

Introduction au DevOps

Docker

Jolan PHILIPPE

12 Septembre 2025

Université d'Orléans

Les rôles des outils d'IaC

Management d'application

Customiser, configurer
tester l'application
et la **conteneuriser**

Provisionnement d'infrastructure

Demander ressources
physiques ou virtuelles;
configurer le réseau;
et règles sécurité

Installation et Configuration

Installer les services
(app + deps)
configurer les services;
et les intégrer

Orchestration de cycle de vie

Upgrades auto;
Backup et recovery;
Surveillance;
Passage à l'échelle

<https://www.docker.com/blog/docker-for-devops/>

Un conteneur c'est quoi ?

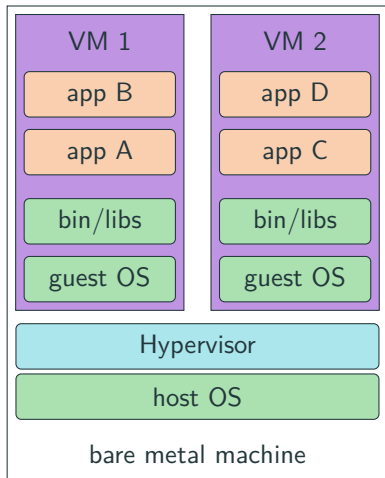
A propos du kernel Linux

Le kernel c'est le cœur d'un système d'exploitation (OS)

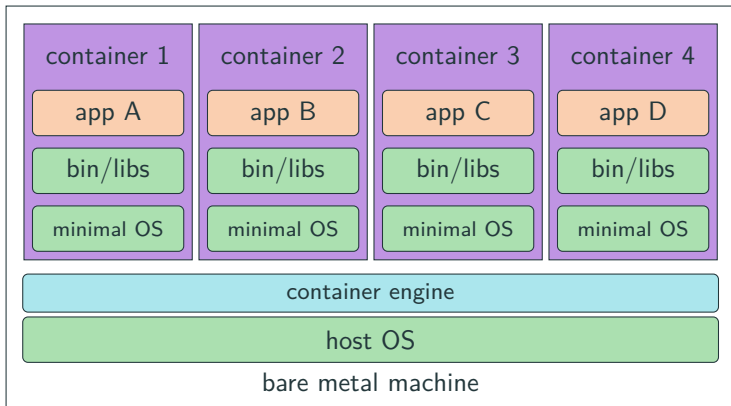
- C'est une partie de l'OS toujours chargé en mémoire.
- Il contrôle les ressources physiques, ou hardware, (e.g., I/O, mémoire, cryptographie, CPU) en utilisant des pilotes, ou drivers.
- Il arbitre les conflits et la concurrence entre les processus.
- Il optimise l'utilisation de ressources (e.g., cache, mémoire, CPU, système de fichiers, réseau)

Le kernel est l'un des premiers programmes chargé au démarrage.

Comparaison à gros grains entre les VMs et les conteneurs



VMs avec un hyperviseur



Containers

Avantages des conteneurs

- Isolation
- Portabilité
- Limitations des ressources dupliquées
- Impact limité sur les performances
- Démarrage rapide

Le modèle est différent et la façon dont les applications sont déployées est différente !

Que se cache-t-il dessous ?

Un peu plus à propos de Linux

Dans Unix/Linux, tout est fichier : un fichier, un dossier, un périphérique, etc.

Système de fichiers

- Organisation des fichiers hiérarchique sous la forme d'un arbre
- / est la racine du système de fichiers
- /sys contient les fichiers du système
- /etc contient les fichiers de configuration et les scripts
- /media contient les partitions de disques, périphériques, etc.
- etc.

Un conteneur est aussi un ensemble de fichiers

Une image de conteneur est une archive de fichiers, incluant:

- une racine de système de fichiers
- des bibliothèques, des packages, etc. (i.e., des dépendances)
- l'application, ou le service, à démarre
- etc.

L'image contient l'environnement requis pour exécuter l'application ou le service sur le noyau hôte.

Cet environnement est **portable** d'un hôte à un autre, si un kernel compatible est présent (**WSL2 sur Windows**, ou **VM sur MacOS**)

Attention: La limite, c'est l'architecture matérielle : par exemple une image construite pour amd64 ne tournera pas nativement sur un hôte arm64 (Mac M1/M2) sans émulation (e.g., qemu).

- Les images Docker sont stockées dans des **registres** (e.g., *Docker Hub* <https://hub.docker.com/>).
- La commande `docker pull` permet de **télécharger une image** vers la machine locale.
- Exemple :

```
> docker pull MyImage
```
- Résultat :
 - Télécharge l'image `MyImage`
 - Rend l'image disponible localement pour créer des conteneurs.

Example: Alpine

L'image docker de Linux Alpine est souvent utilisée par les conteneurs.

- Distribution très légère de Linux
- https://hub.docker.com/_/alpine

Télécharger une image alpine

```
> docker pull alpine
```

Démarre un conteneur en utilisant l'image alpine et démarre une invite de commande sh interactive dans celui-ci

```
> docker run -it alpine /bin/sh
```

Une fois démarrer, on peut interagir avec le système de fichier:

```
in alpine > ls -al
```

Mais il n'y a pas de kernel:

```
in alpine > ls /boot
```

D'une image vers un conteneur

Comme vu précédemment, une image est une archive d'un système de fichiers. Créer un conteneur consiste à:

- Donner une **quantité limitée de ressources** au conteneur
- Créer un environnement isolé au processus du conteneur
- assigner la racine du système de fichiers de l'image à la **la racine du système de fichiers du conteneur**

D'une image vers un conteneur

`cgroups`

Linux Control Groups (`cgroups`) limite la quantité de ressource qu'un processus peut utiliser (CPU, mémoire, latence, etc).

`namespaces`

Linux Namespaces assure que chaque processus voit sa propre vue personnelle du système (fichiers, processus, interfaces réseaux, hostname, etc).

`chroot`, `pivot_root`

Change la racine du système de fichiers d'un processus.

Docker

Vue d'ensemble de Docker

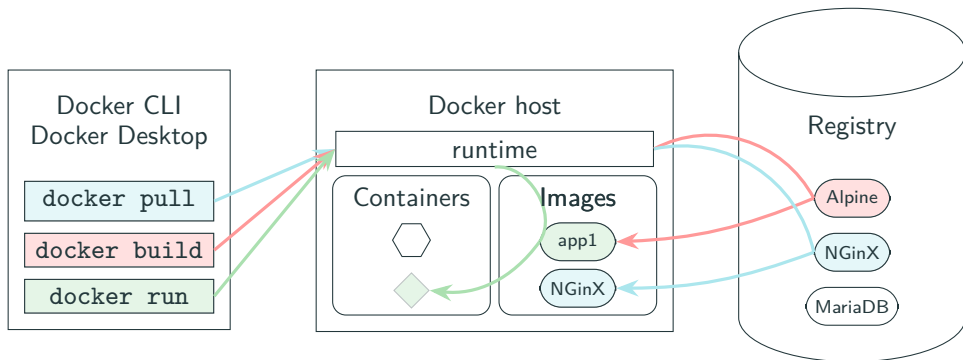
Les différents éléments de Docker

- CLI (Command Line Interface)
- Docker runtime
- Les images et le registre
- Les conteneurs

Les rôles du Docker runtime

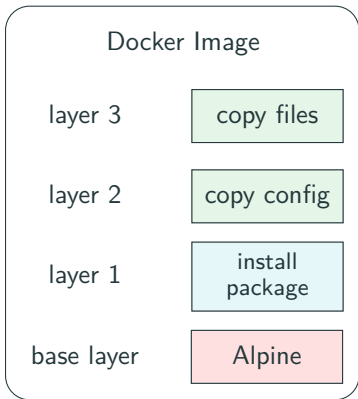
- Démarrer (*start*) et arrêter (*stop*) des conteneurs
- Manager des images
- Manager les réseaux
- Manager les volumes
- etc.

Vue d'ensemble de Docker



Structure d'une image Docker

Une image Docker est construite en assemblant différentes **couches**.



Optimisation de stockage

- Les couches sont partagées par différentes images pour optimiser le stockage;
- Pour se faire, chaque couche est identifiée par une fonction de hashage.

Écrire dans un conteneur ?

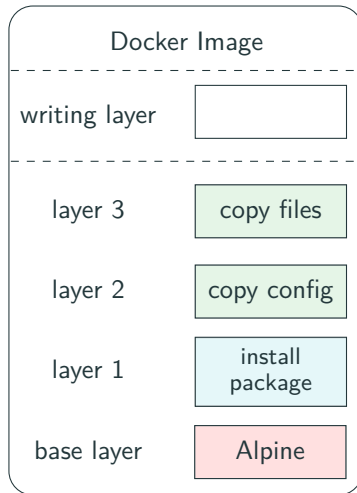
Une image Docker est **immuable**! C'est à dire qu'elle ne peut pas être modifiée.

Au runtime, une couche virtuelle est créée dans le conteneur, au dessus de l'image

- Il est possible d'écrire dans cette couche;
- Cette couche n'est pas partagée par les autres conteneurs;
- Cette couche est détruite avec le conteneur.

Volumes

Si les données doivent être **persistantes et partagées** entre les conteneurs, un **volume** doit être utilisé. Voir <https://docs.docker.com/engine/storage/volumes/>



Créer une image avec un Dockerfile

Un *Dockerfile* contient un ensemble de commandes pour créer une image Docker.

- Empêche de construire des images manuellement, “from scratch”;
- Offre une manière pour Docker de construire des couches et empêcher les commandes inutiles;
- Un Dockerfile ressemble à un fichier bash avec des instructions à appliquer.

Exemple de Dockerfile

- FROM pour indiquer quelle image de base utiliser pour construire notre image;
- RUN pour exécuter une commande par dessus l'image de base;
- ENV pour déclarer des variables d'environnement;
- ENTRYPOINT pour définir quelle commande doit être exécutées au démarrage du conteneur.

La documentation complète est accessible ici:

<https://docs.docker.com/reference/dockerfile/>

```
1 FROM alpine
2 RUN apt update
3 RUN apt install -y htop
4 ENV TERM=xterm
5 ENTRYPOINT /bin/htop
```

Exemple de Dockerfile

- FROM avec une version d'image;
- WORKDIR pour indiquer le dossier de travail dans lequel se placer quand le conteneur démarrera;
- ADD pour ajouter des fichiers depuis la machine locale (i.e., hôte) à l'image du conteneur;
- CMD pour définir quelles commandes doivent être exécutées au démarrage du conteneur.

```
1 FROM python:3.8-alpine
2 RUN apt updateWORKDIR /app
3 ADD . /app/
4 RUN pip install -r requirements.txt
5 CMD ["python", "movie.py"]
```

Quelques subtilités des commandes Dockerfile

- `ADD` \neq `COPY`: `ADD` fait la même chose que `COPY` mais permet aussi de décompresser automatiquement des archives locales, et télécharger une ressource depuis une URL
- `ENTRYPOINT` \neq `CMD`: `ENTRYPOINT` sert à démarrer un programme principal fixe (e.g., `python`, `java`) alors que `CMD` sert à remplacer les arguments passé après `docker run image` (e.g., `/bin/sh` dans le premier exemple)

Construire une image à partir d'un Dockerfile

Commande

```
docker build [OPTIONS] PATH
```

```
> docker build . -t "monapp:latest"
```

- `docker build` est la commande pour construire une image Docker;
- `.` est le chemin vers le Dockerfile;
- `-t` est une option pour nommer cette image (ici `"monapp:latest"` se réfère à un nom et une version);
- Par défaut, le Dockerfile est `PATH/Dockerfile`, mais vous pouvez donner un autre nom en utilisant l'option `-f`.

Docker CLI

- > `docker -h` donne l'aide racine de la CLI
- > `docker images -h` donne l'aide pour les images

- > `docker pull hello-world` télécharge l'image `hello-world`
- > `docker images`, ou `docker images ls`, donne une liste des images téléchargées
- > `docker image inspect id` donne des détail sur l'image correspondante à l'id

- > `docker run hello-world` créer et démarre un conteneur à partir d'une image
- > `docker ps` affiche les conteneurs en cours d'exécution
- > `docker ps -a` affiche tous les conteneurs, y compris ceux aux statut `exited`
- > `docker container inspect id` donne des détail sur le conteneur correspondant à l'id

- > `docker container prune` supprime tous les conteneurs qui ne sont plus utilisés
- > `docker image rm hello-world` supprime l'image locale `hello-world`

Quelques bonnes pratiques

Dans les versions plus anciennes de Docker, chaque ligne dans le Dockerfile créait une couche.
En conséquences:

- Trop de couches intermédiaires, pouvant être coûteuses;
- Le temps de build pouvait être impacté;
- Les optimisations de stockage pouvaient être impossibles;
- Aujourd'hui, `RUN`, `COPY` et `ADD` uniquement créent de nouvelles couches.

Bonne pratique 1

Pensez à vos couches quand vous utiliser `RUN`, `COPY` et `ADD` dans votre Dockerfile

Bonne pratique 2

N'installez uniquement que les dépendances nécessaires dans votre Dockerfile

- Si vous utilisez `apt` pour installer des packages, utilisez l'option `--no-install-recommends`
- Si possible, supprimez les fichiers intermediaires non requis quand vous appliquez `RUN`

Construction en plusieurs étapes

Bonne pratique 3

- Réduisez la taille des images en supprimant les dépendances de compilation dans l'image finale.
- L'image finale contient uniquement les dépendances nécessaires à l'exécution du service.
- Une image de base bien adaptée aux fichiers exécutables uniquement est `scratch` ou `alpine`.

```
1  FROM golang as builder
2  RUN apt update && apt install -y git protobuf-compiler golang-goprotobuf-def && \
3      git clone https://gitlab.imt-atlantique.fr/url && \
4      cd productcatalogservice && \
5      go mod download && \
6      mkdir genproto && \
7      protoc --go_out=plugins=grpc:genproto -I . productcatalogservice.proto && \
8      CGO_ENABLED=0 go builder
9
10 FROM scratch
11 COPY --from=builder /go/productcatalogservice/productcatalogservice /
12 COPY --from=builder /go/productcatalogservice/products.json /
13 ENTRYPOINT ["/productcatalogservice"]
```

N'importe qui peut déposer une image Docker sur Docker Hub !

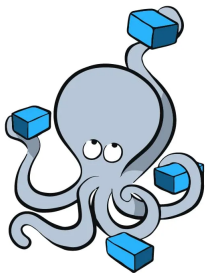
Bonne pratique 4

- Préférez toujours les images Docker officielles.
- Vérifiez que l'image Docker est régulièrement mise à jour.
- Soyez sûr que l'image contient ce que vous pensez en utilisant
 - `> docker history image_name`
 - des outils comme [dive](#)
- Soyez sûr de mettre à jour les images que vous utilisez.

Un peu plus dans les Dockerfiles

- Exposer ses ports dans un Dockerfile
 - `EXPOSE 80` pour exposer un numéro de port
 - `EXPOSE 56/udp` pour exposer un numéro de port pour un protocole précis
- Ajouter des données purement informatives
 - `LABEL maintainer="Jolan PHILIPPE"`
- Ajouter des variables d'environnement
 - `ENV ADMIN_USER="Alice"`
 - ou directement via la CLI: `docker run -e ADMIN_USER="Bob"`
- Ajouter des volumes depuis l'hôte
 - `VOLUME /myapp/data`

Déployer une pile logicielle avec Docker Compose



docker
Compose

- Déploie facilement une pile logicielle conteneurisée
- Défini votre déploiement avec un seul fichier YAML (conteneurs, volumes, réseaux, etc.)
- Les fichiers de déploiement deviennent faciles à partager, à contrôler leurs versions, etc. (c'est de l'IaC :wink:)

Structure de compose.yaml

Spécification complète: <https://docs.docker.com/reference/compose-file/>

- **services**

- Nom du service

- **image** Docker ou chemin de **build** pour le Dockerfile
 - **ports** exposés par le service
 - **networks** utilisés par le service
 - **volumes** utilisés par le service
 - variables d'**environment** utilisés par le service avec sa valeur
 - **depends_on** un autre service

- **volumes**

- **networks**

```
1  version: '3.3'
2
3  services:
4    lychee:
5      image: linuxserver/lychee:4.7.0
6      container_name: Gdsn-photos-web
7      restart: always
8      networks:
9        - web
10       - default
11      volumes:
12        - ./conf:/config
13        - /srv/gdsn_photos_lychee:/pictures
14      labels:
15        - "traefik.http.routers.gdsn_photos.rule=Host(`photos.gdsn.fr`)"
16
17    db:
18      image: mysql:5.7
19      container_name: Gdsn-photos-db
20      restart: always
21      networks:
22        - default
23      volumes:
24        - /srv/gdsn_photos_db_data:/var/lib/mysql
25      environment:
26        MYSQL_ROOT_PASSWORD: secret
27        MYSQL_DATABASE: lychee
28        MYSQL_USER: lychee
29        MYSQL_PASSWORD: secret
30      labels:
31        - "traefik.enable=false"
32
33  networks:
34    web:
35      name: traefik_web
36      external: true
```

Il est très important de comprendre que Docker compose crée un DNS pour que les conteneurs puissent s'appeler les uns les autres sans connaître leurs adresses IP.

Quelques commandes importantes :

- > `docker-compose build` pour construire, ou reconstruire, des images
- > `docker-compose up` pour créer et démarrer des conteneurs, réseaux, etc.
- > `docker-compose stop` pour arrêter des conteneurs, réseaux, etc.
- > `docker-compose down` pour arrêter et supprimer des conteneurs, réseaux, etc.

Toute la documentation : <https://docs.docker.com/compose/reference/>

Quelques exemples : <https://github.com/docker/awesome-compose>