

# **Les conteneurs -- Docker et Docker Compose**

**Thomas Ropars**

**Email: [thomas.ropars@univ-grenoble-alpes.fr](mailto:thomas.ropars@univ-grenoble-alpes.fr)**

**Website: [tropars.github.io](https://tropars.github.io)**

# References

Les ressources suivantes ont été utilisées pour préparer ces slides:

- Les formations proposées par [Jérôme Petazzoni](#)
- En particulier, les slides sur [Docker](#)

# **Programme de la formation**

- Lundi matin (4h): Présentation -- Les conteneurs avec Docker et Docker Compose
- Lundi après midi (3h): TP Docker
- Mardi matin:
  - 1h30 de cours/discussions
  - 1h30 de TP avancé
  - A adapter en fonction de notre progression

# Agenda

- Pourquoi les conteneurs?
- Les utilisations des conteneurs?
- Introduction à Docker
- Les images Docker
- La création d'images
- Publier des images sur Docker Hub
- Manipuler des conteneurs
- La gestion du réseau
- Gérer les données dans Docker
- Docker Compose

# **Pourquoi les conteneurs?**

# Evolution de l'industrie logicielle

## Avant

- Applications monolithiques
- Cycles de dev long
- Un seul environnement

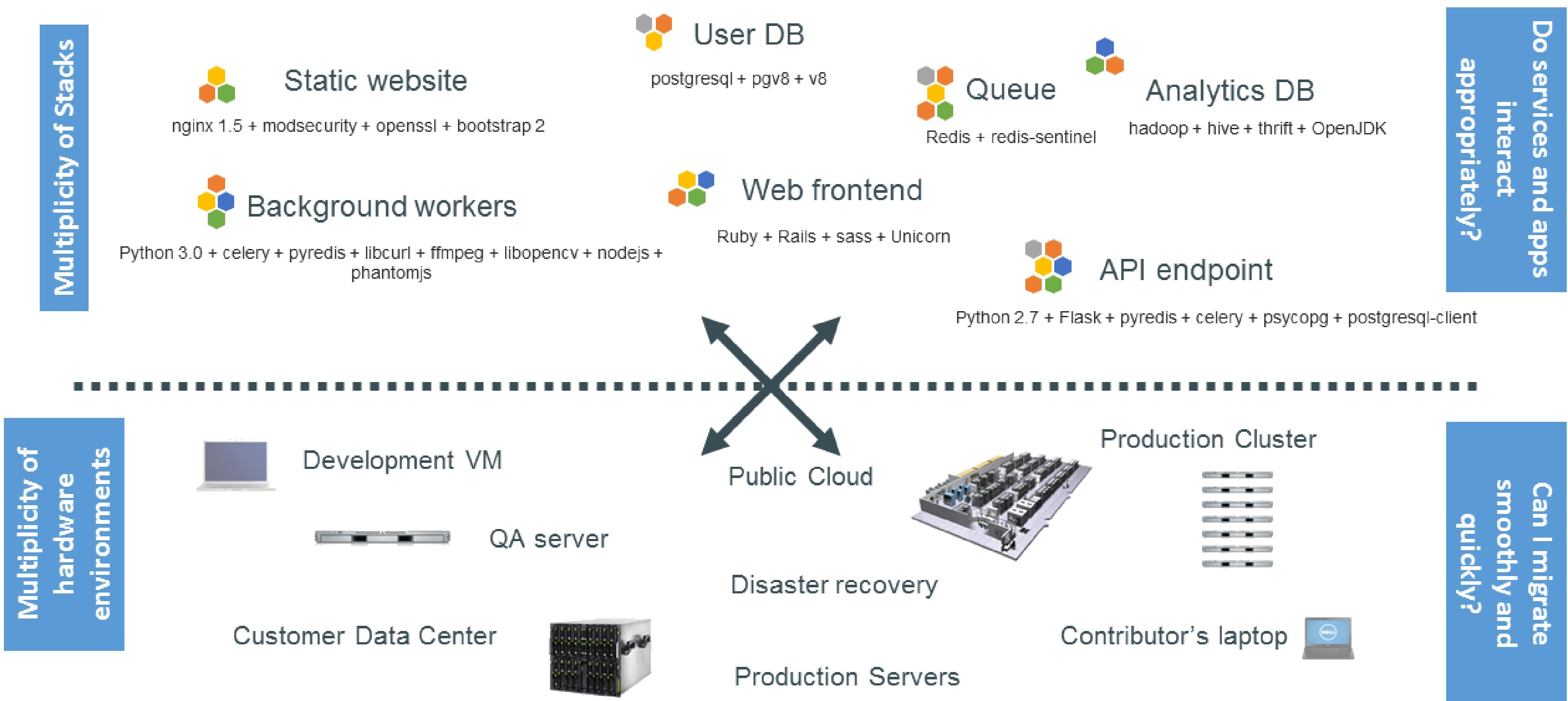
## Après

- Ensemble de services
- Améliorations rapides, cycles itératifs
- De nombreux environnements
  - Des serveurs de prod (dans le cloud?)
  - Des laptops de dev
  - Des serveurs de test (gitlab?)
  - etc.

# **Le déploiement devient complexe**

- Des piles logicielles différentes
  - Langages
  - Frameworks
  - Databases
- De nombreux environnements!

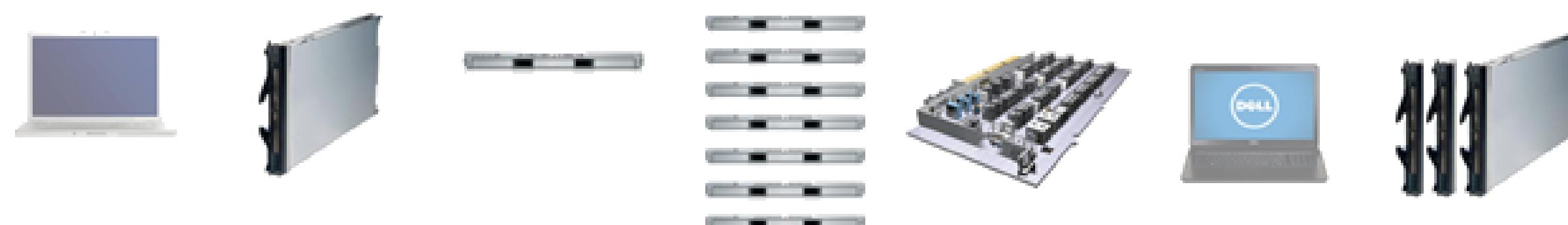
# Le déploiement devient complexe



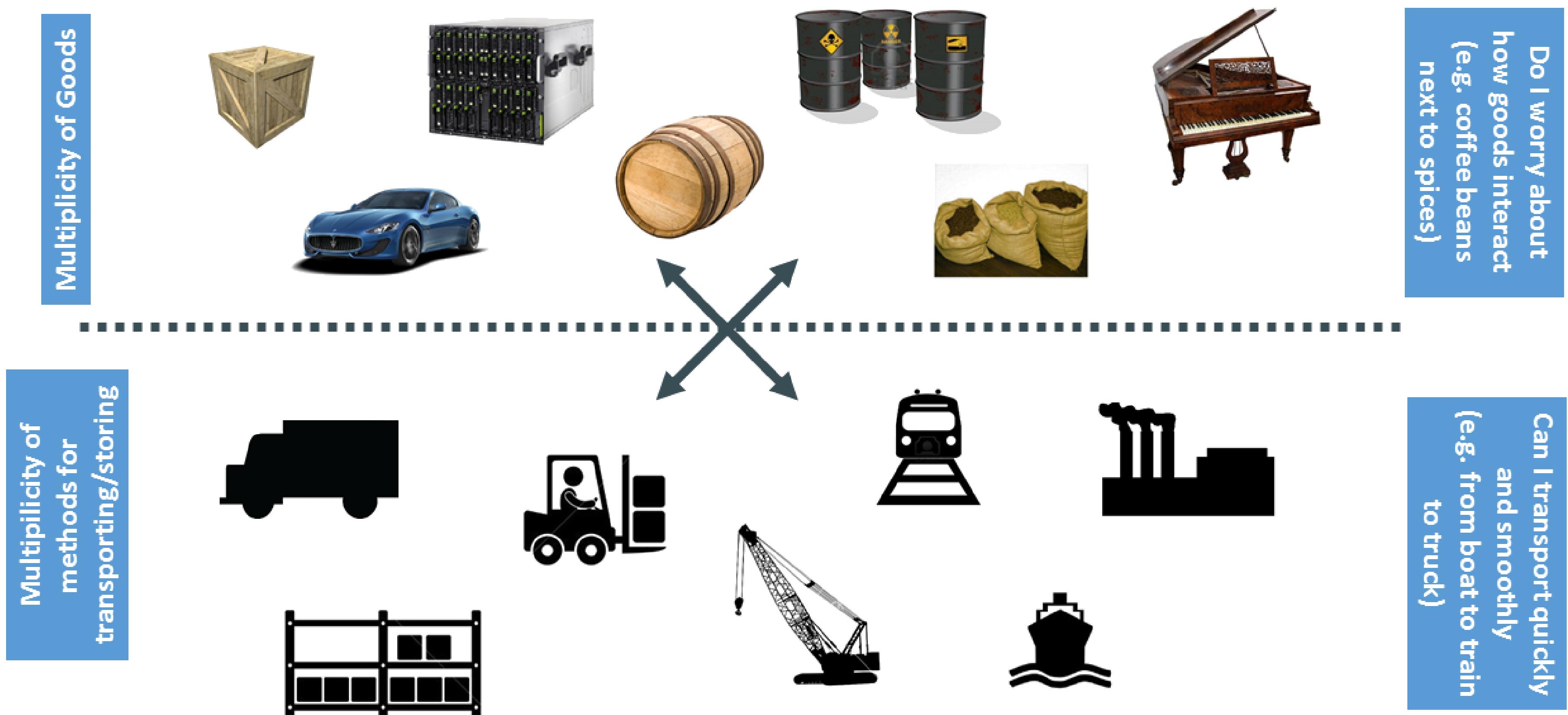
Credits: B. Golub

# The matrix from hell

	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers	



# Transport de marchandise (pré 1960)



Credits: B. Golub

# Une autre matrice ...

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?

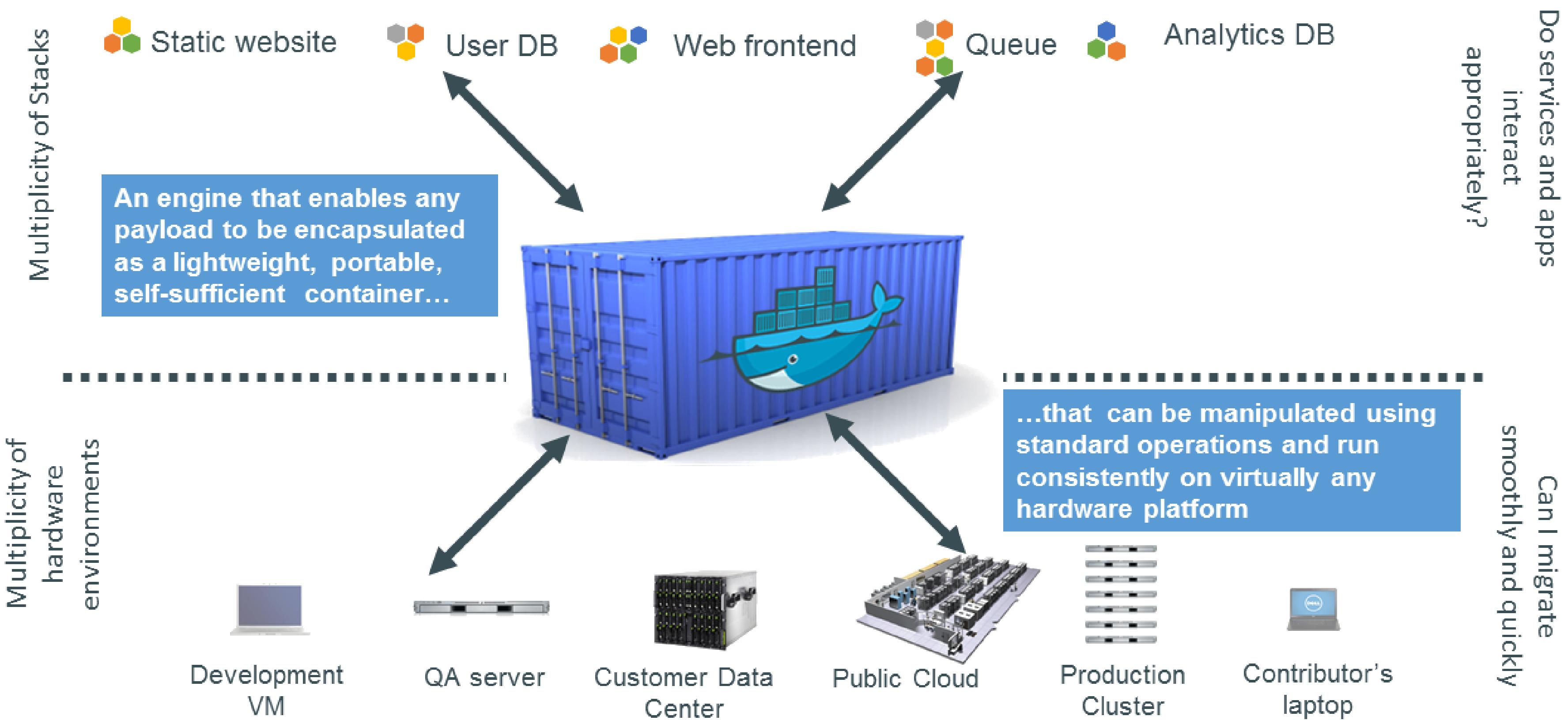
# Solution: des conteneurs!



Manipulation simplifiée d'un ensemble d'objets (ou d'applications) grâce à une interface standardisée.

Credits: B. Golub

# Des conteneurs pour le logiciel



Credits: B. Golub

# **Fin du "pourtant ça marche sur ma machine"**

1. Ecrire les instructions d'installation dans un fichier INSTALL.txt
2. Créer un script install.sh qui fonctionne
3. Le transformer en un Dockerfile
4. Créer une image Docker à partir de ce Dockerfile

**Vous êtes prêts à exécuter sur n'importe quelle machine**

# Etre opérationel sur un projet rapidement

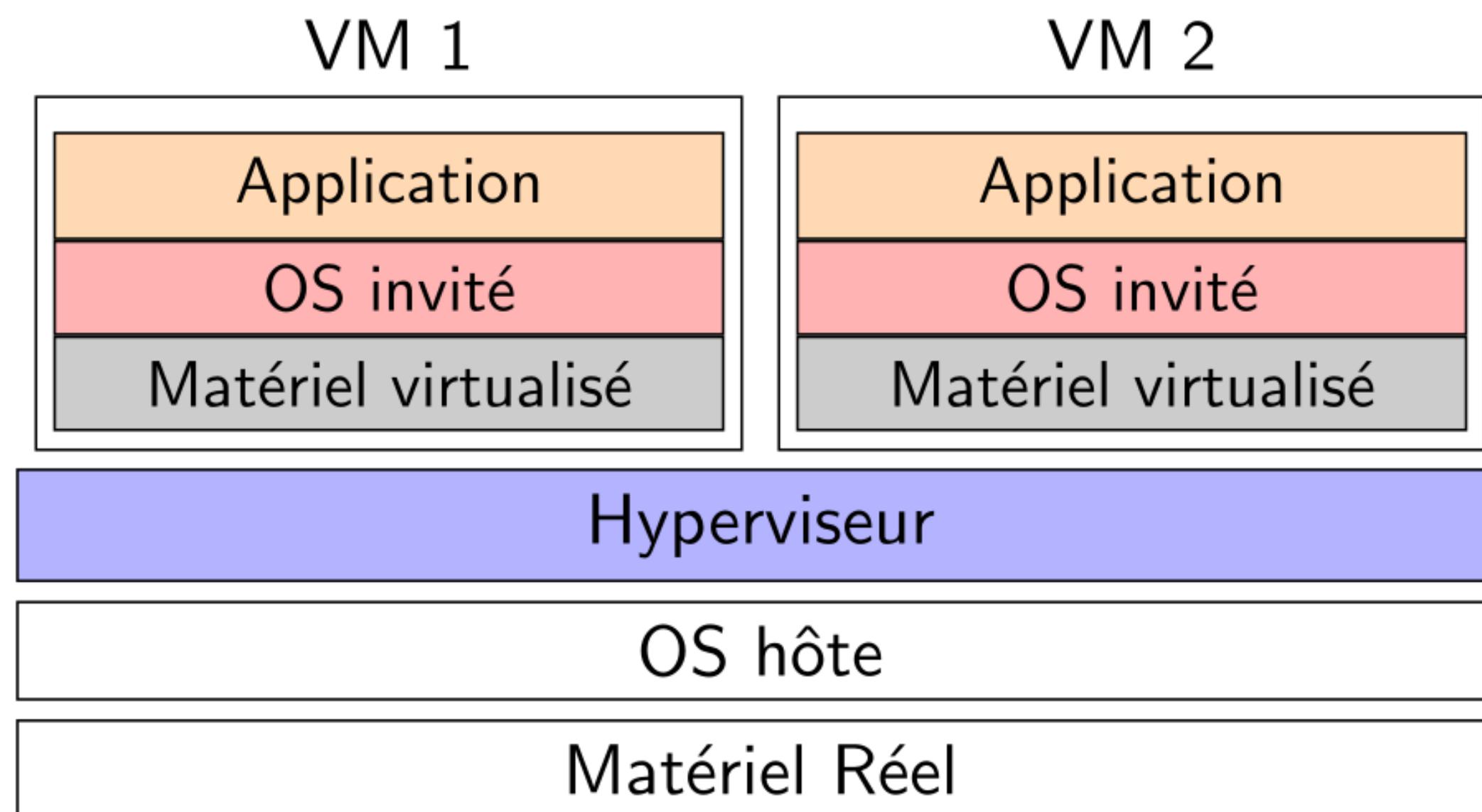
1. Ecrire des Dockerfile pour vos composants
2. Utiliser des images dispo sur Docker Hub (mysql, ect.)
3. Décrire votre application avec un Compose

N'importe qui devient opérationnel en 2 commandes

```
git clone ...
docker compose up
```

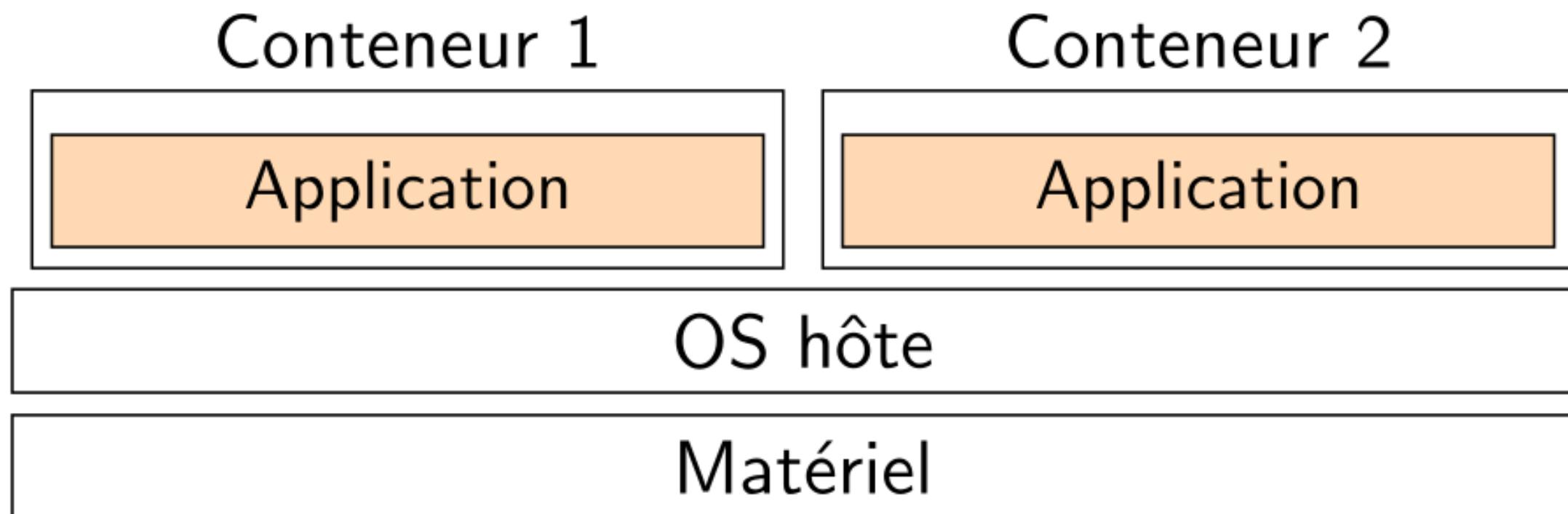
# **Les utilisations des conteneurs**

# Conteneurs vs machines virtuelles



- Une image de machine virtuelle contient un OS complet

# Le cas des conteneurs



- Une image de conteneur ne contient que les bibliothèques nécessaires au composant conteneurisé
- Environnement d'exécution isolé au dessus du système d'exploitation
  - Les conteneurs partagent le même noyau et une grande partie des services de l'OS hôte
  - *Virtualisation* de l'environnement d'exécution (espace de nommage, système de fichier isolé, quotas, etc )

# Avantages des conteneurs

- Meilleures performances
  - Accès direct au matériel
- Démarrage beaucoup plus rapide
  - Pas besoin de démarrer un système complet
- Images plus légères
  - Ne contient que les informations en lien avec l'application
  - Moins coûteux en terme d'espace de stockage
  - Plus rapide à transférer

# Utilisation pour de l'intégration continue (CI)

1. Environnement de test créé à partir d'une image Docker
  - Les tests peuvent être exécutés sur n'importe quelle plateforme (plateforme de CI)
2. Un nouveau conteneur créé pour chaque étape de tests
  - Pas de *pollution* entre les étapes d'un test
  - Pas de *pollution* entre plusieurs exécutions des tests
3. Les tests peuvent être exécutés très souvent

# **Utilisation pour la distribution d'artefacts**

1. Construire notre application à partir de Dockerfiles
2. Stocker les images dans un registre
  - Stockage pérenne
  - Accessible pour tout le monde
3. Exécuter en production
  - Les images contiennent toutes les dépendances pour vos applications
4. Versionage
  - Possibilité de tagger les images

# Découpage de la "plomberie" et de la logique applicative

1. Utiliser des noms de services dans votre code (bd, api, etc)
2. Utiliser une composition pour démarrer votre application
  - (ou un orchestrateur si sur plusieurs serveurs)
3. Résolution de nom faite de manière automatique
4. On peut redimensionner, faire de l'équilibrage de charge, répliquer sans changer le code

# **Passage du dev à la production (DevOps)**

1. Créer une image de conteneur et une composition
2. L'image peut être directement déployé en production
  - Même environnement pour le dev, le test, et la production
  - Déploiement simplifié
  - Les devs peuvent se charger du déploiement
3. Le passage en production est fluidifié

# Infrastructure as code

- L'infrastructure est décrite dans des fichiers texte
  - Les Dockerfile (= du code)
  - Sert aussi de documentation de l'infrastructure
- Cette description peut être versionnée (dépot git)
  - Suivi des modifications
  - Possibilité de revenir en arrière
- Mise en place automatique de l'infrastructure
  - Déterministe
  - Reproductible

# **Introduction à Docker**



- Crée en 2011
  - A popularisé l'utilisation de conteneurs
- D'autres technologies de conteneurs **existent**
  - Singularity
  - Podman
  - etc.
- Une partie gratuite et une partie payante

# **Services fournis par Docker**

- Construction d'images
- Gestion d'images
  - Localement sur une machine
  - Globalement (Registre -- Docker hub)
- Exécution et gestion de conteneurs

# Qu'est ce qu'un conteneur?

Les technologies de conteneurs offrent une solution de *packaging* pour une application et ses dépendances.

## Les images de conteneurs

- Package de l'application et de ces dépendances
  - Peut être exécutée sur différents environnements
  - Décris par un fichier texte.
    - Infrastructure-as-code

## Un Conteneur

- Une instance d'une image de conteneur
  - S'exécute dans un environnement isolé

## Analogie POO

- Une image = une classe
- Un conteneur = une instance

# Docker: les briques principales

## Docker engine

Un environnement d'exécution et un ensemble de services pour manipuler des conteneurs docker sur une machine

- Une application **client-serveur**
  - Le serveur -- Un *daemon* (processus persistant) qui gère les conteneurs sur une machine
  - Le client -- Une interface en ligne de commande

## Un/des registres d'images docker

- Bibliothèque d'images disponibles
- Docker Hub

# Registre Docker

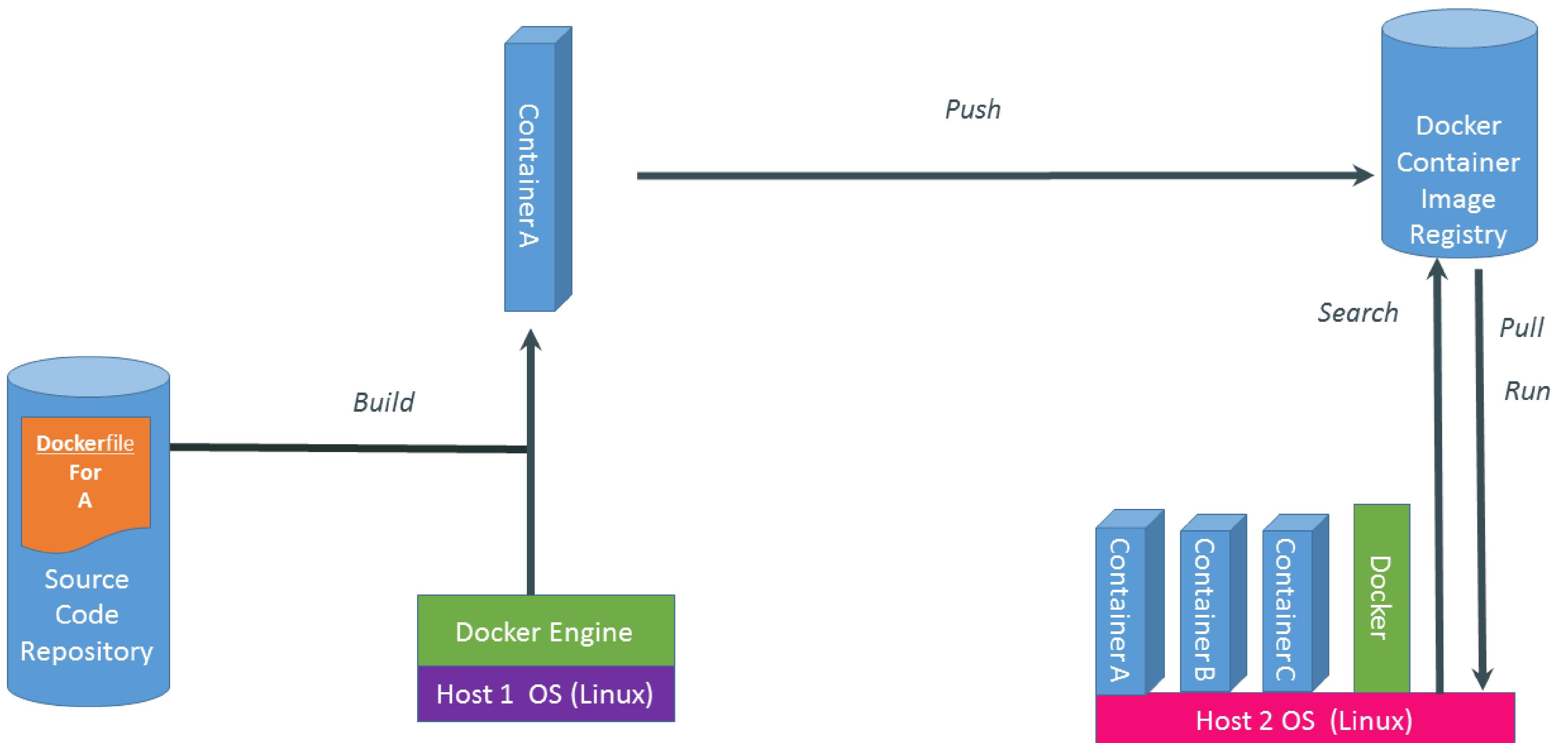
## Principe

- Un serveur stockant des images docker
- Possibilité de récupérer des images depuis ce serveur (*pull*)
- Possibilité de publier de nouvelles images (*push*)

## Docker Hub

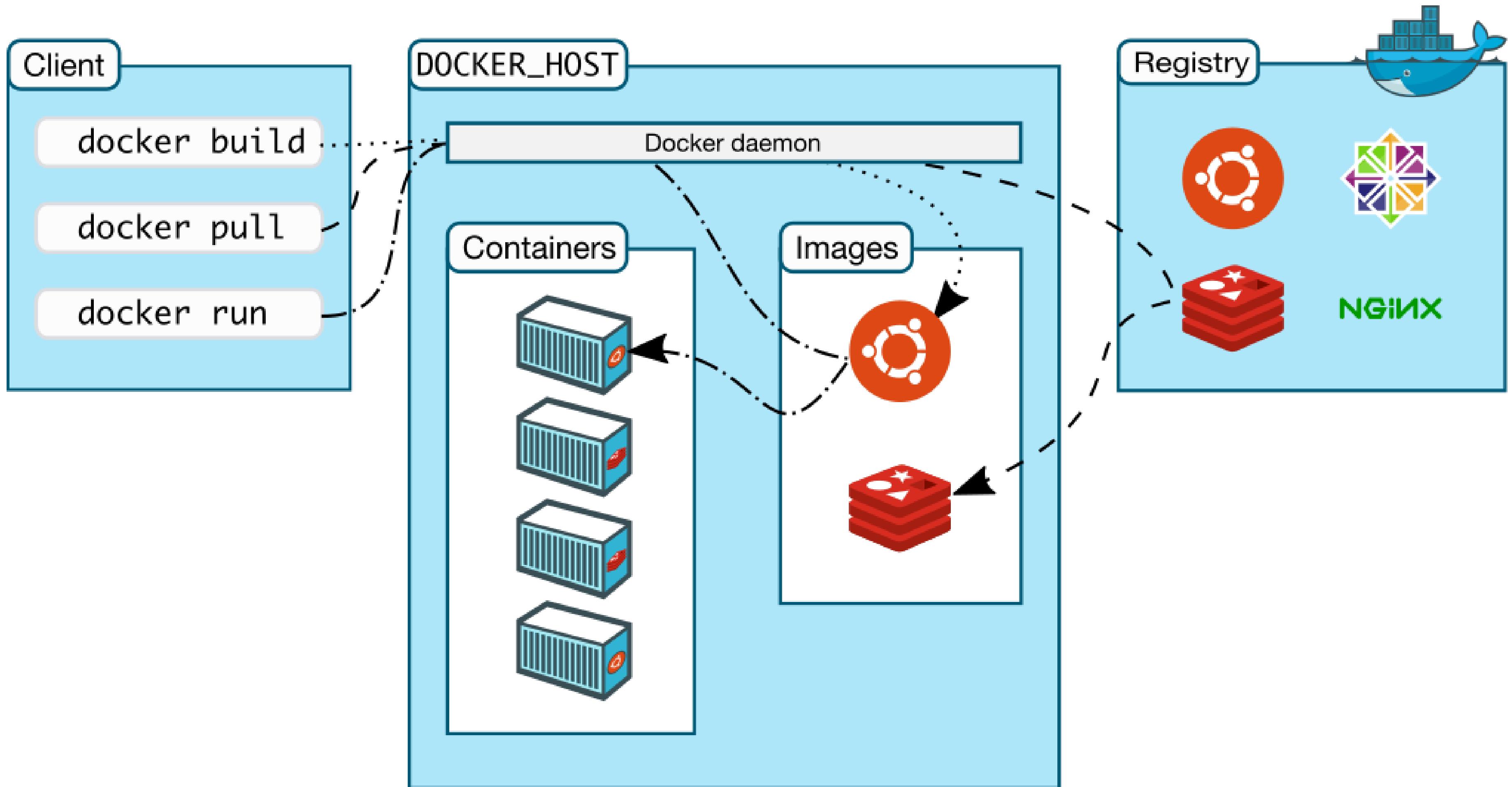
- Dépôt publique d'images Docker

# Principes de fonctionnement de Docker



Crédits: B. Golub

# Architecture de Docker



Crédits: <https://docs.docker.com/get-started/overview/>

# **Les images Docker**

# **Images Docker vs Images de VM**

## **Les images de machines virtuelles**

- Sauvegarde de l'état de la VM (Mémoire, disques virtuels, etc) à un moment donné
- La VM redémarre dans l'état qui a été sauvegardé

## **Les images Docker**

- Une copie d'une partie d'un système de fichier
- Pas de notion d'état

# Principes des images Docker

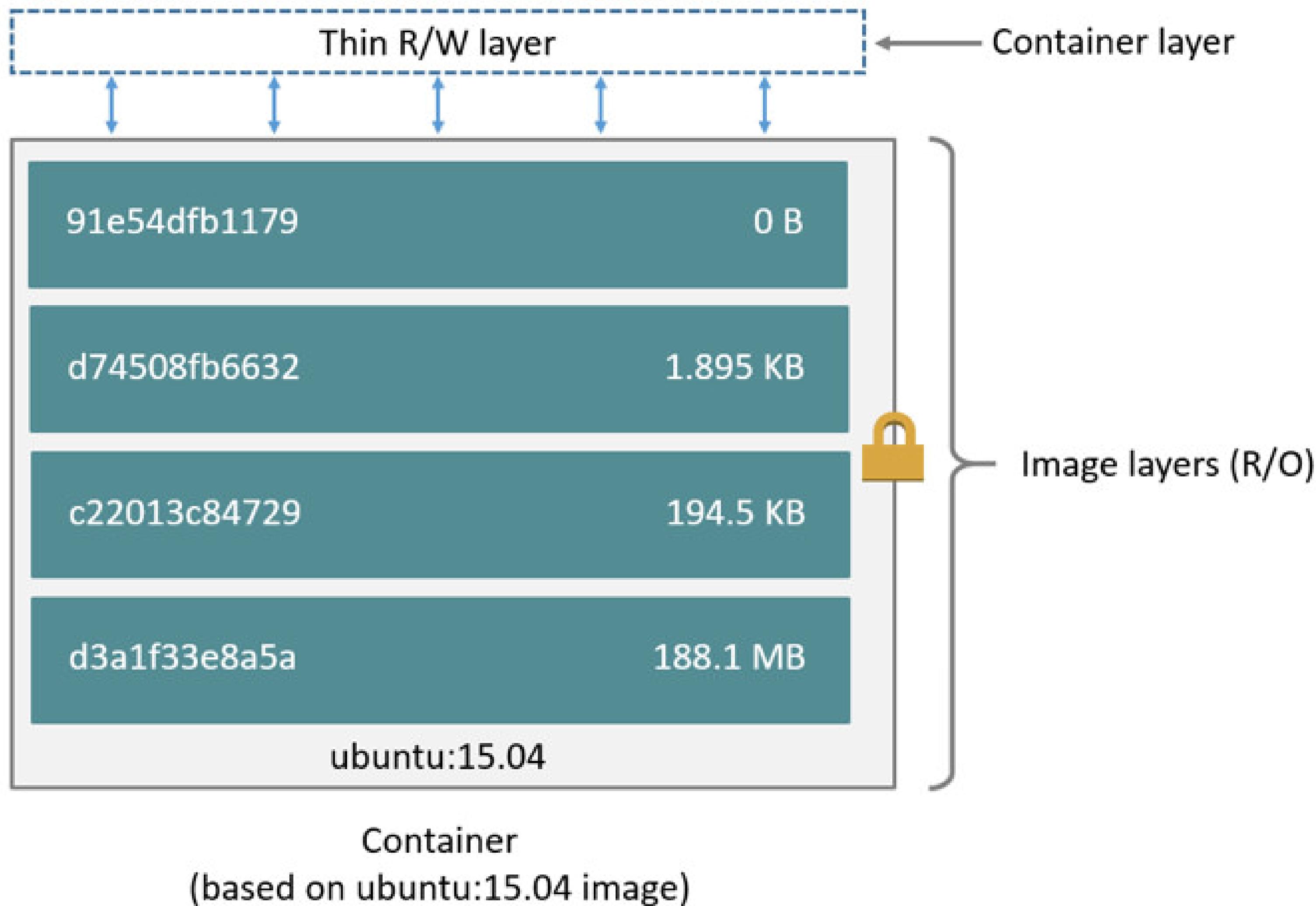
## Fondé sur l'utilisation d'un Union File System

- Crée la vision d'un système de fichier cohérent à partir de fichiers/répertoires appartenant à des systèmes de fichiers différents

## Un ensemble de couches

- Une image est composée d'un ensemble de couches (*layers*)
- L'Union File System est utilisé pour combiner ces couches
  - Le file system utilisé par défaut s'appelle `overlay2`
- Chaque couche correspond à une instruction dans le fichier `Dockerfile` décrivant l'image.

# Les couches



# **Les couches**

## **Images de base**

- Toute image est définie à partir d'une image de base
- Des images de base officielles sont déjà fournies
- Exemples: `ubuntu:latest`, `ubuntu:14.04`, `opensuse:latest`, `alpine:latest`
  - `alpine`: Image de base minimaliste (5MB) -- intéressant pour les performances

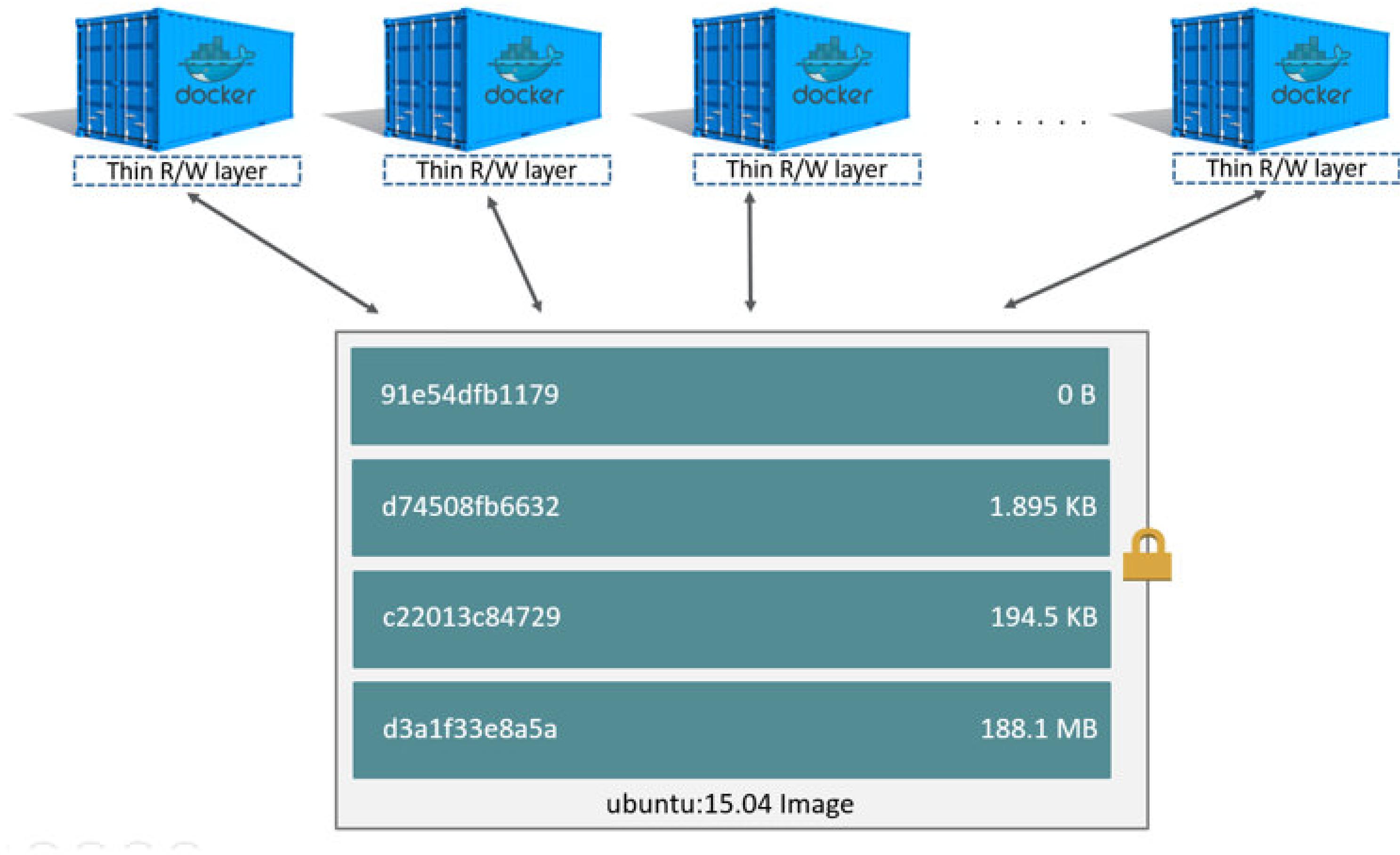
## **Relation avec le système d'exploitation hôte**

- Une image n'inclut que les bibliothèques et services du système d'exploitation mentionné
- Le conteneur utilise le noyau du système hôte

# Les autres couches

- Les couches correspondent aux différentes modifications qui sont faites pour construire l'image à partir de l'image de base.
  - Pour sauvegarder une nouvelle image, il suffit de sauvegarder les nouvelles couches qui ont été créées au dessus de l'image de base
  - Chaque couche capture les écritures faites par une commande exécutée lors de la création de l'image
  - Faible espace de stockage utilisé
- Dans un conteneur en cours d'exécution, il existe une couche supplémentaire accessible en écriture
  - Toutes les écritures vers le système de fichier faites à l'exécution du conteneur sont stockées dans cette couche.
    - Les autres couches, définies au sein de l'image utilisée pour instancier le conteneur, ne sont accessibles qu'en lecture.
  - Cette couche est supprimée à la suppression du conteneur.

# Partage de couches



Crédits: <https://docs.docker.com/storage/storagedriver/>

# **Optimisation des performances**

## **Partage des couches identiques entre conteneurs**

- Diminue l'espace de stockage utilisé par les conteneurs

## **Copie sur écriture (Copy-on-Write -- COW)**

- Un fichier accédé en lecture reste dans sa couche
- Copie dans la couche du dessus lors d'un accès en écriture
  - Mécanisme utilisé lors de la création de l'image et de l'exécution des conteneurs
  - Une couche ne contient vraiment que les données écrites/modifiées

# Plus d'informations sur les couches

## Affichage de l'ensemble des couches d'une image

```
$ docker history image_name
```

## Avantages liés aux mécanismes de couches

- Chaque couche est stockée une seule fois localement
  - Si certaines couches nécessaires pour une image à télécharger sont déjà présentes, pas besoin de les télécharger à nouveau.
  - Réduction de l'espace de stockage
- Optimisations à l'exécution
  - Démarrage rapide
    - Démarrer un conteneur nécessite simplement de créer la couche accessible en écriture
  - Faible utilisation de l'espace disque
    - Si plusieurs containers sont instanciés à partir de la même image, ils partagent les couches en lecture seule.

# **Identification des couches/images (voir ICI)**

Dans la version courante de Docker

- Chaque couche est identifiée par un digest qui est un hash du contenu de la couche
- Une image est définie par un ensemble de metadonnées (principalement la liste ordonnée des identifiants des couches constituant l'image)
  - L'identifiant de l'image est créé en calculant un hash de ces metadonnées

# **Les namespace d'images**

## **3 namespaces**

- les images officielles
  - ex: ubuntu, busybox
- Les images d'utilisateurs/organisations
  - ex: tropars/myapp
- Les images hébergées (en dehors de docker hub)
  - ex: registry.example.com:5000/my-private/image

# Les images officielles (espace de nommage *root*)

- Images sélectionnée par Docker Inc mais en général produites et maintenues par d'autres personnes
- Cela inclut:
  - Des petites images *couteau suisse* (ex: busybox)
  - Des images de distributions à utiliser comme images de base
  - Des composants prêts à être utilisés (ex: redis, mysql)

# **Les images perso (espace de nommage utilisateur)**

Pour les images des utilisateurs de Docker Hub: tropars/myapp

- tropars est le nom de l'utilisateur
- myapp est le nom de l'image

# **Les images self-hosted**

- Pour les images stockées sur d'autres registres que Docker Hub
  - Elles contiennent le hostname ou l'adresse IP, et optionnellement le numéro de port, du serveur.
- `registry.example.com:5000/my-private/image`

# Liste d'images stockées localement

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jekyll-github-page	latest	65736152d458	4 days ago	735MB
maven	3.8.1-jdk-11	4e8fadf07a8c	7 months ago	658MB
openjdk	11-jdk	a9c7e4ad6720	8 months ago	647MB
ubuntu	20.04	7e0aa2d69a15	8 months ago	72.7MB

# Chercher des images

Il n'est pas possible de lister toutes les images d'un dépôt mais on peut faire des recherches par mot clé.

\$ docker search spark				
NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
sequenceiq/spark	An easy way to try Spark	453		[OK]
gettyimages/spark	A debian:jessie based Spark container	131		[OK]
mesosphere/spark	DCOS Spark	112		
singularities/spark	Apache Spark	62		[OK]
bde2020/spark-master	Apache Spark master <b>for</b> a standalone cluster	49		[OK]
bde2020/spark-worker	Apache Spark worker <b>for</b> a standalone cluster	27		[OK]

- STARS indique la popularité de l'image
- OFFICIAL indique si l'image est officielle
- AUTOMATED veut dire que l'image est reconstruite automatiquement par Docker Hub (son Dockerfile est disponible)

# Télécharger des images

- docker pull: télécharge une image explicitement
- docker run: Commande servant à créer un conteneur à partir d'une image
  - Télécharge l'image si elle n'est pas présente localement

# Télécharger des images

```
$docker pull redis
Using default tag: latest
latest: Pulling from library/redis
a2abf6c4d29d: Pull complete
c7a4e4382001: Pull complete
4044b9ba67c9: Pull complete
c8388a79482f: Pull complete
413c8bb60be2: Pull complete
1abfd3011519: Pull complete
Digest: sha256:db485f2e245b5b3329fdc7eff4eb00f913e09d8feb9ca720788059fdc2ed8339
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```

Par défaut, si aucun tag n'est défini, l'image avec le tag `latest` est téléchargée

- Exemples de tags: `redis:6.2.6`, `redis:alpine`

# A propos des tags

## Pas besoin de tags quand

- On prototype/teste
- On veut la dernière version d'une image

## Tags à utiliser quand

- Quand on va en production
- Pour s'assurer que la même version va être utilisée partout
- Pour avoir de la reproductibilité

# **La création d'images**

# **Créer des images**

2 manières:

- Interactivement
- En utilisant un Dockerfile

# Avant de commencer: Premiers pas avec docker

L'outil à la ligne de commande pour exécuter des commandes Docker:

```
$ docker
```

```
docker run hello-world
```

- **docker**: Nous voulons exécuter une commande docker
- **run**: Commande pour créer et exécuter un conteneur docker
- **hello-world**: Nom de l'image à partir de laquelle est construit le conteneur

# Premiers pas

```
$ docker run hello-world
```

## Que va-t-il se passer?

- L'image à charger est `hello-world:latest`
- Vérification: est ce que l'image est présente localement?
- Sinon, télécharger l'image depuis Docker Hub
- Charger l'image dans le conteneur et exécuter la commande par défaut définie pour ce conteneur

# **Construire une image interactivement**

## **Objectifs**

- Créer une image à partir d'une image de base dans laquelle nous allons installer cowsay

## **Les étapes**

1. Créer un conteneur à partir de l'image de base
2. Installer le logiciel manuellement dans le conteneur et en faire une nouvelle image
3. Jouer avec:
  - docker commit
  - docker tag
  - docker diff

# Configuration du conteneur

## Démarrer un conteneur Ubuntu

```
$ docker run -it ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
ea362f368469: Pull complete
Digest: sha256:b5a61709a9a44284d88fb12e5c48db0409cfad5b69d4ff8224077c57302df9cf
Status: Downloaded newer image for ubuntu:latest
root@f461e2e7afff:/#
```

f461e2e7afff est l'id du conteneur créé

## Installer le programme dans le conteneur

```
root@f461e2e7afff:/# apt-get update
root@f461e2e7afff:/# apt-get install cowsay
```

# Inspection des changements

Quitter la session interactive:

```
root@f461e2e7afff:/# exit
```

Inspecter les changements:

```
$ docker diff f461e2e7afff
C /var
C /var/log
C /var/log/apt
C /var/log/apt/history.log
A /var/log/apt/term.log
C /var/log/apt/eipp.log.xz
C /var/log/dpkg.log
...
...
```

[Doc de docker diff](#)

- C: fichier ou répertoire modifié
- A: fichier ou répertoire ajouté
- Rappel: ensemble des changements dans la couche accessible en écriture du conteneur

# A propos de ces changements

## Rappels

- Seule la dernière couche créée à l'exécution du conteneur est accessible en écriture
- Stratégie de CoW pour efficacité
  - Pas de copies au démarrage de l'image

## Avantages

- Tous les changements sont capturés dans une couche
  - Peuvent être affichés avec `git diff`
- Sécurité
  - Des conteneurs peuvent aussi être démarrés en lecture seule
    - Des écritures peuvent quand même être faites mais dans des volumes

# Sauvegarder les changements dans une nouvelle image

```
$ docker commit <yourContainerId>  
<newImageId>
```

- Sauvegarde les changements dans une nouvelle couche et sauvegarde l'image

## Exécution de la nouvelle image

```
$ docker run -it <newImageId>  
root@7267696dc8c6:/# /usr/games/cowsay bonjour  
... ca marche
```

# Tagger une image

On peut tagger une image pour lui associer un nom (plus facile à manipuler qu'un identifiant)

```
$ docker tag <newImageId> cowsay
```

On peut aussi tagger lors de la création de l'image

```
$ docker commit <containerId> cowsay
```

L'image peut maintenant être exécutée en utilisant ce nom:

```
$ docker run -it cowsay
```

**Il faut maintenant automatiser le processus!**

# **Les Dockerfile**

- Un Dockerfile est une recette décrivant comment construire une image
  - Contient une suite d'instructions
- La commande `docker build` permet de créer une image à partir d'un Dockerfile

# Éléments de syntaxe

- **FROM:** Définit l'image à partir de laquelle la nouvelle image est créée
- **LABEL:** Associe des meta-données à la nouvelle image (par exemple, l'auteur de l'image)
- **RUN:** Définie une commande exécutée dans la couche au dessus de l'image courante lors de la construction de l'image
- **CMD:** Définie la commande exécutée au démarrage du conteneur
- **EXPOSE:** Informe docker que le conteneur va écouter sur le port réseau défini
- **COPY:** Copier un fichier/répertoire depuis le contexte de construction de l'image vers la nouvelle couche
  - La destination peut être un chemin absolu ou un chemin relatif depuis **WORKDIR**

*Les commandes de Dockerfile ne sont pas sensibles à la casse. On les note en majuscule par convention (facilite la lecture)*

# Notre premier Dockerfile

1. La création d'un Dockerfile **doit se faire dans un nouveau répertoire vide**

```
mkdir my_image
```

2. Créer le Dockerfile

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install cowsay
```

- Les commandes RUN doivent être non-interactives
  - L'option -y de apt évite qu'il demande si on est sur de vouloir installer

# Construire notre image

```
$ docker build -t cowsay .
```

- `-t` permet de tagger avec un nom l'image qui va être créée
- `.` indique le contexte de construction de l'image (où se trouve le Dockerfile)

# Que se passe-t-il?

```
$ docker build -t cowsay .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
--> d13c942271d6
Step 2/3 : RUN apt-get update
--> Running in 80f5510281d9
Removing intermediate container 80f5510281d9
--> cb1643c4393c
Step 3/3 : RUN apt-get -y install cowsay
--> Running in 01834650511b
Removing intermediate container 01834650511b
--> 9ca55c5ccc54
Successfully built 9ca55c5ccc54
Successfully tagged cowsay:latest
```

# Que se passe-t-il?

- Le build context est envoyé vers le démon docker (contenu du répertoire .)
- A chaque étape:
  - Un conteneur est créé pour exécuter l'étape (Running in ...)
  - Les modifications sont committées dans une nouvelle image (- - -> ...)
  - Le conteneur est supprimé
  - La nouvelle image est utilisée pour la prochaine étape

# Construire une image

## **docker build**

Crée une image à partir d'un fichier Dockerfile

```
docker build -t docker-whale .
```

- Crée l'image docker-whale avec le contexte correspondant au répertoire courant (le `.` est nécessaire)
- Par défaut, le Dockerfile est cherché à la racine du contexte
  - Option -f pour changer le chemin
  - Dans tous les cas le fichier doit se trouver dans le contexte

Voir <https://docs.docker.com/engine/reference/commandline/build/>

# Construire une image

## Build à partir d'une URL:

```
docker build github.com/creack/docker-firefox
```

- L'URL peut pointer vers un dépôt git, une archive, un fichier texte
- Cas d'un dépôt git:
  - Clone le dépôt et utilise le clone comme context

```
docker build https://github.com/docker/rootfs.git#container:docker
```

- Utilise le répertoire docker de la branche container comme contexte
- Voir la doc pour plus de détails

# Manipuler les images

## Quelques commandes

```
docker image COMMAND
```

```
docker image ls
```

- liste des images existantes localement
  - Option -a permet d'afficher toutes les images locales
  - docker image ls ubuntu: liste des images nommées ubuntu
  - Les images <none>:<none> sont des images intermédiaires nécessaires à la constructions d'images locales

```
docker image rm
```

- Supprime une image

# Dangling image ("image qui pendouille")

- **Définition:** Une image intermédiaire qui n'est plus référencée par aucune image ni aucun conteneur
- Supprimer ces images:

```
docker image prune
```

# A propos du contexte

Lors de la construction d'une nouvelle image, un contexte de construction de l'image est défini:

- Souvent le répertoire dans lequel on exécute la commande pour créer l'image
- Tous les fichiers appartenant au contexte sont envoyés au daemon docker
  - Seuls les fichiers appartenant au contexte peuvent être copiés vers la nouvelle image

## Bonne pratique

Créer un répertoire contenant le Dockerfile et seulement les fichiers dont vous avez besoin dans le contexte

- Ne pas utiliser "/" comme contexte !!
- Un fichier .dockerignore peut être utilisé
  - Défini des règles pour ignorer certains fichiers du contexte

# Buildkit

- Un nouveau moteur pour construire des images docker
  - Installé par défaut avec Docker Desktop
  - Doit être activé sous Linux quand on utilise Docker Engine

## Avantages

- Ne transfère que les parties du contexte dont on a besoin
  - Et détecte les fichiers non modifiés entre deux builds
- Exécute les étapes de construction de l'image en parallèle quand c'est possible
- Cache optimisé
- Fonctionne en mode *rootless*

# Syntaxe shell vs exec

Les commandes telles que RUN ou CMD ont 2 syntaxes possibles

## La syntaxe shell

```
RUN apt-get install cowsay
```

- La commande est exécutée dans un shell
  - /bin/sh -c est utilisé par défaut

## La syntaxe exec

```
RUN ["apt-get", "install", "cowsay"]
```

- Commande parsée en JSON et exécutée directement
  - Nécessite " pour chaque chaîne de caractères

# **Syntaxe shell vs exec**

## **La syntaxe shell**

- Plus facile à lire
- Interprète les expressions shell (ex: \$HOME)

## **La syntaxe exec**

- N'essaye pas d'interpréter les arguments
- Ne nécessite pas que /bin/sh soit présent dans l'image

# CMD et ENTRYPOINT

## A propos de CMD

- Défini la commande par défaut à exécuter dans le conteneur quand aucune commande n'est fournie à l'exécution de docker run
- Peut être insérée à n'importe quel endroit dans le Dockerfile mais seul la dernière est conservée

## A propos de ENTRYPOINT

- Permet de définir une commande toujours exécutée par le conteneur
- Il est recommandé d'utiliser la syntaxe exec
  - Avec la syntaxe shell, le passage de paramètres à /bin/sh risque d'être incorrect

# **CMD et ENTRYPOINT : cas d'utilisation**

- CMD: Cas d'une image qui inclut plusieurs exécutables (par ex: busybox)
- ENTRYPOINT: Cas d'un binaire conteneurisé (l'image n'inclut qu'un exécutable)

# CMD et ENTRYPOINT

## Les règles principales

- Chaque Dockerfile doit spécifier au moins CMD ou ENTRYPOINT
- Quand CMD ou ENTRYPOINT sont utilisés en même temps
  - ENTRYPOINT définit la commande de base
  - CMD définit les paramètres par défaut
  - Ils doivent tous les 2 utiliser la syntaxe exec
  - A l'exécution seule la partie CMD sera écrasée si des paramètres sont passés.

Voir <https://docs.docker.com/engine/reference/builder/#entrypoint>

# CMD et ENTRYPOINT : illustration

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "-y", "install", "cowsay"]
ENTRYPOINT ["/usr/games/cowsay", "-e", "%%"]
CMD ["hello world"]
```

```
docker build -t mycowsay .
```

# CMD et ENTRYPOINT : illustration

```
$ docker run mycowsay
< hello world >
-----
\   ^__^
 \  (%%)\_____
 (__)\       )\/\
    ||----w |
    ||     ||
# Commande exécutée: /usr/games/cowsay -e %% hello world
```

```
$ docker run mycowsay salut
-----
< salut >
-----
\   ^__^
 \  (%%)\_____
 (__)\       )\/\
    ||----w |
    ||     ||
# Commande exécutée: /usr/games/cowsay -e %% salut
```

# Le cache d'images

Lors de l'exécution d'un Dockerfile, à chaque commande, Docker évalue si une image **équivalente** existe déjà. Equivalente veut dire:

1. Construite à partir la même image
  2. Exécute la *même* commande
- 
- Si **oui**, l'image existante est réutilisée pour la commande
    - *Si on build 2 fois de rang le même Dockerfile sans rien changer, la deuxième exécution est instantanée*
  - Si **non**, la commande est exécutée pour créer une nouvelle image
    - Et toutes les commandes suivantes dans le Dockerfile sont réexécutées

Plus d'infos: [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/#leverage-build-cache](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#leverage-build-cache)

# Commandes identiques: les règles

- Pour la commande RUN, les chaînes de caractères sont simplement comparées
  - Les commandes suivantes ne sont pas équivalentes:
    - RUN apt-get install cowsay curl
    - RUN apt-get install curl cowsay
  - RUN apt-get update n'est pas ré-exécuté quand les dépôts sont mis à jour
- Pour les commandes ADD et COPY, des checksums des fichiers à copier sont calculés pour déterminer si ils ont été modifiés ou non.

# Quelques conseils liés à l'utilisation du cache

## Ordre des commandes

Penser à l'ordre des commandes pour ne pas ré-exécuter des étapes longues inutilement

- Mettre COPY vers la fin si possible
- Mettre les étapes d'installation couteuses au début

## Désactiver l'utilisation du cache

Pour reconstuire une image en ignorant le cache:

```
docker build --no-cache ...
```

# Voir l'historique d'une image

```
$ docker history mycowsay
```

IMAGE	CREATED	CREATED BY	SIZE	COMM
5439d59a9167	31 minutes ago	/bin/sh -c #(nop)  CMD ["hello world"]	0B	
62cf4b34d556	31 minutes ago	/bin/sh -c #(nop) ENTRYPOINT ["/usr/games/c..."]	0B	
68dd4625b571	37 minutes ago	apt-get -y install cowsay	45.4MB	
cb1643c4393c	8 hours ago	/bin/sh -c apt-get update	32.6MB	
d13c942271d6	38 hours ago	/bin/sh -c #(nop)  CMD ["bash"]	0B	
<missing>	38 hours ago	/bin/sh -c #(nop) ADD file:122ad323412c2e70b...	72.8MB	

- Montre l'ensemble des couches composants une image
  - Avec des infos sur chaque couches
- Chaque couche correspond à une commande dans le Dockerfile

# Exemple d'utilisation de COPY

Dockerfile:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
CMD /hello
```

- `hello.c` doit appartenir au contexte
- Le répertoire de travail par défaut du conteneur est "/"

# Optimiser les images

## Nombre de couches

Chaque commande du Dockerfile crée une nouvelle couche:

- Limiter le nombre de couches peut améliorer les performances
- Commentaire surtout valable pour les anciennes versions de Docker

## Build multi-stages

Permet de réduire la taille des images.

- Permet par exemple de sélectionner seulement certaines des modifications générées lors d'une étape de construction pour les intégrer dans l'image finale.
- <https://docs.docker.com/develop/develop-images/multistage-build/>

# Multi-stage builds

Dans notre dernier exemple, l'image finale contient:

1. L'exécutable hello
2. Le fichier source
3. Le compilateur installé

**Seul le premier est strictement nécessaire**

# Multi-stage builds

## Principes

- Plusieurs instructions FROM peuvent être utilisées dans un Dockerfile
  - Chacune définit le début d'une nouvelle *stage*
  - Chacune peut utiliser une image de base différente
- Seule la dernière *stage* est conservée dans l'image finale
- On peut COPY des fichiers d'une étape précédente dans la nouvelle étape

```
COPY --from=0 /file/from/first/stage /location/in/current/stage
```

- Chaque étape est numérotée, en partant de 0
  - FROM ... AS ... permet d'associer un nom à une étape

# Exemple de Multi-stage build

Dockerfile permettant de créer une image plus petite que précédemment:

```
FROM ubuntu AS compiler
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
FROM ubuntu
COPY --from=compiler /hello /hello
CMD /hello
```

# **Publier des images sur Docker Hub**

# **Tags et adresse des registres**

Publier une image nécessite de la tagger

- Tagger ne renome pas l'image, ça ajoute juste un pointeur vers l'image
- Exemple: `registry.example.net:5000/image`
  - Le tag à ajouter doit inclure l'adresse du registre où on veut publier l'image

## **Cas des images sur Docker Hub**

- `tropars/clock`
  - est en fait: `index.docker.io/tropars/clock`
- `ubuntu`
  - est en fait: `library/ubuntu` -> `index.docker.io/library/ubuntu`

# Publier mon image sur Docker Hub

Tagger l'image à publier:

```
docker tag mycowsay mydockeraccount/cowsay:latest
```

- mydockeraccount: Mon login sur Docker Hub

Se connecter à docker hub:

```
docker login
```

Publier l'image:

```
docker push mydockeraccount/cowsay
```

L'image peut maintenant être récupérée par d'autres:

```
docker pull mydockeraccount/cosway
```

# **Manipuler des conteneurs**

# Démarrer un conteneur

```
docker run --name mytest mycowsay
```

- Démarrer un conteneur nommé `mytest`
  - Si on ne spécifie pas l'option `--name` un nom aléatoire est assigné au conteneur
  - Un numéro d'identifiant est aussi assigné au conteneur
- Exécute la commande spécifiée par `ENTRYPOINT/CMD` dans le Dockerfile
- Plusieurs options (par exemple pour limiter les ressources utilisées)
- `-it`: Crée une session interactive avec le conteneur et ouvre un pseudo-terminal

```
docker run --name test -it debian
```

# Manipuler des conteneurs

## Commande parent

```
docker container COMMAND
```

## Quelques commandes

- **ls**: Montre les conteneurs en cours d'exécution
- **start**: Démarrer un conteneur arrêté
- **stop/restart**: Arrête/redémarre un conteneur en cours d'exécution
- **rm**: Supprime un conteneur arrêté
- **prune**: Supprime tous les conteneurs arrêtés
- **logs**: Récupère les logs d'un conteneur
- **stats**: Obtenir les informations sur la consommation de ressources d'un conteneur
- **top**: affiche la liste des processus du conteneur

Voir <https://docs.docker.com/engine/reference/run/>

# Débugger dans un conteneur

Comment observer ce qu'il se passe dans un conteneur:

- Se connecter à un conteneur en cours d'exécution avec `docker exec`
- Permet d'exécuter une commande ou d'ouvrir un shell

```
$ docker exec -it f136fa721110 bash  
root@f136fa721110:/#
```

# **La gestion du réseau**

# Remarques introductives

Les conteneurs sont souvent utilisés pour déployer des applications communiquant sur le réseau:

- Des applis web
- Des applis composées de plusieurs services
- Des bases de données
- Etc.

## Plusieurs questions:

- Comment communiquer avec un appli conteneurisé depuis l'extérieur?
- Comment communiquer entre conteneurs?
  - Sur un même noeud?
  - Sur des noeuds différents?

# Exemple avec un server web

## Lancer un serveur web en arrière plan

```
$ docker run -d -P nginx
```

- -d lance le conteneur en arrière plan
- -P rend accessible tous les ports exposés par le conteneur au niveau de la machine hôte

## Observation du conteneur créé

```
docker ps
CONTAINER ID        IMAGE               COMMAND                  PORTS
f136fa721110      nginx              "/docke...           0.0.0.0:49153->80/tcp, :::49153->80/tcp
                                         xe
```

- Le serveur utilise le port 80 du conteneur
- Ce port a été associé au port 49153 de la machine hôte
  - Envoyer une requête sur le port 49153 de la machine permet d'atteindre le serveur web dans le conteneur
  - curl localhost:49153

# Quelques commentaires

## Autre manière d'obtenir le port de la machine hôte

```
$ docker port <containerID> 80
```

## Choisir le numéro de ports sur la machine hôte

```
$ docker run -d -p 8000:80 nginx
```

- Port 8000 de la machine hôte associé au port 80 du conteneur
- -p port-on-host:port-on-container

# Adresse IP d'un conteneur

Chaque conteneur démarré se voit associé une adresse IP

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID>
172.17.0.2
```

Depuis l'hôte, je peux *pinger* cette adresse:

```
ping 172.17.0.2
```

# Les réseaux docker

Un réseau docker est un switch virtuel

- Crée un sous réseau avec une plage d'adresse IP privée (172.17.0.0/16 par défaut)
  - Une adresse IP est associée à chaque conteneur
  - Chaque conteneur a son espace de nommage privé pour ces numéros de ports
- Un service DNS se charge de la résolution de noms
- Des mécanismes de NAT (Network Address Translation) permettent de router le traffic entrant sur la machine vers le bon conteneur
- Un réseau docker est implémenté par un *driver*

## Possibilité de créer plusieurs réseaux docker sur une machine

- Des réseaux différents peuvent permettre d'isoler des conteneurs s'exécutant sur la même machine
- Les conteneurs peuvent être connectés à plusieurs réseaux

# Les drivers réseaux

Les drivers réseaux sont chargés de fournir les fonctionnalités réseaux principales.

- Plusieurs drivers existent
- L'utilisateur choisit celui qu'il veut utiliser

Les 4 drivers principaux sont:

- bridge
- host
- overlay
- none

## Remarque

Quand des conteneurs sont déployés par un orchestrateur (par ex. Kubernetes), c'est l'orchestrateur qui est en charge de la gestion du réseau

Voir <https://docs.docker.com/network/>

# Les principaux drivers

## Bridge

- Driver par défaut
- A utiliser pour déployer un/des conteneurs sur une seule machine
- Détailé dans les slides suivant

## Host

- Utilise le réseau de la machine hôte directement
- Moins simple à utiliser mais plus performant que bridge

## Overlay

- Sert pour interconnecter les conteneurs s'exécutant sur différentes machines
- Docker Swarm

## None

- Sert à isoler le conteneur du réseau

# Observer les réseaux docker

## Observer les réseaux présents

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
2d9005266c5f	bridge	bridge	local
93a4dd2904a4	host	host	local
9507122f6adf	none	null	local

Le réseau par défaut est simplement appelé bridge

- Par défaut les conteneurs sont connectés à ce réseau

# Les bridges

Un bridge est un système qui permet aux conteneurs qui y sont connectés de communiquer entre eux

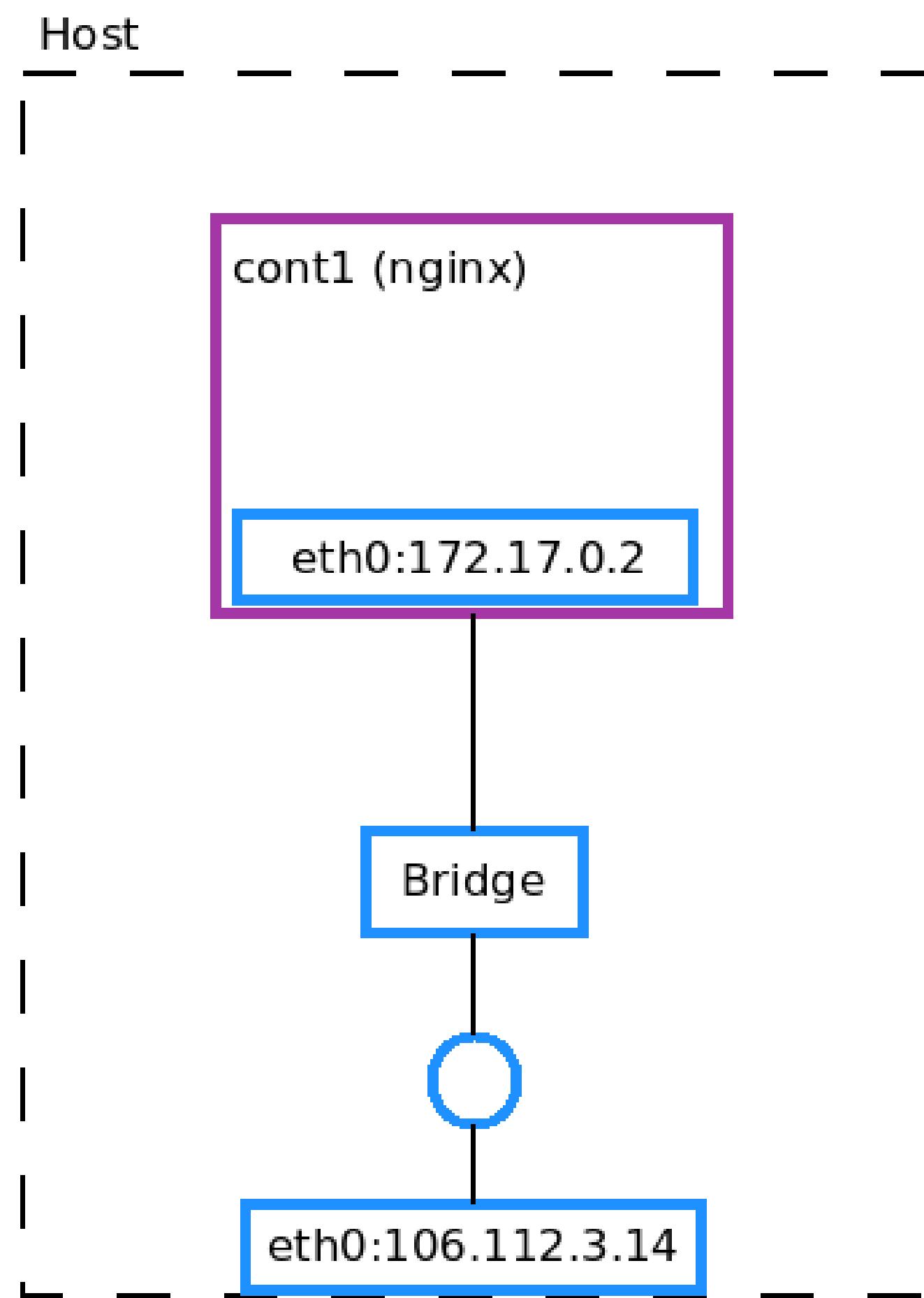
- Interconnecte des conteneurs s'exécutant sur la même machine
- Assure que chaque conteneur a un espace de nommage privé
  - !! Ce n'est pas le cas avec le driver host

## Bridge créé par l'utilisateur

- Le bridge par défaut à certaines limitations
  - Pas de résolution de noms
  - Potentiellement des conteneurs qui n'ont rien à voir sur le même réseau
- **L'utilisateur peut créer ses propres bridges** (recommandé)
  - Et décider quels conteneurs sont connectés à quels bridges
  - La résolution de noms est assurée pour les bridges utilisateurs

# Illustration

```
$ docker run -d --name cont1 nginx
```



# Bridge utilisateur

## Créer un bridge

```
$ docker network create -d bridge my_bridge
```

## Attacher un conteneur à un bridge

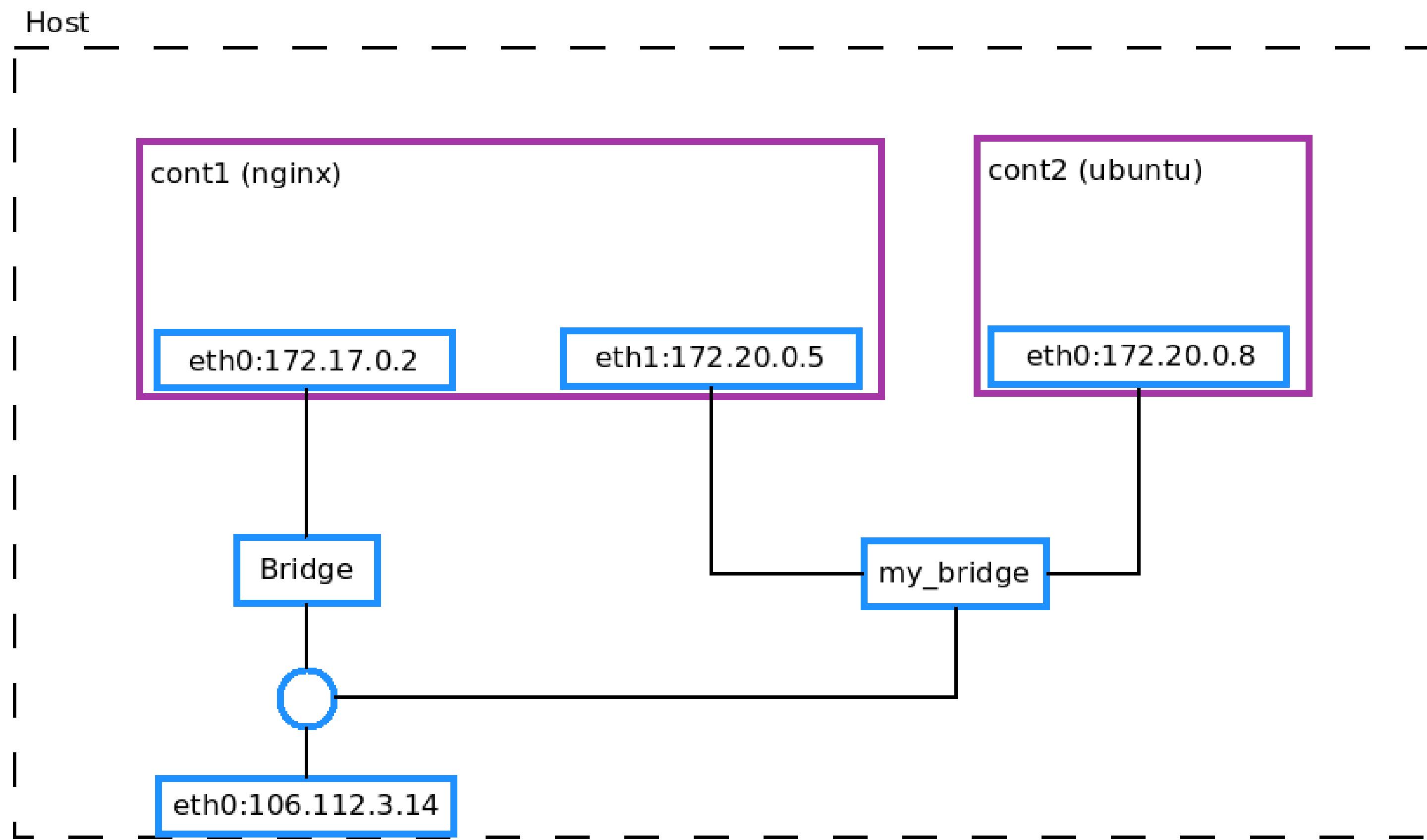
Lors de la création d'un conteneur:

```
$ docker run -d --network my_bridge --name cont2 ubuntu
```

Pour un conteneur existant:

```
$ docker network connect my_bridge web
```

# Illustration



- cont1 peut communiquer avec cont2 au travers du bridge my\_bridge
- Si un conteneur n'est associé qu'au bridge par défaut, il ne pourra pas communiquer avec cont2

# Découverte de services

- Dans la configuration précédente, cont1 peut contacter cont2 en utilisant son adresse IP (172.20.0.8)
- Cependant ce n'est pas très pratique
  - Comment connaître l'IP à la création de l'image?

## Service de découverte de service

- Le service DNS associé au bridge utilisateur permet à cont1 d'utiliser le nom du conteneur pour contacter cont2
- Remarques:
  - Le même nom de conteneur peut être utilisé pour 2 conteneurs qui sont sur des réseaux différents
  - Ce n'est pas possible pour 2 conteneurs sur le même réseau

# A propos de la gestion des numéros de ports

## Problème courant

- Plusieurs conteneurs à exécuter potentiellement sur la même machine utilisent les mêmes numéros de port
  - Port 80 pour http
- Modifier l'image pour que l'application conteneurisée écoute sur un autre port n'est pas forcément simple

## Solution

- Le mécanisme de NAT associé au bridge nous permet de résoudre le problème
- Associer chaque conteneur avec un port différent de la machine hôte
  - Conteneur 1 écoute sur le port 80 -> associé au 8001
  - Conteneur 2 écoute sur le port 80 -> associé au 8002

# **Gérer les données dans Docker**

# **Ecriture dans le conteneur**

On peut stocker les données manipulées par une application directement dans le conteneur.

## **Les inconvénients:**

- Inclure des données en lecture seule dans les images augmente leur taille et leur font perdre leur généricité.
- Pour les données écrites pendant l'exécution du conteneur:
  - Elles disparaissent avec le conteneur
  - Elles sont difficilement accessibles depuis l'extérieur du conteneur
  - C'est peu efficace en terme de performance (coût de la gestion des couches)

# Les alternatives pour le stockage des données

## Les volumes:

Stockage géré par Docker dans le système de fichier hôte

- Partager des données entre plusieurs conteneurs
- Stocker des données sur un support distant (ex: dans un cloud) en utilisant un driver

## Les bind mounts

Stockage à n'importe quel endroit du système de fichier hôte

- Partager des fichiers de configuration avec le système hôte
- Utiliser du code source présent sur la machine hôte

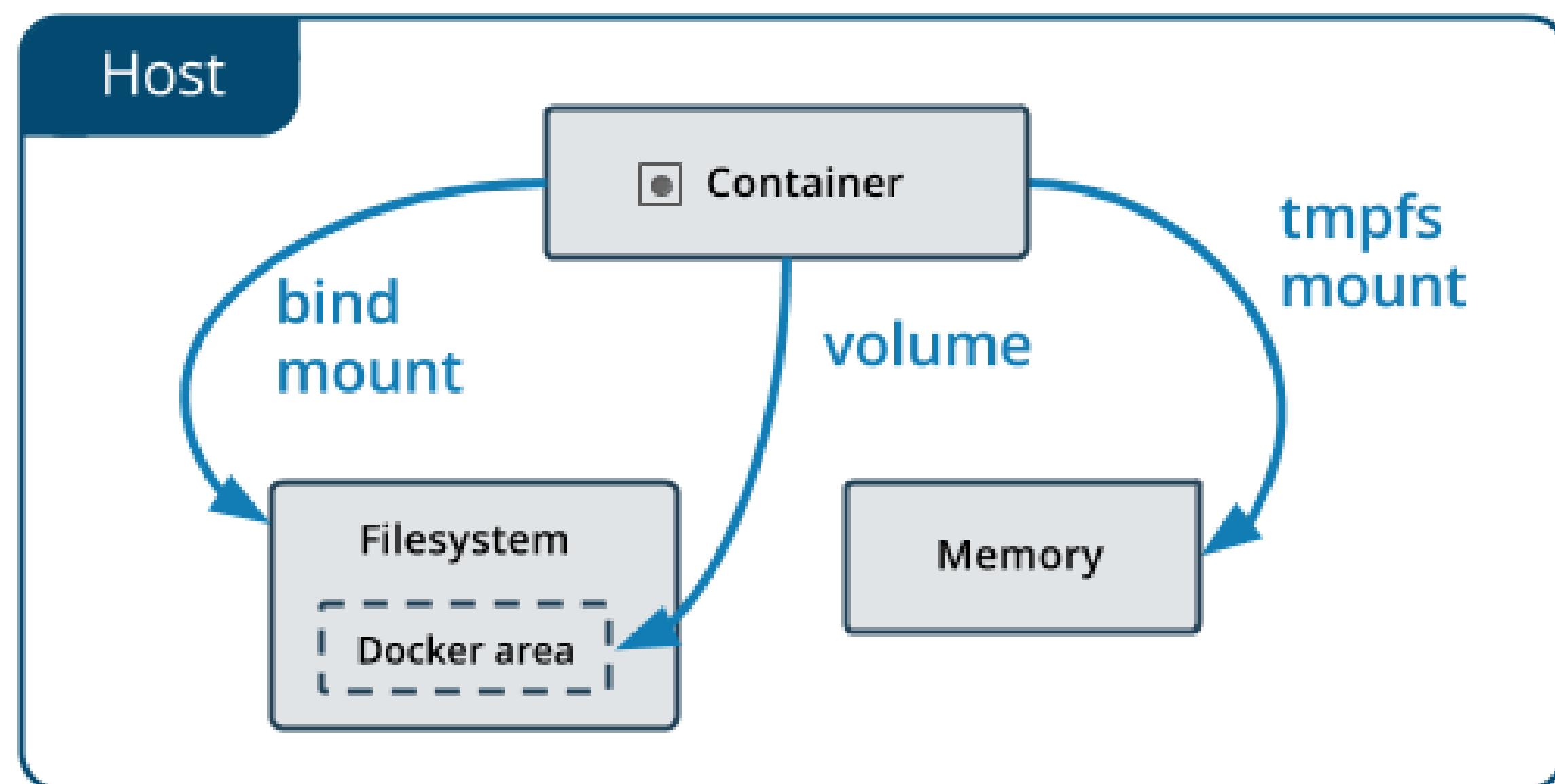
## Les montages tmpfs:

Stockage en mémoire

- Plus efficace qu'un volume lorsque les données n'ont pas besoin de persistance

Voir <https://docs.docker.com/storage/>

# Les solutions pour le stockage des données



# Les volumes

Création d'un volume:

```
$ docker volume create my-vol
```

## Utilisation d'un volume

- Option `--mount` pour la commande `docker run`
  - Utilisation du volume `my-vol` avec comme répertoire destination au sein du conteneur `/app`:

```
docker run -d --name devtest --mount source=my-vol,target=/app debian
```

## Commentaires:

- Le volume n'a pas besoin d'être créé à l'avance
- Si le répertoire destination contient déjà des données, elles sont copiées dans le volume
- L'option `readonly` peut être utilisée pour empêcher les données d'un volume d'être modifiées

Voir <https://docs.docker.com/storage/volumes/>

# Les bind mounts

- Option `--mount` pour la commande `docker run` avec l'option `type=bind`

```
$ docker run -d -it --name devtest \
  --mount type=bind,source="$(pwd)"/code,target=/app debian:latext
```

## Commentaires

- Si le répertoire `/app` existe déjà au sein du conteneur, son contenu est remplacé celui du répertoire de l'hôte qui est *bindé*.

# **Docker Compose**

# Motivations

Configurer correctement une application multi-services peut être complexe

- Démarrage de l'ensemble des conteneurs avec les bonnes options
- Configuration du réseau
- Configuration des volumes
- Etc.

## Solution

- Une approche *Infrastructure-as-code*
- Docker Compose

# Qu'est ce que Docker Compose?

- Un outil complémentaire du *Docker Engine*
  - Présenté dans les versions récentes comme un plugin de *Docker Engine*
  - Ancienne syntaxe: `docker-compose` command
  - Nouvelle syntaxe: `docker compose` command
- Permet de décrire une application construite à base de conteneurs dans un fichier YAML
  - Déploiement sur une seule machine

## Scénario visé

- Checkout du code sur un dépôt
- Executer `docker compose up`
- ... Et c'est tout!
  - L'application est configurée et démarrée

# Principes de Docker Compose

- L'utilisateur décrit son application (multi) conteneurs dans un fichier YAML appelé `docker-compose.yml`
- Executer `docker-compose up` pour démarrer l'application
  - Compose télécharge automatique les images, les build (si nécessaire), et démarre les conteneurs
  - Compose peut configurer des volumes, le réseau, et toutes autres options liées à Docker
- Compose aggrège les sorties des différents conteneurs (si applicable)

# Un exemple

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
  volumes:
    - .:/code
    - logvolume01:/var/log
  depends_on:
    - redis
  redis:
    image: redis
  volumes:
    logvolume01:
```

Voir: <https://docs.docker.com/compose/#compose-documentation>

# Sur l'exemple précédent

- 2 services: `web` et `redis`
  - Par défaut, Compose crée un réseau bridge indépendant du réseau par défaut
  - La découverte de service fonctionne sur ce réseau
    - `web` peut contacter `redis` en utilisant son nom
- Un conteneur pour chaque service est créé
  - Pour `web`, une image est d'abord re-créé à partir du `Dockerfile` présent dans le répertoire courant
  - Pour `redis`, l'image `redis` est récupérée depuis Docker Hub
- Configuration du réseau
  - Le port 5000 de la machine hôte est associé au port 5000 du conteneur `web`

# Sur l'exemple précédent

- Controle de l'ordre de démarrage
  - Compose démarre les conteneurs dans un ordre correspondant aux dépendances introduites dans la composition
    - `depends_on` permet de définir une dépendance explicite
    - D'autres balises induisent des dépendances implicites (`links`, `volumes_from`, etc.)
    - **Attention:** Compose attend que les conteneurs aient démarrés, mais pas que les services au sein des conteneurs soient prêt à fonctionner
- Les volumes
  - La version présentée utilise l'[ancienne syntaxe](#)
  - Déclaration d'un volume nommé: `logvolume01`
    - La déclaration est nécessaire pour partager un volume entre services
    - Le volume est ici utilisé par le conteneur web
  - Le conteneur web utilise aussi un bind mount

# Les volumes (nouvelle syntaxe)

```
version: "3.9"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
      - type: bind
        source: ./static
        target: /opt/app/static
volumes:
  mydata:
```

# A propos de build et image

Pour chaque service d'un composition, `build` ou `image` doivent être définis

- Si `build` est défini, il indique le chemin vers le Dockerfile à utiliser pour construire l'image
- Si `image` est défini, il indique le nom de l'image à utiliser (présente dans le cache ou téléchargée depuis un registre)
- Si les deux sont spécifiés
  - Le Dockerfile spécifié par `build` est utilisé
  - `image` définit le nom de l'image créée

# Les commandes de Compose

## Quelques commandes

- docker-compose up: démarre la composition
  - Option -d pour démarrer les conteneurs en arrière plan
- docker-compose ps: liste les conteneurs démarrés
- docker-compose kill: arrête les conteneurs
- docker-compose rm: supprime les conteneurs arrêtés
- docker-compose down: supprime tout ce qui a été créé par docker-compose up
  - Arrête les conteneurs
  - Supprime les conteneurs, images, volumes, réseaux

Voir <https://docs.docker.com/compose/reference/>

# Gestion des volumes et mise à jour de services

## Mise à jour de services

Si la composition est déjà démarrée et on appelle docker-compose up:

- Si l'image associée à un service a changée, le service est recréé automatiquement en utilisant la nouvelle image

## Gestion des volumes

Lors du redémarrage d'une composition, si un volume d'une exécution précédente existe toujours, il est ré-utilisé sans écraser les données:

- Permet de conserver l'état pour les services *stateful*

# C'est la fin

*A vous de jouer!*