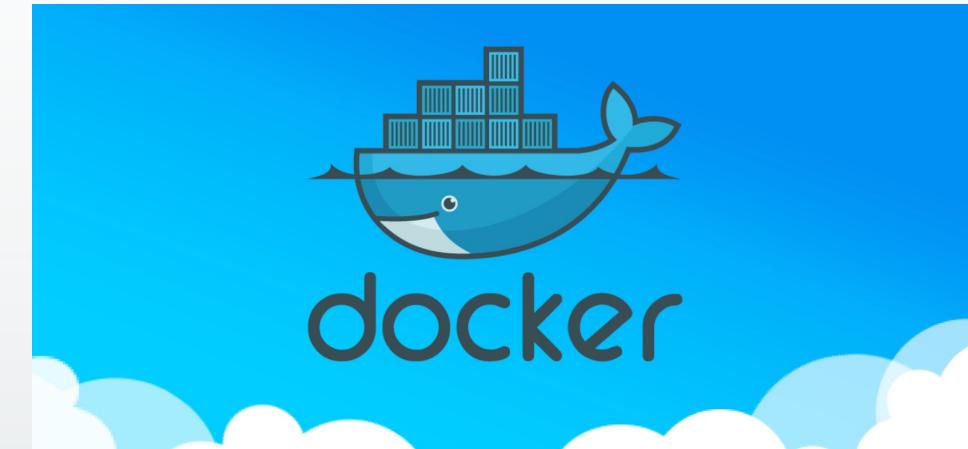


1

Docker

Cours dispensé par
Tancrède SUARD
Promo 24/25

cv@tancredesuard.fr



Plan de cours

Chapitre 1 Principes fondamentaux

Chapitre 2 Installation

Chapitre 3 Premiers pas avec Docker

Chapitre 4 Création et gestion d'images Docker

Chapitre 5 Mise en oeuvre pratique

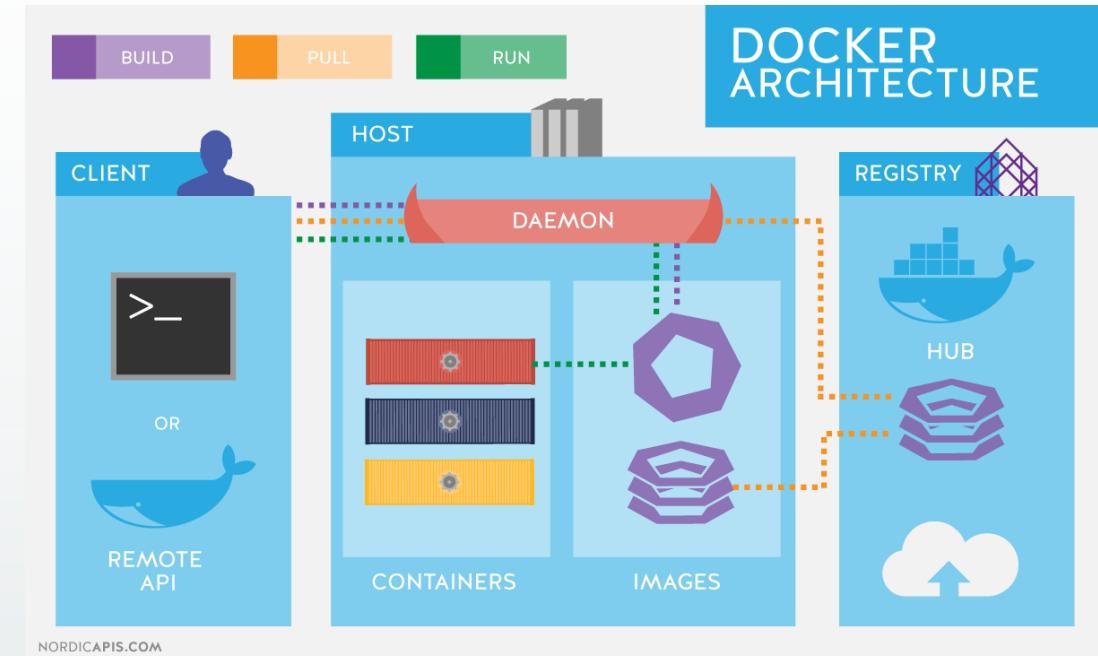
Chapitre 6 Orchestration par Docker Compose

Chapitre 7 Déploiement en cluster par une usine logicielle

Chapitre 1

Principes fondamentaux

- ▶ 1.1. Positionnement de Docker
- ▶ 1.2. Principe de conteneurs
- ▶ 1.3. Les fondements de Docker
- ▶ 1.4. Les plus de Docker
- ▶ 1.5. Architecture de services



1.1. Positionnement de Docker

- ▶ 1.1.1. La problématique racine
- ▶ 1.1.2. Approche par augmentation des ressources physiques
- ▶ 1.1.3. Approche par parallélisation des traitements
- ▶ 1.1.4. L'approche par virtualisation
- ▶ 1.1.5. Tentatives de réduction de la consommation de ressources
- ▶ 1.1.6. Comment Docker règle radicalement le problème
- ▶ 1.1.7. Positionnement de Docker par rapport à la virtualisation
- ▶ 1.1.8. Positionnement de Docker par rapport aux clusters
- ▶ 1.1.9. Positionnement de Docker par rapport aux approches de bacs-à-sable

1.1.1. La problématique racine

- ▶ Dans les systèmes d'information traditionnels, plusieurs problèmes freinent l'agilité et la scalabilité des applications :
 - **Incohérences d'environnement** : une application fonctionne en développement mais échoue en production.
 - **Déploiements lourds** : l'installation et la configuration sont manuelles, longues et souvent sources d'erreurs.
 - **Surcoûts liés aux ressources** : chaque application nécessite un OS complet, ce qui entraîne une surconsommation mémoire/CPU.
 - **Faible portabilité** : les dépendances sont liées à la machine hôte, rendant les migrations complexes.
- ▶👉 Ces limites deviennent critiques avec l'essor du cloud, de la microservicisation et des cycles de livraison rapides (DevOps).

1.1.2. Approche par augmentation des ressources physiques

- ▶ Pendant longtemps, la réponse aux besoins croissants des applications passait par une stratégie simple :
 - ▶ **Ajouter du matériel (scale-up)**
 - Plus de CPU, de RAM, de disques pour supporter des applications plus lourdes.
 - Solution **coûteuse**, peu **flexible** et rapidement **atteinte par des limites physiques**.
 - Ne répond pas aux besoins de **modularité** et de **réPLICATION rapide** des environnements.
 - ▶  Cette approche n'est plus viable dans un monde où l'élasticité, la rapidité de déploiement et la réduction des coûts sont clés.

1.1.3. Approche par parallélisation des traitements

- ▶ Pour améliorer les performances, une autre approche a été utilisée :
 - ▶ **Paralléliser les traitements (multitâche / multithreading / multiprocessing)**
 - Exploite les architectures **multi-cœurs** pour exécuter plusieurs tâches simultanément.
 - Permet un meilleur **temps de réponse** et une **augmentation du débit**.
 - **Limité par la nature de l'application** : certaines tâches sont difficilement parallélisables.
 - Ne répond pas à la problématique de **déploiement**, de **portabilité**, ni d'**isolement des environnements**.
- ▶ ➡️ Utile pour la performance brute, mais insuffisant pour les enjeux d'infrastructure moderne.

1.1.4. L'approche par virtualisation

- ▶ **La virtualisation** permet d'exécuter plusieurs systèmes d'exploitation sur une même machine physique :
 - Chaque **VM (machine virtuelle)** embarque un OS complet + les applications.
 - Fonctionne grâce à un **hyperviseur** (ex. : VMware, VirtualBox, Hyper-V).
 - Permet **l'isolement, le partage des ressources** et une **meilleure utilisation du matériel**.
- ▶ **Limites de la virtualisation :**
 - **Lourde en ressources** : chaque VM réplique un OS complet.
 - **Temps de démarrage lent**.
 - **Maintenance complexe** des images et des configurations.
- ▶ ➤ Bien qu'efficace, cette solution reste trop lourde face aux besoins actuels de **vitesse, légereté et scalabilité**.

1.1.5. Tentatives de réduction de la consommation de ressources

Face aux limites des VM, plusieurs solutions ont été explorées :

- ▶ **Optimisation des hyperviseurs**
 - Hyperviseurs plus légers (ex. : KVM, Xen) pour réduire l'empreinte des VM.
- ▶ **Utilisation de systèmes d'exploitation minimalistes**
 - Ex : CoreOS, Alpine Linux, TinyCore.
 - Réduction de l'espace disque et de la mémoire consommés.
- ▶ **Partage des ressources entre VM**
 - Mutualisation de certaines librairies ou du noyau, parfois au détriment de l'isolation.
- ▶ **Automatisation des déploiements**
 - Outils comme Vagrant ou Ansible pour limiter les duplications inutiles.
- ▶  Malgré ces efforts, le **coût d'exécution** reste élevé et les **gains marginaux** face aux nouvelles exigences de rapidité et d'élasticité.

1.1.6 Comment Docker règle radicalement le problème

- ▶ Docker propose une approche révolutionnaire :
→ **l'exécution d'applications dans des conteneurs légers, isolés et reproductibles.**
- ▶ **Ce que Docker change :**
 - **Pas de VM complète** : les conteneurs partagent le noyau de l'hôte.
 - **Ultra-rapide** à lancer (en quelques millisecondes).
 - **Très léger** : chaque conteneur ne contient que ce qui est strictement nécessaire.
 - **Portabilité totale** : "ça marche chez moi" devient "ça marche partout".
- ▶ **Résultats :**
 - Réduction massive de la consommation de ressources.
 - Déploiements automatisés, fiables et reproductibles.
 - Parfaitement adapté aux environnements modernes : **CI/CD, microservices, cloud.**

1.1.7. Positionnement de Docker par rapport à la virtualisation

▶ Virtualisation classique (VM) :

- Chaque VM embarque **un OS complet**.
- Nécessite un **hyperviseur**.
- **Lente à démarrer**, consomme beaucoup de ressources.
- Bon isolement, mais faible densité.

▶ Docker (conteneurs) :

- Partage le **noyau de l'hôte**, pas d'OS complet par conteneur.
- **Démarrage quasi-instantané**.
- Très **léger**, permet une **forte densité d'applications** par machine.
- Isolement suffisant pour la majorité des cas d'usage.

▶ Résumé :

Docker ne **remplace pas totalement** les VM, mais les **complète** pour des cas où **vitesse, portabilité et efficacité** sont prioritaires.

1.1.8. Positionnement de Docker par rapport aux clusters

- ▶ Un **cluster** désigne un ensemble de machines (physiques ou virtuelles) coopérant pour exécuter des applications de manière distribuée.
- ▶ **Avant Docker :**
 - Mise en place complexe (configuration, déploiement, équilibrage).
 - Difficulté à reproduire les environnements sur chaque nœud.
- ▶ **Docker apporte :**
 - Des **conteneurs standardisés** : portables et identiques sur chaque nœud.
 - Une base pour des outils d'orchestration (Docker Swarm, Kubernetes).
 - Une gestion simplifiée du **scalabilité horizontale** : ajouter des conteneurs plutôt que des machines complètes.
- ▶ ➡ Docker **ne remplace pas les clusters, il s'intègre et les simplifie**, en les rendant plus **dynamiques, reproductibles et modulaires**.

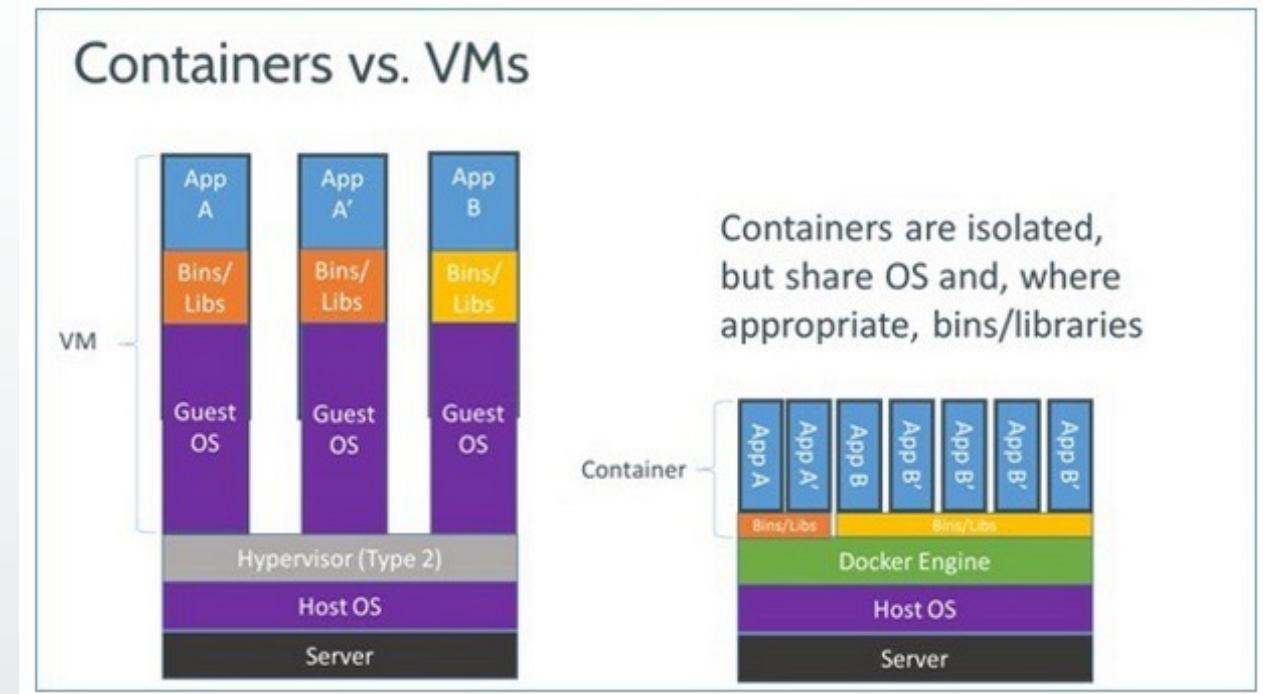
1.1.9. Positionnement de Docker par rapport aux approches de bacs-à-sable

- ▶ Les **bacs-à-sable (sandboxing)** visent à isoler un processus pour le contrôler ou limiter son impact sur le système hôte.
- ▶ **Approches classiques :**
 - Utilisées dans les navigateurs, machines virtuelles Java, environnements sécurisés.
 - Reposent souvent sur des **restrictions logicielles** (droits, quotas, filtres).
- ▶ **Ce que Docker apporte :**
 - Isolation **native et fine** via les **namespaces** Linux (processus, réseau, système de fichiers...).
 - Contrôle des ressources via **cgroups**.
 - Plus léger, plus modulaire et plus proche du système qu'un bac-à-sable classique.
- ▶ **En résumé :**

Docker va **au-delà du sandboxing** : il isole, standardise, et facilite **le déploiement**, tout en conservant des performances proches du natif.

1.2. Principe des conteneurs

- ▶ 1.2.1. Les apports de Docker
- ▶ 1.2.2. Principe des conteneurs industriels
- ▶ 1.2.3. Docker et l'approche normalisée



1.2.1. Les apports de Docker

- ▶ Docker ne se limite pas à l'usage des conteneurs, il propose une **véritable plateforme complète** autour de leur gestion.
- ▶ **Les principaux apports :**
 - **Standardisation** : les applications sont empaquetées avec toutes leurs dépendances dans un conteneur.
 - **Portabilité** : un conteneur Docker peut s'exécuter sur n'importe quelle machine disposant de Docker (Linux, Windows, cloud...).
 - **Isolation légère** : chaque conteneur fonctionne en toute indépendance, sans l'overhead d'une VM.
 - **Reproductibilité** : les environnements sont identiques d'un développeur à l'autre, d'un serveur à l'autre.
 - **Écosystème riche** : Docker Hub, CLI intuitive, API, orchestration, gestion réseau et volumes intégrée.
- ▶ ➤ Docker rend le développement, le test, le déploiement et la scalabilité **plus rapides, fiables et modulaires**.

1.2.2. Principe des conteneurs industriels

► Analogie avec le monde industriel :

- Un **conteneur industriel** (type cargo) est une **boîte standardisée**.
- Il peut transporter **n'importe quel contenu** (voiture, textile, électronique...).
- Il est **facilement manipulable** : par bateau, train, camion, partout dans le monde.
- Les infrastructures (ports, grues, entrepôts) sont **adaptées à ce standard**.

► 🐳 Transposition à l'informatique :

- Un **conteneur Docker** encapsule une application avec toutes ses dépendances.
 - Il est **standardisé, isolé, facile à déplacer** entre environnements (dev, test, prod, cloud...).
 - L'infrastructure devient **agnostique du contenu** : on exécute des conteneurs, peu importe ce qu'ils transportent.
-
- ➤ Le conteneur devient l'unité universelle de **transport logiciel**, comme le conteneur l'est pour la logistique mondiale.

1.2.3. Docker et l'approche normalisée

Docker repose sur des **standards techniques ouverts**, favorisant la compatibilité et l'interopérabilité.

▶ **Normalisation des conteneurs :**

- Format d'image standard (OCI : Open Container Initiative).
- Structure claire : base d'image, dépendances, instructions de build (Dockerfile).
- Comportement prévisible : démarrage via une commande unique (ENTRYPOINT, CMD).

▶ **Normalisation du cycle de vie :**

- **Build → Ship → Run :**
 - docker build : création d'une image.
 - docker push/pull : transport via des registres (Docker Hub, GitLab, etc.).
 - docker run : exécution du conteneur de façon identique partout.

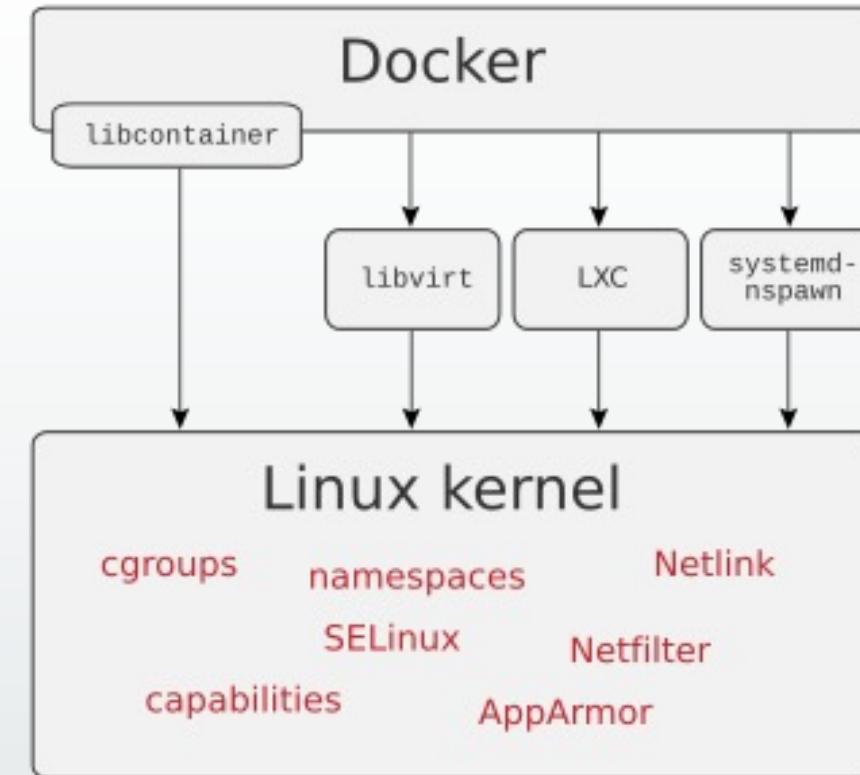
▶ **Avantages de cette normalisation :**

- Compatibilité avec les outils tiers (Kubernetes, CI/CD, monitoring...).
- Adoption massive dans l'industrie.
- Réduction des erreurs liées aux différences d'environnements.

▶ 👉 Docker impose une **discipline simple mais puissante**, au cœur des pratiques DevOps modernes.

1.3. Les fondements de Docker

- ▶ 1.3.1. Les technologies Linux clés pour Docker
- ▶ 1.3.2. Autres dépendances du système
- ▶ 1.3.3. LXC
- ▶ 1.3.4. Architectures complémentaires
- ▶ 1.3.5. Système de fichiers en couches



1.3.1. Les technologies Linux clés pour Docker

Docker s'appuie sur des **mécanismes natifs du noyau Linux** pour fournir isolation, sécurité et gestion des ressources.

▶ **Namespaces**

- Permettent d'isoler les ressources (PID, réseau, montage, utilisateurs...).
- Chaque conteneur a sa **vue restreinte du système**.

▶ **cgroups (control groups)**

- Permettent de **limiter, surveiller et hiérarchiser** l'utilisation des ressources (CPU, RAM, I/O...).
- Garantissent une **cohabitation stable** entre conteneurs.

▶ **UnionFS (OverlayFS, AUFS...)**

- Systèmes de fichiers permettant la **superposition de couches** (lecture seule + lecture/écriture).
- Utilisé pour construire des images Docker légères et modulaires.

▶ ➡ Ces briques Linux permettent à Docker de proposer des conteneurs **isolés, performants et légers**, sans recourir à une virtualisation complète.

1.3.2. Autres dépendances du système

Outre les technologies du noyau Linux, Docker s'appuie sur plusieurs **composants système complémentaires** :

▶ **Le daemon Docker (dockerd)**

- Processus principal côté serveur.
- Gère les conteneurs, images, volumes, réseaux, etc.
- Interagit avec le noyau pour créer et exécuter les conteneurs.

▶ **L'interface en ligne de commande (docker)**

- Client CLI permettant d'interagir avec le daemon via une API REST.
- Toutes les opérations (build, run, pull, exec, etc.) passent par cette interface.

▶ **L'environnement de l'hôte**

- **Système de fichiers** : stockage des images et volumes.
- **Réseau** : Docker configure des interfaces virtuelles (bridge, overlay...).
- **Sécurité** : SELinux, AppArmor, capabilities, etc., renforcent l'isolation des conteneurs.

▶  **Docker n'isole pas tout seul** : il orchestre intelligemment des composants du système d'exploitation pour garantir portabilité, performance et sécurité.

1.3.3. LXC (Linux Containers)

► Qu'est-ce que LXC ?

- **LXC (Linux Containers)** est une technologie de conteneurisation basée sur les fonctionnalités du noyau Linux (namespaces + cgroups).
- Fournit un environnement isolé proche d'un système Linux complet, **sans virtualisation matérielle**.

► Lien avec Docker :

- Docker utilisait **LXC comme backend d'exécution** dans ses premières versions.
- Aujourd'hui remplacé par **runc** (standard OCI), mais LXC a inspiré la philosophie Docker.
- Toujours utilisé dans d'autres solutions (Proxmox, LXD...).

► Différences clés :

- LXC ≈ conteneur "système" (init, services...)
- Docker ≈ conteneur "application" (léger, ciblé, orienté exécution rapide)

► ➤ LXC a été une **brique fondatrice** de la conteneurisation moderne, dont Docker a simplifié l'usage et étendu les cas d'usage.

1.3.4. Architectures complémentaires

Docker s'intègre dans un **écosystème technique plus large**, composé d'architectures complémentaires qui enrichissent ses usages.

- ▶ **OCI (Open Container Initiative)**
 - Standardise le **format des images** et le **runtime des conteneurs**.
 - Permet l'interopérabilité entre outils : Docker, Podman, containerd, CRI-O...
- ▶ **containerd**
 - Runtime léger et modulaire utilisé en interne par Docker.
 - Gère le cycle de vie des conteneurs : création, exécution, snapshot, etc.
 - Utilisé aussi dans Kubernetes.
- ▶ **runc**
 - Outil bas niveau conforme à l'OCI.
 - Exécute un conteneur à partir d'une image et de son bundle de configuration.
- ▶ **Podman**
 - Alternative à Docker, sans daemon, compatible OCI.
 - Approche "rootless" pour plus de sécurité.
- ▶  Ces architectures renforcent la **portabilité**, la **standardisation** et l'**intégration avec les orchestrateurs** comme Kubernetes.

1.3.5. Système de fichiers en couches

Docker utilise un **système de fichiers en couches (UnionFS)** pour construire et exécuter ses images de manière efficace.

► Fonctionnement :

- Une image Docker est composée de **plusieurs couches empilées**.
- Chaque couche correspond à une **instruction** du Dockerfile (FROM, RUN, COPY...).
- Les couches sont **en lecture seule**, sauf la dernière qui est **lecture/écriture** pour le conteneur.

► Avantages :

- **Réutilisation** des couches communes (gain de place, cache).
- **Partage** entre images (optimise le stockage et le réseau).
- **Construction rapide** : seules les couches modifiées sont reconstruites.

► ➡ Cette architecture en couches rend Docker **rapide, modulaire et efficace** lors des builds et des déploiements.

1.3.5. Système de fichiers en couches

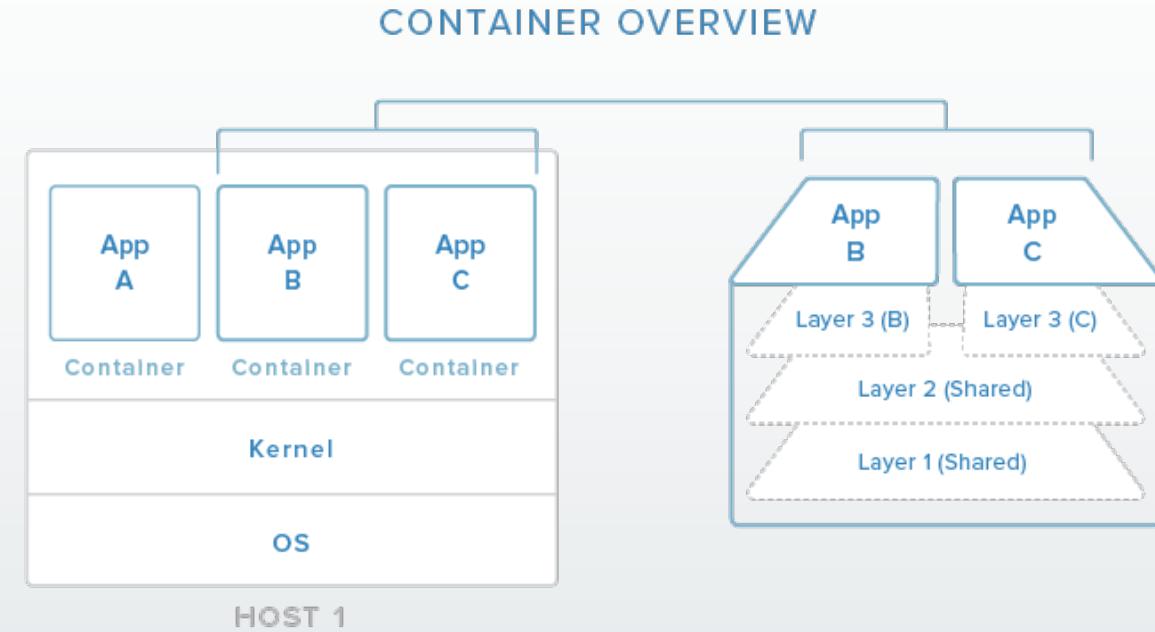
- ▶ Exemple simplifié :

```
FROM python:3.11          # Couche 1 : image de base
COPY . /app                # Couche 2 : ajout des fichiers
RUN pip install -r requirements.txt # Couche 3 : install des dépendances
CMD ["python", "app.py"]      # Couche 4 : commande de lancement
```

- ▶ 👉 Cette architecture en couches rend Docker **rapide, modulaire et efficace** lors des builds et des déploiements.

1.4. Les plus de Docker

- ▶ 1.4.1. Au-delà du cloisonnement Linux
- ▶ 1.4.2. L'approche un conteneur = un processus
- ▶ 1.4.3. L'écosystème Docker



1.4.1. Au-delà du cloisonnement Linux

Docker ne se contente pas d'utiliser les mécanismes d'isolation du noyau Linux : il construit une **expérience complète et accessible** autour de la conteneurisation.

▶ Abstraction simplifiée

- Docker masque la complexité des namespaces, cgroups, montage de volumes, etc.
- Il propose une **interface unifiée** (CLI, API) pour gérer conteneurs, images, réseaux et volumes.

▶ Portabilité native

- Une image Docker peut être **exécutée à l'identique** sur tout système supportant Docker (local, cloud, CI/CD...).
- Supprime les écarts entre environnements (dev/test/prod).

▶ Gestion centralisée

- Docker centralise les aspects réseau, stockage, sécurité et logs dans un modèle cohérent.
- Permet d'**industrialiser facilement** le déploiement applicatif.

▶👉 Docker **dépasse le cloisonnement Linux** en offrant une solution de **gestion d'applications conteneurisées** complète, standardisée et portable.

1.4.2. L'approche un conteneur = un processus

Docker adopte une philosophie simple mais puissante :

→ **chaque conteneur exécute un unique processus principal.**

► Pourquoi cette approche ?

- **Simplicité** : le conteneur démarre, vit et s'arrête avec ce processus.
- **Observabilité** : les logs et le comportement sont directement liés au processus principal.
- **Robustesse** : échec du processus = arrêt explicite du conteneur.
- **Scalabilité** : chaque composant d'une application peut être **isolé et répliqué indépendamment**.

► Exemple :

```
docker run nginx
```

- Lance un conteneur avec **un seul processus actif** : le serveur Nginx.

► 🧠 Bonne pratique :

Un conteneur = un rôle = un processus.

Pour plusieurs services → architecture multi-conteneurs avec orchestration (ex : Docker Compose).

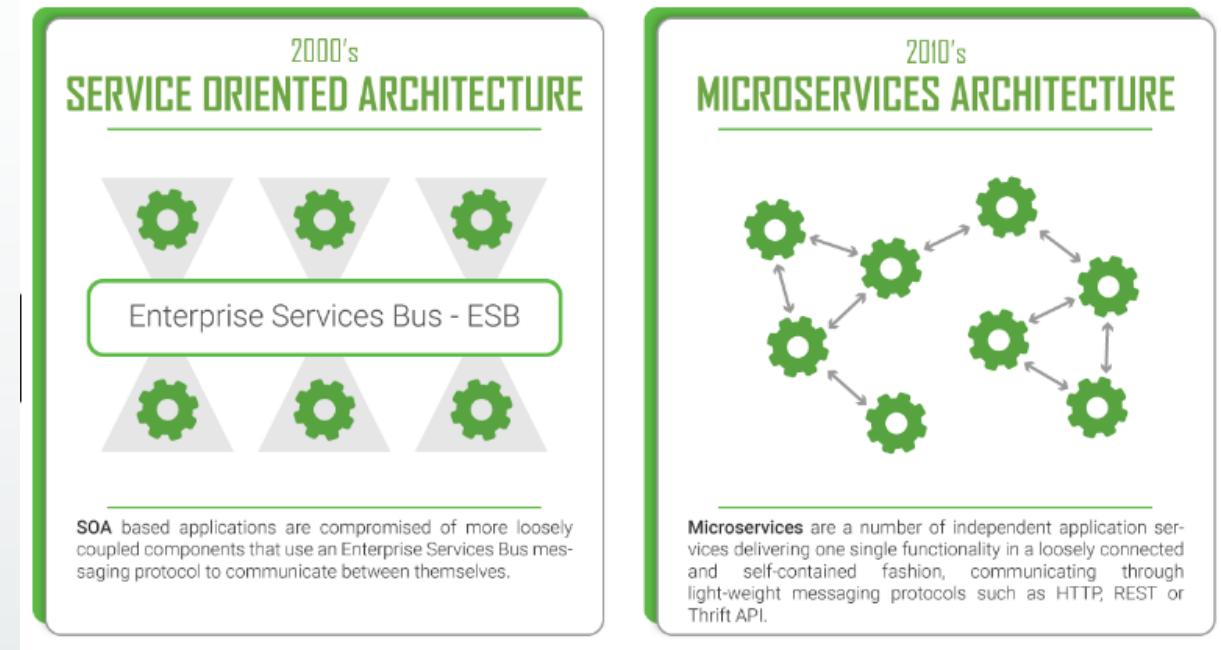
1.4.3. L'écosystème Docker

Docker ne se limite pas à la création de conteneurs :
il s'accompagne d'un **écosystème riche et intégré** pour couvrir tout le cycle de vie applicatif.

- ▶ **Docker CLI & Docker Engine**
 - Interface en ligne de commande + moteur d'exécution.
 - Permet de **construire, exécuter, gérer** des conteneurs localement.
- ▶ **Docker Hub**
 - Registre public d'**images prêtes à l'emploi** (officielles ou communautaires).
 - Permet de partager et déployer rapidement ses propres images.
- ▶ **Docker Compose**
 - Décrit des **architectures multi-conteneurs** via un fichier docker-compose.yml.
 - Pratique pour les environnements de dev ou de test.
- ▶ **Intégration CI/CD & Orchestration**
 - Compatible avec GitLab CI, GitHub Actions, Jenkins...
 - Intégration avec des orchestrateurs : **Docker Swarm, Kubernetes**.
- ▶ 👉 Docker offre bien plus qu'un outil de conteneurisation : c'est une **plateforme complète** au service de la productivité DevOps.

1.5. Architectures de services

- ▶ 1.5.1. Historique des architectures de services
- ▶ 1.5.2. Architecture de microservices
- ▶ 1.5.3. Apport de Docker
- ▶ 1.5.4. Fil conducteur



1.5.1. Historique des architectures de services

L'évolution des architectures de services a suivi les besoins croissants en **modularité, scalabilité et maintenabilité** :

- ▶ **Architecture monolithique**
 - Application unique, déployée d'un bloc.
 - **Facile à développer au départ**, mais difficile à faire évoluer.
 - Un changement peut impacter l'ensemble du système.
- ▶ **Architecture n-tiers (3-tiers classique)**
 - Séparation des **couches présentation / logique / données**.
 - Meilleure organisation, mais **fort couplage entre les services**.
- ▶ **SOA (Service-Oriented Architecture)**
 - Introduction de services réutilisables et indépendants.
 - Utilisation d'un **ESB (Enterprise Service Bus)** pour la communication.
 - **Complexité accrue**, notamment dans la gestion des flux.
- ▶ ➤ Ces évolutions ont préparé le terrain à une approche plus **souple, légère et modulaire** : les **microservices**.

1.5.2. Architecture de microservices

Les **microservices** sont une réponse moderne aux limites des architectures traditionnelles.

▶ Principe

- L'application est décomposée en **services indépendants**, chacun dédié à une **fonction métier** précise.
- Chaque service est **déployable, scalable et maintenable** indépendamment.

▶ Caractéristiques clés :

- **Découplage fort** : les services communiquent via API (REST, gRPC...).
- **Hétérogénéité** possible : langages, bases de données, technologies différentes par service.
- **Résilience** accrue : un service peut tomber sans affecter tout le système.
- Favorise **l'agilité**, le **déploiement continu**, et les **équipes autonomes**.

▶ Défis :

- Complexité de communication (réseau, supervision, sécurité...).
- Nécessite une **infrastructure adaptée** : monitoring, déploiement, orchestration.

▶ ➤ Les microservices sont au cœur des architectures modernes, et **Docker les rend concrets et gérables**.

1.5.3. Apport de Docker

Docker est un **accélérateur naturel** pour la mise en œuvre des architectures modernes, notamment les microservices.

- ▶ **Conteneurisation des services**
 - Chaque microservice peut être **isolé dans un conteneur**.
 - Facilite la **gestion indépendante** des composants (langages, dépendances, cycles de vie).
- ▶ **Portabilité et reproductibilité**
 - Les conteneurs Docker garantissent que l'application fonctionne **de la même manière partout**.
 - Déploiement identique en local, en test, en prod ou dans le cloud.
- ▶ **Scalabilité simplifiée**
 - Lancement rapide de **plusieurs instances d'un même service**.
 - Compatible avec les solutions d'orchestration (Docker Compose, Swarm, Kubernetes).
- ▶ **CI/CD optimisé**
 - Intégration fluide dans les pipelines de **build, test et déploiement continu**.
- ▶  Docker **rend les microservices concrets, rapides à développer et faciles à opérer**, même à grande échelle.

1.5.4. Fil conducteur

L'architecture microservices, facilitée par Docker, devient la **colonne vertébrale** des systèmes modernes.

▶ **Ce que Docker a permis jusqu'ici :**

- Standardiser les environnements.
- Isoler proprement les services.
- Déployer rapidement, de façon fiable et reproductible.

▶ **Ce que nous allons explorer ensuite :**

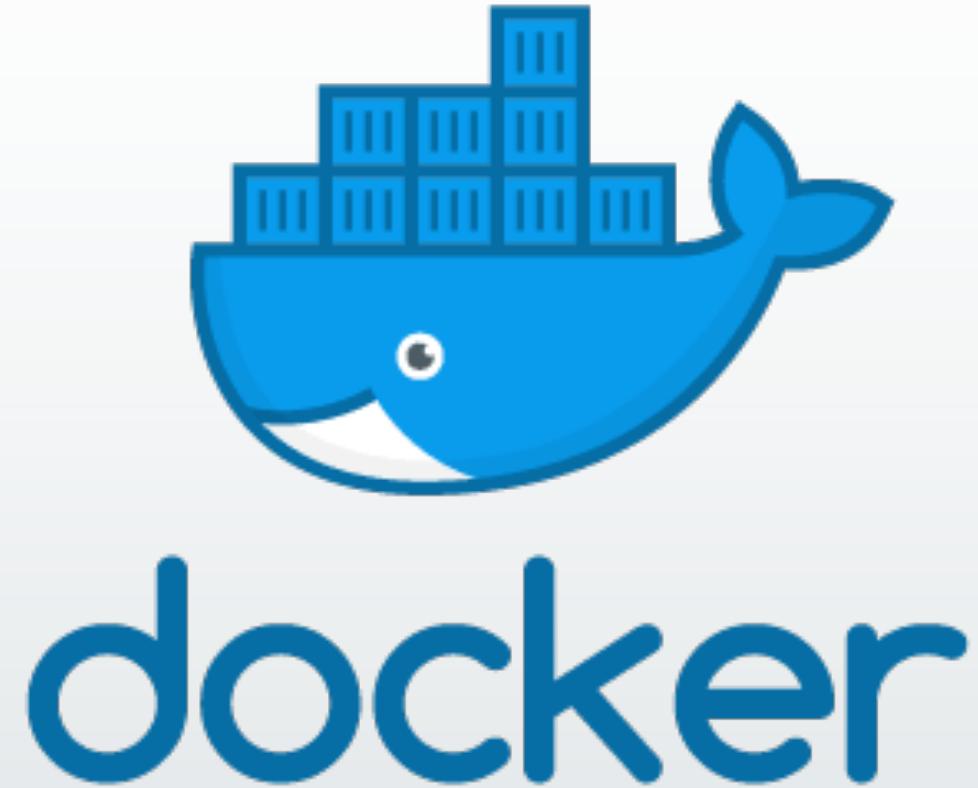
- **Créer nos propres images Docker.**
- **Gérer des conteneurs dans des environnements complexes.**
- **Composer des services interconnectés** avec Docker Compose.
- **Automatiser le déploiement à l'échelle**, en intégrant Docker dans une chaîne logicielle.

▶  Docker ne se limite pas à l'outillage : c'est une **nouvelle façon de concevoir, livrer et exploiter des applications.**

Chapitre 2

Installation

- ▶ 2.1. Editions et canaux
- ▶ 2.2. Utiliser des machines dans le cloud
- ▶ 2.3. Installation de Docker



2.1. Editions et canaux

- ▶ 2.1.1. Un standard sous linux
- ▶ 2.1.2. Les canaux

2.1.1. Un standard sous linux

Docker est **nativement conçu pour Linux**, et y est devenu une **référence incontournable** en matière de conteneurisation.

- ▶ **Intégration au cœur de l'écosystème Linux**
 - Utilise des **fonctionnalités du noyau Linux** (namespaces, cgroups, overlayfs...).
 - Compatible avec toutes les grandes distributions : **Debian, Ubuntu, CentOS, Fedora, Alpine, etc.**
- ▶ **Une adoption massive**
 - Présent dans la majorité des **distributions serveurs** modernes.
 - Intégré à de nombreux outils DevOps : GitLab CI, Jenkins, Kubernetes...
- ▶ **Support natif du CLI et des démons système**
 - Utilisation facilitée via **systemd, packages officiels et dépôts dédiés**.
- ▶👉 Sur Linux, Docker est **à la fois standard, stable et optimisé** pour un usage professionnel et industriel.

2.1.2. Les canaux

Docker propose différents **canaux de distribution** pour répondre à divers besoins : stabilité, nouveauté, support.

▶ **Stable**

- Version **testée et validée** pour la production.
- Mises à jour régulières mais prudentes.
- Recommandée pour les environnements **critiques**.

▶ **Test / Edge (ou nightly)**

- Intègre les **fonctionnalités récentes** dès leur développement.
- Idéale pour les développeurs curieux ou contributeurs.
- Moins stable, peut comporter des bugs.

▶ **Canary / Nightly (selon les cas)**

- Builds générées automatiquement à chaque changement de code.
- **Usage réservé aux tests très avancés** ou à la veille technologique.

▶ **Choisir son canal ?**

- **Stable** : pour prod, équipes DevOps, infrastructure.
- **Test** : pour formation, sandbox, PoC, contributions à Docker.

▶  Le choix du canal dépend du **niveau de maturité du projet** et du **besoin de stabilité vs innovation**.

2.2. Utiliser des machines dans cloud

- ▶ 2.2.1. Amazon AWS
- ▶ 2.2.2. Microsoft Azure
- ▶ 2.2.3. Google Cloud Platform



2.2.1. Amazon AWS

AWS permet de déployer Docker facilement grâce à plusieurs services adaptés aux conteneurs.

- ▶ **EC2 (Elastic Compute Cloud)**
 - **Machines virtuelles classiques.**
 - Permet d'installer Docker manuellement sur une VM Linux.
 - Idéal pour un **contrôle total** de l'environnement.
- ▶ **Amazon ECS (Elastic Container Service)**
 - Service managé pour exécuter des conteneurs à grande échelle.
 - Intégré avec Docker, support natif des images Docker via ECR (Elastic Container Registry).
 - **Simplifie l'orchestration** sans gérer d'infrastructure complexe.
- ▶ **AWS Fargate**
 - Exécution de conteneurs **sans gérer de serveurs** (serverless).
 - Déploiement à partir d'une image Docker, facturation à l'usage.
 - Parfait pour des workloads **élastiques et temporaires**.
- ▶  AWS offre **plusieurs niveaux d'abstraction** pour exécuter des conteneurs Docker, du plus manuel au plus automatisé.

2.2.2. Microsoft Azure

Azure propose plusieurs services pour exécuter des conteneurs Docker dans le cloud.

▶ **Azure VM (Virtual Machines)**

- Déploiement classique d'une machine Linux/Windows avec **Docker installé manuellement**.
- Adapté aux tests, prototypes ou environnements custom.

▶ **Azure Container Instances (ACI)**

- **Conteneurs Docker sans infrastructure** à gérer.
- Démarrage rapide, facturation à la seconde.
- Idéal pour des **tâches ponctuelles, tests ou microservices légers**.

▶ **Azure Kubernetes Service (AKS)**

- Plateforme d'orchestration complète basée sur Kubernetes.
- Intègre naturellement **les conteneurs Docker**.
- Convient aux architectures complexes ou aux déploiements à l'échelle.

▶ ➤ Azure fournit un **écosystème intégré** pour déployer des conteneurs Docker **de manière simple ou avancée**, selon le besoin.

2.2.3. Google Cloud Platform

Google Cloud offre un environnement cloud **nativement pensé pour les conteneurs**, avec un fort historique autour de Kubernetes.

► **Google Compute Engine (GCE)**

- Machines virtuelles classiques (Linux/Windows).
- Docker peut être installé manuellement sur une instance.
- Bon choix pour un **contrôle fin** ou une approche personnalisée.

► **Google Cloud Run**

- Exécution de conteneurs **serverless**.
- Déploiement direct à partir d'une **image Docker**.
- Scalable automatiquement, tarif basé sur l'usage réel.
- Parfait pour les **APIs, microservices, jobs ponctuels**.

► **Google Kubernetes Engine (GKE)**

- Orchestrateur Kubernetes **managé** par Google.
- Intégration native avec Docker, Helm, monitoring, etc.
- Idéal pour les **applications complexes et distribuées**.

► 👍 GCP met les conteneurs au cœur de son offre cloud, avec une prise en charge **native, performante et évolutive** de Docker.

2.3. Installation de Docker

- ▶ 2.3.1. Installation de Docker sur Linux
- ▶ 2.3.2. Le paradoxe Docker sous Windows
- ▶ 2.3.3. L'outil Docker pour Windows
- ▶ 2.3.4. Docker pour Windows Server
- ▶ 2.3.5. Utilisation de Docker avec Vagrant



2.3.1. Installation de Docker sur Linux.

Docker peut être installé rapidement sur la majorité des distributions Linux via les dépôts officiels.

▶ **Étapes principales (ex. Ubuntu / Debian) :**

```
curl -sSL https://get.docker.com/ | sh
```

▶ **Vérification :**

```
docker --version
```

▶ **Remarques :**

- Nécessite souvent d'ajouter l'utilisateur au groupe docker :

```
sudo usermod -aG docker $USER
```

- Redémarrage de session nécessaire après cette commande.

▶  Docker s'installe en quelques minutes, avec un support natif sur Linux et une documentation riche.

2.3.2. Le paradoxe Docker sous Windows

Docker repose sur des **fonctionnalités du noyau Linux**, absentes de Windows nativement.
Cela crée un paradoxe :

👉 "Comment faire tourner Docker sur un système qui ne comprend pas les conteneurs Linux ?"

▶ La réalité :

- Docker **ne peut pas s'exécuter directement** sur Windows (hors Windows Server avec conteneurs Windows).
- Une couche de compatibilité est nécessaire pour exécuter des **conteneurs Linux**.

▶ Solution historique :

- Utilisation d'une **VM Linux embarquée** (via Hyper-V ou WSL2).
- Le **client Docker (Windows)** communique avec le **daemon dans la VM**.

▶ Conséquence :

- Docker sur Windows ≠ Docker “natif”.
- L'expérience dépend fortement de la **configuration système** (Hyper-V, WSL2, Windows Edition...).

▶👉 Ce paradoxe est **gommé par les outils modernes**, mais reste une notion clé pour comprendre l'architecture Docker sur Windows.

2.3.3. L'outil Docker pour Windows

- ▶ **Docker Desktop for Windows** : C'est l'outil **officiel** pour exécuter Docker sur Windows de manière fluide et intégrée.
- ▶ **Fonctionnement :**
 - Docker Desktop utilise **WSL2** (ou **Hyper-V**) pour faire tourner une **VM Linux légère**.
 - Fournit à l'utilisateur une **expérience unifiée** : terminal, intégration Docker CLI, gestion graphique des conteneurs.
- ▶ **Fonctionnalités incluses :**
 - Docker Engine, Docker CLI, Docker Compose.
 - Intégration avec Kubernetes (optionnel).
 - Interface de gestion visuelle des conteneurs, images, volumes...
- ▶ **Prérequis :**
 - Windows 10/11 **Pro ou Entreprise** (ou Home avec WSL2 activé).
 - Sous Windows Home, **WSL2 est requis** (avec noyau Linux).
- ▶  Docker Desktop **masque la complexité** du paradoxe Windows/Linux et offre une **expérience fluide et professionnelle** aux développeurs.

2.3.4. Docker pour Windows Server

Contrairement à Windows "Desktop", Windows Server propose une **prise en charge native de Docker**, mais avec des spécificités.

▶ Conteneurs Windows

- Docker peut exécuter des **conteneurs Windows** basés sur des images comme mcr.microsoft.com/windows/servercore.
- Ces conteneurs utilisent les **API Windows**, pas Linux.

▶ Fonctionnement :

- Pas de VM intermédiaire : Docker s'appuie directement sur le système hôte.
- Deux types d'isolation possibles :
 - **Process isolation** (plus léger, mais dépend de la compatibilité du noyau).
 - **Hyper-V isolation** (plus isolé, via mini-VM).

▶ Limitations :

- **Incompatibilité avec les images Linux** : pas de conteneur Linux sans VM.
- Écosystème d'images plus restreint que sur Linux.

▶ Cas d'usage typiques :

- Conteneurisation d'**applications .NET Framework**, services Windows, ou outils internes d'entreprise.

▶ 👍 Docker sur Windows Server vise les **workloads Windows historiques**, avec une intégration adaptée à l'environnement entreprise.

2.3.5. Utilisation de Docker avec Vagrant

Vagrant est un outil d'automatisation permettant de **provisionner des environnements reproductibles**, souvent via des machines virtuelles.

► **Objectif :**

- Créer une **machine de développement prête à l'emploi**, avec Docker déjà installé.
- Idéal pour **standardiser les environnements entre développeurs** ou en formation.

► **Avantages :**

- Automatisation du setup (OS, Docker, configurations...).
- Portabilité garantie : le même Vagrantfile fonctionne sur toutes les machines hôtes.
- Support multi-plateformes (Windows, macOS, Linux).

► **Exemple minimal de Vagrantfile :**

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/noble64"
  config.vm.provision "docker"
end
```

- Lance automatiquement une VM Ubuntu avec Docker installé.
- 🤝 Vagrant permet de **simplifier le démarrage d'environnements Docker** sur n'importe quel poste, sans effort de configuration manuel.

Chapitre 3

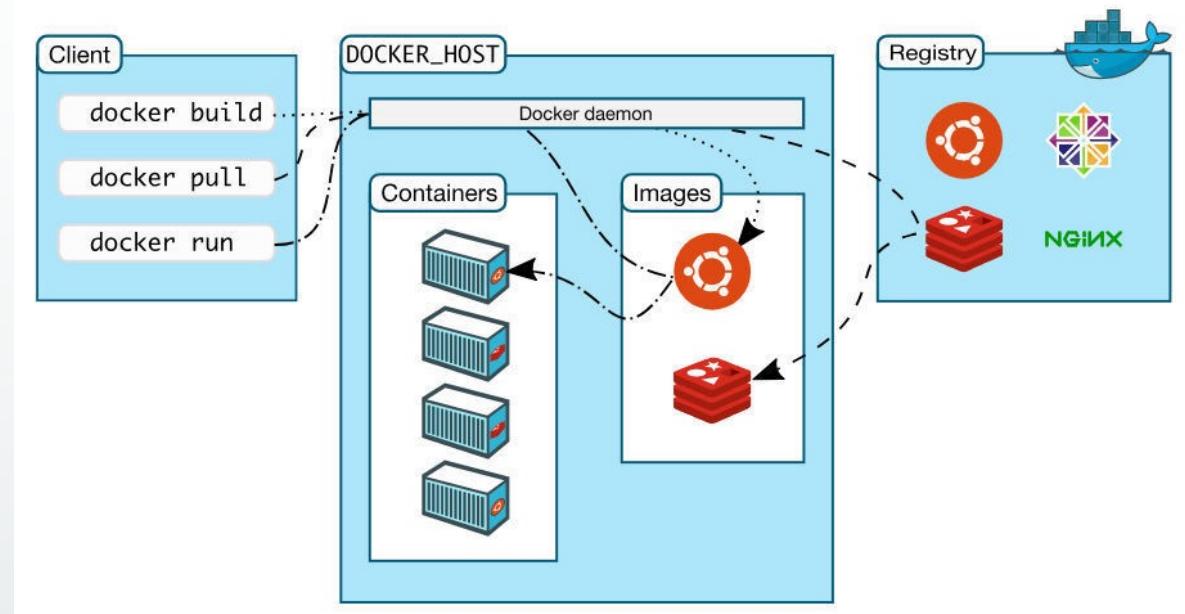
Premiers pas avec Docker

- ▶ 3.1. Hello World, Docker
- ▶ 3.2. Utiliser des images Docker préexistantes
- ▶ 3.3. Un second conteneur
- ▶ 3.4. Retour sur les premiers pas



3.1. Hello Word, Docker

- ▶ 3.1.1. Démarrage d'un conteneur simple
- ▶ 3.1.2. Détails des opérations effectuées



3.1.1. Démarrage d'un conteneur simple

- ▶ La commande la plus simple pour exécuter un conteneur Docker est :

```
docker run hello-world
```

- ▶ Que fait cette commande ?

- Télécharge l'image **hello-world** depuis Docker Hub (si elle n'est pas déjà en local).
- Crée un conteneur basé sur cette image.
- Exécute un script dans le conteneur qui affiche un message de bienvenue.
- Arrête le conteneur une fois le message affiché (car c'est un processus éphémère).

- ▶ Résultat attendu :

```
Hello from Docker!  
This message shows that your installation appears to be working correctly.  
...
```

- ▶ 👏 Ce test est une **vérification rapide** que Docker est installé, configuré et que les images peuvent être récupérées et exécutées.

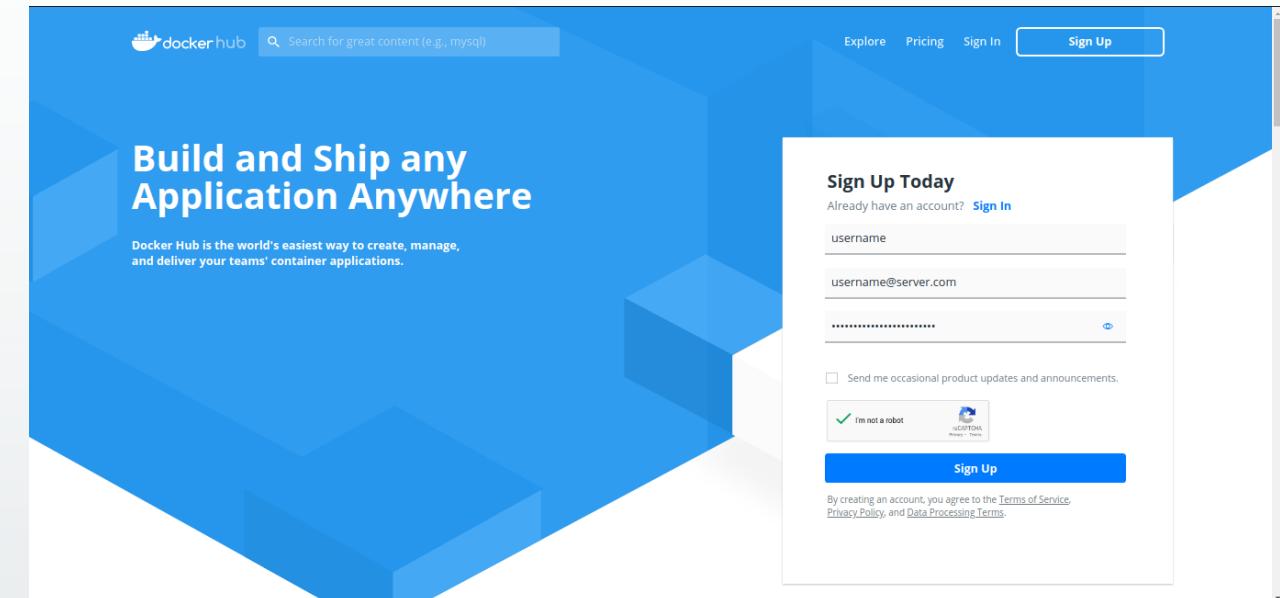
3.1.2. Détails des opérations effectuées

Lors de l'exécution de docker run hello-world, plusieurs étapes sont automatiquement réalisées :

- ▶ **1. Vérification locale**
 - Docker vérifie si l'image hello-world est **présente en local**.
- ▶ **2. Téléchargement (si nécessaire)**
 - Si l'image n'existe pas localement, Docker la **télécharge depuis Docker Hub**, le registre public.
- ▶ **3. Création du conteneur**
 - Docker **crée un conteneur** à partir de l'image (une instance de l'image).
- ▶ **4. Exécution du conteneur**
 - Le **processus principal de l'image est lancé** (dans ce cas : affichage d'un message).
- ▶ **5. Arrêt automatique**
 - Une fois le message affiché, le conteneur **s'arrête**, car le processus principal est terminé.
- ▶ **6. Possibilité de relancer**
 - Le conteneur peut être relancé, ou supprimé à l'aide de commandes comme docker start, docker rm.
- ▶  Cette commande illustre la **philosophie Docker** : tout est automatisé, du téléchargement à l'exécution, pour une **expérience fluide et rapide**.

3.2. Utiliser des images Docker préexistantes

- ▶ 3.2.1. Le registre Docker Hub
- ▶ 3.2.2. Gestion du compte Docker Hub et dépôts privés



3.2.1. Le registre Docker Hub

Docker Hub est le **registre officiel** d'images Docker, accessible publiquement à l'adresse :
👉 <https://hub.docker.com>

▶ **Rôle du registre :**

- **Stocke et distribue** les images Docker.
- Permet de **rechercher, télécharger** et **publier** des images.
- Utilisé automatiquement par la commande docker run si aucun registre n'est précisé.

▶ **Types d'images :**

- **Images officielles** : maintenues par Docker ou des éditeurs reconnus (ex : nginx, mysql, python).
- **Images communautaires** : proposées par les utilisateurs Docker.
- **Tags** pour spécifier une version (ubuntu:20.04, node:18-alpine, etc.).

▶ **Exemple d'utilisation :**

```
docker pull nginx      # Télécharge la dernière version de l'image nginx
docker run nginx       # Lance un conteneur basé sur cette image
```

▶👉 Docker Hub est **la principale source d'images Docker**, et constitue un **réflexe de base** pour tout utilisateur.

3.2.2. Gestion du compte Docker Hub et dépôts privés

Docker Hub permet de **gérer ses propres images Docker** via un compte utilisateur ou une organisation.

► **Création de compte :**

- Inscription gratuite sur hub.docker.com.
- Permet de **pousser ses images**, gérer ses dépôts, et accéder aux **dépôts privés**.

► **Dépôts publics vs privés :**

- **Publics** : accessibles à tous, idéal pour les projets open-source ou les partages simples.
- **Privés** : restreints à un ou plusieurs utilisateurs, **gratuits en nombre limité**, utiles pour des projets confidentiels.

3.2.2. Gestion du compte Docker Hub et dépôts privés

► Commandes utiles :

```
docker login          # Authentification avec ses identifiants Docker Hub
docker push moi/monimage:1.0 # Envoi d'une image vers son dépôt
docker pull moi/monimage:1.0 # Récupération d'une image privée (si autorisé)
```

► Organisations et équipes :

- Gestion fine des accès (lecture/écriture).
 - Pratique en entreprise pour **collaborer sur des projets conteneurisés**.
-
- ➤ Le compte Docker Hub est **indispensable** pour gérer et partager ses propres images dans un cadre professionnel ou collaboratif.

3.3. Un second conteneur

- ▶ 3.3.1. Récupération de l'image
- ▶ 3.3.2. Explication des tags
- ▶ 3.3.3. Premier lancement
- ▶ 3.3.4. Lancement en mode interactif
- ▶ 3.3.5. Persistance des modifications sous forme d'une image
- ▶ 3.3.6. Prise en main du client Docker
- ▶ 3.3.7. Manipulation des conteneurs



3.3.1. Récupération de l'image

Avant d'exécuter un conteneur, il faut **récupérer l'image** correspondante depuis un registre (comme Docker Hub).

▶ **Commande de base :**

```
docker pull ubuntu
```

- → Télécharge la dernière version disponible de l'image ubuntu.

▶ **Précision d'une version (tag) :**

```
docker pull ubuntu:24.04
```

- → Permet de spécifier **la version exacte** du système de base.

3.3.1. Récupération de l'image

▶ **Vérification de la récupération :**

```
docker images
```

- → Affiche toutes les images disponibles localement, avec leur **nom, tag, ID et taille**.

▶ **Bon à savoir :**

- docker run exécute automatiquement un pull si l'image n'existe pas localement.
 - Les images sont **stockées en local**, et **réutilisables** sans re-téléchargement.
- ▶ ➡ docker pull est la **première étape manuelle** pour maîtriser le cycle de vie des conteneurs.

3.3.2. Explication des tags

Un **tag** dans Docker désigne une **version spécifique** d'une image. Il permet de **maîtriser précisément** le comportement et l'environnement d'un conteneur.

► Syntaxe :

```
<nom-de-l'image>:<tag>
ubuntu:22.04
node:23.10-alpine3.20
mysql:8.1
```

► Sans tag ?

- Par défaut, Docker utilise le tag latest :

```
docker pull ubuntu           # équivalent à docker pull ubuntu:latest
```

- Attention : latest ne garantit **pas la version la plus récente**, mais celle définie comme "par défaut" par le mainteneur de l'image.

► Pourquoi utiliser des tags ?

- Reproductibilité : même image = même comportement.
- Maîtrise des mises à jour.
- Comparaison ou test de plusieurs versions côté à côté.

► ➤ Utiliser un **tag explicite** est une **bonne pratique** pour garantir stabilité et cohérence des environnements.

3.3.3. Premier lancement

Une fois l'image récupérée, on peut **lancer un conteneur** pour exécuter un processus.

► **Commande de base :**

```
docker run ubuntu
```

- Lance un conteneur basé sur l'image ubuntu.
- Sans commande précisée, Docker utilise le **CMD par défaut** de l'image.
- Ici, l'image se termine immédiatement (pas de processus actif en arrière-plan).

3.3.3. Premier lancement

▶ Exemple avec une commande :

```
docker run ubuntu echo "Bonjour Docker"
```

- Exécute echo "Bonjour Docker" dans un conteneur Ubuntu.
- Affiche le message, puis le conteneur s'arrête.

▶ Astuce : affichage du conteneur juste après exécution

```
docker ps -a
```

- → Liste les conteneurs, y compris ceux terminés.
- ▶ ➡ docker run est **la porte d'entrée** pour tester rapidement une image dans un environnement isolé.

3.3.4. Lancement en mode interactif

Docker permet de **prendre la main à l'intérieur d'un conteneur** grâce au mode interactif.

▶ **Commande :**

```
docker run -it ubuntu bash
```

▶ **Détails des options :**

- -i : mode **interactif** (stdin reste ouvert).
- -t : allocation d'un **pseudo-terminal** (interface en ligne de commande).
- ubuntu : image de base.
- bash : processus lancé dans le conteneur (shell ici).

3.3.4. Lancement en mode interactif

► Résultat :

- Un terminal interactif s'ouvre dans le conteneur Ubuntu.
- Vous êtes dans un **environnement isolé**, mais fonctionnel :

```
root@<container_id>:/#
```

► Sortie du conteneur :

```
root@<container_id>:/ exit
```

- → Le conteneur s'arrête une fois le shell fermé (sauf si daemonisé).
- ➡ Le mode interactif est idéal pour **explorer, tester ou déboguer** dans un environnement propre.

3.3.5. Persistance des modifications sous forme d'une image

Après avoir modifié un conteneur en mode interactif, on peut **en créer une image personnalisée**.

► **Étapes :**

1. Lancer un conteneur et modifier son contenu :

```
docker run -it ubuntu bash
# apt update && apt install -y curl
# exit
```

2. Lister les conteneurs arrêtés :

```
docker ps -a
```

3.3.5. Persistance des modifications sous forme d'une image

3. Créer une image à partir du conteneur :

```
docker commit <ID_du_conteneur> monimage:custom
```

4. Utiliser la nouvelle image :

```
docker run -it monimage:custom
```

► Bonnes pratiques :

- Utiliser docker commit uniquement pour des cas simples ou exploratoires.
 - Préférer l'utilisation d'un Dockerfile pour des images reproductibles (vu plus tard).
- ➤ docker commit permet de **capturer l'état d'un conteneur** et de le transformer en image réutilisable.

3.3.6. Prise en main du client Docker

Le **client Docker (CLI)** permet de piloter toutes les actions : images, conteneurs, volumes, réseaux, etc.

▶ **Commandes fondamentales :**

-  Télécharger une image :

```
docker pull <image>
```

-  Lancer un conteneur :

```
docker run [options] <image> [commande]
```

-  Lister les conteneurs actifs :

```
docker ps
```

3.3.6. Prise en main du client Docker

- 📁 Lister tous les conteneurs (y compris arrêtés) :

```
docker ps -a
```

- 💥 Supprimer un conteneur :

```
docker rm <id-du-container>
```

- 🗑️ Supprimer une image :

```
docker rmi <image>
```

3.3.6. Prise en main du client Docker

- 🧹 Nettoyage global :

```
docker system prune
```

- ▶ ➡ Le client Docker est **simple, puissant et cohérent**, utilisable en local comme en automatisation (scripts, CI/CD...).

3.3.7. Manipulation des conteneurs

Docker permet de **gérer facilement le cycle de vie des conteneurs** : démarrage, arrêt, reprise, suppression...

- **Démarrer un conteneur arrêté :**

```
docker start <id>
```

- **Arrêter un conteneur en cours d'exécution :**

```
docker stop <id>
```

- **Supprimer un conteneur :**

```
docker rm <id>
```

3.3.7. Manipulation des conteneurs

- Exécuter une commande dans un conteneur **actif** :

```
docker exec -it <id> <commande>
```

Exemple :

```
docker exec -it monconteneur bash
```

- Afficher les logs d'un conteneur :

```
docker logs <id>
```

3.3.7. Manipulation des conteneurs

- Renommer un conteneur :

```
docker rename ancien_nom nouveau_nom
```

- ▶ ➡ Ces commandes permettent de **maîtriser les conteneurs au quotidien**, en mode développement ou production.

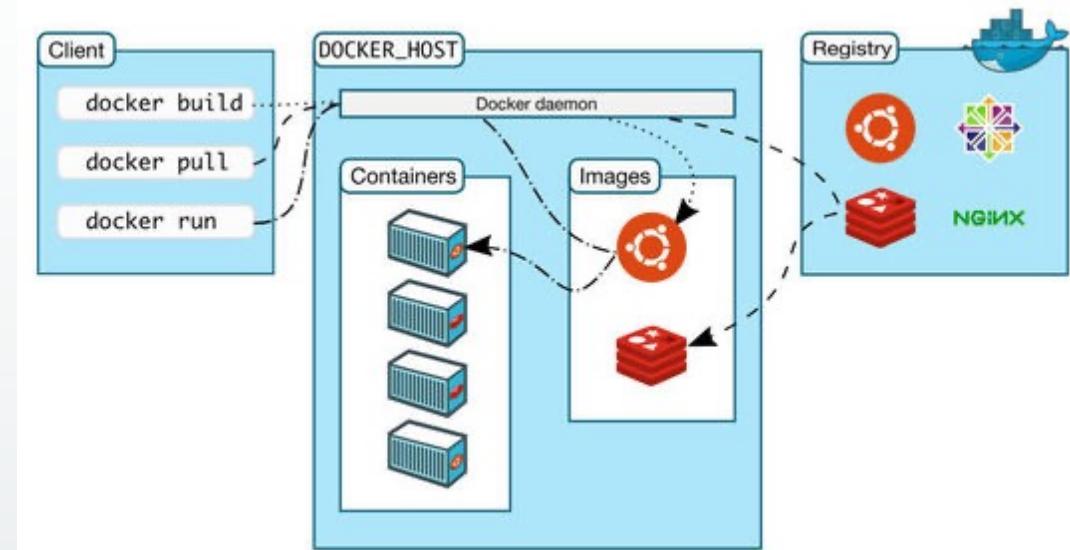
3.4. Retour sur les premiers pas

- ▶  **Ce que nous avons vu jusqu'ici :**
 -  **Téléchargement d'images** depuis Docker Hub (pull, run)
 -  **Exécution de conteneurs** simples et interactifs
 -  **Utilisation des tags** pour cibler des versions précises
 -  **Modifications dans un conteneur** et sauvegarde via commit
 -  **Commandes de base** du client Docker (ps, rm, exec, etc.)
 -  **Cycle de vie des conteneurs** : démarrage, arrêt, suppression
- ▶  **Ce que l'on retient :**
 - Un conteneur = un processus **isolé, jetable, réproductible**
 - Docker permet de **tester rapidement**, sans installer sur le système hôte
 - La base est **facile à prendre en main**, même pour un débutant
- ▶  Ces premières manipulations posent les **fondations concrètes** pour aller plus loin : **création d'images personnalisées, orchestration, automatisation...**

Chapitre 4

Création et gestion d'images Docker

- ▶ 4.1. Création manuelle d'une nouvelle image
- ▶ 4.2. Utilisation d'un fichier Dockerfile
- ▶ 4.3. Partage et réutilisation simple des images
- ▶ 4.4. Bonnes pratiques
- ▶ 4.5. Mise en œuvre d'un registre privé
- ▶ 4.6. Incorporation dans le cycle de développement



4.1. Cration manuelle d'une nouvelle image

- ▶ 4.1.1. Objectif
- ▶ 4.1.2. Approche
- ▶ 4.1.3. Difficultes
- ▶ 4.1.4. Conclusion

4.1.1. Objectif

Avant d'automatiser la création d'images Docker via un Dockerfile, il est essentiel de comprendre **comment une image peut être créée manuellement**.

▶ **Objectif de cette étape :**

- Apprendre à **construire une image étape par étape**, sans script ni fichier automatisé.
- Maîtriser le fonctionnement de :
 - La **modification d'un conteneur en direct**.
 - La **persistence** de ces modifications sous forme d'image.

▶ **Pourquoi c'est utile :**

- Compréhension fine du fonctionnement de Docker.
- Permet de **tester et valider rapidement** un environnement avant de le formaliser.
- Base pour des usages ponctuels, des corrections rapides ou du prototypage.

▶ ➡ Cette méthode manuelle est un **préalable pédagogique** avant d'aborder l'approche professionnelle via Dockerfile.

4.1.2. Approche

- ▶ **Étapes pour créer manuellement une image Docker :**

- 1. Lancer un conteneur interactif** à partir d'une image de base :

```
docker run -it ubuntu bash
```

- 2. Effectuer des modifications dans le conteneur :**

```
apt update && apt install -y curl
```

- 3. Quitter proprement le conteneur :**

```
exit
```

4.1.2. Approche

4. Lister les conteneurs récemment utilisés :

```
docker ps -a
```

5. Créer une image à partir du conteneur modifié :

```
docker commit <ID_conteneur> monimage:custom
```

6. Tester l'image créée :

```
docker run -it monimage:custom
```

4.1.2. Approche

- ▶ **Résultat :**
 - L'image monimage:custom contient toutes les modifications apportées manuellement.
 - Elle peut être utilisée comme base pour d'autres conteneurs.

- ▶  Cette méthode permet de **capturer un état précis** d'un environnement et de le **réutiliser immédiatement**.

4.1.3. Difficultés

Créer une image manuellement via docker commit est **possible**, mais présente plusieurs **limitations importantes** :

- ▶ **Non-reproductibilité**
 - Impossible de **retracer précisément** les actions effectuées dans le conteneur.
 - Difficile à **rejouer à l'identique** sur une autre machine.
- ▶ **Absence de versionning**
 - Aucun historique des modifications.
 - Pas de contrôle fin sur les évolutions d'image.
- ▶ **Risque d'oubli ou d'erreur**
 - Une étape oubliée = image incomplète ou instable.
 - Aucune validation automatique possible.
- ▶ **Inadaptée à la collaboration**
 - Impossible de **partager un processus clair** de construction d'image avec une équipe.
 - Mauvaise intégration dans les pipelines CI/CD.
- ▶  Cette approche est utile pour des **tests ponctuels**, mais elle est **peu fiable pour un usage professionnel** ou en équipe.

4.1.4. Conclusion

► Ce que nous avons vu :

- Il est possible de **créer une image Docker manuellement** à partir d'un conteneur modifié.
- Cette méthode repose sur la commande :

```
docker commit <conteneur> <nouvelle_image>
```

- Elle permet de **capturer un état fonctionnel rapidement**.

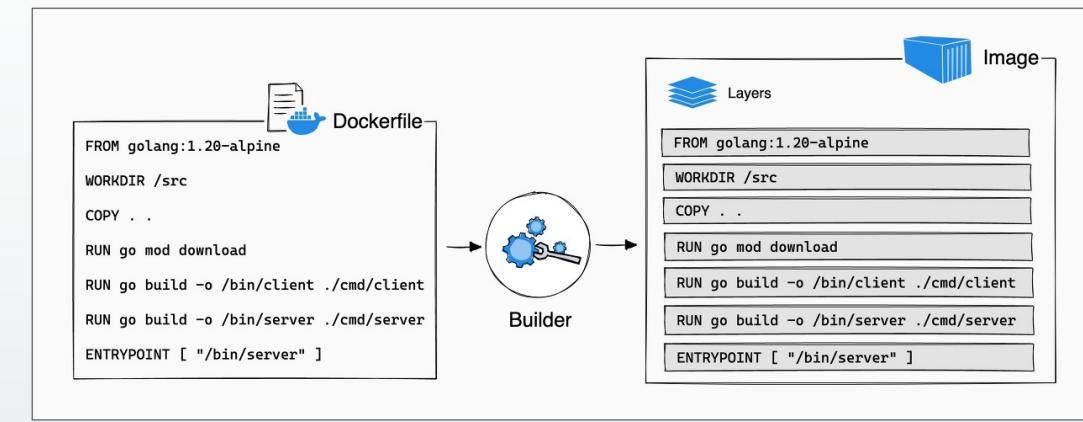
► Mais elle présente des limites :

- Peu de **tracabilité**, pas de **reproductibilité**.
- Méthode **non maintenable** à grande échelle.
- Inadaptée à une **collaboration ou une automatisation**.

► ➤ C'est une **étape pédagogique essentielle**, mais dans un contexte professionnel, on privilégiera l'approche **automatisée via Dockerfile**.

4.2. Utilisation d'un fichier Dockerfile

- ▶ 4.2.1. Intérêt des fichiers Dockerfile
- ▶ 4.2.2. Utilisation d'un fichier Dockerfile
- ▶ 4.2.3. Anatomie d'un fichier Dockerfile
- ▶ 4.2.4. Notre premier fichier Dockerfile
- ▶ 4.2.5. Commandes additionnelles



4.2.1. Intérêt des fichiers Dockerfile

Un Dockerfile est un **fichier texte** qui décrit étape par étape **comment construire une image Docker**.

► Pourquoi utiliser un Dockerfile ?

- **✓ Reproductibilité**
 - Même Dockerfile = même image, peu importe la machine.
- **✓ Automatisation**
 - Intégrable dans des scripts, pipelines CI/CD, déploiements cloud.
- **✓ Traçabilité**
 - Historique clair des instructions d'installation, configuration, exposition de ports, etc.
- **✓ Partage facile**
 - Versionnable via Git, partageable entre développeurs ou équipes.
- **✓ Maintenance simplifiée**
 - Facile de modifier une instruction sans tout reconstruire manuellement.

►  Le Dockerfile est un **standard incontournable** pour créer des images Docker de façon **claire, maintenable et professionnelle**.

4.2.2. Utilisation d'un fichier Dockerfile

Un Dockerfile permet de **définir une image personnalisée** que Docker peut construire automatiquement.

► **Étapes d'utilisation :**

1. **Créer un fichier nommé Dockerfile**
2. **Écrire les instructions** (ex : FROM, RUN, CMD...).
3. **Construire l'image** à partir du Dockerfile :

```
docker build -t monimage:1.0 .
```

- `-t` : nom + tag de l'image.
- `.` : contexte de build (dossier courant).

4.2.2. Utilisation d'un fichier Dockerfile

4. Exécuter un conteneur basé sur cette image :

```
docker run monimage:1.0
```

- ▶ **Organisation :**
 - Le Dockerfile est placé à la **racine du projet**.
 - Il peut être accompagné d'un `.dockerignore` pour exclure des fichiers inutiles.
- ▶  Cette approche permet de **formaliser la construction d'images**, et s'intègre facilement dans une chaîne de développement moderne.

4.2.3. Anatomie d'un fichier Dockerfile

Un Dockerfile est composé d'**instructions successives** qui décrivent comment construire une image.

► **Principales instructions :**

- **FROM** : définit l'image de base (*obligatoire*)

```
FROM ubuntu:20.04
```

- **RUN** : exécute des commandes lors de la construction de l'image

```
RUN apt update && apt install -y curl
```

- **COPY** ou **ADD** : copie des fichiers vers l'image

```
COPY . /app
```

4.2.3. Anatomie d'un fichier Dockerfile

- **WORKDIR** : définit le répertoire de travail dans l'image (le crée s'il n'existe pas)

```
WORKDIR /app
```

- **CMD** : commande exécutée par défaut au lancement du conteneur

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

- ▶ **Autres instructions utiles :**
 - ENV, EXPOSE, ENTRYPOINT, ARG, etc.
- ▶  L'ordre des instructions est **important** : chaque ligne crée une **nouvelle couche** dans l'image.

4.2.4. Notre premier fichier Dockerfile

Voici un exemple minimal pour créer une image Docker qui exécute un script Python.

► **Exemple de Dockerfile :**

```
FROM python:3.11-slim

WORKDIR /app

COPY app.py .

CMD ["python", "app.py"]
```

► **Structure du projet :**

```
mon-projet/
└── Dockerfile
└── app.py
```

4.2.4. Notre premier fichier Dockerfile

► **Commandes associées :**

- **Build de l'image :**

```
docker build -t monapp:1.0 .
```

- **Exécution du conteneur :**

```
docker run monapp:1.0
```

- ➤ Ce fichier suffit à créer une image **reproductible et portable**, prête à être partagée ou déployée.

4.2.5. Commandes additionnelles

En plus des instructions de base, Dockerfile propose des **commandes avancées** pour affiner le comportement de l'image.

- ▶ **ENV** – Définir des variables d'environnement

```
ENV APP_ENV=production
```

- ▶ **EXPOSE** – Documenter le port utilisé (non obligatoire pour l'ouverture réelle)

```
EXPOSE 8080
```

- ▶ **ENTRYPOINT** – Spécifie le binaire principal du conteneur

```
ENTRYPOINT ["python"]
CMD ["app.py"] # devient un argument par défaut
```

4.2.5. Commandes additionnelles

- ▶ **ARG** – Paramètres passés au moment du build (et non à l'exécution)

```
ARG VERSION
RUN echo "Building version $VERSION"
```

- Usage :

```
docker build --build-arg VERSION=1.2 .
```

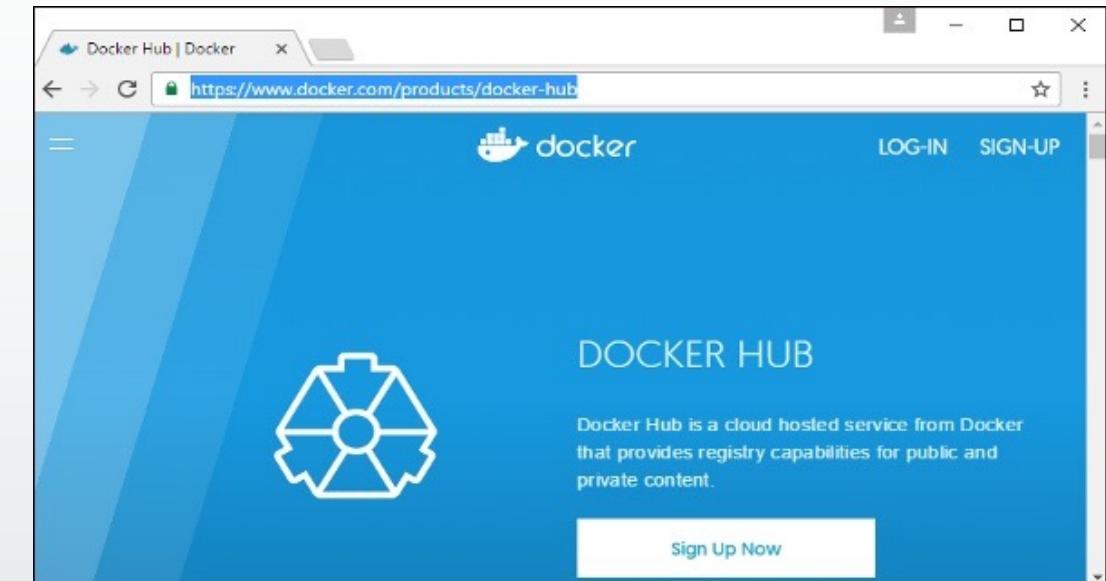
- ▶ **.dockerignore** – Exclure des fichiers du contexte de build

```
node_modules
*.log
.git
```

- ▶👉 Ces instructions permettent de **personnaliser finement** le comportement du conteneur et de **gagner en efficacité** lors du build.

4.3. Partage et réutilisation simple des images

- ▶ 4.3.1. Envoi sur votre compte Docker Hub
- ▶ 4.3.2. Export et import sous forme de fichiers



4.3.1. Envoi sur votre compte Docker Hub

Docker Hub permet de **stocker et partager vos images Docker** en ligne.

► **Prérequis :**

- Avoir un compte sur <https://hub.docker.com>
- Être connecté en local :

```
docker login
```

► **Taguer l'image pour votre dépôt :**

```
docker tag monapp:1.0 monuser/monapp:1.0
```

- Format attendu : utilisateur/nom_image:tag

4.3.1. Envoi sur votre compte Docker Hub

- ▶ Envoyer l'image sur Docker Hub :

```
docker push monuser/monapp:1.0
```

- ▶ Télécharger depuis un autre poste :

```
docker pull monuser/monapp:1.0
```

- ▶ ➤ Le **push/pull** d'images est la méthode la plus simple pour **partager des images entre développeurs, machines ou environnements.**

4.3.2. Export et import sous forme de fichiers

Docker permet de **sauvegarder et transférer une image** sans passer par un registre distant.

▶ **Export d'une image au format .tar :**

```
docker save monimage:1.0 -o monimage.tar
```

- Crée un fichier archivé contenant l'image complète.

▶ **Transfert :**

- Le fichier .tar peut être **copié** sur un autre poste (clé USB, SSH, réseau...).

▶ **Import sur une autre machine :**

```
docker load -i monimage.tar
```

4.3.2. Export et import sous forme de fichiers

► Vérification :

```
docker images
```

→ L'image est maintenant disponible localement.

► Avantages :

- Fonctionne **hors ligne**, sans dépendance à Docker Hub.
 - Pratique pour des **environnements isolés** ou du **déploiement interne sécurisé**.
- ➤ *docker save / docker load* sont les outils idéaux pour une **portabilité physique des images Docker**.

4.4. Bonnes pratiques

- ▶ 4.4.1. Principe du cache local d'images
- ▶ 4.4.2. Principe du cache à la compilation
- ▶ 4.4.3. Conséquences sur l'écriture des fichiers Dockerfile
- ▶ 4.4.4. Ajout d'une image de cache intermédiaire
- ▶ 4.4.5. Mise en œuvre d'un cache de paquetages
- ▶ 4.4.6. Conséquences sur le choix des images de base
- ▶ 4.4.7. Arborescence recommandée
- ▶ 4.4.8. La question du processus unique



4.4.1. Principe du cache local d'images

Docker conserve localement les **images déjà téléchargées ou construites**, afin d'optimiser les performances.

▶ **Fonctionnement du cache local :**

- Toute image téléchargée (docker pull) ou créée (docker build) est **stockée localement**.
- Lors d'un nouveau build ou run, Docker vérifie si l'image est déjà disponible en cache.
- **Évite les téléchargements/redéploiements inutiles**, sauf si forcé ou si l'image a changé.

▶ **Commandes utiles :**

- Voir les images disponibles :

```
docker images
```

- Supprimer une image manuellement :

```
docker rmi <image>
```

4.4.1. Principe du cache local d'images

- Nettoyer les images inutilisées :

```
docker image prune
```

- ▶ Avantage :
 - Gain de **temps** et de **bande passante**, surtout en développement.
- ▶ ➡ Le cache local est un **mécanisme central** dans l'efficacité de Docker au quotidien.

4.4.2. Principe du cache à la compilation

Lors de l'exécution d'un docker build, Docker **réutilise les couches d'image précédemment construites**, si elles n'ont pas changé.

► **Fonctionnement :**

- Chaque **instruction** d'un Dockerfile (RUN, COPY, ENV, etc.) crée une **couche (layer)**.
- Si l'instruction **n'a pas changé**, Docker **utilise le cache existant** au lieu de la reconstruire.
- Le cache est **invalidé** dès qu'une instruction **ou une couche précédente** change.

► **Exemple :**

```
FROM python:3.11
COPY . /app
RUN pip install -r /app/requirements.txt
```

- Si les fichiers copiés changent, COPY invalide le cache et le RUN sera exécuté à nouveau.

► **Avantages :**

-  **Builds plus rapides**
-  **Tests itératifs efficaces**
-  **Réduction de la consommation réseau et CPU**

►  Comprendre le fonctionnement du cache de build permet d'**écrire des Dockerfile optimisés** et **éviter des reconstructions inutiles**.

4.4.3. Conséquences sur l'écriture des fichiers Dockerfile

Le fonctionnement du **cache à la compilation** influe directement sur la manière d'écrire un Dockerfile efficace.

► **Ordre des instructions stratégique :**

- Placer en **haut** les instructions **les moins susceptibles de changer** (FROM, ENV, RUN apt install, etc.).
- Regrouper les commandes stables pour **maximiser le cache**.

► **Exclure les fichiers volatils :**

- Les changements fréquents (ex : fichiers source, config) doivent apparaître **le plus bas possible** dans le Dockerfile.
- Utiliser un fichier .dockerignore pour éviter d'invalider le cache inutilement.

4.4.3. Conséquences sur l'écriture des fichiers Dockerfile

► Exemple d'écriture optimisée :

```
FROM node:20

# Dépendances rarement modifiées
COPY package.json package-lock.json ./ 
RUN npm install

# Code source modifiable
COPY . .
```

► Bénéfices :

- Builds **plus rapides**.
- Meilleure **expérience en développement**.
- Moins de **ressources consommées**.

► ➡️ Un bon Dockerfile doit être **structuré pour exploiter intelligemment le cache**.

4.4.4. Ajout d'une image de cache intermédiaire

Il est possible d'utiliser des **images intermédiaires** pour **réutiliser une partie du build** et accélérer les futures constructions.

► Principe :

- Une étape intermédiaire d'un Dockerfile peut être **taguée** comme image temporaire.
- Elle peut être **réutilisée comme base** dans un autre Dockerfile ou un nouveau build.

```
# Étape 1 : build des dépendances
FROM node:18 AS build-deps
WORKDIR /app
COPY package*.json ./
RUN npm install

# Étape 2 : build final
FROM node:18
WORKDIR /app
COPY --from=build-deps /app/node_modules ./node_modules
COPY . .
CMD [ "node", "app.js" ]
```

4.4.4. Ajout d'une image de cache intermédiaire

```
# Étape 1 : build des dépendances
FROM node:18 AS build-deps
WORKDIR /app
COPY package*.json ./
RUN npm install

# Étape 2 : build final
FROM node:18
WORKDIR /app
COPY --from=build-deps /app/node_modules ./node_modules
COPY ..
CMD ["node", "app.js"]
```

- ▶ **Avantages :**
 - Réutilisation des **couches de dépendances** entre builds.
 - Moins d'opérations à répéter → **builds plus rapides**.
 - Meilleure **séparation des responsabilités**.
- ▶  Utiliser des images intermédiaires permet une **approche modulaire et performante**, surtout dans les projets complexes.

4.4.5. Mise en œuvre d'un cache de paquetages

L'installation de dépendances (npm, pip, apt, etc.) peut être **longue et répétitive** si mal gérée.
Docker permet d'**optimiser ces étapes** avec un cache bien structuré.

- ▶ **Objectif :**
 - Ne réinstaller les dépendances **que si les fichiers de configuration ont changé**.
- ▶ **Exemple : projet Node.js**

```
FROM node:20-alpine

WORKDIR /app

# Étape 1 : copier uniquement les fichiers de dépendances
COPY package*.json ./
RUN npm install

# Étape 2 : copier le reste du code
COPY . .

CMD [ "node", "app.js" ]
```

- ▶ Si seul le code source change, npm install **reste en cache** ✓

4.4.5. Mise en œuvre d'un cache de paquetages

► Autres cas :

- Python : requirements.txt → pip install
- PHP : composer.json → composer install
- Java : pom.xml → mvn install

► Astuce :

- Bien structurer .dockerignore pour éviter l'invalidation inutile du cache.

► ➡ Cette méthode **réduit drastiquement les temps de build et soulage les serveurs de CI/CD.**

4.4.6. Conséquences sur le choix des images de base

Le choix de l'image de base impacte **la taille, la sécurité et la performance** de votre image finale.

▶ Types d'images de base :

- **Complètes** : ex. *ubuntu*, *debian*
 - Riches en outils, mais lourdes.
 - 🟡 Utiles en dev, mais à éviter en prod si non nécessaire.
- **Minimales** : ex. *alpine*, *slim*
 - Très légères (quelques Mo), idéales pour la production.
 - ⚠ Parfois plus difficiles à utiliser (librairies manquantes).

```
FROM node:18          # ~120 Mo
FROM node:18-alpine  # ~30 Mo
```

▶ Recommandations :

- Choisir l'image **la plus légère compatible avec vos besoins**.
- Privilégier les variantes **officielles et maintenues**.
- Tenir compte des **problèmes de compatibilité potentiels** (libc, paquets...).

▶ ➤ Une bonne image de base = **plus de performance, moins d'exposition aux failles, et des builds plus rapides**.

4.4.7. Arborescence recommandée

Une **bonne organisation de projet** facilite la construction, la maintenance et le partage d'images Docker.

- Exemple d'arborescence simple :

```
mon-projet/
├── Dockerfile
├── .dockerignore
├── README.md
├── src/
│   └── main.py
└── tests/
    └── test_main.py
```

4.4.7. Arborescence recommandée

► Bonnes pratiques :

- Placer le Dockerfile à la **racine du projet**.
- Inclure un .dockerignore pour **exclure les fichiers inutiles** (logs, venv/, node_modules/, .git/...).
- Séparer clairement :
 - Le **code source** (/src)
 - Les **dépendances** (requirements.txt, package.json)
 - Les **tests**, données, etc.

► Objectif :

- Avoir un **contexte de build propre** et rapide.
- Faciliter l'intégration dans un dépôt Git ou une chaîne CI/CD.

► ➤ Une arborescence claire permet des **Dockerfile simples, efficaces et durables**.

4.4.8. La question du processus unique

Docker encourage le principe "**1 conteneur = 1 processus**".

▶ Pourquoi ce principe ?

- Chaque conteneur doit exécuter **une seule tâche principale**.
- Permet :
 - Une **meilleure isolation**.
 - Un **démarrage plus rapide**.
 - Une **gestion simplifiée** (logs, supervision, redémarrage...).

▶ Mauvaise pratique :

- Lancer plusieurs services dans un seul conteneur via supervisord, bash, etc.
- Problèmes de monitoring, de plantage partiel, de redémarrage...

▶ Bonne pratique :

- Un service = un conteneur.
- Coordination via un outil d'orchestration (ex : **Docker Compose** ou **Kubernetes**).

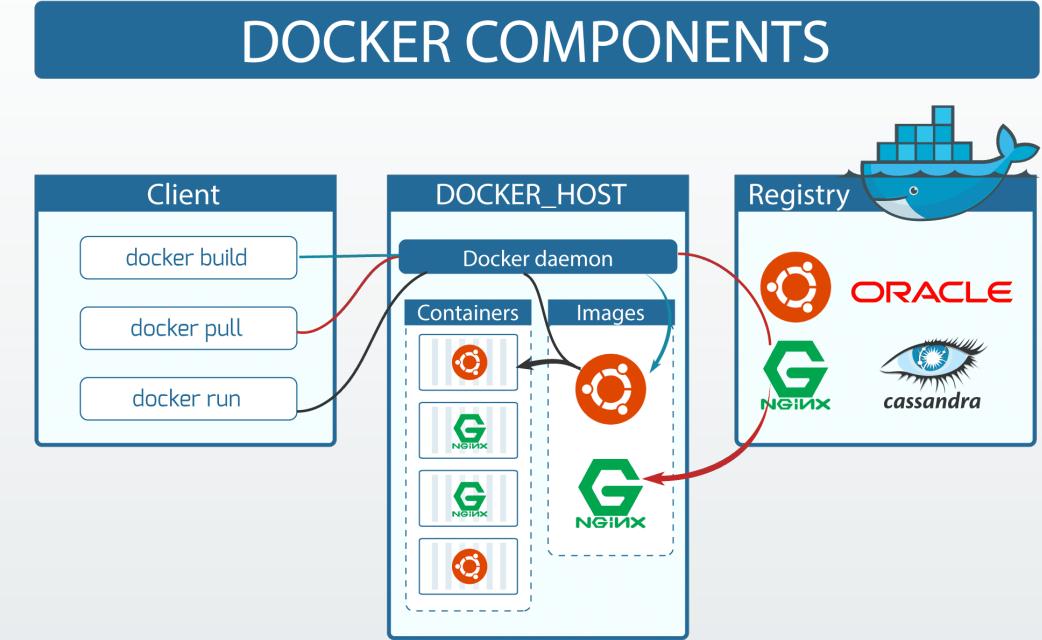
▶ Exemple :

- Web : nginx
- App : node, python, etc.
- BDD : postgres, mongo...

▶ ➡ Respecter ce principe favorise une **architecture modulaire, maintenable et scalable**.

4.5. Mise en œuvre d'un registre privé

- ▶ 4.5.1. Objectifs
- ▶ 4.5.2. Votre registre en complète autonomie
- ▶ 4.5.3. Les limites du mode autonome
- ▶ 4.5.4. Utilisation d'un service cloud déjà préparé
- ▶ 4.5.5. Approches complémentaires



4.5.1. Objectifs

Un **registre privé Docker** permet de **stocker, partager et contrôler** les images en dehors de Docker Hub.

▶ Pourquoi utiliser un registre privé ?

- **Confidentialité** : ne pas exposer les images sur un registre public.
- **Maîtrise des accès** : gestion fine des droits (lecture, écriture, suppression).
- **Indépendance** : éviter la dépendance aux services externes comme Docker Hub.
- **Performances** : hébergement en interne pour un **accès plus rapide** dans une infrastructure fermée.

▶ Cas d'usage typiques :

- Déploiement en environnement **on-premise** ou en **intranet**.
- Gestion d'**images métiers propriétaires**.
- **Chaînes CI/CD internes** nécessitant un stockage maîtrisé.

▶ ➤ L'objectif est de disposer d'un **espace de stockage privé et contrôlé pour les images Docker**, en accord avec les contraintes techniques et de sécurité.

4.5.2. Votre registre en complète autonomie

Docker permet de **déployer un registre privé local** en toute simplicité, grâce à une image officielle.

▶ Démarrer un registre en local :

```
docker run -d -p 5000:5000 --name mon-registre registry:2
```

- L'image officielle registry:2 met à disposition un **registre HTTP basique**.
- Accessible via localhost:5000.

▶ Pousser une image dans le registre :

```
docker tag monimage localhost:5000/monimage
docker push localhost:5000/monimage
```

4.5.2. Votre registre en complète autonomie

- ▶ Récupérer l'image depuis le registre :

```
docker pull localhost:5000/monimage
```

- ▶ Caractéristiques :
 - Simple à mettre en œuvre.
 - Fonctionne sans connexion Internet.
 - Utilisable pour du **test, du développement ou de la CI locale**.
- ▶ ➡ Cette approche vous donne **le contrôle total** sur votre registre, avec une mise en place rapide.

4.5.3. Les limites du mode autonome

Le registre autonome (local avec `registry:2`) est **fonctionnel**, mais présente plusieurs **limitations importantes** en production.

- ▶ **Absence de sécurité par défaut :**
 - Pas de **chiffrement HTTPS** intégré.
 - Pas de **gestion des utilisateurs** ou d'authentification native.
- ▶ **Pas d'interface graphique :**
 - Aucune UI pour parcourir ou gérer les images.
 - Gestion uniquement via API/CLI.
- ▶ **Stockage non persistant (par défaut) :**
 - Les images sont perdues si le conteneur est supprimé sans volume dédié.
- ▶ **Accès limité :**
 - Non accessible en dehors du réseau local sans configuration réseau spécifique.
- ▶ **Non adapté à l'échelle :**
 - Gestion manuelle des logs, de la réPLICATION, des sauvegardes...
- ▶  Ce mode reste utile pour du **prototypage, du test ou de la formation**, mais **pas pour un usage industriel sans renforts**.

4.5.4. Utilisation d'un service cloud déjà préparé

Pour éviter la complexité d'un registre auto-hébergé, il est possible d'utiliser un **registre managé dans le cloud**, prêt à l'emploi.

- ▶ **Exemples de solutions populaires :**
 - **Docker Hub (privé)** : support des dépôts privés avec gestion des accès.
 - **GitHub Container Registry (GHCR)** : intégré à GitHub, gestion par organisation.
 - **GitLab Container Registry** : intégré aux projets GitLab avec pipeline CI/CD.
 - **Amazon ECR / Google Artifact Registry / Azure ACR** : registres managés par les grands cloud providers.
- ▶ **Avantages :**
 - Sécurité intégrée (authentification, HTTPS, ACL).
 - Disponibilité et **maintenance assurée**.
 - Intégration native avec les **outils CI/CD**.
 - Support du **versioning**, de la gestion des tags, du nettoyage automatique.
- ▶ **Inconvénients potentiels :**
 - \$ Coût à l'usage (souvent au-delà d'un quota gratuit).
 -  Dépendance à une plateforme externe.
- ▶  Ces services offrent une **solution fiable, scalable et rapide à mettre en œuvre** pour gérer vos images en production.

4.5.5. Approches complémentaires

La mise en place d'un registre d'images Docker peut combiner **plusieurs stratégies**, selon les besoins et les contextes.

- ▶ **Approche hybride : local + cloud**
 - Utiliser un **registre local pour le développement** rapide et les tests internes.
 - Pousser les versions validées vers un **registre cloud sécurisé** pour le déploiement.
- ▶ **Proxy cache Docker Registry**
 - Permet de **cacher les images publiques** (ex : library/nginx) en local.
 - Réduction de la bande passante et **accès plus rapide**.
- ▶ **Authentification et contrôle d'accès**
 - Intégration d'**outils externes** (comme nginx, oauth2-proxy) pour sécuriser un registre auto-hébergé.
 - Utilisation de **solutions d'entreprise** avec gestion fine des rôles et permissions.
- ▶ **Intégration CI/CD**
 - Les registres doivent être **intégrés dans les pipelines** pour automatiser build, push et déploiement.
- ▶  Choisir une approche adaptée permet de **concilier sécurité, performance, maîtrise des coûts et flexibilité opérationnelle**.

4.6. Incorporation dans le cycle de développement

- ▶ 4.6.1. Positionnement de Docker dans une usine logicielle
- ▶ 4.6.2. Exemple de mise en œuvre
- ▶ 4.6.3. Docker comme une commodité



4.6.1. Positionnement de Docker dans une usine logicielle

Docker joue un **rôle central** dans les chaînes de développement modernes, dites **usines logicielles**.

► Rôle dans les étapes clés :

- **Développement**
 - → Environnement reproductible pour tous les développeurs.
 - → Intégration dans des outils comme VS Code, GitPod...
- **Intégration continue (CI)**
 - → Build des images via GitHub Actions, GitLab CI, Jenkins...
 - → Exécution des tests dans des conteneurs isolés.
- **Déploiement continu (CD)**
 - → Déploiement automatisé d'images vers différents environnements (recette, prod...).
 - → Utilisation avec Docker Compose, Swarm, ou Kubernetes.
- **Monitoring & Observabilité**
 - → Conteneurs traçables, observables, facilement redémarrables.

4.6.1. Positionnement de Docker dans une usine logicielle

► Avantages dans la chaîne :

- Cohérence entre dev, test et prod ✓
- Vitesse d'exécution ✓
- Isolation et maîtrise des dépendances ✓

► ➤ Docker est une **brique clé de la démarche DevOps**, du premier commit au déploiement final.

4.6.2. Exemple de mise en œuvre

Objectif : Automatiser le cycle **build** → **test** → **push** → **déploiement** d'une application Dockerisée via une pipeline CI/CD.

▶  **Étapes d'une chaîne simplifiée avec GitLab CI :**

1. **Push du code** sur Git
2. Le pipeline **build l'image Docker** :

```
docker build -t registry.gitlab.com/groupe/app:latest .
```

3. **Tests automatisés** dans un conteneur :

```
docker run --rm groupe/app:latest pytest
```

4.6.2. Exemple de mise en œuvre

4. Push de l'image dans un registre privé :

```
docker push registry.gitlab.com/groupe/app:latest
```

5. Déploiement automatique sur un serveur ou cluster :

- Via SSH, Docker Compose, ou Helm/Kubernetes

▶ Résultat :

- Livraison continue sans intervention manuelle
- Traçabilité, reproductibilité, qualité garantie

▶ 👉 Docker devient un **maillon technique fluide** entre le développeur et la production.

4.6.3. Docker comme une commodité

Avec sa large adoption, Docker est devenu une **infrastructure de base**, au même titre que Git ou un système de build.

▶ **Un outil devenu omniprésent :**

- Utilisé dans le **développement**, le **test**, la **CI/CD**, le **déploiement**, la **formation**, etc.
- Supporté nativement par tous les grands clouds, outils DevOps, IDE...

▶ **Ce que ça change :**

- Moins besoin de "connaître Docker" pour l'utiliser : il est **abstrait dans les outils**.
- Ce qui compte : **bien l'intégrer** dans un cycle maîtrisé et sécurisé.
- Docker devient une **commodité** : invisible mais indispensable.

▶ **Enjeux pour les équipes :**

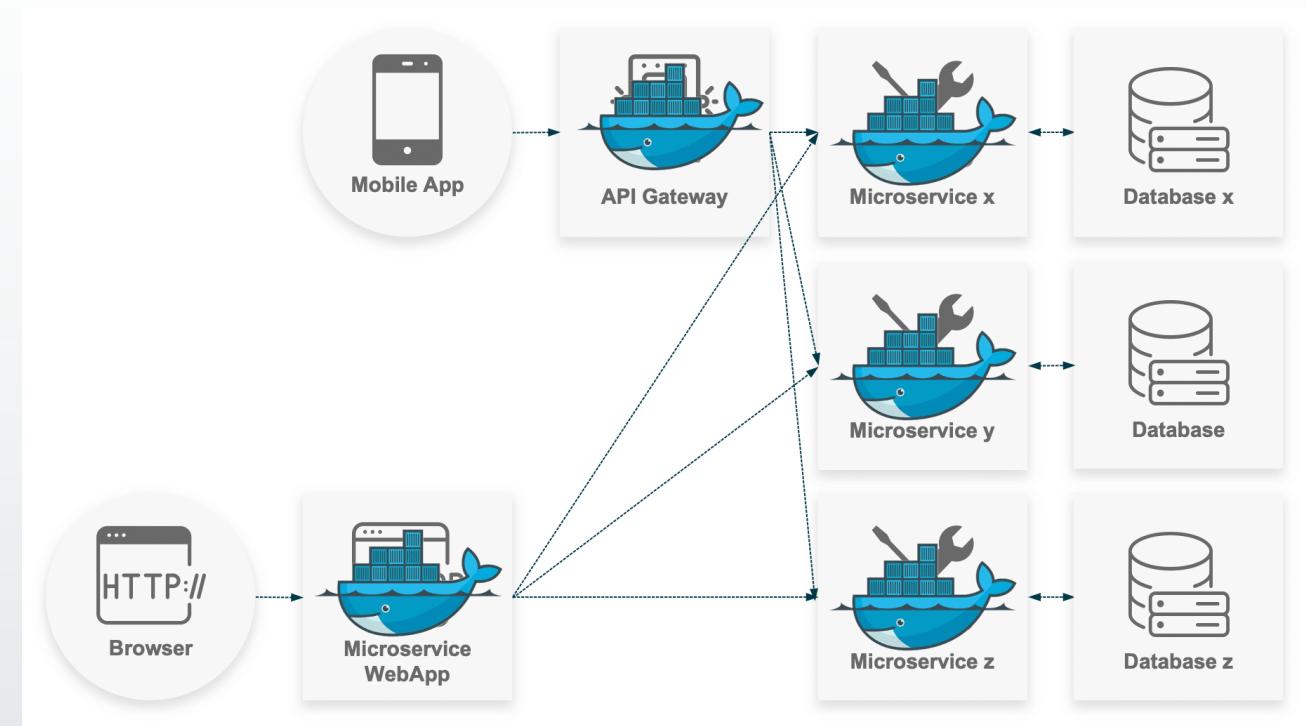
- Ne pas "juste dockeriser", mais **concevoir des architectures adaptées**.
- Raisonner en termes de **cycle de vie des images**, de **qualité des Dockerfile**, de **flux CI/CD**.

▶ ➡ Docker n'est plus une techno isolée : c'est un **standard de fait**, une **brique invisible mais critique** du cycle logiciel moderne.

Chapitre 5

Mise en œuvre pratique

- ▶ 5.1. Présentation de l'application exemple
- ▶ 5.2. Adaptation à Docker de l'application exemple



5.1. Présentation de l'application exemple

- ▶ 5.1.1. Architecture
- ▶ 5.1.2. Création d'un fichier des mots de passe
- ▶ 5.1.3. Mise en place des certificats de sécurité
- ▶ 5.1.4. Prérequis
- ▶ 5.1.5. Paramétrage
- ▶ 5.1.6. Première utilisation
- ▶ 5.1.7. Utilité
- ▶ 5.1.8. Principe à l'oeuvre



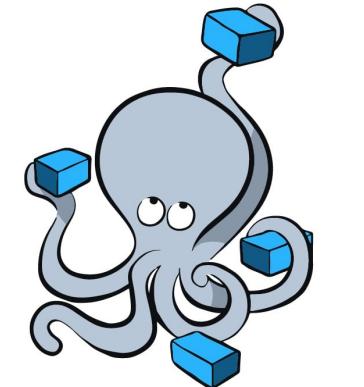
5.2. Adaptation à Docker de l'application exemple

- ▶ 5.2.1. Préparation de l'environnement
- ▶ 5.2.2. Principes de construction
- ▶ 5.2.3. Détails du service RecepteurMessages
- ▶ 5.2.4. Lancement de RecepteurMessages en mode Docker
- ▶ 5.2.5. Adaptation de Visual Studio
- ▶ 5.2.6. Retour sur le problème de secret
- ▶ 5.2.7. Problème des paramètres liés à XKCD
- ▶ 5.2.8. Traitement du projet API
- ▶ 5.2.9. Traitement du projet Server
- ▶ 5.2.10. Cas particulier des paramètres du projet Client
- ▶ 5.2.11. Test de l'application passée en Docker
- ▶ 5.2.12. État atteint

Chapitre 6

Orchestration par Docker Compose

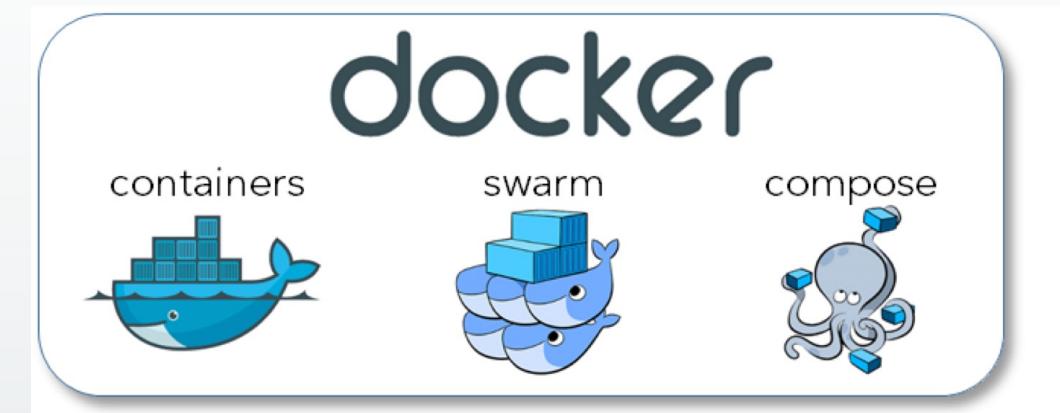
- ▶ 6.1. Redéployer automatiquement avec Docker Compose
- ▶ 6.2. Fonctionnalités supplémentaires de Docker Compose
- ▶ 6.3. Exploitation d'une infrastructure Docker
- ▶ 6.4. Exemples de fichier Docker pour d'autres plateformes



docker
Compose

6.1. Redéployer automatiquement avec Docker Compose

- ▶ 6.1.1. Principe du Docker Compose
- ▶ 6.1.2. Ecriture du fichier docker-compose.yml
- ▶ 6.1.3. Mise en œuvre
- ▶ 6.1.4. Mise en œuvre avec IAM externalisée
- ▶ 6.1.5. Débogages complémentaires du mode Docker Compose



6.1.1. Principe du Docker Compose

Docker Compose est un outil permettant de **décrire, configurer et lancer plusieurs conteneurs** à partir d'un simple fichier YAML.

► **Objectif :**

- Gérer des **architectures multi-conteneurs** (ex. : application + base de données + cache...).
- Définir l'environnement complet dans un seul fichier lisible et versionnable.

► **Avantages :**

- Déploiement unifié : un seul fichier pour tout lancer.
- Reproductibilité : même configuration sur tous les postes.
- Maintenabilité : configuration claire et centralisée.
- Réseau privé automatique entre services.

► **Fonctionnement :**

- Fichier docker-compose.yml → contient les services, volumes, réseaux...
- Commande clé :

```
docker compose up
```

► 👉 Docker Compose permet de **modulariser, automatiser et simplifier** le lancement d'applications conteneurisées complexes.

6.1.2. Ecriture du fichier docker-compose.yml

- Le fichier docker-compose.yml décrit une **architecture multi-conteneurs** dans un format lisible et déclaratif.

- Structure de base :**

- Principaux éléments :**

- services : définition de chaque conteneur.
- build* ou *image* : source du conteneur.
- ports* : mappage des ports.
- volumes*, *networks*, *environment*, etc. : options complémentaires.
- depends_on* : gère l'ordre de lancement.

- Bonnes pratiques :**

- Indenter correctement (YAML est sensible).
- Versionner le fichier avec le projet (.yml à la racine).

- 👉 Le docker-compose.yml est le **point central de configuration d'un environnement applicatif Dockerisé**.
- Tips : le fichier peut maintenant s'appeler également compose.yml

```
version: "3.9" # Déprécié dans les versions actuelles

services:
  web:
    image: nginx
    ports:
      - "8080:80"

  app:
    build: ./app
    depends_on:
      - db

  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: exemple
```

6.1.3. Mise en œuvre

Une fois le fichier docker-compose.yml rédigé, **le déploiement complet se fait en une seule commande.**

► **Lancement de l'environnement :**

```
docker compose up
```

- Démarre tous les services définis.
- Affiche les logs en direct dans le terminal.

► **Exécution en arrière-plan :**

```
docker compose up -d
```

- Mode détaché (background).
- Les conteneurs tournent en tâche de fond.

6.1.3. Mise en œuvre

► Vérification des services :

```
docker compose ps
```

► Arrêt et suppression des conteneurs :

```
docker compose down
```

► Avantages :

- Déploiement rapide, sans script shell complexe.
- Simplifie la vie des développeurs, testeurs, formateurs...

► ➤ Docker Compose permet de **répliquer une architecture entière en une seule commande**, localement ou sur un serveur.

6.1.4. Mise en œuvre avec IAM externalisée

Dans un contexte professionnel, on peut coupler Docker Compose à un **système d'authentification et d'autorisation externe** (IAM = Identity & Access Management).

► Objectif :

- **Sécuriser l'accès** à des services sensibles (API, bases de données, front-end admin...).
- **Externaliser la gestion des identités** via un fournisseur IAM (Keycloak, Auth0, Azure AD...).

► Exemple d'intégration (Keycloak + reverse proxy) :

- L'authentification est **déléguee à Keycloak** via un reverse proxy configuré en front.
- Le reste des services est **protégé sans modifier leur code**.

► Bonnes pratiques :

- Séparer les **préoccupations métier / sécurité**.
- Centraliser la gestion des accès pour **tous les services** Dockerisés.

►👉 En intégrant un IAM, Docker Compose devient une **brique fiable pour des architectures sécurisées**.

```
services:  
  keycloak:  
    image: quay.io/keycloak/keycloak  
    ports:  
      - "8080:8080"  
    environment:  
      KEYCLOAK_ADMIN: admin  
      KEYCLOAK_ADMIN_PASSWORD: admin  
  
  nginx:  
    image: nginx  
    volumes:  
      - ./nginx.conf:/etc/nginx/nginx.conf  
    depends_on:  
      - keycloak
```

6.1.5. Débogages complémentaires

Docker Compose offre plusieurs outils intégrés pour **observer, analyser et corriger les problèmes** dans les environnements multi-conteneurs.

► **Logs en temps réel :**

```
docker compose logs -f
```

- Affiche les logs de tous les services.
- Utiliser -f pour suivre en continu, --tail pour limiter.

► **Exécution d'une commande dans un conteneur :**

```
docker compose exec <service> bash
```

- Permet d'**entrer dans un conteneur** pour inspecter, tester, éditer...

► **Rebuild complet d'un service :**

```
docker compose up --build
```

- Reconstruit l'image même si elle existe déjà.

6.1.5. Débogages complémentaires

► Vérification des dépendances :

```
docker compose config
```

- Affiche le fichier YAML fusionné et validé → utile pour repérer les erreurs de syntaxe ou de logique.

► Nettoyage complet :

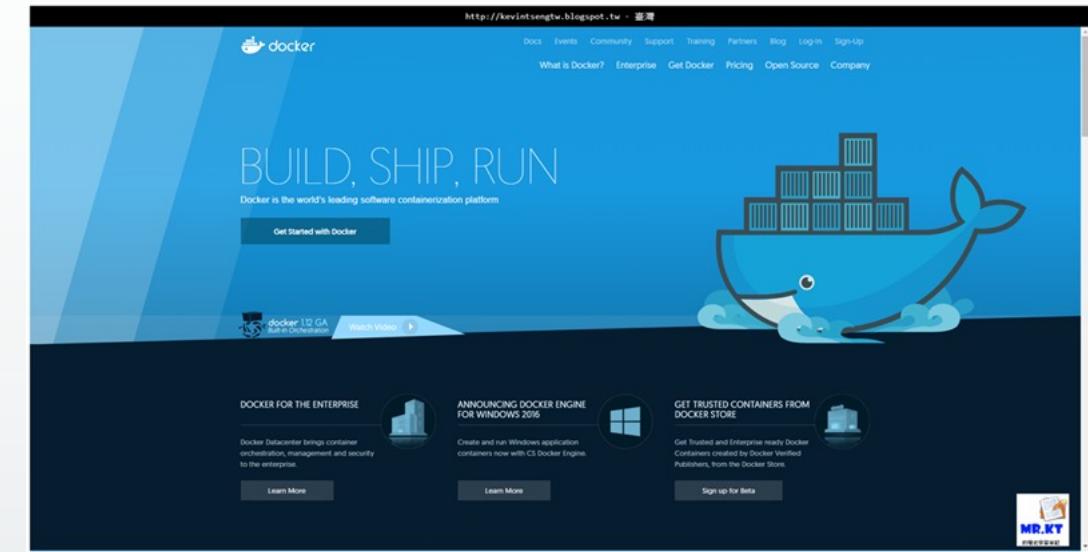
```
docker compose down -v --rmi all
```

- Supprime conteneurs, volumes et images associés.

► ➤ Docker Compose fournit les **outils nécessaires pour diagnostiquer finement un environnement en erreur.**

6.2. Fonctionnalités supplémentaires de Docker Compose

- ▶ 6.2.1. Retour sur la gestion des conteneurs
- ▶ 6.2.2. Parallélisation des traitements
- ▶ 6.2.3. Autre avantage de l'approche par conteneurs
- ▶ 6.2.4. Limites
- ▶ 6.2.5. Intégration dans Visual Studio Code



6.2.1. Retour sur la gestion des conteneurs

Docker Compose simplifie la **gestion coordonnée de plusieurs conteneurs**, tout en s'appuyant sur les principes vus précédemment.

► **Regroupement logique**

- Tous les conteneurs définis dans le fichier docker-compose.yml sont traités comme un **ensemble cohérent**.
- Gestion collective : build, démarrage, arrêt, suppression.

► **Commandes utiles (rappel) :**

```
docker compose up          # Démarrage des services
docker compose down        # Arrêt + nettoyage
docker compose ps          # État des conteneurs
docker compose logs -f     # Logs en direct
docker compose exec <name> bash # Entrer dans un conteneur
```

► **Gestion facilitée :**

- Noms automatiques (projet_service_index).
- Réseau dédié généré automatiquement.
- Volume partagé défini dans le YAML.

► 👍 Docker Compose devient une **interface de gestion unique et claire** pour une architecture multi-conteneurs.

6.2.2. Parallélisation des traitements

L'architecture multi-conteneurs avec Docker Compose **favorise naturellement la parallélisation** des traitements.

► **Conteneurs = processus indépendants**

- Chaque service tourne dans son propre conteneur, avec ses propres ressources.
- Exécution **concurrente** dès le démarrage :

```
docker compose up
```

- Tous les services sont lancés en parallèle, sauf dépendances explicites (depends_on).

► **Cas d'usage typiques :**

- Traitement parallèle de données (ex : workers).
- Microservices interconnectés (API, BDD, cache, etc.).
- Tâches asynchrones (ex : file de messages + consommateur).

► **Optimisations possibles :**

- **Répartition de charge** sur plusieurs conteneurs d'un même service.
- Utilisation de **CPU/RAM par conteneur configurable** (avec deploy → en Swarm/K8s).

► ➡ Docker Compose permet de **modulariser et distribuer le travail**, pour des applications plus réactives et scalables.

6.2.3. Autre avantage de l'approche par conteneurs

L'approche par conteneurs ne se limite pas à l'exécution parallèle : elle offre des **bénéfices structurels majeurs**.

► Isolation des composants

- Chaque service s'exécute dans un **espace isolé**, avec ses propres dépendances.
- Pas de conflit de versions, pas d'interférence entre services.

► Portabilité totale

- Une architecture définie dans un docker-compose.yml peut être **déployée à l'identique** sur :
 - Une machine locale
 - Un serveur distant
 - Un environnement de CI/CD
 - Le cloud (via Swarm ou Kubernetes)

► Maintenabilité et évolutivité

- Chaque service peut être **modifié, mis à jour ou remplacé** sans impacter les autres.
- Favorise les **architectures découplées** (ex : microservices).

► Environnement reproduit

- Tous les développeurs/testeurs utilisent exactement la **même stack**, sans configuration manuelle.

►👉 L'approche conteneurisée transforme l'infrastructure en **composants modulaires, sûrs et déployables partout**.

6.2.4. Limites

Même si Docker Compose est puissant, son utilisation **présente des limites**, en particulier dans les contextes complexes ou critiques.

▶ **Environnements non persistants**

- Conteneurs sont **éphémères** : perte de données si volumes non définis.
- Requiert une bonne gestion des **volumes** pour conserver l'état.

▶ **Pas de montée en charge native**

- Docker Compose ne gère pas le **scaling automatique** des services.
- Pas de **load balancing intégré** (au-delà du réseau interne Docker).

▶ **Supervision limitée**

- Pas d'outils de monitoring, d'alertes ou de logs centralisés intégrés par défaut.

▶ **Déploiement mono-machine**

- Docker Compose est conçu pour un **usage local ou sur une seule machine**.
- Pas adapté à une infrastructure **multi-nœuds ou distribuée**.

▶ **Sécurité simplifiée**

- Pas de gestion native des secrets ou des règles d'accès avancées (IAM, RBAC).

▶ ➤ Ces limites justifient, dans certains cas, le **passage à des outils d'orchestration plus avancés** (Swarm, Kubernetes).

6.2.5. Intégration dans Visual Studio Code

Visual Studio Code (VS Code) propose une **intégration native et puissante de Docker et Docker Compose** grâce à des extensions dédiées.

► **Extension Docker officielle :**

- Disponible sur le marketplace : `ms-azuretools.vscode-docker`
- Fonctionnalités :
 - Exploration des **images, conteneurs, volumes, réseaux**
 - **Build, run, push** et **pull** via interface graphique
 - Génération assistée de Dockerfile et docker-compose.yml

► **Support de docker compose**

- Détection automatique du fichier docker-compose.yml
- Lancement direct depuis l'interface :
 - Bouton "**Compose Up**" / "**Compose Down**"
 - Accès aux **logs par service**
 - Terminal intégré

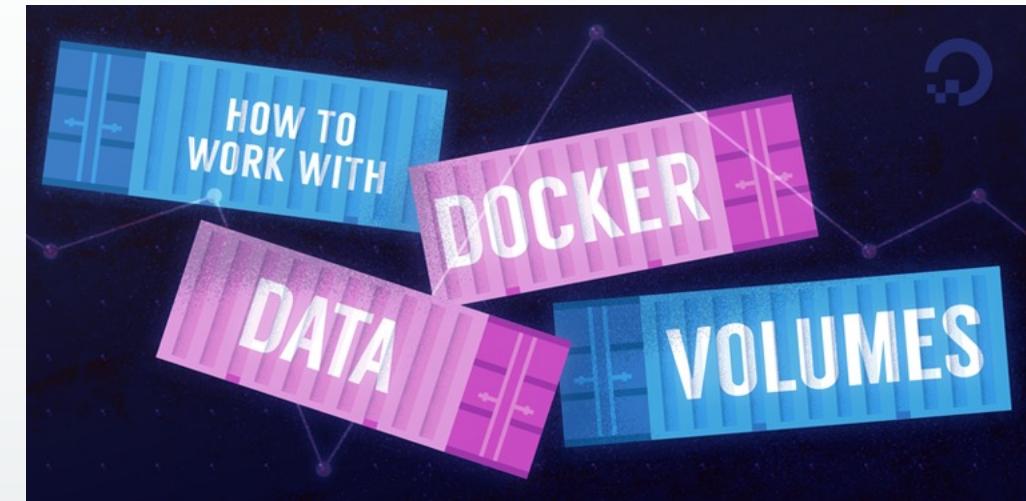
► **Avantages pédagogiques :**

-  Visualisation simplifiée pour les débutants
-  Productivité accrue grâce à l'autocomplétion, la coloration YAML, le terminal intégré
-  Idéal pour les tests locaux rapides

►  L'intégration VS Code rend **l'usage de Docker plus accessible** et fluide, même sans ligne de commande.

6.3. Exploitation d'une infrastructure Docker

- ▶ 6.3.1. Le réseau dans Docker
- ▶ 6.3.2. Les volumes Docker



6.3.1. Le réseau dans Docker

Docker crée des **réseaux virtuels isolés** pour permettre la communication entre conteneurs, tout en garantissant leur isolation.

► Types de réseaux par défaut :

- bridge (par défaut) :
 - Réseau local entre conteneurs sur une même machine.
 - Nom des services utilisables comme **hostnames** (db, web, etc.).
- host :
 - Partage directement le réseau de la machine hôte (pas d'isolation réseau).
 - Utilisé pour des performances maximales ou compatibilité.
- none :
 - Pas de réseau attaché au conteneur.

6.3.1. Le réseau dans Docker

► Avec Docker Compose et des networks customs :

- web et api communiquent via le réseau frontnet.
- api et db échangent via backnet.
- **Isolation logique et meilleure sécurité** en séparant les flux.

► ➡️ Les conteneurs d'un même réseau peuvent se contacter **par leur nom de service**, les autres sont isolés.

```
services:  
  web:  
    image: nginx  
    networks:  
      - frontnet  
  
  api:  
    image: my-api  
    networks:  
      - frontnet  
      - backnet  
  
  db:  
    image: postgres  
    networks:  
      - backnet  
  
networks:  
  frontnet:  
  backnet:
```

6.3.1. Le réseau dans Docker

► Avec Docker Compose :

- Un **réseau est créé automatiquement** pour tous les services du projet.
- Communication possible par **nom de service** :

```
app:  
  depends_on:  
    - db
```

→ L'app peut se connecter à la base via l'adresse db:5432

► Commandes utiles :

```
docker network ls          # Lister les réseaux  
docker network inspect <id> # Détails sur un réseau
```

- ➤ Docker fournit une **infrastructure réseau simple et isolée**, idéale pour reproduire un mini-système distribué.

6.3.2. Les volumes Docker

- ▶ Les **volumes** permettent de **persister des données** au-delà de la durée de vie des conteneurs.
- ▶ **Pourquoi utiliser des volumes ?**
 - Les données internes à un conteneur sont **éphémères**.
 - Un volume permet de :
 - **Conserver des bases de données**
 - **Partager des fichiers entre services**
 - **Sauvegarder des logs ou des fichiers de configuration**
- ▶ **Déclaration dans docker-compose.yml :**

```
services:  
  db:  
    image: postgres  
    volumes:  
      - db-data:/var/lib/postgresql/data  
  
volumes:  
  db-data:
```

- Le volume db-data stocke les données en dehors du conteneur db.

6.3.2. Les volumes Docker

► Commandes utiles :

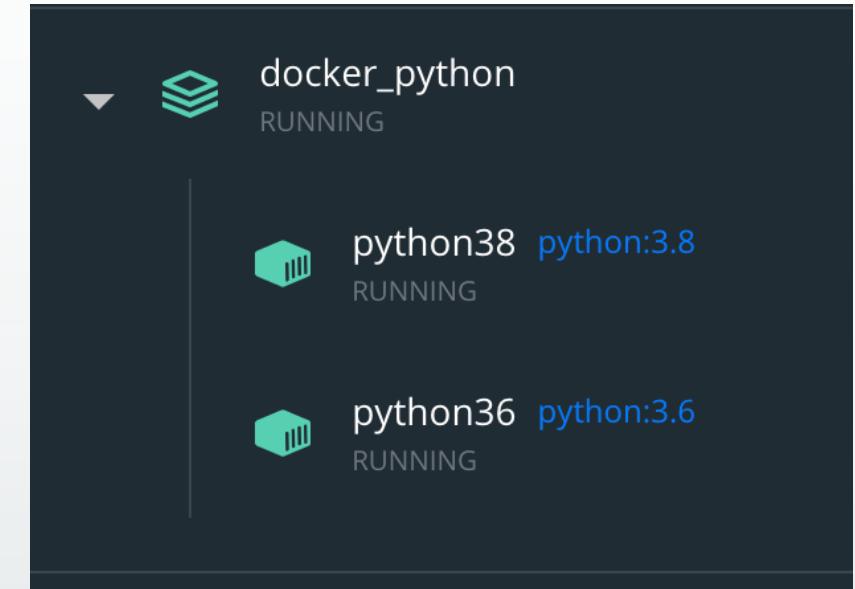
```
docker volume ls          # Liste des volumes
docker volume inspect <nom> # Détails d'un volume
docker volume rm <nom>     # Suppression manuelle
```

► Avantage :

- 📦 Séparation claire **conteneur ≠ données**
 - ⚡ Recréation des conteneurs sans perte d'information
- 👉 Les volumes sont **indispensables** dès que des **données doivent être conservées ou partagées.**

6.4. Exemples de fichier Docker pour d'autres plateformes

- ▶ 6.4.1. Java
- ▶ 6.4.2. PHP
- ▶ 6.4.3. Node.js
- ▶ 6.4.4. Go
- ▶ 6.4.5. Python
- ▶ 6.4.6. Un mot sur les DevContainers



6.4.1. Java

Une application Java packagée en .jar peut être exécutée facilement dans un conteneur.

► **Exemple de Dockerfile :**

```
FROM openjdk:17-jdk-slim

WORKDIR /app

COPY target/monapp.jar app.jar

CMD ["java", "-jar", "app.jar"]
```

► **Avantages :**

- Déploiement rapide d'une **application Java autonome**.
- Compatible avec Maven, Gradle, Spring Boot...

► ➤ Docker permet de **standardiser l'exécution de projets Java** sur tout type d'environnement.

6.4.2. PHP

Une application PHP peut être exécutée avec un **serveur web intégré** (dev) ou avec Apache/Nginx (prod).

► **Exemple de Dockerfile (mode dev) :**

```
FROM php:8.2-cli

WORKDIR /app

COPY . .

CMD ["php", "-S", "0.0.0.0:8000", "-t", "."]
```

► **Utilisation :**

```
docker build -t monapp-php .
docker run -p 8000:8000 monapp-php
```

- Accès à l'app : <http://localhost:8000>

6.4.2. PHP

► Variante avec Apache :

```
FROM php:8.2-apache  
  
COPY . /var/www/html/
```

- PHP est directement servi via **Apache** (port 80 exposé par défaut)

► Bonnes pratiques :

- Monter un volume pour le code source en dev :

```
-v $(pwd):/app
```

- Utiliser un conteneur **MySQL/MariaDB** séparé via docker compose
- ➡ Docker facilite le **lancement rapide d'environnements PHP isolés**, pour du développement ou des tests.

6.4.3. Node.js

- ▶ Node.js se prête très bien à la conteneurisation grâce à sa structure modulaire et ses outils CLI.
- ▶ **Exemple de Dockerfile :**

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 3000
CMD [ "npm", "start" ]
```

- ▶ **Étapes :**

1. Construire l'image :

```
docker build -t monapp-node .
```

2. Lancer l'application :

```
docker run -p 3000:3000 monapp-node
```

6.4.3. Node.js

► Variante avec docker compose :

```
services:  
  web:  
    build: .  
    ports:  
      - "3000:3000"  
    volumes:  
      - ./app  
      - /app/node_modules
```

- Idéal pour le développement avec **rechargement automatique** (via nodemon, par exemple).
- ➡ Docker permet de **standardiser l'environnement Node.js** et d'éviter les problèmes de version locale.

6.4.4. Go

Go permet de compiler une application en **binaire autonome**, idéal pour des images Docker **légères**.

► Dockerfile avec build multi-étapes :

```
# Étape 1 : compilation
FROM golang:1.21 AS builder

WORKDIR /app
COPY . .
RUN go build -o monapp

# Étape 2 : image minimale
FROM alpine:latest

WORKDIR /app
COPY --from=builder /app/monapp .

CMD [ "./monapp" ]
```

6.4.4. Go

► Étapes d'utilisation :

```
docker build -t monapp-go .
docker run monapp-go
```

► Avantages :

- Image finale très légère (~10 Mo avec Alpine).
- Déploiement rapide, pas besoin de runtime Go à l'exécution.

► Astuce :

- Ajouter EXPOSE, variables ENV, ou ENTRYPOINT selon le contexte.

► ➡ Le modèle de build de Go s'adapte parfaitement à une **approche Docker performante et portable**.

6.4.5. Python

Docker permet d'exécuter une application Python avec ses dépendances dans un environnement **isolé et reproductible**.

► Exemple de Dockerfile :

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD [ "python", "app.py" ]
```

► Étapes d'utilisation :

```
docker build -t monapp-python .
docker run -p 5000:5000 monapp-python
```

6.4.5. Python

► Bonnes pratiques :

- Utiliser un **virtualenv** dans le conteneur (optionnel).
- Ajouter EXPOSE si serveur web (Flask, FastAPI...) :

```
EXPOSE 5000
```

► Variante avec docker compose :

```
services:  
  app:  
    build: .  
    ports:  
      - "5000:5000"
```

- ➤ Docker est parfait pour **éviter les conflits de versions Python et partager un environnement identique** à toute l'équipe.

6.4.6. Un mot sur les DevContainers

Les **DevContainers** permettent de définir un **environnement de développement complet et portable**, basé sur Docker.

► Qu'est-ce qu'un DevContainer ?

- Une **spécification** (dossier `.devcontainer/`) contenant :
 - Un Dockerfile ou une image
 - Un fichier `devcontainer.json` (config VS Code)
- Permet d'ouvrir un projet dans **Visual Studio Code avec un environnement prêt à l'emploi**.

► Avantages :

-  Même environnement pour tous les développeurs
-  Tests locaux réalistes (ex : même que CI/CD)
-  Reproductibilité totale du poste de travail

6.4.6. Un mot sur les DevContainers

► Exemple minimal :

1. `.devcontainer/devcontainer.json`

```
{  
  "name": "Mon Projet",  
  "image": "python:3.11",  
  "postCreateCommand": "pip install -r requirements.txt"  
}
```

► Support :

- Nativement intégré dans **Visual Studio Code** (avec l'extension *Dev Containers*)
 - Compatible avec **GitHub Codespaces**
- ➤ Les DevContainers sont un **levier puissant de productivité**, en particulier pour les projets en équipe ou en formation.

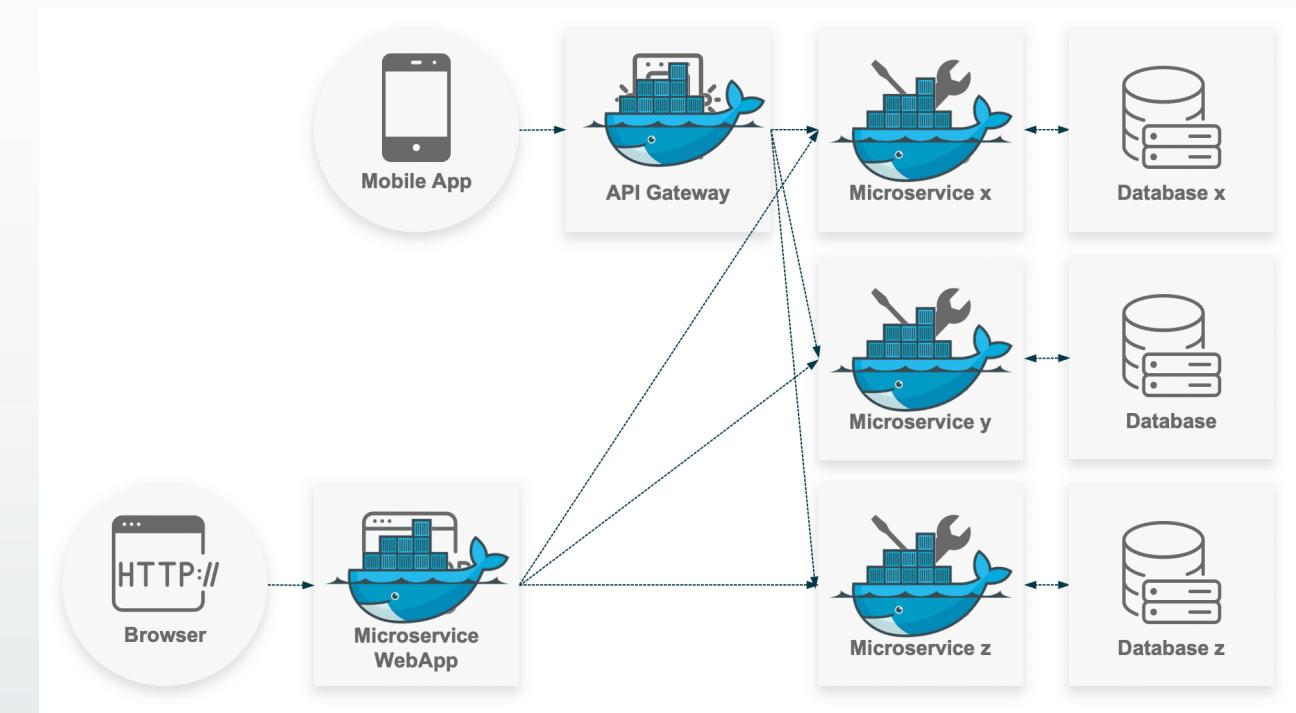
Chapitre 7

Déploiement en cluster par une usine logicielle

- ▶ 7.1. Les besoins d'orchestration
- ▶ 7.2. L'approche Docker Swarm
- ▶ 7.3. Outils avancés d'exposition
- ▶ 7.4. Introduction à Kubernetes
- ▶ 7.5. Intégration et déploiement continus
- ▶ 7.6. Azure Container Instances

7.1. Les besoins d'orchestration

- ▶ 7.1.1. Objectif
- ▶ 7.1.2. Approche théorique
- ▶ 7.1.3. Lien aux microservices
- ▶ 7.1.4. Fonctionnement pratique



7.1.1. Objectif

Lorsque le nombre de conteneurs augmente, il devient **nécessaire d'orchestrer leur exécution** de manière automatisée et fiable.

► **Problématique :**

- Un conteneur seul est facile à gérer...
- Mais **des dizaines ou centaines** de conteneurs :
 - doivent être **déployés automatiquement**
 - doivent pouvoir **se redémarrer en cas de panne**
 - doivent être **mis à l'échelle** (scalés) selon la charge

► **Objectif de l'orchestration :**

- Coordonner le **lancement, l'arrêt, la mise à jour et la supervision** des conteneurs.
- Répartir intelligemment la charge sur **plusieurs machines (nœuds)**.
- Faciliter l'**automatisation et la résilience** des environnements en production.

►  L'orchestration est une **brique indispensable** pour passer du conteneur local à un **système distribué et scalable**.

7.1.2. Approche théorique

L'orchestration repose sur un ensemble de **principes structurants** pour piloter des conteneurs à grande échelle.

▶ **Composants clés d'un orchestrateur :**

- **Cluster** : ensemble de machines (nœuds) unifiées pour exécuter les conteneurs.
- **Scheduler** : répartit les conteneurs selon la charge, les ressources disponibles et les règles.
- **Control Plane** : supervise l'état global (qui tourne ? où ? dans quel état ?).
- **Worker Nodes** : exécutent réellement les conteneurs.

▶ **Fonctions assurées :**

- **Déploiement automatisé** : en un seul fichier ou commande.
- **Redondance** : relance automatique en cas d'échec.
- **Scalabilité** : montée/descente en charge (scale up/down).
- **Mise à jour continue** : rolling updates sans interruption.

▶ **Exemple d'abstraction :**

"Je veux 3 instances de mon service API, accessibles via un seul point d'entrée, et automatiquement redémarrées si l'une tombe."

▶  L'orchestration fournit une **couche de pilotage intelligente** au-dessus de l'exécution brute des conteneurs.

7.1.3. Lien aux microservices

L'orchestration est **étroitement liée** à l'architecture de microservices, où l'application est **décomposée en services indépendants**.

▶ **Caractéristiques des microservices :**

- Chaque composant (API, base de données, front, etc.) est **isolé, déployable et scalable indépendamment**.
- Architecture distribuée, dynamique, et potentiellement multi-langages.

▶ **Besoins spécifiques :**

- **Nombre élevé de conteneurs** à coordonner.
- Besoin d'un **réseau interne fiable**, de **découverte de services**, de **mise à l'échelle**, etc.
- **Résilience** : si un service tombe, les autres doivent continuer à fonctionner.

▶ **Apport de l'orchestration :**

- Supervision de **l'état de chaque service**.
- **Auto-réparation** et relancement des services défaillants.
- **Montée en charge ciblée** sur les services les plus sollicités.

▶  Les microservices ne peuvent être **exploitables à grande échelle** que s'ils sont **orchestrés intelligemment**.

7.1.4. Fonctionnement pratique

Dans un environnement orchestré, le **développeur ne gère plus les conteneurs un par un**, mais **décrit l'état souhaité du système**.

► **Déploiement déclaratif :**

- On décrit les services dans un fichier (YAML, manifeste, etc.) :
 - Nombre de réplicas
 - Ports exposés
 - Volumes
 - Variables d'environnement
- Exemple : "Je veux 3 instances du service API et 1 base de données persistante"

► **Ce que fait l'orchestrateur :**

- **Planifie le déploiement** sur les nœuds disponibles.
- Surveille l'état des services (up/down).
- **Redémarre automatiquement** les conteneurs si besoin.
- Applique les mises à jour **progressivement** (rolling update).

7.1.4. Fonctionnement pratique

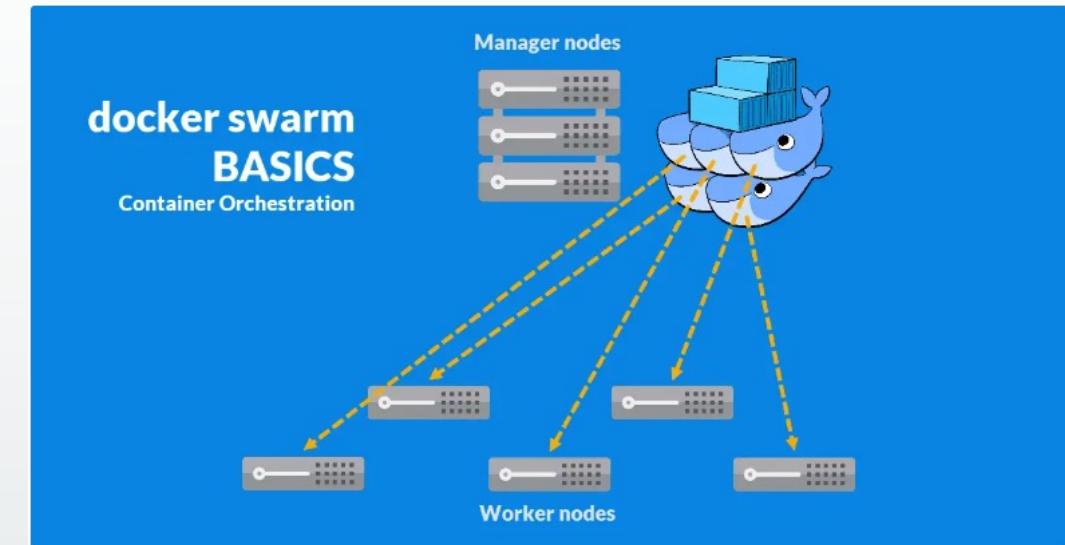
- ▶ Exécution typique :

```
docker stack deploy -c stack.yml monapp
```

- ▶ Docker (Swarm) se charge de tout le cycle de vie des services.
- ▶ ➡ Le fonctionnement pratique repose sur un **pilotage centralisé et automatisé**, pour garantir **disponibilité, résilience et évolutivité**.

7.2. L'approche Docker Swarm

- ▶ 7.2.1. Gestion du cluster Swarm
- ▶ 7.2.2. Test du cluster avec un cas simple
- ▶ 7.2.3. Déploiement manuel sur le cluster Swarm
- ▶ 7.2.4. Déploiement de l'application microservices
- ▶ 7.2.5. Considérations additionnelles



7.2.1. Gestion du cluster Swarm

Docker Swarm est le **mode natif d'orchestration de Docker**, permettant de transformer plusieurs machines en un **cluster coordonné**.

▶ **Initialisation du cluster :**

```
docker swarm init
```

- Initialise le **nœud manager** (point de contrôle central du cluster)

▶ **Ajout de nœuds workers :**

```
docker swarm join --token <token> <ip_manager>:2377
```

- Commande fournie automatiquement à l'init (sécurité via token)

▶ **Vérification du cluster :**

```
docker node ls
```

- Affiche la liste des nœuds et leur rôle (Manager, Worker)

7.2.1. Gestion du cluster Swarm

► **Rôles :**

- **Manager** : orchestre le cluster, prend les décisions
- **Worker** : exécute les services, reçoit les instructions du manager

► **Avantages :**

- Simple à mettre en place (1 ligne).
- Inclus dans Docker, sans outil externe.

► ➡ Docker Swarm permet de **transformer plusieurs machines Docker en un système distribué**, sans ajouter de complexité externe.

7.2.2. Test du cluster avec un cas simple

Une fois le cluster Docker Swarm en place, on peut le **tester avec un service simple** pour vérifier son bon fonctionnement.

▶ **Déployer un service de test :**

```
docker service create --name webtest --replicas 3 -p 8080:80 nginx
```

- Crée un service webtest avec **3 répliques** du conteneur **nginx**
- Expose le port 80 du conteneur sur le port 8080 du cluster

▶ **Vérifier les services actifs :**

```
docker service ls
```

▶ **Voir la répartition des conteneurs :**

```
docker service ps webtest
```

- Affiche **sur quels nœuds** les répliques ont été déployés

7.2.2. Test du cluster avec un cas simple

▶ Accès à l'application :

- ▶ Naviguer vers `http://<IP_Manager>:8080`
- ▶ Chaque appel est **load balancé automatiquement** entre les réplicas

▶ Nettoyage :

```
docker service rm webtest
```

- ▶ 👉 Ce test simple valide que le cluster **répartit, exécute et expose correctement** les services.

7.2.3. Déploiement manuel sur le cluster Swarm

- ▶ Docker Swarm permet de déployer manuellement une **application conteneurisée complète**, en définissant chaque service à la main.
- ▶ **Exemple : déploiement d'une API Node.js + BDD**
 1. Créer un réseau interne partagé :

```
docker network create --driver overlay monreseau
```

2. Déployer la base de données :

```
docker service create --name db \
--network monreseau \
-e POSTGRES_PASSWORD=secret \
postgres
```

7.2.3. Déploiement manuel sur le cluster Swarm

3. Déployer l'API :

```
docker service create --name api \
--network monreseau \
-p 3000:3000 \
monimage-api
```

- ▶ L'image monimage-api doit avoir été **préalablement poussée** sur un registre accessible par tous les nœuds (Docker Hub, registre privé...).

4. Vérifier le déploiement :

```
docker service ls
docker service ps api
```

▶ Résultat :

- Les services tournent sur différents nœuds.
 - L'API peut accéder à la base via le nom db (grâce au réseau overlay).
 - L'application est exposée sur le port 3000 du manager.
- ▶ 👉 Ce mode permet de **déployer sans fichier YAML**, utile pour des tests, de l'apprentissage ou du déploiement ponctuel.

7.2.4. Déploiement de l'application microservices

- ▶ Pour une architecture composée de plusieurs services, **Docker Swarm permet un déploiement global via un fichier stack.**
- ▶ **Exemple de fichier stack.yml :**

```
services:  
  api:  
    image: monregistry/api:latest  
    ports:  
      - "3000:3000"  
    networks:  
      - backend  
  
  db:  
    image: postgres  
    environment:  
      POSTGRES_PASSWORD: secret  
    volumes:  
      - db-data:/var/lib/postgresql/data  
    networks:  
      - backend  
  
  networks:  
    backend:  
      driver: overlay  
  
  volumes:  
    db-data:
```

7.2.4. Déploiement de l'application microservices

▶ **Déploiement :**

```
docker stack deploy -c stack.yml monapp
```

▶ **Contrôle :**

```
docker stack services monapp
docker stack ps monapp
```

▶ **Avantages :**

- Tous les services sont **déployés, liés et exposés automatiquement**.
- **Réseaux, volumes, secrets** intégrés dans une seule définition.
- Adapté aux **applications réelles en production**.

▶👉 L'approche stack permet de **gérer une application complète de manière déclarative**, avec un minimum d'effort manuel.

7.2.5. Considérations additionnelles

L'utilisation de Docker Swarm en production nécessite de prendre en compte plusieurs aspects complémentaires.

► **Mise à l'échelle dynamique :**

```
docker service scale api=5
```

- Permet d'ajuster **à chaud** le nombre de réplicas d'un service.

► **Mise à jour continue (rolling update) :**

```
docker service update --image monregistry/api:v2 api
```

- Swarm met à jour les conteneurs **progressivement**, sans interruption du service.

7.2.5. Considérations additionnelles

▶ Secrets et configuration :

- Gestion native des **secrets Docker** (fichiers, mots de passe...)

```
docker secret create db_pass secrets/db_pass.txt
```

- Injectables uniquement dans les services autorisés.

▶ Résilience et auto-réparation :

- Si un nœud tombe, Swarm **reprogramme les services automatiquement** sur un autre nœud.
- L'équilibrage de charge est assuré **en natif** entre les réplicas.

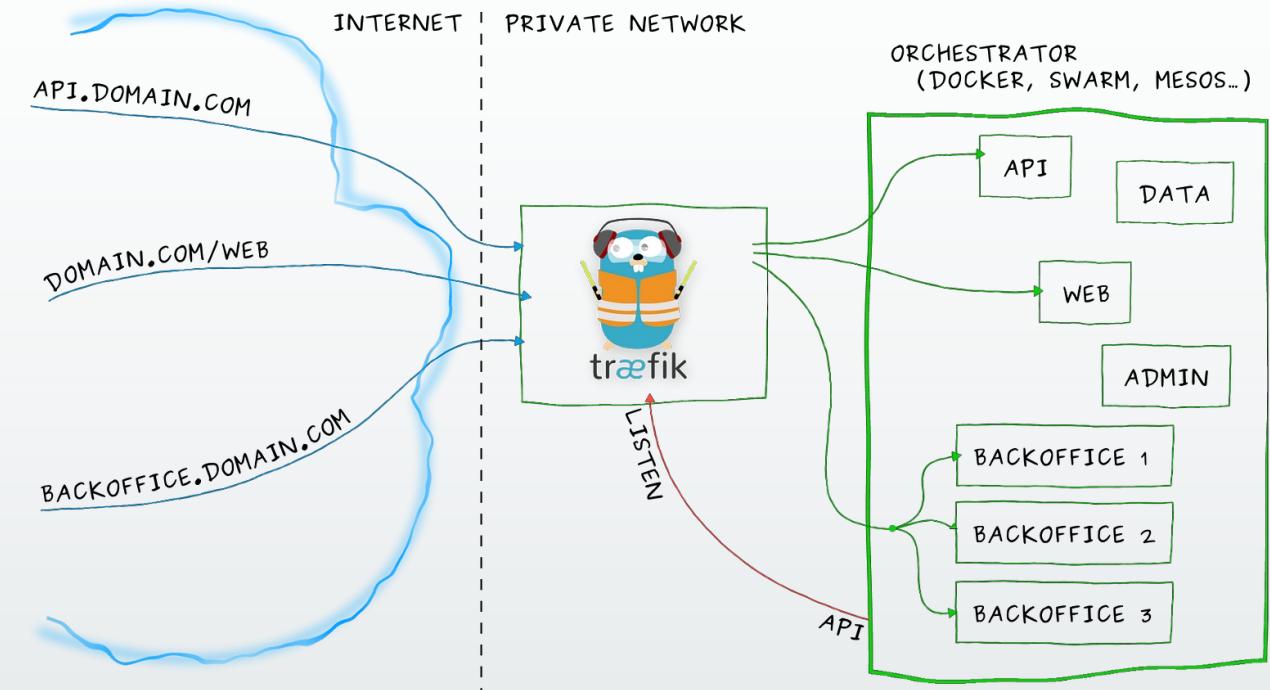
▶ Limites de Swarm :

- Moins d'outils et d'extensions que Kubernetes.
- Moins utilisé dans les grandes entreprises, mais **parfait pour des déploiements simples et efficaces**.

▶ ➡ Docker Swarm est un **excellent point d'entrée** dans l'orchestration, simple à prendre en main et suffisant pour de nombreux cas concrets.

7.3. Outils avancés d'exposition

- ▶ 7.3.1. Traefik
- ▶ 7.3.2. Caddy



7.3.1. Traefik

Traefik est un **reverse proxy** moderne conçu pour l'**orchestration de conteneurs**. Il détecte automatiquement les services Docker et les expose via HTTP/HTTPS.

► Principaux atouts :

-  Intégration native avec Docker, Swarm, Kubernetes...
-  Configuration **automatique** via les labels Docker
-  Support natif de **Let's Encrypt** pour les certificats TLS
-  Routage avancé : path-based, host-based, load balancing, middlewares...

► Fonctionnement :

- Traefik **observe les conteneurs Docker en temps réel**
 - Il configure les routes HTTP/S automatiquement à partir des **labels**
 - Il peut **gérer les certificats SSL** sans intervention manuelle
-  Traefik est un **outil central pour exposer des services en cluster**, de manière dynamique et sécurisée.

7.3.1. Traefik

► Exemple avec Docker Compose :

```
services:  
  traefik:  
    image: traefik:v2.10  
    ports:  
      - "80:80"  
      - "443:443"  
    volumes:  
      - /var/run/docker.sock:/var/run/docker.sock  
    command:  
      - --providers.docker  
      - --entrypoints.web.address=:80  
  
  whoami:  
    image: traefik/whoami  
    labels:  
      - "traefik.http.routers.whoami.rule=Host(`monsite.com`)"
```

7.3.2. Caddy

Caddy est un **serveur HTTP moderne** qui agit aussi comme **reverse proxy avec HTTPS automatique**, très simple à configurer.

▶ **Caractéristiques principales :**

- 🌐 Configuration ultra-simple avec un fichier Caddyfile
- 🔒 Gestion automatique des certificats **Let's Encrypt**
- 🧠 Redémarrage et rechargement **gracieux**
- 📦 Image officielle Docker disponible

▶ **Avantages :**

- Prise en main très rapide (moins verbeux que Traefik)
- Idéal pour les petits projets ou prototypage
- Support de HTTP/3, compression, cache, header policies...

▶ 👉 Caddy est un **excellent choix pour exposer facilement des conteneurs**, avec **HTTPS natif et sans configuration complexe**.

7.3.2. Caddy

- ▶ Exemple de Caddyfile :

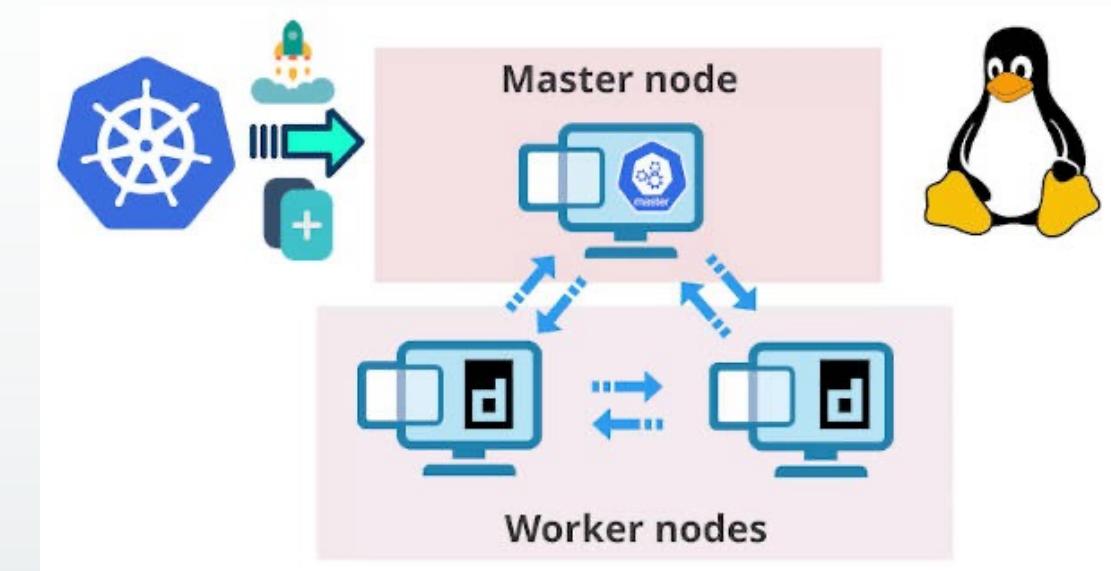
```
monsite.com {  
    reverse_proxy api:3000  
}
```

- ▶ Exemple docker-compose.yml :

```
services:  
    caddy:  
        image: caddy:2  
        ports:  
            - "80:80"  
            - "443:443"  
        volumes:  
            - ./Caddyfile:/etc/caddy/Caddyfile  
  
    api:  
        image: monapp/api
```

7.4. Introduction à Kubernetes

- ▶ 7.4.1. Positionnement
- ▶ 7.4.2. Comparaison avec Docker Swarm
- ▶ 7.4.3. Concepts
- ▶ 7.4.4. Installation
- ▶ 7.4.5. Premier déploiement par ligne de commande



7.4.1. Positionnement

Kubernetes est aujourd'hui la **plateforme d'orchestration de conteneurs la plus utilisée** dans le monde.

- ▶ **Origine :**
 - Crée par Google, désormais maintenu par la **Cloud Native Computing Foundation (CNCF)**
 - Inspiré de l'outil interne **Borg**
- ▶ **Objectif :**
 - Orchestrer le **déploiement, la mise à l'échelle, la résilience** et la **gestion du cycle de vie** des conteneurs.
 - Fonctionne dans des environnements **multi-nœuds**, sur site ou dans le cloud.
- ▶ **Cas d'usage :**
 - Architecture **microservices** à grande échelle
 - Déploiements **multi-environnements** (dev, staging, prod)
 - Intégration dans des **chaînes CI/CD avancées**
- ▶ **Support :**
 - Disponible sur **tous les grands clouds** (GKE, AKS, EKS)
 - Utilisable en local (Minikube, k3s, kind...)
- ▶ ➤ Kubernetes est aujourd'hui le **standard de facto** pour gérer des applications conteneurisées en production.

7.4.2. Comparaison avec Docker Swarm

► Vision générale :

Critère	Docker Swarm	Kubernetes
 Complexité initiale	Faible	Élevée
 Déploiement	Simple (inclus dans Docker)	Plus complexe (outils externes)
 Configuration	Déclarative mais limitée	Très fine et modulable
 Mises à jour / scaling	Basique (rolling updates)	Avancé (readiness, probes, autoscale)
 Secrets & RBAC	Basique	Très complet
 Écosystème	Limité	Très riche (Helm, Istio, Prometheus)
 Observabilité native	Faible	Écosystème complet

► Conclusion :

- **Swarm** : idéal pour des **déploiements simples et rapides**
- **Kubernetes** : adapté aux **environnements complexes, multi-équipes ou industriels**
-  Kubernetes est **plus puissant, mais plus exigeant**. Il offre **une granularité et un écosystème bien supérieurs** à Swarm.

7.4.3. Concepts

Kubernetes repose sur un **modèle déclaratif** : on décrit **l'état souhaité** de l'application, et le système s'assure qu'il est respecté.

► **Objets fondamentaux :**

- **Pod**
 - Unité minimale de déploiement (1 ou plusieurs conteneurs partagent réseau/volume)
- **Deployment**
 - Gère le cycle de vie des Pods (réplication, mise à jour, rollback)
- **Service**
 - Point d'accès stable à un ou plusieurs Pods (abstraction réseau + load balancing)
- **ConfigMap / Secret**
 - Gestion de la configuration externe et des données sensibles
- **Namespace**
 - Espace logique d'isolation au sein d'un cluster
- **Node**
 - Machine (physique ou VM) qui exécute des Pods (\simeq worker)

► **Architecture :**

- **Master / Control Plane** : prend les décisions (scheduling, surveillance)
 - **Worker Nodes** : exécutent les Pods
- ➡ Kubernetes est un **système déclaratif, modulaire et auto-réparant**, pensé pour le **déploiement d'applications distribuées**.

7.4.4. Installation

Kubernetes peut être installé de plusieurs façons, selon le **besoin (local, cloud, prod)** et le **niveau de complexité** souhaité.

► En local (pour apprendre et tester) :

- **Minikube**
► Démarrer un cluster local en VM ou conteneur
- **Kind (Kubernetes IN Docker)**
► Cluster léger basé sur Docker, idéal pour CI
- **k3s**
► Distribution Kubernetes allégée, très rapide à installer

► En production (cloud) :

- **GKE / EKS / AKS** (Google, AWS, Azure)
► Clusters managés avec mise à l'échelle automatique, intégration IAM, monitoring...
- **kubeadm**
► Méthode officielle pour installer un cluster manuellement
- **k3s**
► Distribution Kubernetes allégée, très rapide à installer

7.4.4. Installation

► **Outils nécessaires :**

- kubectl : ligne de commande pour interagir avec le cluster
- kubeconfig : fichier de configuration pour accéder à un cluster

►  L'installation dépend du **contexte d'utilisation** :

- local pour le développement
- managé ou automatisé pour la production

7.4.5. Premier déploiement par ligne de commande

Déployons une **application web simple (nginx)** dans un cluster Kubernetes à l'aide de la CLI kubectl.

▶ Étape 1 : création d'un déploiement

```
kubectl create deployment web --image=nginx
```

- → Crée un **Deployment** avec un **Pod** exécutant nginx

▶ Étape 2 : exposer le service

```
kubectl expose deployment web --type=NodePort --port=80
```

- → Crée un **Service** qui rend l'app accessible via un port du cluster

7.4.5. Premier déploiement par ligne de commande

- ▶ ◆ **Étape 3 : vérifier les ressources**

```
kubectl get deployments  
kubectl get pods  
kubectl get services
```

- ▶ ◆ **Étape 4 : accéder à l'application**

```
minikube service web
```

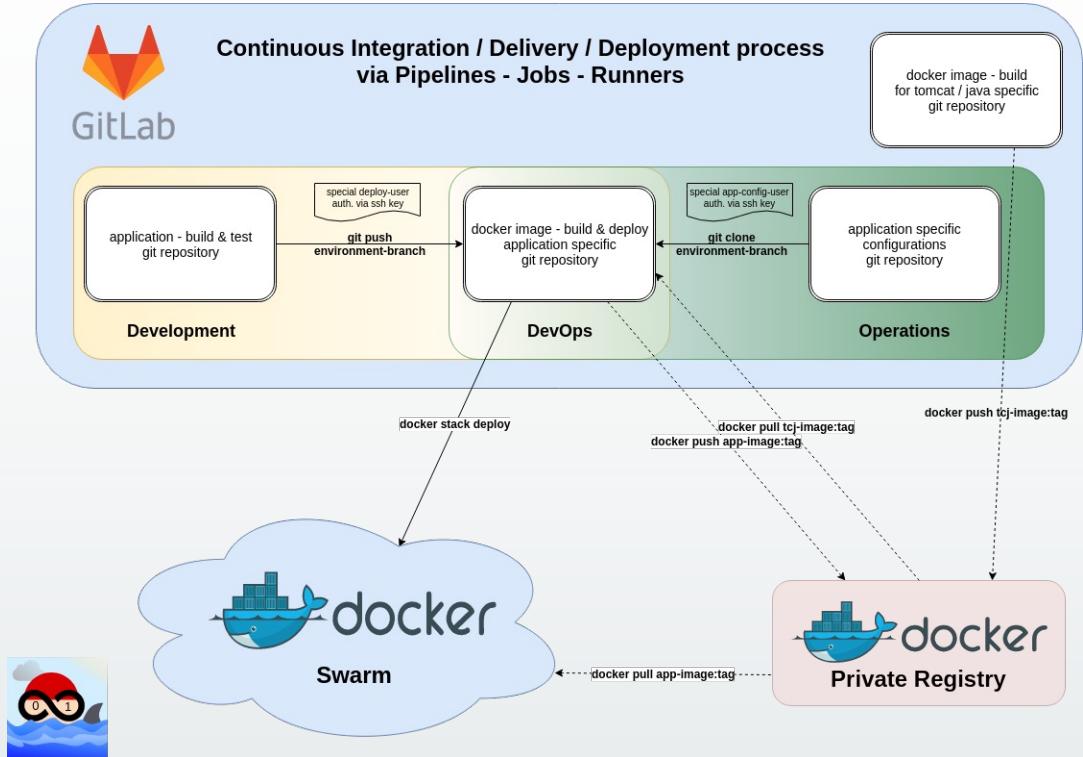
- Ouvre automatiquement l'URL dans le navigateur (via Minikube)
- ▶ ◆ **Nettoyage :**

```
kubectl delete service web  
kubectl delete deployment web
```

- ▶ ➤ Ce déploiement minimal permet de **comprendre le cycle complet** : image → pod → service → accès.

7.5. Intégration et déploiement continus

- ▶ 7.5.1. Approche
- ▶ 7.5.2. Intégration continue de l'application exemple
- ▶ 7.5.3. Déploiement de l'application microservices
- ▶ 7.5.4. Déploiement continu avec Azure DevOps
- ▶ 7.5.5. Fonctionnalités supplémentaires d'usine logicielle



7.5.1. Approche

L'**intégration continue (CI)** et le **déploiement continu (CD)** visent à **automatiser le cycle de vie applicatif**, du code source à la production.

- ▶ **Objectifs :**
 - **CI** : détecter rapidement les erreurs, tester automatiquement chaque modification.
 - **CD** : déployer automatiquement les versions validées sur des environnements de test, préprod, prod...
- ▶ **Rôle de Docker :**
 - Les conteneurs assurent :
 - Un environnement **constant et maîtrisé** pour la compilation et les tests.
 - Un **paquetage unique** pour l'application (image Docker).
 - Les pipelines CI/CD construisent, testent et **poussent l'image** dans un registre.
- ▶ **Exemple de workflow :**
 1. Push du code sur Git
 2. Build + tests automatisés (GitHub Actions, GitLab CI, Azure DevOps...)
 3. Création et push d'une image Docker
 4. Déploiement automatique dans un cluster (Swarm, Kubernetes...)
- ▶ ➤ **CI/CD + conteneurs = automatisation fiable, rapide et reproduitible** du processus de livraison logicielle.

7.5.2. Intégration continue de l'application exemple

Mettons en place une **pipeline CI** pour construire et tester automatiquement une application conteneurisée.

- ▶ **Exemple : pipeline GitHub Actions (Node.js + Docker)**
 - `.github/workflows/ci.yml`
- ▶ **Étapes couvertes :**
 1. **Checkout du code**
 2. **Build de l'image Docker**
 3. **Exécution des tests dans un conteneur isolé**
- ▶ **Bénéfices :**
 - Exécution **rapide, isolée et cohérente** à chaque push
 - **Pas de dépendances locales** nécessaires
 - Préparation du déploiement (image prête à être poussée)
- ▶  Cette étape assure que **chaque commit produit une version testée et conteneurisée**.

```
name: CI

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Build Docker image
        run: docker build -t monapp .

      - name: Run unit tests
        run: docker run --rm monapp npm test
```

7.5.3. Déploiement de l'application microservices

Dans une architecture microservices, le **déploiement automatisé** de plusieurs conteneurs est essentiel pour la cohérence, la rapidité et la stabilité.

▶ **Objectif :**

- Déployer automatiquement tous les services (API, base, cache, front...) à chaque mise à jour validée.

▶ **Approche possible :**

- **Avec Docker Compose (Swarm) :**

```
docker stack deploy -c stack.yml monapp
```

- ▶ Utilisé dans un pipeline (GitLab CI, GitHub Actions, etc.)

- **Avec Kubernetes (ex. via kubectl) :**

```
kubectl apply -f k8s/
```

- ▶ Le dossier k8s/ contient les fichiers YAML pour chaque service.

7.5.3. Déploiement de l'application microservices

► Exemple de pipeline (extrait) :

```
- name: Deploy to cluster
  run: |
    kubectl apply -f k8s/
```

► Bonnes pratiques :

- Versionner les fichiers YAML (stack.yml, deployment.yaml, etc.)
- Utiliser des **tags d'image explicites** (v1.2.3, sha256...) pour éviter les effets de cache
- Ajouter des **probes** (readiness/liveness) pour un déploiement sûr

► ➤ Un bon déploiement microservices doit être **automatisé, traçable et reproductible**.

7.5.4. Déploiement continu avec Azure DevOps

Azure DevOps permet de configurer une **chaîne CI/CD complète**, de la compilation au déploiement dans un cluster Docker ou Kubernetes.

► **Principe :**

- **Pipeline CI** : build + tests + image Docker
- **Pipeline CD** : déploiement automatisé via Docker ou kubectl

► **Exemple de pipeline YAML (simplifié) :**

```
trigger:  
  branches:  
    include: [ main ]  
  
stages:  
- stage: Build  
  jobs:  
    - job: BuildImage  
      steps:  
        - script: docker build -t monregistry.azurecr.io/app:${{Build.BuildId}} .  
        - script: docker push monregistry.azurecr.io/app:${{Build.BuildId}}  
  
- stage: Deploy  
  jobs:  
    - job: DeployToK8s  
      steps:  
        - script: kubectl apply -f k8s/
```

7.5.4. Déploiement continu avec Azure DevOps

▶ Intégrations possibles :

- Azure Container Registry (ACR)
- Azure Kubernetes Service (AKS)
- Secrets et connexions sécurisées via **Service Connections**

▶ Avantages :

-  Déploiement **automatisé et traçable** après chaque validation
 -  Gestion des identités et secrets centralisée
 -  Suivi des exécutions et rollback possibles
- ▶  Azure DevOps facilite la **mise en place de pipelines robustes** pour des projets conteneurisés à grande échelle.

7.5.5. Fonctionnalités supplémentaires d'usine logicielle

Une usine logicielle moderne va bien au-delà du simple build & deploy. Elle intègre des outils pour **améliorer la qualité, la sécurité et la traçabilité** du cycle de vie applicatif.

▶ Qualité & tests :

-  **Linting** automatique (code style, Dockerfile)
-  **Tests unitaires, d'intégration et E2E** automatisés
-  **Mesure de couverture** (SonarQube, Coveralls...)

▶ Sécurité :

-  **Scan d'images Docker** (ex : Trivy, Snyk, GitHub Advanced Security)
-  Vérification des **dépendances** (CVEs, licences)

▶ Observabilité :

-  Intégration de **logs** (ELK, Loki), **métriques** (Prometheus, Grafana), **traces** (Jaeger)
-  Intégration avec des outils de monitoring ou d'alerting

7.5.5. Fonctionnalités supplémentaires d'usine logicielle

► Collaboration & validation :

-  **Review automatique** de code via PRs
-  Déclenchement de pipelines **en fonction des branches ou des reviewers**
-  **Feature flags**, blue/green deployments, canary releases

► Gouvernance :

-  **Templates de pipelines**, règles de validation
-  **Historique des déploiements**, notifications

►  Une usine logicielle bien conçue est un **levier de qualité, de rapidité et de sécurité**, au cœur des pratiques DevOps.

7.6. Azure Container Instances

- ▶ 7.6.1. Principe
- ▶ 7.6.2. Préparation d'une image
- ▶ 7.6.3. Lancement du conteneur
- ▶ 7.6.4. Correction de l'erreur et relance
- ▶ 7.6.5. Coût et effort

