Cours complet sur Docker Compose

1. Introduction : Les limites de Docker seul

Docker nous permet de **créer**, **exécuter** et **gérer** des conteneurs facilement.
MAIS, lorsqu'on commence à construire une application plus complexe, on se heurte à plusieurs limites :

Limite	Explication
Applications multi-conteneurs	Une application réelle a souvent plusieurs services : une base de données, un backend, un frontend, etc. Gérer tout à la main avec `docker run` devient très lourd.
Reproductibilité	Difficile de garantir que tout le monde démarre les bons conteneurs avec les bonnes configurations.
Configuration complexe	À force de multiplier les `docker run`, la gestion des ports, réseaux, volumes devient compliquée.
Orchestration minimale	Impossible de lancer et arrêter tous les conteneurs ensemble de manière élégante.

---

2. La solution : Docker Compose

👉 **Docker Compose** est un **outil officiel de Docker** qui permet de **définir et de gérer** plusieurs conteneurs **dans un seul fichier** (`docker-compose.yml`).

Son but :

- Décrire toute l'architecture d'une application dans un **fichier unique**.
- Lancer toute l'infrastructure d'un seul coup (`docker compose up`).
- Faciliter la gestion de réseaux, volumes et environnements multi-conteneurs.

---

3. Pourquoi Docker Compose est important ?

- **Facilite le développement** : tu peux cloner un repo, faire `docker compose up` et ton environnement est prêt.
- **Versionné avec le code** : le `docker-compose.yml` est dans le repo Git.
- **Simule un environnement de production** : tu reproduis presque exactement ta prod en local.
- **Standardisé** : tout le monde utilise les mêmes services, versions et configurations.

---

4. Comment fonctionne Docker Compose ?

- Tu **définis** tes services (backend, base de données, cache...) dans un fichier **YAML** (`docker-compose.yml`).
- Tu **lances** tous les services avec une seule commande :
- `docker compose up`
- Tu peux **arrêter** tout avec :
- `docker compose down`
- Docker Compose crée :
    - Un **réseau par défaut** pour que les conteneurs puissent communiquer.
    - Les **volumes** pour persister les données.
    - Les **images** s'il faut les construire.

---

5. Structure d'un fichier docker-compose.yml

Voici un exemple simple :

```
version: '3.8' # version de la syntaxe Compose

services:
  db:
    image: postgres:14
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
      POSTGRES_DB: mydb
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - app-network

  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgres://myuser:mypassword@db:5432/mydb
    depends_on:
      - db
    networks:
      - app-network

volumes:
  db-data:

networks:
  app-network:
```

✍ Détail des parties :

Partie	Description
`version`	La version du format de fichier docker-compose.

Partie	Description
`services`	Liste des conteneurs/services que tu veux lancer.
`volumes`	Définir des volumes pour persister les données.
`networks`	Définir des réseaux personnalisés.

---

6. Bonnes pratiques avec Docker Compose

☑ **Utiliser un `.env` pour les variables sensibles** (mots de passe, utilisateurs...).

☑ **Ne jamais coder les secrets directement** dans `docker-compose.yml`.

☑ **Spécifier les dépendances** avec `depends_on`, mais savoir que cela ne garantit pas que l'application est prête (utiliser des scripts de "wait-for" si besoin).

☑ **Utiliser des volumes** pour persister les données importantes.

☑ **Structurer le projet** avec un dossier par service si tu as du build (exemple : `/backend`, `/frontend`...).

☑ **Utiliser des noms de réseau explicites** pour éviter des conflits si plusieurs projets utilisent Compose sur la même machine.

---

7. Inconvénients de Docker Compose

Inconvénient	Détail
Pas fait pour la production à grande échelle	Compose est parfait pour dev, tests, staging, mais pour des dizaines/centaines de conteneurs, on utilise d'autres outils.
Pas de réplication automatique	Un service tourne une fois, pas de scaling auto comme avec Kubernetes ou Swarm.
Gestion basique des erreurs	Pas de reprise automatique en cas de crash de conteneur (sauf si tu configures des `restart: always`).

---

8. Différence entre Docker Compose, Swarm et Kubernetes

Critère	Docker Compose	Docker Swarm	Kubernetes
Cible	Développement local	Orchestration simple (production légère)	Orchestration avancée (production complexe)
Installation	Super simple (juste Docker)	Simple (déjà intégré dans Docker)	Plus complexe

Critère	Docker Compose	Docker Swarm	Kubernetes
Scalabilité	Limitée	Bonne (réplication facile)	Excellente
HA (High Availability)	Non	Oui (simplement)	Oui (ultra-robuste)
Réseaux/Secrets avancés	Limité	Oui	Très avancé
Mises à jour sans coupure	Non	Oui	Oui
Utilisation	Projet local / petits environnements	Production légère ou moyenne	Production lourde (Google, Amazon, Netflix...)

🔥 Résumé visuel :

```
Docker seul => Trop simple pour projet sérieux
↓
Docker Compose => Idéal pour dev multi-conteneurs
↓
Docker Swarm => Premier pas en orchestration
↓
Kubernetes => Orchestration avancée pour production géante
```
