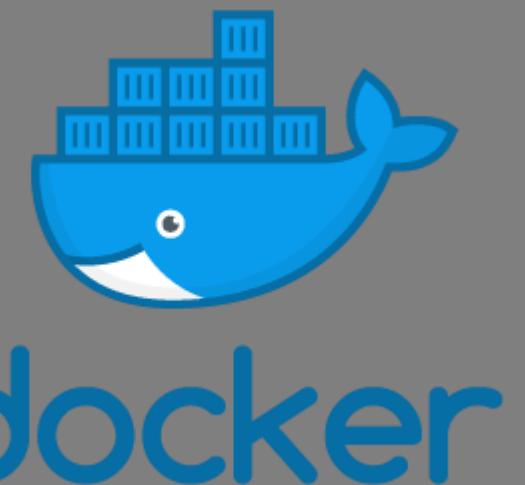


# Docker

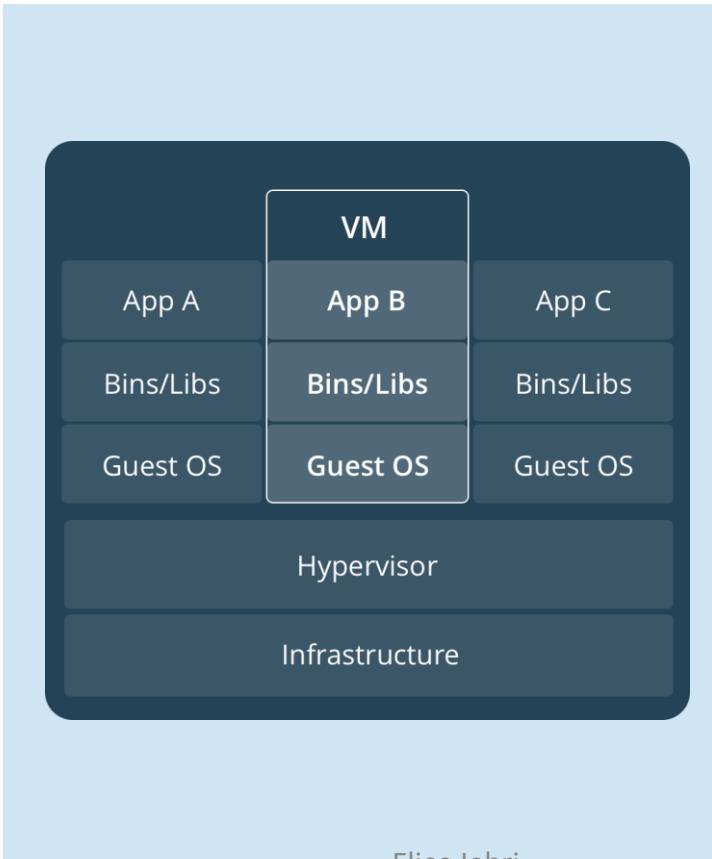
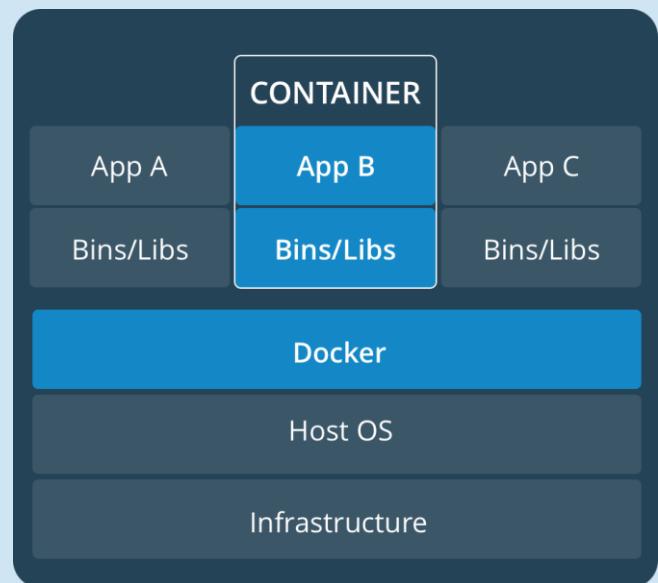
Elies Jebri



# Concepts Docker

- **Flexible** : même les applications les plus complexes peuvent être conteneurisées.
- **Léger** : les conteneurs exploitent et partagent le noyau hôte, ce qui les rend beaucoup plus efficaces en termes de ressources système que les machines virtuelles.
- **Portable** : vous pouvez créer localement, déployer sur le cloud et exécuter n'importe où.
- **Faiblement couplé** : les conteneurs sont hautement autonomes et encapsulés, ce qui vous permet de remplacer ou de mettre à niveau l'un sans en perturber les autres.
- **Évolutif** : vous pouvez augmenter et distribuer automatiquement les répliques de conteneurs dans un centre de données.
- **Sécurisé** : les conteneurs appliquent des contraintes et des isolements agressifs aux processus sans aucune configuration requise de la part de l'utilisateur.

# Conteneurs et machines virtuelles

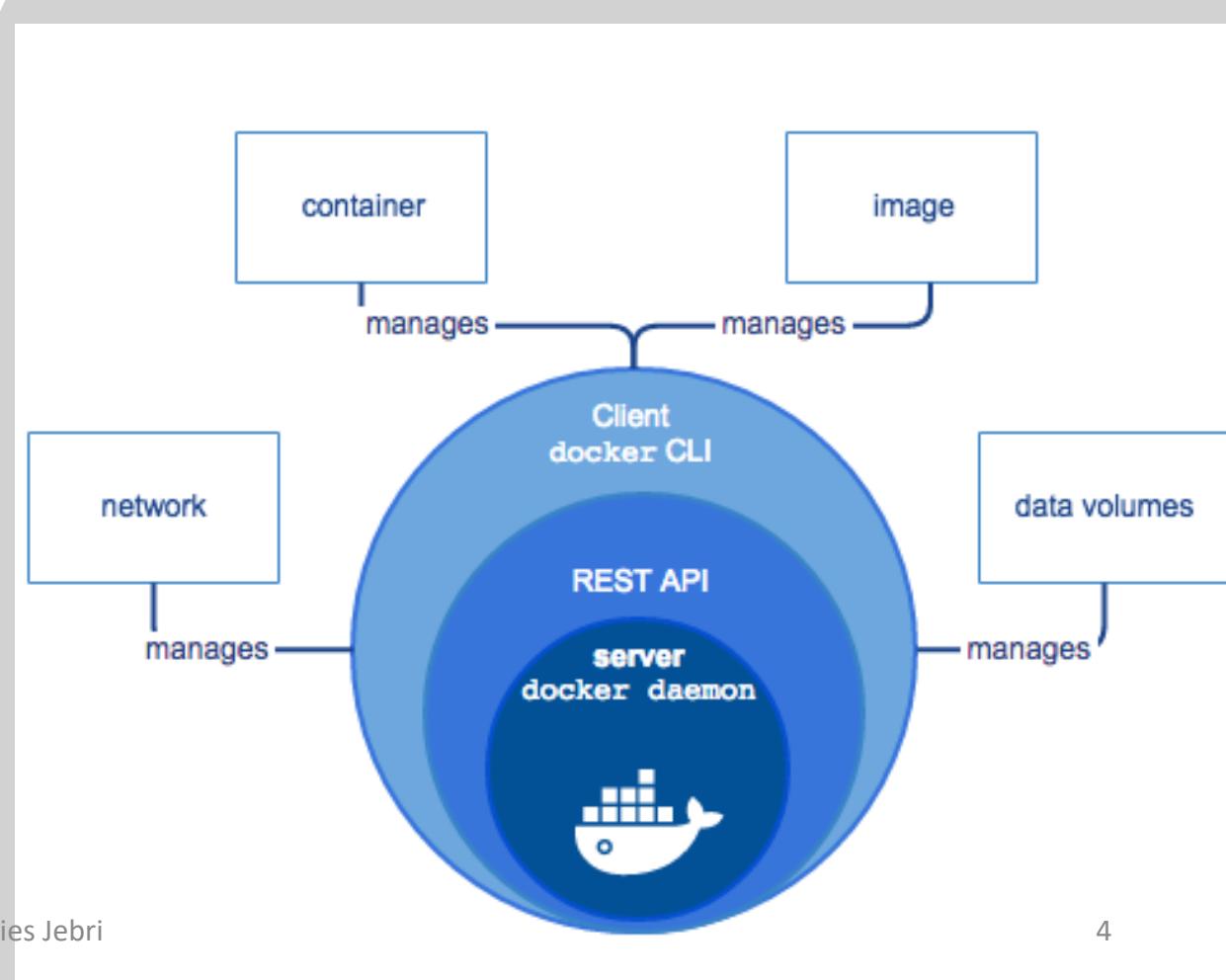


Elies Jebri

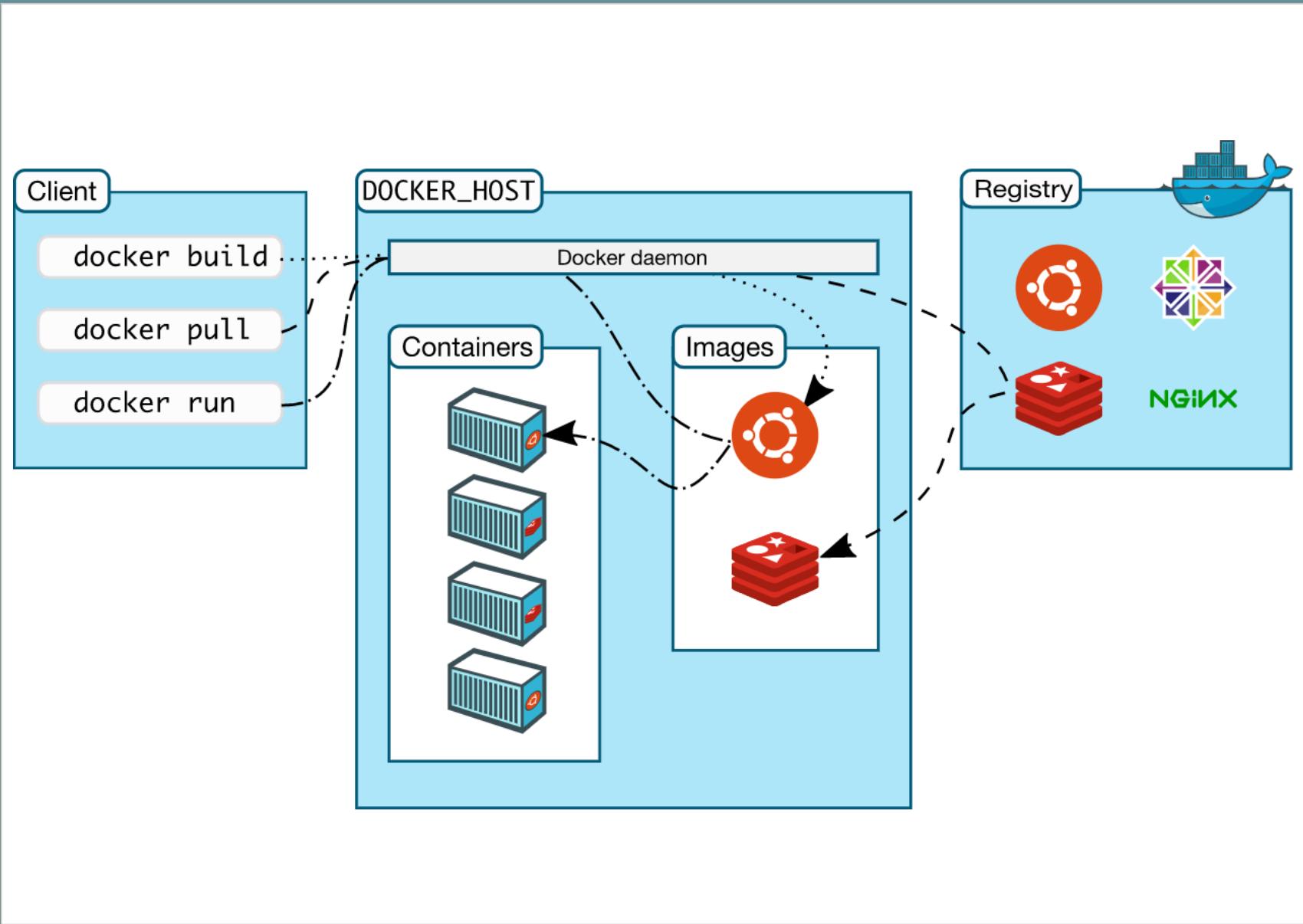
- Un conteneur s'exécute nativement sur Linux et partage le noyau de la machine hôte avec d'autres conteneurs.
- Il exécute un processus discret, ne prenant pas plus de mémoire que tout autre exécutable, ce qui le rend léger.
- Une machine virtuelle (VM) exécute un système d'exploitation «invité» à part entière avec un accès virtuel aux ressources hôte via un hyperviseur.
- En général, les machines virtuelles entraînent beaucoup de frais généraux au-delà de ce qui est consommé par la logique de votre application.

# Docker Engine

- Docker Engine est une application client-serveur avec ces composants majeurs:
  - Un serveur de type longue durée d'exécution appelé processus démon (dockerd).
  - Une API REST qui spécifie les interfaces que les programmes peuvent utiliser pour parler au démon et lui indiquer quoi faire.
  - Un client d'interface de ligne de commande (CLI) (docker).

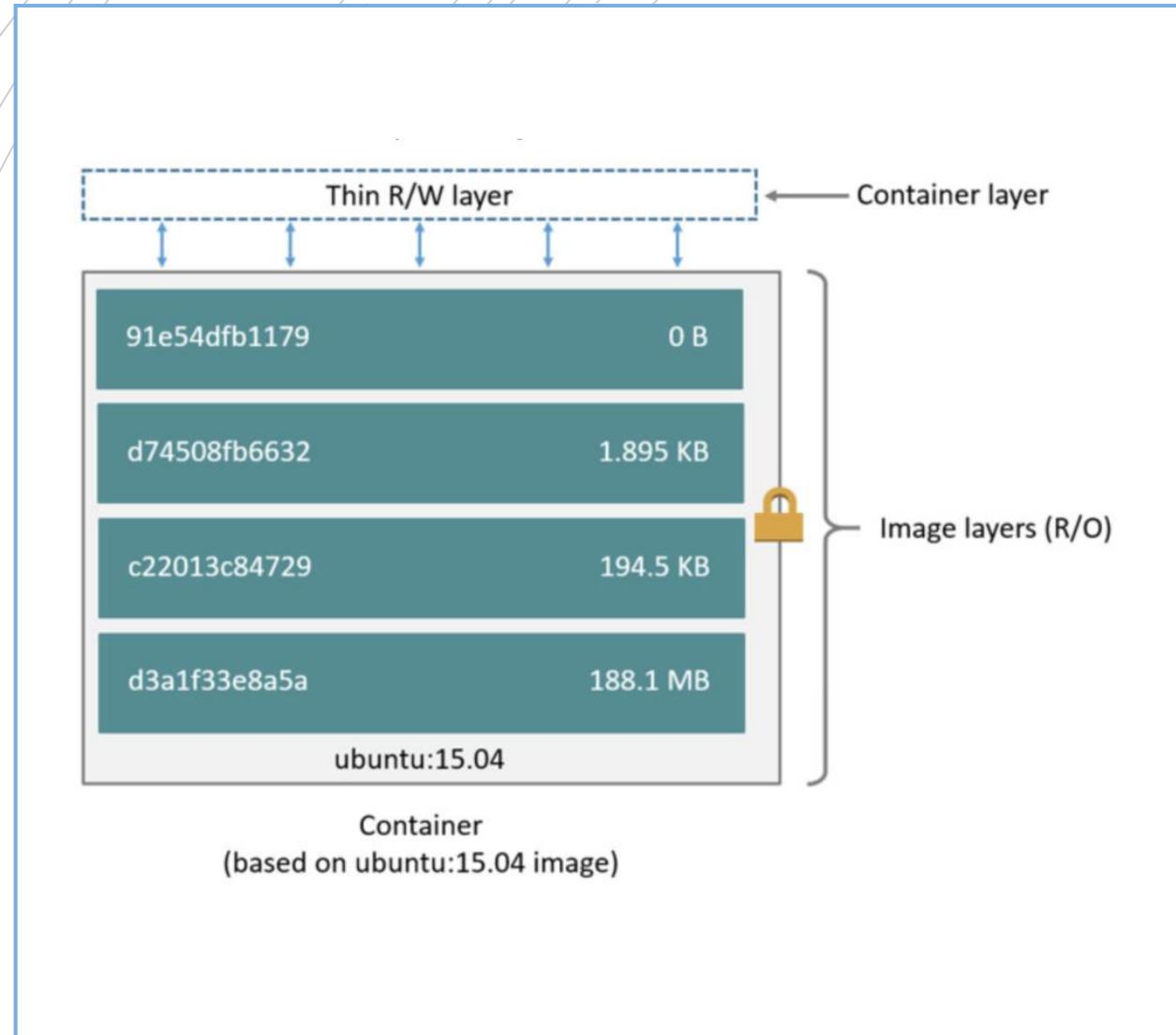


# Architecture Docker



# Images et conteneurs

- Un conteneur est un processus en cours d'exécution, avec quelques fonctionnalités d'encapsulation supplémentaires afin de le maintenir isolé de l'hôte et des autres conteneurs.
- Un aspect important de l'isolation est que chaque conteneur interagit avec son propre système de fichiers privé
- Ce système de fichiers est fourni par une image Docker .
- Une image comprend tout ce qui est nécessaire pour exécuter une application - le code ou le binaire, les environnements d'exécution, les dépendances et tout autre objet du système de fichiers requis.

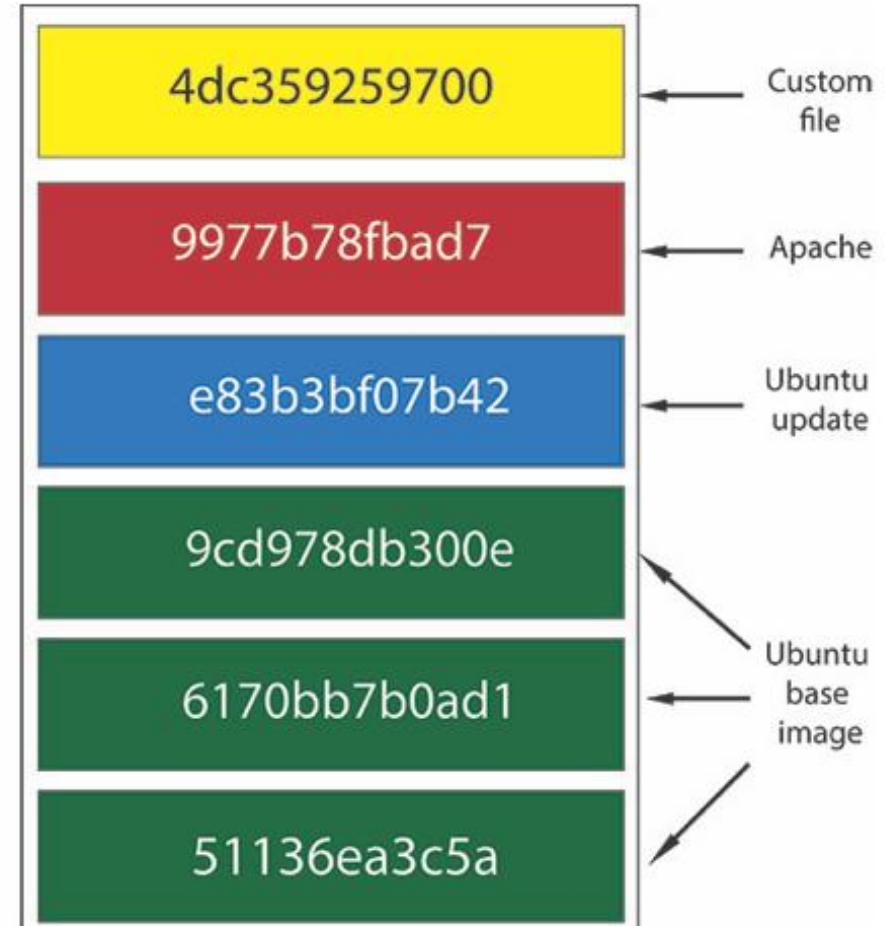


# Les images Docker

- Une image se décompose en layers
- Une image Docker fait référence à une liste de couches en lecture seule qui représentent les différences dans le système de fichiers.
- Une conteneur = une image + un layer r/w
- Une image, peut servir de base pour plusieurs conteneurs

# Layers

- La création de conteneurs d'applications à utiliser implique la création d'une image de base Docker sur laquelle certains ou tous les conteneurs d'applications sont basés.
- Les layers peuvent être réutilisés entre différents conteneurs
- Gestion optimisée de l'espace disque.
- L'utilisation d'une image de base permet de réutiliser les configurations d'image car de nombreuses applications partageront des dépendances, des bibliothèques et une configuration.

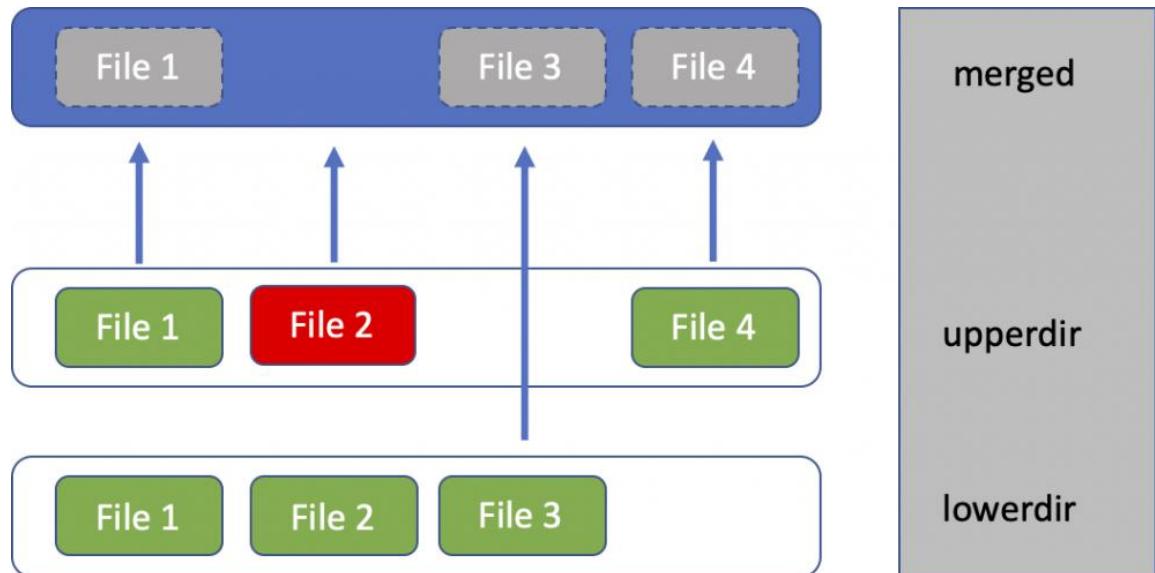


# Différence entre image et conteneur

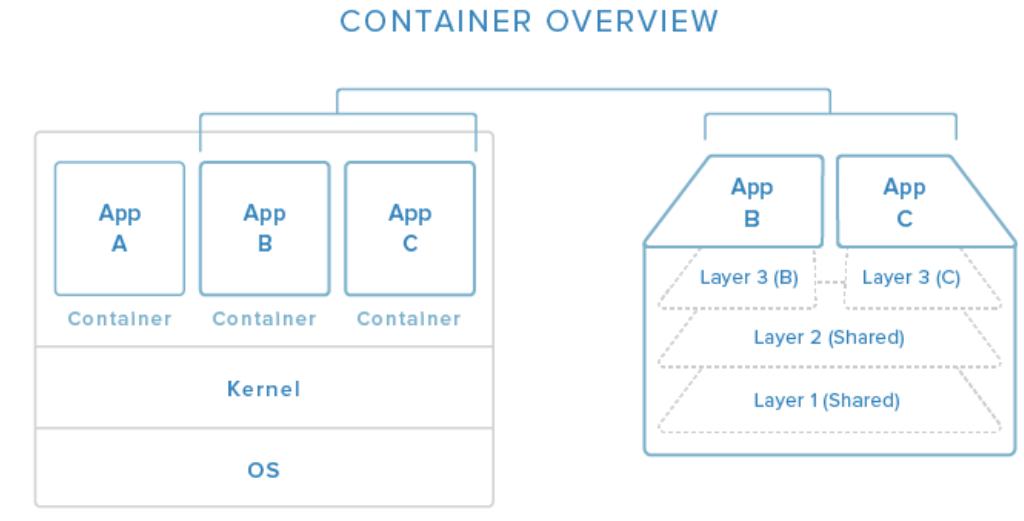
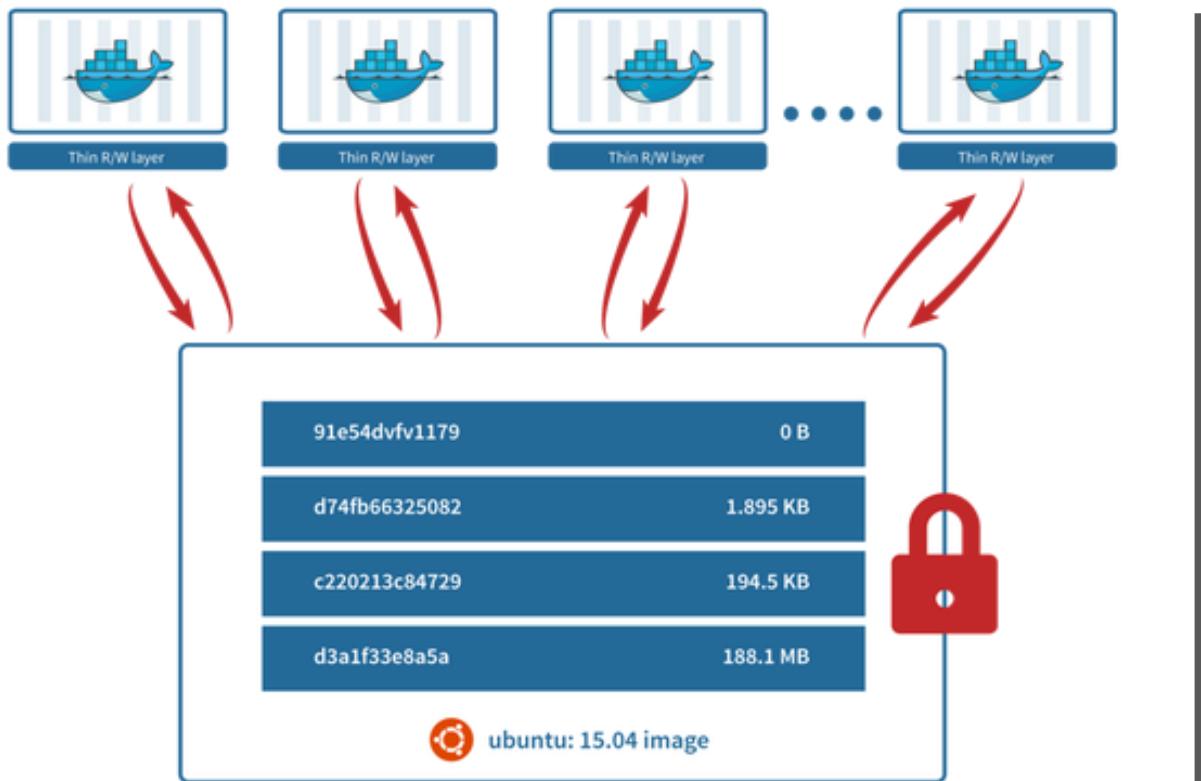
- La principale différence entre un conteneur et une image est la couche inscriptible supérieure.
- Toutes les écritures dans le conteneur qui ajoutent de nouvelles ou modifient des données existantes sont stockées dans cette couche inscriptible.
- Lorsque le conteneur est supprimé, la couche inscriptible est également supprimée.
- L'image sous-jacente reste inchangée.
- Étant donné que chaque conteneur a sa propre couche de conteneur inscriptible et que toutes les modifications sont stockées dans cette couche de conteneur, plusieurs conteneurs peuvent partager l'accès à la même image sous-jacente tout en ayant leur propre état de données

# Overlay File System

- La couche de base s'appelle « **lowerdir** » : c'est là que résident les fichiers originaux.
- Toute modification effectuée par le client sera reflétée dans le « **upperdir** » :
  - Si vous modifiez un fichier, la nouvelle version y sera écrite (Fichier 1 dans notre exemple).
  - Si vous supprimez le fichier, une marque de suppression sera créée sur ce calque (Fichier 2 dans notre exemple).
  - Si vous créez un nouveau fichier (Fichier 4 dans notre exemple), il sera placé ici.
- Enfin, « **merged** » est la vue finale de toutes les couches fusionnées



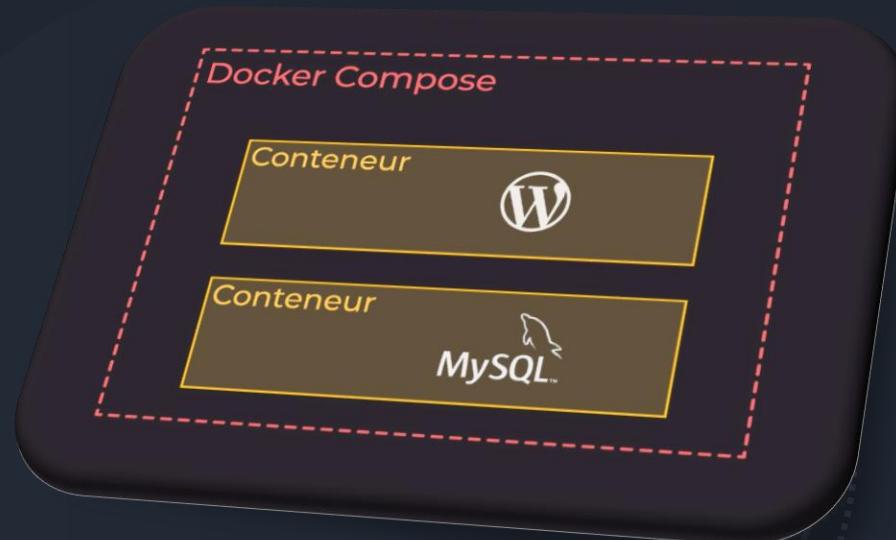
# Plusieurs conteneurs



# Composition de conteneurs

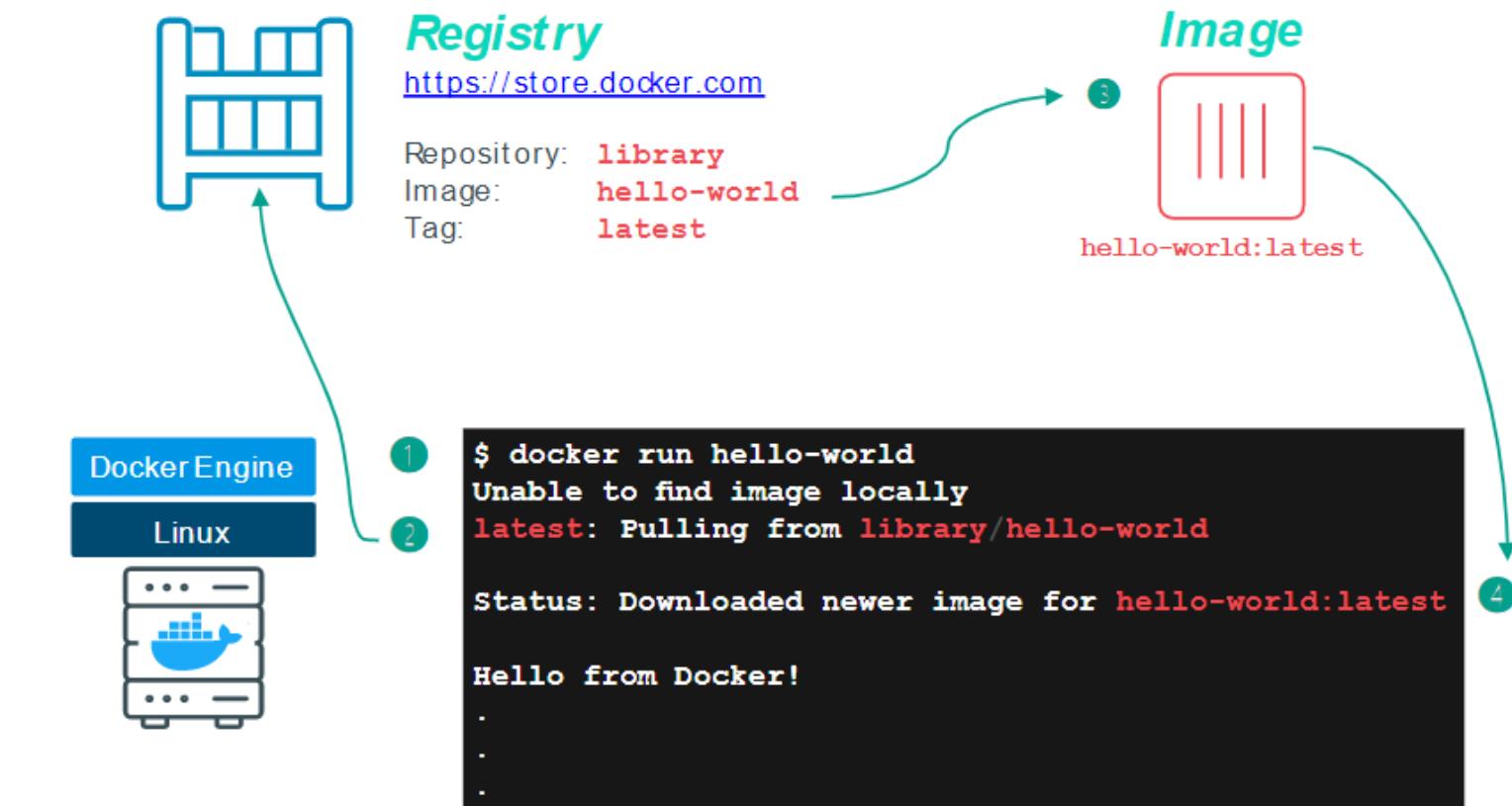
En général, le workflow de développement ressemble à ceci:

- Créez et testez des conteneurs individuels pour chaque composant de votre application en créant d'abord des images Docker.
- Assemblez vos conteneurs et votre infrastructure de support dans une application complète.
- Testez, partagez et déployez votre application conteneurisée complète.

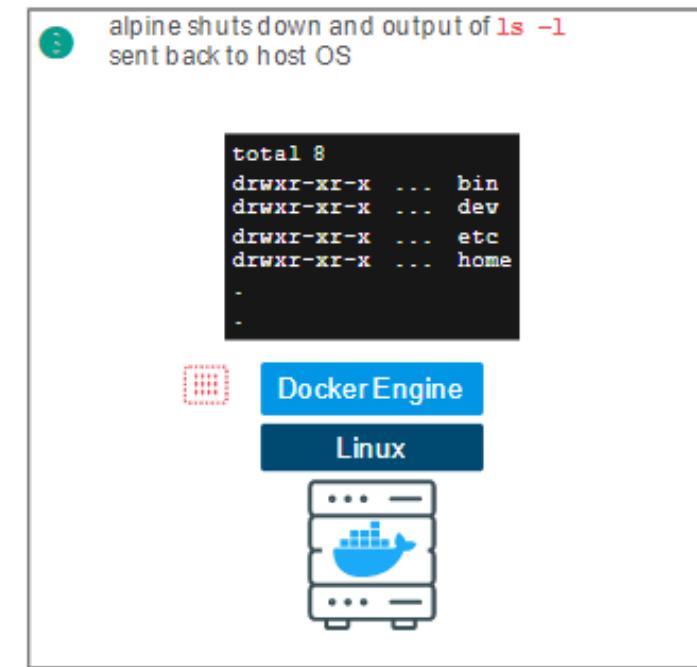
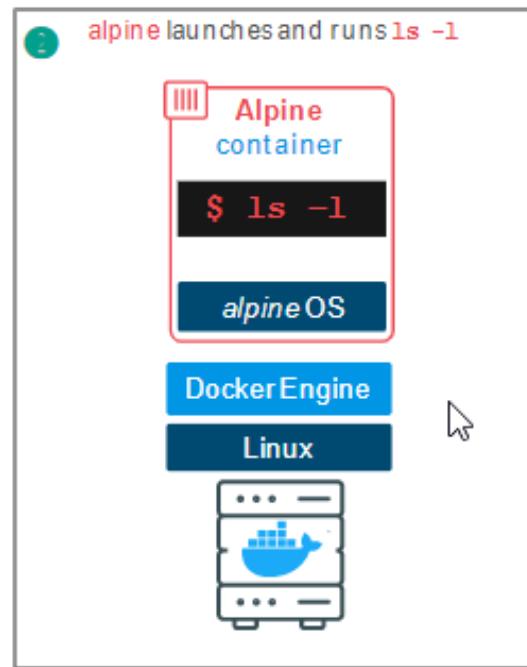
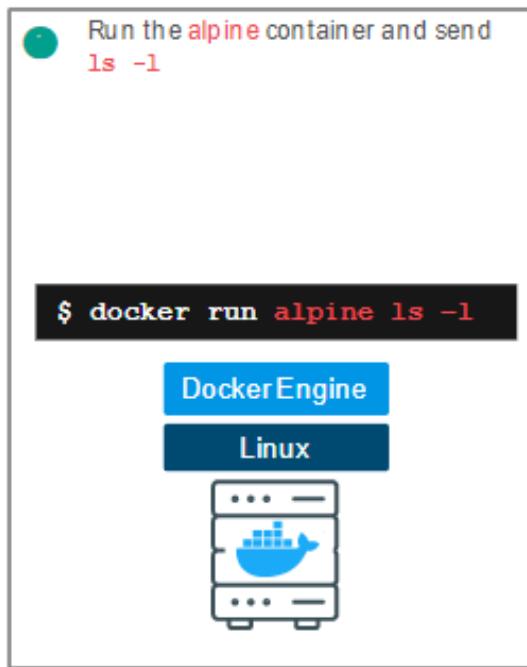


Créez et exécutez un conteneur

## Hello World: What Happened?



## *docker run* Details



# docker run

# Docker Container Instances

Output of docker container ls -a

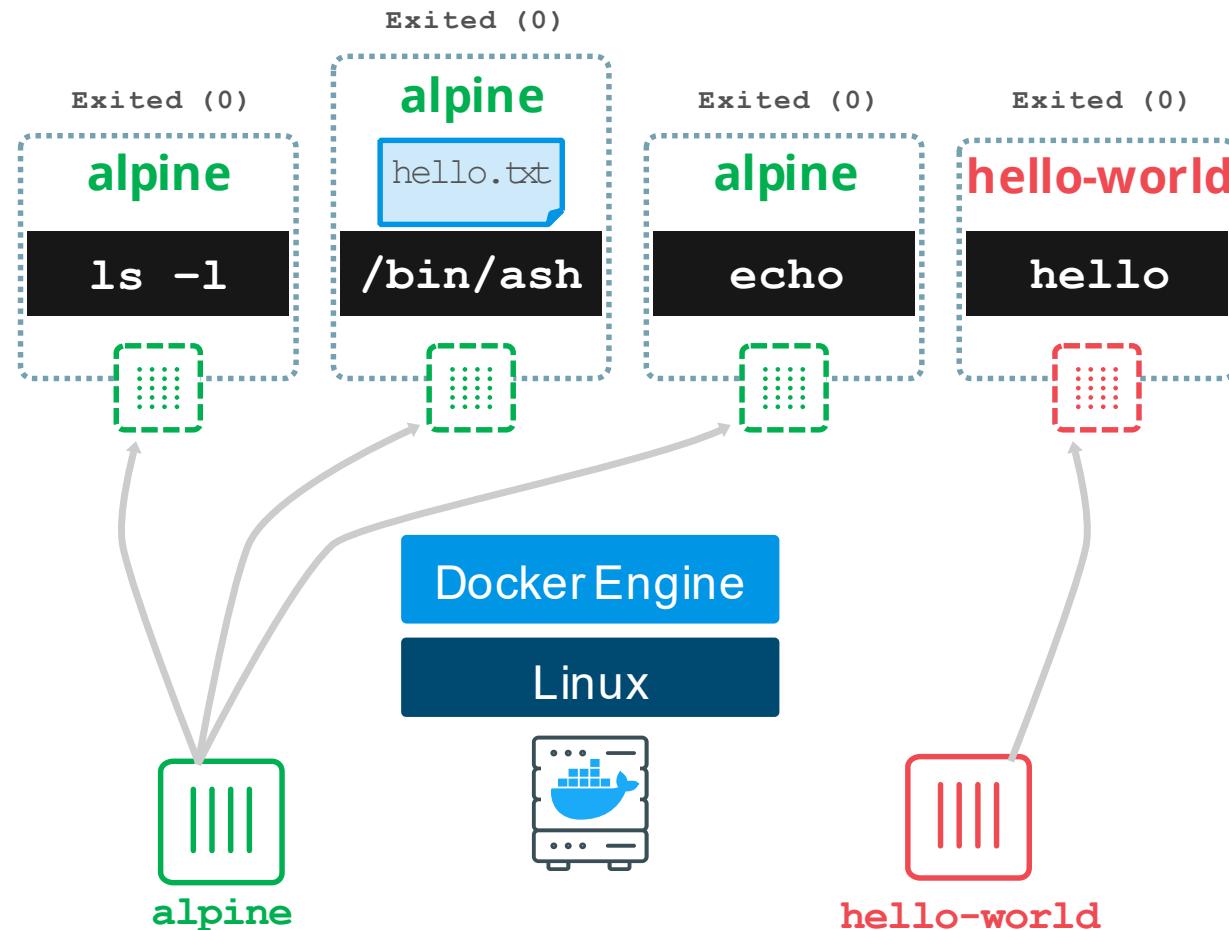
## Container Instances

	Exited (0)	Exited (0)	Exited (0)	Exited (0)
Container IDs	ff0a5c3750b9	36171a5da744	a6a9d46d0b2f	c317d0a9e3d2
Container Names	elated_ramanujan	fervent_newton	lonely_kilby	stupefied_mcclintock
	 alpine ls -l	 alpine /bin/sh	 alpine echo	 hello-world /hello

# Docker Container Isolation

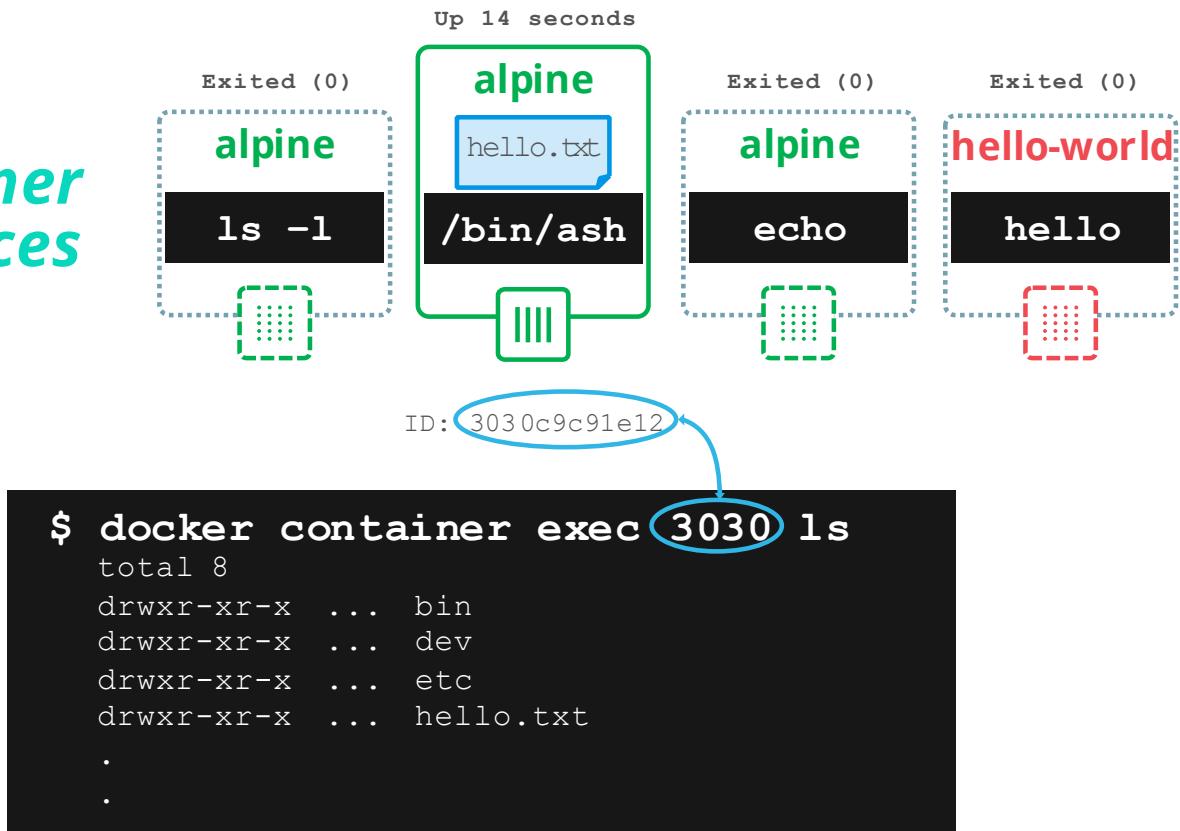
*Container Instances*

*Images*



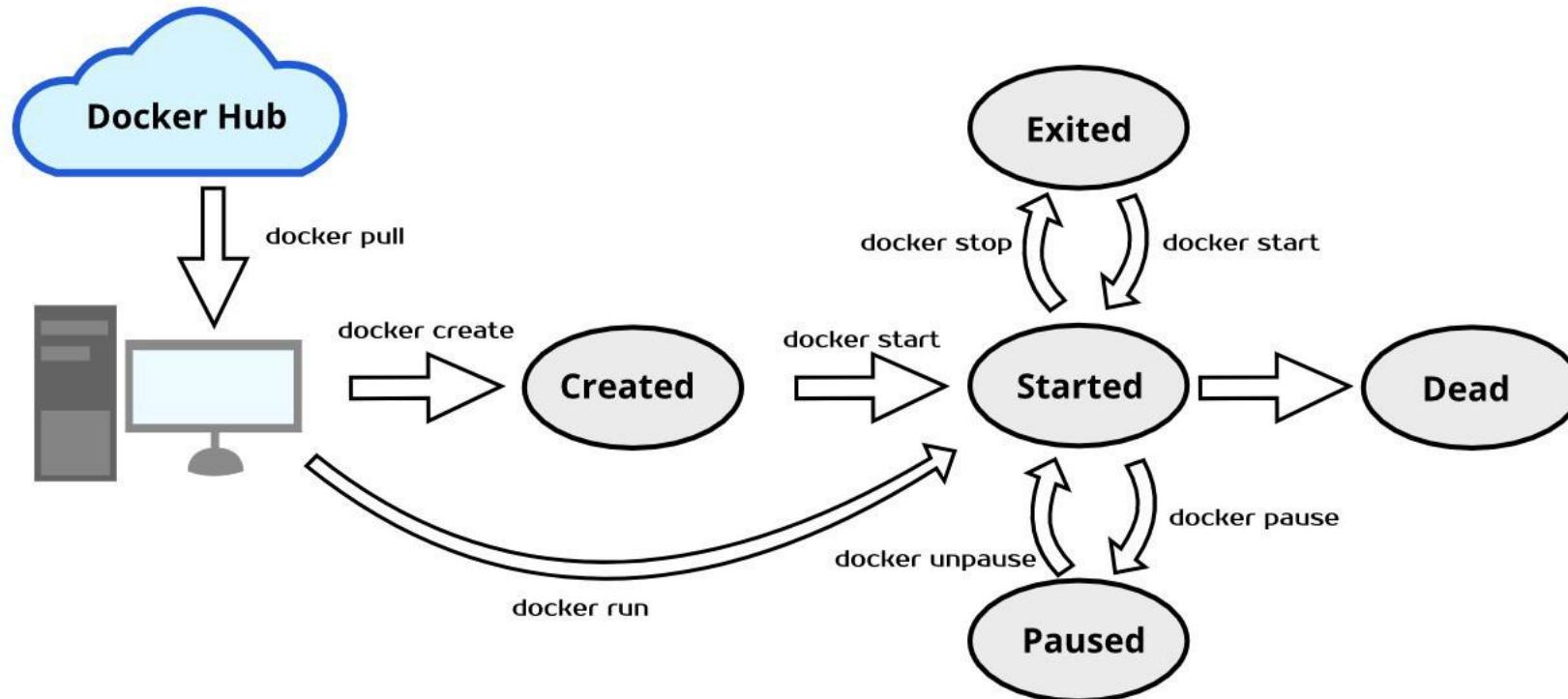
# docker container exec

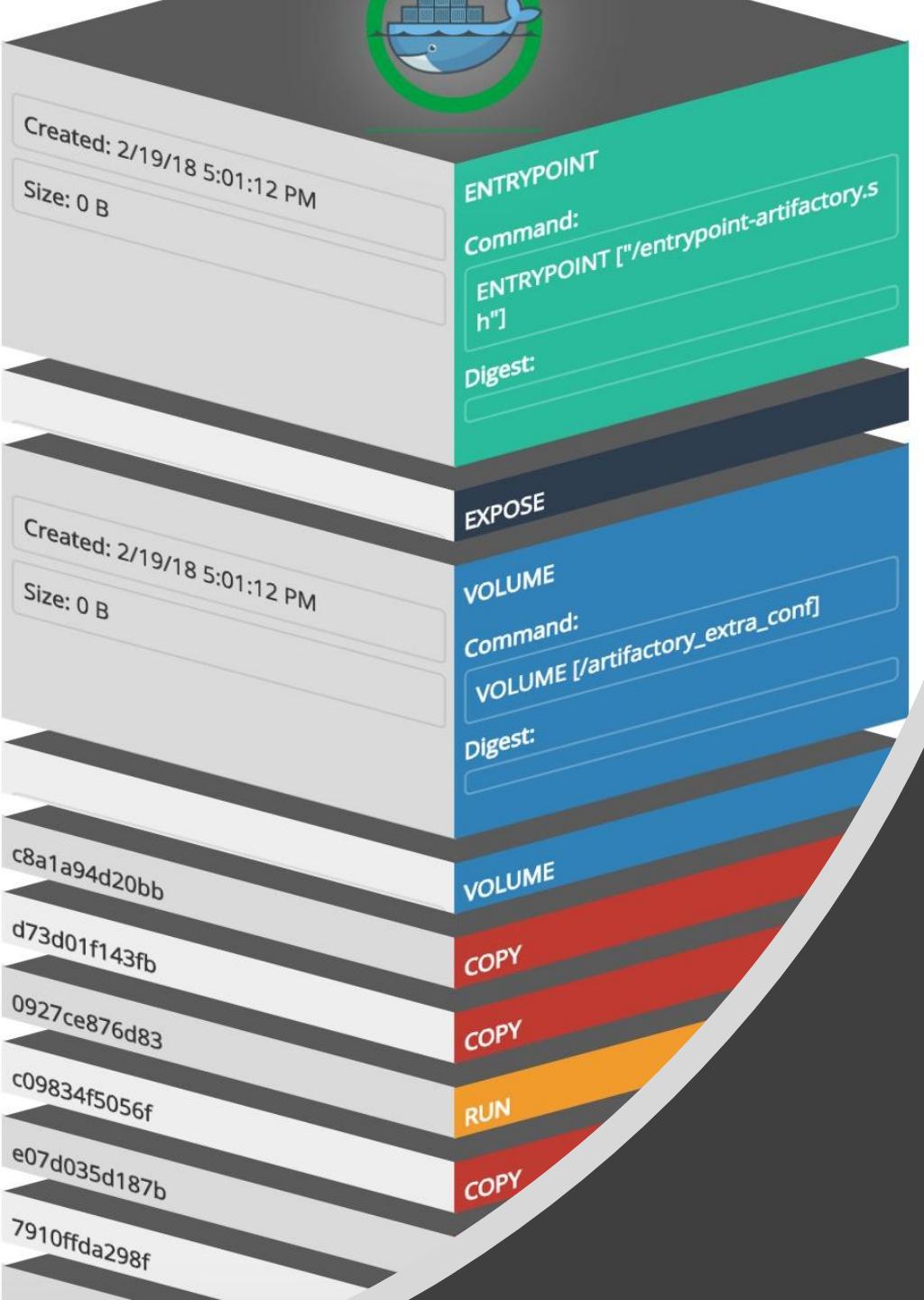
## Container Instances



Container Isolation

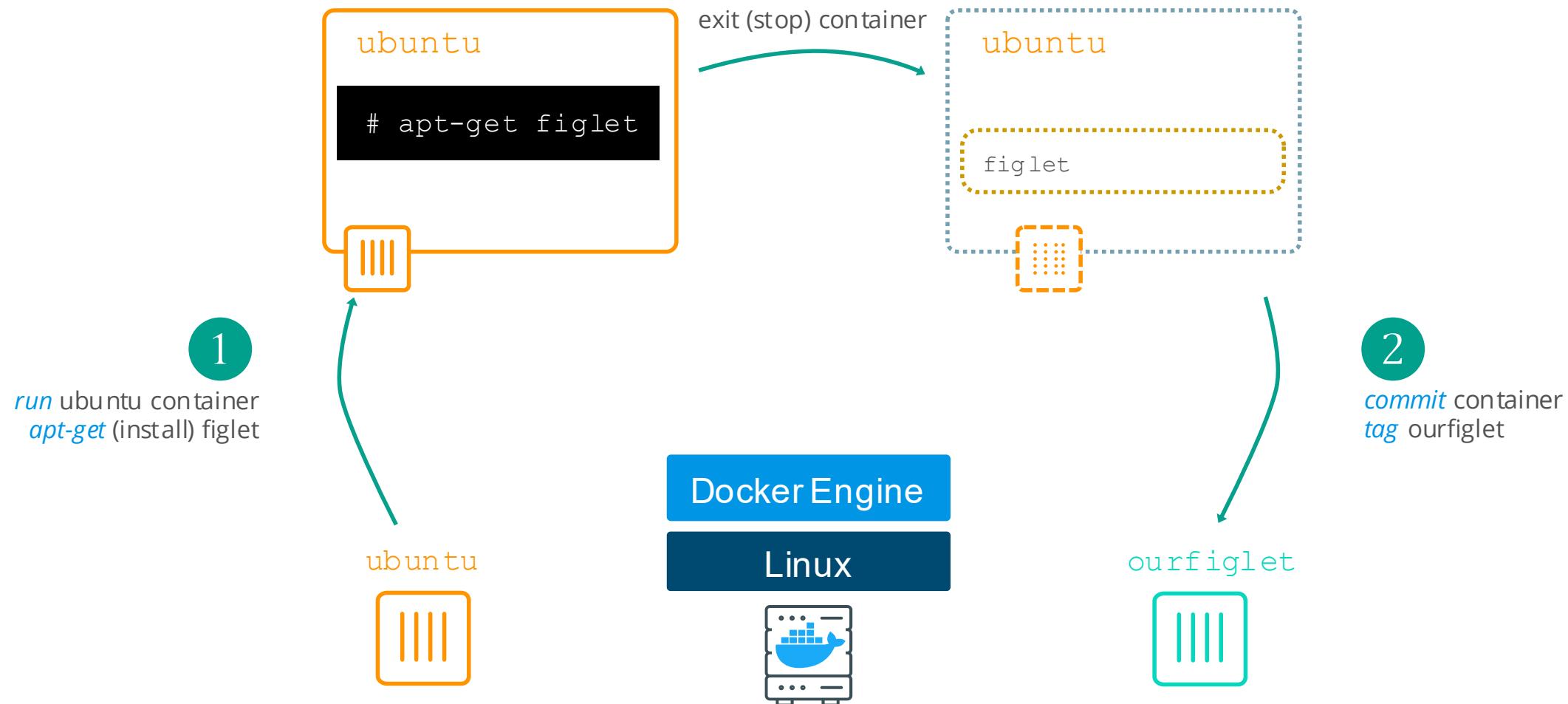
# Docker Lifecycle





# Créez une image Docker

# Image Creation: Instance Promotion



# Dockerfile

- Au lieu de créer une image binaire statique, nous pouvons utiliser un fichier appelé **Dockerfile** pour créer une image.
- Le résultat final est essentiellement le même, mais avec un Dockerfile, nous fournissons les instructions pour construire l'image, plutôt que juste les fichiers binaires bruts.
- Ceci est utile car il devient beaucoup plus facile de gérer les changements, d'autant plus que vos images deviennent plus grandes et plus complexes.

# Dockerfile : rôle et philosophie

---

22

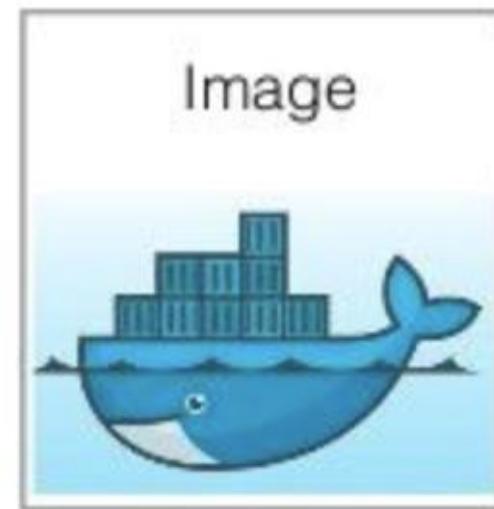
- Dockerfile = recette de build immuable
- Chaque instruction crée un **layer**
- Ordre des instructions = impact :
  - cache
  - taille
  - Sécurité

Un Dockerfile se **lit de haut en bas**, comme un pipeline.

```
FROM node:14.0.0
WORKDIR /app
COPY package.json /app
RUN npm install
EXPOSE 3001
CMD ["node", "index.js"]
```

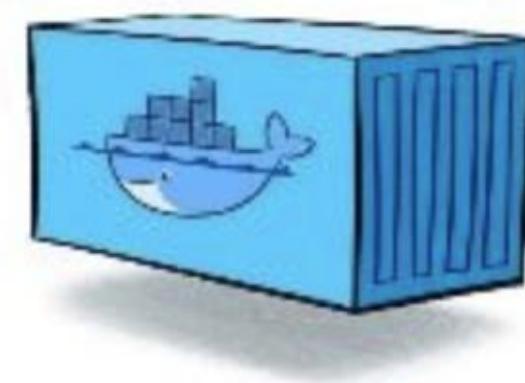
Dockerfile

build



Docker Image

run



Docker Container

Docker build |

# Dockerfile

## Dockerfile:

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node", "index.js"]
```

**FROM** alpine



**RUN** apk

apk update  
apk add nodejs

**COPY** ./app

host/index.js -->  
container/app

**WORKDIR** /app  
**CMD** "node" "index.js"

\$ cd /app  
\$ node index.js



Elies Jebri

# Instructions Dockerfile essentielles

---

25

- **FROM** : image de base
- **WORKDIR** : répertoire de travail
  - Image officielle et versionnée
  - Pas de latest
  - Toujours définir WORKDIR
- **COPY / ADD** : ajout de fichiers
  - **COPY** : simple, prévisible
  - **ADD** : extraction automatique, téléchargement distant
- **RUN** : exécution au build
  - Chaque RUN = un layer
  - Grouper les commandes
  - Nettoyer après installation
- **CMD / ENTRYPOINT** : lancement du conteneur
  - **CMD** : peut être surchargé
  - **ENTRYPOINT** : point d'entrée fixe

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

# USER & sécurité

---

26

- Éviter root
- Par défaut, un conteneur s'exécute en root
- Root dans un conteneur ≠ root sur l'hôte
- Réduction de l'impact en cas de compromission
- Parfois exigé par les policies sécurité
- Créer un utilisateur non-root (image classique)

```
RUN addgroup -S appgroup \
&& adduser -S appuser -G appgroup
USER appuser
```

# Images Docker modernes : distroless & scratch

---

- Images classiques : OS minimal + runtime
- Images distroless :
  - Pas de shell
  - Pas de package manager
  - Seulement l'application + ses dépendances
- Image scratch :
  - Image vide
  - Taille minimale
  - Cas d'usage très ciblés

# Pourquoi distroless & scratch en DevOps

---

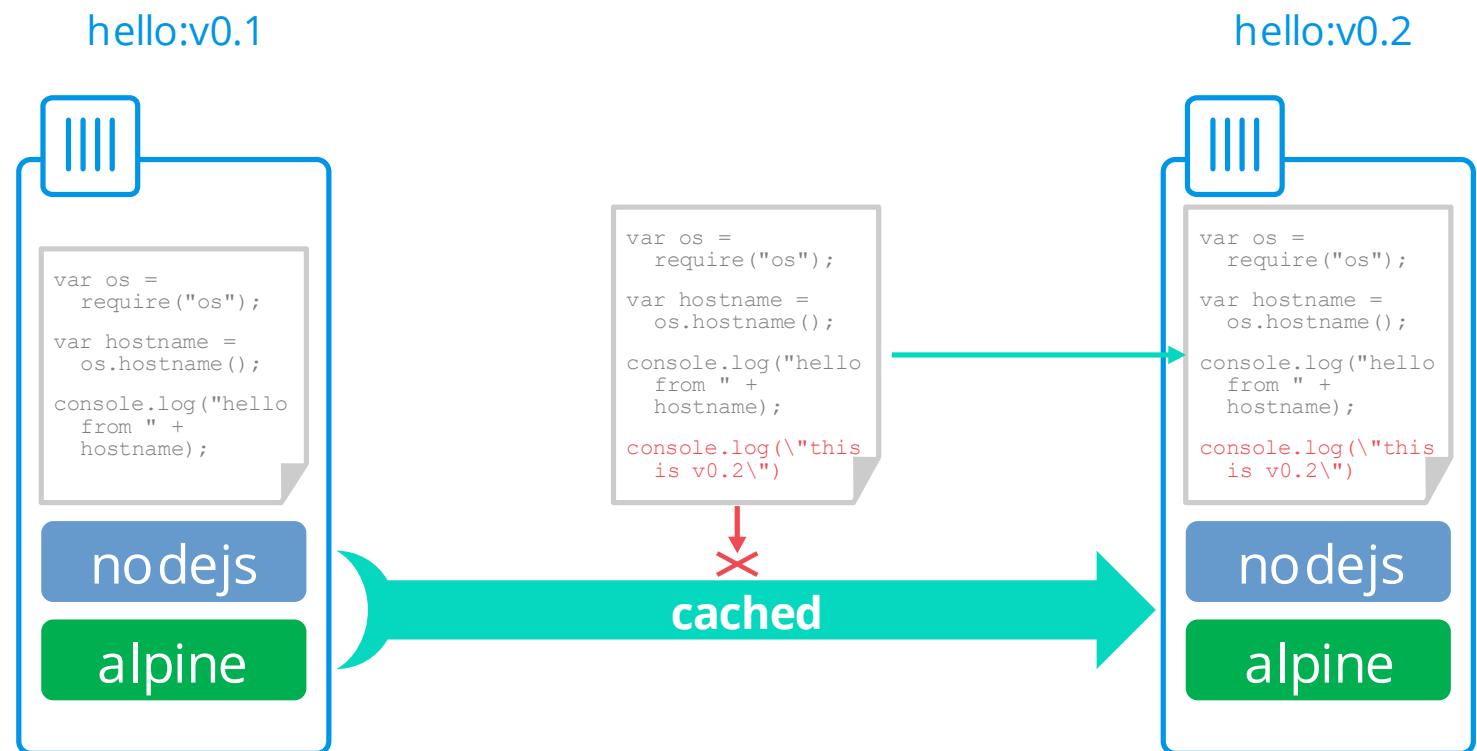
28

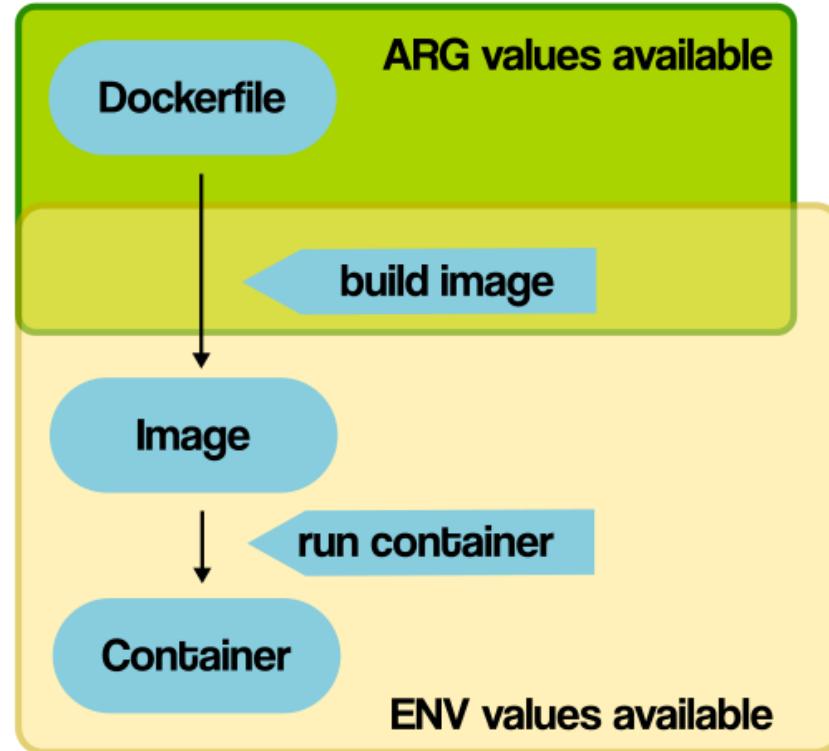
- Sécurité renforcée (moins de CVE)
- Images plus légères
- Déploiements plus rapides
- Debug dans le conteneur plus difficile

# cache

- Docker reconnaît que nous avons déjà construit certaines de ces couches dans nos précédentes versions d'image
  - Comme rien n'a changé dans ces couches, il utilise une version mise en cache de la couche, plutôt que d'extraire le code une deuxième fois et d'exécuter ces étapes.

# Layers & Cache



**Dockerfile content:**

```
ARG required_var      # expects a value
ARG var_name=def_value # set default ARG value
ENV foo=other_value   # set default ENV value
ENV bar=${var_name}    # set default ENV value from ARG
```

**Override Dockerfile ARG values on build:**

```
$ docker build --build-arg var_name=value
```

**Override Docker image ENV values on run:**

```
$ docker run -e "foo=other_value" [...]
$ docker run -e foo [...] # pass from host
$ docker run --env-file=env_file_name [...] # pass from file
```

<https://vsupalov.com/docker-env-vars>

# Dockerfile arguments |

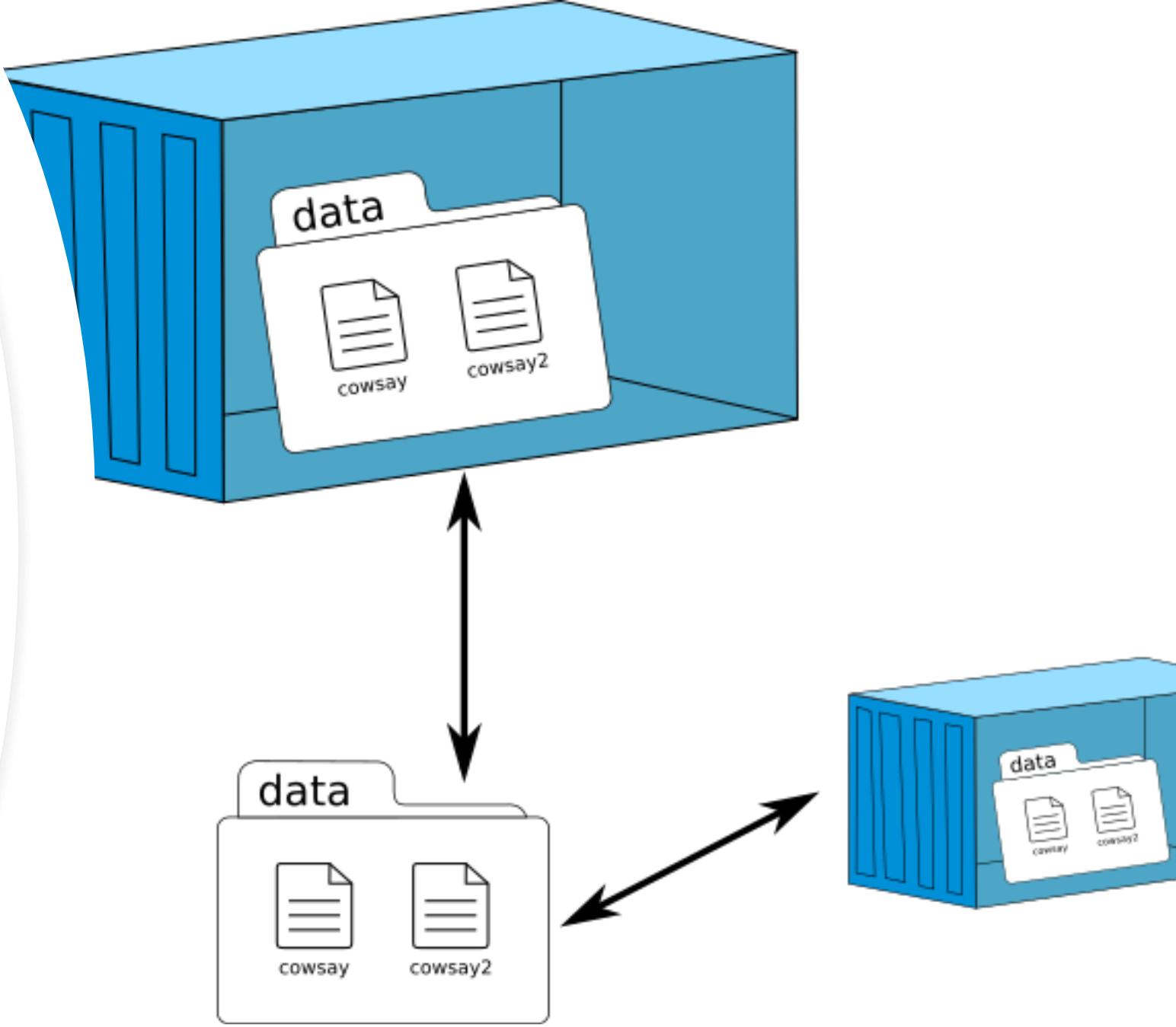
# docker init

---

31

- Génère automatiquement :
- Dockerfile
- .dockerignore
- docker-compose.yml (optionnel)
- Basé sur le type de projet détecté
- Gain de temps pour démarrer un projet
- Fichiers conformes aux bonnes pratiques Docker
- Réduction des erreurs humaines
- Base commune pour les équipes

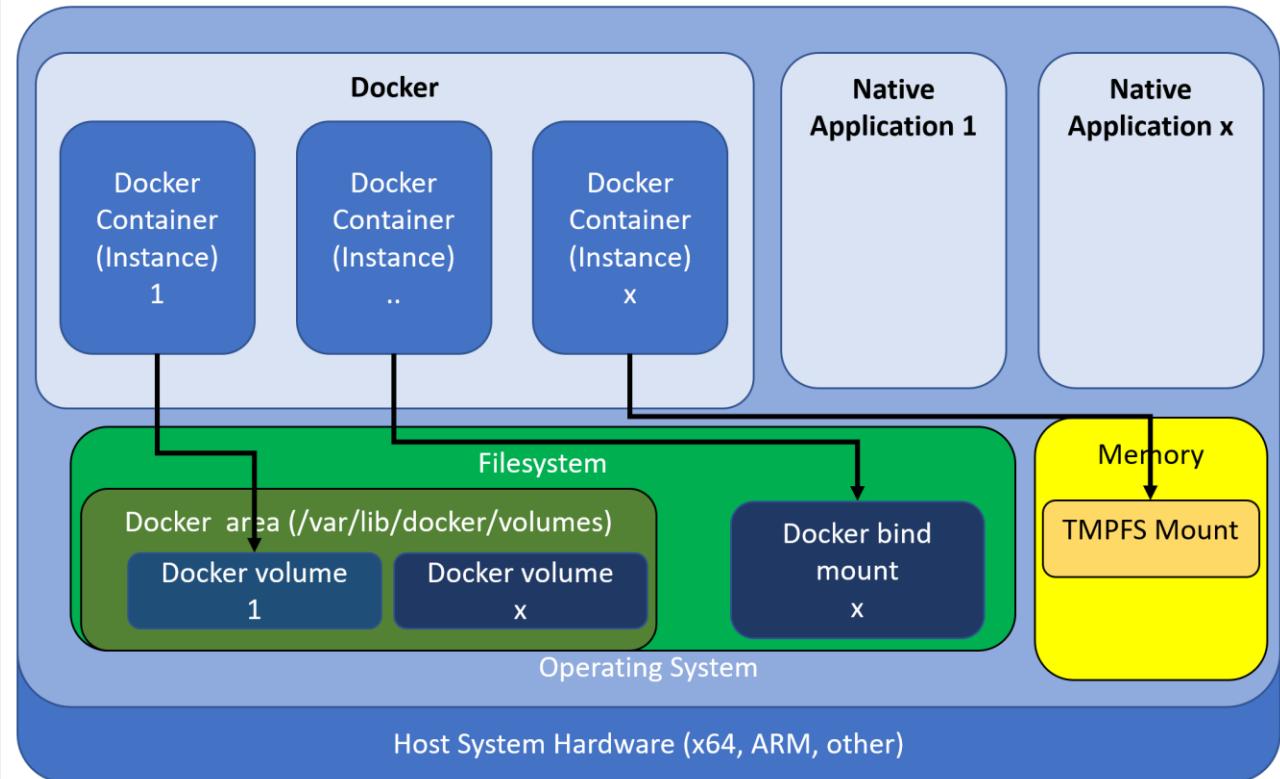
# Volumes



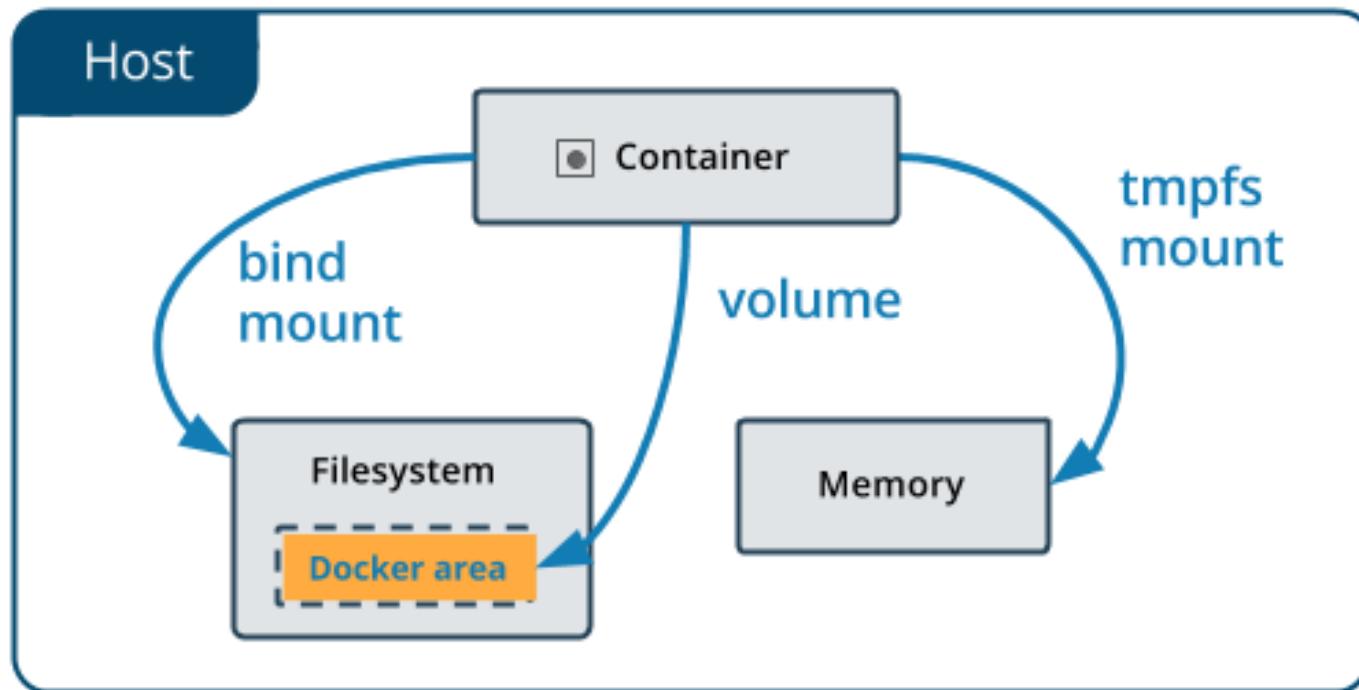
# Stockage persistant sur des conteneurs

- Docker propose différentes manières de stocker des données persistantes telles que
  - les volumes,
  - les Bind mount (ou volumes hôtes),
  - Vous pouvez stocker des données non persistantes sur un montage TMPFS où elles sont stockées en mémoire.

Les volumes étant la méthode préférée.



# volume



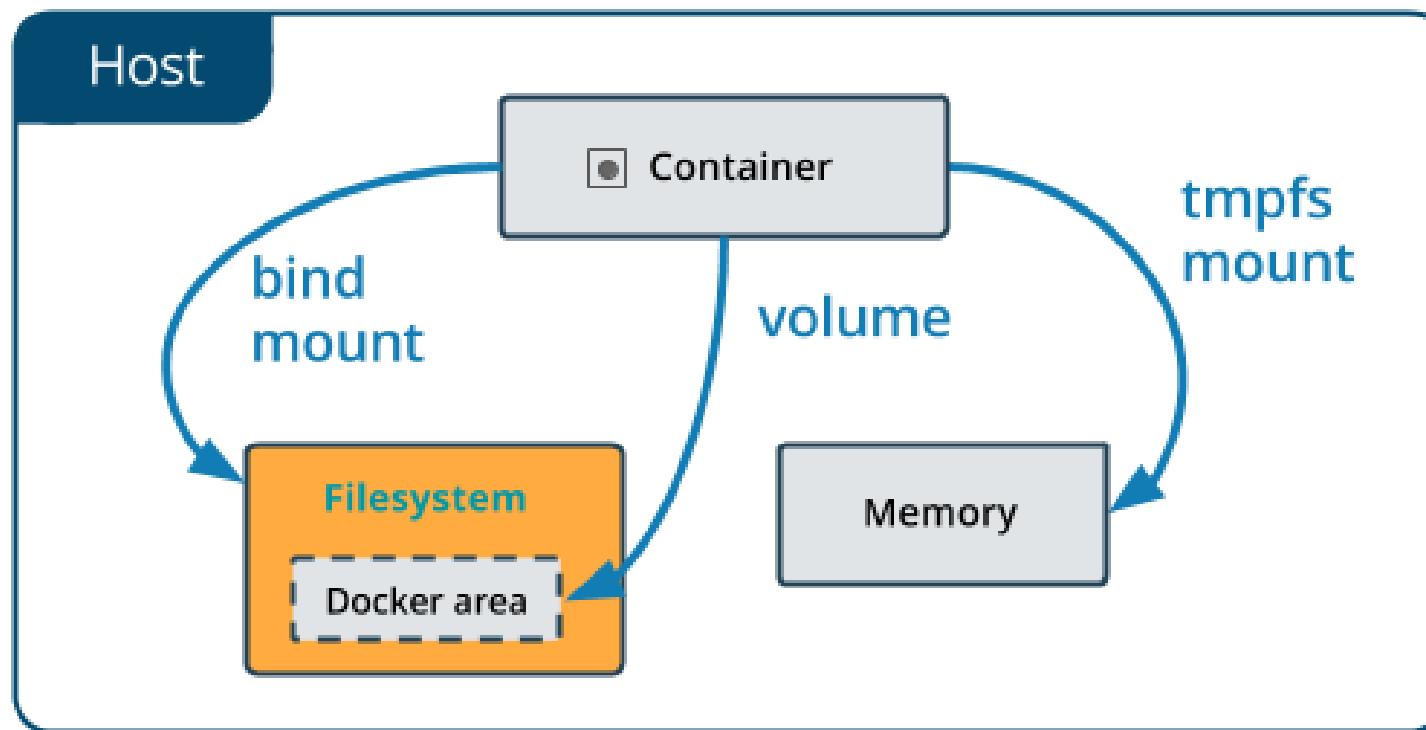
- Les volumes sont le mécanisme préféré pour la persistance des données générées par et utilisées par les conteneurs Docker.
- Alors que les montages de liaison dépendent de la structure de répertoires et du système d'exploitation de la machine hôte, les volumes sont entièrement gérés par Docker.

```
root@ubuntu2204A:/home/user01# docker volume create MyVol_001
MyVol_001
root@ubuntu2204A:/home/user01# docker volume ls
DRIVER      VOLUME NAME
local        MyVol_001
root@ubuntu2204A:/home/user01# docker volume inspect MyVol_001
[
  {
    "CreatedAt": "2022-11-28T10:43:45-08:00",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/MyVol_001/_data",
    "Name": "MyVol_001",
    "Options": {}
```

## Volume

*docker volume create MyVol\_001*

# Bind mount



- Le fichier ou le répertoire n'a pas besoin d'exister déjà sur l'hôte Docker.
- Il est créé à la demande s'il n'existe pas encore.
- Les montages bind sont très performants, mais ils dépendent du système de fichiers de la machine hôte ayant une structure de répertoire spécifique disponible.

# Bind mount

---

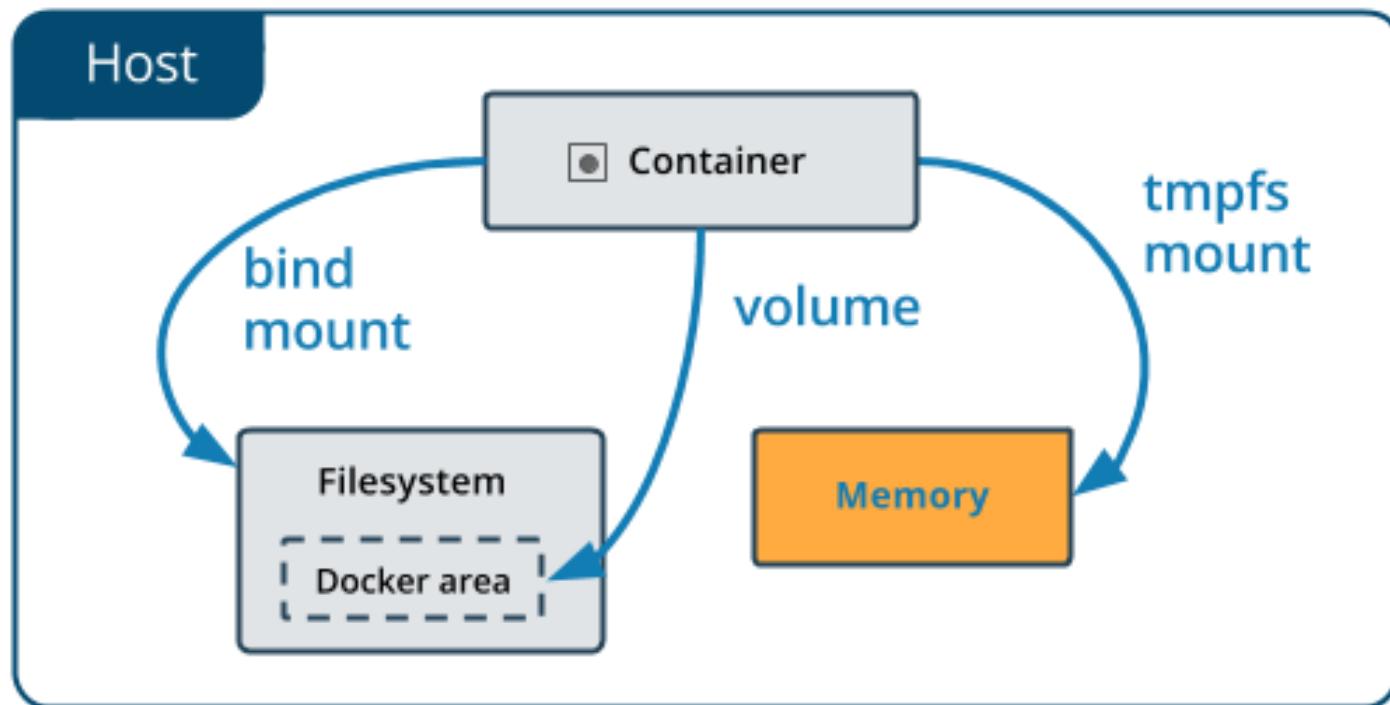
- Vous pouvez créer un montage de type **bind** à l'aide de la commande *docker run* avec l'option **--mount** (recommandée) ou son raccourci **-v**
- Utilisation de l'option `--mount` (syntaxe recommandée) :

37

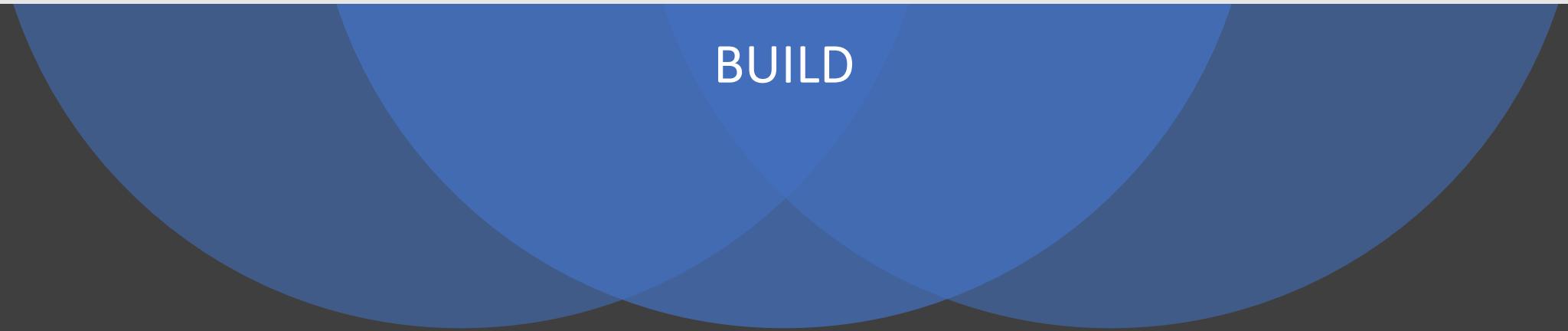
```
docker run -d --name devtest --mount type=bind,source="$(pwd)"/target,target=/app nginx:latest
```

- **type=bind** : Spécifie le type de montage comme un montage « bind ».
  - **source (ou src)** : Le chemin absolu vers le fichier ou le répertoire sur la machine hôte. L'utilisation de `\$(pwd)` garantit que vous fournissez un chemin absolu si vous utilisez un chemin relatif dans votre terminal.
  - **target (ou dst, destination)** : Le chemin où le fichier ou le répertoire sera monté dans le conteneur.
  - Utilisation de l'option **-v** (syntaxe abrégée) :
- ```
docker run -d --name devtest -v "$(pwd)"/target:/app nginx:latest
```
- L'option **-v** exige que le chemin de l'hôte et le chemin du conteneur soient séparés par un deux-points (:), dans cet ordre précis.

# tmpfs



- Les volumes et les montages bind vous permettent de partager des fichiers entre la machine hôte et le conteneur afin que vous puissiez conserver les données même après l'arrêt du conteneur.
- Si vous exécutez Docker sous Linux, vous avez une troisième option: les montages **tmpfs**. Lorsque vous créez un conteneur avec un montage tmpfs, le conteneur peut créer des fichiers en dehors de la couche inscriptible du conteneur.



Multi-stage

BUILD

# Multi-stage build : le problème à résoudre

- Bien que l'objectif principal des multi-stage builds Docker soit la création d'images de production optimisées (ce qui implique intrinsèquement un processus de build),
- Vous pouvez utiliser le même modèle à d'autres fins, telles que les tests, l'exécution de plusieurs étapes d'intégration continue ou la séparation des environnements de développement et de production au sein d'un même Dockerfile.
- Images classiques :
  - contiennent outils de build
  - Compilateurs
  - dépendances inutiles en prod
- Conséquences :
  - images lourdes
  - surface d'attaque élevée
  - plus de CVE

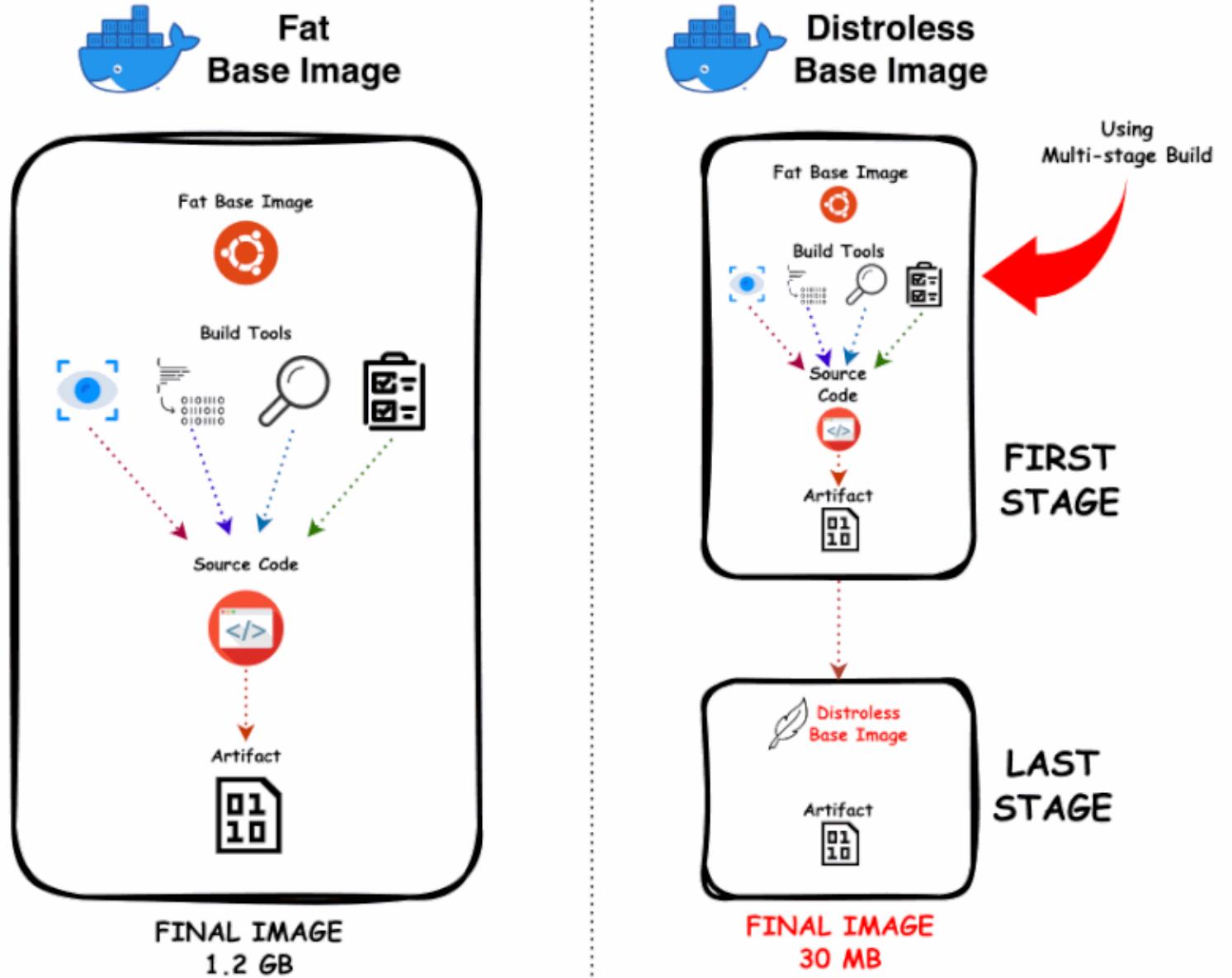
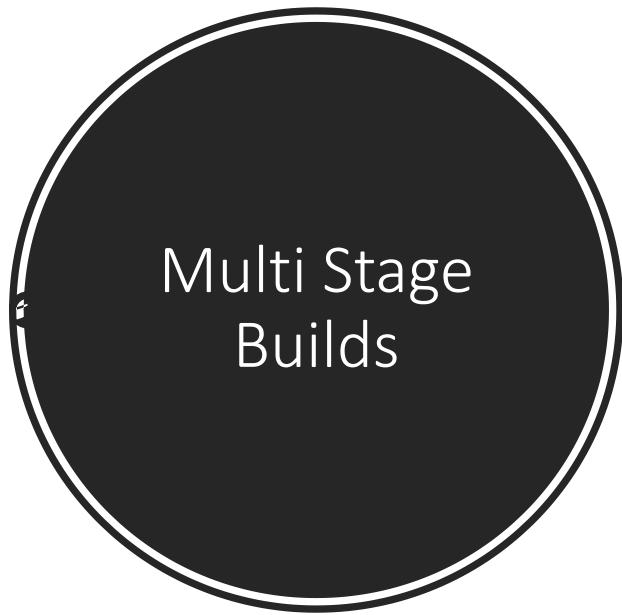
# Multi-stage build : principe

- Plusieurs étapes (FROM) dans un seul Dockerfile
- Séparation :
  - **build stage**
  - **runtime stage**
- Seuls les artefacts nécessaires sont copiés

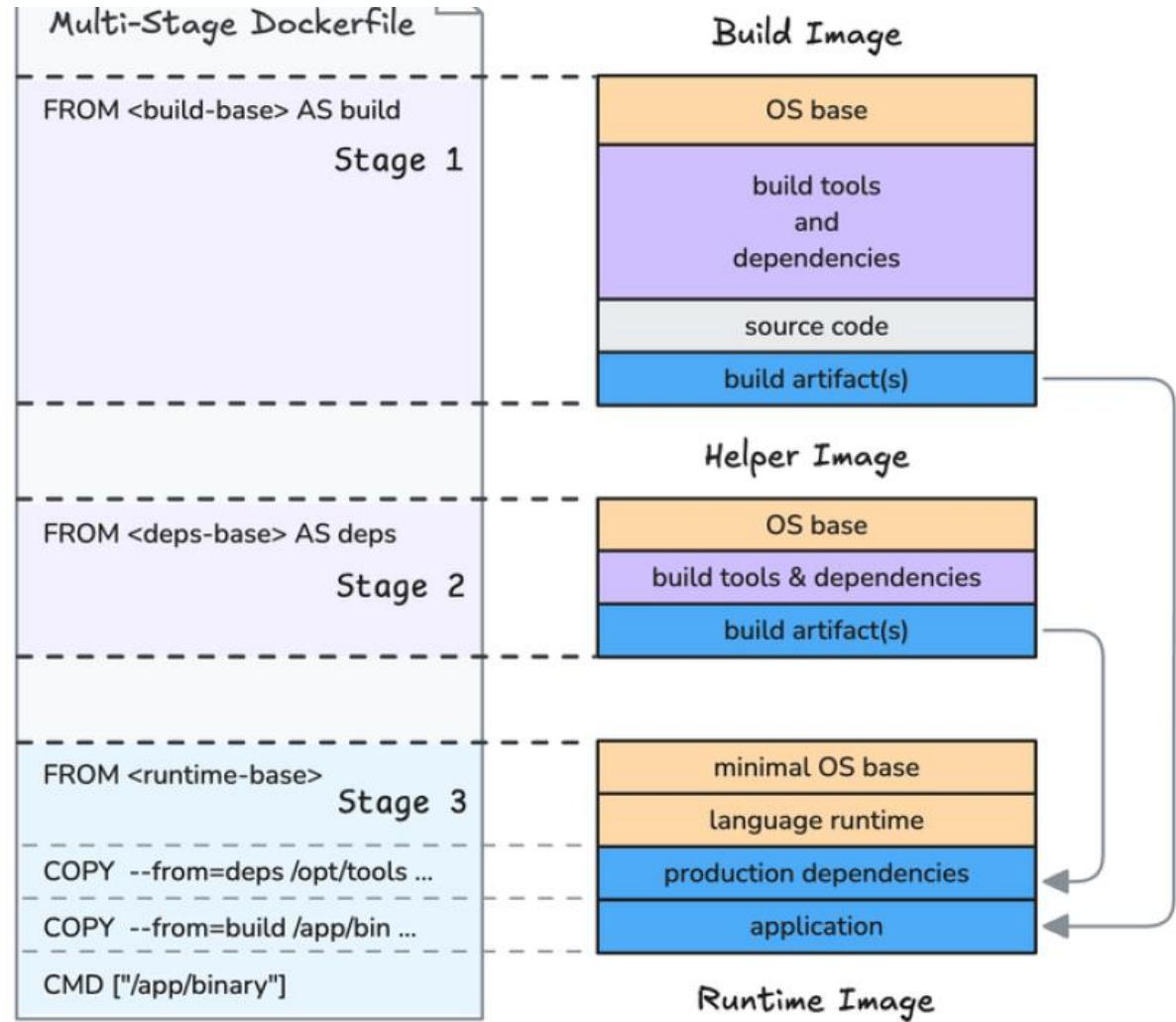
**Schéma mental**  
build → copy → run

```
FROM node:ls AS test_stage
# Install dev dependencies and run tests
WORKDIR /app
COPY package*.json .
RUN npm install
COPY ..
RUN npm test
```

```
FROM node:ls-slim AS final_stage
# Copy only necessary application files for runtime
# ...
```



# Multi Stage Dockerfile



Stages can be built separately using `--target <stage>` flag.

```
docker build --target build ...
```

Independent stages can be executed in parallel.

```
docker build --target deps ...
```

Building a stage also builds all stages it depends on.

```
docker build -t app:v1.2.3
```

# Docker networking

# Pilotes réseau

Le sous-système réseau de Docker est enfichable, à l'aide de pilotes.

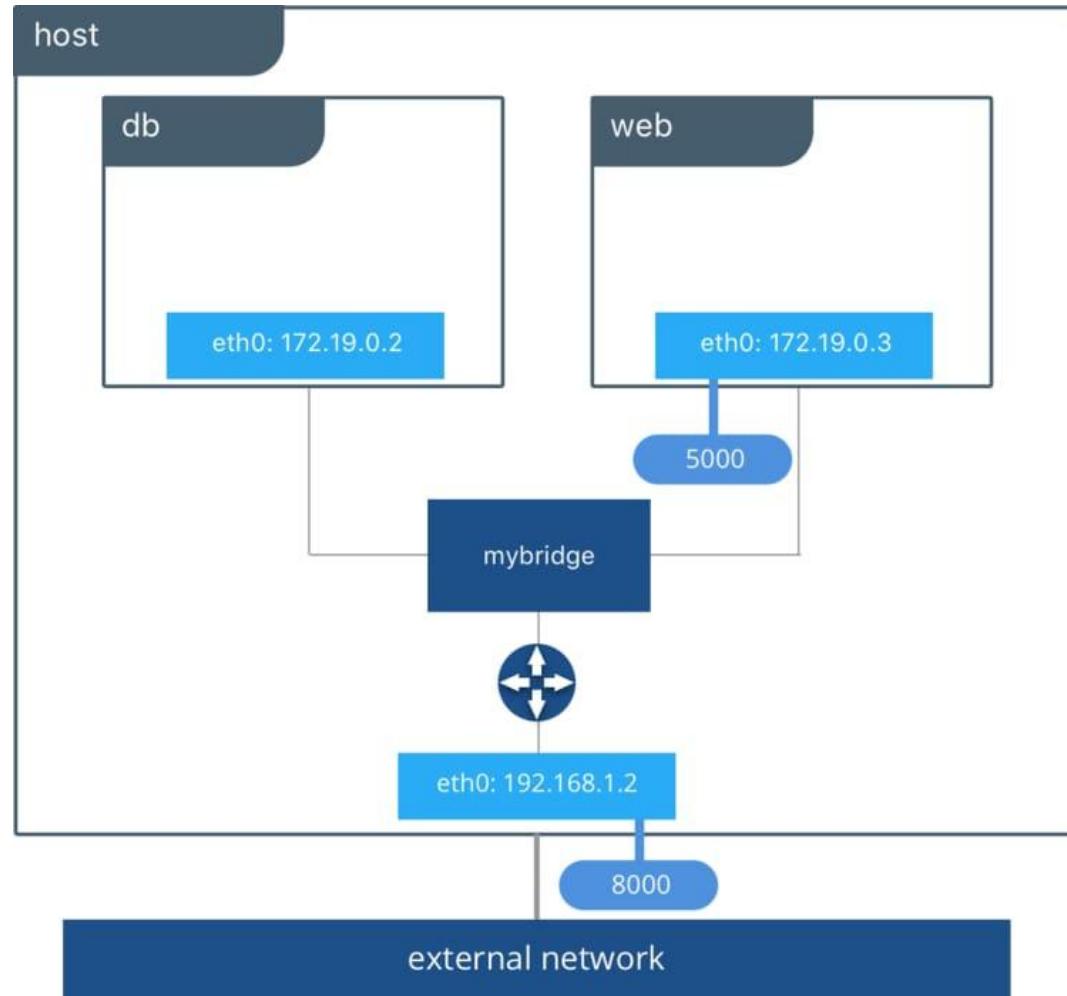
- **bridge**: Pilote réseau par défaut. Les réseaux bridge sont généralement utilisés lorsque vos applications s'exécutent dans des conteneurs autonomes qui doivent communiquer.
- **host**: Pour les conteneurs autonomes, supprimez l'isolation réseau entre le conteneur et l'hôte Docker et utilisez directement la mise en réseau de l'hôte.
- **overlay**: Les réseaux overlay connectent plusieurs nœuds Docker ensemble et permettent aux services Swarm de communiquer entre eux ou pour faciliter la communication entre un service Swarm et un conteneur autonome, ou entre deux conteneurs autonomes sur différents nœuds Docker. Cette stratégie supprime la nécessité d'effectuer un routage au niveau du système d'exploitation entre ces conteneurs.

# Pilotes réseau - suite

- **macvlan**: permet d'attribuer une adresse MAC à un conteneur, le faisant apparaître comme un périphérique physique sur votre réseau. Le démon Docker achemine le trafic vers les conteneurs par leurs adresses MAC.
- **none**: Pour ce conteneur, désactivez tous les réseaux. Habituellement utilisé avec un pilote réseau personnalisé.
- **Plugins réseau** : vous pouvez installer et utiliser des plugins réseau tiers avec Docker. Ces plugins sont disponibles auprès de Docker Hub ou auprès de fournisseurs tiers.

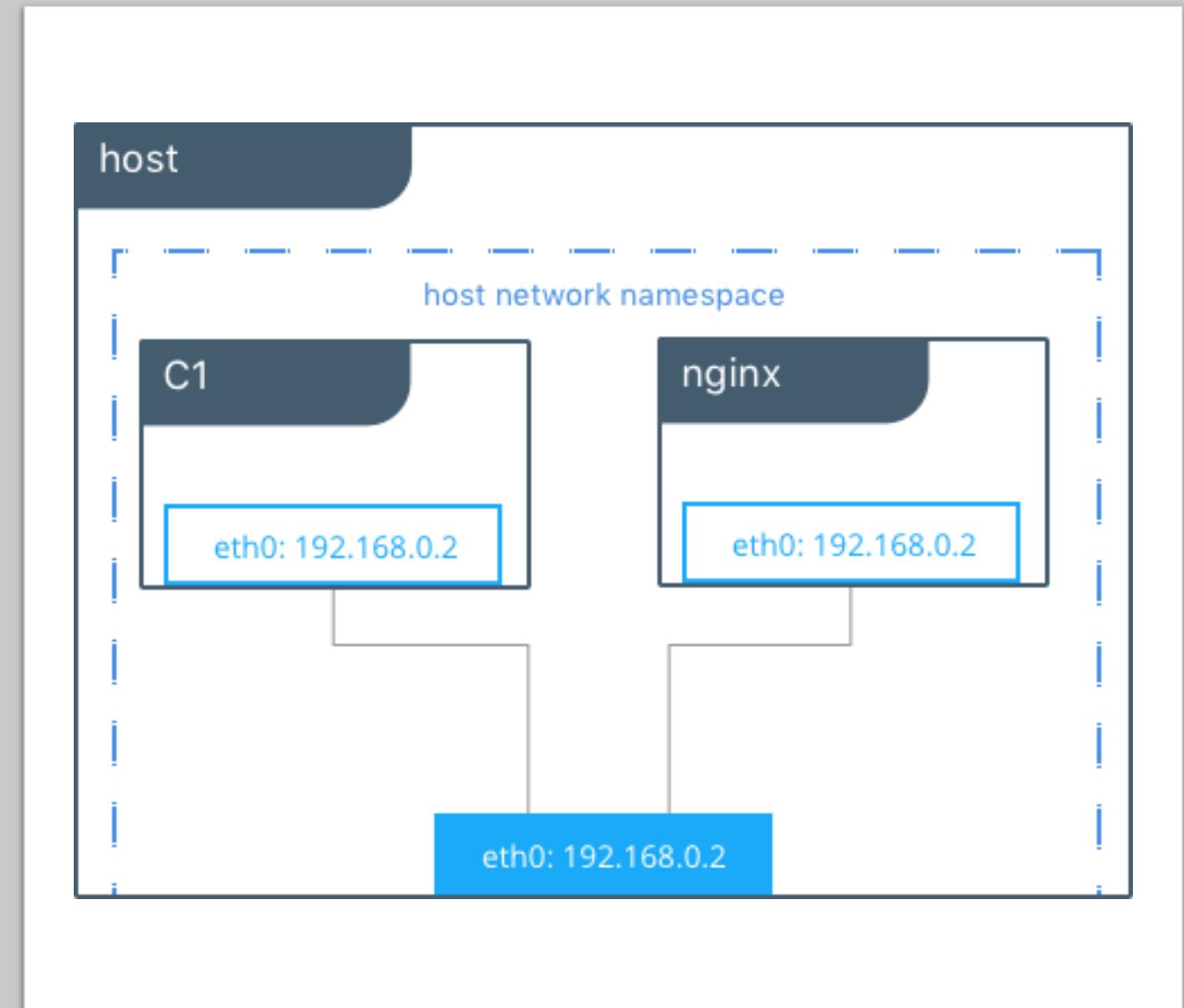
# Le driver Bridge

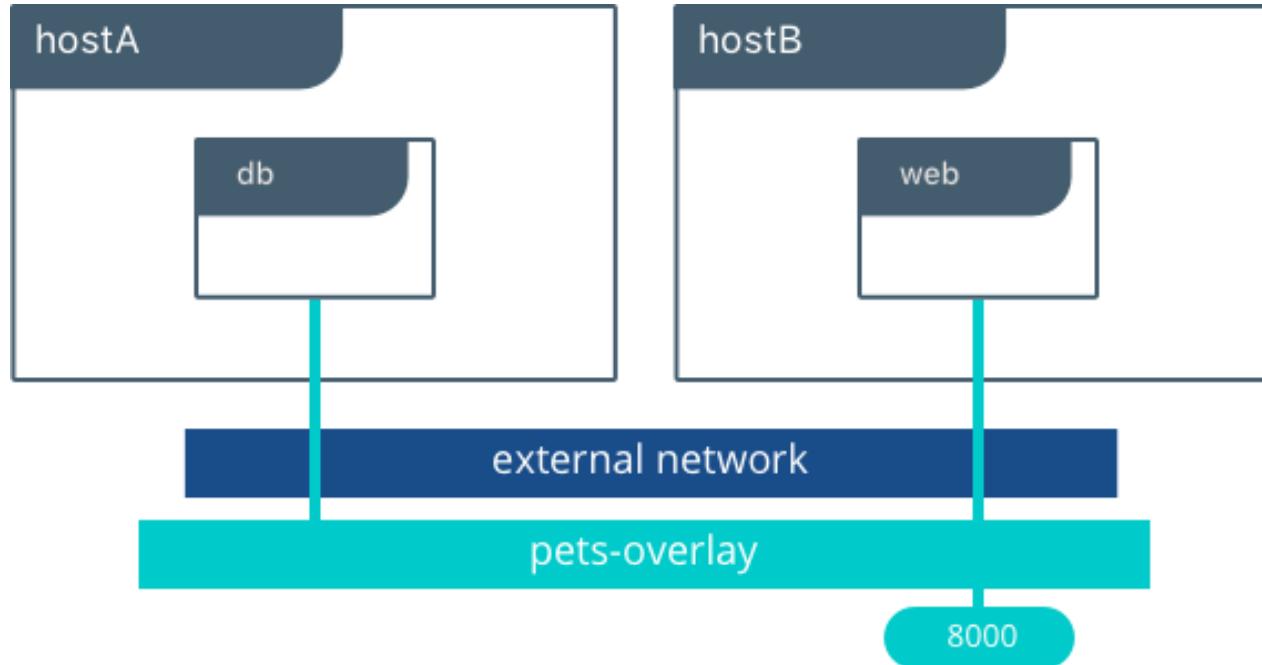
- Le réseau bridge est le type de réseau le plus couramment utilisé.
- Il est limité aux conteneurs d'un hôte unique exécutant le moteur Docker.
- Les conteneurs qui utilisent ce driver, ne peuvent communiquer qu'entre eux, cependant ils ne sont pas accessibles depuis l'extérieur.
- Pour que les conteneurs sur le réseau bridge puissent communiquer ou être accessibles du monde extérieur, vous devez configurer le mappage de port.



# Le driver host

- Ce type de réseau permet aux conteneurs d'utiliser la même interface que l'hôte.
- Il supprime donc l'isolation réseau entre les conteneurs et seront par défaut accessibles de l'extérieur.
- De ce fait, il prendra la même IP que votre machine hôte.



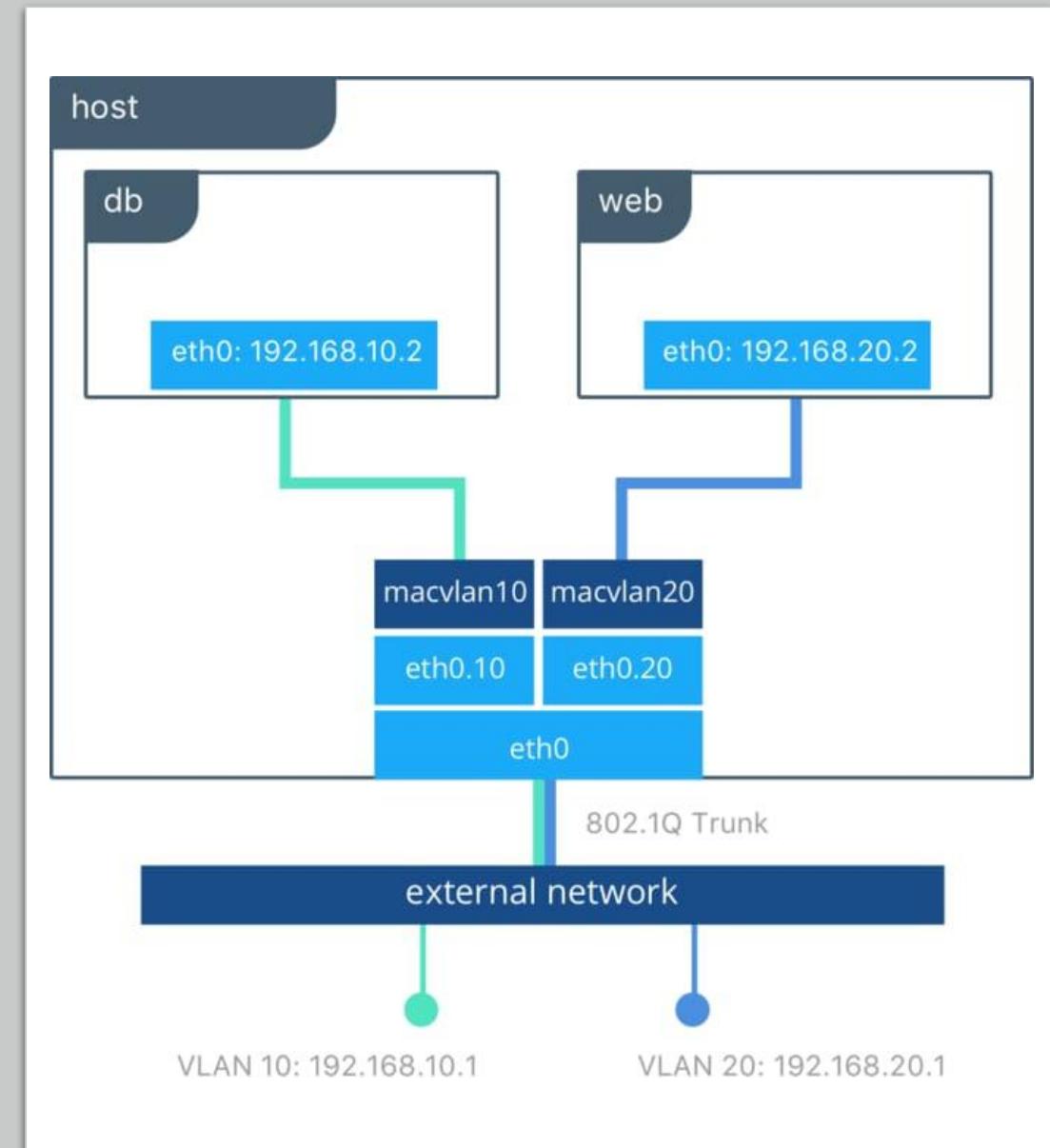


- Si vous souhaitez une mise en réseau multi-hôte native, vous devez utiliser un driver overlay.
- Il crée un réseau distribué entre plusieurs hôtes possédant le moteur Docker.
- Docker gère de manière transparente le routage de chaque paquet vers et depuis le bon hôte et le bon conteneur.

## Le driver overlay

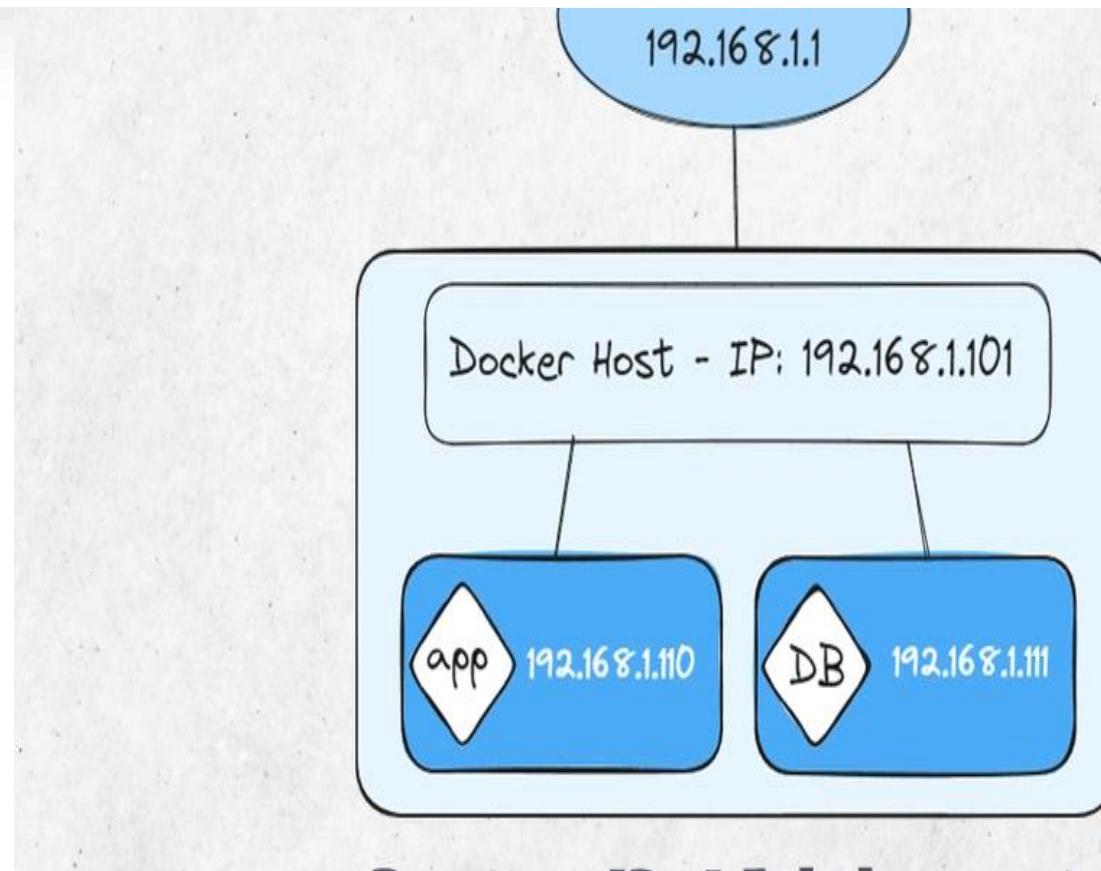
# Le driver macvlan

- L'utilisation du driver macvlan est parfois le meilleur choix lorsque vous utilisez des applications qui s'attendent à être directement connectées au réseau physique, car le driver Macvlan vous permet d'attribuer une adresse MAC à un conteneur, le faisant apparaître comme un périphérique physique sur votre réseau.
- Le moteur Docker route le trafic vers les conteneurs en fonction de leurs adresses MAC.



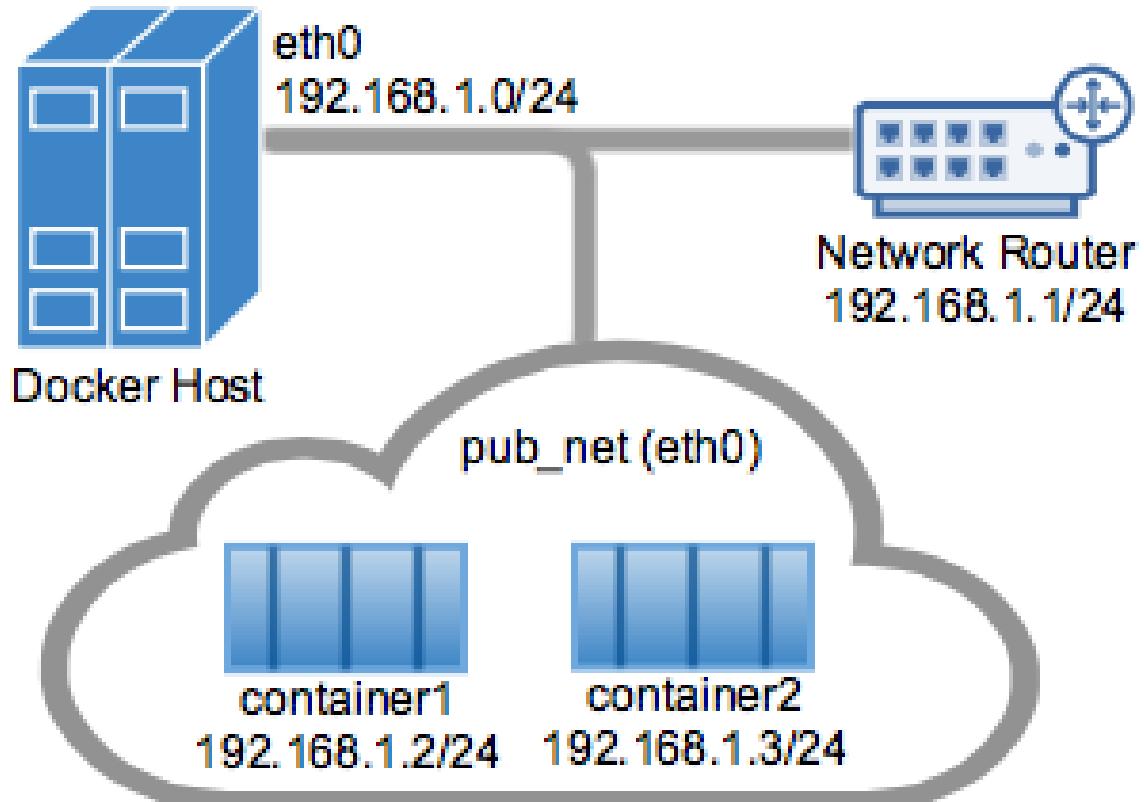
# Docker Network IPVlan

- Permet aux conteneurs :
- D'avoir une adresse IP propre sur le réseau physique
- D'être directement intégrés au réseau existant
- Sans utiliser de NAT (contrairement au bridge)
- Il fonctionne au niveau L2 ou L3 du modèle OSI



## IPvlan L2 mode

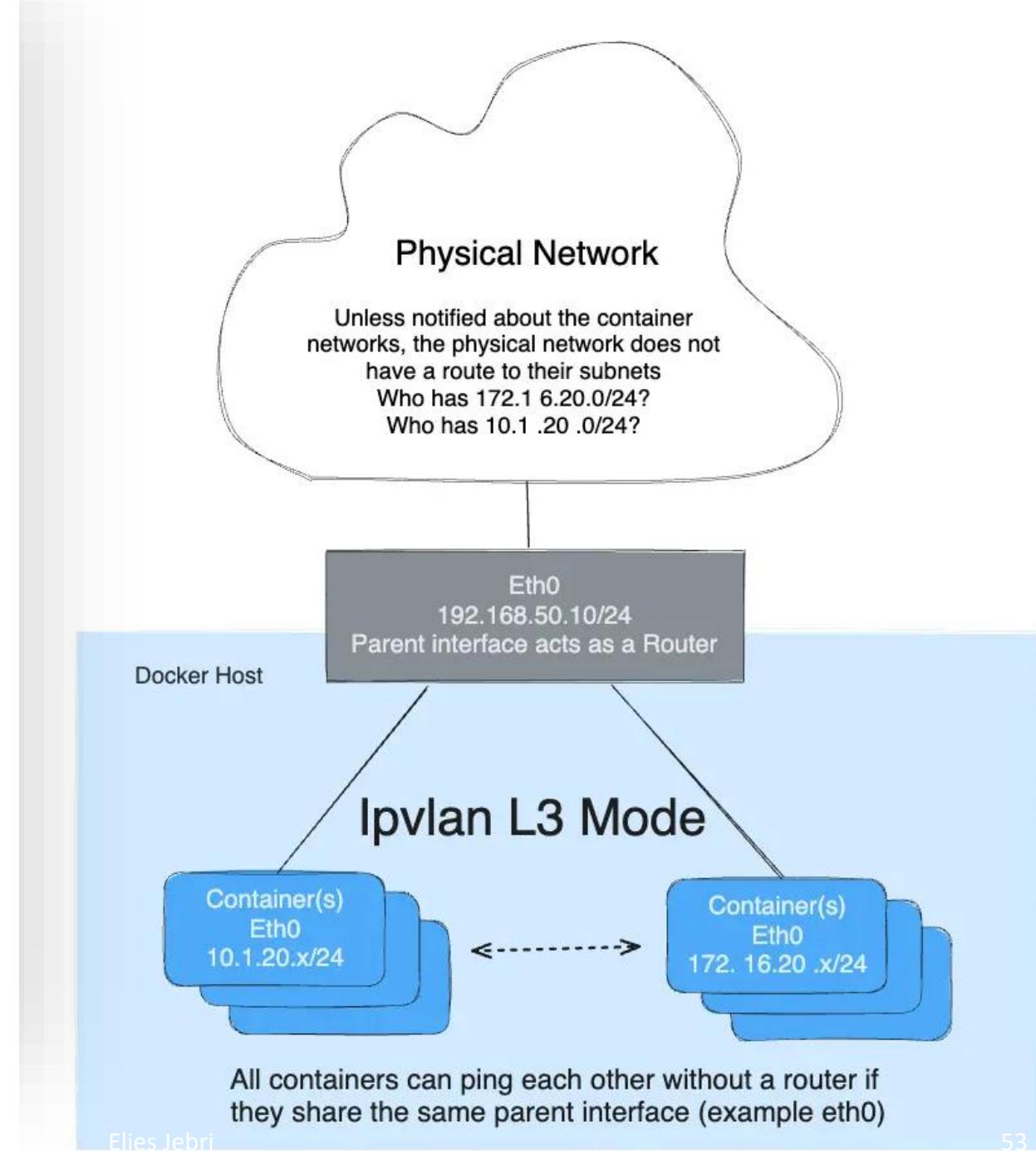
- Les conteneurs sont sur le **même sous-réseau que l'hôte**
- Communication directe via le switch physique
- Comportement proche du macvlan (mais sans MAC multiple)



```
docker network create -d ipvlan \
    --subnet=192.168.1.0/24 \
    --gateway=192.168.1.1 \
    --parent=eth0 pub_net
```

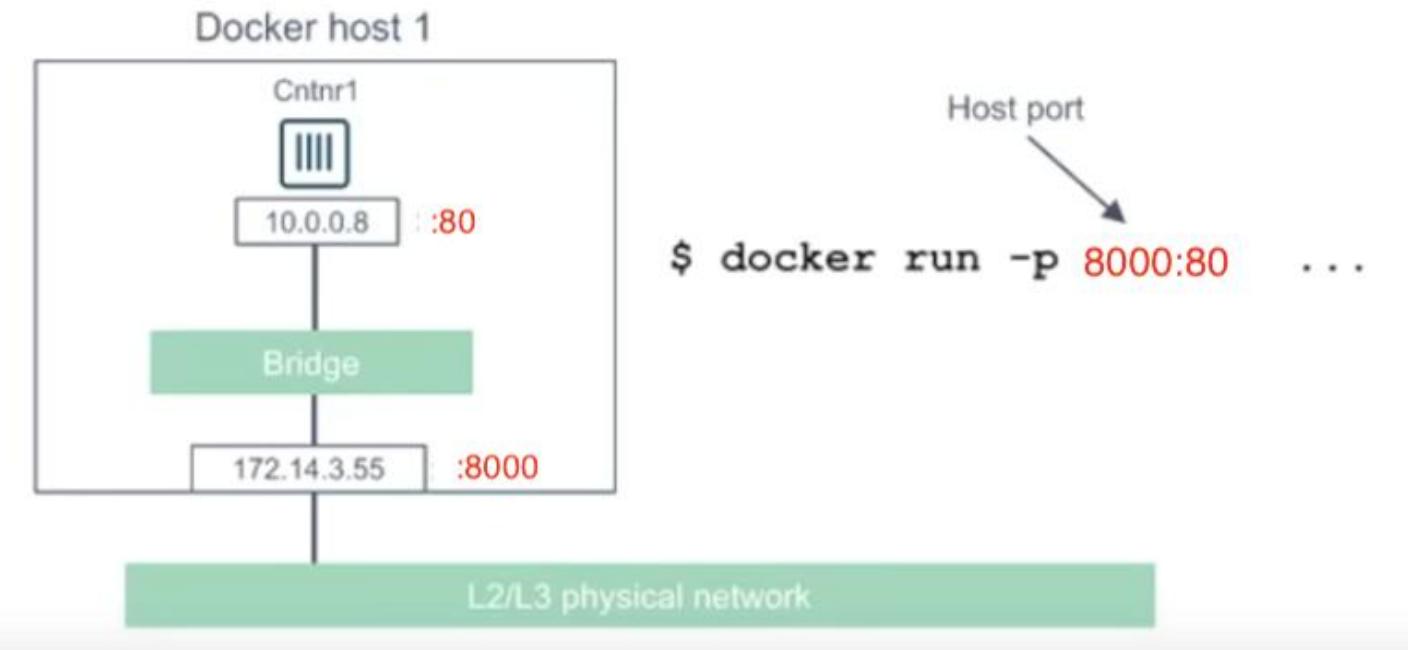
# IPvlan L3 mode

- Les conteneurs sont sur un **sous-réseau différent**
- Routage via l'hôte
- Plus scalable en environnement multi-segments



# Port Mapping

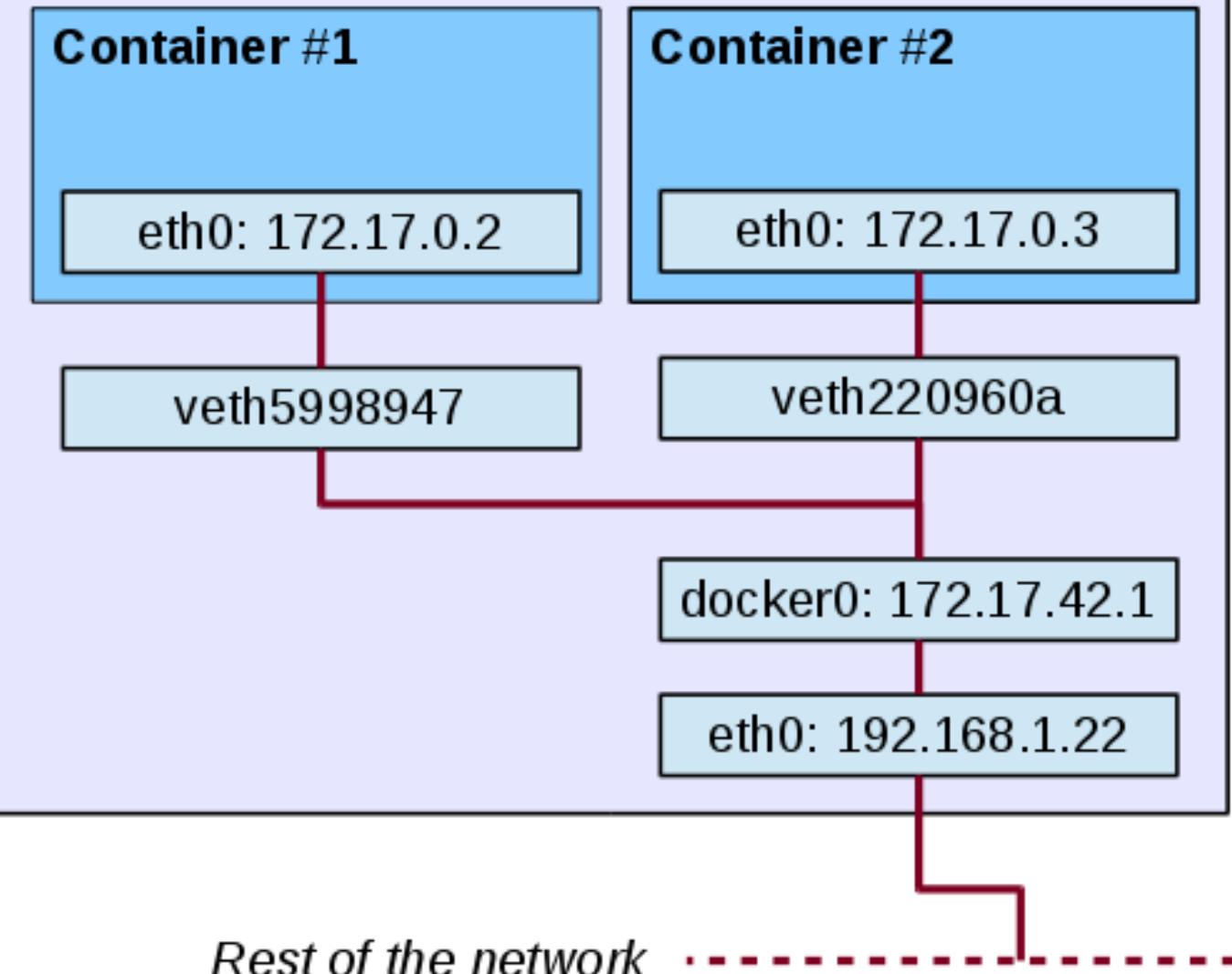
- Une fois le conteneur opérationnel, il expose un ou plusieurs ports (par exemple, le conteneur d'une API peut exposer le port 80).
- Étant donné que nous devons accéder au conteneur à partir de notre hôte local, nous devons mapper un port local sur l'un des ports exposés par le conteneur.
- Si le conteneur expose le port 80, nous pouvons exécuter docker run -p 8000:80 image-name.
- L'indicateur -p mappe les ports externes et internes, nous permettant d'accéder au conteneur en naviguant vers localhost:8000.



# Bridge Docker0

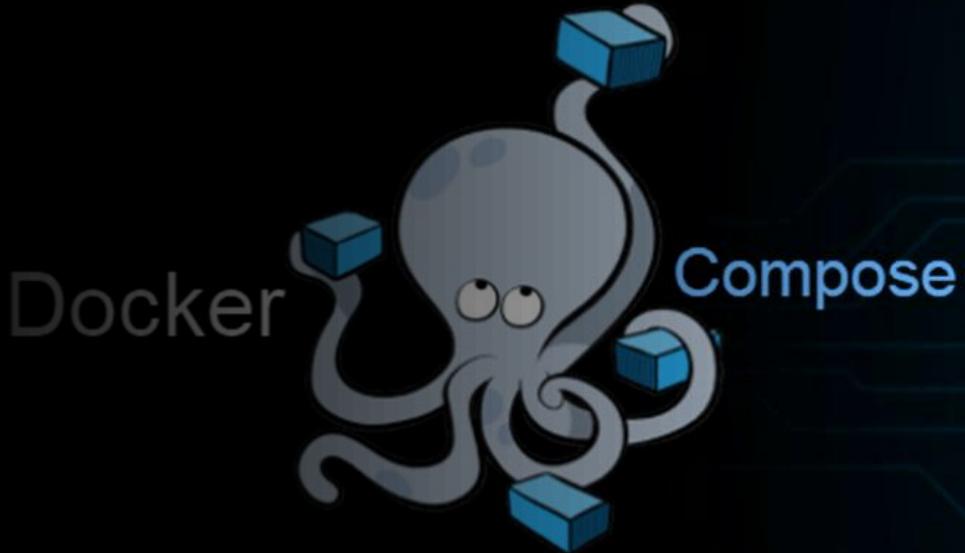
- Le pont (Bridge) **docker0** a l'adresse .1 du réseau Docker qui lui a été assigné, c'est généralement quelque chose autour de 172.17 ou 172.18.
- Les conteneurs se voient attribuer une interface **veth** qui est attachée au pont docker0.
- Les conteneurs créés sur le réseau Docker par défaut reçoivent l'adresse .1 comme route par défaut.

## Host running Docker



# Docker compose

---



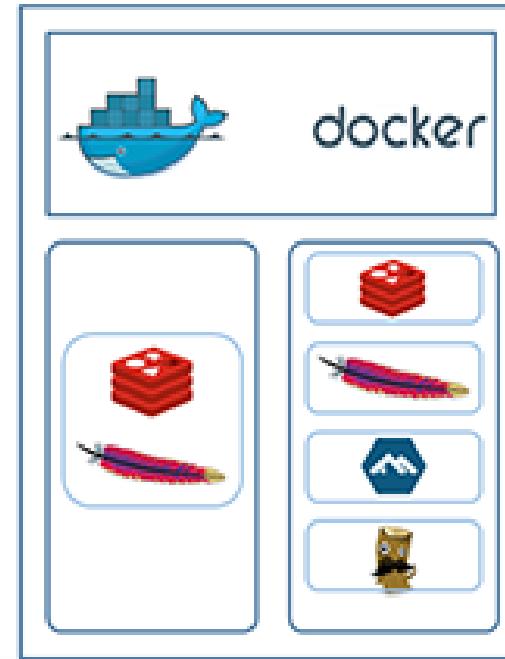
# Introduction

\$ docker-compose up

**docker-compose.yml**

```
version: '3'
services:
  web:
    build: .
    volumes:
      - web-data:/var/www/data
  redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      - default
```

**Docker Host**



- Docker Compose est un outil permettant de définir le comportement de vos conteneurs et d'exécuter des applications Docker à conteneurs multiples.
- La config se fait à partir d'un fichier YAML, et ensuite, avec une seule commande, vous créez et démarrez tous vos conteneurs de votre configuration.

# Docker compose

- Avec les applications à plusieurs niveaux (tiers), Dockerfile et les commandes d'exécution deviennent de plus en plus complexes.
- Docker Compose est un outil pour rationaliser la définition et l'instanciation des applications Docker multi-niveaux et multi-conteneurs.
- Compose nécessite un seul fichier de configuration et une seule commande pour organiser et augmenter le niveau d'application.
- Docker Compose simplifie la conteneurisation d'une application à plusieurs niveaux et à plusieurs conteneurs, qui peuvent être assemblées à l'aide du fichier de configuration **docker-compose.yml** et de la commande **docker-compose** pour fournir un service d'application unique.
- Le fichier Compose permet de documenter et de configurer toutes les **dépendances** de service de l'application (bases de données, files d'attente, caches, API de service Web, etc.)

# Docker compose - suite

Docker Compose définit et exécute des services complexes:

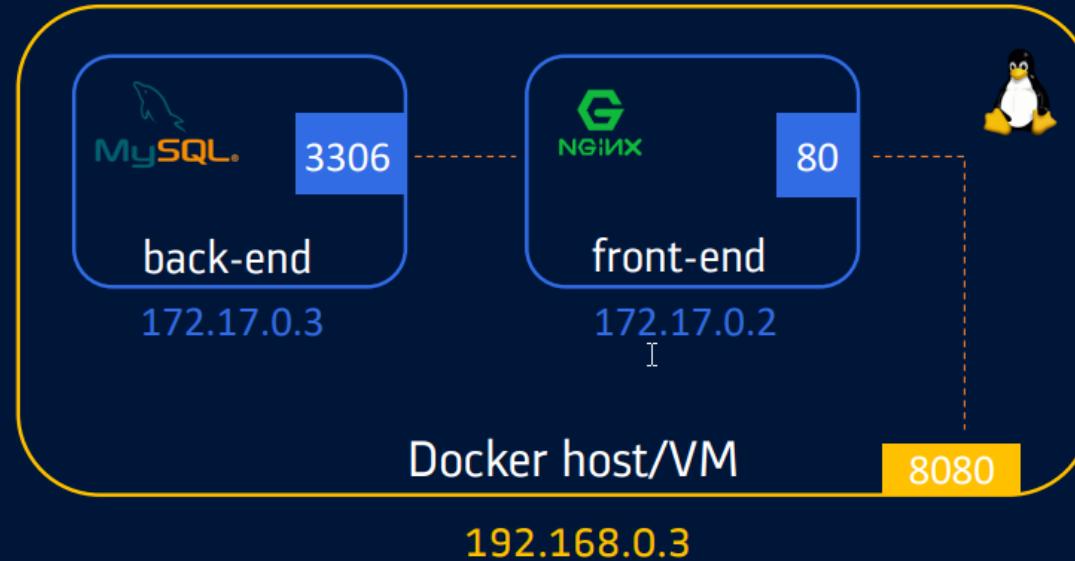
- définit des conteneurs uniques via Dockerfile
- décrit une application multi-conteneurs via un fichier de configuration unique ( docker-compose.yml )
- gère la pile d'applications via un seul binaire ( docker-compose up )
- lie des services via Service Discovery

# Application Multi-Tier

compose file: `docker-compose.yml`

Bring up the app: `docker-compose up -d`

Bring down the app: `docker-compose down`



- Par exemple, une application nécessite à la fois des conteneurs NGNIX et MySQL, vous pouvez créer un fichier qui démarrerait les deux conteneurs en tant que service (docker compose) ou démarrer chacun séparément (docker run)
- Tous les services doivent être définis au format YAML

# Docker-compose.yml

- Le fichier de configuration Docker Compose spécifie les services, les réseaux et les volumes à exécuter:
  - **services** - l'équivalent de passer des paramètres de ligne de commande à l'exécution de docker
  - **networks** - analogues aux définitions du réseau docker créer
  - **volumes** - analogues aux définitions du volume docker create

# YAML

- Le fichier de configuration Compose est au format déclaratif YAML :
- **YAML A in't M arkup L angage (YAML)**
- La philosophie de YAML est que «lorsque les données sont faciles à visualiser et à comprendre, la programmation devient une tâche plus simple»
- Convivial et compatible avec les langages de programmation modernes pour les tâches courantes
- Structure minimale pour un maximum de données:
  - L'indentation peut être utilisée pour la structure
  - Les deux-points (:) séparent les paires de clé : valeurs
  - Les tirets sont utilisés pour créer des listes à puces

# Exemple

```
version: "3"
services:
  web:
    build: .
    volumes:
      - web-data:/var/www/data
  redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      - default
```

# Docker Compose

## Compose file syntax

**version:** '3' # if no version is specified then v1 is assumed.

**services:** # containers. same as docker run

**service\_name:** # container name. this is also DNS name inside network

**image:** # name of the image

**command:** # Optional, replace the default CMD specified by the image

**environment:** # same as -e in docker run

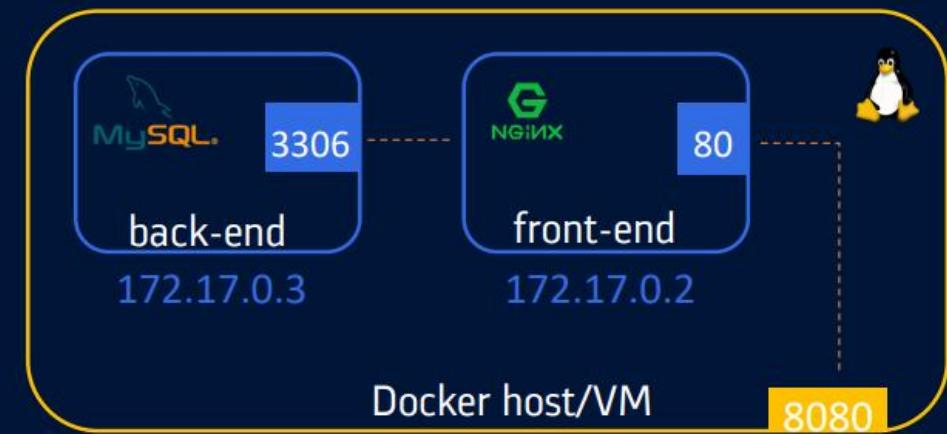
**ports:** # same as -p in docker run

**volumes:** # same as -v in docker run

**service\_name2:**

**volumes:** # Optional, same as docker volume create

**networks:** # Optional, same as docker network create



# Versions Docker compose

- **Version 1**
  - Les fichiers Compose qui ne déclarent pas de version sont considérés comme "version 1«
  - Ne prend pas en charge les volumes nommés, les réseaux définis par l'utilisateur ou les arguments de construction
  - Chaque conteneur est placé sur le réseau de pont par défaut et est accessible depuis tous les autres conteneurs à son adresse IP. Vous devez utiliser des liens pour activer la découverte entre les conteneurs
  - Aucune résolution DNS à l'aide de noms de conteneurs
- **Version 2**
  - Les liens sont obsolètes. Résolution DNS via les noms de conteneurs
  - Tous les services doivent être déclarés sous la clé "services«
  - Les volumes nommés peuvent être déclarés sous la clé volumes, et les réseaux peuvent être déclarés sous la clé réseaux
  - Nouveau réseau de ponts pour connecter tous les conteneurs
- **Version 3**
  - Prise en charge de l'essaim docker



# Docker Compose

```

version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
  volumes:
    mysql-data:
  networks:
    my_net:

```

```

  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net

```

```

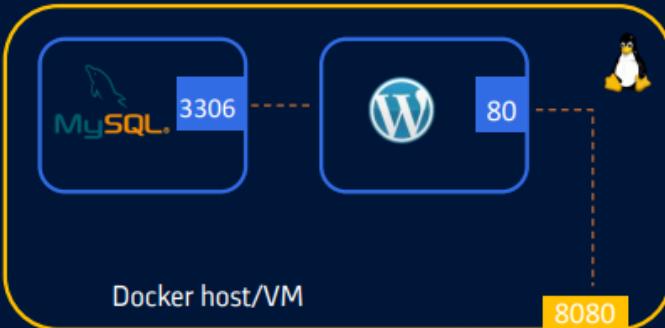
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net

```

```

  volumes:
    mysql-data:
  networks:
    my_net:

```





# Docker Compose

```

version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
volumes:
  mysql-data:
networks:
  my_net:

```

Services file: `docker-compose.yml`

Bring up: `docker compose up -d`

Bring down: `docker compose down`

Process state: `docker compose ps`

```

root@docker-master:/home/osboxes/docker# docker compose up -d
Creating network "docker_my_net" with the default driver
Creating volume "docker_mysql-data" with default driver
Pulling mysql (mariadb:...)

latest: Pulling from library/mariadb
23884877105a: Pull complete
bc38caa0f5b9: Pull complete
2910811b6c42: Pull complete
36505266dcc6: Pull complete
e69dcc78e96e: Pull complete
222f44c5392d: Pull complete
efc64ea97b9c: Pull complete
9912a149de6b: Extracting [=====] 115B/115B

```



| CONTAINER ID | IMAGE     | COMMAND                  | CREATED       | STATUS       | PORTS                |
|--------------|-----------|--------------------------|---------------|--------------|----------------------|
| 9784a2cc6e02 | wordpress | "docker-entrypoint.s..." | 5 minutes ago | Up 5 minutes | 0.0.0.0:8080->80/tcp |
| 2657b6db3f94 | mariadb   | "docker-entrypoint.s..." | 5 minutes ago | Up 5 minutes | 3306/tcp             |



# Docker Compose

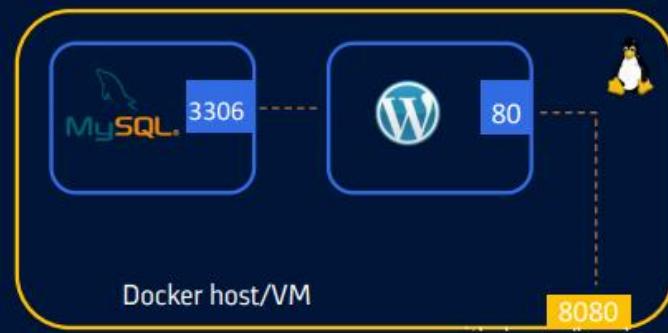
```
version: '3.3'  
services:  
  wordpress:  
    image: wordpress  
    depends_on:  
      - mysql  
    ports:  
      - 8080:80  
    environment:  
      WORDPRESS_DB_HOST: mysql  
      WORDPRESS_DB_NAME: wordpress  
      WORDPRESS_DB_USER: wordpress  
      WORDPRESS_DB_PASSWORD: wordpress  
    volumes:  
      - ./wordpress-data:/var/www/html  
    networks:  
      - my_net  
  mysql:  
    image: mariadb  
    environment:  
      MYSQL_ROOT_PASSWORD: wordpress  
      MYSQL_DATABASE: wordpress  
      MYSQL_USER: wordpress  
      MYSQL_PASSWORD: wordpress  
    volumes:  
      - mysql-data:/var/lib/mysql  
    networks:  
      - my_net  
  volumes:  
    mysql-data:  
  networks:  
    my_net:
```

## Deployment through imperative commands

```
docker network create --driver bridge my_net  
docker volume create mysql-data
```

```
docker run --name wordpress -p 8080:80 -v ./wordpress-data:/var/www/html \  
--net my_net -e WORDPRESS_DB_HOST=mysql \  
-e WORDPRESS_DB_NAME=wordpress \  
-e WORDPRESS_DB_USER=wordpress \  
-e WORDPRESS_DB_PASSWORD=wordpress \  
wordpress
```

```
docker run --name mariadb -p 3306 -v mysql-data:/var/lib/mysql --net my_net  
-e MYSQL_ROOT_PASSWORD=wordpress \  
-e MYSQL_DATABASE=wordpress \  
-e MYSQL_USER=wordpress \  
-e MYSQL_PASSWORD=wordpress \  
mariadb
```





# Docker Compose

## Image build

```

version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
  volumes:
    mysql-data:
  networks:
    my_net:
  
```

If the image has to be built before deployment, include dockerfile in compose file

```

version: "3"
services:
  wordpress:
    build:
      context: .
      dockerfile: Dockerfile-wordpress
      image: wordpress
      container_name: wordpress
  
```



To build only images: **docker compose build**  
 build + deploy: **docker compose up -d**

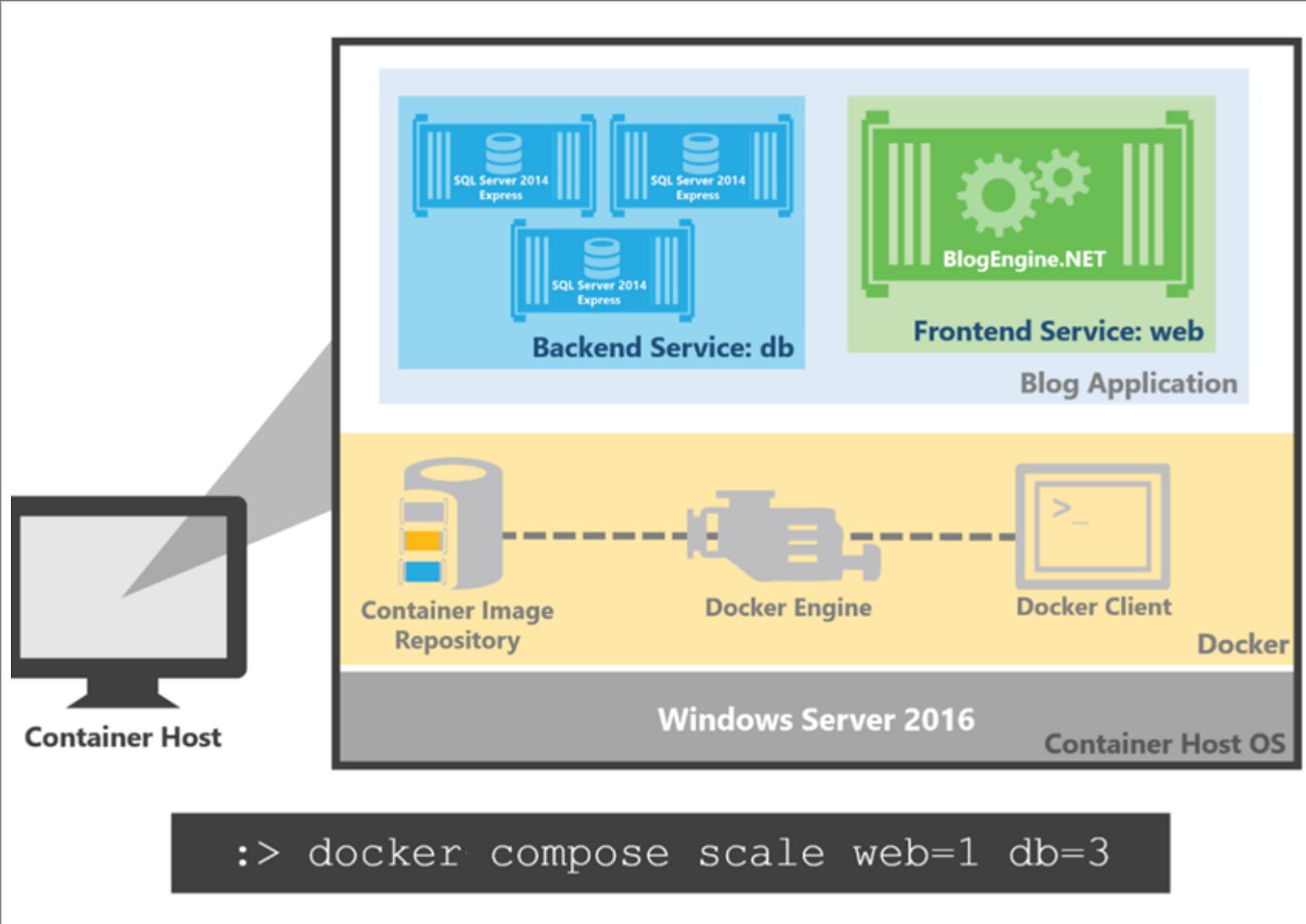


# Exemple fichier Docker-compose – Blog App

The diagram illustrates the Docker ecosystem for a blog application. It shows a Container Host (monitor icon) running a Docker Engine (train engine icon). The Docker Engine interacts with a Container Image Repository (cylinder icon) and a Docker Client (terminal icon). The Docker Client runs a Docker container (blue box with a gear icon). Inside the container, there are two services: Backend Service: db (SQL Server 2014 Express icon) and Frontend Service: web (BlogEngine.NET icon). The entire setup is labeled "Blog Application".

```
# docker-compose.yml
version: '2'
services:
  web:
    image: blogengine
    ports:
      - "80:80"
    depends_on:
      - db
    tty:
      true
  db:
    image: blogdb
    expose:
      - "1433"
    tty:
      true
networks:
  default:
    external:
      name: "nat"
```

# Scale-out avec Docker- compose



# Docker compose replicas

```
1 version: '3'  
2 services:  
3   web:  
4     image: nginx:latest  
5     deploy:  
6       replicas: 3  
7       endpoint_mode: dnsrr  
8     volumes:  
9       - ./webapp:/usr/share/nginx/html  
10    depends_on:  
11      - db  
12  
13  db:  
14    image: mysql:latest  
15    environment:  
16      - MYSQL_ROOT_PASSWORD=password  
17      - MYSQL_DATABASE=myapp  
18      - MYSQL_USER=user  
19      - MYSQL_PASSWORD=password  
20    volumes:  
21      - dbdata:/var/lib/mysql  
22    volumes:  
23      dbdata:
```

# Docker - Port Mapping

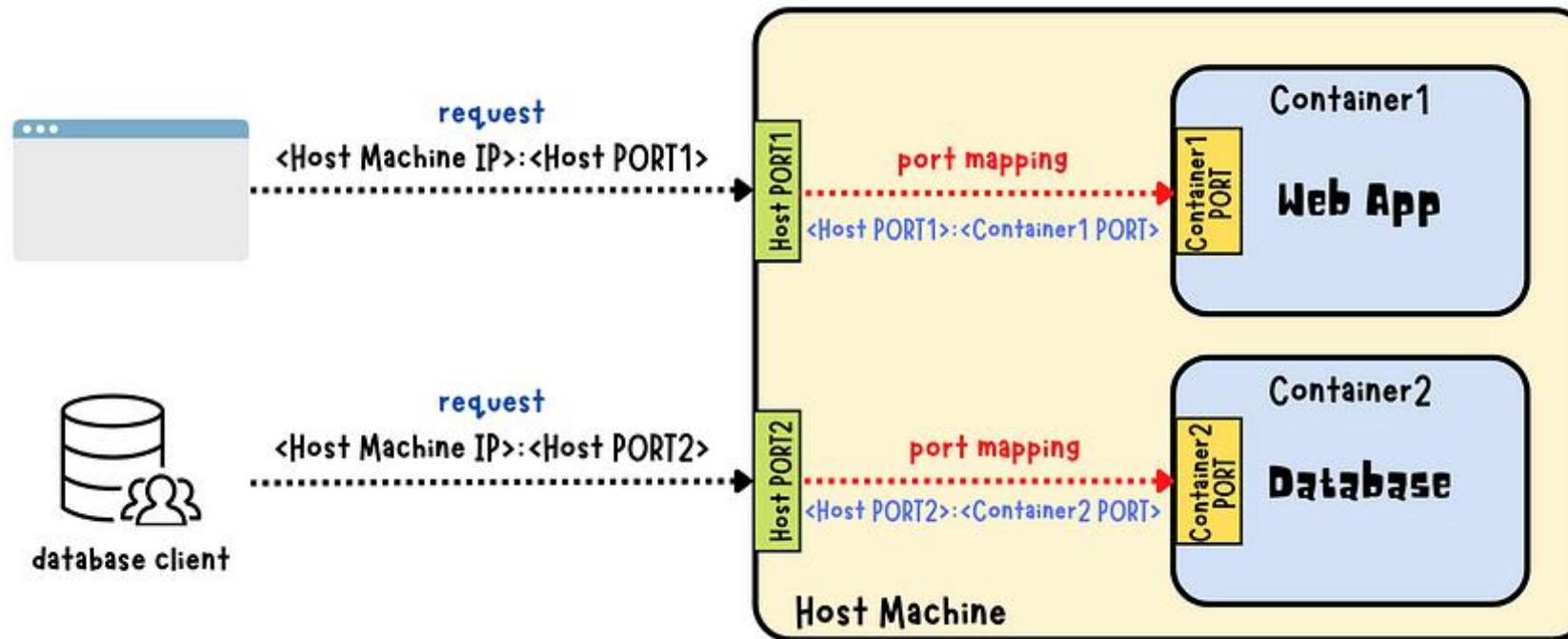


- 1 `docker run -p <Host_PORT>:<Container_PORT> <image>`
- 2 `docker-compose.yml`

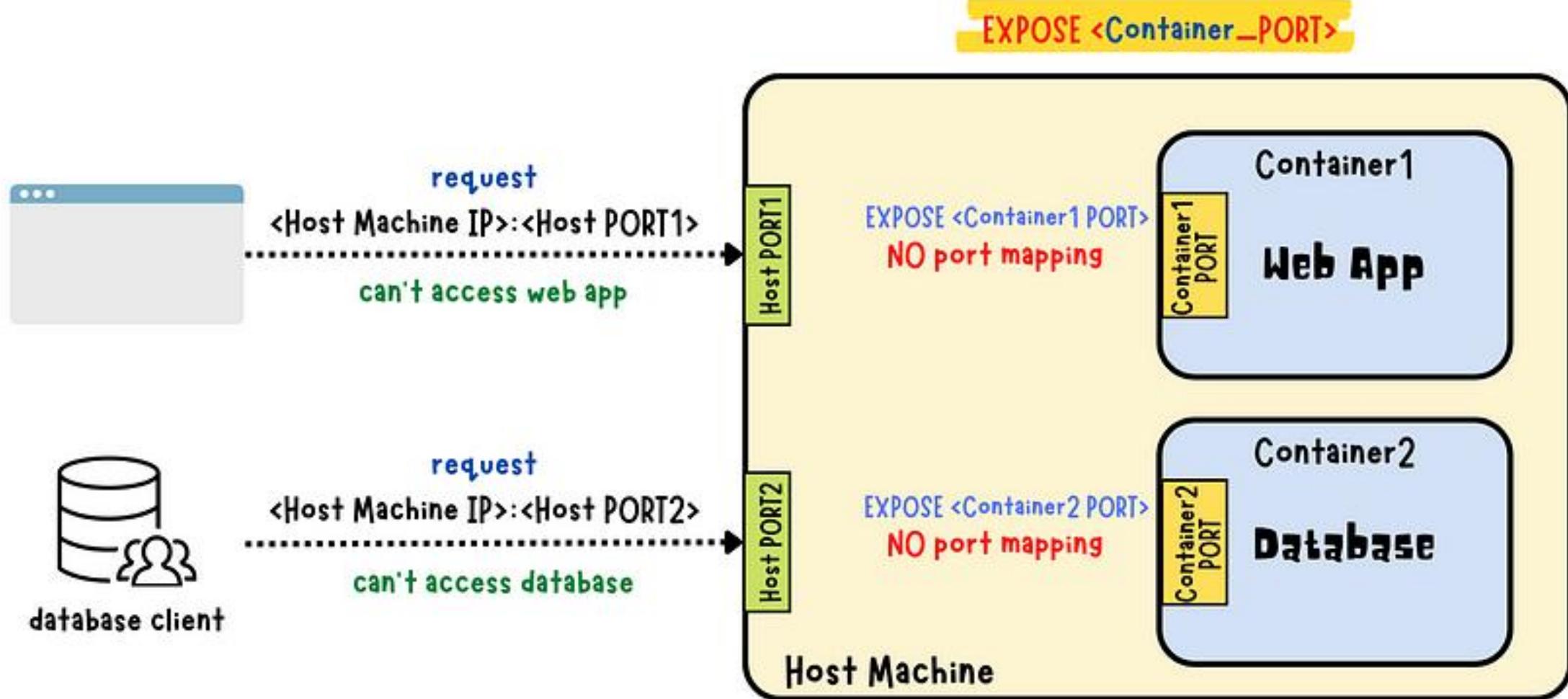
<Host\_PORT>:<Container\_PORT>

ports:

<Host\_PORT>:<Container\_PORT>



# EXPOSE

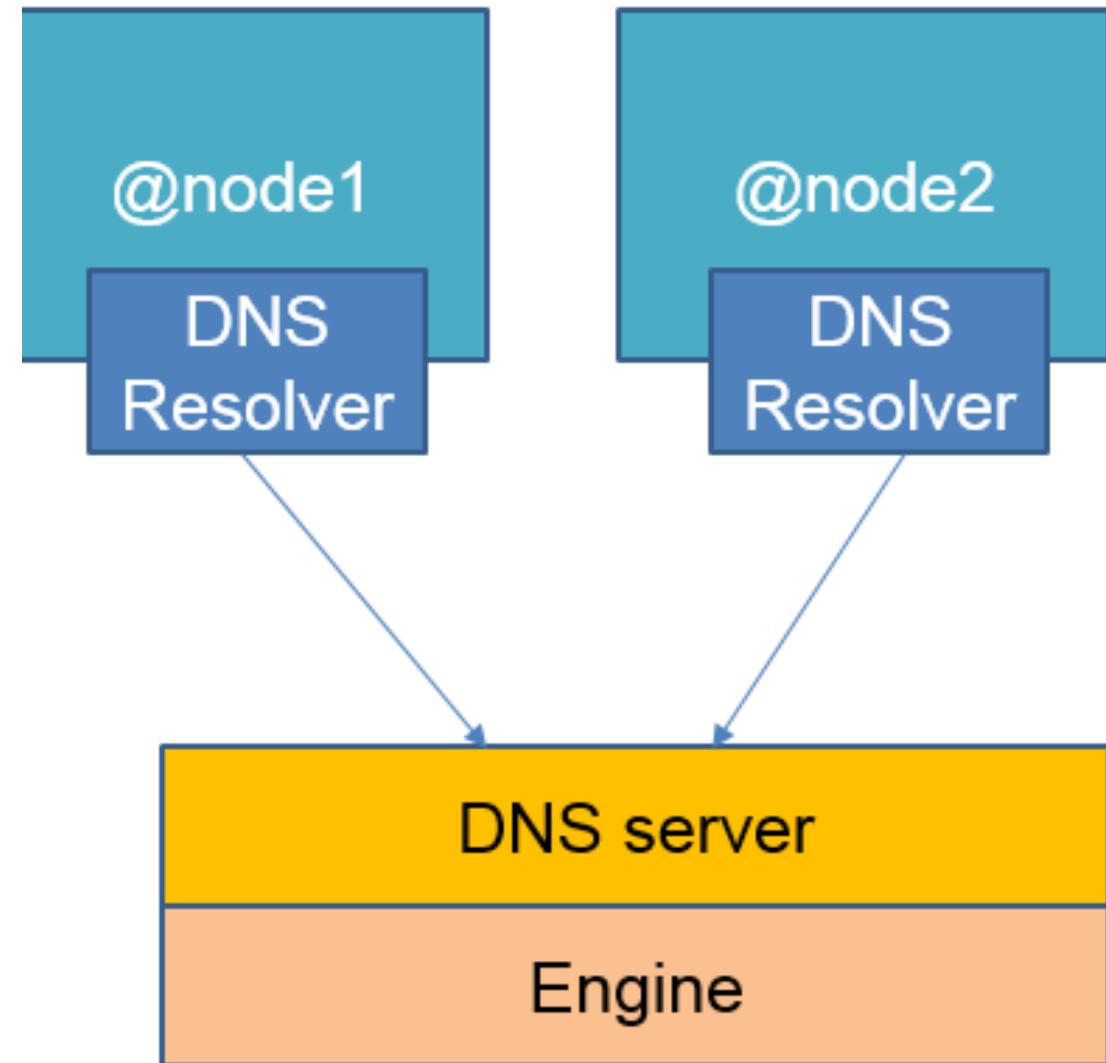


# Service Discovery

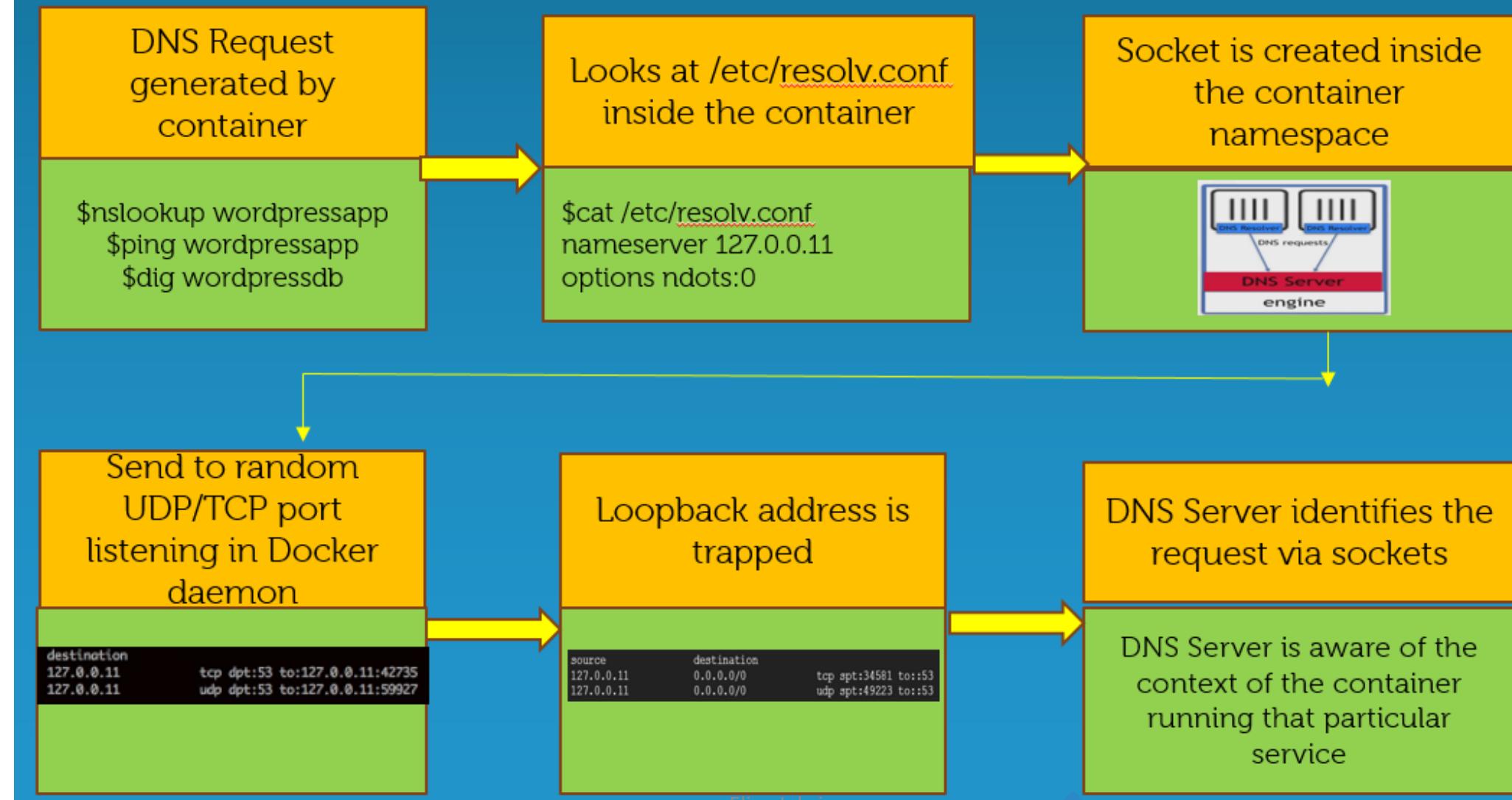
- Service Discovery en lui-même est une exigence clé pour faire évoluer les applications multiservices à l'aide de l'équilibrage de charge (scale-out) basé sur DNS.
- La découverte de service (Service Discovery ) est intégrée à Docker, et offre deux avantages clés:
- l'enregistrement du service et le mappage du nom du service vers l'IP (DNS).
- La découverte de services est particulièrement utile dans le contexte d'applications évolutives, car elle permet de découvrir et de référencer les services multi-conteneurs de la même manière que les services à conteneur unique;
- avec Service Discovery, la communication intra-application est simple et concise: n'importe quel service peut être référencé par son nom, quel que soit le nombre d'instances de conteneur utilisées pour exécuter ce service.

# Service Discovery

- Service fourni par du DNS embarqué
- Hautement disponible (HA)
- Utilise le Network Control Plane pour identifier les états
- Peut être utilisé pour découvrir les services et les conteneurs



# How Embedded DNS resolve unqualified names?





# Docker Swarm

---

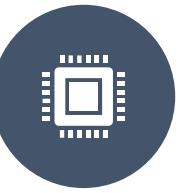


# Docker Swarm

- Solution native (officielle) de Docker pour faire du clustering
- Transformer un ensemble d'hôtes Docker en un unique hôte virtuel Docker.
- Utilise la même API
- Tous les outils qui communiquent avec un daemon Docker peuvent utiliser Swarm (ex: Dokku, Compose, Machine, Jenkins et bien sur le client Docker).
- Alternatives : Shipyard, Mesos, Kubernetes, Rancher,
- ...

# Remarque : Swarm avant Docker 1.12

---



LE  
DÉPLOIEMENT  
D'UN CLUSTER  
DE SERVEUR  
DOCKER AVEC  
SWARM ÉTAIT  
LOURD



GÉNÉRER DES  
CERTIFICATS



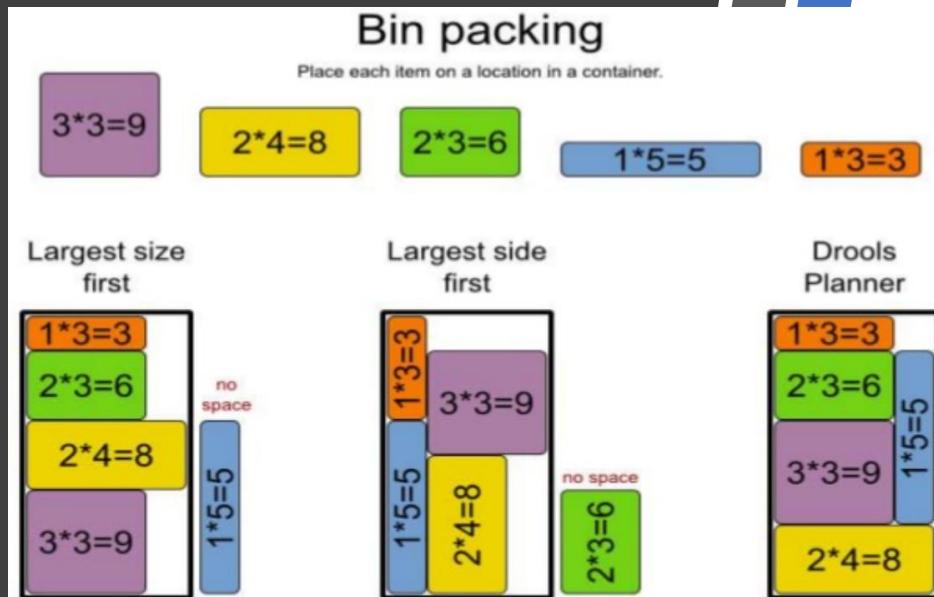
UTILISER UN  
SERVICE  
DISCOVERY



CONFIGURER  
CHAQUE  
NŒUD

# Docker Swarm

Permet d'héberger et d'ordonnancer une grappe de container Docker



Le scheduling backend peut être modifié/remplacé facilement (principe du Swap plug'n play).  
Par défaut : Bin packing.

# Création d'un cluster Swarm

- Pull d'une image « Docker Swarm »
- Configurer le Swarm Manager et les Workers
- (nœuds physique apte à héberger des containers).
- Ouvrir un port TCP sur chaque nœud pour communiquer avec le Swarm manager.
- Installer Docker sur chaque nœuds (>1.12)
- Créer et gérer des certificats TLS pour sécuriser son cluster.

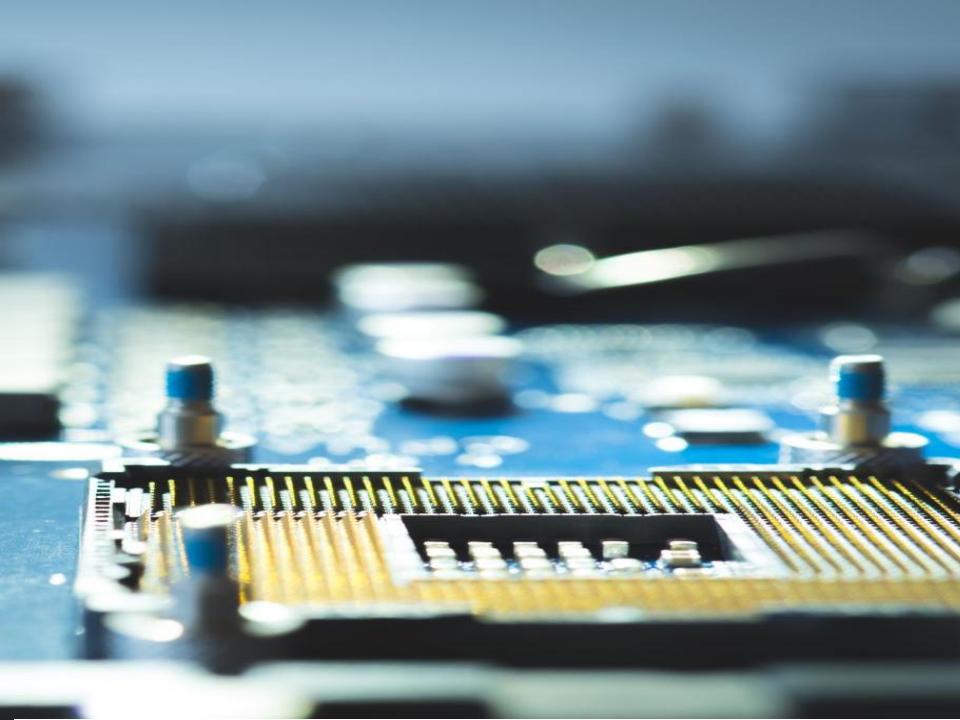
# Installation d'un Docker Swarm Cluster

- Deux méthodes :
  - Exécuter une image Swarm dans un container
  - Installer et utiliser le binaire sur son système.
- Avantages de la première méthode
  - Image construite par Docker et mise à jour régulièrement (`docker run ... swarm:latest`)
  - Pas d'installation de binaire sur son système pour utiliser l'image
  - Une seule commande (`docker run`) pour obtenir et exécuter la version la plus récente
  - Le container isole Swarm de votre environnement.

# Création d'un Docker Swarm

# Docker Swarm init

- Initialisation d'un Swarm. Le nœud physique devient un nœud Manager.  
Docker Swarm init génère deux token aléatoires
  - Un Worker token
  - Un Manager token
- Utiliser l'option --advertise-addr si la machine a plusieurs adresses IP.
- Pour ajouter un nouveau Worker au cluster Swarm on utilisera le Worker token.



```
$ docker swarm init --advertise-addr 192.168.99.121
Swarm initialized: current node (bvz81updecsj6wjz393c09vti) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token SWMTKN-1-3pu6hszjas19xyp7ghgosyx9k8atbfcr8p2is99znpv26u21kl-1awxwuwd3z9j1z3puu7rcgdbx \
172.17.0.2:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

# Docker Swarm join

- Permet d'ajouter un nœud à un Swarm.
- Passer en paramètre le Worker token.

```
$ docker swarm join --token SWMTKN-1-3pu6hszjas19xyp7ghgosyx9k8atbfcr8p2is99znpv26u2lk1-1awxwuwd3z9j1z3puu7rcgdbx 192.1  
68.99.121:2377
```

This node joined a swarm as a worker.

```
$ docker node ls
```

| ID                        | HOSTNAME   | STATUS | AVAILABILITY | MANAGER STATUS |
|---------------------------|------------|--------|--------------|----------------|
| 7ln70f122uw2dvjn2ft53m3q5 | worker2    | Ready  | Active       |                |
| dkp8vy1dq1kxleu9g4u78tlag | worker1    | Ready  | Active       | Reachable      |
| dvxpx4zseq4s0rih1selh0d20 | * manager1 | Ready  | Active       | Leader         |

# Docker Swarm leave

- Permet à un *Worker* de quitter le Swarm.
- A partir du manager :

```
$ docker node ls
ID          HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
7ln70fl22uw2dvjn2ft53m3q5  worker2  Ready  Active
dkp8vy1dq1kxleu9g4u78tlag  worker1  Ready  Active
dvfxp4zseq4s0rih1selh0d20 *  manager1  Ready  Active      Leader
```

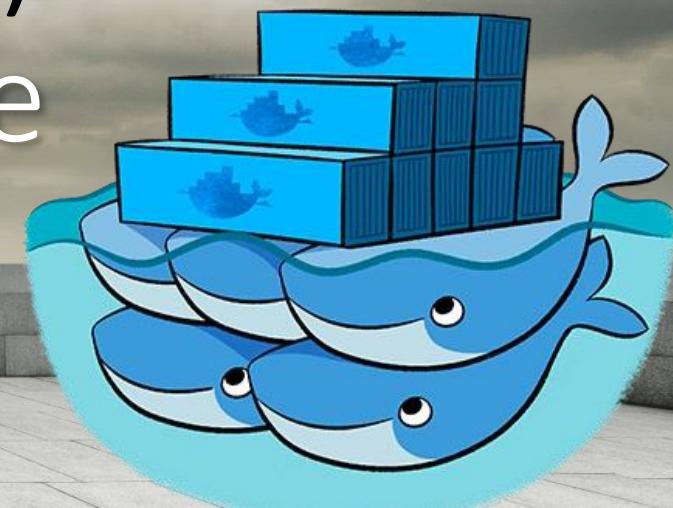
- A partir d'un worker :

```
$ docker swarm leave
Node left the default swarm.
```

- Pour supprimer un nœud inactif *docker node rm*

# Usage d'un Docker Swarm (déploiement de services)

## Mise en œuvre



# Déploiement

- Déploiement d'un service exposé sur nos nœuds et *load balancé* dans notre cluster.
  - Création d'un service qui expose le port 8001 de notre serveur vers 5 containers nginx.

```
docker service create --name nginx --replicas 5 -p 8001:80/tcp nginx
```

- Lister les containers

```
root@kube:~# docker service tasks nginx
ID          NAME      SERVICE
cmmni6a83fmqnk7h1wqb7yrdl  nginx.1  nginx
5o26sravvboq4d6femhmcjvh  nginx.2  nginx
1gnokp6s37fhrxn8warwu63tt  nginx.3  nginx
at7lmzqd4pq8e8f3bv5yt4sxl  nginx.4  nginx
dcyj70t557tz7o63cyvwps1v0  nginx.5  nginx
root@kube:~#
```

# Besoin de plus : Scale !

- On peut *scale* le service selon nos besoins

```
docker service scale nginx=40
```

```
root@k8s-node-01:~# docker service ls
          ID      NAME  REPLICAS  IMAGE
216bzwgp1lzutlmffputvw203  nginx.24
88wlk3s0l0lziaa3v9wclxmpe  nginx.25
ec5b6zbsd7i7t90m21veufh58  nginx.26
08kqc6q4301o0wgp494zr1i3r  nginx.27
bajmwp0766s32dpqj2uac5drb  nginx.28
d7onzqls9srifs72bqxfu6r2c  nginx.29
6er79s2g5qa4mmfk5efqc5mon  nginx.30
5u5dbzezne61mwuwc1qqtd7dc  nginx.31
9x9y0e95suuty7xrrf2lofl3j  nginx.32
4wgpj14j9f3xfe0es6hl1quna  nginx.33
4nouu3najhv8yvsasxt6t67   nginx.34
de0ks32qnbs290k8iiqczs6li  nginx.35
9mycc8dq8vgj7iurgqhm3crf4  nginx.36
01aqcgzbl3ynu5eiboxp104fj  nginx.37
dnveqceuty4fe0ta3bs12buqm  nginx.38
8h4hh1dtt8mq2jecnyot02l4b  nginx.39
1l9o7rconzta7huqaubnggqh1  nginx.40
root@kube:~# _
```