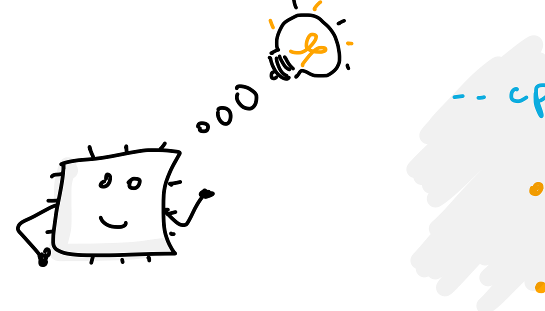
```
$ docker run --cpus .5 my-image:tag
```

Dedicated CPUs

→ Allows you to separate containers onto different CPUs or cores

`--cpuset.cpus 1,2,4`



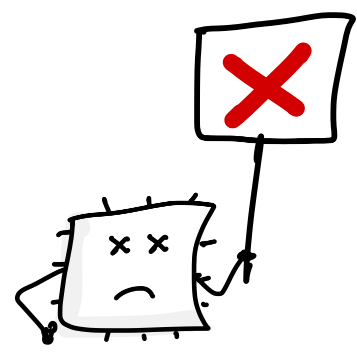


--cpuset-cpus means:

- a list 1,2,4
- a range 0-3

Be careful, --cpuset--cpus use basic index starting from 0!

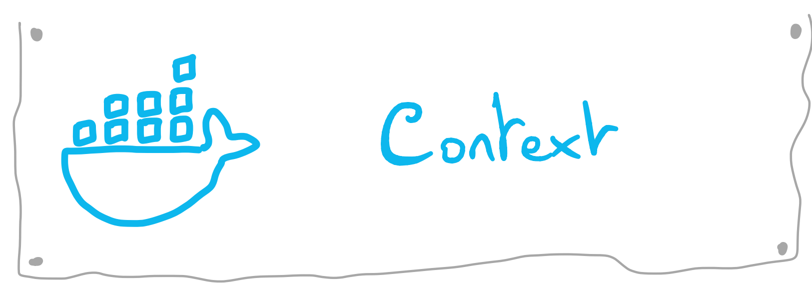
→ Defining an invalid CPU index will throw an error



How To ?

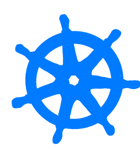
- Run a container & allocate the 2nd and 3rd CPU

```
$ docker run --cpuset-cpus 1,2 my-image:tag
```

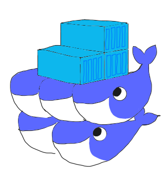


→ Allows to handle the connection :

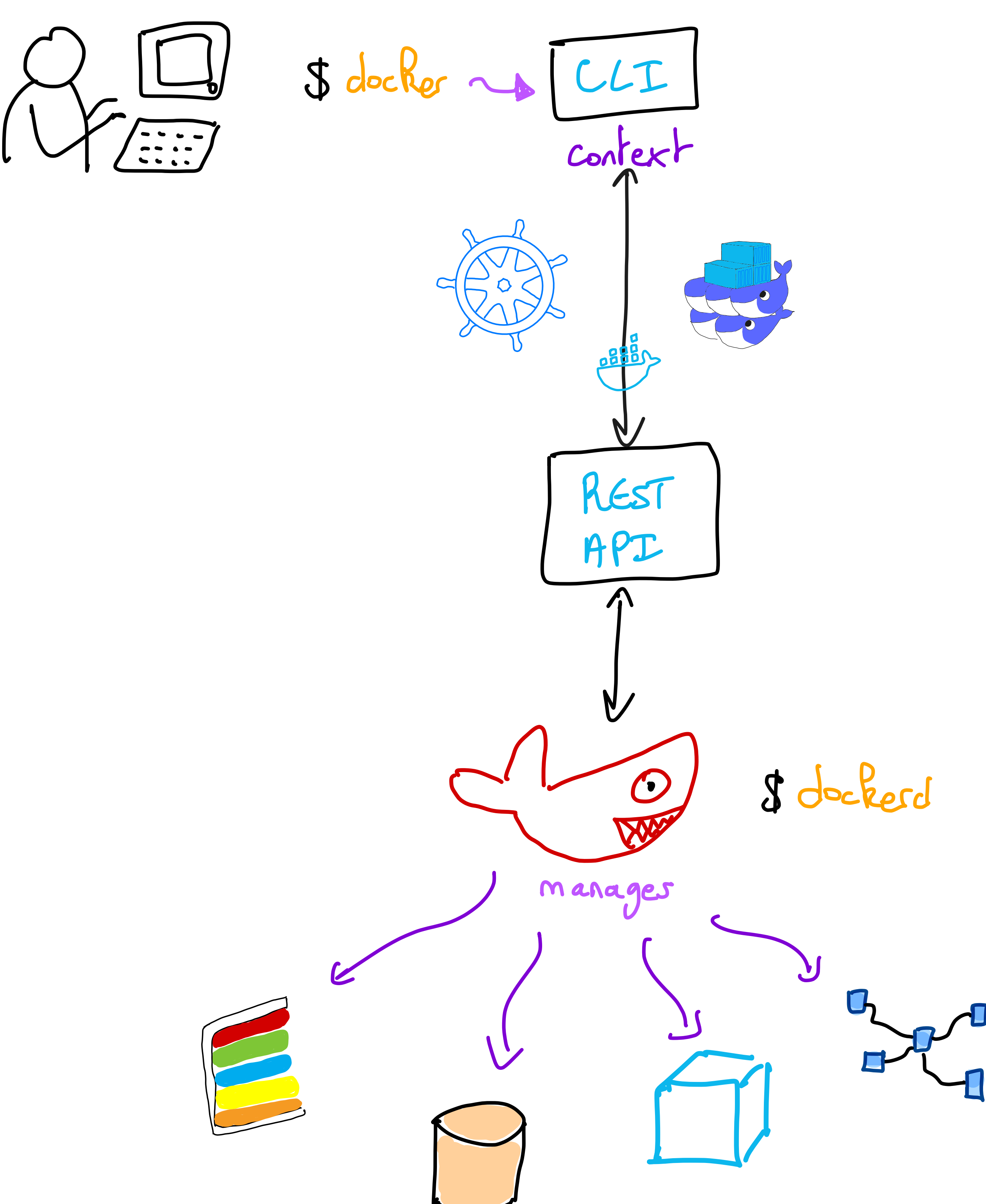
- remote Docker instances / nodes



- Kubernetes clusters



- Swarm clusters


$ docker
CLI
context
REST
API
$ dockerd
manages

By default, a default context is created & equals to your Docker local installation

How To ?

List existing contexts

```
$ docker context ls
```

Create a new context to our another Docker host

```
$ docker context create dev
--description "Development environment
--docker "host=ssh://$IP"
```

Switch to our dev context

```
$ docker context use dev
```

Now, you can list existing containers running in the second host

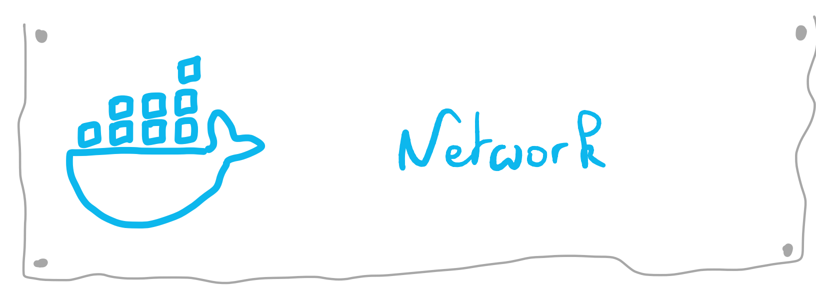
```
$ docker container ls
```

Run a new container in our default context

```
$ docker --context default run -d -p 80:80 nginx
```

Save your context

```
$ docker context export dev
```



→ Containers in a same network can communicate to each others

→ A Form of isolation

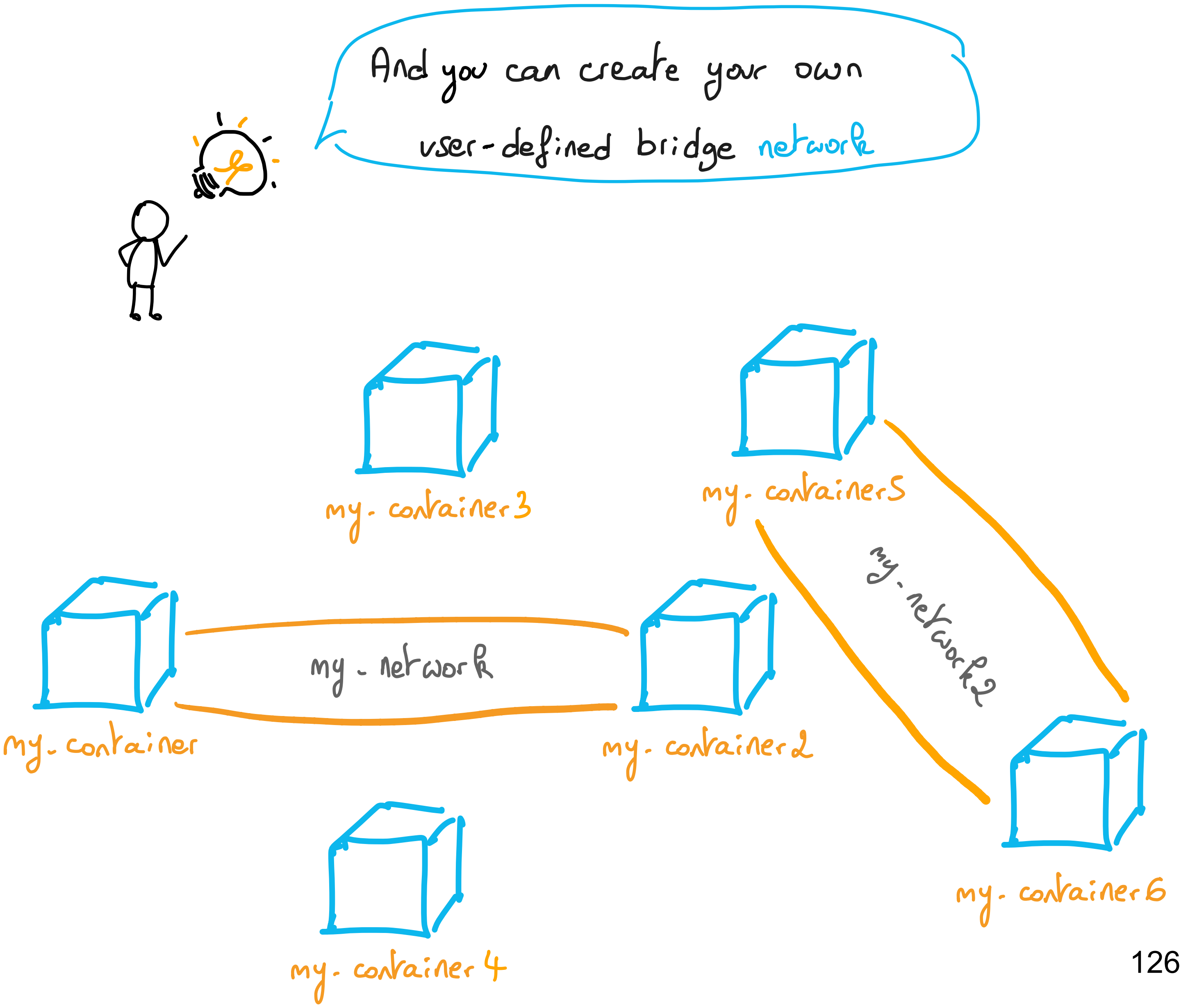
→ By default, a container is attached to the default bridge network

→ 3 networks are automatically created by default

bridge host none

bridge

- Default network driver
- Allows containers conected to the same bridge network to communicate to each others
- Isolate other containers not conected to that bridge network



host

- For standalone containers
- Remove network isolation between the container & Docker Host
- Used host's networking directly

none

- Disable all networking for a container

overlay

- Connect multiple Docker daemons together
- Used to facilitate communication between a swarm service & a standalone container

macvlan

- Assign a MAC address to a container
- Making it appear as a physical device
- Docker daemons routes traffic to containers by their MAC addresses

Third-party network plugins can also be installed

How To ?

- List existing networks

```
$ docker network ls
```

- Run a container that need to be isolated

```
$ docker run --network=none ubuntu
```

- Connect a container to an existing network

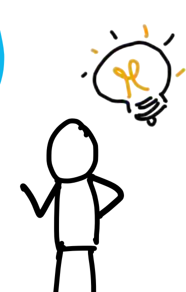
```
$ docker network connect my-network nginx
```

- Disconnect a container from a user-defined bridge network

```
$ docker network disconnect my-network nginx
```

Let's troubleshoot network issues

Without changing an existing running container, we can run a new container to debug / troubleshoot it



① Launch a container (that has potentially an issue)

```
$ docker run -d —name my-broken-container nginx
```

② Run another container that share the same network

```
$ docker run -it --network container:my-broken-container alpine
```

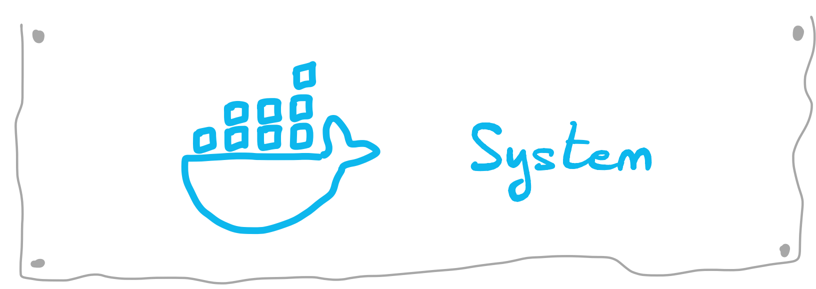
we create a namespace that can be shared for troubleshooting troubles

Then, inside it, install tools for debugging:

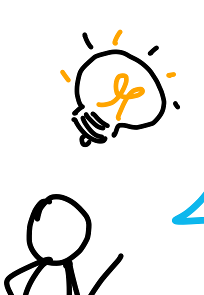
```
$ apk add --update-cache iproute2 bind-tools net-tools
```

And finally debug!

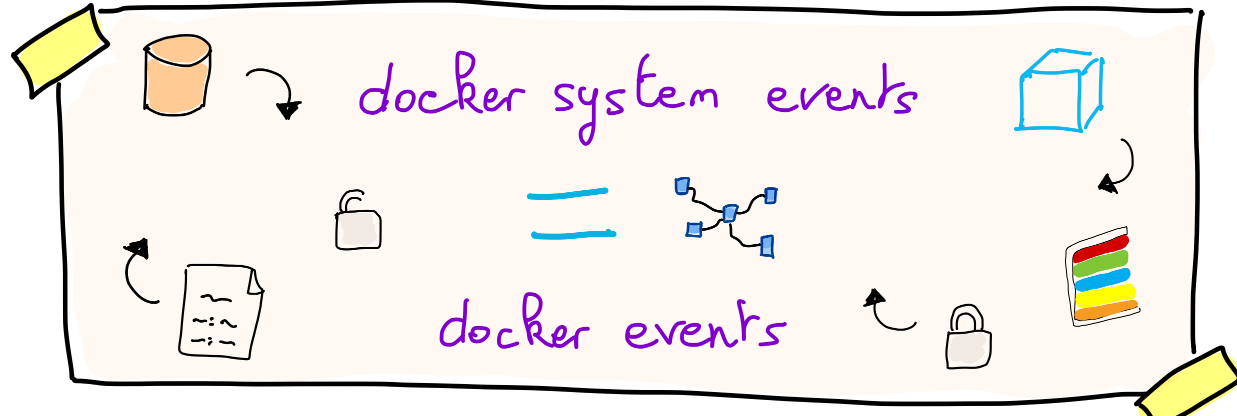
```
$ nslookup localhost
$ netstat -laptn
```



→ Useful commands to query Docker internally



The half of docker system commands have shortcuts



→ Real-time events from server

docker system info
=
docker info

client
runc
kernel
experimental
Debug mode
CPU
Server

→ Display sytem-wide information

How To ?

- Display all sytem-wide information

```
$ docker system info
```

- Display Kernel version system information only

```
$ docker system info -f '{{ .KernelVersion }}'
```

- Display Docker disk usage

```
$ docker system df
```

- Display detailed Docker disk usage

```
$ docker system df -v
```

- Display Docker disk usage in JSON

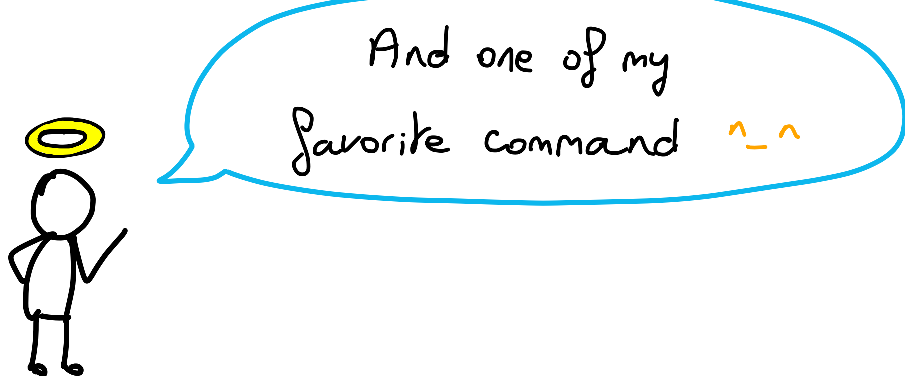
```
$ docker system df --format '{{ json . }}'
```

〉 Display events in real-time

```
$ docker system events
```

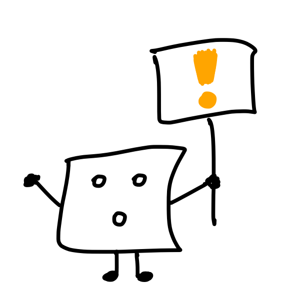
〉 Display events for my-container container since 15 minutes

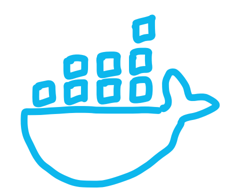
```
$ docker system events
--filter 'container=my-container' --since '15min'
```



〉 Delete all unused images, containers, networks, volumes, dangling images & build cache

```
$ docker system prune -a --volumes
```

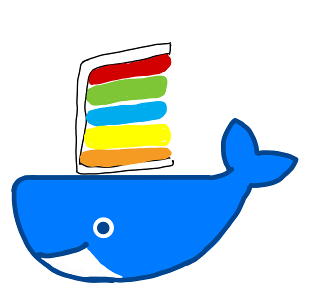




Manifest

experimental

→ Information about an image layers:

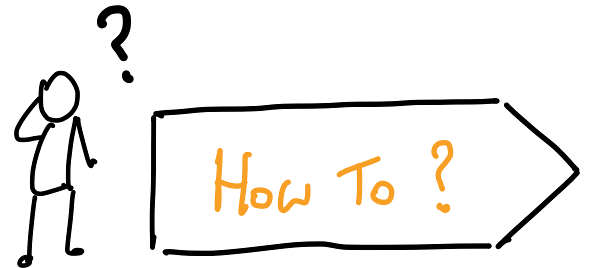


- size
- digest / hash
- architecture
- OS

→ Each manifest entry represents a different variant of the image (x86, AMD64, ARM64 ...)

Docker uses manifests to know if an image is compatible with the current device.

& how to start a new container.



- Display the manifest for busybox image

```
$ docker manifest inspect busybox
```

- Display the manifest for busybox image + the image's tag, architecture & OS

```
$ docker manifest inspect busybox -v
```

- Display the manifest for an image from an insecure registry

```
$ docker manifest inspect --insecure 127.0.0.1/my-image:tag
```

Create multi-architecture images (AMD64 & ARM64)

```
# AMD64 Architecture
$ docker build -t my-image:amd64 .
$ docker push my-image:amd64

# ARM architecture
$ docker build -t my-image:arm64 .
$ docker push my-image:arm64

# Combine the manifests
$ docker manifest create my-image:my-tag
  --amend my-image:amd64
  --amend my-image:arm64

# Push the manifest
$ docker manifest push my-image:my-tag
```

Thanks to this manifest, AMD64 & ARM64 users can create containers from this image

Or

```
$ docker buildx build
--platform darwin/amd64,darwin/arm64
--tag my-image:my-tag
```

Annotate an image

```
$ docker manifest annotate my-image:my-tag my-image:amd64
--os-version darwin --arch amd64
```

Delete the manifest for my-image

```
$ docker manifest rm my-image:my-tag
```

When an error occurs, keep calm & analyze step by step :)

1 Check containers

→ Display all containers (even stopped / inactive / terminated)

```
$ docker container ls -a
```

2 Show logs containers

→ Don't hesitate to watch container's logs in order to try to understand what is happening

→ Cool, but I want to stream the logs in realtime :

```
$ docker logs -f my-container
```

-f / --follow : stream logs from STDOUT & STDERR

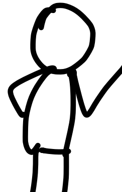
③ Exec in a container

→ Debug in your container & execute commands directly into it :

```
$ docker exec -it ubuntu bash
```

-it / --interactive --tty : interactive shell

You can watch specific logs



```
$ tail -f /var/log/access_log
```

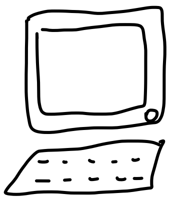
④ List published ports exposed by a container

My container exposes ports ... but which ones?

```
$ docker port my-container
80/tcp -> 0.0.0.0:28600
443/tcp -> 0.0.0.0:33000
```

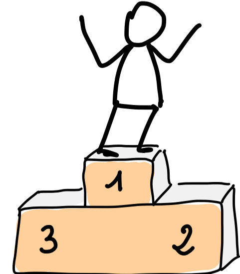
remote ↗ ↖ locally

OK, so now you can simply curl it!



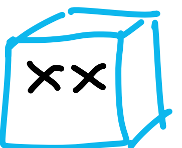
```
$ curl http://localhost:28600
```

```
$ docker top my-container
```



6 Get memory & CPU usage

→ Useful to monitor and/or analyze a potential memory leak



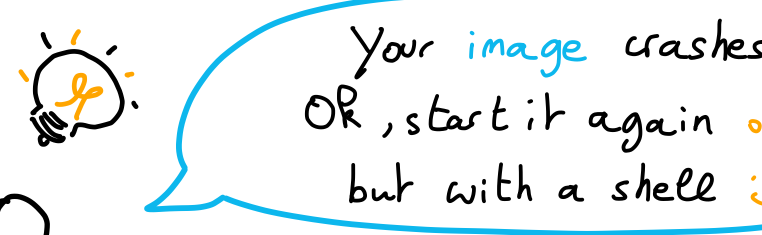
```
$ docker stats my-container
```

7 Get container's IP address

It's me again, now I want to know the IP!

```
$ docker inspect
--format '{{NetworkSettings.IPAddress}}' my-container
```

8 Bypass actual container's entrypoint



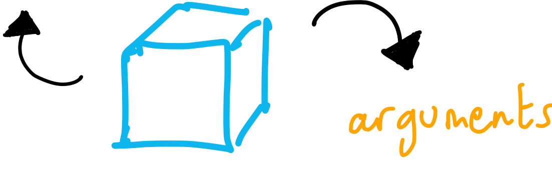

```
$ docker run --entrypoint "/bin/sh folder/" my-image
```

→ Now, the container is running, so you can execute a shell into it & debug its configuration for example!

9 View container's details

→ You can't inspect not only IP address but also others useful information:

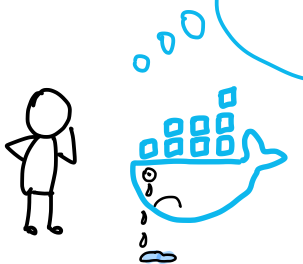
environments variables




```
$ docker inspect my-container
```

10 Watch Docker events

My container is restarting again & again ...



> Run container & display in realtime related container's events

```
$ docker run xxx & docker events
--filter 'container=$(docker ps -lq)'
```

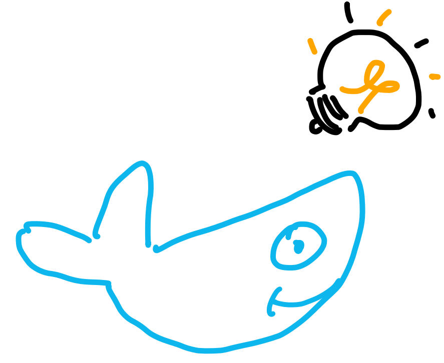
- -l : last
- -q : quiet

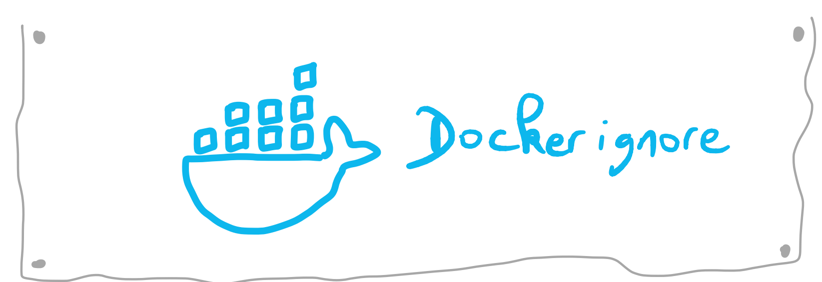
11 And ... you can also ...

→ Run docker command in debug mode (Will print more information)

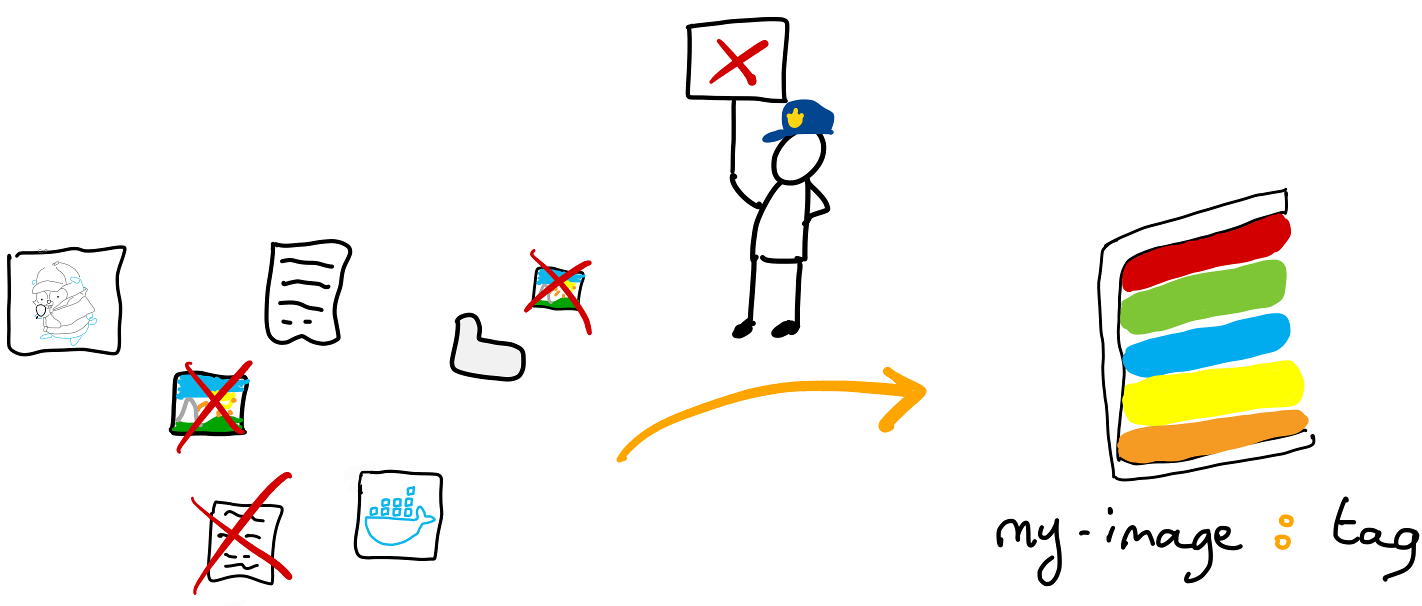
```
$ docker run -D xxx
```

-D / --debug : debug mode



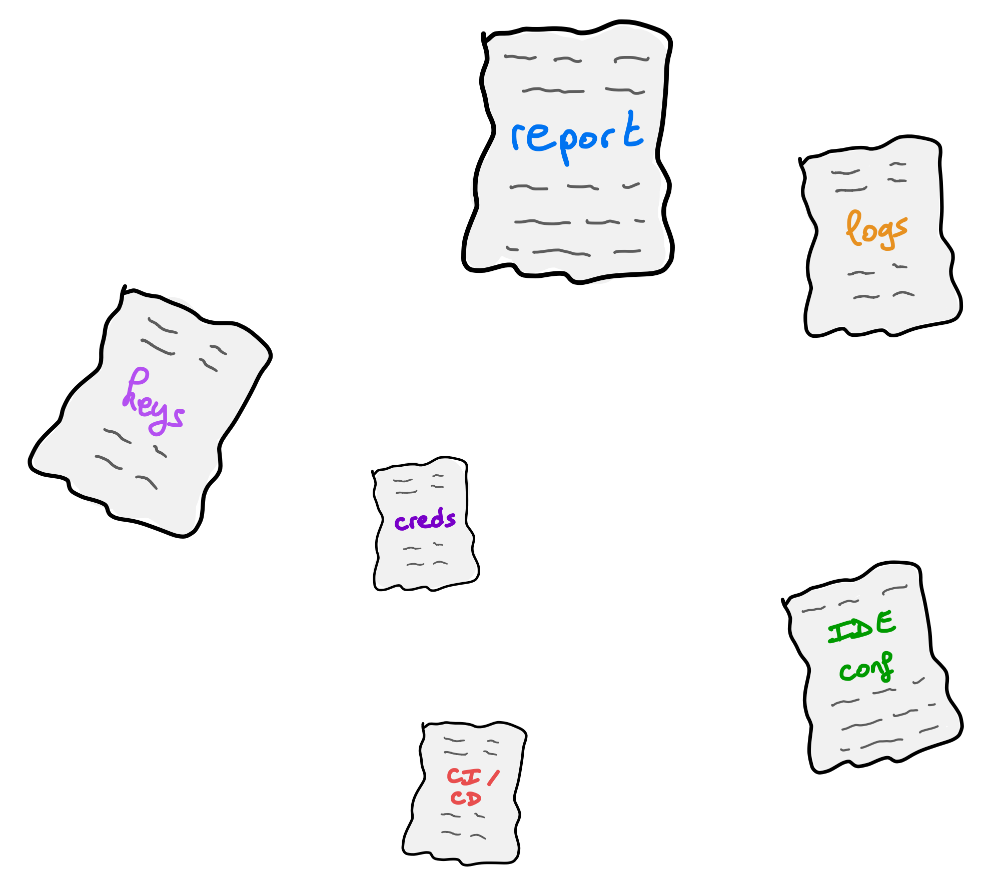


→ Define rules & exceptions for files and folders to be excluded from the build context

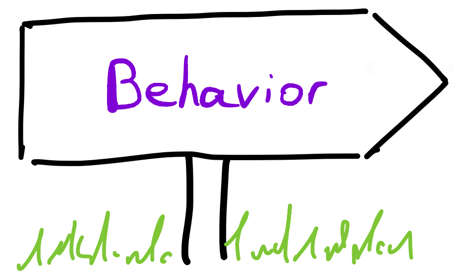


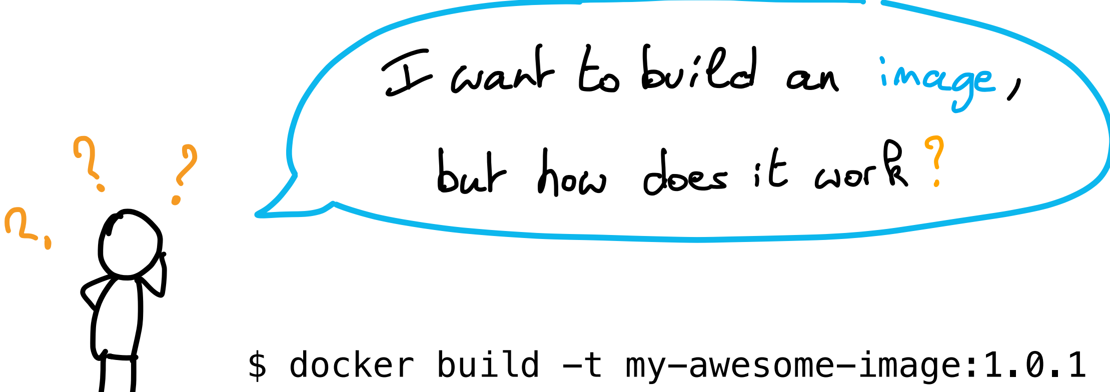
→ Same as .gitignore but for Docker

→ Allows to not copy / include in the image sensitive information

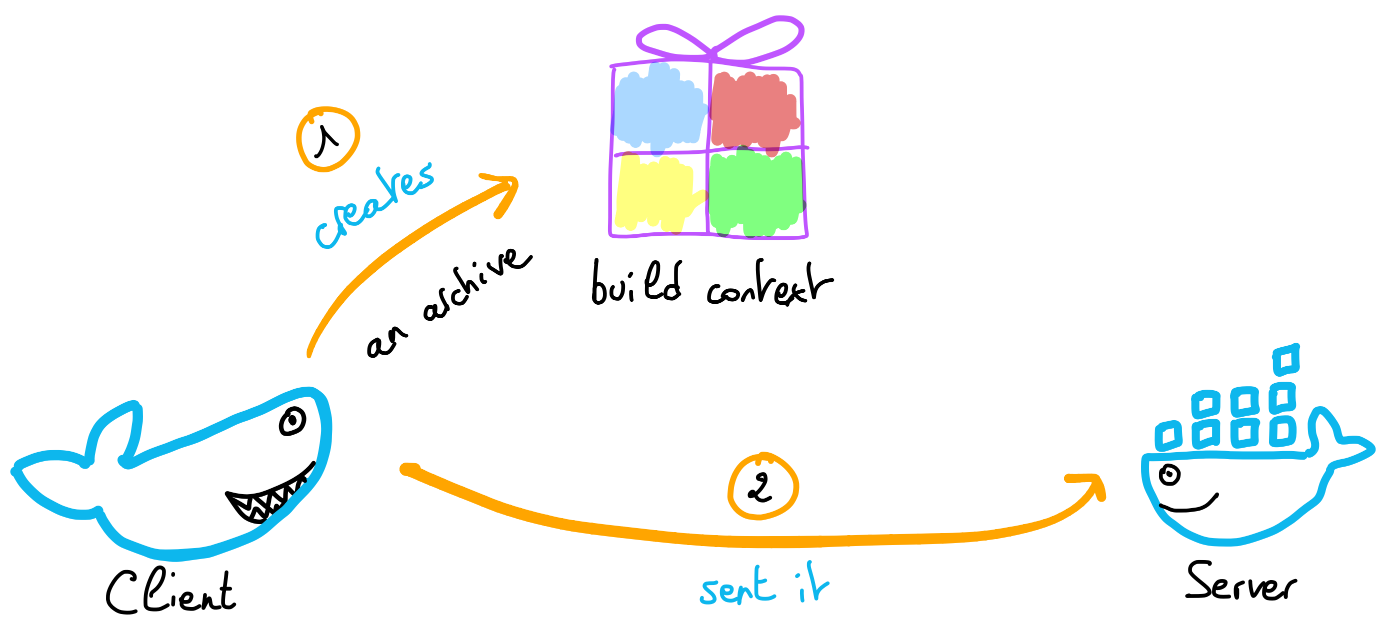


Helps to reduce images size






```
$ docker build -t my-awesome-image:1.0.1 .
```



For the build context creation, I follow the DockerFile & exclude files listed in .dockerignore

What I need to follow/do

FROM —
ADD —
COPY —

DockerFile

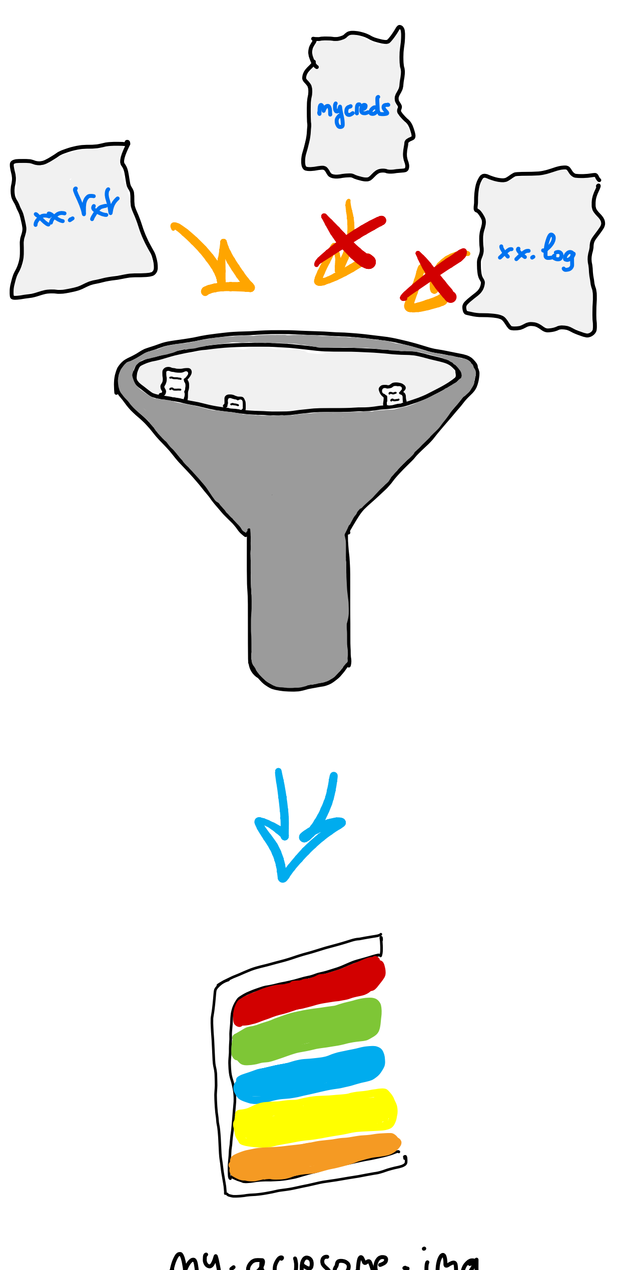
Files I need to exclude

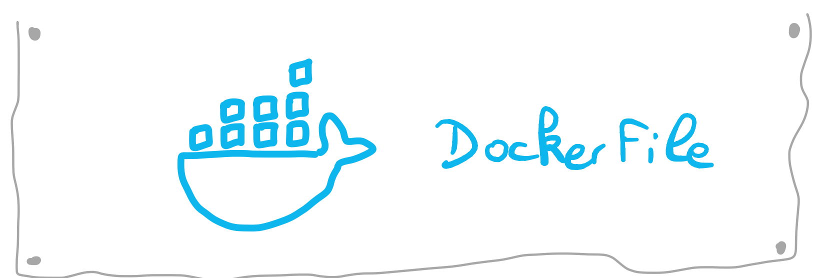
.aws
mycreds.txt
sensitive.log

.dockerignore

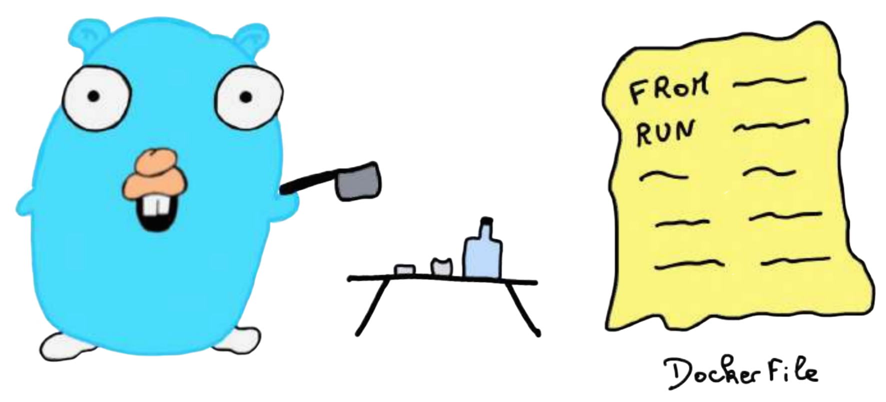
The final Docker image contains only files you want, even if you do a COPY or an ADD in the DockerFile

Don't add files in the build context if you don't need them



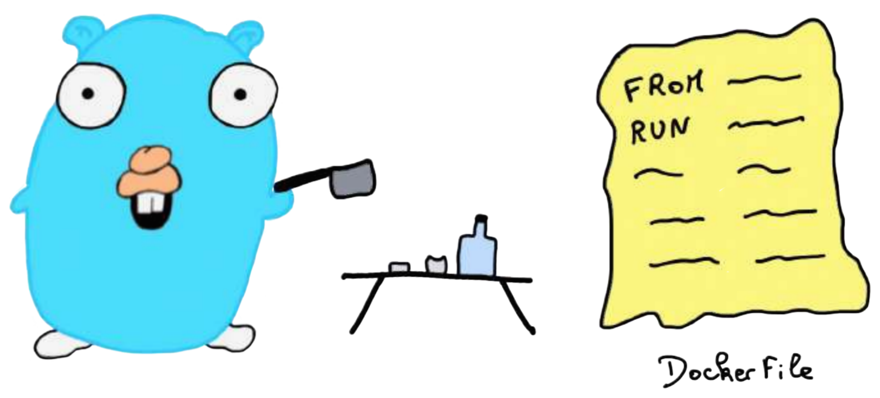


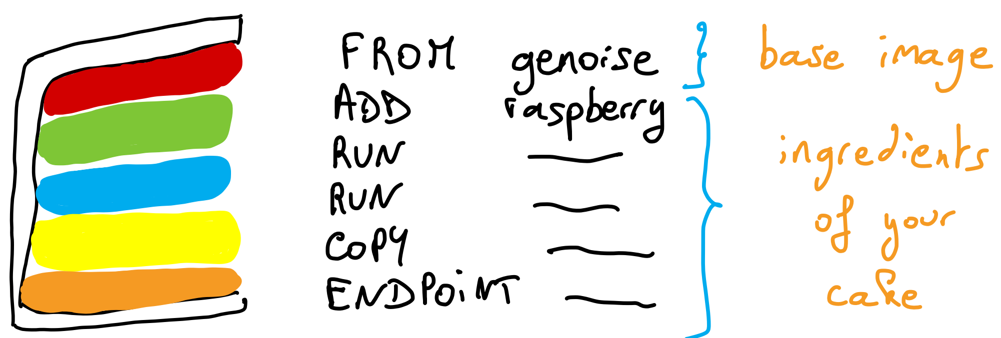
→ DockerFile contains a set of instructions that Docker follows to build an image

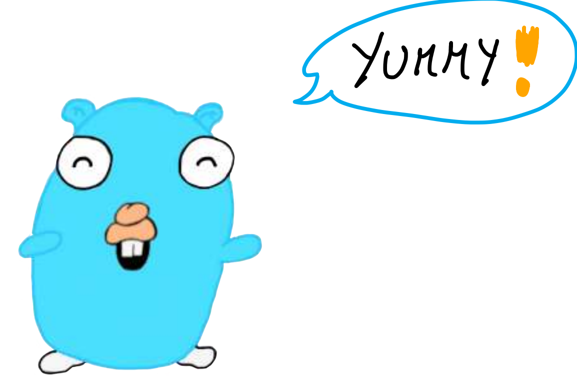


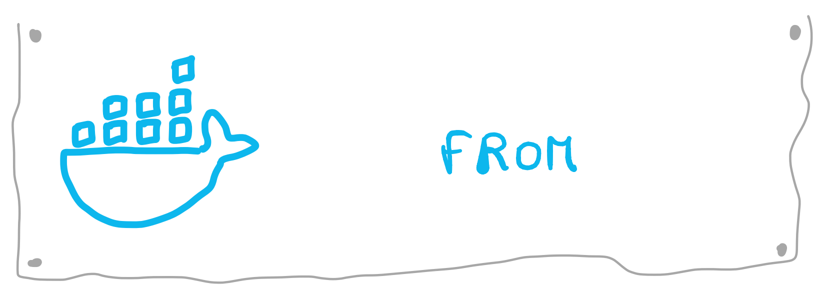
Through a DockerFile, Docker defines What to run & How to run

Lines in a DockerFile are ingredients of our image



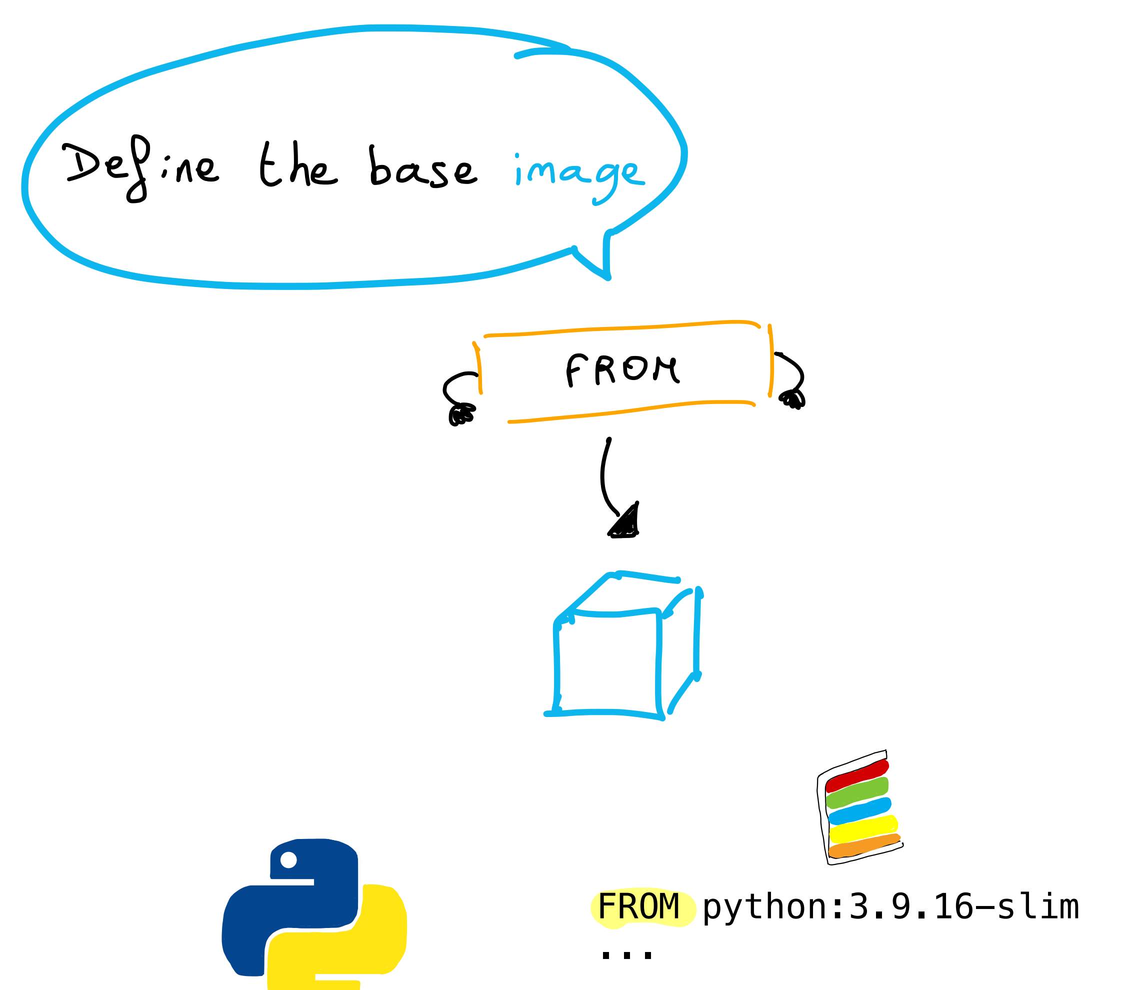


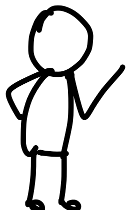




→ To define an instruction that initializes a DockerFile

FROM instruction

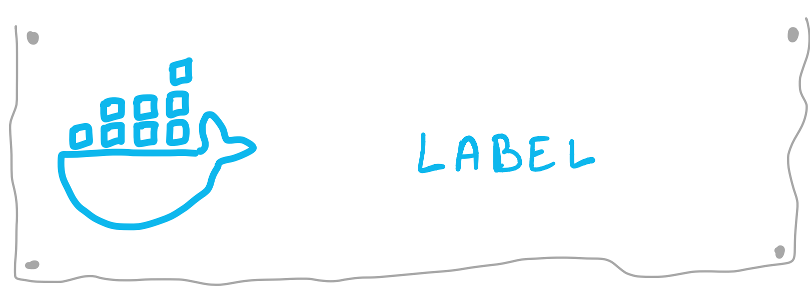




It's possible to specify the target platform

```
FROM --platform linux/arm64 my-image:tag
```

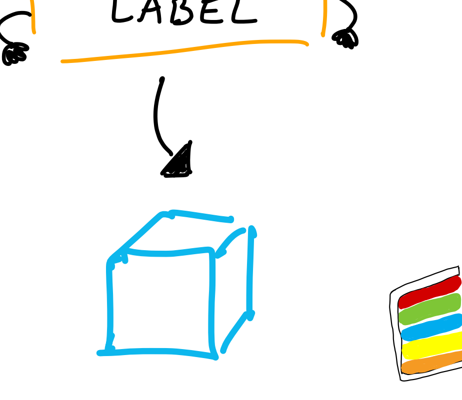
ARG is the only instruction you can put before a FROM



→ Adds metadata to an image (version, description, project ...)

LABEL instruction

Adds information to an image




```
FROM my-image:tag

LABEL version="2.0.0"
LABEL description="my text \
in several lines"
LABEL com.scraly.app="My app"
```

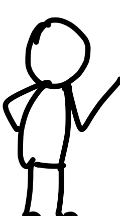
Use LABEL instruction instead of deprecated MAINTAINER

How To ?

Display labels of an image

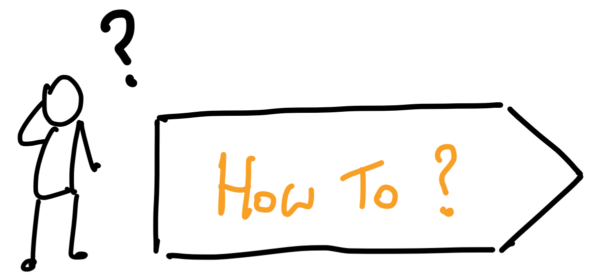
```
docker image inspect
--format '{{ .ConfigLabels }}' my-image:tag
```

Base image's labels are inherited by your image

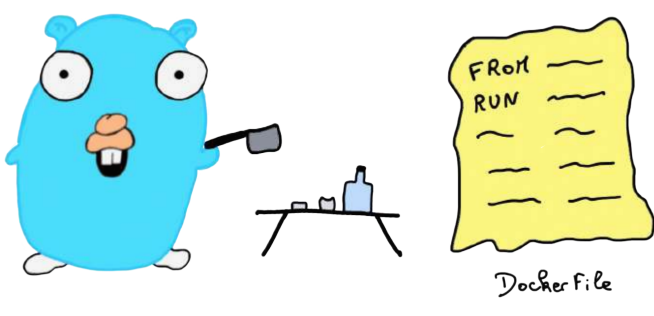


ARG instruction

- Defines an argument that can be passed at build time
- Allows to override build arguments
- Useful to not hardcode information in Dockerfile



1 Pass image name and version & override values in DockerFile




```
ARG MY_IMAGE=golang
ARG VERSION=latest
FROM $MY_IMAGE:VERSION as build
```

> Build golang:latest docker image

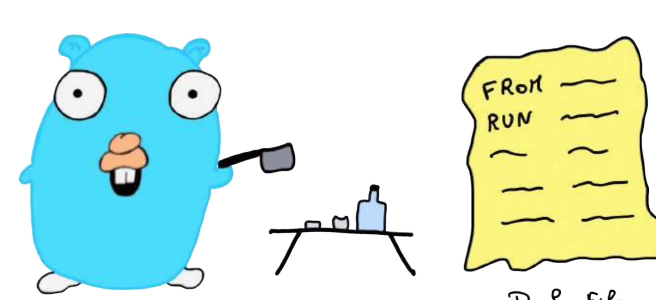
```
$ docker build -t my-image:tag .
```

> Build bitnami/golang:alpine3.12 docker image

```
$ docker build -t my-image:tag
--build-arg MY_IMAGE=bitnami/golang
--build-arg VERSION=alpine3.12
.
```

ARG instruction should be defined before the FROM instruction in order to have an effect to the FROM.

2 Pass information to DockerFile



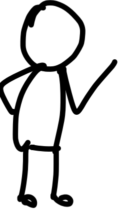

```
FROM busybox
ARG user
RUN echo "user is $user"
```

> Build image with user value

```
$ docker build -t my-image:tag
--build-arg user=moby
.
```

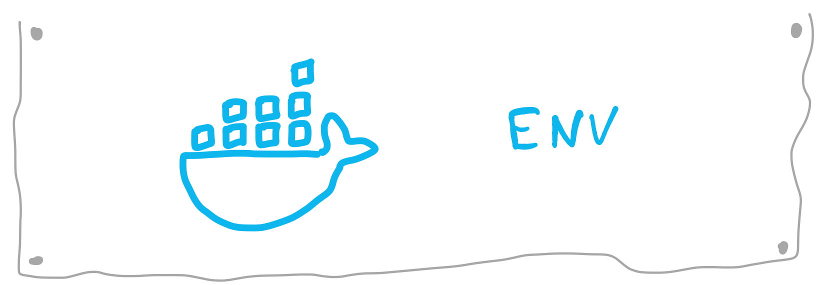
Otherwise, define ARG instruction after FROM.

Yes, ARG have a scope ! ◡̈



It's cool, but don't use ARG instructions for sensitive information (credentials, passwords...)

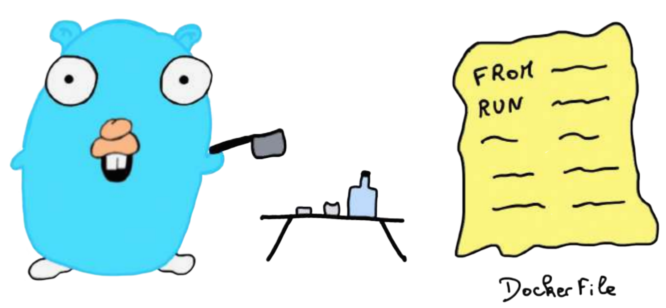
Build-time variable are visible through docker history command



ENV instruction

- Defines an environment variable that can be passed at build time
- Set environment variables to a container

1 Pass an environment variable from argument

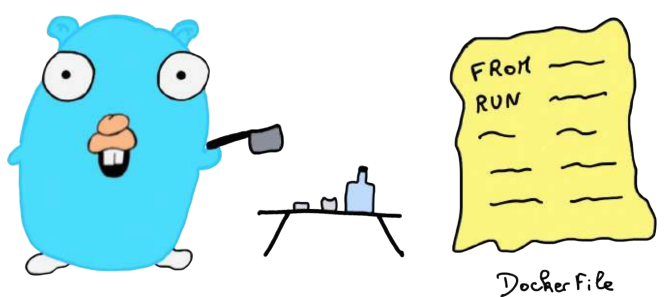



```
FROM alpine:latest
ARG PORT_ARG=3000
...
ENV PORT=$PORT_ARG
COPY ./package.json .
...
```

> Build image & run a container

```
$ docker build -t my-image:tag .
$ docker run my-image:tag
```

② Pass environment variables to our favorite container

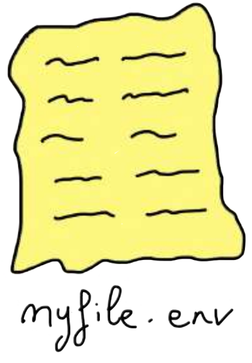



```
FROM golang:1.19
ENV ENV=INT
ENV REGION=eu-central-1
COPY ./entrypoint.sh /app/entrypoint.sh
WORKDIR /app
ENTRYPOINT ["./entrypoint.sh"]
```

```
$ docker run
-e "ENV=prod"
-e "REGION=eu-central-1"
my-image:tag
```

-e
--env } same!

3 Pass environment variables from an environment file



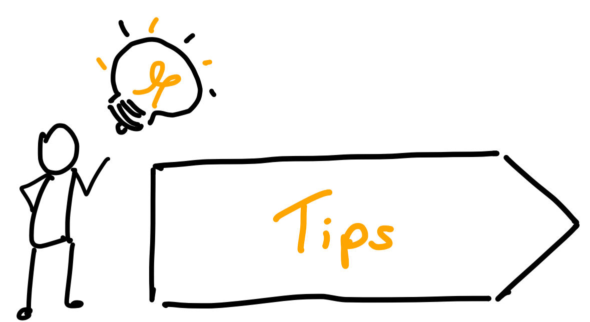

```
ENV=prod
REGION=eu-central-1
```

Run a container

```
$ docker run --env-file=my-file.env my-image:tag
```

Run a container from busybox image & display env variables

```
$ docker run
-e KEY=VAL
--env-file=my-file.env
busybox env
```



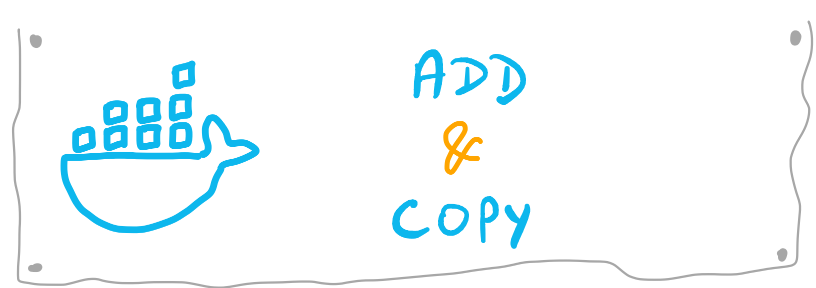
If you pass an environment value through -e & --env-file, the final value will be the one provided by -e option

1. Display the content of our environment file

```
$ cat myfile.env

ENV=prod
```

2. Run a container & display environment values equals to "ENV"

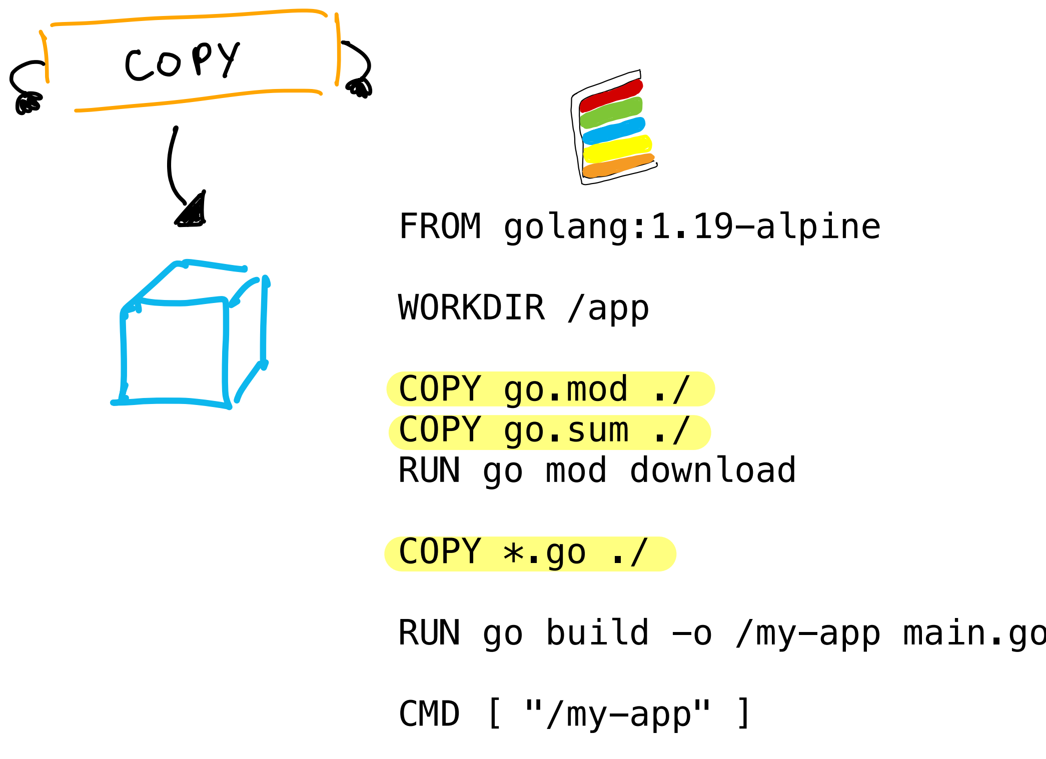
```
$ docker run -e ENV=qa --env-file=myfile.env
ubuntu:latest env | grep ENV

ENV=prod
```



→ To copy local files & directory in an image, specify at least ADD or COPY instructions in the DockerFile

COPY instruction

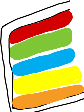
Adds files & directory from a host to an image

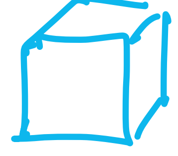


Use COPY if you just want to copy files / directory into the build context

ADD instruction

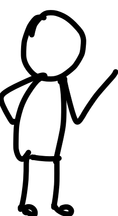
Like COPY + more features
(local-only tar extraction
& remote URL support)





```
FROM busybox:latest
WORKDIR /app

ADD https://go.dev/dl/go1.19.4.linux-amd64.tar.gz ./

RUN tar -xzf go1.19.4.linux-amd64.tar.gz \
   && rm go1.19.4.linux-amd64.tar.gz

CMD ls -alrt
```

Allows <src> input as file or an URL. But be careful for cache usage it's recommended to use curl command in RUN instruction instead

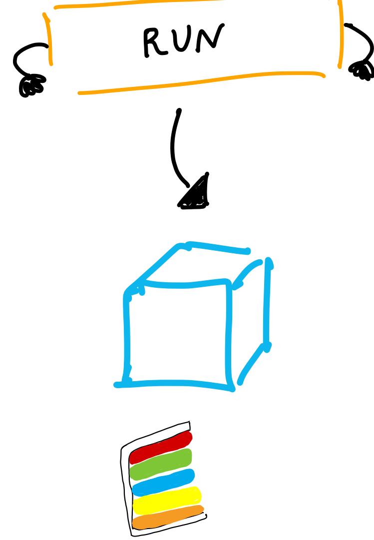


Copy is preferred for basic copy, no "magic tricks"

→ To define commands to execute

RUN instruction

Execute commands & use the result in other commands




```
FROM ubuntu

## Install dependencies
RUN apt update && \
    apt install curl cowsay -y

RUN cp /usr/games/cowsay /usr/local/bin/cowsay
```

How does it work?

1. Build this image

```
$ docker build -t cowsay .
```

2. Display the layers

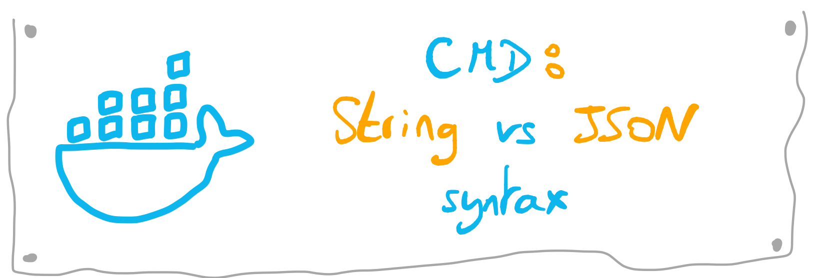
```
$ docker image history cowsay
```

3. Run the container & execute cowsay with a message

```
$ docker run -it cowsay
```

root@af73e1b0c1f3:~$ cowsay hello

```
  _____
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```



CMD instruction

→ You can define CMD instruction in two ways

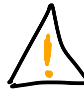
String syntax ≠ JSON syntax

String syntax

→ Be careful, this instruction executes a command **with** a shell

```
CMD ["./hello"]
```

```
/bin/sh -c "./hello"
```



There's no shell in every images!

This command can cause an error:

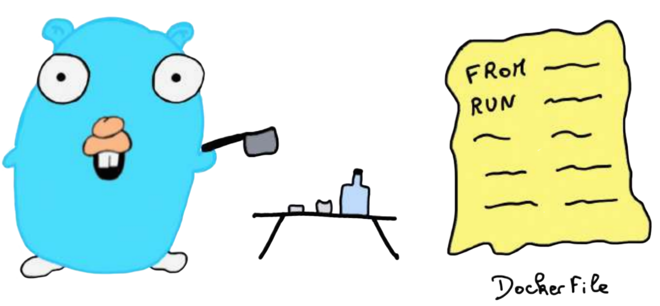
```
exec: "/bin/sh": stat /bin/sh:
no such file or directory
```

JSON syntax

→ This command executes arguments without a shell

```
CMD ["./my-command"]
```

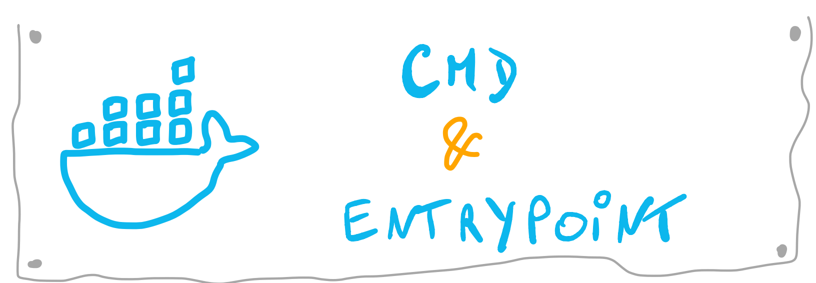
→ Let's take a look with a concrete example:




```
FROM golang as build
COPY hello.go .
RUN go build hello.go

FROM scratch
COPY --from=build /go/hello.
CMD ["./hello"]
```

→ Execute a command with a different shell (e.g., bash):

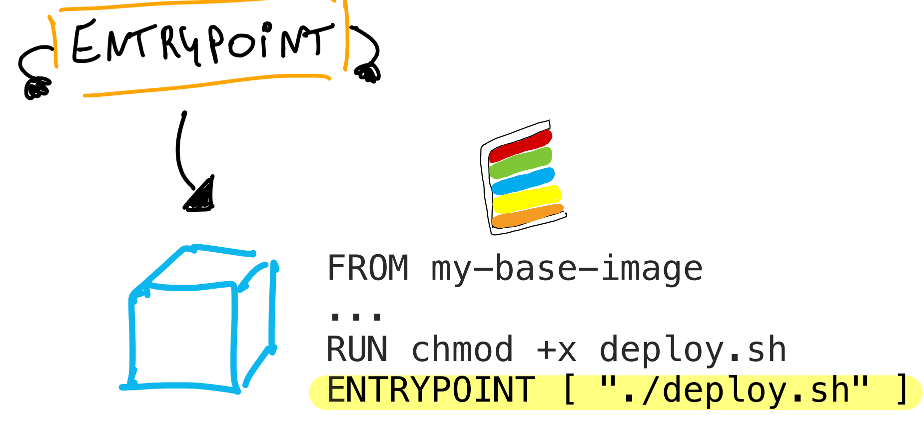
```
CMD ["bash","-c", "echo $HOME"]
```



→ Specifies at least CMD or ENTRYPOINT instructions in the DockerFile

ENTRYPOINT instruction

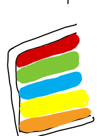
Defines executable to run when the container starts

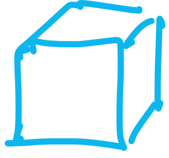


⚠ ENTRYPOINT instruction (command & parameters) are not ignored when the command docker run is executed with parameters

ONLY CMD

Defines executable + parameters for a container





```
FROM node:alpine
WORKDIR /usr/src/app
COPY package.json .
RUN npm install --only=production
COPY . .
CMD [npm start]
```

With ENTRYPOINT

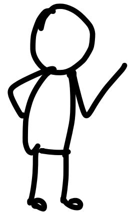
Defines executable to run when the container starts

ENTRYPOINT — CMD

Add options

```
$ kubectl exec my-pod -it -- ls
```

```
FROM golang:1.19.4
COPY --from=build /src/bin/redis-tools /redis-tools
ENTRYPOINT [ "/redis-tools"]
CMD [ "--help" ]
```

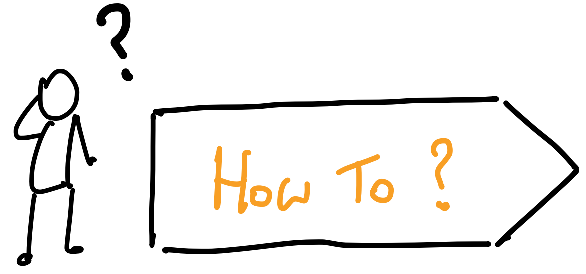


What happens if I only specify CMD instruction?

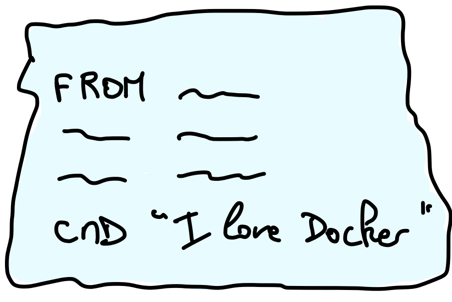
Implicit entrypoint is `/bin/sh -c`

Use ENTRYPOINT & CMD combo

when you run the same executable every time :)



Run a container that displays a message "I love Docker"



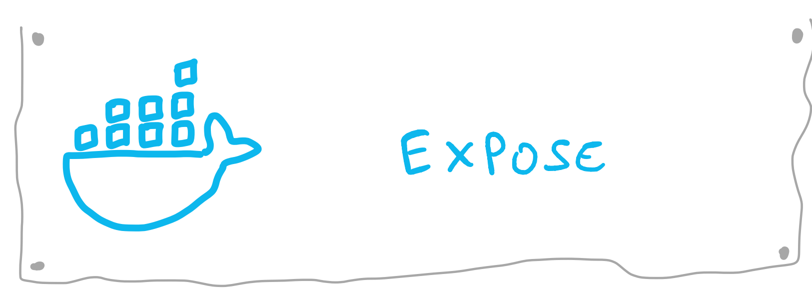

```
$ docker run -it my-image
```

Run a container with the same image but overrides the default command that allows you to run/exec into it

```
$ docker run -it my-image /bin/bash
```

Run a container & override ENTRYPOINT instruction

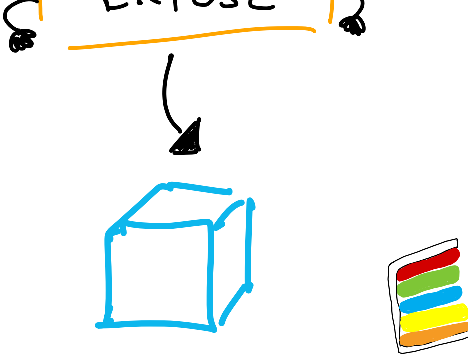
```
$ docker run my-image --entrypoint=‘’
```



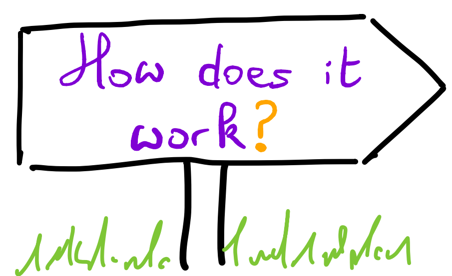
→ To define the port & the protocol that the container listens on

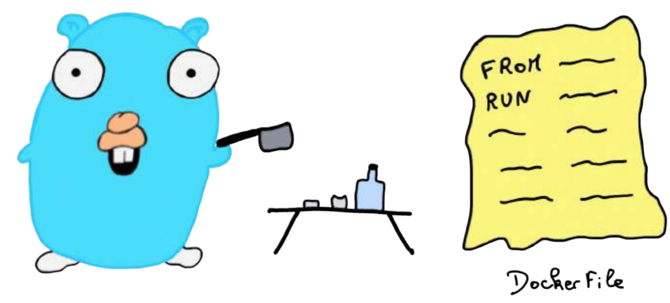
EXPOSE instruction

It's like a documentation to know which port to publish




```
FROM golang:1.19-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./

RUN go build -o /my-app main.go

EXPOSE 8080
CMD [ "/my-app" ]
```






```
FROM my-image:tag

...

EXPOSE 80/tcp
EXPOSE 80/udp

...
```

1. Build this image

```
$ docker build -t my-image:tag .
```

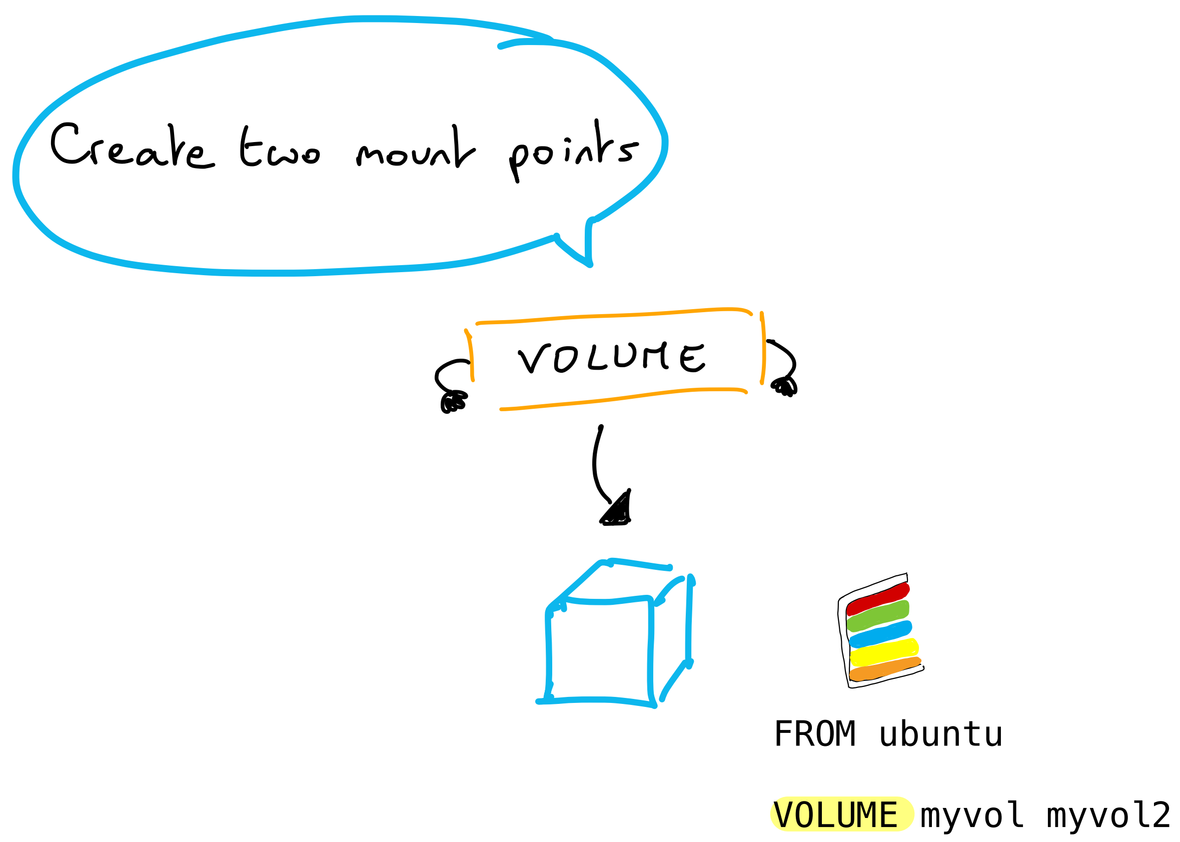
2. Run the container & expose port for TCP and UDP

```
$ docker run -p 80:80/tcp -p 80:80/udp my-image:tag
```

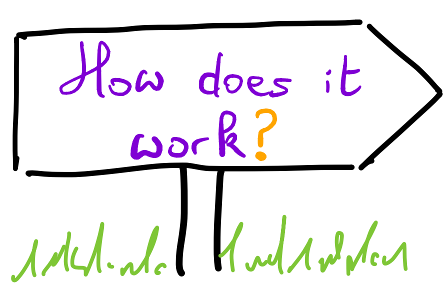
By default, Docker exposes the TCP protocol

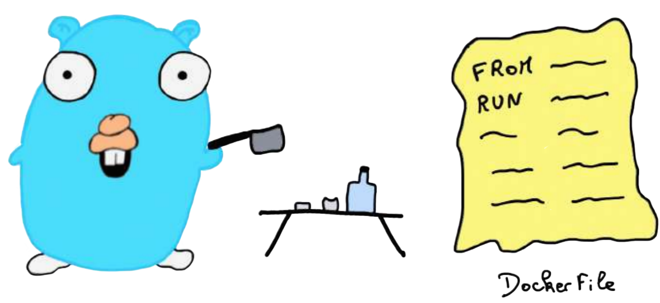
→ To define one or several mount points

VOLUME instruction




```
FROM ubuntu

VOLUME myvol myvol2
```






```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello docker lovers" > /myvol/docker
VOLUME myvol
```

```
$ docker build -t my-image-with-vol .
```

2. Run the container & verify the existance of the volume and the content of the "docker" file

```
$ docker run -it my-image-with-vol
```

```
root@af73e1b0c1f3:~$ ls /myvol
docker

root@af73e1b0c1f3:~$ cat /myvol/docker
hello docker lovers
```

3. List existing volumes

```
$ docker volume ls
```

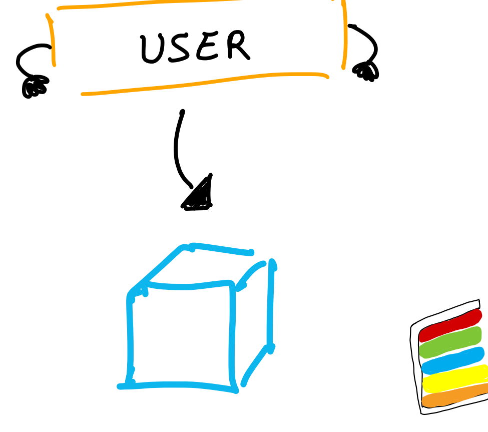
" VOLUME myvol " :

No need to specify / before a path, because the default working directory within a DockerFile is /

→ To define the user name (& group) to use as the default user (& group)

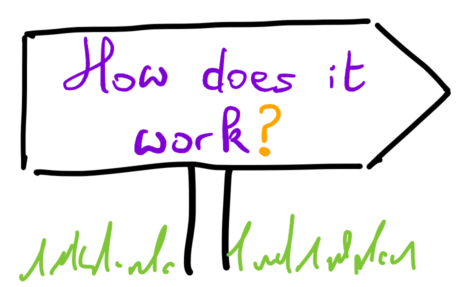
USER instruction

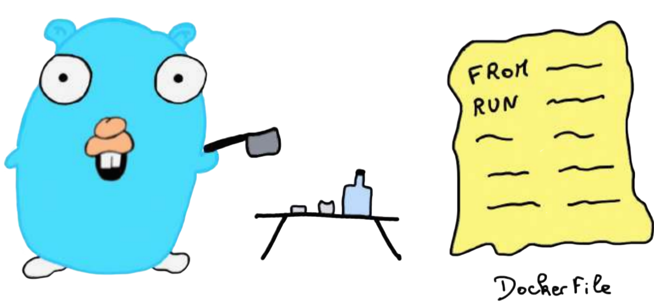
I tell which user, & optionally group to use



create the user & the group first

```
FROM ubuntu:latest

RUN groupadd -r mygroup \
   && useradd -r -g mygroup -ms /bin/bash myuser

USER myuser
WORKDIR /home/myuser
```






```
FROM ubuntu:latest

RUN groupadd -r mygroup \
  && useradd -r -g mygroup -ms /bin/bash myuser

USER myuser
WORKDIR /home/myuser
```

- -m : user's home directory
- -s : user's new shell

1 Build this image

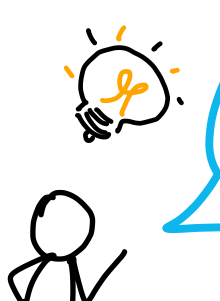
```
$ docker build -t my-image-with-user .
```

② Create a container

& verify you are not root but myuser

& the working directory is myuser's home

```
$ docker run -it my-image-with-user

myuser@af73e1b0c1f3:~$ pwd
/home/myuser
```



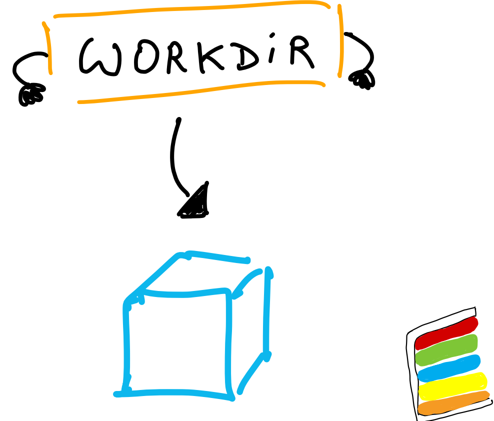
If the user doesn't have
any primary group,
the group by default is
equals to root

The user must exist before the USER
instruction.
Else, the switch cannot work.

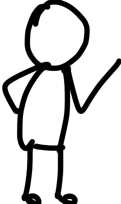
→ To define the working directory for other instructions

WORKDIR instruction

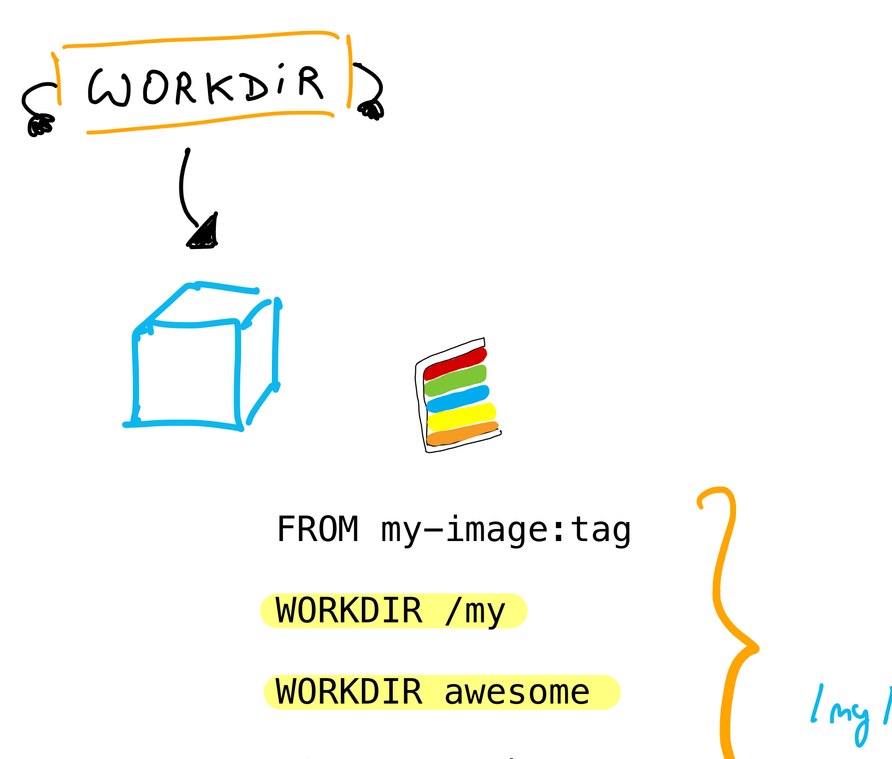
I define the working directory at a given step




```
FROM my-image:tag

# My directives
...
WORKDIR /usr/src/app

# Others directives
...
WORKDIR /tmp/
```



If a relative path is defined, other paths will be appended to the previous one




```
FROM my-image:tag
WORKDIR /my
WORKDIR awesome
WORKDIR path
RUN pwd
```

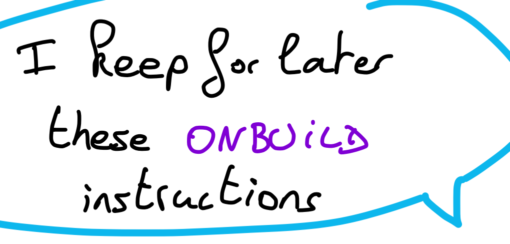
pwd:

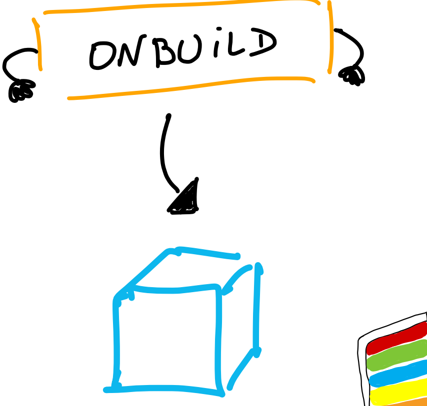
/my/awesome/path

If you don't define a WORKDIR,
by default it is set to /

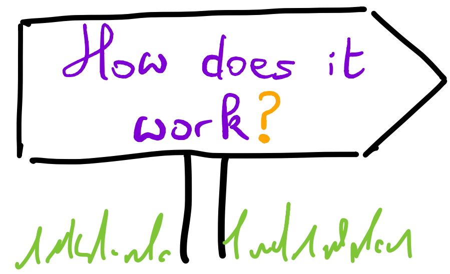
→ To define instructions that will be executed when building a new image

ONBUILD instruction






```
FROM node:0.12.6

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD COPY package.json /usr/src/app/
ONBUILD RUN npm install
ONBUILD COPY . /usr/src/app

CMD ["npm", "start"]
```



1. Build this image

```
$ docker build -t node:0.12.6-onbuild .
```

2. Use it in the FROM instruction in your Dockerfile

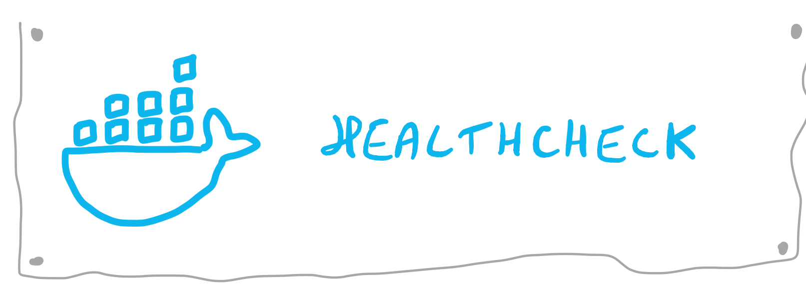
```
FROM node:0.12.6-onbuild
#
...
```

3. Build your image

```
$ docker build -t node:0.12.6-mine .
```

The 3 ONBUILD instructions of the base image are executed just after your FROM instruction

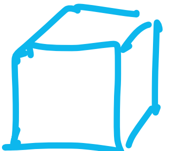
Useful when the image is used as a base image



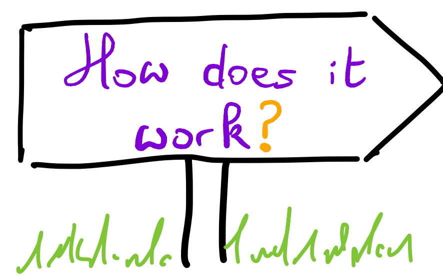
→ To define instructions that Docker needs to test if a container is still working

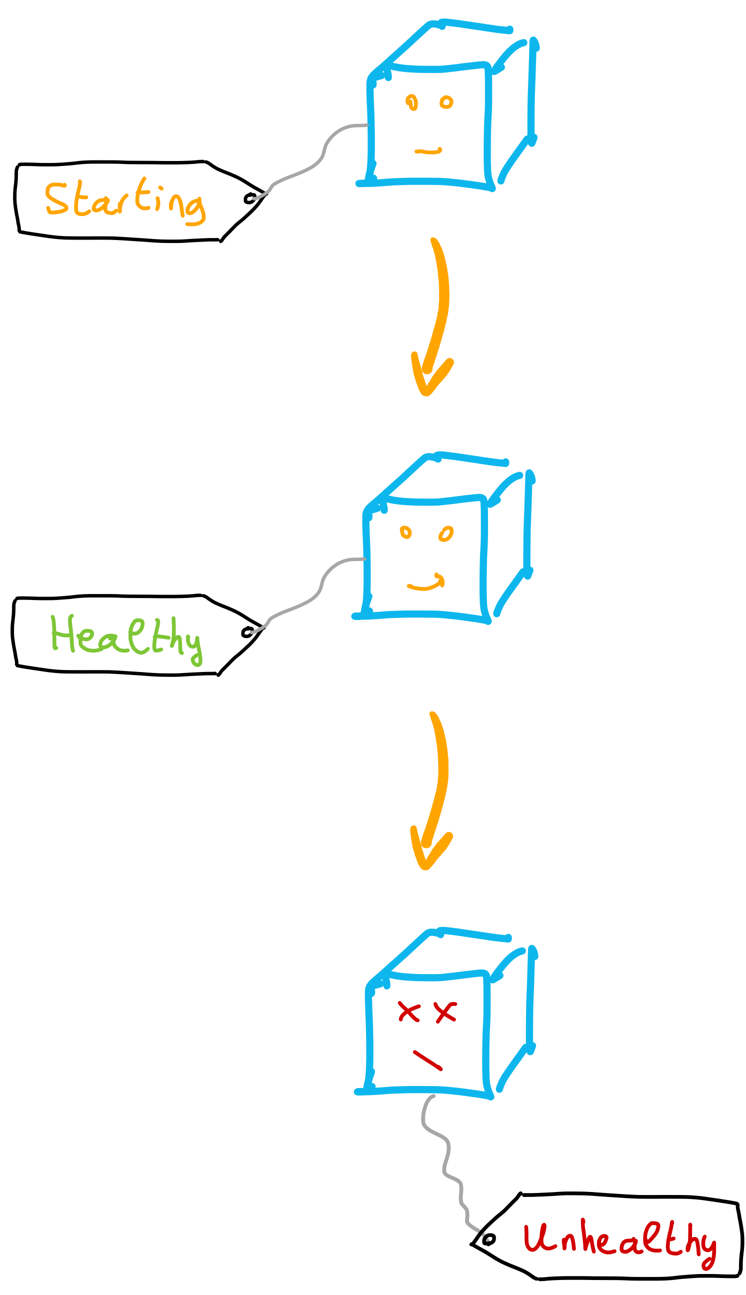
HEALTHCHECK instruction

Hey container,
are you still
alive?



```
FROM golang:1.19-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download

COPY *.go ./

RUN go build -o /my-app main.go

HEALTHCHECK CMD curl --fail
http://localhost:8080 || exit 1

CMD [ "/my-app" ]
```



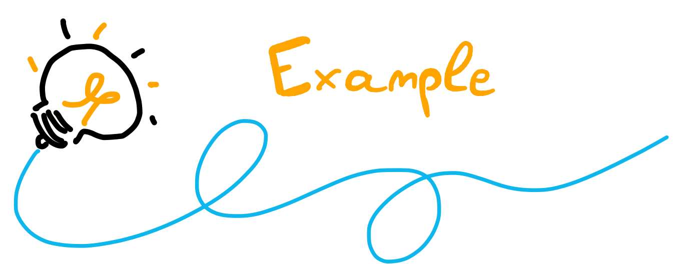


2

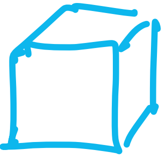
HEALTHCHEK instruction is executed

3

If instruction failed x times, container become unhealthy

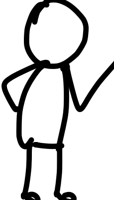


> After a start period of 8 seconds, check every 10 seconds.
> If a check takes longer than 5 seconds, it fails.
> After 5 failed retries, the container is unhealthy.



```
FROM nginx:1.23.3

HEALTHCHECK --interval=10s --timeout=5s
--start-period=8s --retries=5 \
  CMD curl --fail http://localhost:80 || exit 1

EXPOSE 80
```

If you define several HEALTHCHECK instructions, the last will be taken in account

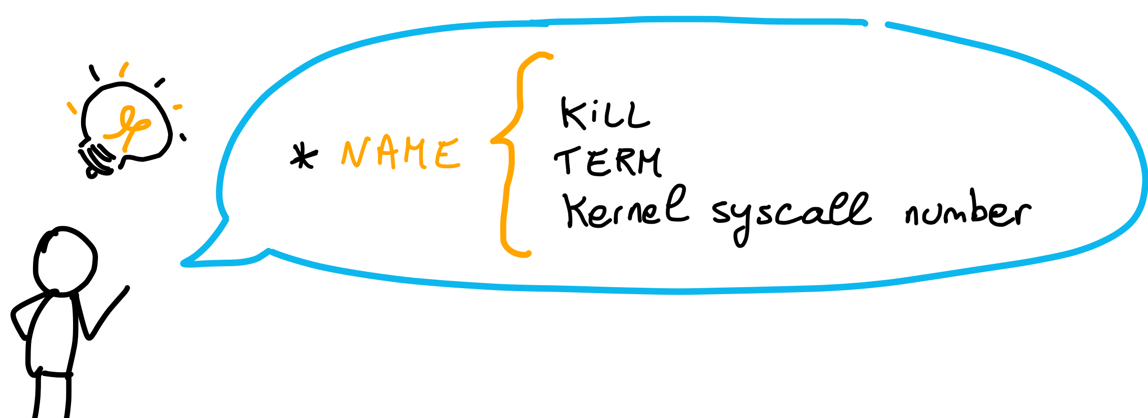


It's also possible to define an HEALTHCHECK instruction in Docker Compose

→ Override the default signal sent to a container to exit

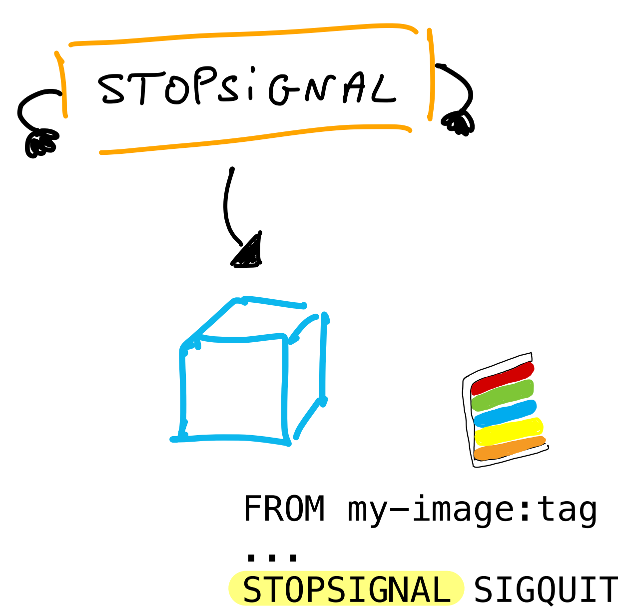
→ The default signal is equals to SIGTERM

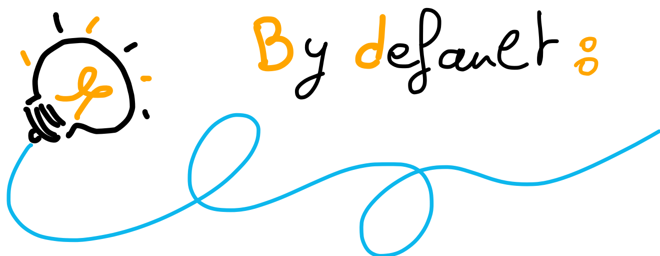
STOPSIGNAL = SIG<NAME>*



STOPSIGNAL instruction

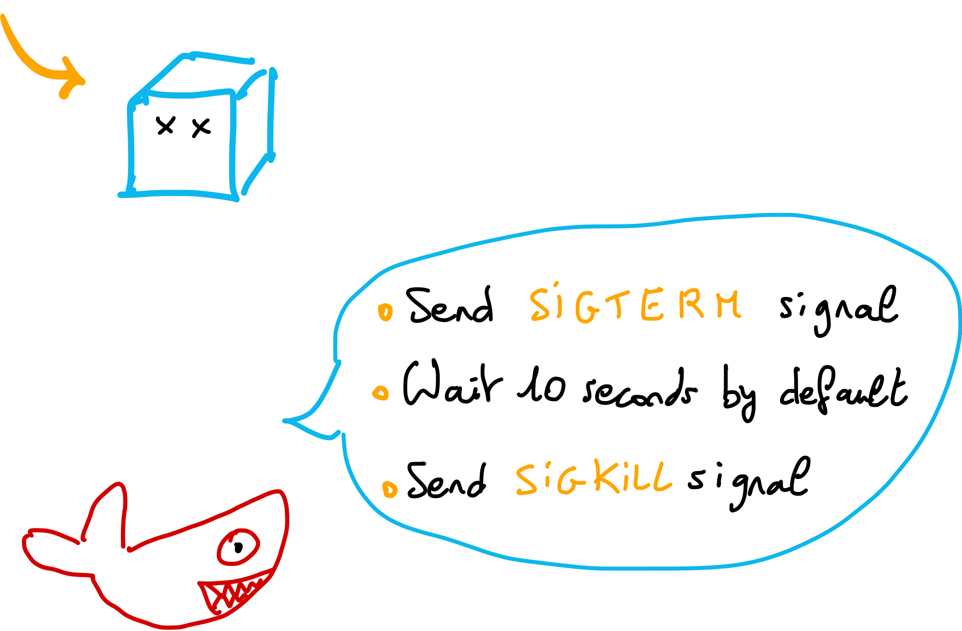
Hey container, here is the new signal when you need to stop



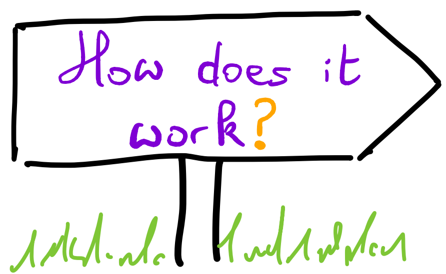


→ docker stop command stops any running containers

```
$ docker stop my-container
```



Ok, but ...



1. Build this image

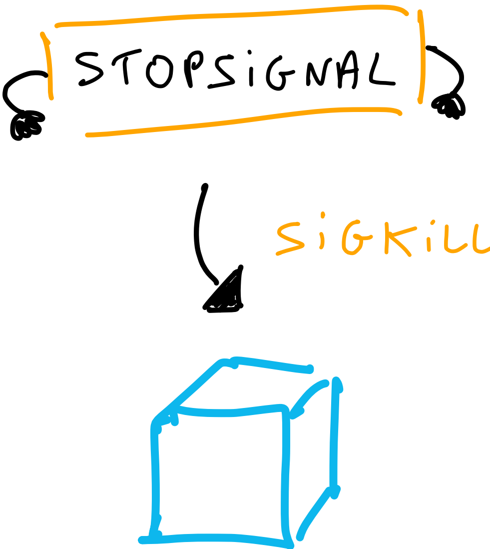
```
$ docker build -t my-app .
```

2. Start a container

```
$ docker start my-app
```

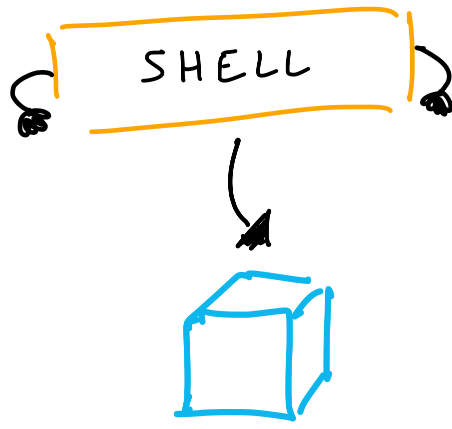
3. Stop the container

```
$ docker stop my-app
```



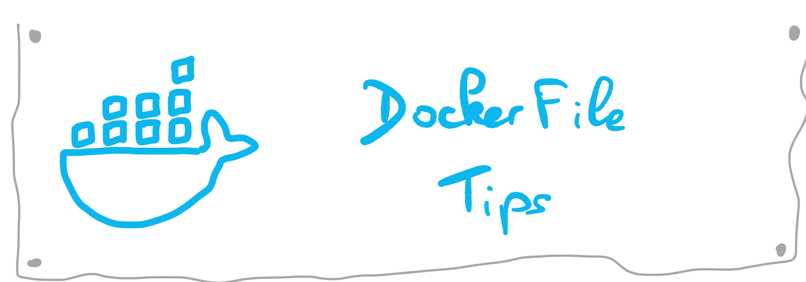
→ Default shell on Linux is ["/bin/sh", "-c"] & on Windows is ["cmd", "/S", "/C"], SHELL overrides the default shell

SHELL instruction



```
FROM windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S"", "/C"]
RUN echo hello
```

Useful for Windows because the OS have two native shells



For a unique need, you can create the DockerFile with different manners & with different instructions.
It will work ... but, maybe it will not be very efficient.
Several tips exist to reduce the size of the images.

Smaller images

=

Faster builds

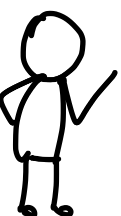
Faster deploys

Smaller attack surface

- Merge multiple RUN statements into single one:

```
RUN apt-get update
RUN apt-get install -y nginx
```

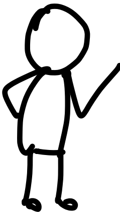
```
RUN apt-get update \
&& apt-get install -y nginx
```



Reduce layers to reduce size

Or > Squash at the end of the build

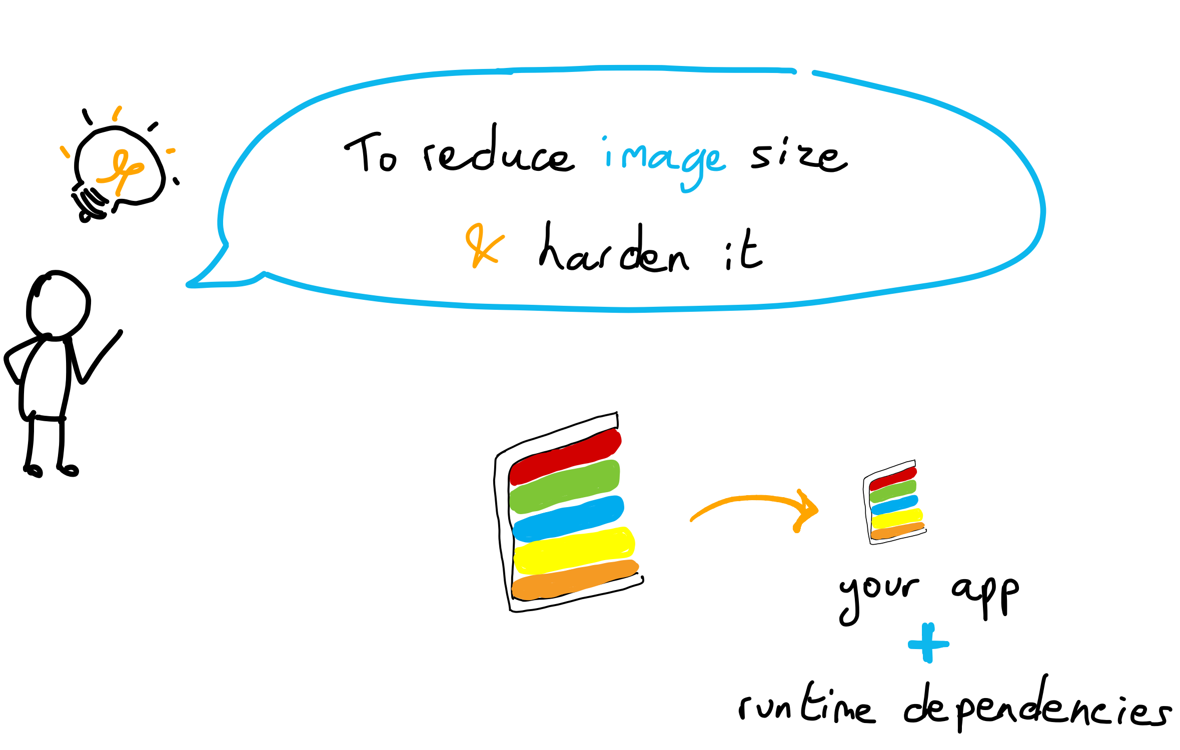
```
$ docker build --squash -t my-image .
```



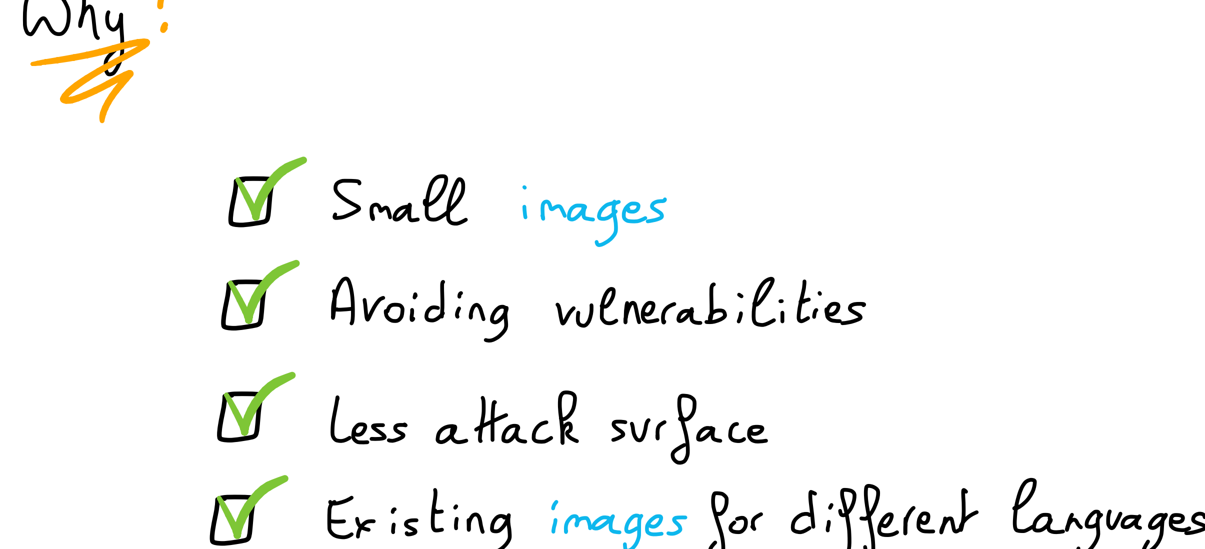
Allows to merge all "filesystem layers" in only one

--squash is only supported on a Docker daemon with experimental features

Use Distroless base image




```
FROM gcr.io/distroless/nodejs18-debian11
```



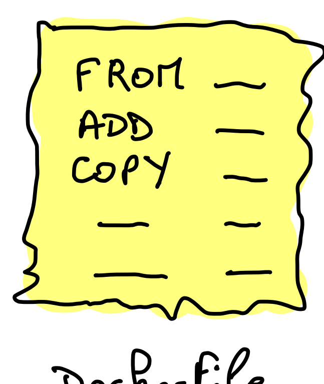
Distroless do not contain a shell!

So, CMD & ENTRYPOINT must be specified in vector form:

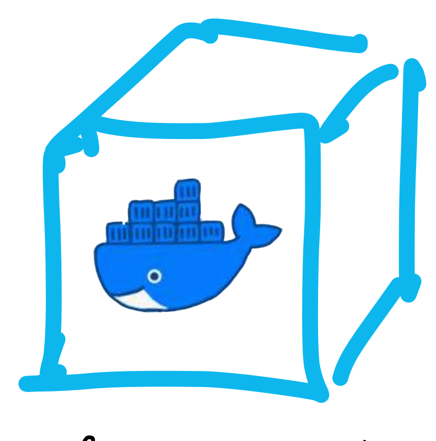
```
FROM gcr.io/distroless/nodejs18-debian11
...
CMD ["/app"]
```

> Multi-stages build

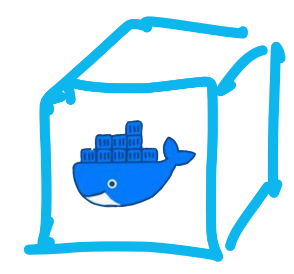
→ Only one DockerFile in where we define Build & Runtime environments



Dockerfile

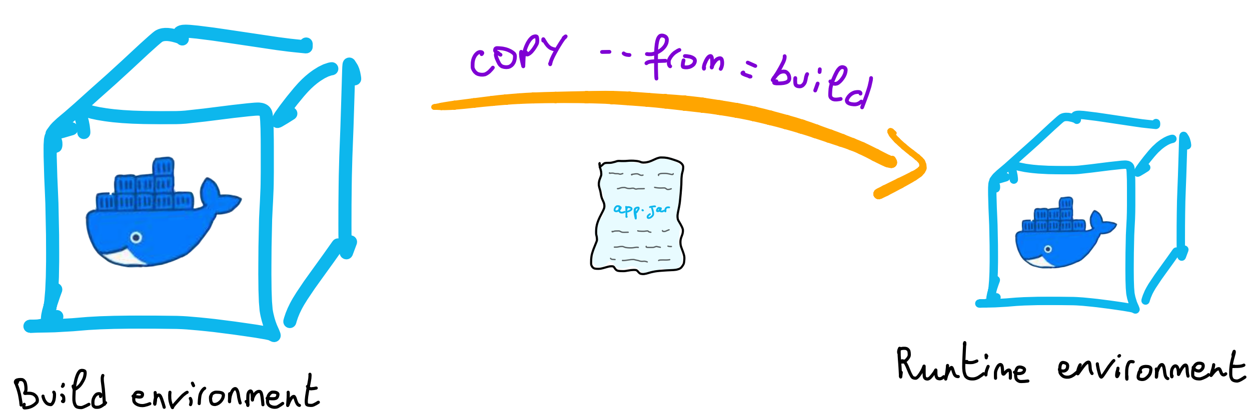


Build environment

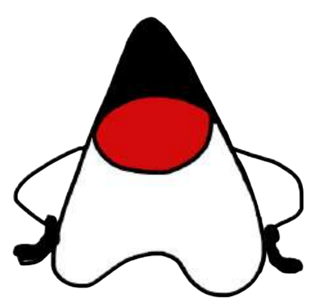


Runtime environment

→ Allows to copy artifacts from one stage to another



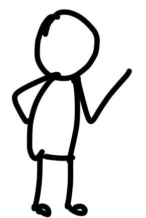
name of the stage



```
# Build env
FROM maven:3-jdk-11 AS build
COPY src /app/src
COPY pom.xml /app
RUN mvn -f /app/pom.xml clean package


# Runtime env
FROM gcr.io/distroless/java:11
COPY --from=build /app/app.jar /usr/app/app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/usr/app/app.jar"]
```

Images contain only apps & dependencies, so they provide a minimal attack surface.

It's possible to build
a specified stage



```
$ docker build --target build -t my-image:tag .
```

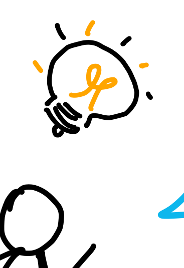
Ok, but why?

Debugging a stage

Using a debug stage with tools ...

Using a test stage with fake data ...

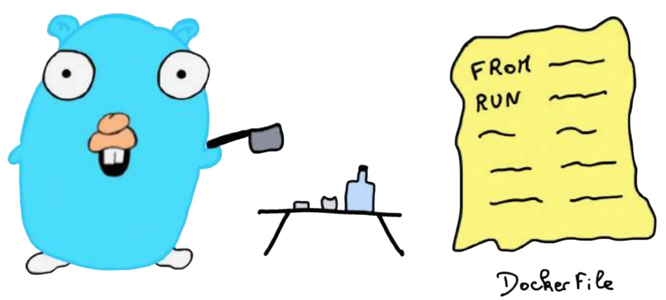
...



An artifact can be copied from an external image

```
FROM my-image:tag
...
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```

→ With BuildKit, only needed stages are built




```
FROM my-image:tag AS base
RUN echo "hello from base"

FROM base AS stage1
RUN echo "hello from stage1"

FROM base AS stage2
RUN echo "hello from stage2"
```

Let's build only base & stage2

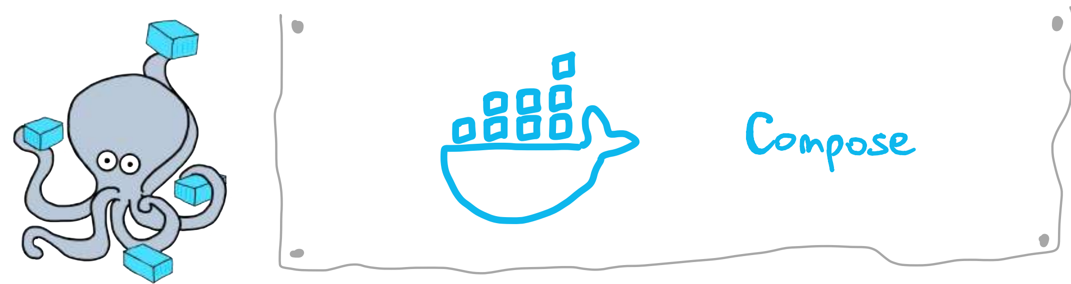
```
$ DOCKER_BUILDKIT=1 docker build --no-cache --target stage2 .
```

FROM scratch

→ To build base images (debian, busybox ...)

→ or minimal images (only one binary)

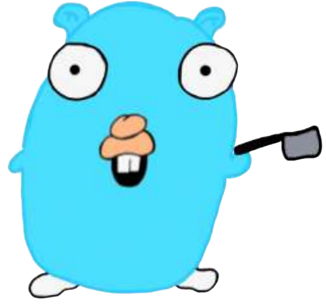
```
FROM scratch
COPY hello /
CMD ["/hello"]
```



→ Handle several containers at once

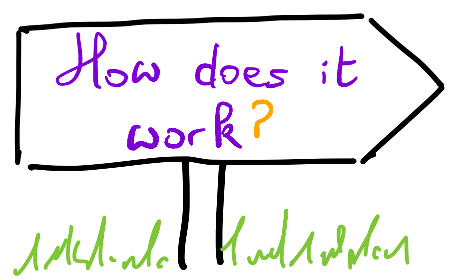
→ A container's configuration is called a service

→ Services are defined in a docker-compose.yml file



docker-compose.yml

Almost every line in a docker-compose.yml file corresponds to Docker command.



// build my own image

$ docker build -t cowsay myapp/

```
services:
  my-app:
    build:
      context: ./myapp
      dockerfile: Dockerfile
    image: cowsay:latest
    container_name: my-app
    ports:
      - 8080:5000
  db:
    image: my-db:1.0.0
    ...
```

Docker will create 2 containers

// expose port 5000 of the container on port 8080 in the host

// pull existing image

$ docker pull my-db:1.0.0

my-app

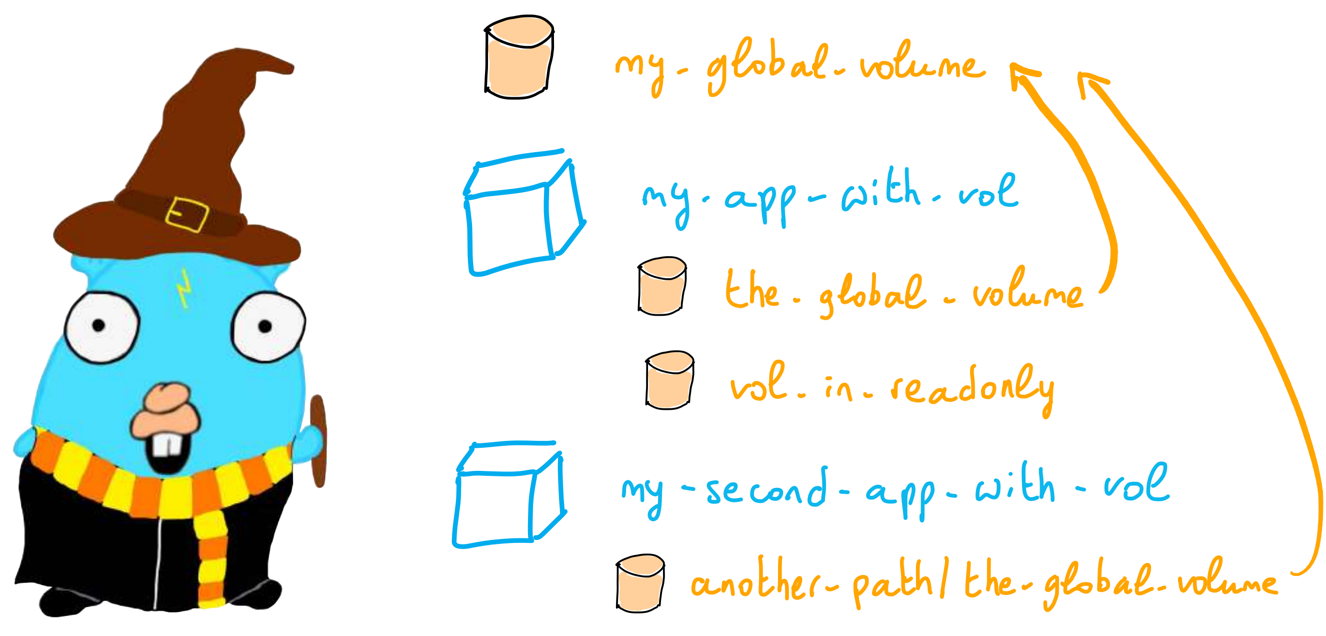
db

→ Volumes can be local
(accessible in one container)
& global
(accessible by all containers)

```
services:
  myapp-with-vol:
    image: my-image:tag
    volumes:
      - my-global-volume:/the-global-volume
      - /my-folder:/vol-in-readonly:ro
  my-second-app-with-vol:
    image: alpine:latest
    volumes:
      - my-global-volume:/another-path/the-global-volume
volumes:
  my-global-volume:
```

read only volume only for my-app-with-vol

global volume accessible by all containers/services



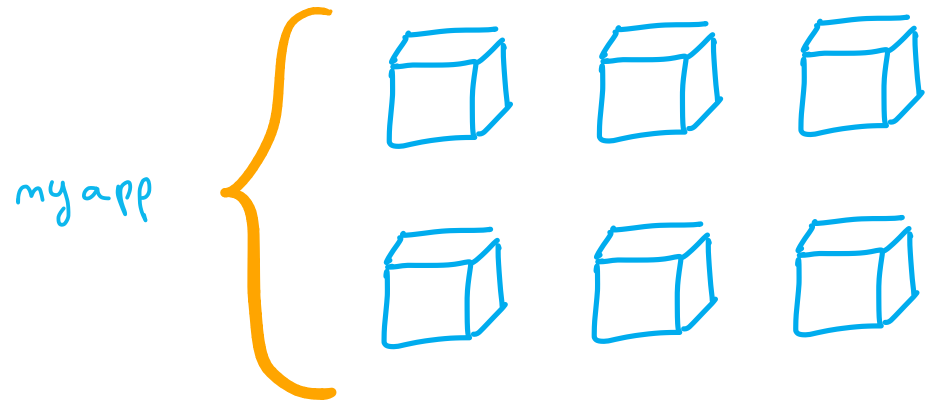
→ It's possible to ask Docker to start/handle a service after another one

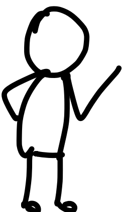
```
services:
  myapp:
    image: my-image:tag
    depends-on:
      - db
  db:
    image: my-db:latest
```

(2) myapp, (1) db

→ Insted of having 1 service = 1 container, you can define the number of containers per service

```
services:
  myapp:
    image: my-image:tag
    deploy:
      mode: replicated
      replicas: 6
```





A lot of other features exists on Docker Compose

How To ?

- Start the services

```
$ docker compose up
```

- Start the services in background

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

- Display running services

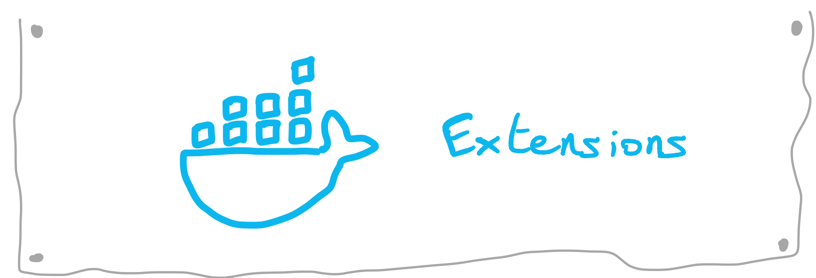
```
$ docker compose ps
```

- Stop the services

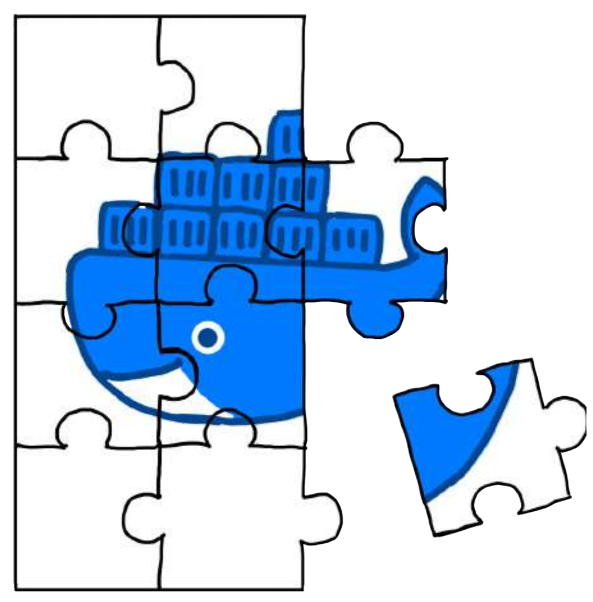
```
$ docker compose stop
```

- Remove the services & created volumes

```
$ docker compose down --volumes
```

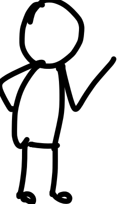


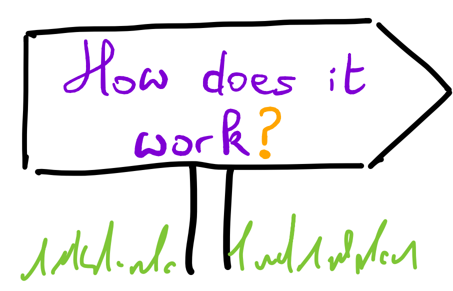
→ Extends Docker Desktop



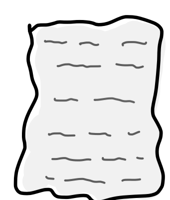
→ Extensions are accessible in the Marketplace

You can install your own extensions ☺





→ An extension is composed of several required files and folders :



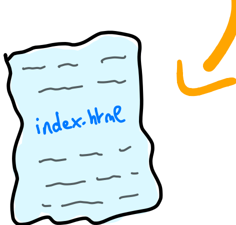
metadata.json

information about the extension :

Dockerfile

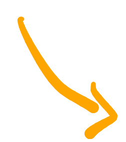
everything you need to build the extension

ui

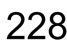


UI / front of the extension

xxx.svg



an icon



How To ?

Initialize a new extension

```
$ docker extension init my-extension
```

Build your extension

```
$ docker build -t myuser/my-extension:1.0.0
```

myuser → your DockerHub user

Install your extension

```
$ docker extension install myuser/my-extension:1.0.0
```

Update the extension

```
$ docker extension update myuser/my-extension:1.0.0
```

List installed extensions

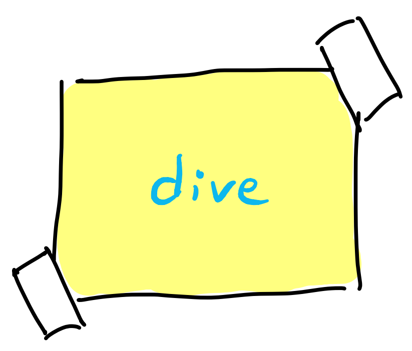
```
$ docker extension ls
```

Remove an extension

```
$ docker extension rm myuser/my-extension:1.0.0
```

Validate your metadata.json file

```
$ docker extension validate metadata.json
```



"Examine filesystem changes between different layers of your images"

https://github.com/wagoodman/dive

Usage

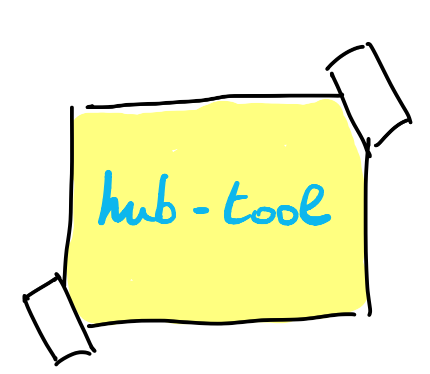
```
$ dive gcr.io/distroless/java
```

or :

```
$ dive build -t my-image .
```

In CI/CD chain

```
CI=true dive my-image
```



with Desktop 3.0

"Interact with Docker Hub through a CLI"

Display user information & user limits

```
$ hub-tool account info
```

Display user rate limiting

```
$ hub-tool account rate-limiting
```

List my repositories

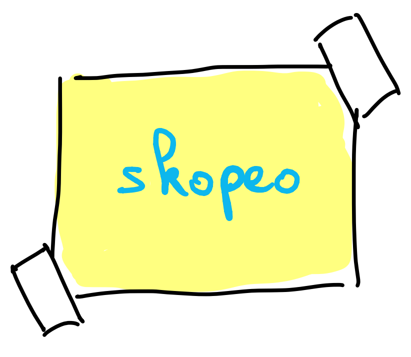
```
$ hub-tool repo ls
```

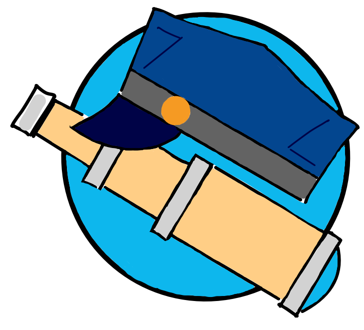
Show tags for repository

```
$ hub-tool tag ls scraly/simpleapp –format=json
```

Remove my repository scraly/toto

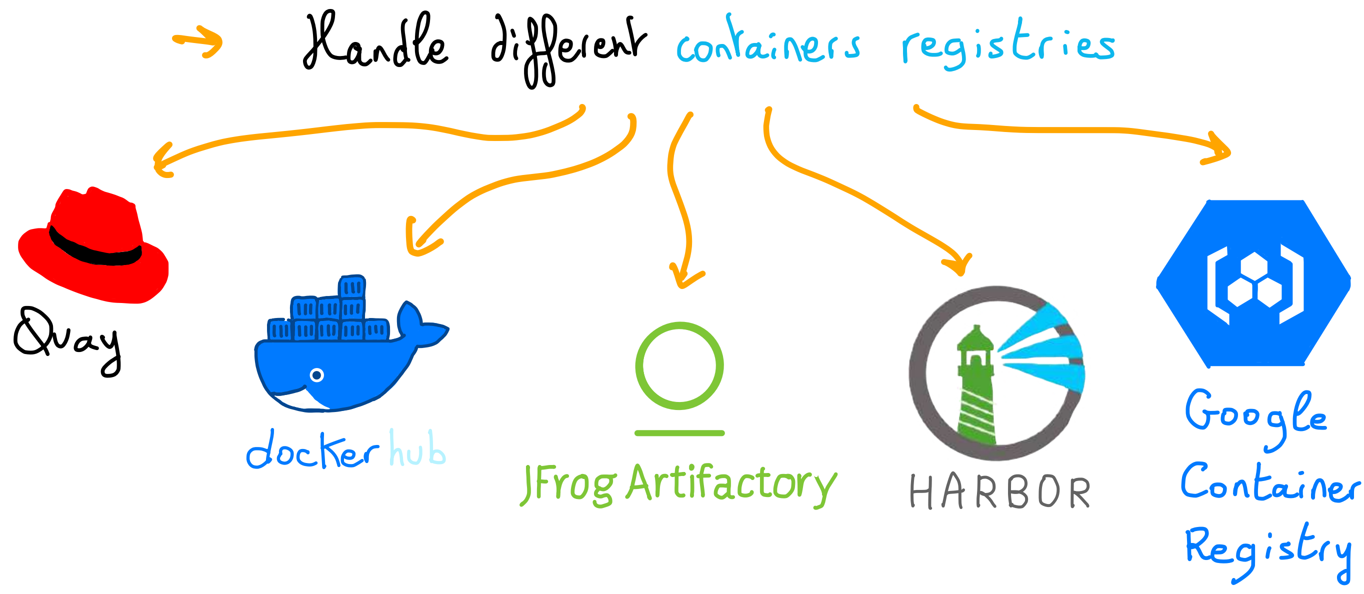
```
$ hub-tool repo rm scraly/to to –force
```





" Handle many operations
to containers images "

https://github.com/containers/skopeo



→ Don't run as root user ☺

Login

```
$ skopeo login my-registry.com -u my-user
```

Copy an image from DockerHub to my private registry

```
$ skopeo copy docker://dockersamples/k8s-wordsmith-web
docker://my-registry.com/dockersamples/k8s-wordsmith-
web
```

Delete an image from a remote registry

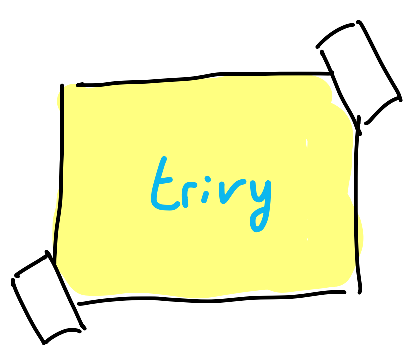
```
$ skopeo delete docker://my-registry.com/my-image:my-tag
```

Inspect a remote image (no need to pull it!)

```
$ skopeo inspect docker://my-registry.com/my-image:my-tag
```

Synchronize images between a registry and a folder

```
$ skopeo sync --src docker --dest dir
my-registry.com/busybox /my/folder
```



" Security scanner for containers, clusters ... "

https://github.com/aquasecurity/trivy

→ Vulnerabilities detection :

- ☑ of OS packages
- ☑ Application dependencies

  (npm, yarn, cargo, pipenv, composer, bundler ...)

→ Misconfiguration detection

(Kubernetes, Docker, Terraform ...)

→ Secret detection

→ Simple and fast scanner

→ Easy integration for CI

→ A Kubernetes operator

Scan an image

```
$ trivy image python:3.11-alpine
```

Scan and save in a JSON report

```
$ trivy image golang:1.19-alpine -f json -o results.json
```

In the same Collection :

Available on Paperback on Amazon

& on digital on Gumroad

https://www.youtube.com/AurelieVache

Who am I ?

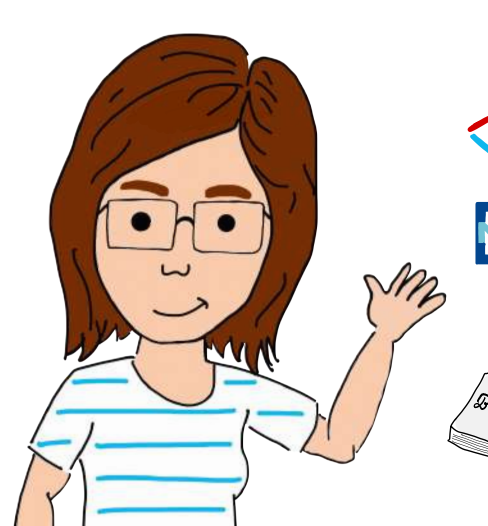
DevRel ❤ DevOps - + 17 years xp

Google Developer Expert in Cloud technologies

CNCF Ambassador - Docker Captain

Conferences organizer & Mentor

Technical writer - Speaker - Sketchnoter



Aurélie Vache

Contact me !

Abstract

Understanding Docker can be difficult or time-consuming.
I created this collection of sketchnotes about Docker
in order to try to explain the technology in a visual way.

Included :

- ☑ Docker components
- ☑ Resources
- ☑ Concretes examples
- ☑ Tips & Tools