

DOCKER

PLAN

Intro

Premiers pas

Reseaux Docker

Volumes Docker

Docker Compose

Créer une image

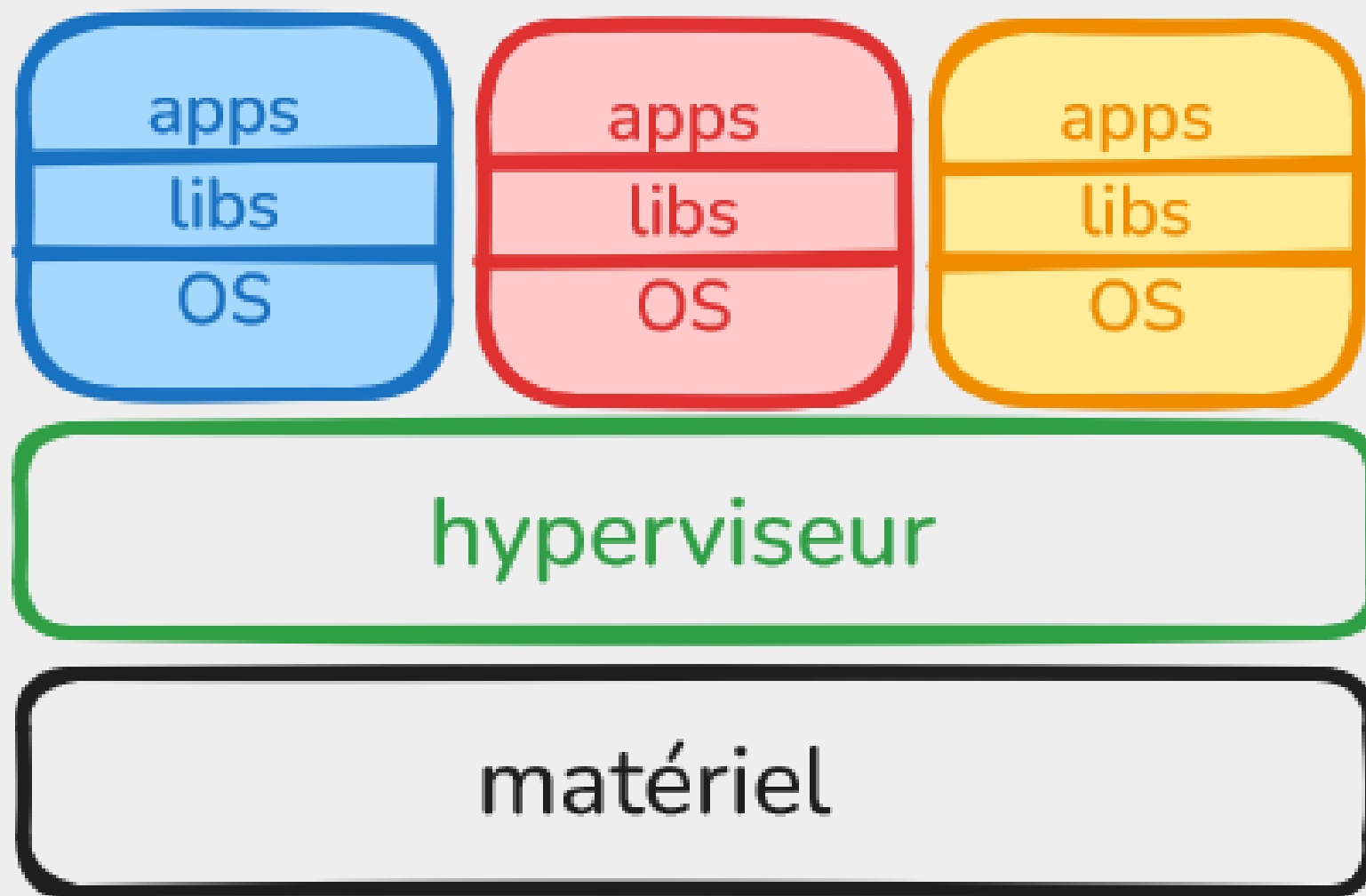
Registre d'images

Techniques avancées

Virtualisation

soit **partage** de ressources matérielles

- hyperviseur
 - Fine couche de système d'exploitations **OS** = Noyau léger
 - Alloue des **ressources physiques**
 - i.e *CPU, RAM, E/S*
 - Pour Créer des **Machines Virtuelles**
 - i.e des OS « invités » différents (unix / windows)
 - Ex: *VMWare, Xen, KVM, Hyper-V...*



Enjeux

- Optimisation des ressources matérielles
 - Gain énergétique: réduction des coûts
- Flexibilité
 - Déplacement, Sauvegarde / Restauration des Vms
- Incident / Sinistre : **PRA** « Plan de Relance de l'Activité »
 - **RPO** « Recovery Point Objective » Δ_{\max} . save / restore
 - **RTO** « Recovery Time Objective » Δ_{\max} interruption

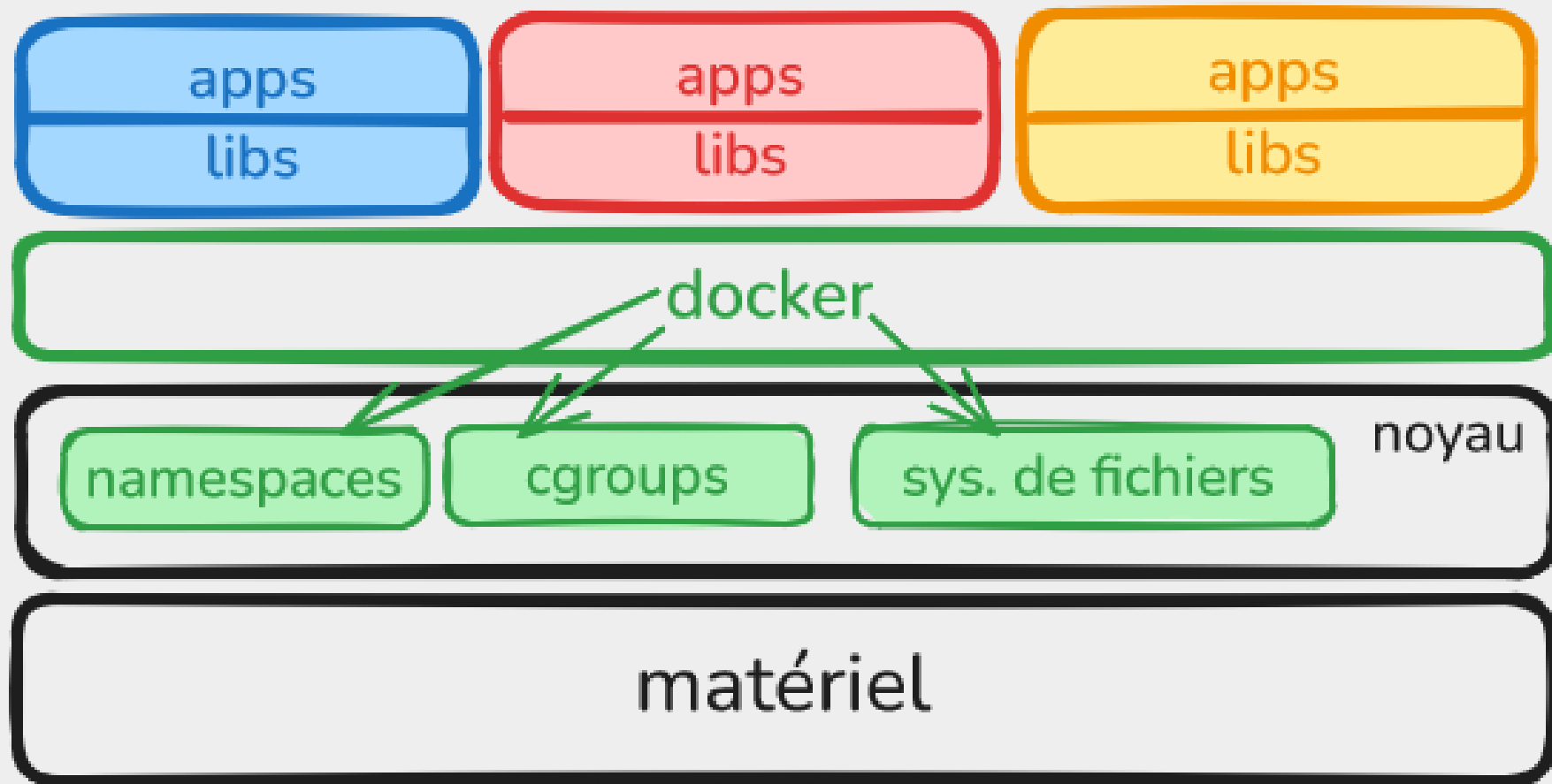
Performances

- Dégradées car la couche logicielle supplémentaire (hyperviseur)
- Moindres dégradations grâce aux technos comme VT-x ou AMD-V
 - accès direct aux drivers E/S pour les VMs « para-virtualisation »
 - architectures processeur facilitant le traitement d'instructions d'une vm étiquetées par l'hyperviseur

Conteneurisation

soit **isolation** de processus par le **noyau**

- logiciel de conteneurisation (comme Docker)
 - Créé des conteneurs
 - i.e, Isole des **processus** du reste de l'OS hôte
 - Avec leur propre « vision » du système => **namespaces**
 - Limités dans les ressources utilisables => **cgroups**
 - dont l'environnement sont assis sur des ensembles de *systemes de fichiers*, des **images**, contenant des librairies et des binaires

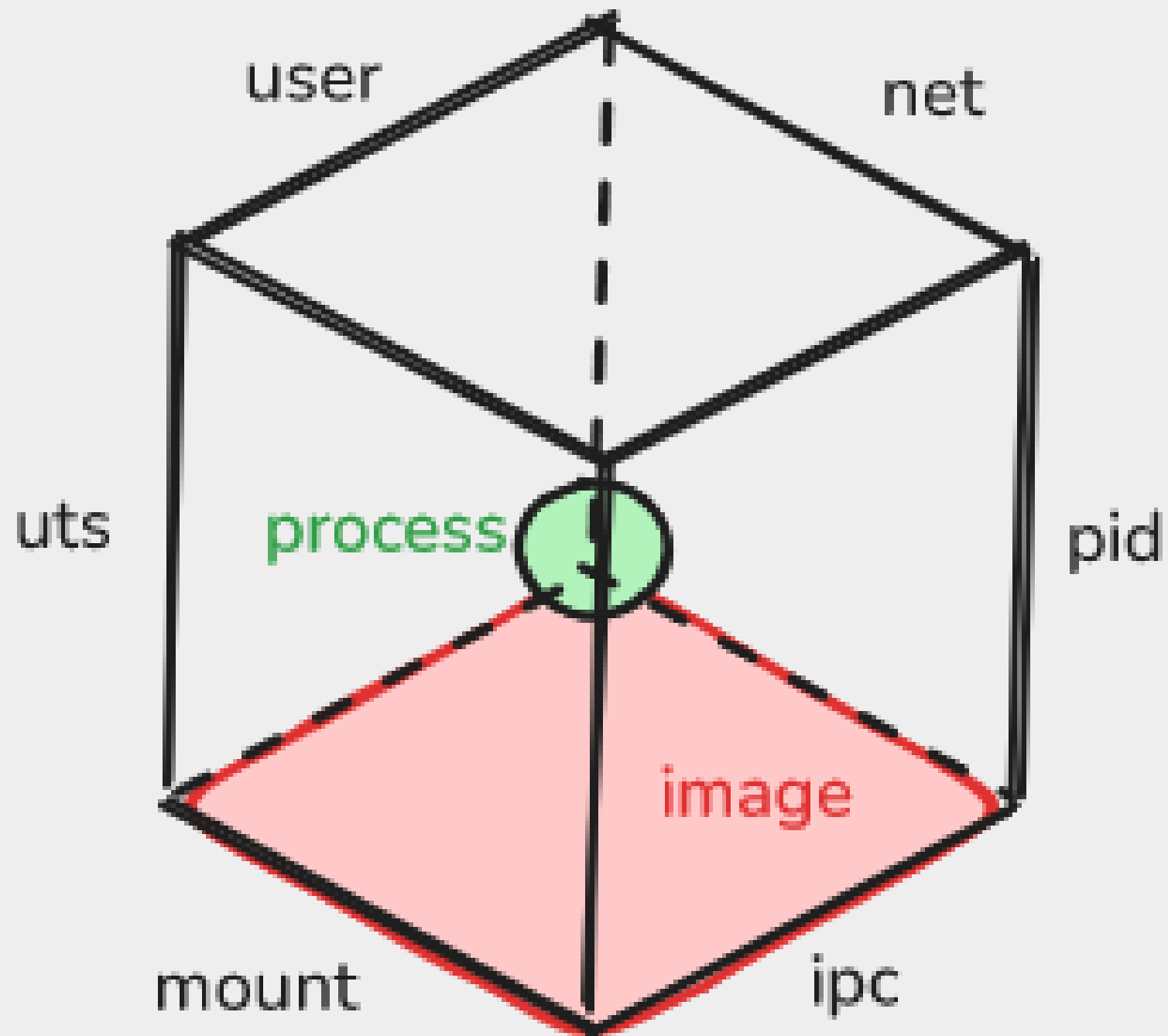


virtualisation vs conteneurs

- Les conteneurs:
 - Plus **légers**, ne comprennent que des logiciels de haut niveau
 - Plus **rapides** à créer et déplacer
- Les Vms
 - mieux **isolés** de l'extérieur
alors que le noyau de l'hôte est partagé entre conteneurs
 - Plus **dynamiques**: conçues pour les interactions utilisateur
alors qu'un conteneur n'accède pas à la *GUI / systemd / ...*

namespaces linux

- **pid:** isolation des *Pids, PPids* dans une autre table de processus
- **net:** isolation réseau *interfaces, addr. IP, routes, firewall* dans une autre pile réseau
- **mount:** isolation des *points de montage* => volumes docker
- **uts:** isolation du nom d'hôte
- **ipc:** « inter processus communication » pas de cnx avec les *sémaphores, segments de mémoire partagés, mutexes*
- **user:** mapping des UID/GID entre l'hôte et les conteneurs host
uid xxx → ctn uid 0 (par défaut)

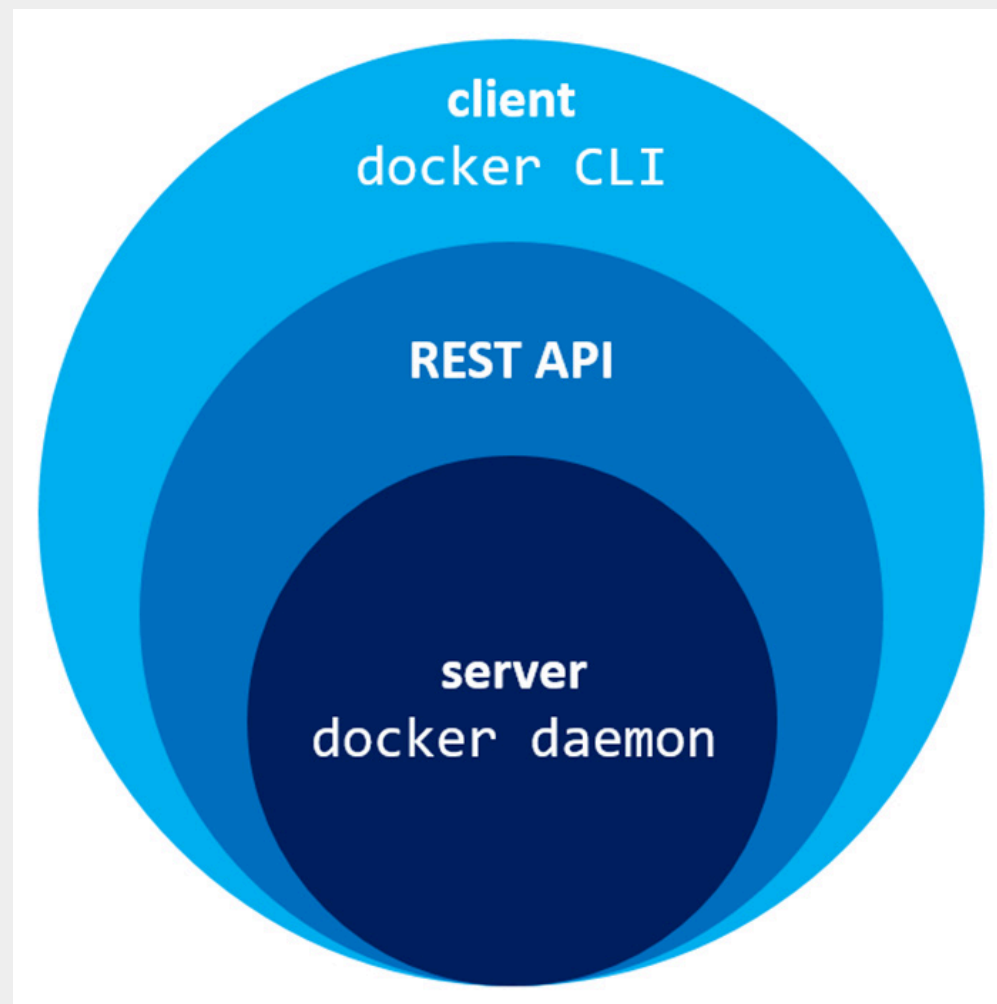


cgroups linux

- régulent l'accès aux ressources des processus:
- **RAM**
 - limites physiques et réservations
- **CPU**
 - proportion du pool CPU
 - répartition des traitements sur les coeurs
- **Entrées / Sorties :**
 - débits en lecture / écriture

docker: client/serveur

- connexion avec une socket
- même machine => **socket unix**
 - *unix:///run/docker.sock*
- machine distante => **socket tcp**
 - *tcp://host:2375*
- TLS (sécurisé): *tcp://host:2376*



composants de docker

- client: **CLI** + plugins
 - *Compose / Buildx / Swarm*
- **registre d'images** public ou privé
- serveur d'API REST: **docker daemon**
 - responsable de la construction des images *build*
- moteur d'exécution haut-niveau: **containerd**
 - responsable des aspects *stockage et réseau*
- moteur d'exécution bas-niveau: **runc**
 - fabrique un *conteneur* à partir d'une *image et de namespaces*

tcp://host:2375 ou
unix:///run/docker.sock

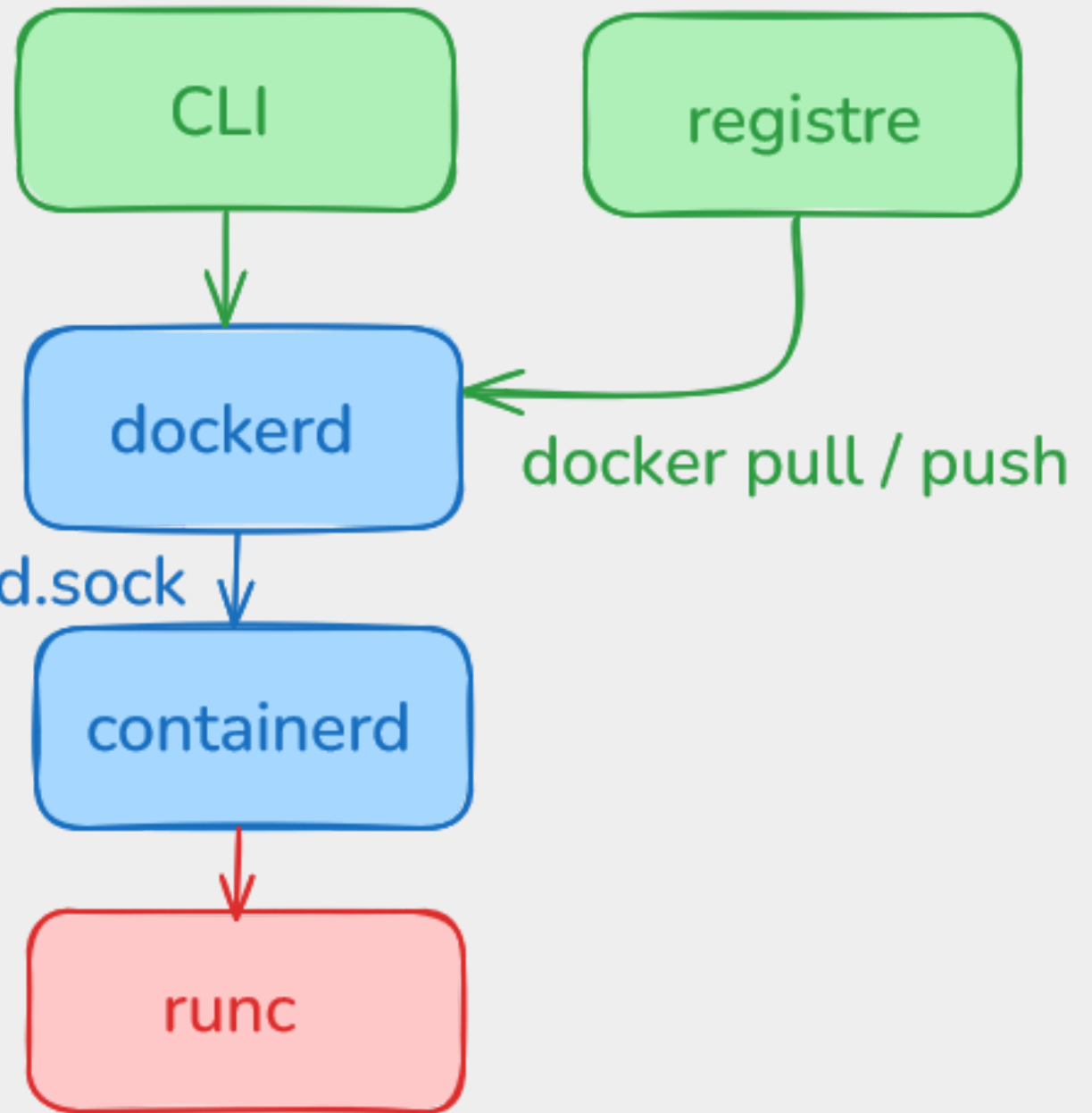
unix:///run/containerd/containerd.sock

appels système

unshare(): crée les namespaces

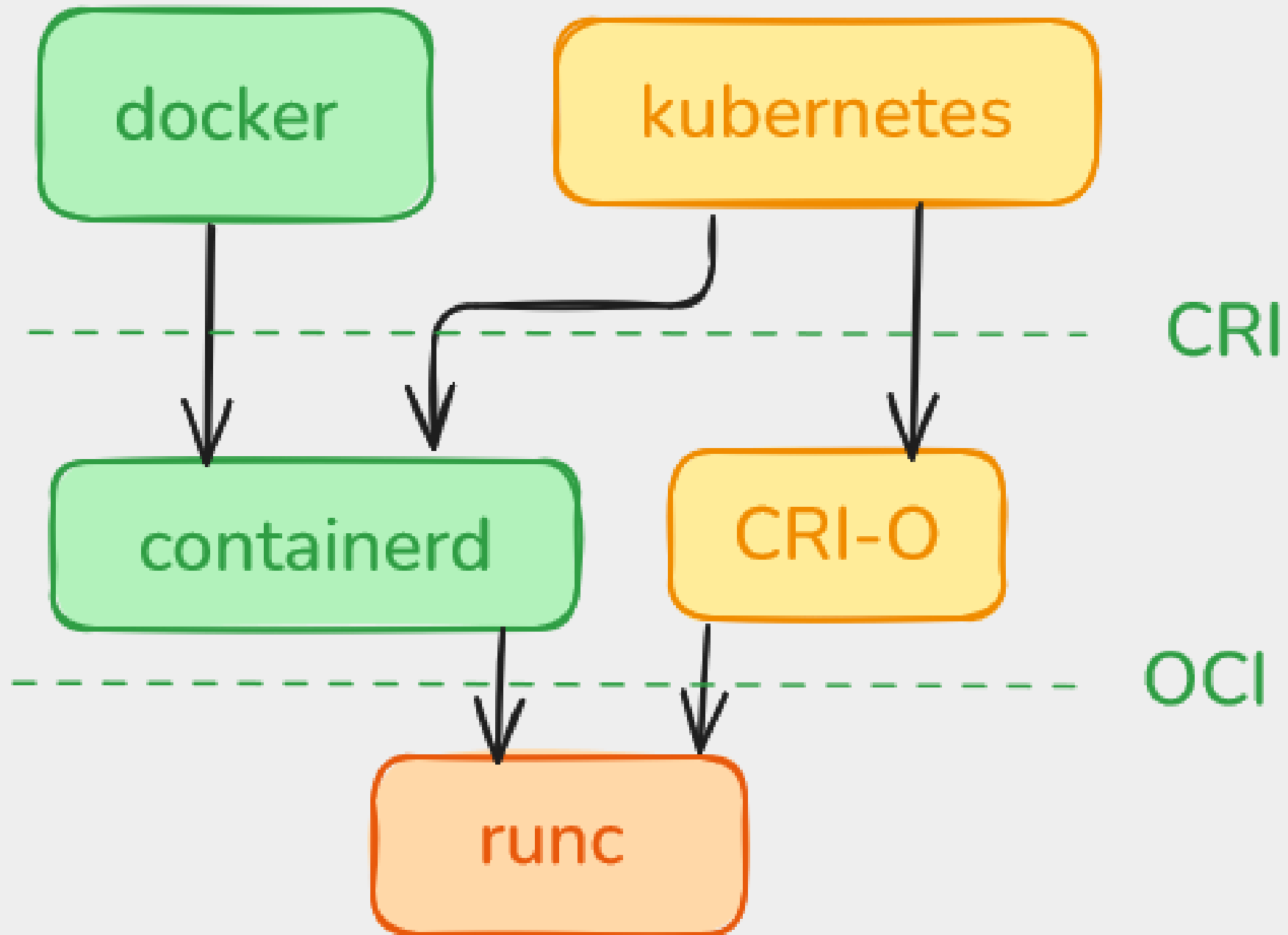
pivot_root(): installe l'image

clone(): lance un process



standards

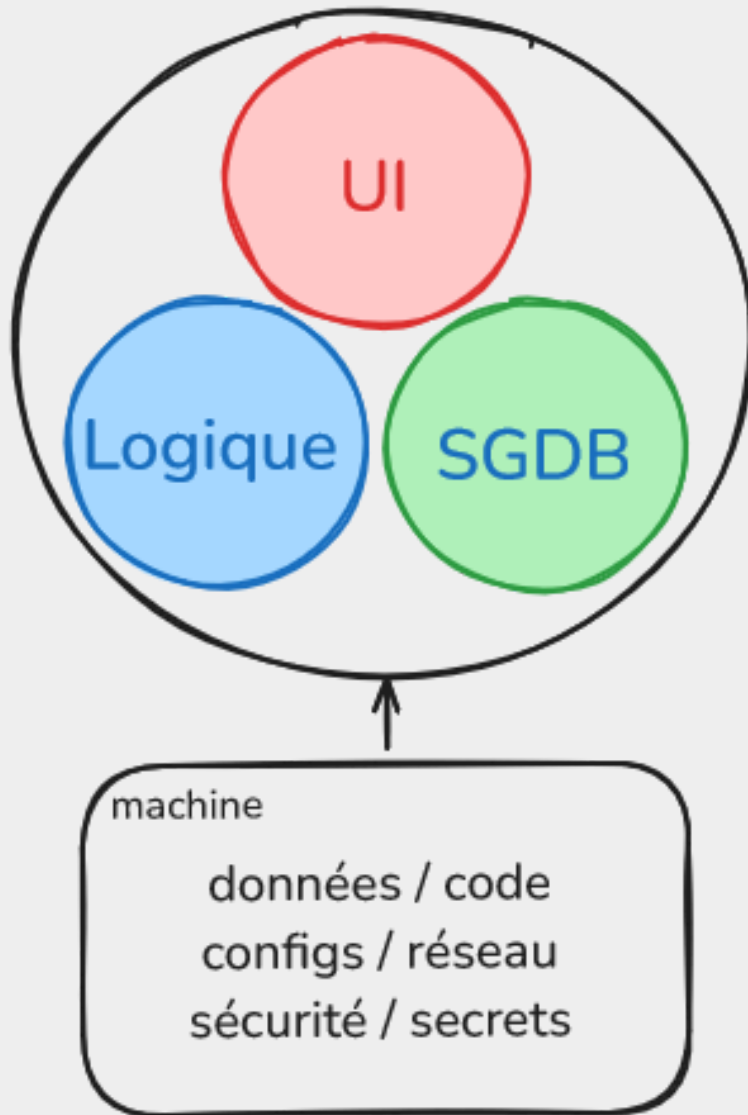
- **containerd** est conforme au standard **CRI**
 - *Container Runtime Interface*
- **runc** est conforme au standard **OCI**
 - *Open container Initiative*
- ces 2 standards rendent Docker **compatible** avec les autres solutions technologiques de conteneurs
 - *Kubernetes / Openshift / Rancher / ...*



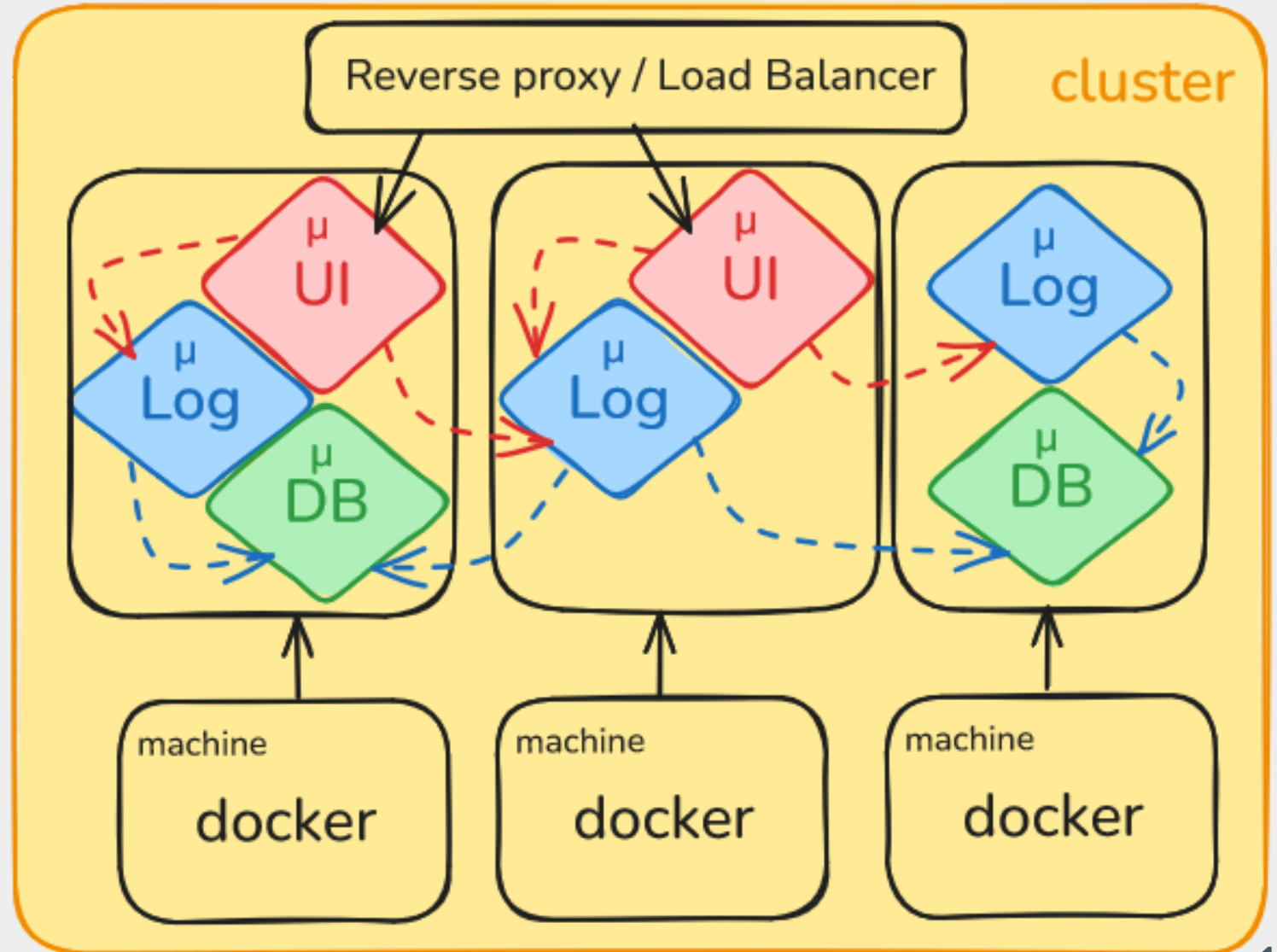
architecture microservices

- versus *monolithique*
 - **décomposition** d'une application en tiers ou **(micro)services**
 - *UI* (serveur web) / *logique* (PHP, java) / *données* (SGDB, fichiers, objets...)
 - *sécurité, surveillance et autres middlewares...*
 - services *mis en réseau* dans une infrastructure en *cluster ou cloud*
 - **connectés par des API** web => *couplage faible*
- “ docker permet facilement d'installer et isoler les microservices ”

Monolithique



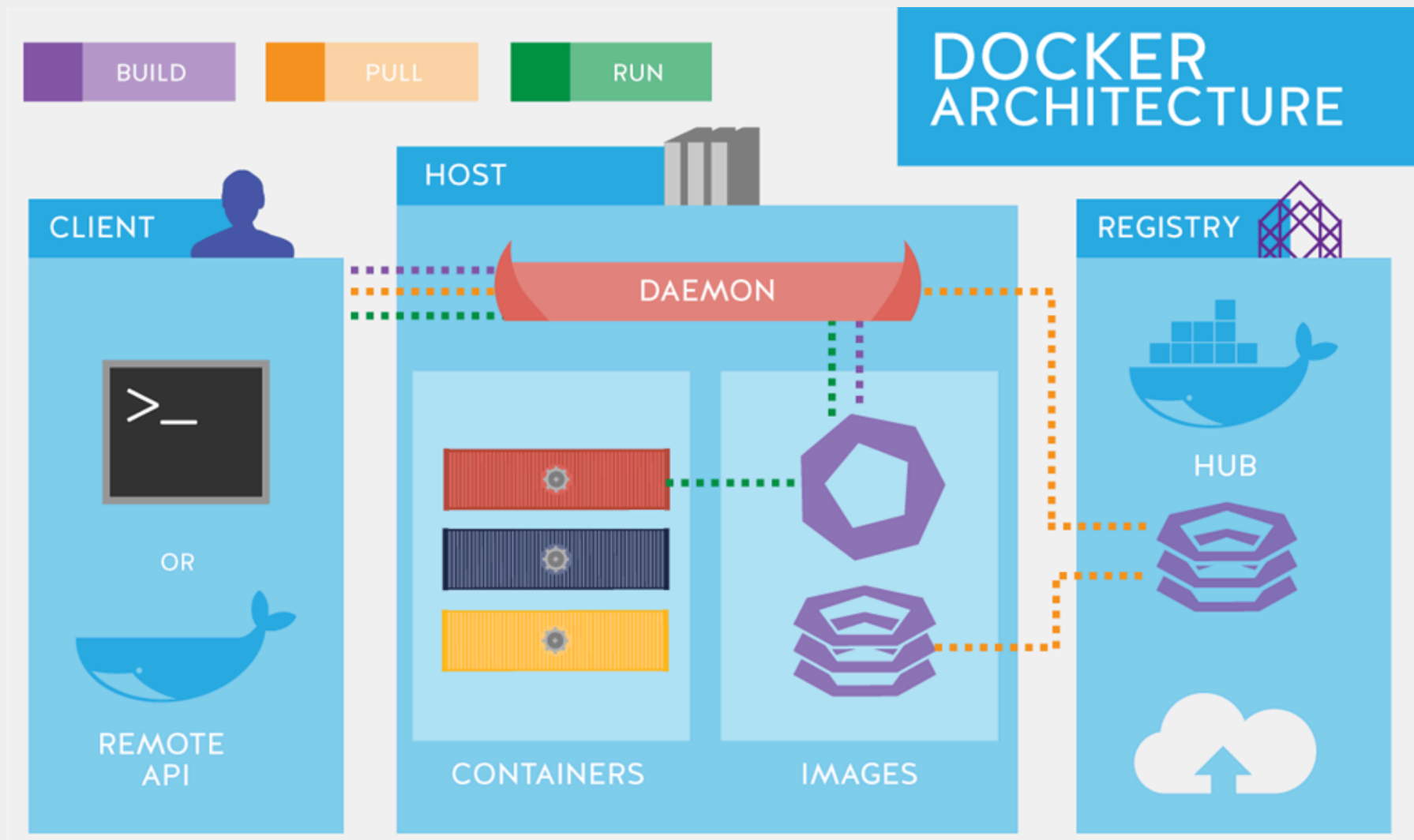
Microservices



intérêts et problématiques

- Un conteneur docker est créé / supprimé / mis à jour facilement et isolé du reste de la machine - **couplage faible**
- Un conteneur docker est déplacé / *répliqué* facilement dans un cluster pour assurer une *mise en échelle horizontale* - **scalabilité**
 - augmente les *performances* / *disponibilité* / *fiabilité* (MTRS)
- L'architecture microservices *génère bcp de flux* / **complexité**
 - *ordonnancer* / *mettre à jour* / *superviser* des services répliqués
- On doit s'adjoindre un outil d'**orchestration** *Swarm / Kubernetes...*

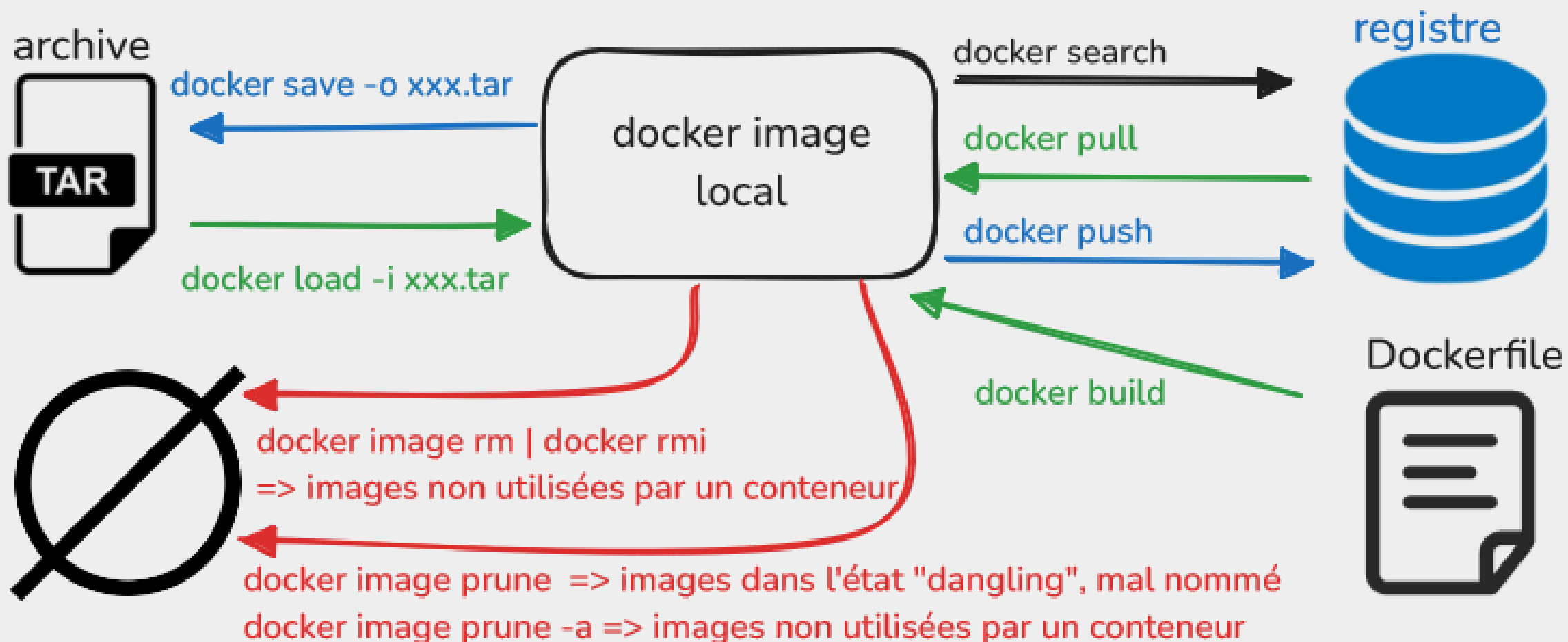
Premiers pas



télécharger des images

- par défaut on trouve les images sur le portail public **Docker Hub**
- infos: `docker search <image>` mais *interface pauvre*
- composition d'un image: `nom:tag[@sha256:xxxxxxxxxx]`
 - **nom:** lié à la technologie souhaitée (ex: nginx)
 - **tag:** *obligatoire* lié à la version et/ou la distro
 - **digest:** *optionnel* sha256 lié à la plateforme (par défaut amd64)
- `docker pull nom:tag[@sha256:xxxxxxxxxx]`
- voir les images téléchargées en *local*:
`docker image ls` ou `docker images`

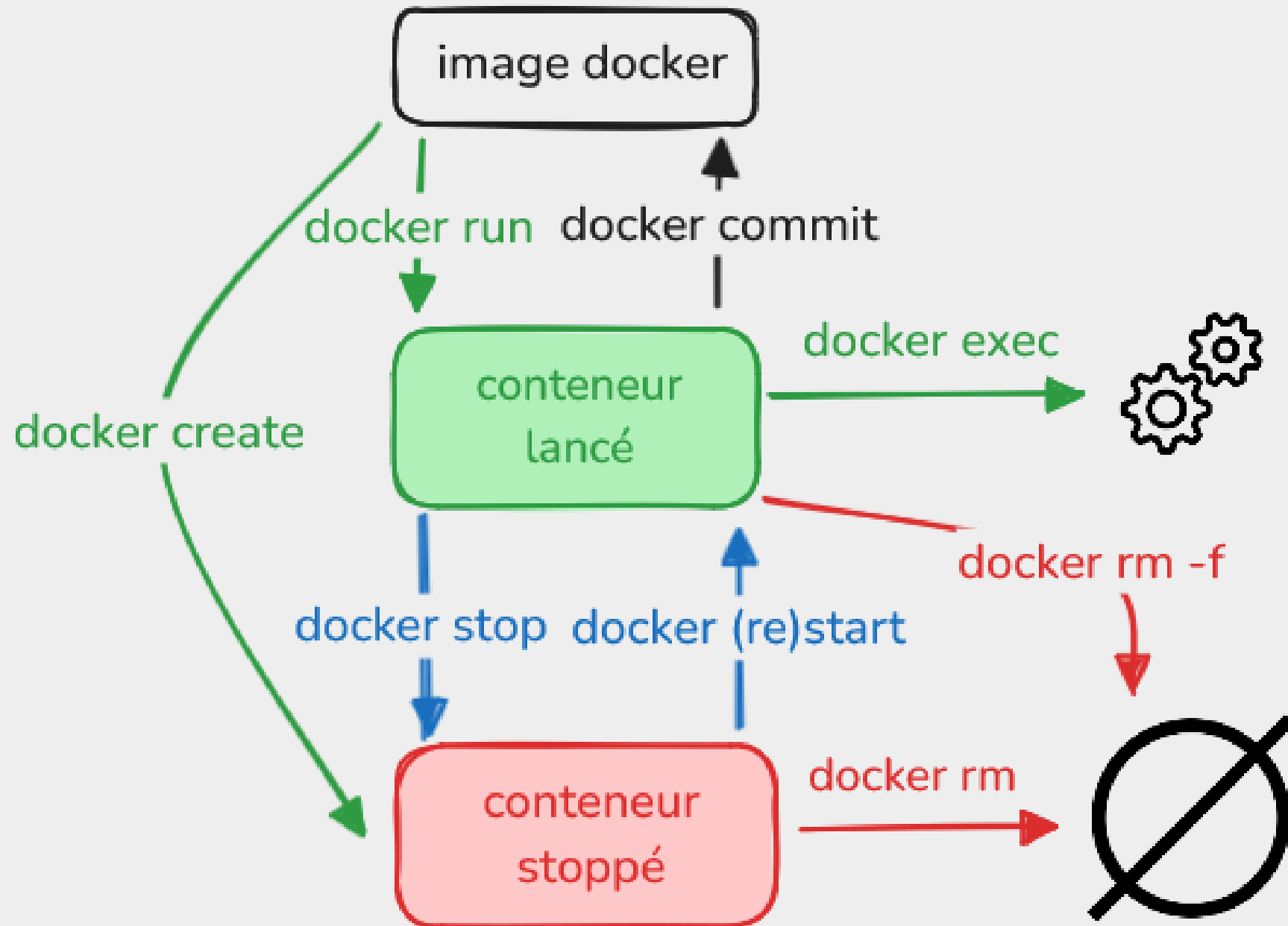
cycle des états des images



lancer un conteneur

- créé un conteneur: `docker run <image:tag>`
 - *"pull" l'image* si elle n'existe pas en local
 - **isole l'image** sous jacente
 - et **lance une commande** liée à l'image
- par défaut, le processus lancé est lié au terminal, *en avant-plan*
- REMARQUES
 - le terminal est bloqué si le processus ne retourne pas ! *daemon*
 - si le processus est un *terminal*, celui-ci s'arrête immédiatement !

cycle des états des conteneurs



lancer un conteneur de type daemon

- `docker run --name <name> -d --restart unless-stopped <image:tag>`
 - **--name <name>**: fixer le nom d'un conteneur
 - **-d**: lancement du processus en *arrière-plan*
 - **--restart [unless-stopped | always]**: politique de redémarrage du conteneur en cas de crash
 - on voit l'**identifiant** du conteneur en sortie *hash sha256*
- `docker stop <name | ID>`: stoppe le conteneur => le processus
- `docker [re]start <name | ID>`: (re) démarre le conteneur => le processus

introspection: voir les conteneurs

- `docker ps`: voir les conteneurs lancés
- `docker ps -a`: voir tous les conteneurs - *lancés et stoppés*

CTN ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e9b2f8b	alpine	<code>"/bin/sh"</code>	8s ago	Exited(0) 7s ago		shy_gauss
aa8f8b8	nginx	<code>"xxxxxxx"</code>	About 1mn ago	Up About 1mn	80/tcp	my-nginx

- `docker ps -aq`: affiche seulement les identifiants *e9b2f8b aa8f8b8*
- `docker ps --filter "name=my-"`: filtrer selon les nom et les valeurs des champs

introspection: configuration du conteneur

- `docker inspect <name | ID>` : affiche la configuration du conteneur
 - sous forme d'*objet JSON*

```
# afficher un ou des champs avec les TEMPLATES GO
docker inspect --format "{{.Id}} {{.State.Status}}"
# afficher un objet ou une liste en json
// --format "{{json .State}}"
# // avec des champs séparés
// --format "{{join .State ' ' , ' ' }}"
# similaire à docker
(echo "ID NAME STATUS"; docker inspect $(docker ps -q) \
--format '{{printf "%.10s" .Id}} {{.Name}} \
{{upper .State.Status}}') | column -t -s ' '
```

introspection: observer les logs d'un conteneur

- `docker logs [-f] <name | ID>`
 - affiche les logs du processus du `run`
 - **-f**: affichage en temps réel - *bloquant*
- les logs sont gérés par docker
 - *par défaut* en tant que fichier json *ICI*
 - `docker inspect --format='{{.LogPath}}' <name | ID>`
- d'autres techniques - *drivers* de logging sont disponibles
 - `docker info`

modifier le processus au lancement

- `docker run [options] <image:tag> <COMMAND>`
- le paramètre *à droite de l'image* remplace la commande par défaut

CTN ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e9b2f8b	alpine	"COMMAND"	8s ago	Exited(0) 7s ago		shy_gauss

- si la commande est à usage unique *one shot*
 - **--rm:** *supprime le conteneur* après la fin du processus, soit l'arrêt du conteneur

configurer le processus

- `docker run -e VARIABLE=value -e VARIABLE2=value2 ...`
 - créer / éditer des **variables d'environnement** pour configurer le processus au moment du `run`
- `docker run --env-file ./env ...`
 - même chose en chargeant un *fichier de variables* `.env` en local

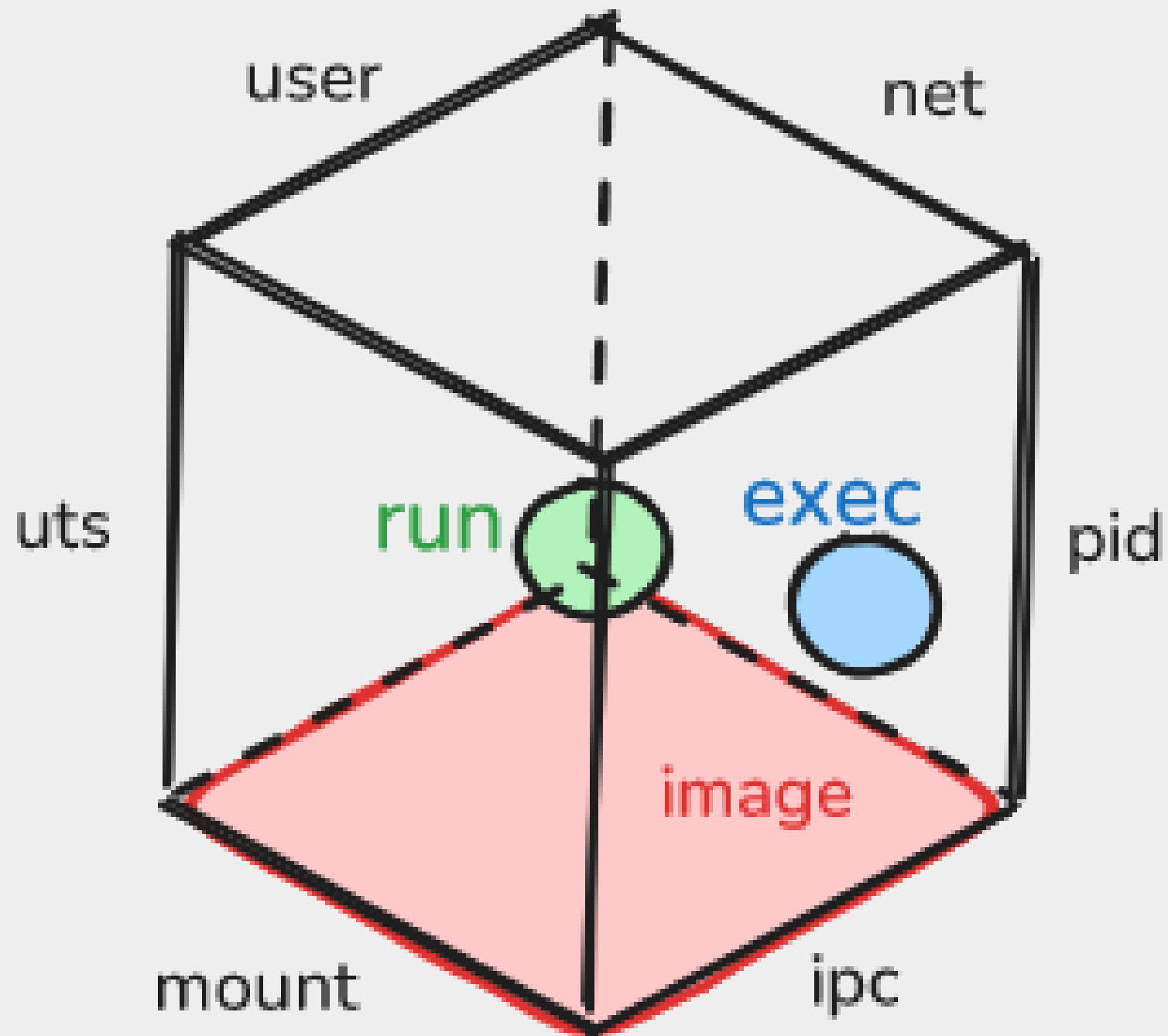
```
MY_VARIABLE1=value  
MY_VARIABLE2=value2
```

lancer un conteneur en mode interactif

- `docker run -it <image:tag> [/bin/[ba|...]sh]`
 - les 2 options `it` du `run` permettent de
 - rendre **persistent** et
 - donc d'utiliser un terminal dans le conteneur
 - pour *travailler* dans le conteneur
 - **-t** : attache un pseudo-tty
 - **-i** : attache le flux d'entrée *stdin* du conteneur
- “ *si la commande par défaut de l'image n'est pas un terminal on peut la remplacer par un shell (sh, bash, ...), si l'existe !!!* ”

exécuter un autre processus dans un conteneur

- `docker exec [-it] <ctn-name | ID> <COMMAND>`
- si un conteneur est **déjà lancé** avec `run`
 - `exec` exécute une commande supplémentaire dans le conteneur
- si l'on veut *travailler dans le conteneur* avec un terminal
 - on ajoute les options `it` avec une commande de type shell



s'attacher / se détacher du processus

- pour sortir d'un processus de type shell (interactif)
- `exit`: sort ET termine le terminal
 - Rien à dire si le shell a été lancé avec `exec`
 - MAIS *stoppe le conteneur* si le shell a été lancé avec `run`
- `Ctrl + P + Q`: détache le flux d'entrée et *laisse le shell vivant*
 - `docker attach`: peut se rattacher au shell

Reseaux Docker

- le **namespace network** créé une copie de la pile réseau de l'hôte
- on peut y attacher des:
 - *interfaces réseau virtuelles*
 - *adresses*
 - *tables de routages*
 - *règles de Firewall*
- une telle configuration réseau est établie par un **driver réseau docker**

drivers réseau docker

- **bridge:** *par défaut*, sous-réseau local virtuel permettant la communication entre conteneurs sur le hôte
- **host:** accès direct des conteneurs aux interfaces de l'hôte
- **none:** *pas de réseau*, le conteneur ne communique pas
- **overlay** : pour un *cluster swarm* - *surcouché* réseau virtuelle qui agrège les réseaux associés à plusieurs hôtes en LAN ou WAN
- `docker network ls` : afficher les réseaux docker disponibles

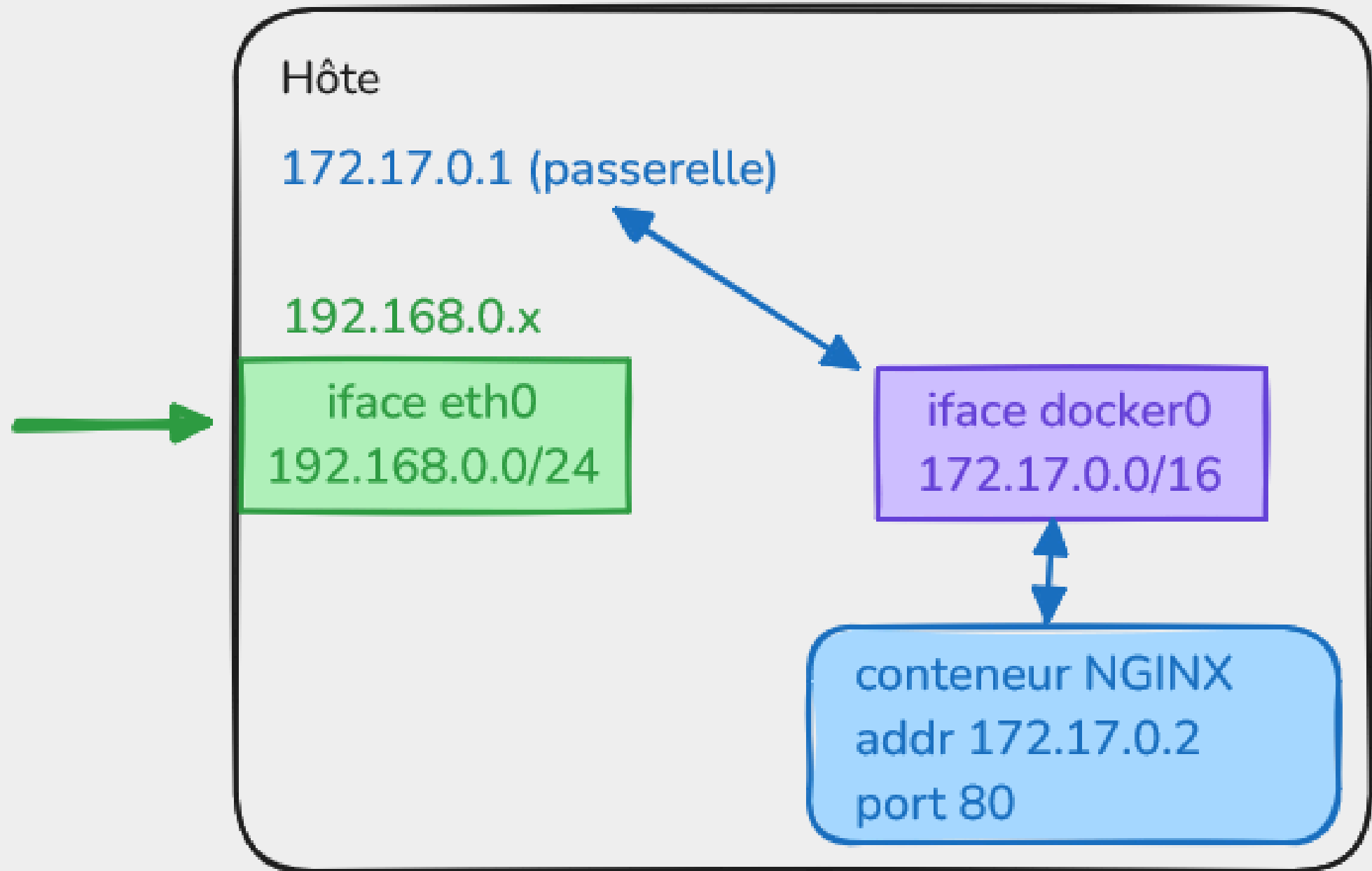
bridge par défaut et l'interface docker0

- par défaut une interface **docker0** de type *bridge* est disponible

```
$ ip a
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ... state UP group default
    link/ether 02:42:87:f5:46:1c brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
```

- par défaut, un conteneur lancé reçoit une adresse IP sur ce bridge

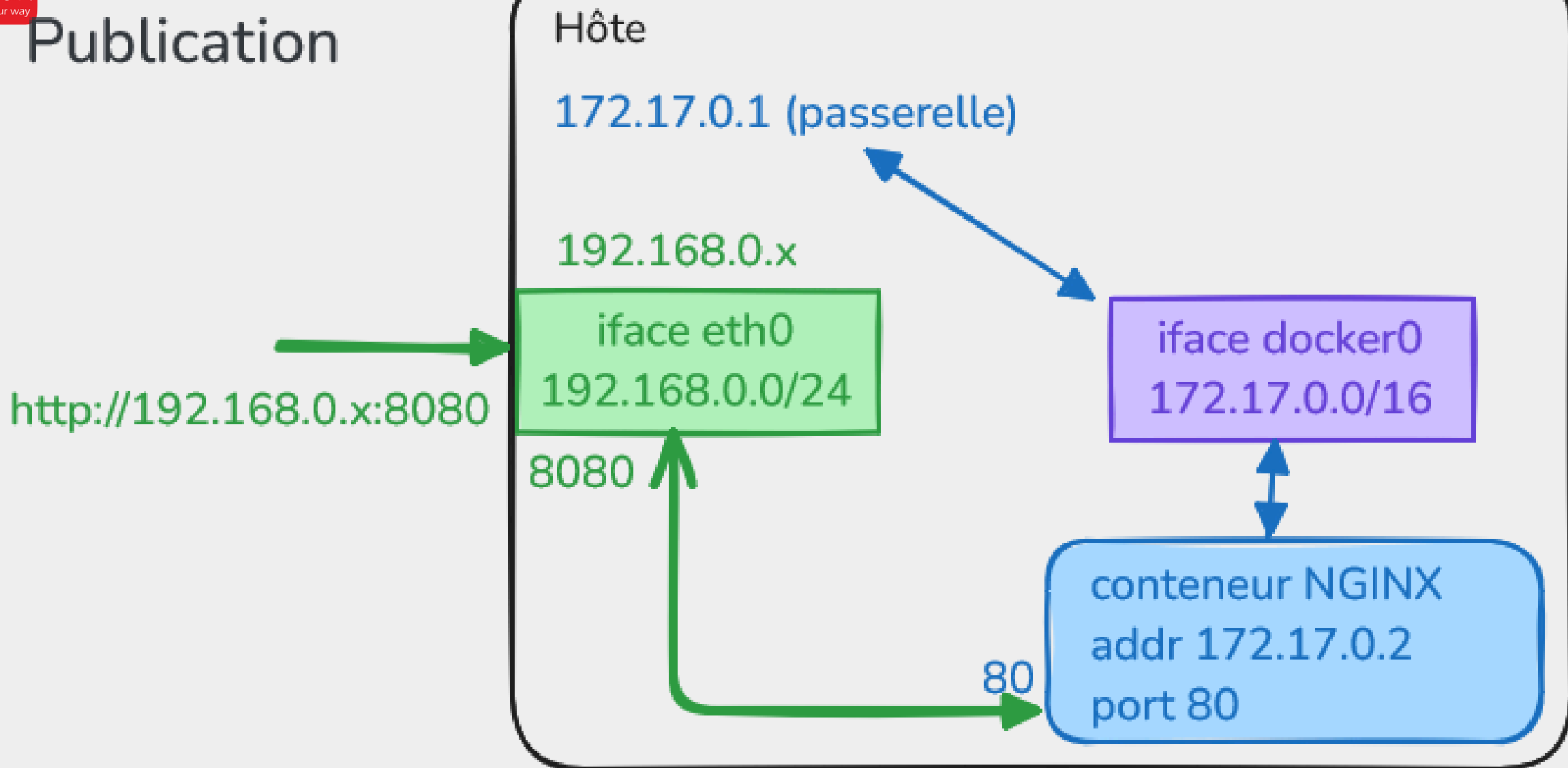
```
docker network inspect bridge
{ ..., "Containers": { "aa8f8b...": {
    "Name": "my-nginx",
    "IPv4Address": "172.17.0.2/16"}, ...}
```



exposition et publication des ports

- le champs "PORTS" du `docker ps` affiche un ou plusieurs ports réseau, dits **exposés**, liés à la techno du conteneur (*nginx => 80*)
- c'est une *simple information*, utilisable par l'extérieur
 - *cette info peut être fausse !!!*
- *par défaut* **ces ports ne sont pas accessibles** en dehors du réseau docker
- `docker run -p <EXTERN_PORT:INTERN_PORT> ...`
 - **-p:** redirige un port "interne" du conteneur sur un port "externe" sur *toutes les interfaces du hôte par défaut*
 - on appelle ce mécanisme la **publication**

Publication



`docker run -p 8080:80 ...`

types de publication

```
## redirection sur une INTERFACE PARTICULIERE  
## ADDRESS:EXTERN_PORT:INTERN_PORT  
docker run -p 192.168.0.x:8080:80 ...
```

```
## choix du port externe dans un intervalle  
## + choix de la couche transport  
docker run -p 8080-8090:80/tcp ...
```

```
## publier TOUS les ports internes EXPOSES sur des ports externes > 32768  
docker run -P (--publish-all) ...
```

“ *les publications automatiques sont utilisées par les orchestrateurs, pour répliquer des conteneurs à publier* ”

problématique du réseau docker0

- La communication entre conteneurs sur *docker0* se fait:
 - par les **ips**, peu pratique car les valeurs sont imprévisibles ips
 - `docker run --ip 172.17.x.y ...` pour une ip fixe
 - par un **dns** (alias réseau)
 - `docker run --link <ID | name | name:alias> ...` *déprécié*
 - en pratique on va associer un ensemble de conteneurs par un **réseau ad hoc de type bridge**
- “ *on utilise pas le réseau docker0 en exploitation
on créé plutôt un réseau custom par application conteneurisée* ”

créer un réseau bridge custom

```
# options avec les valeurs par défaut
docker network create \
--driver=bridge --subnet=172.18.0.0/16 --gateway=172.18.0.1 \
<network_name>
```

- `docker run --net <network_name> ...`
 - lancer un conteneur *sur ce réseau*
- `docker [dis]connect <network_name> <ID | ctn_name>`
 - *(dé)connecter* un conteneur *préexistant* au(du) réseau

“ **sur un bridge custom le nom de conteneur est un DNS !!!**

”

Volumes Docker

- *REMARQUE n°1:*
 - les images publiques sont créées avec une *config. par défaut*
 - on peut utiliser les *var. d'environnement* à la marge
 - MAIS comment **injecter un fichier/dossier** dans un conteneur ?
- *REMARQUE n°2:*
 - en cas de suppression d'un conteneur
 - *les données modifiées* depuis la création seront **PERDUES**
 - comment peut on **faire persister les données** d'exploitation

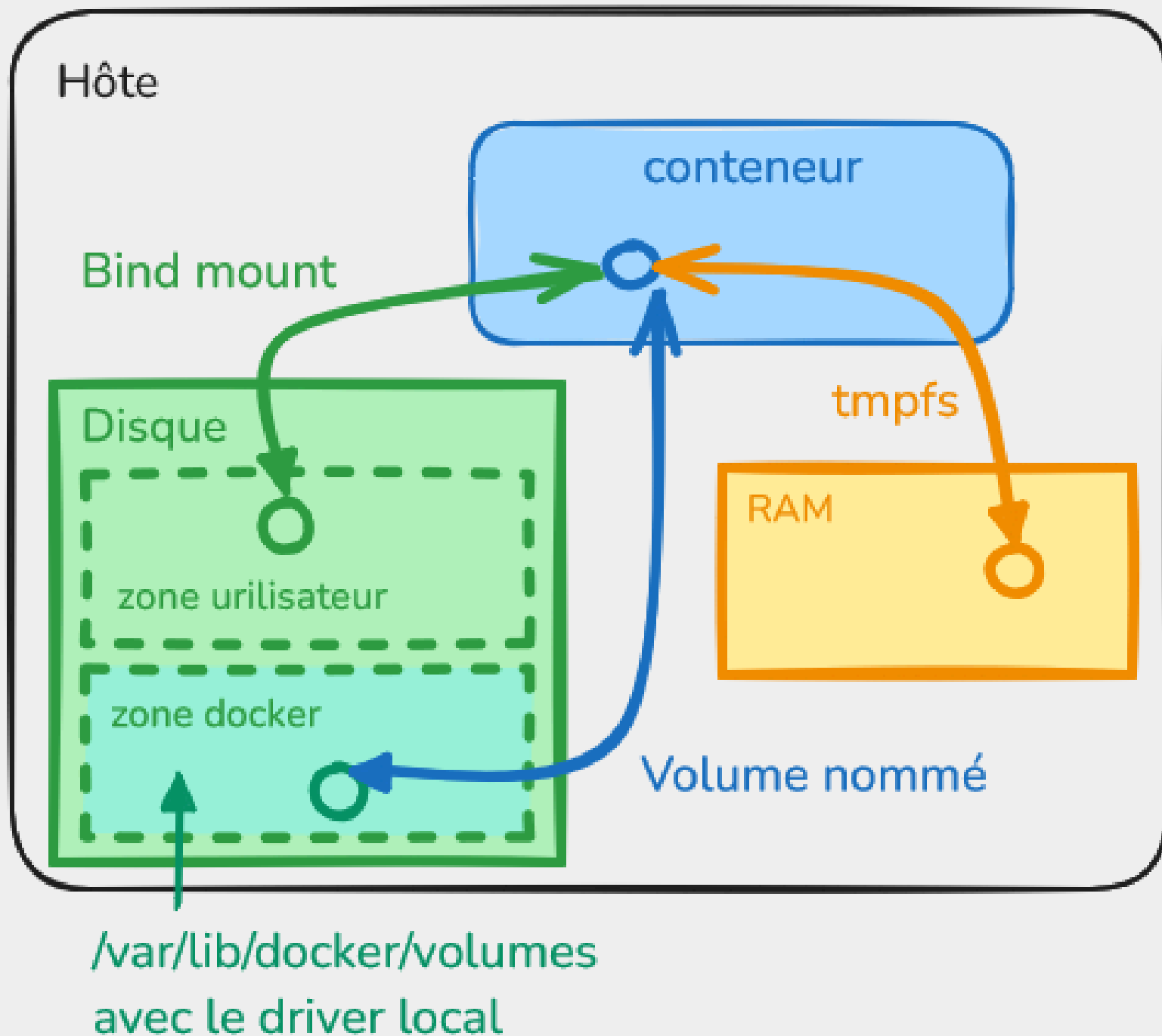
copier un fichier/dossier

- `docker cp <src_path> CTN:<dest_path>`
 - copie un objet en *local dans le conteneur*
- `docker cp CTN:<src_path> <dest_path>`
 - copie un objet du *conteneur dans le hôte*
- L'objet destination **n'est pas** le fichier source - *inodes différents !*
- docker cp produit un flux tar, on peut le rediriger sur *stdout* avec
 - `docker cp CTN:<src_path> - | tar x -0`

types de volumes

- **bind mount:** point de montage i.e *même inode*
 - d'un fichier/dossier local *géré par l'utilisateur*
 - sur un fichier/dossier dans *un ou plusieurs conteneurs*
- **volume nommé:** point montage
 - d'un fichier/dossier d'un conteneur
 - sur un fic/dos *géré par docker* selon un *driver de stockage*
 - *partageable* par d'autres conteneurs
- **tmpfs:** point de montage du conteneur dans la *RAM*
 - persistant tant que le conteneur existe

volumes



Drivers de stockage

- Par défaut, docker utilise le **driver local overlay2**
 - volumes créés sur l'hôte avec le système de fichiers **overlayFS**
- On peut spécifier *d'autres systèmes de fichiers*
 - tmpfs => RAM,
 - sshfs => machine distante,
 - nfs => partage NFS,
 - CIFS => partage samba ...)
- Les volumes docker pilotent en réalité **l'appel système mount**, d'une manière ou d'une autre

utiliser les volumes

- `docker run -v /path/to/obj:/path/on/ctn/obj:opt1, ..., optn, ...`
 - **bind mount:** le choix du chemin sur le hôte *est libre*
 - **options:** propriétés du montage *Ex. ro => ReadOnly ...
 - `docker run -v volume_name:/path/on/ctn:ro ...`
 - **volume nommé:** un label *volume_name* représente un emplacement géré par docker
- “ *ATTENTION: le chemin sur le hôte dans le cas d'un bind mount doit être soit un chemin absolu ou un chemin relatif commençant par . ou .. sinon il paraîtrait comme un NOM !!!* ”

Écriture longue

```
# bind mount
docker run --mount src=path/to/obj,dst=/path/on/ctn/obj,readonly ...
# volume nommé: PEU PRATIQUE !!!
docker run --mount \
src=volume_name,dst=/path/on/ctn/obj,volumedriver=local,volume-opt=type=nfs...
```

- cette écriture longue permet d'utiliser des *options avancées*
- pour un **volume nommé**, la config.
 - *driver*
 - *système de fichier*
 - ...

Création de volumes nommés

```
## PREFERER CELA à la config run --mount !!!
```

```
docker volume create \  
--driver=local -o opt1=val1 -o opt2=val2 \  
<volume_name>
```

```
docker volume ls
```

```
docker volume inspect <volume_name>
```

```
# suppression de volume non lié à un conteneur en exécution
```

```
docker volume rm <volume_name>
```

```
# suppression des volumes non utilisés
```

```
docker volume prune
```

monter tous les volumes d'un conteneur

```
docker run --name=toto -v ./obj:/obj -v volume_name:/path/on/ctn/obj2 ...  
docker run --name=tata --volumes-from=toto ...
```

- **--volumes-from:**

- permet de monter tous les volumes d'un conteneur dans un autre
- *condition:* les 2 conteneurs doivent être *le même réseau*

Docker Compose

- *plugin* client docker
 - Pilote les commandes docker CLI depuis un **fichier de configuration au format YAML**
 - les *commandes docker compose* permettent de lancer un ensemble de conteneurs inter-connectés et dûment configurés
=> outil d'**infrastructure as code**
 - installation: `sudo apt-get install -y docker-compose-plugin`
- “ *docker compose compose une architecture microservice !!!* ”

format YAML

- format de représentation de données par *sérialisation*, conçu pour être *aisément modifiable et lisible*
- Dérivé de la représentation d'un objet *JSON déplié*

```
{                                     ---
  "key" : "value",                   key: value
  "other key" : "other value", => other key: other value
  "num" : "3.14"                     num: 3.14
  "bool" : "false",                 bool: false
  "none" : "null"                   none: null
}                                     "with quotes": "possible"
```

format YAML : imbrications vs JSON

- objets **{ ... }** ==> **indentation de 2 ou 4** char.
- listes **[...]** ==> **indentation de 2 ou 4** char. + « **- [espace]** »
- YAML est *compatible avec JSON*

```
{                                ---
  "object" : {                  object:
    "key" : "value", =>         key: value
    "items" : [                items:
      "item1",                  - item1
      { "k1": "v1", "k2": ... }, - k1: v1
      ...                      k2: v2
      "item3"                  - item3
    ]}}}
```


Correspondances clés / options commandes CLI

```
services:                                # service = app conteneurisée répliquable
  app:                                   # nom du service ARBITRAIRE
    container_name:                      # --name  nom du conteneur ARBITRAIRE
    image:                               # nom de l'image ARBITRAIRE
    build:                               # docker build d'un Dockerfile
      context: .                          # chemin du Dockerfile (contexte de build)
      args: [...]                        # --build-arg
    restart:                             # --restart
    depends_on: [...]                   # dépendance à des services (qui doivent exister !)
    env_file: [...]                     # fichier de variables d'environnement
    environment:                         # variables directement dans le document
      - VAR=${PARAM:default}             # valeurs par défaut
    ports: [8080:80]                     # -p EXT_PORT:INT_PORT ou PORT seul pour exposition
    volumes:                             #
      - type: bind                        # --mount
        source: .                         #
        target: /                         #
      - /src:/dest:opts                  # -v
    networks: [...]                      # --net
```

créations des réseaux et des volumes

```
networks:
  network-name:           # ARBITRAIRE
    name: network-name    # fixe la politique de nommage
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/16
          gateway: 172.18.0.1
  extern-network:
    name: extern-network
    external: true

volumes:
  volume-name:           # ARBITRAIRE
    name: volume-name    # fixe la politique de nommage
```

politique de nommage

- un projet docker compose commence par créer un *dossier* abritant:
 - le fichier **compose.yml** terme *déecté par Docker*
 - plusieurs fic/dos configurations / code ...
- par défaut, au lancement des services:
 - le nom des réseaux / volumes seront:
nom du dossier + "_" + clé, sauf si on ajoute la clé `name: value`
 - le nom des conteneurs seront
nom du dossier + "-" + nom du service + "-[0-9]+"
sauf si on ajoute la clé `container_name: value`

Commandes docker compose principales

- `docker compose up [-d]` : lance l'ensemble des conteneurs
 - créé les *réseaux / volumes (si absents) / services*
 - **-d**: lance *tous* les services en *arrière plan*
- `docker compose down [-v]` : détruit tout sauf les volumes
 - **-v**: détruit aussi les volumes
- commandes analogues à la CLI *mais agrégées ou pour un service*

```
docker compose ps, logs, rm [-f], stop, [re]start [<service-name>]  
docker compose run [--rm|it], exec [-it] <service-name>
```

dépendances entre services

- la clé **depends_on**: attaché aux clés sous **services**:
- présente une liste de *noms de services* qui doivent être lancés avant le *service courant*

```
services:  
  app:  
    image: xxxx  
    depends_on:  
      - db  
  
  db:  
    image: yyyy
```

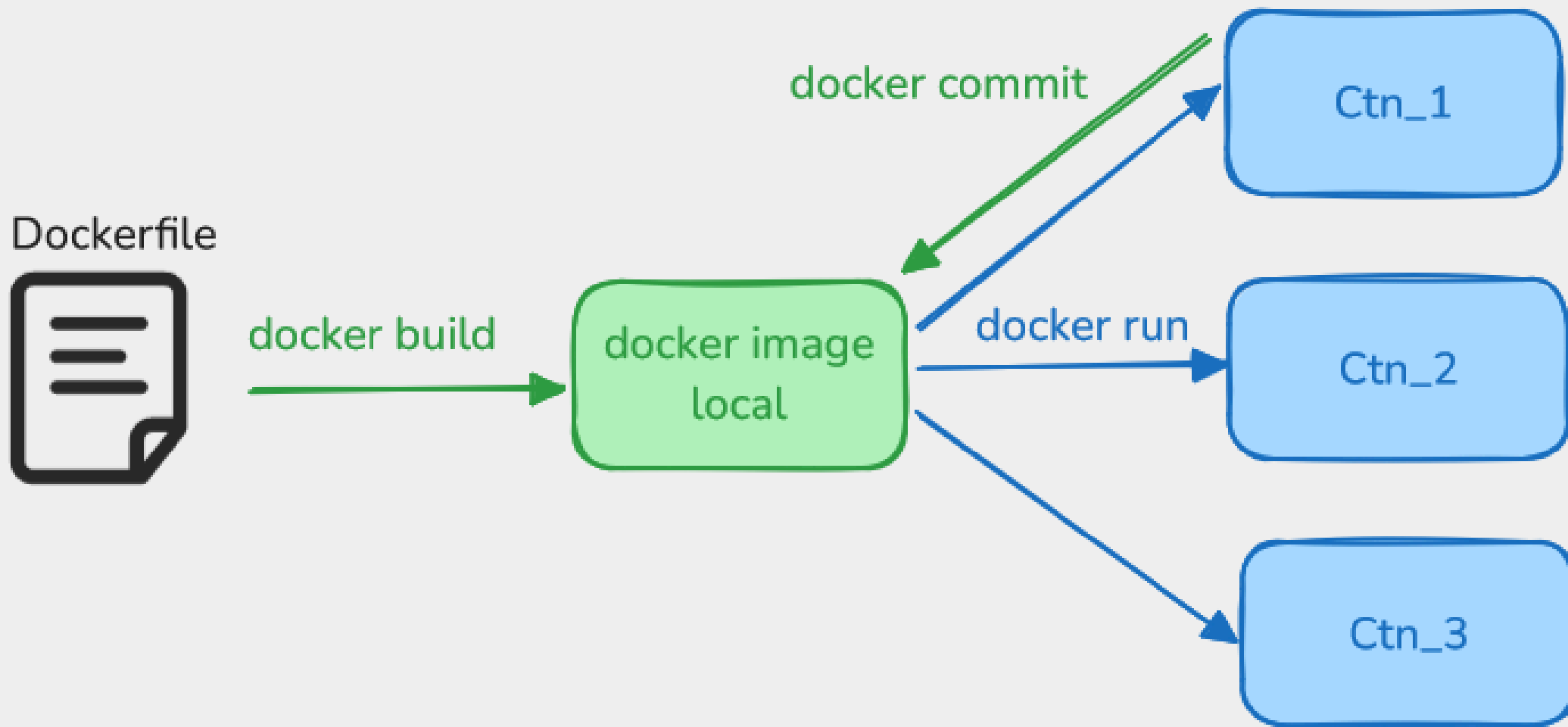
Utilisation des profiles

- La clé **profiles:** attachée aux clés sous **services:**
- présente une liste d'éléments arbitraires qui sélectionnent une série de services qui fonctionnent ensemble

```
# docker compose --profile profile-name up, down
services:
  app:
    ...
    profiles:
      - profile-name
```

“ *en utilisant les profiles, tous les services doivent avoir un profile et l'option --profile doit être utilisée !!!*

Créer une image



retransformer un conteneur en image

- `docker diff <ID | ctn_name>` : affiche les **modifs** opérées dans un conteneur depuis le `docker run`

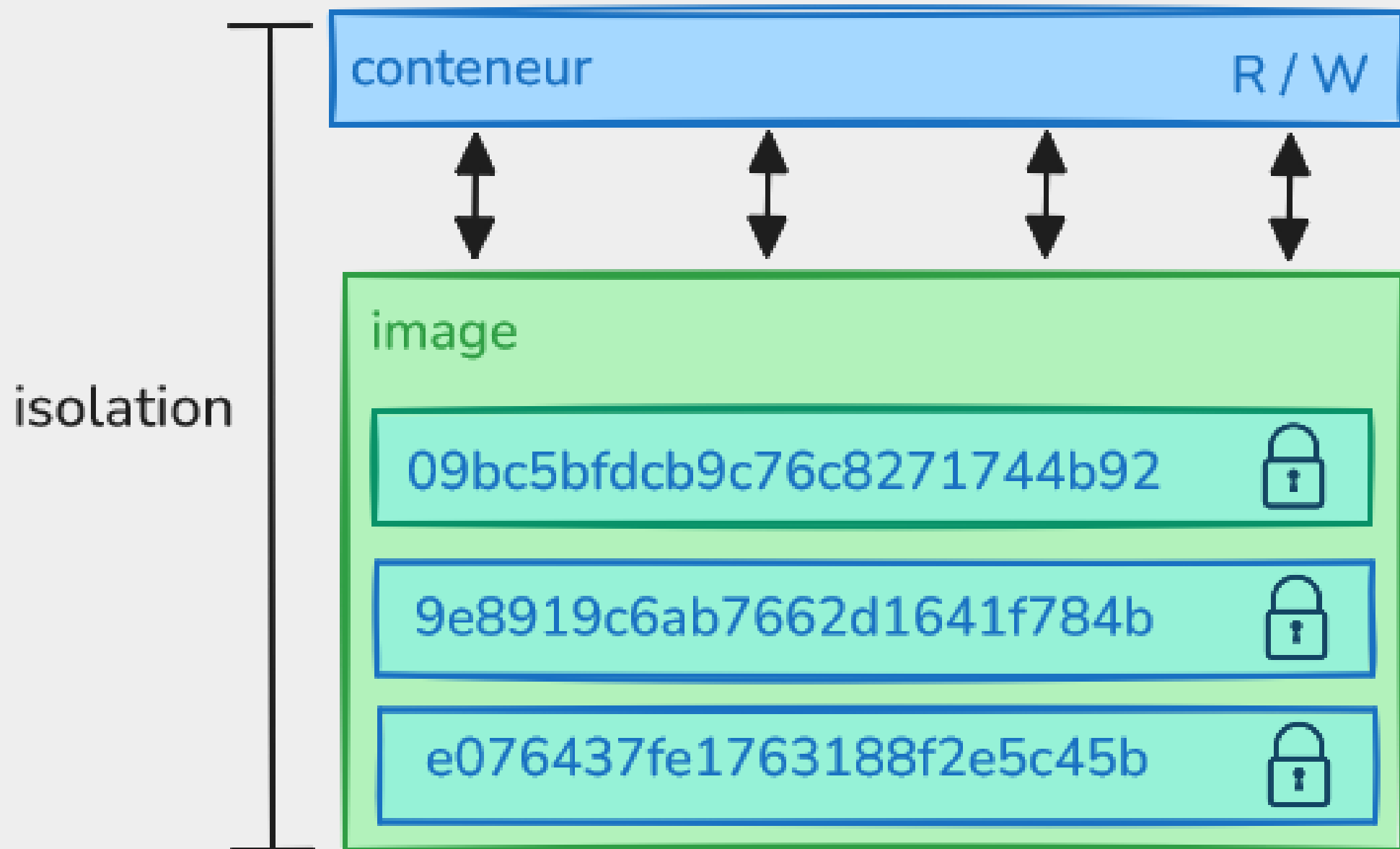
```
A /my_dir/my_file # Add
C /var/cache/apk  # Change
D /home           # Deletion
```

- `docker commit [-a author] [-m msg] <ID | ctn_name> :<tag>` :
"ajoute" ces modifs dans la nouvelle image `:<tag>`
- “ *commit ne considère pas les données partagées des volumes !!!
commit s'opère à froid ou avec --pause* ”

composition d'une image

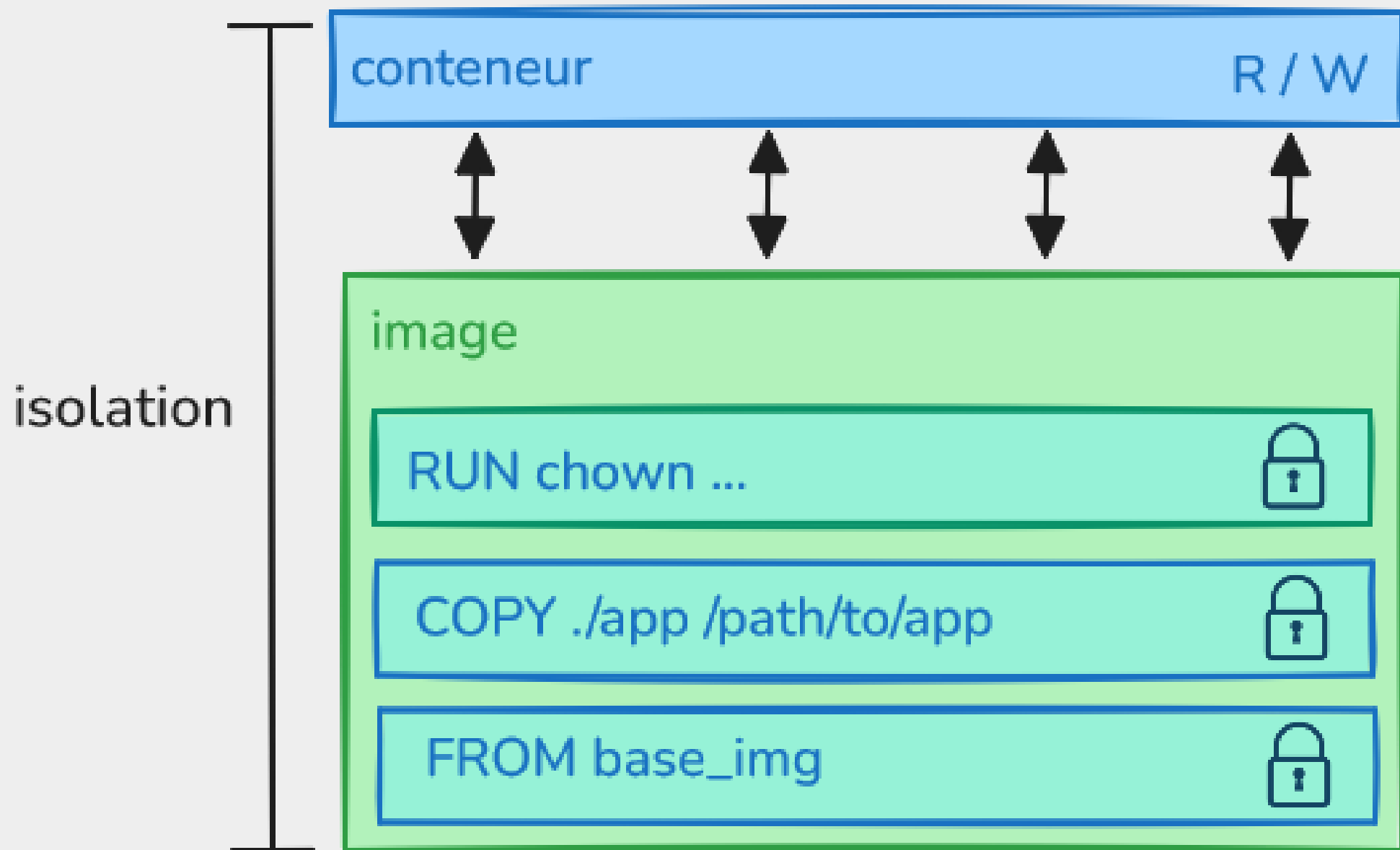
- Une image Docker est un **empilement de couches de système de fichier en lecture seule** - par défaut Docker utilise **OverlayFS**
- Le conteneur consiste principalement en une **fine couche en lecture/écriture** au dessus de l'image
- plusieurs conteneurs batis sur la même image
=> partagent les couches communes en lecture seule
- On peut utiliser *au maximum 127 couches* pour une image

```
# size : taille de la couche RW  
# virtual size : taille RW + RO (mutualisées)  
docker ps -s
```



Le fichier Dockerfile

- rassemble une série d'**instructions** docker
- les instructions principales *créent une couche* de sys. de fichier
 - *FROM / COPY / ADD / RUN / WORKDIR*
- certaines instructions créent ou modifient des *métadonnées*
 - *LABEL / ENV / ARG / USER / EXPOSE*
- et d'autres *déclenchent une action*
 - *VOLUME / ENTRYPOINT / CMD / HEALTHCHECK*



les instructions

```
# image et tag de base OU scratch: pour une image de distrubtion linux
# --platform: sélectionne l'architecture voulue (amd64 / arm64 / ...)
FROM [ --platform ] <image>:<tag>
```

```
# copie des éléments de l'hôte dans l'image
# utilisation de wildcards ( *, ? ) pour les éléments source
# --chown: changer le propriétaire des éléments copiés
COPY [ --chown ] <src> <dest>
```

```
# comme COPY mais gestion de protocoles réseaux pour la source
ADD [ --chown ] <src> <dest>
```

```
# exécute une commande shell
RUN <cmd>
```

```
# répertoire par défaut à partir duquel le conteneur est lancé
WORKDIR <path>
```

```
# métadonnées arbitraires associées à l'image et au conteneur
# utilisé pour documenter, filtrer, voire configurer (ex. Traefik)
# docker image ls -f label=<label_name>
# docker ps -f label=<label_name>=<label_value>
LABEL <key> <value>

# crée ou modifie des variables d'environnement dans l'image
# Ces variables ne peuvent être modifiées qu'au lancement du conteneur !!!
ENV VAR=value

# IDEM SAUF que ces variables ne sont spécifiées qu'au build de l'image !!!
ARG VAR[=value]

# fixe l'utilisateur utilisé au moment du lancement du conteneur
# l'utilisateur DOIT EXISTER !!!
USER <user[:group]>

# déclare les ports réseau sur lesquels le service installé dans l'image DOIT ECOUTER
EXPOSE <ports>
```

```
# création de volumes persistant les chemins renseignés dans Docker
# ce sont des volumes anonymes !!!
# docker run --volumes-from: pour les exploiter
VOLUME <paths>

# commande automatiquement lancée avec un conteneur de cette image
# commande non-substituable par docker run => ERROR
ENTRYPOINT <cmd>

# IDEM SAUF que la commande est substituable par docker run
# on peut combiner ENTRYPOINT + CMD
CMD <cmd>

# commande automatiquement lancée après les premières ci dessus
# DOIT prouver l'état valide du conteneur
HEALTHCHECK [ --options ] CMD <cmd>
```

le build: exécuter le Dockerfile

- on crée **un dossier** pour le *Dockerfile* et tous les éléments constitutifs de l'image *configurations, code, ...*
- ce dossier est nommé **contexte de build**
- le fichier **.dockerignore** ignore les éléments du contexte inutiles au build:
 - indisponible pour les instructions COPY / ADD
 - intérêt: contexte chargé côté serveur => *bande passante !!*

types de build

- `docker build -t <img:tag> [proto://]path/to/context`
 - **-t <img:tag>**: nommer la nouvelle image avec son tag
- `docker build -t <image : tag> - < [proto://]/path/to/Dockerfile`
build sans contexte

```
# build paramétré par les instructions ARG !!!  
docker build -t <img:tag> \  
--build-arg PARAM=val \  
--build-arg PARAM2=val2 ...
```

le builder

- par défaut, le **plugin client buildx** et le **backend buildKit** s'occupent du build
 - chargement du *contexte de build* dans buildKit
 - exécution des *instructions => couches*
- les couches sont **mises en cache** pour accélérer les builds successifs

```
docker buildx du # voir le cache
docker buildx prune [--filter until=1h] # vider le cache
docker build --no-cache ... # build sans cache
```

test le build

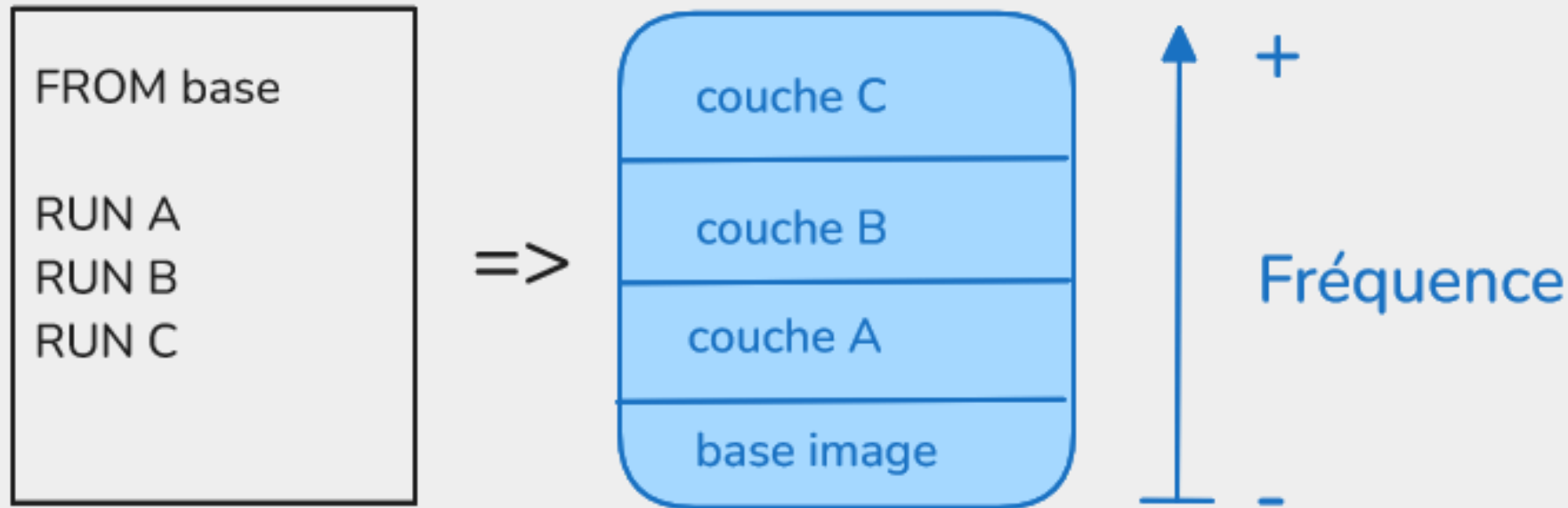
```
# ENTRYPOINT OU/ET CMD ...  
HEALTHCHECK \  
    --start-period=5s \  
    --timeout=30s \  
    --interval=5s \  
    [--retries=3 \  
CMD <cmd>
```

- après 5s, exécute CMD avec Δ_{max} de 30s,
- si CMD est en erreur, **réessaie** 3x en attendant 5s *entre essais*
- sans `--retries=n`, **HEALTHCHECK** devient *périodique*

“ on voit **"healthy"** ou **"unhealthy"** dans *docker ps / inspect* !!!

bonnes pratiques de performances

1. Ajout des couches par *fréquence de mise à jour* croissante
stable => instable



2. minimiser le nb de couches

- *chaîner les RUN* avec **&& ** ou **<< EOF**
- ADD => RUN curl

3. enchaîner *création / modif / suppression* dans la même couche

4. travailler en flux **|** et non en fichier **&&**

5. privilégier les images/paquets *les plus légers*

```
RUN apt-get update && apt-get install -y \  
xxx && \  
... && \  
rm -rf /var/lib/apt/lists/*
```

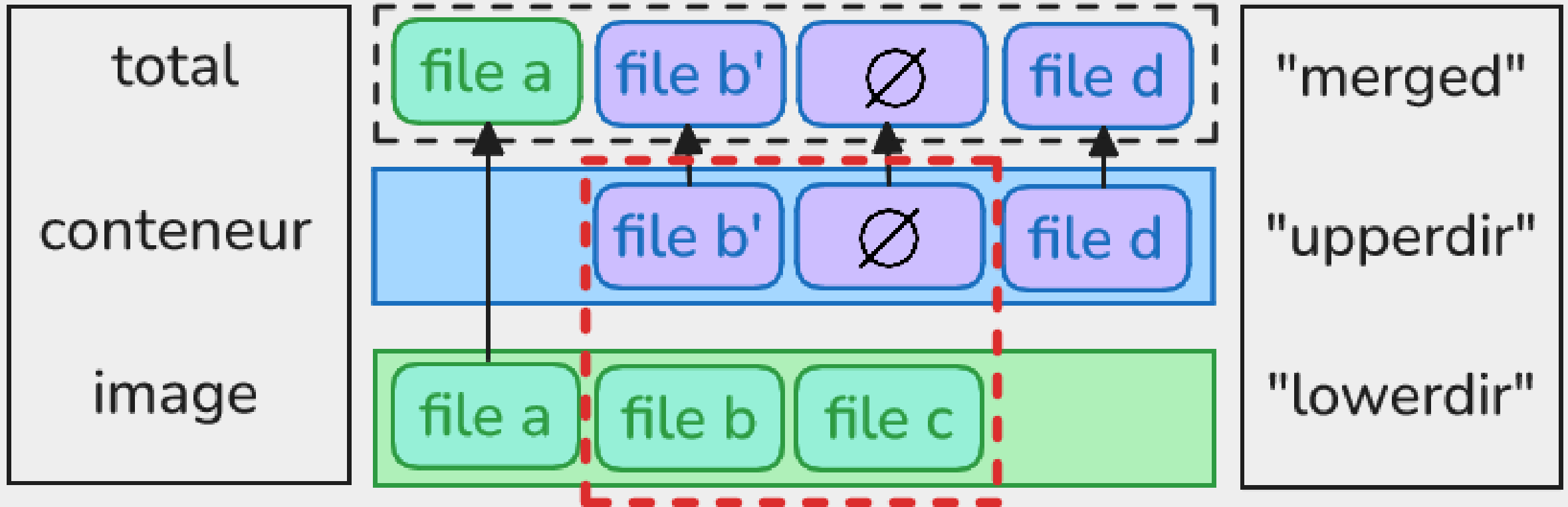
```
RUN curl https://xxx.xx | \  
tar xvz -C ...
```

Mécanisme « Copy on Write » de OverlayFS

- **CoW** permet de **réutiliser les objets** d'une couche inférieure dans les couches supérieures *TANT QUE* ces objets *ne sont pas modifiés*
 - => *en cas de changement*, **l'objet au complet est copié**
 - le **build** consiste à *itérativement*:
 - créer une nouvelle couche R/W utilisant le CoW
 - et rendre la nouvelle couche exécutée en RO
 - le **run** créer une nouvelle couche R/W utilisant le CoW
- “ *modifier/supprimer un objet dans une couche supérieure laisse toujours l'objet précédent dans l'image !!!*
=> *bonne pratique 3. !!!* ”

ce qu'on voit !

OverlayFS



réalité sur le disque !!

«builder» avec docker compose

```
services:
  svc:
    image: <image>:<tag>
    build:
      context: /path/to/context
      dockerfile: Dockerfile
      # build sans cache
      # no_cache: true
    args:
      - VAR=value
      # piloter l'argument par une var. d'env.
      # voire une valeur par défaut
      - VAR=${VAR:-default}
```


les labels et healthcheck avec docker compose

```
services:
  svc:
    ...
    labels:
      - author=bob
      - created_at="2020-03-14"
      - stack.image="svc"
    healthcheck:
      test: ["executable", "arg"]
      start_period: 5s
      timeout: 30s
      interval: 5s
      retries: 3
```

Registre d'image privé

- pour stocker les images custom buildée:
 - docker Hub (public) + compte docker (privé)
 - installer un conteneur de l'**image registry** dans l'env. pro
- configurations principales: `/etc/docker/registry/config.yml`
 - accès http sur le *port 5000* ou **http + TLS** sur le port *443*
 - authentication: *htpasswd, Jeton JWT, ...*
 - stockage, cache, ...

utiliser le registre

logging sur le registre

```
docker login -u [username] -p [password]
```

l'image à pousser DOIT PREFIXER l'adresse (ip ou dns) du registre

```
docker push <addr>:<port>/[<path>/]<img>:<tag>
```

"changer" le nom d'une image locale

i.e, créer une référence //

```
docker tag <img>:<tag> <addr>:<port>/[<path>/]<img>:<tag>
```

builder une image directement dans le registre

```
docker build -t <addr>:<port>/[<path>/]<img>:<tag> --push /path/to/context
```

pull

```
docker pull <addr>:<port>/[<path>/]<img>:<tag>
```

contournement d'un certificat TLS auto-signé

- éditer le fichier config. du serveur docker `/etc/docker/daemon.json`

```
{  
  ...  
  "insecure-registries": [ "xxx.yyy.zzz.ttt:443", "domain.name:443" ]  
}
```

```
sudo systemctl daemon-reload  
sudo systemctl restart docker
```

Techniques avancées

mécanisme «multi staging build»

- utilise *plusieurs instructions FROM* comme étapes de construction, images auxiliaires ou **stage**
- on exploite les stages *RUN, COPY, ADD, ...*
et on injecte les objets intéressants dans l'*image buildée*

```
FROM <image>:<tag> AS <stage_name>
...
FROM <build_image:tag>
COPY --from=<stage_name> [stage_path_src] [build_image_dest]
```

build «multi-plateforme »

- méthode avec *Docker Engine + QEMU*
- prérequis (2 possibilités)
 - utiliser le *dépôt d'images de containerd* (expérimental)
 - **OU** créer un *builder custom*
- installation de l'émulation des architectures via QEMU

```
docker run --privileged --rm \
tonistiigi/binfmt \
--install all | linux/amd64,linux/arm64...
```

```
cat /proc/sys/fs/binfmt_misc/qemu-arm | grep "F" # OK!!
```

dépôt d'images de containerd (I)

```
// dans la CONFIG SERVEUR de docker /etc/daemon.json  
{  
  ...  
  "features": {  
    ...  
    "containerd-snapshotter": true  
  }  
}
```

- `sudo systemctl restart docker` : redémarre le service docker

exécuter le build multi-plateforme (I)

```
docker buildx ls
```

NAME/NODE	DRIVER/ENDPOINT	STATUS	BUILDKIT	PLATFORMS
default*	docker			
_ default	_ default	running	v0.16.0	linux/amd64, linux/arm64

```
FROM alpine
```

```
RUN uname -m > /arch
```

- l'architecture d'un conteneur d'une image multi-plateforme est **ajustée à l'hôte**

```
docker build --platform linux/amd64,linux/arm64 -t multiplat .
```

```
docker run --rm multiplat cat /arch # x86_64
```


créer un builder custom (II)

```
# en utilisant l'image moby/buildkit:buildx-stable-1
docker buildx create \
    --name multiplat \
    --driver docker-container \
    --bootstrap \           # lancer
    --use                   # utiliser

## ce builder ne peut pas utiliser le dépôt d'image
## => build dans une archive OU LE REGISTRY
## le builder DOIT pouvoir se connecter en TLS dans le REGISTRY
BUILDER=$(sudo docker ps | grep buildkitd | cut -f1 -d' ')
sudo docker cp \
    /etc/docker/certs.d/formation.lan:443/ca.crt \
    $BUILDER:/usr/local/share/ca-certificates/ca.crt
sudo docker exec $BUILDER update-ca-certificates
sudo docker restart $BUILDER
```

exécuter le build multi-platforme (II)

```
docker buildx ls
NAME/NODE    DRIVER/ENDPOINT  ...
multiplat*   docker-container ...
  \_ mutliplat0 \_ unix:///var/run/docker.sock ...

## option1
[docker buildx use multiplat]
docker buildx build \
    --platform linux/amd64,linux/arm64 -t formation.lan:443/multiplat . \
    --push

## option2
docker build \
    --builder multiplat \
    --platform linux/amd64,linux/arm64 -t formation.lan:443/multiplat . \
    --push

docker run --rm multiplat cat /arch # x86_64
```

partager un serveur X11 (GUI) dans un conteneur

```
ARG USER=<SAME_user_than_in_host>
ARG UID=<SAME_uid_than_in_host>
ARG HOME=/home/$USER
RUN useradd -d $HOME -s /bin/bash -m $USER -u $UID -U
WORKDIR $HOME
USER $USER
ENV HOME=$HOME
```

environment:

DISPLAY: *\$DISPLAY # partager la MEME valeur de DISPLAY*

hostname: *\${HOSTNAME} # MEME hostname*

volumes:

- */tmp/.X11-unix/:/tmp/.X11-unix # partager la socket x11*
- *\$HOME/.Xauthority:\${HOME}/.Xauthority # MEME config xauth*

mécanisme «watch»

- la structure **develop: watch:** attachée aux clés **services:**
- met à jour en temps réel les services en fonction de changements sur les fichiers de ces derniers
- la clé **action:** permet 3 types de mise en jour:
 - *sync*: Δ sur un fichier de l'hôte => **maj** du fichier du conteneur
=> pour les *fichiers statiques*
 - *sync + restart*: idem + **redémarrage** du conteneur
=> pour les *fichiers de configuration*
 - *rebuild*: on **rebuild** le service
=> pour les *fichiers constituant le Dockerfile*

```
## obj: file or directory
develop:
  watch:
    - action: sync
      path: /path/to/watch/obj
      target: /path/on/ctn/obj
      # if obj is directory
      # do not watch these
      ignore:
        - /path/to/watch/obj/something
        - ...

    - action: sync+restart
      path: /path2/to2/watch2/obj2
      target: /path2/on2/ctn2/obj2

    - action: rebuild
      path: /path3/to3/watch3/obj3
```

déclencher le mécanisme «watch»

- 2 commandes **docker compose** pour le déclenchement:
 - `docker compose watch`: **build + watch**
 - `docker compose up [--build] --watch`: **[build] +watch + logs**
 - **-d** ne fonctionne pas avec ces déclenchements
- “ **Le mécanisme ne fonctionne pas avec les Binds Mounts !!!**
=> avec les fichiers liés aux instructions COPY/ADD/RUN ”

scanner les images Docker

- utilisation de **Docker Scout CLI** => *besoin d'un compte Docker*

```
# INSTALLATION
if [ ! -d "~/.docker" ]; then
  mkdir ~/.docker
fi
curl -sSfL \
https://raw.githubusercontent.com/docker/scout-cli/main/install.sh | sh -s --
```

- Compte Docker => Avatar => Account settings
=> Personal Access Tokens => create token

```
# CONNEXION au compte
echo "xxx-token-xxx" | docker login -u <username> --password-stdin
```

évaluer une image

- `docker scout quickview [proto://]<image:tag>`: affiche le nb de vulnérabilités de l'*image* est celle de l'instruction *FROM (base)*
- sources *[proto]*
- **local:** locale
- **registry:** registre (*pb! self-signed certs*)
- **image:** *par défaut*, locale ou registre (*pb! self-signed certs*)
- **archive:** `docker save -o img.tar`


```
$ docker scout quickview mariadb:11.5
```

```
✓ SBOM of image already cached, 206 packages indexed
```

```
i Base image was auto-detected. To get more accurate results, \
  build images with max-mode provenance attestations.
  Review docs.docker.com 🔗 for more information.
```

Target	mariadb:11.5	3C	35H	14M	7L	5?
digest	980042c20069					
Base image	ubuntu:24.04	0C	0H	1M	4L	
Updated base image	ubuntu:24.10	0C	0H	0M	0L	
				-1	-4	

What's next:

```
View vulnerabilities \
```

```
→ docker scout cves mariadb:11.5
```

```
View base image update recommendations \
```

```
→ docker scout recommendations mariadb:11.5
```

scanner les vulnérabilités

- `docker scout cves [proto://]mariadb:11.5`: décrit l'image et liste les **vulnérabilités** liées aux **librairies installées** dans l'image
- `// --format only-packages`:
voir uniquement la liste
- `// --format only-packages --only-package-type golang`:
Filtrer par le *type de librairies* (deb, pypi, golang, composer ...)
- `// --format only-packages --only-severity critical, high`
Filtrer par *criticité* (critical, high, medium, low, unspecified)
- `// --format only-packages --only-base`: uniquement pour le *FROM*

```
$ docker scout cves mariadb:11.5
✓ SBOM of image already cached, 206 packages indexed
X Detected 7 vulnerable packages with a total of 64 vulnerabilities
```

Overview

	Analyzed Image
Target	mariadb:11.5
digest	980042c20069
platform	linux/amd64
vulnerabilities	3C 35H 14M 7L 5?
size	137 MB
packages	206

Packages and Vulnerabilities

```
3C 35H 12M 1L 5? stdlib 1.18.2
pkg:golang/stdlib@1.18.2
```

```
X CRITICAL CVE-2024-24790
https://scout.docker.com/v/CVE-2024-24790
Affected range : <1.21.11
Fixed version : 1.21.11
```

```
docker scout cves mariadb:11.5 --format only-packages
```

```
✓ SBOM of image already cached, 206 packages indexed
```

```
✗ Detected 7 vulnerable packages with a total of 64 vulnerabilities
```

Name	Version	Type	Vulnerabilities			
acl	2.3.2-1build1	deb	0C	0H	0M	0L
adduser	3.137ubuntu1	deb	0C	0H	0M	0L
apt	2.7.14build2	deb	0C	0H	0M	0L
attr	1:2.5.2-1build1	deb	0C	0H	0M	0L
audit	1:3.1.2-2.1build1	deb	0C	0H	0M	0L
base-files	13ubuntu10.1	deb	0C	0H	0M	0L
base-passwd	3.6.3build1	deb	0C	0H	0M	0L
bash	5.2.21-2ubuntu4	deb	0C	0H	0M	0L

chercher des recommandations

- `docker scout recommendations [proto://]mariadb:11.5:`
affiche un rapport d'**améliorations**

```
## Recommended fixes
```

```
Base image is ubuntu:24.04
```

Vulnerabilities		0C	0H	1M	4L
Packages		130			
OS		24.04			

```
Change base image
```

```
The list displays new recommended tags in descending order, \
where the top results are rated as most suitable.
```

Tag	Details	Pushed	Vulnerabilities
24.10	Benefits:	2 weeks ago	0C 0H 0M 0L
Minor OS version update	• Minor OS version update		-1 -4

exporter un rapport de vulnérabilités

- `docker scout quickview|cves|rec. --format format -o <path> ...`
- **--format:**
 - texte: packages (verbeux), only-packages (simple)
 - dialectes JSON: sarif, spdx, **gitlab**, sbom
 - markdown
- **-o:** sortie dans un fichier