# CONTAINÉRISATION DOCKER

VINCENT LAINE

# SOMMAIRE

- Conteneurisation Docker
- Bénéfices des containers
- Apports pour les Dev-Ops
- Docker hub
- NOTIONS DE BASE DOCKER
- OFFRES COMMERCIALES
- DOCKER COMPOSE
- DOCKER MACHINE
- DOCKER SWARM
- DOCKER ENGINE
- DOCKER HUB / REGISTRY
- FONCTIONNEMENT DES VOLUMES DE STOCKAGE EN CLUSTER
- Le réseau dans Docker (Que les 3 principaux)
- Variables d'environnement
- Run – port mapping
- Run – volume mapping
- Docker Commands
- Inspect container
- Container logs
- DOCKER FILE
- Build Multistage
- TAG / PUSH / PULL

# CONTENEURISATION DOCKER

- Qu’est ce que Docker ?

- Docker permet d'embarquer une application dans un ou plusieurs containers logiciels qui pourra s'exécuter sur n'importe quel serveur machine.

Docker est une plate-forme de virtualisation par conteneur qui, à la différence de la virtualisation par hyperviseur qui nécessite de créer une machine virtuelle embarquant un système d'exploitation invité pour s'assurer de son indépendance avec la machine principale.

Il va permettre d'isoler une ou plusieurs applications dans des conteneurs en utilisant les ressources de la machine hôte.

Docker doit pouvoir être exécutée sur n'importe quel environnement, quel que soit son contenu, évitant ainsi divers problèmes de compatibilité entre les applications et l'hôte qui les exécute.

# DOCKER

**Avantages**

- Un conteneur n'a besoin que de quelques centaines de Mo d'espace disque pour être déployé, contrairement à une VM. Un conteneur va directement utiliser les ressources dont il a besoin sur le serveur physique.
- La possibilité d'exploiter directement une application sans être contraint de modifier les librairies installées sur la machine hôte ou de se soucier des versions des logiciels installés sur cette dernière.
- C'est un gain de temps non négligeable, l'administrateur système n'a plus autant à s'accorder avec l'équipe de développement pour être certain que son serveur possédera toutes les dépendances nécessaires au déploiement d'une application.
- La possibilité de partager la configuration complète d'une application sur n'importe quel poste grâce aux Dockerfiles.
- Une simple exécution du fichier permet de déployer des conteneurs prêts à l'emploi, avec la possibilité d'en modifier certains aspects sans affecter leur configuration.

# DOCKER

**Inconvénients**

- Comme les conteneurs utilisent directement les ressources de la machine hôte, il est par exemple impossible de déployer un conteneur basé sur un système d'exploitation Windows sur un serveur physique Linux.
- Certaines applications ne se prêtent pas à un déploiement morcelé par couche car certaines d'entre elles ne peuvent fonctionner que de façon monolithique, il est donc préférable d'effectuer des tests de conteneurisation avant de se lancer dans un déploiement d'application déjà existante.
- Il peut être parfois difficile de définir une sécurité poussée entre les différents environnements Docker d'une même machine physique, les conteneurs utilisant les ressources du noyau et des composants systèmes de cette dernière, certaines situations ont plus de chance de se répercuter sur le système d'exploitation sous-jacent ou sur d'autres conteneurs.

# BÉNÉFICES DES CONTAINERS

- Agilité
  - Les containers sont rapides et simples à mettre en œuvre
- Portabilité
  - Les containers sont portables sur tout type d'environnement
  - Du poste développeur, aux serveurs de l'entreprise, au plateformes cloud privées ou publiques
- Consistance
  - Les containers ne subissent aucune modification du poste du développeur à l'environnement de production
  - Ils sont conçus pour être réutilisés
- Optimisation des coûts
  - Que ce soit en en termes de ressources nécessaires (RAM, stockage..) ou de management (agilité, portabilité…)
  - Profite des coûts d'investissement plus bas du Cloud provider CAPEX/OPEX
- Elasticité
  - Orchestrateur !
  - Autoscaling

ANDN SERVICES

# APPORTS POUR LES DEV-OPS

- Développeurs
  - « build once… run everywhere »
  - Le container est un environnement sécurisé, stable, portable sur tous environnements
  - Il résout les problèmes de dépendances ou de packages manquant lors du déploiement
  - Plusieurs versions de librairies peuvent être utilisées dans différentes containers du fait de leur isolation
  - Forte automatisation de toutes les phases (test, intégration, packaging..)
  - Réduit voire élimine les problèmes d'environnement clients différents de ceux de l'éditeur
  - Rapidité de déploiement, de retour arrière comparativement aux VMs
  - En définitif, tout semble simple et rapide !

# APPORTS POUR LES DEVOPS

- Administrateurs
  - « configure once… run everywhere »
  - Rend le cycle de vie des applications plus efficace, consistant, et  reproductible
  - Améliore le rendu de la qualité du code développé
  - Élimine définitivement les inconsistances entre les différents environnements de développement, test, et production
  - Apporte une répartition des responsabilités claire avec les développeurs
  - Améliore sensiblement la vitesse et l'agilité des procédés d'intégration et de déploiement continu
  - Améliore beaucoup d'aspects comparativement aux VMs du fait de leur taille réduite : performance, coûts, déploiement, portabilité..

# APPORTS POUR L'ENTREPRISE

- Stratégie cloud
  - Facilite l'évolution vers le(s) cloud(s) public(s)
  - Facilite l'évolution vers l'hybridation et le multi-clouds
- Stratégie de modernisation applicative
  - Containerisation des applications « legacy »
  - Transformation des applications « legacy » en microservices
  - Accélère le développement de nouvelles applications sur le modèle microservices
- Stratégie DevOps
  - Résout les problèmes de gestion des différents environnements par la consistance des containers
  - Facilite les démarches d'intégration et de déploiement continu
  - Facilite l'autoscaling des applications

# LIMITES - MOBILITÉ DES CONTAINERS

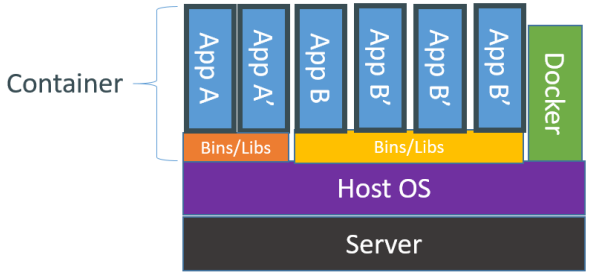
- Registre
  - Registre Docker public
  - Registre privé
  - Registre en mode SaaS
  - Quay.io
- …

- Problématique des données
  - Stockage objet
  - Bases de données !

- Kubernetes
  - 2017 meilleure gestion du stockage stateless/statefull sur les pods

# LIMITES - SÉCURITÉ DES CONTAINERS

- Isolation
  - Isolation native entre les containers
  - Partage de librairies pour l'optimisation mais en lecture seule




- Sécurité
  - Kernel « hardened » > Grsecurity / PaX
  - Linux Security Module
  - SE Linux / AppArmor
  - Os Minimaux / réduction de la surface d'attaque > CoreOS / Fedora Atomic / VMWare Photon…

# DOCKER

Docker est une technologie de conteneurisation reposant sur le noyau Linux et ses fonctionnalités de virtualisation par conteneurs (LXC pour Linux Containers), notamment :

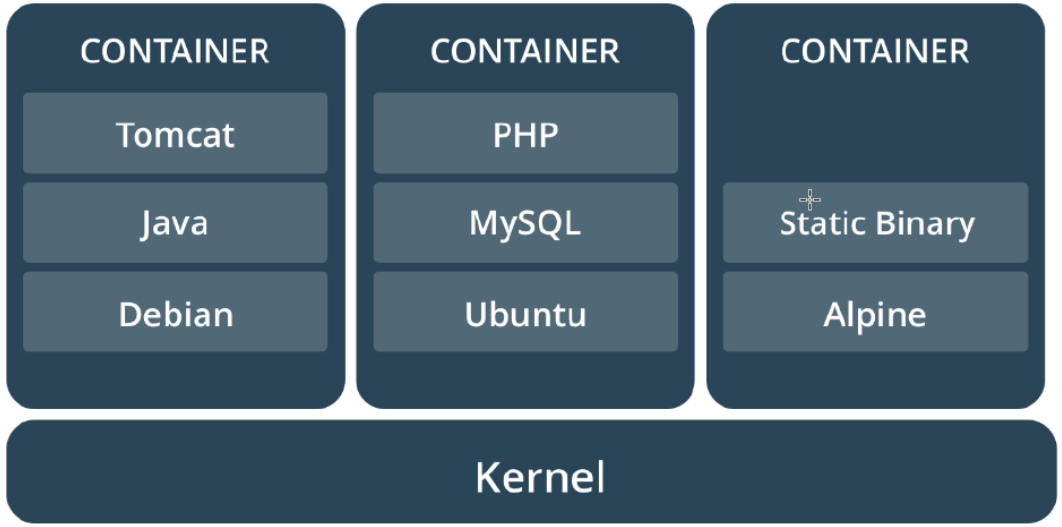
- le composant cgroups pour contrôler et limiter l'utilisation des ressources pour un processus ou un groupe de processus (utilisation de la RAM, CPU entre autres) associé au système d'initialisation systemd qui permet de définir l'espace utilisateur et de gérer les processus associés ;
- les espaces de noms ou namespaces qui permettent de créer des environnements sécurisés de manière à isoler les conteneurs et empêcher par exemple qu'un groupe puisse « voir » les ressources des autres groupes.

Docker utilise des fonctionnalités natives au noyau Linux, comme les cgroups ou les namespaces, mais offre les outils pour le faire de manière simplifiée pour permettre, entre autres :

- la duplication et la suppression des conteneurs ;
- l'accessibilité des conteneurs à travers la gestion des API et CLI ;
- la migration (à froid ou à chaud) de conteneurs.

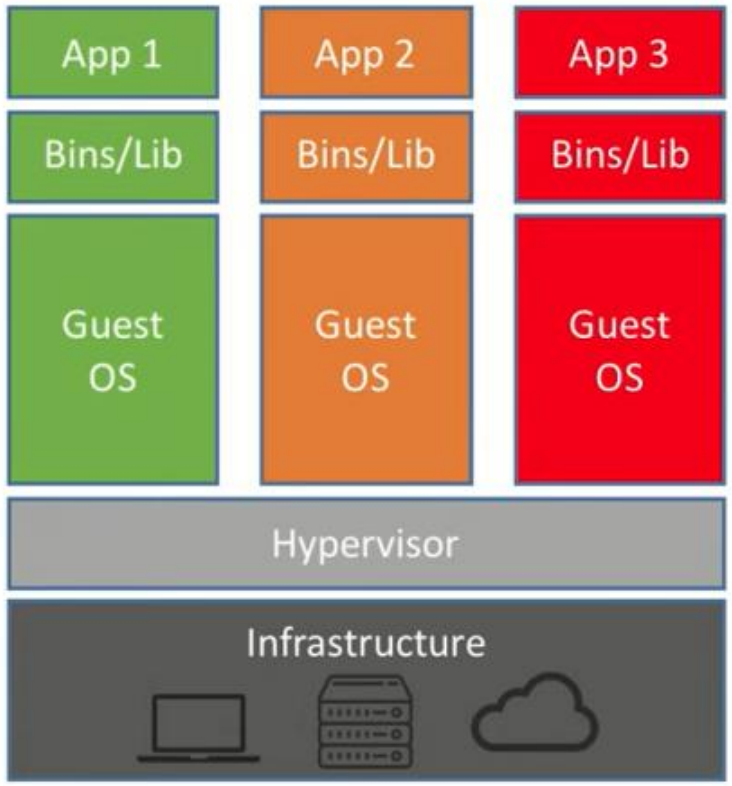
# DOCKER

- Les conteneurs Docker sont construits à partir des images Docker. Une image Docker est un package léger, autonome et exécutable d'un logiciel qui inclut tout ce qui est nécessaire pour l'exécuter :
  - code de l'application,
  - environnement d'exécution (runtime),
  - outils système et librairies, etc.
- Disponible pour les applications basées sur Linux et Windows, le logiciel conteneurisé fonctionnera toujours de la même manière, quel que soit l'environnement.

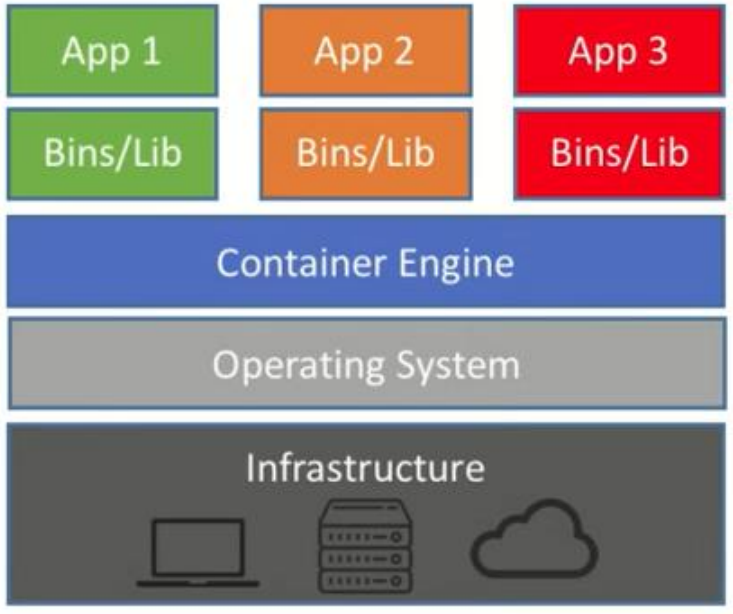


# DOCKER

- Différence entre Docker et Machine Virtuelle




Machine Virtualization




Containers

ANDN SERVICES

# DOCKER

Machine Virtuelle

- Basée sur un Hyperviseur
- Permet d'émuler des machines complètes : un Os avec une ou plusieurs applications
- Machine virtuelles lourdes (plusieurs giga)

Docker

- Plus léger, baser sur un moteur de conteneur (Docker)
- Sur la couche supérieure, on retrouve les applications installées dans chacun des conteneurs
- Un seul système d'exploitation, celui de la machine hôte qui fait bénéficier de ses ressources aux différents conteneurs
- Conteneur plus léger (rarement au-dessus de 500 Mo)
- Permet de segmenter une application en micro-service (un docker par service : apache, base de données …)

# POURQUOI TANT D'ENGOUEMENT ? QUELLES SONT LES UTILISATIONS POSSIBLES, EFFICACES ET PERTINENTES DE DOCKER ?

- Déployer rapidement un service lorsque l'on a besoin de le déployer plusieurs fois : cette reproductibilité est la base de docker, c'est typiquement l'utilisation que peut en faire un fournisseur de cloud.
- Distribuer une application : Docker en tant que "système de distribution d'une application" : n'ont plus à packager une application sous différents systèmes (deb, rpm, etc)
- Développer et tester une application, Docker permet :
  - de concevoir une architecture de test plus agile, chaque conteneur de test pouvant par exemple intégrer une brique de l'application (base de données, langages, composants, …), le développeur pourra tester sur la même machine plusieurs versions d'un même logiciel en inter changeant le conteneur correspondant.
  - de développer une application selon le concept d'architecture de micro-services avec pour chaque couche des conteneurs isolant les composants de l'application ;
  - faciliter le process de mise à jour de l'application : les images Docker sont versionnées et permettent une mise à jour simplifiée et maîtrisée. Le process de rollback est aussi simplifié : on redéploie la version précédente de l'image.
  - d'avoir un environnement de développement identique à l'environnement de production

# DOCKER HUB

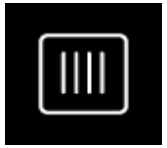
- C'est quoi ?
- https://hub.docker.com/
- Regroupe toutes les images docker disponibles
- Permet de retrouver toutes les images officielles
- Et celle de la communauté
- Tout le monde peut déposer son propre conteneur (inscription obligatoire)

# TECHNOLOGIES SOUS-JACENTES

- Développement
  - Docker est développé en langage GO
  - Le daemon Docker se base sur le pilote d'exécution RunC
- Control Group CGROUPS
  - Allocation de ressources RAM et CPU aux containers
- Namespaces
  - Isolation des containers
  - système de fichiers, nom d'hôte, utilisateurs, réseau, processus
- UNION FILE SYSTEM UnionFS
  - Stockage sur plusieurs couches en lecture seule
  - Support d'autres pilotes de stockage en fonction des distributions (AUFS, devicemapper, BTRFS, Overlay)
- Format des containers Docker
  - Namespaces + Cgroup + UnionFS
  - Ce format par défaut est appelé LIBCONTAINER

# NOTIONS DE BASE DOCKER

- Docker image (Dockerfile)
  - Équivalent à une application complète composée de différentes couches liées.



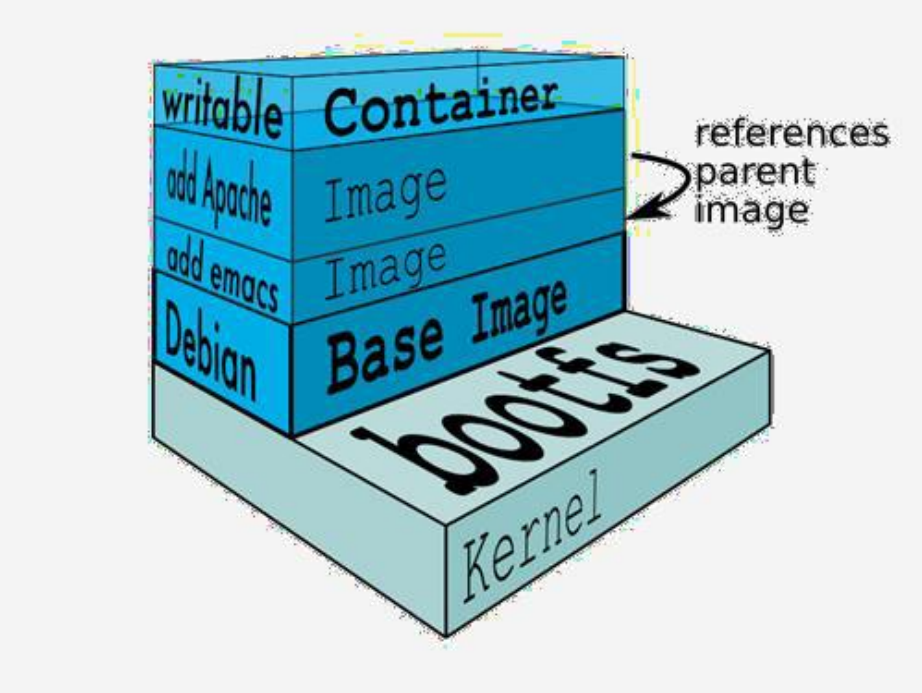
- Docker container (docker-compose)
  - Correspond à l'exécution d'une image Docker + une couche de lecture/écriture

- Docker engine
  - Crée, lance et exécute les containers Docker sur les plateformes physiques, virtuelles, en local, dans le datacenter ou chez un fournisseur de cloud

- Docker hub / Registry service
  - Service de stockage et de distribution des images dans le cloud ou sur un stockage d'entreprise

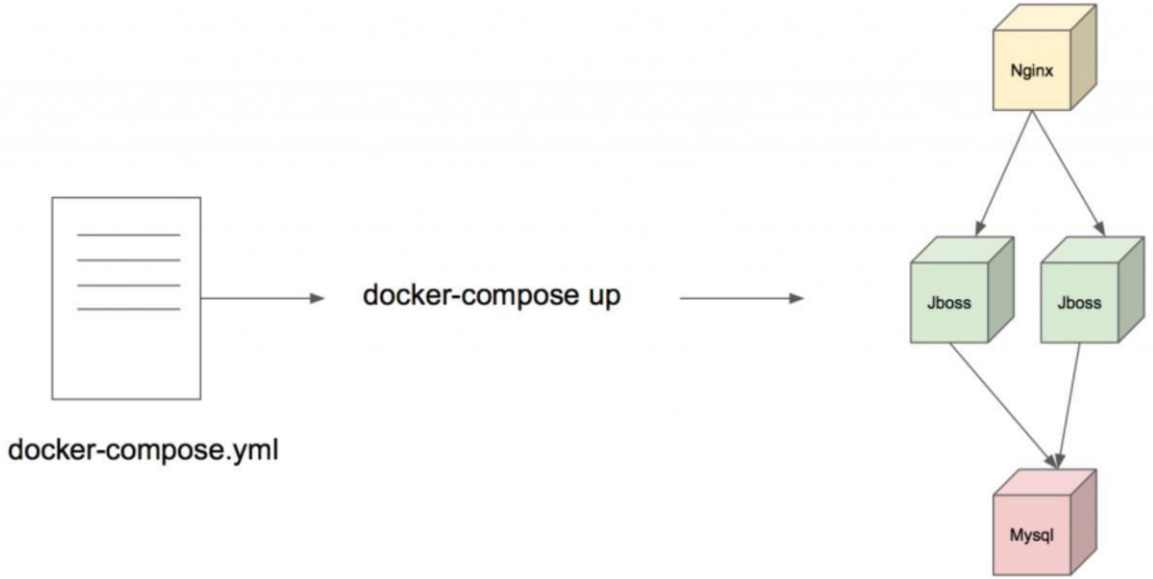
# SYSTÈME EN COUCHES

- Chaque couche est liée à la couche sous-jacente
- Copy-on-WRITE
- 127 couches possibles



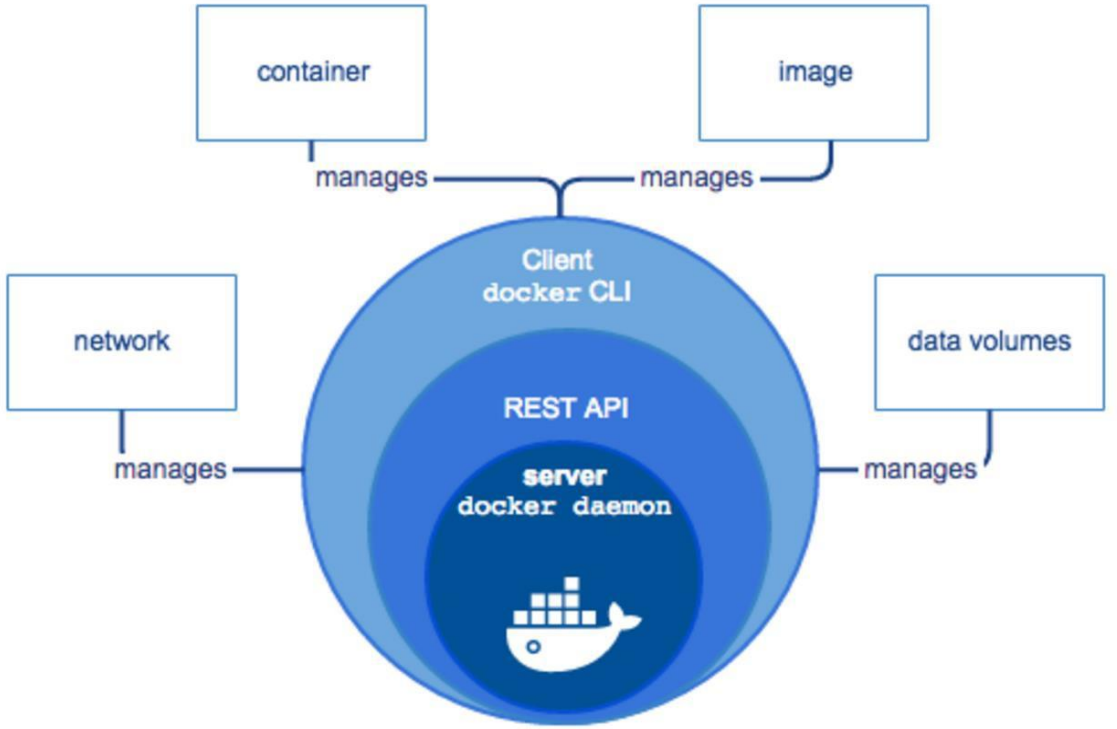
# DOCKER COMPOSE

- Liens de containers
  - Description YAML



# DOCKER ENGINE

- Moteur Docker
  - Docker Daemon
  - REST API pour piloter le daemon
  - Command Line Interface




ANDN
SERVICES

# QUEL OS POUR L'HÔTE ?

- Distributions minimales
  - Le choix idéal ! CoreOS, Fedora Atomic, RancherOS…
  - • + Sécurisé + Cluster/Réseau + Léger
  - - mature / stable ?
  - Production ?
- Distribution
  - Ubuntu / CentOS / Debian / Suse ….
  - Adapté à la production : en Entreprise & Cloud providers
  - + Support + Maitrisé + GUI
  - - sécurisé ? - cluster ?
- Poste de développement
  - Poste Linux : Ubuntu ou autre
  - Poste Mac/Windows :VirtualBox / Distri minimale Boot2Docker

ANDN SERVICES

# DOCKER HUB / REGISTRY

- Analogie
  - Docker Hub est équivalent à un repository Github mais pour stocker des images de containers.
- Docker Hub
  - Service de stockage d'images Docker en mode SaaS
    - La version publique est gratuite et accessible par tous
    - La version privée est payante et sécurisée
  - API : push/pull/search images
  - Intègre une GUI avec statistiques et gère des droits d'accès utilisateur (version privée)
- Docker registry
  - Repo de container à la Docker Hub
  - Installation Onpremise
  - API : push/pull/Search

# LES VOLUMES DANS DOCKER

- Lorsque vous supprimer le conteneur, toutes vos modifications sont perdus car les fichiers sont stockés par défaut dans ce conteneur
- Pour éviter ce problème, il est possible d'utiliser un volume qui sera situé sur la machine hôte
- Vous pourrez supprimer le conteneur, et le recréer en conservant vos fichiers modifiés

# FONCTIONNEMENT DES VOLUMES DE STOCKAGE EN CLUSTER

- Pour améliorer la performance et la portabilité d'une infrastructure, il est recommandé de ne pas stocker les données importantes directement sur la couche d'écriture des conteneurs.
- Pour cela il faut utiliser les volumes.
- Les volumes sont des espaces de stockages qui peuvent être mappés sur plusieurs conteneurs/services en même temps.
- Docker met à disposition 3 types de volumes :
  - Volume Data
  - Volume Bind
  - Volume tmpfs

# VOLUME DATA

- Description :
- C'est le type de stockage par défaut de docker.
- Elle offre de nombreux avantages :
  - Elle n'augmente pas la taille de la couche d'écriture du conteneur.
  - Cycle de vie indépendant du conteneur/service.
  - Peut être managé directement via la CLI Docker.
  - Fonctionne aussi bien sous Linux que Windows.
  - Facile à déplacer/sauvegarder.
  - Permet l'utilisation de « volumes driver » pour stocker les données sur des espaces de stockages distants de l'hôte (NFS, SAMBA, cloud, etc…).
- Les volumes docker de type data sont visibles dans le répertoire : /var/

# VOLUME BIND

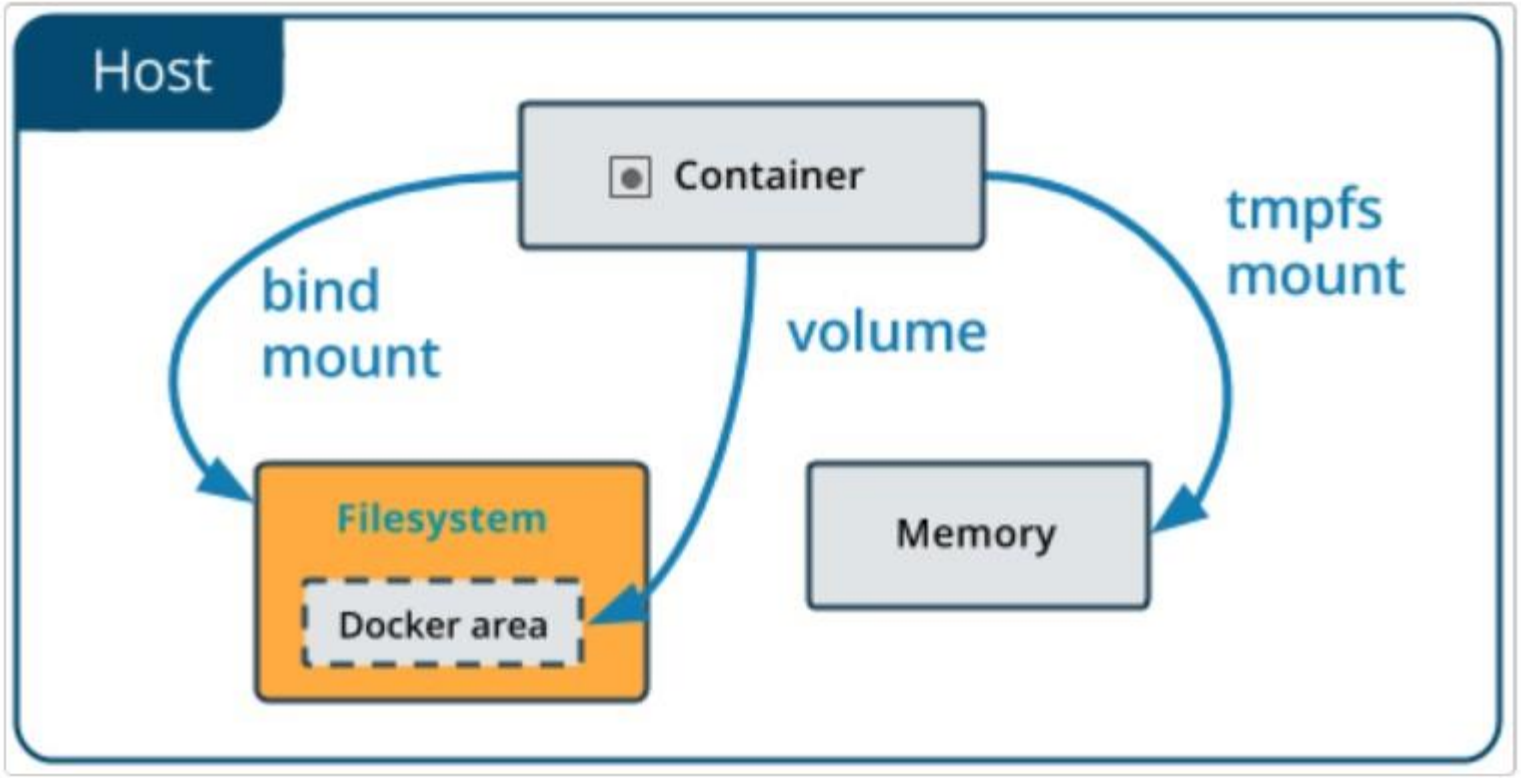
- Description :
  - Le volume de type « bind » consiste à monter un chemin du filesystem de l'hôte directement dans le conteneur.
  - Ce type de volume est historique à docker, mais il est préférable aujourd'hui d'utiliser les volumes de type « data ».
  - Ce type de volume ne peut pas être managé via la CLI de Docker et est donc plus « difficile » à administrer.
  - Si le chemin du volume « bind » n'existe pas à la création du conteneur, il sera automatiquement créé.
  - Ce type de volume est dépendant de la structure de répertoire du filesystem hôte.

# VOLUME TMPFS

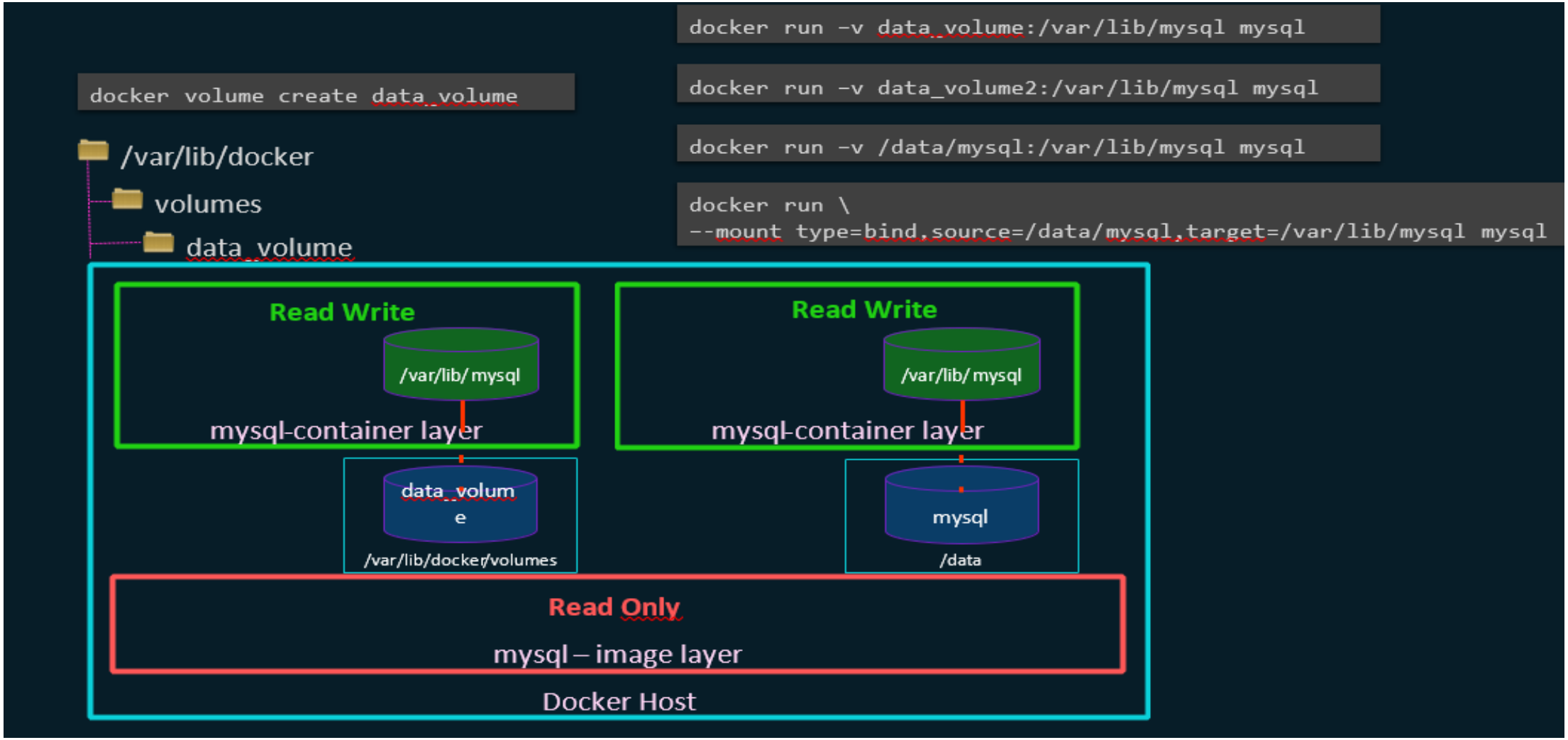
- Description :
  - Ce type de volume permet de stocker de la donnée volatile directement sur la RAM de l'hôte.
  - Ce type de volume dispose d'une performance très élevée.

- Contraintes :
  - Uniquement disponible sur les hôtes docker Linux.
  - Le cycle de vie du volume tmpfs est identique à celui du conteneur qui l'utilise.
  - Un volume tmpfs ne peut être partagé entre plusieurs  conteneurs/services.

# RÉSUMÉ


Host
Container
bind
mount
volume
tmpfs
mount
Filesystem
Docker area
Memory

# RÉSUMÉ


docker run -v data_volume:/var/lib/mysql mysql
docker volume create data_volume
docker run -v data_volume2:/var/lib/mysql mysql
/var/lib/docker
volumes
data_volume
docker run -v /data/mysql:/var/lib/mysql mysql
docker run \
--mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql
Read Write
/var/lib/ mysql
mysql-container layer
Read Write
/var/lib/ mysql
mysql-container layer
data_volum
e
/var/lib/docker/volumes
mysql
/data
Read Only
mysql – image layer
Docker Host

# VOLUMES PERSISTANT DANS DOCKER

- La commande suivante va permettre à docker de mapper le répertoire local /srv/data/html/ dans l'arborescence du container /usr/share/nginx/html/
  - Docker run -tid -p 8080 :80 -v /srv/data/html/:/usr/share/nginx/html/ --namenginx nginx:latest

- Autre manière : créer un volume local directement avec docker :
  - docker volume create nginxlocal
- Pour avoir un résumé de la configuration du volume, taper :
  - docker volume inspect nginxlocal
- Le volume est situé ici : /var/lib/docker/nginxlocal/_data
- Installer un conteneur nginx utilisant ce volume :
  - docker run -tid –name nginx -p 8080:80 –mountsource=nginxlocal,target=/usr/share/nginx/html nginx:latest

# VOLUMES PERSISTANT DANS DOCKER

- docker run -tid --name conteneur1 -v volume01:/usr/share/nginx/html/ -p 80:80 nginx
- docker volume inspect volume01
  - Vous aurez le Mountpoint qui vous dira ou est le volume
- docker inspect conteneur1
  - Vous retrouverez l'équivalence entre le conteneur et le volume
- On peut faire une sauvegarde tar.gz
  - Lancer un conteneur de sauvegarde léger type alpine
  - Monter le volume à sauvegarder volume01:/tmp/src
  - Monter le volume de destination /tmp/:/tmp/dest
  - Ajouter la commande de sauvegarde type tar -czvf <dest> <source>
  - Prévoir la suppression automatique
- Se rendre dans le conteneur01 avec docker exec puis aller dans le dossier qui contient index.html et le modifier pour tester
- Pour sauvegarder ensuite :
  - docker run --rm -v volume01:/tmp/src/ -v /tmp/:/tmp/dest -u root alpine tar -czvf /tmp/dest/backup.tar.gz /tmp/src/

# LE RÉSEAU DANS DOCKER

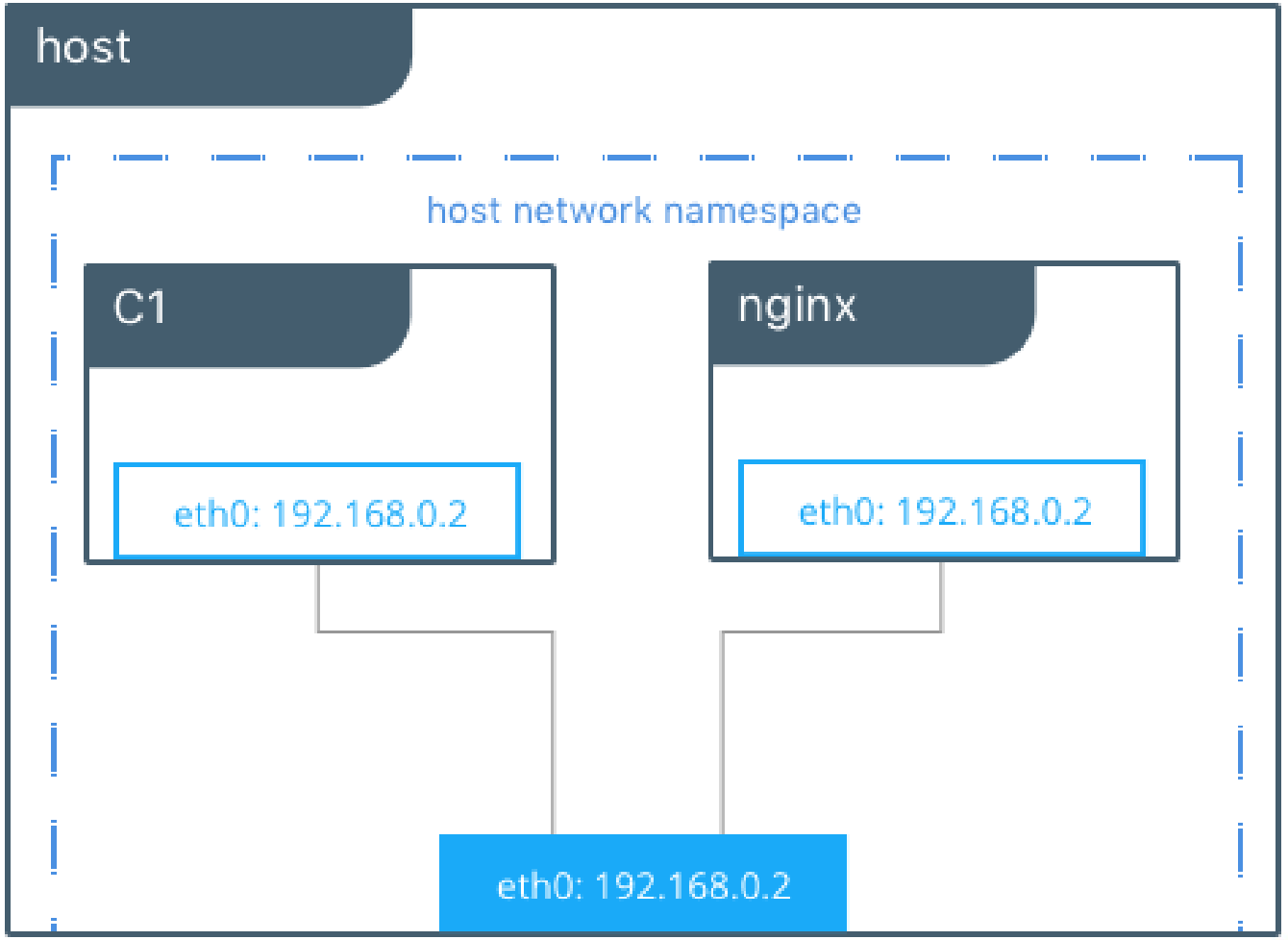
- Par défaut le réseau utilisé pour tous les conteneurs est le 172.17.0.0/16
- Docker assigne une ip automatiquement et dynamiquement dans ce range pour tout conteneur créé.
- Attention, si un conteneur redémarre, son ip peut changer.
- **Par défaut**, tous les conteneurs peuvent communiquer entre eux car ils sont créés dans le bridge .
- Il est possible de créer ses propres réseaux afin d'isoler les conteneurs entre eux.
- Les commandes utiles :
- docker network ls => liste les réseaux disponibles
- docker inspect monréseau => montre le détail du réseau nommé « monréseau »
- docker network create -d bridge –subnet 172.18.0.0/24 monréseau => créé le réseau 172.18.0.0/24 nommé monréseau

# LE RÉSEAU DANS DOCKER

- Il existe 8 types de réseau utilisable dans docker
  - bridge
  - host
  - The NONE Network
  - overlay
  - user defined bridge
  - macvlan (bridge)
  - macvlan (802.1q mode)
  - IPvlan (Layer 2)
  - IPvlan (Layer 3)

# HOST

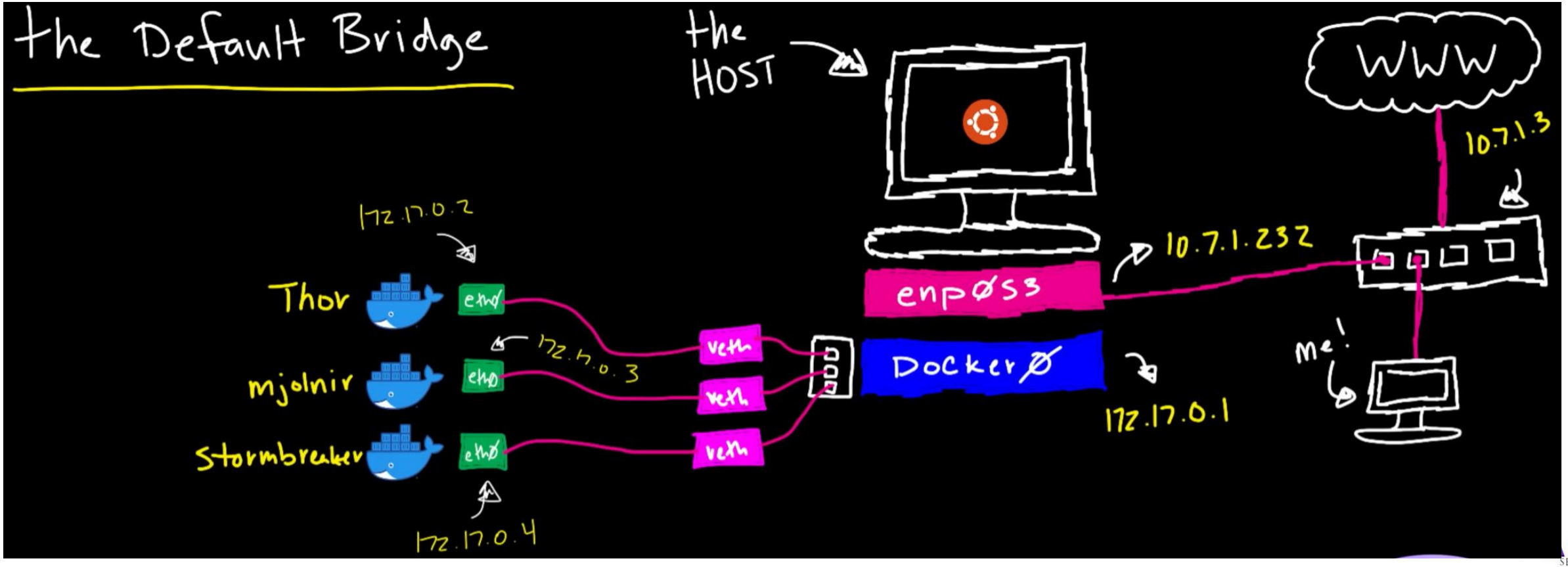
- Ce type de réseau permet aux conteneurs d'utiliser la même interface que l'hôte.
- Il supprime l'isolation réseau entre les conteneurs et seront par défaut accessibles de l'extérieur. Il prendra donc la IP que votre machine hôte.




ANDN SERVICES

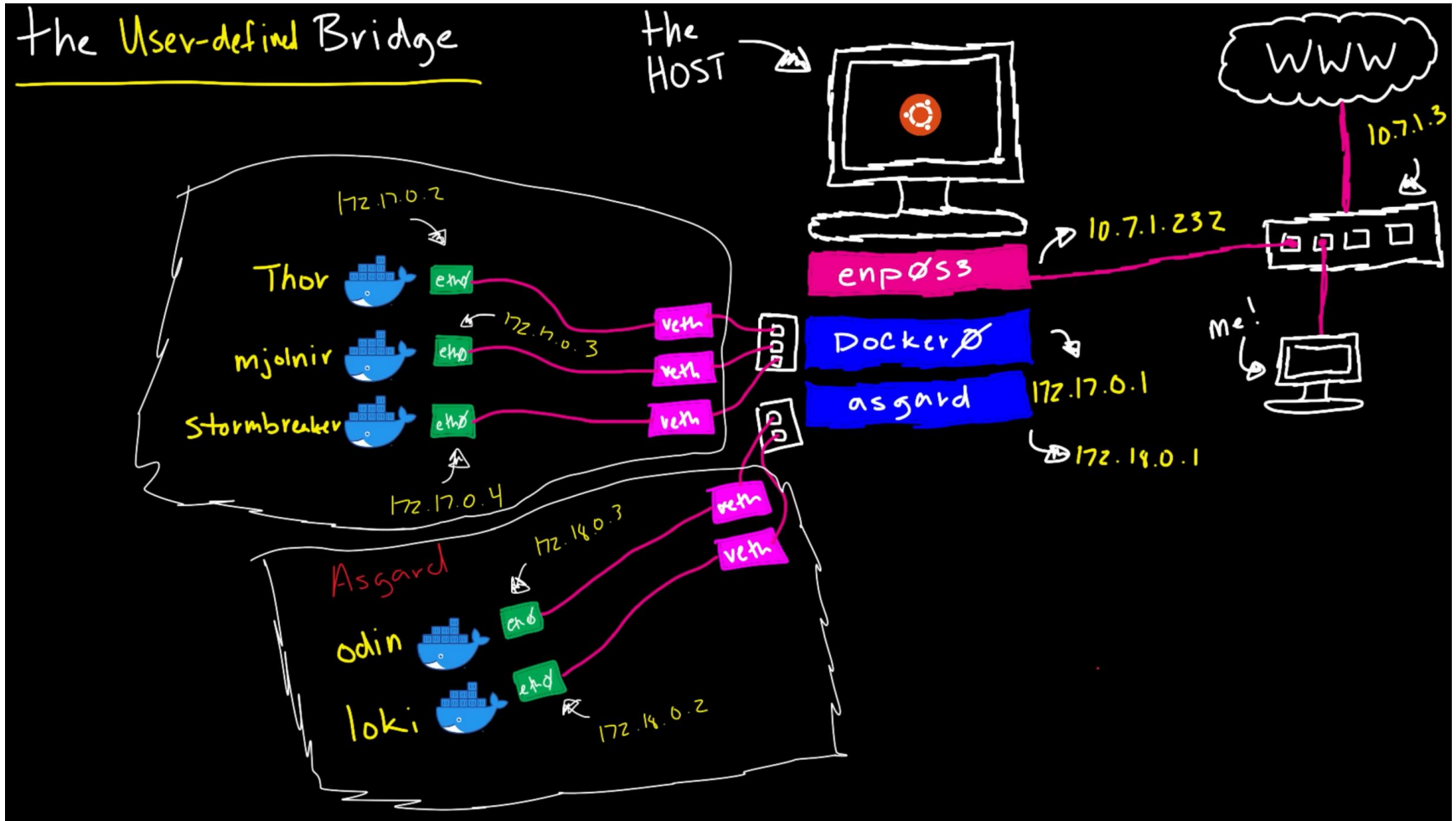
# DEFAULT BRIDGE

Sous réseau naté (NAT) sur le réseau normal (hote) - 172.17.0.0



# USER DEFINED BRIDGE

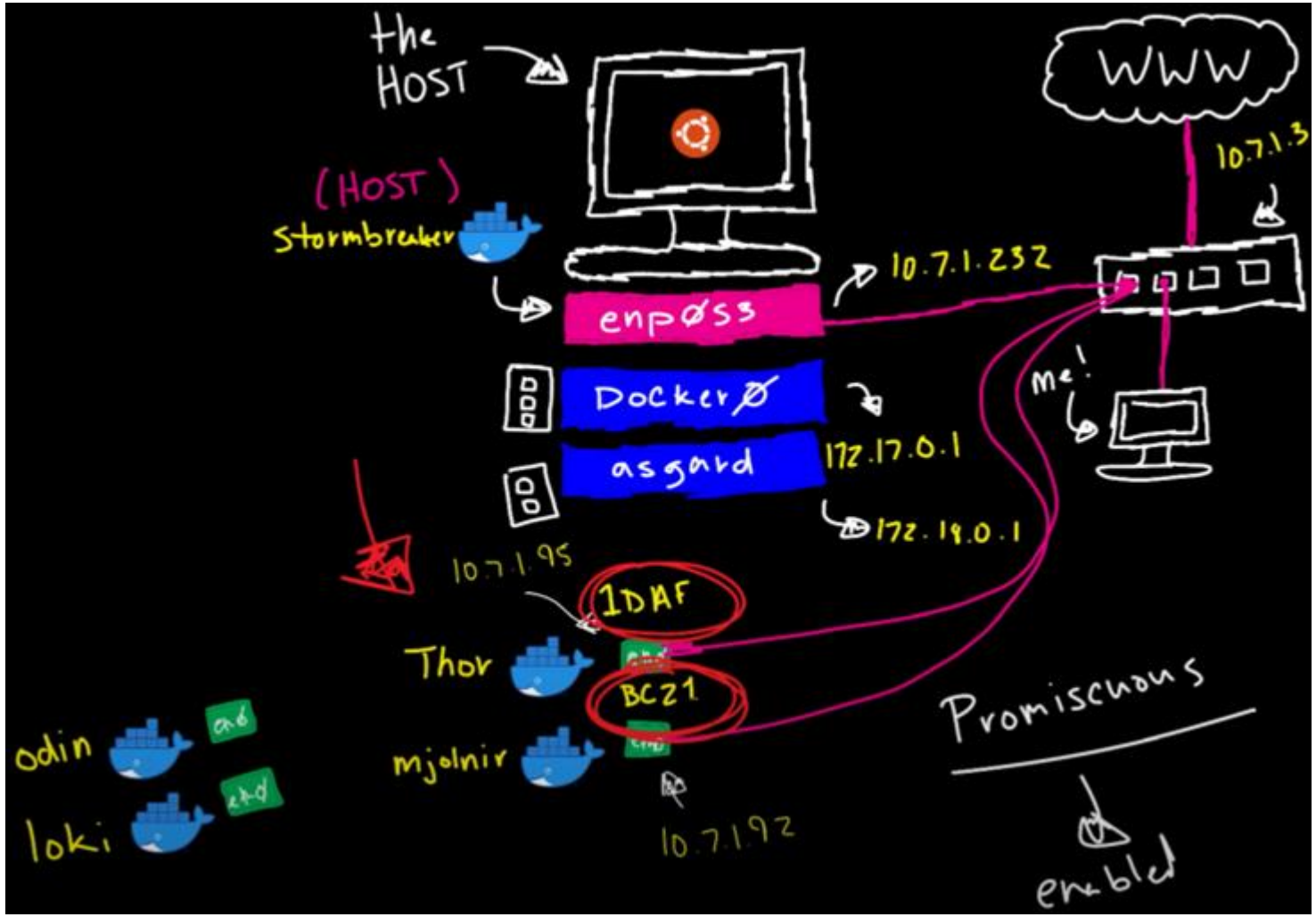
Autre réseau bridge, isolé les un des autres



# MACVLAN (BRIDGE)

- macvlan  (bridge mode)  - Connecté directement a notre réseau physique, un peu comme une VM - tu peux lui attribuer une adresse de ton réseau local
- Donner son subnet local et gateway (box, routeur) et son parent (l'interface réseau de l'hôte) pour mapper le reseau macvlan dessus
- Issue:  Cela distribue une mac address au container et une interface réseau ne peux pas avoir plusieurs mac sur une interface
- Il faut donc accepter le mode promiscuité
  - ip link set eth0 promisc on
  - et dans l'hyperviseur (workstation, esxi…)
- pas de dhcp - SPECIFIER UNE IP ADDRESS POUR LE CONTAINER POUR EVITER UN CONFLIT AVEC L'IP DE L'HÔTE (VOIR DOC POUR LES OPTIONS AU DEPLOY)

# MACVLAN (BRIDGE)

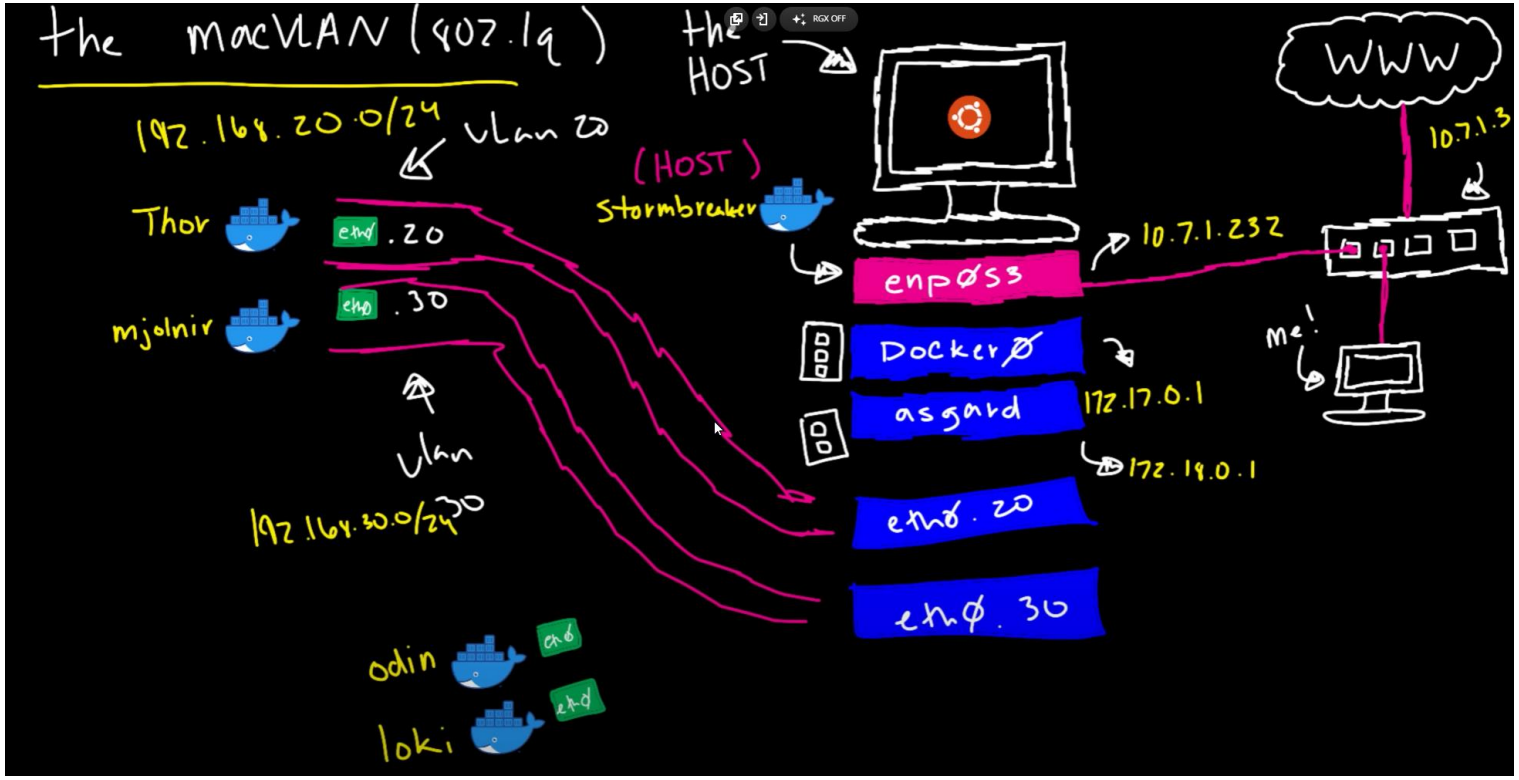


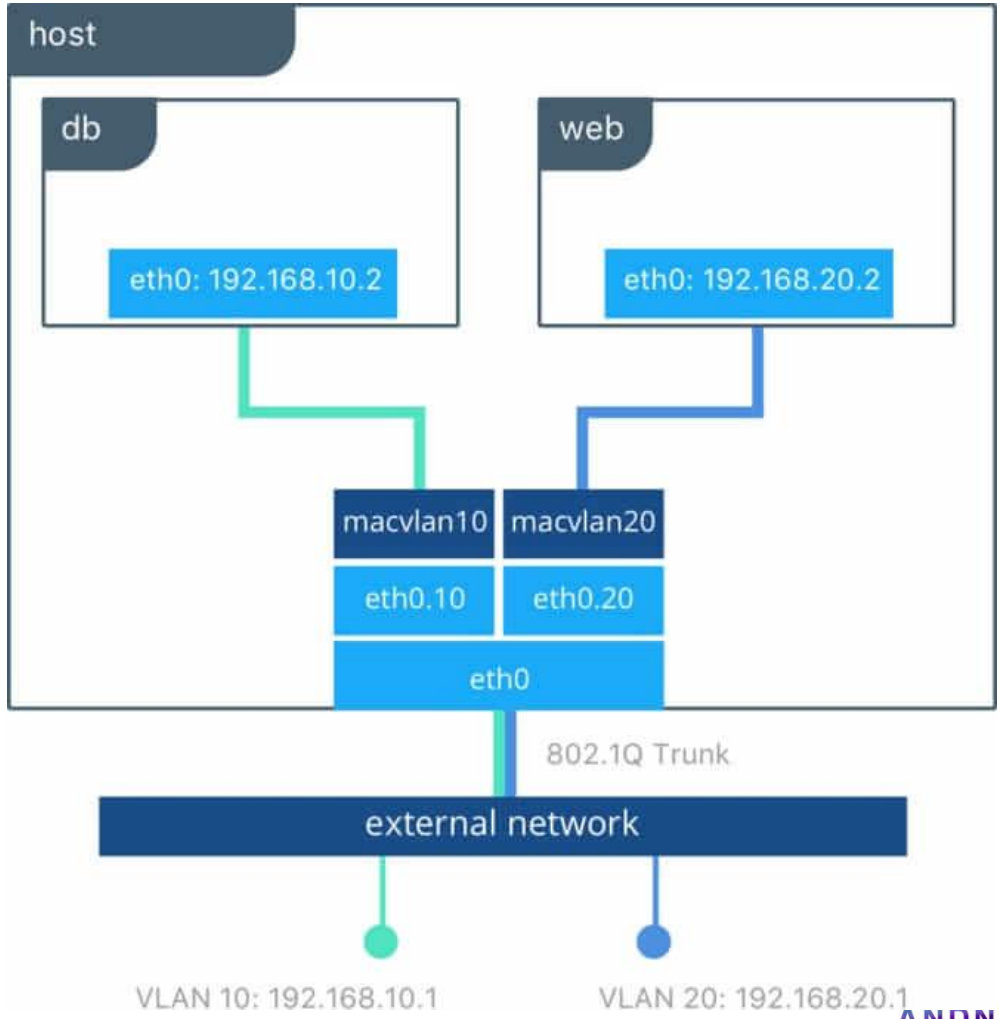

ANDN SERVICES

# MACVLAN (802.1Q (TRUNKED))

- macvlan  (802.1q mode) - Connecté directement a notre réseau physique, un peu comme une VM - tu peux lui attribuer une adresse de ton réseau local, similaire à macvlan mais avec une notion de VLAN et de sous interfaces.
- Réseau sous forme de vlan : - eth0.20 eth0.30 etc..
- Donner son subnet local et gateway (box) et son parent (l'interface réseau de l'hote avec .20 (num du vlan) eg: eth0.20) pour la mapper dessus
- Ce réseau crée au hôte des sous interfaces
  - docker network create -d macvlan \
  - --subnet 192.168.20.0/24
  - --gateway 192.168.20.1 \
  - -o parent=eth0.20 \
  - macvlan20

# MACVLAN (802.1Q (TRUNKED))



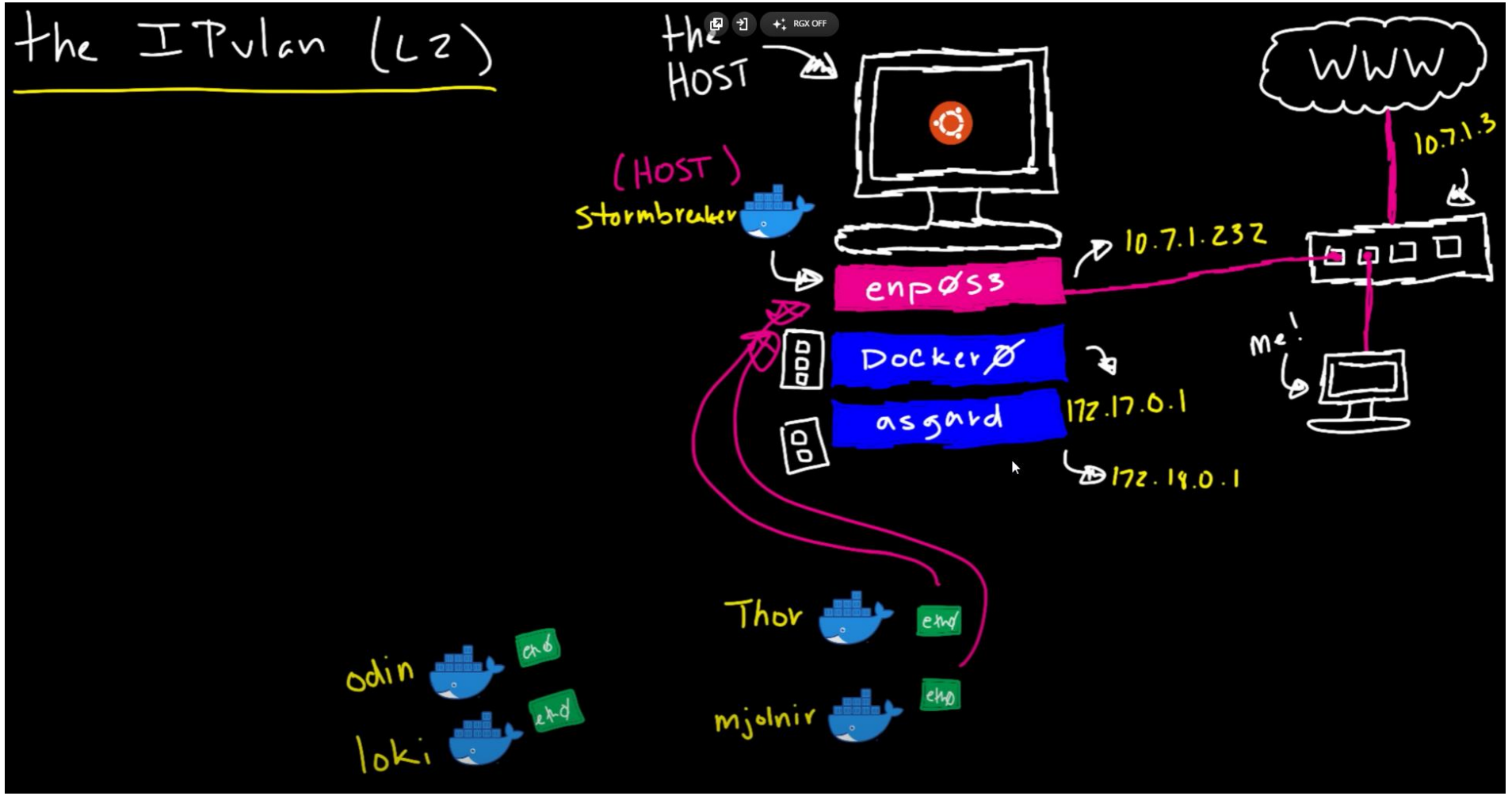



# IP VLAN (L2)

- Les container vont pouvoir avoir leur propre ip sur notre réseau local mais cette fois en partageant la mac address de l'hôte
  - docker network create -d ipvlan \
  - --subnet 192.168.1.0/24
  - --gateway 192.168.1.1 \
  - -o parent=eth0 \
  - new_ipvlan_network

- Puis run un container :
  - docker run -tid --rm --network new_ipvlan_network \
  - --ip 192.168.1.4/24
  - -name container01 busybox

# IPVLAN (L2)


the IPvlan (L2)
the HOST
WWW
10.7.1.3
(HOST)
Stormbreaker
10.7.1.232
enp0s3
me!
Docker
asgard
172.17.0.1
172.18.0.1
Thor
eth0
odin
eth0
loki
eth0
mjolnir
eth0

# IP VLAN (L3)

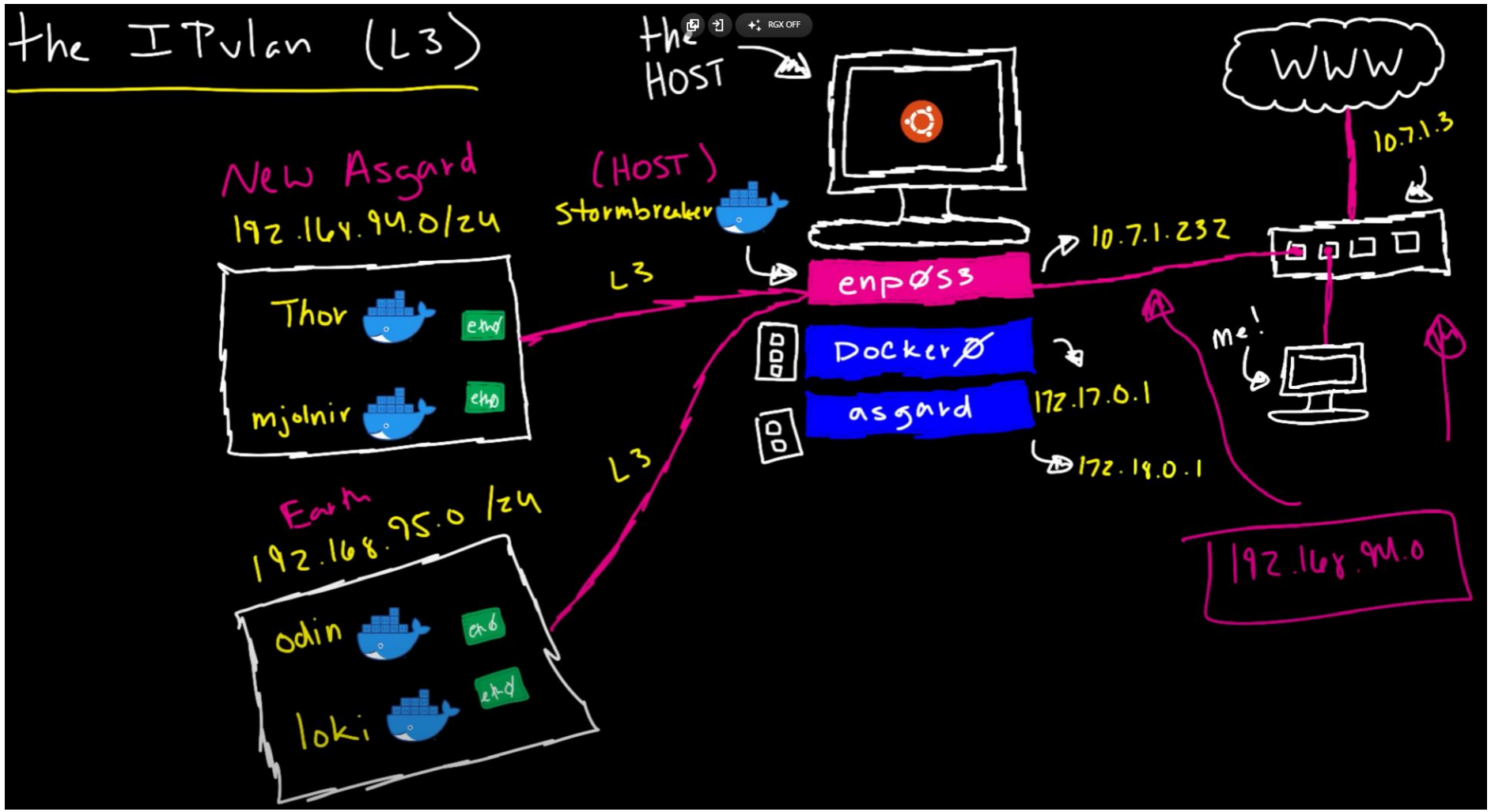
- Le container (le network docker) considère l'host (son ip) comme étant le routeur, donc l'ip du host est la gateway
- Nous aurons donc besoin de créer des routes statiques sur l'hôte et/ou routeur pour faire communiquer nos réseaux.
  - docker network create -d ipvlan \
  - --subnet 192.168.94.0/24 #le réseau que nous créons
  - -o parent=eth0  -o ipvlan_mode=l3 \ #pas de gateway car le parent est la gateway
  - --subnet 192.168.95.0/24 #on fait ça si on veux créer plusieurs réseau sur la même interface
  - new_ipvlan_network

```
docker run -tid --rm --network new_ipvlan_network \
  --ip 192.168.94.8/24
  -name container01 busybox

docker run -tid --rm --network new_ipvlan_network \
  --ip 192.168.95.9/24
  -name container02 busybox
```

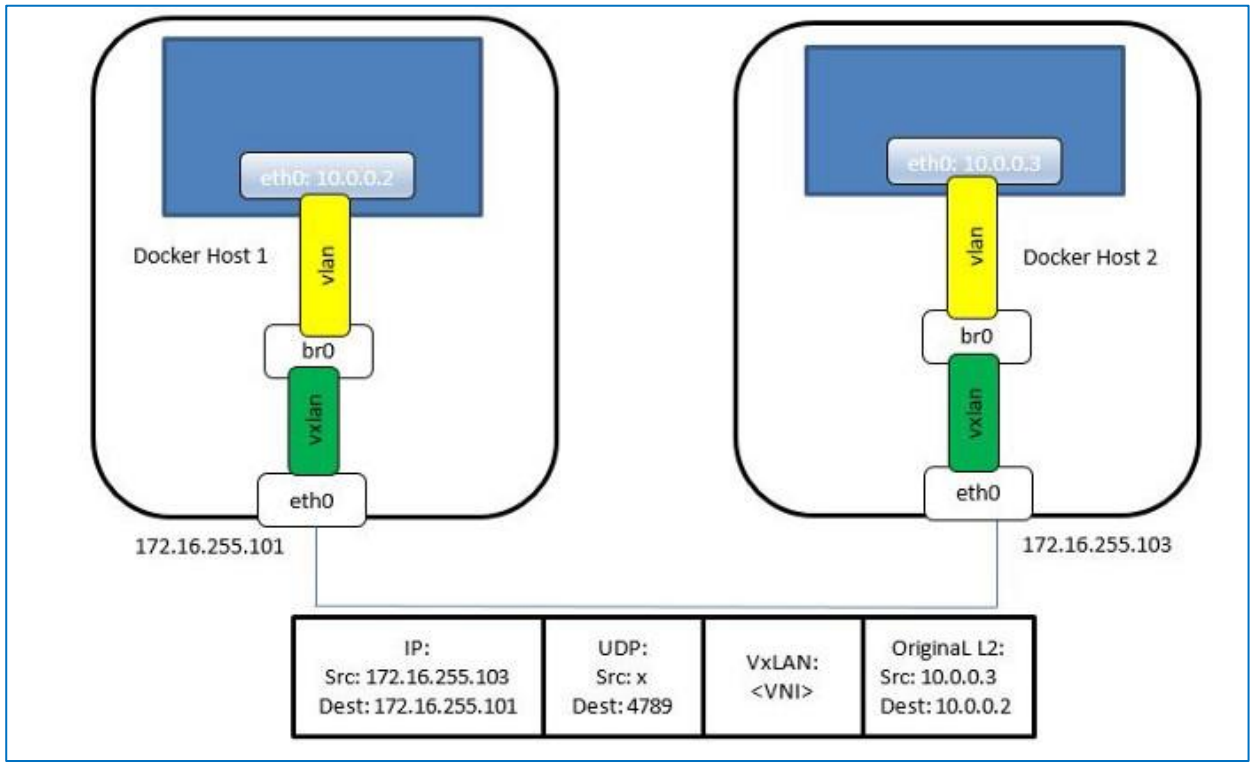
docker inspect new_ipvlan_network

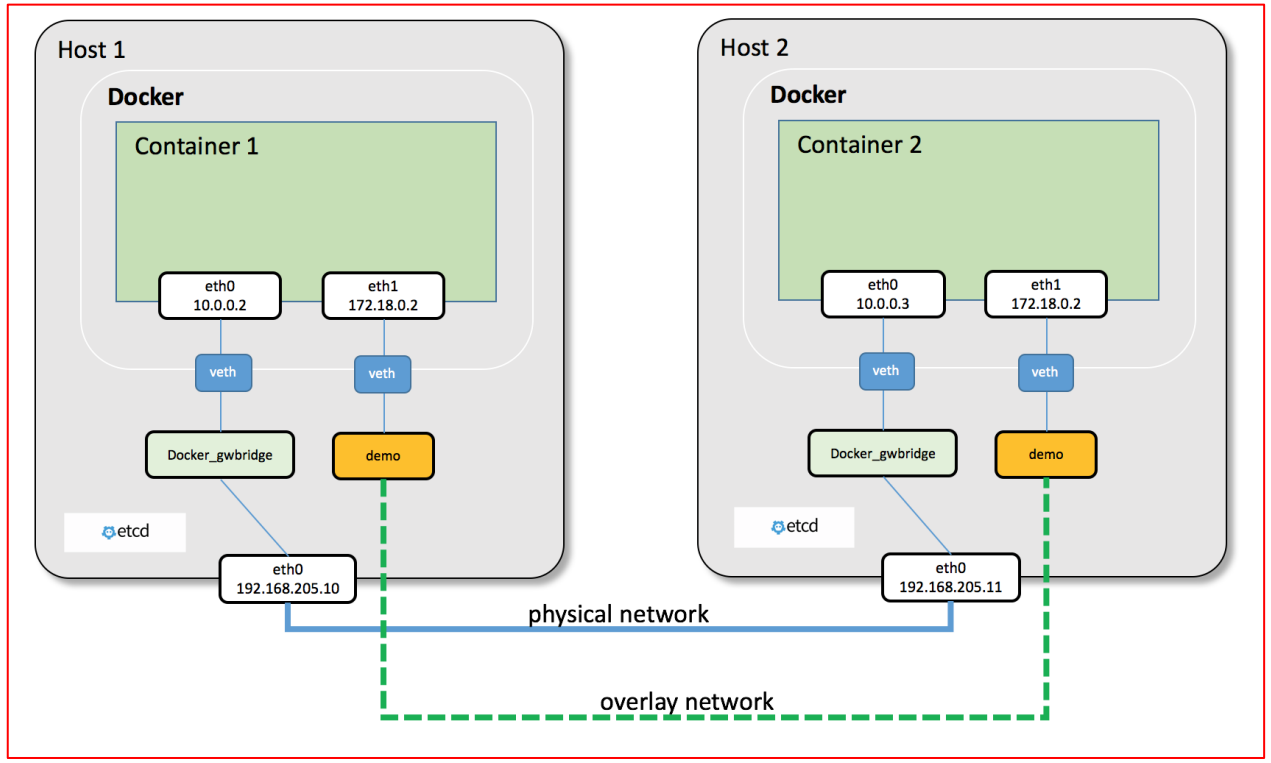
- les container ont pas internet, pas de routes
- peut pinger les autres container et par le nom aussi
- donc ajouter une vrai route sur le réseau si on veux faire communiquer à l'extérieur
- no broadcast traffic

# IPVLAN (L3)


the IPvlan (L3)
the HOST
WWW
10.7.1.3
New Asgard
192.164.94.0/24
(HOST)
Stormbreaker
L3
Thor
eth0
mjolnir
eth0
enp0s3
10.7.1.232
Docker
asgard
172.17.0.1
172.18.0.1
me!
L3
Earth
192.168.95.0 /24
odin
eth0
loki
eth0
192.168.94.0

# OVERLAY (UTILISÉ DANS L'ORCHESTRATION) #SWARM



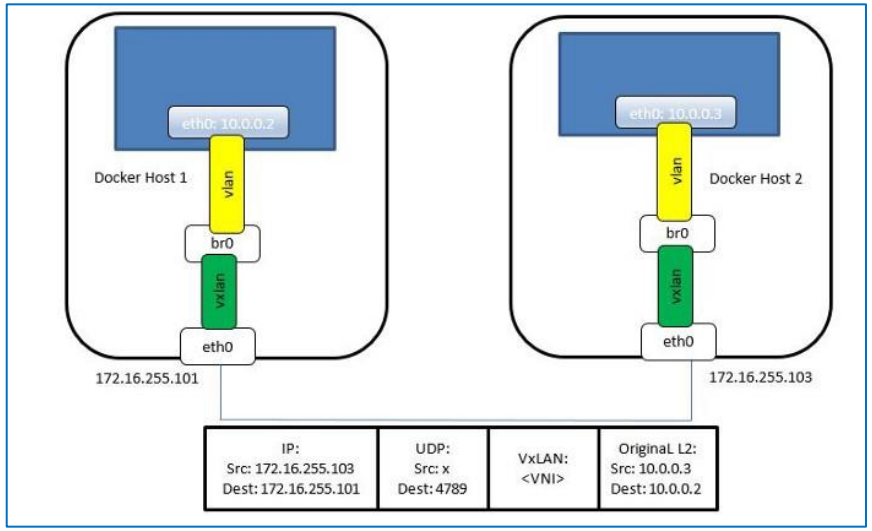

```
(Host) Docker01 : 172.16.255.101
Container1 : eth0: 10.0.0.2

(Host) Docker03 : 172.16.255.103
Container2 : eth0: 10.0.0.3
```

Dans un environnement docker avec 2 hôtes, chaque hôte a 1 conteneur à l'intérieur :

ANDN SERVICES

# OVERLAY (UTILISÉ DANS L'ORCHESTRATION) #SWARM

- Lorsque vous créez un réseau overlay, Docker crée un espace de noms pour le réseau sur l'hôte.
- Il créera ensuite un périphérique de pont (par exemple br0) et une interface vxlan.
- Lorsque vous créez un conteneur attaché à ce réseau, il sera attaché au bridge.
- Lorsque vous envoyez ensuite du trafic entre les conteneurs sur différents hôtes, le périphérique réseau sur le conteneur l'envoie au périphérique vxlan et au pont br0, jusqu'à l'hôte.
- L'hôte utilise l'en-tête vxlan pour acheminer le paquet vers le nœud de destination.




- La méthode recommandée pour créer un réseau superposé de nos jours se fait en 2 étapes :

1. Créez un réseau swarm entre les nœuds que vous souhaitez mettre en réseau.
2. Créez un overlay sur le dessus et les nodes de swarm se découvriront automatiquement.

Avant de commencer à créer un réseau superposé à l'aide de Swarm, assurez-vous que les ports suivants sont ouverts et accessibles sur tous les nœuds hôtes Docker :

- Port TCP 2377
- Port TCP et UDP 7946
- Port UDP 4789

ANDN SERVICES

# NONE (NULL) NETWORK

- Il n'y a pas d'IP, juste l'adresse du localhost.

# NETWORK COMMAND

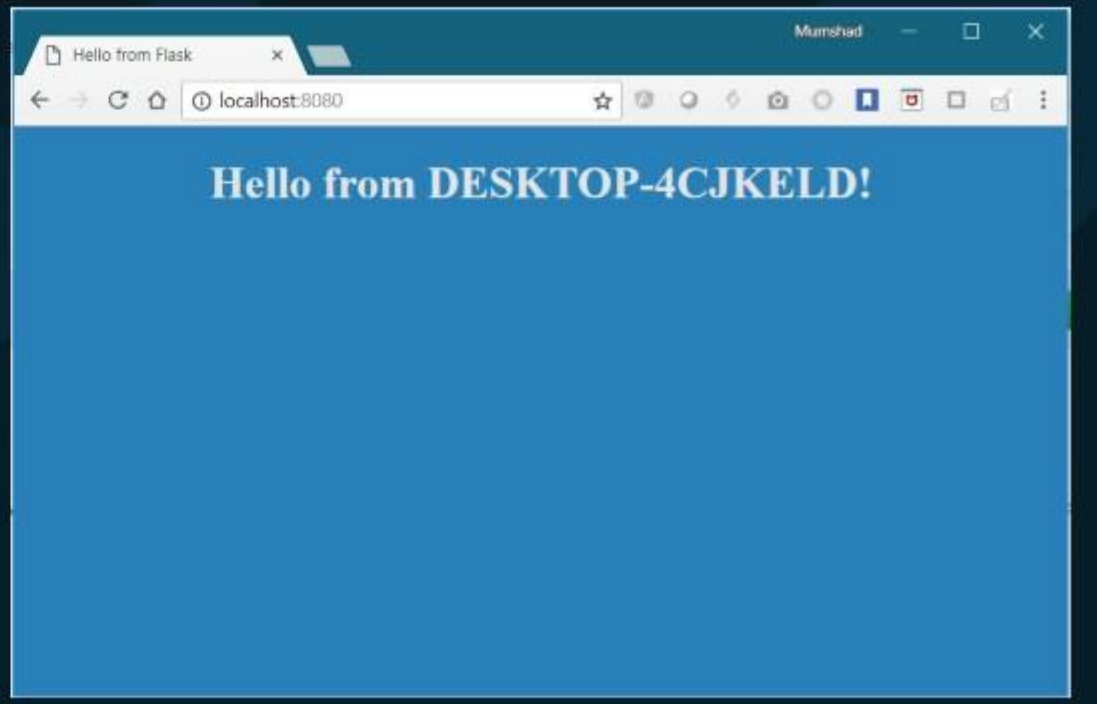
```
## Créer un réseau docker

docker network create --driver <DRIVER TYPE> <NETWORK NAME>

# Lister les réseaux docker

docker network ls

## Supprimer un ou plusieurs réseau(x) docker

docker network rm <NETWORK NAME>

## Récolter des informations sur un réseau docker

docker network inspect <NETWORK NAME>

    -v ou --verbose : mode verbose pour un meilleur diagnostique

## Supprimer tous les réseaux docker non inutilisés

docker network prune

    -f ou --force : forcer la suppression

## Connecter un conteneur à un réseau docker

docker network connect <NETWORK NAME> <CONTAINER NAME>

## Déconnecter un conteneur à réseau docker

docker network disconnect <NETWORK NAME> <CONTAINER NAME>

    -f ou --force : forcer la déconnexion

## Démarrer un conteneur et le connecter à un réseau docker

docker run --network <NETWORK NAME> <IMAGE NAME>
```

ANDN SERVICES

# VARIABLES D'ENVIRONNEMENT
# VARIABLES D'ENVIRONNEMENT EN PYTHON

app.py

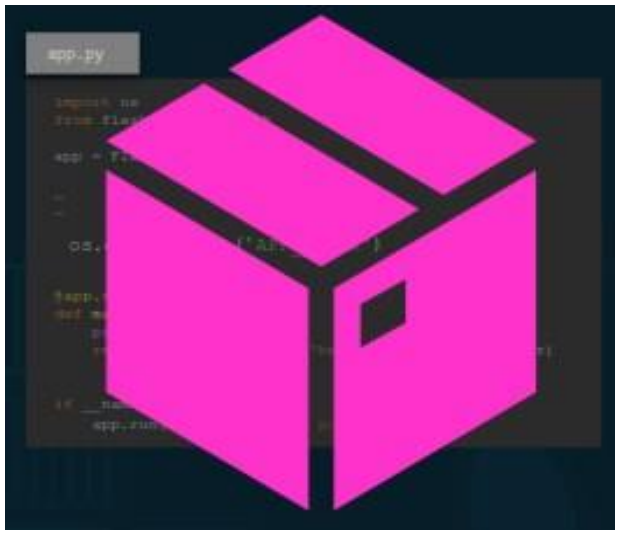
```python
import os
from flask import Flask

app = Flask(__name__)

...
...

color = os.environ.get('APP_COLOR')

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```




```
export APP_COLOR=blue; python app.py
```

ANDN SERVICES

# VARIABLES D'ENVIRONNEMENT
# VARIABLES D'ENVIRONNEMENT DANS DOCKER




```
docker run -e APP_COLOR=blue    simple-webapp-color
```

```
docker run  -e APP_COLOR=green simple-webapp-color
```

```
docker run  -e APP_COLOR=pink simple-webapp-color
```

# VARIABLES D'ENVIRONNEMENT
# INSPECTER LES VARIABLES D'ENVIRONNEMENT

```
docker inspect blissful_hopper
```

```
[
    {
        "Id": "35505f7810d17291261a43391d4b6c0846594d415ce4f4d0a6ffbf9cc5109048",
        "State": {
            "Status": "running",
            "Running": true,
        },

        "Mounts": [],
        "Config": {
            "Env": [
                "APP_COLOR=blue",
                "LANG=C.UTF-8",
                "GPG_KEY=0D96DF4D4110E5C43FBFB17F2D347EA6AA65421D",
                "PYTHON_VERSION=3.6.6",
                "PYTHON_PIP_VERSION=18.1"
            ],
            "Entrypoint": [
                "python",
                "app.py"
            ],
      }
]
```

ANDN SERVICES

# DOCKER

```
curl -fsSL https://get.docker.com | sh; >/dev/null
curl -sL "https://github.com/docker/compose/releases/download/v2.17.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

- Installer Docker :
  - Ajouter les repos suivant votre distribution
    - Exemple Debian: https://docs.docker.com/engine/install/debian/
  - Debian/ubuntu : sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
- Installer une image et la lancer :
  - docker run ***<image_name>***
  - docker run -di ***<image_name>*** (d : detach, I : interactive => permet de garder la main sur l'image)

- Lorsqu'on lance un « docker run », docker cherche d'abord l'image en local, si elle n'est pas disponible, il la télécharge depuis Docker Hub (par défaut). Une fois l'image présente en local, le conteneur est lancé.

# DOCKER COMMANDS
# RUN – PORT MAPPING

```
docker run kodekloud/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

http://172.17.0.2:5000 Internal IP

```
docker run -p 80:5000 kodekloud/simple-webapp
docker run -p 8000:5000 kodekloud/simple-webapp
docker run -p 8001:5000 kodekloud/simple-webapp
docker run -p 3306:3306 mysql
docker run -p 8306:3306 mysql
docker run -p 8306:3306 mysql
```


```
root@osboxes:/root # docker run -p 8306:3306 -e MYSQL_ROOT_PASSWORD=pass mysql
docker: Error response from daemon: driver failed programming external connectivity on endpoint boring_bhabha (
5079d342b7e8ee11c71d46): Bind for 0.0.0.0:8306 failed: port is already allocated.
```

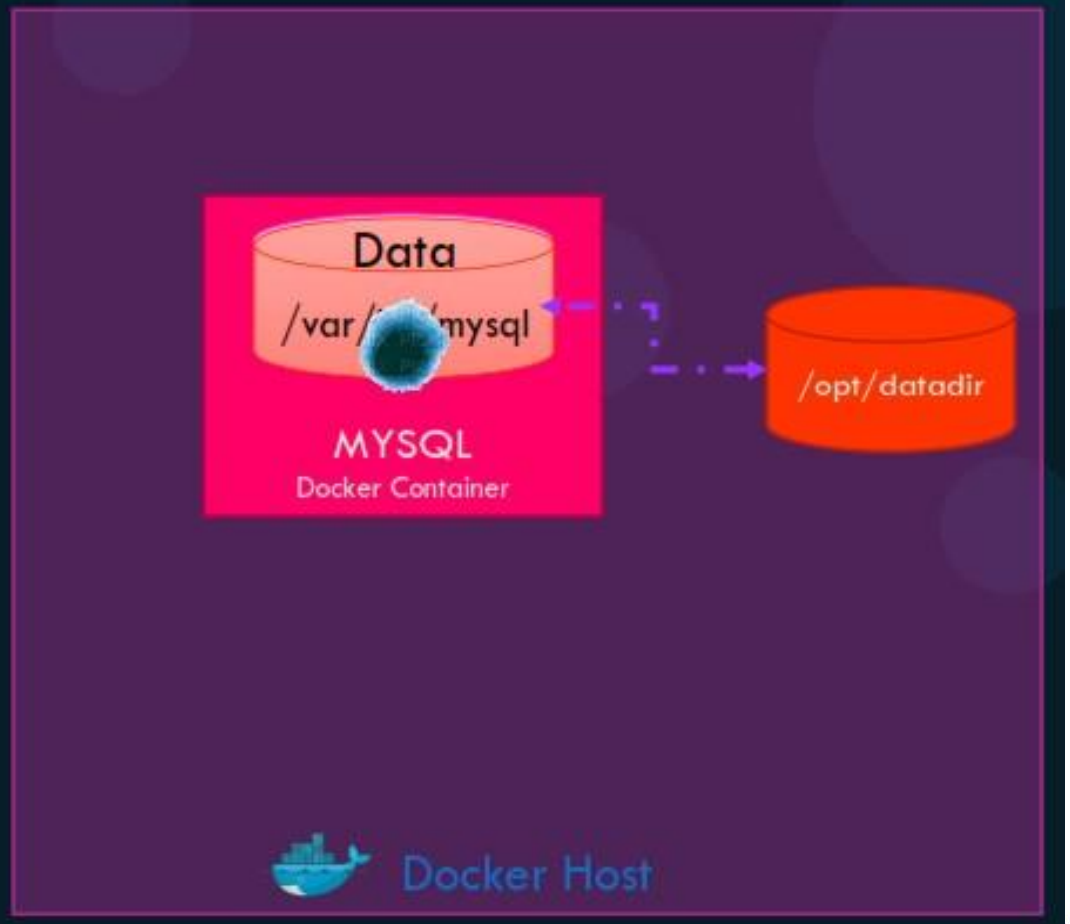
ANDN SERVICES

# DOCKER COMMANDS
# RUN – VOLUME MAPPING

```
docker run mysql
```

```
docker stop mysql
docker rm mysql
```

```
docker run -v /opt/datadir:/var/lib/mysql mysql
```



# DOCKER COMMANDS
# INSPECT CONTAINER

```
docker inspect blissful_hopper
```


```
[
    {
        "Id": "35505f7810d17291261a43391d4b6c0846594d415ce4f4d0a6ffbf9cc5109048",
        "Name": "/blissful_hopper",
        "Path": "python",
        "Args": [
            "app.py"
        ],
        "State": {
            "Status": "running",
            "Running": true,
        },

        "Mounts": [],
        "Config": {
           "Entrypoint": [
                "python",
                "app.py"
            ],
        },
        "NetworkSettings": {..}
      }
]
```


ANDN SERVICES

# DOCKER COMMANDS
# CONTAINER LOGS

```
docker logs blissful_hopper
 This is a sample web application that displays a colored background.
 A color can be specified in two ways.

 1. As a command line argument with --color as the argument. Accepts one of
red,green,blue,blue2,pink,darkblue
 2. As an Environment variable APP_COLOR. Accepts one of
red,green,blue,blue2,pink,darkblue
 3. If none of the above then a random color is picked from the above list.
 Note: Command line argument precedes over environment variable.


No command line argument or environment variable. Picking a Random Color =blue
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

ANDN SERVICES

# DOCKER

- docker ps : permet de lister les processus
- docker ps -a : permet de lister tous les processus docker, même ceux qui sont terminés
- docker exec -ti nomduconteneur sh : Se connecter au conteneur en shell

- Exemple avec un serveur web, nginx :
- docker run -tid -p 8080:80 --name monnginx nginx:latest
  - /usr/share/nginx/html/index.html
    - Modifier le contenu et tester
- Explication :
- Lancer le conteneur nginx, le nommer monnginx, le laisser fonctionner. Rediriger son port 80 sur le 8080 de notre machine locale

- Accéder au paramétrage du conteneur :
- docker inspect nom du conteneur (permet de voir l'adressage ip entre autre)

# QUELQUES COMMANDES DE BASE

| Commande | Explication |
|---|---|
| **docker image ls** | Lister les images existantes |
| **docker container ls -a** | Lister les conteneurs |
| **docker ps -a** | Lister les conteneurs |
| **docker run alpine:latest** | Lancer ou créer le conteneur alpine |
| **docker run -di --name alpinetest alpine:latest** | Lancer ou créer le conteneur alpine, le nommer alpinetest et le laisser fonctionner (mode Detach Interactive) |
| **docker run -di --name alpinetest alpine:latest sleep infinity** | Comme au dessus mais en plus on conserve un process qui tourne sur le conteneur |
| **docker run -tid -p 8080:80 --name monnginx nginx:latest** | Lancer le conteneur nginx, le nommer monnginx, le laisser fonctionner. Rediriger son port 80 sur le 8080 de notre machine locale |
| **docker run -tid --name conteneur2 --link conteneur1 alpine** | Lancer un conteneur alpine que l'on nomme conteneur2, le linker avec conteneur1 (conteneur2 pourra pinguer conteneur1) |
| **docker start moncontenеur** | Redémarrer un conteneur arrêté |
| **docker exec -ti monnginx sh** | Se connecter au conteneur nommé monnginx avec le shell et le bach |
| **docker stop <nom du conteneur>** | Arrêter un conteneur |
| **docker container kill $(docker ps -q)** | Tuer tous les conteneurs |
| **docker start <nom du conteneur>** | Redémarrer un conteneur |
| **docker rm -f <nom du conteneur>** | Supprimer un conteneur (-f permet de forcer, même si le conteneur tourne) |
| **docker rm $(docker ps -a -q)** | Supprimer tous les conteneurs |
| **docker inspect <nom du conteneur>** | Obtenir les infos sur un conteneur |
| **docker inspect -f "{{.NetworkSettings.IPAddress}}" <nom du conteneur>** | Obtenir l'ip d'un conteneur |

# CHEAT SHEET

| Commande | Description |
| --- | --- |
| docker ps | Visualiser les conteneurs actifs |
| docker ps -a | Visualiser tous les conteneurs |
| docker rm [container] | Supprimer un conteneur inactif |
| docker rm -f [container] | Forcer la suppression d'un conteneur actif |
| docker images | Lister les images existantes |
| docker rmi [image] | Supprimer une image docker |
| docker exec -t -i [container] /bin/bash | Exécuter des commandes dans un conteneur actif |
| docker inspect [container] | Inspecter la configuration d'un conteneur |
| docker build -t [image] . | Construire une image à partir d'un Dockerfile |
| docker history [image] | Visualiser l'ensemble des couches d'une image |
| docker logs --tail 5 [container] | Visualiser les logs d'un conteneur (les 5 dernières lignes) |

ANDN SERVICES

# CHEAT SHEET - REGISTRY

| Commande | Description |
| --- | --- |
| docker login | Se connecter au registry |
| docker search [name] | Rechercher une image |
| docker pull [image] | Récupérer une image |
| docker push [image] | Pouser une image du cache local au registry |
| docker tag [UUID] [image]:[tag] | Tagger une image |

# DOCKER-COMPOSE

- Compose est un outil pour définir et exécuter des applications multi-conteneurs, aussi appelé **"stack".**
- Les conteneurs étant idéaux pour des applications basées sur des micro services, il devient évident que rapidement on doit faire fasse à l'interconnexion de plusieurs conteneurs et à la gestion de multi-conteneurs.
- Un fichier docker-compose s'écrit avec le langage YAML.
- Il se présentera sous la forme **docker-compose.yml**
- Références officiels : https://docs.docker.com/compose/reference/
- Documentation d'installation : https://docs.docker.com/compose/install/#install-compose

# DOCKER-COMPOSE

- Arguments de docker-compose !
  - image qui permet de spécifier l'image source pour le conteneur ;
  - build qui permet de spécifier le Dockerfile source pour créer l'image du conteneur ;
  - volume qui permet de spécifier les points de montage entre le système hôte et les conteneurs ;
  - restart qui permet de définir le comportement du conteneur en cas d'arrêt du processus ;
  - environment qui permet de définir les variables d'environnement
  - depends_on qui permet de dire que le conteneur dépend d'un autre conteneur ;
  - ports qui permet de définir les ports disponibles entre la machine host et le conteneur.

# DOCKER-COMPOSE

- Pour utiliser docker-compose, vous devez d'abord créer un fichier yml, ici quelques exemples de docker-compose.yml

```yaml
version: "3.9" # optional since v1.27.0
services:
 web:
  build: .
  ports:
   - "8000:5000"
  volumes: - .:/code - logvolume01:/var/log
  depends_on:
   - redis
 redis:
  image: redis
volumes:
 logvolume01: {}
```

```yaml
services:
 webapp:
  image: examples/web
  ports:
   - "8000:8000"
  volumes:
   - "/data"
```

```yaml
version: "3"

services:
 web:
  image: "apache:${PHP_VERSION}"
  restart: 'always'
  depends_on:
   - mariadb
  ports:
   - '8080:80'
  links:
   - mariadb
 mariadb:
  image: "mariadb:${MARIADB_VERSION}"
  restart: 'always'
  volumes:
   - "/var/lib/mysql/data:${MARIADB_DATA_DIR}"
   - "/var/lib/mysql/logs:${MARIADB_LOG_DIR}"
   - /var/docker/mariadb/conf:/etc/mysql
  environment:
   MYSQL_ROOT_PASSWORD: "${MYSQL_ROOT_PASSWORD}"
   MYSQL_DATABASE: "${MYSQL_DATABASE}"
   MYSQL_USER: "${MYSQL_USER}"
   MYSQL_PASSWORD: "${MYSQL_PASSWORD}"
```

# DOCKER-COMPOSE

- Si par exemple vous voulez installer un wordpress avec docker-compose (en partant du principe que vous n'utilisez pas déjà l'image existante sur le hub ;)
- Vous devrez installer une base de donnée et un serveur web.
- docker-compose up -d
- Puis http://127.0.0.1:8000 pour accéder à votre wordpress

```
version: '3'
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress

volumes:
  db_data: {}
```

ANDN SERVICES

# DOCKER COMPOSE - CHEAT SHEET

| Commande | Description |
| --- | --- |
| docker-compose up -d | Démarre un ensemble de conteneurs en arrière-plan |
| docker-compose down | Stoppe un ensemble de conteneurs |
| docker-compose exec [service] [command] | Exécute une commande au sein d'un service |
| docker-compose ps | Affiche les stacks |
| docker-compose logs -f --tail 5 | Afficher les logs du stack |
| docker-compose stop | Arrêter un stack |
| docker-compose config | Valider la syntaxe de votre fichier docker-compose |

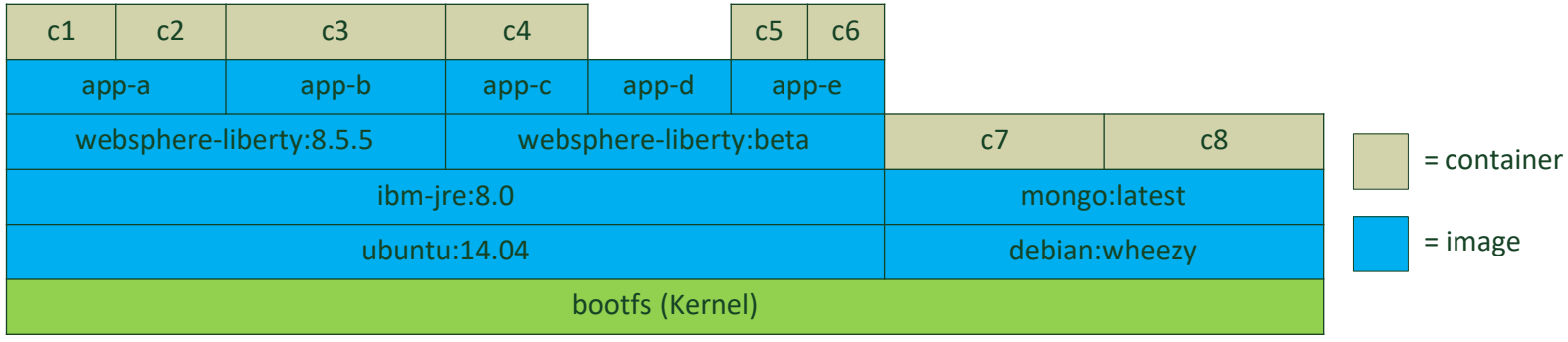
ANDN SERVICES

# TP DOCKER-COMPOSE

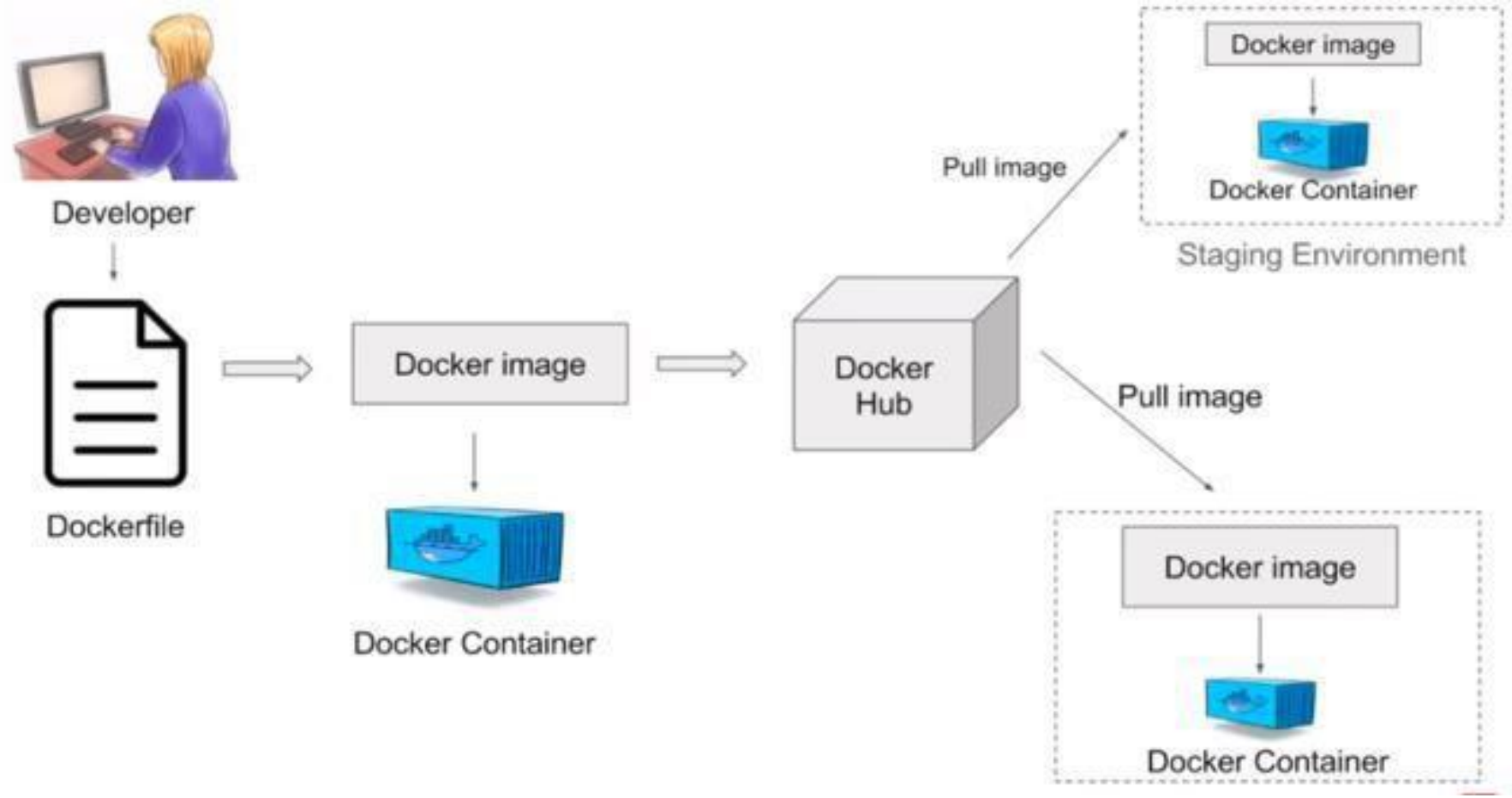
- Monter une stack LAMP avec docker compose.
- LAMP : LINUX APACHE MYSQL PHP
- help :
  - https://linuxconfig.org/how-to-create-a-docker-based-lamp-stack-using-docker-compose-on-ubuntu-18-04-bionic-beaver-linux
  - https://connect.ed-diamond.com/Linux-Pratique/lp-111/creer-son-environnement-de-developpement-lamp-grace-a-docker-compose
  - https://github.com/sprintcube/docker-compose-lamp
  - https://cours.brosseau.ovh/tp/docker/docker_compose.html

# DOCKER IMAGES

- Rappel :
  - container = plusieurs images/couches en lecture seul + couche Copy-On-Write + RUN
- IMAGES
  - Chaque image est en lecture seule
  - Partage des images entre containers sur le host
  - Cache des couches de base en mémoire

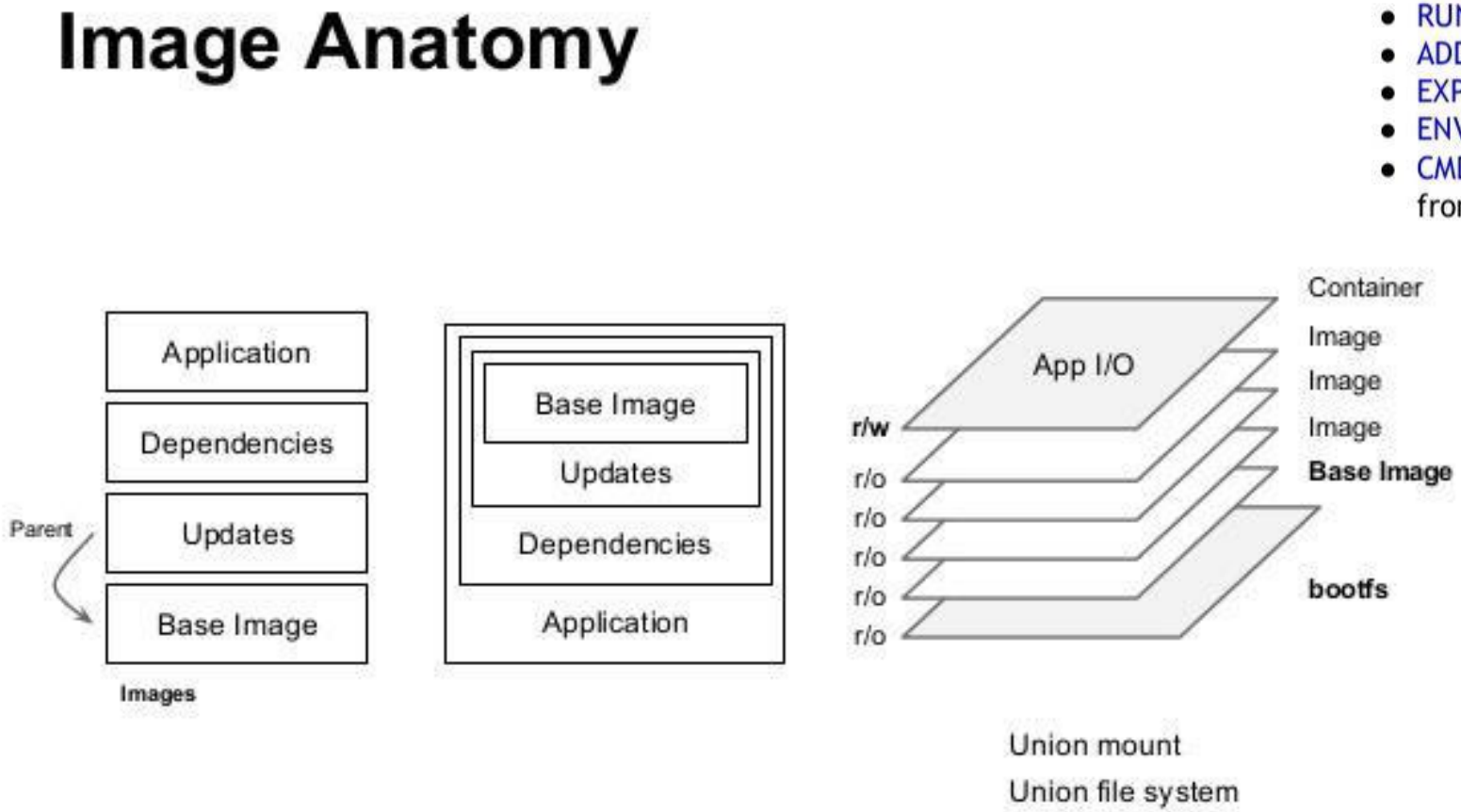


# DOCKER IMAGES - WORKFLOW D'UNE IMAGE



# DOCKER IMAGES - WORKFLOW D'UNE IMAGE

- FROM : Specify the base image
- MAINTAINER : Specify the image maintainer
- RUN : Run a command
- ADD : Add a file or directory
- EXPOSE : expose ports to be accessed
- ENV : Create an environment variable
- CMD : What process to run when launching a container from this image.




- **Lecture seule**
- **Réutilisation**
- **Couche (Layer)**

ANDN SERVICES

# DOCKERFILE - CRÉER UNE IMAGE

- Un dockerfile est un fichier de configuration qui à pour objectif de créer une image.
- Nous y retrouverons une séquence d'instruction
  - FROM : Sur quelle image on se base
  - ENV : variables d'environnement
  - EXPOSE : exposer le port du conteneur
  - VOLUME : Définition des volumes
  - COPY : cp entre host et conteneur
  - ENTRYPOINT : processus maitre, car l'idée d'un conteneur est d'avoir un seul process qui tourne

# DOCKERFILE - CRÉER UNE IMAGE

- L' intérêt d'un dockefile est de relancer une création d'image à tout moment
- Meilleure visibilité sur ce qui est fait
- Partage facile et possibilité de gitter
- Script d'édition de docker file (variables...)
- Ne pas se poser de question lors du docker run du conteneur
- Création images prod // dev - CI // CD (continuous integration/deployment)

# DOCKERFILE - CRÉER UNE IMAGE

- FROM ubuntu:latest
- MAINTAINER vincent
- RUN apt-get update \
- && apt-get install -y vim git \
- && apt-get clean \
- && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

- docker build -t nomimage:version .
  - -t nom de l'image
  - Ne pas oublier le "." qui symbolise le dockerfile

# DOCKER FILE - CRÉER SON IMAGE

Dockerfile

```
FROM Ubuntu

RUN apt-get update
RUN apt-get install python

RUN pip install flask
RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

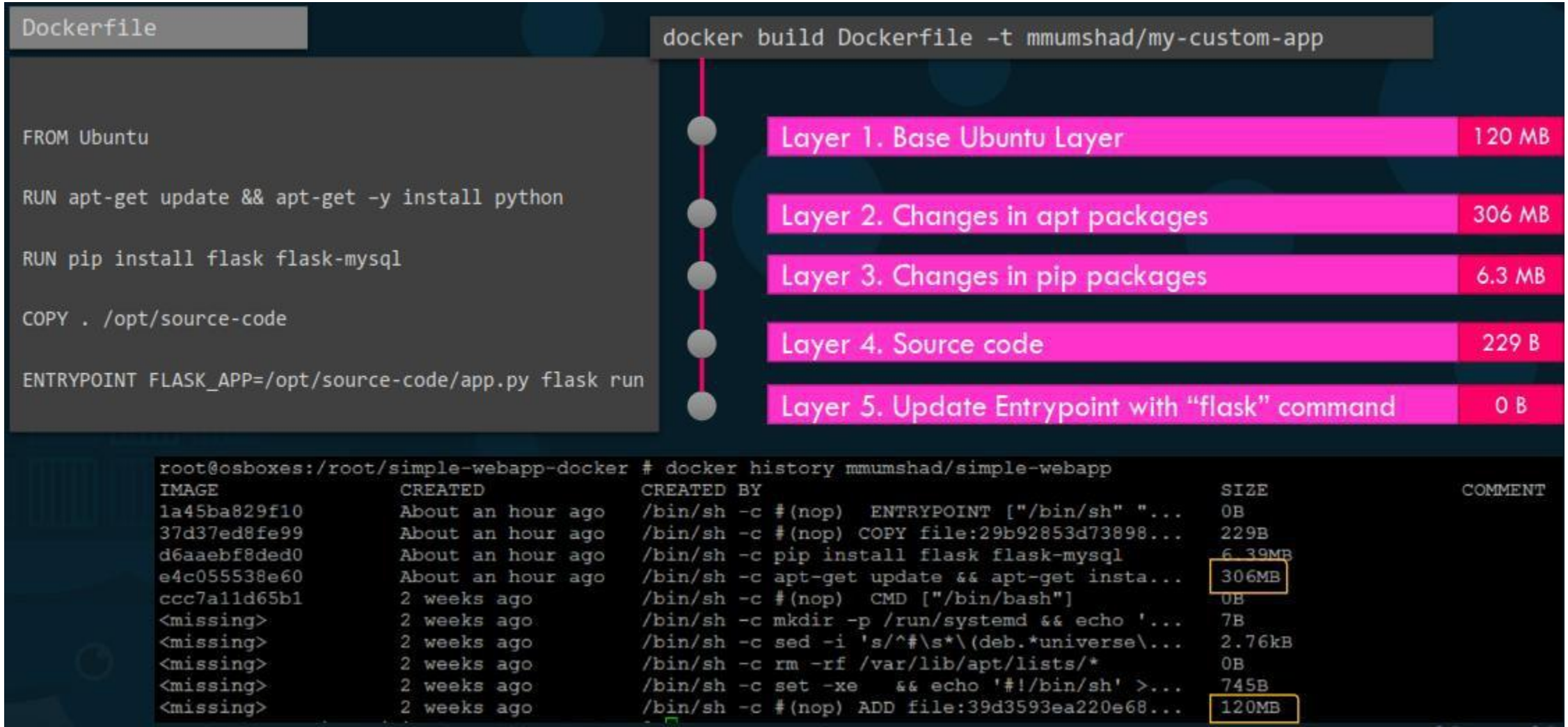
1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using “flask” command

```
docker build Dockerfile -t mmumshad/my-custom-app
```

```
docker push mmumshad/my-custom-app
```


ANDN
SERVICES

# COMMENT CRÉER SA PROPRE IMAGE DOCKER DOCKERFILE :ARCHITECTURE EN COUCHES



# COMMENT CRÉER SA PROPRE IMAGE DOCKER
# DOCKERBUILD : OUTPUT

```
root@osboxes:/root/simple-webapp-docker # docker build .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM ubuntu
 ---> ccc7a11d65b1
Step 2/5 : RUN apt-get update && apt-get install -y python python-setuptools python-dev
 ---> Running in a7840dbfad17
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:4 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [46.3 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:6 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [440 kB]
Step 3/5 : RUN pip install flask flask-mysql
 ---> Running in a4a6c9190ba3
Collecting flask
  Downloading Flask-0.12.2-py2.py3-none-any.whl (83kB)
Collecting flask-mysql
  Downloading Flask_MySQL-1.4.0-py2.py3-none-any.whl
Removing intermediate container a4a6c9190ba3
Step 4/5 : COPY app.py /opt/
 ---> e7cdab17e782
Removing intermediate container faaaaf63c512
Step 5/5 : ENTRYPOINT FLASK_APP=/opt/app.py flask run --host=0.0.0.0
 ---> Running in d452c574a8bb
 ---> 9f27c36920bc
Removing intermediate container d452c574a8bb
Successfully built 9f27c36920bc
```

# DOCKERFILE - INSTRUCTIONS

1. FROM
    1. Image parente
2. LABEL
    1. Ajout de métadonnées
3. RUN
    1. Commande(s) utilisée(s) pour construire l'image
4. ENV
    1. Variable d'environnement
5. CMD
    1. Exécuter une commande au démarrage du conteneur
6. EXPOSE
    1. Port(s) écouté(s) par le conteneur
7. ARG
    1. Variables passées comme paramètres à la construction de l'image
8. ADD
    1. Ajoute un fichier dans l'image
9. COPY
    1. Ajoute un fichier dans l'image
10. ENTRYPOINT
    1. Exécuter une commande au démarrage du conteneur
11. WORKDIR
    1. Permet de changer le chemin courant (cd tier)
12. VOLUME
    1. Crée un point de montage
13. USER
    1. Nom d'utilisateur ou UID à utiliser
14. ONBUILD
    1. Instructions exécutées lors de la construction d'images enfants

# COPY VS ADD

- COPY et ADD sont les deux instructions qui servent à des fins similaires. **Ils vous permettent de copier des fichiers d'un emplacement spécifique dans une image Docker.**

- **COPY** prend un **src** et une **destination** . Il vous permet uniquement de **copier dans un fichier ou un répertoire local de votre hôte** (la machine créant l'image Docker) dans l'image Docker elle-même.

- **ADD** vous permet de le faire aussi, **mais il prend également en charge 2 autres sources**. Tout d'abord, vous **pouvez utiliser une URL** au lieu d'un fichier/répertoire local. Deuxièmement, vous pouvez **extraire un fichier tar de la source directement dans la destination.**

- Dans la plupart des cas, si vous utilisez une URL, vous **téléchargez un fichier zip et utilisez ensuite la RUN commande pour l'extraire.**

- Cependant, vous pouvez tout aussi bien **utiliser RUN avec curl au lieu d' ADD** afin de tout enchaîner **en une seule RUN** pour créer une image Docker plus petite.

- Un cas d'utilisation valide pour **ADD est lorsque vous souhaitez extraire un fichier tar local dans un répertoire spécifique de votre image** Docker. C'est exactement ce que fait l'image Alpine ADD rootfs.tar.gz / .

- Si vous copiez des fichiers locaux sur votre image Docker, utilisez toujours COPY car c'est plus explicite.

ANDN SERVICES

# DOCKERFILE : ENTRYPOINT VS CMD

- Entrypoint est le processus principal sur lequel va tourner le conteneur qui va permettre de lancer l'image.
- CMD est la commande par défaut (arguments/paramètres par défaut)
- CMD seule remplace l'entrypoint

```
FROM alpine
MAINTAINER Vincent
CMD ["ping", "--help"]
```

- Mais ne prend pas les arguments passés en CLI.
- Si docker run --rm demo google.fr
  - Vous aurez un message d'erreur car le CMD ne fera la distinction entre paramètre et process.

```
FROM alpine
MAINTAINER Vincent
ENTRYPOINT ["ping", "--help"]
```

- docker build -t demo -f Dockerfile .
- docker run --rm demo
- La commande passe puis en passant docker run --rm demo google.fr, vous ne chargerez pas le container.

# DOCKERFILE : ENTRYPOINT VS CMD

FROM alpine

MAINTAINER Vincent

CMD ["--help"]

ENTRYPOINT ["ping"]

- docker build -t demo -f Dockerfile .
- docker run --rm demo google.fr
- La commande passe puis en passant docker run --rm demo google.fr.
- On précise le paramètre par défaut dans le CMD et le process dans le entrypoint !

# TP : CRÉEZ VOTRE PREMIÈRE IMAGE

- Il s'agit ici de créer une image docker afin de conteneuriser une application web statique,
- Télécharger les fichiers de l'application à l'aide de git clone « https://github.com/sadofrazer/static-website-example.git »
- conteneuriser cette application à l'aide de l'image de base nginx.

- Il s'agit ici de créer une image docker afin de conteneuriser une application web statique,
- conteneuriser cette application sans télécharger les fichiers au préalable en local à l'aide de l'image de base ubuntu.
- Fichiers de l'application se trouvant dans le repo : https://github.com/daviddias/static-webpage-example.git

# MEILLEURES PRATIQUES DANS LA  CRÉATION DES IMAGES

- Points d'attention :
  - Les conteneurs créés par l'image doivent être le plus « stateless » possible.
  - Build context
  - Use a .dockerignore file
  - Use multi-stage builds
  - Gestion des paquets
  - Chaque conteneur doit avoir une utilité précise.
  - Minimize the number of layers
  - Sort multi-line arguments
  - Build cache

# BUILD CONTEXT

- Description :
  - Lorsque l'on utilise la commande « docker build », le daemon docker créé un « context » qui contient tous les fichiers et dossiers du répertoire contenant le fichier « Dockerfile ».
  - Lors de la création de l'image, tous les dossiers et fichiers contenus dans le « build context » sont envoyés au daemon docker, même si ils ne sont pas nécessaires à la création de l'image.
  - Plus le « Build Context » est volumineux, plus la charge de travail du daemon docker pour créer l'image sera importante !
- Meilleure pratique :
  - Il est donc nécessaire de s'assurer que seuls les fichiers réellement
  - indispensables à la création de l'image sont dans le « Build Context ».

# .DOCKERIGNORE FILE

- Description:
  - Le fichier caché « .dockerignore » doit être placé dans le même répertoire que le fichier « Dockerfile ».
  - C'est l'équivalent d'un « .gitignore » pour Git.
  - Il permet « d'ignorer » les fichiers présent dans le « context » qui ne sont pas nécessaire à la création de l'image.
  - Ce fichier permet ainsi de créer les images plus rapidement et de diminuer la charge de travail envoyée au daemon Docker.

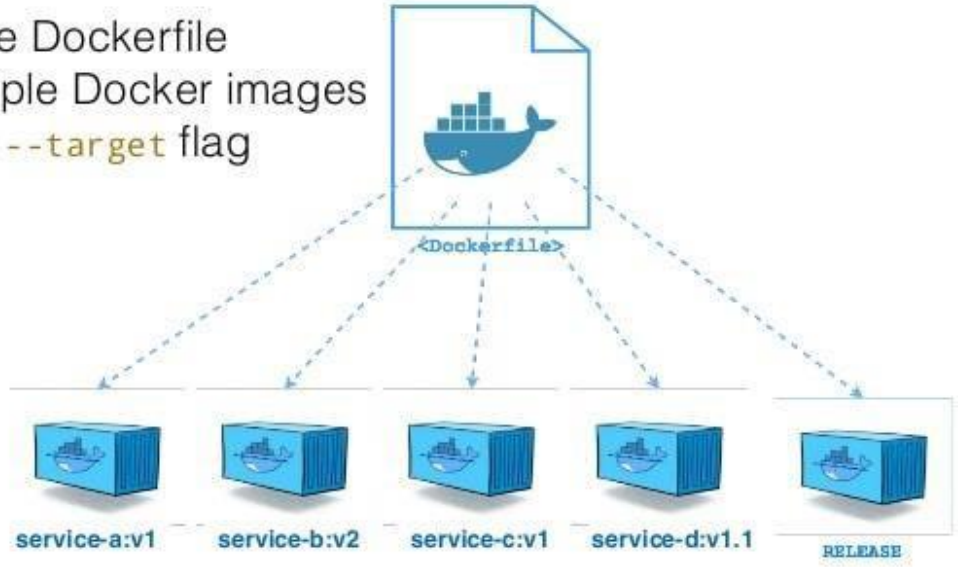
# MULTI-STAGE BUILD

- Description :
  - Le « multi-stage builds » est une nouvelle façon de construire les images de conteneurs applicatifs apportés avec la version 17.05 de Docker.
  - Cette technique est la plus puissante pour réduire drastiquement la taille d'une image docker.
  - Le principe est d'utiliser un « Dockerfile » qui aura deux parties.
    - La première partie déclare une image « intermédiaire » et qui sert à construire l'application.
    - La deuxième partie déclare une image « minimaliste » (souvent depuis « scratch », qui est l'image la plus petite possible), et qui exécute l'application compilée lors de la première partie.
    - Il y a donc deux fois l'instruction « FROM » dans le Dockerfile.
  - Au final on obtient une image minimaliste parfaitement fonctionnelle et ne contient que notre application.

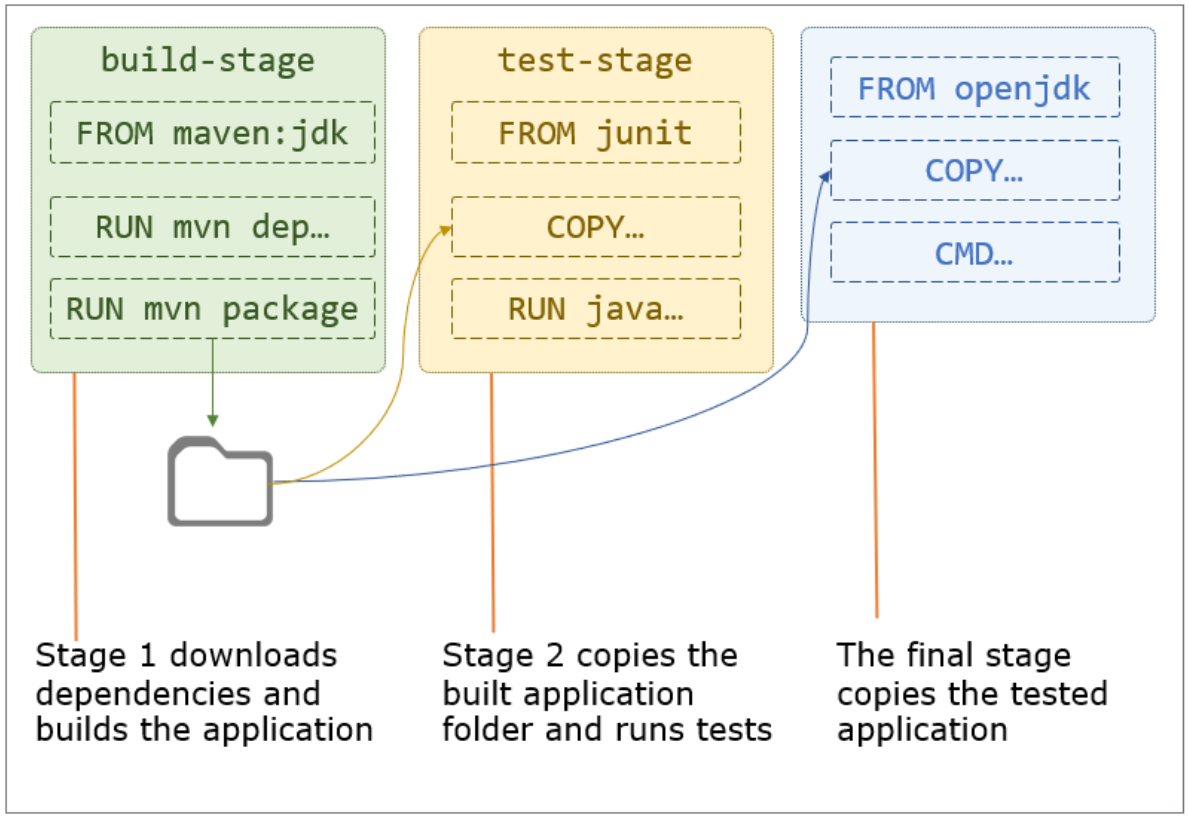
# BUILD MULTISTAGE
# CAS D'UTILISATION

## Multi-build Use Case

- ✓Same Dockerfile
- ✓Multiple Docker images
- ✓With `--target` flag




```
$ docker build -t repo/service-a:v1 --target service-a .
$ docker build -t repo/service-b:v2 --target service-b .
```



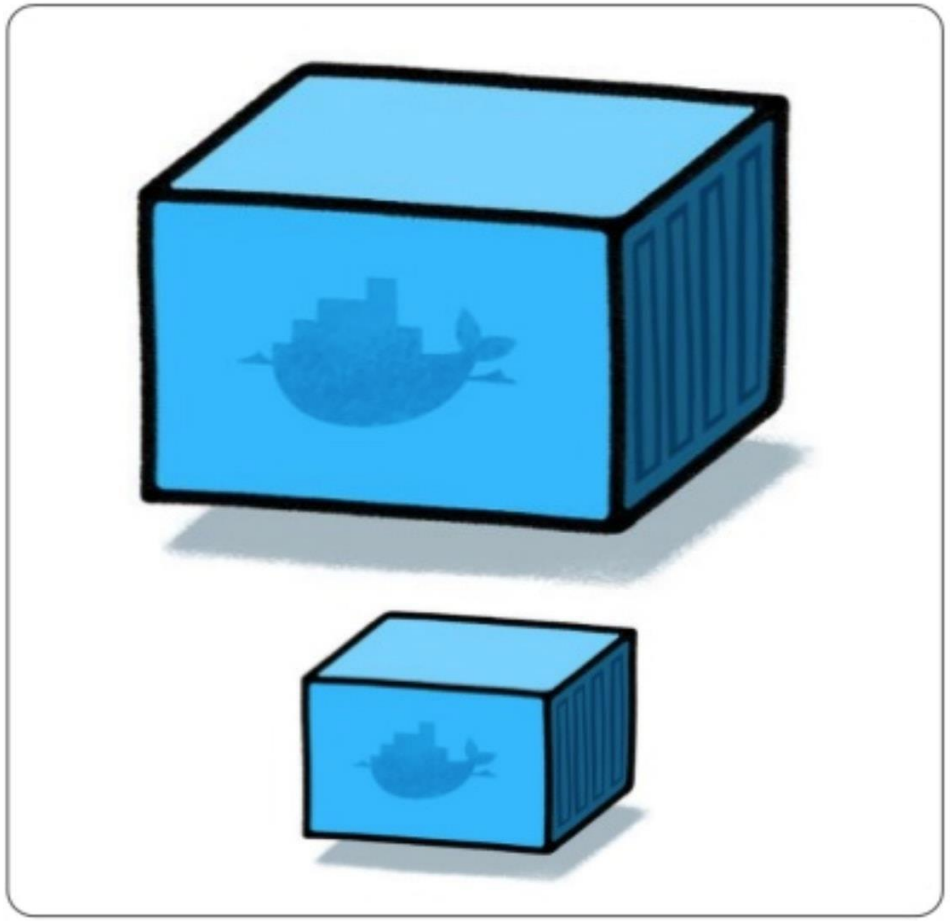

ANDN SERVICES

# BUILD MULTISTAGE
# EXEMPLE D'UTILISATION

1

2

```
# Dockerfile
# build stage
FROM buildbase as build
...
...
...

# production ready stage
FROM runbase
...
COPY --from=build
/artifact /app
```



# DOCKERFILE – PACKAGING D'UNE APP- MULTI-STAGE BUILD (TOUTES CES INSTRUCTIONS DANS UN DOCKERFILE)

- FROM alpine/git as clone
- WORKDIR /app
- RUN git clone https://github.com/XXXXXXXXXXXX

- Depuis l'image avec comme nom clone
- On va dans le dossier /app (répertoire de travail)
- On fait une git clone dans le workdir

- FROM truc/truc as build
- WORKDIR /app
- COPY --from=clone /app/helloworld /app
- RUN truc package

- Depuis l'image avec comme nom clone
- On va dans le dossier /app (répertoire de travail)
- Depuis l'image clone, monte le /app/helloworld dans le /app de l'image truc/truc
- Packager l'application

- FROM tomcat
- COPY –from=build /app/target/helloworld.sh /usr/local/webapps/ROOT
- COPY –from=build /app/target/helloworld/ /usr/local/webapps/ROOT
- EXPOSE 8080

- FROM tomcat, cette partie sera l'image final donc pas besoin du « as »
- Ensuite depuis l'image build tu récupère les dossiers/fichiers nécessaires dans le dossier de tomcat pour avoir l'appli
- Puis on expose sur le port 8080

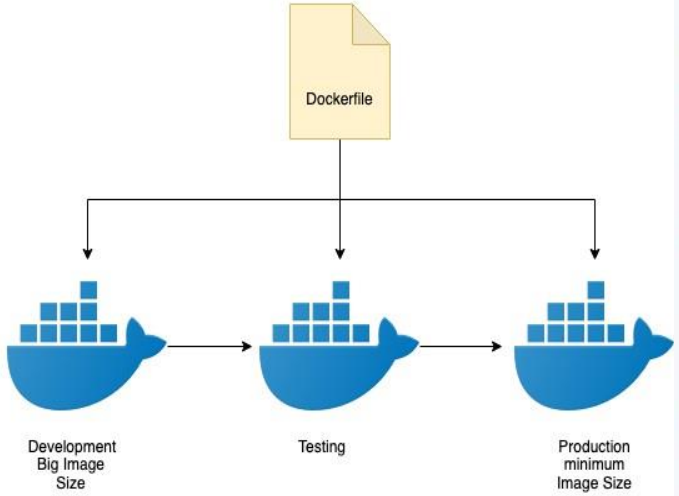
# BUILD MULTISTAGE
# EXEMPLE D'UTILISATION

```
# Stage - Development
FROM Dev-Image as dev
. . .
. . .

# Stage - Testing
FROM Test-Image as test
COPY --from=dev /src/app .
. . .
. . .

# Stage - Production
FROM Minimum-Image as prod
COPY --from=dev /src/app .
. . .
. . .
```

stage 0

stage 1

stage 2




```
# syntax=docker/dockerfile:1
FROM golang:1.16
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go ./
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]
```

```
# syntax=docker/dockerfile:1
FROM alpine:latest AS builder
RUN apk --no-cache add build-base

FROM builder AS build1
COPY source1.cpp source.cpp
RUN g++ -o /binary source.cpp

FROM builder AS build2
COPY source2.cpp source.cpp
RUN g++ -o /binary source.cpp
```

```
# syntax=docker/dockerfile:1
FROM golang:1.16 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    ./
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]
```

# TP – 19 : CRÉATION D'IMAGES DOCKER

- Utiliser le build multistage avec 3 stages afin de conteneuriser l'application précédente dans le but de réduire la taille finale de notre image.
- Nous devons avoir 03 stages, le premier nous permettant de récupérer les fichiers source avec comme base ubuntu, le second stage nous permettant d'héberger notre app web à l'aide de apache et le troisième à l'aide de nginx.
- Créez à l'aide de ce dockerfile 02 images docker ayant pour tag « name:httpd » et « name:nginx » que vous pousserez ensuite vers votre dockerhub,
- Sauvegarder votre fichier (dockerfile) sur votre repo github

# GESTION DES PAQUETS

- Description :
  - La gestion des paquets lors de la création des images est capitale pour diminuer la taille de l'image et pour éviter les problèmes de paquets corrompus.
  - Il faut installer dans l'image uniquement les paquets strictement nécessaires au bon fonctionnement de l'application. Cela peut paraître évident, mais il n'est hélas pas rare de voir des images avec des paquets inutiles !
  - Une version doit systématiquement être indiquée pour chaque paquet à installer dans l'image pour éviter de créer des images avec des paquets en version « latest » qui pourrait être corrompues, ou non supportés par l'application les utilisant.

# CHAQUE CONTENEUR DOIT AVOIR UNE  UTILITÉ PRÉCISE.

- Description:
  - Chaque conteneur doit avoir une fonctionnalité bien précise.
  - Segmenter une application en plusieurs « processus » permet  d'améliorer la scalabilité horizontale de l'application et favorise la  réutilisation des images de conteneurs.

# GESTION DES COUCHES

- Description
  - Docker stocke les images par layer.
  - Si une même couche est présente sur différentes images, la couche n'est pas dupliquée sur le disque du node.
  - En suivant ce principe de déduplication des couches en mémoire, il est recommandé de construire les images dockers en ordonnant les couches, de la moins fréquemment changée à la plus fréquemment changée.
  - Cela permet ainsi de drastiquement diminué l'espace de disque utilisé par les images.
- Eléments à prendre en compte :
  - Dans le Dockerfile, seules les instructions « ADD », « COPY » et « RUN » ajoutent des couches à l'image.
  - Toutes les autres instructions ajoutent des couches temporaires qui n'augmentent pas la taille de l'image finale.

# SORT MULTI-LINE ARGUMENTS

- Description :
  - Lorsque c'est possible, arrangez le Dockerfile pour avoir les instructions
  - les plus petites possible. Cela peut se faire en ayant recours au symbole « \ » avant de faire un retour à la ligne.
  - Si vous devez installer plusieurs paquets, utilisez le « multi-ligne » pour installer tous les paquets par ordre alphabétique avec une seule commande RUN.
  - Cela permet de faciliter les mises à jour et les relectures du Dockerfile par la suite.
  - Exemple :

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

# BUILD CACHE

- Description :
  - Par défaut, lors de la création d'une image, Docker vas chercher dans son cache local si il y a déjà des couches utilisables pour la nouvelle image à construire.
  - L'utilisation du cache d'image Docker permet de construire plus rapidement les images.
  - Il est possible de désactiver l'utilisation du cache d'images docker en ajoutant l'argument « --no-cache=true » à la commande « docker build ».

# GÉRER LES VARIABLES AU SEIN DES IMAGES

- Objectifs:
  - Il est parfois nécessaire de variabiliser les fichiers nécessaires à la création d'une image. (Par exemple pour indiquer une base de données postgresql , ou un Bucket Amazon à une application).
  - Cela permet de personnaliser des images Docker construites en se basant sur les mêmes fichiers source / Dockerfile.
  - La technique la plus utilisée est la suivante :
    - Créer sur l'hôte docker des variables d'environnements.
    - Appeler ces variables d'environnement dans les fichiers sources / Dockerfile via : ${VARIABLE}
- En général, ces variables d'environnement servent à définir les variables déclarées dans les instructions « ENV » et « ARG » des Dockerfile.

# TAG / PUSH / PULL

- docker tag <imagesource>:<version> <imagedest>:<version>
  - Sert à donner un tag à son image, pour y ajouter la version latest qui est par defaut par exemple
- Puis docker-build –t myalpine:v1.0
- Docker images ls
- On a maintenant une image myalpine en v1.0
  - Docker run –ti myalpine
  - Il ne la trouve pas car il n'y a pas de tag latest
- Docker tag myalpine:v1.0 myalpine:latest
- On a maintenant une image mais avec 2 tags, v1.0 et latest
  - Docker run –ti myalpine

# TAG / PUSH / PULL

- Docker login
  - Indiquer login/mdp de docker Hub
- Docker login <chemin_du_registre>
  - Pour se connecter à un autre registre
- Docker push <regsitre>/<nomimage>:version
- Docker pull <regsitre>/<nomimage>:version
  - Docker push myalpine:latest
- Si vous êtes sur un autre repository, vous devrez tag votre image avec le lien complet du regsitre :
  - Docker tag myalpine:v2.0 registry.gitlab.com/vlne/docker/myalpine:v2.0
  - Docker push registry.gitlab.com/vlne/docker/myalpine:v2.0

# TP 1 - CRÉATION D'UN CONTAINER

Lancez un premier premier container docker, l'image à utiliser devra être « nginx »

Vérifier si votre container est bien fonctionnel à l'aide de la commande « docker ps »

Stoppez votre container

Supprimez votre container et s'assurer qu'il a bien été supprimé.

Supprimer également les volumes qui auront été créé

# TP 2 - LES IMAGES

Combien d'images docker existent-ils dans votre machine à ce stade ?

Relancer à nouveau un container docker à l'aide de l'image nginx

Combien de temps cela prend ? Est-ce le même temps que précédemment?

Supprimer l'image nginx de votre machine

Avez-vous pu supprimer la dite images? Pourquoi?

Supprimer le container nginx

Supprimer à nouveau l'image nginx de votre machine

# TP 3 - INTÉRACTION AVEC UN CONTAINER

Lancer le container Ubuntu en mode détaché et interactive avec un tag différent de latest

Que constatez vous?

Créez une nouvelle machine dans l'environnement de labs

Dans cette machine, lancez à nouveau le container ubuntu en mode interactive + nouveau terminal

Que constatez vous?

Supprimez les différents containers présent dans votre machine

Lancez le container Ubuntu en mode détaché avec la commande « sleep 4500»

Utiliser la commande « docker exec » afin d'exécuter une commande à l'intérieur du container (commande de création d'un dossier portant votre prénom »

Utiliser docker exec afin d'afficher l'ensemble des fichier et répertoire à l'aide de la commande « ls »

Utiliser docker exec et passez la commande « /bin/bash »

Que constatez vous?

Observez et expliquez le comportement de ces différentes commandes puis supprimez vos environnements

# TP 3 - COMMANDES

Lancer le container ubuntu

Vérifier que ce container soit bien lancé et qu'il fonctionne correctement

Pourquoi le container s'est-il arreté

Supprimer le container ubuntu

Lancer à nouveau le container ubuntu en mode attach et en passant en paramètre la commande « sleep 100 » Que constatez vous?

Lancez à nouveau le container ubuntu en mode attach passant la commande « ls » Que constatez vous?

Lancez le container ubuntu en mode détaché avec la commande « sleep 4500 » Observez et expliquez le comportement de ces différentes commandes

Supprimez l'ensemble des containers présents (fonctionnels ou arrêtés) sur votre machineSupprimez également les images exisatantes

# TP – 5- PORT MAPPING

- Lancer le container nginx en mode détaché et en exposant le port 80 du container sur le port 8080 de l'hôte
- Vérifier le bon fonctionnement de votre container
- Vérifier qu'il est bien joignable à travers le port 8080 de votre hôte.
- Que constatez vous? Quel est le contenu de la page web affichée?
- A l'aide de docker exec, connectez vous à ce container afin de créer un fichier « index.html » dans lequel vous feriez
- une briève présentation de vous
- Déplacer ce fichier index.html précédemment créé vers le répertoire « /usr/share/nginx/html» de votre container
- Arrêtez votre container et redémarrez le à nouveau
- Essayez à nouveau d'y accéder à travers le port 8080 de votre hote… Que constatez vous?

# TP – 6 : OPÉRATIONS ET INSPECTION D'UN CONTAINER

- Créez un fichier « index.html » en local sur votre machine contenant les mêmes informations que celui du TP 5
- Lancez le container nginx en l'exposant sur le port 8080 et en ajoutant l'option (-v fichier_index.html:/usr/share/nginx/html//index.html)
- Vérifier que le container soit bien lancé
- Vérifiez que l'application de notre container nginx est bien consommable à partir du port 8080 de notre hote
- Que constatez vous? Quel est le contenu de la page web afficheé?
- Supprimer les containers, images et votre environnement.

# TP – 7 : MANIPULATION VARIABLES D'ENVIRONNEMENT

- Lancez un container à partir de l'image kodekloud/webapp-color en exposant son port 8080 sur le port 5000 de votre  hote
- Connectez vous sur le port 5000 de votre hote afin de consommer l'application
- Lancez à nouveau un autre container à base de la même image mais cette fois ci en définissant la variable
- d'environnement APP_COLOR=red. (exposez sur le port 8000 )
- Consommez l'application. Que constatez vous?
- Supprimer votre environnement