# Docker

**Qu’est-ce que Docker ?**

Docker est une plateforme qui permet de créer, déployer et exécuter des applications dans des conteneurs légers et portables. Un conteneur empaquette une application avec toutes ses dépendances (bibliothèques, outils, configuration) pour garantir qu’elle fonctionne de la même façon quel que soit l’environnement d’exécution.

**► Comment ça fonctionne (en bref)**

- **Images** : fichiers immuables définissant l’environnement et l’application (basés sur un Dockerfile).
- **Conteneurs** : instances en cours d’exécution d’une image ; isolés les uns des autres mais partagent le noyau du système hôte.
- **Docker Engine** : moteur (daemon) qui gère création, exécution et réseau des conteneurs.
- **Dockerfile** : script textuel décrivant comment construire une image (base image, commandes, copies, variables d’environnement, ports exposés).
- **Registry** : dépôt d’images (Docker Hub, registres privés). On "push" et "pull" des images.
- **Réseau et volumes** : mécanismes pour connecter conteneurs entre eux et pour persister des données hors du cycle de vie d’un conteneur.

**► Principaux atouts**

- **Portabilité** : même image, mêmes comportements sur postes dev, CI, serveurs, cloud.
- **Légèreté** : conteneurs plus légers que les machines virtuelles car ils partagent le noyau hôte.
- **Démarrage rapide** : instanciation quasi instantanée comparée aux VM.
- **Isolation** : séparation des dépendances et environnements entre applications.
- **Reproductibilité** : builds déclaratifs (Dockerfile) garantissant la même image à chaque build.

- **Écosystème riche** : outils (Docker Compose, Docker Swarm, Kubernetes intégration), images publiques.
- **CI/CD friendly** : s'intègre facilement dans chaînes d'intégration et déploiement continu.

► **Limitations et points d'attention**

- **Pas une VM complète** : partage du noyau, donc isolation moindre sur certains aspects comparée aux VM.
- **Sécurité** : nécessite bonnes pratiques (images minimales, scans de vulnérabilités, gestion des privilèges).
- **Stockage/State** : nécessite gestion explicite des volumes pour données persistantes.
- **Complexité réseau & orchestration** : à grande échelle, nécessite orchestration (Kubernetes, Swarm) et compréhension des concepts réseau.

► **Cas d'usage typiques**

- Packaging d'applications (microservices).
- Tests reproductibles et environnements de développement identiques.
- Déploiement continu et scaling d'applications.
- Isolation d'outils et versions (ex. différentes versions de base de données).

► **Commandes de base (exemples)**

- Construire une image : docker build -t monimage:tag .
- Lister les images : docker images
- Lancer un conteneur : docker run -d --name moncontenant -p 8080:80 monimage:tag
- Lister conteneurs : docker ps (ou docker ps -a)
- Supprimer conteneur : docker rm <id> ; image : docker rmi <image>
- Volumes : docker volume create, monter avec -v /host/path:/container/path.

## ► Qu’est-ce que Docker Compose ?

Docker Compose est un outil qui permet de définir et exécuter des applications multi-conteneurs à l’aide d’un fichier de configuration YAML (généralement docker-compose.yml). Il orchestre le lancement, le réseau et les volumes de plusieurs services définis ensemble, facilitant le développement local et les environnements reproductibles.

## ► Comment ça fonctionne (en bref)

- **Fichier docker-compose.yml** : décrit les services, images/build, volumes, réseaux, variables d’environnement, ports, dépendances (depends_on), commandes d’entrée (command), etc.
- **Services** : chaque entrée représente un conteneur (ou un ensemble répliqué) — ex. web, db, cache.
- **Réseaux et volumes déclarés** : Compose crée des réseaux et volumes nommés pour interconnecter et persister les données entre services.
- **Commandes principales** : docker compose up (lance et crée les services), docker compose down (arrête et supprime les ressources), docker compose build, docker compose ps, docker compose logs.
- **Mode de fonctionnement** : Compose traduit la configuration YAML en appels Docker Engine pour construire images, créer réseaux/volumes et démarrer conteneurs dans le bon ordre.

## ► Principaux atouts

- **Simplicité** : regroupe tous les services d’une application dans un seul fichier lisible.
- **Reproductibilité** : facilite la reproduction d’un environnement complet (dev, test).
- **Isolation & configuration centralisée** : variables d’environnement, commandes d’initialisation, liens réseau explicites.
- **Développement local efficace** : hot-reload, montages de volumes pour code source, logs consolidés.
- **Orchestration légère** : gère dépendances de démarrage (depends_on) et scaling basique (replicas avec Compose V2/override).

- **Intégration CI** : facilite tests d'intégration en levant l'ensemble des services requis.

### ► Limitations et points d'attention

- **Pas un orchestrateur de production complet** : pour du scaling massif, tolérance aux pannes et scheduling avancé, privilégier Kubernetes ou Swarm.
- **Gestion des secrets** : Compose supporte des fichiers d'environnement/secrets mais nécessite prudence pour la sécurité en prod.
- **Ordre de démarrage** : depends_on n'attend pas la disponibilité applicative (juste le démarrage du conteneur) — prévoir des checks readiness ou scripts d'attente.
- **Réseaux par défaut** : Compose crée un réseau par projet ; attention aux noms et collisions entre projets.



```yaml
version: "3.9"

services:
  web:
    build: ./web
    ports:
      - "8080:80"
    environment:
      - DATABASE_URL=postgres://user:pass@db:5432/app
    depends_on:
      - db
      - redis
    volumes:
      - ./web:/app

  db:
    image: postgres:15
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=app
    volumes:
      - db-data:/var/lib/postgresql/data

  redis:
    image: redis:7
    volumes:
      - redis-data:/data

volumes:
  db-data:
  redis-data:
```

Commandes utiles :

- Lancer en avant-plan : docker compose up
- Lancer en détaché : docker compose up -d
- Arrêter et supprimer ressources : docker compose down --volumes --remove-orphans
- Reconstruire : docker compose up --build
- Voir logs : docker compose logs -f
- Exécuter une commande dans un service : docker compose exec web sh

**► Bonnes pratiques**

- Utiliser des images légères et fixes (tags, pas :latest).
- Versionner le docker-compose.yml avec le code.
- Externaliser les secrets (Docker secrets, vault, variables d'environnement CI).
- Ajouter des checks de readiness pour dépendances critiques.
- Utiliser des profils (profiles) pour activer des services selon l'environnement (dev vs prod).
- Nettoyer volumes non utilisés régulièrement.

**► Qu'est-ce que Portainer ?**

Portainer est une interface graphique web légère pour gérer des environnements de conteneurs Docker, Docker Compose, Docker Swarm et Kubernetes. Elle facilite l'administration (création, surveillance, déploiement) sans passer exclusivement par la ligne de commande.

**► Comment ça fonctionne (en bref)**

- Portainer s'exécute comme un conteneur et se connecte au Docker Engine (via socket /var/run/docker.sock ou endpoint distant) ou à un cluster Kubernetes.
- L'UI expose des écrans pour gérer images, conteneurs, volumes, réseaux, stacks (Compose), secrets, utilisateurs et rôles.
- Les actions via l'UI déclenchent des appels au moteur Docker/Kubernetes pour créer, modifier ou supprimer les ressources.

- Supporte l’import et le déploiement de fichiers docker-compose.yml en tant que "stacks".

► **Principaux atouts**

- **Simplicité d’utilisation** : interface visuelle pour utilisateurs non experts CLI.
- **Déploiement de stacks** : importer et lancer des docker-compose.yml en quelques clics.
- **Gestion multi-endpoints** : centralise plusieurs hôtes Docker/Kubernetes depuis une seule console.
- **Contrôle d’accès** : gestion des utilisateurs, équipes et rôles (RBAC) en version Business.
- **Surveillance basique** : vue des logs, métriques et des statuts des conteneurs/services.
- **Opérations courantes** : créer containers, images, volumes, backups, snapshots, exec dans un conteneur via UI.
- **Extensible** : templates d’application pour déployer rapidement des stacks courants.

► **Limitations et points d’attention**

- **Sécurité du socket Docker** : monter /var/run/docker.sock dans Portainer donne des droits élevés — suivre bonnes pratiques (utilisateurs, TLS, endpoints distants).
- **Fonctionnalités avancées** : pour orchestration à grande échelle ou politiques complexes, Kubernetes natif + outils dédiés restent nécessaires.
- **Version Business payante** : certaines fonctions avancées (authentification externe poussée, RBAC étendu, scoping d’équipes) sont en licence commerciale.
- **Surface d’attaque** : exposer l’UI sans protection (auth, TLS, pare-feu) peut être risqué.

► **Cas d'usage typiques**

- Console d'administration pour petites équipes DevOps.
- Enseignement / démonstrations et développeurs qui préfèrent l'UI.
- Déploiement et gestion rapide de stacks Compose en dev/test.
- Centralisation de plusieurs environnements Docker/Kubernetes.

**Déploiement rapide (exemple)**

- Lancer Portainer (Docker) :

```bash
docker volume create portainer_data
docker run -d -p 9000:9000 \
  --name portainer \
  --restart=always \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v portainer_data:/data \
  portainer/portainer-ce:latest
```

- Accéder à http://:9000 pour créer le premier utilisateur et ajouter des endpoints (local ou distants via agent).

► **Fonctionnalités pratiques à connaître**

- **Stacks** : importer un docker-compose.yml, éditer et redéployer via UI.
- **Templates** : déployer rapidement des applications préconfigurées.
- **Agent** : déployer Portainer Agent sur des hôtes distants pour gestion sécurisée multi-hôtes.
- **Snapshots & backups** : exporter configuration et volumes critiques.
- **Logs & console** : voir logs en temps réel et ouvrir un shell dans un conteneur.
- **Integrations** : LDAP/OAuth (selon édition), webhooks pour CI.

► **Bonnes pratiques**

- Protéger l'accès : HTTPS, authentification forte, limiter accès réseau.
- Éviter de laisser /var/run/docker.sock exposé à des services non sûrs ; utiliser Portainer Agent pour séparation.
- Versionner stacks et templates dans VCS; utiliser CI pour déploiements reproductibles.
- Surveiller et restreindre comptes utilisateurs et rôles.