# 🐳 Docker — Volumes, réseaux et redémarrage

# Objectifs

À la fin de cette page, tu seras capable de :

- Expliquer pourquoi les données d'un conteneur sont perdues par défaut
- Créer et utiliser un volume nommé pour persister des données
- Utiliser un bind mount pour développer en temps réel
- Connecter plusieurs conteneurs via un réseau Docker
- Configurer le redémarrage automatique d'un conteneur

# 5 notions-clés

1. **Volume nommé** — Stockage géré par Docker, persiste même si le conteneur est supprimé
2. **Bind mount** — Lien direct entre un dossier de ta machine et un dossier dans le conteneur
3. **Réseau bridge** — Réseau virtuel privé créé par Docker où les conteneurs peuvent se parler
4. **Réseau nommé** — Réseau bridge personnalisé : les conteneurs se contactent par leur **nom de service**
5. **Restart policy** — Règle qui définit si et quand Docker relance automatiquement un conteneur

# Les volumes

## Le problème : un conteneur est éphémère

Quand tu crées un fichier ou modifies une base de données à l'intérieur d'un conteneur, ces données vivent **dans le conteneur**. Dès que tu le supprimes, tout disparaît.

```
docker run -d --name ma-db mysql:8.0 -e MYSQL_ROOT_PASSWORD=test
# ... tu crées des tables, tu insères des données ...
docker rm -f ma-db
docker run -d --name ma-db mysql:8.0 -e MYSQL_ROOT_PASSWORD=test
# → Toutes les données ont disparu ❌
```

La solution, c'est le **volume**.

## Volume nommé — pour les données importantes

Un volume nommé est un espace de stockage géré par Docker, qui **vit indépendamment** des conteneurs.

```
# Créer un volume manuellement
docker volume create mes-donnees-mysql

```

```
# Lancer MySQL avec ce volume
docker run -d \
  --name ma-db \
  -e MYSQL_ROOT_PASSWORD=secret \
  -v mes-donnees-mysql:/var/lib/mysql \
  mysql:8.0
```

La syntaxe `-v` : `nom_du_volume:chemin_dans_le_conteneur`

```
# Lister les volumes existants
docker volume ls

# Inspecter un volume (voir son emplacement réel sur le disque)
docker volume inspect mes-donnees-mysql

# Supprimer un volume
docker volume rm mes-donnees-mysql

# Supprimer tous les volumes non utilisés
docker volume prune
```

💡 **Dans Docker Compose**, si tu déclares un volume sans préciser qu'il est externe, Docker le crée automatiquement. Le nom sera préfixé du nom du dossier projet : `monprojet_mes-donnees-mysql`.

**Exemple Compose avec volume :**

```
services:
  db:
    image: mysql:8.0
    volumes:
      - mysql_data:/var/lib/mysql   # volume nommé

volumes:
  mysql_data:   # déclaration du volume
```

---

# Bind mount — pour le développement

Un bind mount crée un **lien direct** entre un dossier de ta machine et un chemin dans le conteneur. Toute modification faite d'un côté est immédiatement visible de l'autre.

C'est idéal quand tu développes : tu modifies ton code dans VS Code, le conteneur voit le changement en temps réel.

```
# Syntaxe : chemin_absolu_hôte:chemin_dans_conteneur
docker run -d \
  --name mon-site \
  -p 8080:80 \
  -v "C:\Users\toi\projets\mon-site:/usr/share/nginx/html" \
  nginx:1.25
```

Sur Windows, Docker Desktop accepte les chemins Windows avec des guillemets, ou la notation WSL (`/mnt/c/Users/...`).

**Exemple Compose avec bind mount :**

```yaml
services:
  web:
    image: php:8.3-apache
    ports:
      - "8080:80"
    volumes:
      - ./src:/var/www/html   # chemin relatif au fichier compose.yml
```

## Résumé : volume nommé vs bind mount

| Critère | Volume nommé | Bind mount |
|---|---|---|
| Géré par Docker | ✅ Oui | ❌ Non (c'est ton dossier) |
| Chemin à connaître | ❌ Non | ✅ Oui |
| Idéal pour | Données de production (BDD, uploads) | Développement (code source) |
| Portabilité | ✅ Excellente | ⚠️ Dépend du chemin hôte |
| Performances | ✅ Optimisées | ⚠️ Légèrement plus lentes sous Windows |

# Les réseaux

## Pourquoi les réseaux ?

Par défaut, deux conteneurs lancés séparément **ne peuvent pas se parler**. Si tu veux que ton conteneur PHP joigne ton conteneur MySQL, ils doivent être sur le **même réseau Docker**.

## Le réseau bridge par défaut

Quand tu lances un conteneur sans préciser de réseau, il rejoint le réseau `bridge` par défaut de Docker. Les conteneurs sur ce réseau peuvent se contacter **uniquement par leur adresse IP** (qui change à chaque redémarrage).

```bash
# Voir les réseaux existants
docker network ls

# Inspecter le réseau bridge par défaut
docker network inspect bridge
```

## Les réseaux nommés — la bonne pratique

Avec un réseau **nommé**, les conteneurs peuvent se contacter **par leur nom** (ou par le nom du service dans Compose). Plus besoin de connaître les IPs.

```bash
# Créer un réseau nommé
docker network create mon-reseau

# Lancer MySQL sur ce réseau
docker run -d \
  --name ma-db \
  --network mon-reseau \
  -e MYSQL_ROOT_PASSWORD=secret \
```

```
  mysql:8.0

# Lancer phpMyAdmin sur le même réseau
docker run -d \
  --name mon-pma \
  --network mon-reseau \
  -p 8080:80 \
  -e PMA_HOST=ma-db \
  phpmyadmin:latest
```

Ici, `PMA_HOST=ma-db` fonctionne parce que `ma-db` est le **nom du conteneur MySQL** sur le même réseau. Docker fait la résolution de nom automatiquement.

**Avec Docker Compose**, les réseaux nommés sont créés automatiquement. Tous les services d'un même fichier Compose partagent un réseau par défaut et peuvent se contacter par leur nom de service.

```yaml
services:
  db:
    image: mysql:8.0
    # accessible depuis les autres services sous le nom "db"

  app:
    image: php:8.3-apache
    # peut contacter MySQL avec le hostname "db"
    # ex: mysqli_connect("db", "root", "secret", "mabase")
```

# Les types de réseaux Docker

| Type | Description | Utilisation typique |
|---|---|---|
| `bridge` | Réseau virtuel privé (défaut) | Projets locaux, développement |
| `host` | Partage l'interface réseau de la machine hôte | Cas avancés sous Linux uniquement |
| `none` | Aucun réseau | Tests d'isolation, sécurité |
| Réseau nommé (`bridge` custom) | Comme bridge mais avec résolution de noms | ✅ Recommandé pour tous les projets |

# Commandes réseau utiles

```
# Créer un réseau
docker network create mon-reseau

# Lister les réseaux
docker network ls

# Connecter un conteneur existant à un réseau
docker network connect mon-reseau mon-conteneur

# Déconnecter un conteneur d'un réseau
docker network disconnect mon-reseau mon-conteneur

# Inspecter un réseau (voir quels conteneurs y sont connectés)
docker network inspect mon-reseau

```

```bash
# Supprimer un réseau (doit être vide)
docker network rm mon-reseau
```

# Le redémarrage automatique

## Pourquoi ?

Si tu héberges un service (serveur web, base de données, outil de monitoring), tu veux qu'il soit toujours disponible, même après :

- Un redémarrage du PC / serveur
- Un redémarrage du service Docker lui-même
- Un crash inattendu de l'application

## Les 4 politiques de redémarrage

```bash
docker run --restart [politique] ...
```

| Politique | Comportement |
|---|---|
| `no` | Ne redémarre jamais automatiquement (comportement par défaut) |
| `always` | Redémarre toujours, même si arrêté manuellement avec `docker stop` |
| `unless-stopped` | Redémarre automatiquement, **sauf** si arrêté manuellement ✅ |
| `on-failure` | Redémarre uniquement si le conteneur se termine avec une erreur |
| `on-failure:3` | Comme `on-failure`, mais maximum 3 tentatives |

## Quelle politique choisir ?

- **`unless-stopped`** → La plus utilisée pour les services permanents. Elle redémarre après un reboot du système, mais respecte quand tu l'arrêtes volontairement.
- **`always`** → Utile en production quand le service **ne doit jamais être arrêté**, même manuellement.
- **`on-failure`** → Pour les tâches/scripts qui peuvent échouer et qu'on veut retenter.
- **`no`** → Pour les conteneurs de test ou les jobs ponctuels.

## Exemples

```bash
# Serveur web permanent
docker run -d \
  --name site-web \
  --restart unless-stopped \
  -p 80:80 \
  nginx:1.25

# Service critique (ne doit jamais être arrêté)
docker run -d \
  --name surveillance \
  --restart always \
  mon-outil-monitoring

# Script qui peut échouer (3 tentatives max)
docker run \
  --restart on-failure:3 \
  mon-script-backup
```

## Dans Docker Compose

```yaml
services:
  web:
    image: nginx:1.25
    restart: unless-stopped    # ← même syntaxe, sans tirets

  db:
    image: mysql:8.0
    restart: unless-stopped
```

## Modifier la politique d'un conteneur existant

```bash
# Changer la politique de redémarrage d'un conteneur déjà créé
docker update --restart unless-stopped mon-conteneur

# Vérifier la politique actuelle
docker inspect mon-conteneur | grep RestartPolicy
```

---

# Tout ensemble — exemple complet

Voici un exemple qui combine volumes, réseau nommé et restart policy :

```yaml
services:

  db:
    image: mysql:8.0
    container_name: projet_db
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: ${DB_ROOT_PASSWORD}
      MYSQL_DATABASE: ${DB_NAME}
    volumes:
      - db_data:/var/lib/mysql        # volume nommé pour les données
    networks:
      - projet_net                    # réseau nommé

  app:
    image: php:8.3-apache
    container_name: projet_app
    restart: unless-stopped
    ports:
      - "8080:80"
    volumes:
      - ./src:/var/www/html           # bind mount pour le dev
    networks:
      - projet_net                    # même réseau = peut contacter "db"
    depends_on:
      - db
```

```
27
28  volumes:
29    db_data:
30
31  networks:
32    projet_net:
```

Variables dans `.env` :

```
1  DB_ROOT_PASSWORD=supersecret
2  DB_NAME=monprojet
```

Démarrage :

```
1  docker compose up -d
```