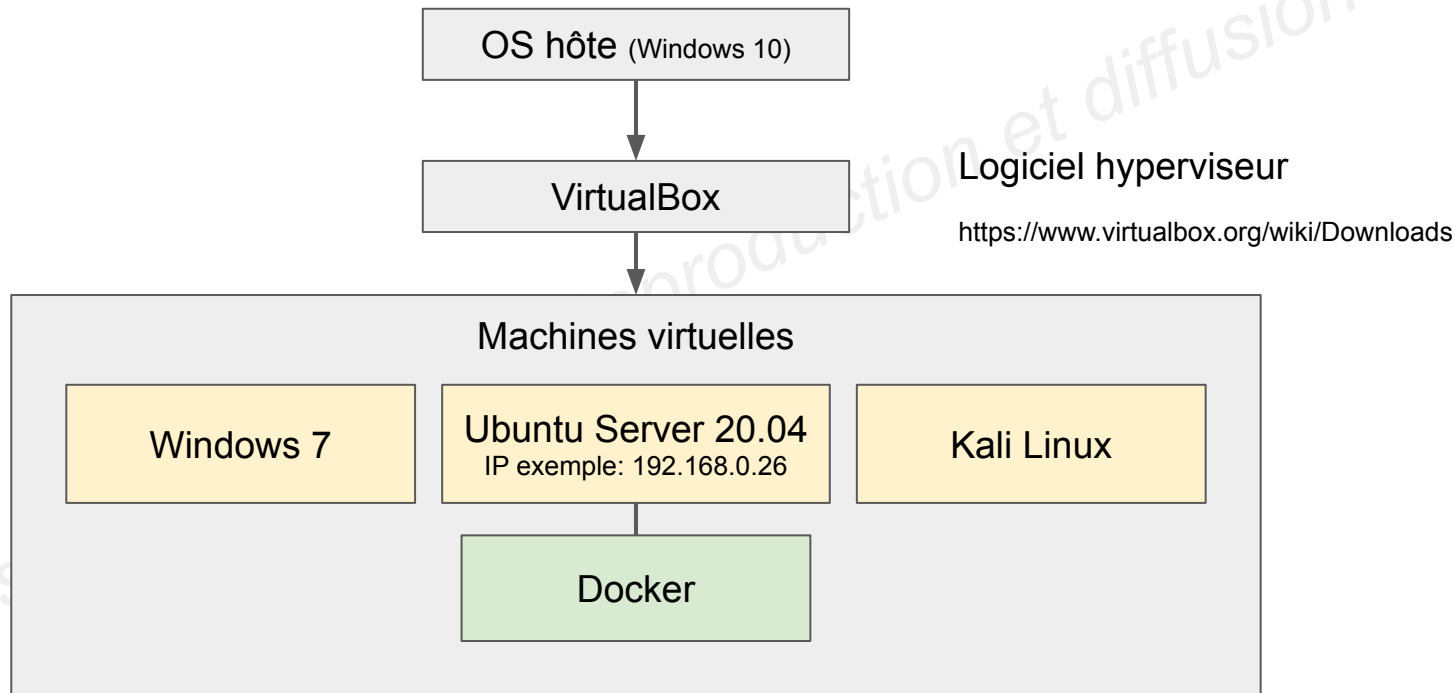


# Docker

Christophe Dufour



# Infrastructure (prof)



# Virtualisation

- Faire fonctionner un/plusieurs OS/apps comme un simple logiciel sur un/plusieurs ordinateurs
- Eviter d'avoir un seul OS pour une seule machine physique
- La virtualisation OS permet de faire fonctionner simultanément, sur une seule machine physique, différents OS comme si ces derniers fonctionnaient sur des machines distinctes

# Historique

- La virtualisation commence dans les années 1970 sur des machines IBM
- Années 80, virtualisation permettant d'émuler des machines Dos/Windows sur Mac/Unix

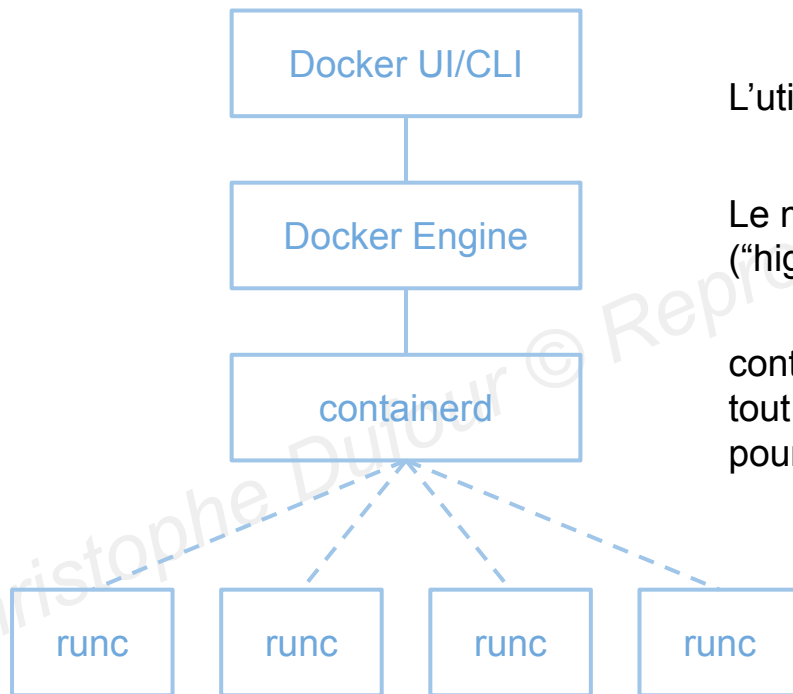
# Hyperviseur

- Manière la plus courante de simuler des espaces de travail avec des OS différents
- Deux types
  - **Type 1**: OS très léger gérant l'accès des OS invités. "Paravirtualisation": hyperviseur et OS invité coopèrent. Ex: VMWare VSphere, Microsoft Hyper-V, KVM
  - **Type 2**: Logiciel installé sur un OS "hôte". Les OS invités, les machines virtuelles, ne savent pas qu'elles sont virtualisées. Ex: VMware Server, VirtualBox, Parallels

# Docker - intro

- Moteur de virtualisation/conteneurisation le plus populaire
- Logiciel libre, open source depuis 2013
- Développé au départ par Solomon Hykes pour un projet interne à la société française dotCloud
- Utilise des services fournis par le noyau linux
  - cgroups, apparmor/selinux, namespaces
- Se base sur LXC (Linux Container) au départ pour en étendre les capacités

# Docker - architecture



L'utilisateur interagit avec le moteur Docker

Le moteur Docker communique avec containerd  
("high level runtime")

containerd utilise runc ("low level runtime") ou  
tout autre runtime conforme aux normes OCI  
pour exécuter des conteneurs

# LXC/LXD - intro

- Outil de virtualisation linux, 2014
- basé sur les conteneurs Linux
- respect du OCI (Open Container Initiative)
- A inspiré docker mais indépendant
- Virtualisation système
- LXD: API de gestion, CLI
- Utilise les cgroups et namespaces linux

```
root@lxc:~# lxc launch images:alpine/3.10 monconteneur
Creating monconteneur
Starting monconteneur
root@lxc:~# lxc list
```

NAME	STATE	IPV4	
monconteneur	RUNNING	10.207.45.178 (eth0)	fd42:



# Cgroups (groupes de contrôle)

Contrôlent les ressources utilisées par un ou plusieurs processus. Les processus contrôlés sont réunis dans des groupes sur lesquels agissent des contrôleurs par type de ressource:

- cpuset: allocation CPU
- cpuacct: consommation CPU
- memory: contrôle de la mémoire vive et de la mémoire swap
- devices: contrôle de l'accès aux périphériques
- blkio: contrôle de l'accès aux périphériques de type bloc (ex: disque dur)
- net\_cls: contrôle de l'accès aux périphériques réseau

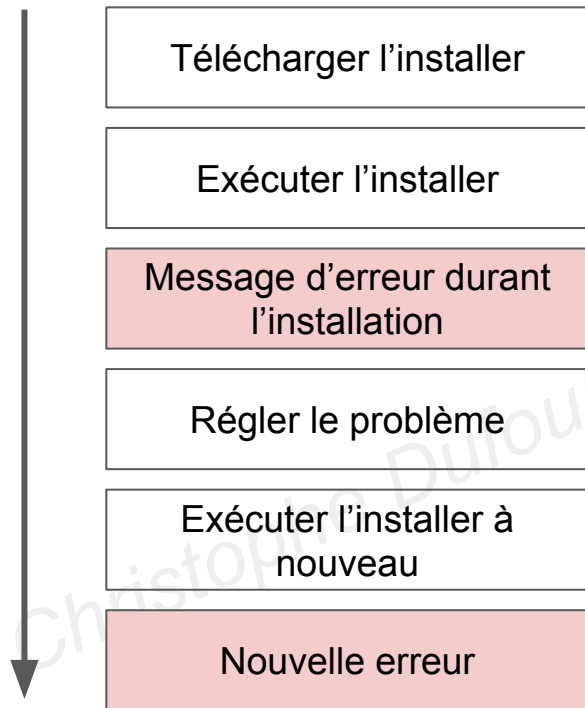
# Namespaces (espaces de nom)

Il existe différents types de namespaces s'appliquant à une ressource spécifique. Chaque espace de noms crée des barrières entre les processus. Voici quelques types d'espace de noms que docker utilise :

- pid: isole les processus (pid: Process ID)
- net: permet la gestion des interfaces réseau (net: networking)
- ipc: gère les accès inter-processus (ipc: InterProcess Communication)
- mnt: gère les points de montage (mnt: mount)
- uts: utilisé pour isoler le noyau et identificateurs de version (uts: Unix Timesharing System)

# Pourquoi utiliser Docker ?

*Installer un logiciel*



Ce à quoi Docker tente de remédier

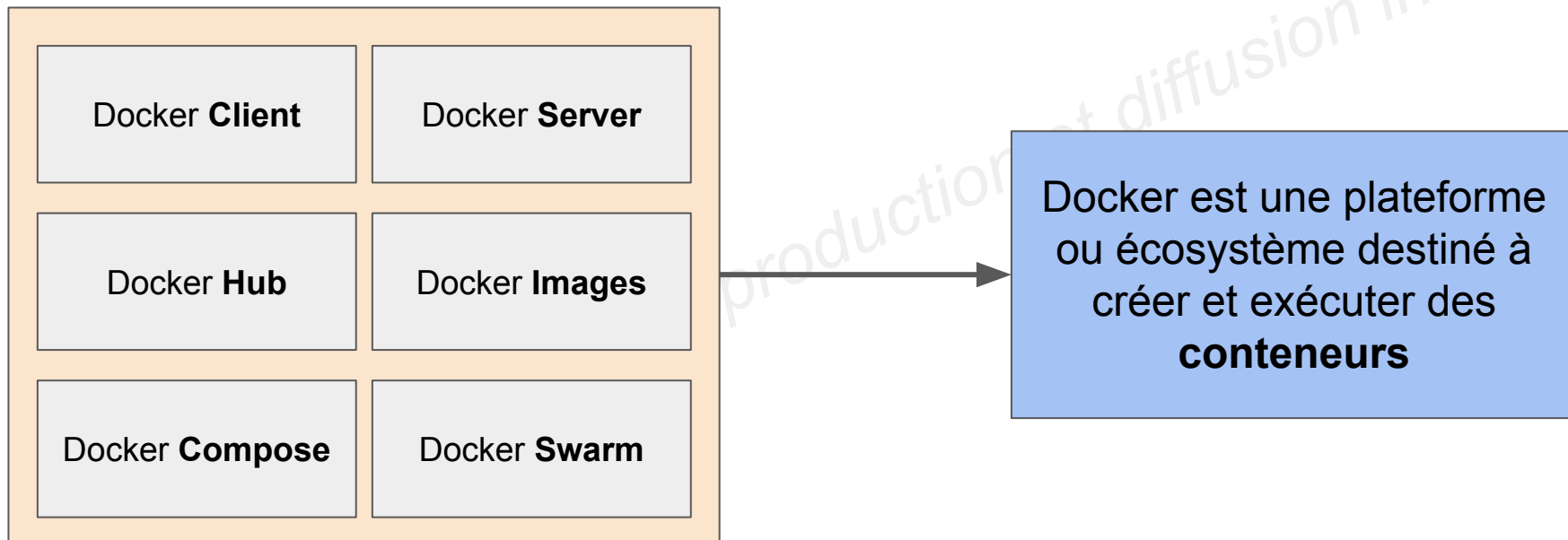
# Pourquoi utiliser Docker ?

Pourquoi l'utiliser ?

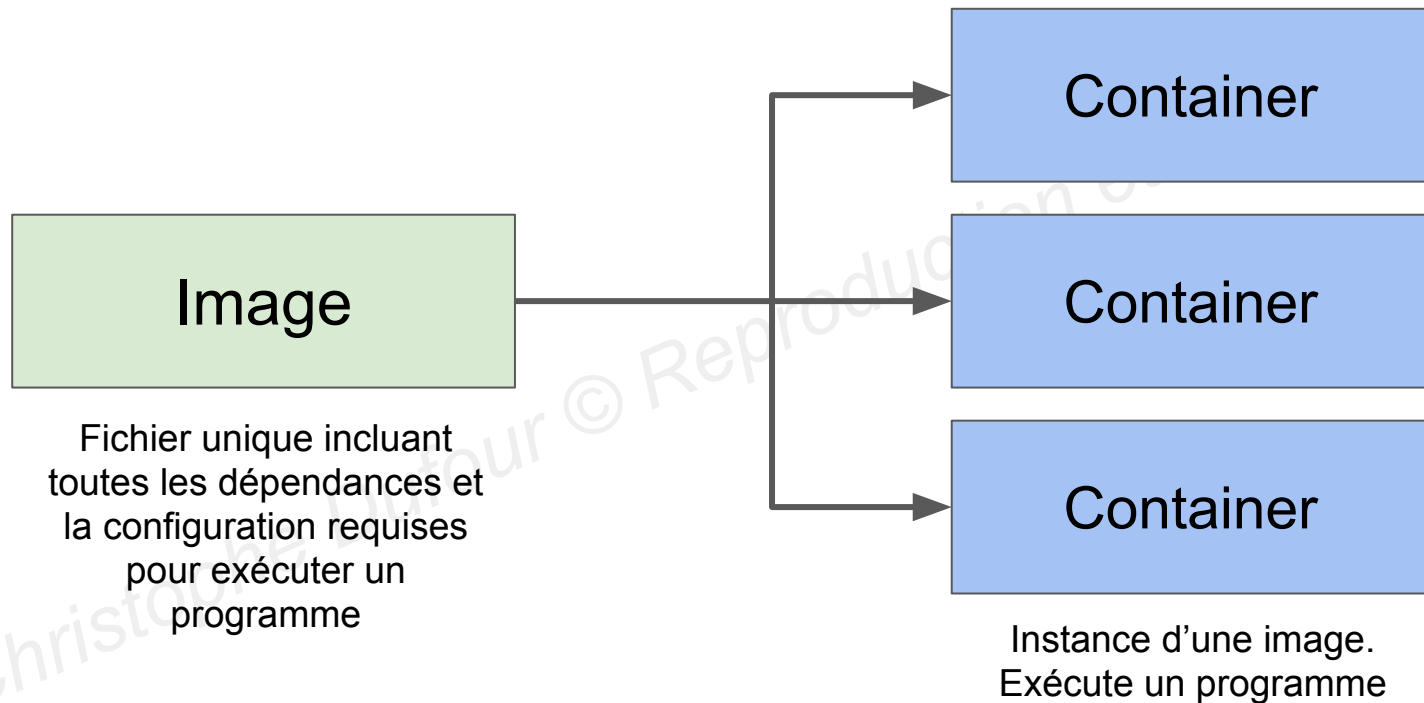


Docker permet très facilement d'installer et exécuter des logiciels sans souci du paramétrage et des dépendances

# L'écosystème Docker

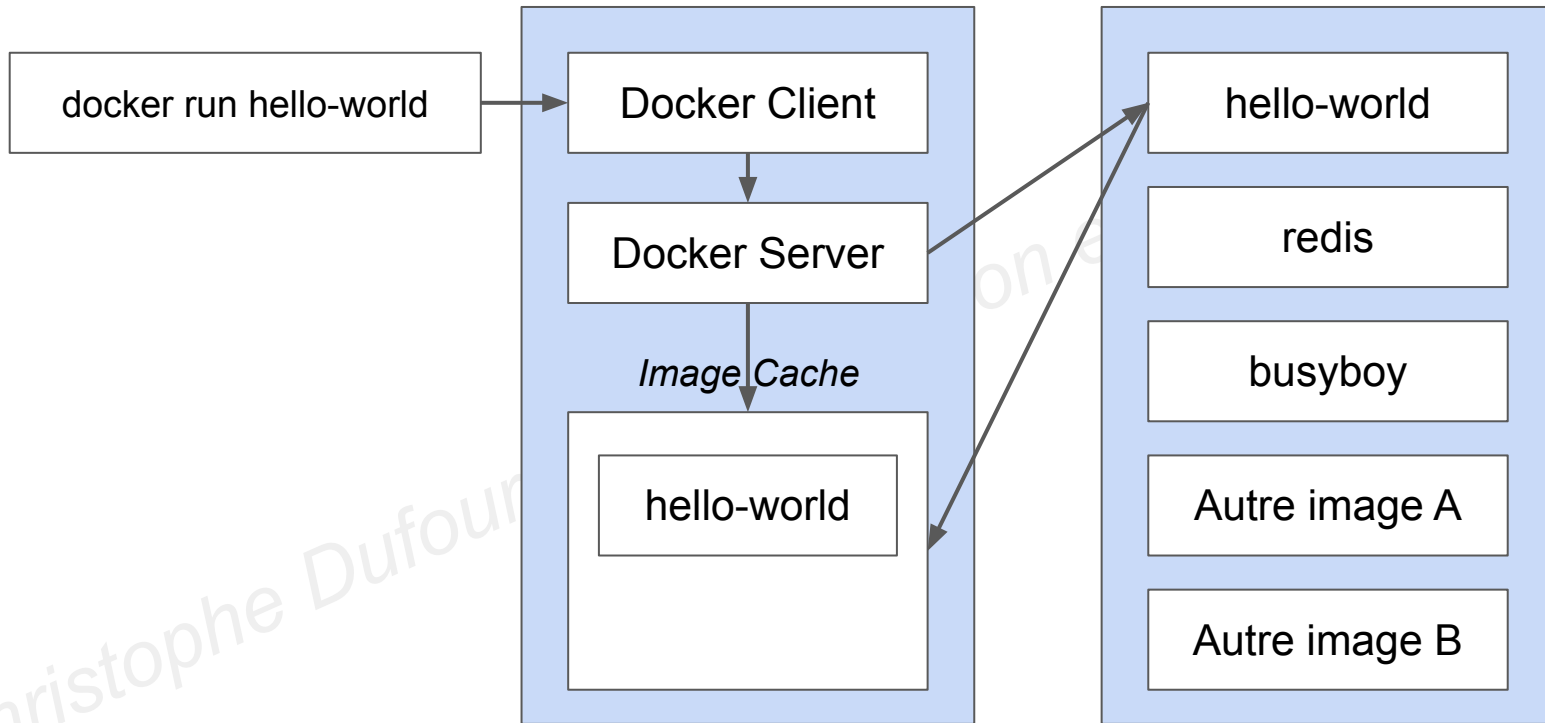


# L'écosystème Docker

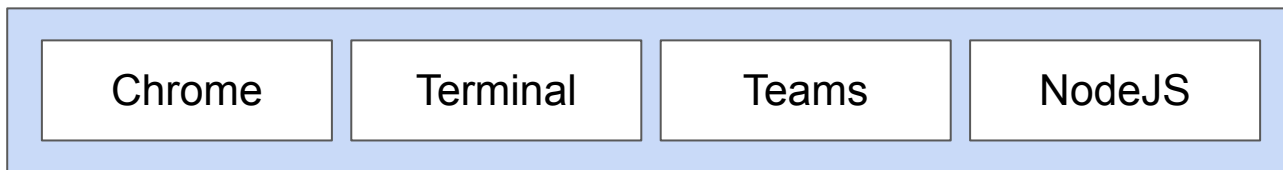


## Mon ordinateur

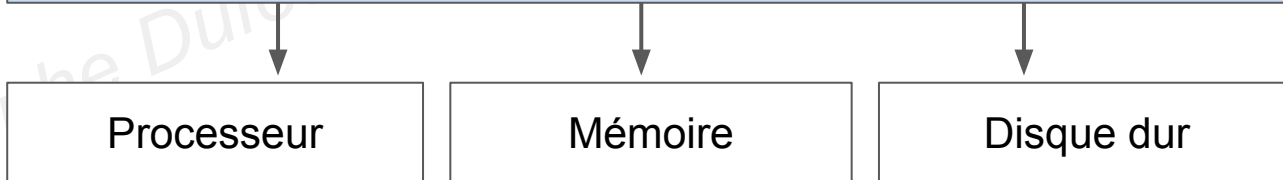
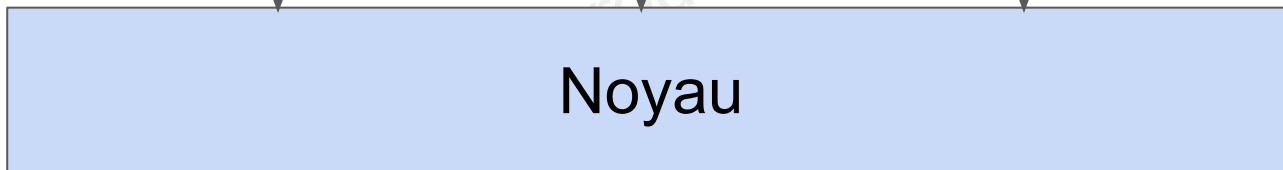
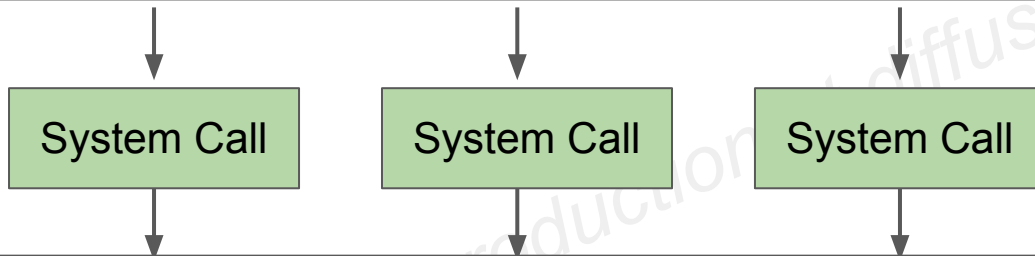
## Docker Hub



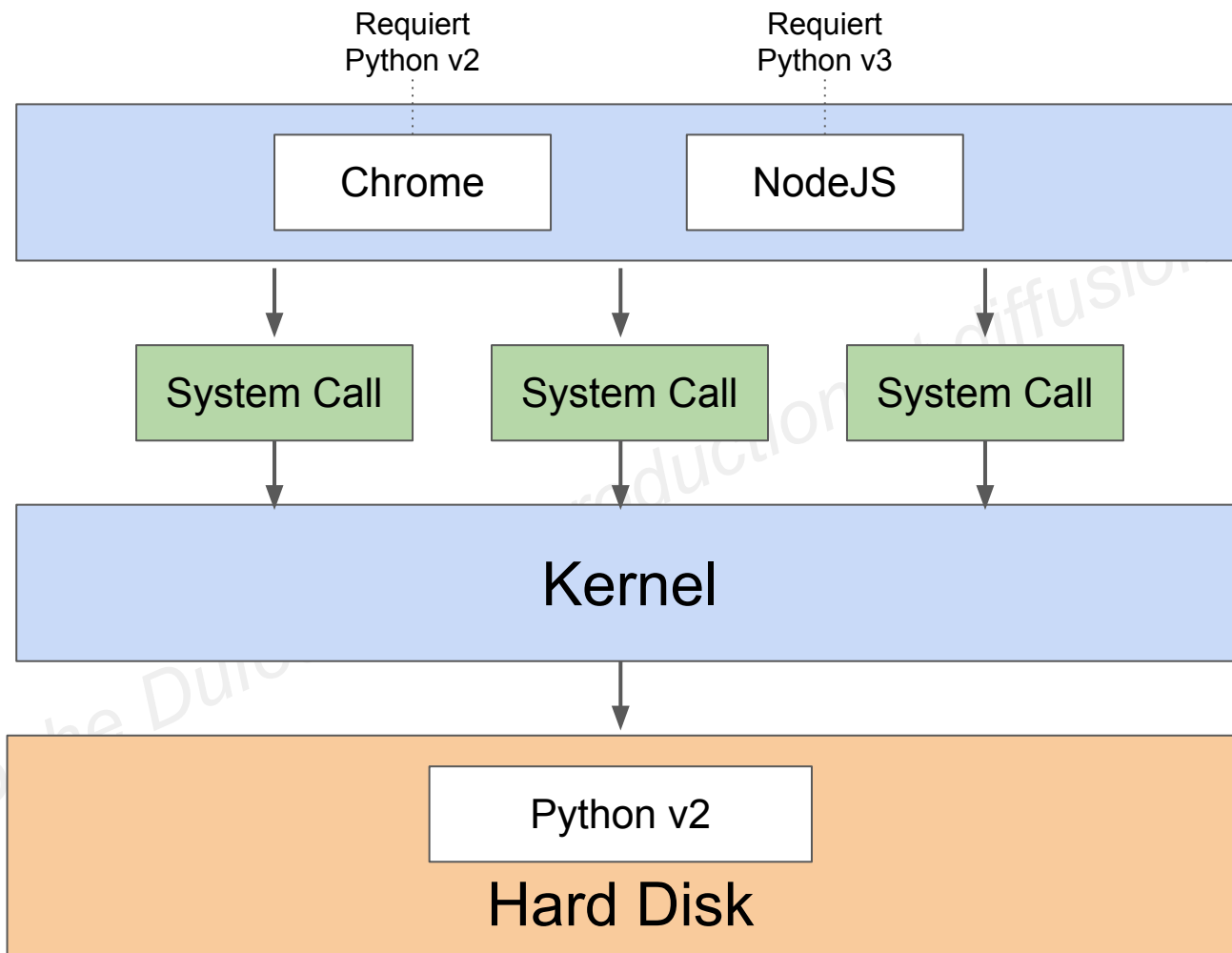
Processus en  
cours d'exécution  
sur mon  
ordinateur



Appels systèmes  
vers le noyau afin  
d'obtenir des accès  
bas niveau







Requiert  
Python v2

Requiert  
Python v3

Chrome

NodeJS

System Call  
d'accès au HD

Kernel

Quel processus  
effectue le SysCall ?

Segment HD  
pour Chrome

Python v2

conflit potentiel

Python v3

Segment HD  
pour NodeJS

Hard Disk

## Namespacing

*Isolation de ressources  
par process ou groupe  
de process*

Processes

Hard drive

Network

Users

Hostnames

Inter Process  
Communication

## Control Groups (cgroups)

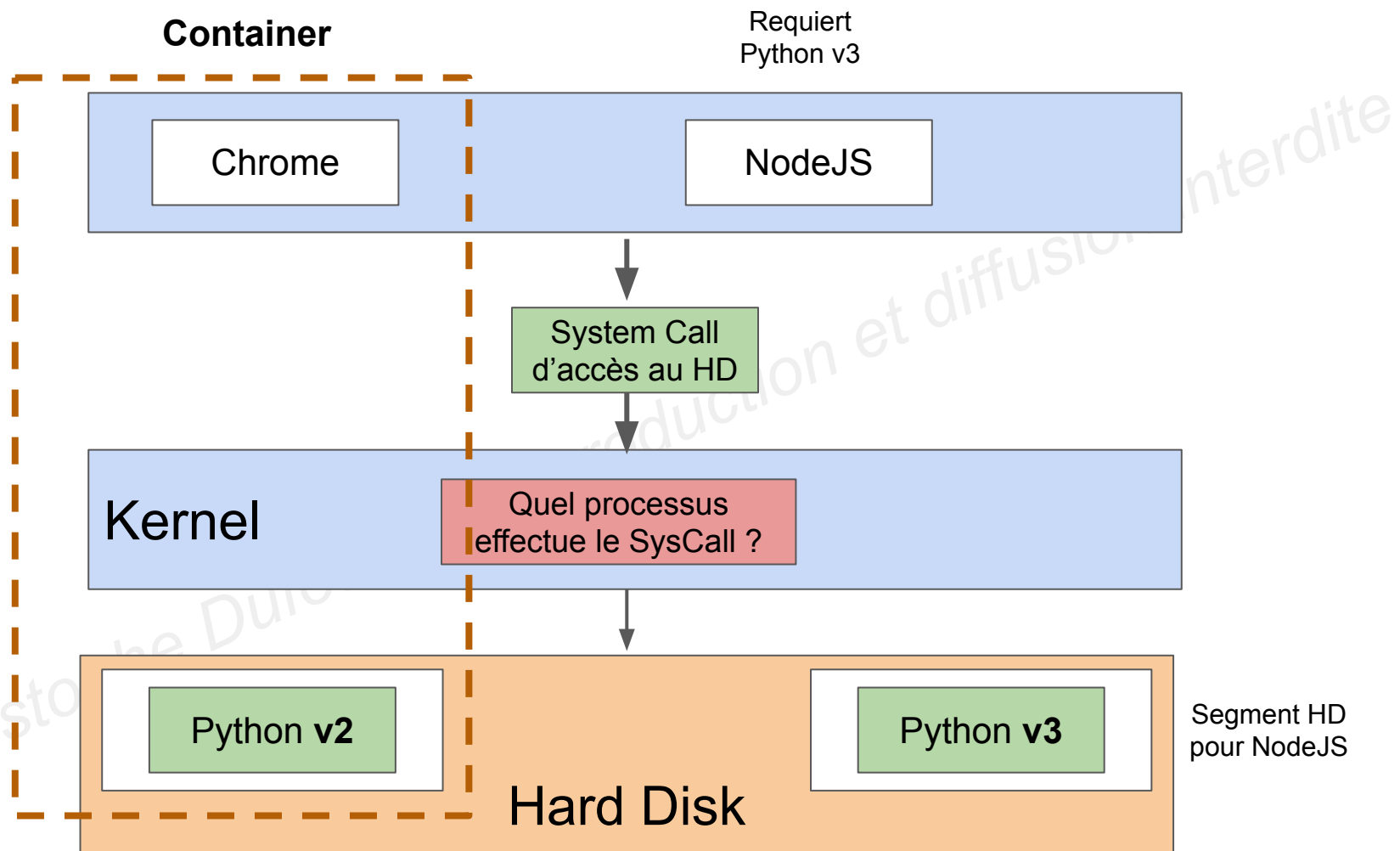
*Montant limité de  
ressources utilisées par  
process*

Memory

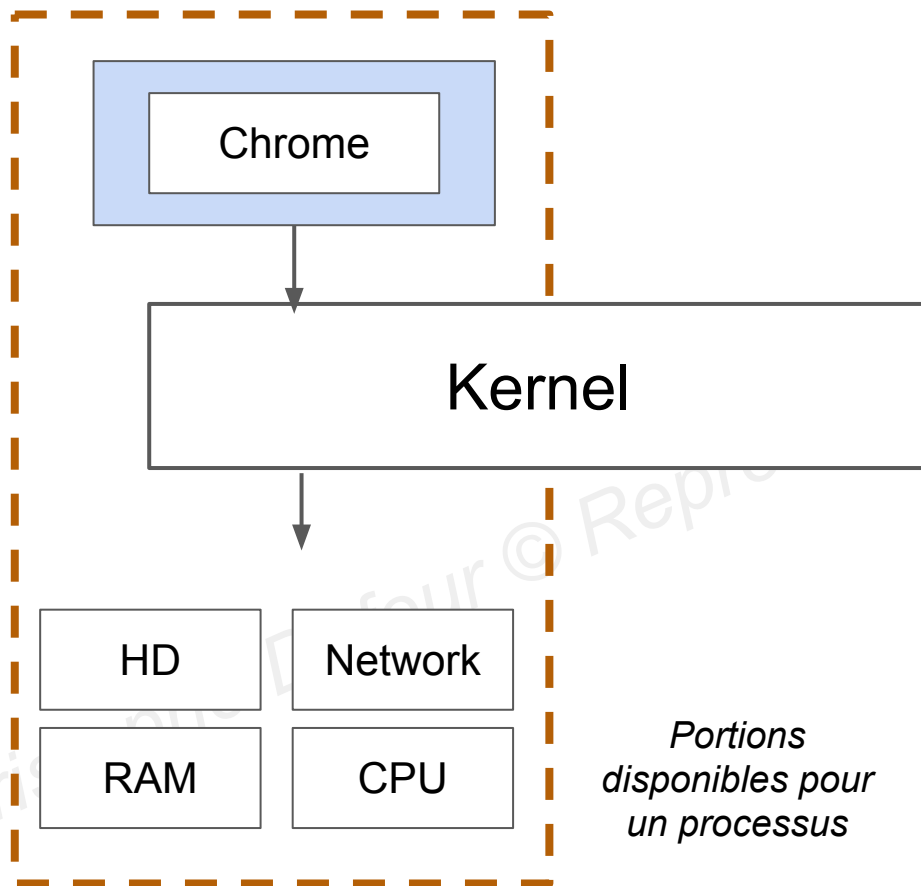
CPU Usage

HD I/O

Network  
Bandwidth

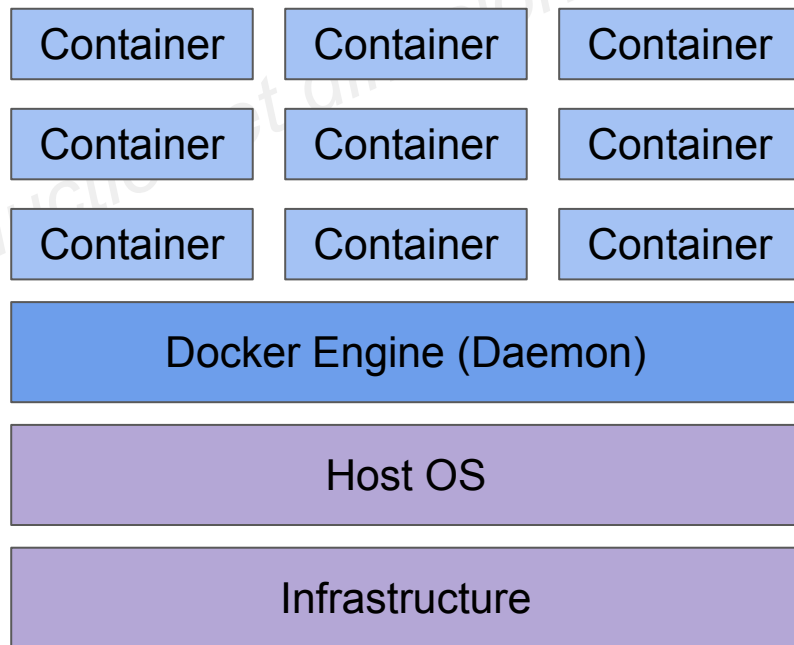
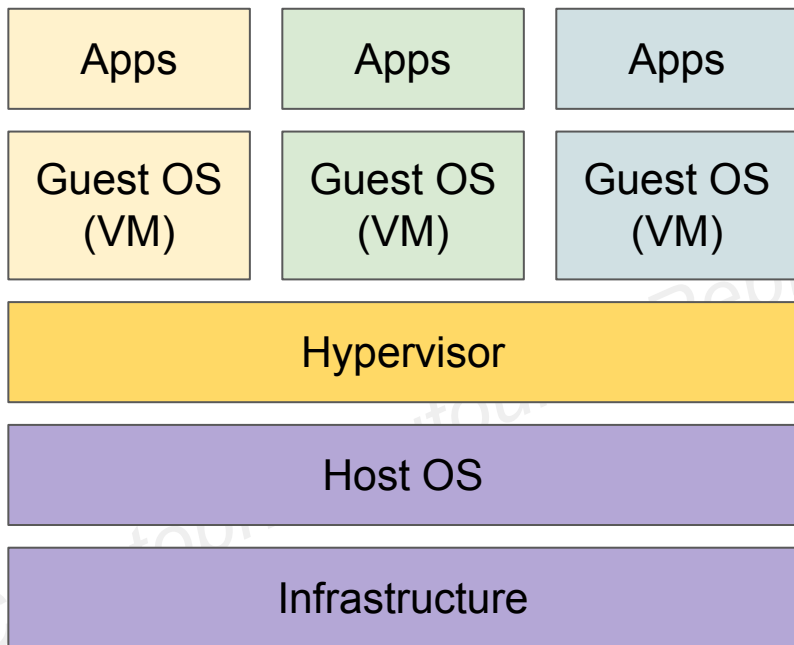


## Container



*Portions  
disponibles pour  
un processus*

# VM vs Docker



# VM vs Docker

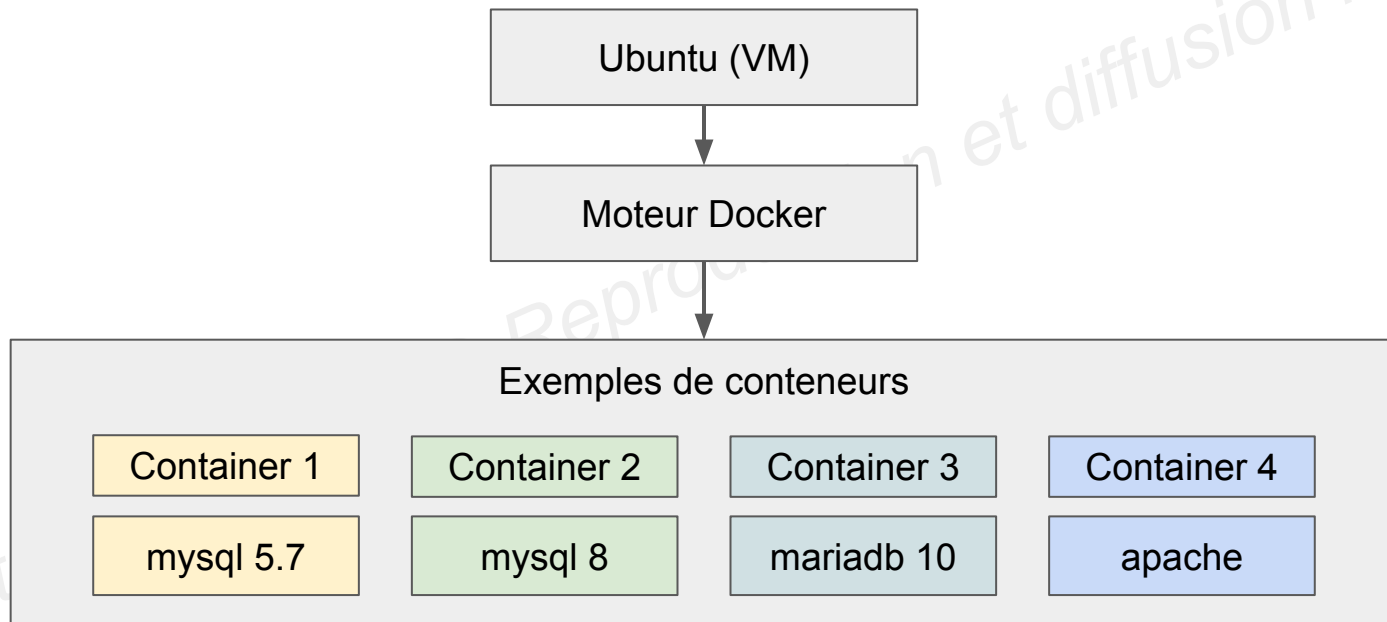
Isolation des processus au niveau matériel	Isolation des processus au niveau du système d'exploitation
Chaque machine virtuelle a un système d'exploitation distinct	Chaque conteneur peut partager le système d'exploitation
Démarrage en quelques minutes	Démarrage en quelques secondes
Poids en Go	Poids en Ko/Mo
Les VM prêtes à l'emploi sont difficiles à trouver	Les conteneurs Docker pré-construits sont facilement disponibles
Les VM peuvent facilement se déplacer vers un nouvel hôte	Les conteneurs sont détruits et recréés au lieu de se déplacer
Temps de création relativement long	Création en quelques secondes
Plus d'utilisation des ressources	Moins d'utilisation des ressources

# VM vs Docker - que privilégier ?

- Docker s'il s'agit de faire tourner un grand nombre d'applications - potentiellement dans différentes versions - sur un nombre de serveurs limité
- VM s'il s'agit de faire tourner un nombre d'applications limité sur un grand nombre de serveurs (différents OS)
- VM si l'isolation maximale (sécurité) est prioritaire
- Dans la vie réelle, on trouve généralement des approches "hybrides" avec des conteneurs tournant dans des VMs



# VM + Docker: approche hybride



# Installation sur Ubuntu

- Mise à jour apt  
`sudo apt update`
- Installation de paquets utiles  
`sudo apt install -y apt-transport-https ca-certificates curl gnupg lsb-release`
- Ajout de la clé GPG officielle de Docker  
`curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg`
- Ajout du dépôt stable aux sources  
`echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null`
- Mise à jour apt  
`sudo apt update`
- Installation des paquets Docker  
`sudo apt-get install -y docker-ce docker-ce-cli containerd.io`
- Ajout de l'utilisateur courant au groupe docker  
`sudo usermod -aG docker $USER`

# Image Docker

- Image Docker = plusieurs couches en lecture seule (RO)
- Couche = différence au niveau du système de fichiers
- Création d'un nouveau conteneur = ajout d'une fine couche inscriptible
- Les changements faits dans un conteneur en cours d'exécution sont écrits dans cette couche
- Conteneur détruit = couche inscriptible détruite. L'image sous-jacente reste inchangée
- De multiples conteneurs peuvent partager l'accès (lecture) à la même image sous-jacente tout en disposant de leurs propres données
- Le pilote de stockage Docker gère les couches d'image et la couche inscriptible des conteneurs

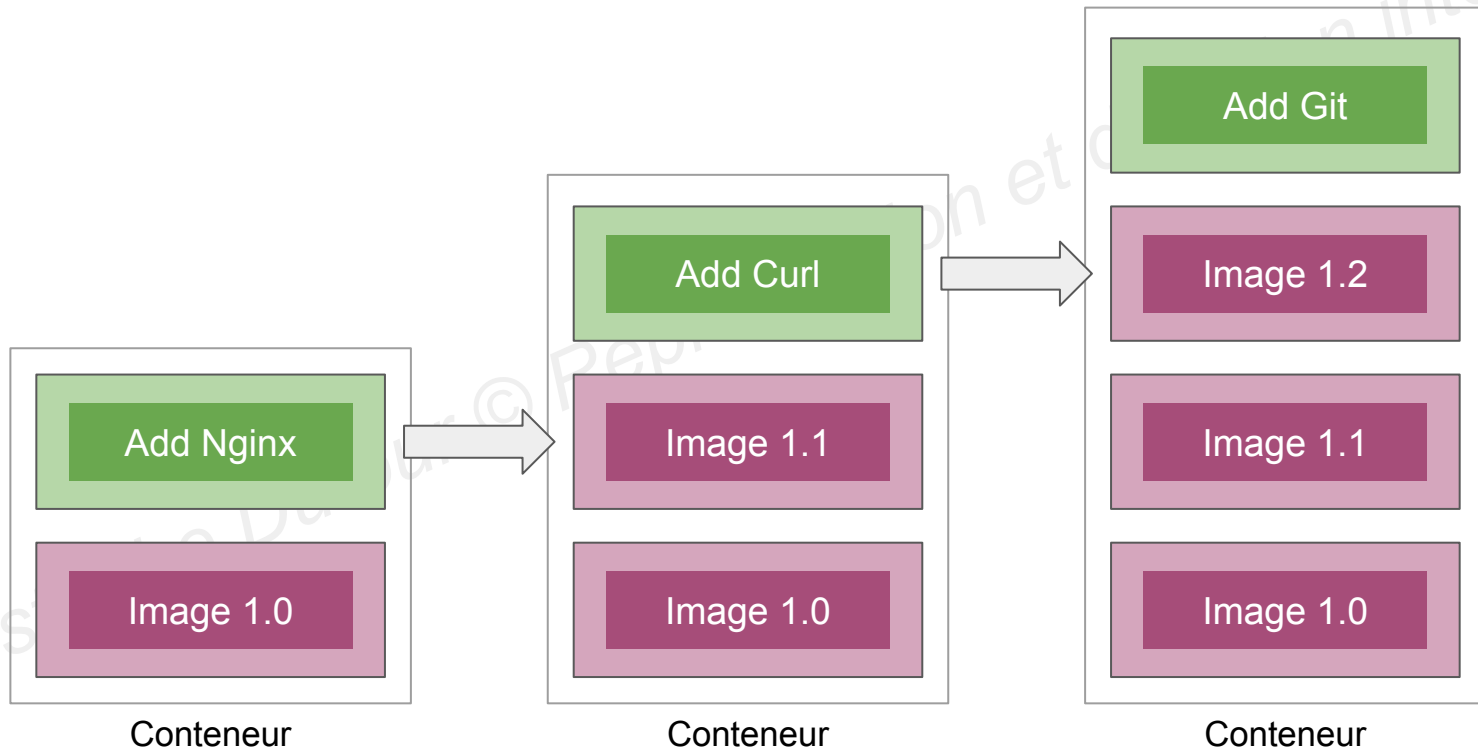
# Image Docker (suite)

- La couche inscriptible est aussi petite que possible
- Chaque couche est placée dans la zone de stockage local de Docker
- Les dossiers sont accessibles, sur linux: `/var/lib/docker/<storage-driver>`
- Les noms de dossier ne correspondent pas aux IDs d'image

```
root@opusidea:~# ls /var/lib/docker/overlay2/  
005131a79ee51bc2be153df8e78ab368e1089de1cb09c48853d1ad07a3ead90e  
ff89802f3eaf324fb7c5fb7b267dc3af596cc8eb6f5c44eaf95780b6dfcd13bd  
03b81a2ddbc5a1f42387b1ea09ccff0676a4dfe3749e846e6a8e51d0faceecf5  
0508a08a902a816d46cd375379e468f4ad4642b0ace33c5b1a54b9fdf20dbcb0
```

```
root@opusidea:/var/lib/docker/overlay2/ff89802f3eaf324fb7c5fb7b267dc3af596cc8eb6f5c44eaf95780b6dfcd13bd# ls  
committed diff link lower work
```

# Conteneurs et couches (layers)



# Docker run - exemple

docker

docker cli

run

commande

busybox

image

echo Ciao !

commande de  
démarrage

```
vagrant@ubuntu-bionic:~$ docker run busybox echo Ciao !
Ciao !
vagrant@ubuntu-bionic:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
1161506f27e5	busybox	"echo Ciao !"	39 seconds ago	Exited

# Docker run - exemple

docker run

=

docker create

+

docker start

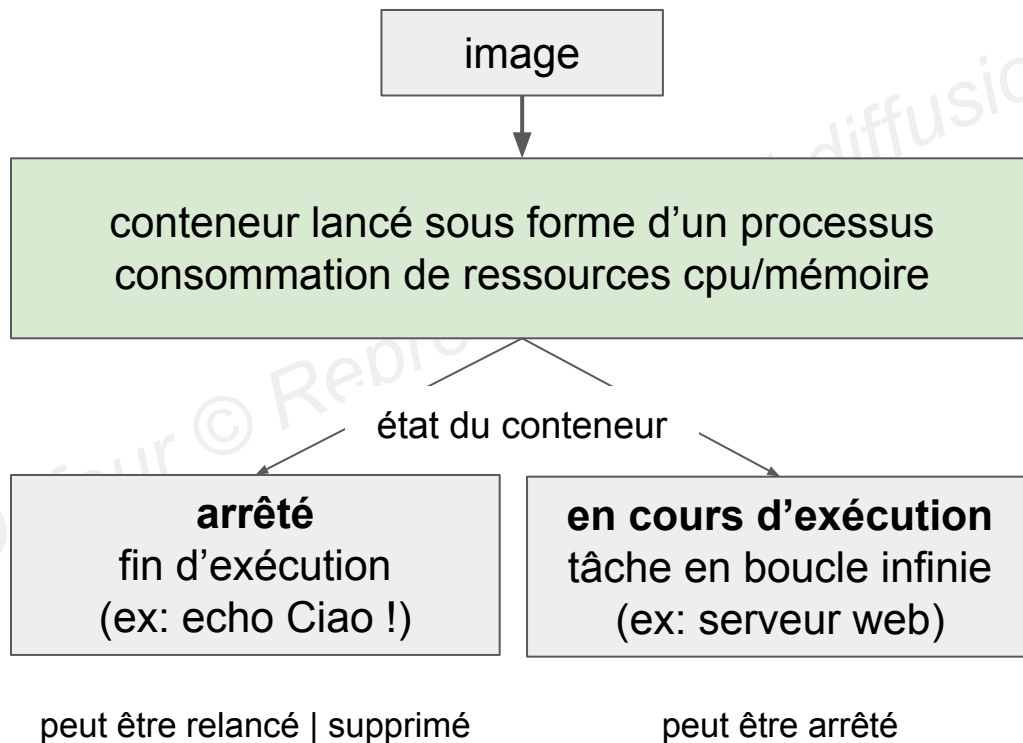
```
vagrant@ubuntu-bionic:~$ docker create busybox echo Coucou !
b189fa6a73fe4ca09ea51e3e19e1405d19befc5e2de781e3b5d0f366a3a53376
vagrant@ubuntu-bionic:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
b189fa6a73fe	busybox	"echo Coucou !"	4 seconds ago	Created

```
vagrant@ubuntu-bionic:~$ docker start -a b189
Coucou !
vagrant@ubuntu-bionic:~$ docker start -a b189
Coucou !
vagrant@ubuntu-bionic:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
b189fa6a73fe	busybox	"echo Coucou !"	About a minute ago	Exited

# Fonctionnement





# Commandes - gestion des conteneurs

Commande	Description
<code>docker create image [ commande ]</code> <code>docker run image [ commande ]</code>	créer le conteneur = create + start
<code>docker start conteneur...</code> <code>docker stop conteneur...</code> <code>docker kill conteneur...</code> <code>docker restart conteneur...</code>	démarre le conteneur arrêt "doux" (SIGTERM) arrêt "brutal" (SIGKILL) = stop + start
<code>docker pause conteneur...</code> <code>docker unpause conteneur...</code>	suspend l'exécution du conteneur reprend l'exécution du conteneur
<code>docker rm [-f] conteneur...</code>	détruit le conteneur = docker kill + docker rm

# Commandes - inspection des conteneurs

Commande	Description
<code>docker ps</code> <code>docker ps -a</code>	affiche les conteneurs en cours d'exécution affiche tous les conteneurs (all)
<code>docker logs conteneur</code>	affiche la sortie du conteneur (stdout + stderr)
<code>docker top conteneur</code>	liste les processus en cours dans le conteneur
<code>docker diff conteneur</code>	montre les différences avec l'image
<code>docker inspect conteneur...</code>	affiche les infos bas-niveau (format json)

# Commandes - interaction avec les conteneurs

Commande	Description
<code>docker attach conteneur</code>	attache le terminal actif au conteneur (stdin/out/err)
<code>docker cp conteneur:chemin hôte:chemin</code> <code>docker cp hôte:chemin conteneur:chemin</code>	copie des fichiers depuis le conteneur copie des fichiers dans le conteneur
<code>docker export conteneur</code>	exporte le contenu du conteneur (archive tar)
<code>docker exec conteneur args...</code>	exécute une commande dans le conteneur
<code>docker wait conteneur</code>	attend que le conteneur se termine (exit code)
<code>docker commit conteneur image</code>	commit une nouvelle image docker (snapshot du conteneur)

# Commandes - gestion des images

Commande	Description
<code>docker images</code> <code>docker history image</code> <code>docker inspect image</code>	liste toutes les images locales affiche l'historique de l'image (ses calques) affiche les infos bas-niveau (format json)
<code>docker tag image tag</code>	tague une image
<code>docker commit conteneur image</code>	crée une image à partir d'un conteneur
<code>docker import chemin</code>	créer une image à partir d'un tarball
<code>docker rmi image...</code>	supprime l'image

# Commandes - transfert d'image

Commande	Description
<code>docker pull repo[:tag]...</code> <code>docker push repo[:tag]...</code> <code>docker search text</code>	télécharge une image/repo depuis un registre envoie une image/repo vers un registre recherche une image dans le registre officiel
<code>docker login</code> <code>docker logout</code>	se connecte à un registre se déconnecte d'un registre
<code>docker save repo[:tag]...</code> <code>docker load</code>	exporte une image/repo en tant que tarball charge une image depuis un tarball

# Créer ses propres images

- Docker permet d'utiliser des images faites par d'autres (depuis le registre docker.hub par exemple)
- On peut aussi créer des images personnalisées
- Avantages: conteneurs "sur-mesure" disposant de fonctionnalités précises
- Etapes de création d'une image
  - rédaction d'un Dockerfile
  - construction de l'image (docker build -f dockerfile)

# Anatomie d'une image: les couches (layers)

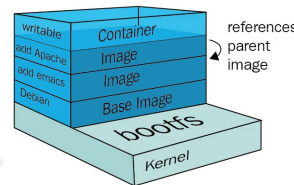


Image Couche v3 démarrée comme conteneur, accessible par les utilisateurs

0a4114a14ded

Lecture/écriture  
Couche conteneur

1. Image Couche v1 démarrée comme conteneur
2. Serveur http configuré et démarré ("yum install...")
3. Nouvelle couche v2 produite(committed)

bf2feb98a4f5

Image en lecture seule  
Couche v2

1. Image de base (docker.io/centos) démarrée comme conteneur
2. Paquets de l'image de base mis à jour (yum update)
3. Nouvelle couche v1 produite (committed)

d62409846594

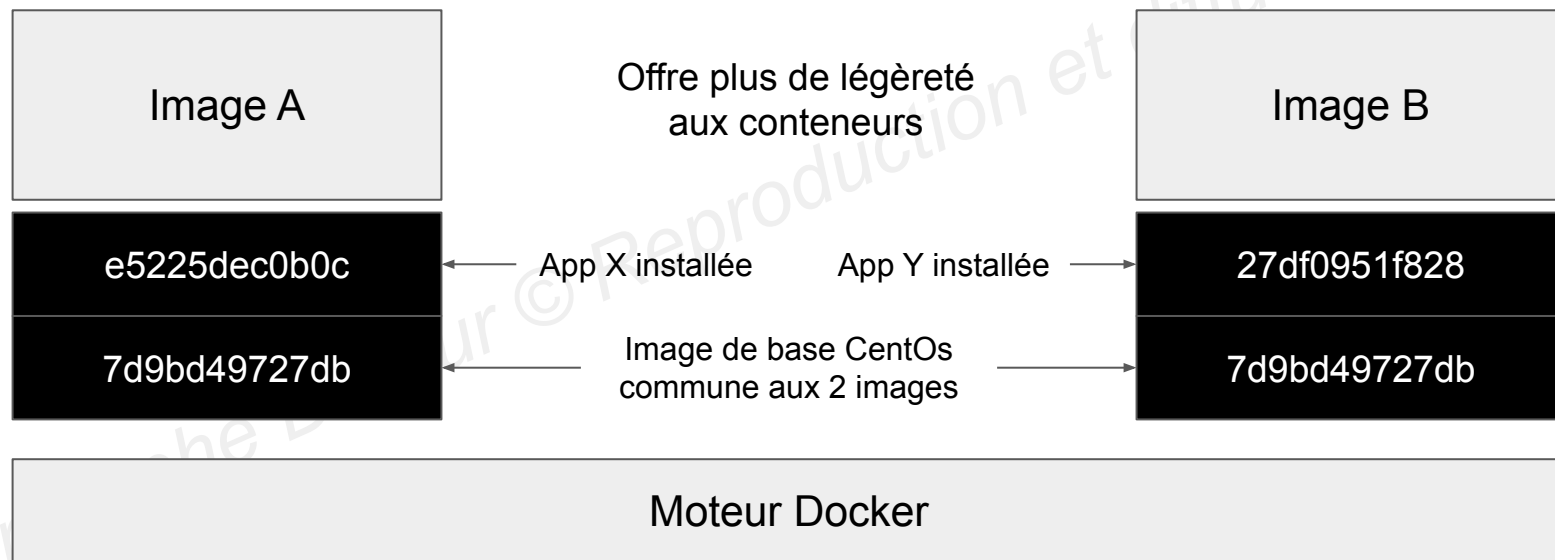
Image en lecture seule  
Couche v1

Image CentOS téléchargée depuis Docker Hub par "docker pull". **Repo: docker.io/centos**

7d9bd49727db

Image de base

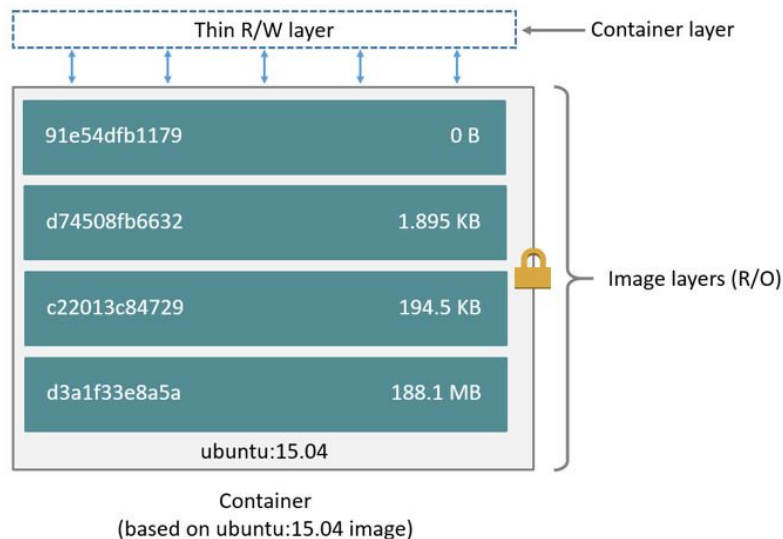
# Mutualisation des couches





# Union File System

- Union des contenus de différents système de fichiers
- La couche la plus haute remplace tout fichier similaire trouvé dans les systèmes
- Docker implémente le pilote de stockage **overlay2** (plus performant que l'ancien pilote **aufs**)

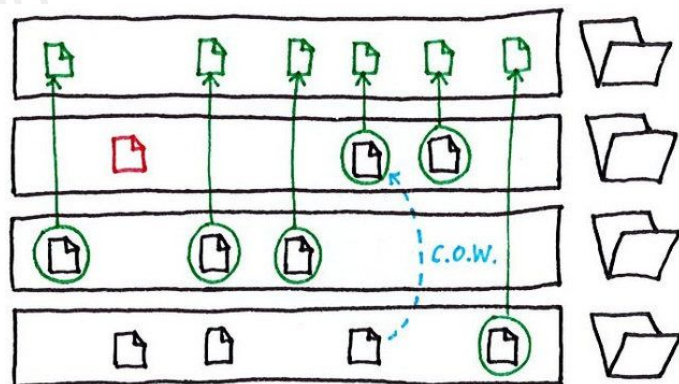


MERGED

UPPER DIR

LOWER DIR

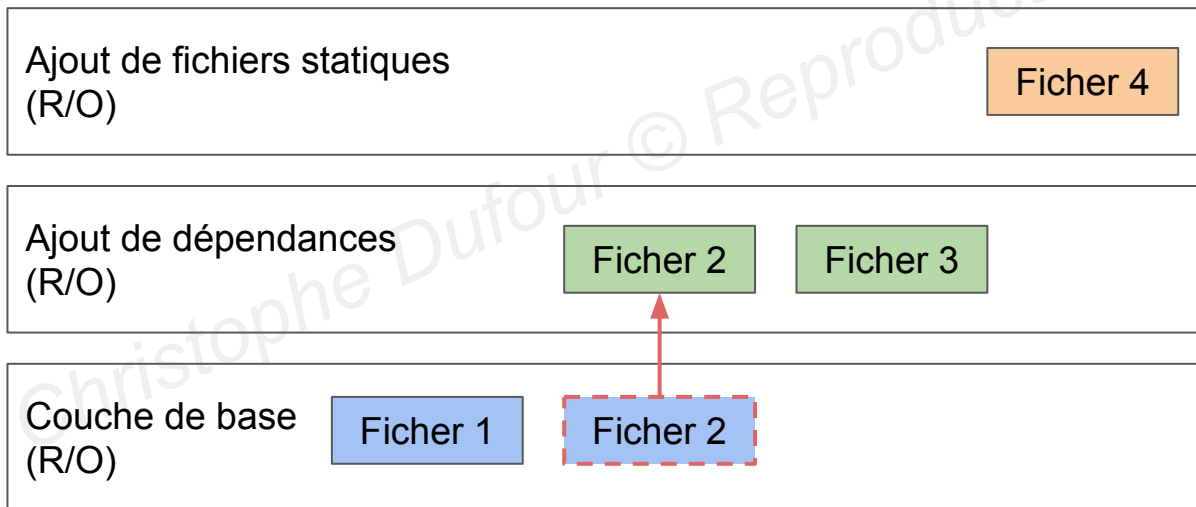
LOWER DIR



# Copy-on-write (COW)

Docker utilise la stratégie COW afin de partager et copier les fichiers efficacement.

Si un conteneur utilise un fichier/dossier disponible dans une des couches inférieures, l'accès en lecture est direct. S'il s'agit de le modifier, le fichier/dossier est d'abord copié dans la couche cible puis modifié.



La seconde couche veut modifier **Fichier 2**, présent dans la couche de base. Ce fichier est donc copié puis modifié



# Dockerfile

- Fichier texte définissant les règles de construction d'une image
- Toute commande modifiant le système de fichier produit une nouvelle couche

FROM

ARG

WORKDIR

RUN

ADD | COPY

ENV

CMD

ENTRYPOINT

EXPOSE

# Dockerfile

Modifie les méta-données de l'image, pas le FS, aucune couche additionnelle produite


Modifie le FS, produit une couche additionnelle

Modifie les méta-données de l'image, pas le FS, aucune couche additionnelle produite

```
FROM ubuntu:18.04
LABEL author="chris@oi.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

The diagram illustrates the effects of various Dockerfile instructions. A central box on the right contains the Dockerfile code. Three arrows point from this box to three separate boxes on the left. The first arrow points from the 'FROM' instruction to the top box, which states it only modifies metadata. The second arrow points from the 'COPY' and 'RUN' instructions to the middle box, which states it modifies the filesystem and produces a new layer. The third arrow points from the 'CMD' instruction to the bottom box, which states it only modifies metadata.

# Dockerfile: création



Spécifier une image de base

Exécuter quelques commandes  
d'installation de programmes  
additionnels

Spécifier une commande à exécuter  
au démarrage

```
FROM alpine:3.12
RUN apk update
RUN apk add redis
CMD ["redis-server"]
```

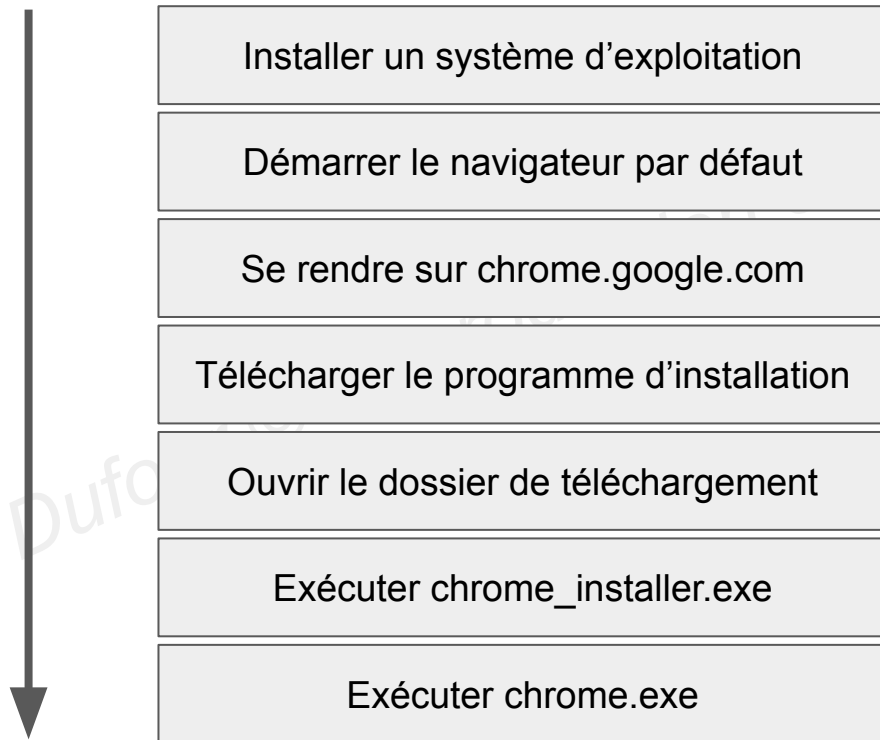
# Exercice

Créer une image démarrant un serveur  
Redis et incluant le logiciel bash

# Exercice: solution

FROM	alpine
RUN	apk add --update redis
RUN	apk add bash
CMD	["redis-server"]

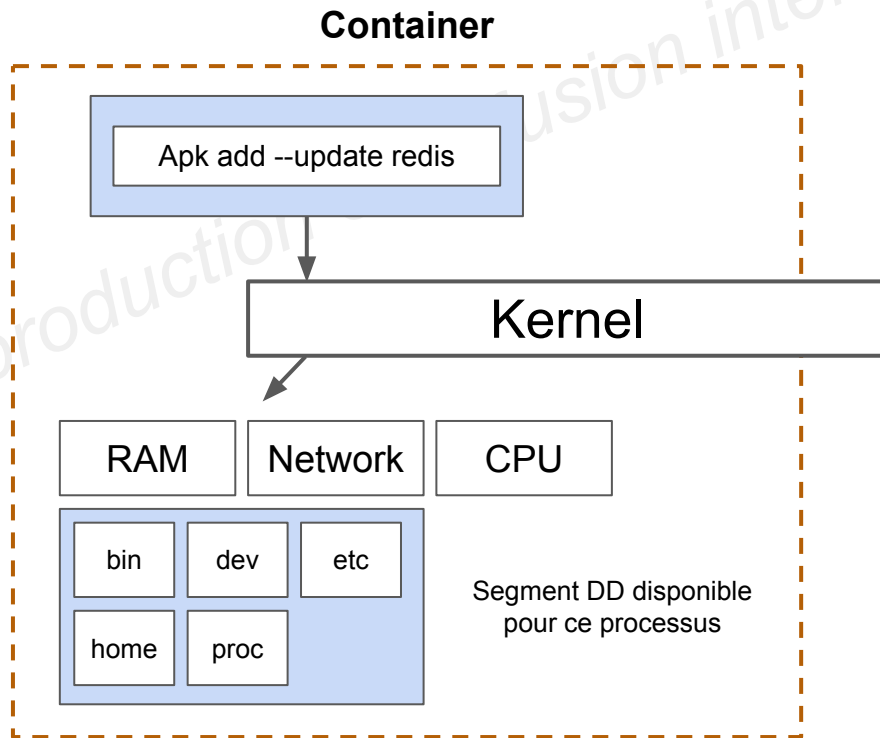
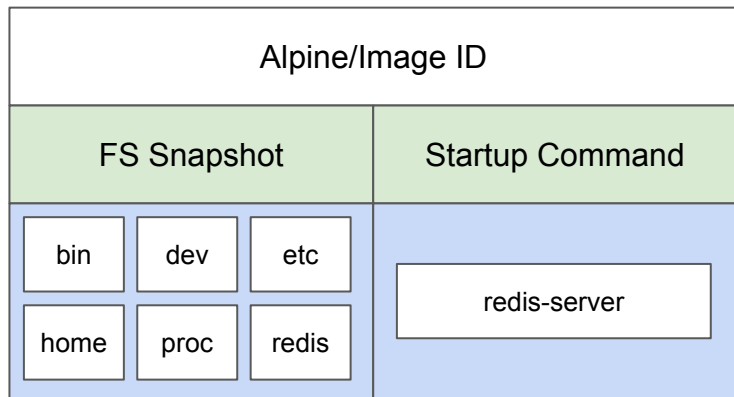
# Exemple d'installation de chrome





# Build process

FROM	alpine
RUN	Apk add --update redis
CMD	["redis-server"]



# Atelier

“Conteneuriser” un serveur nodejs/express

# Atelier: solution

```
import express from 'express';

const app = express();

app.get('/', (req, res) => {
  res.send('Je suis un serveur nodejs/express');
})

app.listen(3000, () => {
  console.log('Serveur écoutant le port 3000...');
})
```

```
docker build . -t opusidea/simpleweb:v1
```

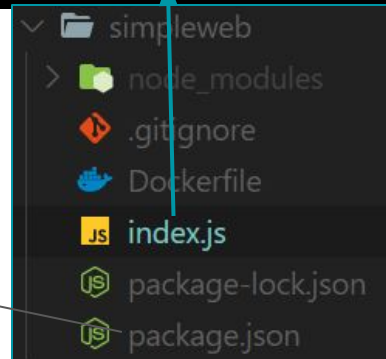
```
docker run --rm -p 3000:3000 opusidea/simpleweb:v1
```

```
FROM node:14.17-alpine

COPY index.js .
COPY package.json .

RUN npm install

CMD ["node", "index.js"]
```

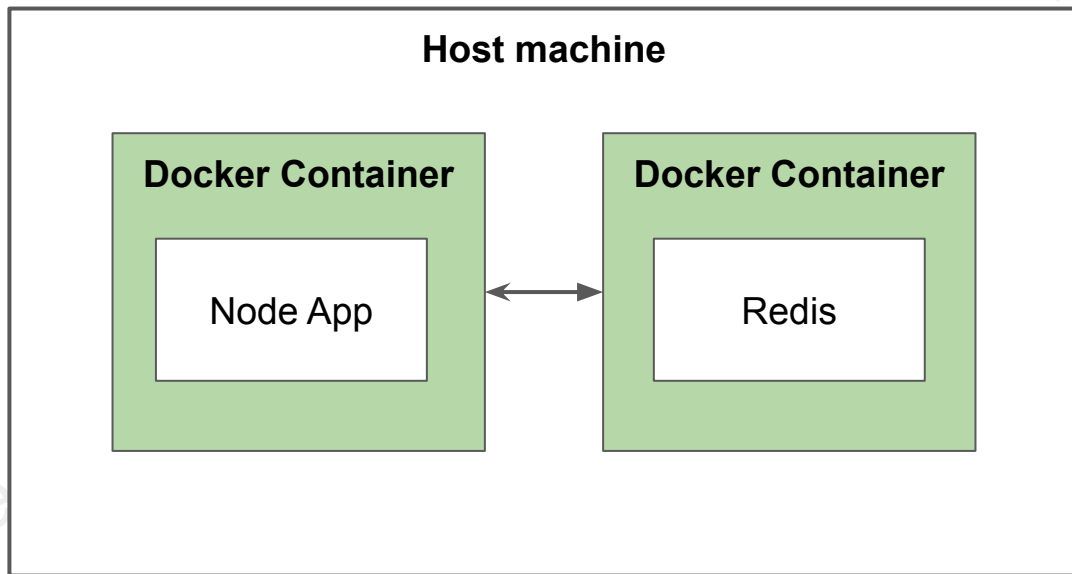


# Application multi-conteneurs

Créons une application nodejs communiquant avec un serveur Redis

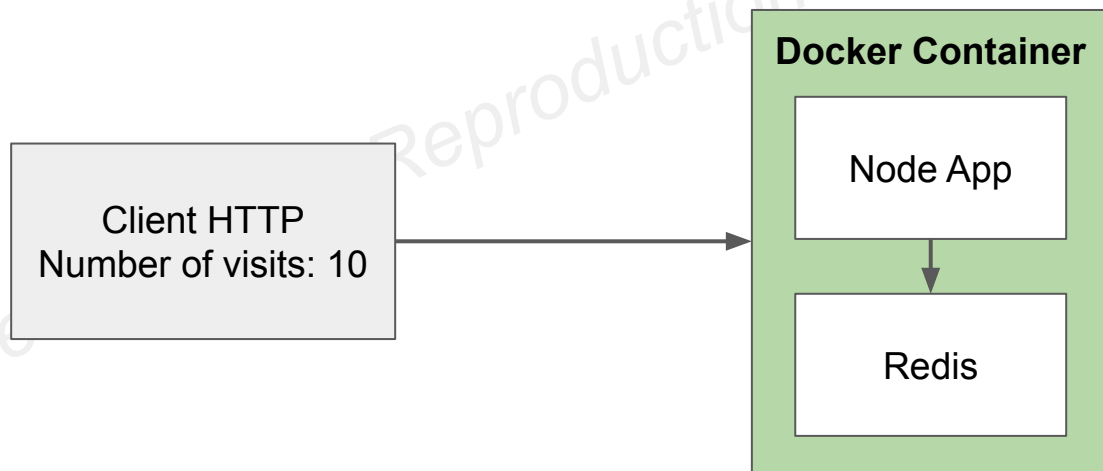


# Application multi-conteneurs



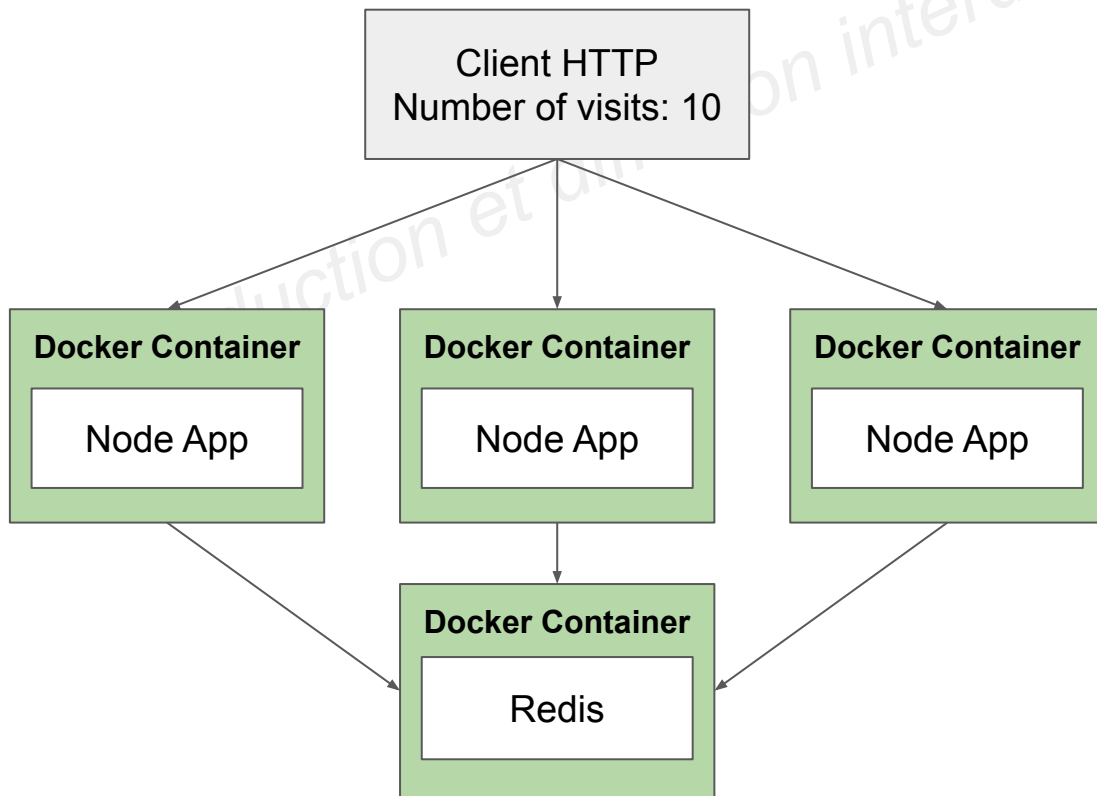
# Application multi-conteneurs

Les serveurs nodejs et redis sont encapsulés dans le même conteneur. Ce manque de découplage est dangereux en cas de problème rencontré par une application. Limite la mise à échelle (scalability).



# Application multi-conteneurs

Les serveurs nodejs et redis sont encapsulés dans des conteneurs différents. Ce découplage permet une meilleure résilience à la faille, une mutualisation de la couche donnée, il favorise aussi la mise à échelle (scalability)

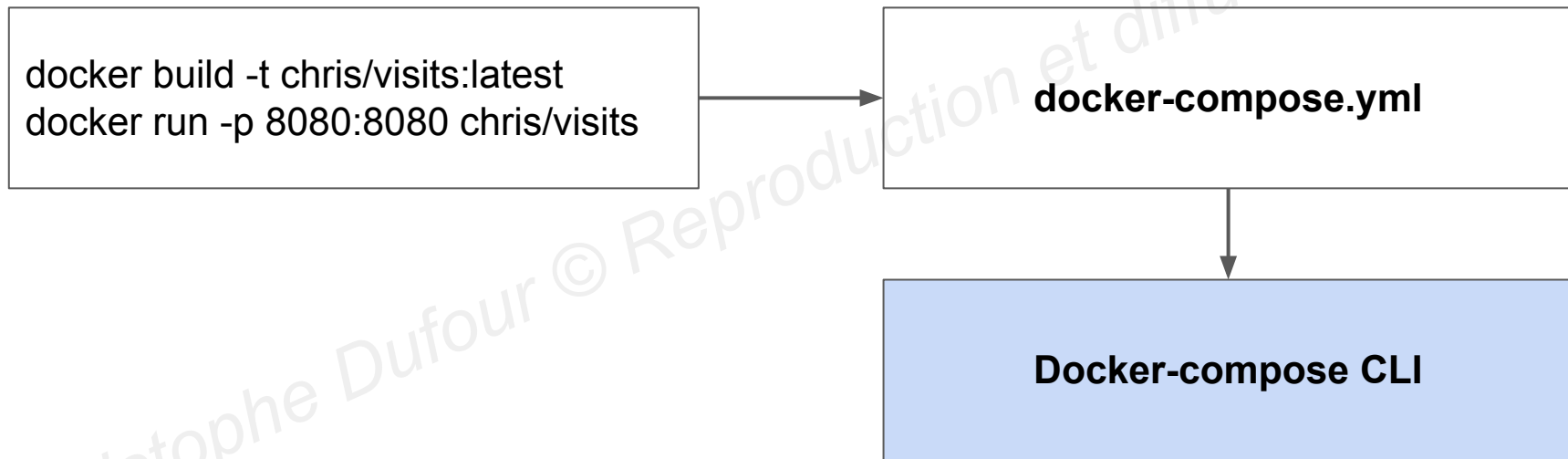


# Docker Compose

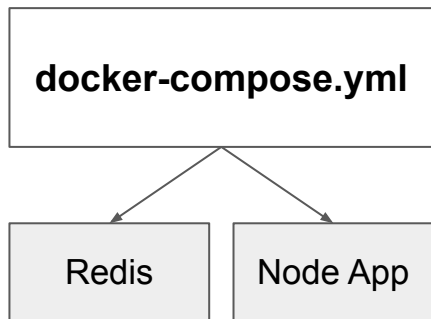
- CLI distinct du docker CLI
- Utilisé pour gérer de multiples conteneurs en même temps
- On parle de service
- Permet de configurer des infrastructures multi-services complexes
- Fichier texte au format YAML (diffusable, partageable, versionable, etc.)
- Toutes les fonctionnalités Docker sont configurables:
  - mapping de port
  - variables d'environnement
  - création de volume
- Docker-compose crée un réseau privé auquel les conteneurs (services) sont attachés



# Docker Compose



# Docker Compose



Le fichier docker-compose définit deux services.  
Le service node-app est joignable de l'extérieur sur le port 4001.

Le service redis-app n'est pas joignable de l'extérieur. L'application node y accède par le nom du service (host: redis-app)

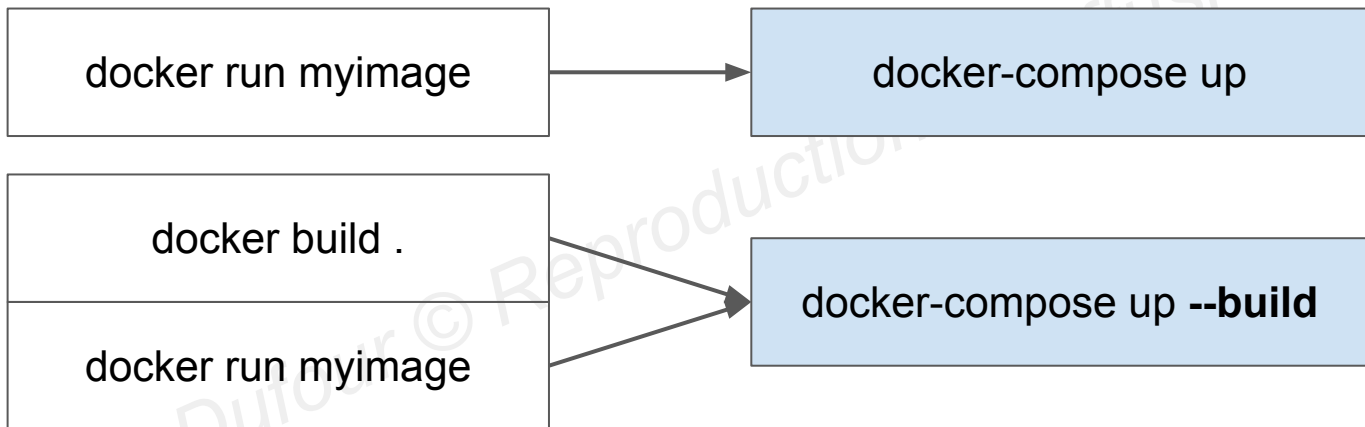
```
version: '3'
services:

  redis-app: # dns name pour l'application node
    image: 'redis'

  node-app:
    build: . # cible le Dockerfile (dossier courant)
    ports:
      - "4001:8081"
```

```
const client = redis.createClient({
  host: 'redis-app',
  port: 6379
});
```

# Docker Compose



# Docker Compose

## Démarrage en tâche de fond

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

## Arrêts des conteneurs

```
docker-compose down
```

## Liste (depuis dossier contenant .yaml)

```
docker-compose ps
```

# Conteneur stateless vs conteneur stateful

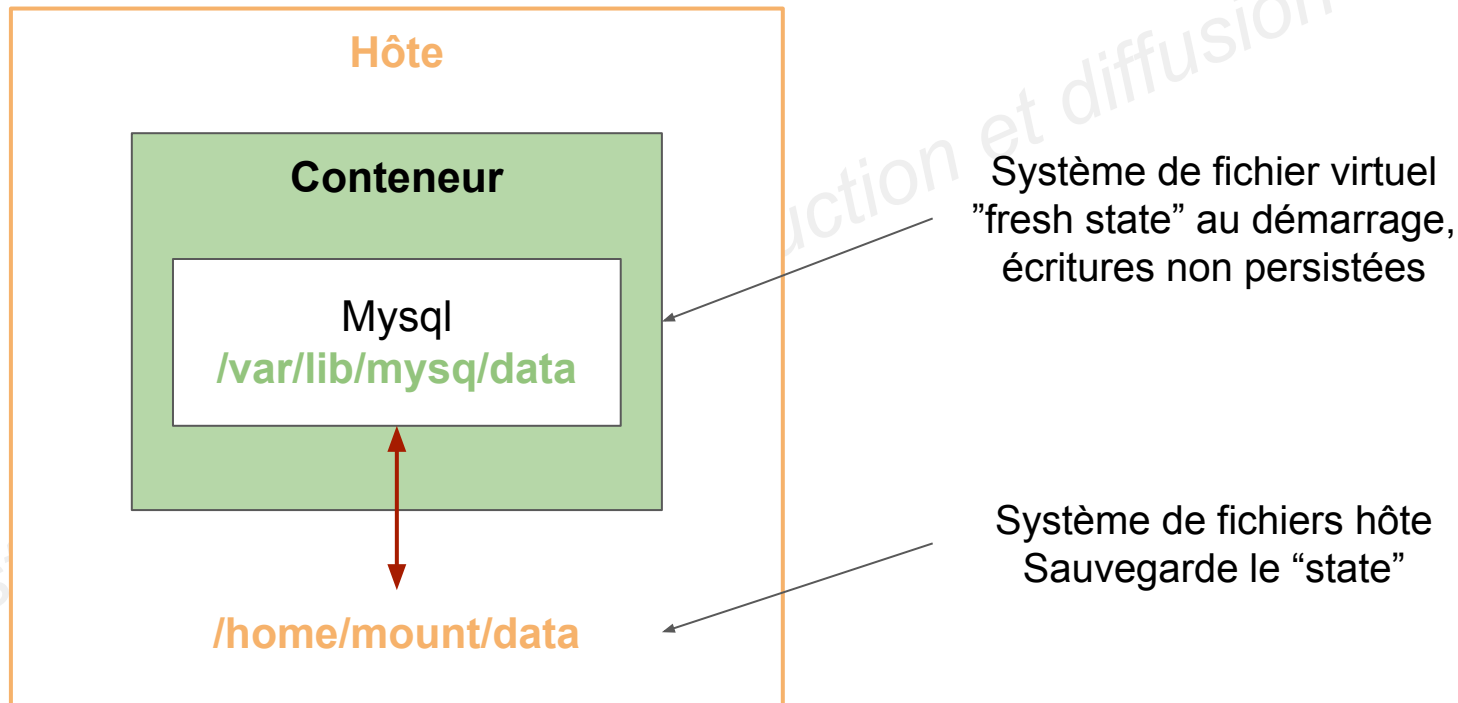
- Dès l'origine, les conteneurs ont vocation à travailler sans état, c'est-à-dire, sans mémoire, sans trace ou historique des transactions passées, à la manière d'une fonction pure, gage de forte mobilité
- Avec la popularisation des conteneurs, et l'encapsulation d'applications fonctionnant en mode stateful (nécessité de connaître un état passé - exemple, contenu de précédentes requêtes), le besoin d'un état sauvegardé s'est accru
- Solutions pour apporter un état à un conteneur
  - bind mount
  - persistent volume

# Docker Volumes

- Utile pour les applications “stateful” (ex: bases de données)
- Permet de persister des données hors conteneur
- Permet de synchroniser des données entre le conteneur et le hôte
- Offre un stockage partagé entre plusieurs conteneurs
- Améliore les performances pour les opérations d'écriture intensives\*

*\*Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers.*

# Docker Volumes

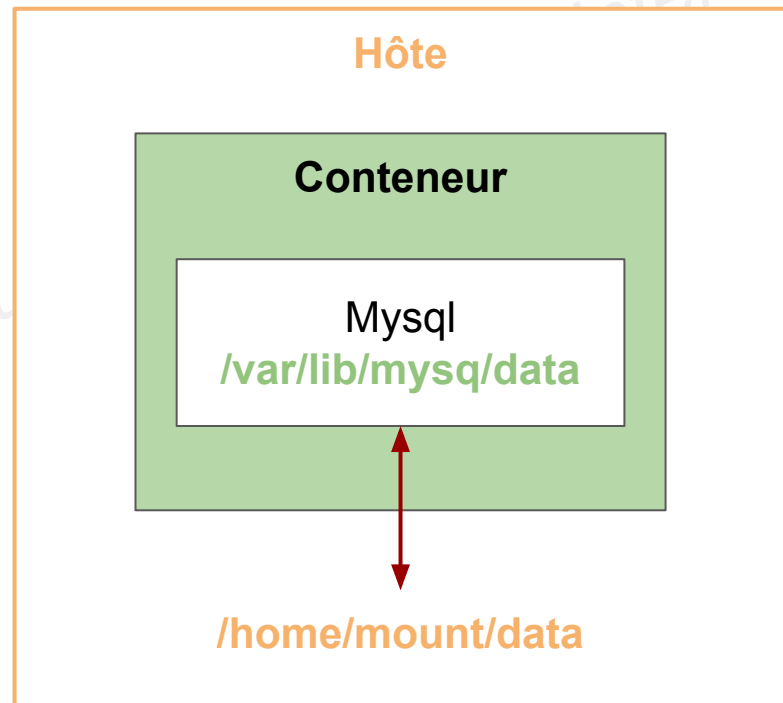


# 3 types de volumes

docker run

-v /home/mount/data:/var/lib/mysql/data

Volume hôte (type: bind)



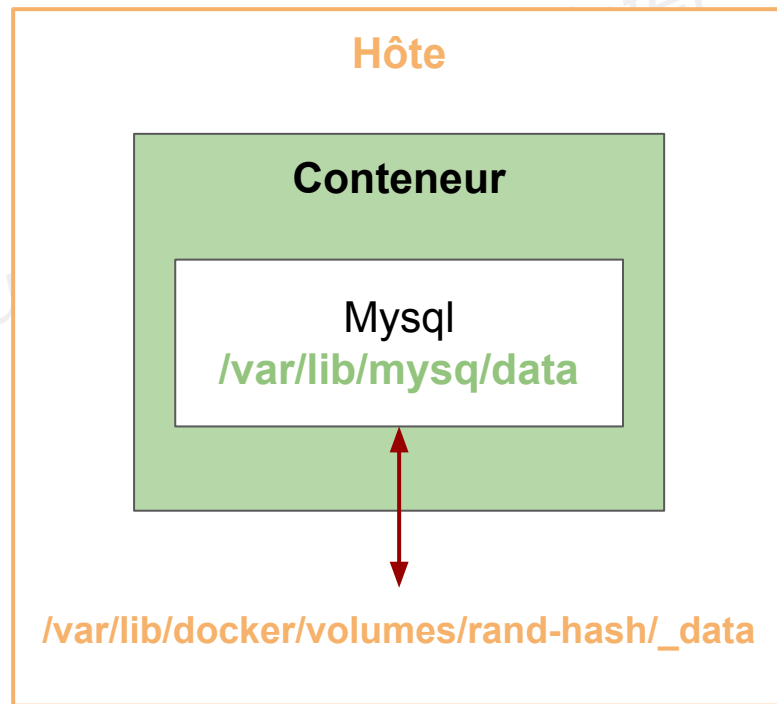


# 3 types de volumes

docker run

-v /var/lib/mysql/data

Volume anonyme



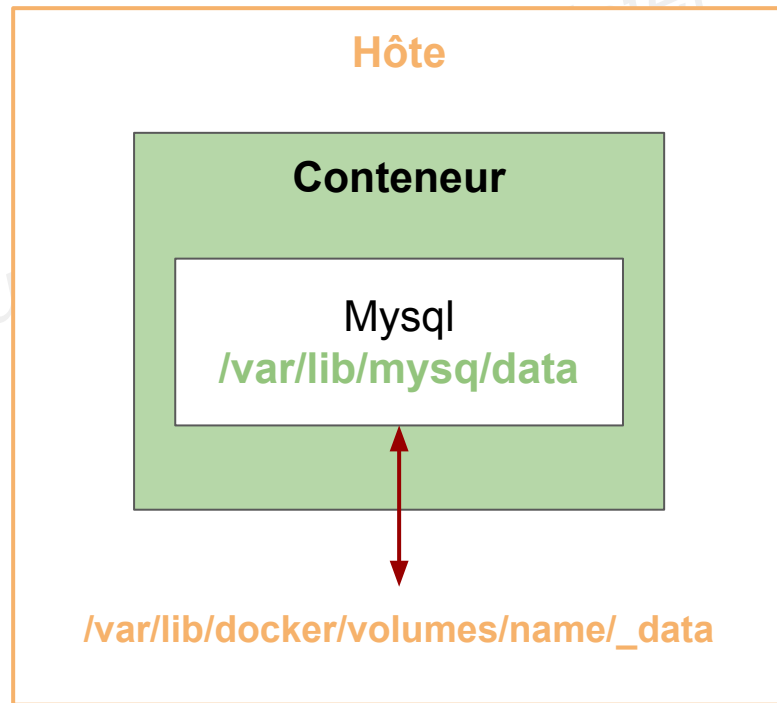
automatiquement créé par Docker

# 3 types de volumes

docker run

-v name:/var/lib/mysql/data

Volume nommé



automatiquement créé par Docker

# Volumes dans docker-compose

Volume  
nommé



```
version: '3'
# default path storage for mongo:
/db/data
services:
  mongodb:
    image: 'mongo:latest'
    ports:
      - 27017:27017
    volumes:
      - db-data:/db/data
volumes:
  db-data:
```

## Partage de volume

Une même référence à un volume peut être utilisée par plusieurs conteneurs

# Docker Volumes

```
docker run -p 3000:3000 -v /app/node_modules -v $(pwd):/app cid
```

```
docker volume ls
```

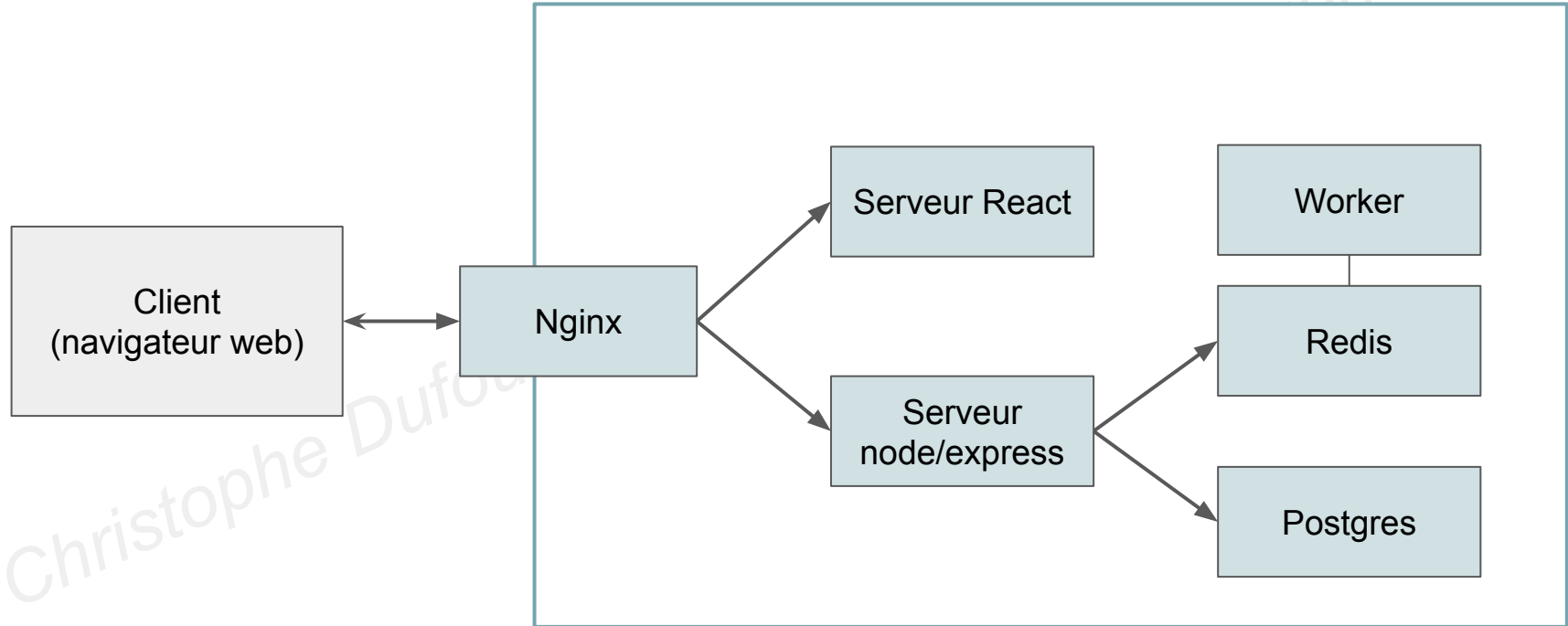
# Docker Network

```
docker network create mynetwork --subnet 172.50.0.0/16
```

```
docker run -tid --name toto--network mynetwork redis:5
```

```
docker run -tid --name tata --network mynetwork redis:5
```

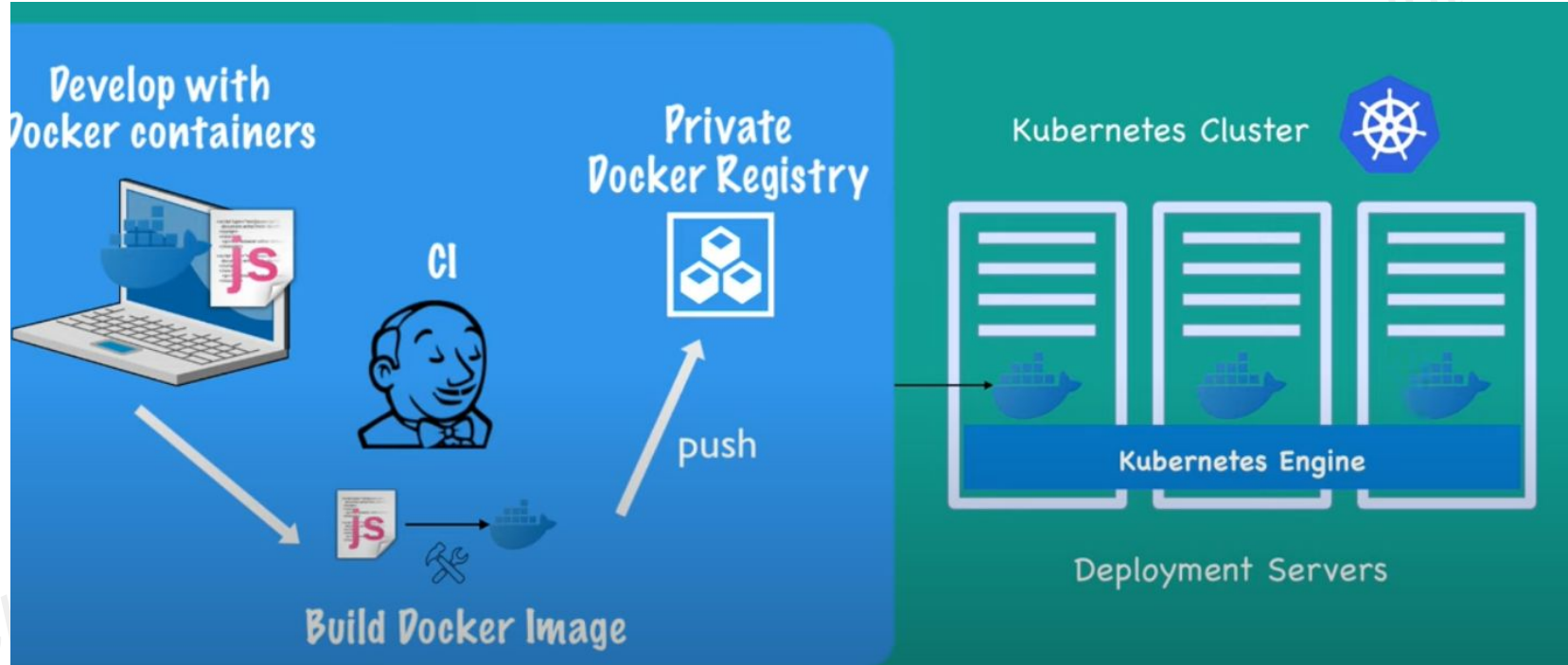
# TP “Complex”: infrastructure multi-services



# Docker vs K8S

Docker	Kubernetes
Conteneurs, environnement isolé pour applications	Infrastructure pour gérer de multiples conteneurs
Construction et déploiement automatisés d'applications - CI	Planification et gestion automatisés de conteneurs d'applications (micro-services)
Plateforme de conteneurisation pour configurer, construire et distribuer des conteneurs	Ecosystème pour gérer un cluster de conteneurs (docker ou autre)

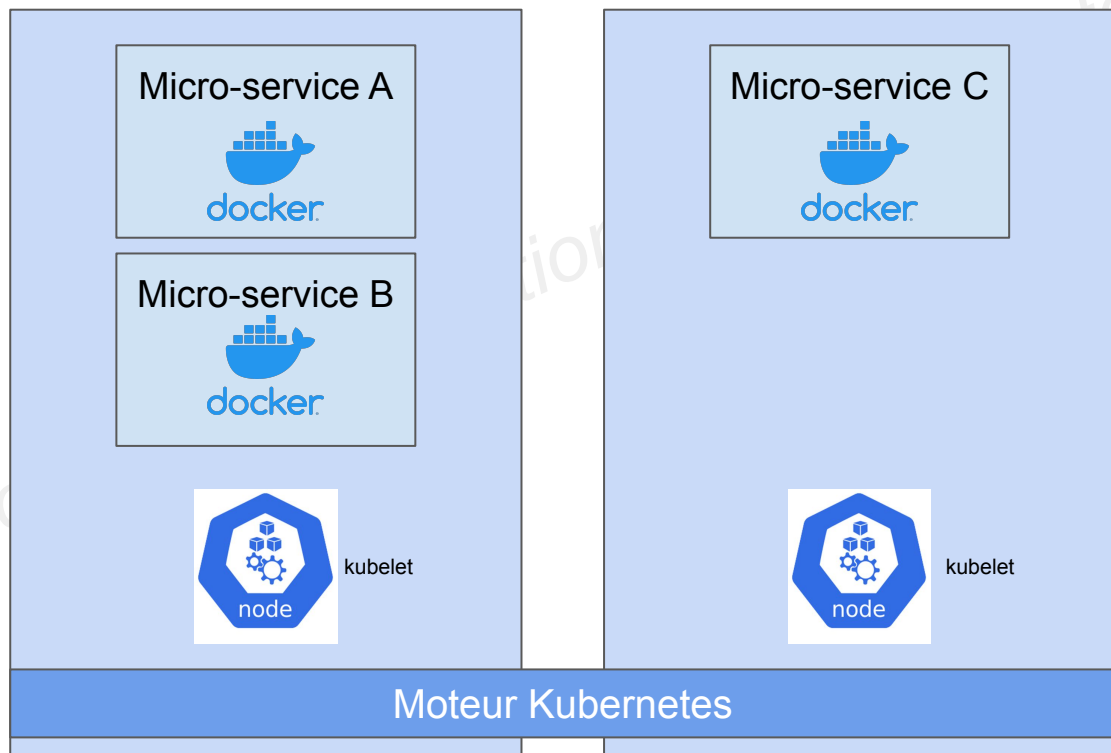
# Docker et K8S dans le processus de dév





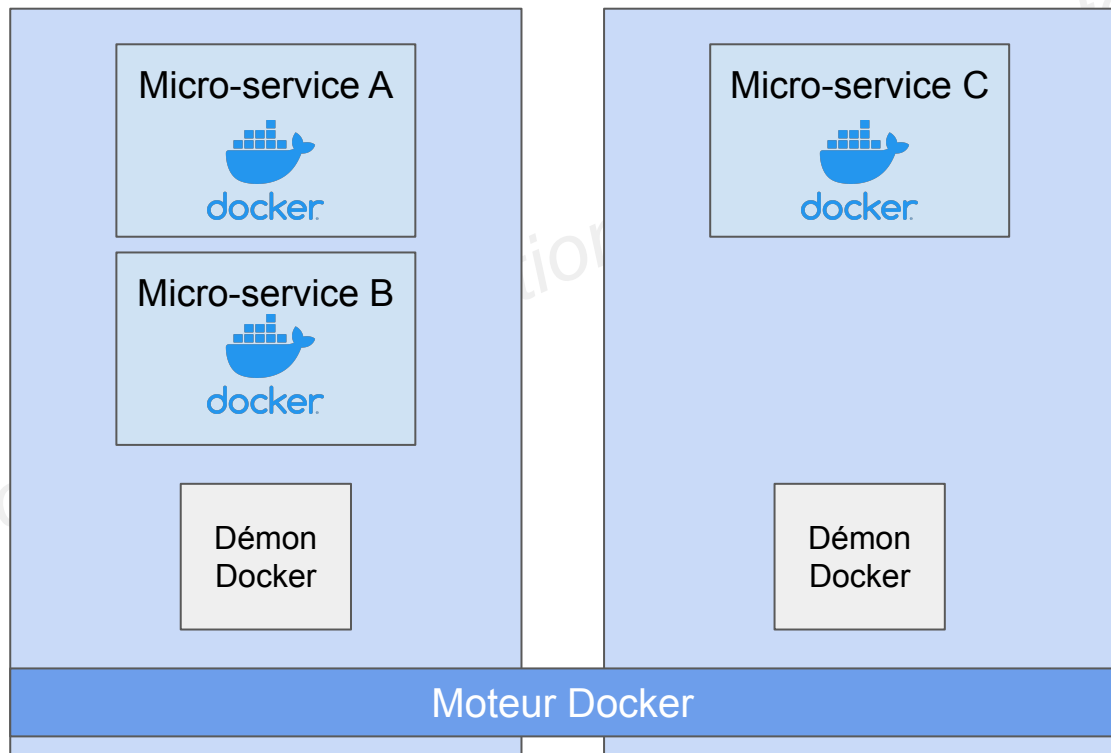
# Kubernetes

Chaque noeud k8s a un  
kubelet (agent) installé



# Docker Swarm

Chaque worker swarm a  
un démon docker  
(agent) installé



# Docker Swarm vs K8S

Swarm	Kubernetes
Installation simple	Installation complexe
Plus léger et facile d'utilisation mais limité en fonctionnalités	Plus complexe, apprentissage plus long mais plus puissant
Scaling manuel	Supporte auto-scaling
Plugins pour monitoring	Monitoring intégré
Equilibrage de charge automatique	Config manuelle du LB
Cli intégré à docker	Cli différent