



# Conteneurisation

Présentation: Stéphane Lavirotte  
Auteurs: ... et al\*

(\*) Cours réalisé grâce aux documents de :  
Stéphane Lavirotte, Nikita Rousseau

Mail: Stephane.Lavirotte@univ-cotedazur.fr

Web: <http://stephane.lavirotte.com/>

Université Côte d'Azur



# Introduction

**Virtualisation vs Conteneurisation**



## ✓ Définition de la virtualisation

- Exécuter plusieurs systèmes d'exploitation et/ou applications isolés sur un même hôte physique

## ✓ Deux manières de réaliser cette virtualisation

- Machine virtuelle (avec OS et émulation machine):

- Isoler un système d'exploitation sur une machine hôte
- Consiste à émuler le matériel d'une machine (stockage, carte graphique, réseau, ...)
- Pour installer un système d'exploitation complet
- Le partage de l'architecture processeur permet d'accélérer la virtualisation
- Ex: Qemu, VirtualBox, VMWare, ...

- Virtualisation (sans OS supplémentaire) => Conteneurisation

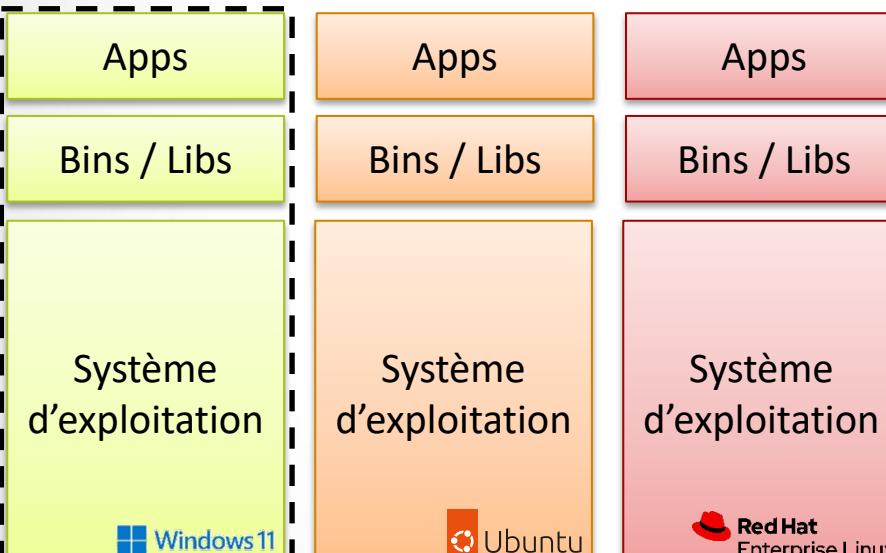
- Isoler une/des processus sur une machine hôte
- Utilise des fonctionnalités du noyau pour limiter le champ d'exécution d'un(de) processus donné(s)
- Ex: docker, LXC, Podman, runC, containerd, ...



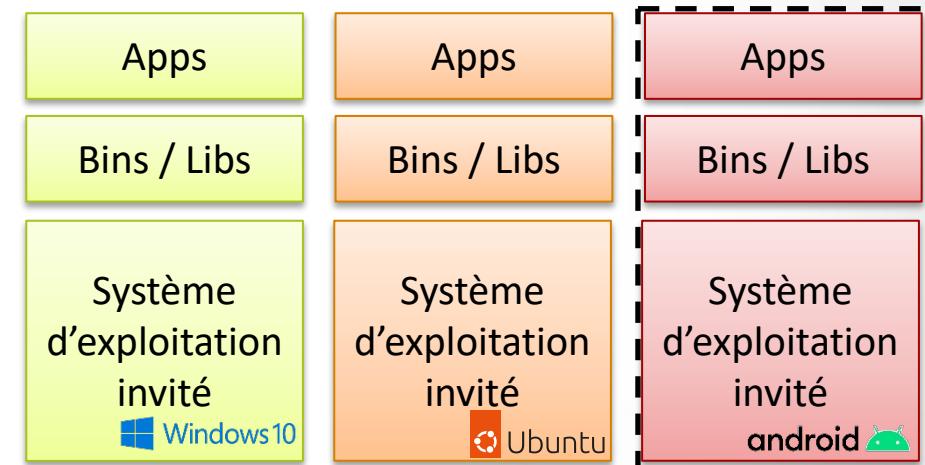
# Virtualisation et Hyperviseur

## ✓ Hyperviseur: moniteur de machine virtuelle

VM – Type 1 (ou niveau 1)



Type 2 (ou niveau 2)



Infrastructure / Matériel (*Hardware*)

Infrastructure / Matériel (*Hardware*)



# Virtualisation vs Conteneurisation

## ✓ Machine Virtuelle

VM

Apps

Apps

Apps

Bins / Libs

Bins / Libs

Bins / Libs

Système  
d'exploitation  
invité  
 Windows 10

Système  
d'exploitation  
invité  
 Ubuntu

Système  
d'exploitation  
invité  
 android

Hyperviseur Type 2 VirtualBox

Système d'Exploitation Hôte Windows 11

Infrastructure / Matériel (*Hardware*)

## ✓ Conteneurisation

- Légèreté et isolation pour l'exécution de processus

Conteneur

Apps

Apps

Bins / Libs

Bins / Libs

Moteur de  
Conteneur



Système d'Exploitation Hôte Linux

Infrastructure / Matériel (*Hardware*)



# Virtualisation vs Conteneurisation

## ✓ Une machine virtuelle permet, via un hyperviseur:

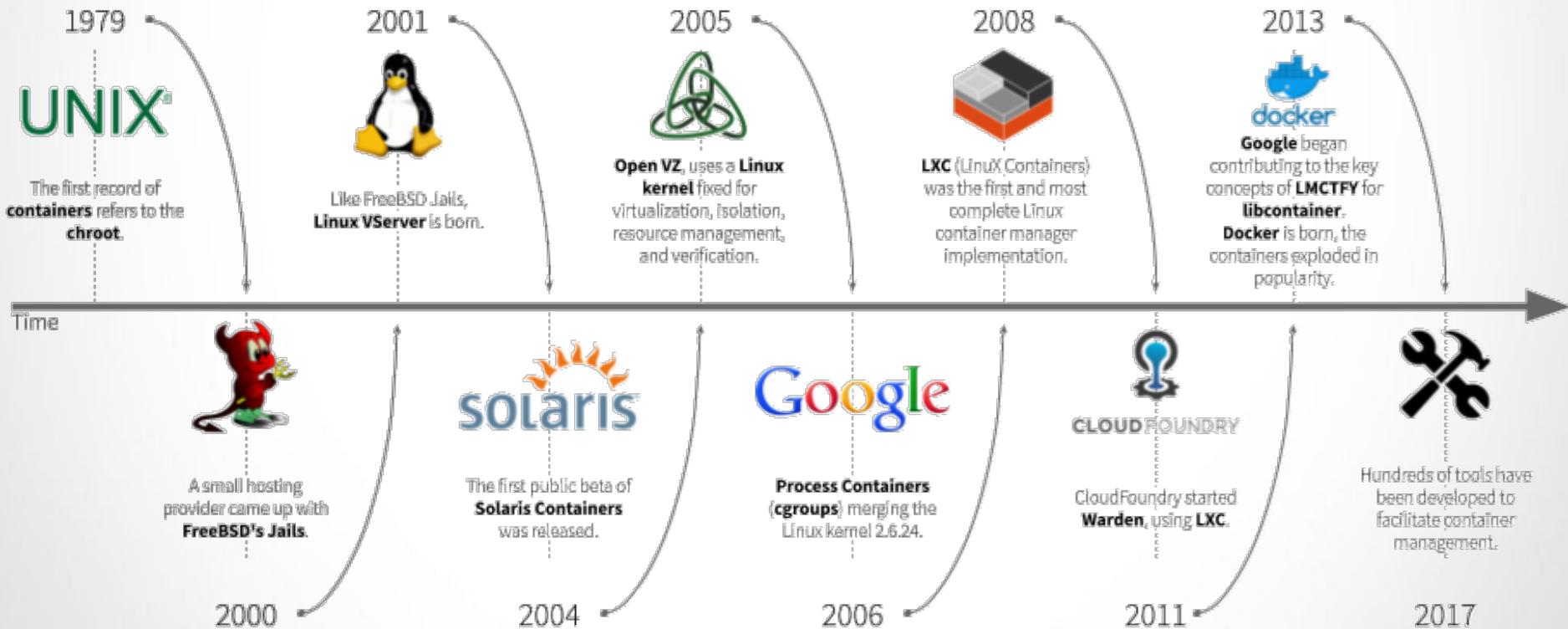
- de simuler une ou plusieurs machines physiques
- de les exécuter sous forme de machines virtuelles (VM)
- Ces VM intègrent elles-mêmes un OS sur lequel des applications sont exécutées
- Ne pourrait-on pas se passer d'un deuxième OS ?

## ✓ Un Conteneur Logiciel:

- Grande légèreté
- Fait directement appel à l'OS (noyau) de sa machine hôte pour réaliser ses appels système et exécuter ses applications
- Isolation des applications, de leurs dépendances (ex: bibliothèques) et de leur environnement d'exécution
- Evite d'avoir à gérer la cohérence globale de toutes les dépendances sur un même système (gain de temps/productivité)



# Histoire de la Technologie des Conteneurs



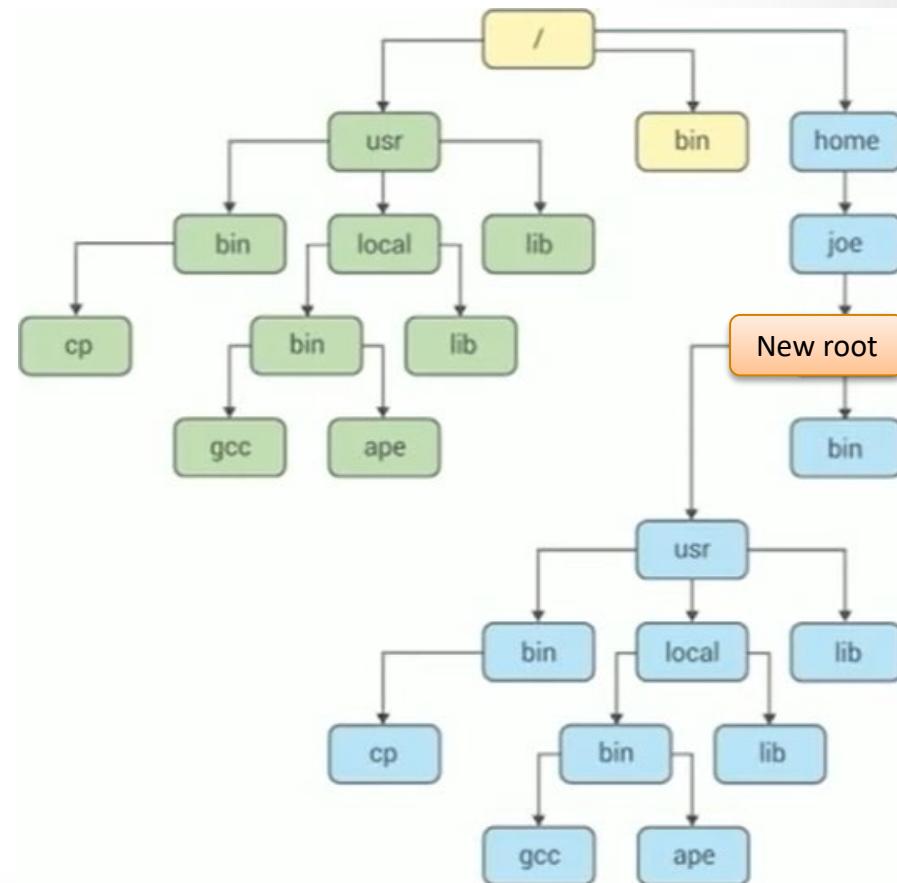


# Besoins pour mettre en œuvre l'isolation des Conteneurs

- ✓ chroot
  - Changer de racine de répertoire (isolation du système de fichiers)
- ✓ Namespace
  - Donner une vue sur une ressource globale partagée (entre tous les processus)
- ✓ Cgroups
  - Création de groupes de processus et gestion des droits
- ✓ Seccomp
  - Sécurité autour des appels systèmes
- ✓ Ces fonctionnalités sont fournies par le noyau (via des appels systèmes):
  - Syscall: API des fonctionnalités mises à disposition par le noyau
  - Actuellement (6.3): 397 fonctions sur x86-64 (437 sur x86-32)
  - Bascule entre le « User Space » et le « Kernel Space »
  - Cf slide 14 et 15 du cours « Processus et Threads »



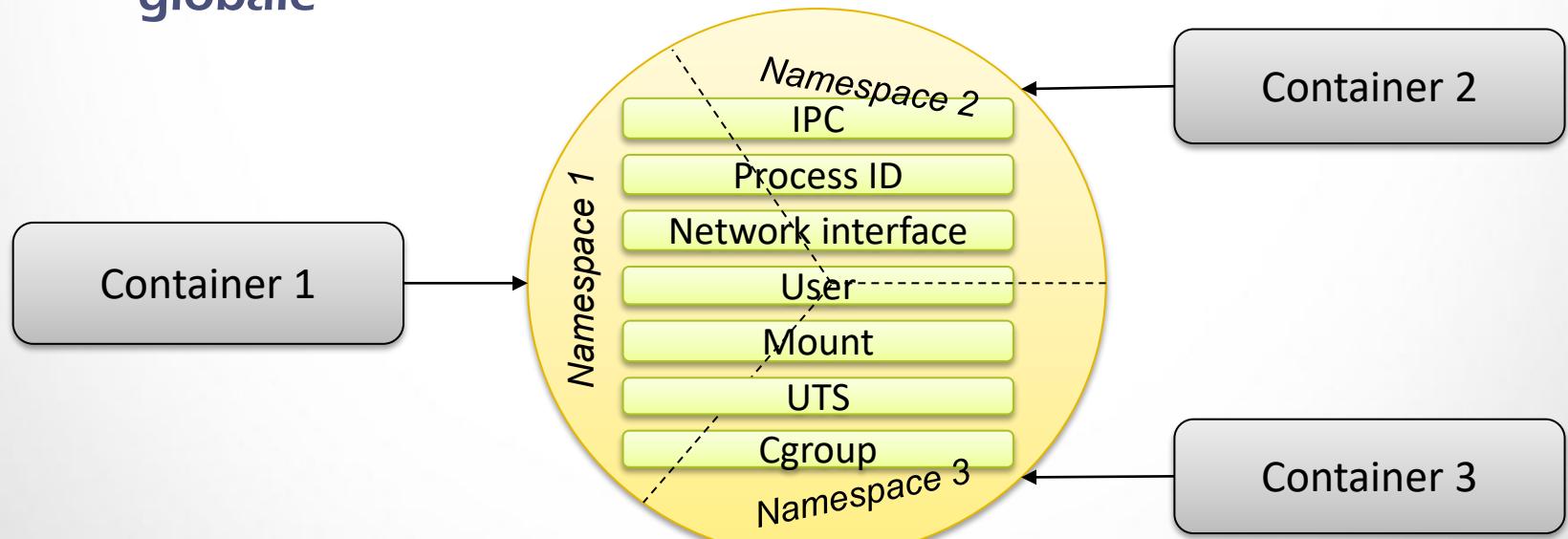
- ✓ **Changer de racine de système de fichiers**
  - Le plus vieux mécanisme (1979)
  - N'avoir accès qu'à une partie du système de fichiers (FS) pour un processus
  - Si une faille permet un accès à une application
    - Application sans chroot, accès à tous le FS
    - Application avec chroot, accès uniquement à une portion de FS
  - Ex: Un serveur Web peut faire un chroot sur l'arborescence des fichiers qu'il expose (et pas sur tout le FS depuis la racine)





# Container Namespace

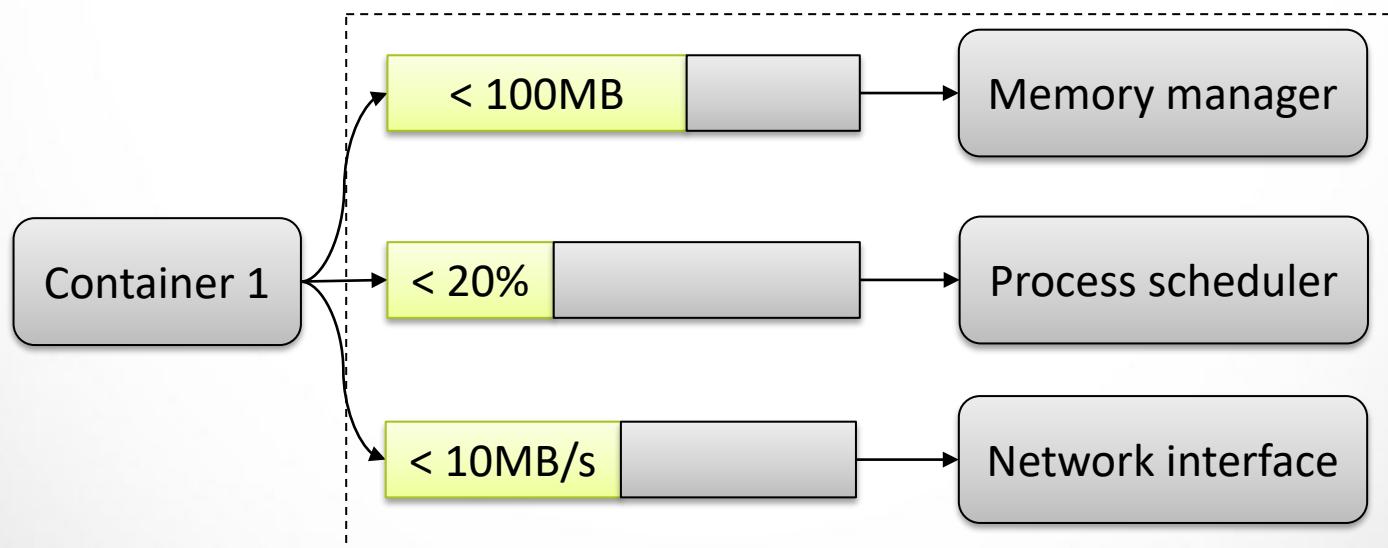
- ✓ Container Namespace: « *What you can see* »
  - Enveloppe une ressource système globale
  - Dans une abstraction qui donne l'impression aux processus d'un *Namespace*
  - Qu'ils possèdent leur propre instance isolée de la ressource globale





## ✓ Control Groups: « *What you can use* »

- Limiter les ressources utilisées par un conteneur (groupe de processus)
- Contrôle fin de l'allocation, de la priorisation, du refus et de la gestion des ressources du système



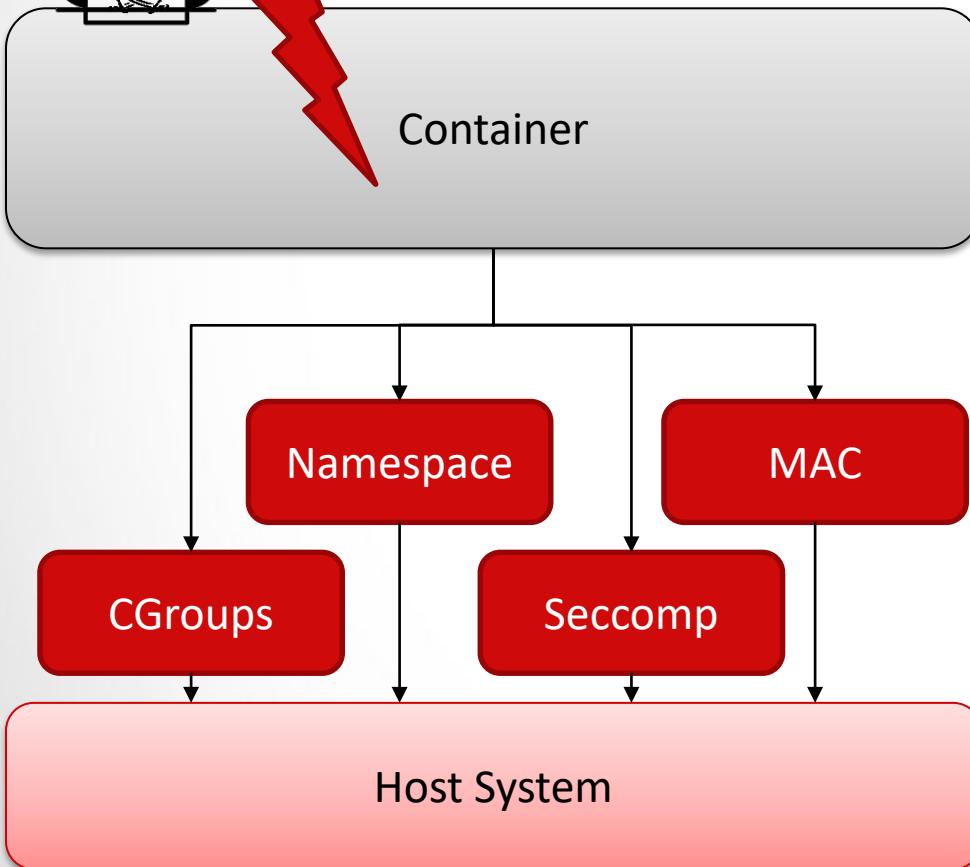


# Cgroups controllers

Cgroups controllers	V1	V2	Description
cpu (and V1 cpacct)	2.6.24	4.15	<i>Guaranteed a minimum number of "CPU shares"</i>
cpuset	2.6.24	5.0	<i>Bind the processes in a cgroup to a specified set of CPUs</i>
freezer	2.6.28	5.2	<i>Suspend and restore (resume) all processes in a cgroup</i>
hugetlb		5.6	<i>Supports limiting the use of huge pages by cgroup</i>
io (V1 blkio)	2.6.33	4.5	<i>Controls and limits access to specified block devices by applying IO control</i>
memory	2.6.25	4.5	<i>Report and limit process memory, kernel memory, and swap</i>
perf_event	2.6.39	4.11	<i>Allows perf monitoring of the set of processes grouped</i>
pids	4.3	4.5	<i>Limits the number of process created</i>
rdma	4.11	4.11	<i>Limits the use of RDMA/IB- specific resources</i>
devices	2.6.26		<i>Controls devices creation (mknod) and access</i>
net_cls	2.6.29		<i>Places a classid on network packets</i>
net_prio	3.3		<i>Priorities to be specified, per network interface</i>
hugetlb	3.5		<i>Limits the use of huge pages</i>



- ✓ **Filtre devant tous les syscall**
  - Bloque à l'entrée de l'appel système
  - Réalise à base de BPF (Berkeley Packet Filters)
- ✓ **Utilisé à partir de libseccomp**
  - Autoriser uniquement les appels systèmes que l'on souhaite
- ✓ **Permet de rendre moins vulnérable le système**

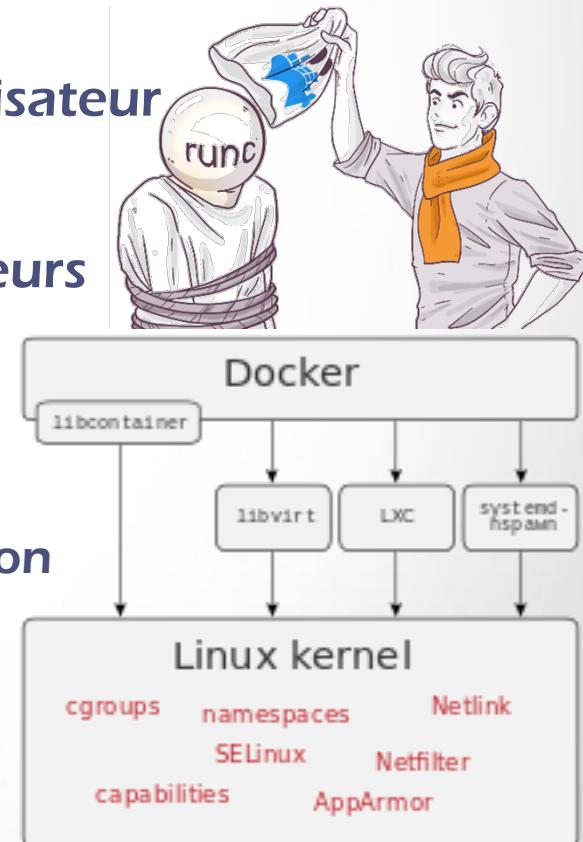


- ✓ **Cgroups: limite l'accès aux ressources**
- ✓ **Namespace :virtualise l'accès aux ressources**
- ✓ **Seccomp: limite les accès aux appels systèmes**
- ✓ **Politique de contrôle d'accès obligatoire (MAC: Mandatory Access Control, i.e. type AppArmor)**
- ✓ **Conteneurs sans racines**



# Interfaces utilisateur

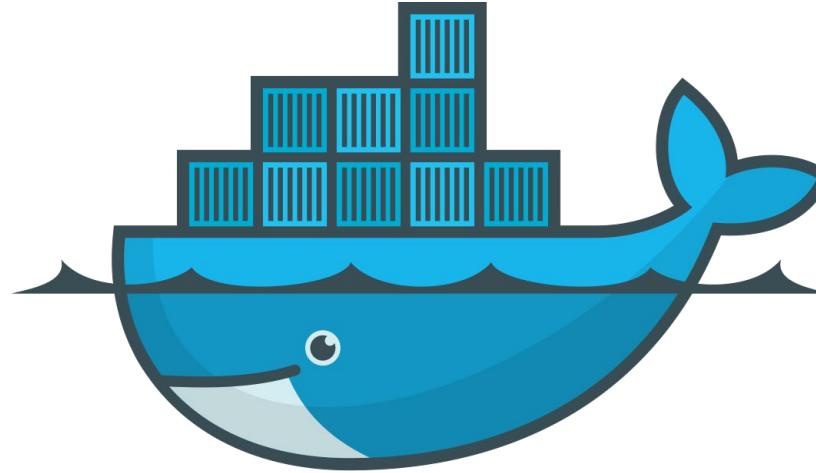
- ✓ Ces différentes fonctionnalités (`chroot`, `namespace`, `cgroups`, `seccomp`, ...)
  - Sont offertes par le noyau
  - Accessibles par `/sys` depuis l'espace utilisateur
- ✓ Besoin d'interfaces utilisateur
  - Pour faciliter la manipulation des conteneurs
  - Différentes interfaces
    - LXC: API bas niveau
    - LXD: API haut niveau
    - Podman: moteur de conteneur sans daemon
    - CRI-O: optimisé pour Kubernetes
    - containerd: std indus simple et robuste
    - Docker: utilise runc (initialement LXC)





# Open Container Initiative (OCI)

- ✓ **OCI: Projet de la Linux Foundation**
- ✓ **Concevoir des normes ouvertes pour la virtualisation par isolation au niveau des systèmes d'exploitation**
- ✓ **Deux spécifications (toujours en évolution):**
  - Runtime-spec: spécification de l'exécution
    - RunC (utilisé par Docker): Go: 80%, Shell: 15%, C: 5%
      - CLI pour la création et l'exécution de conteneurs sur Linux conformément à la spécification OCI
    - RailCar : Rust: 99%, Shell: 1%
      - Implémentation Rust de la spécification OCI runtime
    - Image-spec: spécification du format de sauvegarde
- ✓ **Plus de 13 implémentations différentes des conteneurs en 2016**



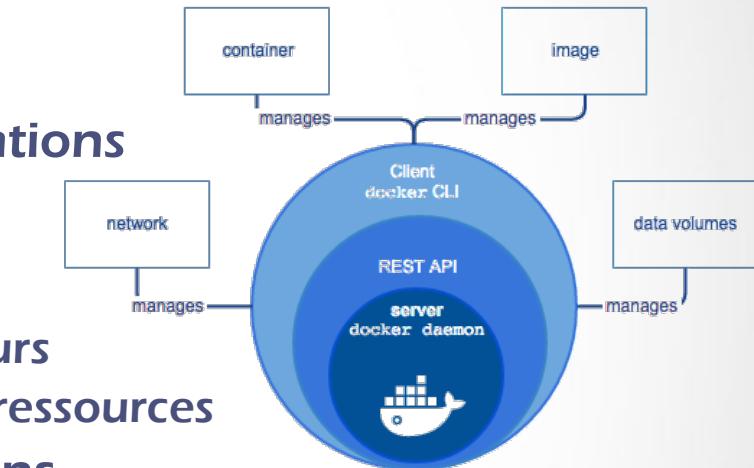
# Conteneurs

Cas d'usage avec

**docker**



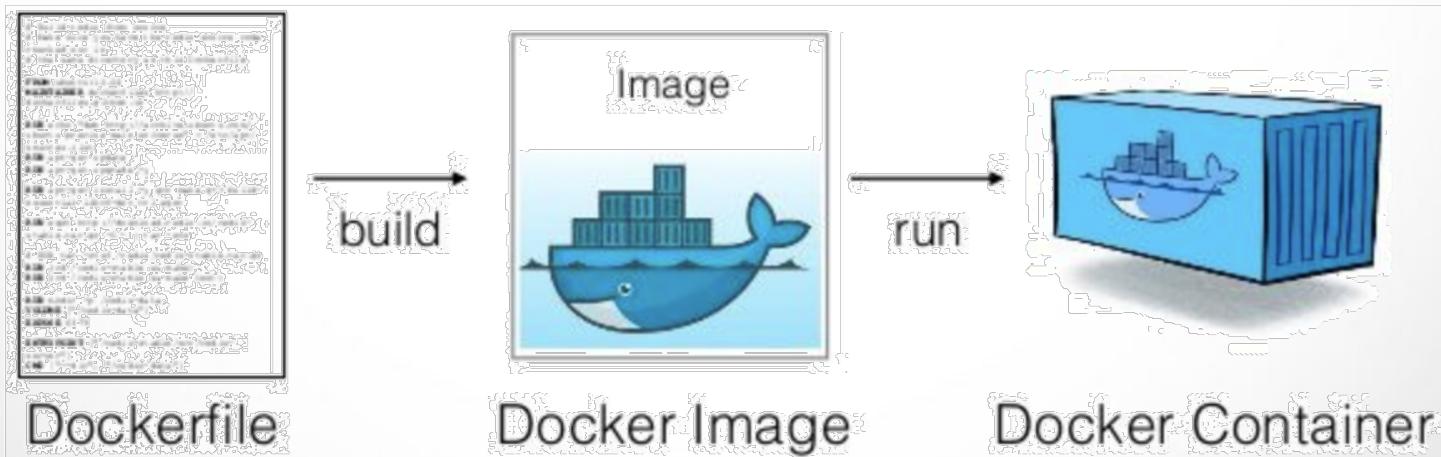
- ✓ **Plateforme de conteneurisation**
  - Créer, déployer, exécuter des applications
- ✓ **Basé sur plusieurs éléments:**
  - **Moteur Docker: composant principal**
    - Permet de créer et gérer les conteneurs
    - Utilise le noyau Linux pour isoler les ressources
  - **Images Docker: modèles d'applications**
    - Système de fichiers
    - Contient le code, les bibliothèques, les fichiers de configuration, les dépendances
  - **Conteneurs Docker: instance en cours d'exécution d'une image**
    - Crée à partir d'une image
    - Léger et démarrage rapide (plus rapide qu'une VM)
  - **Registres Docker: dépôt où stocker les images Docker**
    - Public ou privé
  - **API: manipulation des images, conteneurs, ...**





# Images vs Conteneurs

- ✓ Dockerfile (**patron/modèle**):
  - Instructions pour construire une image
- ✓ Image (**classe**):
  - Système de fichiers constitué d'un ensemble de strates (*layers*)
- ✓ Conteneur (**instance**):
  - Une instance en cours d'exécution d'une image

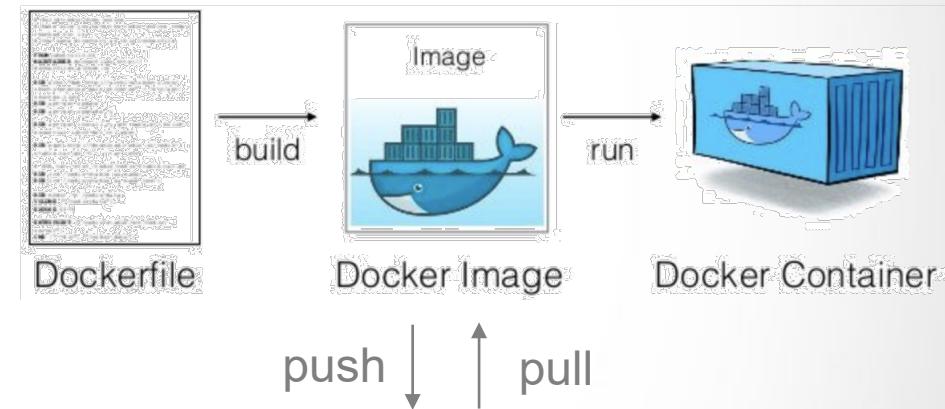




# Images et Registres

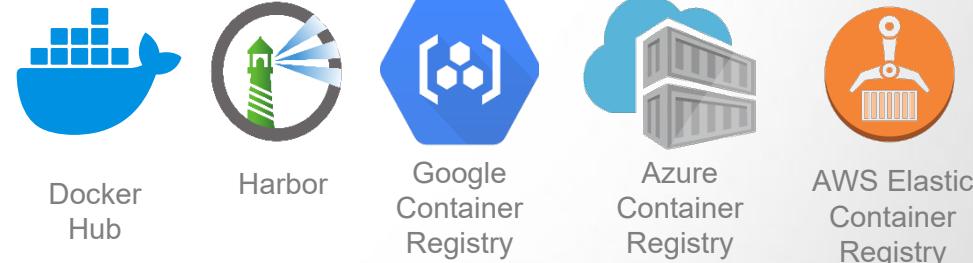
## ✓ Registre local

- Stockage de toutes les images en cache/construites
- Lister: docker images



## ✓ Registre externe

- Permet de partager les images
- Dépôts publics/privés
- Publier: docker push
- Obtenir: docker pull





✓ **CLI (Command Line Interface)**

- Permet de piloter le moteur
- **docker [action] [arguments]**

✓ **Actions principales:**

- build: **construit une image depuis un fichier Dockerfile**
- exec: **exécute une commande dans un conteneur en cours d'exécution**
- image: **gestion des images**
- images : **liste les images**
- ps: **liste les conteneurs**
- pull: **télécharge une image depuis un registry**
- push: **télécharge une image vers un registry**
- restart: **redémarre un conteneur**
- rm: **supprime un ou plusieurs conteneurs**
- rmi: **supprime une ou plusieurs images**
- run: **créé et exécute un nouveau conteneur à partir d'une image**
- start: **démarre un ou plus conteneurs stoppés**
- stop: **arrête un ou plusieurs conteneurs**
- tag: **donne un identifiant spécifique à l'image**



# Définition de Docker

- ✓ Docker est une plateforme qui permet d'empaqueter nos applications dans des exécutables déployables - appelés conteneurs, avec toutes les bibliothèques et dépendances nécessaires au système d'exploitation
  
- ✓ Conséquence:
  - Nécessite d'embarquer dans le conteneur toutes les dépendances (bibliothèque, variable d'environnement, ...)
  - i.e. nécessite d'embarquer les bibliothèques partagées dans le conteneur
    - ldd pour savoir quelles bibliothèques partagées intégrer dans le conteneur
    - Perd une partie de l'intérêt des bibliothèques partagées

# Mise en œuvre de Docker sur d'autres OS que Linux

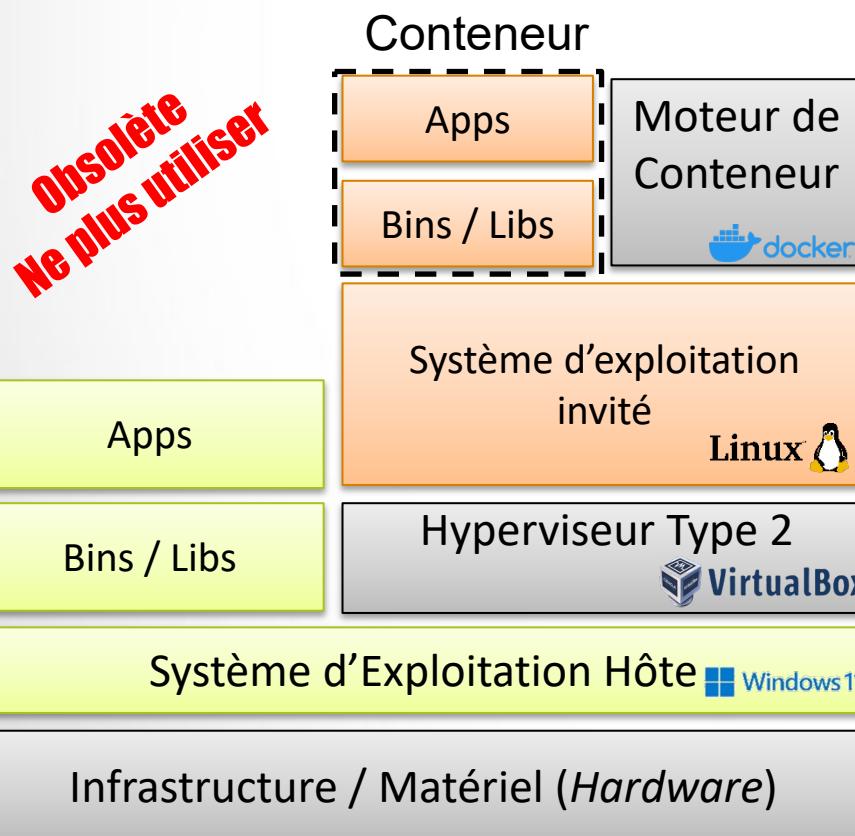
- ✓ **La conteneurisation repose sur des mécanismes proposés par le noyau**
  - Aujourd'hui, seul le noyau Linux propose ces mécanismes
  - Mais pas le noyau Windows NT ou le noyau Mac OS...
  
- ✓ **Comment cela peut fonctionner sous d'autres OS que Linux ?...**
  - Docker Desktop sous Windows ou Mac



# Illustration de Docker sur Windows

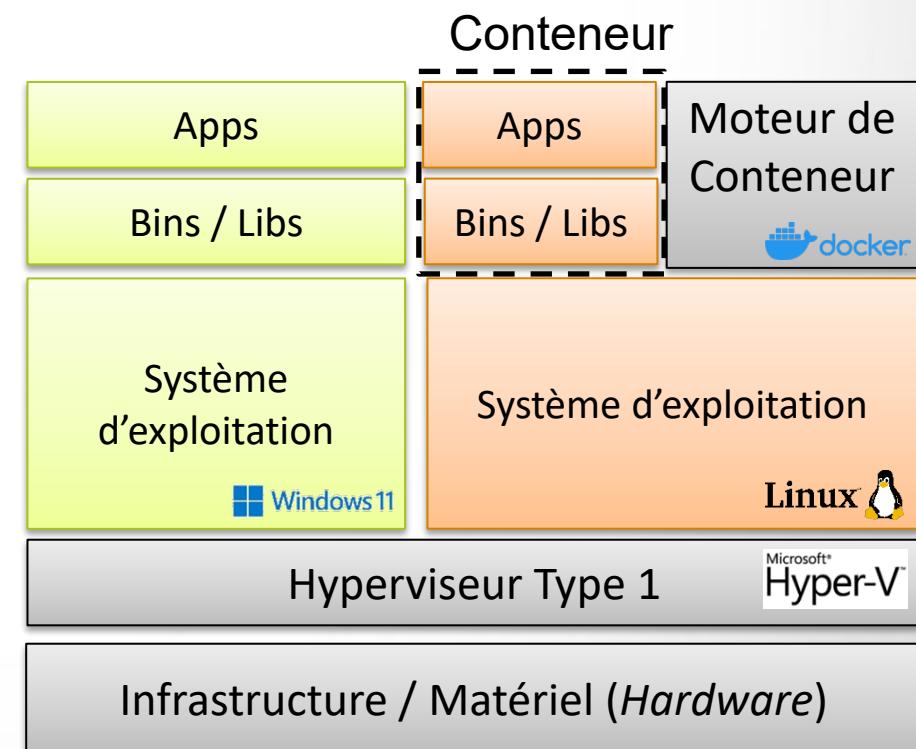
## ✓ Docker toolbox

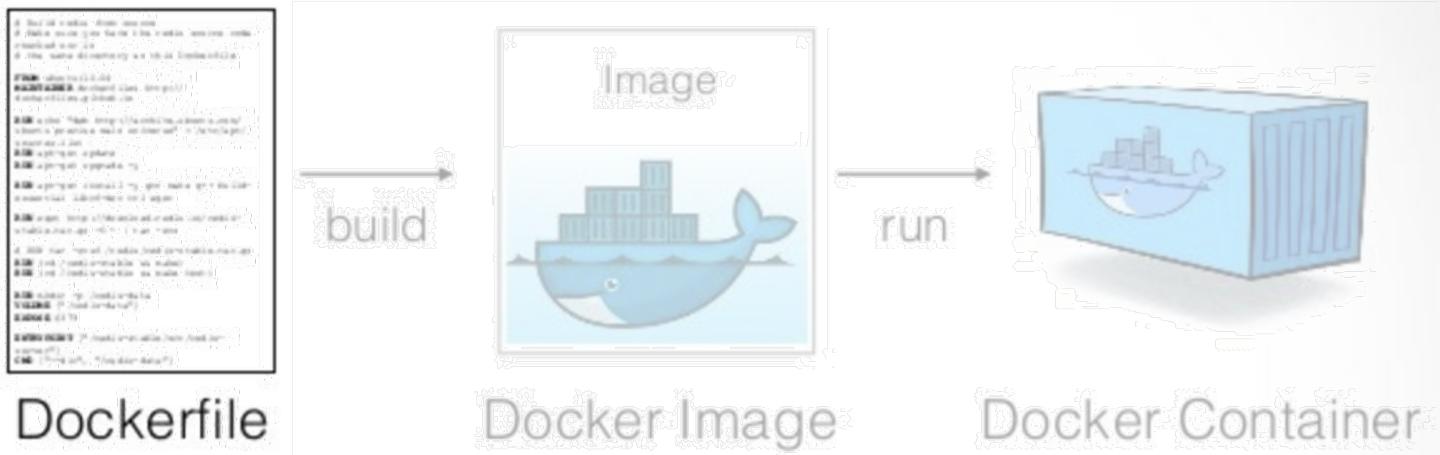
- Utilise VirtualBox



## ✓ Docker desktop

- Utilise WSL





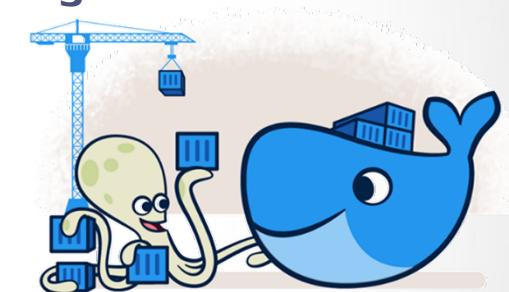
# Dockerfile

Patron/Modèle pour construire une image



## ✓ Dockerfile

- Fichier décrivant les étapes de construction d'une image docker
- Constituée des éléments suivants
  - # Comment
  - INSTRUCTION arguments
- [Documentation de Référence](#)



## ✓ Crédit d'une image à partir d'une image existante:

- FROM <image de base>
- Initialisation de la construction (obligatoire / « première instruction »)
- scratch, alpine, busybox, debian, ubuntu, ...

## ✓ Commande docker pour construire l'image

- docker build [<options>] <path> | <url> | -
- Exemples:
  - docker build . | docker build - < Dockerfile
  - docker build -t [<repository>/]<name>[:<tag>] .



# Instructions Dockerfile

## 1/3

- ✓ FROM [option] <name>[:<tags>] [AS <name>]
  - **Initialise la construction à partir d'une image de base**
- ✓ ENV <key>=<value>
  - **Définit des variables d'environnement pour le container**
- ✓ ARG <var>[=<value>]
  - **Définit des variables pour la construction (peut être modifié lors du build)**
- ✓ WORKDIR <path>
  - **Equivalent à cd (avec création si nécessaire)**
  - **Commandes successives relatives à ce répertoire**
- ✓ RUN <cmd args> | RUN ["cmd", "arg1", "arg2"]
  - **Exécute une commande dans un Shell Unix**
- ✓ ADD ou COPY [options] <src> <dest>
  - **Ajoute des fichiers locaux à l'host dans l'image**



# Instructions Dockerfile

## 2/3

### ✓ Point d'entrée / commande

- Définir la commande à lancer au démarrage du conteneur
- CMD **et/ou** ENTRYPOINT
  - [Comprendre l'interaction entre CMD et ENTRYPOINT](#)

### ✓ CMD **et** ENTRYPOINT

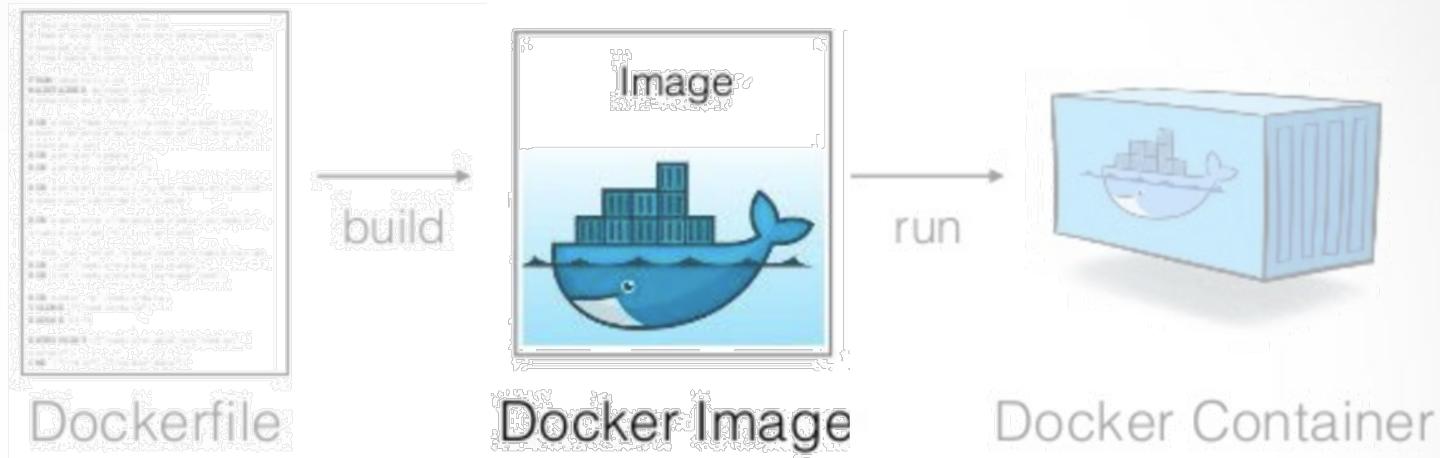
- CMD ["cmd", "arg1", "arg2"]
  - Spécifie la commande à lancer au démarrage du container
- CMD ["arg1", "arg2"]
  - Spécifie les options à ENTRYPOINT
- ENTRYPOINT ["cmd"]
  - Spécifie la commande principale à lancer au démarrage du container



# Instructions Dockerfile

## 3/3

- ✓ LABEL <key>=<value> ...
  - Ajoute des métadonnées à l'image
- ✓ USER <user>[:<group>]
  - Définit l'utilisateur courant (change de root)
- ✓ EXPOSE port[/protocol]
  - Port sur lequel le container sera à l'écoute des connexions
- ✓ VOLUME ["path"]
  - Crée un point de montage
- ✓ STOPSIGNAL <signal>
  - Définit le signal qui sera envoyé au conteneur pour le quitter
- ✓ HEALTHCHECK [options] CMD <command> | NONE
  - Vérifie l'état du conteneur en exécutant une commande à l'intérieur du conteneur



# Docker image

Système de fichiers constitué de states  
Plus quelques éléments...

Artefact de construction



# Image == Layer

## ✓ Layer = Image

- Chaque *layer* est une image, avec un tag autogénéré
- Chaque *layer* stocke les modifications par rapport à l'image sur laquelle il est basé
- Chaque instruction dans un Dockerfile résulte en un *layer* (sauf pour les constructions en multi-étapes)
- Les *layers* sont utilisés pour éviter les informations redondantes
- Les *layers* permettent de faire une construction incrémentale (comme pour un Makefile, on ne reconstruit que ce qui est nécessaire)

## ✓ Une terminologie un peu confuse

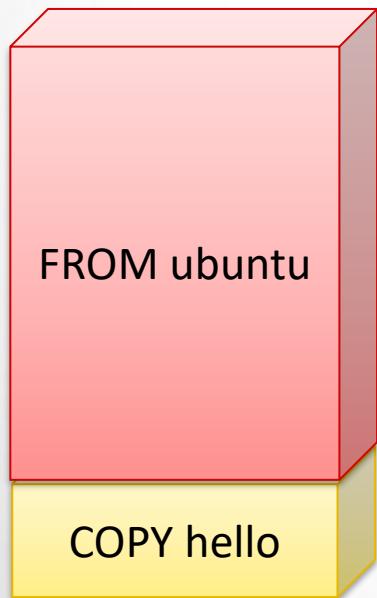
- Quand on parle d'image, on parle généralement d'un ensemble de *layers* avec un tag attribué par un humain



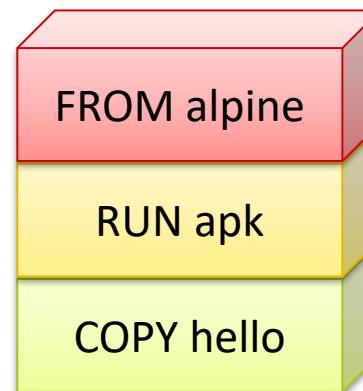
# Deep dive into Image

## ✓ Explorer les *layers* d'une image: dive

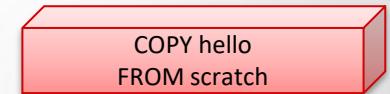
```
FROM ubuntu:3.17.3
COPY hello .
CMD ["./hello"]
```



```
FROM alpine:3.17.3
RUN apk add gcompat
COPY hello .
CMD ["./hello"]
```



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





✓ Chaque layer est identifié par un *tag*

– Autogénéré

- **version courte:** c28b9c2faac4070

- **version longue:** c28b9c2faac407005d4d657e49f372fb3579a47dd4e4d87d13e29edd1c912d5c

– Nom donné par son auteur

- [<repository>/]<name>[:<tag>]

▪ **Exemples:**

- hello-world

- alpine:latest alpine:3.17.3

- python:3.11.3-bullseye python:3.11.3-alpine3.17

- ubiquarium/wol-http:latest



# Optimiser vos images Docker en taille (1/2)

## ✓ Minimiser la taille de l'image

- Installer uniquement les paquets nécessaires à l'exécution
  - Ne pas conserver les packages nécessaires à la compilation
  - Ou utiliser les constructeurs multi-niveaux (faire une image pour construire une image)
- Ne pas conserver le cache d'installation des paquets
  - Dans le même RUN, faire la mise à jour, installation et nettoyage
- Ne copier que les fichiers nécessaires
  - Utiliser un `.dockerignore` (identique à `.gitignore`)

## ✓ Minimiser le nombre de *layers*

- Ne pas faire n commandes RUN, mais une qui regroupe les commandes
- La compression des *layers* sera aussi plus efficace



# Optimiser vos images Docker en taille (2/2)

## ✓ Minimiser la taille de l'image (suite)

- Utiliser Alpine quand c'est possible
  - alpine utilise busybox (et musl libc)...
  - Image de base d'environ 2 à 3Mio
- Mais certaines fois, alpine ne peut être utilisé
  - Code qui ne compile pas ou core dump avec musl (bibliothèque C ultra-compacte)
- Utiliser debian-slim (image de base d'environ 20Mio)
- Utiliser [distroless](#)

## ✓ Minimiser l'espace de stockage hôte utilisé

- Utiliser les mêmes versions de l'image de base si plusieurs containers
- Réutiliser une image existante pour en construire une nouvelle
  - Minimise le nombre de layer à stocker

# Optimiser le développement de vos images

- ✓ Mettre l'installation des paquets dans un layer à part pour éviter de retélécharger
  - Va à l'encontre du minimum de layer
  - A optimiser pour la production une fois l'image fonctionnelle
- ✓ Utilise le « *multi-stage* »
  - Faire une image pour la construction
  - Copier le résultat de la construction dans l'image à distribuer
- ✓ Ne pas copier des sources incluant des fichiers que l'on est en train de modifier pour la construction de l'image (comme le Dockerfile par exemple)
  - Change la signature donc refait le *layer...* et les suivants

# Optimiser la construction de vos images

- ✓ **Fixer les numéros de version de vos dépendances !**
  - Au moins le numéro majeur et mineur pour ne laisser que le numéro patch non contraint (et encore...)
- ✓ **Copiez les fichiers le plus tard possible dans la création de l'image**
  - Evite de refaire les *layers* suivants
  - Mais pas toujours possible
- ✓ **En cas de dev sur une machine embarquée, ne compilez pas sur la cible, mais sur une machine performante**
  - Utiliser la cross-compilation avec `docker buildx`
  - Utiliser l'intégration continue pour produire l'image à chaque « `git push` »



# Cross-compiler sur votre station de travail

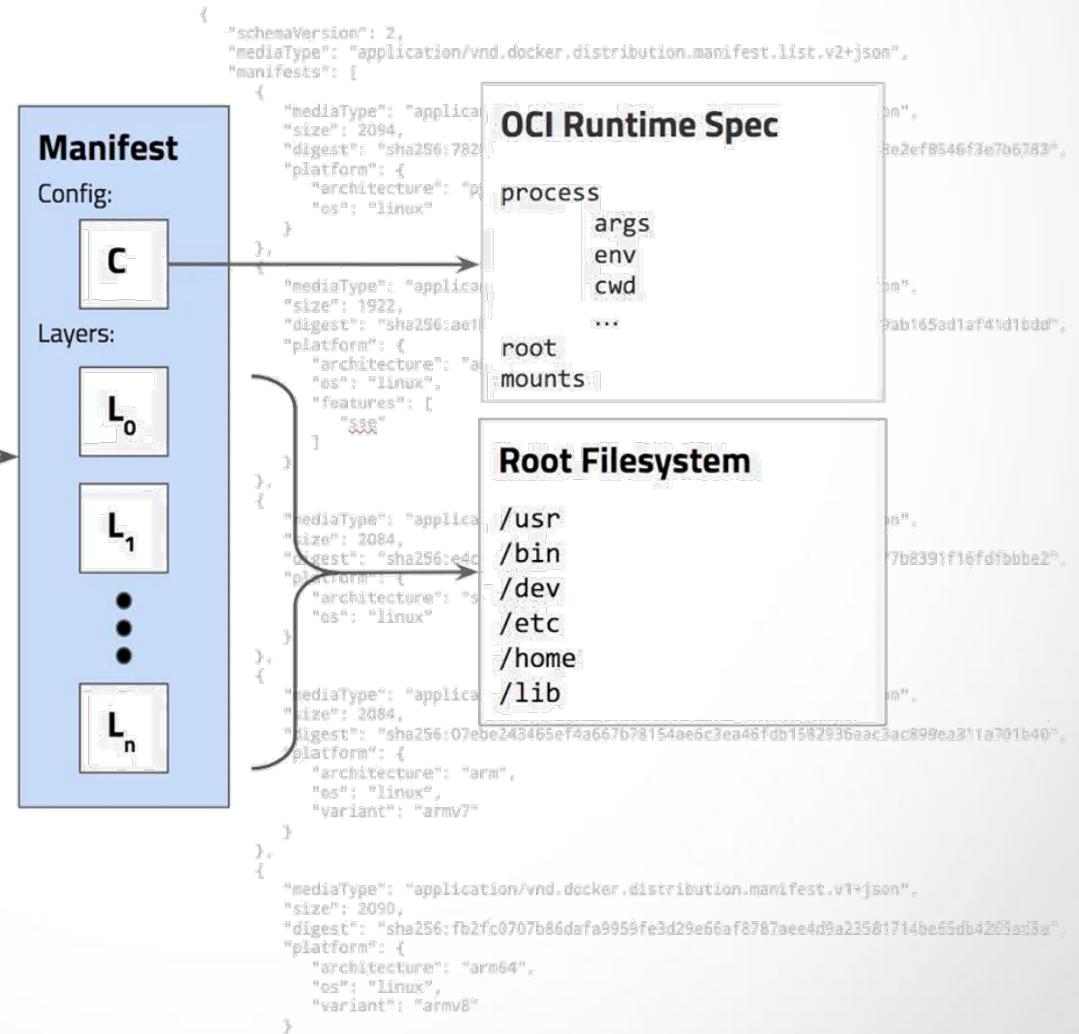
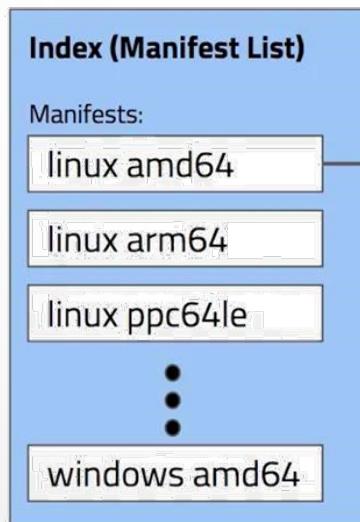
- ✓ Il est possible de créer des images pour d'autres architectures processeur
  - **Cross-compilation:** docker buildx

## ✓ Exemple:

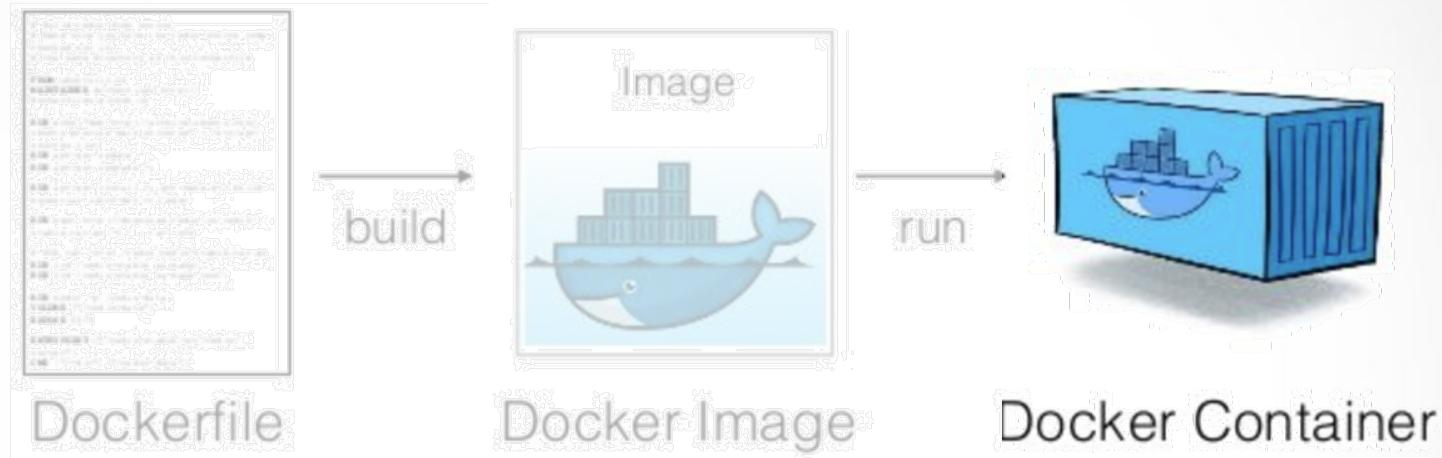
```
docker buildx ls
docker buildx create --name mybuilder
docker buildx use mybuilder
docker buildx inspect --bootstrap
docker buildx build --platform
linux/amd64,linux/arm/v7,linux/arm64-t username/demo:latest --
push .
docker buildx imagetools inspect username/demo:latest
```

# Multi-arch Manifest

## Image Manifests



Source: DockerCon 2017



# Docker conteneur

*Instance d'une image qui s'exécute*



## ✓ Un principe de base

- 1 conteneur = 1 application | 1 service

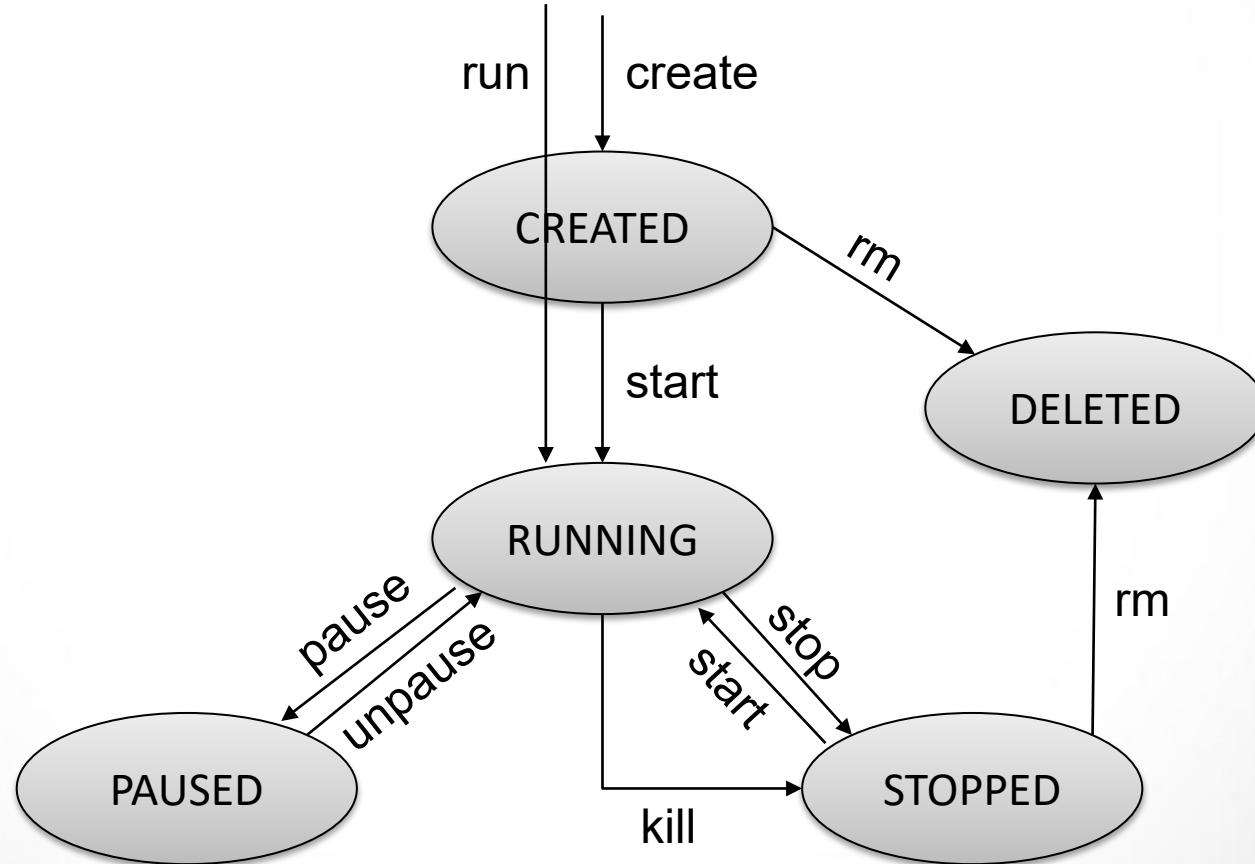
## ✓ Crée et lance l'exécution d'un conteneur

- docker run --help: liste toutes les options de l'action run
- Options les plus courantes:

- **-d: lance le conteneur en background (i.e. &)**
- **-e: spécifie les variables d'environnement**
- **-h: spécifie le nom de l'hôte pour le conteneur**
- **-i: conserve STDIN même si le conteneur est détaché**
- **-m: limite la quantité de mémoire**
- **--name: donne un nom spécifique au conteneur**
- **--rm: retire automatiquement le conteneur quand il se termine**
- **-t: alloue un pseudo tty**
- **-u: spécifie le nom d'utilisateur**



# Cycle de vie d'un conteneur



# Commandes gérant le Cycle de vie d'un conteneur

## ✓ Gestion du cycle de vie d'un conteneur

- docker run: **créé et exécute un nouveau conteneur**
- docker start: **lance l'exécution d'un conteneur existant**
- docker stop: **arrête un conteneur sans le détruire (pause)**
- docker restart: **redémarre un conteneur**
- docker rm: **supprime un conteneur (qui est stoppé, sinon -f)**
- docker ps -a: **liste l'ensemble des conteneurs (-a = même si arrêté)**

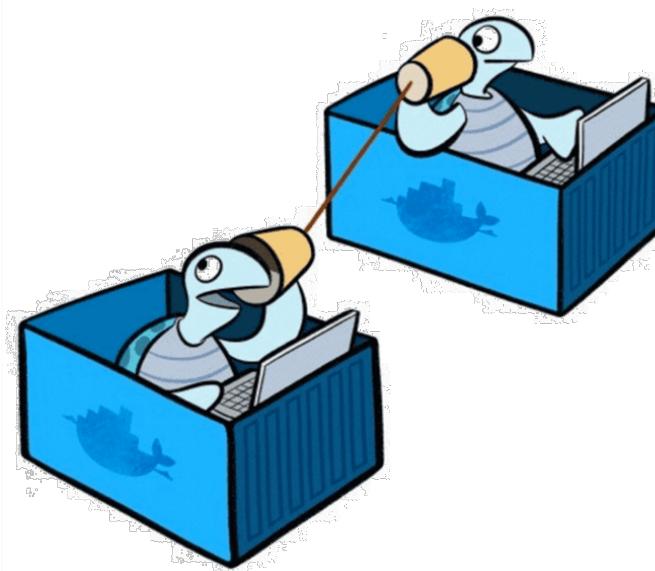
## ✓ Gestion automatique

- Docker fournit des politiques de redémarrage automatique
- Option: --restart
  - no: **ne pas redémarrer automatiquement (défaut)**
  - on-failure: **redémarre s'il a quitté sur une erreur**
  - always: **redémarre toujours le conteneur s'il s'arrête**
  - unless-stopped: **semblable à always, sauf quand le conteneur est arrêté**



# Lancer un processus dans un conteneur

- ✓ **Le Dockerfile permet de spécifier le(s) processus qui sera(seront) lancé(s) dans le conteneur**
  - CMD **et/ou** ENTRYPOINT
  - **Quand le dernier processus lancé s'arrête, le conteneur s'arrête**
  
- ✓ **Il est possible de demander l'exécution d'un processus dans le conteneur une fois qu'il s'exécute**
  - docker exec [options] <conteneur> <cmd> [<args>]
  - **Le nom du conteneur est juste après exec**
    - Car la commande peut avoir un nombre inconnu d'arguments
  - **Exemple très courant:**
    - **Exécuter un Shell dans le conteneur et interagir avec lui**
    - docker exec -it <container> /bin/sh



# Docker Networking

Exposer des ports vers l'extérieur  
ou entre conteneurs



# Aperçu de la mise en réseau

## ✓ Puissance des conteneurs Docker

- On peut créer un réseau de conteneur parlant entre eux
- Pas de connaissance par les applications/services qu'ils sont dans un conteneur

## ✓ Utilise les règles iptables sur Linux

## ✓ Le sous-système de mise en réseau de Docker

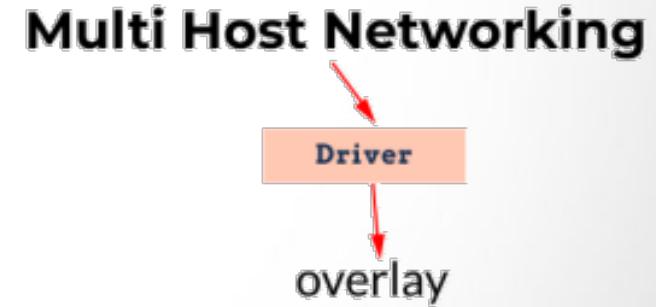
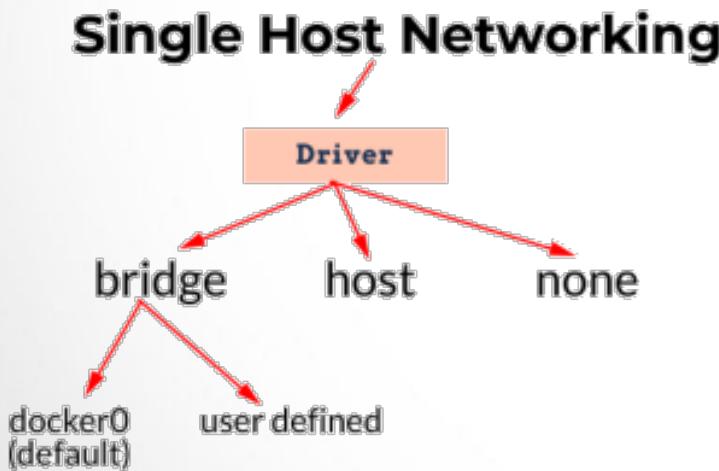
- Utilise des pilotes (drivers)
  - bridge: le pilote par défaut
  - host: supprimer l'isolation réseau entre le conteneur et l'hôte Docker
  - none: désactive toute communication réseau du conteneur
  - overlay: connecter plusieurs moteurs Docker entre eux et permet au service d'essaimage de communiquer entre eux
  - ipvlan: donner aux utilisateurs un contrôle total sur l'adressage IPv4 et IPv6
  - macvlan: permettent d'attribuer une adresse MAC à un conteneur (le fait apparaître comme un appareil physique sur le réseau)



# Réseau Docker

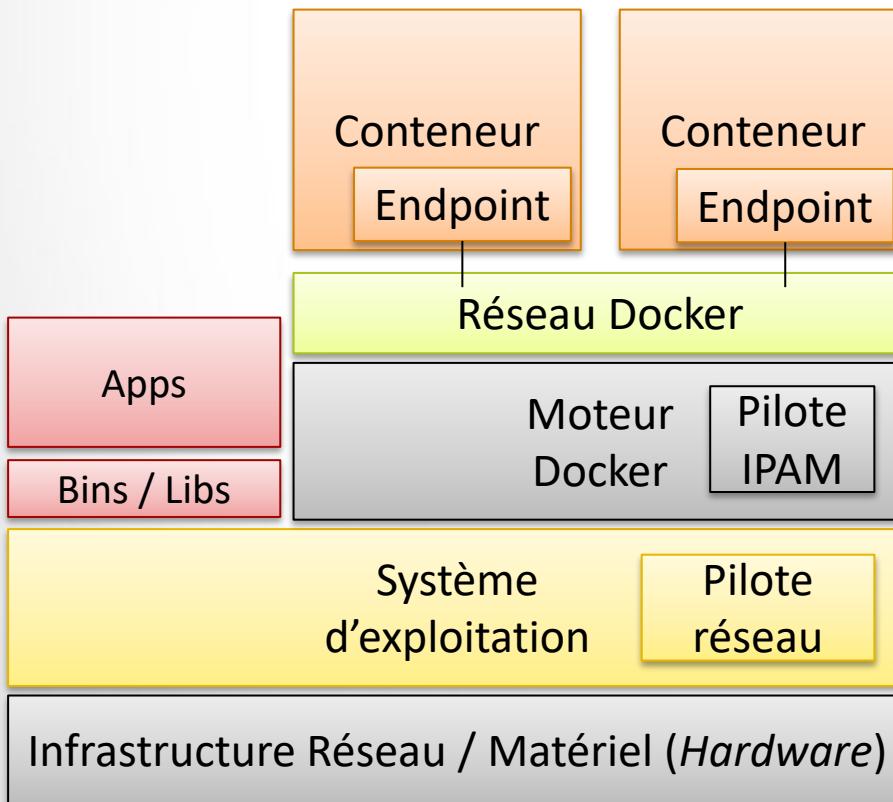
## 1/2

### Docker Networking

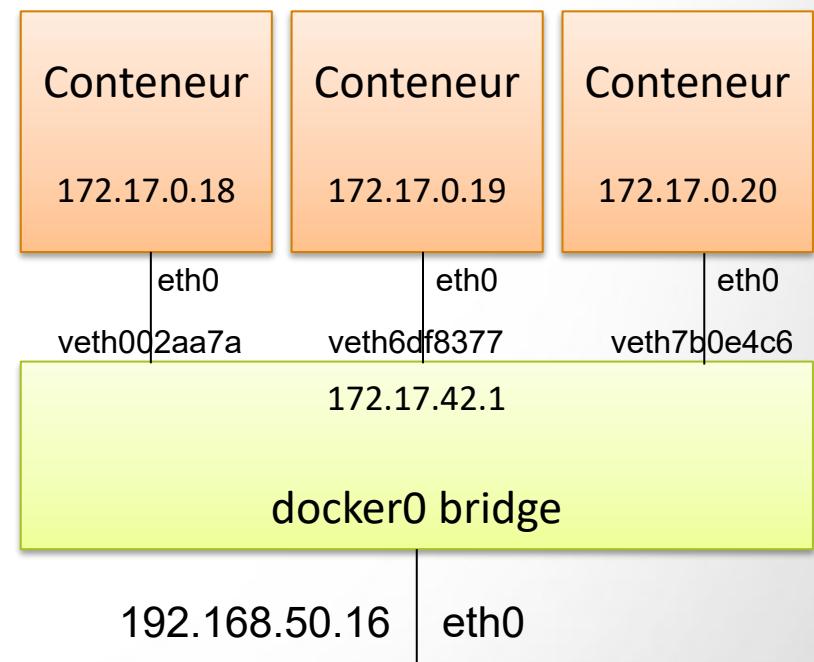


# Réseau Docker

## ✓ Comment fonctionne le réseau Docker



## ✓ Illustration

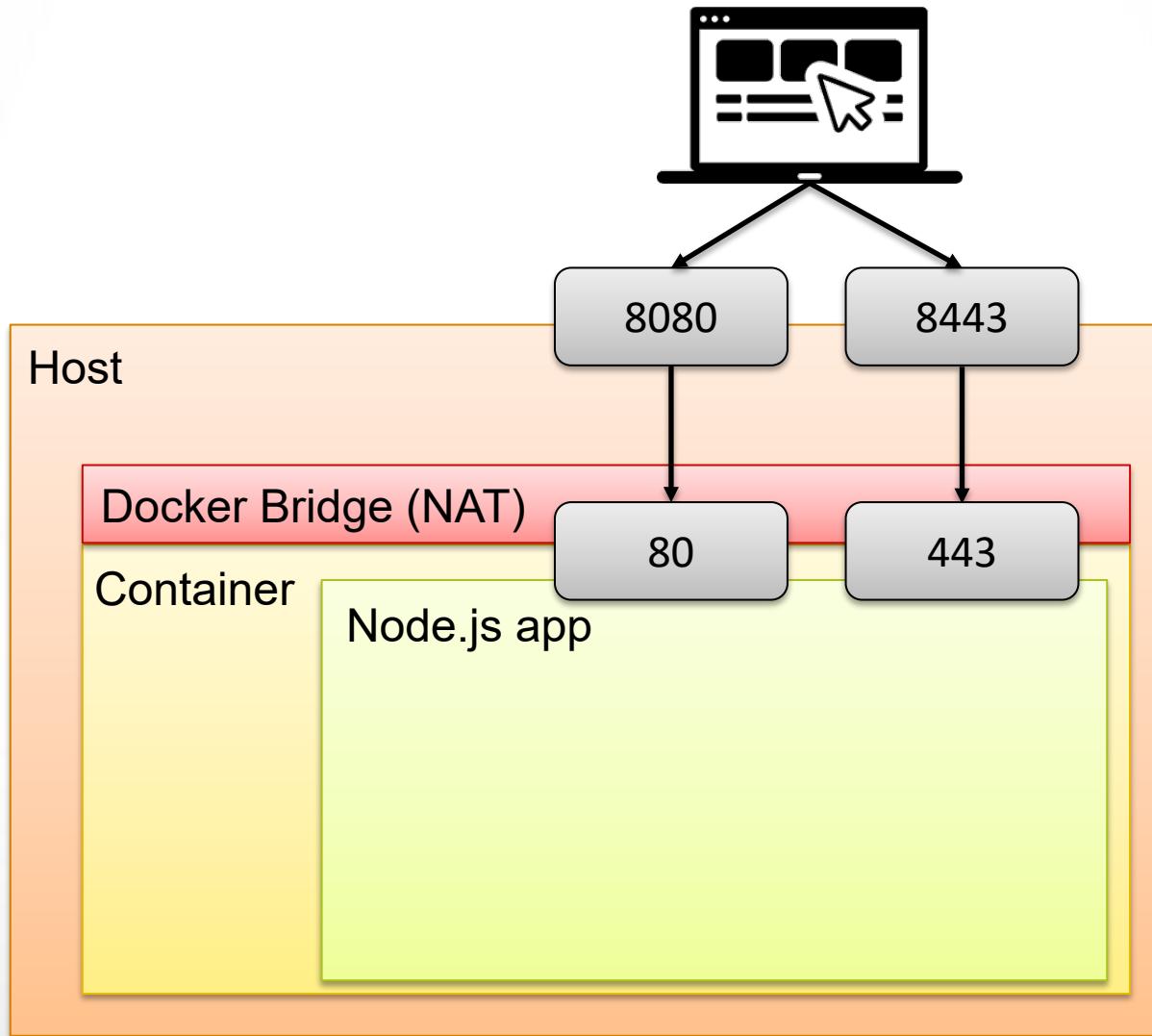


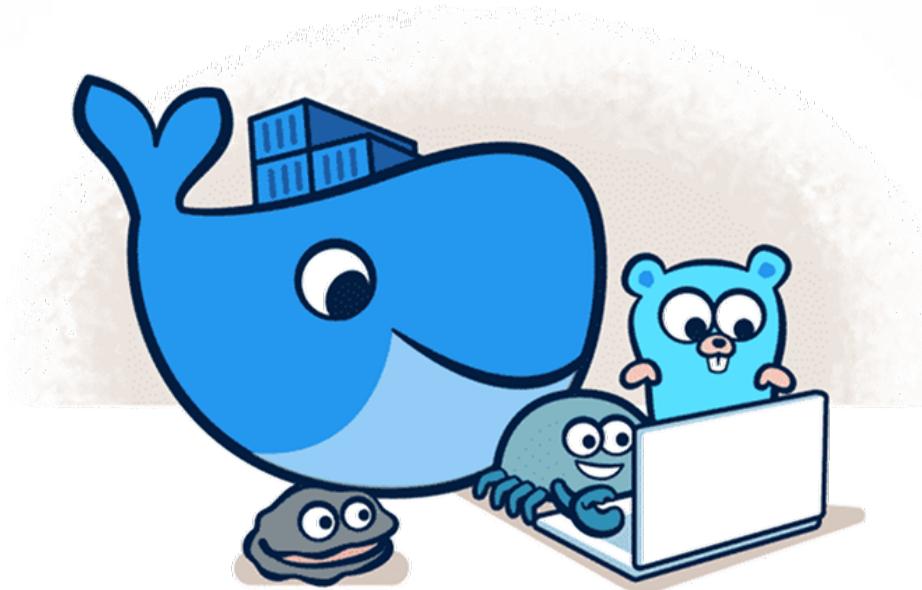


# Gestion des ports réseaux

- ✓ **L'instruction EXPOSE dans le Dockerfile informe Docker que le conteneur écoute sur les ports réseaux**
  - L'instruction EXPOSE ne publie pas les ports
  - Sorte de documentation
- ✓ **Pour publier un port au lancement du conteneur**
  - docker run -p <port\_host>:<port\_container>
- ✓ **Le mapping de port entre hôte et conteneur**
  - Permet de lancer plusieurs instances d'une même application / service que l'on peut atteindre par différents ports de l'hôte (passage à l'échelle)
  - Permet de n'exposer que certains ports, même si plusieurs ouverts dans le conteneur (sécurité)
  - Meilleure maîtrise de la sécurité

# Cartographie des ports





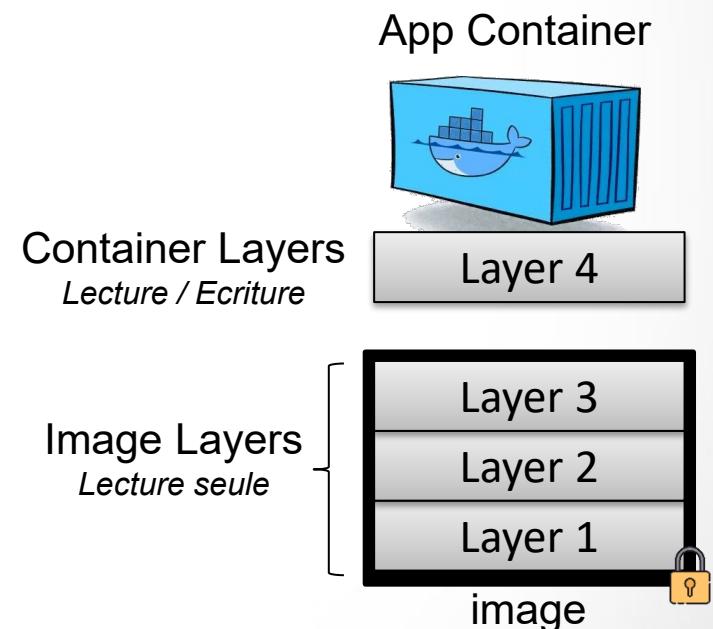
# Docker Volumes

Persistance des données



# Problème de persistance

- ✓ **Les images sont en lecture seule**
  - Elle sont immuables une fois construites
  - On ne peut qu'en dériver et ajouter des *layers*
- ✓ **Quand le conteneur produit des données**
  - Ces données sont ajoutées à un *layer* de l'exécution du conteneur
  - Ce *layer* est en lecture écriture
  - Mais il n'a que la durée de vie du conteneur



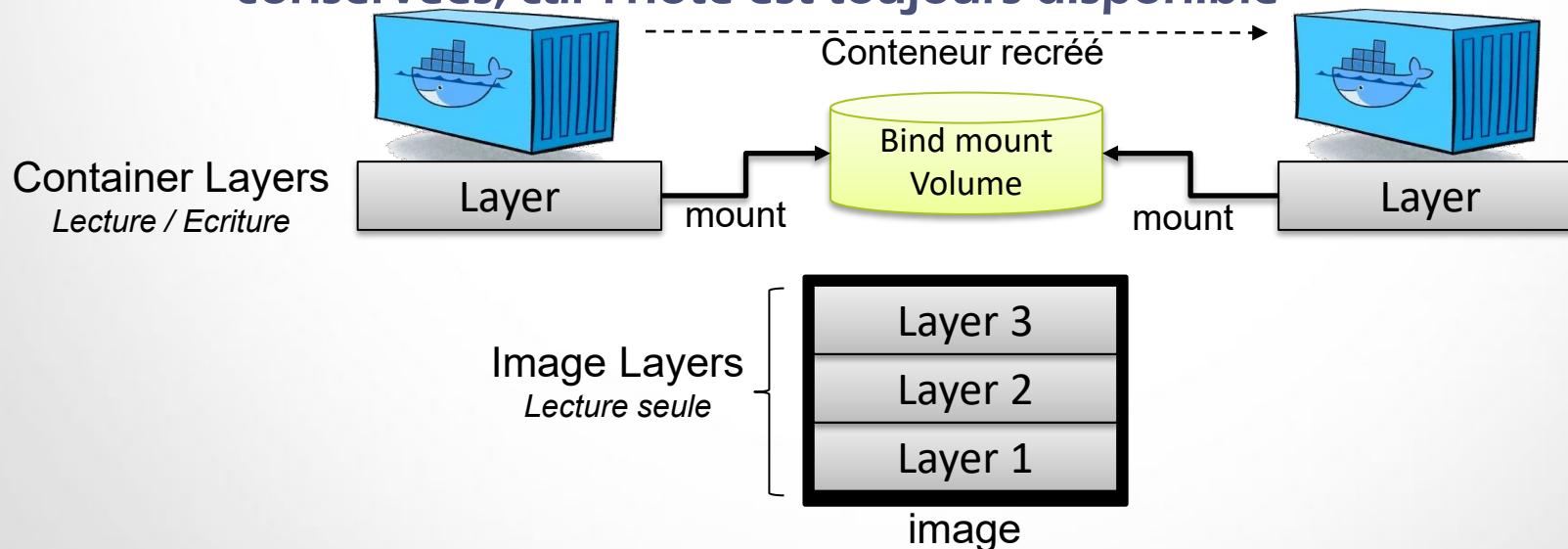
# La solution au problème de persistance

## ✓ Le montage

- Procédé consistant à rendre des fichiers et des répertoires accessibles à l'utilisateur via le système de fichiers

## ✓ Pour un conteneur

- Monte un fichier/dossier du système de fichiers du conteneur vers un fichier/dossier du système de fichiers de l'hôte
- Assure que si le conteneur supprimé, ses données sont conservées, car l'hôte est toujours disponible





# Type de montage

## ✓ 2 types de montages

- Bind mounts

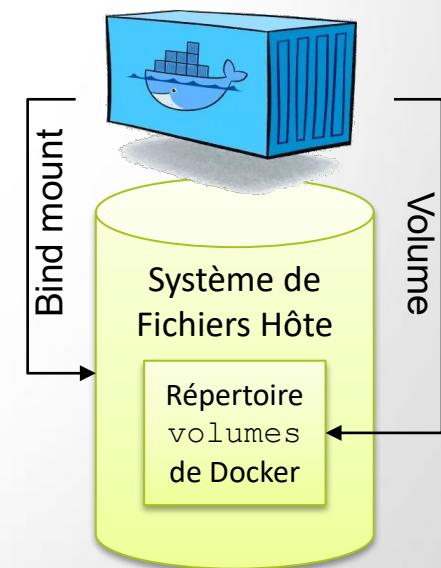
- Peut-être stocké n'importe où sur le système de fichiers hôte

- Volumes

- Stocké dans une partie du système de fichiers hôte géré par Docker (Linux: /var/lib/docker/volumes/)
  - Peut avoir un nom
  - Peut être partagé par plusieurs conteneurs sur des hôtes différents

## ✓ Peu importe le type utilisé, perçu de la même manière dans le conteneur

- Donc transparent pour l'application / service





- ✓ **Pourquoi les volumes sont préférés au Bind mounts**
  - Portabilité: moins de couplage à la structure du FS hôte
  - Sécurité: Bind mounts peut modifier le FS hôte (donc affecter d'autres processus). Volume dans un espace réservé à Docker
  - Facilité de gestion: gestion avec le docker CLI
  - I/O performance on Docker Desktop (Mac & Windows) car stocké dans la VM Linux plutôt que sur l'hôte
  - Accessibilité: pour stocker vos données dans le cloud, souvent le seul choix possible
- ✓ **Commandes de manipulation des volumes**
  - docker volume create
  - docker volume rm
  - docker volume ls
  - docker volume inspect
  - docker volume prune



# Utilisation lors du lancement du conteneur

## ✓ Options

- **-v ou -volumes**: spécifie un volume à monter (bind ou volume)
- Peut être utilisé plusieurs fois dans la commande docker run

## ✓ Bind mount: /path/to

- docker run -v /path/to/host:/path/to/container ...

## ✓ Volume: name du volume

- docker run -v name:/path/to/container ...

## ✓ Peut ajouter un troisième paramètre optionnel

- Par défaut partage en lecture écriture
- :ro : permet de spécifier montage en lecture seule

## ✓ Ex:

- docker run -v /etc/localtime:/etc/localtime:ro \  
-v /etc/timezone:/etc/timezone:ro \  
-v data:/usr/app/data ...

← Volume

Bind mount



# Quelques principes de sécurité

*Si on veut éviter de se faire voler la marchandise dans le conteneur...*



# Sécurité Docker

- ✓ **4 domaines principaux lors de l'examen de la sécurité de Docker**
  - la sécurité intrinsèque du noyau et sa prise en charge les namespaces et les cgroups
  - la surface d'attaque du moteur Docker lui-même
    - Requière les privilèges de super-utilisateur
    - Seuls les utilisateurs de confiance devraient être autorisés à contrôler votre moteur Docker
  - les failles dans le profil de configuration du conteneur, soit par défaut, soit lorsqu'il est personnalisé par les utilisateurs
  - les caractéristiques de sécurité du noyau et la façon dont elles interagissent avec les conteneurs
- ✓ **Les conteneurs Docker sont, par défaut, assez sécurisés, mais attention**
  - Aux attaques externes: par ce qui vient de l'extérieur du conteneur
  - Aux attaques internes: par ce qui se trouve dans le conteneur



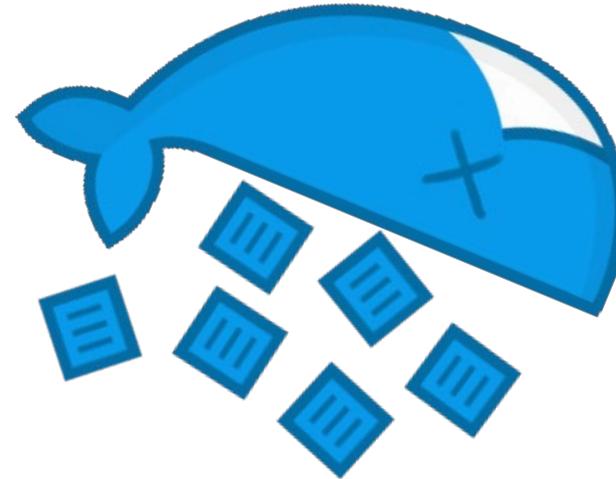
# Bonne pratiques

- ✓ Utiliser des images officielles / certifiées
  - Le moteur Docker peut être configuré pour n'exécuter que des images signées
  - Attention si vous récupérez n'importe quelle image
- ✓ Exécuter vos processus en tant qu'utilisateur non privilégié dans le conteneur
  - root dans le conteneur = root sur la machine
  - si un conteneur est compromis, il ne faut pas grand-chose pour compromettre l'ensemble de l'hôte
- ✓ Ne pas mettre d'informations personnelles dans l'image
  - Pas d'identifiant ou de mot de passe
- ✓ Sécuriser l'hôte
  - si l'hôte est compromis l'isolation des conteneurs et les mesures de sécurité ne feront pas une grande différence



# Isolation du conteneur

- ✓ **Eviter** --network host **ou** --privileged
  - **Sauf si cela est vraiment nécessaire (par exemple pour modifier la configuration de la plateforme sous-jacente)**
  - **Avec** --network:
    - **Par d'isolation réelle côté réseaux (donc tous les ports exposés par les conteneurs sont accessibles)**
  - **Avec** --privileged:
    - **Cache le fait que votre image inclue toutes les dépendances car accès à l'ensemble du FS**



# Conclusion

Pour en finir...



# Conclusion

## ✓ Avantages de la conteneurisation

- Facilite le déploiement d'application
  - Aucune dépendance en dehors du conteneur, sauf accès hardware (cf SI5)
- Isolation (=> agilité)
  - Pas d'interférence entre les conteneurs et « isolation » de l'OS
- « Portabilité »
  - Déployer dans différents environnements
- Capacité de mise à l'échelle
  - Déploiement d'applications basées sur des micro-services (cf SI4)
- Tolérance aux pannes
  - Redondance possible et facilitée



# Et encore plus...



- ✓ **Et d'autres choses à découvrir et mettre en pratique dès PS6**
  - Le déploiement d'applications composées de plusieurs conteneurs
  - docker-compose
  - La communication entre conteneurs
  - ...