

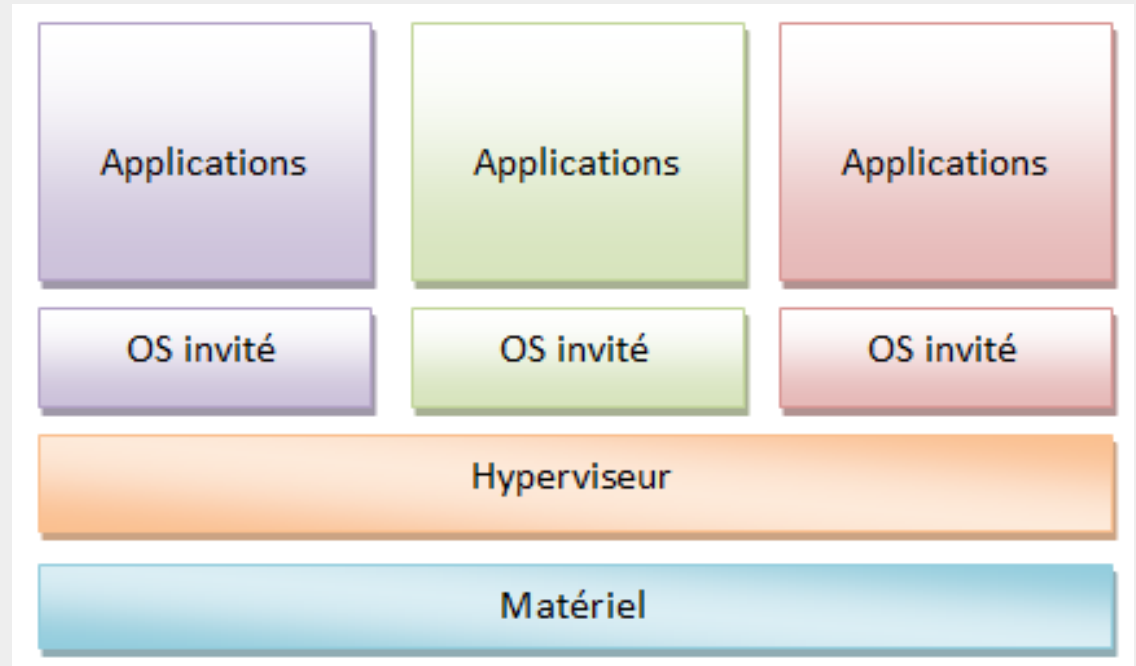
DOCKER

Virtualisation vs conteneurs

■ Virtualisation = partage de ressources physiques

■ Hyperviseur :

- Fine couche d'OS = Noyau léger
- Alloue des ressources physiques
- i.e instructions CPU, adresses RAM
- Pour Créer des Machines Virtuelles
- i.e des OS « invités » différents
- Ex : VMWare, Xen, KVM, Hyper-V



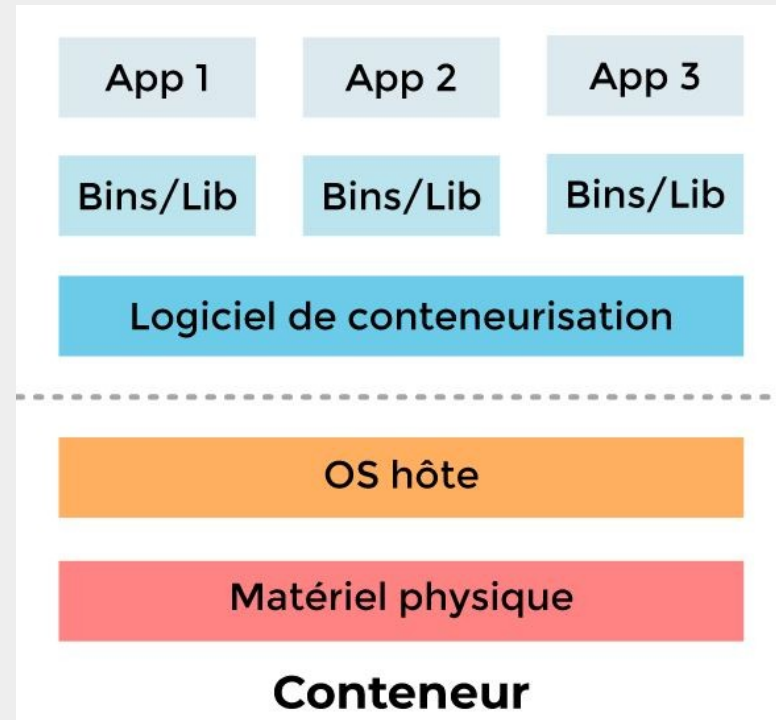
Virtualisation vs conteneurs

■ Enjeux de la virtualisation

- Meilleure adaptation aux besoins du parc informatique
- Gain énergétique : réduction des coûts
- Plus de flexibilité : déplacement, sauvegarde / restauration des Vms
 - PRA : Plan de Relance de l'Activité
 - RPO : « Recovery Point Objective », durée maximale d'enregistrement des données qu'il est acceptable de perdre lors d'une panne => objectif de sauvegarde
 - RTO : « recovery time objective », durée maximale d'interruption que l'on est prêt à supporter
=> objectif de restauration
- Gain en Qualité de Service (QOS)
- Moindres dégradations de Performances grâce au « Super - BIOS » de technos VT-x ou AMD-V
 - accès direct sur certains drivers E/S pour les os invités « para-virtualisation »
 - architectures processeur facilitant le traitements d'instructions d'un os invité étiquetées par l'hyperviseur

Virtualisation vs conteneurs

- Conteneurisation = partage des ressources logicielles du noyau de l'OS
- Logiciel de conteneurisation :
 - Créé des conteneurs
 - Isolent des processus du reste de l'OS hôte
 - Avec leur propre « vision » du système (namespaces)
 - Limités dans les ressources utilisables



Virtualisation vs conteneurs

■ Comparaison

■ Les conteneurs :

- **Plus légers**, ne comprennent que des logiciels de haut niveau (~Mo vs ~Go pour les Vms avec leur noyau et les librairies bas niveau)
- **Plus rapides** à créer et déplacer

■ Les Vms

- **Isolation** totale des processus entre Vms, alors que le noyau de l'hôte est partagé entre conteneurs
- **Plus dynamiques** : Les Vms sont conçues pour les interactions utilisateur, les conteneurs pour assurer les bonnes dépendances / configuration statiques du processus à isoler

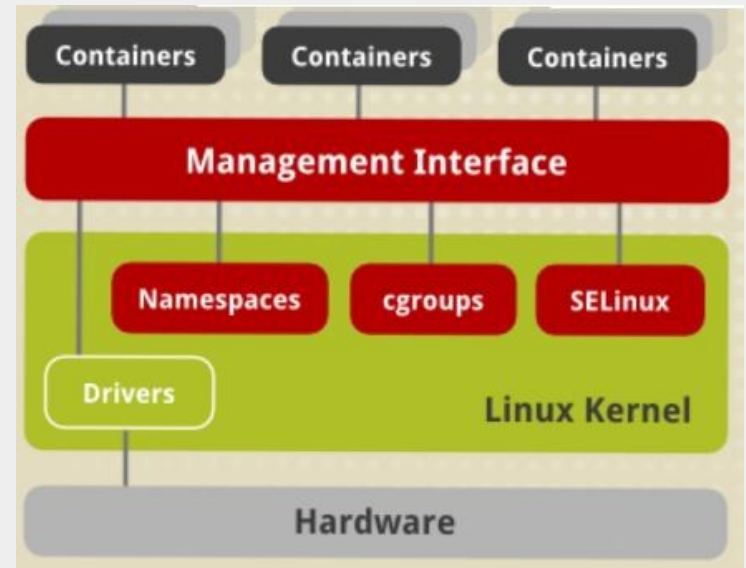
■ Les usages sont différents

conteneurs Linux

■ Les namespaces

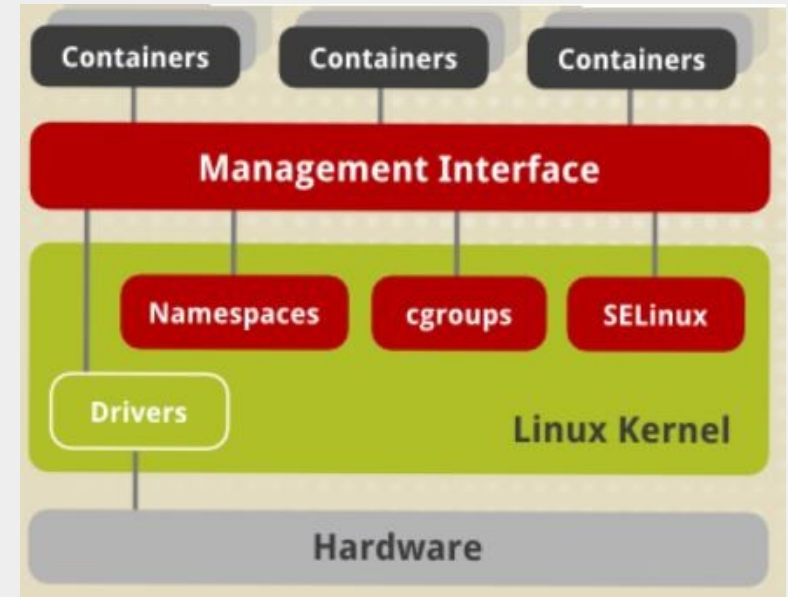
■ Fonctionnalités du noyau linux qui assurent certains aspects de **l'isolation** des conteneurs :

- pid : isolation des processus (1 table de process par ns)
 - (Pid, Ppid) dans le ctn != (Pid, Ppid) dans le host
- net : isolation d'interfaces réseau (1 pile réseau par ns)
 - interfaces, addresses IP , routes, firewall
- mount : isolation des systèmes de fichiers
 - systèmes de fichiers des images docker
- uts : isolation du nom d'hôte
- ipc : « inter processus communication » entre ns
 - interdiction des cnx avec les sémaphores, segments de mémoire partagés, files
- user : mapping des UID/GID entre l'hôte et les conteneurs host uid xxx → ctn uid 0 (par défaut)



conteneurs Linux

- Les Cgroups :
- Fonctionnalités du noyau linux qui régulent l'accès aux **ressources** des processus :
 - RAM
 - limites physiques et logicielles
 - CPU
 - proportion du pool CPU
 - répartition des traitements sur les coeurs
 - I/O
 - Network
 - limitation des débits lecture / écriture



Microservices

■ Architecture monolithique vs microservices :

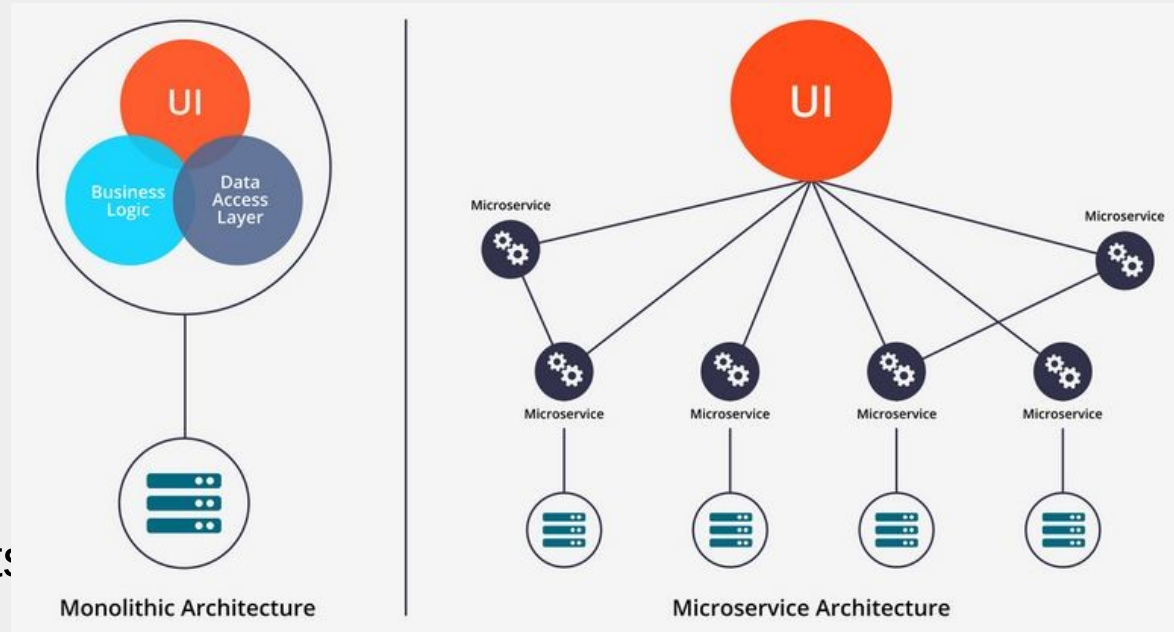
■ Une application :

- Décomposés en **services**
- Packagés dans des conteneurs
- Disséminés dans le **cloud**
- Qui communiquent sur le **réseau**
- Via des **APIs**

■ **Couplage faible** entre composants

■ Pros / cons

- Meilleure **stabilité** des infra, meilleur **MTRS** (Mean Time to Restore Service)
- Plus **complexe** à mettre en œuvre => utilisation **d'orchestrateurs**



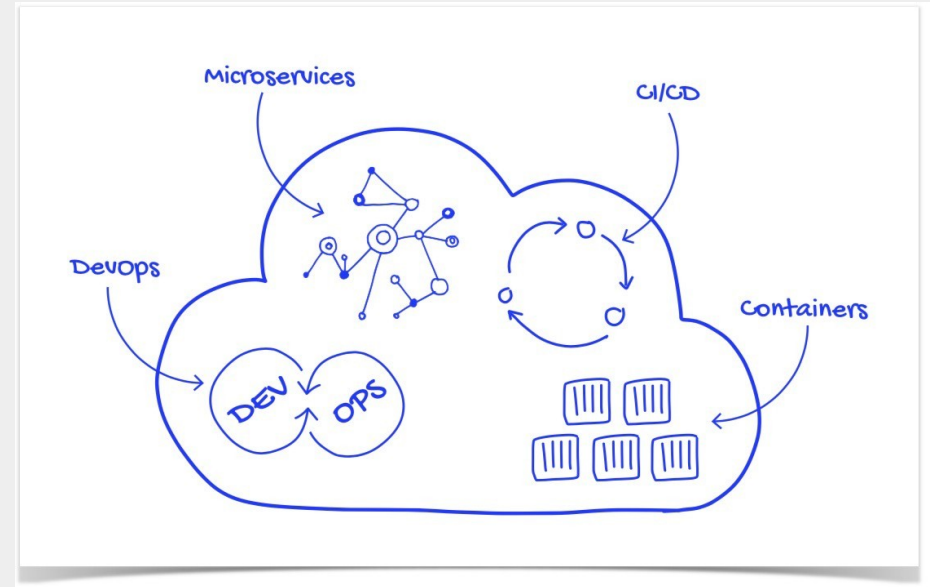
Applications Cloud Native

■ Définition :

- Cloud Native Computing Foundation
- Application orientée Microservice
- Dont le cycle de vie est géré par un orchestrateur
- **Scaling horizontal** facilité
- Aptitude au **stockage et au calcul distribués**

■ Exemples : cncf.io

- Kubernetes
- Prometheus
- Gitlab
- ...

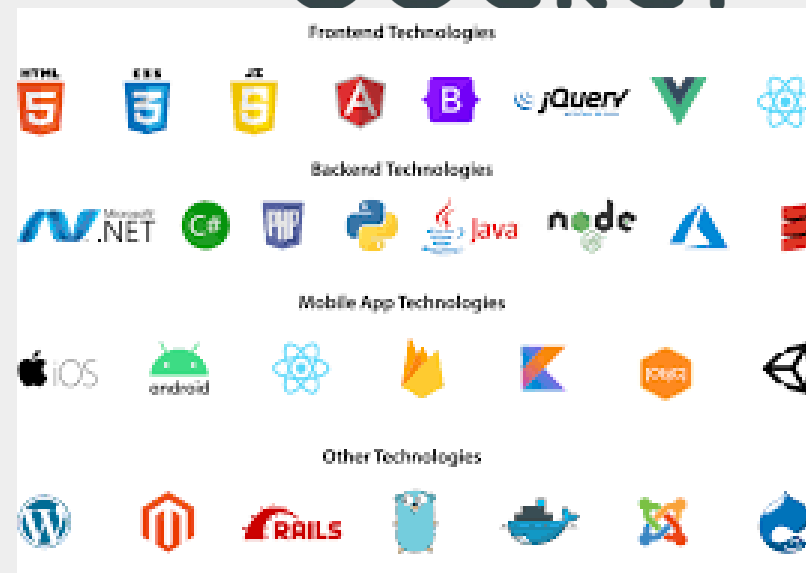
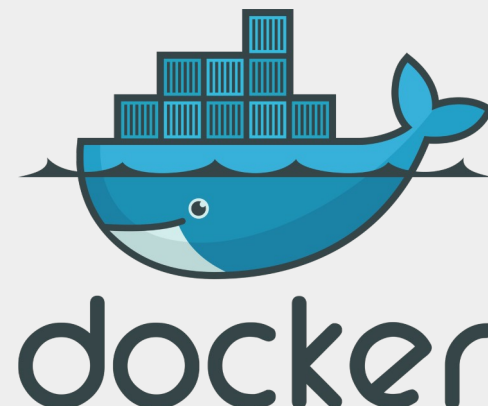


DOCKER

■ Docker : société commerciale et technologie

■ technologie :

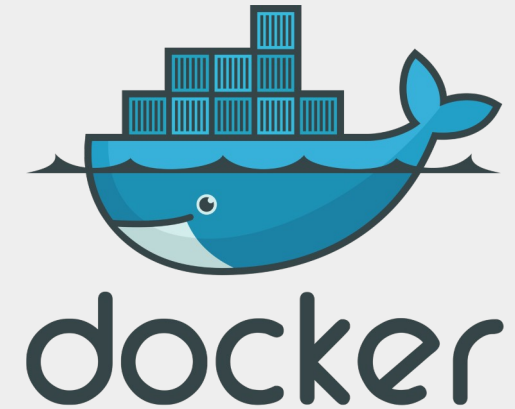
- Nombreuses applications prêtes l'emploi
- Décrites par des **images** (instantanés de systèmes de fichiers)
- Téléchargeables depuis un **registre** privé ou public (docker hub)
- Packagées dans des conteneurs



DOCKER

■ Offre open source docker-ce:

- Distribuée à partir de 2013
- <https://github.com/docker/docker-ce/blob/master/CHANGELOG.md>
- Concurrence
 - LXC (Linux)
 - Rkt (CoreOs – RedHat)
 - LXD (Canonical – ubuntu)



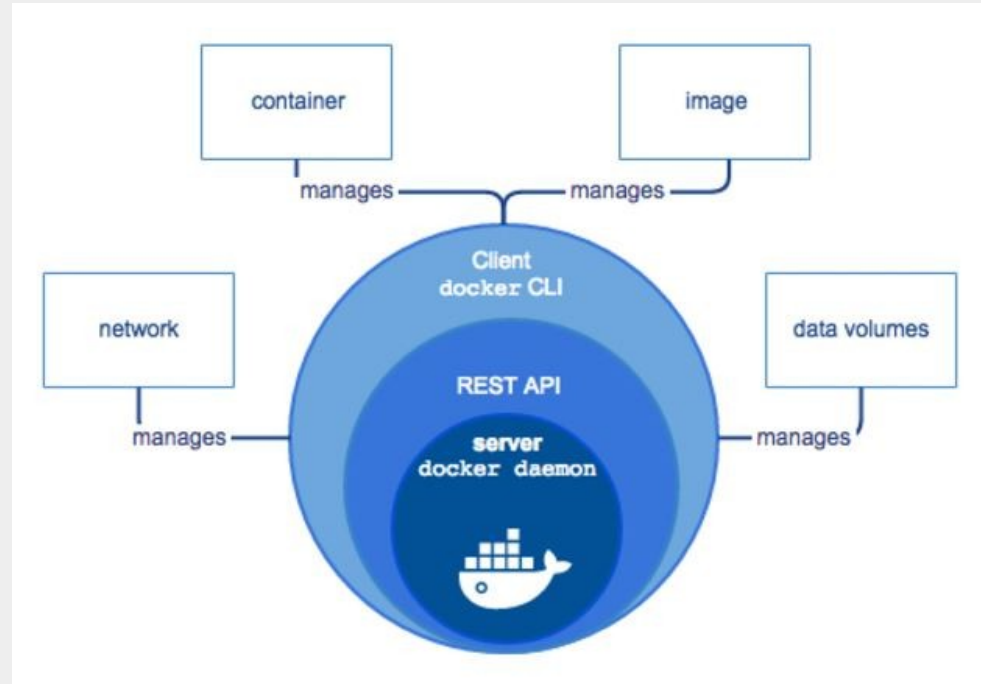
■ Offre payante

- Docker Business

DOCKER

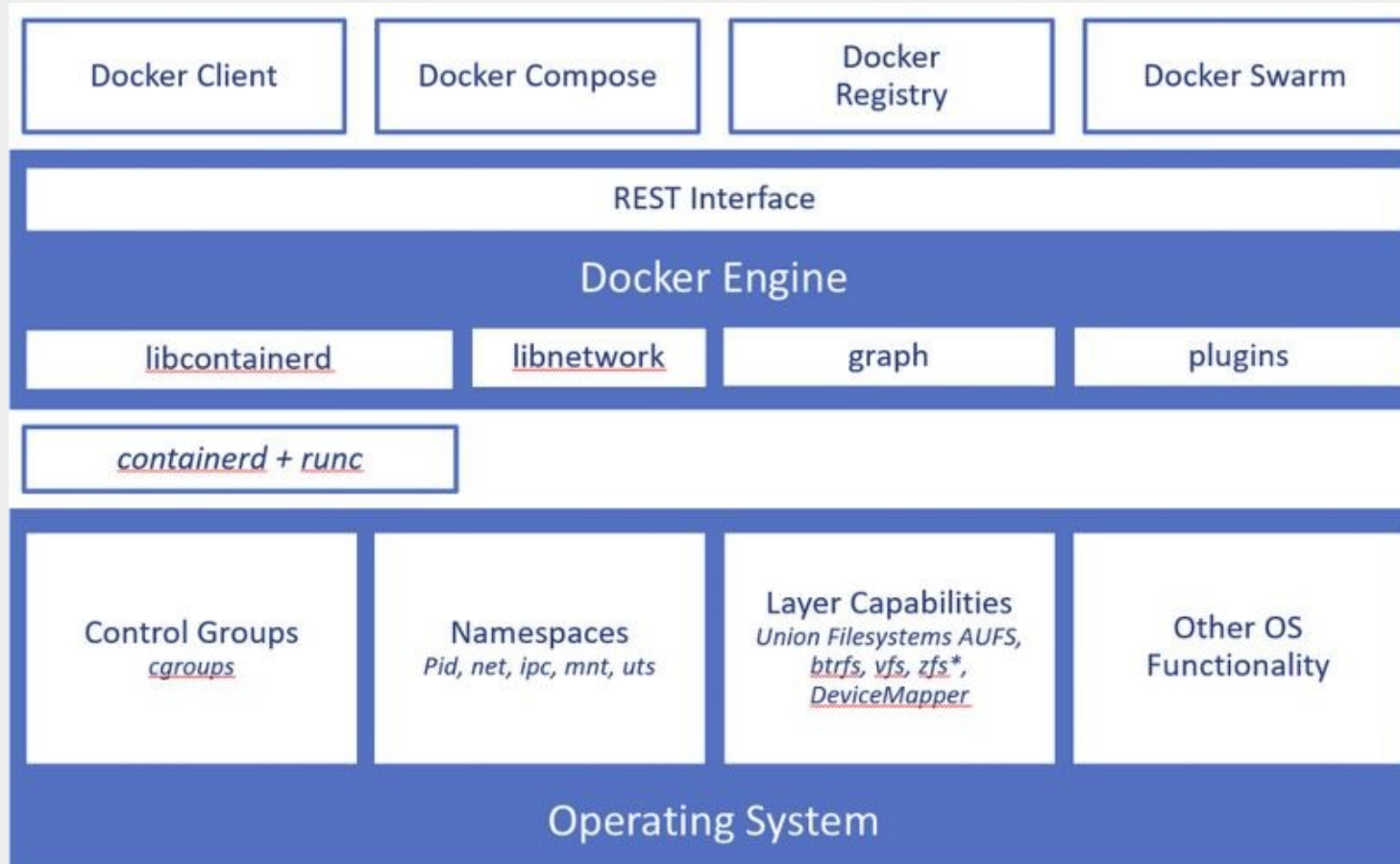
■ Une technologie client-serveur:

- Docker Cli Client
- Dockerd : server d'API REST
- Briques fonctionnelles
 - conteneurs
 - images de conteneurs
 - interfaces réseau
 - Volumes de données



DOCKER

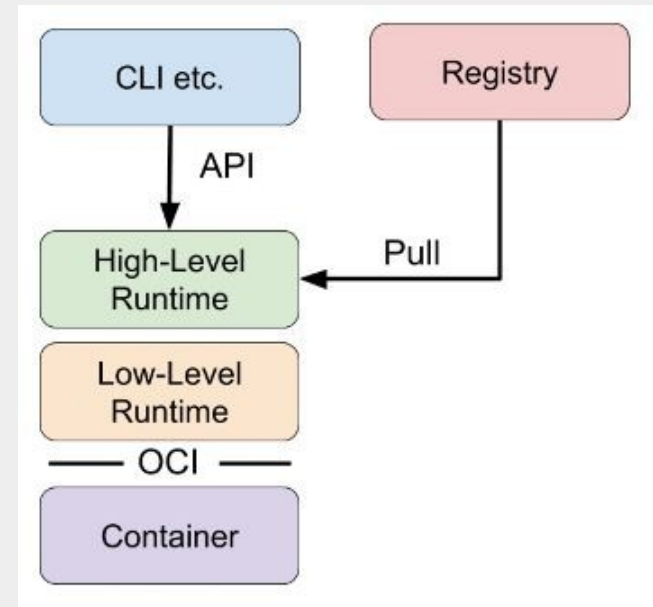
■ Plus précisément:



DOCKER

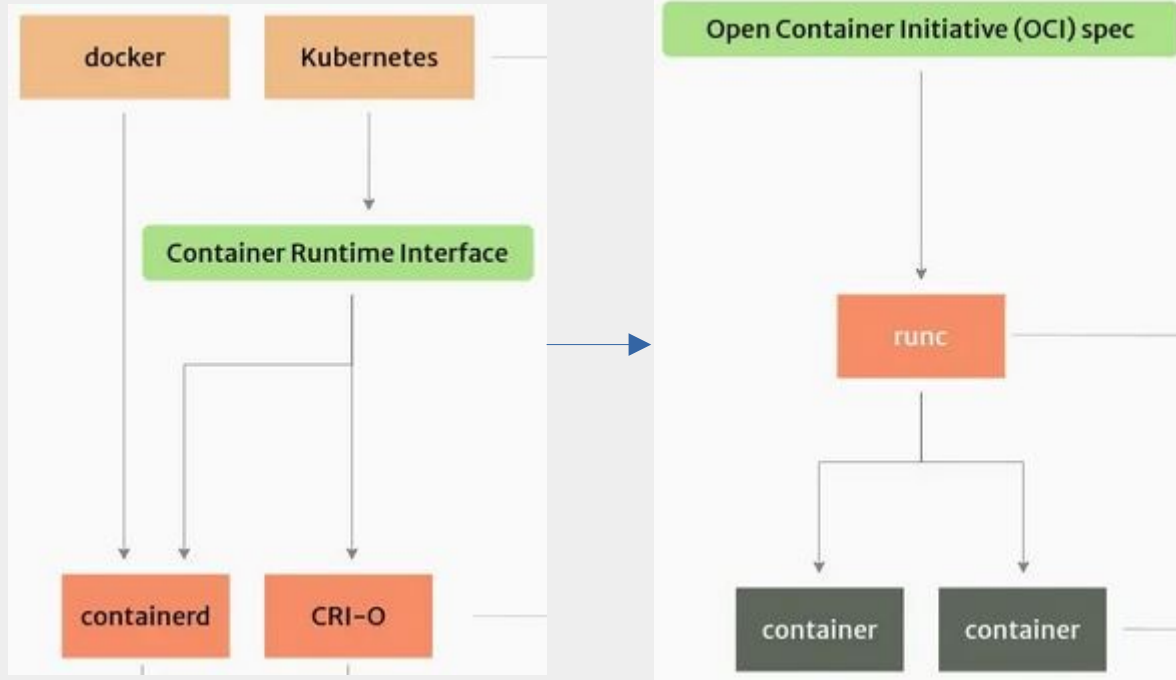
■ responsabilités :

- Dockerd gère
 - les call d'api et
 - le **build des images**
- Containerd (CNCF) gère
 - les push et pull d'image (vers le **registre**)
 - le **stockage** de données
 - les couches **réseaux**
 - l'**API** de création de containers
- Runc gère
 - la manipulation de **namespaces** pour créer les containers



DOCKER

Standards pour la conteneurisation

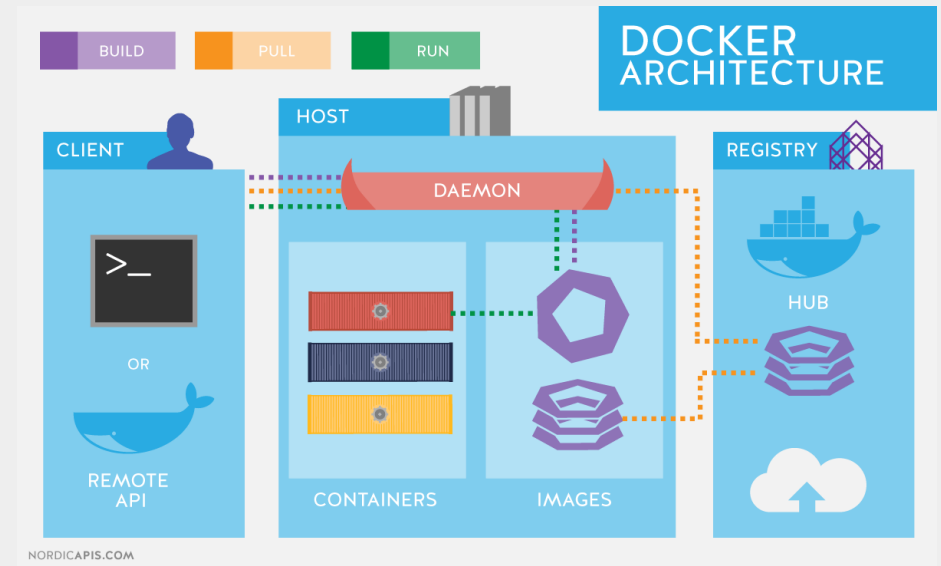


- CRI : standard pour les runtimes de Kubernetes
Containerd est Compatible CRI
- CRI-O : runtime haut niveau de Kubernetes
- OCI : standard pour les containers Linux
Runtime bas niveau
- runc est compatible OCI

Premiers pas

■ Téléchargement d'une image depuis le docker hub (hub.docker.com)

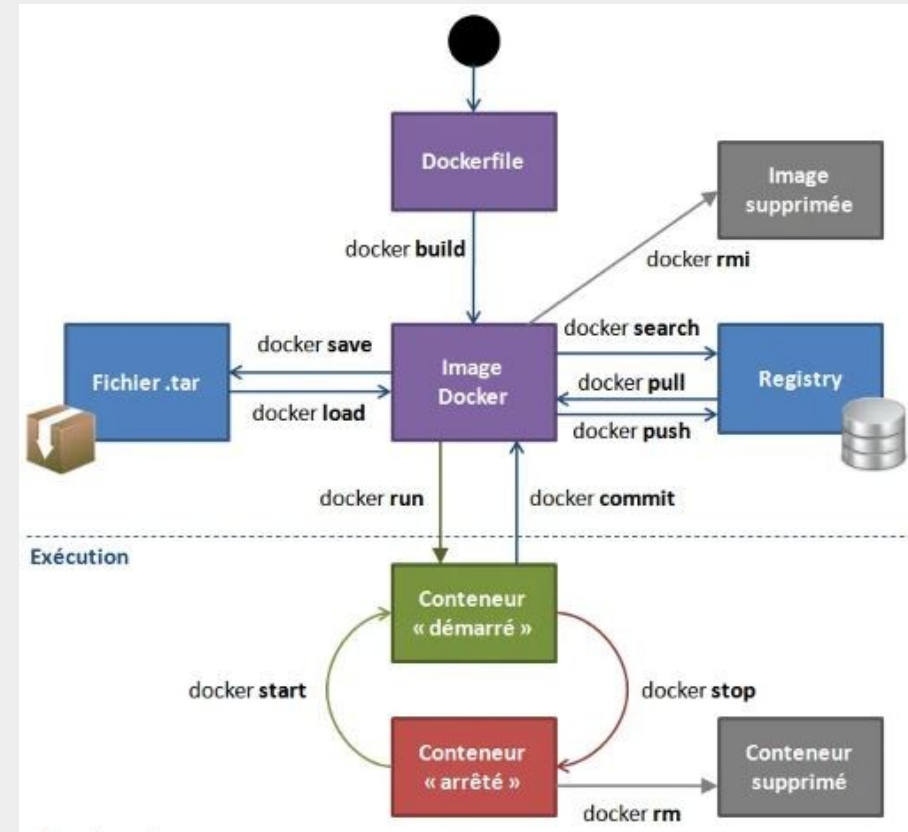
- Une **image** décrit une technologie prête à l'emploi
- Un **tag** précise les paquets de base et la version
- Téléchargement : **docker pull <image>:<tag>**
- Recherche : **docker search <image>**
- Options courantes :
 - * --filter **key=value** (stars, is-official, is-automated)
 - * --format "**Go template : {{ .Name }}**"



Premiers pas

■ Cycle des états d'images

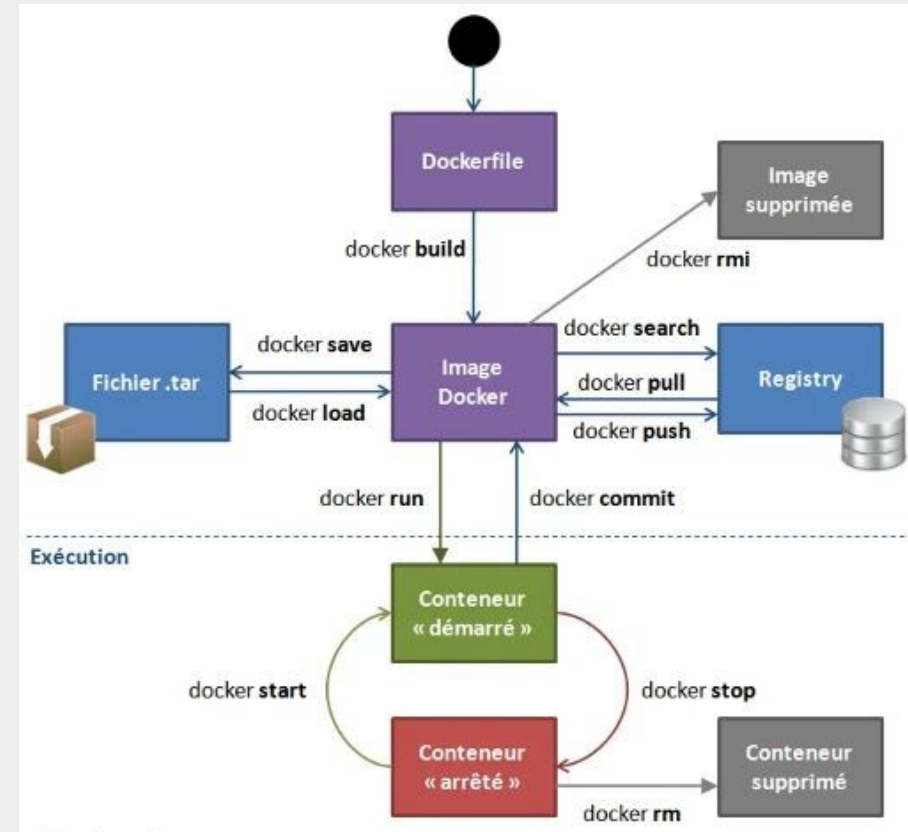
- Lister : **docker image ls** ou **docker images**
- Supprimer une image non utilisée :
docker image rm ou **docker rmi**
- Supprimer les images non utilisées sans tags
docker image prune « dangling image »
- Supprimer les images non utilisées
docker image prune -a



Premiers pas

■ Cycle des états de conteneurs

- Créer un conteneur : **docker create**
- Démarrer / stopper : docker **start / stop / restart**
- Exécuter une commande : docker **exec [cmd]**
- **Lancer : docker run <img:tag> [cmd]**
= create + start + exec
- Supprimer une fois stoppé : **docker rm**
- Supprimer les conteneurs stoppés
docker container prune [-f | --force]



■ Détail du **docker run**

- Création d'un conteneur à partir d'une image avec des **namespaces et des cgroups**
=> puis, exécution d'une commande programmée **dans l'image sous jacente**
=> le processus associé est **isolé** du reste du host par les namespaces...
- Si le processus s'arrête normalement ou en erreur, dans tous les cas le conteneur est stoppé
- On peut le (re)démarrer avec **docker (re)start**
- On peut également, ajouter un autre processus dans le même conteneurs (i.e dans les namespaces)
=> avec **docker exec <ctn> <cmd>**

- Par conséquent, un conteneur est vivant s'il contient un processus vivant :
 - un **conteneur « one shot »** exécute un processus **éphémère** qui doit modifier l'état du host d'un autre conteneur à travers le réseau ou le stockage (volumes)
 - un **conteneur « permanent »** exécute un processus de type **« daemon », un serveur** qui ne retourne pas



■ Introspection

- Voir la liste des conteneurs en exécution : **docker ps**
 - Voir la liste des conteneurs créés : **docker ps -a**
 - Filtrer la liste : **docker ps -f key=value** (id, name, status)
 - Liste d'identifiants (placer en entrée d'une commande): `$(docker ps -a -q)`
-
- Afficher les informations disponibles sur un conteneur : **docker inspect**
 - Afficher seulement certaines informations : `docker inspect --format "Go template : {{ .Name }}"`



■ Format « GO template »

- Technique de requêter « parser » des données,
 - en particulier les obkets json utilisés par Docker

```
# afficher un ou des champs
```

```
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 ' '
```

Premiers pas

■ Options principales d'exécution (docker run)

- **--name** : nom du container, plus compréhensible que l'id
- **--restart** : condition de redémarrage si un erreur survient (**always**, **unless-stopped**)
- Redirection des commandes exécutées dans les containers sur la sortie « **stdout** »
- **--rm** : le container est supprimé quand il est stoppé
- **-e VAR=value** ; créer / mettre à jour une variable d'environnement dans le container
- **--env-file <path>** : injecter les variables du fichier en paramètre dans le container

Premiers pas

■ Conditions d'exécution (docker run | exec)

- **-i** : ajout d'un flux d'entrée « **stdin** » pour exécuter des commandes dans le container
- **-t** : ajout d'un driver « **pseudo-tty** » pour bénéficier des propriétés d'un terminal
- **-it** : **Mode interactif**. Une fois lancé :
 - **Ctrl + P + Q** : revenir sur le shell de l'hôte sans interrompre le terminal container, **docker attach** pour revenir
 - **exit** pour interrompre le terminal container et revenir au shell de l'hôte
- **-d** : **Mode détaché**, lancer le container en arrière plan
 - sans flux d'entrées / sorties
 - le container s'arrête en même temps que son processus interne
- En mode détaché, on affiche la trace du processus du container avec **docker logs**

■ Le namespace « network » et les drivers réseau

- Ce namespace du noyau Linux permet d'obtenir une copie logique (virtuelle) de la pile réseau de l'hôte, en la configurant spécifiquement pour un container. Cela inclus :
 - l'utilisation d'interfaces réseaux, adresses et tables de routages spécifiques
 - des règles de Firewall spécifiques

- Les drivers réseau de docker permettent de spécifier le type d'interfaces réseau à mettre à disposition des containers
 - **driver bridge : par défaut**, réseau virtuel permettant à des containers de communiquer entre eux sur une machine (un démon docker)
 - driver host : utilisation directe d'une interface réseau de l'hôte pour un container
 - driver none : pas de couche réseau, le container ne communique pas
 - **driver overlay : pour un cluster swarm** (cf infra) : agrège les réseaux associés à plusieurs démons docker (machines sur un lan ou un wan)

- Liaisons entre container : `--link` pour le réseau par défaut « `docker0` »
 - Pour faire communiquer plusieurs containers sur `docker0` on peut passer :
 - par les ips, peu pratique car on ne peut pas toujours prévoir ces ips (`--ip <ip_address>` pour une ip fixe)
 - par un alias réseau via l'option `--link <ID | Name | Name : Alias >`
 - Dans les faits, on préférera utiliser un **réseau créé par l'utilisateur** (cf infra)

```
# lien entre le container courant et un container ctn  
docker run ... --link ctn:alias
```

- Liaisons entre containers : réseaux créés par l'utilisateur
 - L'utilitaire **docker network** permet de créer des réseaux virtuels pour faire communiquer les containers
 - Lister : **docker network ls**
 - Créer

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

- Inspecter : **docker network inspect**
- **Sur ces réseaux, le nom des containers est un alias réseau (hostname) par défaut**

■ réseaux créés par l'utilisateur : ajouts de containers

➤ Création

```
docker run ... --network my_network_name
```

➤ Pour un container déjà créé

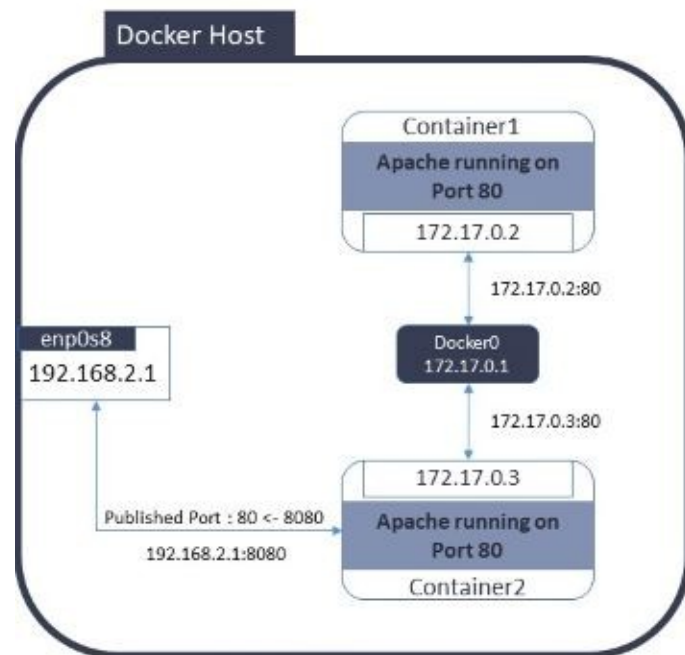
```
docker network connect my_network_name ctn_name
```

- Pour déconnecter : **docker network disconnect**
- Supprimer un ou des réseaux vides de container : **docker network rm <networks>**
- Supprimer tous les réseaux non utilisés : **docker network prune**

■ Publication de ports (-p | --publish)

- L'installation de docker comprend la création d'une interface réseau virtuelle « **docker0** » de type bridge en **172.17.0.1/16**
- Par défaut, un container est créé avec une ip allouée sur ce réseau. Les ports réseaux du container ne sortent pas
- **Publier** un port, c'est rediriger un port de l'interface docker sur une interface de l'hôte

```
# interface:port_externe:port_interne
docker run ... -p enp0s8:8080:80
# toutes les interfaces, choix de port, couche transport
-p 8080-8090:80/tcp
# publier tous les ports internes sur des ports > 10000
-P | --publish-all
```



STOCKAGE

- Copier des fichiers / dossiers entre l'hôte et un container

```
# copie sur le container
```

```
docker cp <src_path> CTN :<dest_path>
```

```
# copie depuis le container
```

```
docker cp CTN :<src_path> <dest_path>
```

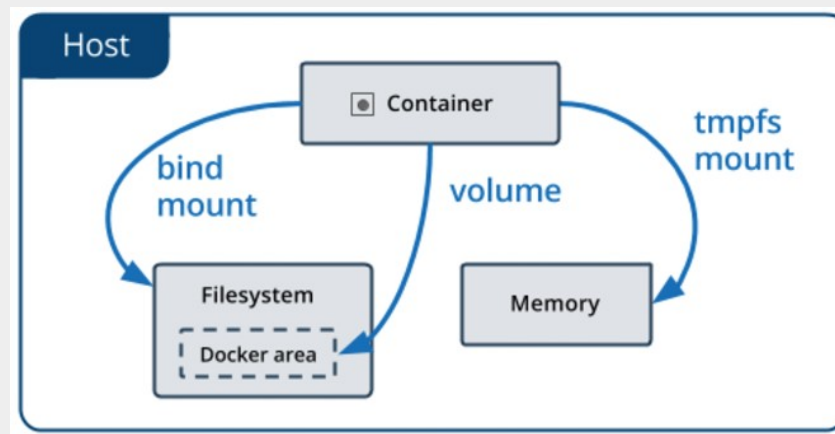
- Sur le container, le répertoire d'accueil est la racine /
- docker cp produit un flux tar, on peut le rediriger sur stdout avec

```
docker cp CTN:<src_path> - | tar x -O
```

STOCKAGE

■ Persistance des données dans les containers

- Toutes les données à l'intérieur d'un container sont supprimées avec le container
- Les **volumes** de données permettent de partager des données entre l'hôte et les containers
 - **bind mounts** : partage entre un **dossier utilisateur** et un dossier dans un ou plusieurs containers
 - **volumes nommés** : partage entre un **dossier géré par docker** et un dossier dans un ou plusieurs containers
 - **tmpfs** : partage temporaire enregistré en RAM, non partageable entre containers
- De fait, les données partagées existent toujours sur l'hôte



■ Drivers de stockage

- Par défaut, docker utilise le driver local **overlay2**, qui enregistre les volumes sur l'hôte en utilisant le système de fichiers **overlayFS du noyau linux > 3.18**

docker info

- 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 la commande linux **mount**, d'une manière ou d'une autre

STOCKAGE

■ docker run --mount vs -v

- L'option **-v** permet de spécifier rapidement une correspondance entre un élément de l'hôte et sa contrepartie sur le container, ainsi que des options de montage

```
# bind mount
docker run ... -v path/to/bind_mount:/path/on/container:opt1,opt2,...
# volume nommé
docker run ... -v volume_name:/path/on/container:ro...
```

- L'option **--mount** permet la même chose, avec une interface plus complète, notamment le réglage d'un **driver spécifique via les paramètres volume-opt**

```
# bind mount
docker run ... --mount src=path/to/bind_mount,dst=/path/on/container,readonly
# volume nommé
docker run ... --mount src=volume_name,dst=/path/on/container,volume-
driver=local,volume-opt=type=nfs...
```


■ Création de volumes nommés

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

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

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

```
docker volume rm <volume_name> | ID
```

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

```
docker volume prune
```

CREER DES IMAGES

■ Retransformer un container en images: docker commit

- On peut **enregistrer l'état d'un container** lancé avec des modifications en tant qu'image
- Afficher les modifications depuis le lancement

```
docker diff <ctn ID | ctn name >
```

- docker commit met le container en pause pour éviter d'enregistrer un état instable et recrée une image à partir de cet état du container

```
docker commit <ctn ID | ctn name > <image name>:<tag>
```

```
# méradonnées du commit
```

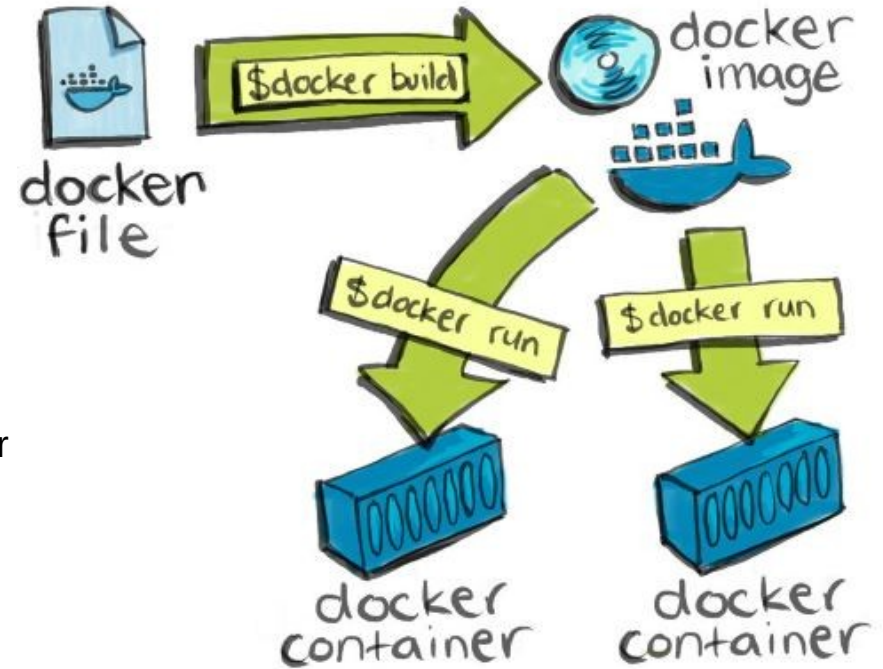
```
-a <author> -m <message>
```

- Le commit ne tient pas compte des données partagées dans des **volumes**

CREER DES IMAGES

■ Le fichier Dockerfile

- Contient des **directives**
- Qui modifient l'état d'une image de base
- De multiples façon :
 - exécution de commandes
 - injection de fichiers / dossier
 - changement d'utilisateur / répertoire home
 - port réseau exposé sur l'interface docker
 - commandes à exécuter au lancement du container
 - ...



CREER DES IMAGES

■ Dockerfile : principales directives

- **FROM <image>:<tag>** : image et tag de base
- **LABEL <key> <value>** : métadonnées de l'image (permet de filtrer les images)
- **COPY [--option] <src> <dest>** : copie des éléments de l'hôte dans l'image
 - utilisation de wildcards (*, ?) pour les éléments source
 - possibilité de changer le propriétaire des éléments sur l'image (--chown)
- **ADD [--option] <src> <dest>** : comme COPY mais gestion de protocoles réseaux pour la source
- **RUN <cmd>** : exécute une commande
- **USER <user[:group]>** : règle l'utilisateur par défaut dans l'image
- **WORKDIR <path>** : répertoire home par défaut de l'image
- **EXPOSE <ports>** : déclare les ports réseau sur lesquels le service installé sur l'image écoute
- **VOLUME <paths>** : création de volumes persistant les chemins renseignés
- **ENTRYPOINT <cmd>** : commande automatiquement lancée avec un container de cette image
- **CMD <cmd>** : termine et dynamise l'ENTRYPOINT, remplaçable par le paramètre de docker run

CREER DES IMAGES

■ Mécanisme du build d'image

- Docker build crée l'image à partir du Dockerfile et du **contexte de build**
 - çàd le répertoire contenant ce dernier

```
docker build -t <image : tag> path/to/context
```

- Le fichier **.dockerignore** permet de ne pas considérer les éléments qu'il contient dans le contexte
 - éléments indisponibles pour les directives COPY et ADD
 - car le contexte est chargé côté démon docker => bande passante
- On peut construire une image sans contexte

```
docker build -t <image : tag> - < /path/to/Dockerfile
```

CREER DES IMAGES

■ Paramétrer un build

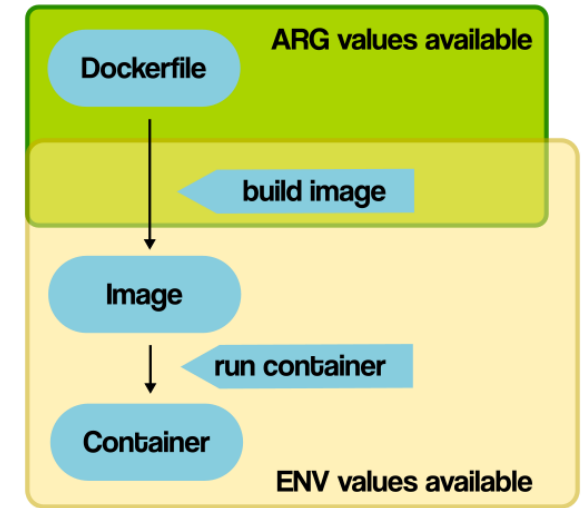
- La directive **ENV** permet de déclarer ou initialiser des variables d'environnement dans l'image

```
ENV VAR
ENV VAR=default
```

- Ces variables peuvent être modifiées au lancement du container
 - docker run ... -e VAR=some_value

- La directive **ARG** permet de déclarer des variables initialisées au moment du build

```
ARG PARAM
...
docker build -t <image : tag> --build-arg PARAM=val
```



CREER DES IMAGES

■ Tester un build

- La directive **HEALTHCHECK** permet d'exécuter une commande au moment du docker run

```
HEALTHCHECK --interval=1s --timeout=30s --retries=30 --start-period=0 CMD <cmd>
```

- Cette commande doit **démontrer** que le service dans le container s'est lancé convenablement
- À l'issue du lancement, un réponse positive du HEALTHCHECK ajoute la valeur « **(healthy)** » au **statut du container** dans la sortie de **docker ps**

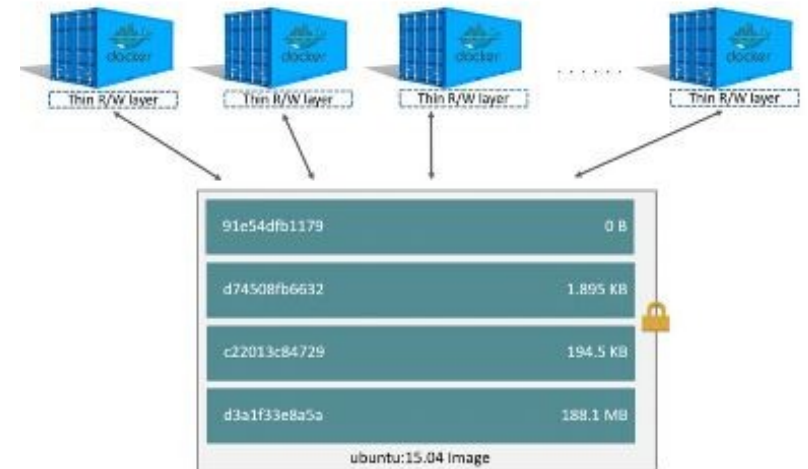
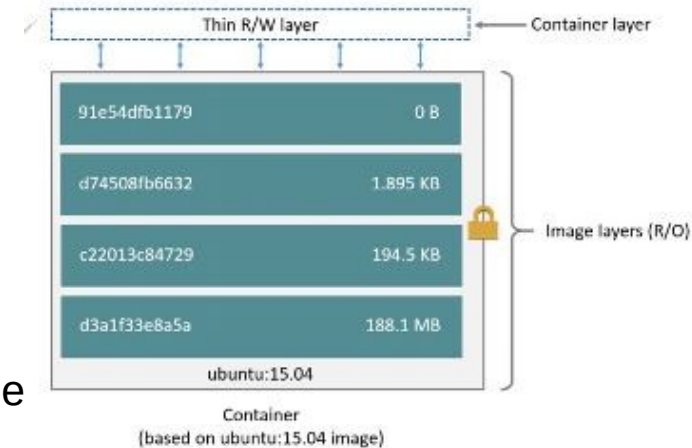
CREER DES IMAGES

■ Composition d'une image

- Chaque directive du Dockerfile crée une couche de système de fichier **en lecture seule**
- Le container rajoute une **fine couche en lecture écriture**
- Les containers **partagent** les couches communes en lecture seule

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

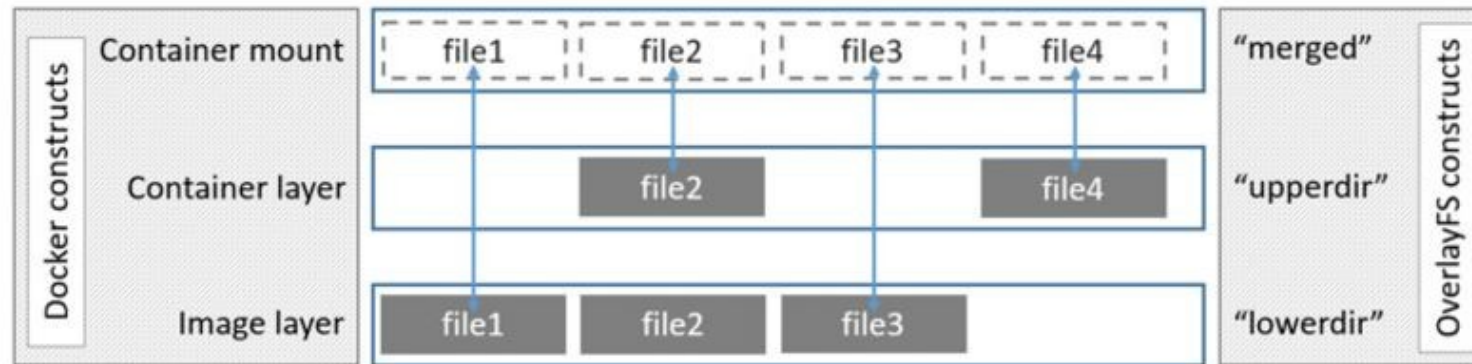
- On peut utiliser **au maximum 127 couches** par image



CREER DES IMAGES

■ Mécanisme « Copy on Write » CoW d'overlayFS

- stratégie de partage et de copie de fichiers pour une efficacité maximale.
- Si un élément existant dans une couche inférieure de l'image est utilisé **en lecture** par une autre couche (y compris la couche container) doit y accéder en lecture, **il n'est pas copié**.
- Le même élément, **modifié** par une autre couche est **copié** sur cette couche et utilisé par les autres couches depuis cette nouvelle couche



CREER DES IMAGES

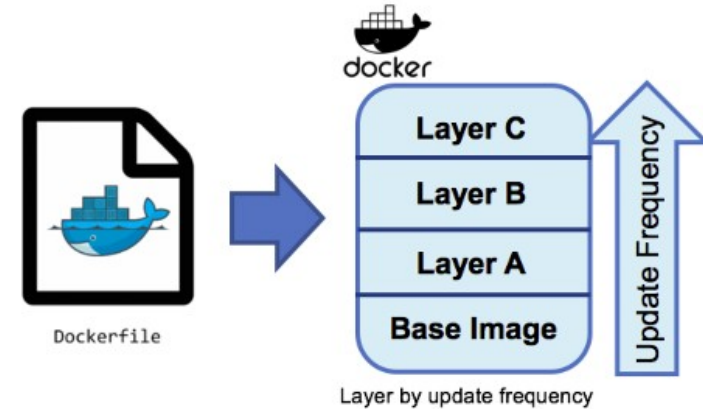
■ Bonnes pratiques

- Les couches doivent être ajoutées par **fréquence de mise à jour croissante**
- Puisqu'on ne peut réutiliser une couche que si les couches parentes n'ont pas changé
- Minimiser le nombre de couches est aussi plus performant

2 couches
RUN cmd1
RUN cmd2

1 couche
**RUN cmd1 && **
cmd2

- Les builds en erreur créent des « **dangling images** » qu'on supprime avec **docker image prune**



■ Définition

- Les Cgroups forment une fonctionnalité du **noyau Linux** qui gère les modalités d'**accès aux ressources système** (RAM, CPU, I/O, Network) pour des ensembles de processus donnés
- Les Cgroups sont organisés en **hierarchies**, avec héritage de propriétés d'un Cgroup parent
- Chaque Cgroup définit les **sous systèmes** sur lesquels il désire réguler l'accès
 - **blkio** : entrées / sortie sur les stockages en mode bloc
 - **memory** : accès RAM
 - **cpu** : % de temps CPU global
 - **cpuset** : sélection spécifique de cpus
 - **net_cls** : gestion des paquets réseau distingués par un identifiant de classe
 - **pids** : gestion du nombre max de process exécutables
- **/sys/fs/cgroup**
- Docker crée un **cgroup spécifique** à la création d'un container

docker stats

■ Limiter la mémoire (docker run)

- On utilise les options
 - **--memory** ou **-m** pour limiter la quantité de ram max autorisée hors SWAP
 - **--memory-swap** pour la même chose SWAP inclus
 - « **memory == memory-swap** » ⇒ **pas de Swap**
- Si le processus dans le container essaie d'allouer plus de RAM qu'autorisé, le **processus est tué** et le container stoppé (sauf avec **--oom-kill-disable**)
- **--memory-reservation** fixe une limite basse en cas de conflits d'accès à la RAM entre containers
- Attention, la RAM autorisée n'est pas la RAM disponible ! Par ex. la commande free ne donne pas la bonne valeur
- Voir dans le container

```
cat /sys/fs/cgroup/memory/memory.limit_in_bytes
```

■ Limiter l'accès aux cpus (docker run)

- **--cpus** : valeur nominale à rapporter au nb total de CPUs disponibles
 - ex : `--cpus='1.5'`, pour 2 cpus \Rightarrow 75 %
- **--cpuset-cpus** : sélection de CPUs autorisés pour le container
 - numérotés à partir de 0 : `--cpuset-cpus=0,3`
 - sélectionner une gamme : `--cpuset-cpus=0-3`
- On peut accéder à ces métriques via l'utilitaire **htop** lancé sur l'hôte

■ Limiter les débits d'entrées / sorties (docker run)

- Il faut d'abord trouver le device sur lequel est monté le container (**/dev/sda** par défaut sur Ubuntu par exemple)
 - **df -h, docker inspect <CTN ID | Name> | grep Device**
- **--device-read-bps et --device-write-bps :**
 - **--device-read-bps /dev/sda:1mb**
- On peut accéder à ces métriques via l'utilitaire **iostat** lancé sur l'hôte

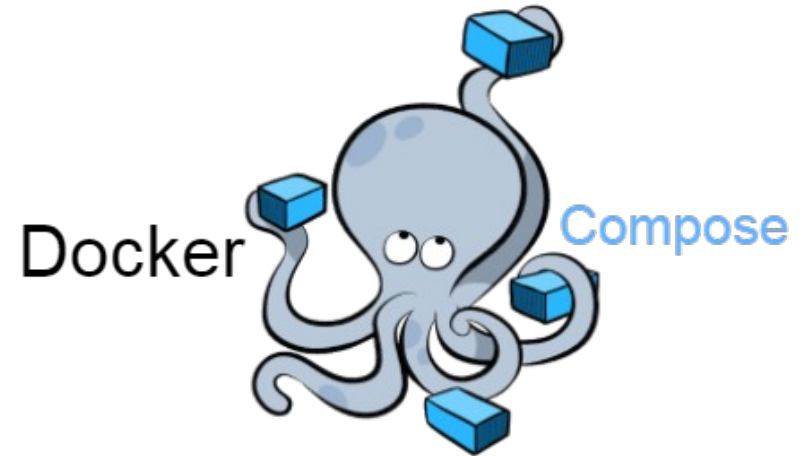
■ Tester les Limites

- Utiliser l'image **polinux/stress**
- **docker exec -it <ID> stress -h**
- **<https://hub.docker.com/r/polinux/stress-ng>**

DOCKER COMPOSE

■ Infrastructure as Code :

- Piloter les commandes docker depuis un **fichier de configuration** au format **YAML**
- Gérer en ligne de commande plusieurs conteneurs comme un ensemble de services inter-connectés
=> **microservices**
- Disponible comme plugin docker



```
sudo apt-get install docker-compose-plugin
```


DOCKER COMPOSE

■ Le 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",
  "other key" : "other value",
  "num" : "3.14"
  "bool" : "false",
  "none" : "null"
}
```

=>

```
---
key: value
other key: other value
num: 3.14
bool: false
none: null
"with quotes": "possible"
```

■ Le format YAML : imbrications

➤ Par rapport à JSON :

- les objets { ... } sont remplacés par une **indentation** de 2 ou 4 caractères
- les listes [...] sont remplacés par une indentation de 2 ou 4 caractères préfixée par « - **[espace]** »
- YAML accepte les listes et objets **JSON** comme valeur de clés

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

DOCKER COMPOSE

■ Correspondances clé / option commandes docker (run, build, network, volume, ...)

```
services:                # service = app containerisée
  app:                   # nom du service (arbitraire)
    container_name:      # --name
    image:               # nom de l'image
    build:               # docker build d'un Dockerfile
      context: .          # chemin du Dockerfile
      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]      # EXT_PORT:INT_PORT ou PORT seul pour exposition
    volumes:             #
      - type: bind        # --mount
        source: .
        target: /
      - /src:/dev:opts    # -v
    networks: [...]      # --net
```

DOCKER COMPOSE



Commandes docker compose principales

- docker compose **up -d** : lancement de la stack (container) dans l'ordre et en mode détaché
- docker compose **down -v** : stopper les container et les réseaux et détacher les volumes
- docker compose **rm -f** : supprimer les containers si stoppés
- docker compose **run <ctn name> <cmd>** : lancer un service standalone avec une commande
- docker compose **logs, ps** : versions agrégées des commandes docker

REGISTRE DOCKER

■ L'image registry

- Le dépôt public **docker hub** est installé par docker
- Avec l'image **registry:2**
- On peut installer son propre registre d'images privées grâce à cette image
- Pour échanger des images avec le registre, on utilise les commandes docker suivantes
 - docker **pull**
 - docker **tag** (ou docker build)
 - docker **push**

