DOCKER CFITECH

Présentation de Docker

N°	Sujet	Détails
1 -	Introduction	1 - Présentation des points à retenir du cours
2 -	Premiers pas avec la conteneurisation	2 - Microservice vs approche monolithique Conteneurs vs virtualisation
3 -	Introduction à Docker	3 - Introduction Docker, architecture, fonctionnalités, registre
4 -	Configuration de Docker	4 - Installation sur Linux et Windows Docker Desktop
5 -	Utilisation des images et des conteneurs	5 - Image Docker, conteneurs, commandes
6 -	Gestion de Dockerfile	6 - Création et gestion de Dockerfiles, brève introduction de Dockerfile et détails de l'application conteneurisée.
7 -	Docker Network	7 - Pilote réseau (Bridge, host, none)
8 -	Docker Volume	8 - —----------
9 -	Docker Compose	

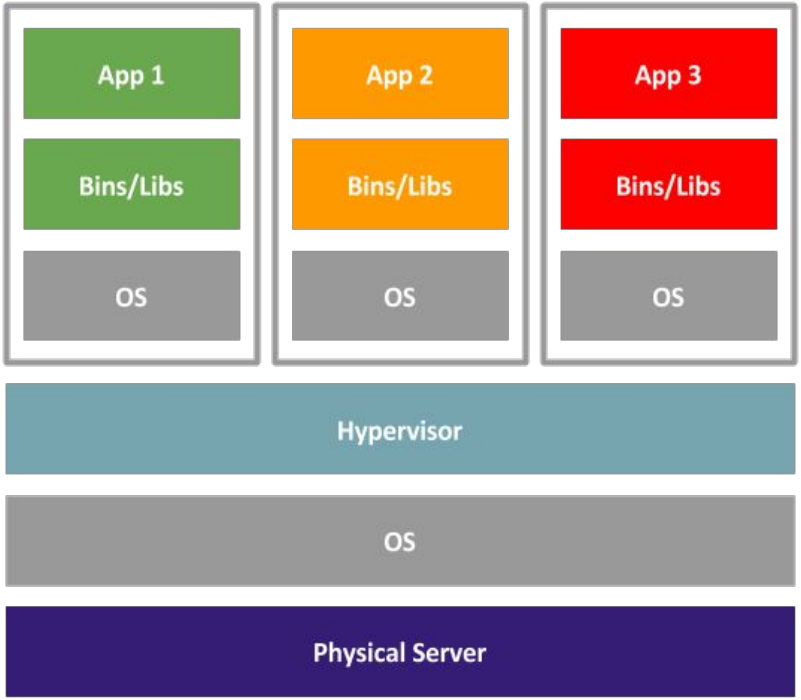
Avantages du cours et que recevrez-vous ?

- **Apprenez** les bases de Docker et comment l'utiliser pour créer et déployer des applications.
- **Acquérez** une expérience pratique avec Docker en effectuant des travaux pratiques.
- **Improvisez** votre compréhension de la conteneurisation et de ses avantages.
- **Préparez-vous** à une carrière dans le cloud computing ou DevOps.

Docker - Brève introduction

- Une plateforme de conteneurisation.
- Regroupez une application et ses dépendances dans une seule image.
- Exécutez sur n'importe quelle machine installée sur Docker.
- Crée, gère et a également la capacité d'orchestrer des conteneurs.

Avant la conteneurisation



- Avant 2000, des serveurs pour chaque application.

- Différentes infrastructures et dépendances et incapacité à évaluer les besoins en ressources

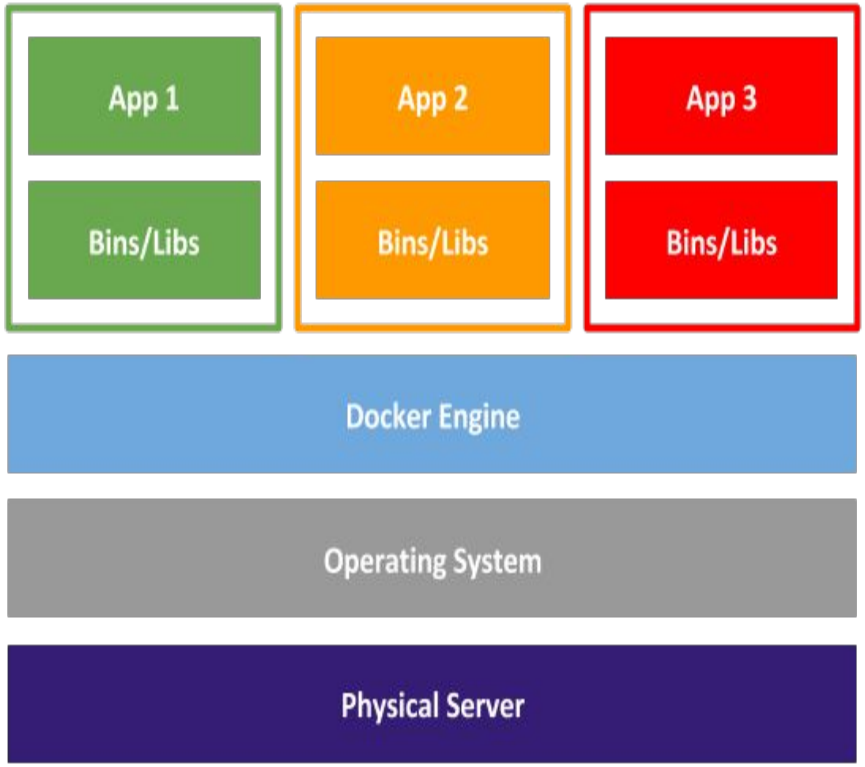
- Coûteux, gaspillage de ressources et nombreux serveurs à gérer !

- Au début des années 2000, la technologie de virtualisation était populaire comme moyen d'isoler les applications et les services sur un seul serveur physique.

- Plusieurs applications sur un seul serveur, partage des ressources, mais.

- Licences, gourmandes en ressources et entraînant souvent une surcharge de performances.

Conteneurisation



- En 2008, une nouvelle technologie appelée **conteneurs Linux** a fourni une alternative plus légère à la virtualisation.

- En 2013, **Docker** a été créé en tant que projet **open source** pour simplifier et améliorer le processus de création, de déploiement et d'exécution d'applications à l'aide de conteneurs Linux.

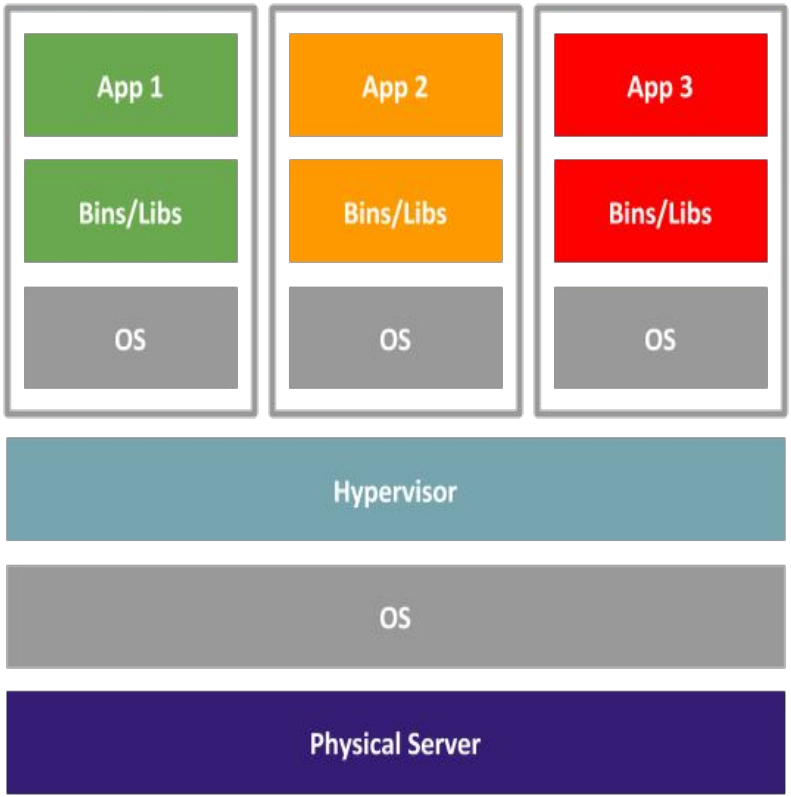
- Une méthode pour **packager** une application afin qu'elle puisse être exécutée, avec ses dépendances, isolée des autres processus.

Conteneurs vs VM

- Différence majeure entre les **conteneurs** et le modèle **VM** (OS, ressources matérielles, frais de licence, portables et rapides, une fois utilisés, ils fonctionnent partout).

- Les **conteneurs** permettent de réaliser des économies sur les fronts **cap-ex** (argent) et **op-ex** (ressources humaines).

Virtualisation



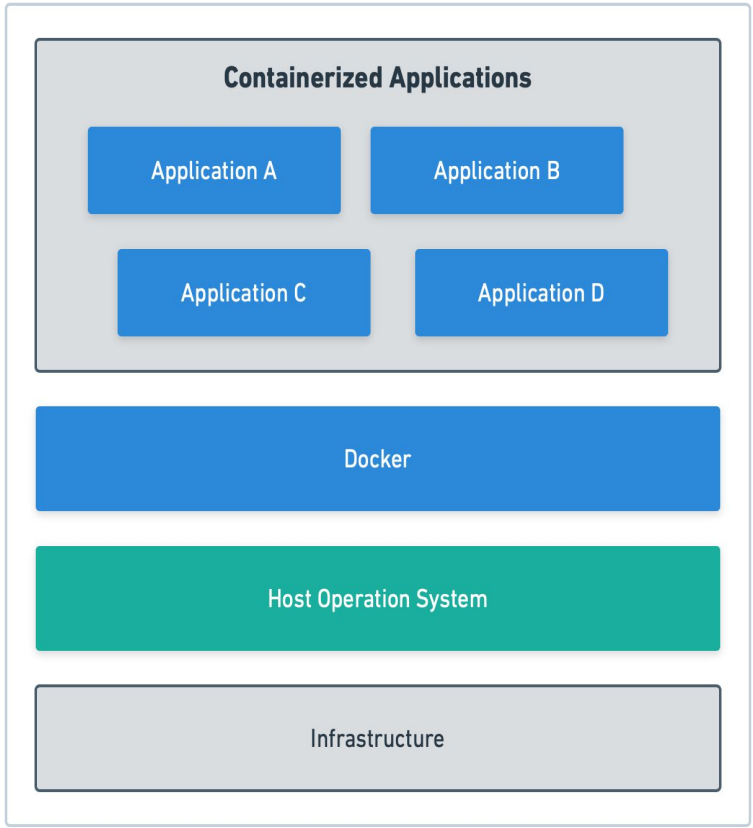
- Introduction de la VM en 1999, VMWare

- Certains avantages (plusieurs applications sur un seul serveur, différents systèmes d'exploitation et dépendances sur le même serveur, meilleurs et économiques).

- Et d'autres inconvénients (le système d'exploitation consomme beaucoup de ressources, licences).

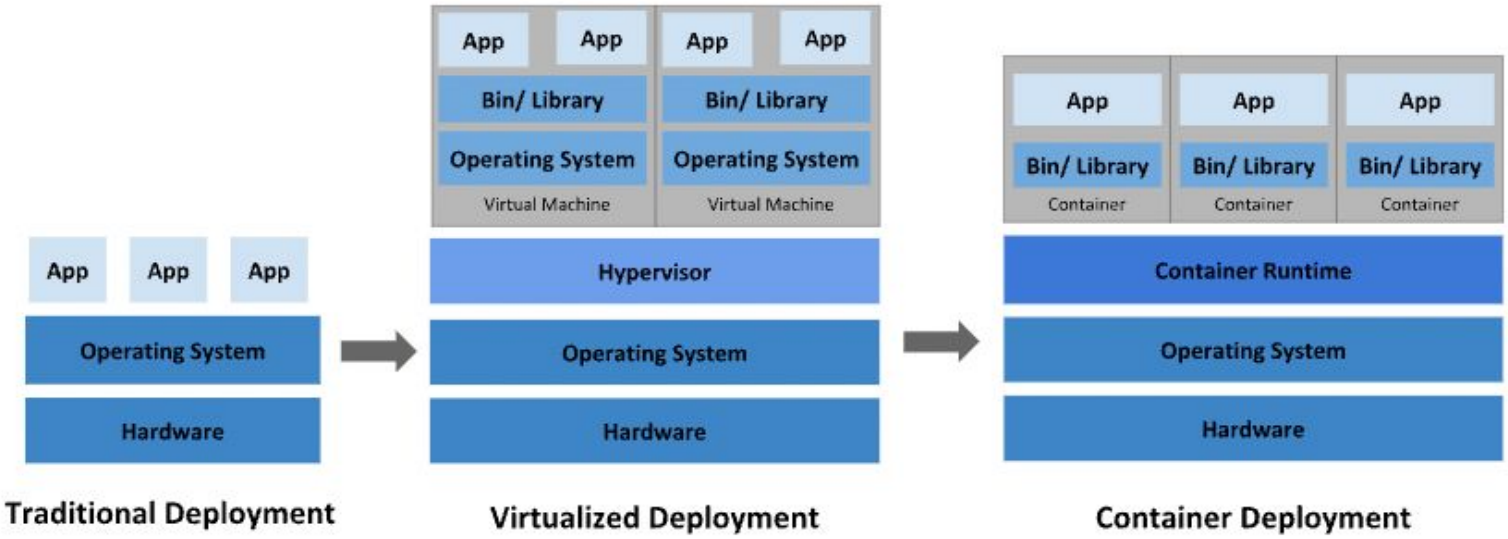
- **Hyperviseur** isole le système d'exploitation et l'application du matériel informatique sous-jacent.

Conteneurisation



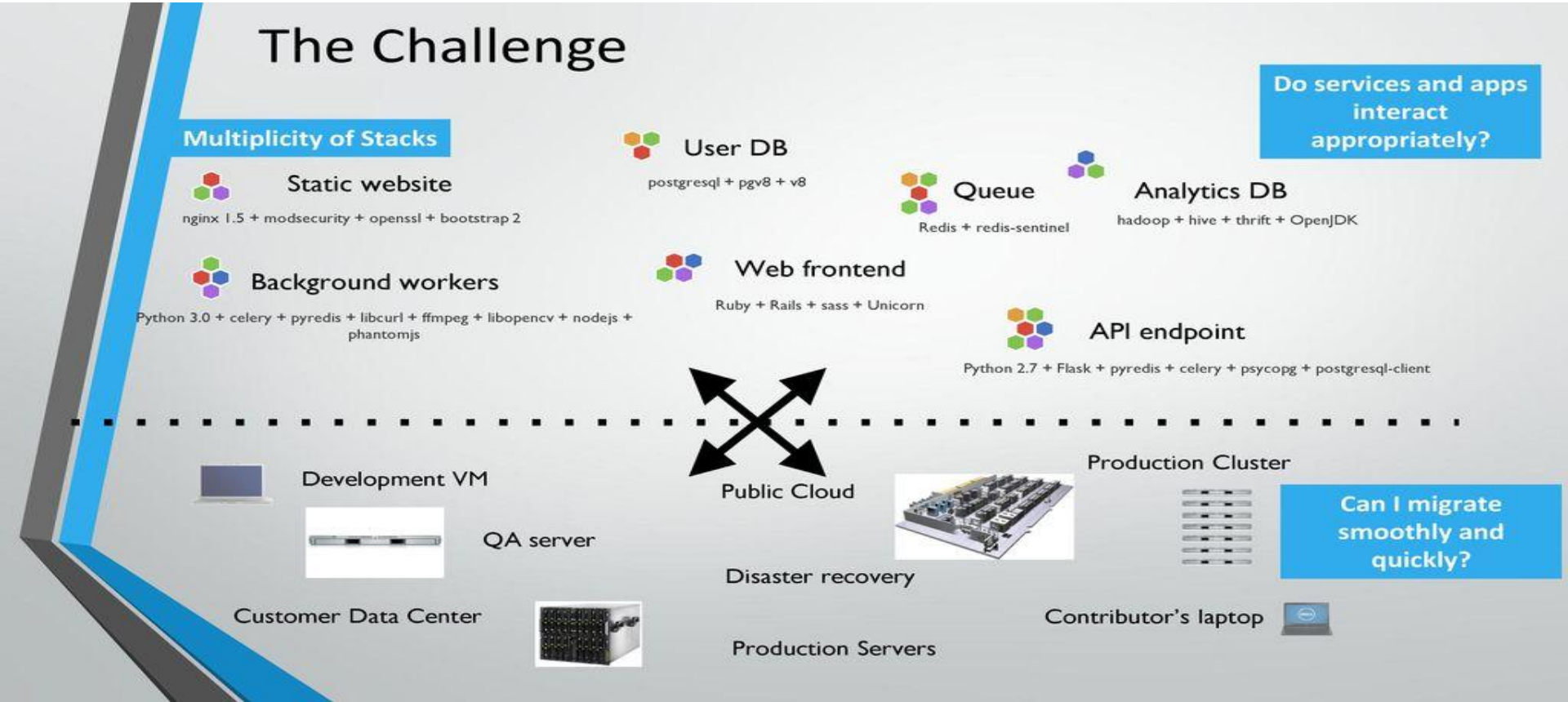
- Unité logicielle standard.
- Codes et dépendances des packages.
- Peut être partagé sous forme d'images Docker.
- Plusieurs conteneurs peuvent être exécutés simultanément.
- **Portable** utilisé avec n'importe quel système d'exploitation.
- **Léger** utilise le système d'exploitation hôte.
- **Sécurisé** fonctions d'isolation par défaut puissantes.
- Parfois utilisé avec des **VM.**

Comparaison

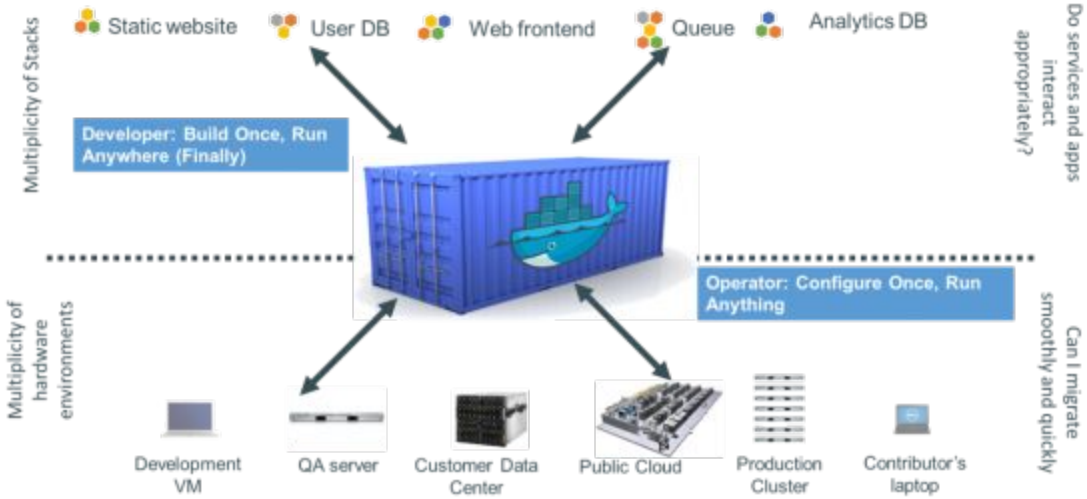


Technologie	heure de début	heure de fin
Conteneurs Docker	< 50 ms	< 50 ms
Machines virtuelles	30 - 45 sec	5-10 sec

Le problème


The Challenge
Do services and apps interact appropriately?
Multiplicity of Stacks
User DB
postgresql + pgv8 + v8
Static website
nginx 1.5 + modsecurity + openssl + bootstrap 2
Queue
Redis + redis-sentinel
Analytics DB
hadoop + hive + thrift + OpenJDK
Background workers
Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs
Web frontend
Ruby + Rails + sass + Unicorn
API endpoint
Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client
Development VM
Public Cloud
Production Cluster
QA server
Can I migrate smoothly and quickly?
Disaster recovery
Customer Data Center
Contributor's laptop
Production Servers

Conteneurs pour applications



L'industrie du logiciel évolue

- **Avant** applications monolithiques, cycles de développement longs, environnement unique, montée en puissance lente.

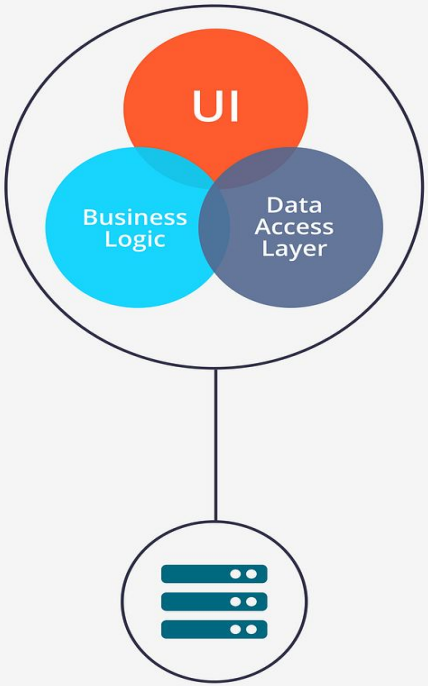
- **Maintenant** services découpés, rapides, itératifs, améliorations, environnements multiples, montée en puissance rapide.

Le déploiement devient complexe

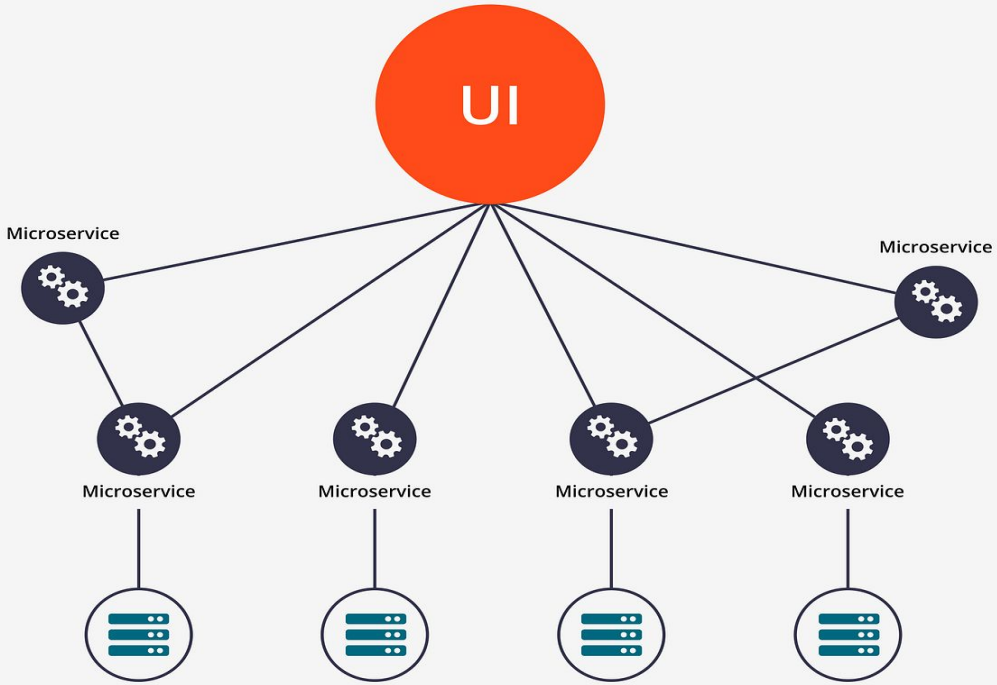
- *De nombreuses piles différentes* (langages, frameworks, bases de données),

- *De nombreuses cibles différentes* (environnements de développement individuels, pré-production, QA, staging..., production, sur site, cloud, hybride).

Monolith vs Microservices



Monolithic Architecture



Microservice Architecture

Monolith

- Simple à utiliser/tester/déployer/mettre à l'échelle.
- Trop volumineux et complexe.
- Une grande taille d'application peut ralentir le temps de démarrage.
- Redéploiement complet.
- Niveau élevé requis pour le débogage
- Déploiement continu difficile

Microservices

- Diviser l'application en un ensemble de services plus petits et interconnectés.
- Développement, compréhension et maintenance plus rapides.
- Développé indépendamment.
- Déployé indépendamment.
- Mise à l'échelle indépendante.

Dockers

- Une technologie d'exécution et d'orchestration de conteneurs.

- Un logiciel open source (appelé Moby maintenant)

- Fonctionne sous Windows et Linux.

- **Docker, Inc.** dans une entreprise basée à San Francisco par l'entrepreneur **Solomon Hykes**

Pourquoi

Exécuter partout

- Quelle que soit la version du noyau
- Quelle que soit la distribution hôte
- Physique ou virtuel, cloud ou non
- L'architecture du conteneur et de l'hôte doit correspondre...

Exécuter n'importe quoi

- Si cela peut s'exécuter sur l'hôte, cela peut s'exécuter dans le conteneur

- Si cela peut s'exécuter sur un noyau Linux, cela peut s'exécuter

Quoi

Haut niveau

une machine virtuelle légère - propre espace de processus - propre interface réseau - peut exécuter des éléments en tant que root - peut avoir son propre /sbin/init (différent de l'hôte)

<< conteneur de machine >>

Bas niveau

Chroot sous stéroïdes - Ne peut pas non plus avoir son propre /sbin/init - Conteneur = processus isolés - partage le noyau avec l'hôte.

<< conteneur d'application>>

Cas d'utilisation et fonctionnalités

- Développer des applications qui fonctionnent sur ***n'importe quel système d'exploitation***

- Applications faciles à ***partager*** entre les équipes

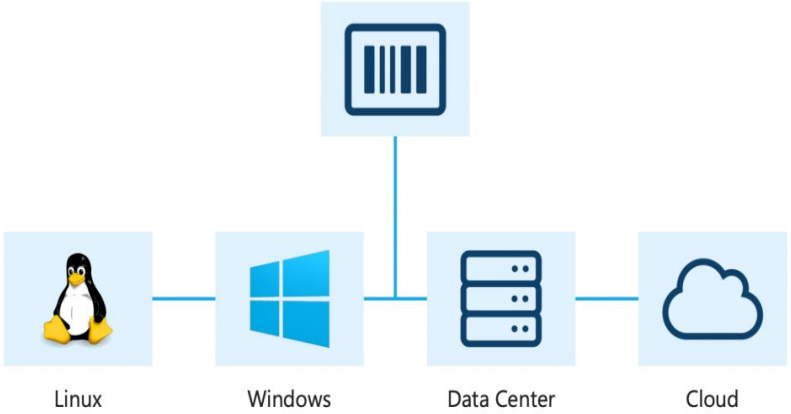
- Faciles à ***se déplacer*** sur plusieurs serveurs

- Les grandes applications peuvent être divisées en ***plusieurs conteneurs*** - un pour chaque microservice

- Excellente solution pour le ***Cloud Computing***

- Grande communauté et ***bibliothèque*** d'images Docker

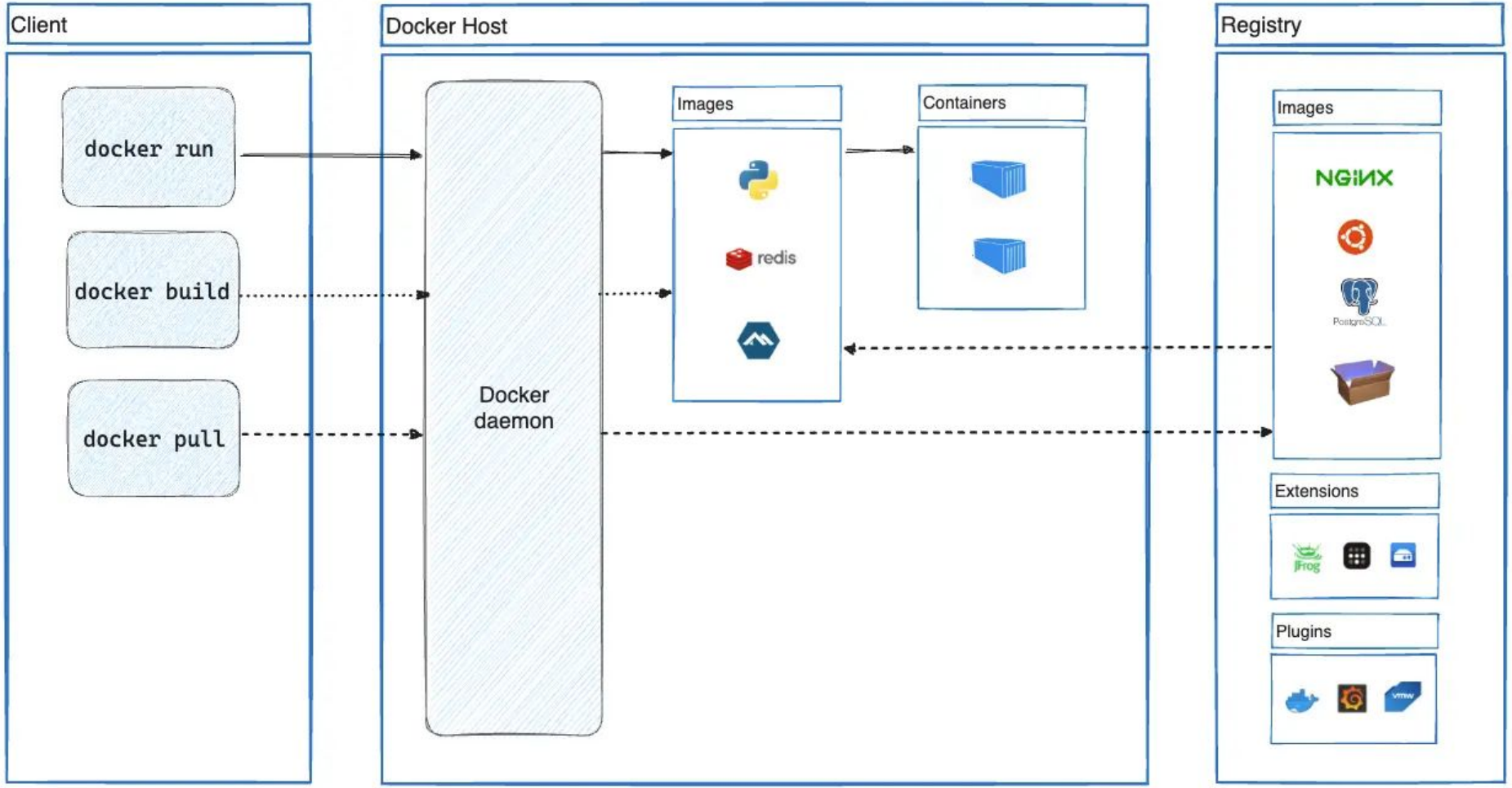
Conteneur



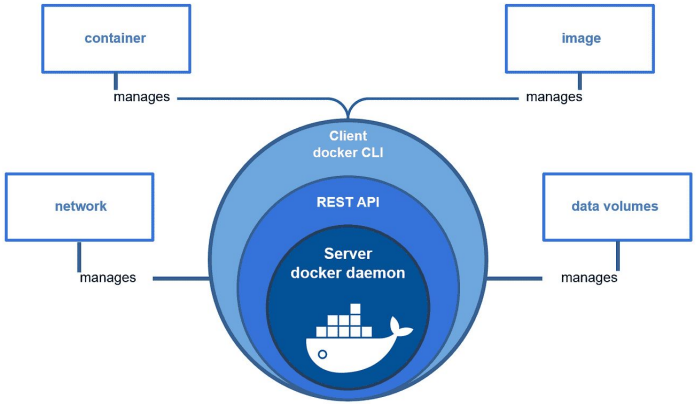
- Un type de package logiciel qui inclut tout le code et les dépendances nécessaires à l'exécution d'une application ou d'un service.

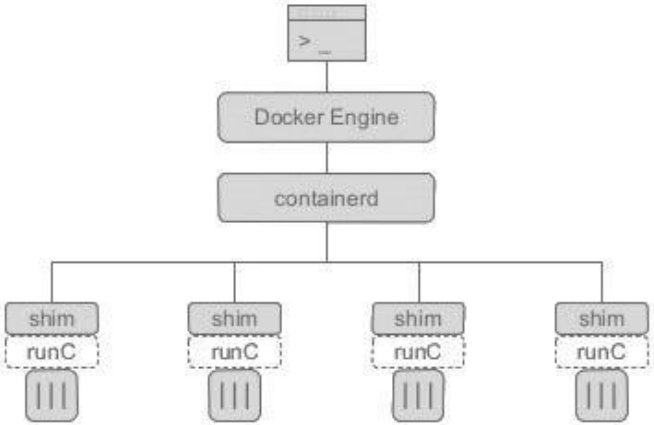
- Fournit un environnement léger et portable qui permet aux applications de s'exécuter de manière cohérente.

- Les conteneurs sont souvent utilisés dans les flux de travail de développement et de déploiement de logiciels, tels que DevOps, car ils peuvent simplifier le processus de packaging, de déploiement et de mise à l'échelle des applications.


Client
docker run
docker build
docker pull
Docker Host
Docker daemon
Images
redis
Containers
Registry
Images
NGINX
PostgreSQL
Extensions
Plugins

Docker Engine





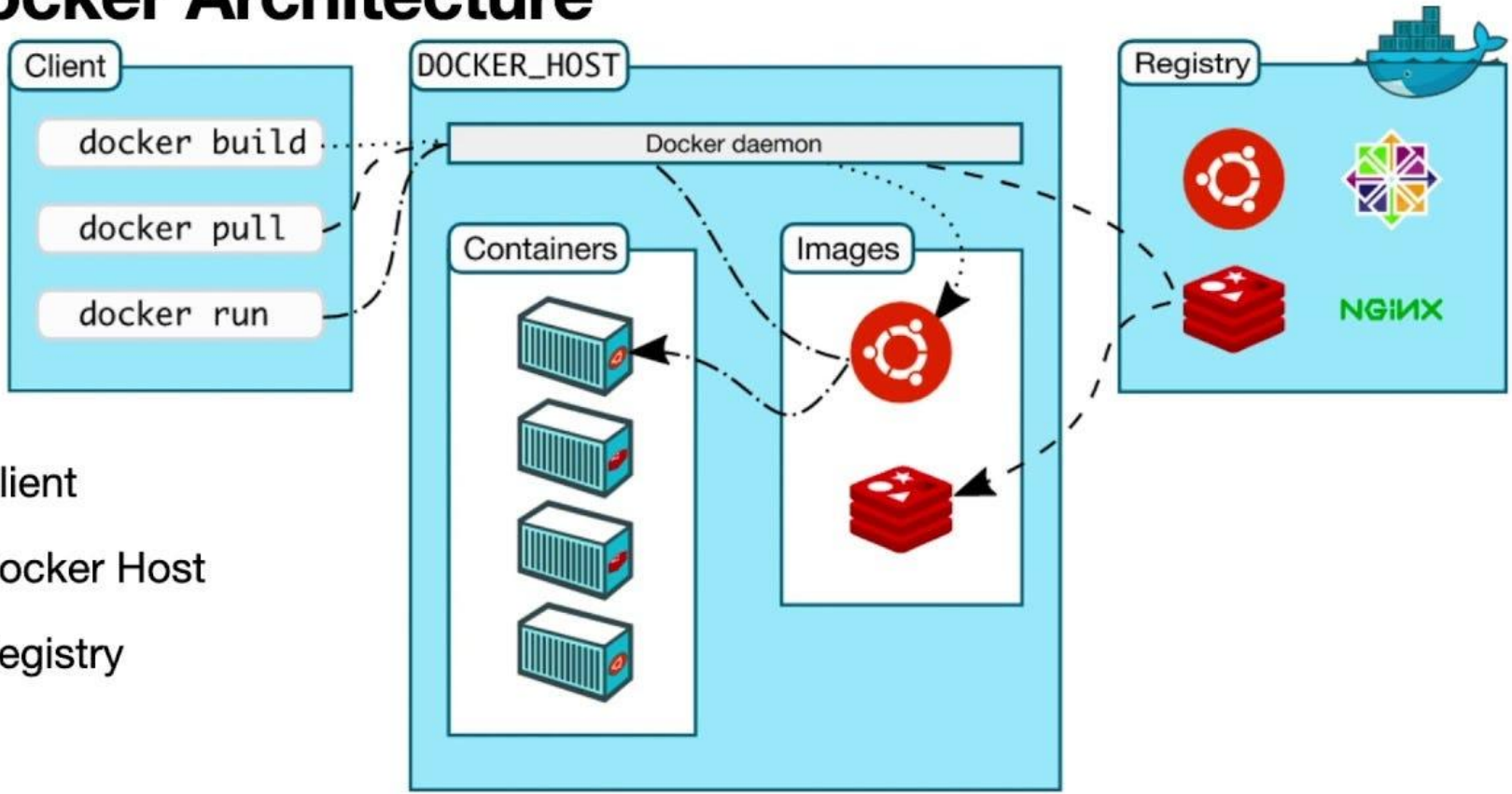
- Docker Engine est un logiciel de base qui exécute et gère les conteneurs.

- Une conception modulaire avec un composant échangeable.

Composant principal:

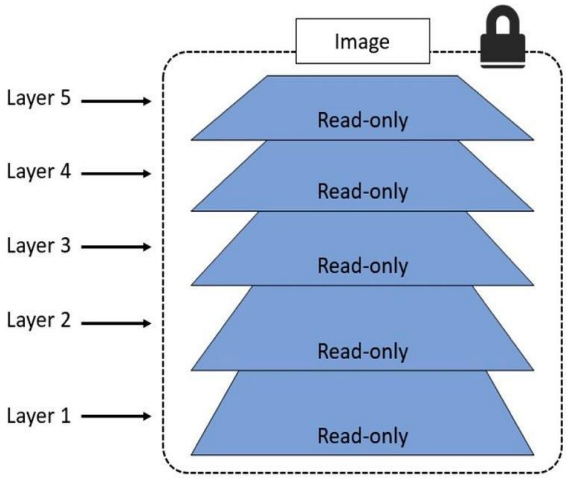
- Client Docker
- Docker Daemon
- Container
- Runc

Docker Architecture



- Client
- Docker Host
- Registry

Docker Images



- Une ***image de conteneur*** est un package logiciel léger, autonome et exécutable qui comprend tout ce qui est nécessaire pour exécuter une application :

  a - Code | Runtime | Outils système | Bibliothèques système

  b - Paramètres

- Les images deviennent des conteneurs lorsqu'elles s'exécutent sur Docker Engine.

- Plusieurs couches pour constituer une image.

Images (métadonnées)

- *Image = Fichier + Métadonnées*

- Les métadonnées peuvent indiquer un certain nombre de choses, par exemple :

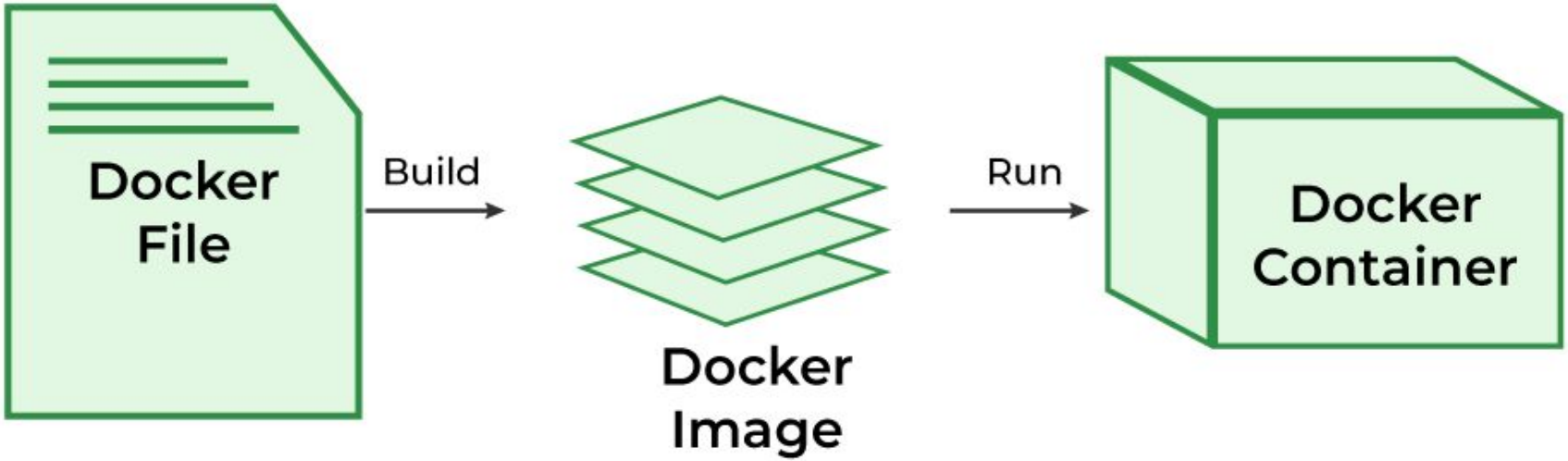
- L'auteur de l'image

- Le conteneur à exécuter dans le conteneur lors de son démarrage

- Variable d'environnement à définir

- Les images sont constituées de couches, empilées conceptuellement les unes sur les autres

```
cfitech@docker:~$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
dafa2b0c44d2: Already exists
Digest:
sha256:dfc10878be8d8fc9c61cbff33166cb1d1fe44391
539243703c72766894fa834a
Status: Downloaded newer image for
ubuntu:latest
docker.io/library/ubuntu:latest
```


Docker
File
Build
Docker
Image
Run
Docker
Container

Registry

- ***Registry*** est un service chargé d'héberger et de distribuer les images Docker.

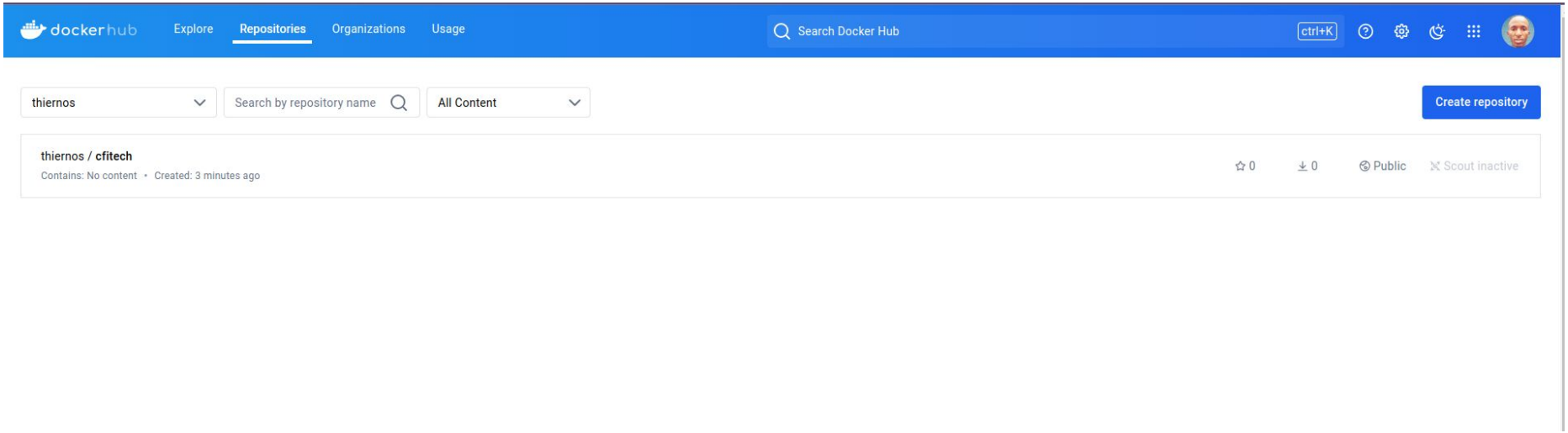
- ***Docker Hub*** est le registre public par défaut fourni par Docker, mais nous pouvons configurer et utiliser votre propre registre privé.

Registry - Docker Hub

- Il existe de nombreuses options de registres disponibles, le registre le plus courant est Docker Hub (https://hub.docker.com).

- Le Docker client est également opiniâtre et utilise par défaut Docker Hub.

- Docker Hub est le moyen le plus simple au monde de créer, de gérer et de livrer les applications de conteneurs de votre équipe.


dockerhub
Explore
Repositories
Organizations
Usage
Search Docker Hub
ctrl+K
thiernos
Search by repository name
All Content
Create repository
thiernos / cfitech
Contains: No content • Created: 3 minutes ago
0
0
Public
Scout inactive

Registry - Docker Hub

Dépôts officiels et non officiels:

- **nginx** - https://hub.docker.com/_/nginx/
- **busybox** - https://hub.docker.com/_/busybox/
- **redis** - https://hub.docker.com/_/redis/
- **mysql** - https://hub.docker.com/_/mysql/

Dockerfile

- Dockerfile est un script utilisé pour créer une image Docker. Il contient un ensemble d'instructions qui sont exécutées afin de créer une application conteneurisée.

Avantages / Utilisations:

Il est utilisé pour organiser et aider au déploiement du début à la fin.

- **FROM** définit l'image de base à utiliser pour démarrer le processus de construction.

- **MAINTAINER** cette commande non exécutable déclare l'auteur, définissant ainsi le champ auteur des images.

- **RUN** exécute une commande et valide le résultat dans le système de fichiers image.

- **ARG** disponible uniquement pendant la construction d'une image Docker.

- **VOLUME** monte le répertoire hôte sur l'exécution du conteneur.

- **EXPOSE** documenté les ports qui sont destinés à être publiés au moment de l'exécution.

- **WORKDIR** définit le répertoire de travail.

- **COPY** copie les fichiers de l'hôte dans le système de fichiers image.

- **ADD** copie les fichiers de l'hôte dans le système de fichiers image.

- **ENV** définit une variable d'environnement à l'intérieur de l'image Docker.

- **CMD** définit la commande par défaut pour les conteneurs.

- **ENTRYPOINT** définit l'exécutable par défaut pour les conteneurs.

- La portabilité peut l'utiliser de n'importe où dans le monde.

- Enregistrez votre image dans le registre comme Docker Hub (en envoyant l'image vers le ***Docker Hub*** https://hub.docker.com/)

- Après avoir créé un compte, vous pouvez simplement envoyer votre image vers le référentiel Docker Hub.

- Pour envoyer une image vers Docker Hub, nous avons besoin que nos images soient créés sous ***username/repository:tag***

```
FROM ubuntu:20.04
LABEL maintainer="Cfitech"
ENV TZ=Europe/Brussels
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && \
   apt-get -y install apache2 vim curl elinks && \
   ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
ADD myfile.tar /var/www/container1/
COPY index.html /var/www/container1/
RUN cp /etc/apache2/sites-available/000-default.conf /etc/apache2/sites-available/container1.conf
&& \
   sed -i 's=DocumentRoot /var/www/html=DocumentRoot /var/www/container1/='
/etc/apache2/sites-available/container1.conf && \
   sed -i '/container1/ a \\tServerName localhost' /etc/apache2/sites-available/container1.conf
&& \
   a2ensite container1 && \
   a2dissite 000-default.conf
ENV MY_VAR="This is my server"
ENV PASSWD="HismyPass"
WORKDIR /data/
EXPOSE 8080
VOLUME /myvol
ENTRYPOINT ["bash", "-c", "service apache2 restart && exec bash"
```

Docker Networking

- Les conteneurs communiquent entre eux et avec le monde extérieur via la machine hôte, il doit y avoir une couche de **réseau** impliquée.

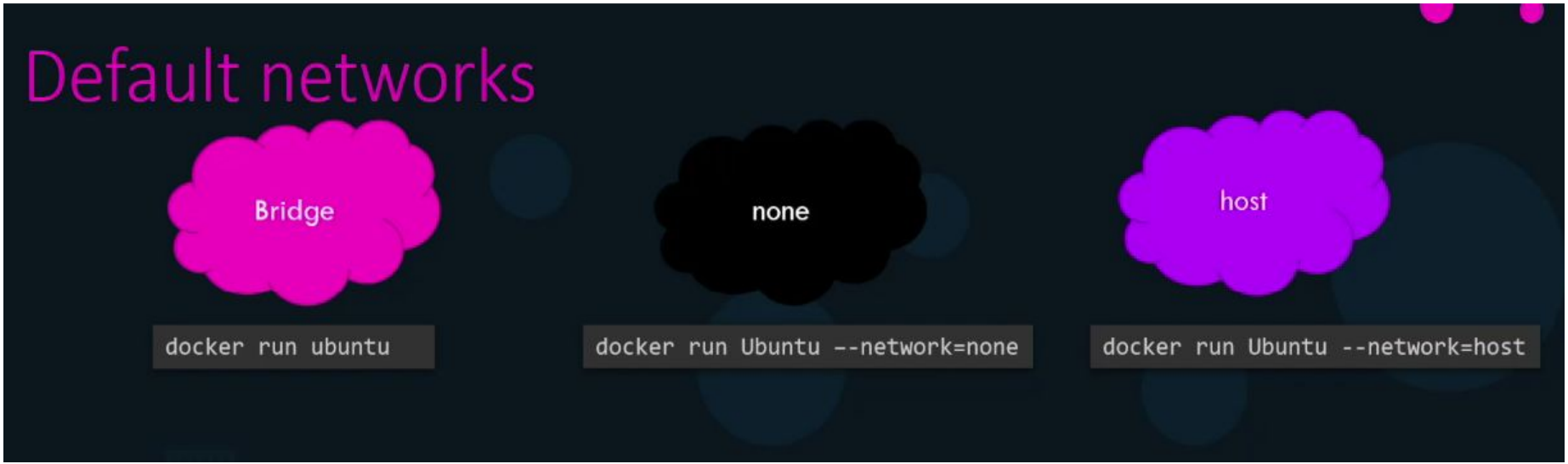
- Docker prend en charge différents types de réseaux, trois réseaux automatiquement lors de l'installation :

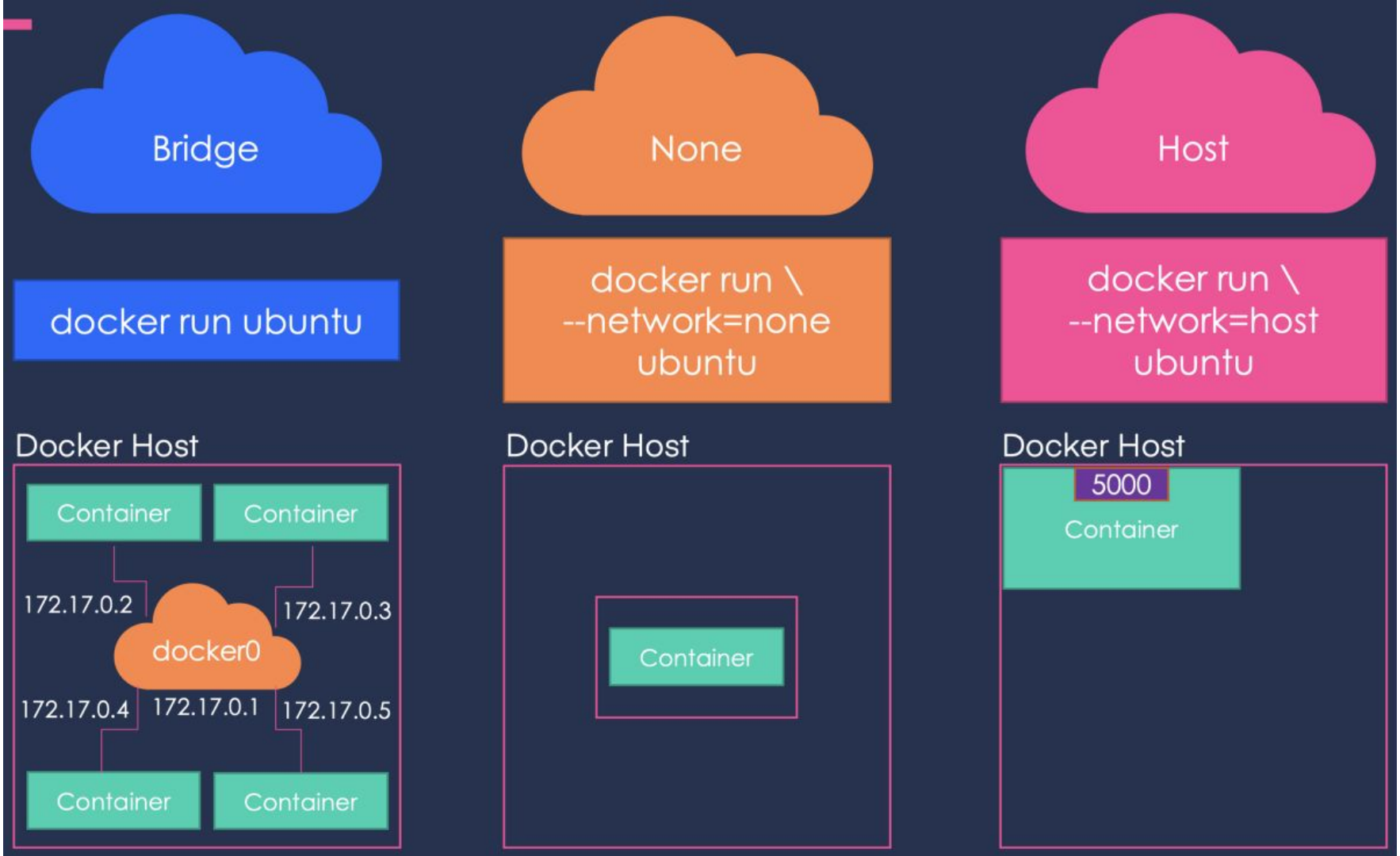
bridge (par défaut),

host,

none

Default networks




Bridge
docker run ubuntu
Docker Host
Container
Container
172.17.0.2
172.17.0.3
docker0
172.17.0.4
172.17.0.1
172.17.0.5
Container
Container
None
docker run \
--network=none
ubuntu
Docker Host
Container
Host
docker run \
--network=host
ubuntu
Docker Host
5000
Container

Networking of Docker

- Par défaut, il s'agit d'un réseau **pont.**

- Les conteneurs communiquent avec l'hôte Docker et le monde extérieur.

- L'hôte Docker peut communiquer avec tous les conteneurs en utilisant leurs adresses IP.

- Le réseau pont par défaut (interface) est visible sur l'ordinateur hôte sous le nom **docker0.**

Bridge

- Le réseau par défaut.

- Assure l'isolation de votre machine hôte, les conteneurs proviennent du réseau hôte.

- Les conteneurs peuvent se référer les uns aux autres par adresse IP sur le réseau pont.

- Le type de réseau le plus courant.

- Les adresses sont allouées sur un sous-réseau interne privé. (Docker utilise 172.17.0.0/16 par défaut).

Host

- Le pilote connecte le conteneur directement à la pile réseau de l'hôte.
- Permet à un conteneur de se connecter au réseau de votre hôte.
- Pas d'isolement entre les conteneurs ou entre les conteneurs et l'hôte.

None

- Fournit une ***isolation sandbox,***

- Aucune implémentation pour la mise en réseau.

- Offre une pile réseau spécifique au conteneur qui n'a pas d'interface réseau.

- N'a qu'une interface de bouclage locale (c'est-à-dire ? aucune interface réseau externe).

Initialisation le service Swarm sur le master node.

docker swarm init

Sur un autre hôte, comment rejoindre le cluster swarm ?

docker swarm join

Une fois cela fait, vous pouvez créer le pilote réseau Overlay sur le nœud maître.

Comment créer un réseau Overlay ?

docker network create -d overlay my-overlay

IPVLAN

● Le pilote **IP vlan** donne aux utilisateurs un contrôle total sur l'adressage **IPv4** et **IPv6**.

● Cela signifie que l'utilisateur a le contrôle du balisage **VLAN** de couche 2 et même du routage **IP vlan** L3 pour les utilisateurs intéressés par l'intégration du réseau sous-jacent.

● C'est la raison pour laquelle **IPVLAN** est une véritable technique de virtualisation de réseau.

● Pour les autres pilotes réseau, nous utilisons des sous-interfaces appelées **VERTH**. Dans **IPVLAN**, nous attachons le conteneur directement à l'interface hôte Docker, comme eth0, etc.

MACVLAN

● Certaines applications qui surveillent le trafic réseau nécessitent une connexion directe. Cela signifie qu'elles doivent avoir

une adresse **MAC** unique.

● Dans ce type de situation, vous pouvez utiliser le pilote réseau macvlan pour attribuer une adresse MAC à chaque

interface réseau virtuelle du conteneur, ce qui lui donne l'apparence d'une interface réseau physique

directement connectée au réseau physique.

● Toutes les fonctionnalités d'IPVLAN sont adoptées par MACVLAN. Dans Macvlan, tous les conteneurs auront une

adresse MAC dédiée pour le monde extérieur.

● Ainsi, la pratique du MACVLAN n'est pas nécessaire.

Docker Volumes

- Tous les fichiers créés dans un conteneur sont stockés sur une couche de conteneur accessible en écriture.

- Les données ne persistent pas lorsque ce conteneur n'existe plus.

- La couche accessible en écriture d'un conteneur est étroitement couplée.

- Essentiel lorsque nous voulons assurer la persistance des données, partager des données entre conteneurs ou gérer des données qui doivent survivre au redémarrage et aux suppressions de conteneurs.

- Docker dispose d'options permettant aux conteneurs de stocker des fichiers sur la machine hôte, afin qu'ils puissent être conservés même après l'arrêt/la suppression du conteneur :

(Volume nommé, montages de liaison).

Types de volumes - Volumes nommés

- Les volumes nommés sont stockés sur l'hôte Docker à (**/var/lib/docker/volumes/ sous Linux**).

- Peuvent être utilisés par n'importe quel conteneur sur cet hôte.

- Fournissent un moyen pratique de partager des données entre plusieurs conteneurs exécutés sur le même hôte.

- Les processus non Docker ne doivent pas modifier cette partie du système de fichiers.

Types de volumes - Montages de liaison

- Permet de monter un répertoire hôte dans un conteneur.

- Utile pour partager des données entre l'hôte et le conteneur, permettant un accès facile aux fichiers et répertoires sur le système hôte depuis le conteneur.

- Les processus non Docker sur l'hôte Docker ou un conteneur Docker peuvent les modifier à tout moment.

Comparaison

#	Named Volumes	Bind Mounts
Host Location	Docker chooses	You control
Mount Example(using -v)	my-volume:/usr/local/data	/path/to/data:/usr/local/data
Populates new volume with container contents	Yes	No
Supports Volume Drivers	Yes	No

Avantages des volumes

- Persistance des données
- Partage de données entre conteneurs
- Simplification du développement d'applications

Commands

- Créer | Lister | Inspecter | Supprimer

```
docker volume create my-vol
docker volume ls
docker volume inspect my-vol
docker volume rm [VOLUME_NAME]
docker volume rm $(docker volume ls -q)
```

Monter | -v | service

```
docker run -d --name devtest -p 80:80 --mount source=myvol11, target=/usr/share/nginx/html nginx:latest
docker run -d --name devtest2 -p 8080:80 -v myvol11:/usr/share/nginx/html nginx:latest
docker run -d --name devtest3 -p 8081:80 -v myvol11:/usr/share/nginx/html:ro nginx:latest
docker exec -it devtest2 /bin/bash
echo 'Hi Cfitech' > /usr/share/nginx/html/index.html
```

- Bind Mount Volume

```
mkdir cfietch
echo "Hello from CFITECH" >> cfietch/index.html
docker run -d -it --name testdevt -p 85:80 --mount type=bind, source="$(pwd)"/cfietch,
tartget=/usr/share/nginx/html nginx
```

Mount | Read only

```
docker run -d --name=ng --mount source=nginx-vol, destination=/usr/share/nginx/html
nginx:latest
docker run -d --name=ng -v nginx-vol:/usr/share/nginx/html:ro nginx:latest
```

Backup volume

```
docker run -v /dbdata --name dbstore ubuntu /bin/bash
docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata
```

Docker Compose

Environnement déclaratif avec Docker Compose

Définition des services

Définition des volumes

Définition des réseaux

Docker Compose CLI

Environnement déclaratif avec Docker Compose

Compose pour les piles de développement

Les fichiers Docker sont parfaits pour créer des images de conteneurs.

Mais que se passe-t-il si nous travaillons avec une pile complexe composée de plusieurs conteneurs ?

Finalement, nous voudrions écrire des scripts personnalisés et des automatisations pour créer, exécuter et connecter nos conteneurs ensemble.

Il existe une meilleure façon : utiliser Docker Compose.

Compose est un outil permettant de définir et d'exécuter des applications Docker multi-conteneurs.

Docker Compose permet de définir et de coordonner plusieurs conteneurs.

L'idée générale de Compose est de permettre un flux de travail d'intégration très simple et puissant :

Vérifiez votre code.

Exécutez docker-compose up.

Votre application est opérationnelle !

Présentation de Compose

Voici comment travailler avec Compose :

Vous décrivez un ensemble (ou une pile) de conteneurs dans un fichier YAML appelé

docker-compose.yml.

Vous pouvez utiliser une extension .yml ou .yaml pour ce fichier. Ils fonctionnent tous les deux

Vous exécutez docker-compose

Compose récupère automatiquement les images, crée les conteneurs et les

démarre.

Compose peut configurer des liens, des volumes et d'autres options Docker pour vous.

Compose peut exécuter les conteneurs en arrière-plan ou au premier plan.

Lorsque les conteneurs s'exécutent au premier plan, leur sortie agrégée est affichée

Vérifier si Compose est installé

Si vous utilisez Docker pour Mac/Windows ou Docker Toolbox,

Compose est fourni avec eux.

Si vous utilisez Linux (environnement de bureau ou serveur), vous

devrez installer Compose à partir de sa page de publication ou avec pip install

docker-compose.

Vous pouvez toujours vérifier qu'il est installé en exécutant :

> ***docker-compose --version***

Lancement de notre première pile avec Compose

Voici un exemple de fichier Docker Compose
Pour démarrer l'application :
> **docker-compose up**

```yaml
services:
 db:
   image: mysql
   container_name: mysql_db
   restart: always
   environment:
     - MYSQL_ROOT_PASSWORD=secret
     - MYSQL_PASSWORD=secret
   networks:
     - mynetwork
   volumes:
     - myNamedVolume:/var/lib/mysql

 web:
   image: apache
   build: ./webapp
   depends_on:
     - db
   container_name: apache_web
   restart: always
   ports:
     - "8080:80"
   networks:
     - mynetwork
```

Structure du fichier Compose

Version est obligatoire.

Nous devons utiliser « 2 » ou une version ultérieure ; la version 1 est obsolète. Services est obligatoire.

Service est une ou plusieurs répliques de la même image exécutées en tant que conteneurs.

Networks est facultatif et indique à quels réseaux les conteneurs doivent être connectés.

Par défaut, les conteneurs seront connectés sur un réseau privé, par fichier de composition.

Volumes est facultatif et peut définir les volumes à utiliser et/ou partagés par les conteneurs.

Versions du fichier Compose

La version 1 est une version héritée et ne doit pas être utilisée.

(Si vous voyez un fichier Compose sans version ni services, il s'agit d'un fichier v1 hérité.)

La version 2 a ajouté la prise en charge des réseaux et des volumes.

La version 3 a ajouté la prise en charge des options de déploiement (mise à l'échelle, mises à jour continues, etc.).

La documentation Docker contient d'excellentes informations sur le format de fichier Compose si vous avez besoin d'en savoir plus sur les versions.

Yaml pour Docker Compose

Le format YAML est censé être lisible par l'homme et pratique à saisir. Et c'est le cas.

C'est le plus populaire par rapport aux formats JSON ou XML pour une raison les fichiers peuvent se terminer par ".yml" ou "yaml"

Tabs And Spaces

Pour les identifiants, vous pouvez utiliser des espaces ou des tabulations.

Restez fidèle à l'un de ces deux formats pour les retraits (vous pouvez les mélanger, mais ce n'est pas conseillé)

The Strings.

" et ' fonctionnent tous les deux. Et dans la plupart des cas, les deux manières d'écrire des chaînes sont

équivalentes pour la plupart des raisons pratiques (les chaînes entre guillemets peuvent contenir

des caractères d'échappement).

```yaml
version: "3"

annot: >
       a string written
       in folded style
# will become "a string written in folded style\n"
```

Yaml pour Docker Compose

Clés, valeurs et blocs

Les fichiers YAML sont constitués de clés qui sont utilisées pour accéder aux valeurs attribuées.

Une clé peut avoir une valeur unique comme un entier (5), une chaîne (« hi »), une liste (« hi », « there ») ou un dictionnaire (ensemble de mappages clé-valeur).

```yaml
# A single, boring value
immastring: "3"
immanint: 3

# A list
immalist:
 - "hi"
 - "there"

# Also a list (inline)
immalist2: ["hi", "there"]

# A dictionary
immadict:
 akey: "a value"
 anotherkey: "another value"

# Another dictionary (inline)
immadict2: {akey: "a value", anotherkey: "another value"}
```

Yaml pour Docker Compose

Les clés, les valeurs et les blocs peuvent être combinés

Voici une liste de dictionnaires :

```
weirdlist:
- key1: "hi"
key2: "there"
- key1: "hi2"
key2: "there2"
# You could write it like this
weirdlist: [{key1: "hi", key2: "there"}, {key1: "hi2", key2: "there2"}]
```

Définition des services

Chaque service dans le fichier YAML doit contenir soit build, soit image.

build indique un chemin contenant un Dockerfile.

image indique un nom d'image (local ou sur un registre).

Si les deux sont spécifiés, une image sera construite à partir du répertoire build et de l'image nommée.

context

Contient soit un chemin vers un répertoire contenant un Dockerfile, soit une URL vers un référentiel git.

args

Ajouter des arguments de build, qui sont des variables d'environnement accessibles uniquement

pendant le processus de build.

labels

Ajouter des métadonnées à l'image résultante

Définition de service

Une définition de service contient une configuration qui est appliquée à chaque

conteneur démarré pour ce service, un peu comme la transmission de paramètres de ligne de commande à docker run.

De même, les définitions de réseau et de volume sont analogues à

docker network create et docker volume create.

Comme avec docker run, les options spécifiées dans le Dockerfile, telles que

CMD, EXPOSE, VOLUME, ENV, sont respectées par défaut - vous n'avez pas besoin de les spécifier à nouveau dans docker-compose.yml.

Vous pouvez utiliser des variables d'environnement dans les valeurs de configuration avec un ${VARIABLE} de type Bash.

Paramètres du conteneur

La commande indique ce qu'il faut exécuter (comme CMD dans un Dockerfile).

Les ports se traduisent par une (ou plusieurs) option(s) -p pour mapper les ports.

Vous pouvez spécifier des ports locaux (c'est-à-dire x:y pour exposer le port public x).

Les volumes se traduisent par une (ou plusieurs) option(s) -v.

Vous pouvez utiliser des chemins relatifs ici.

Volumes montés sur l'hôte

Syntaxe : /host/path:/container/path

Le chemin de l'hôte peut être défini comme un chemin absolu ou relatif.

```
version: '3'
services:
 app:
   image: nginx:alpine
   ports:
     - "80:80"
   volumes:
     - /var/opt/my_website/dist:/usr/share/nginx/html:ro
```

Volumes nommés Docker / Volumes nommés internes

Les volumes nommés peuvent être définis comme internes (par défaut) ou externes.

Les volumes nommés internes Docker compose ont la portée d'un fichier Docker-compose unique et Docker les créer s'ils n'existent pas

Syntaxe : named_volume_name:/container/path

```yaml
version: '3'
volumes:
 web_data:
services:
 app:
   image: nginx:alpine
   ports:
     - "80:80"
   volumes:
     - web_data:/usr/share/nginx/html:ro
```

À partir de la version 3.4 de Docker Compose, le nom du volume peut être

généré dynamiquement à partir de variables d'environnement placées dans un fichier .env

(ce fichier doit se trouver dans le même dossier que docker-compose.yml).

Volumes nommés Docker / Volumes nommés externes

Les volumes nommés externes Docker Compose peuvent être utilisés

dans l'installation Docker et ils doivent être créés par l'utilisateur

(sinon, ils échouent) à l'aide de la commande docker volume create.

La même syntaxe en yaml que les volumes internes

Exemple :

Définit le volume web_data :

```
docker volume create --driver local \
        --opt type=none \
        --opt device=/var/opt/my_website/dist \
        --opt o=bind web_data
```

docker-compose.yml

```
version: '3'
volumes:
 web_data:
   external: true
services:
 app:
   image: nginx:alpine
   ports:
     - "80:80"
   volumes:
     - web_data:/usr/share/nginx/html:ro
```

Networking dans Compose

Par défaut, Compose configure un réseau unique pour votre application.

Chaque conteneur d'un service rejoint le réseau par défaut et est à la fois accessible

par d'autres conteneurs sur ce réseau et détectable par eux sur un

nom d'hôte identique au nom du conteneur.

Par exemple, supposons que votre application se trouve dans un répertoire appelé myapp et que votre

docker-compose.yml ressemble à ceci :

```
version: "3"
services:
 web:
   build: .
   ports:
     - "8000:8000"
 db:
   image: postgres
   ports:
     - "8001:5432"
```

Lorsque vous exécutez **docker-compose up**, voici ce qui se passe :

1. Un réseau appelé **myapp_default** est créé.

2. Un conteneur est créé à l'aide de la configuration de web. Il rejoint le réseau **myapp_default** sous le nom web.

3. Un conteneur est créé à l’aide de la configuration de db. Il rejoint le réseau **myapp_default** sous le nom db.

Chaque conteneur peut désormais rechercher le nom d'hôte web ou db et récupérer

l'adresse IP du conteneur approprié.

Par exemple, le code d'application web peut se connecter à l'URL **postgres://db:5432** et commencer à utiliser la base de données Postgres.

Liens

Les liens vous permettent de définir des alias supplémentaires par lesquels un service est accessible

à partir d'un autre service.

Ils ne sont pas nécessaires pour permettre aux services de communiquer - par défaut, tout

service peut accéder à tout autre service au nom de ce service.

Dans l'exemple suivant, db est accessible à partir de web aux noms d'hôte db et

database :

```yaml
version: "3"
services:
 web:
   build: .
   depends_on:
     - db
 db:
   image: postgres
```

Spécifier des réseaux personnalisés

Au lieu d'utiliser simplement le réseau d'application par défaut, vous pouvez spécifier vos propres réseaux avec la clé de réseaux de niveau supérieur.

Cela vous permet de créer des topologies plus complexes et de spécifier des pilotes et des options réseau personnalisés.

Chaque service peut spécifier à quels réseaux se connecter avec la clé de réseau de niveau de service, qui est une liste de noms référençant des entrées sous

la clé de réseau de niveau supérieur.

```yaml
version: "3"
services:
  proxy:
    build: ./proxy
    networks:
      - frontend

  app:
    build: ./app
    networks:
      - frontend
      - backend

  db:
    image: postgres
    networks:
      - backend

networks:
  frontend:
    # Use a custom driver
    driver: custom-driver-1

  backend:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

Docker Compose CLI

Nous avons déjà vu docker-compose up

Il existe une autre commande : docker-compose build.

Elle exécutera docker build pour tous les conteneurs mentionnant un chemin de build.

Elle peut également être invoquée automatiquement au démarrage de l'application :

> docker-compose up --build

Une autre option courante consiste à démarrer les conteneurs en arrière-plan :

> docker-compose up -d

Mise à l'échelle via la CLI docker-compose

> docker-compose up -d --scale web=5

Vérifier l'état du conteneur

Il peut être fastidieux de vérifier l'état de vos conteneurs avec docker ps, en particulier lorsque vous exécutez plusieurs applications en même temps.

Compose facilite les choses ; avec docker-compose ps, vous ne verrez

que l'état des conteneurs de la pile actuelle :

Mettre à l'échelle les conteneurs

Aussi simple que de spécifier le nombre d'instances dans le fichier yaml

$ docker-compose scale web=3

Nettoyage

Si vous avez démarré votre application en arrière-plan avec Compose et que vous souhaitez l'arrêter facilement, vous

pouvez utiliser la commande kill :

> docker-compose kill

De même, docker-compose rm vous permettra de supprimer des conteneurs (après confirmation).

Alternativement, docker-compose down arrêtera et supprimera les conteneurs.

Il supprimera également d'autres ressources, comme les réseaux qui ont été créés pour l'application.

Utilisez docker-compose down -v pour tout supprimer, y compris les volumes.

Quelques commandes Docker

```
$ docker version
$ docker images
$ docker pull <image_name>
$ docker run ubuntu
$ docker image ls
$ docker run -it ubuntu
root@9e7b5ad0ff35:/# ls
root@9e7b5ad0ff35:/# echo "This is our
first file in container" >
/tmp/file1.txt
root@9e7b5ad0ff35:/#  cat
/tmp/file1.txt
root@9e7b5ad0ff35:/#  exit
```

```
$ docker image ls
$ docker run --name=web1 -d -p 8400:80 nginx
$ docker ps
$ docker container inspect <CONTAINER  ID>
$ curl <IP Address>
$ docker container ls
$ docker container ls -a
$ docker exec -it web1 bash
$ docker exec -it <CONTAINER ID> cat /tmp/file1.txt
$ docker exec -it <CONTAINER ID> date
$ docker exec -it <CONTAINER ID> hostname
$ docker run --name=web2 -d -p 8500:80 nginx
$ docker ps || $ docker ps -a
$ docker pull httpd
$ docker run --name=web3 -d -p 8600:80
engineerbaz/htmlfile:2
$ docker container inspect web3
$ curl <IP Address>
$ docker run --name=web4 -d -p 8700:80
engineerbaz/htmlfile:5
$ curl <IP Address>
```

Docker Commandes en profondeur (Images)

```
# List images ls
docker image ls
# Retag existing image
docker image tag nginx
btraversy/nginx
# Build image from DockerFile
docker image build -t [REPONAME] .
# Add tag to push to Dockerhub
docker image tag nginx-website:latest
btraversy/nginx-website:latest
# Upload image to docker hub
docker image push bradtraversy/nginx
# Login to Dockerhub
docker login
```

```
# Add tag to new image
docker image tag bradtraversy/nginx
bradtraversy/nginx:testing
# Pull down mysql image to test
docker pull mysql
# Inspect and see image
docker image inspect mysql
# Visualise docker image history
docker history image_name
# Search for an image on Dockerhub
docker search mysql
```

Docker Containers

```
# Create and run a container from an image, with a
custom name:
$ docker run --name [container] [image name]
# Run a container with and publish a container's
port(s) to the host
$ docker run -p
[host port]:[container port][image name]
# Run a container in the background
$ docker run -d <image name>
# Stop all running containers:
docker container stop $(docker container ls
-aq)
# Remove all running containers:
docker container rm $(docker container ls -aq)
# Stop and remove all containers:
docker stop $(docker container ls -aq) &&
docker rm $(docker container ls -aq)
```

```
# Stop and remove all containers
(shorthand):
docker rm -f [container name]
#Remove all caches images:
docker rmi $(docker images -a -q)
# Get interactive shell on specified
container
docker exec -it [container name or
container ID] bash
# Rename container
docker rename old name new name
# Fetch and follow the logs of a container
$ docker logs -f [container name]
# To inspect a running container
$ docker inspect [container name] ||
[container id]
```

Docker Networks

```
# Create network
docker network create [NETWORK_NAME]
# Create container on network
docker container run -d --name [NAME] --network [NETWORK_NAME] nginx
# Connect existing container to network
docker network connect [NETWORK_NAME] [CONTAINER_NAME]
# Disconnect existing container to network
docker network disconnect [NETWORK_NAME] [CONTAINER_NAME]
# List all Networks
docker network ls
# Inspect Network
docker network inspect [NETWORK_NAME]
# Remove unused Networks
docker network prune
# Get All Instances IP Addresses
docker ps -q | xargs -n 1 docker inspect --format '{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}} {{ .Name }}'| sed 's/ \// /'
```

Docker Volumes

```
# List all Volumes
docker volume ls
# Remove specific Volume
docker volume rm [VOLUME_NAME]
# Remove all unused Volume
docker volume rm $(docker volume ls -q)
```

Docker compose

```
# To run docker-compose
docker-compose up
# Run in background
docker-compose up -d
# down
docker-compose down
```

Docker Hub

```
# Login into Docker
$ docker login -u <username>

# Publish an image to Docker Hub
$ docker push <username>/<image_name>

# Search Hub for an image
$ docker search <image_name>

# Pull an image from a Docker Hub
$ docker pull <image_name>
```

Autres ressources

https://github.com/LeCoupa/awesome-cheatsheets/blob/master/tools/docker.sh