# Docker : virtualiser des applications avec des conteneurs

Comprendre Docker : conteneurs, images, écosystème et mise en pratique sous Windows

## 🎯 Objectifs du cours

À la fin de ce cours, tu seras capable de :

| Savoir/Faire | Objectif |
|---|---|
| **Reproduire** | Installer Docker Desktop sous Windows et exécuter des commandes de base |
| **Comprendre** | Expliquer la différence entre un conteneur Docker et une machine virtuelle |
| **Comprendre** | Identifier les composants de l'écosystème Docker (image, conteneur, Hub, Compose) |
| **Appliquer** | Lancer des services courants (serveur web, base de données) via Docker |
| **Généraliser** | Reconnaître les situations où Docker est la bonne solution |

## 🔑 5 notions-clés

1. **Conteneur** — Une application et tout ce dont elle a besoin, isolée du reste du système
2. **Image** — Le "moule" à partir duquel on crée un conteneur (lecture seule, réutilisable)
3. **Dockerfile** — Le fichier de recette qui décrit comment construire une image
4. **Docker Hub** — Le dépôt public où l'on télécharge et partage des images
5. **Docker Compose** — L'outil pour orchestrer plusieurs conteneurs qui travaillent ensemble

## 🔥 Le problème que Docker résout

### "Ça marche sur ma machine..."

Tu as peut-être déjà vécu cette situation :

Tu développes une application PHP sur ton PC sous Windows avec XAMPP.
Tu l'envoies à un camarade qui a une version différente de PHP.
Ça ne fonctionne plus. 😤

Ou encore :

L'application tourne parfaitement en développement.
On la déploie sur le serveur de production (Ubuntu).
Surprise : erreur au démarrage. Une bibliothèque manque. Une version ne correspond pas.

Ce problème s'appelle le problème de l'**environnement**.

Une application ne tourne pas seule : elle dépend d'un système d'exploitation, d'une version de langage (PHP 8.1 vs 8.3), de bibliothèques, de variables d'environnement, de fichiers de configuration...

© 2025 Mr. Lambrechts

**Docker résout ce problème en encapsulant l'application ET son environnement dans une boîte autonome : le conteneur.**

Un conteneur Docker tourne de la même façon sur :

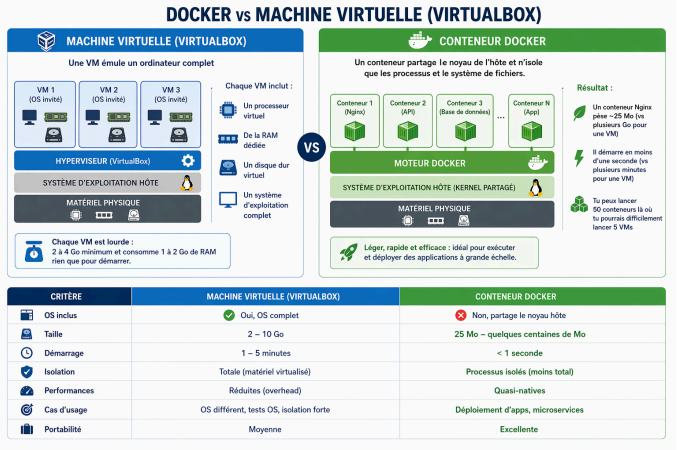
- Ton PC Windows
- Le PC de ton camarade (macOS)
- Un serveur Linux en production
- Un service cloud (AWS, Azure, Google Cloud)

💡 **Analogie** : Pense à un conteneur Docker comme à un aquarium auto-suffisant. Le poisson (l'application) vit dans son eau (son environnement). Peu importe où tu poses l'aquarium, les conditions intérieures restent identiques.

---

# ⚖️ Docker vs Machine Virtuelle (VirtualBox)

C'est la question que tout le monde se pose. Docker et VirtualBox permettent tous les deux de faire tourner des environnements isolés, mais leur approche est radicalement différente.




## La machine virtuelle (VirtualBox)

Une VM émule un **ordinateur complet** :

- Un processeur virtuel
- De la RAM dédiée
- Un disque dur virtuel
- Un **système d'exploitation complet** (Windows, Ubuntu, etc.)

VirtualBox installe un "hyperviseur" qui fait croire à chaque VM qu'elle possède son propre matériel. Chaque VM est **lourde** : un Ubuntu dans VirtualBox pèse 2 à 4 Go minimum et consomme 1 à 2 Go de RAM rien que pour démarrer.

## Le conteneur Docker

Un conteneur **ne virtualise pas le matériel**. Il partage le **noyau (kernel) du système d'exploitation hôte** et n'isole que les processus et le système de fichiers.

Résultat :

- Un conteneur Nginx pèse **~25 Mo** (vs plusieurs Go pour une VM)
- Il démarre en **moins d'une seconde** (vs plusieurs minutes pour une VM)
- Tu peux lancer **50 conteneurs** là où tu pourrais difficilement lancer 5 VMs

## Tableau comparatif

| Critère | Machine Virtuelle (VirtualBox) | Conteneur Docker |
|---|---|---|
| **OS inclus** | ✅ Oui, OS complet | ❌ Non, partage le noyau hôte |
| **Taille** | 2–10 Go | 25 Mo – quelques centaines de Mo |
| **Démarrage** | 1–5 minutes | < 1 seconde |
| **Isolation** | Totale (matériel virtualisé) | Processus isolés (moins total) |
| **Performances** | Réduites (overhead) | Quasi-natives |
| **Cas d'usage** | OS différent, tests OS, isolation forte | Déploiement d'apps, microservices |
| **Portabilité** | Moyenne | Excellente |

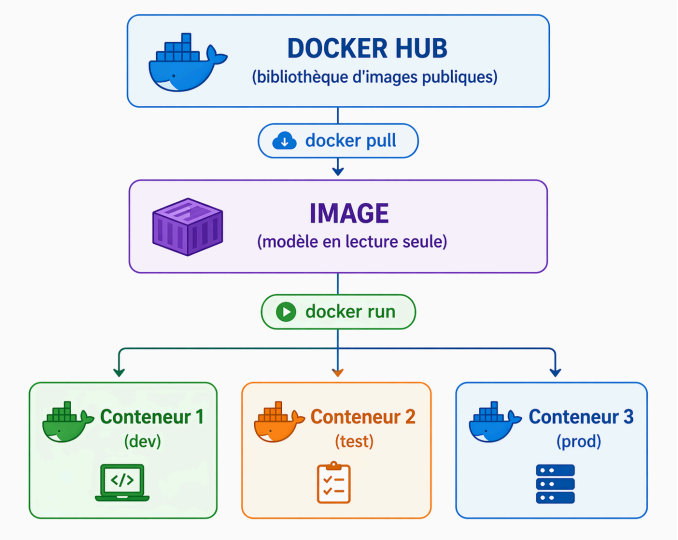
## Quand utiliser quoi ?

- **VirtualBox / VMware** → Tu as besoin d'un vrai autre système d'exploitation (ex: tester une app Windows sur macOS, lab réseau avec plusieurs OS)
- **Docker** → Tu veux déployer une application rapidement, de façon reproductible, sans te soucier de l'environnement

🧠 **Note** : Sous Windows, Docker Desktop utilise en arrière-plan une petite VM Linux légère (WSL 2 — Windows Subsystem for Linux) pour faire tourner le moteur Docker. L'utilisateur n'a pas à s'en occuper, c'est transparent.

---

# 🧩 L'écosystème Docker




## 1. Le Docker Engine (le moteur)

C'est le cœur du système. Il tourne en arrière-plan sur ta machine (on appelle ça un **daemon**). Il est responsable de :

- Télécharger des images
- Créer et gérer les conteneurs
- Gérer les réseaux virtuels entre conteneurs
- Gérer les volumes (stockage persistant)

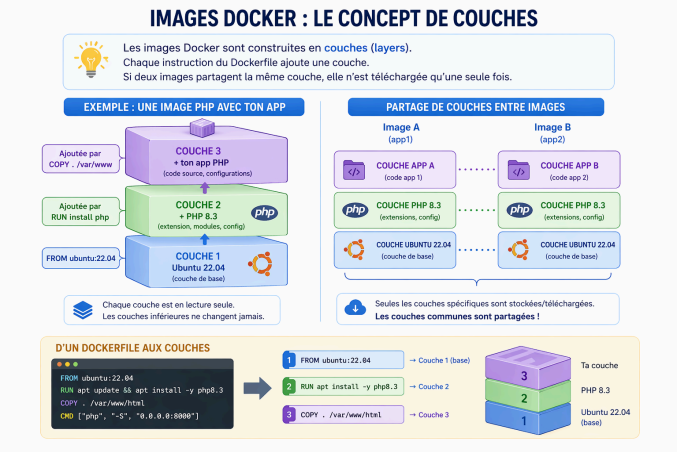
# 2. L'Image Docker

Une image est un **modèle en lecture seule**. Elle contient :

- Un mini-système de fichiers
- L'application et ses dépendances
- Les instructions de démarrage

Les images sont construites en **couches** (layers). Chaque instruction du Dockerfile ajoute une couche. Si deux images partagent la même couche de base (ex: Ubuntu), elle n't est téléchargée qu'une seule fois.

```
Image Ubuntu 22.04  ← couche de base
    └── + PHP 8.3   ← couche ajoutée
        └── + ton app PHP  ← ta couche
```




# 3. Le Conteneur

Un conteneur est une **instance en cours d'exécution** d'une image. C'est la "boîte vivante".

- Une image ⟶ on peut en créer **autant de conteneurs qu'on veut**
- Chaque conteneur est **isolé** : ses modifications ne touchent pas l'image source
- Si on supprime un conteneur, ses données sont **perdues** (sauf si on utilise un volume)

**Analogie code** : Une image = une classe. Un conteneur = une instance (objet).

```
Image MySQL  →  conteneur "db-dev"    (base de données de développement)
             →  conteneur "db-test"   (base de données de test)
             →  conteneur "db-prod"   (base de données de production)
```

# 4. Le Dockerfile

C'est le fichier texte qui contient la **recette** pour construire une image personnalisée.

```dockerfile
# On part d'une image de base officielle
FROM php:8.3-apache

# On copie nos fichiers dans le conteneur
COPY ./src /var/www/html

# On expose le port 80
EXPOSE 80
```

Avec la commande `docker build`, Docker lit ce fichier et crée ton image.

# 5. Docker Hub

C'est le **registre officiel** d'images Docker. L'adresse : hub.docker.com

On y trouve :

- Des images **officielles** maintenues par les éditeurs (MySQL, PHP, Nginx, Node.js, Ubuntu...)
- Des images **communautaires** partagées par des développeurs
- Ton propre compte pour héberger tes images privées ou publiques

`docker pull nginx` → télécharge l'image officielle Nginx depuis Docker Hub.

# 6. Docker Compose

Quand une application a besoin de **plusieurs services** qui coopèrent (ex: une app PHP + une base de données MySQL + phpMyAdmin), Docker Compose permet de les définir dans **un seul fichier** `docker-compose.yml` et de tout démarrer avec une seule commande.

```yaml
# docker-compose.yml — exemple simplifié
services:
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: secret

  phpmyadmin:
    image: phpmyadmin
    ports:
      - "8080:80"
    depends_on:
      - db
```

`docker compose up` → démarre tous les services définis.

# 7. Docker Desktop

C'est l'**application graphique** pour Windows et macOS. Elle installe et gère automatiquement :

- Le moteur Docker
- WSL 2 (sous Windows)
- Une interface visuelle pour voir les conteneurs, images, volumes
- Docker Compose

---

# 💼 Cas d'utilisation courants

## En développement (le plus fréquent pour vous)

| Besoin | Solution Docker |
|---|---|
| Base de données MySQL sans l'installer | `docker run mysql` |
| Serveur Apache/Nginx pour tester | `docker run nginx` |
| Tester une app sur une version PHP différente | Image `php:8.1` vs `php:8.3` |
| Environnement identique pour toute l'équipe | `docker-compose.yml` partagé via Git |
| Isoler des projets qui ont des dépendances conflictuelles | Un conteneur par projet |

## En production / DevOps

- **Microservices** : chaque composant d'une grosse application tourne dans son propre conteneur
- **CI/CD** : les pipelines de tests (GitHub Actions, GitLab CI) utilisent Docker pour créer des environnements de test propres à chaque build
- **Déploiement reproductible** : "ça marche sur ma machine" devient "ça marchera partout"
- **Mise à l'échelle** : avec Kubernetes, on peut lancer automatiquement 10 copies d'un conteneur si le trafic augmente

## Pour un technicien informatique

- Lancer un **serveur DNS** (Pi-hole) dans un conteneur sur un Raspberry Pi
- Installer **Portainer** (interface web pour gérer Docker) en une commande
- Déployer un **serveur de fichiers**, un **Wiki**, un **serveur de monitoring** sans "salir" le système hôte
- Tester des configurations réseau ou serveur en toute sécurité

---

# 🏭 Docker est-il prêt pour la production ?

**Oui, absolument.** Docker est utilisé en production par des entreprises comme Netflix, Spotify, Airbnb, et des milliers d'autres.

Quelques chiffres :

- Plus de **13 millions** d'images sur Docker Hub
- Utilisé dans la majorité des pipelines CI/CD modernes
- Base de **Kubernetes**, le standard de l'orchestration de conteneurs

## Limites à connaître

| Limite | Explication |
|---|---|
| **Stateless par défaut** | Les données d'un conteneur disparaissent à sa suppression → utiliser des **volumes** |
| **Sécurité** | Un conteneur mal configuré peut devenir un vecteur d'attaque (ne pas lancer les conteneurs en `root`) |
| **Complexité à grande échelle** | Des dizaines de conteneurs → nécessite **Kubernetes** ou **Docker Swarm** |
| **Performance réseau** | Une légère latence supplémentaire sur les communications inter-conteneurs |

## Bonne pratique : l'écoresponsabilité 🌱

Docker permet de **mieux utiliser les ressources** d'un serveur. Là où il fallait auparavant un serveur physique par application (beaucoup de gaspillage CPU/RAM), on peut maintenant en faire tourner des dizaines sur la même machine. **Moins de serveurs = moins de consommation électrique = moins d'impact environnemental.** C'est l'un des arguments écologiques du Cloud-native.

---

# 💻 Installation sous Windows

## Prérequis

- Windows 10 version 21H2 ou Windows 11 (64 bits)
- WSL 2 activé (Windows Subsystem for Linux)
- Virtualisation activée dans le BIOS (VT-x / AMD-V)

# Étape 1 : Activer WSL 2

Ouvre PowerShell **en administrateur** et exécute :

```
wsl --install
```

Redémarre le PC si demandé.

# Étape 2 : Télécharger Docker Desktop

Télécharge l'installeur sur : **docker.com/products/docker-desktop**

Lance l'installeur et coche **"Use WSL 2 instead of Hyper-V"**.

# Étape 3 : Vérifier l'installation

Ouvre un terminal (PowerShell ou CMD) et tape :

```
docker --version
docker run hello-world
```

Si tu vois `Hello from Docker!`, c'est bon ! 🎉

# Interface Docker Desktop

Une fois lancé, Docker Desktop te propose :

- **Containers** → voir et gérer les conteneurs actifs/arrêtés
- **Images** → voir les images téléchargées localement
- **Volumes** → gérer le stockage persistant
- **Dev Environments** → environnements de développement préconfigurés

---

# ⌨️ Commandes essentielles en ligne de commande

## Gestion des images

```
# Télécharger une image depuis Docker Hub
docker pull nginx

# Lister les images locales
docker images

# Supprimer une image
docker rmi nginx
```

## Gestion des conteneurs

```
# Lancer un conteneur (télécharge l'image si besoin)
docker run nginx
```

```
# Lancer en arrière-plan (-d = detached)
docker run -d nginx

# Lancer avec un nom et un port (hôte:conteneur)
docker run -d --name mon-serveur -p 8080:80 nginx

# Lister les conteneurs actifs
docker ps

# Lister TOUS les conteneurs (actifs + arrêtés)
docker ps -a

# Arrêter un conteneur
docker stop mon-serveur

# Supprimer un conteneur
docker rm mon-serveur

# Arrêter ET supprimer automatiquement (--rm)
docker run --rm -p 8080:80 nginx
```

## Inspecter et déboguer

```
# Voir les logs d'un conteneur
docker logs mon-serveur

# Logs en temps réel (suivi)
docker logs -f mon-serveur

# Ouvrir un terminal dans un conteneur actif
docker exec -it mon-serveur bash

# Inspecter les détails d'un conteneur
docker inspect mon-serveur
```

## Nettoyage

```
# Supprimer tous les conteneurs arrêtés
docker container prune

# Supprimer toutes les images inutilisées
docker image prune

# Nettoyage global (images, conteneurs, réseaux non utilisés)
docker system prune
```

## Docker Compose

```
# Démarrer tous les services (en arrière-plan)
docker compose up -d
```

```
3
4  # Arrêter tous les services
5  docker compose down
6
7  # Voir les logs de tous les services
8  docker compose logs -f
9
10 # Reconstruire les images et redémarrer
11 docker compose up -d --build
```

---

# 🧪 Exercice 1 — Ton premier serveur web avec Nginx

**Objectif** : Lancer un serveur web Nginx en une commande et y accéder depuis ton navigateur.

**Niveau Bloom** : Reproduire / Appliquer

## Contexte

Nginx est un serveur web très utilisé en production. Normalement, l'installer sous Windows est compliqué. Avec Docker, c'est une seule commande.

## Instructions

**1.** Ouvre un terminal PowerShell.

**2.** Lance le conteneur :

```
1  docker run -d --name mon-nginx -p 8080:80 nginx
```

Décortiquons cette commande :

| Argument | Signification |
|---|---|
| `run` | Crée et démarre un conteneur |
| `-d` | En arrière-plan (detached) |
| `--name mon-nginx` | Donne un nom au conteneur |
| `-p 8080:80` | Redirige le port 8080 de ta machine vers le port 80 du conteneur |
| `nginx` | Le nom de l'image à utiliser (téléchargée depuis Docker Hub) |

**3.** Ouvre ton navigateur et va sur : **http://localhost:8080**

Tu devrais voir la page de bienvenue de Nginx ✅

**4.** Vérifie que le conteneur tourne :

```
1  docker ps
```

**5.** Lis les logs du serveur :

```
1  docker logs mon-nginx
```

Recharge la page du navigateur et regarde les logs se mettre à jour.

**6.** Arrête et supprime le conteneur :

```
docker stop mon-nginx
docker rm mon-nginx
```

## Questions de réflexion

- Que se passe-t-il si tu essaies d'accéder à http://localhost:8080 après avoir arrêté le conteneur ?
- Que se passerait-il si tu lançais deux fois la même commande avec le même `--name` ?
- Quel est le port par défaut d'un serveur web HTTP ?

---

# 🧪 Exercice 2 — Une base de données MySQL sans installation

**Objectif** : Lancer un serveur MySQL complet via Docker, s'y connecter et créer une base de données.

**Niveau Bloom** : Appliquer / Comprendre

## Contexte

Tu connais MySQL depuis tes cours. L'installer sur un PC prend du temps et "pollue" le système. Avec Docker, tu lances une instance MySQL propre en quelques secondes, que tu peux supprimer après utilisation.

## Instructions

**1.** Lance MySQL dans un conteneur :

```
docker run -d \
  --name mon-mysql \
  -p 3306:3306 \
  -e MYSQL_ROOT_PASSWORD=motdepasse \
  -e MYSQL_DATABASE=mabase \
  mysql:8.0
```

**Note Windows** : Sur PowerShell, remplace les `\` par des backticks `` ` `` pour les retours à la ligne, ou écris tout sur une seule ligne.

Décortiquons les nouveaux arguments :

| Argument | Signification |
|---|---|
| `-p 3306:3306` | Expose le port MySQL standard |
| `-e MYSQL_ROOT_PASSWORD=motdepasse` | Variable d'environnement : mot de passe root |
| `-e MYSQL_DATABASE=mabase` | Crée automatiquement une base "mabase" au démarrage |
| `mysql:8.0` | Image MySQL version 8.0 |

**2.** Attends quelques secondes que MySQL démarre. Surveille les logs :

```
docker logs -f mon-mysql
```

Quand tu vois `ready for connections`, appuie sur `Ctrl+C` pour quitter les logs.

**3.** Connecte-toi à MySQL depuis l'intérieur du conteneur :

```
docker exec -it mon-mysql mysql -u root -pmotdepasse
```

Tu es maintenant dans le shell MySQL.

**4.** Teste quelques commandes :

```
SHOW DATABASES;
USE mabase;
CREATE TABLE eleves (id INT AUTO_INCREMENT PRIMARY KEY, nom VARCHAR(100));
INSERT INTO eleves (nom) VALUES ('Alice'), ('Bob');
SELECT * FROM eleves;
EXIT;
```

**5.** Connecte-toi aussi depuis **MySQL Workbench** ou **HeidiSQL** sur ta machine hôte :

- Hôte : `127.0.0.1`
- Port : `3306`
- Utilisateur : `root`
- Mot de passe : `motdepasse`

**6.** Supprime le conteneur :

```
docker stop mon-mysql
docker rm mon-mysql
```

## ⚠️ Observation importante

Recrée le conteneur et retourne voir la table `eleves`. Elle a **disparu**. C'est normal : sans **volume**, les données ne sont pas persistées.

Pour garder les données, on utilise un volume :

```
docker run -d \
  --name mon-mysql \
  -p 3306:3306 \
  -e MYSQL_ROOT_PASSWORD=motdepasse \
  -v mysql_data:/var/lib/mysql \
  mysql:8.0
```

Le `-v mysql_data:/var/lib/mysql` crée un volume nommé `mysql_data` qui survit à la suppression du conteneur.

---

# 🧪 Exercice 3 — MySQL + phpMyAdmin avec Docker Compose

**Objectif** : Lancer deux services qui coopèrent (MySQL + phpMyAdmin) avec un seul fichier de configuration.

**Niveau Bloom** : Appliquer / Généraliser

# Contexte

Dans la vraie vie, les applications ont souvent besoin de plusieurs services. Docker Compose permet de les définir ensemble, de gérer leur démarrage dans l'ordre, et de les faire communiquer sur un réseau interne.

# Instructions

**1.** Crée un dossier `docker-mysql-lab` sur ton Bureau.

**2.** Dans ce dossier, crée un fichier `docker-compose.yml` :

```yaml
services:

  # Service 1 : la base de données MySQL
  db:
    image: mysql:8.0
    container_name: lab_mysql
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: monprojet
      MYSQL_USER: dev
      MYSQL_PASSWORD: devpass
    volumes:
      - mysql_data:/var/lib/mysql
    networks:
      - lab_network

  # Service 2 : phpMyAdmin pour gérer la base via navigateur
  phpmyadmin:
    image: phpmyadmin:latest
    container_name: lab_phpmyadmin
    restart: unless-stopped
    ports:
      - "8080:80"
    environment:
      PMA_HOST: db          # Le nom du service MySQL dans le réseau Docker
      PMA_PORT: 3306
    depends_on:
      - db
    networks:
      - lab_network

# Déclaration du volume persistant
volumes:
  mysql_data:

# Déclaration du réseau interne
networks:
  lab_network:
```

**3.** Ouvre un terminal dans ce dossier et lance tout :

```
docker compose up -d
```

Docker va :

- Télécharger les images manquantes
- Créer le réseau `lab_network`
- Créer le volume `mysql_data`
- Démarrer MySQL en premier (car `depends_on`)
- Démarrer phpMyAdmin ensuite

**4.** Ouvre **http://localhost:8080** dans ton navigateur.

Connecte-toi avec :

- Serveur : `db`
- Utilisateur : `root`
- Mot de passe : `secret`

**5.** Crée une table, insère des données.

**6.** Arrête les services :

```
docker compose down
```

Relance avec `docker compose up -d`. Tes données sont **toujours là** grâce au volume.

**7.** Pour tout supprimer y compris les volumes :

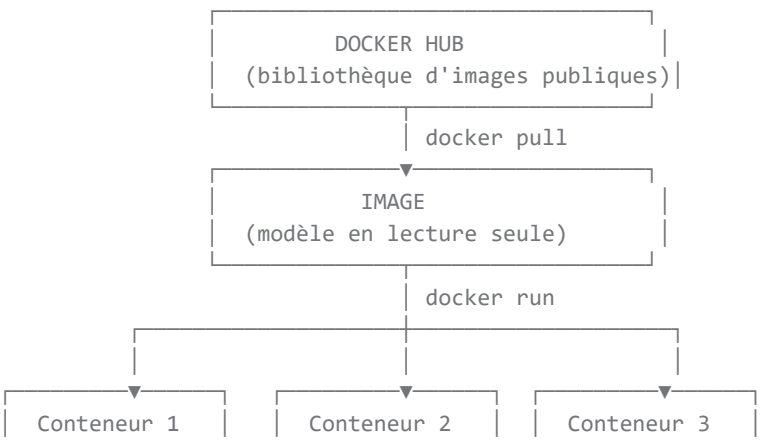
```
docker compose down -v
```

# Pour aller plus loin

- Ajoute un troisième service : une application PHP qui se connecte à la base MySQL
- Modifie le port de phpMyAdmin pour utiliser `8888:80` et relance
- Explore l'onglet **Containers** de Docker Desktop pour visualiser les deux services

---

# 📊 Synthèse : ce qu'il faut retenir




| Commande | Action |
|---|---|
| `docker pull <image>` | Télécharger une image |
| `docker run -d -p X:Y <image>` | Lancer un conteneur |
| `docker ps` | Voir les conteneurs actifs |
| `docker stop <nom>` | Arrêter un conteneur |
| `docker rm <nom>` | Supprimer un conteneur |
| `docker logs <nom>` | Voir les journaux |
| `docker exec -it <nom> bash` | Entrer dans un conteneur |
| `docker compose up -d` | Démarrer les services Compose |
| `docker compose down` | Arrêter les services Compose |

---

# 🌱 Écoresponsabilité numérique

Docker favorise la **densification des serveurs** : là où il fallait 10 serveurs physiques, on peut héberger 10 applications isolées sur un seul serveur bien dimensionné. Cela réduit la consommation électrique, la production de chaleur et les déchets électroniques. C'est l'un des fondements du mouvement **Green IT** dans les datacenters modernes.

---

© 2025 Mr. Lambrechts