Formation Docker

Brahim HAMDI

brahim.hamdi.consult@gmail.com

Octobre 2024

Présentation du formateur

Brahim HAMDI

- Consultant/Formateur DevOps & Cloud
- Expert DevOps, Cloud, CyberOps et Linux

Plan de la formation

- Comprendre les containers
- Docker
- Ecosystème Docker
- Gestion des conteneurs
- Les images Docker
- Création d’images
- Les volumes
- Les réseaux Docker
- Docker compose
- Docker swarm

Objectifs pédagogiques

- Manipuler l'interface en ligne de commande de Docker pour créer des conteneurs
- Maîtriser la création des images docker
- Mettre en œuvre et déployer des applications dans des conteneurs
- Administrer des conteneurs

Public concerné

- Administrateurs systèmes
- Développeurs
- Architectes
- Experts DevOps

Connaissance requise

- Commandes de base Linux

Références bibliographiques

- https://docs.docker.com/

Comprendre les containers

- Machines virtuelles : avantages et inconvénientsDifférences entre VM et conteneur
- Le conteneur et la virtualisation
- Différences entre VM et conteneur
- Conteneur Linux
- Namespaces
- Control Groups
- Copy-On-Write

Machines virtuelles : avantages et inconvénients

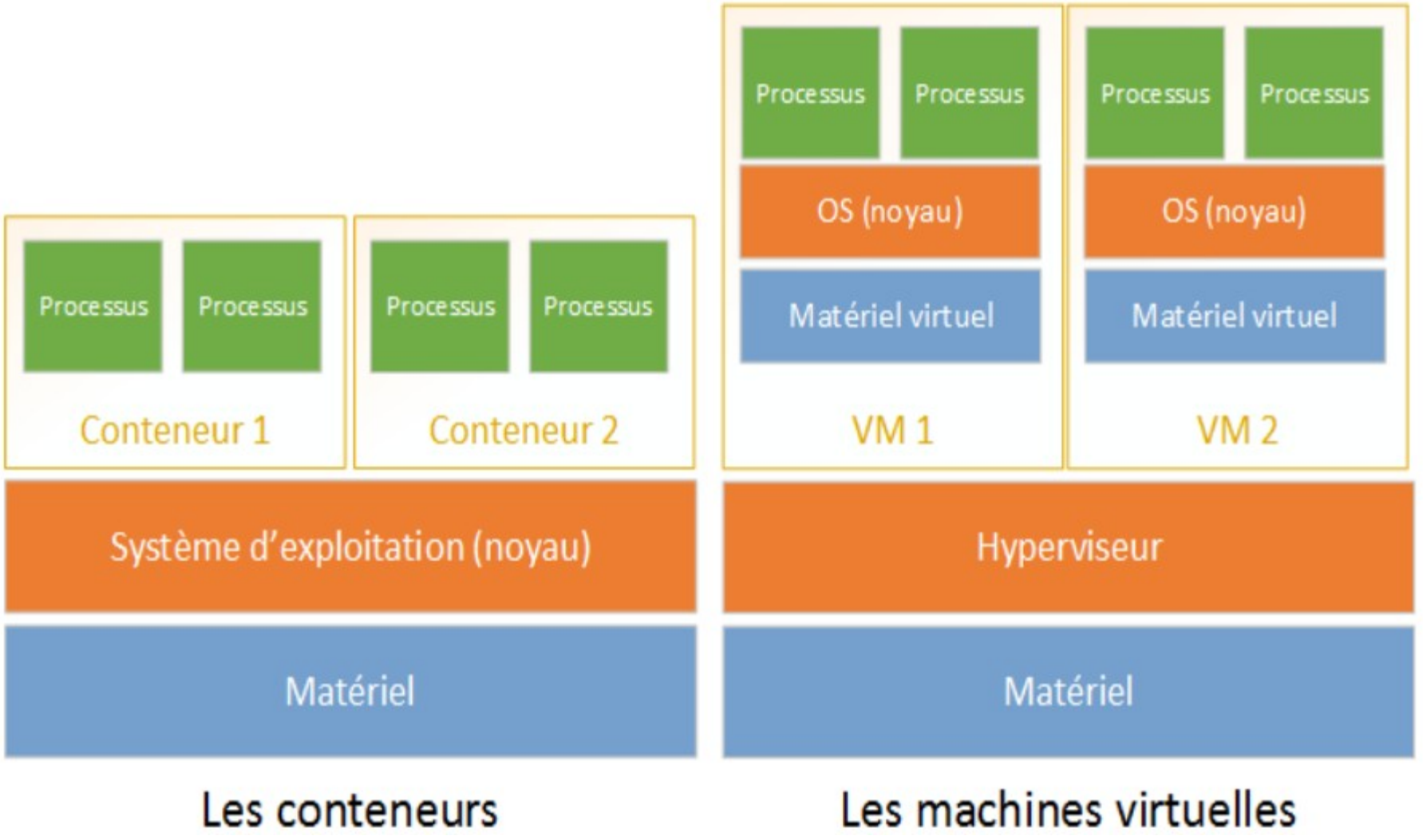
- Avantages :
  - Emulation bas niveau
  - Sécurité/compartimentation forte hôte/VMs et VMs/VMs
- Inconvénients
  - Usage disque important
  - Impact sur les performances

Le conteneur et la virtualisation

- Les mêmes idées que la virtualisation, mais sans virtualisation
- Isolation et automatisation
- Peu d’overhead par rapport à une VM !
- Un super chroot

- Certains parlent de virtualisation ”niveau OS” ou ”légère”, isolation applicative

Différences entre VM et conteneur



Conteneur Linux

- Processus isolé par des mécanismes noyaux.
- 3 éléments fondamentaux:
  - Namespaces
  - Cgroups
  - Copy-On-Write

Les namespaces

- Apparue dans le noyau 2.4.19 en 2002, réellement utiles en 2013 dans le noyau 3.8
- Limite ce qu'un processus peut voir du système:
  - un processus ne peut utiliser/impacter que ce qu'il peut voir
- Types: pid, net, mnt, uts, ipc, user
- Chaque processus est dans un namespace de chaque type

Control Groups

- Fonctionnalité du noyau, apparue en 2008 (noyau 2.6.24)
- Des fonctionnalités du noyau
- Contrôle précis de l'allocation des ressources pour un/groupe de processus :
  - Allocation
  - Monitoring
  - limite
- Types :
  - cpu, memory, network, block io, device

Copy-on-Write

- CoW : Donne un pointeur vers la même ressource plutôt que donner une copie.
- Réduit l’usage disque et le temps de création
- Plusieurs options disponibles
  - device mapper (niveau fichier)
  - btrfs, zfs (niveau fs)
  - aufs, overlay (niveau fichiers)

Docker

- Banaliser l’utilisation des conteneurs
- Docker Engine
- Exécuter des conteneurs partout
- Distribution efficace des conteneurs
- Historique de docker
- Qu’est ce que docker?
- Que contient docker?
- Les avantages et inconvénients de docker
- Sans et avec docker

Banaliser l’utilisation des conteneurs

- Une plate-forme ouverte pour créer, déployer et exécuter des applications réparties, empaquetées, de manière isolée et portable.

Docker Engine

- Développé par dotCloud en mars 2013
- Développé en langage Go de Google
- Fonctionne en mode démon
- Utilisait la lib de *lxc*
- Utilise désormais la *libcontainer*
- Démon accessible via une API REST
- Gestion des conteneurs, images, builds, et plus

Exécuter des conteneurs partout

- Sur n’importe quelle plate-forme: physique, virtuel, cloud, etc ..
- Pouvoir passer de l’une à l’autre des plate-formes,
- Maturité des technologies (cgroups, namespaces, copy-on-write)

Distribution efficace des conteneurs

- Distribuer des images, au format standard
- Optimiser l’utilisation disque, mémoire et réseau

Les avantages de Docker

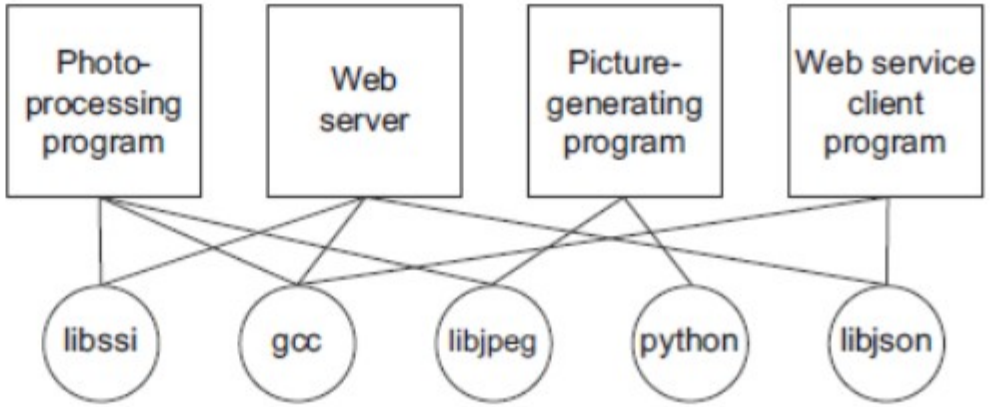
- Remplace les machines virtuels (VM).
- Permet le packaging d’applications.
- Ouverture vers les *microservices*.
- Modélisation d’un réseau informatique avec un budget réduit.
- Permet une certaine productivité mais avec des machines déconnectées.
- Réduire le temps de recherche des bugs.
- Renforce la documentation dès le début du cycle de vie d’une mise en production.
- Permet la mise en place du *Continuous Delivery* (CD).

Les inconvénients de Docker

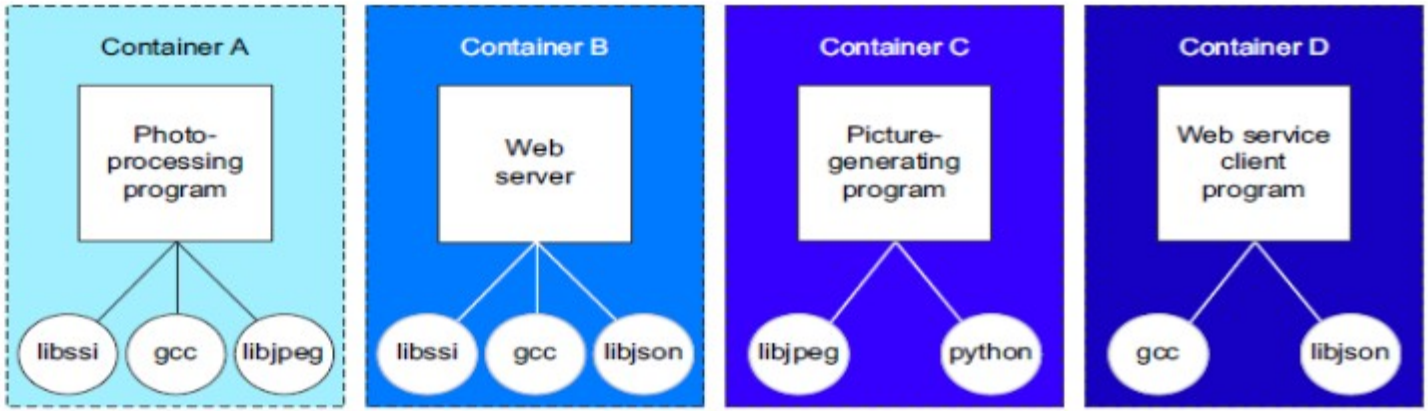
- Fonctionne que sur des noyaux Linux récents > version 3.10.
- Rapide, mais pas aussi rapide que si vous utilisez directement votre machine physique.
- Pas encore complètement sécurisé.
  - Pas encore prêt pour passer en production mais,
  - Certaines sociétés le font déjà. (Red Hat, Google ...).
- Nécessite une remise en cause des équipes de sysadmin et nécessite également un certain temps d’apprentissage.

Sans / Avec Docker

Sans Docker



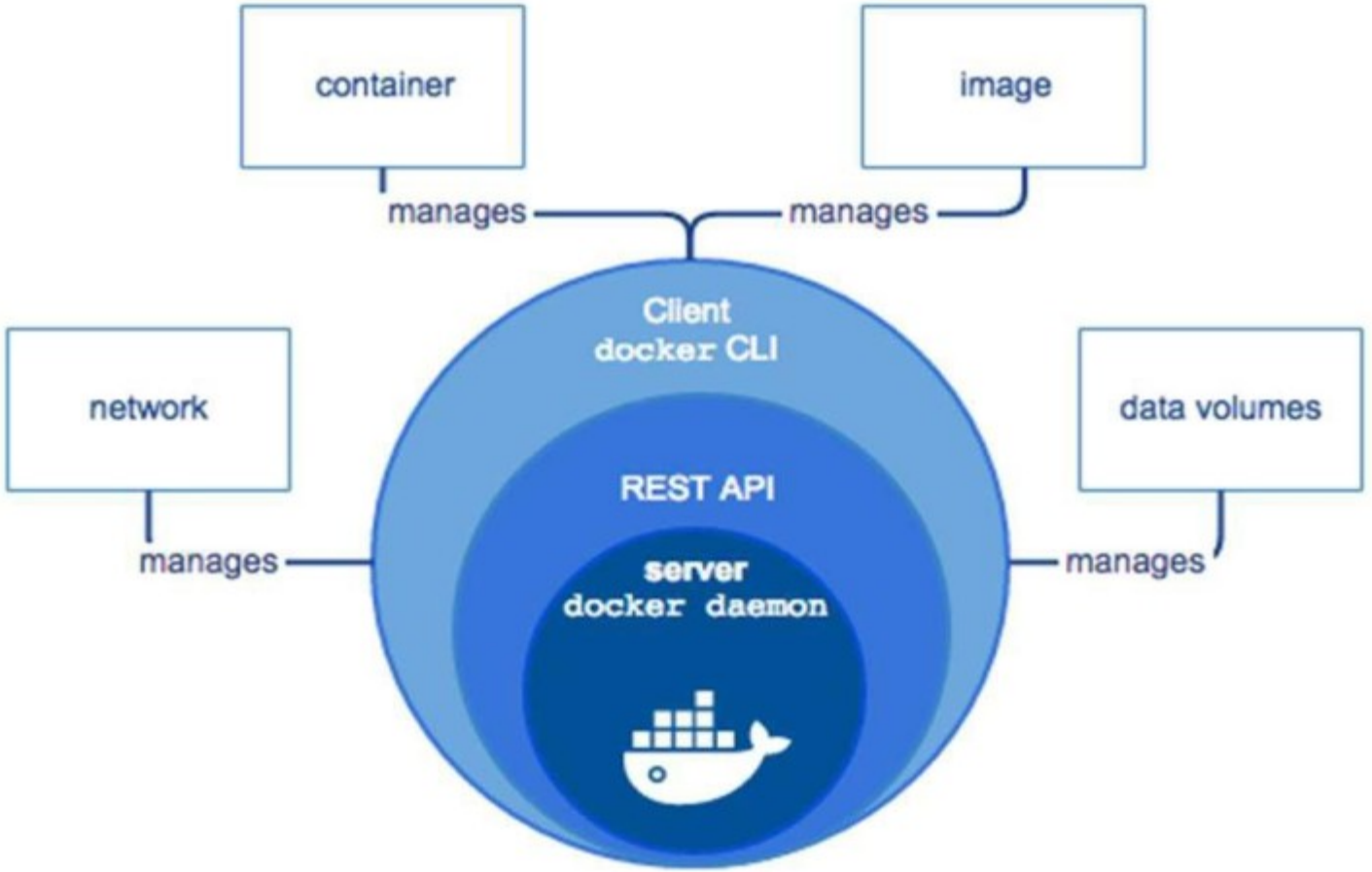
Avec Docker



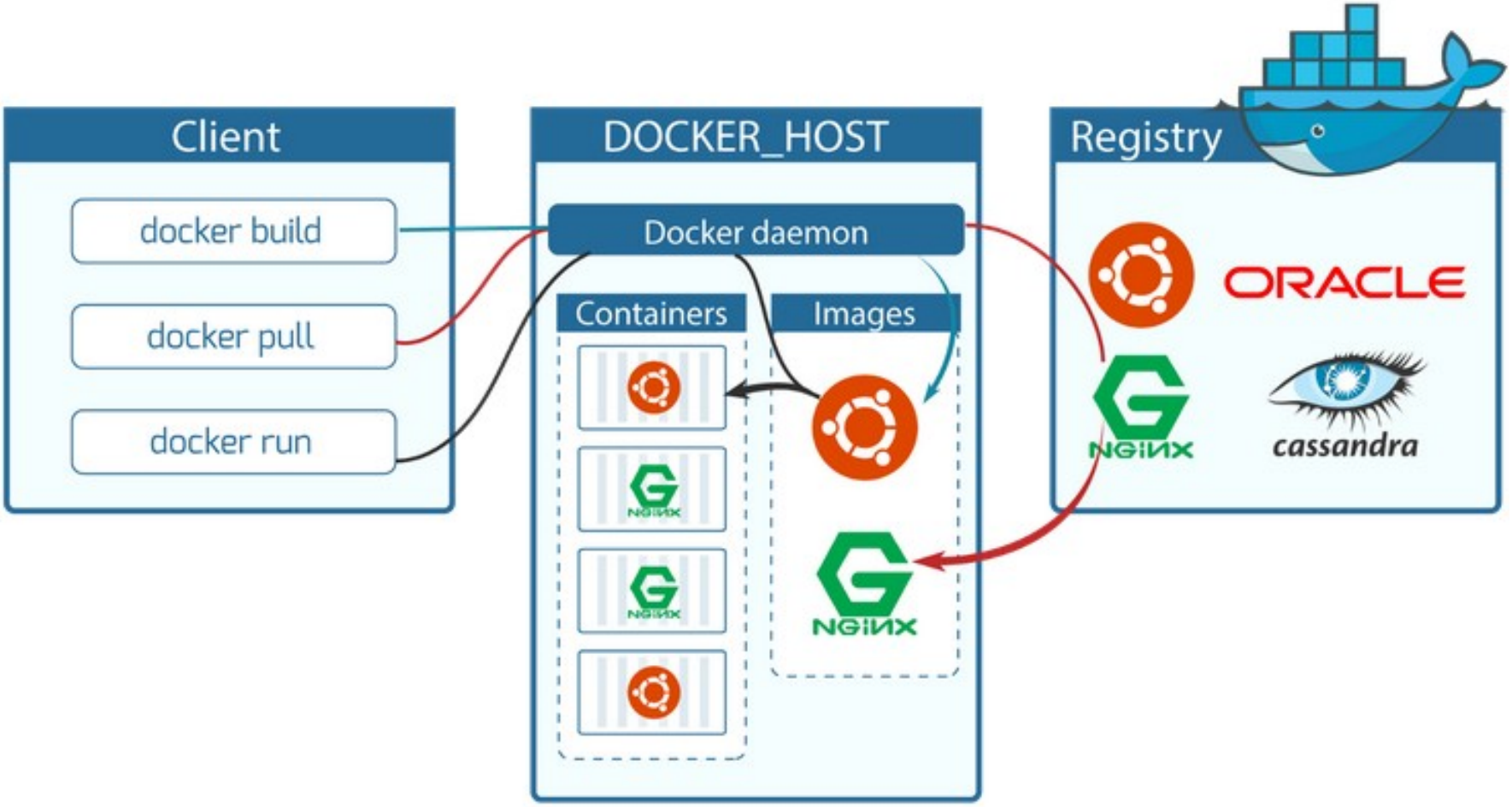
Ecosystème de Docker

- Composants de docker
- Architecture
- Les images Docker
- Registre Docker
- Docker container
- Commandes Docker

Architecture (1)



Architecture (2)

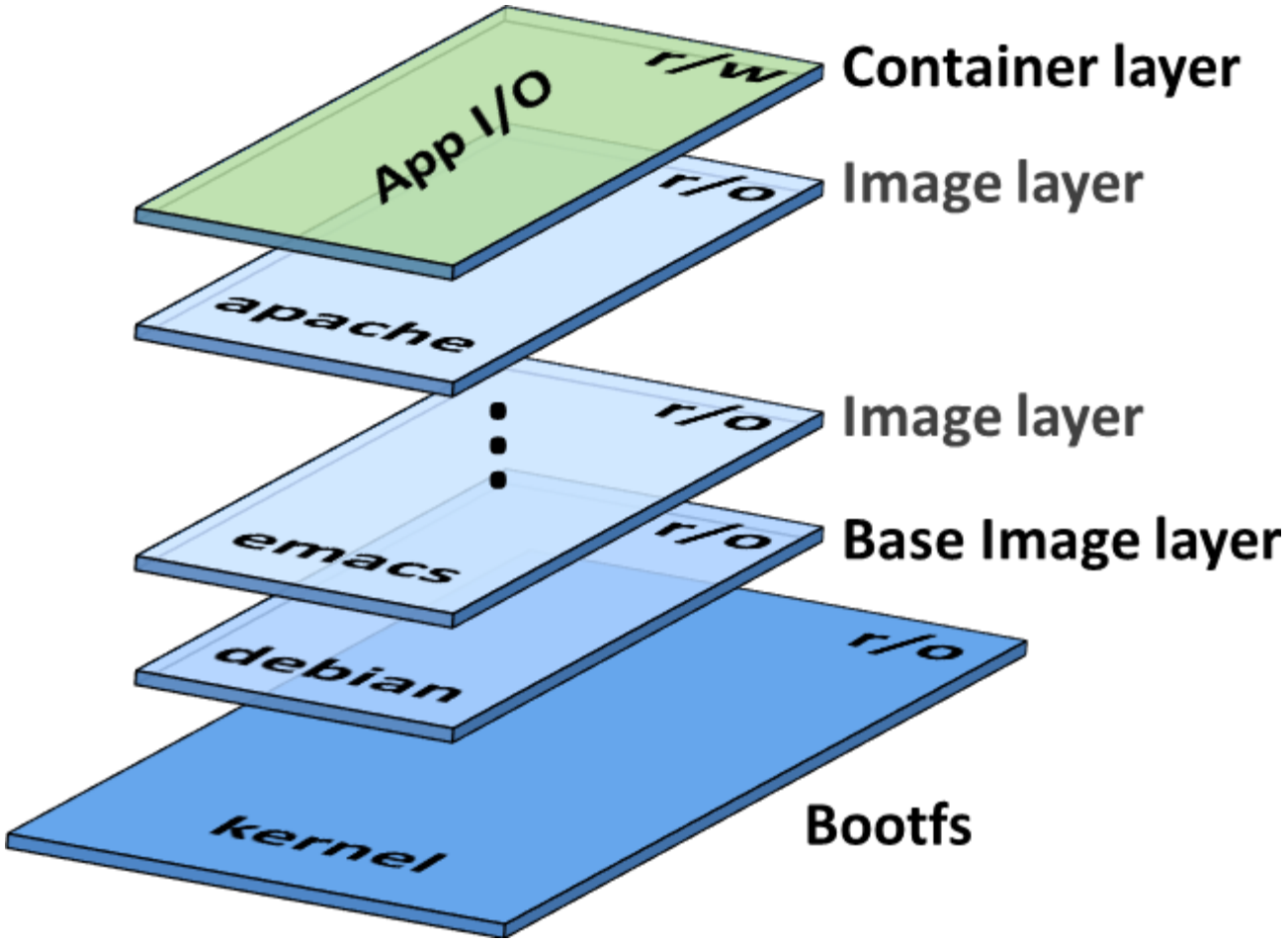


Les images Docker

- 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

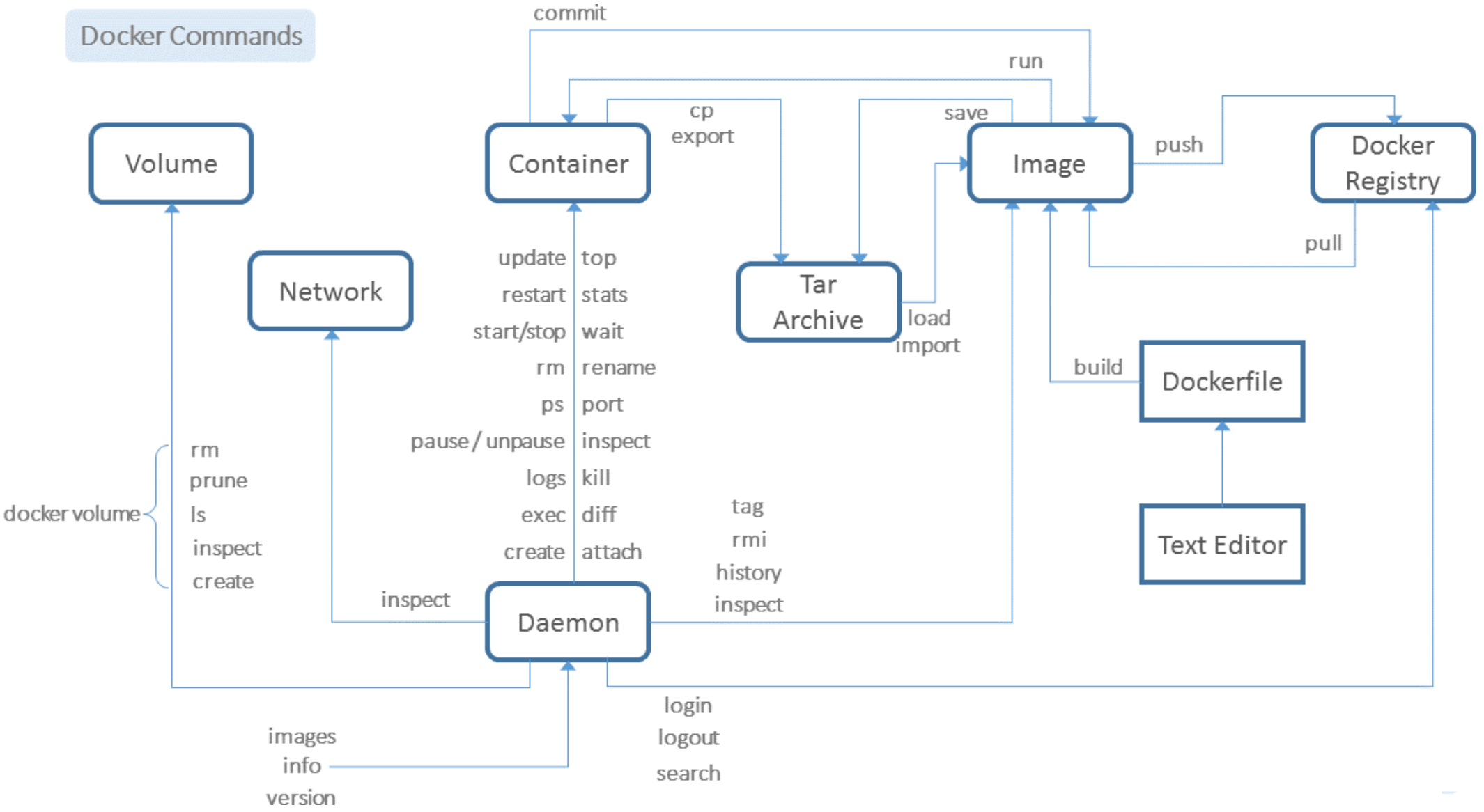
Les images Docker



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

Commandes Docker



Les images Docker

- Qu’est ce qu’une image
- Qu’est ce qu’une couche
- Registre DockerHub
- Registre auto-hébergé
- Les espaces de nom des images
- Rechercher et récupérer des images
- Tagger une image
- Modifier une image

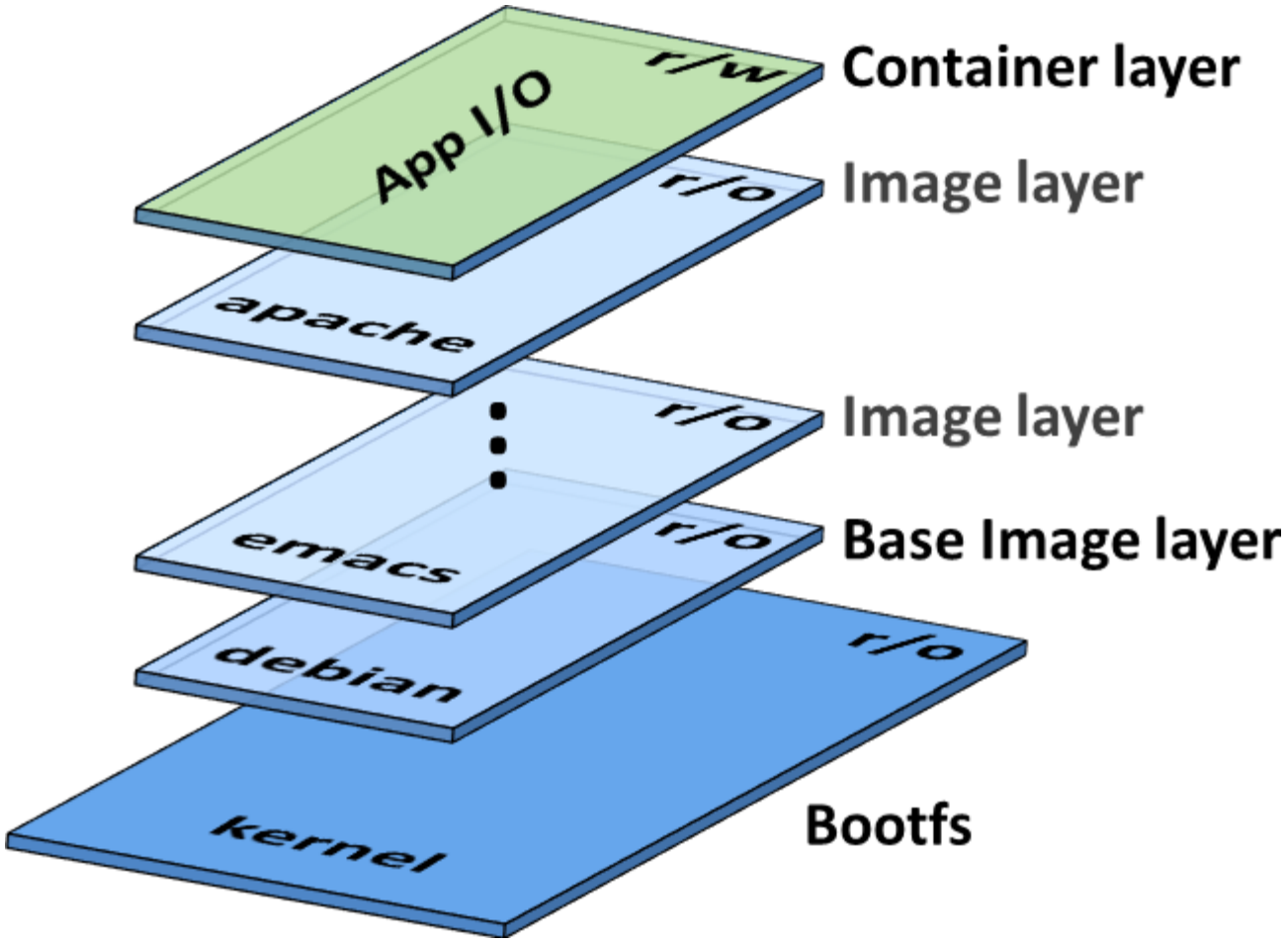
Qu’est ce qu’une image ?

- Collection de fichiers + méta-données
  - Ces fichiers forment le FS racine du conteneur
- Composées de couches, en théorie superposées
- Chaque couche peut rajouter, modifier, supprimer des fichiers
- Des images peuvent partager des couches pour optimiser
  - L’usage disque
  - Les temps de transfert

Qu’est ce qu’une image ?

- Systèmes de fichiers lecture-seule
- Un conteneur est un ensemble de processus s’exécutant dans une copie en lecture-écriture de ce système de fichiers
- Pour optimiser les ressources, le CoW est utilisé au lieu de copier le système de fichiers.
- docker run démarre un conteneur depuis une image

Qu’est ce qu’une couche ?



Docker Hub

- Le registre héberge les images, et les met à disposition.
- Les images sont récupérées en local depuis un registre distant.
- Il existe 2 types de registre:
  - Docker Hub
    - Auto-hébergés
- DockerHub est un service en ligne, officiel de Docker, pour distribuer les images:

  https://hub.docker.com/
- Par défaut, les commandes de Docker liées aux images utilisent le Docker Hub.

Registre auto-hébergé

- Registre hébergé en interne.
- Docker fournit un conteneur pour héberger son propre Registre
- Protocole open-source: différentes implémentations existent.

Espaces de nom

- Il existe 3 manières de nommer des images
- Images officielles:
  - Ubuntu
  - debian
- Images utilisateur sur DockerHub:
  - brahimhamdi/myapp
- Images auto-hébergées:
  - registry.example.com:5000/my-private/image

Rechercher et récupérer une image

Vous pouvez le faire via l'interface web, ou via la CLI :

- *$ docker search nginx*
- *NAME DESCRIPTION STARS OFFICIAL AUTOMATED*
- *nginx*
- *.....*

Faire un pull pour récupérer l'image en local.

$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
aa1efa14b3bf: Pull complete
...
Digest: sha256:75a55d33ecc73c2a242450a9f1cc858499d468f077ea942867e662c247b5e412
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

Elle sera ensuite utilisable pour créer un conteneur avec la commande *docker run*

- NB: *docker run* fait un pull si l'image n'est pas disponible en local

Lister les images

- Lister les images disponibles localement avec *docker images* ou *docker image ls*:

```
$ docker images
REPOSITORY                  TAG       IMAGE ID       CREATED         SIZE
dockercoins_worker          latest    1f49d8a6bd9d   2 days ago      55.2MB
dockercoins_webui           latest    00662d83fad5   2 days ago      218MB
nginx                       latest    62d49f9bab67   2 weeks ago     133MB
```

Tagger une image

- Les images peuvent avoir des tags
- Un tag définira une version, une variante différente d'une image
- Par défaut le tag est latest:
- *docker run ubuntu == docker run ubuntu:latest*
- Un tag est juste un alias, un surnom pour un identifiant d'image
- Exemple :
  - *$ docker tag debian brahimhamdi/debian:10*

  - *$ docker images |grep debian*
  - *brahimhamdi/debian 10 0d587dfbc4f4 2 weeks ago 114MB*
  - *debian latest 0d587dfbc4f4 2 weeks ago 114MB*

Modifier une image

- Si une image est en lecture-seule, comment est-ce que l'on la modifie?
- On ne la modifie pas,
  - On crée un conteneur depuis cette image,
  - On fait nos modifications dans ce conteneur,
  - On transforme ces modifications en une couche,
  - On crée une nouvelle image en validant cette couche par dessus celles de l'image de base.

Création des images Docker

- Méthodes de création d’images Docker
- Premier dockerfile
- Dockerfile et les instructions
- FROM et RUN
- COPY et ADD
- CMD et ENTRYPOINT
- EXPOSE
- VOLUME
- Autres instructions

Méthodes de création d’image Docker

- Deux méthodes :
- **Création usuelle** en utilisant la commande *docker commit*
  - Elle sauvegarde les modifications apportées à un conteneur dans une nouvelle couche : La couche du conteneur convertie en une couche de la nouvelle image
- **Création automatisée** avec la commande *docker build*
  - séquence d’instructions répétables, ***Dockerfile***
  - Méthode recommandée

Premier Dockerfile

- On travaille dans un dossier qui va contenir le Dockerfile propre à notre future image (***Context***)
- *$ mkdir -p ~/docker/nginx*

- On se place dans ce dossier et on ouvre un fichier Dockerfile
- *$ cd ~/docker/nginx/*
- *$ edit Dockerfile*

- Exemple de contenu du Dockerfile:
- *FROM debian*
- *RUN apt-get update*
- *RUN apt-get install -y nginx*

Dockerfile et les instructions

- Un **Dockerfile** est composé d’instructions, une par ligne.
- Le Dockerfile définit les étapes pour créer une nouvelle image.
- Tout commence par une image de base existante.
- L’image nouvellement créée succède à l’image de base.
- Quelques instructions d’un Dockerfile ajoutent des couches à l’image finale.

FROM et RUN

- **FROM**: image de base à utiliser pour notre future image
  - Ajoute les couches de l'image de base
- **RUN**: commandes shell à exécuter sur l'image de base (généralement)
  - seront exécutées durant le processus de build
  - non-interactive: aucun input possible durant le build
  - Chaque RUN ajoute une couche
- Il existe deux formats pour l'instruction RUN :
  - ***Format Shell*** (ou command) qui respecte la syntaxe du shell :
  - *RUN <command>*
  - ***Format exec***, obligatoire si y a pas de shell :
  - *RUN ["executable", "param1", "param2"]*

COPY et ADD

- Copier des fichiers/dossiers dans l'image (lors de la construction de l'image)
- Chaque instruction COPY/ADD ajoute une couche
- **ADD** : la source peut être locale ou distante (URL)s
  - Fichier local de type archive automatiquement décompressé en un dossier, URL non.
- **COPY** : la source doit être locale, et il n'est pas possible de spécifier une URL
  - Ne décompresse pas automatiquement une archive tar compatibleY

CMD et ENTRYPOINT

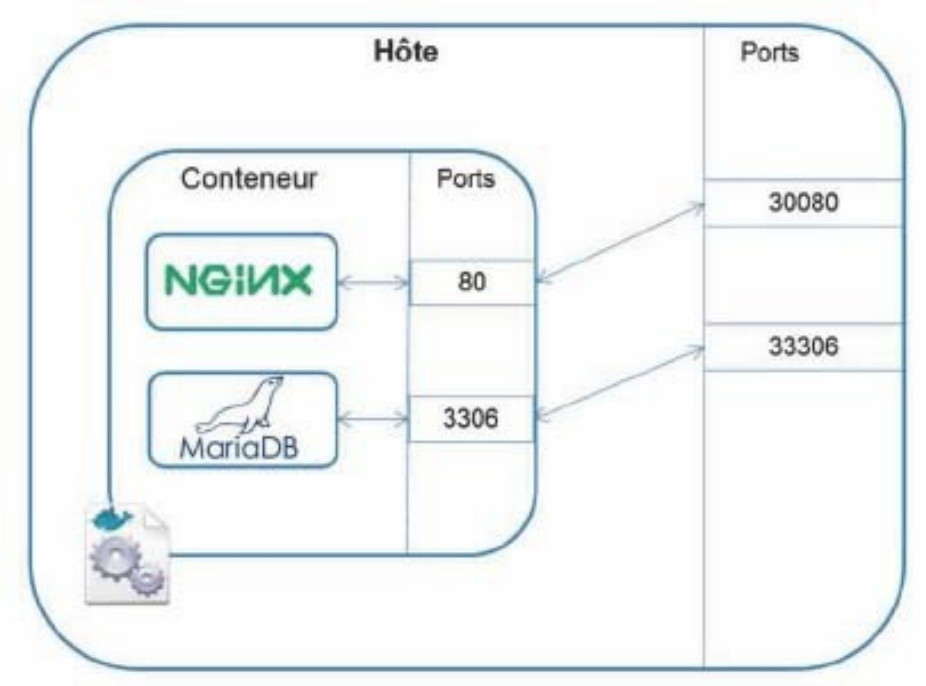
- Exécuter une commande (par exemple, une commande Unix) au démarrage du conteneur.
- N’est pas jouée lors de la construction de l’image, mais lors du démarrage du conteneur ⇒ N’ajoute pas une couche
- Un Dockerfile peut contenir plusieurs instructions **CMD** / **ENTRYPOINT**, cependant seule la dernière instruction du fichier sera exécutée.
- Deux formats comme RUN : ***shell*** et ***exec***
- Il est possible de charger l’instruction **CMD**, par exemple :
  - *docker run monimage echo bonjour*
- Il est possible de surcharger l’instruction **ENTRYPOINT** grâce à l’option --entrypoint, par exemple :
  - *docker run monimage --entrypoint echo bonjour*

CMD et ENTRYPOINT : Différentes combinaisons

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", " p1_entry"]
No CMD	error, not allowed	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

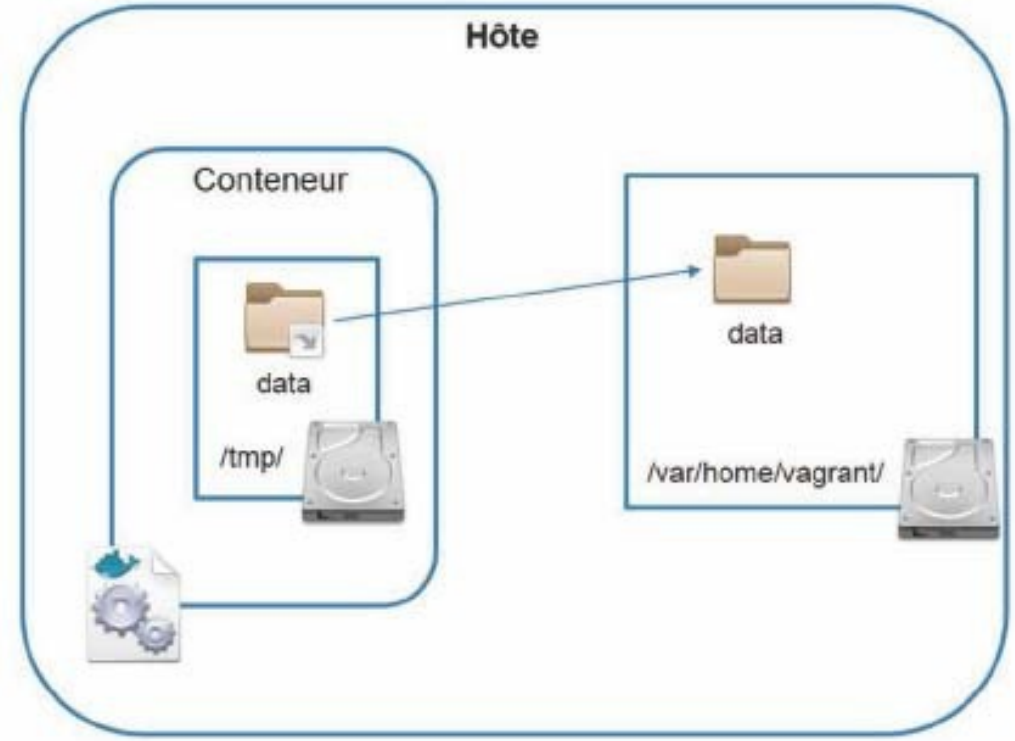
EXPOSE

- Décrit les ports que le conteneur expose et sur lesquels il écoute.
- Un port exposé n’est pas directement accessible, et il devra ensuite être mappé avec un port de l’hôte exécutant le conteneur.



VOLUME

- Permet de créer un point de montage dans l'image
- Le point de montage référera à un emplacement, appelé **volume** dans l'hôte.



Autres instructions Dockerfile

- **ENV** : Définit les variables d’environnement pour le build process et le container runtime
- **WORKDIR** : Modifie le répertoire actuel
- **USER** : Modifie le statut de membre de l’utilisateur et de groupe
- **LABEL** : Ajouter des métadonnées à l’image

Les volumes

- Introduction
- Volume
- Création de volume nommé
- Lister et inspecter les volumes
- Suppression des volumes

Introduction

- Une image Docker est constituée de plusieurs couches, chacune d’elles en lecture seule.
- Pour toute image lancée depuis un conteneur, Docker ajoute une nouvelle couche inscriptible au système.
- Chaque fois qu’un fichier est modifié, Docker crée une copie de celui-ci à partir des couches en lecture seule, qu’il ajoute à la couche inscriptible supérieure.
- Le fichier original (en lecture seule) n’est donc pas modifié.
- En supprimant un conteneur, vous perdrez également sa couche inscriptible supérieure.
- ⇒ Toute modification effectuée après le lancement du conteneur est alors supprimée.

Volume

- Un **volume** de conteneur vous permet de conserver vos données, même en cas de suppression d'un conteneur Docker.
- Également, une solution pratique pour l'**échange de données** entre l'hôte et les conteneurs.
- Un volume consiste en un dossier que se partagent le conteneur et l'ordinateur hôte.
  - Un « **point de montage** » sur l'un des répertoires de l'hôte.
- Plusieurs conteneurs peuvent également **partager** un même volume.

Création d’un volume nommé

- La commande *docker volume create* a pour effet de créer un volume qu’il vous revient de nommer.
- Cette action permet de trouver facilement les volumes Docker pour les attribuer aux conteneurs correspondants.
- Pour créer un volume, utilisez la commande suivante :
- *sudo docker volume create - - name [volume name]*

Utiliser le volume dans un conteneur Docker

- Pour lancer un conteneur qui utilise le volume, ajoutez l'argument suivant à la commande docker run :
- *-v [volume name]:[container directory]*

- Exemple :
- *docker run -d --name my-volume-test -v data-volume:/data nginx*

Lister et inspecter les volumes

- Utilisez la commande suivante pour lister tous les volumes Docker sur votre système :
- *sudo docker volume ls*
- Celle-ci renvoie la liste de tous les volumes Docker déjà créés sur l’hôte.

- Pour obtenir des informations détaillées sur un volume nommé data par exemple, utiliser la commande suivante :
- *docker volume inspect data-volume*

Suppression des volumes

- Il est impossible de supprimer un volume utilisé par un conteneur existant.
- Avant de supprimer celui-ci, vous devez arrêter et supprimer le conteneur Docker.
- Par exemple, pour supprimer le volume *data*, il faut commencer par arrêter et supprimer le conteneur qui l'utilise :
- *docker stop my-volume-test*
- *docker rm my-volume-test*
- Utilisez ensuite la commande suivante pour supprimer le volume de données :
- *docker volume rm data-volume*

Orchestration Docker

- Problématiques
- Axes d’orchestration

Problématiques

- Jusqu’à présent, nous avons travaillé avec une application monolithique, sur un seul hôte Docker.
- Nous voulons déployer et gérer des applications de type micro-services, sur plusieurs hôtes.

Axes d’orchestration

- Réseau
- Docker Compose
- Docker Swarm

Les réseaux Docker

- Introduction
- Les réseaux prédéfinis
- Les pilotes et le réseau Bridge
- Créer et inspecter les réseaux Docker
- Connecter/déconnecter un conteneur au réseau

Introduction

- Une couche de mise en réseau est nécessaire pour que les conteneurs Docker puissent communiquer :
  - entre eux.
  - avec le monde extérieur via la machine hôte
- Cette couche réseau rajoute une partie d'isolation des conteneurs, et permet de créer des applications Docker qui fonctionnent ensemble de manière sécurisée.
- Docker prend en charge différents types de réseaux qui sont adaptés à certains cas d'utilisation

Les réseaux prédéfinis

- Quand vous installez Docker, 3 réseaux sont créés automatiquement, bridge, none, et host

- *$ docker network ls*
- *NETWORK ID NAME DRIVER SCOPE*
- *3c1cc953ba68 bridge bridge local*
- *b66d434b3d42 host host local*
- *037a5f2bfbe4 none null local*

- Ce sont des réseaux prédéfinis : On ne peut pas les supprimer

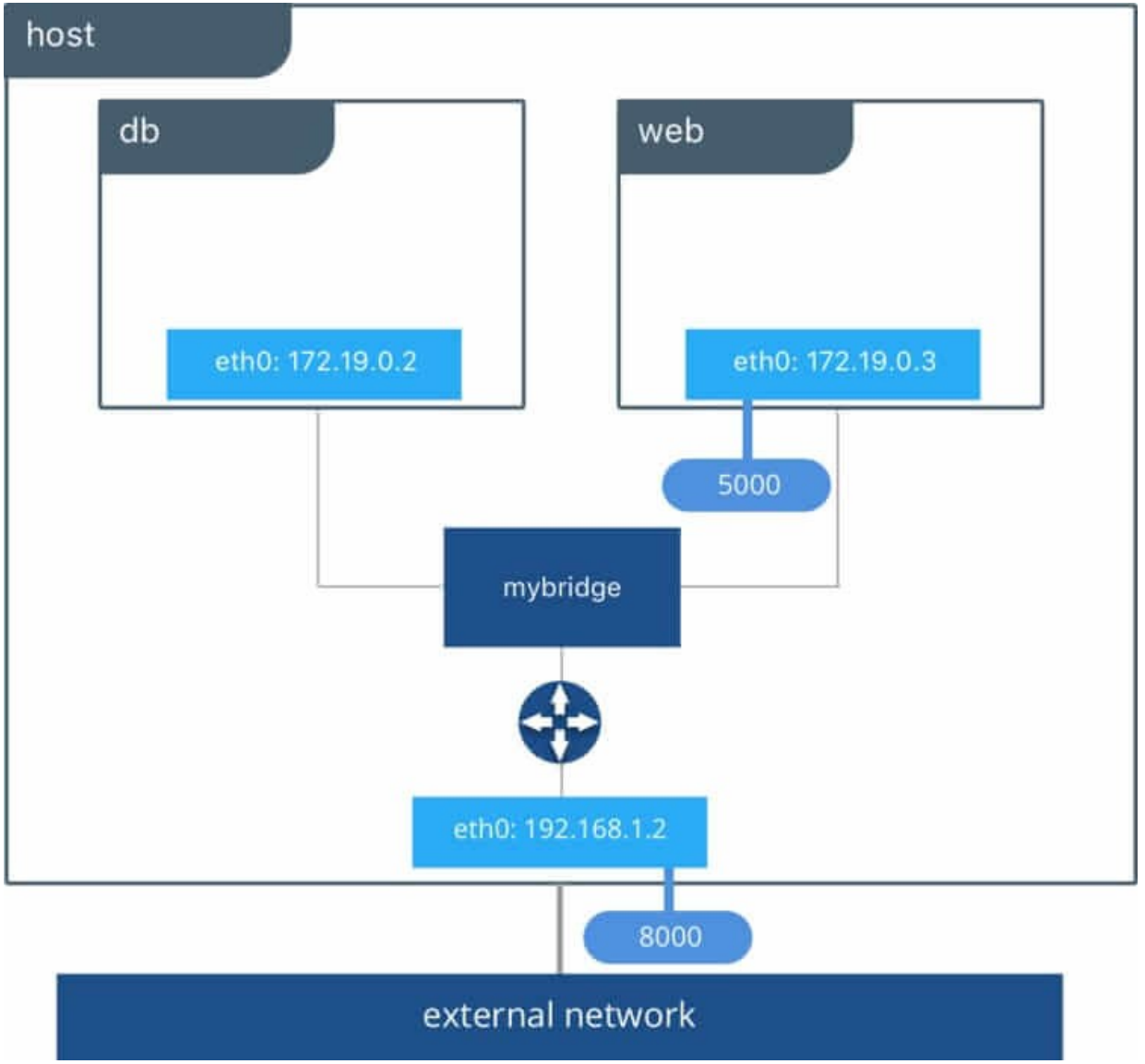
Types de réseaux Docker

- Le système réseau de Docker utilise des drivers (pilotes).
- Ils fournissent des fonctionnalités différentes.
- Les 5 drivers Docker :
  - Bridge
  - None
  - Host
  - Overlay
  - Macvlan

Le réseau Bridge

- C’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.
- Lorsque vous installez Docker pour la première fois, il crée automatiquement un réseau **bridge** nommé **bridge** connecté à l'interface réseau **docker0**
  - Chaque nouveau conteneur Docker est automatiquement connecté à ce réseau, sauf si un réseau personnalisé est spécifié.

Le réseau Bridge

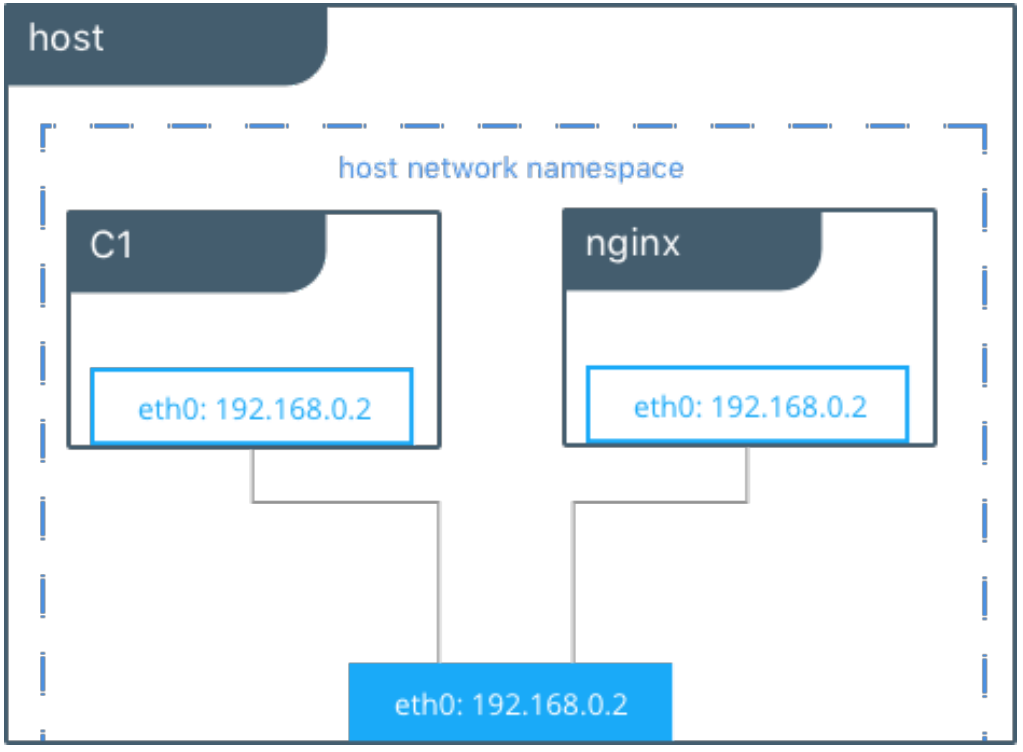


Réseau None

- None:
  - type null: aucun réseau pour un conteneur sur un réseau de ce type.
  - Interdire toute communication interne et externe avec votre conteneur
  - conteneur dépourvu de toute interface réseau

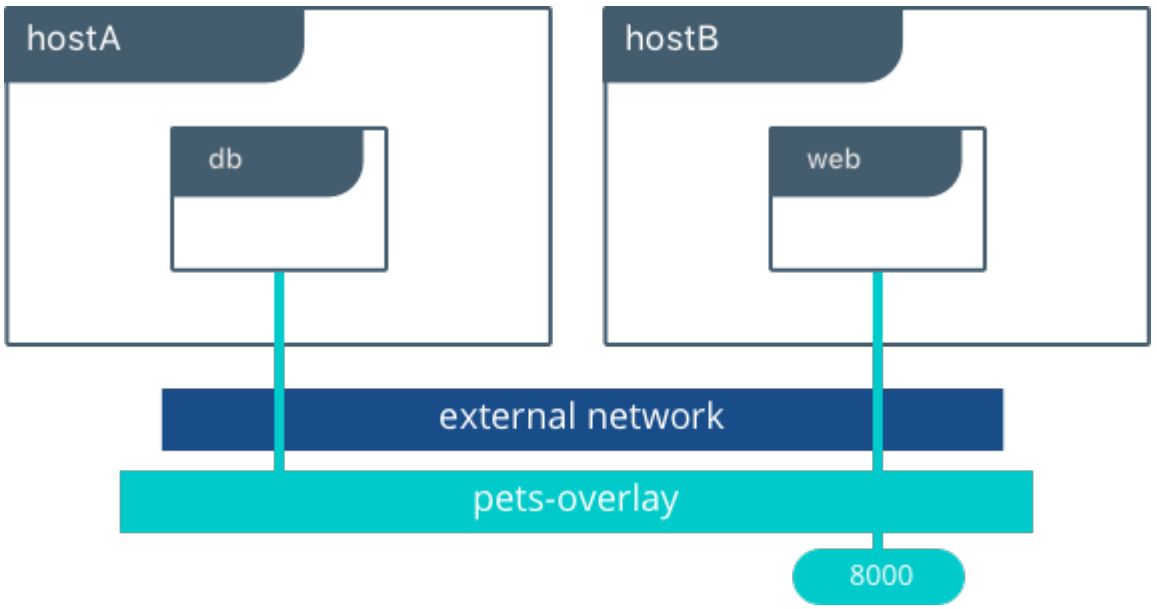
Réseau Host

- Host
  - type host: stack réseau identique à l’hôte
  - Supprime l'isolation réseau entre les conteneurs et seront par défaut accessibles de l'extérieur



Le réseau Overlay

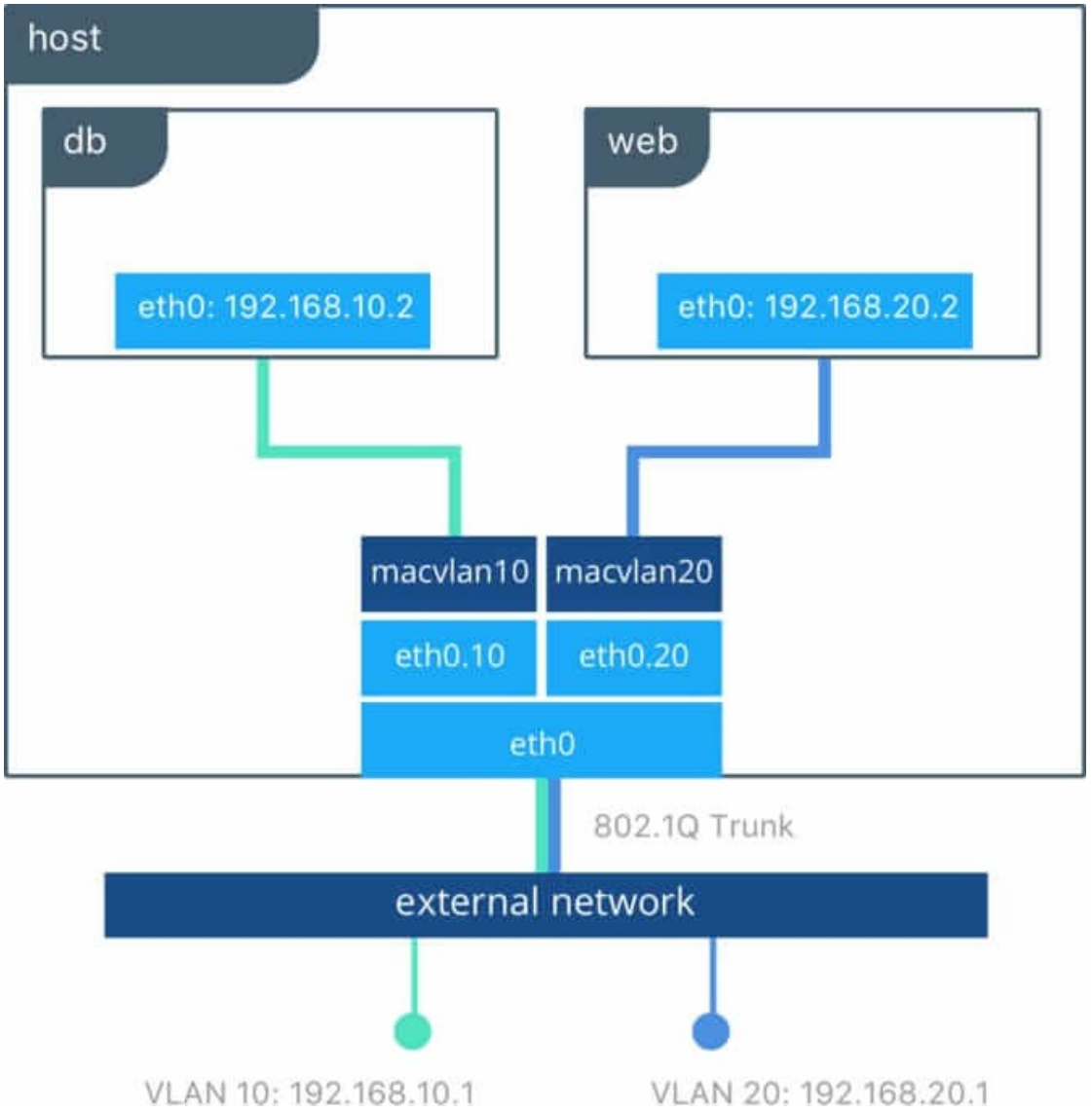
- Si vous souhaitez une mise en réseau multi-hôte native, vous devez utiliser un driver **overlay**.
- Il crée un réseau distribué entre plusieurs hôtes possédant le moteur Docker.
- Docker gère de manière transparente le routage de chaque paquet vers et depuis le bon hôte et le bon conteneur.



Le réseau Macvlan

- Parfois le meilleur choix lorsque vous utilisez des applications qui s'attendent à être directement connectées au réseau physique
  - Permet d'attribuer une adresse MAC à un conteneur et le faisant apparaître comme un périphérique physique sur le réseau.
- Le moteur Docker route le trafic vers les conteneurs en fonction de leurs adresses MAC.

Le réseau Macvlan



Créer et inspecter les réseaux Docker

- Utiliser la commande docker network create pour créer un nouveau réseau, exemple :
  - *docker network create --driver bridge mon-bridge*
- Utiliser la commande suivante pour lister les réseaux :
  - *docker network ls*
- La commande suivante permet de récolter des informations détaillées sur le réseau mon-bridge :
  - *docker network inspect mon-bridge*

Connecter/Déconnecter un conteneur au réseau

- II est possible de connecter un conteneur au réseau :
  - à sa création
  - Pendant son exécution
- Exemple, pour créer un conteneur *mon-container* et le connecter au réseau *mon-réseau*:
- *Docker run -d --name mon-container --network mon-réseau*
- La commande *docker network connect/disconnect*, permet de connecter/déconnecter un ou plusieurs conteneurs à un réseau, exemples :
  - *Docker network connect mon-réseau mon-container*
  - *Docker network disconnect mon-réseau mon-container*

Docker compose

- Bilan
- Problématique
- Exemple: Wordpress
- UTILISATION De Docker Compose
- Les Services
- Exemple Docker-compose.Yml
- Démarrage D’une Application
- Information De Mon Application
- Conteneurs Classiques
- Logs
- Passage À L’échelle : Scale

Bilan

- On sait créer des images:
- On sait lancer des conteneurs
  - partager les données avec des volumes
  - les interconnecter sur le réseau

Problématique

- On veut coordonner des conteneurs
- On veut simplifier la gestion multi-conteneurs
- On ne veut pas utiliser de scripts shell complexes
- On veut une interface standardisée avec l’API Docker

Solution :

- **Compose** permet d'éviter de gérer individuellement des conteneurs qui forment les différents services de votre application.
- Outil qui définit et exécute des applications multi-conteneurs
- Utilise un fichier de configuration dans lequel vous définissez les services de l'application
- A l'aide d'une simple commande, vous contrôlez le cycle de vie de tous les conteneurs qui exécutent les différents services de l'application.

Utilisation de docker compose

- Définir l’environnement de l’application pour qu’il soit possible de la générer de n’importe où.
  - à l’aide de Dockerfile
  - à l’aide d’image officielle
- Définir les services dans un fichier docker-compose.yml
- pour les exécuter et les isoler.
- Exécuter docker-compose qui se chargera d’exécuter l’ensemble de l’application

Les services

- Compose introduit une notion de service :
- Concrètement, un conteneur exécutant un processus
- Chaque conteneur exécute un service inter-dépendant
- Le service peut-être évolutif en lançant plus ou moins d’instances du conteneur avec Compose.
- Exemple Wordpress:
  - Service db
  - Service wordpress

Exemple de docker-compose.yml

```
version: '3'
services:
  wordpress:
    image: wordpress:4.9.8
    restart: always
    volumes:
      - ./wp-content:/var/www/html/wp-content
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_NAME: wpdb
      WORDPRESS_DB_USER: user
      WORDPRESS_DB_PASSWORD: password
    ports:
      - 8080:80
  db:
    image: mysql:8
    command: "--default-authentication-plugin=mysql_native_password"
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: wpdb
      MYSQL_USER: user
      MYSQL_PASSWORD: password
```

Démarrage de l’application

- *$ docker-compose up -d*
- *Creating wordpress ... done*
- *Creating mysql   ... done*

- Les différents services qui composent mon application ont
- été démarrés, avec la configuration et l’environnement qui va bien.

Informations sur l’application

- On utilise la commande ps deCompose:

- *$ docker-compose ps*
- *Name Command State Ports*
- *--*
- *mysql docker-entrypoint.sh --def ... Up 3306/tcp, 33060/tcp*
- *wordpress docker-entrypoint.sh apach ... Up 0.0.0.0:8080→80/tcp*

- *$ docker-compose ps wordpress*
- *Name Command State Ports*
- *--*
- *wordpress docker-entrypoint.sh apach ... Up 0.0.0.0:8080->80/tcp*

Conteneurs classiques

- Les services s’exécutent via des conteneurs sur l’hôte. Les commandes docker classiques sont toujours fonctionnelles.

- *$ docker ps*
- *CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*
- *70f081415426 wordpress:4.9.8 "docker-entrypoint.s..." 5 minutes ago Up 3 minutes 0.0.0.0:8080->80/tcp wordpress*
- *21cf410942c9 mysql:8 "docker-entrypoint.s..." 5 minutes ago Up 4 minutes 3306/tcp, 33060/tcp mysql*

logs

- Logs d’une application:
- *$ docker-compose logs*
- *Attaching to wordpress, mysql*
- *…*
- Logs d’un service :
- *$ docker-compose logs db*
- *Attaching to mysql*
- *…*

Passage à l’échelle

- On peut passer à l’échelle un service.
- Autrement dit, on peut augmenter/diminuer le nombre de conteneurs exécutant un service
- Par défaut, Compose exécute chaque service avec un conteneur.

Scale

- On utilise la commande scale pour changer le nombre de réplicas d’un service:

- *$ docker-compose scale db=2*
- *Creating test_db_2 ... done*

- *$ docker-compose ps db*
- *Name Command State Ports*
- *---*
- *wordpress_db_1 docker-entrypoint.sh --def ... Up 3306/tcp, 33060/tcp*
- *wordpress_db_2 docker-entrypoint.sh --def ... Up 3306/tcp, 33060/tcp*

Docker Swarm

- Problématique
- Solution
- Mode swarm
- Notion de nœud
- Services et tâches
- Diagramme des services et tâches
- Répartition de charge
- Commande node
- Créer notre swarm
- Vérifier l’état de notre swarm
- Première commande en mode swarm
- Sous le capot
- Notion de token
- Ajout d’un worker et manager au cluster
- Promouvoir un worker en manager
- Quitter un swarm
- exécuter et lister les services
- Passage à l’échelle

Problématique

- Vous avez plusieurs hôtes Docker.
- Vous désirez les utiliser sous forme de cluster, et répartir de manière transparente l’exécution de conteneur.

Solution

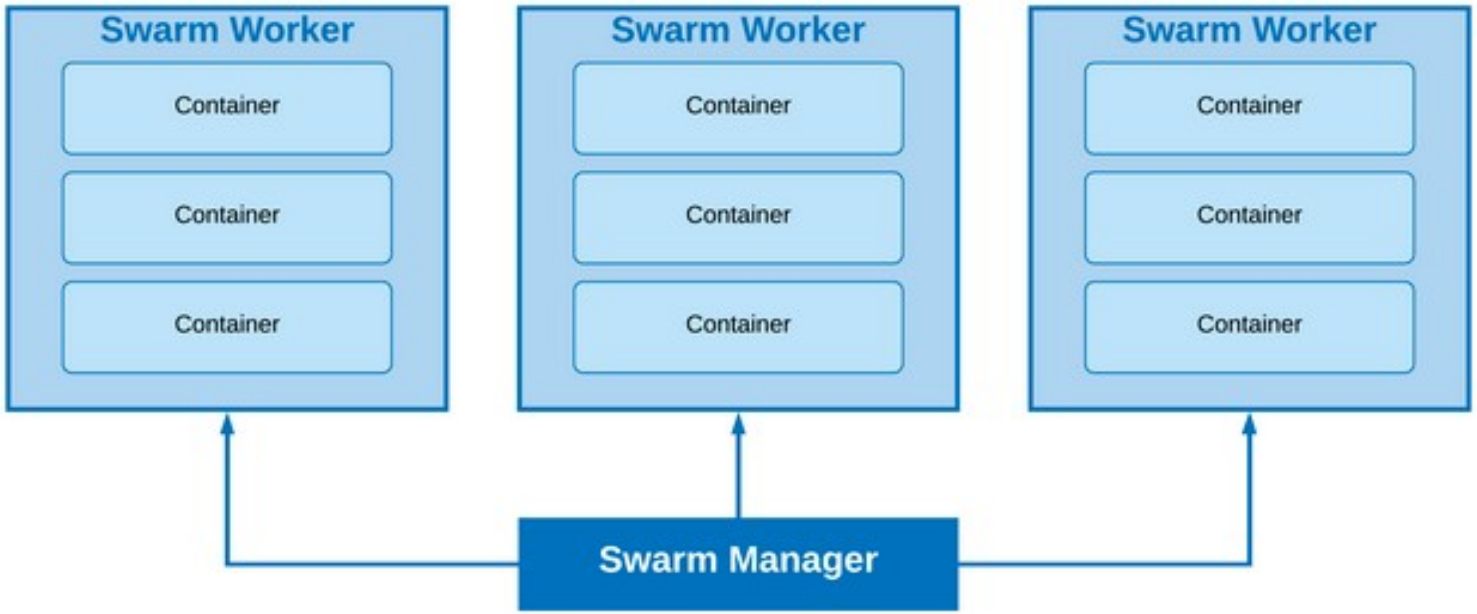
- Docker Engine 1.12 inclut le mode swarm pour nativement gérer un cluster de Docker Engines qu'on nomme un swarm (essaim).
- On utilise la CLI Docker pour :
  - créer un swarm,
  - déployer des service d'application sur un swarm,
  - gérer le comportement du swarm.

Mode swarm

- Les fonctionnalités de gestion et orchestration de cluster incluses dans le Docker Engine.
- Les Moteurs Docker participant à un cluster s'exécutent en mode swarm.
- Un swarm (essaim) est un cluster de Docker Engines sur lequel vous déployez des services.
- La CLI Docker inclut la gestion des nœuds d'un swarm
  - ajout de noeuds,
  - déploiement de services,
  - gestion de l'orchestration des services

Notion de nœud

- Un nœud est une instance Docker Engine participant à un swarm.
- 2 types de nœuds :
  - Manager
  - Worker



Nœud de type manager

- déploie les applications suivant les définitions de services
- dispatche les tâches au nœud de type worker, Gestion du cluster, orchestration
- Un leader est choisi parmi les managers pour gérer les tâches d’orchestration

Nœud de type worker

- Reçoit et exécute les tâches depuis les managers
- Un manager est également un worker
- Notifie les managers de son état pour l'orchestration

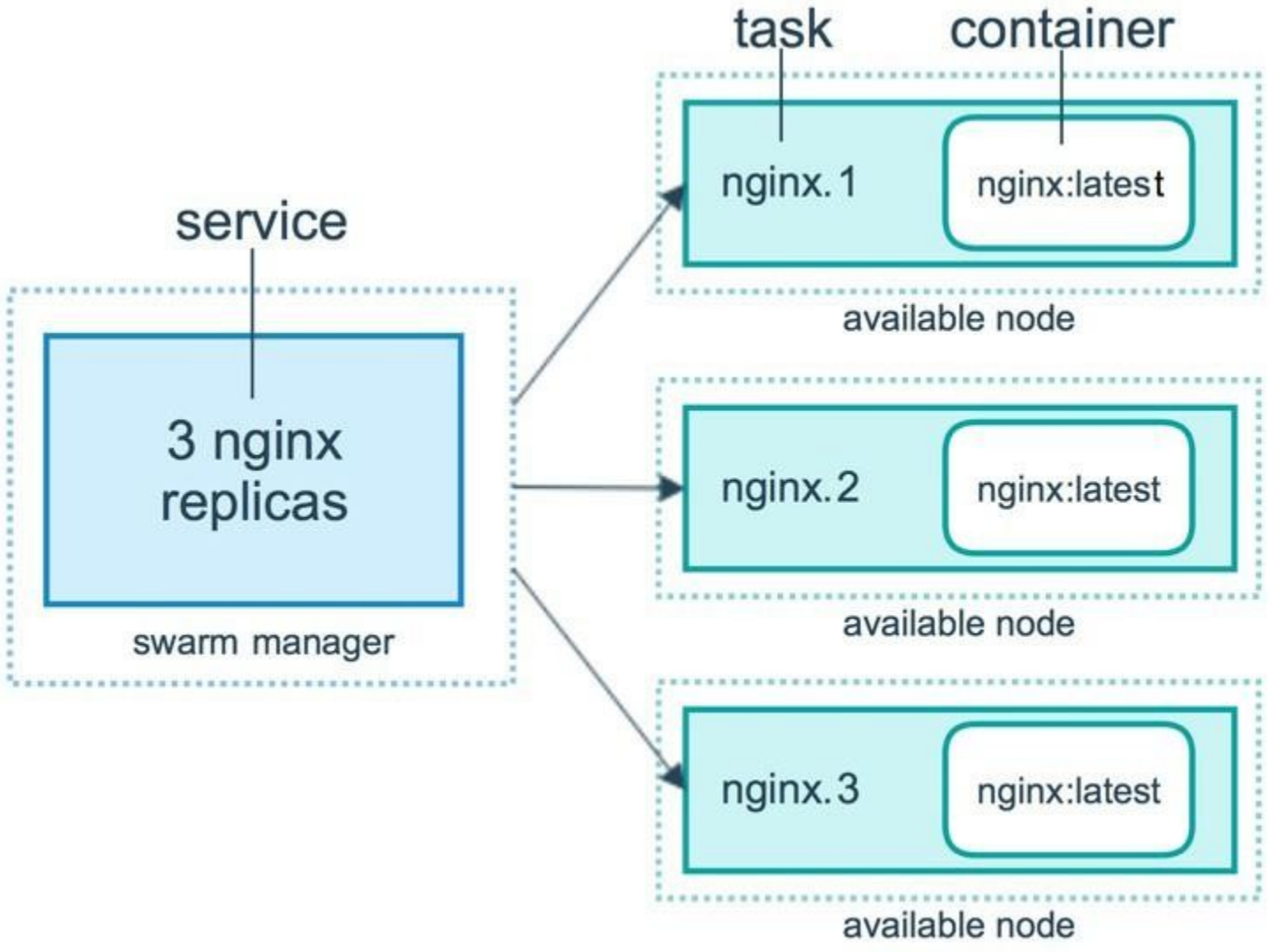
Services et tâches (1)

- Un service est la définition de tâches à exécuter par les workers
  - exécution d’une commande via un conteneur utilise une image
- deux modes d’exécution de service:
  - repliqué: un manager distribue un nombre donné de tâches sur chaque noeud
  - global: exactement une tâche est exécuté par noeud

Services et tâches (2)

- tâche: unité atomique d’exécution d’un swarm
  - représente le conteneur et la commande à y exécuter
  - assignée à un worker par un manager suivant le nombre de réplica défini par le service
  - ne peut changer de nœud, s'exécute sur ce nœud ou échoue.

Diagramme des services et des tâches



Répartition de charge

- Le manager utilise une répartition de charge vers tous les workers pour exposer les ports des services
- Le port rendu public est également accessible sur tout worker du swarm
- Chaque service du swarm à son propre nom DNS interne:
- Le manager utilise une répartition de charge interne pour distribuer les requêtes des services du cluster.

Commande node

- *$ docker node ls*
- *Error response from daemon: This node is not a swarm manager. Use "docker swarm init" or "docker swarm join" to connect this node to swarm and try again.*

- Un cluster est initialisé avec docker swarm init. A exécuter une fois sur un hôte.

Créer notre swarm

- *$ docker swarm init --advertise-addr 192.168.99.1*
- *Swarm initialized: current node (8ohd998buvm3pdi4bulojekjh) is now a manager.*

- *To add a worker to this swarm, run the following command:*
- *docker swarm join --token SWMTKN-1-400ojx3v2lkmchcl6sjwsjorueo4en7v27zilq72v9f2c5tejr-2t2uhff9rnr2xdnjh731v8g8b 192.168.99.1:2377*

- Dans la sortie de la commande, un message indique la
- commande à exécuter pour ajouter un worker au nouveau swarm.

Vérifier l’état du swarm

- On utilise la classique commande docker info:

- *$ docker info*
- *…*
- *Swarm: active*
- *NodeID: 8ohd998buvm3pdi4bulojekjh*
- *Is Manager: true*
- *ClusterID: yn9s0kwn9hiz36rt0qiitwrlf*
- *Managers: 1*
- *Nodes: 1*
- *…*

Première commande en mode swarm

- Pour voir les informations des noeuds du swarm:

- *$ docker node ls*
- *ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION*
- *8ohd998buvm3pdi4bulojekjh * DevOps Ready Active Leader 19.03.14*

Sous le capot

- Lors du docker swarm init, un certificat Racine TLS a été créé.
- Puis une paire de clés pour le premier nœud, signée avec le certificat.
- Pour chaque nouveau noeud sera créée sa paire de clé signée avec le certificat.
- Toutes les communications sont ainsi chiffrées en TLS.

Notion de token

- Docker a généré 2 tokens de sécurité (équivalent d'une passphrase ou password) pour le cluster, à utiliser lors de l'ajout de nouveaux nœuds:
  - Un token pour les workers
  - Un token pour les managers

- Récupération des tokens:
- *$ docker swarm join-token worker*
- *$ docker swarm join-token manager*

Ajout d’un worker au cluster

- Se connecter à un autre serveur Docker:

- *$ docker swarm join --token SWMTKN-1-400ojx3v2lkmchcl6sjwsjorueo4en7v27zilq72v9f2c5tejr-2t2uhff9rnr2xdnjh731v8g8b 192.168.99.1:2377*
- *This node joined a swarm as a worker.*

Cluster de 2 nœuds

- *$ docker node ls*
- *ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION*
- *8ohd998buvm3pdi4bulojekjh * DevOps Ready Active Leader 19.03.14*
- *zqda5t1vg5pmvgd3owxoj8waq host1 Ready Active 19.03.12*

Ajout d’un manager

- *$ docker swarm join --token SWMTKN-1-400ojx3v2lkmchcl6sjwsjorueo4en7v27zilq72v9f2c5tejr-4cm91hmbr8c2pnzh6e80e7c87 192.168.99.1:2377*
- *This node joined a swarm as a manager.*

Cluster de 3 nœuds

- $ docker node ls
- ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION
- 8ohd998buvm3pdi4bulojekjh * DevOps Ready Active Leader 19.03.14
- zqda5t1vg5pmvgd3owxoj8waq host1 Ready Active 19.03.12
- 42i7q0ld6f7hi0p0up0b691e8 host2 Ready Active Reachable 19.03.12

Promouvoir un worker en manager

- *$ docker node promote host1*
- *Node host1 promoted to a manager in the swarm.*

- *$ docker node ls*
- *ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION*
- *8ohd998buvm3pdi4bulojekjh * DevOps Ready Active Leader 19.03.14*
- *zqda5t1vg5pmvgd3owxoj8waq host1 Ready Active Reachable 19.03.12*
- *42i7q0ld6f7hi0p0up0b691e8 host2 Ready Active Reachable 19.03.12*

- L’inverse : demote
- *$ docker node demote host2*
- *Manager host2 demoted in the swarm.*

- *$ docker node ls*
- *ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION*
- *8ohd998buvm3pdi4bulojekjh * DevOps Ready Active Leader 19.03.14*
- *zqda5t1vg5pmvgd3owxoj8waq host1 Ready Active Reachable 19.03.12*
- *42i7q0ld6f7hi0p0up0b691e8 host2 Ready Active 19.03.12*

Quitter un swarm

- Sur host2 par exemple :
- *$ docker swarm leave*
- *Node left the swarm.*

- *$ docker node ls*
- *ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION*
- *8ohd998buvm3pdi4bulojekjh * DevOps Ready Active Leader 19.03.14*
- *zqda5t1vg5pmvgd3owxoj8waq host1 Ready Active Reachable 19.03.12*
- *42i7q0ld6f7hi0p0up0b691e8 host2 Down - 19.03.12*

- *$ docker node rm host2*

Exécuter un service

- On utilise la commande service sur un manager en mode swarm

- *$ docker service create --replicas 1 --name helloworld alpine ping docker.com*
- *za9zewvlaozwclcd8ies0shmi*
- *overall progress: 1 out of 1 tasks*
- *1/1: running   [==>]*
- *verify: Service converged*

Lister les services

- $ docker service ls
- ID NAME MODE REPLICAS IMAGE PORTS
- 2as0rkv51p4x helloworld replicated 1/1 alpine:latest

ps

- $ docker service ps helloworld
- ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS
- rqka35u1ykvy helloworld.1 alpine:latest DevOps Running Running 24 seconds ago

Passage à l’échelle : exemple

- $ docker service scale helloworld=5
- helloworld scaled to 5

- $ docker service ps helloworld
- ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR
- rqka35u1ykvy helloworld.1 alpine:latest DevOps Running Running 3 minutes ago
- sgzrb4967c00 helloworld.2 alpine:latest host1 Running Running 12 seconds ago
- kuy233ozz4fe helloworld.3 alpine:latest DevOps Running Running 23 seconds ago
- ign6un71o1ti helloworld.4 alpine:latest host2 Running Running 11 seconds ago
- j9vnrg3f3ig0 helloworld.5 alpine:latest host1 Running Running 11 seconds ago