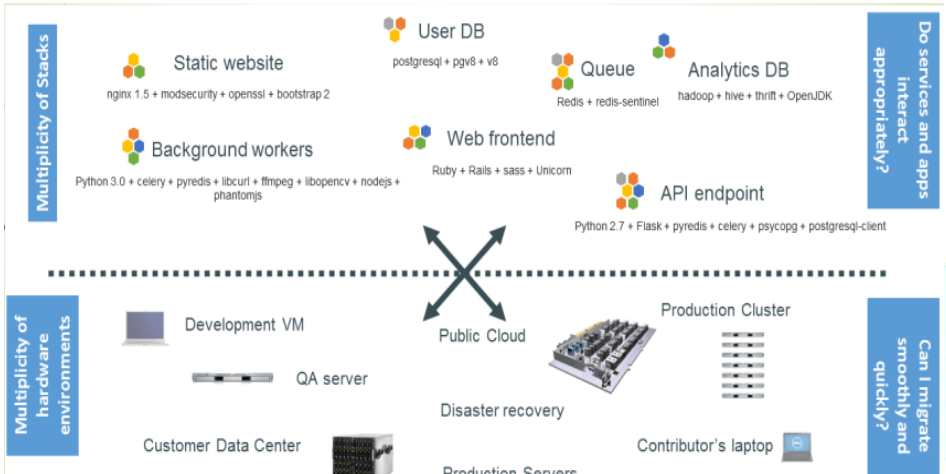
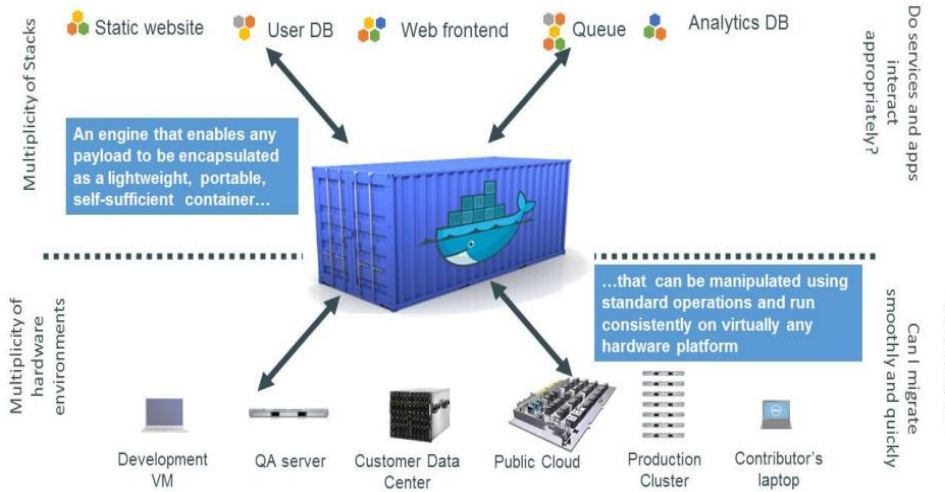
Docker & Container

Pourquoi les conteneurs ??



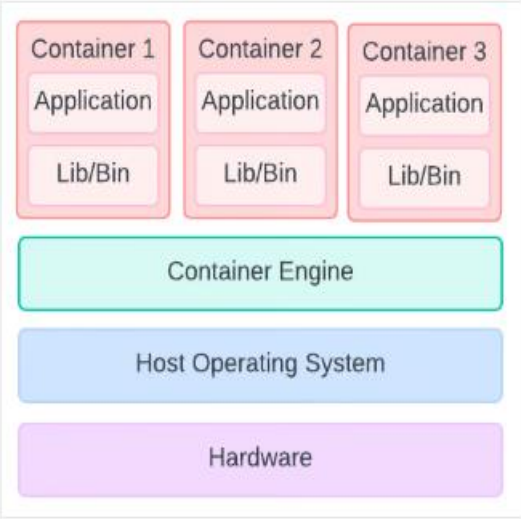


Qu'est-ce qu'une machine virtuelle ?

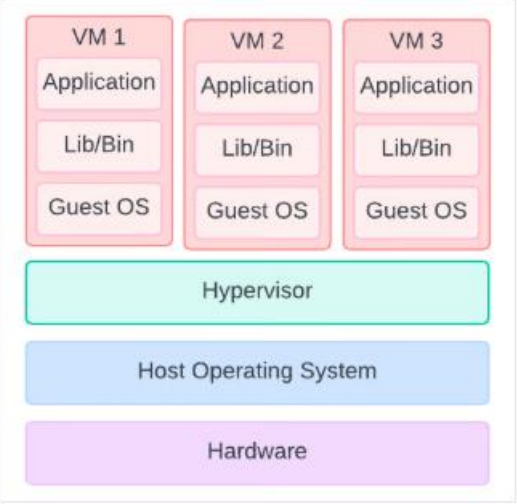
Une machine virtuelle (VM) est une technologie qui permet la virtualisation au niveau du matériel, ce qui permet à plusieurs systèmes d'exploitation de fonctionner sur une seule machine. Chaque VM agit comme un système isolé avec son propre système d'exploitation, sa propre application et ses propres dépendances

Qu'est-ce qu'un conteneur ?

Un conteneur est une forme de virtualisation qui opère au niveau du système d'exploitation, permettant à plusieurs applications de fonctionner sur le même noyau de système d'exploitation



conteneur

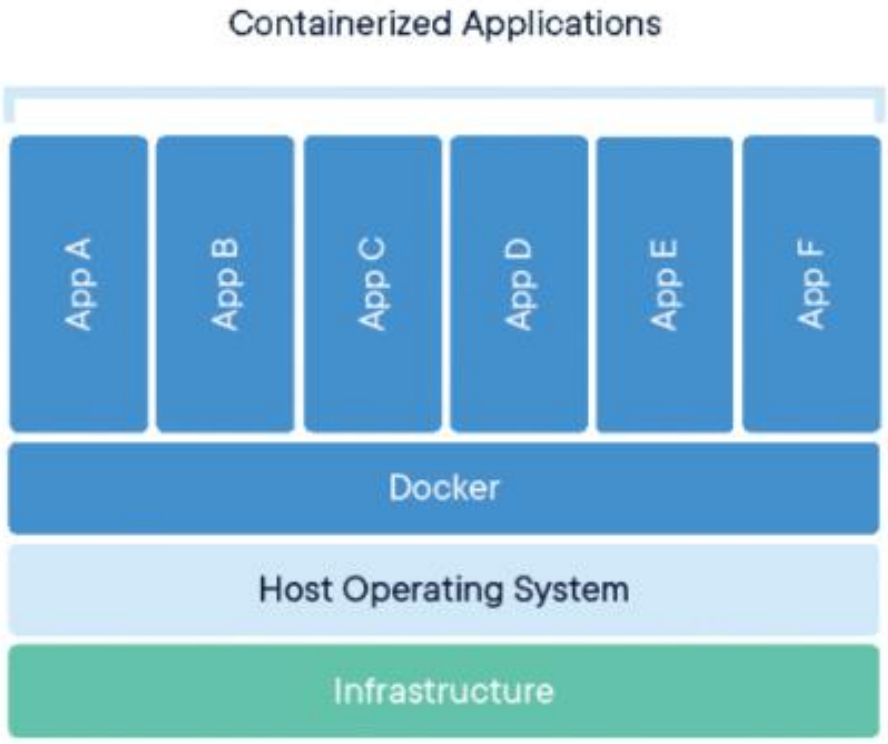


machine virtuelle

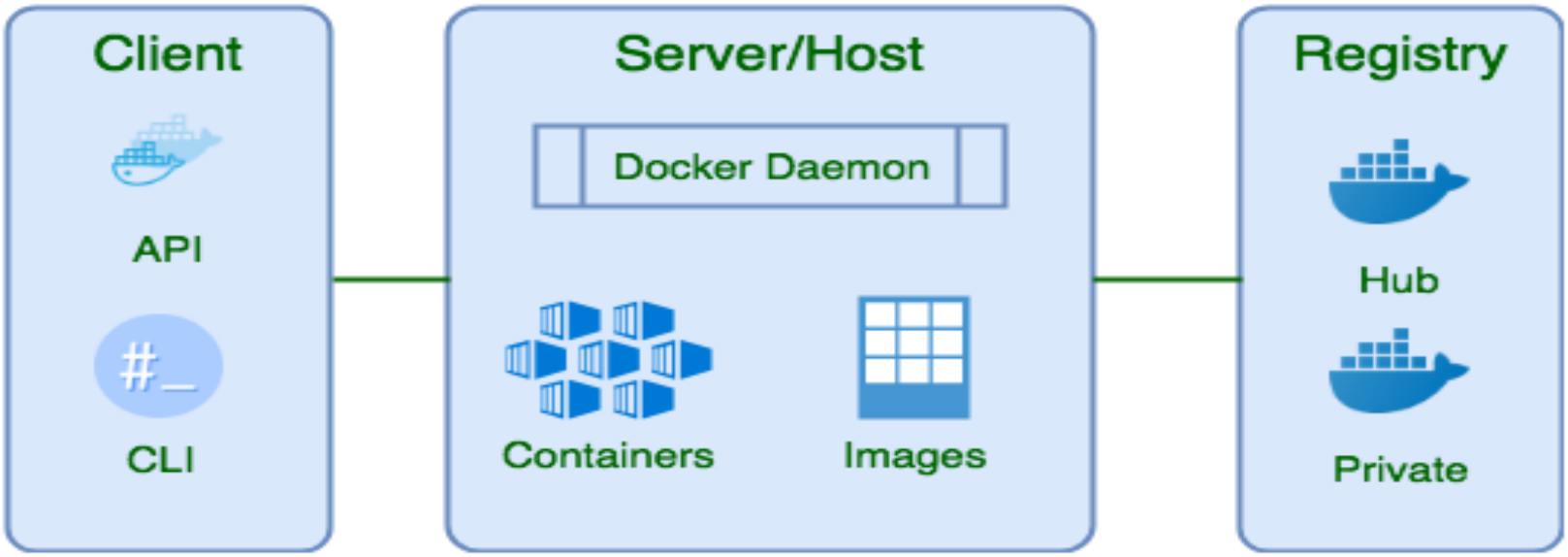
	Conteneurs	Machines virtuelles (VM)
L'architecture	Partager le noyau du système d'exploitation hôte ; n'empaqueter que l'application et les dépendances.	Exécution au-dessus des hyperviseurs ; inclusion du système d'exploitation, des applications et des dépendances.
Utilisation des ressources	Utilisation réduite des ressources (CPU et RAM) grâce à un système d'exploitation partagé	Utilisation accrue des ressources en raison d'un système d'exploitation distinct pour chaque machine virtuelle
Temps de démarrage	Démarrage rapide (quelques secondes), idéal pour les pipelines CI/CD	Temps de démarrage plus long dû à la configuration du système d'exploitation, d'où une surcharge plus importante.
Isolement et sécurité	Isolement partiel ; risque de sécurité si le noyau partagé est compromis	Sécurité accrue grâce à une isolation complète (chaque VM possède son propre système d'exploitation)
Portabilité	Grande portabilité dans tous les environnements ; moins de problèmes de compatibilité	Encombrant et moins portable ; des problèmes de compatibilité peuvent se poser entre les différents environnements.

Introduction à Docker

Docker est un projet open-source qui automatise le déploiement d'applications à l'intérieur de conteneurs de logiciels en fournissant une couche supplémentaire d'abstraction et d'automatisation de la virtualisation au niveau du système d'exploitation



Architecture Docker

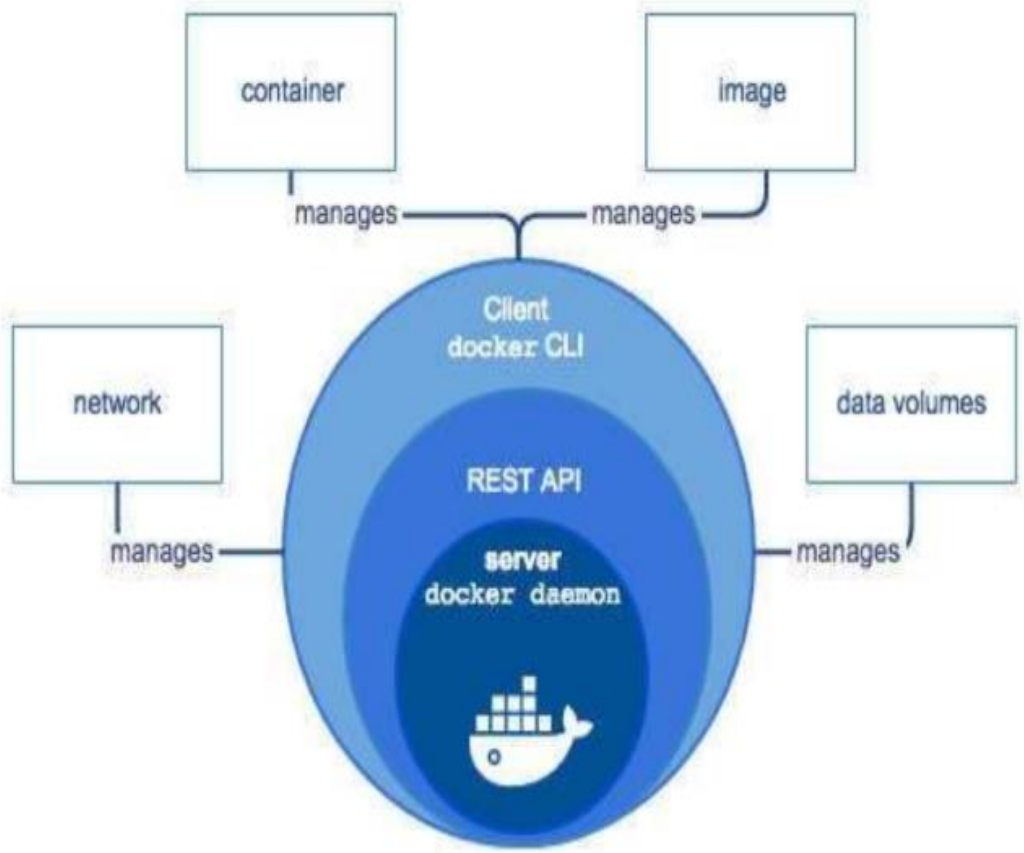


•**Docker Host:** Docker host est le cœur de Docker et le moteur de conteneurisation. Il gère les opérations essentielles telles que la création, l'exécution et la suppression de conteneurs. Il écoute les demandes de l'API Docker et gère les objets Docker tels que les images, les conteneurs, les réseaux et les volumes.

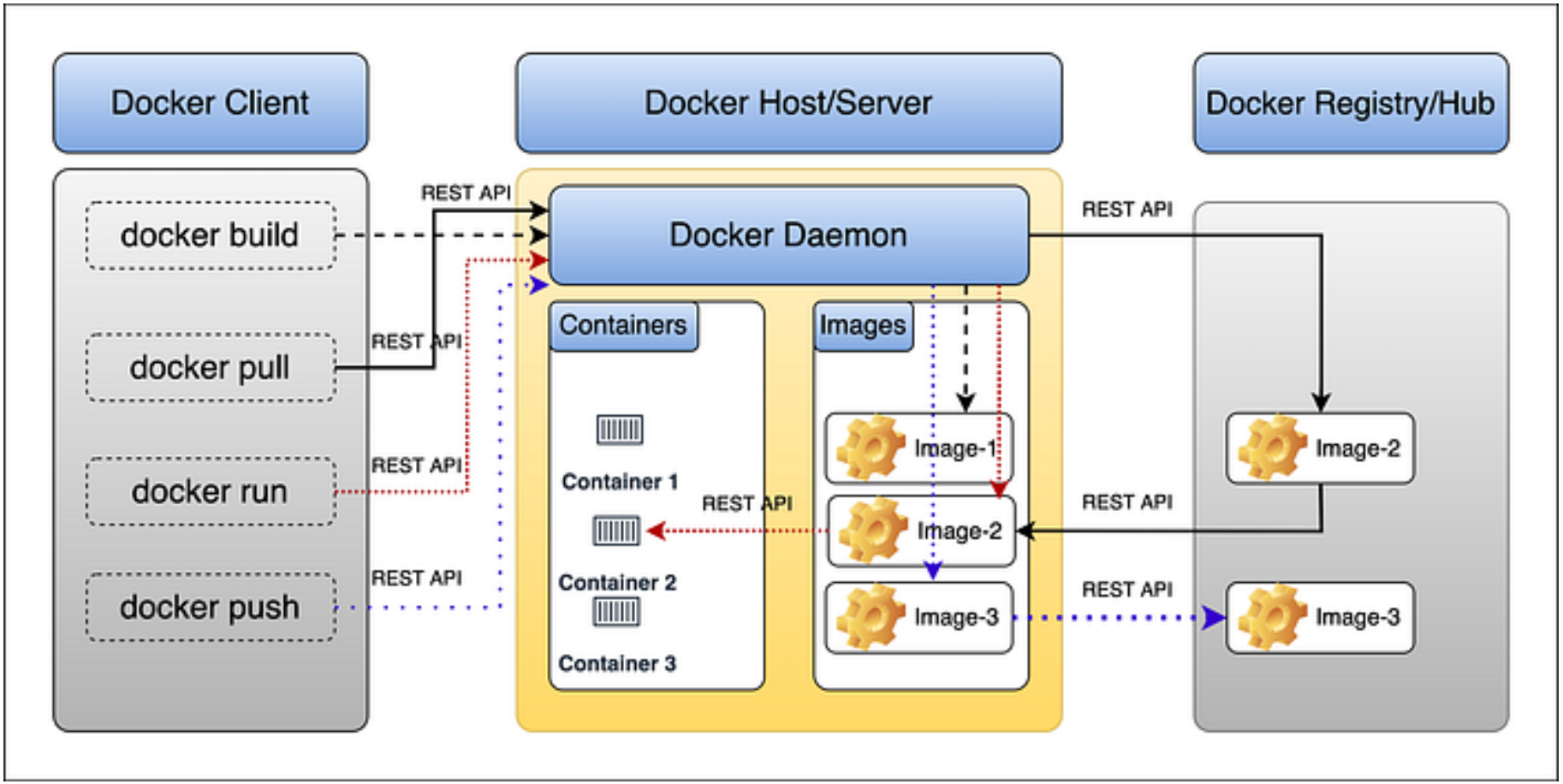
•**Docker Client:** Le Docker Client est une interface en ligne de commande (CLI) qui permet aux utilisateurs d'interagir avec le daemon Docker. Il permet de donner des instructions au Docker Engine pour créer, exécuter, arrêter et gérer des conteneurs, ainsi que pour interagir avec d'autres parties de l'écosystème Docker.

•**Docker Registry:** un référentiel centralisé qui stocke les images Docker. Il facilite le partage et la distribution d'images entre les développeurs et les équipes, favorisant ainsi la collaboration et le déploiement d'applications.

•**Les images Docker** constituent des modèles immuables (ce qui signifie que son contenu ne peut pas être modifié) qui incluent le système d'exploitation, les bibliothèques et les dépendances nécessaires à l'exécution d'une application.



•**Les conteneurs Docker** sont des instances en cours d'exécution de ces images, isolées les unes des autres et du système hôte.



Types de stockage Docker (Docker Storage)

Docker propose plusieurs mécanismes pour gérer et persister les données entre conteneurs et hôtes :

1.Volumes

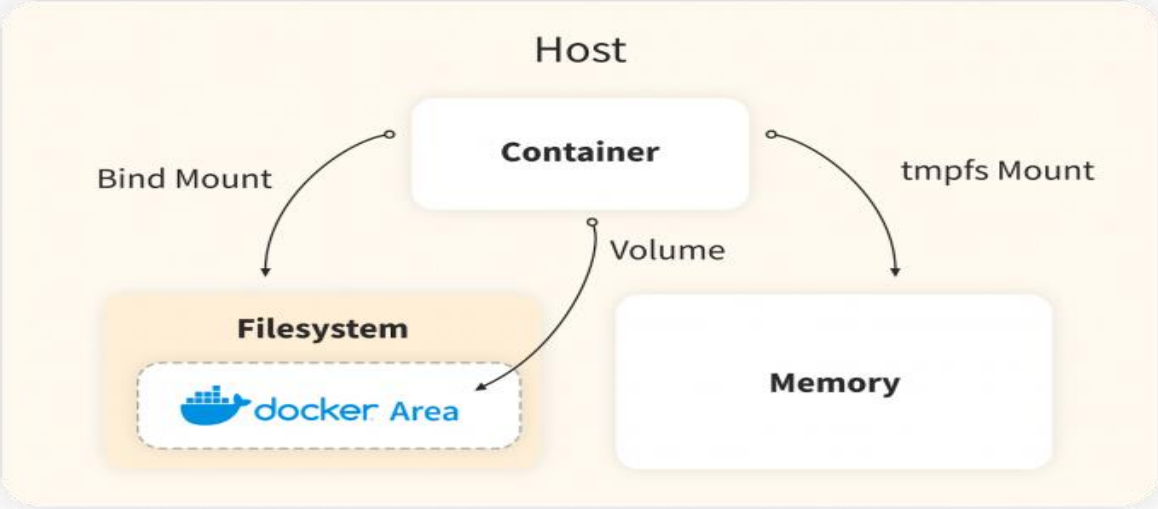
- Gérés directement par Docker et stockés dans un répertoire dédié du système hôte (ex. /var/lib/docker/volumes/).

2.Bind Mounts

- Permet de monter un fichier ou répertoire de la machine hôte directement dans un conteneur.
- Très utile en développement (ex. partager du code source entre hôte et conteneur).

3.tmpfs Mounts

- Stockage en mémoire (RAM), jamais écrit sur le disque de l'hôte.



Types de réseaux Docker

1.Bridge (Pont)

- Réseau par défaut utilisé lorsque plusieurs conteneurs sur le même hôte Docker doivent communiquer entre eux.

2.Host

- Supprime l'isolation réseau entre conteneur et hôte. Le conteneur utilise directement l'interface réseau de l'hôte.

3.Overlay

- Permet la communication entre conteneurs répartis sur plusieurs hôtes.
- Utilisé notamment dans Docker Swarm.

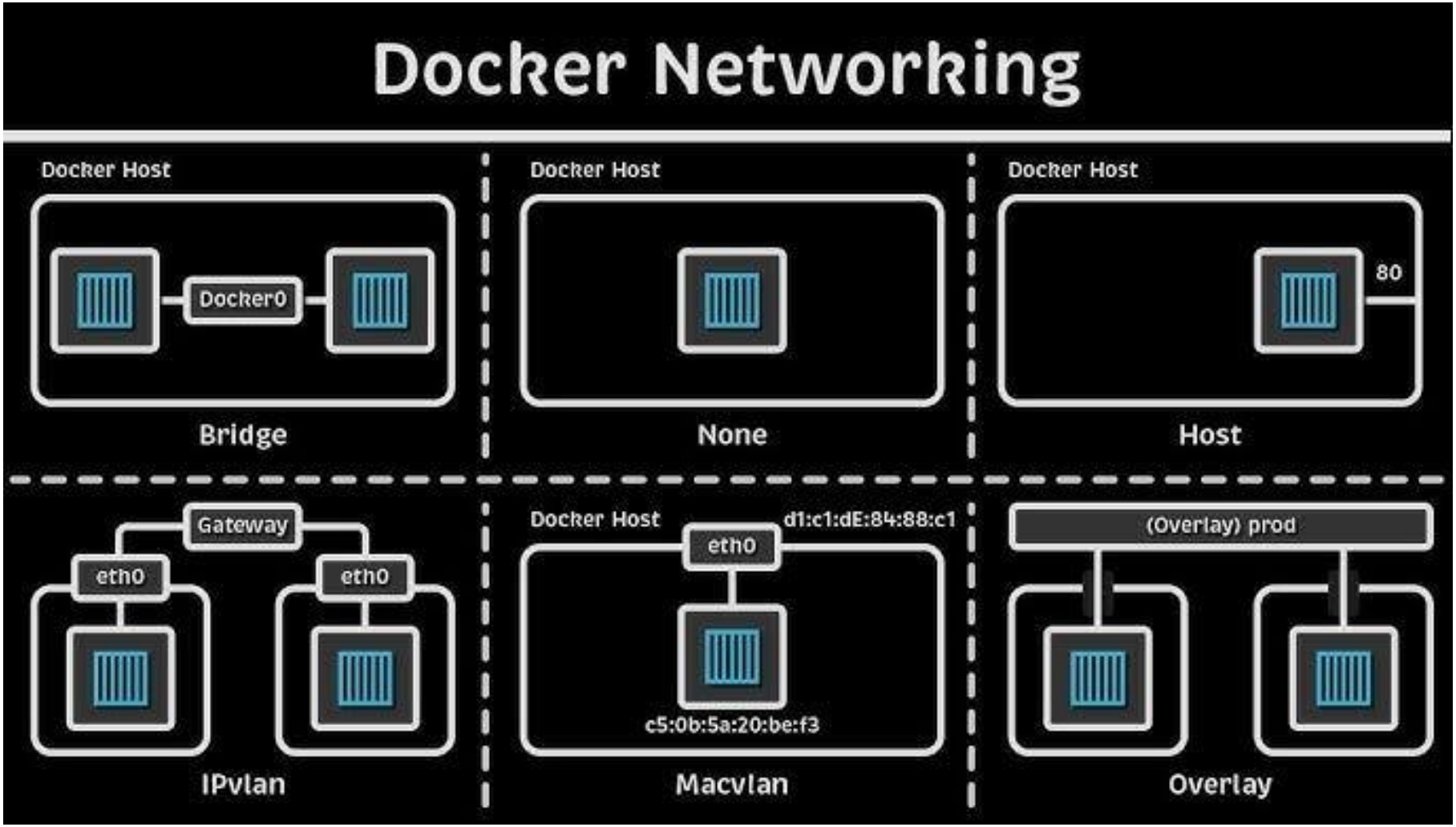
4.None

- Désactive tout réseau pour le conteneur.

5.Macvlan

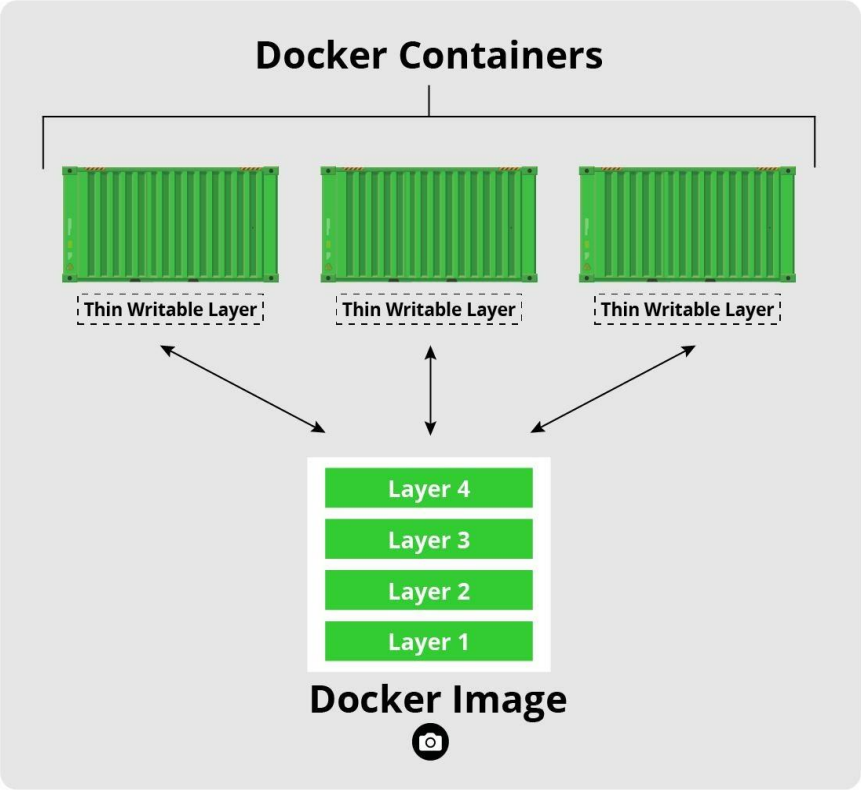
- Attribue une adresse MAC unique au conteneur.
- Le conteneur est vu comme un appareil physique distinct sur le réseau.

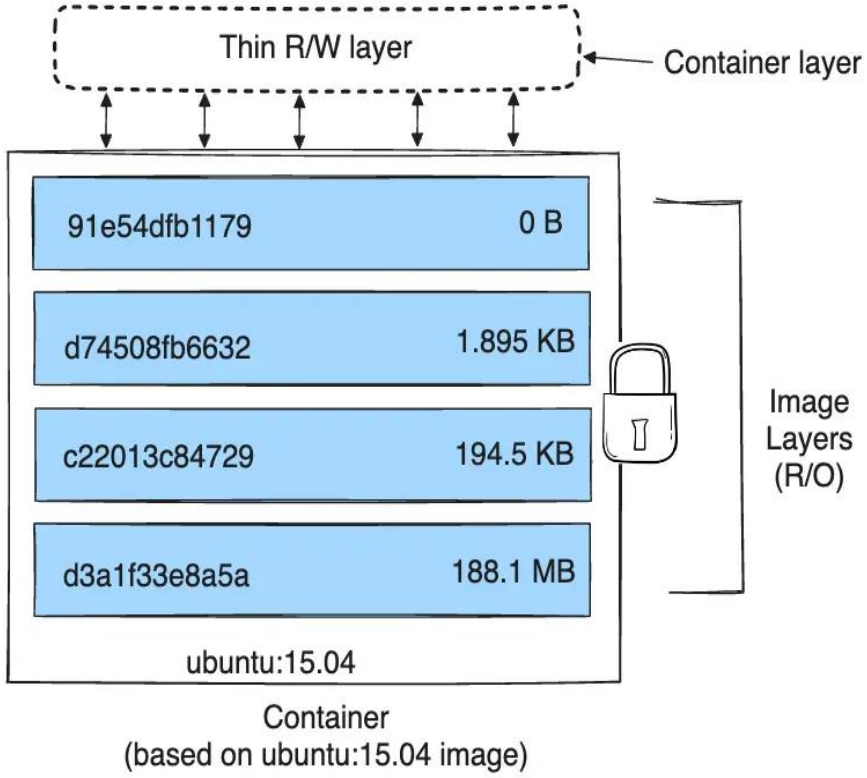
Types de réseaux Docker



Les images Docker

Quand on commence à vouloir conteneuriser une application, on entend vite parler des images de conteneurs. Mais qu'est-ce que c'est exactement ? **Une image**, c'est un peu comme un modèle figé : elle contient tout ce qu'il faut pour faire tourner une application – code, dépendances, système de fichiers. C'est à partir de cette image que les conteneurs seront lancés.



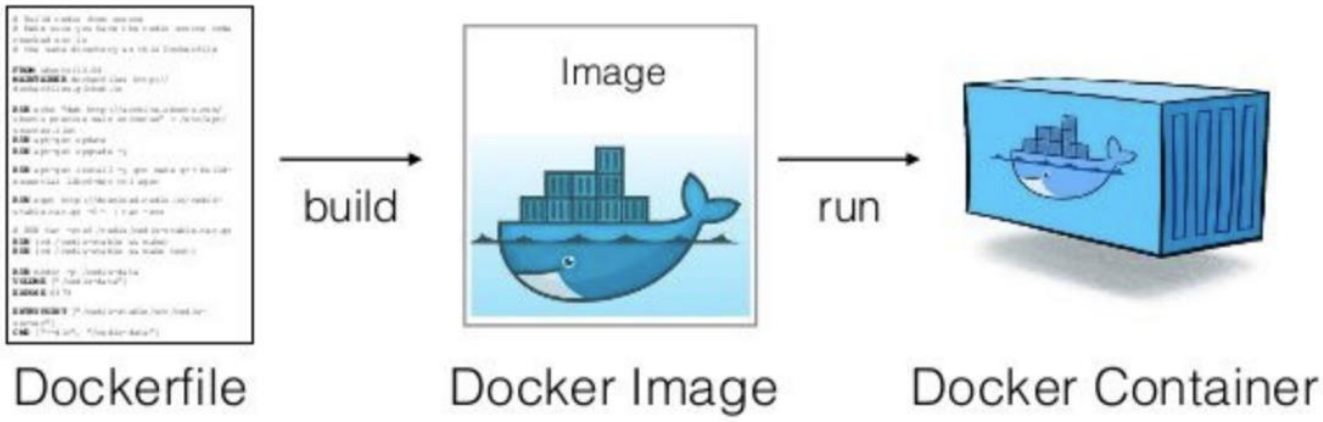


La création d'images

On peut construire une image Docker de deux manières :

- **À partir d'un conteneur existant**, en le modifiant puis en utilisant **docker commit** pour créer une nouvelle image, méthode rapide mais non reproductible.

- **À partir d'un Dockerfile**, en écrivant les instructions (FROM, COPY, RUN, CMD) et en exécutant **docker build**, ce qui crée des couches réutilisables .



➢ À partir d'un conteneur existant

Pour créer une image à partir d'une image de base dans laquelle nous allons installer cowsay :

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

- **docker commit**
- **docker tag**
- **docker diff**

❑ Configuration du conteneur

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

❑ **Installer le programme dans le conteneur**

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

❑ **Quitter la session interactive**

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

❑ **Inspecter les changements**

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

C: fichier ou répertoire modifié
A: fichier ou répertoire ajouté

❑ Sauvegarder les changements dans une nouvelle image

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

❑ Exécution de la nouvelle image

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

❑ Tagger une image

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

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

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

Commandes de base Docker

Commandes générales

- **docker --version** → Afficher la version de Docker installée.
- **docker info** → Informations sur le moteur Docker (daemon, images, conteneurs, etc.).
- **docker help** → Aide générale ou sur une commande (docker run --help).

Gestion des conteneurs

- **docker ps** → Lister les conteneurs en cours d'exécution.
- **docker ps -a** → Lister tous les conteneurs (y compris arrêtés).
- **docker run <image>** → Créer et lancer un conteneur à partir d'une image.
- **docker run -d <image>** → Lancer un conteneur en arrière-plan (detached mode).
- **docker run -it <image> bash** → Lancer un conteneur en mode interactif (accès au terminal).
- **docker stop <id>** → Arrêter un conteneur en cours.
- **docker start <id>** → Redémarrer un conteneur arrêté.
- **docker restart <id>** → Redémarrer un conteneur.
- **docker rm <id>** → Supprimer un conteneur.

Gestion des images

- **docker imag**es → Lister les images disponibles localement.
- **docker pull <image>** → Télécharger une image depuis un registre (ex. Docker Hub).
- **docker build -t <nom_image>** . → Construire une image à partir d'un Dockerfile.
- **docker rmi <image>** → Supprimer une image.

Inspection et logs

- **docker logs <id>** → Afficher les logs d'un conteneur.
- **docker inspect <id>** → Obtenir les détails d'un conteneur ou d'une image.
- **docker exec -it <id> bash** → Entrer dans un conteneur en cours d'exécution.

Rechercher une image

```
root@mHAJMABROUK-PC:~# docker search nginx | head
NAME                                      DESCRIPTION                                     STARS     OFFICIAL
nginx                                     Official build of Nginx.                        20989     [OK]
nginx/nginx-ingress                       NGINX and  NGINX Plus Ingress Controllers fo…   110
nginx/nginx-prometheus-exporter           NGINX Prometheus Exporter for NGINX and NGIN…   50
nginx/unit                                This repository is retired, use the Docker o…   66
nginx/nginx-ingress-operator              NGINX Ingress Operator for NGINX and NGINX P…   2
nginx/docker-extension                                                                    0
nginx/nginx-quic-qns                      NGINX QUIC interop                              1
nginx/nginxaas-loadbalancer-kubernetes                                                    1
nginx/unit-preview                        Unit preview features                           0
root@mHAJMABROUK-PC:~#
```

Puller une image

```
root@mHAJMABROUK-PC:~# docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
d107e437f729: Downloading [=>                                                 ]  581.6kB/28.23MB
cb497a329a81: Downloading [>                                                  ]  454.7kB/44.07MB
f1c4d397f477: Download complete
f72106e86507: Waiting
899c83fc198b: Waiting
a785b80f5a67: Waiting
6c50e4e0c439: Waiting
```

Lister les images

```
root@mHAJMABROUK-PC:~# docker images
REPOSITORY                                                                                  TAG
AGE ID       CREATED         SIZE
europe-west3-docker.pkg.dev/ow-agentow-1wg-11368-hp/fan-test-mon-projet-pyhton/mon-projet-python           test
26552670e0   7 weeks ago     371MB
europe-west3-docker.pkg.dev/ow-agentow-1wg-11368-hp/fan-test-mon-projet-pyhton/mon-projet-python           latest
854fd713cf   7 weeks ago     371MB
registry.gitlab.tech.orange/win/explo/ow-data/ow-genai/agent-hub/poc-agentique-2a4/fan_test_agent_bonjour   main
06603e72a2   7 weeks ago     467MB
europe-west3-docker.pkg.dev/ow-gow-m2c-9931-spoke-hp/spoke-hp-repo/proxy                    0.6.0
f96b438244   5 months ago    458MB
```

Run une image

```
root@mHAJMABROUK-PC:~# docker run nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
d107e437f729: Downloading [=======>                                           ]  4.092MB/28.23MB
d107e437f729: Downloading [==========>                                        ]  5.861MB/28.23MB
cb497a329a81: Downloading [======>                                            ]  5.313MB/44.07MB
f72106e86507: Waiting
f72106e86507: Download complete
```

```
root@mHAJMABROUK-PC:~# docker run -d -p 8080:80 nginx:latest
```

Lister les centenaires en cours d'exécution

```
root@mHAJMABROUK-PC:~# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
eb26ead3c16e   nginx     "/docker-entrypoint.…"   44 seconds ago   Up 44 seconds   80/tcp    cool_visvesvaraya
root@mHAJMABROUK-PC:~#
```

Connecter à un contenaire

```
root@mHAJMABROUK-PC:~# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
e425362dae05   nginx     "/docker-entrypoint.…"   53 seconds ago   Up 52 seconds   80/tcp    vigorous_allen
root@mHAJMABROUK-PC:~# docker exec -it e425362dae05 bash
root@e425362dae05:/#
```

Stopper un centenaire en cours d'exécution

```
root@mHAJMABROUK-PC:~# docker stop eb26ead3c16e
eb26ead3c16e
root@mHAJMABROUK-PC:~# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
root@mHAJMABROUK-PC:~#
```

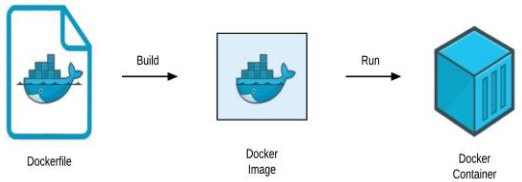
Supprimer une image

```
root@mHAJMABROUK-PC:~# docker rmi nginx:lates
```

Dockerfile

Un **Dockerfile** est un **fichier texte** qui contient une suite d'instructions permettant d'automatiser la création d'une image Docker avec la commande docker build
Il décrit comment **construire une image** (quelle base utiliser, quels fichiers copier, quelles dépendances installer, quelle commande exécuter au démarrage, etc.).

➢ Elément de syntaxe dockerfile



FROM: Définit l'image à partir de laquelle la nouvelle image est créée

LABEL : Associe des meta-données à la nouvelle image (par exemple, l'auteur de l'image)

RUN: Définie une commande exécutée dans la couche au dessus de l'image courante lors de la construction de l'image

CMD/ ENTRYPOINT: Définie la commande exécutée au démarrage du conteneur

EXPOSE: Informe docker que le conteneur va écouter sur le port réseau défini

COPY / ADD : Copier un fichier/répertoire depuis le contexte de construction de l'image vers la nouvelle couche

```
FROM ubuntu:20.04
CMD ["echo", "Bonjour avec CMD"]
```

- `docker run mon-image`
  ➡️ affiche : **Bonjour avec CMD**
- `docker run mon-image ls`
  ➡️ affiche : **contenu du dossier** (le `ls` remplace totalement le `CMD`)

VS

```
FROM ubuntu:20.04
ENTRYPOINT ["echo", "Bonjour avec ENTRYPOINT:"]
```

- `docker run mon-image`
  ➡️ affiche : **Bonjour avec ENTRYPOINT:**
- `docker run mon-image Mohamed`
  ➡️ affiche : **Bonjour avec ENTRYPOINT: Mohamed**
  (le mot *Mohamed* est ajouté à la suite de la commande)

Image Version Pinning missing (DL3006,DL3007)

Base Image
```
FROM ubuntu :12.04
```

Maintainer or maintainer email missing (DL3012,D4000)

Env. Variable
```
MAINTAINER John Doe <joe@doe.org>
ENV USE_HTTP 0
```

Comment
```
# Add proxy settings
COPY  ./setenv.sh /tmp/
```

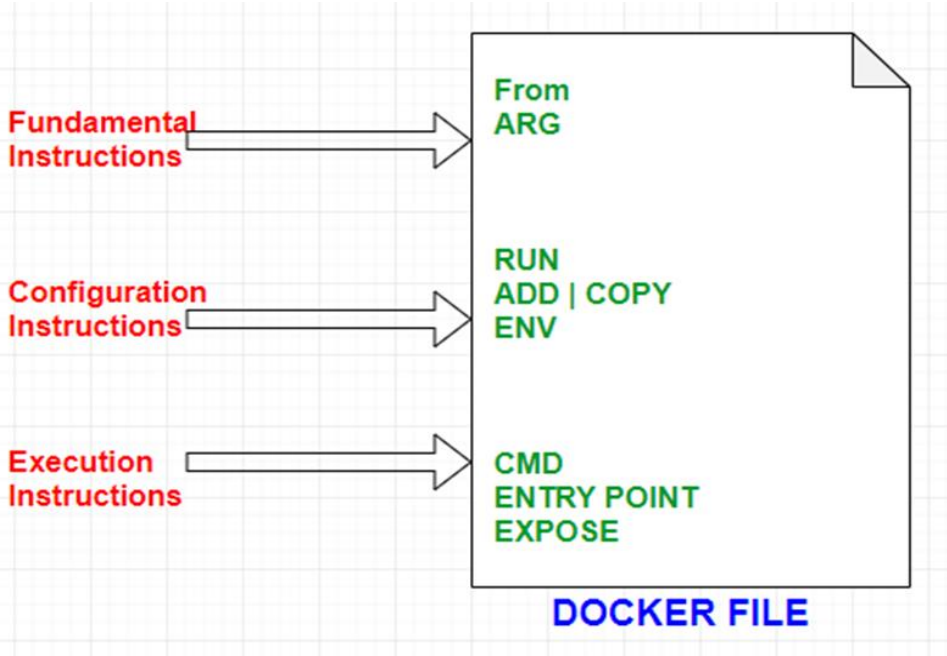
`RUN` can execute any shell command
```
RUN sudo apt-get update
RUN sudo apt-get upgrade -y
```

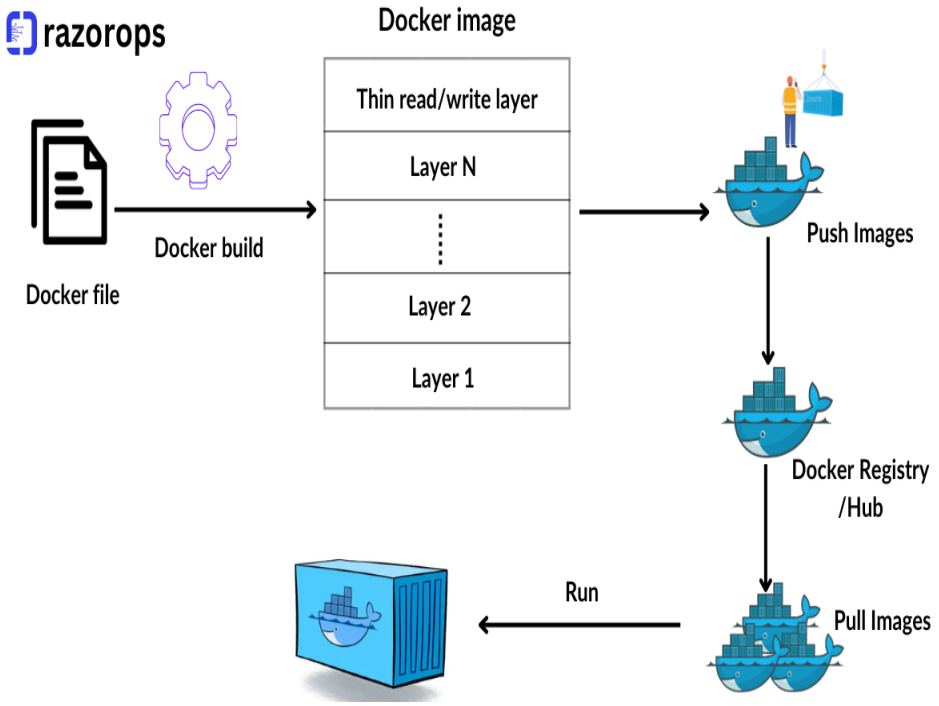
Installing dependencies
```
RUN apt-get install -y wget :1.12
```
Dependency Version Pinning missing (DL3008,DL3013)
```
RUN sudo -E pip install scipy :0.18.1
```

Installing software (compiling, linking, etc.)
```
RUN cd /usr/src/vertica-sqlalchemy;
    sudo python ./setup.py install
```

Open Port
```
EXPOSE 8888
# CMD ipython notebook --ip=* …
ADD runcmd.sh /
```
Using ADD instead of COPY (DL3020)
```
RUN chmod u+x /runcmd.sh
```

Start Process
```
CMD ["/runcmd.sh"]
```





Notre premier Dockerfile

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

```
root@mHAJMABROUK-PC:~# mkdir my_docker
root@mHAJMABROUK-PC:~# cd my_docker/
root@mHAJMABROUK-PC:~/my_docker#
```

```
root@mHAJMABROUK-PC:~/my_docker# vim dockerfile
```

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

Builder l'image via dockerfile

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

-t permet de tagger l'image qui va être créée

. indique le contexte de construction de l'image (où se trouve le Dockerfile)

Que se passe-t-il?

Le build context est envoyé vers le démon docker (contenu du répertoir .)
A chaque étape:

- Un conteneur est créé pour exécuter l'étape (Running in ...)
- Les modifications sont committées dans une nouvelle image (---> ...)
- Le conteneur est supprimé
- La nouvelle image est utilisée pour la prochaine étape

Voir l'historique de l'image

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

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

Exemple d'utilisation de COPY

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

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

Publier mon image sur Docker Hub

Tagger l'image à publier:

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

- mydockeraccount: Mon login sur Docker Hub

Se connecter à docker hub:

```
docker login
```

Publier l'image:

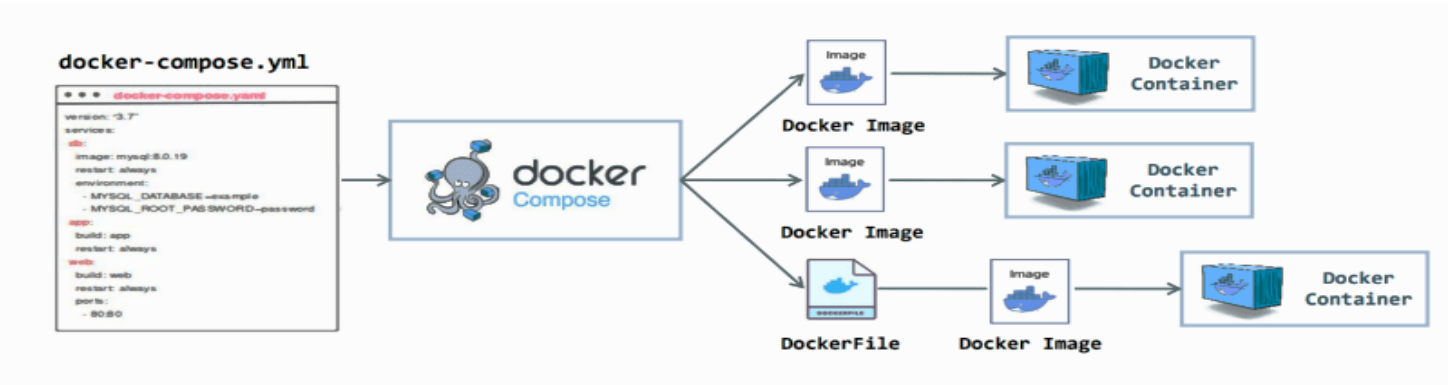
```
docker push mydockeraccount/cowsay
```

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

```
docker pull mydockeraccount/cosway
```

Docker-compose

Docker Compose est un outil qui permet de **définir et gérer plusieurs conteneurs Docker** comme une seule application.
Au lieu de lancer manuellement chaque conteneur avec plusieurs commandes docker run, on décrit la configuration complète dans un fichier YAML (docker-compose.yml).



Commandes principales Docker Compose

- **docker compose up** → Démarrer l'application (tous les services).
- **docker compose up -d** → Démarrer en arrière-plan (detached mode).
- **docker compose down** → Arrêter et supprimer les conteneurs, réseaux, volumes créés.
- **docker compose ps** → Lister les services en cours d'exécution.
- **docker compose logs** → Voir les logs des services.

```yaml
version: "3.9"

services:
  web:
    build: .
    container_name: nginx-web
    ports:
      - "8080:80"
    networks:
      - app-network

  db:
    image: mysql:8
    container_name: mysql-db
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: testdb
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

```yaml
version: "3"
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    networks:
      - common-network
  redis:
    image: redis
    volumes:
      - redis-data:/var/lib/redis
    networks:
      - common-network
volumes:
  redis-data:
    driver: local
networks:
  common-network:
    driver: bridge
```

local image

Exposed port mapping

Used volume

Connected network

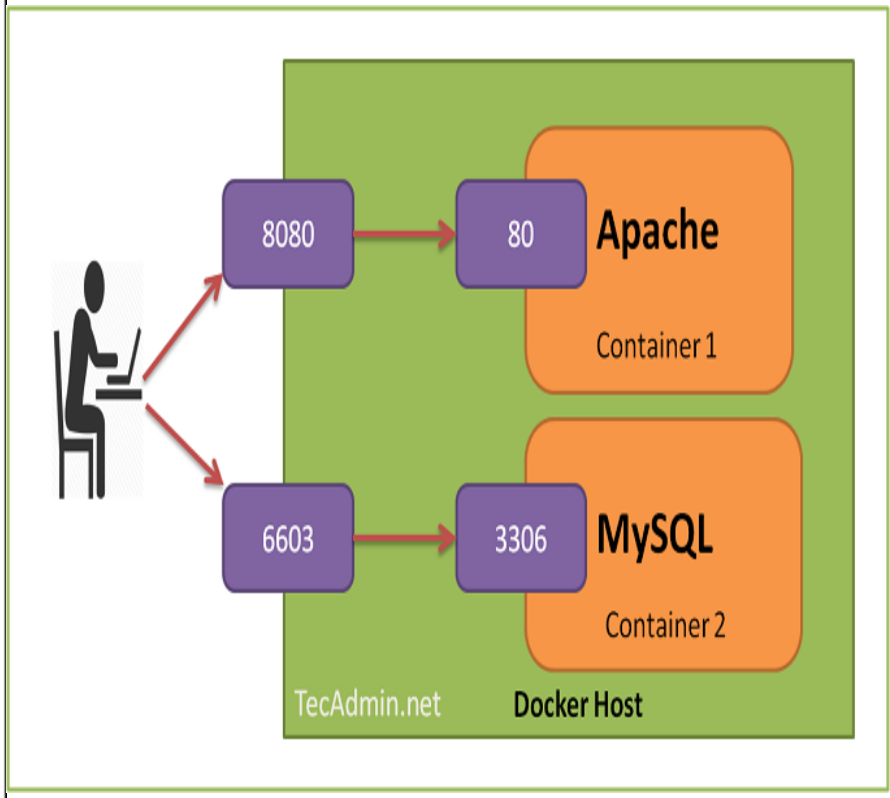
Image from online registry

Component 1

Component 2

Named volume

Network

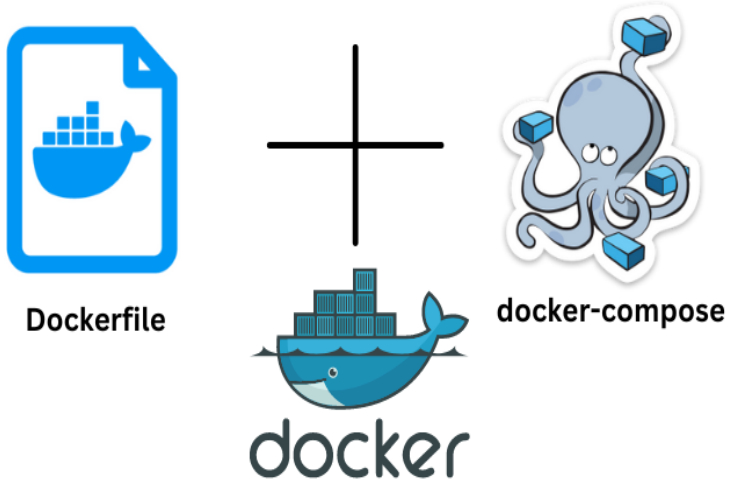


Résumé Dockerfile/Docker-Compose

Dockerfile = "la recette de cuisine" (construction d'une image).

Docker Compose = "le menu complet" (comment lancer plusieurs conteneurs ensemble).

Ils sont complémentaires : souvent, tu écris un Dockerfile pour ton app, puis tu l'utilises dans un docker-compose.yml pour orchestrer app + DB + autres services.



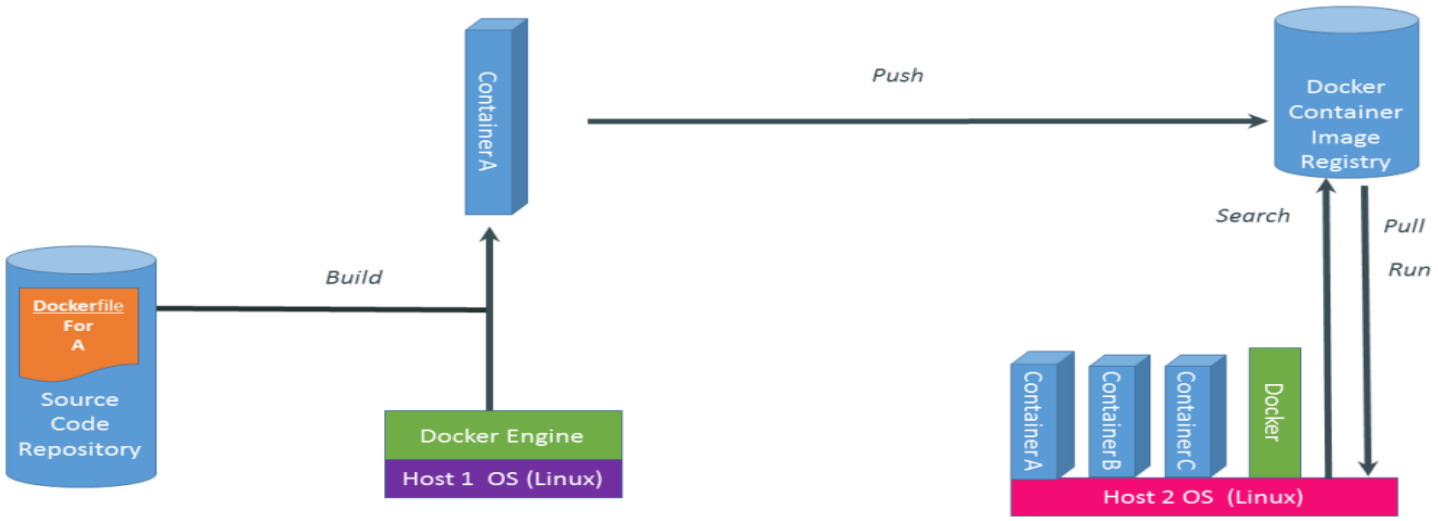

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    container_name: flask_hello
```

Avec dockerfile
. = le dossier courant (le contexte de build)
docker compose va chercher un fichier Dockerfile dans ce dossier
le contexte de build peut être modifié selon le besoin

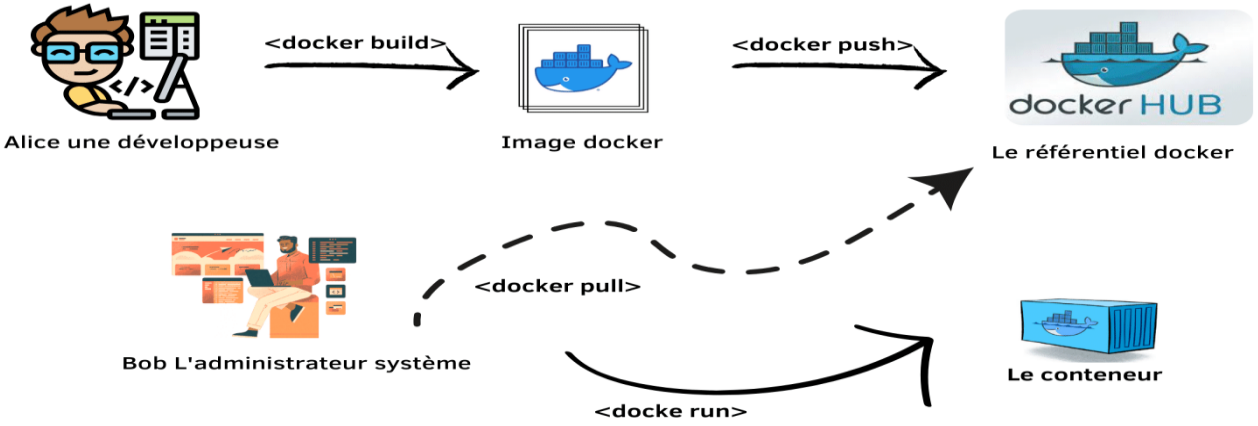
Sans dockerfile

```yaml
version: "3.9"
services:
  nginx:
    image: nginx:alpine
    ports:
      - "8080:80"
    container_name: nginx_server
```

Maintenant que nous avons exploré les fondements de l'architecture Docker vous pouvez vous demander : « **Mais comment cela fonctionne-t-il réellement** ?«

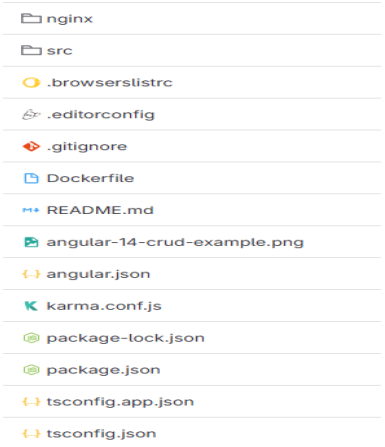


Le **Docker daemon** est en cours d'exécution en arrière-plan pour gérer ce processus de construction.




LAB-Docker

angular-14

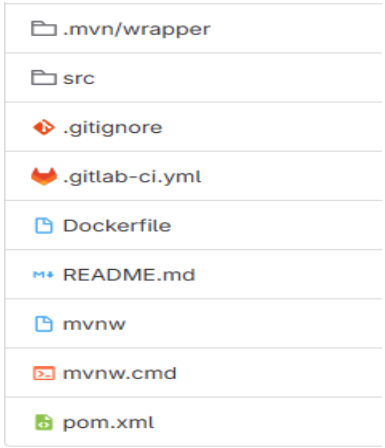


Rédigez un fichier Dockerfile multi stage :

Partie 1 – Étape de build (construction de l'application)
- Utiliser l'image de base : node:18
- Définir le répertoire de travail : /app
- Copier les fichiers package.json et package-lock.json
- Installer les dépendances avec npm install
- Copier l'ensemble du projet dans le conteneur
- Compiler l'application en mode production avec
  npm run build --prod

Partie 2 – Étape de déploiement (serveur web)
- Utiliser l'image de base : nginx:alpine
- Copier le fichier de configuration personnalisé nginx.conf dans /etc/nginx/nginx.conf
- Copier depuis l'étape précédente le dossier compilé /app/dist/angular-14-crud-example/ vers /usr/share/nginx/html

spring-boot



Rédigez un fichier Dockerfile Multi stage :

Partie 1 – Étape de build (compilation de l'application)
- Utiliser l'image de base : maven:3.9.6-eclipse-temurin-17
- Définir le répertoire de travail : /spring-boot-server
- Copier le fichier pom.xml dans le conteneur
- Télécharger les dépendances du projet avec la commande :
  mvn dependency:go-offline –B
- Copier l'ensemble du code source dans le conteneur
- Compiler et empaqueter l'application sans exécuter les tests avec la commande :
  mvn clean package –DskipTests

Partie 2 – Étape de runtime (exécution de l'application)
- Utiliser l'image de base : eclipse-temurin:17-jdk-jammy
- Définir le répertoire de travail : /app
- Copier depuis l'étape précédente le fichier .jar généré dans /app/app.jar
- Exposer le port 9090 (port d'écoute de l'application)
- Définir une variable d'environnement JAVA_OPTS pour gérer la mémoire :
  -Xmx512m -Xms256m
- Lancer l'application via une commande d'entrée (entrypoint) adaptée :
  ["sh", "-c", "java $JAVA_OPTS -jar app.jar"]

Dockerfile

```
FROM node:18 AS build-stage

WORKDIR /app

COPY package.json package-lock.json ./

RUN npm install

COPY . .

RUN npm run build --prod


FROM nginx:alpine

COPY ./nginx/nginx.conf
/etc/nginx/nginx.conf

COPY --from=build-stage
/app/dist/angular-14-crud-example/
/usr/share/nginx/html
```

Dockerfile

```
FROM maven:3.9.6-eclipse-temurin-17 AS build

WORKDIR /spring-boot-server

COPY pom.xml .

RUN mvn dependency:go-offline -B

COPY . .

RUN mvn clean package -DskipTests

FROM eclipse-temurin:17-jdk-jammy AS runtime

WORKDIR /app

COPY --from=build /spring-boot-
server/target/*.jar app.jar

EXPOSE 9090

ENV JAVA_OPTS="-Xmx512m -Xms256m"

ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar
app.jar"]
```

1- Recherche et test de l'image officielle

•Recherchez et récupérez la dernière image officielle **nginx** depuis Docker Hub.

•Lancez un conteneur basé sur cette image et testez-le en ouvrant votre navigateur sur **http://localhost:8080** pour vérifier que la page par défaut de nginx s'affiche correctement.

2- Création d'une image Docker personnalisée

•Créez un **Dockerfile** qui :

- Utilise l'image nginx officielle comme base,
- Copie le fichier `index.html` fourni dans le conteneur pour remplacer la page par défaut de nginx,
- Expose le port 80.

•Construisez et lancez le conteneur de manière à ce que votre navigateur affiche **Hello World** sur `http://localhost:8080`.

3- Docker Compose avec volumes et réseau

•Créez un fichier `docker-compose.yml` qui :

- Déploie votre conteneur nginx personnalisé,
- Monte le fichier `index.html` via un **volume**,
- Mappe le port 8080 de l'hôte vers le port 80 du conteneur,
- Utilise un **réseau bridge** nommé `lab-docker`.

Livrables

•Dockerfile

•docker-compose.yml

•Capture d'écran de la page affichant **Hello World** sur `http://localhost:8080`