



La conteneurisation

A2S2

La conteneurisation

vs. virtualisation et émulation

Introduction à la conteneurisation



Virtualisation et conteneurisation jouent un rôle clé dans la gestion efficace et sécurisée d'applications et d'environnements de travail.

Virtualiser = exécuter plusieurs systèmes sur une même machine physique

Conteneuriser = isoler des process en partageant un même noyau

Émuler = simuler un matériel différent. Intéressant pour du reverse ou l'utilisation d'apps qui ne sont pas cross-arch. Peut être combiné à la virtu ou conteneurisation

?

Exemples de techno ?

?

Exemples de techno ?

Conteneurisation

Partage du noyau, isolation légère, plus rapide et efficace. Mais dépendant du noyau hôte.

Meilleure portabilité et scalabilité rapide

Virtualisation

Chaque VM embarque un OS complet. Indépendant mais plus lourd.

Type 1 = bare metal (ESXi, Hyper-V, ...)

Type 2 = hosted (VBox, VMWare Workstation, ...)

Introduction à la conteneurisation

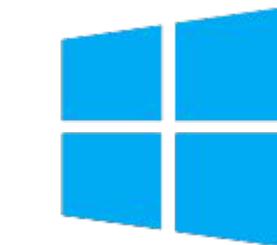
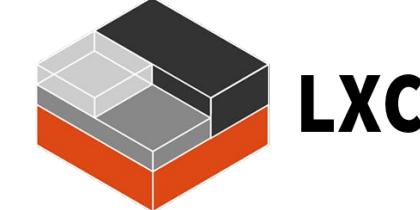
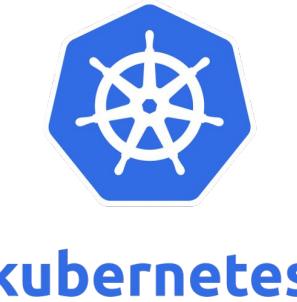


Virtualisation et conteneurisation jouent un rôle clé dans la gestion efficace et sécurisée d'applications et d'environnements de travail.

Virtualiser = exécuter plusieurs systèmes sur une même machine physique

Conteneuriser = isoler des process en partageant un même noyau

Émuler = simuler un matériel différent. Intéressant pour du reverse ou l'utilisation d'apps qui ne sont pas cross-arch. Peut être combiné à la virtu ou conteneurisation



Microsoft
Hyper-V



Conteneurisation

Partage du noyau, isolation légère, plus rapide et efficace. Mais dépendant du noyau hôte.

Meilleure portabilité et scalabilité rapide

Virtualisation

Chaque VM embarque un OS complet. Indépendant mais plus lourd.

Type 1 = bare metal (ESXi, Hyper-V, ...)

Type 2 = hosted (VBox, VMWare Workstation, ...)

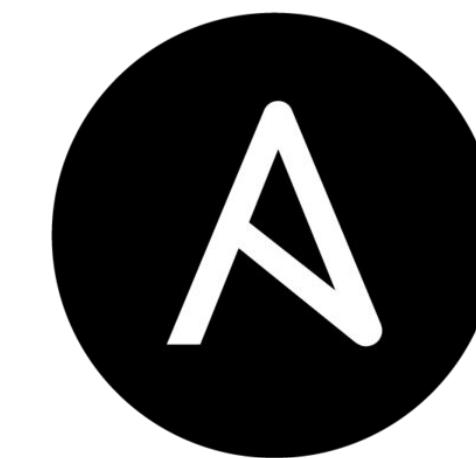
Comprendre le reste de l'écosystème

26
05



HashiCorp
Vagrant

Déployer, automatiser et manager des VMs
(repose sur des hyperviseurs)

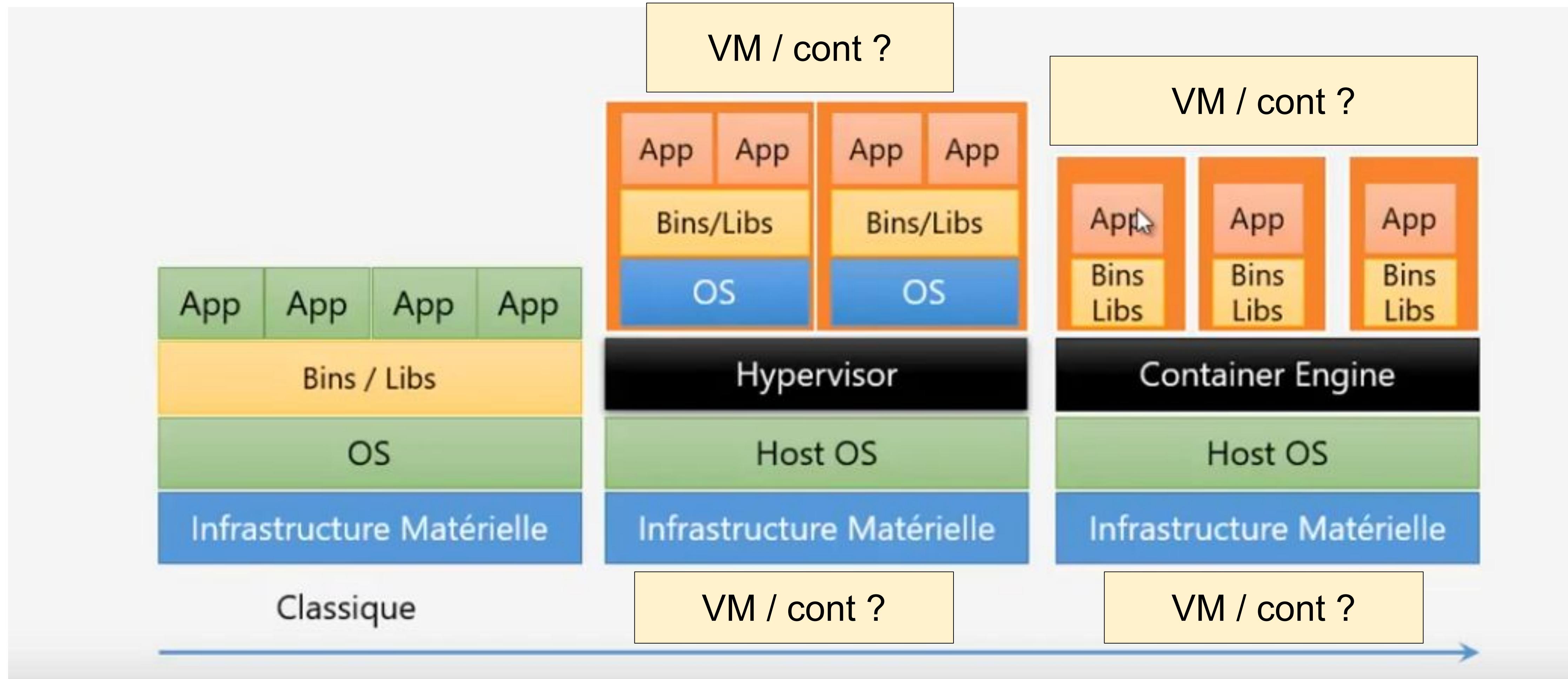


A N S I B L E

Configuration des machines, principe de playbooks
YAML

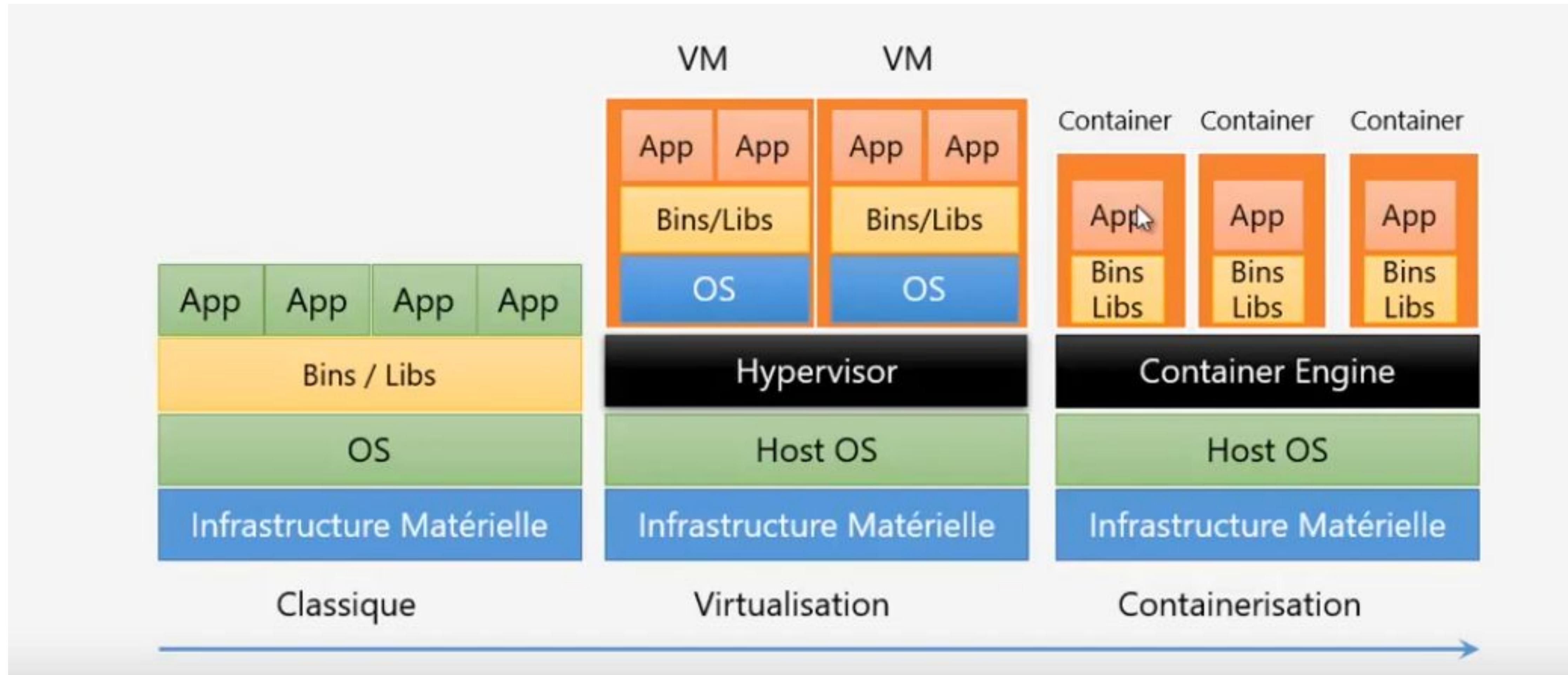
Introduction à la conteneurisation

zo
gr



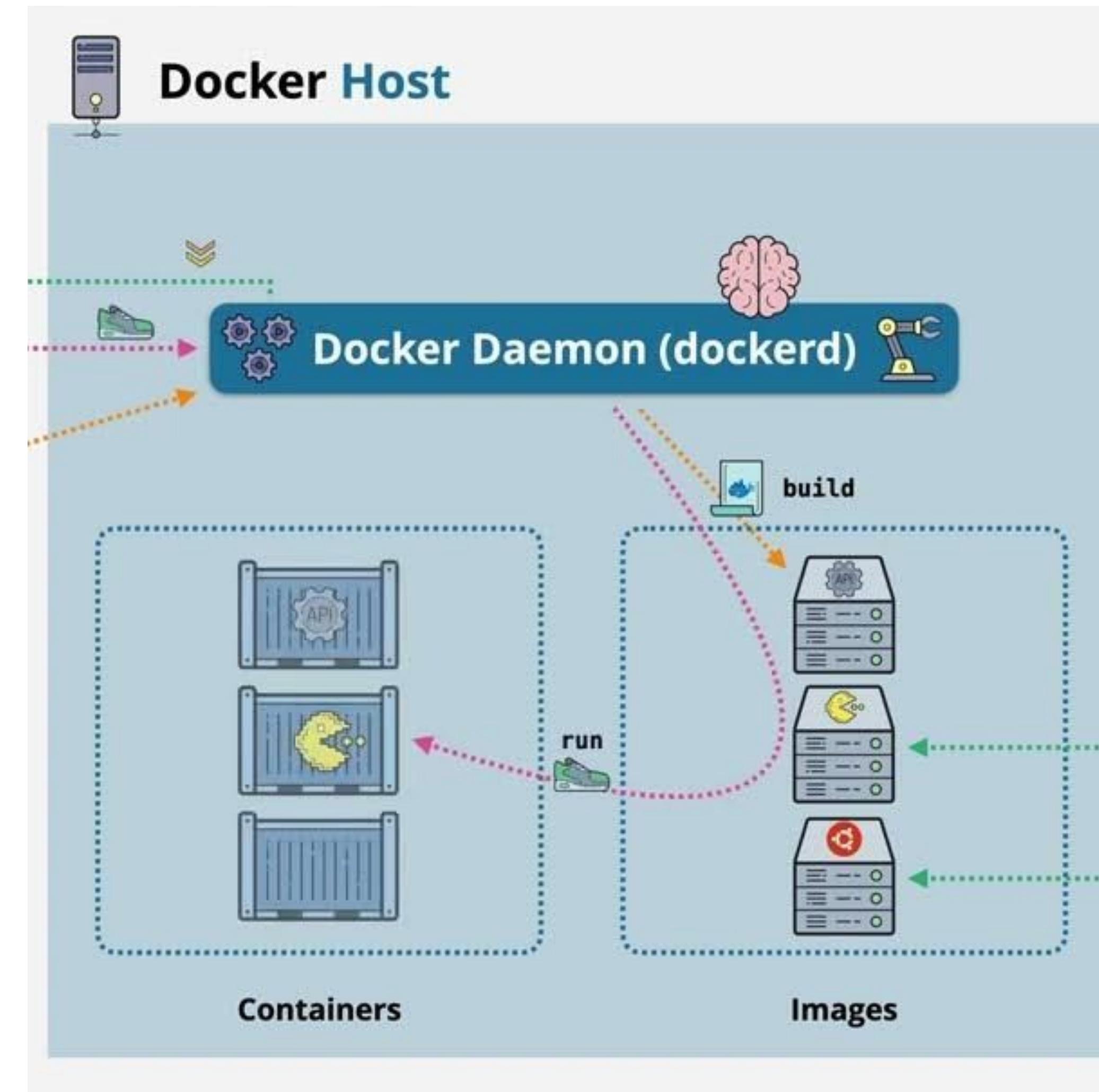
Introduction à la conteneurisation

zo
gr



Architecture de Docker

26
GR



Docker, la fraude du siècle

20
08

1979 - **chroot** - pour changer le root directory d'un process et isoler l'accès au fs

2002 - **namespaces** - diverses isolations (PID, réseau, utilisateurs) empêchant un processus d'interagir avec d'autres en dehors de son namespace

2007 - **groups** - contrôler l'allocation de ressources à un groupe de process

Technos utilisées dès 2008 dans LXC mais complexe à déployer et configurer

2012 - Docker - ajoute un format d'image standardisé, une API, un CLI

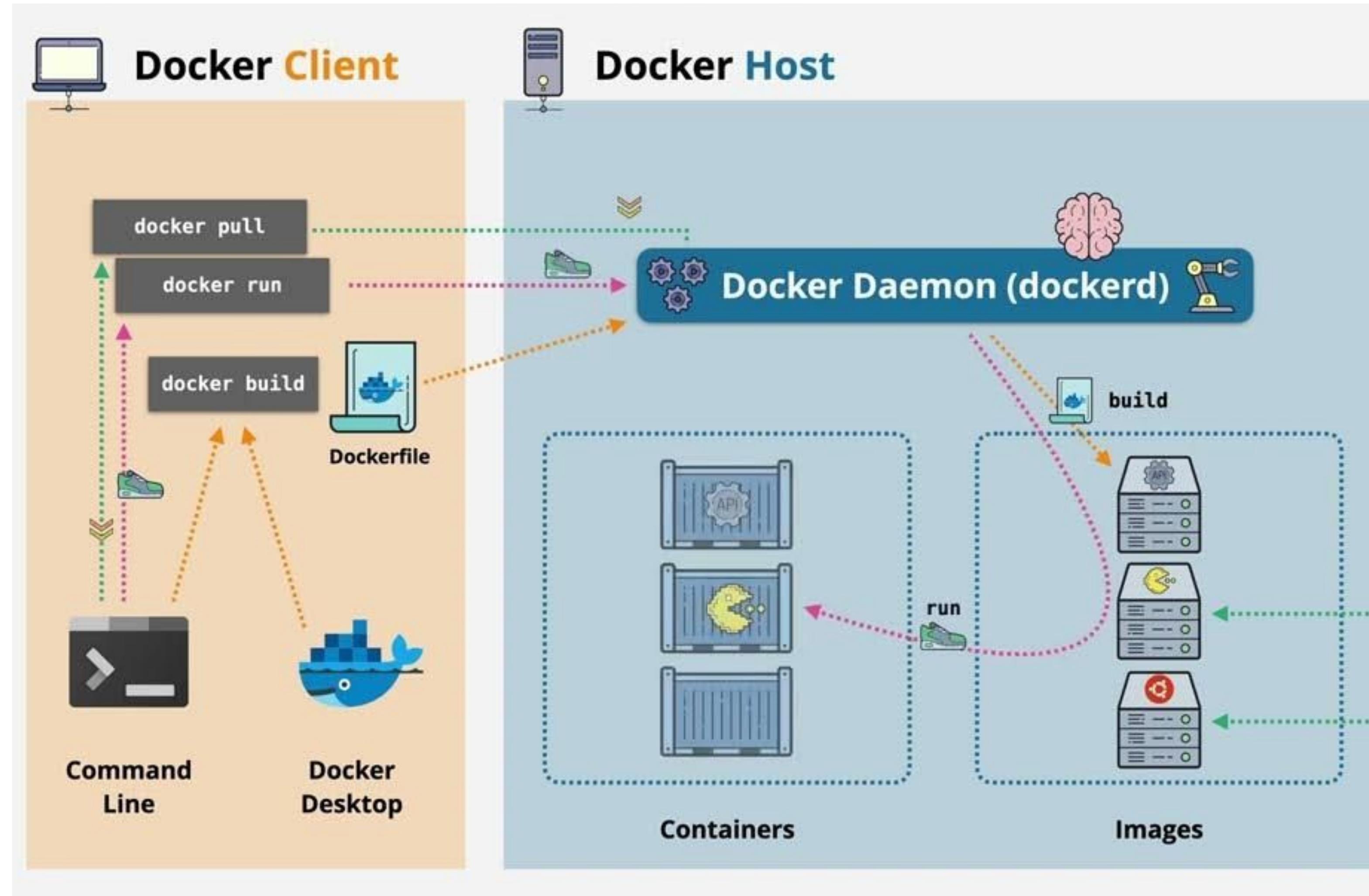
"user-friendly", et un registre public pour distribuer des images. Mais dans le fond ça s'appuie sur ces technos existantes depuis 30 ans.

Docker a donc fortement popularisé ces principes, qui étaient alors des concepts techniques trop poussés pour le moindre quidam.



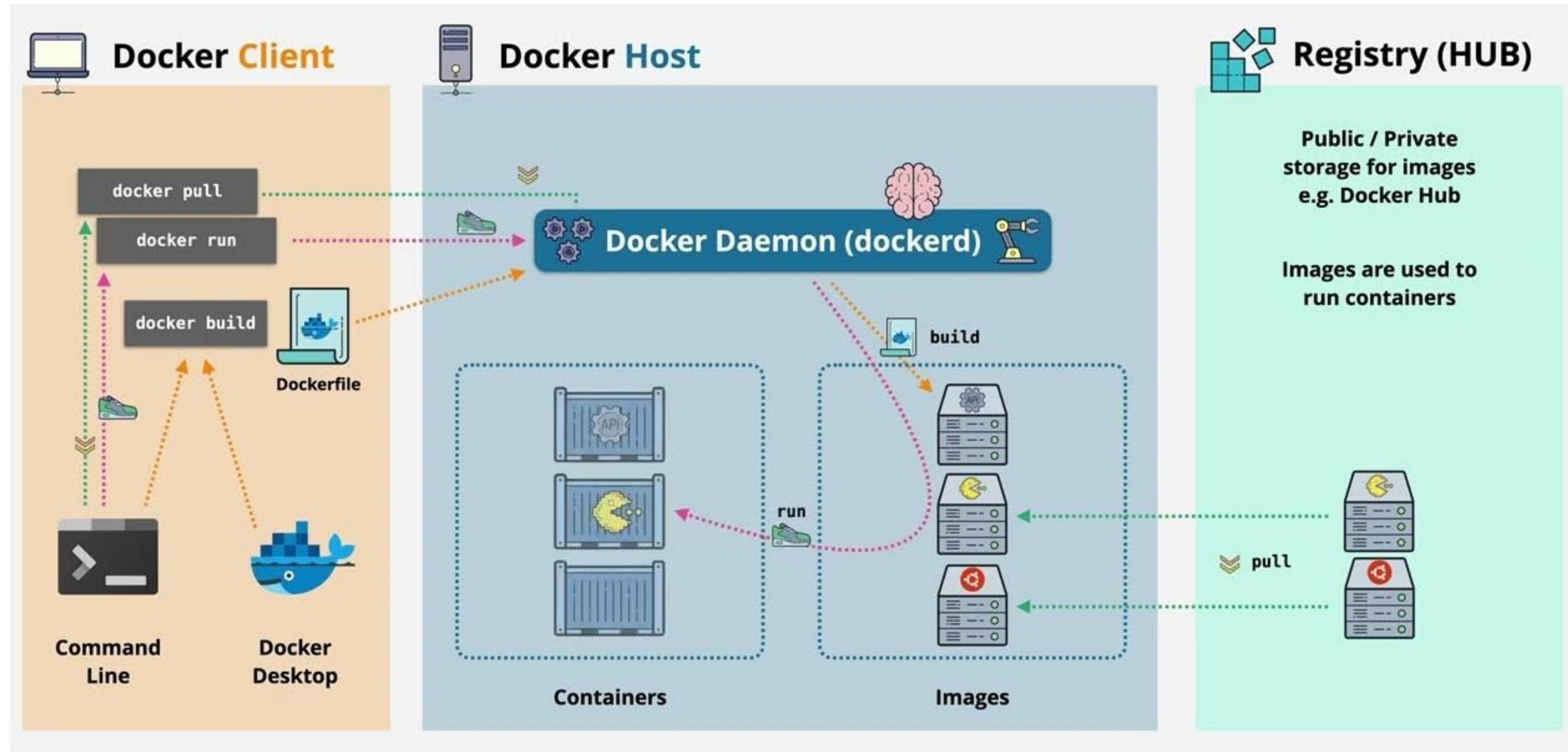
Architecture de Docker

zo
gr



Architecture de Docker

zo
gr



Installation de Docker



Windows

<https://docs.docker.com/desktop/setup/install/windows-install/>

The screenshot shows the Docker Docs website with a blue header. The navigation bar includes links for 'dockerdocs', 'Get started', 'Guides', 'Manuals' (which is underlined), 'Reference', a search bar, and AI tools. The left sidebar has a 'Manuals' section with dropdown menus for 'OPEN SOURCE' (Docker Engine, Docker Build, Docker Compose, Testcontainers) and 'PRODUCTS' (Docker Desktop, Setup, Install, Mac, Understand permission require..., Windows, Understand permission require..., Enterprise deployment, Linux). The main content area displays the 'Install Docker Desktop on Windows' page, which includes a breadcrumb trail (Home / Manuals / Docker Desktop / Setup / Install / Windows), a title, a 'Docker Desktop terms' section, a note about commercial use requiring a paid subscription, instructions for installation, and download links for 'Docker Desktop for Windows - x86_64' and 'Docker Desktop for Windows - Arm (Beta)'. A tip section is at the bottom. On the right side, there's a 'Table of contents' and links for system requirements, installing Docker Desktop on Windows, interactive installation, command-line installation, starting Docker Desktop, and where to go next. There are also 'Edit this page' and 'Request changes' buttons.

https://docs.docker.com/desktop/setup/install/windows-install/

dockerdocs

Get started

Guides

Manuals

Reference

Search

Ask AI

Edit this page

Request changes

Table of contents

System requirements

Install Docker Desktop on Windows

Install interactively

Install from the command line

Start Docker Desktop

Where to go next

Manuals

OPEN SOURCE

Docker Engine

Docker Build

Docker Compose

Testcontainers

PRODUCTS

Docker Desktop

Setup

Install

Mac

Understand permission require...

Windows

Understand permission require...

Enterprise deployment

Linux

Home / Manuals / Docker Desktop / Setup / Install / Windows

Install Docker Desktop on Windows

Docker Desktop terms

Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than \$10 million USD in annual revenue) requires a [paid subscription](#).

This page contains the download URL, information about system requirements, and instructions on how to install Docker Desktop for Windows.

[Docker Desktop for Windows - x86_64](#)

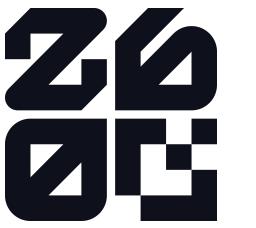
[Docker Desktop for Windows - Arm \(Beta\)](#)

For checksums, see [Release notes](#)

System requirements

Tip

Installation de Docker Desktop



UNIX-like (éviter Docker Desktop, pas besoin de ça)

<https://docs.docker.com/engine/install/>

The screenshot shows the Docker Engine installation documentation page. The top navigation bar includes links for dockerdocs, Get started, Guides, Manuals (which is the active tab), and Reference. A search bar and an AI icon are also present. The left sidebar has a 'Manuals' dropdown menu with sections for OPEN SOURCE (Docker Engine, Install, Ubuntu, Debian, RHEL, Fedora, Raspberry Pi OS (32-bit)), Storage, Networking, and Containers. The main content area displays the 'Install Docker Engine' page, which describes how to install Docker Engine on Linux (Docker CE) and mentions Docker Desktop for Windows, macOS, and Linux. It links to an 'Overview of Docker Desktop'. Below this, a 'Supported platforms' section provides a table mapping operating systems to supported architectures:

Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS (32-bit)			✓		
RHEL	✓	✓		✓	
SLES (s390x)					✓

On the right side of the page, there are links for 'Edit this page', 'Request changes', 'Table of contents', 'Supported platforms', 'Other Linux distributions', 'Release channels', 'Support', 'Upgrade path', 'Licensing', 'Reporting security issues', and a 'Get started' button.

Installation de Docker Desktop Orbstack



macOS

<https://orbstack.dev/>

The screenshot shows the OrbStack website's landing page for macOS. At the top, there's a navigation bar with the OrbStack logo, Pricing, Blog, Docs, a blue 'Download' button, and a user icon. The main headline is 'Feel the difference'. Below it, a sub-headline reads: 'Skip the wait and leave your charger behind with OrbStack's unmatched performance and efficiency.' A comparison table follows, showing provisioning times for different environments:

Speed	Build	CPU & Battery	CPU & Battery
Open edX	PostHog	Kubernetes	Supabase

Below the table, two progress bars are shown: one for OrbStack (17 min) and one for Docker Desktop (45 min). The text 'Time to provision development environment (lower is better)' is displayed above the bars.

as of August 2023



Docker sur Windows & macOS



Docker fonctionne différemment sur Windows et macOS par rapport à Linux, car il repose sur des solutions de virtualisation pour exécuter des conteneurs Linux.

Sur Windows, Docker tourne dans WSL, qui lui-même tourne dans une VM Linux dédiée sous une version allégée d'Hyper-V (Virtual Machine Platform). C'est transparent pour l'utilisateur, mais il y a donc une couche de virtualisation avant Docker qui limite l'accès hardware (notamment USB).

Sur macOS, c'est la même chose avec le framework d'Apple Hypervisor

Les limites :

- sur macOS et Windows, l'accès aux fichiers depuis le conteneur vers l'OS hôte est plus lent qu'en natif sous Linux.
- certaines restrictions CPU/mémoire peuvent ne pas fonctionner exactement comme sous Linux.
- les utilisations avancées telles que le partage de device physique/USB
- limites réseau, notamment mode host, sur Docker Desktop Windows et Linux (c'est "supported" en partie mais pas un vrai bridge)



Docker 101

Comprendre et manipuler images et conteneurs

Images Docker



Une image est un modèle de conteneur. On n'exécute rien dans une image. Une image se déploie.

Elle contient un système de fichiers, des dépendances et des instructions pour exécuter une application.

Types d'images (type abstrait)

Une image Docker peut être

- basée sur un **OS** : Alpine, Debian, ...
- Basée sur un **runtime** spécifique (mais qui inclut tout de même un OS derrière) : Python, Node.js, Go, ...
- Basée sur une **application** spécifique (idem) : Nginx, PostgreSQL, Redis

Elle peut être téléchargée depuis Dockerhub (ou autre registre), ou construite en local (Dockerfile)

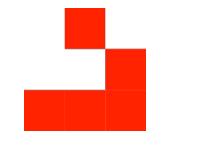
Les couches

Une image = des layers (couches) superposées.

Chaque couche est une modification apportée à la couche n-1.

Chaque commande du Dockerfile crée une nouvelle couche.

Elles sont immutables (remplaçables mais pas modifiables)



Instructions Dockerfile



Un Dockerfile est un script qui décrit comment construire une image Docker.

Il existe plusieurs instructions pour définir l'environnement, installer des logiciels, configurer l'exécution du conteneur et optimiser la gestion des ressources.

FROM : spécifier l'image de base

LABEL : ajouter des métadonnées à l'image (cf. `docker inspect`)

RUN : exécuter des commandes lors de la construction

ONBUILD : exécuter des commandes si l'image est utilisée comme base d'une autre (comme si on faisait un **RUN** après le **FROM**)

COPY : copier des fichiers locaux dans l'image (i.e. `COPY SRC DST`)

ADD : similaire à **COPY**, mais peut décompresser automatiquement les archives, et télécharger un fichier distant

WORKDIR : comme son nom l'indique, définir le dossier de travail

CMD : définir la commande exécutée au démarrage du container

ENTRYPOINT : comme **CMD**, mais n'est pas remplaçable, c'est une commande obligatoire

ENV : définir une variable d'environnement

EXPOSE : exposer un port (ce qui est généralement plutôt défini dans la commande `docker run` ou dans le `docker-compose.yml`)

VOLUME : monter un volume

USER : spécifier un utilisateur

SHELL : changer l'interpréteur de commandes

HEALTHCHECK : définition une vérification d'état du container



Exemple de Dockerfile simple

On peut changer la dernière commande, auquel cas les 3 premières couches sont inchangées. Elles sont même mises en cache 🚀

```
# Couche 1 : Image de base
FROM ubuntu:22.04

# Couche 2 : Mise à jour et installation de Python
RUN apt update && apt install -y python3

# Couche 3 : Ajout d'un script Python
COPY app.py /app.py

# Couche 4 : Définition de la commande à exécuter
CMD ["python3", "/app.py"]
```



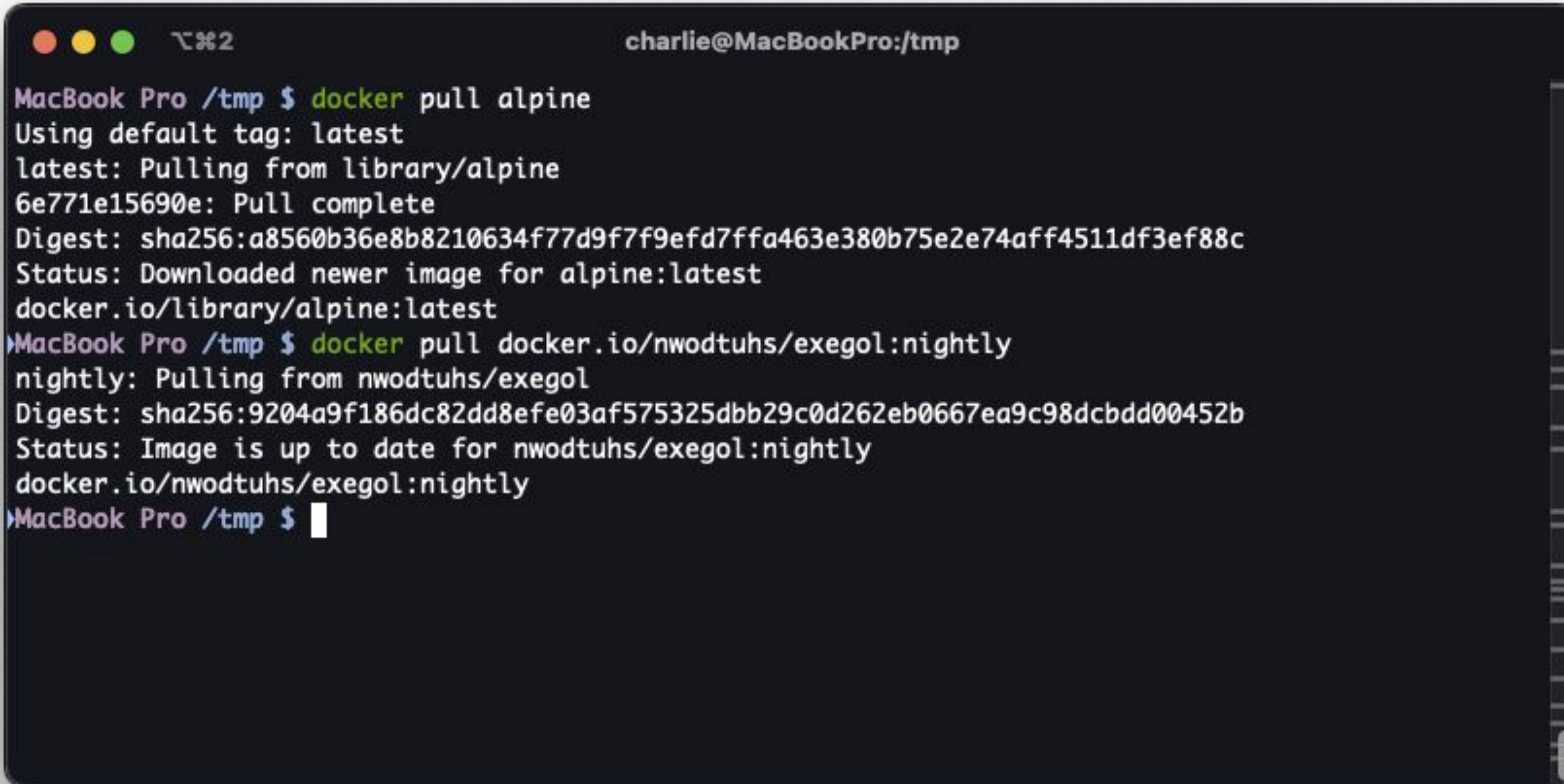
1a Télécharger une image



Depuis un registre distant (généralement Docker Hub).

Syntaxe de l'argument : [REGISTRY/] [LIBRARY/] NAME[:TAG | @DIGEST]

⚠ Ce ne sont pas forcément des images "endorsed" par Docker.



A screenshot of a terminal window on a MacBook Pro. The window title bar shows three colored dots (red, yellow, green) and the text "charlie@MacBookPro:/tmp". The terminal prompt is "MacBook Pro /tmp \$". The user has run two "docker pull" commands:

```
MacBook Pro /tmp $ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
6e771e15690e: Pull complete
Digest: sha256:a8560b36e8b8210634f77d9f7f9efd7ffa463e380b75e2e74aff4511df3ef88c
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
MacBook Pro /tmp $ docker pull docker.io/nwodtuhs/exegol:nightly
nightly: Pulling from nwodtuhs/exegol
Digest: sha256:9204a9f186dc82dd8efe03af575325dbb29c0d262eb0667ea9c98dcbdd00452b
Status: Image is up to date for nwodtuhs/exegol:nightly
docker.io/nwodtuhs/exegol:nightly
MacBook Pro /tmp $
```

1b Construire une image



Depuis un Dockerfile

- ▲ L'image de base (FROM) doit être présente localement, ou disponible en remote.

```
charlie@MacBookPro:tmp
MacBook Pro /tmp $ docker build --tag "2600_01" --file ./Dockerfile .
[+] Building 8.9s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 274B
=> [internal] load metadata for docker.io/library/debian:bullseye
=> [auth] library/debian:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/3] FROM docker.io/library/debian:bullseye@sha256:a6cec654bb08bdc6a563cd7507b62f57c91
=> => resolve docker.io/library/debian:bullseye@sha256:a6cec654bb08bdc6a563cd7507b62f57c91
=> => sha256:a6cec654bb08bdc6a563cd7507b62f57c916290e195142e79a0d41a70ebb2 4.52kB / 4.52kB
=> => sha256:83a9ea0c701aafba43c0dec873b0f1274a7932182823213ffa8716c252f0d 1.04kB / 1.04kB
=> => sha256:88bd349b4fa362fc928b3dc7e23882e5f4b2e66915eb8373a4c97cd11a5f379d 468B / 468B
=> => sha256:7e1cab756c27ddad3e1de86c2aaaf2bca04f012bff531cd99d37f988960 52.25MB / 52.25MB
=> => extracting sha256:7e1cab756c27ddad3e1de86c2aaaf2bca04f012bff531cd99d37f98896026ca4
=> [internal] load build context
=> => transferring context: 53B
=> [2/3] RUN apt update && apt install -y python3
=> [3/3] COPY app.py /app.py
=> exporting to image
=> => exporting layers
=> => writing image sha256:1b82897c7a3d4f531706c6dfaec9f3b410b21b77bf66c4977c5e205a962fa90
=> => naming to docker.io/library/2600_01
MacBook Pro /tmp $
```

2 Créer un container

Créer un container à partir d'une image.

⚠ Impossible de modifier les volumes, capabilities, contraintes CPU/RAM, [...] définies à la création

```
charlie@MacBookPro:/tmp

MacBook Pro /tmp $ docker create --name "2600_cont_01" 2600_01:latest
5e5c261b1200c3b67167e6bb7530137883c082a33630436a74ea7cf014fd98e9
MacBook Pro /tmp $ docker ps --all --filter "name=2600_cont_01"
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5e5c261b1200 2600_01:latest "python3 /app.py" 3 seconds ago Created
2600_cont_01
MacBook Pro /tmp $ docker inspect 2600_cont_01
[
  {
    "Id": "5e5c261b1200c3b67167e6bb7530137883c082a33630436a74ea7cf014fd98e9",
    "Created": "2025-03-11T16:37:56.622268485Z",
    "Path": "python3",
    "Args": [
      "/app.py"
    ],
    "State": {
      "Status": "created",
      "Running": false,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 0,
      "ExitCode": 0,
      "Error": ""
    }
  }
]
```

3 Démarrer un container

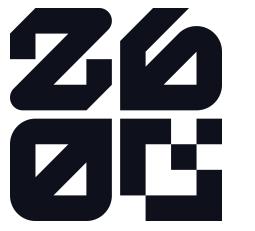
26
01

Démarrer un container existant

```
● ● ●  ▾%2                               charlie@MacBookPro:/tmp

MacBook Pro /tmp $ docker start 2600_cont_01
2600_cont_01
MacBook Pro /tmp $ docker ps --all --filter "name=2600_cont_01"
CONTAINER ID   IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
AMES
5e5c261b1200   2600_01:latest "python3 /app.py"  5 minutes ago   Exited (0) 3 seconds ago
2600_cont_01
MacBook Pro /tmp $ docker start --attach 2600_cont_01
hello 2600
MacBook Pro /tmp $ docker ps --all --filter "name=2600_cont_01"
CONTAINER ID   IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
AMES
5e5c261b1200   2600_01:latest "python3 /app.py"  6 minutes ago   Exited (0) 3 seconds ago
2600_cont_01
MacBook Pro /tmp $
```

2+3 Combiner create et start



Créer et démarrer un container en une seule commande. Ce qui est généralement utilisé.

Pleins d'options possibles (comme create et start)

```
-d, --detach : Exécuter le conteneur en arrière-plan.  
-it, --interactive --tty : Mode interactif avec terminal.  
-p, --publish : Mapper un port hôte vers un port du conteneur.  
-v, --volume : Monter un volume pour la persistance des données.  
-e, --env : Définir une variable d'environnement.  
--name : Attribuer un nom spécifique au conteneur.  
--rm : Supprimer le conteneur après exécution.  
--memory : Limiter la RAM utilisée par le conteneur.  
--cpus : Limiter le nombre de CPU utilisés.  
--restart : Définir une stratégie de redémarrage automatique.  
--network : Connecter le conteneur à un réseau spécifique.  
--privileged : Donner un accès root complet au conteneur.
```



charlie@MacBookPro:tmp

```
MacBook Pro /tmp $ docker run --attach STDOUT 2600_01  
hello 2600  
MacBook Pro /tmp $
```

Zoom sur les options possibles



Une image est un modèle de conteneur. On n'exécute rien dans une image. Une image se déploie.

Elle contient un système de fichiers, des dépendances et des instructions pour exécuter une application.

```
-d, --detach : Exécuter le conteneur en arrière-plan.  
-it, --interactive --tty : Mode interactif avec terminal.  
-p, --publish : Mapper un port hôte vers un port du conteneur.  
-v, --volume : Monter un volume pour la persistance des données.  
-e, --env : Définir une variable d'environnement.  
--name : Attribuer un nom spécifique au conteneur.  
--rm : Supprimer le conteneur après exécution.  
--memory : Limiter la RAM utilisée par le conteneur.  
--cpus : Limiter le nombre de CPU utilisés.  
--restart : Définir une stratégie de redémarrage automatique.  
--network : Connecter le conteneur à un réseau spécifique.  
--privileged : Donner un accès root complet au conteneur.
```



Docker compose 101

Ou comment gérer plusieurs conteneurs facilement

Docker compose



Définir et exécuter plusieurs conteneurs avec un simple fichier YAML

- Facilite le **déploiement** d'applications multi-conteneurs (ex. : API + base de données).
- Remplace les commandes docker run multiples par **une seule configuration centralisée**.
- Permet de gérer les **réseaux, volumes et variables** d'environnement plus facilement.

services : définit les conteneurs (app et db).

build : permet de construire l'image de my_app avec un Dockerfile.

ports : expose le port 5000 de l'application.

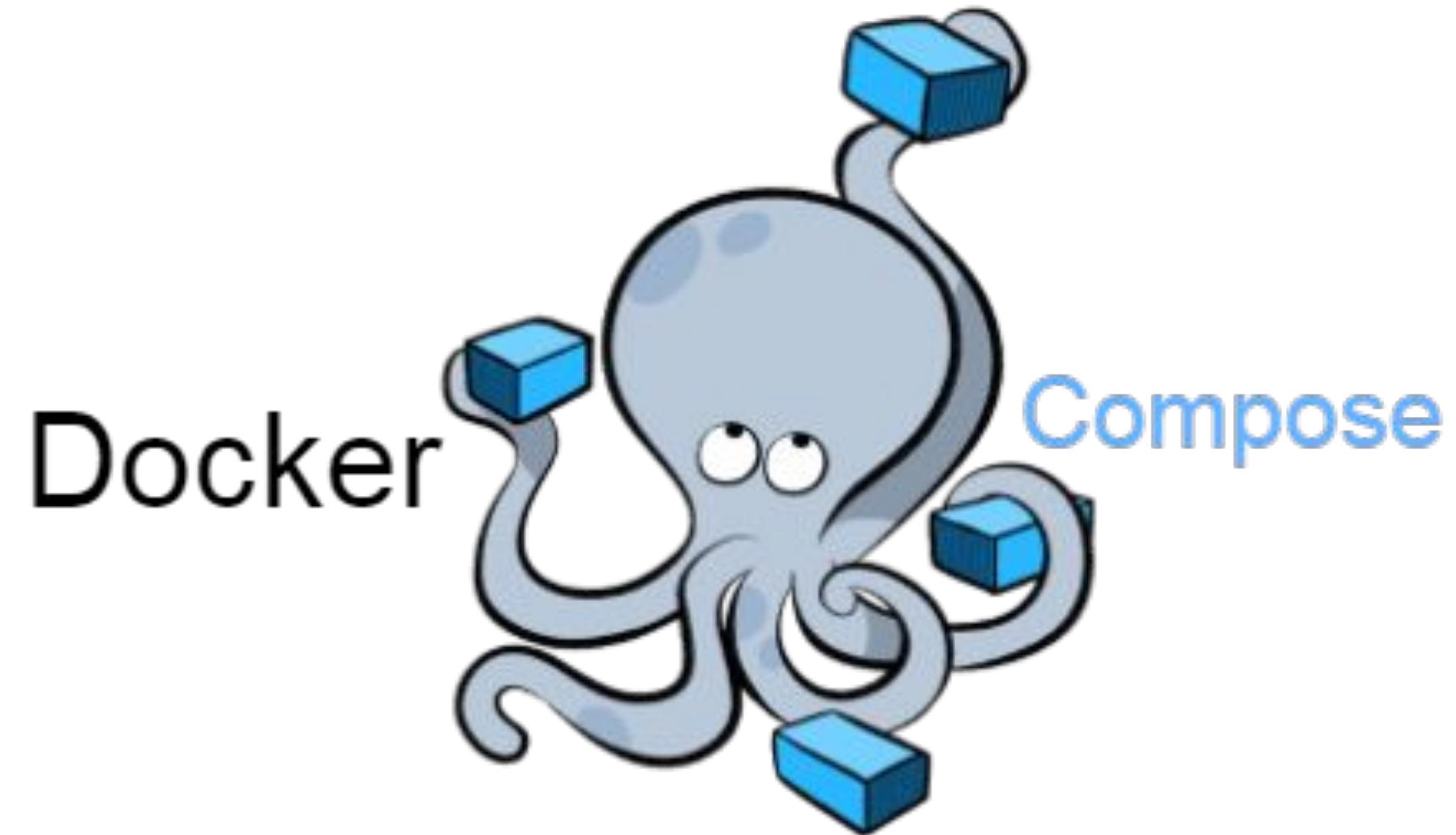
depends_on : assure que db démarre avant app.

volumes : stocke les données de PostgreSQL de manière persistante.

docker-compose up/down pour démarrer ou stopper les services

docker-compose down -v supprime également les volumes persistants

docker-compose logs affiche les logs des services



Docker compose



Exemple de `docker-compose.yml`

```
version: '3.8'

services:
  app:
    image: my_app
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

L'équivalent sans docker-compose

```
# Créer un réseau personnalisé pour que app et db puissent communiquer
docker network create my_network

# Créer un volume pour stocker les données de PostgreSQL
docker volume create db_data

# Lancer la base de données PostgreSQL
docker run --detach --name db --network my_network \
  -e POSTGRES_USER=user -e POSTGRES_PASSWORD=password \
  --volume db_data:/var/lib/postgresql/data \
  postgres

# Construire l'image de l'application
docker build -t my_app .

# Lancer l'application en la connectant à la base de données
docker run -d --name app --network my_network -p 5000:5000 my_app
```

Mise en pratique

Comprendre et manipuler images et conteneurs

Exercice noté 01



Objectifs

- Groupes de 3, durée : 3 heures
- Créer un docker-compose.yml avec un environnement attaqué vulnérable à une SQLi (service web, et BDD), et un environnement attaquant (base debian, avec SQLMap).
- Au lancement du docker-compose, les services doivent être montés et la machine attaquant doit procéder au dump complet de base de données, en autonomie.

Contraintes

- Ne pas utiliser de service préconfiguré, que ce soit pour l'environnement attaqué ou attaquant, tel que DVWA, Kali, Exegol, etc.

Libertés

Vous pouvez vous aider de ChatGPT & consort, mais vous devrez être en mesure d'expliquer vos choix

Exercice noté 01 (critères)



Sur 20 points (4 * 5 pts), constitue la première partie de la note du cours

- Structure et fonctionnement du docker-compose.yml : tous les services sont bien définis (web, db, attacker), le réseau est correctement configuré, les dépendances entre services sont gérées
- Création et exécution correcte de l'application vulnérable : l'image web est bien construite depuis le Dockerfile : la BDD est fonctionnelle et contient les données (init), l'app est accessible sur <http://localhost:8080> et la faille SQLi est exploitable.
- Configuration correcte de la machine attaquante : attacker est bien construite depuis le Dockerfile, connecté au réseau et peut interagir avec l'app, sqlmap est installé et fonctionnel
- Automatisation de l'attaque SQLi : l'attaque est autonome, sqlmap extrait les bases de données et affiche les résultats

Livrable : envoyer [rapport.pdf](#) à charlie@2600.eu (contenant les snippets docker-compose, dockerfiles, ressources, ...). Présenter, expliquer les choix, démontrer l'exécution. Entête avec le nom des membres

Docker, aller plus loin

Creuser dans les layers, émuler une autre architecture, ...

Comprendre les layers



Quel est le problème dans l'image suivante ? (sachant que .env contient un token secret)

```
FROM python:3.10-slim

COPY .env /app/.env
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY auth.py /app/auth.py
RUN rm /app/.env

CMD ["python3", "/app/auth.py"]
```

MacBook Pro /tmp \$ docker run -ti --entrypoint bash challenge_secret:

root@e1786f8d70f0:/app# ls -alh

total 8.0K

drwxr-xr-x 1 root root 8 Mar 11 18:43 .

drwxr-xr-x 1 root root 12 Mar 11 19:03 ..

-rw-r--r-- 1 root root 694 Mar 11 18:42 auth.py

-rw-r--r-- 1 root root 23 Mar 11 18:43 requirements.txt

root@e1786f8d70f0:/app# cat /app/.env

cat: /app/.env: No such file or directory

root@e1786f8d70f0:/app# █

Comprendre les layers



Identifier la couche en cause et son digest avec docker history, docker inspect ou dive

The screenshot shows the 'dive challenge_secret' interface. On the left, there's a tree view of 'Layers' with a color-coded legend (green for blobs, purple for buildkit). The layers listed are:

Cmp	Size	Command
97 MB	FROM blobs	
9.2 MB	RUN /bin/sh -c set -eux; apt-get update; apt-g	
46 MB	RUN /bin/sh -c set -eux; savedAptMark="\$(apt-m	
0 B	RUN /bin/sh -c set -eux; for src in idle3 pip3 pyd	
43 B	COPY .env /app/.env # buildkit	
0 B	WORKDIR /app	

On the right, under 'Current Layer Contents', there's a tree view of the 'app' directory containing a '.env' file.

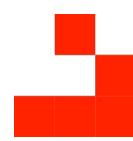
Below the layers, 'Layer Details' are shown:

- Tags: (unavailable)
- Id: blobs
- Digest: sha256:2ab82eacf571e28bb0cf4250bd8b27ca51e7026a75f8e6536c1d2a4d7fc02387
- Command: COPY .env /app/.env # buildkit

Under 'Image Details':

- Image name: challenge_secret
- Total Image size: 163 MB
- Potential wasted space: 5.5 MB
- Image efficiency score: 97 %

At the bottom, there's a navigation bar with commands: ^C Quit, Tab Switch view, ^F Filter, ^L Show layer changes, and ^A Show aggregated changes.



Comprendre les layers



Exporter les layers et récupérer la donnée intéressante

Docker + QEMU



QEMU est un émulateur de processeur. Permet d'exécuter des apps ou conteneurs conçus pour une archi différente.

On peut combiner émulation et conteneurisation ! Les intérêts : utiliser/RE des outils limités à une seule archi, tester le déploiement/dev à destination d'autres archi.

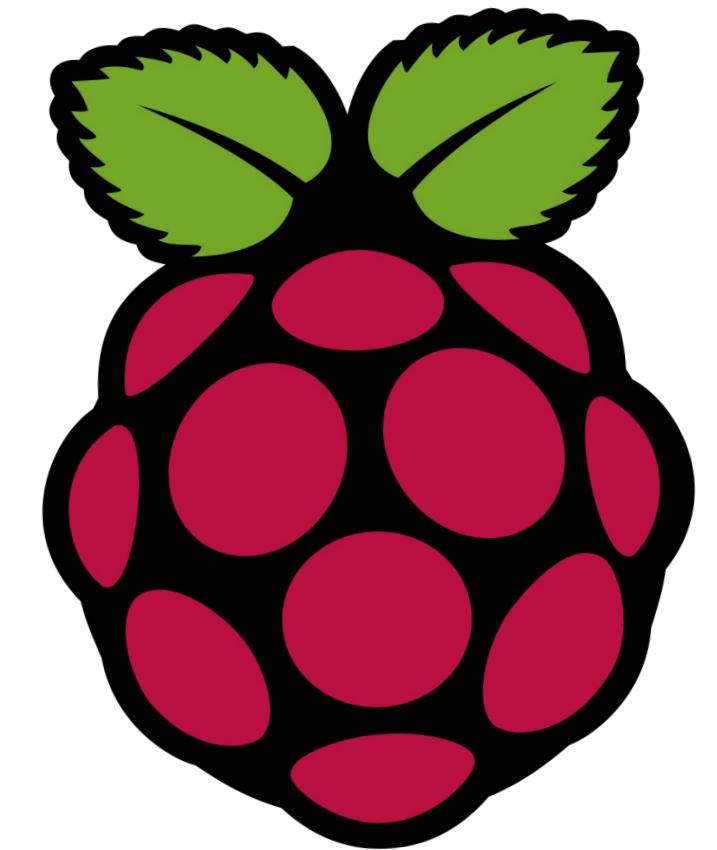
```
# Installer QEMU et activer l'émulation multi-architecture (⚠️ modifie le kernel)
docker run --rm --privileged tonistiigi/binfmt --install all

# Vérifier les architectures supportées
docker buildx ls

# Activer buildx pour gérer plusieurs architectures
docker buildx create --use

# Construire une image pour x86_64 et ARM64
docker buildx build --platform linux/amd64,linux/arm64 -t mon_image_multarch .

# Exécuter un conteneur en spécifiant l'archi utilisée
docker run --rm --platform linux/arm64 monrepo/mon_image_multarch
```



Capabilities



Sur Linux, les capabilities sont des permissions système qui permettent de décomposer les priviléges root en plusieurs droits spécifiques.

Sur Docker, le principe est le même. Les capabilities sont des permissions système accordées aux processus dans un conteneur. Elles permettent de contrôler finement les droits du conteneur sans lui donner un accès root complet.

Elles peuvent être contrôlées avec les options **--cap-add** et **--cap-drop** lors d'un **docker run** ou **docker create**.

Quelques capabilities principales :

- **NET_ADMIN** : permet de modifier la configuration réseau, tel qu'ajouter des interfaces, modifier les routes ou activer/désactiver des connexions.
- **SYS_MODULE** : permet de charger et décharger des modules kernel
- **SYS_RAWIO** : donne un accès direct aux opérations d'entrée/sortie sur les périphériques, ce qui peut permettre de manipuler du matériel bas niveau.
- **SYS_ADMIN** : fournit de nombreux priviléges avancés, comme le montage de systèmes de fichiers, la modification de namespaces, et d'autres actions proches du contrôle root.

⚠ Warning : attention, en fonction des capabilities qui sont set, la porosité augmente !

Mode privileged

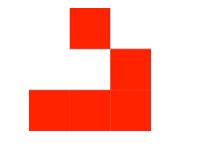


Le mode **--privileged** est une option Docker (`docker create`, `docker run`) qui donne au conteneur tous les priviléges root possibles, y compris l'accès à l'ensemble des devices et des capabilities du système hôte. Donne quasiment les mêmes droits qu'un process root sur l'hôte.

⚠ Warning : y a-t-il réellement besoin d'expliquer les risques ?

Il y a des cas d'usages légitimes : accès aux périphériques spécifiques (USB, cartes réseau, GPU), exécution de conteneurs nécessitant des modules noyau (conteneurs systemd, installation de QEMU via binfmt qui modifie le noyau Linux) → **MAIS** en général, on peut remplacer le mode privilégié par l'attribut des bonnes capabilities, principe du moindre privilège





Parlons de podman



Podman (POD Manager), similaire à Docker, mais avec une approche plus sécurisée.

- **Sans daemon** : ne fonctionne pas avec un démon comme Docker le fait avec dockerd (qui a un accès privilégié -root- à tout le système).
Une faille dans dockerd = une compromission complète. "Facilite" les cas d'escape to takeover.
- **Rootless** : Podman exécute les conteneurs sans droits root. Docker supporte désormais ce mode mais ça n'a pas toujours été le cas.
- **Docker-compatible** : même commandes et images (`docker pull`, `docker run`, `docker-compose`)
- **Kubernetes-like** : sait gérer plusieurs "conteneurs" (i.e. pods) un peu comme k8s

Bien que podman soit plus sécurisant, Docker reste largement adopté pour plusieurs raisons :

- **Adoption massive** (existe depuis plus longtemps, plus de ressources communautaires, plus connu)
- Docker utilisé un daemon, ce qui est moins sécurisant, mais **plus simple**
- Fonctionne avec **docker-compose**, mais nécessite l'installation d'un wrapper podman-compose
- Intégration native avec **Kubernetes**
- Podman pas encore tout à fait stable sur Windows (mais Docker Desktop, c'est pas glorieux non plus 💩)
- **Support commercial** et outils pro (Docker Hub, Docker Desktop, Docker Trusted Registry, ...)





Comprendre un cas réel

Parlons d'Exegol (évidemment **00**)





L'orchestration

Kubernetes

L'orchestration, kesako ?

zo
gr

Ou comment gérer automatiquement le lifecycle des containers dans un environnement distribué.

Gérer un seul container est simple, mais dès que l'on passe à plusieurs instances, sur plusieurs machines, ça l'est moins.

L'orchestration aide au déploiement et scalability, redémarrage automatique, gestion réseau, allocation de ressources etc.

Principales solutions d'orchestration :

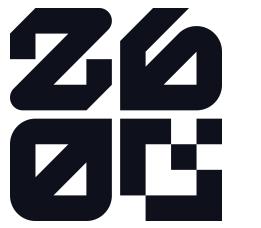
- **Docker Compose** : d'une certaine manière, c'est de l'orchestration aussi, mais sur une même machine, donc plus simple mais pas scalable
- **Docker Swarm** : simple, intégré à Docker, adapté aux déploiements de petite échelle
- **Kubernetes (k8s)** : référence, conçue pour gérer des architectures complexes et multi-noeuds



reboot manuel
pas scalable
risques de coupure
config manuelle

auto-healing
auto-scaling
rolling update
auto-routing

Docker Swarm



Ou comment orchestrer plusieurs containers sur plusieurs machines sans complexité excessive.

Le mode d'orchestration **intégré à Docker**, permet de gérer plusieurs containers sur plusieurs nœuds comme un seul et même cluster.

Plus simple que Kube (syntaxe similaire à Docker Compose, natif dans Docker), mais offre moins de fonctionnalités avancées.

Architecture de Docker Swarm

- **Managers** : gèrent l'état du cluster, prennent les décisions de planification et répliquent les services.
- **Workers** : exécutent les containers, reçoivent les instructions des managers.
- **Services** : définissent le nombre de réplicas et la configuration des containers.
- **Overlay network** : permet aux services de communiquer entre eux, quel que soit le nœud sur lequel ils s'exécutent.

<https://docs.docker.com/engine/swarm/>

The screenshot shows the Docker Docs website with a blue header. The navigation bar includes links for 'dockerdocs', 'Get started', 'Guides', 'Manuals' (which is underlined), and 'Reference'. On the right side of the header are 'Search' and 'Ask AI' buttons. The main content area has a dark background. On the left, there's a sidebar with 'Manuals' expanded, showing sections for 'OPEN SOURCE', 'Docker Engine', 'Install', 'Storage', and 'Networking'. The main content area shows the breadcrumb path: Home / Manuals / Docker Engine / Swarm mode. The title 'Swarm mode' is prominently displayed. Below it is a 'Note' section with the text: 'Swarm mode is an advanced feature for managing a cluster of Docker daemons.' To the right of the content are several interactive buttons: 'Edit this page', 'Request changes', 'Table of contents', 'Feature highlights' (which is highlighted in grey), and brief descriptions of 'Cluster management integrated with Docker Engine', 'Decentralized design', and 'Declarative service model'.

Pratiquer Docker Swarm



Par groupes de 4

- **Manager Swarm** (1 personne) → Initialiser et superviser le cluster.
- **Administrateurs de nœuds workers** (2 personnes) → Rejoindre le cluster et surveiller les containers.
- **Testeur** (1 personne) → Tester l'application et analyser la réaction du cluster aux pannes.

Etape 1 : initialiser le cluster

```
# Faire de l'hôte un Manager du cluster Swarm (Manager)
docker swarm init

# Afficher un token pour que des Workers puissent rejoindre le cluster (Manager)
docker swarm join-token worker

# Rejoindre le cluster en tant que Worker (Administrateurs)
docker swarm join --token <TOKEN> <IP_DU_MANAGER>:2377

# Vérifier l'état du cluster (Manager)
docker node ls
```

Pratiquer Docker Swarm



Etape 2 : déployer un service répliqué

```
# Créer un service Nginx avec 5 replicas (Manager)
docker service create --name web --replicas 5 -p 8080:80 nginx

# Vérifier que les conteneurs sont répartis (Manager)
docker service ps web

# Accéder à l'application (Testeur)
curl <IP_MANAGER>:8080
```

Pratiquer Docker Swarm



Etape 3 : simuler une panne

```
# Surveiller les services (Manager)
watch docker service ps web

# Surveiller la conso (Administrateur)
docker stats

# Surveiller les events (Administrateur)
docker events

# Rendre un noeud indisponible de force (Manager) (cf. docker service)
docker node update --availability drain <NODE_ID>

# Vérifier que l'application n'en souffre pas (Testeur)
curl <IP_MANAGER>:8080

# Vérifier que le noeud est redéployé (Manager)
docker service ps web
```

Pratiquer Docker Swarm



Etape 4 : scaler

```
# Surveiller les services (Manager)
watch docker service ps web

# Surveiller la conso (Administrateur)
docker stats

# Surveiller les events (Administrateur)
docker events

# Augmenter le nombre de replicas (Manager)
docker service scale web=8

# Vérifier que l'application n'en souffre pas (Testeur)
curl <IP_MANAGER>:8080

# Constater les changements dans le watch, stats, events (Administrateur)
```

Kubernetes [coo-ber-net-ease]



Kubernetes (k8s) est une plateforme d'orchestration de conteneurs permettant d'automatiser le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Il a été conçu pour gérer des applications distribuées en production.

Fun fact : pourquoi k8s (k ubernetes) ? Comme pour i18n (i nternationalization) ou l10n (l ocalization)

- Gère automatiquement le cycle de vie des conteneurs.
- Assure la haute disponibilité et l'auto-réparation des applications.
- Offre une scalabilité dynamique adaptée à la charge de travail.
- Standardise le déploiement multi-cloud et on-premise.

Composants clés

- **Pods** : plus petite unité de Kubernetes, regroupe un ou plusieurs conteneurs
- **Deployments** : gestion déclarative des déploiements, mise à jour progressive des pods
- **Services** : expose les pods et permet leur découverte réseau
- **ConfigMaps et Secrets** : gestion des configurations et variables d'environnement
- **Nodes** : machines (physiques ou virtuelles) qui exécutent les pods
- **API Server** : interface centrale pour gérer l'état du cluster



L'architecture de Kubernetes



Kubernetes reprend les concepts de base de Docker Swarm tout en les enrichissant avec des composants spécialisés, permettant une gestion plus fine et plus robuste du cluster. Son architecture modulaire apporte des fonctionnalités avancées.

En comparaison de Docker Swarm, k8s présente les rôles suivants

Docker Swarm	k8s	Description
Manager	Control Plane	Gérer l'orchestration
Worker	Worker Node	Exécuter les conteneurs
Tasks	Pod	Instance d'un service
	etcd	BDD qui stocke l'état du cluster
Configs & Secrets	ConfigMaps & Secrets	Gestion des configs et variables sensibles
	Ingress Controller	Gestion du trafic et de l'accès aux services
Stack	Namespaces	Isolation des ressources pour organiser des events distincts

Kubernetes utilise les mêmes mécanismes que Docker pour isoler les ressources (CPU, mémoire, réseau).

Mais pas de démon unique comme dockerd. Kubelet peut tourner en rootless et applique des règles RBAC.

K8s apporte des éléments dont Docker Swarm manque : Network Policies & Service Accounts

L'unité de base : les pods



Représente un ou plusieurs conteneurs qui partagent le même environnement d'exécution.

Contrairement à Docker où un conteneur est isolé, Kubernetes fonctionne avec des Pods qui regroupent un ou plusieurs conteneurs fonctionnant ensemble.

Dans Docker Swarm, on avait vu qu'un noeud Worker gérait plusieurs Tasks avec un conteneur par Task.

Avec k8s, un Noeud Worker gère des Pods, qui peuvent être multi-conteneurs (e.g., un serveur web et un collecteur de logs dans le même Pod). Permet la gestion unifiée du cycle de vie de conteneurs qui doivent fonctionner ensemble.

Les conteneurs d'un Pod partagent la même IP et peuvent communiquer entre eux via **localhost**. Dans Swarm, chaque conteneur a généralement sa propre IP. E.g., Une application backend et un cache Redis dans un même Pod peuvent communiquer directement sans passer par un service réseau.

Un Pod peut être redémarré et déplacé sur un autre nœud si nécessaire. Kubernetes gère des stratégies comme les **Rolling Updates** (remplacement progressif des Pods pour éviter une coupure de service), ou **l'Auto-scaling** (augmente ou réduit dynamiquement le nombre de Pods).

Installation de k8s

ZD
NET

L'installation de Kubernetes dépend du contexte :

- **En local pour le développement** → kind, minikube, OrbStack
- Sur un serveur ou VM → kubeadm
- Sur le cloud → GKE (Google Cloud), EKS (AWS), AKS (Azure)



Déploiement de conteneurs



Contrairement à Docker où un conteneur est lancé avec `docker run`, Kubernetes utilise des "Deployments", qui permettent de :

- Automatiser le lancement et la gestion des conteneurs.
- Gérer les mises à jour sans interruption.
- Assurer la haute disponibilité en répliquant les instances.
- Redémarrer automatiquement les pods en cas de panne.

```
# Créer le déploiement
kubectl apply -f manifest.yaml

# Vérifier les pods en cours d'exec
kubectl get pods

# Observer l'état du déploiement
kubectl get deployments

# MAJ l'image d'un déploiement
kubectl set image deployment/nginx-deployment nginx=nginx:1.21
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:latest
    ports:
      - containerPort: 80
```

ClusterIP, NodePort, LoadBalancer



Les Pods Kubernetes ont une adresse IP éphémère et ne sont pas accessibles directement depuis l'extérieur du cluster. Kubernetes utilise des **Services pour permettre la communication** entre Pods et l'exposition des applications.

Trois types de Services permettent cette communication :

- **ClusterIP** : communication interne uniquement (type de Service par défaut).
- **NodePort** : accès externe via un port fixe exposé sur chaque nœud du cluster.
- **LoadBalancer** : accès externe via une IP publique fournie par un provider cloud.



ClusterIP, NodePort, LoadBalancer



Déployer un service ClusterIP et vérifier qu'il est uniquement accessible depuis l'intérieur du cluster.

```
# nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

```
# nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

```
kubectl apply -f nginx-deployment.yaml
kubectl apply -f nginx-service.yaml
# Vérifier que les Pods sont bien en cours
d'exécution
kubectl get pods
# Lister les services et noter l'IP
ClusterIP attribuée
kubectl get services
# Tester l'accès au service
kubectl run --rm -it --image=busybox
--restart=Never test-pod -- wget
-q0- http://nginx-service
```

ClusterIP, NodePort, LoadBalancer



Déployer un service NodePort et vérifier qu'il est accessible depuis l'extérieur du cluster.

```
# nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

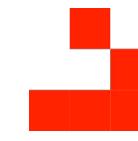
```
# nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30080
  type: NodePort
```

```
kubectl apply -f nginx-service.yaml

# Lister les services et noter l'IP
ClusterIP attribuée
kubectl get services

# Récupérer l'IP d'un nœud du cluster
kubectl get nodes -o wide

# Tester l'accès au service
curl <NODE_IP>:30080
```



Namespaces



Par défaut, toutes les ressources (Pods, Services, ConfigMaps, etc.) sont créées dans le namespace default.

Les namespaces permettent de :

- **Isoler** les environnements (exemple : dev, staging, production).
- **Gérer les accès** et permissions via RBAC.
- **Limiter l'utilisation** des ressources (quotas CPU/RAM par namespace).
- **Éviter les conflits** de noms

Namespaces



Par défaut, toutes les ressources (Pods, Services, ConfigMaps, etc.) sont créées dans le namespace default.

```
# nginx-dev.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-dev
  namespace: dev
spec:
  containers:
    - name: nginx
      image: nginx
```

```
# nginx-prod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-prod
  namespace: prod
spec:
  containers:
    - name: nginx
```

```
# Créer deux namespaces distincts
kubectl create namespace dev
kubectl create namespace prod

# Vérifier
kubectl get namespaces

# Appliquer les fichiers
kubectl apply -f nginx-dev.yaml
kubectl apply -f nginx-prod.yaml

# Lister les pods
kubectl get pods --all-namespaces

# Tester l'accès dev → prod
kubectl run --rm -it --image=busybox -n dev
--restart=Never test-pod -- wget -qO-
http://nginx-prod

# Cleanup
kubectl delete namespace dev
kubectl delete namespace prod
```

Volumes et persistance des données



Ou comment stocker et conserver les données des conteneurs au-delà de leur cycle de vie.

Par défaut, les Pods sont éphémères (Pod supprimé ou redémarré = données perdues, pas de conservation des fichiers après MAJ ou crash).

Kube propose plusieurs solutions:

- **EmptyDir** : stockage temporaire, supprimé avec le Pod, partage de fichiers entre conteneurs
- **HostPath** : montage d'un répertoire du nœud, non recommandé en production (stockage dépend du nœud où le Pod est exécuté)
- **PersistentVolume (PV) et PersistentVolumeClaim (PVC)** : solution standard pour la persistance, avec support des backends externes (NFS, Ceph, cloud).

Séparation du stockage physique et de son utilisation par les Pods

- Un PV est une **ressource de stockage** physique (disque, NFS, Cloud Storage).
- Un PVC est une **demande de stockage** faite par un Pod.
- Kubernetes fait correspondre un PVC à un PV disponible automatiquement.

PV et PVC



Où comment stocker et conserver les données des conteneurs au-delà de leur cycle de vie.

```
# pv-pvc.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-test
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/test-data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-test
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

```
# pvc-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pvc-pod
spec:
  containers:
    - name: busybox
      image: busybox
      stdin: true
      tty: true
      volumeMounts:
        - mountPath: "/data"
          name: storage
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: pvc-test
```

```
# Créer les ressources
kubectl apply -f pv-pvc.yaml
kubectl apply -f pvc-pod.yaml

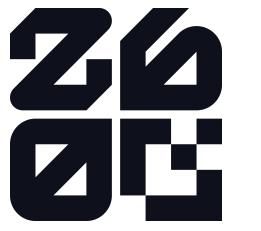
# Vérifier que le PV et le PVC sont liés
kubectl get pv
kubectl get pvc
kubectl get pods

# Écrire un fichier dans un pod
kubectl exec -it pvc-pod -- sh
echo "Test de persistance" > /data/test-file.txt
ls /data
cat /data/test-file.txt
exit

# Supprimer le Pod, le recréer, afficher la donnée persistée
kubectl delete pod pvc-pod
kubectl apply -f pvc-pod.yaml
kubectl exec -it pvc-pod -- sh -c "ls /data && cat /data/test-file.txt"

# Cleanup
kubectl delete pod pvc-pod
kubectl delete pvc pvc-test
kubectl delete pv pv-test
```

ConfigMaps et Secrets



Gestion de la configuration et les données sensibles des applications de manière modulaire et sécurisée.

Les applications conteneurisées nécessitent des configurations dynamiques et des informations sensibles (clés API, mots de passe). Stocker ces valeurs directement dans les fichiers YAML des déploiements pose des problèmes de sécurité et de gestion (modification difficile des configurations sans redéploiement, risques de sécurité si les mots de passe sont stockés en clair dans les manifestes k8s)

- **ConfigMap** : stocke des configurations non sensibles (variables d'environnement, fichiers de configuration). Stocké en clair, accessible.
- **Secret** : stocke des données sensibles encodées en Base64 (clés API, certificats, mots de passe). Stocké encodé, mais accès restreint (RBAC)

```
# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: "production"
  API_URL: "https://api.example.com"
```

```
# secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  DB_PASSWORD: U3VwZXJtZWNyZXQ=
```

```
# configmap-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: app
      ...
      env:
        - name: APP_ENV
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: APP_ENV
        - name: LOG_LEVEL
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: DB_PASSWORD
      volumes:
        - name: config-volume
          configMap:
            name: app-config
```



Liveness et Readiness Probes

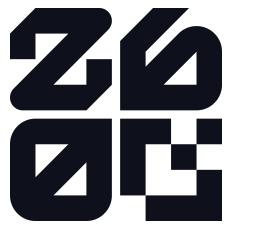


Pour surveiller l'état des applications et gérer leur disponibilité.

Dans un cluster Kubernetes, un conteneur peut être techniquement en cours d'exécution mais non fonctionnel. Kubernetes utilise les probes pour détecter ces situations et agir en conséquence.

- **Readiness Probe** : vérifie si un conteneur est prêt à recevoir du trafic. Tant que la probe échoue, Kubernetes ne redirige pas de requêtes vers ce Pod (e.g., print un fichier).
- **Liveness Probe** : vérifie si un conteneur est toujours vivant. Si la probe échoue, Kubernetes redémarre le conteneur (e.g., interagir avec le service).

```
# pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod
spec:
  containers:
    - name: app
      image: nginx
      readinessProbe:
        exec:
          command: ["cat", "/tmp/ready"]
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 10
```



Qui peut faire quoi dans un cluster.

RBAC (Role-Based Access Control) permet de définir précisément les permissions des utilisateurs et des services dans un cluster Kubernetes. Sans RBAC, n'importe quel utilisateur avec accès à kubectl pourrait modifier ou supprimer des ressources critiques.

```
# role-pod-reader.yaml
apiVersion:
rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default # <-- Namespace
  restraint
  name: pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list"]
```

```
# rolebinding-pod-reader.yaml
apiVersion:
rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: default # <-- Namespace
  restraint
  subjects:
    - kind: User
      name: alice
      apiGroup: rbac.authorization.k8s.io
  roleRef:
    kind: Role
    name: pod-reader
    apiGroup: rbac.authorization.k8s.io
```

```
# cluster-role-reader.yaml
apiVersion:
rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list"]
```

```
# cluster-pod-reader-binding.yaml
apiVersion:
rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-pod-reader-binding
  subjects:
    - kind: User
      name: bob
      apiGroup: rbac.authorization.k8s.io
  roleRef:
    kind: ClusterRole
    name: cluster-pod-reader
    apiGroup: rbac.authorization.k8s.io
```

Network Policies



Par défaut, tous les Pods dans un cluster peuvent communiquer librement entre eux.

Ce qui pose des problèmes de sécu et d'isolation :

- Un service compromis peut attaquer d'autres Pods.
- Des bases de données internes peuvent être accessibles par erreur.
- Aucune segmentation réseau entre environnements (dev, prod, test).

Les Network Policies permettent de restreindre et filtrer le trafic réseau entre Pods.

```
# Interdire les connexions sortantes d'un Pod
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: restricted-app
  policyTypes:
    - Egress # Règle pour le trafic sortant
  egress: []
```

```
# Bloquer tout accès à un Pod sauf depuis un autre Pod précis
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: backend # La règle s'applique aux Pods "backend"
  policyTypes:
    - Ingress # Règle pour le trafic entrant
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend # Seuls les Pods "frontend"
peuvent accéder
  ports:
    - protocol: TCP
      port: 80
```

Service Account



Ou comment Kubernetes gère l'authentification des Pods pour interagir avec l'API du cluster.

Par défaut, les Pods n'ont pas d'identité propre, mais ils doivent parfois :

- **Lire** des ConfigMaps ou Secrets.
- **Communiquer** avec l'API Kubernetes.
- **Automatiser** des tâches via des contrôleurs internes.

Un Service Account est un compte spécifique attribué aux Pods, leur permettant de s'authentifier auprès de l'API du cluster.

Le Service Account utilisé par défaut a des permissions très limitées.

```
# Lister les Service Accounts
kubectl get serviceaccounts

# Voir les détails d'un Service Account
kubectl describe serviceaccount custom-sa

# Vérifier le token d'un Pod utilisant un SA
kubectl exec -it pod-with-sa -- cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

Service Account



```
# Service Account
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-sa
  namespace: default
```

```
# Service Account is Bound to a Role
apiVersion:
rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
subjects:
- kind: ServiceAccount
  name: custom-sa
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

```
# Role definition
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list"]
```

```
# Pod uses Service Account
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-sa
spec:
  serviceAccountName: custom-sa
  containers:
    - name: app
      image: busybox
```

Kubernetes dans le Cloud



La complexité du on-prem en moins, des environnements très managés, avec du scaling très important, et du usage-based billing

Principaux fournisseurs : **AWS EKS** (Elastic Kubernetes Service), **Google Cloud GKE** (Google Kubernetes Engine), **Azure AKS** (Azure Kubernetes Service), **IBM Cloud IKS** (IBM Kubernetes Service), **Oracle Cloud OKE** (Oracle Kubernetes Engine), **DigitalOcean DOKS** (DigitalOcean Kubernetes)

- Le cloud provider gère le control plane (API Server, etcd, Scheduler).
- L'utilisateur gère uniquement les worker nodes et les ressources déployées.
- Intégration avec le cloud simplifie le stockage (EBS, GCS, Azure Disk) et le réseau (LoadBalancer, DNS)
- Intégrations natives pour la sécurité (IAM)
- Auto-scaling (minReplicas, maxReplicas, averageUtilization target pour cpu par exemple)

Mise en pratique

Déployer un cluster vulnérable

Exercice noté 02



Objectifs

- Groupes de 3, durée : 3 heures
- Déployer un cluster Kubernetes sur vos machines, avec un control plane et des worker nodes.
- Configurer les services du cluster en appliquant les concepts vus ensemble (ConfigMaps, Secrets, RBAC, Network Policies, Services, etc.).
- Introduire des vulnérabilités intentionnelles permettant un chemin d'attaque précis.
- Exploiter le cluster en appliquant les principes d'attaque et d'escalade.

Scénario proposé

- Service web exposé sur le réseau, vulnérable à une injection de commande → escalade de privilège via un sudo NOPASSWD sur python → découverte réseau d'un autre pod qui n'était pas accessible à l'extérieur mais qui l'est entre Pods → découverte d'un service SSH accessible à root avec mot de passe faible → escape du Pod qui était alors privileged (peut exécuter des commandes système sur l'hôte) → prise de contrôle totale, et arrêt, du cluster (ClusterRole trop permissif, appliqué au namespace dans lequel le Pod se trouve)

✓ Exercice noté 02 (critères)



Sur 20 points, constitue la première partie de la note du cours

- (8 points) Mise en place du cluster et des concepts vu en cours
- (7 points) Progression dans la mise en place du chemin d'attaque
- (5 points) Explications / rapport
- (Bonus 2 points) Propositions de remédiation

Livrable : envoyer [rapport.pdf](#) à charlie@2600.eu (contenant les snippets). Présenter, expliquer les choix, démontrer l'exécution. Entête avec le nom des membres

