



Docker & Ansible 2025



Une formation présentée par Andromed.

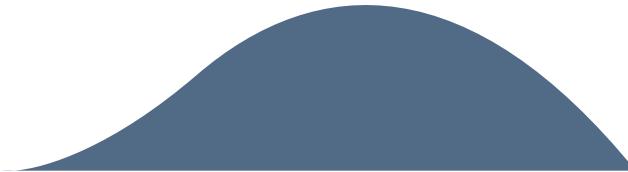


Appuyez sur espace pour la page suivante →

Jimmylan Surquin

Fondateur  [Andromed](#)

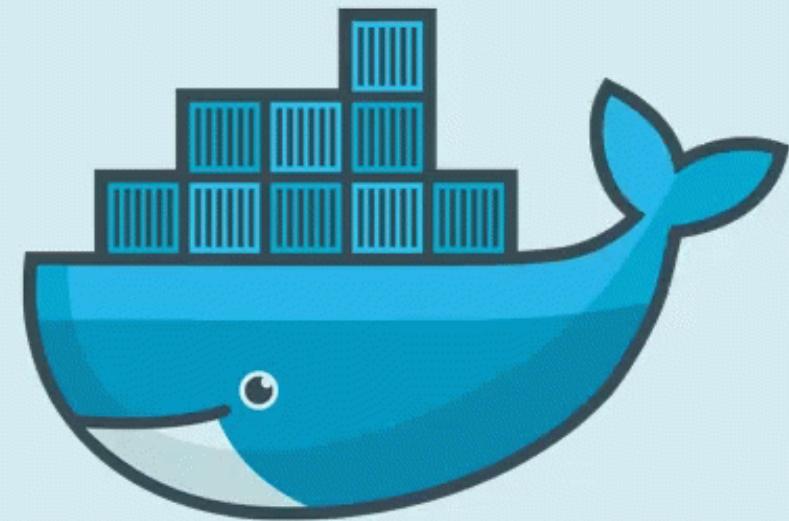
- Lille, France 
- Création de contenu sur  [jimmylansrq](#)
- Blog & Portfolio [jimmylan.fr](#)



DISCLAIMER



Dans cette formation nous allons voir les commandes principales de Docker et Ansible en 2025.



DOCKER & ANSIBLE

SOMMAIRE DOCKER



Formation pratique en 3 jours

CI/CD & Microservices

Virtualisation vs conteneurisation

Introduction & Définitions Docker

Premier contact Docker

Le CLI Docker

Réseaux & Volumes

Exercices CLI Docker

Docker Compose

Exercices Compose

Dockerfile et images

Exercices Dockerfile

Fondamentaux Ansible

Exercices Ansible

QCM Ansible

PROGRAMME 3 JOURS

JUL
17

Structure pédagogique optimisée

Jour 1 - Fondamentaux Docker

- CI/CD et microservices
- Virtualisation vs conteneurisation
- Introduction & définitions Docker
- CLI Docker et commandes essentielles
- Exercices CLI pratiques (3 niveaux)
- Premier contact pratique

Jour 2 - Docker avancé

- Dockerfile et bonnes pratiques
- Exercices Dockerfile (3 niveaux)
- Réseaux et communication
- Volumes et persistance
- Docker Compose multi-containers
- Exercices Compose (3 niveaux)

Jour 3 - Ansible et intégration

- Introduction à Ansible
- Playbooks et inventaires
- Modules essentiels
- Ansible + Docker
- Projet final

Comprendre le CI/CD & les micro-services

Comprendre le CI/CD



Définition et contexte

Le **CI/CD** (Continuous Integration / Continuous Deployment) est devenu l'épine dorsale du développement logiciel moderne. Cette méthodologie permet d'automatiser entièrement le cycle de vie d'une application, de la phase de développement jusqu'à la mise en production.



[Revenir au sommaire](#)

Pourquoi le CI/CD ? 🎯

Pourquoi le CI/CD est-il essentiel en 2025 ?

- **Réduction des erreurs** : Détection précoce des bugs et problèmes d'intégration
- **Déploiements plus fréquents** : Livraison continue de nouvelles fonctionnalités
- **Feedback rapide** : Retour immédiat sur la qualité du code
- **Collaboration améliorée** : Synchronisation automatique entre les équipes



[Revenir au sommaire](#)

CI/CD & IA en 2025 🤖

L'impact de l'intelligence artificielle sur le CI/CD et la conteneurisation

En 2025, l'IA révolutionne la façon dont on met en place des pipelines CI/CD et des environnements Docker. De nombreux outils assistés par l'IA permettent de gagner un temps précieux et d'automatiser des tâches complexes.



[Revenir au sommaire](#)

Métaphore automobile du CI/CD 🚗

Intégration Continue (CI) 🔧

Imaginez que vous dirigez une usine automobile moderne produisant 500 voitures par jour. L'**Intégration Continue** consiste à :

- **Contrôler chaque pièce** avant de l'installer sur la chaîne de montage
- **Tester chaque assemblage** au fur et à mesure (moteur, freins, électronique)
- **Valider la qualité** à chaque poste de travail, pas seulement à la fin
- **Déetecter immédiatement** si une pièce est défectueuse ou incompatible



Déploiement Continu (CD)

Une fois que tous les composants sont validés et l'assemblage perfectionné, le **Déploiement Continu** permet de :

- **Finaliser automatiquement** la voiture sans intervention manuelle
- **Livrer immédiatement** dès que tous les tests sont passés
- **Maintenir la qualité** constante pour chaque véhicule produit
- **Répéter le processus** de manière fiable sur toute la chaîne de production



Génération automatique de Dockerfile 🐳⚡

- **Outils IA** : Des plateformes comme [Docker AI](#), [GitHub Copilot](#), ou [ChatGPT](#) génèrent des Dockerfile optimisés à partir de simples descriptions de projet.
- Liste non exhaustive d'outils IA pour le CI/CD :

Pour générer du code , à utiliser avec la précaution de comprendre à 100% ce que vous générez.

- [Cursor](#)
- [Claude](#)
- [Gemini](#)
- [Grok](#)
- [Perplexity](#)



Pour simplifier Docker, il existe des outils qui génèrent des Dockerfile optimisés à partir de simples descriptions de projet.

- [nixpacks](#) : Génère un Dockerfile à partir de la description d'un projet.
- [coolify](#) : Gère en grosse partie lui même l'intégration continue et le déploiement.
- [railway](#) : Créez vous même votre Dockerfile, il vous propose d'heberger gratuitement votre container en quelques clics.



Plateformes de déploiement simplifié 🚀

- [render](#) : Créez vous même votre Dockerfile, il vous propose d'heberger gratuitement votre container en quelques clics.
- [netlify](#) : Gère en grosse partie lui même l'intégration continue et le déploiement.
- [vercel](#) : Gère en grosse partie lui même l'intégration continue et le déploiement.
- **Avantages :**
 - Génération instantanée de Dockerfile adaptés à votre stack
 - Suggestions de bonnes pratiques de sécurité et d'optimisation
 - Détection automatique des dépendances et des ports à exposer



Plateformes de déploiement simplifié 🚀

- **Coolify** : Plateforme open-source qui permet de déployer des applications Docker, Node.js, PHP, etc. en quelques clics, avec gestion automatique des certificats SSL, des bases de données et du scaling.
- **Netlify** : Déploiement ultra-rapide de sites statiques et d'APIs serverless, intégration continue native, preview automatiques.
- **Vercel** : Déploiement instantané d'applications front-end et back-end, preview automatiques pour chaque pull request.
- **Render, Railway, Fly.io** : Alternatives modernes pour déployer des containers ou des microservices sans gestion manuelle de l'infrastructure.



[Revenir au sommaire](#)

L'IA pour automatiser et sécuriser le pipeline

- **Détection automatique de failles** dans les images Docker grâce à des outils comme Snyk, Trivy, ou les scanners intégrés aux plateformes CI/CD modernes.
- **Optimisation des builds** : L'IA propose des étapes de build plus rapides, détecte les redondances et suggère des améliorations.
- **Monitoring intelligent** : Analyse prédictive des incidents, alertes proactives, et recommandations de scaling automatique.



[Revenir au sommaire](#)

En résumé

L'IA et les plateformes modernes transforment le CI/CD et la conteneurisation en 2025 :

- Génération de Dockerfile et de pipelines en quelques secondes
- Déploiement simplifié sur des plateformes comme Coolify, Netlify, Vercel, etc.
- Sécurité et optimisation automatisées
- Plus de temps pour l'innovation, moins pour la configuration manuelle !



[Revenir au sommaire](#)

Les Pipelines CI/CD en Pratique 🛠

Qu'est-ce qu'un pipeline ?

Un **pipeline CI/CD** est une chaîne automatisée d'étapes qui transforme votre code source en application déployée et opérationnelle.



[Revenir au sommaire](#)

Schéma d'un pipeline



Phases essentielles

Les phases essentielles d'un pipeline moderne

- **Source** : Récupération du code depuis le repository (Git)
- **Build** : Compilation et construction de l'application
- **Test** : Exécution des tests unitaires, d'intégration et de sécurité
- **Package** : Création des artefacts déployables (containers Docker)
- **Deploy** : Déploiement automatisé vers les environnements cibles



Outils et technologies de pipeline 2025



Plateformes CI/CD populaires

- **GitHub Actions** : Intégration native avec GitHub, YAML-based
- **GitLab CI/CD** : Solution complète intégrée à GitLab
- **Jenkins** : Solution open-source extensible et mature
- **Azure DevOps** : Écosystème Microsoft complet
- **CircleCI** : Pipeline cloud optimisé pour la vitesse



[Revenir au sommaire](#)

Les microservices

[Revenir au sommaire](#)

Architecture Microservices

Définition et philosophie

L'**architecture microservices** consiste à décomposer une application monolithique en services indépendants, chacun ayant une responsabilité spécifique et pouvant être développé, déployé et mis à l'échelle de manière autonome.



Métaphore du supermarché

Métaphore du supermarché

Imaginez un supermarché moderne où chaque rayon fonctionne comme un microservice :

- **Rayon fruits & légumes** : Gestion des produits frais, stocks, prix
- **Boulangerie** : Production, cuisson, vente de produits de boulangerie
- **Caisse** : Traitement des paiements, fidélité client
- **Stock** : Approvisionnement, inventaire, logistique



Indépendance des rayons



Chaque rayon peut :

- Fonctionner indépendamment des autres rayons
- Avoir ses propres employés et processus
- Être mis à jour sans affecter les autres
- Communiquer avec les autres via des interfaces définies



[Revenir au sommaire](#)

Avantages des Microservices



Bénéfices techniques

- **Scalabilité granulaire** : Mise à l'échelle service par service selon les besoins
- **Technologie polyglotte** : Chaque service peut utiliser la technologie la plus adaptée
- **Isolation des pannes** : Une défaillance n'affecte pas l'ensemble du système
- **Déploiements indépendants** : Livraison continue sans impact sur les autres services



[Revenir au sommaire](#)

Bénéfices organisationnels 👤

Bénéfices organisationnels

- **Équipes autonomes** : Chaque équipe possède et maintient ses services
- **Développement parallèle** : Accélération du développement global
- **Responsabilité claire** : Ownership et accountability bien définis
- **Innovation technique** : Liberté d'expérimenter sur des services isolés



[Revenir au sommaire](#)

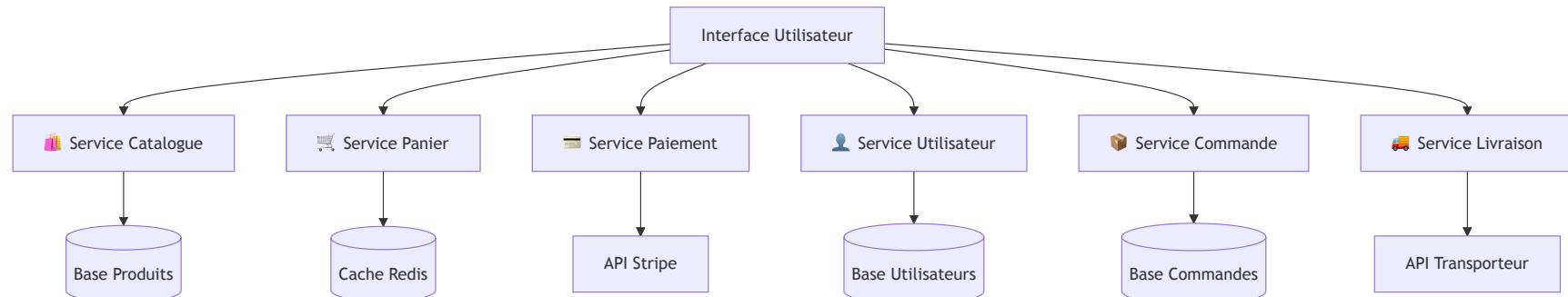
Exemple Concret : E-commerce 🛍

Architecture microservices d'une plateforme e-commerce



[Revenir au sommaire](#)

Architecture e-commerce



Service Catalogue Produits 🛍️

🛍️ Service Catalogue Produits

- **Responsabilité** : Gestion des produits, catégories, prix, promotions
- **Technologie** : Node.js + MongoDB pour flexibilité des données
- **API** : REST pour consultation, GraphQL pour recherche complexe



[Revenir au sommaire](#)

Service Panier

Service Panier

- **Responsabilité** : Gestion des paniers clients, calculs de totaux
- **Technologie** : Redis pour performance et session management
- **API** : WebSocket pour mise à jour temps réel



[Revenir au sommaire](#)

Service Paiement



Service Paiement

- **Responsabilité** : Traitement sécurisé des transactions
- **Technologie** : Java Spring Boot pour robustesse et sécurité
- **Intégrations** : Stripe, PayPal, Apple Pay, Google Pay



[Revenir au sommaire](#)

Communication entre microservices



Patterns de communication

- **Synchrone** : API REST/HTTP pour les opérations immédiates
- **Asynchrone** : Message queues (RabbitMQ, Kafka) pour les tâches en arrière-plan
- **Event-driven** : Publication/souscription pour les notifications système



Relation avec Docker

Relation avec Docker

Pourquoi cette architecture nous mène vers Docker ?

- **Isolation** : Chaque microservice dans son propre container
- **Portabilité** : Déploiement identique sur tous les environnements
- **Scalabilité** : Multiplication des containers selon la charge
- **Orchestration** : Kubernetes pour gérer l'ensemble des services

Cette approche microservices constitue le fondement parfait pour comprendre l'intérêt de la conteneurisation avec Docker ! 



[Revenir au sommaire](#)

🎯 Live Tuto : Déployer en 5 minutes

Projets minimalistes pour tester Vercel et Render.com

Mettons les mains dans le cambouis avec 2 exemples ultra-simples !



[Revenir au sommaire](#)



Projet 1 : Site statique pour Vercel

```
# 1. Créer le dossier  
mkdir mon-site-vercel  
cd mon-site-vercel  
  
# 2. Créer le fichier HTML : index.html  
<!DOCTYPE html>  
<html>  
<head>  
    <title>Mon site déployé avec Vercel !</title>  
    <style>  
        body { font-family: Arial; text-align: center; padding: 50px; background: linear-gradient(135deg, #28a745, #ffc107); }  
        .card { background: rgba(255,255,255,0.1); padding: 30px; border-radius: 15px; backdrop-filter: blur(10px); }  
    </style>  
</head>  
<body>
```





Déployer sur Vercel (2 minutes)

Étapes ultra-simples

1. Push sur GitHub :

```
# Créer un repo sur github.com et récupérer l'URL  
git remote add origin https://github.com/VOTRE-USERNAME/mon-site-vercel.git  
git push -u origin main
```

2. Aller sur [vercel.com](#) → Se connecter avec GitHub

3. Cliquer "New Project" → Sélectionner votre repo

4. Cliquer "Deploy" → C'est tout ! 🎉

Résultat : Site live en 1 minute à <https://mon-site-vercel-xxx.vercel.app>



[Revenir au sommaire](#)



Projet 2 : API Node.js pour Render.com

```
# 1. Créer le projet
mkdir mon-api-render
cd mon-api-render

# 2. Initialiser Node.js
npm init -y

# 3. Installer Express
npm install express

# 4. Créer l'API
# 2. Créer le fichier server.js
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;
```





Dockerfile pour Render.com

```
FROM node:18-alpine

WORKDIR /app

# Copier et installer les dépendances
COPY package*.json ./
RUN npm ci --only=production

# Copier le code
COPY .

# Créer un utilisateur non-root
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```





Déployer sur Render.com (3 minutes)

Étapes simples

1. Push sur GitHub :

```
git init
git add .
git commit -m "feat: add simple API with Docker"
git remote add origin https://github.com/VOTRE-USERNAME/mon-api-render.git
git push -u origin main
```



2. Aller sur [render.com](#) → Se connecter avec GitHub

3. New Web Service → Connecter votre repo

4. Configuration :

- **Environment** : Docker
- **Region** : Frankfurt (plus proche)
- **Instance Type** : Free

5. Deploy → Attendre 2-3 minutes 



[Revenir au sommaire](#)

✓ Test des déploiements

Vérifier que tout fonctionne

Vercel - Tester le site :

```
# Ouvrir dans le navigateur
open https://mon-site-vercel-xxx.vercel.app
```

Render.com - Tester l'API :

```
# Test avec curl
curl https://mon-api-render-xxx.onrender.com/

# Réponse attendue :
{
  "message": "🌟 API déployée sur Render.com !",
  "timestamp": "2025-01-XX...",
  "status": "running",
  "platform": "render"
}
```



⟳ CI/CD automatique activé !

Ce qui se passe automatiquement

À chaque push sur GitHub :

Vercel :

- Build automatique du site
- Déploiement en ~30 secondes
- Preview URL pour chaque branch

Render.com :

- Build de l'image Docker
- Déploiement en ~2 minutes
- Health checks automatiques

Plus besoin de déploiement manuel ! 🎉



[Revenir au sommaire](#)

Aller plus loin

Améliorations possibles

Vercel :

- Ajouter `vercel.json` pour la configuration
- Variables d'environnement via le dashboard
- Domaine personnalisé

Render.com :

- Base de données PostgreSQL (gratuite)
- Variables d'environnement
- Monitoring et logs

Les deux utilisent le même principe : Git push = Déploiement automatique !



[Revenir au sommaire](#)

QCM : Micro-services et CI/CD

QCM sur les micro-services et le CI/CD

1. Quel est l'avantage principal des micro-services ?

- Ils permettent de créer des applications monolithiques.
- Ils permettent de découper une application en plusieurs services indépendants.
- Ils nécessitent moins de ressources que les applications traditionnelles.
- Ils sont plus difficiles à maintenir.

2. Dans l'exemple d'une application de e-commerce, quel micro-service gère les transactions de paiement ?

- Microservice de gestion de produits
- Microservice de gestion de commandes
- Microservice de gestion de paiement
- Microservice de gestion des utilisateurs



3. Pourquoi utiliser les micro-services ?

- Pour rendre l'application plus modulaire, plus facile à maintenir et plus scalable.
- Pour augmenter la complexité de l'application.
- Pour réduire le nombre de développeurs nécessaires.
- Pour éviter l'utilisation de conteneurs.



[Revenir au sommaire](#)

4. Quel est l'objectif principal du CI/CD ?

- Augmenter la complexité du développement logiciel.
- Automatiser le processus de développement, de test et de déploiement.
- Réduire la qualité du code.
- Remplacer les développeurs par des machines.

5. Quel outil est couramment utilisé pour le CI/CD ?

- Docker Hub
- Jenkins
- GitHub Packages
- Quay.io



Réponse(s)

1. Ils permettent de créer des applications modulaires et indépendantes.
2. Microservice de gestion de paiement
3. Pour rendre l'application plus modulaire, plus facile à maintenir et plus scalable.
4. Automatiser le processus de développement, de test et de déploiement.
5. GitHub Packages / Jenkins



[Revenir au sommaire](#)

Virtualisation vs conteneurisation

Virtualisation vs Conteneurisation

Comprendre les différentes approches d'isolation

Pour bien saisir la révolution que représente Docker, il est essentiel de comprendre les différences fondamentales entre la virtualisation traditionnelle et la conteneurisation moderne.

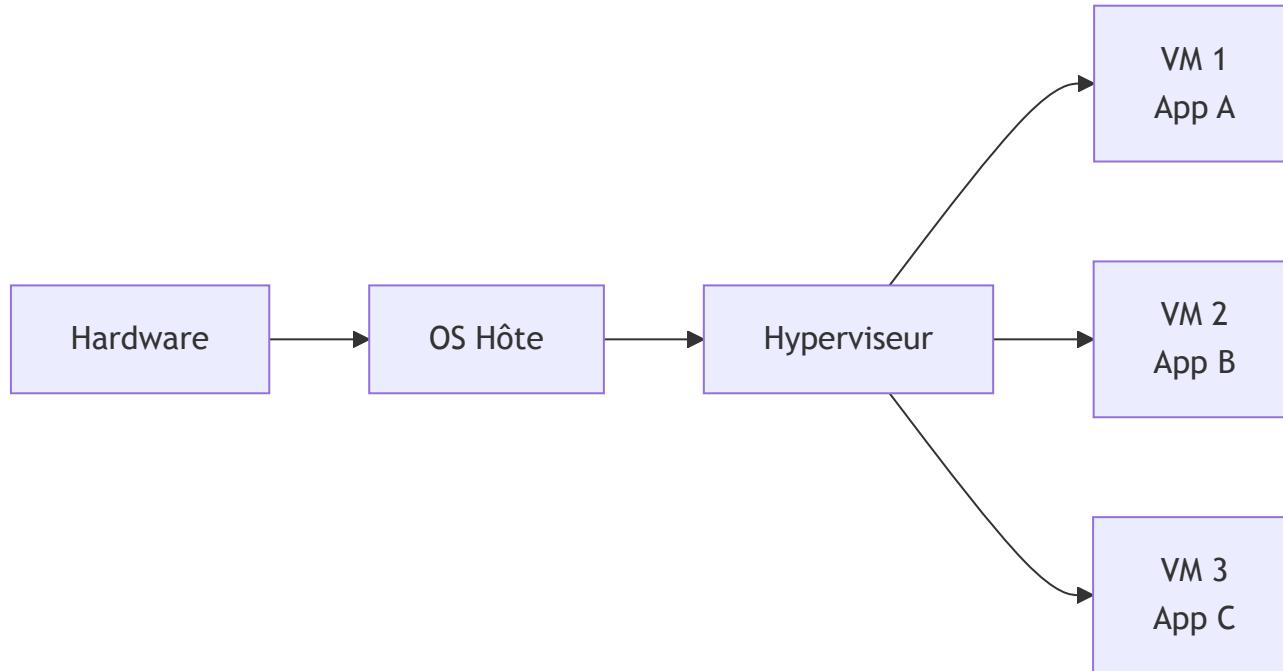


[Revenir au sommaire](#)

Virtualisation Traditionnelle



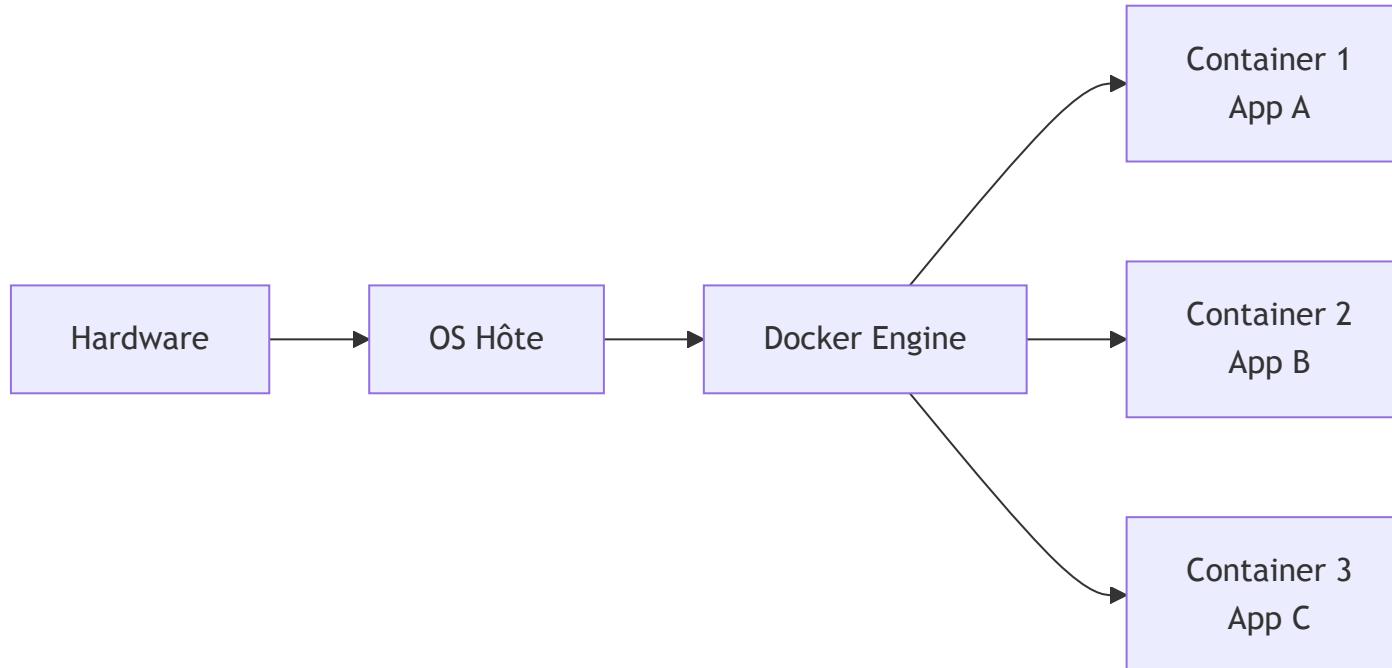
Architecture avec machines virtuelles



Conteneurisation Docker



Architecture avec containers



Comparaison des ressources



Performance et utilisation

Critère	Virtualisation	Conteneurisation
Taille	1-20 GB par VM	10-500 MB par container
RAM	512MB-8GB minimum	10-100MB typique
Démarrage	30s-5min	0.1-2s
CPU Overhead	5-15%	<1%



Avantages/Inconvénients

Virtualisation

 **Avantages** : Isolation maximale, différents OS  **Inconvénients** : Lourd, lent, consomme beaucoup

Conteneurisation

 **Avantages** : Léger, rapide, efficace  **Inconvénients** : Même OS requis, isolation moindre



Quand utiliser la virtualisation ? 🎯

Cas d'usage pour les VMs

- **Applications legacy** nécessitant un OS spécifique
- **Sécurité critique** : Isolation maximale requise
- **Environnements multi-OS** : Windows + Linux
- **Compliance réglementaire stricte**



[Revenir au sommaire](#)

Quand utiliser les containers ? 🐋

Cas d'usage pour Docker

- **Applications modernes** cloud-native
- **Microservices** et architecture découpée
- **Développement agile** avec déploiements fréquents
- **CI/CD** et automatisation



[Revenir au sommaire](#)

L'avenir : Container-First 🌟

Tendance 2025

- 85% des nouvelles applications utilisent des containers
- 40% de réduction des coûts d'infrastructure
- 3x amélioration de la vitesse de déploiement
- Container-First devient la norme par défaut



[Revenir au sommaire](#)

Introduction à Docker



[Revenir au sommaire](#)

Définition simple



Qu'est-ce que Docker exactement ?

Docker est une **plateforme de conteneurisation** qui permet d'emballer une application et toutes ses dépendances dans un conteneur portable, léger et autonome qui peut s'exécuter de manière cohérente sur n'importe quel environnement.



Vocabulaire Docker Essentiel

Les concepts de base à maîtriser

Avant de plonger dans la pratique, il est crucial de comprendre le vocabulaire Docker.

Ces termes reviendront constamment dans votre utilisation quotidienne.



[Revenir au sommaire](#)

Définitions fondamentales



Container vs Image

- **Container** : Un environnement d'exécution isolé et portable qui contient tout ce dont une application a besoin pour fonctionner (code, runtime, outils système, bibliothèques)
- **Image** : Un modèle en lecture seule qui sert de blueprint pour créer des containers. C'est un snapshot figé d'un système de fichiers avec toutes les dépendances



Dockerfile & Écosystème

Composants essentiels

- **Dockerfile** : Un fichier texte contenant une série d'instructions pour construire automatiquement une image Docker personnalisée
- **Docker Hub** : Le registre public officiel où sont stockées et partagées des millions d'images Docker prêtes à l'emploi
- **Docker Registry** : Un service de stockage et de distribution d'images Docker, peut être privé ou public



[Revenir au sommaire](#)

Écosystème complet 2025 🌟

Plateforme moderne

- **Communauté active** : Plus de 10 millions de développeurs dans le monde
- **Docker Desktop** : Interface graphique et outils de développement
- **Plus de 6 millions d'images** disponibles sur Docker Hub
- **85% des nouvelles applications** utilisent Docker ou Podman , son implémentation 100% open source en 2025

Car oui Docker n'est pas à 100% open source, il y a des licences propriétaires. C'est une entreprise qui cherche à capitaliser un minimum sur son produit.

Podman est une alternative open source à Docker, il est plus léger et plus rapide.

Il est possible de faire tourner Docker sur un serveur Linux sans Docker Desktop, pareil sur macOS via `podman` etc.



Architecture Docker moderne



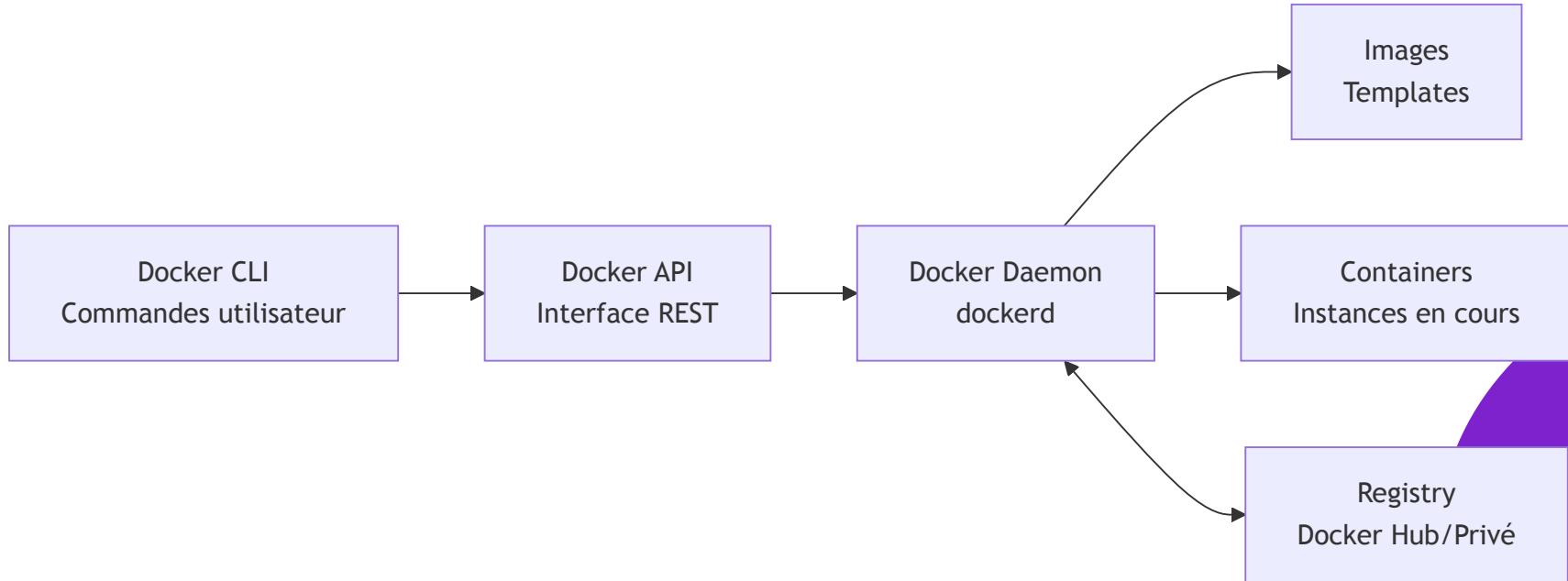
Composants principaux

- **Docker Engine** : Le cœur de Docker qui gère le cycle de vie des containers (création, exécution, arrêt, suppression)
- **Docker Daemon (dockerd)** : Service système qui s'exécute en arrière-plan et gère les objets Docker
- **Docker CLI** : L'interface en ligne de commande qui permet d'interagir avec le Docker Daemon via des commandes



[Revenir au sommaire](#)

Vue d'ensemble du système



Pourquoi Docker révolutionne ? 🚀

Le problème des dépendances

Avant Docker : "Ça marche sur ma machine" 😅

- Conflits de versions entre environnements
- Configuration manuelle complexe
- Incompatibilités système
- Déploiements imprévisibles



[Revenir au sommaire](#)

La solution Docker ✓

Avec Docker : "Ça marche partout" ✓

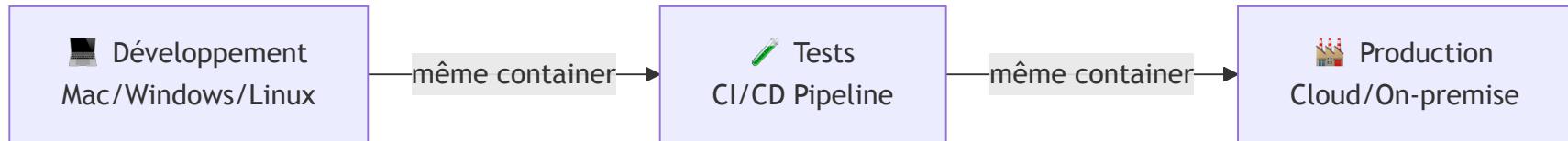
- Environnements identiques garantis
- Dépendances encapsulées
- Déploiement reproductible
- Isolation parfaite



[Revenir au sommaire](#)

Les super-pouvoirs de Docker 💪

Portabilité absolue



Avantages techniques concrets

Performance et efficacité

- **Démarrage ultra-rapide** : 0.1 à 2 secondes vs 30s-5min pour une VM
- **Densité élevée** : 100-1000 containers vs 5-20 VMs par serveur
- **Utilisation mémoire optimisée** : 10-100MB vs 512MB-8GB par instance
- **Performance native** : Overhead <1% vs 5-15% pour la virtualisation



Principes fondamentaux



Isolation et sécurité

- **Isolation** : Chaque container s'exécute dans son propre environnement isolé, séparé des autres containers et du système hôte
- **Namespaces** : Mécanisme Linux qui isole les ressources système (PID, réseau, système de fichiers)
- **Cgroups** : Limitation et contrôle des ressources système (CPU, mémoire, I/O) allouées aux containers



Philosophie Container-First 🎯

Stateless par défaut

Les containers Docker suivent le principe **stateless** :

- **Données éphémères** : Le container peut être détruit et recréé sans perte de fonctionnalité
- **État externalisé** : Les données persistantes sont stockées dans des volumes ou bases de données externes
- **Configuration externalisée** : Variables d'environnement et fichiers de configuration montés depuis l'extérieur



Portabilité et reproductibilité

Garanties Docker

- **Portabilité** : Les containers s'exécutent de manière identique sur tous les environnements supportant Docker (développement, test, production)
- **Immutabilité** : Les images Docker sont immuables, garantissant la reproductibilité des déploiements
- **Infrastructure as Code** : La configuration de l'infrastructure est définie dans du code versionnable et reproduitible



[Revenir au sommaire](#)

Premier contact avec Docker 🎯

Installation rapide 2025

```
# Linux (Ubuntu/Debian)
curl -fsSL https://get.docker.com | sh
# On ajoute l'utilisateur courant au groupe docker
sudo usermod -aG docker $USER

# macOS/Windows : Docker Desktop sur leur site ou :
wsl --install
```

Puis à nouveau la 1ère commande pour installer docker



[Revenir au sommaire](#)

Vérification installation ✓

Test de votre environnement

```
# Version et informations système  
docker --version  
docker info  
  
# Test classique : créer un container hello-world  
docker run hello-world
```



[Revenir au sommaire](#)

Exemple concret : Nginx en action 🌐

Déploiement d'un serveur web en une commande

```
# Lancement d'un serveur Nginx
docker run -d -p 8080:80 --name mon-nginx nginx:alpine

# Vérification du container
docker ps

# Consultation des logs
docker logs mon-nginx
```



Ce qui se passe en coulisses



Analyse du processus

1. **Pull automatique** : Téléchargement de l'image nginx:alpine
2. **Création du container** : Instance isolée avec Nginx
3. **Mapping de port** : Port 8080 de l'hôte → Port 80 du container
4. **Démarrage** : Nginx opérationnel en quelques secondes

Résultat : Serveur web accessible sur <http://localhost:8080>



[Revenir au sommaire](#)

Images vs Containers en pratique



Relation fondamentale

Images Docker 📦

- Templates **immuables** et **versionnés**
- Architecture en **couches** (layers) pour l'optimisation
- Stockées dans des **registries** (Docker Hub, privés)
- Peuvent être **taguées** pour le versioning

Containers Docker 🚤

- **Instances vivantes** créées à partir d'images
- **Environnements isolés** avec leur propre filesystem
- **États mutables** : peuvent être démarrés, arrêtés, modifiés
- **Éphémères** : données perdues à la suppression (sauf volumes)



Concepts avancés 🚀

Réseautage et stockage

- **Docker Network** : Réseau virtuel permettant la communication sécurisée entre containers
- **Docker Volume** : Mécanisme de persistance des données qui survit au cycle de vie des containers
- **Docker Secret** : Gestion sécurisée des informations sensibles (mots de passe, clés API, certificats)



[Revenir au sommaire](#)

Orchestration moderne 🎭

Solutions d'orchestration

- **Docker Swarm** : Solution d'orchestration native pour gérer des clusters de containers
- **Kubernetes** : Plateforme d'orchestration avancée pour le déploiement et la gestion de containers à grande échelle
- **Docker Stack** : Déploiement d'applications multi-services dans un cluster Swarm



Avantages pratiques pour les développeurs ✓

Bénéfices quotidiens

- **Scalabilité horizontale** : Multiplication facile des instances
- **Mise à jour sans interruption** : Remplacement transparent des containers
- **Récupération rapide** : Redémarrage instantané en cas de problème
- **Testing simplifié** : Environnements de test identiques à la production
- **Déploiement uniforme** : Même artefact du développement à la production



[Revenir au sommaire](#)

Votre premier exercice avec Docker

🎯 Exercice Principal Détaillé

Votre première expérience pratique

Maintenant que vous connaissez les concepts **ET** les commandes CLI, mettons les mains dans le cambouis ! Cet exercice vous fait pratiquer ce que vous venez d'apprendre.



[Revenir au sommaire](#)

Vérification de votre installation 🔎

Étape 1 : Vérifier que Docker fonctionne

```
# Vérifiez votre version Docker  
docker --version  
  
# Affichez les informations de base  
docker info | head -10
```

Questions simples :

- Quelle version de Docker avez-vous ?
- Docker est-il bien démarré ?



Premier container : Hello World! 🙌

Étape 2 : Votre tout premier container

```
# Lancez le container de test officiel  
docker run hello-world
```

Ce qui vient de se passer :

1. Docker a **téléchargé** l'image `hello-world`
2. Il a **créé** un container à partir de cette image
3. Le container a **affiché** un message de bienvenue
4. Le container s'est **arrêté** automatiquement



Explorer ce qui existe

Étape 3 : Regarder ce que vous avez maintenant

```
# Listez les images téléchargées  
docker images
```

```
# Listez tous les containers (même arrêtés)  
docker ps -a
```

Questions d'observation :

- Combien d'images avez-vous maintenant ?
- Quel est l'état de votre container hello-world ?



Deuxième container : Un serveur web!

Étape 4 : Lancer quelque chose d'utile

```
# Lancez un serveur web Nginx (en arrière-plan avec un nom)
docker run -d --name mon-premier-site nginx:alpine

# Vérifiez qu'il fonctionne
docker ps
```

Points d'apprentissage :

- -d : Lance en arrière-plan (détaché)
- --name : Donne un nom au container
- nginx:alpine : Version légère de Nginx



Interagir avec votre container



Étape 5 : Regarder ce qui se passe

```
# Voir les logs du serveur  
docker logs mon-premier-site
```

```
# Voir les processus qui tournent dans le container  
docker top mon-premier-site
```

```
# Voir l'utilisation des ressources  
docker stats mon-premier-site --no-stream
```



Nettoyer après vous



Étape 6 : Arrêter et supprimer

```
# Arrêter le serveur web  
docker stop mon-premier-site  
  
# Supprimer le container  
docker rm mon-premier-site  
  
# Vérifier que c'est bien parti  
docker ps -a
```



🟡 Exercice Express 3 : Explorer l'intérieur d'un container (20 min)

Ce qu'on apprend : Mode interactif, différences entre distributions

```
# 1. Entrer dans un container Ubuntu
docker run -it ubuntu:latest bash

# Dans le container, explorer :
ls /
cat /etc/os-release
ps aux
whoami

# Sortir du container
exit

# 2. Comparer avec Alpine Linux
docker run -it alpine:latest sh
```

Questions :

- Quelles sont les différences principales entre Alpine et Ubuntu ?



[Revenir au sommaire](#)

Test de maîtrise : À votre tour !

Étape 7 : Exercice autonome

Mission : Lancez un container Nginx, PostGRES en arrière-plan avec le nom "mon-nginx"

Vous pouvez utiliser la commande suivante pour vérifier si vous avez bien réussi l'exercice :

```
docker ps
```

Essayez de faire un curl sur le container Nginx depuis votre machine, que se passe t-il ?



```
# À vous de jouer ! Essayez sans regarder la solution...
# Indice : nginx:alpine

# Solution (ne regardez qu'après avoir essayé) :
docker run -d --name mon-nginx nginx:alpine
docker run -d --name mon-postgres postgres:alpine

# Vérification
docker ps
docker logs mon-nginx
docker logs mon-postgres

# Nettoyage
docker stop mon-nginx && docker rm mon-nginx
docker stop mon-postgres && docker rm mon-postgres
```



Est ce que votre curl fonctionne ?

```
curl http://localhost:80
```

non, car le container Nginx n'a pas été associé au port de votre PC, nous allons très vite en parler dans le prochain module.



[Revenir au sommaire](#)

Exercice Découverte - Exploration d'images populaires

Mission : Testez différentes images Docker populaires

```
# Essayez ces images une par une
docker run --rm alpine:latest echo "Je suis Alpine Linux!"
docker run --rm ubuntu:latest cat /etc/os-release
docker run --rm python:3.12-alpine python --version
```

Questions :

- Que fait l'option `--rm` ?



[Revenir au sommaire](#)



Félicitations !

Vous venez de maîtriser :

- Vérifier** votre installation Docker
- Lancer** votre premier container (`docker run`)
- Lister** les containers et images (`docker ps` , `docker images`)
- Surveiller** vos containers (`docker logs` , `docker stats`)
- Gérer le cycle de vie** (`docker stop` , `docker rm`)
- Pratiquer** de façon autonome



Prêt pour le CLI !

Vous maîtrisez maintenant les commandes de base.

On peut passer aux commandes avancées avec le CLI !



[Revenir au sommaire](#)



Le CLI Docker

Le CLI Docker

Votre outil de travail quotidien

Le **Docker CLI** est votre interface principale pour interagir avec Docker. Maîtrisons les commandes essentielles pour être productifs au quotidien.



[Revenir au sommaire](#)

Structure des commandes



Syntaxe de base

```
docker [OPTIONS] COMMAND [ARG...]
```

Exemples pratiques :

- docker run -d -p 80:80 nginx : Lance un serveur web
- docker ps -a : Liste tous les containers
- docker build -t myapp . : Construit une image



[Revenir au sommaire](#)

Gestion des containers - Essentiel



Commandes incontournables

Commande	Description	Exemple
docker run	Créer et démarrer	docker run -d -p 80:80 --name web nginx
docker ps	Containers actifs	docker ps
docker ps -a	Tous les containers	docker ps -a
docker stop	Arrêter	docker stop web
docker start	Redémarrer	docker start web
docker rm	Supprimer	docker rm web



[Revenir au sommaire](#)

Options run les plus utiles 🔧

Paramètres essentiels pour docker run

```
# Déattaché avec nom et port  
docker run -d --name mon-app -p 8080:80 nginx  
  
# Variables d'environnement  
docker run -e NODE_ENV=production -e PORT=3000 node-app  
  
# Volumes et répertoire de travail  
docker run -v $(pwd):/app -w /app node:18 npm install  
  
# Limite de ressources  
docker run --memory=512m --cpus=1 mon-app
```



Gestion des images



Images : télécharger, construire, gérer

Commande	Description	Exemple
docker pull	Télécharger	docker pull nginx:alpine
docker build	Construire	docker build -t myapp:v1.0 .
docker images	Lister	docker images
docker tag	Tagger	docker tag myapp:v1.0 myapp:latest
docker rmi	Supprimer	docker rmi myapp:v1.0



Inspection et débogage



Comprendre ce qui se passe

Commande	Usage	Exemple
docker logs	Voir les logs	docker logs -f --tail 100 mon-app
docker exec	Exécuter dans le container	docker exec -it mon-app bash
docker inspect	Détails complets	docker inspect mon-app
docker stats	Utilisation ressources	docker stats



[Revenir au sommaire](#)

Commandes d'inspection pratiques

Débogage rapide

```
# Accès shell interactif
docker exec -it mon-container bash

# exemple en faisant directement la commande dans le container pour la récupérer sur la machine (hôte)
docker exec le_nom_du_container /usr/bin/mysqldump -u votre_utilisateur --password=votre_mot_de_passe

# Logs en temps réel
docker logs -f mon-container

# Monitoring des ressources
docker stats --no-stream

# Processus dans le container
docker top mon-container
```



Volumes et réseaux

Gestion des ressources

Volumes :

```
docker volume create mon-volume  
docker volume ls  
docker volume inspect mon-volume  
docker volume rm mon-volume
```

Réseaux :

```
docker network create mon-reseau  
docker network ls  
docker network connect mon-reseau mon-container
```



Nettoyage et maintenance



Libérer l'espace disque

Commande	Action	Impact
docker system prune	Nettoyage général	Containers arrêtés, réseaux, images dangling
docker container prune	Containers arrêtés	Libère l'espace containers
docker image prune	Images non utilisées	Nettoie les images orphelines
docker volume prune	Volumes non utilisés	Supprime les volumes inutiles



Nettoyage avancé



Surveillance et nettoyage

```
# Voir l'utilisation de l'espace  
docker system df  
  
# Nettoyage complet (ATTENTION !)  
docker system prune -a --volumes  
  
# Forcer la suppression  
docker rm -f $(docker ps -aq)  
docker rmi -f $(docker images -q)
```



Workflow quotidien optimal



Séquence type de développement

```
# 1. Build de l'image  
docker build -t mon-app:dev .  
  
# 2. Lancement en mode développement  
docker run -d -p 3000:3000 -v $(pwd):/app --name dev-app mon-app:dev  
  
# 3. Monitoring  
docker logs -f dev-app  
  
# 4. Debug si besoin  
docker exec -it dev-app bash  
  
# 5. Nettoyage  
docker stop dev-app && docker rm dev-app
```



Commandes avancées pour pros 🚀

Techniques avancées

```
# Copier fichiers container ↔ hôte  
# concretement cela veut dire que vous pouvez copier un fichier de votre machine vers le container  
docker cp mon-file.txt mon-container:/app/mon-file.txt  
# et inversement  
docker cp mon-container:/app/mon-file.txt ./mon-file.txt  
  
# pratique si vous voulez modifier un fichier ou récupérer un dossier dans le container  
  
# Créer image depuis container  
docker commit mon-container mon-image:v2  
  
# Export/Import d'images  
docker save -o mon-app.tar mon-app:latest  
docker load -i mon-app.tar
```



Bonnes pratiques CLI



Conseils pour être efficace

- Utilisez des noms explicites :** --name web-frontend
- Toujours spécifier les tags :** nginx:1.25-alpine
- Nettoyez régulièrement :** docker system prune
- Utilisez les aliases pour les commandes fréquentes**
- Logs avec limites :** docker logs --tail 50

- Évitez latest en production**
- N'oubliez pas de supprimer les containers de test**



Réseaux & Volumes Docker



Pourquoi a-t-on besoin de réseaux et volumes ? 🤔

Les problèmes à résoudre

Sans réseaux Docker :

- Vos containers ne peuvent pas se parler facilement
- Difficile de faire communiquer une app web avec sa base de données

Sans volumes Docker :

- 💀 **Vos données disparaissent** quand vous supprimez un container
- Impossible de partager des fichiers entre containers
- Pas de persistance pour vos bases de données

Ce que Docker résout

- ✓ **Réseaux** : Communication simple entre containers
- ✓ **Volumes** : Vos données survivent aux containers



Volumes - Le problème de base

Que se passe-t-il SANS volumes ?

```
# Créer un container avec des données
docker run -it --name test-data ubuntu:20.04 bash

# Dans le container, créer un fichier important
echo "Mes données importantes" > /app/data.txt
exit

# PROBLÈME : Supprimer le container = PERTE DES DONNÉES
docker rm test-data

# 💀 Le fichier data.txt a DISPARU pour toujours !
```

 **Résultat :** Toutes vos données sont **perdues** !



🔧 Solution : Les Volumes Docker

Qu'est-ce qu'un volume exactement ?

Un **volume** est un **dossier spécial** que Docker gère pour vous :

- **Stocké sur votre disque dur** (pas dans le container)
- **Partageable** entre plusieurs containers
- **Persistent** : survit à la suppression du container
- **Géré par Docker** : sauvegarde, permissions automatiques

Analogie : C'est comme un **disque dur externe** que vous branchez sur différents ordinateurs !

You can also see it as a virtual hard drive managed by Docker.



Types de volumes - Comprendre les différences

3 façons de gérer vos données

Type	Quand l'utiliser	Où sont les données
 Volume anonyme	Par accident/débutant	Docker le gère
 Volume nommé	Production	Docker le gère
 Bind mount	Développement	Sur votre PC



[Revenir au sommaire](#)

Analogie simple : Volumes = Clés USB

Type	Quand l'utiliser	Où sont les données	Analogie 
 Volume anonyme	Débutant / oubli	Docker (nom auto)	 Clé USB sans étiquette jetée dans un tiroir — tu la retrouves jamais
 Volume nommé	Production	Docker (nom choisi)	 Clé USB avec ton nom, rangée dans une boîte — facile à retrouver, réutiliser
 Bind mount	Développement	Sur votre PC	 Tu bosses directement sur un dossier de ton PC, comme si ton app était "en live"



Volumes Anonymes - Ce qui arrive aux débutants

Quand Docker crée des volumes automatiquement

```
# ❌ Commande de débutant (SANS -v)
docker run -d --name mysql-test mysql:8.0

# 🤖 Docker crée automatiquement un volume ANONYME
docker volume ls
# DRIVER      VOLUME NAME
# local       a1b2c3d4e5f6... ← Volume avec nom aléatoire !
```

⚠️ **Problème :** Volume avec nom bizarre, difficile à retrouver !



Pourquoi Docker fait ça automatiquement ?

```
# Voir les détails du container MySQL
docker inspect mysql-test

# Dans les détails, vous verrez :
# "Mounts": [
#   {
#     "Type": "volume",
#     "Name": "a1b2c3d4e5f6...",
#     "Source": "/var/lib/docker/volumes/a1b2c3d4e5f6.../",
#     "Destination": "/var/lib/mysql"
#   }
# ]
```

 **Pourquoi ? MySQL a besoin** de persister ses données, Docker crée donc automatiquement un volume pour `/var/lib/mysql` !



🔥 Volumes Nommés - La bonne pratique

Créer et utiliser un volume avec un nom explicite

```
# ✓ Créer un volume avec un nom clair  
docker volume create mysql-data  
  
# ✓ Utiliser ce volume avec votre container  
docker run -d \  
  --name mysql-prod \  
  -v mysql-data:/var/lib/mysql \  
  -e MYSQL_ROOT_PASSWORD=password123 \  
  mysql:8.0  
  
# ✓ Vérifier que vos données sont là  
docker volume ls  
# DRIVER      VOLUME NAME  
# local        mysql-data    ← Nom clair et lisible !
```



Test de persistance des données

```
# 1. Créer une base de données
docker exec -it mysql-prod mysql -p
# permet de faire deux commandes en même temps, se connecter au container et exécuter une commande
# CREATE DATABASE test_app;
# exit

# 2. SUPPRIMER le container (simulation crash)
docker stop mysql-prod
docker rm mysql-prod

# 3. Recréer un nouveau container avec le MÊME volume
docker run -d \
  --name mysql-nouveau \
  -v mysql-data:/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=password123 \
```

🎉 **Résultat :** Vos données ont **survécu** à la destruction du container !



Bind Mounts - Pour le développement

Lier un dossier de votre PC au container

```
# Créer un dossier sur votre PC
mkdir ~/mon-projet
echo "console.log('Hello Docker!');" > ~/mon-projet/app.js

# Lier ce dossier au container
docker run -it \
  --name dev-container \
  -v ~/mon-projet:/app \
  node:18-alpine \
  sh

# Dans le container :
# cd /app
# ls -la      ← Vous voyez app.js !
# node app.js ← "Hello Docker!"
```



[Revenir au sommaire](#)

Magie du bind mount - Modification en temps réel

```
# Sur votre PC, modifier le fichier  
echo "console.log('Modifié depuis mon PC!');" > ~/mon-projet/app.js  
  
# Dans le container, relancer  
# node app.js ← "Modifié depuis mon PC!"
```

💡 **Magie** : Les modifications sur votre PC apparaissent **instantanément** dans le container !



[Revenir au sommaire](#)

Ce qu'il se passe VRAIMENT avec un bind mount :

- Le dossier de votre PC est **monté directement** dans le container
- Il n'y a **aucune copie** de fichier : c'est le même fichier vu des deux côtés
- Le container lit/écrit directement dans le dossier du host
- Toute modification faite sur le PC est **instantanément visible** dans le container
- Et inversement, ce que fait le container modifie le fichier sur le host
- Vous pouvez vérifier ça avec `cat /app/app.js` ou `cat ~/mon-projet/app.js` → même contenu

 Attention : Si un dossier existe déjà dans le container (ex: `/app`), le bind mount va **masquer** son contenu. Le dossier de votre PC **remplace entièrement** celui du container. Le contenu initial du container à cet endroit est **invisible, mais pas supprimé**

Si vous annulez le bind mount, le dossier du container reprend son contenu initial.



Réseaux Docker - Le problème de communication

Pourquoi les containers ne se parlent pas par défaut ?

```
# Lancer 2 containers séparés
docker run -d --name app1 nginx:alpine
docker run -d --name app2 nginx:alpine

# Essayer de faire communiquer app1 avec app2
docker exec app1 ping app2
# ping: bad address 'app2' ← ÉCHEC !
```

⚠️ **Problème :** Les containers sont **isolés** par défaut !



🔗 Solution : Créer un réseau personnalisé

Les containers peuvent se parler par leur nom

```
# 1. Créer un réseau personnalisé  
docker network create mon-reseau
```

```
# 2. Lancer les containers dans ce réseau  
docker run -d --name app1 --network mon-reseau nginx:alpine  
docker run -d --name app2 --network mon-reseau nginx:alpine
```

```
# 3. Maintenant ils peuvent se parler !  
docker exec app1 ping app2  
# PING app2 (172.20.0.3): 56 data bytes ← ✓ ÇA MARCHE !
```

```
# Ils sont tout les deux sur le même réseau "mon-reseau" et peuvent se parler.
```

👉 Résultat : Communication par **nom de container** !



🏗 Exemple concret : Site web + Base de données

Stack complète qui fonctionne ensemble

```
# 1. Créer l'infrastructure
docker network create webapp-network
docker volume create database-data

# 2. Lancer la base de données
docker run -d \
  --name database \
  --network webapp-network \
  -v database-data:/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=secret \
  -e MYSQL_DATABASE=myapp \
  mysql:8.0
```



Suite : Application web

```
# 3. Lancer l'application web
docker run -d \
--name webapp \
--network webapp-network \
-p 3000:3000 \
-e DATABASE_HOST=database \
-e DATABASE_USER=root \
-e DATABASE_PASSWORD=secret \
-e DATABASE_NAME=myapp \
node:18-alpine \
sh -c "
  npm init -y &&
  npm install express mysql2 &&
  echo 'const express = require(\"express\");
  const app = express();'
```



Test de la stack complète

```
# 4. Tester que tout fonctionne
curl http://localhost:3000
# "App connectée à MySQL!" ← ✓ ÇA MARCHE !

# 5. Voir les containers qui communiquent
docker exec webapp ping database
# PING database (172.21.0.2) ← Communication réseau ✓

# 6. Vérifier la persistance
docker volume inspect database-data
# Les données MySQL sont sauvegardées ✓
```



🔍 Types de réseaux Docker - Les vraies différences

🤔 Pourquoi bridge et host semblent similaires ?

À première vue, **bridge** et **host** ont l'air pareils :

- Les containers peuvent accéder à Internet
- Tu peux exposer des ports
- Ça marche pour tes apps

MAIS la différence est dans **l'isolation réseau et comment ça fonctionne sous le capot !**



📦 Réseau Bridge (défaut) - Isolation sécurisée

```
# Le container a son propre réseau virtuel
docker run -d -p 8080:80 --name web nginx

# ✅ Ce qui se passe :
# - Container a une IP interne (ex: 172.17.0.2)
# - Tu DOIS utiliser -p pour exposer les ports
# - Accessible via localhost:8080 sur ton PC
# - Container isolé du réseau de ton PC
```

🔒 **Isolation** : Container dans sa bulle réseau, plus sécurisé



Réseau Host - Performance maximale

```
# Le container utilise directement le réseau de ta machine
docker run -d --network host --name web nginx

# ⚡ Ce qui se passe :
# - Container utilise l'IP de ton PC
# - PAS besoin de -p → nginx accessible direct sur port 80
# - Plus rapide car pas de couche réseau virtuelle
# - ⚠ Marche QUE sur Linux natif (pas Docker Desktop Mac/Windows)
```

 **Performance** : Container "fusionné" avec ton PC, plus rapide



[Revenir au sommaire](#)

🚫 Réseau None - Isolation totale

```
# Aucun réseau du tout
docker run -d --network none --name isolated alpine

# 🔒 Ce qui se passe :
# - Pas d'accès Internet
# - Pas de communication avec d'autres containers
# - Parfait pour traitement de données sensibles
```

🔑 Sécurité : Container complètement coupé du monde



[Revenir au sommaire](#)

Tableau comparatif - Bridge vs Host vs None

Mode	IP container ?	Isolation ?	Accès aux ports	Performance	Usage typique
📦 bridge	✓ Oui (virtuel)	✓ Sécurisé	via -p	Standard	Apps normales, production
🏡 host	✗ Non (host IP)	✗ Aucune	direct	Maximum	Apps haute perf, debug
🚫 none	✗ Aucune	✓ Totale	aucun	N/A	Traitement isolé
🔗 bridge custom	✓ Oui	✓ Sécurisé	via -p + DNS	Standard	Multi-containers



⌚ Exemples concrets des différences

```
# === BRIDGE (défaut) ===  
docker run -d -p 8080:80 --name web-bridge nginx  
# → Accessible sur http://localhost:8080  
# → Container IP: 172.17.0.2 (réseau virtuel)  
  
# === HOST ===  
docker run -d --network host --name web-host nginx  
# → Accessible sur http://localhost:80 (direct)  
# → Container IP: même que ton PC  
  
# === Comparaison ===  
docker exec web-bridge ip addr      # IP virtuelle Docker  
docker exec web-host ip addr       # IP de ton PC
```



⚠ Limitations importantes

Réseau Host :

- ✗ Ne fonctionne QUE sur Linux natif
- ✗ Docker Desktop (Mac/Windows) → host = bridge automatiquement
- ✗ Moins sécurisé (pas d'isolation)
- ✗ Conflicts de ports possible avec le host

Réseau Bridge :

- ! Containers sur bridge par défaut ne se voient pas par nom
- ✓ Solution : créer un bridge personnalisé



Analogies pour retenir

Bridge :

- **Analogie** : Tu es dans une colocation avec d'autres colocataires (containers) : vous pouvez parler entre vous (même réseau), et vous avez tous accès à Internet.

Host :

- **Analogie** : Tu bosses **seul** sur ton propre PC connecté directement à Internet - aucune cloison, tu fais tout toi-même, vite, mais moins sécurisé.

None :

- **Analogie** : Tu bosses dans une **salle sans Wi-Fi, sans câble, sans rien** : impossible de communiquer, même avec tes voisins.



Commandes essentielles - Diagnostic et debug

Réseaux - Voir ce qui se passe

```
# Lister tous les réseaux  
docker network ls  
  
# Voir les détails d'un réseau (quels containers sont dessus)  
docker network inspect mon-reseau  
  
# Connecter un container existant à un réseau  
docker network connect mon-reseau mon-container  
  
# Tester la connectivité entre containers  
docker exec container1 ping container2  
docker exec container1 nslookup container2
```



Volumes - Gérer vos données

```
# Lister tous les volumes
docker volume ls

# Voir où Docker stocke un volume sur votre disque
docker volume inspect mon-volume

# Nettoyer les volumes non utilisés
docker volume prune

# Voir l'espace utilisé par Docker
docker system df

# Backup d'un volume
docker run --rm -v mon-volume:/data -v $(pwd):/backup alpine \
    tar czf /backup/backup.tar.gz -C /data .
```



⚠️ Erreurs courantes et solutions

"Container can't connect to database"

```
# ❌ Erreur : containers pas sur le même réseau  
docker run -d --name db mysql:8.0  
docker run -d --name app mon-app # Différents réseaux !
```

```
# ✅ Solution : même réseau  
docker network create app-net  
docker run -d --name db --network app-net mysql:8.0  
docker run -d --name app --network app-net mon-app
```



"Data lost after container restart"

```
# ❌ Erreur : pas de volume  
docker run -d --name db mysql:8.0 # Données perdues !  
  
# ✅ Solution : volume nommé  
docker volume create db-data  
docker run -d --name db -v db-data:/var/lib/mysql mysql:8.0
```



"Permission denied in bind mount"

```
# ❌ Erreur : problème de permissions  
docker run -v /host/folder:/container/folder image  
  
# ✅ Solution : utiliser l'option :Z pour SELinux  
docker run -v /host/folder:/container/folder:Z image  
  
# ✅ Alternative : changer les permissions  
chmod 755 /host/folder
```





Bonnes pratiques 2025

Réseaux sécurisés

-  DO - Créez des réseaux séparés par fonction :

```
docker network create frontend-net  
docker network create backend-net  
# Web servers sur frontend-net  
# Databases sur backend-net
```

-  DON'T - Utilisez le réseau bridge par défaut en production



Volumes optimisés

-  DO - Volumes nommés en production :

```
docker volume create app-data
docker run -v app-data:/data mon-app
```

-  DO - Bind mounts en développement :

```
docker run -v $(pwd)/src:/app/src mon-app
```

-  DON'T - Volumes anonymes (sauf cas spéciaux)



Sécurité

 **DO** - Lecture seule quand possible :

```
docker run -v /host/config:/app/config:ro mon-app
```

 **DO** - Réseaux internes pour les bases de données :

```
docker network create --internal db-network
```

 **DON'T** - Exposez les ports de database directement



[Revenir au sommaire](#)

🎯 Récapitulatif - Ce que vous avez appris

Volumes 📁

- **Problème** : Les données disparaissent avec les containers
- **Solution** : Volumes pour la persistance
- **Types** : Anonymes (éviter), nommés (production), bind mounts (dev)

Réseaux 🌐

- **Problème** : Containers isolés par défaut
- **Solution** : Réseaux personnalisés pour la communication
- **Magie** : Communication par nom de container
- **Types** : Bridge (sécurisé), Host (performance), None (isolé)

Next step 🚀

Prêt pour **Docker Compose** qui simplifie tout ça !



[Revenir au sommaire](#)

QCM : Parlons de Docker

QCM sur l'introduction à Docker

1. Qu'est-ce que Docker principalement ?

- Un langage de programmation pour créer des applications
- Une plateforme de conteneurisation pour isoler et déployer des applications
- Un système d'exploitation léger pour serveurs
- Un outil de versioning comme Git

2. Quels sont les trois composants principaux de l'architecture Docker ?

- Image, Container, Registry
- Client, Daemon, Hub
- Client, Daemon, Registry
- Image, Volume, Network



3. Quelle est la différence principale entre une image et un container Docker ?

- Une image est en lecture seule, un container est l'instance exécutable
- Une image est temporaire, un container est permanent
- Une image est locale, un container est distant
- Aucune différence, ce sont des synonymes

4. Quel n'est PAS un avantage de Docker ?

- Isolation des applications
- Portabilité entre environnements
- Amélioration automatique des performances
- Facilité de déploiement



5. Quelle est la principale différence entre Docker et les VMs ?

- Docker partage le kernel de l'hôte, les VMs ont leur propre OS
- Docker est plus lourd que les VMs
- Les VMs sont plus sécurisées par défaut
- Docker ne peut pas fonctionner sur Windows



6. Qu'est-ce que Docker Hub ?

- L'interface graphique de Docker
- Le registry public officiel pour les images Docker
- L'outil de monitoring de Docker
- Le système de fichiers de Docker

7. Que fait la commande `docker run -it ubuntu bash` ?

- Lance un container Ubuntu en arrière-plan
- Télécharge l'image Ubuntu sans la lancer
- Lance un container Ubuntu interactif avec terminal
- Supprime un container Ubuntu existant



8. Comment conserver des données après la suppression d'un container ?

- Les données sont automatiquement sauvegardées
- Utiliser des volumes Docker
- Redémarrer le container
- Impossible, les données sont toujours perdues



[Revenir au sommaire](#)

9. Qu'est-ce qu'une "layer" dans une image Docker ?

- Un fichier de configuration
- Une instruction du Dockerfile qui crée une couche
- Un container en cours d'exécution
- Une sauvegarde automatique



[Revenir au sommaire](#)

10. Comment Docker s'intègre-t-il dans la philosophie DevOps ?

- Il remplace complètement les pratiques DevOps
- Il facilite l'intégration continue et la livraison continue
- Il est uniquement destiné aux développeurs
- Il n'a aucun rapport avec DevOps



11. Un développeur dit : "Mon application fonctionne sur ma machine mais pas en production". Comment Docker peut-il résoudre ce problème ?

- Docker ne peut pas résoudre ce type de problème
- En standardisant l'environnement d'exécution avec des containers
- En installant automatiquement les bonnes versions
- En accélérant l'application

12. Pourquoi Docker est-il particulièrement adapté aux architectures microservices ?

- Il rend les applications plus rapides
- Il permet d'isoler, déployer et mettre à l'échelle chaque service indépendamment
- Il supprime le besoin de bases de données
- Il automatisé le code



Réponses

1. Une plateforme de conteneurisation pour isoler et déployer des applications
2. Client, Daemon, Registry
3. Une image est en lecture seule, un container est l'instance exécutable
4. Amélioration automatique des performances
5. Docker partage le kernel de l'hôte, les VMs ont leur propre OS
6. Le registry public officiel pour les images Docker
7. Lance un container Ubuntu interactif avec terminal
8. Utiliser des volumes Docker
9. Une instruction du Dockerfile qui crée une couche
10. Il facilite l'intégration continue et la livraison continue
11. En standardisant l'environnement d'exécution avec des containers
12. Il permet d'isoler, déployer et mettre à l'échelle chaque service indépendamment



Barème et Correction

Réponses Correctes

Questions 1-6 (Fondamentaux)

1. **B** - Docker est une plateforme de conteneurisation
2. **C** - Client, Daemon, Registry
3. **A** - Image = template, Container = instance



[Revenir au sommaire](#)

Suite des réponses



4. **C** - Docker n'améliore pas automatiquement les performances
5. **A** - Docker partage le kernel, VMs ont leur OS
6. **B** - Docker Hub est le registry public officiel

Réponses pratiques



Questions 7-10 (Pratique) 7. **C** - Lance un container interactif 8. **B** - Utiliser des volumes Docker 9. **B** - Une instruction Dockerfile = une layer 10. **B** - Facilite CI/CD

Score d'évaluation

Score d'évaluation

- **12-13 bonnes réponses :** 🏆 Expert Docker !
- **10-11 bonnes réponses :** 🥇 Très bon niveau
- **8-9 bonnes réponses :** 🥈 Bon niveau, quelques révisions
- **6-7 bonnes réponses :** 🥉 Niveau correct, approfondissez
- **< 6 bonnes réponses :** 📚 Reprenez les concepts de base



[Revenir au sommaire](#)



Explications Détaillées

Pourquoi ces réponses ?

Question 3 - La distinction image/container est fondamentale :

- **Image** : Template en lecture seule (comme une classe en programmation)
- **Container** : Instance exécutable (comme un objet)



Suite explications



Question 5 - Architecture différente :

- **Docker** : Containers partagent le kernel de l'hôte
- **VMs** : Chaque VM a son propre système d'exploitation

Question 8 - Persistance cruciale :

- Par défaut, les données dans un container sont éphémères
- Les volumes permettent la persistance au-delà du cycle de vie du container



Points à retenir ✓

Points à retenir

- ✓ Docker révolutionne le déploiement d'applications
- ✓ La conteneurisation != virtualisation
- ✓ Les volumes sont essentiels pour la persistance
- ✓ Docker facilite DevOps et microservices



[Revenir au sommaire](#)



Suite du parcours

Une fois ce QCM maîtrisé, vous êtes prêt(e) pour :

- **Dockerfiles** : Créer vos propres images
- **Docker CLI** : Maîtriser la ligne de commande
- **Réseaux Docker** : Faire communiquer vos containers
- **Orchestration** : Docker Compose et Kubernetes

Conseil : Si vous avez des difficultés, reprenez la partie théorique et refaites les exercices pratiques !



[Revenir au sommaire](#)

Exercices CLI Docker



3 niveaux DevOps progressifs

Maîtrisez Docker avec des **stacks complètes** : réseaux + volumes + communication inter-containers

!

Pourquoi ces exercices ? 🤔

Objectifs pédagogiques

- 🌐 **Réseaux** : Faire communiquer plusieurs containers
- 💾 **Volumes** : Persister et partager des données
- 🔧 **Variables** : Configurer vos applications
- 👀 **Monitoring** : Voir ce qui se passe en temps réel
- 🚀 **Production** : Préparer des environnements réalistes

Ce que vous allez construire aujourd'hui :

- Base de données avec interface web
- Site WordPress complet
- Environnement DevOps multi-distributions
- Stack de monitoring professionnel



[Revenir au sommaire](#)

Exercice Niveau Simple

Stack PostgreSQL + Interface Web

🎯 Objectif : Votre première base de données avec interface

Ce que vous allez apprendre :

- Créer et utiliser des **réseaux Docker**
- Persister des données avec les **volumes**
- Connecter des containers entre eux
- Avoir un **feedback visuel** de votre base de données



[Revenir au sommaire](#)

Qu'est-ce que PostgreSQL ?

PostgreSQL est une base de données relationnelle très populaire :

- Plus moderne que MySQL pour certains aspects
- Utilisée par Instagram, Spotify, Reddit
- Excellente pour l'apprentissage et la production

phpPgAdmin est une interface web pour PostgreSQL :

- Équivalent de phpMyAdmin mais pour PostgreSQL
- Permet de créer des tables, voir les données visuellement
- Parfait pour débuter sans lignes de commande



🔧 Étape 1 : Préparer l'environnement

```
# Créer le réseau pour que les containers se parlent
docker network create db-network

# Créer les volumes pour la persistence
docker volume create postgres-data
docker volume create postgres-logs

# Vérifier que tout est créé
docker network ls | grep db-network
docker volume ls | grep postgres
```

💡 Pourquoi faire ça ?

- **Réseau** : Les containers pourront se parler par leur nom
- **Volumes** : Vos données survivront aux redémarrages
- **Organisation** : Structure propre et réutilisable



🔧 Étape 2 : Lancer PostgreSQL

```
# Lancer PostgreSQL avec toute la configuration
docker run -d \
--name ma-postgres \
--network db-network \
-e POSTGRES_DB=formation \
-e POSTGRES_USER=docker \
-e POSTGRES_PASSWORD=formation123 \
-v postgres-data:/var/lib/postgresql/data \
-v postgres-logs:/var/log/postgresql \
-p 5432:5432 \
postgres:15-alpine
```



Explication de la commande PostgreSQL

Décortiquons chaque option :

```
--name ma-postgres          # Nom du container (pour s'y référer)
--network db-network         # Rejoint notre réseau personnalisé
-e POSTGRES_DB=formation   # Crée une base "formation"
-e POSTGRES_USER=docker     # Utilisateur avec droits admin
-e POSTGRES_PASSWORD=...    # Mot de passe (OBLIGATOIRE!)
-v postgres-data:/var/lib... # Volume pour les données
-v postgres-logs:/var/log... # Volume pour les logs
-p 5432:5432                 # Port accessible depuis votre PC
postgres:15-alpine          # Image officielle, version légère
```



🔧 Étape 3 : Vérifier que PostgreSQL fonctionne

```
# Attendre 5 secondes que PostgreSQL démarre
echo "⏳ PostgreSQL démarre..."
sleep 5

# Vérifier le statut
docker ps | grep postgres

# Voir les logs de démarrage
docker logs ma-postgres

# Tester la connexion
docker exec ma-postgres pg_isready -U docker
```

✓ Que voir :

- Container en statut "Up"
- Logs sans erreur "database system is ready"
- Message "accepting connections"



Étape 4 : Lancer l'interface web phpPgAdmin

```
# Interface web pour gérer PostgreSQL
docker run -d \
--name phppgadmin \
--network db-network \
-e POSTGRES_HOST=ma-postgres \
-e POSTGRES_PORT=5432 \
-p 8081:80 \
dockage/phppgadmin:latest
```



Explication phpPgAdmin

Décortiquons cette commande :

```
--name phppgadmin          # Nom de l'interface web
--network db-network         # Même réseau que PostgreSQL
-e POSTGRES_HOST=ma-postgres # Se connecte à notre base
-e POSTGRES_PORT=5432        # Port standard PostgreSQL
-p 8081:80                  # Interface accessible sur port 8081
dockage/phppgadmin:latest   # Image avec interface web
```

 **Magie des réseaux** : phpPgAdmin peut contacter ma-postgres par son nom !



[Revenir au sommaire](#)

Étape 5 : Tester votre stack !

```
# Vérifier que tout tourne
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"

echo ""
echo "⌚ VOTRE STACK EST PRÊTE !"
echo "🌐 Interface web: http://localhost:8081"
echo "👤 Serveur: ma-postgres"
echo "🔑 User: docker"
echo "🔒 Password: formation123"
echo ""
echo "🌐 Ouvrez votre navigateur et connectez-vous !"
```



Étape 6 : Expérimenter avec les données

Dans l'interface web (<http://localhost:8081>) :

1. **Connectez-vous** avec les identifiants
2. **Créez une table** utilisateurs
3. **Ajoutez quelques données**
4. **Redémarrez PostgreSQL** : docker restart ma-postgres
5. **Vérifiez** que vos données sont toujours là !

```
# Redémarrage test
docker restart ma-postgres
sleep 5
echo "⌚ PostgreSQL redémarré, vos données sont-elles toujours là ?"
```



🔍 Étape 7 : Explorer les volumes

```
# Voir où Docker stocke vos données  
docker volume inspect postgres-data  
  
# Voir l'espace utilisé  
docker system df  
  
# Voir les fichiers de la base (depuis l'intérieur du container)  
docker exec ma-postgres ls -la /var/lib/postgresql/data
```

 **Comprendre :** Vos données sont **physiquement** stockées sur votre disque, pas dans le container !



Étape 8 : Nettoyage (optionnel)

```
# Script pour tout supprimer proprement
echo "🧹 Nettoyage de la stack PostgreSQL..."

# Arrêter les containers
docker stop phppgadmin ma-postgres

# Supprimer les containers
docker rm phppgadmin ma-postgres

# Supprimer le réseau
docker network rm db-network

# Supprimer les volumes (ATTENTION: perte des données!)
docker volume rm postgres-data postgres-logs
```



🏆 Bilan Niveau Simple

✓ Vous avez maîtrisé :

- Créer des **réseaux** Docker personnalisés
- Utiliser des **volumes** pour la persistance
- Connecter des **containers** entre eux
- Configurer avec des **variables d'environnement**
- Avoir une **interface visuelle** pour vos données

🚀 Prêt pour le niveau intermédiaire !



[Revenir au sommaire](#)

🟡 Exercice Niveau Intermédiaire

Stack WordPress Complète

🎯 Objectif : Site web professionnel avec base de données

Ce que vous allez construire :

- Site **WordPress** complet et fonctionnel
- Base de données **MySQL** dédiée
- Interface **phpMyAdmin** pour gérer la DB
- **Volumes** persistants pour tout sauvegarder
- **Réseau** sécurisé entre les services



[Revenir au sommaire](#)

Qu'est-ce que WordPress ?

WordPress est le CMS le plus populaire au monde :

- Utilise environ **40% des sites web** mondiaux
- Interface d'administration intuitive
- Milliers de thèmes et plugins
- Parfait pour blogs, sites vitrine, e-commerce

MySQL est sa base de données préférée :

- Stocker articles, utilisateurs, commentaires
- Base relationnelle très répandue
- **phpMyAdmin** permet de la gérer visuellement



🔧 Étape 1 : Créer l'environnement WordPress

```
# Environnement dédié au WordPress
docker network create wordpress-network
docker volume create mysql-data
docker volume create wordpress-data

# Vérifier la création
echo "🌐 Réseau créé:"
docker network ls | grep wordpress

echo "💾 Volumes créés:"
docker volume ls | grep -E "(mysql|wordpress)"
```



Étape 2 : Lancer MySQL pour WordPress

```
# Base de données MySQL optimisée pour WordPress
docker run -d \
--name mysql-wordpress \
--network wordpress-network \
-e MYSQL_ROOT_PASSWORD=root123 \
-e MYSQL_DATABASE=wordpress \
-e MYSQL_USER=wpuuser \
-e MYSQL_PASSWORD=wppass \
-v mysql-data:/var/lib/mysql \
--restart unless-stopped \
mysql:8.0
```



Configuration MySQL expliquée

```
# Analysons cette configuration MySQL :  
  
-e MYSQL_ROOT_PASSWORD=root123      # Mot de passe administrateur  
-e MYSQL_DATABASE=wordpress          # Base dédiée à WordPress  
-e MYSQL_USER=wpuser                 # Utilisateur WordPress  
-e MYSQL_PASSWORD=wppass             # Son mot de passe  
-v mysql-data:/var/lib/mysql         # Persistence des données  
--restart unless-stopped            # Redémarre auto (sauf arrêt manuel)  
mysql:8.0                            # Version stable de MySQL
```

 **Sécurité** : WordPress n'a accès qu'à sa base, pas aux autres !



🔧 Étape 3 : Attendre que MySQL soit prêt

```
# MySQL prend du temps à démarrer
echo "⌚ Démarrage de MySQL (peut prendre 30 secondes)..."
sleep 15

# Vérifier que MySQL accepte les connexions
docker exec mysql-wordpress mysqladmin ping -h localhost

# Voir les logs de démarrage
docker logs mysql-wordpress --tail 10
```

💡 Pourquoi attendre ? MySQL doit initialiser la base `wordpress` avant que WordPress se connecte !



[Revenir au sommaire](#)

Étape 4 : Lancer WordPress

```
# WordPress connecté à MySQL
docker run -d \
  --name mon-wordpress \
  --network wordpress-network \
  -e WORDPRESS_DB_HOST=mysql-wordpress \
  -e WORDPRESS_DB_USER=wpuser \
  -e WORDPRESS_DB_PASSWORD=wppass \
  -e WORDPRESS_DB_NAME=wordpress \
  -v wordpress-data:/var/www/html \
  -p 8080:80 \
  --restart unless-stopped \
  wordpress:latest
```



Configuration WordPress expliquée

```
# Configuration de WordPress :  
  
-e WORDPRESS_DB_HOST=mysql-wordpress      # Se connecte à notre MySQL  
-e WORDPRESS_DB_USER=wpuser                # Utilise notre utilisateur  
-e WORDPRESS_DB_PASSWORD=wppass            # Avec le bon mot de passe  
-e WORDPRESS_DB_NAME=wordpress            # Dans la bonne base  
-v wordpress-data:/var/www/html          # Fichiers WordPress persistants  
-p 8080:80                                # Accessible sur port 8080
```

🌐 Réseau magique : WordPress trouve MySQL via le nom mysql-wordpress !



Étape 5 : Ajouter phpMyAdmin pour la base

```
# Interface pour gérer la base MySQL
docker run -d \
--name mysql-admin \
--network wordpress-network \
-e PMA_HOST=mysql-wordpress \
-e PMA_USER=root \
-e PMA_PASSWORD=root123 \
-p 8081:80 \
phpmyadmin:latest

echo "💡 phpMyAdmin disponible sur: http://localhost:8081"
```



Étape 6 : Tester votre stack WordPress !

```
# Vérifier tous les services
echo "🔍 État de votre stack WordPress:"
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}" | grep -E "(wordpress|mysql)"

echo ""
echo "🌐 VOTRE STACK WORDPRESS EST PRÊTE !"
echo ""
echo "🌐 WordPress: http://localhost:8080"
echo "📊 phpMyAdmin: http://localhost:8081"
echo "👤 User: root | Password: root123"
echo ""
echo "🚀 Installez WordPress en suivant l'assistant !"
```



💡 Étape 7 : Installation WordPress complète

Dans votre navigateur :

1. Allez sur <http://localhost:8080>
2. Suivez l'assistant d'installation WordPress
3. Créez votre compte administrateur
4. Connectez-vous au tableau de bord WordPress

```
# Pendant l'installation, surveillez les logs  
docker logs mon-wordpress --follow
```

🎉 Premier article : Créez un article "Hello Docker World!" pour tester !



[Revenir au sommaire](#)

🔍 Étape 8 : Explorer la base de données

Dans phpMyAdmin (<http://localhost:8081>) :

1. **Connectez-vous** avec root/root123
2. **Sélectionnez** la base wordpress
3. **Explorez** les tables WordPress (wp_posts, wp_users, etc.)
4. **Trouvez** votre article dans wp_posts !

```
# Voir les tables WordPress depuis le terminal  
docker exec mysql-wordpress mysql -u wpuser -pwppass wordpress -e "SHOW TABLES;"
```



⌚ Étape 9 : Test de persistence

```
# Test ultime : redémarrer toute la stack
echo "⌚ Test de persistence - redémarrage de tout..."

docker restart mysql-wordpress mon-wordpress mysql-admin

# Attendre le redémarrage
sleep 20

echo "✅ Stack redémarrée !"
echo "🌐 Votre site: http://localhost:8080"
echo "MYSQL Votre base: http://localhost:8081"
echo ""
echo "⌚ Vos données sont-elles toujours là ?"
```



🏆 Bilan Niveau Intermédiaire

✓ Vous avez maîtrisé :

- **Stack multi-containers** complète et fonctionnelle
- **Communication** sécurisée via réseau personnalisé
- **Persistence** totale avec volumes dédiés
- **Variables d'environnement** pour la configuration
- **Interface d'administration** pour la base de données
- **Restart policies** pour la robustesse

🚀 Niveau avancé : environnements DevOps !



[Revenir au sommaire](#)

Exercice Niveau Avancé

Environnement DevOps Multi-Distributions

🎯 Objectif : Simuler un environnement de production DevOps

Ce que vous allez construire :

- **Cluster** de containers avec différentes distributions Linux
- **Workspace partagé** entre tous les containers
- **Logs centralisés** pour le monitoring
- **Outils DevOps** installés et configurés
- **Communication** inter-containers testée



[Revenir au sommaire](#)

Pourquoi plusieurs distributions ?

En production DevOps on gère souvent :

- **CentOS/RHEL** : Serveurs d'entreprise traditionnels
- **Fedora** : Environnements de développement avec outils récents
- **Rocky Linux** : Alternative moderne à CentOS
- **Alpine** : Containers ultra-légers pour les microservices

L'exercice simule :

- Environnement **hétérogène** réaliste
- **Partage de fichiers** entre serveurs
- **Centralisation des logs** comme en production
- **Outils** que vous utiliseriez vraiment



🔧 Étape 1 : Créer l'environnement DevOps

```
# Infrastructure DevOps
docker network create devops-network
docker volume create shared-workspace
docker volume create logs-centralized
docker volume create tools-shared

# Créer un dossier local de travail
mkdir -p ~/docker-devops
cd ~/docker-devops

echo "🏗 Infrastructure DevOps créée"
docker network ls | grep devops
docker volume ls | grep -E "(shared|logs|tools)"
```



Étape 2 : Container CentOS - Serveur Legacy

```
# Serveur CentOS avec outils DevOps traditionnels
docker run -d \
  --name centos-legacy \
  --network devops-network \
  -v shared-workspace:/workspace \
  -v logs-centralized:/var/log/shared \
  -v tools-shared:/opt/tools \
  --hostname centos-srv \
  --privileged \
  centos:7 \
/bin/bash -c "
  yum update -y &&
  yum install -y git vim curl wget htop net-tools &&
  echo 'CentOS Legacy Server Ready' > /var/log/shared/centos.log &&
  tail -f /dev/null
```



Configuration CentOS expliquée

```
# Analysons ce container CentOS :
```

```
--hostname centos-srv          # Nom réseau identifiable
--privileged                     # Accès étendu (nécessaire pour certains outils)
-v shared-workspace:/workspace   # Dossier partagé pour les projets
-v logs-centralized:/var/log/shared # Logs centralisés
-v tools-shared:/opt/tools       # Outils partagés
yum install -y git vim curl...    # Outils DevOps essentiels
echo '...' > /var/log/shared/...   # Log de démarrage
tail -f /dev/null                 # Garde le container actif
```



🔧 Étape 3 : Container Fedora - Environnement Modern

```
# Fedora avec outils modernes de développement
docker run -d \
  --name fedora-modern \
  --network devops-network \
  -v shared-workspace:/workspace \
  -v logs-centralized:/var/log/shared \
  -v tools-shared:/opt/tools \
  --hostname fedora-dev \
  -p 9090:9090 \
  fedora:38 \
  /bin/bash -c "
    dnf update -y &&
    dnf install -y git vim curl wget htop python3 nodejs npm docker &&
    echo 'Fedora Modern Environment Ready' > /var/log/shared/fedora.log &&
    python3 -m http.server 9090 --directory /workspace &
```

🚀 **Bonus :** Fedora expose un serveur web sur le port 9090 pour partager des fichiers !



Étape 4 : Container Rocky Linux - Serveur Production

```
# Rocky Linux comme serveur web de production
docker run -d \
--name rocky-production \
--network devops-network \
-v shared-workspace:/workspace \
-v logs-centralized:/var/log/shared \
-v tools-shared:/opt/tools \
--hostname rocky-prod \
-p 8080:80 \
rockylinux:9 \
/bin/bash -c "
dnf install -y httpd git curl vim &&
echo '<h1>🌐 Rocky Linux Production Server</h1><p>Shared workspace available</p>' > /var/www/html/index.html
echo 'Rocky Production Server Ready' > /var/log/shared/rocky.log &&
httpd -D FOREGROUND"
```

🌐 **Serveur web** : Rocky Linux expose un serveur HTTP sur le port 8080 !



🔧 Étape 5 : Attendre que tout démarre

```
# Laisser le temps aux installations
echo "⌚ Installation des packages sur toutes les distributions..."
sleep 30

# Vérifier que tous les containers tournent
echo "🔍 État de l'environnement DevOps:"
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
grep -E "(centos|fedora|rocky)"

echo ""
echo "🌐 Services exposés:"
echo "📁 Partage de fichiers (Fedora): http://localhost:9090"
echo "🌐 Serveur web (Rocky): http://localhost:8080"
```



Étape 6 : Tester la communication inter-containers

```
# Test de connectivité réseau
echo "🌐 Test de la communication inter-containers:"

# CentOS ping Fedora
docker exec centos-legacy ping -c 3 fedora-dev

# Fedora ping Rocky
docker exec fedora-modern ping -c 3 rocky-prod

# Rocky ping CentOS
docker exec rocky-production ping -c 3 centos-srv

echo "✅ Communication réseau testée!"
```



Étape 7 : Tester le workspace partagé

```
# Créer un fichier depuis CentOS
docker exec centos-legacy bash -c "
  echo '# Projet DevOps 2025' > /workspace/README.md
  echo 'Ce fichier est partagé entre tous les containers' >> /workspace/README.md
  echo 'Créé depuis CentOS Legacy' >> /workspace/README.md
  date >> /workspace/README.md
"

# Le lire depuis Fedora
echo "■ Contenu lu depuis Fedora:"
docker exec fedora-modern cat /workspace/README.md

# Le modifier depuis Rocky
docker exec rocky-production bash -c "
  echo 'Modifié depuis Rocky Production' >> /workspace/README.md
```



🔍 Étape 8 : Explorer les logs centralisés

```
# Voir tous les logs de démarrage
echo "📋 Logs centralisés de toutes les distributions:"
docker exec centos-legacy ls -la /var/log/shared/

echo ""
echo "📄 Contenu des logs:"
docker exec centos-legacy cat /var/log/shared/centos.log
docker exec fedora-modern cat /var/log/shared/fedora.log
docker exec rocky-production cat /var/log/shared/rocky.log

# Ajouter un log personnalisé
docker exec fedora-modern bash -c "
    echo 'Test de monitoring - $(date)' >> /var/log/shared/monitoring.log
"
```



⌚ Étape 9 : Simulation DevOps réaliste

```
# Créer un script de déploiement partagé
docker exec centos-legacy bash -c "
#!/bin/bash
echo '🚀 Déploiement automatisé'
echo 'Serveur: \$(hostname)'
echo 'Distribution: \$(cat /etc/os-release | grep PRETTY_NAME)'
echo 'Date: \$(date)'
echo 'Utilisateur: \$(whoami)'
echo '✅ Déploiement terminé'
chmod +x /workspace/deploy.sh
"

# Exécuter le script depuis chaque distribution
echo "🚀 Exécution du script de déploiement:"
echo ""
```



🏆 Bilan Niveau Avancé

✓ Environnement DevOps maîtrisé :

- **Multi-distributions** Linux en communication
- **Workspace partagé** pour les projets communs
- **Logs centralisés** pour le monitoring
- **Scripts de déploiement** cross-platform
- **Simulation production** réaliste

🔥 Prêt pour le niveau expert avec monitoring !



[Revenir au sommaire](#)

🔥 Exercice BONUS - Expert

Stack de Monitoring Professionnel

🎯 Objectif : Monitoring comme en production

Ce que vous allez construire :

- **Prometheus** : Collecte de métriques
- **Grafana** : Dashboards visuels magnifiques
- **cAdvisor** : Métriques containers
- Stack complète de **monitoring production**



[Revenir au sommaire](#)

Qu'est-ce que le monitoring moderne ?

Prometheus  :

- Base de données de **métriques** temporelles
- Utilisé par Google, SoundCloud, DigitalOcean
- Collecte automatique depuis vos applications
- Système d'**alertes** intégré

Grafana  :

- **Dashboards** magnifiques et interactifs
- Graphiques en temps réel
- Utilisé par PayPal, eBay, Intel
- Interface web moderne et intuitive



[Revenir au sommaire](#)

🔧 Étape 1 : Préparer l'environnement monitoring

```
# Infrastructure de monitoring
docker network create monitoring-network
docker volume create prometheus-data
docker volume create grafana-data

# Créer le dossier de configuration
mkdir -p ~/monitoring-stack
cd ~/monitoring-stack

echo "📊 Infrastructure monitoring créée"
```



🔧 Étape 2 : Configuration Prometheus

```
# Créez la configuration Prometheus
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']

echo "⚙️ Configuration Prometheus créée"
```

📋 Configuration expliquée :

- `scrape_interval: 15s` : Collecte les métriques toutes les 15 secondes
- Surveille **Prometheus lui-même** et **cAdvisor**



Étape 3 : Lancer Prometheus

```
# Prometheus avec configuration personnalisée
docker run -d \
  --name prometheus \
  --network monitoring-network \
  -p 9090:9090 \
  -v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml \
  -v prometheus-data:/prometheus \
  --restart unless-stopped \
  prom/prometheus:latest \
  --config.file=/etc/prometheus/prometheus.yml \
  --storage.tsdb.path=/prometheus \
  --web.console.libraries=/etc/prometheus/console_libraries \
  --web.console.templates=/etc/prometheus/consoles \
  --web.enable-lifecycle
```



🔧 Étape 4 : Lancer cAdvisor (métriques containers)

```
# Node Exporter pour métriques système
docker run -d \
--name node-exporter \
--network monitoring-network \
-p 9100:9100 \
--restart unless-stopped \
prom/node-exporter:latest \
--path.rootfs=/host \
--collector.filesystem.mount-points-exclude="^/(sys|proc|dev|host|etc)(\$|/)"
```



🔧 Étape 5 : Lancer Prometheus

```
# Prometheus avec configuration personnalisée
docker run -d \
  --name cadvisor \
  --network monitoring-network \
  -p 8080:8080 \
  --restart unless-stopped \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:ro \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker:/var/lib/docker:ro \
  --volume=/dev/disk:/dev/disk:ro \
  gcr.io/cadvisor/cadvisor:latest
```

💡 Qu'est-ce que cAdvisor ?

- Développé par **Google** pour monitorer les containers
- Collecte métriques de **tous vos containers** Docker
- CPU, RAM, réseau, I/O par container



🔧 Étape 6 : Lancer Grafana avec dashboard

```
# Grafana avec storage persistant
docker run -d \
  --name grafana \
  --network monitoring-network \
  -p 3000:3000 \
  -v grafana-data:/var/lib/grafana \
  -e GF_SECURITY_ADMIN_PASSWORD=admin123 \
  -e GF_USERS_ALLOW_SIGN_UP=false \
  --restart unless-stopped \
  grafana/grafana:latest
```

🎨 **Grafana** va créer de magnifiques graphiques avec toutes ces métriques !



Étape 7 : Vérifier votre stack monitoring

```
# Attendre que tout démarre
echo "⌚ Démarrage de la stack monitoring (30 secondes)..."
sleep 30

# Vérifier tous les services
echo "📊 État de votre stack de monitoring:"
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}" | grep -E "(prometheus|grafana|node-exporter|cAdvisor)"

echo ""
echo "🌐 STACK DE MONITORING PRÊTE !"
echo ""
echo "📈 Prometheus: http://localhost:9090"
echo "📊 Grafana: http://localhost:3000"
echo "👤 User: admin | Password: admin123"
echo "🌐 cAdvisor: http://localhost:8080"
```



Étape 8 : Configuration Grafana (Hands-on)

Dans Grafana (<http://localhost:3000>) :

1. Connectez-vous avec admin/admin123

2. Ajoutez Prometheus comme source de données :

- URL : http://prometheus:9090
- Click "Save & Test"

3. Importez des dashboards préconfigurés :

- Docker containers : Dashboard ID 193

```
# Pendant que vous configurez, surveillez les métriques
echo "⚠️ Métriques en cours de collecte..."
docker logs prometheus --tail 10
```



[Revenir au sommaire](#)

Étape 9 : Générer de l'activité à moniturer

```
# Créer de l'activité pour voir les métriques bouger
echo "🔥 Génération d'activité pour le monitoring..."

# Lancer quelques containers gourmands
docker run -d --name stress-test alpine:latest \
    sh -c "while true; do echo 'generating load...'; sleep 1; done"

docker run -d --name cpu-test alpine:latest \
    sh -c "while true; do dd if=/dev/zero of=/dev/null bs=1M count=100; done"

# Surveiller en temps réel
echo "📊 Observez vos dashboards Grafana maintenant !"
echo "🔍 Vous devriez voir l'activité CPU et containers augmenter"
```



🔍 Étape 10 : Exploration des métriques

Dans Prometheus (<http://localhost:9090>) :

1. Explorez les métriques disponibles
2. Essayez ces requêtes dans l'onglet "Graph" :

```
# CPU usage
100 - (avg by(instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)

# Memory usage
100 * (1 - ((node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)))

# Container count
count(container_last_seen)
```



🏆 Félicitations ! Stack Expert Maîtrisée

✓ Vous avez construit une infrastructure de monitoring professionnelle :

- **Prometheus** : Base de données métriques comme Netflix, Uber
- **Grafana** : Dashboards visuels comme Tesla, PayPal
- **cAdvisor** : Monitoring containers par Google
- **Configuration** production-ready avec persistence

🎯 Compétences acquises :

- Architecture **observabilité** moderne
- Configuration **time-series databases**
- **Dashboards** interactifs professionnels
- **Métriques** système et containers
- **Stack** utilisée dans le monde réel



[Revenir au sommaire](#)



Récapitulatif Complet des Exercices

Vert Niveau Simple - Base de données

- Réseaux Docker personnalisés
- Volumes persistants
- Communication inter-containers
- Interface web pour bases de données

Jaune Niveau Intermédiaire - Site web complet

- Stack multi-containers (WordPress + MySQL)
- Variables d'environnement avancées
- Restart policies
- Administration de base de données



[Revenir au sommaire](#)

🔴 Niveau Avancé - DevOps multi-distributions

- Environnement hétérogène Linux
- Workspace partagé
- Logs centralisés
- Scripts cross-platform

🔥 Niveau Expert - Monitoring professionnel

- Prometheus + Grafana
- Métriques système et containers
- Dashboards interactifs
- Observabilité production



[Revenir au sommaire](#)

Vous êtes maintenant prêts pour Docker Compose !

Ces exercices vous ont préparés à :

- **Orchestrer** des applications complexes
- **Gérer** des environnements multi-services
- **Monitorer** vos infrastructures
- **Automatiser** vos déploiements

Prochain module : Docker Compose pour simplifier tout ça ! 🎶



[Revenir au sommaire](#)

Docker Compose - Orchestration Multi- Containers

Orchestrez vos applications multi-containers

Docker Compose permet de définir et gérer des applications multi-containers avec un seul fichier de configuration. Fini les commandes `docker run` interminables !



[Revenir au sommaire](#)

Pourquoi Docker Compose ? 🤔

Problème sans Compose

```
# Lancer une stack web manuellement
docker network create app-network
docker run -d --name database --network app-network postgres:15
docker run -d --name redis-cache --network app-network redis:alpine
docker run -d --name web-app --network app-network -p 3000:3000 mon-app
docker run -d --name nginx-proxy --network app-network -p 80:80 nginx
```



Problème sans Compose (suite) 🚨

⚠️ **Problèmes** : Complexe, répétitif, difficile à maintenir !



[Revenir au sommaire](#)

Solution avec Compose ✨

Un seul fichier = toute votre infrastructure

```
# docker-compose.yml
version: '3.8'
services:
  database:
    image: postgres:15
    environment:
      POSTGRES_DB: myapp
      POSTGRES_PASSWORD: secret

  redis:
    image: redis:alpine

  web:
    build: .
    ports:
```



[Revenir au sommaire](#)

Solution avec Compose (résultat) 🚀

Une seule commande : docker compose up 🚀



[Revenir au sommaire](#)

Syntaxe Moderne 2025 ⚡

Nouvelle syntaxe (recommandée)

```
# ✓ Syntaxe moderne Docker 2025
docker compose up -d
docker compose down
docker compose logs -f
docker compose restart web
```



Syntaxe Moderne 2025 (suite)

Ancienne syntaxe (dépréciée)

```
#  Ancienne syntaxe (à éviter)  
docker-compose up -d  
docker-compose down
```



Syntaxe Moderne 2025 (conclusion)



Docker intègre maintenant Compose nativement !



[Revenir au sommaire](#)

Anatomie d'un fichier Compose

```
version: '3.8'

services: # Définition des containers
  web:
    build: .
    ports:
      - '3000:3000'
networks: # Réseaux personnalisés
  app-network:
    driver: bridge

volumes: # Volumes partagés
  db-data:
    driver: local
secrets: # Gestion des secrets
```



Commandes Essentielles 🎯

Cycle de vie complet

Commande	Description	Exemple
<code>docker compose up</code>	Démarrer les services	<code>docker compose up -d</code>
<code>docker compose down</code>	Arrêter et supprimer	<code>docker compose down</code>
<code>docker compose ps</code>	Status des services	<code>docker compose ps</code>



[Revenir au sommaire](#)

Commandes Essentielles (suite)

Commande	Description	Exemple
docker compose logs	Voir les logs	docker compose logs -f web
docker compose exec	Exécuter dans un service	docker compose exec web bash
docker compose restart	Redémarrer	docker compose restart web



Application complète Next.js + PostgreSQL + Nginx

```
version: '3.8'

services:
  # Base de données PostgreSQL
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: webapp
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret123
    volumes:
      - postgres_data:/var/lib/postgresql/data
  healthcheck:
    test: ['CMD-SHELL', 'pg_isready -U app']
    interval: 30s
```



Dockerfile pour notre app Next.js :

```
FROM node:18-alpine AS base

# Installer les dépendances seulement quand nécessaire
FROM base AS deps
WORKDIR /app

# Installer les dépendances basées sur le gestionnaire de packages préféré
COPY package.json package-lock.json ./
RUN npm ci;
# npm install

# Rebuild le code source seulement quand nécessaire
# 2 eme stage, permet de différencier le build de l'app et le runner
FROM base AS builder
# je me base sur base et je crée un stage builder
```



nginx.conf pour le reverse proxy :

```
events {  
    worker_connections 1024;  
}  
  
#  
# on définit le upstream  
http {  
    upstream nextjs {  
        server web:3000;  
    }  
  
    # on définit le server  
    server {  
        listen 80;  
        server_name localhost;
```



Créer l'application Next.js 🚀

Commandes pour créer et préparer l'app :

```
# Créer l'application Next.js avec TypeScript
npx create-next-app@latest mon-app-nextjs --typescript --tailwind --eslint --app --src-dir --import-always

# Aller dans le dossier
cd mon-app-nextjs

# Ajouter la configuration pour standalone
echo 'module.exports = {
  output: "standalone",
  experimental: {
    outputFileTracingRoot: require("path").join(__dirname, "../../"),
  },
}' > next.config.js

# Ajouter la dépendance PostgreSQL
```



[Revenir au sommaire](#)

Cas Concret : Magie de Compose ✨

Une seule commande :

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

Compose fait tout automatiquement :

1. 🚧 Build l'image Next.js à partir du Dockerfile
2. 🚶 Lance PostgreSQL avec healthcheck
3. 🔗 Connecte l'app à la base via le réseau
4. 🌐 Configure Nginx comme reverse proxy
5. ⚡ Démarre toute la stack sur le port 80

Résultat : Stack complète Next.js + PostgreSQL + Nginx fonctionnelle !



[Revenir au sommaire](#)

Variables d'environnement 🔧

Fichier .env pour la configuration

```
# .env
NODE_ENV=development
POSTGRES_DB=webapp
POSTGRES_USER=app
POSTGRES_PASSWORD=secret123
WEB_PORT=80
```

Utilisation dans docker-compose.yml :

```
services:
  web:
    environment:
      NODE_ENV: ${NODE_ENV}
      DATABASE_URL: postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db:5432/${POSTGRES_DB}

  nginx:
    ports:
      - '${WEB_PORT}:80'
```



[Revenir au sommaire](#)

Profiles et environnements



Gestion multi-environnements

```
services:  
  web:  
    image: mon-app:latest  
  
  # Service de développement uniquement  
  dev-tools:  
    image: adminer  
    ports:  
      - '8080:8080'  
    profiles:  
      - dev  
  
  # Service de monitoring en production  
  monitoring:  
    image: grafana/grafana
```



[Revenir au sommaire](#)

Commandes :

```
# Développement  
docker compose --profile dev up
```

```
# Production  
docker compose --profile prod up
```



Scaling et Load Balancing

Mise à l'échelle facile

```
# Lancer 3 instances du service web  
docker compose up --scale web=3  
  
# Avec un load balancer  
docker compose up --scale web=3 --scale worker=5
```

Configuration Nginx pour load balancing :

```
upstream nextjs {  
    server web_1:3000;  
    server web_2:3000;  
    server web_3:3000;  
}
```



Bonnes Pratiques 2025 ✓

Recommandations modernes

🔒 Sécurité :

```
services:  
db:  
  image: postgres:15-alpine  
  environment:  
    POSTGRES_PASSWORD_FILE: /run/secrets/db-password  
  secrets:  
    - db-password
```



Monitoring :

```
services:  
  web:  
    healthcheck:  
      test: [CMD, curl, -f, http://localhost:3000/health]  
      interval: 30s  
      timeout: 10s  
      retries: 3  
      start_period: 60s
```



Debugging et Troubleshooting



Commandes utiles pour déboguer

```
# Voir les services en cours  
docker compose ps  
  
# Logs en temps réel  
docker compose logs -f  
  
# Inspecter un service spécifique  
docker compose logs web
```

Suite des commandes :

```
# Reconstruire les images  
docker compose build --no-cache  
  
# Valider la configuration  
docker compose config  
  
# Nettoyer complètement  
docker compose down -v --remove-orphans
```



[Revenir au sommaire](#)

Intégration CI/CD 🚀

Production avec secrets externes :

```
services:  
  web:  
    image: registry.company.com/mon-app:${VERSION}  
    environment:  
      DATABASE_URL: ${DATABASE_URL}  
    deploy:  
      replicas: 3  
      resources:  
        limits:  
          memory: 512M  
        reservations:  
          memory: 256M
```

Déploiement CI/CD :

```
# Déploiement CI/CD  
export VERSION=v1.2.3  
docker compose -f docker-compose.prod.yml up -d
```



[Revenir au sommaire](#)

Récapitulatif

Ce que vous maîtrisez maintenant

- Orchestration multi-containers** avec un seul fichier
- Syntaxe moderne Docker Compose 2025**
- Gestion des environnements** avec profiles et .env
- Build d'images personnalisées** avec Dockerfile Next.js
- Scaling** et load balancing
- Bonnes pratiques** de sécurité et monitoring
- Debugging** et troubleshooting

 **Prêt pour l'exercice pratique !**

Vous pouvez maintenant créer des applications multi-containers complètes !



[Revenir au sommaire](#)

Exercices Docker Compose



3 niveaux DevOps progressifs

Orchestrez plusieurs containers facilement !



Exercices Express (Warm-up)

3 exercices rapides avec images officielles

Avant les exercices principaux, des exercices courts pour maîtriser les bases !



[Revenir au sommaire](#)

Exercice Express 1 : Ma Première Stack

Stack super simple avec 2 services (15 min) au choix :

- nginx + redis
- nginx + mysql
- nginx + postgres
- nginx + mongo
- nginx + elasticsearch
- nginx + kibana

Ce qu'on apprend : Premier docker-compose.yml, services liés

```
# Créer docker-compose.yml
version: '3.8'

services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
  cache:
    image: votre_choix:alpine
```



[Revenir au sommaire](#)

Exercice Express 2 : Stack Base de Données

Postgres + Adminer pour explorer une BDD (20 min)

Ce qu'on apprend : Variables d'environnement, volumes, interface web

```
# docker-compose.yml plus sophistiqué
version: '3.8'

services:
  database:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: testdb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password123
    volumes:
      - db_data:/var/lib/postgresql/data

  adminer:
    image: adminer:latest
```

```
# Test complet
docker compose up -d
echo "🌐 Interface BDD: http://localhost:8081"
```



[Revenir au sommaire](#)

Exercice Express 3 : Stack Monitoring Simple

Prometheus + Grafana pour surveiller (25 min)

Ce qu'on apprend : Stack monitoring, réseaux personnalisés

```
# docker-compose.yml monitoring
version: '3.8'

services:
  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"
    networks:
      - monitoring

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
```

```
# Déploiement monitoring
docker compose up -d
echo "Metrics: http://localhost:9090"
```



[Revenir au sommaire](#)



Exercices Principaux Détailés



[Revenir au sommaire](#)

Exercice Niveau Simple

WordPress + MySQL (Stack de blog)

Objectif : Déployer un blog WordPress avec base de données

Consignes :

1. Utiliser l'image officielle `wordpress:latest`
2. Base de données `mysql:8.0`
3. Configurer les volumes pour la persistance
4. Accessible sur le port 8080



[Revenir au sommaire](#)

Correction Niveau Simple

```
version: '3.8'

services:
  wordpress:
    image: wordpress:latest
    ports:
      - '8080:80'
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress123
      WORDPRESS_DB_NAME: wordpress
    volumes:
      - wordpress_data:/var/www/html
depends_on:
```



● Test et Gestion Simple

Déploiement et vérification

```
# Lancer la stack
docker compose up -d

# Vérifier les services
docker compose ps

# Voir les logs
docker compose logs wordpress
docker compose logs db

# Tester l'accès
echo "🌐 WordPress: http://localhost:8080"
curl -I http://localhost:8080

# Arrêter proprement
```

✓ **Résultat :** Blog WordPress fonctionnel avec persistance des données



[Revenir au sommaire](#)

Exercice Niveau Intermédiaire

Stack NGINX + Node.js + Redis

Objectif : Application web avec proxy et cache

Consignes :

1. Proxy NGINX sur le port 80
2. Application Node.js (image node:18-alpine)
3. Cache Redis pour les sessions
4. Réseaux séparés frontend/backend



[Revenir au sommaire](#)

Correction Niveau Intermédiaire

Configuration avec réseaux

```
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - '80:80'
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - app
    networks:
      - frontend

  app:
```



Configuration NGINX Proxy

Fichier nginx.conf automatique

```
# Créer la configuration NGINX
mkdir -p nginx
# 2. Créer le fichier nginx.conf
events {
    worker_connections 1024;
}

http {
    upstream app {
        server app:80;
    }

    server {
        listen 80;
```



🟡 Test Stack Intermédiaire

```
# Déployer la stack complète
docker compose up -d

# Vérifier tous les services
docker compose ps

# Tester le proxy
curl http://localhost/
curl http://localhost/health

# Vérifier Redis
docker compose exec redis redis-cli ping

# Monitoring des logs
docker compose logs -f nginx
```

✓ **Résultat :** Stack 3-tiers avec proxy et cache



Exercice Niveau Avancé

Monitoring Stack (Prometheus + Grafana)

Objectif : Infrastructure de monitoring complète

Consignes :

1. Prometheus serveur de monitoring
2. Grafana interface de visualisation
3. Node Exporter pour les métriques système
4. AlertManager pour les alertes par email



[Revenir au sommaire](#)

Correction Niveau Avancé

Stack de monitoring complète

```
version: '3.8'

services:
  prometheus:
    image: prom/prometheus:latest
    ports:
      - '9090:9090'
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
```



Configuration Prometheus

Configuration automatique

```
# Configuration Prometheus
# 2. Créer le fichier prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  # - "first_rules.yml"

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'node-exporter'
```



[Revenir au sommaire](#)

Déploiement et Monitoring

```
# Déployer la stack de monitoring
docker compose up -d

# Attendre que tout soit prêt
sleep 30

# Vérifier tous les services
docker compose ps

# URLs d'accès
echo "🌐 Prometheus: http://localhost:9090"
echo "📝 Grafana: http://localhost:3000 (admin/admin123)"
echo "🔔 AlertManager: http://localhost:9093"
echo "🌐 Node Exporter: http://localhost:9100"
```

✓ **Résultat :** Infrastructure de monitoring production-ready



Exercice Niveau Expert

Stack DevOps Complète (GitLab + Registry + Runner)

Objectif : Plateforme CI/CD complète avec GitLab

Consignes :

1. GitLab CE pour le code source
2. GitLab Registry pour les images
3. GitLab Runner pour les pipelines
4. PostgreSQL comme base de données



[Revenir au sommaire](#)

Correction Niveau Expert

Infrastructure GitLab complète

```
version: '3.8'

services:
  gitlab:
    image: gitlab/gitlab-ce:latest
    hostname: gitlab.local
    ports:
      - '80:80'
      - '443:443'
      - '2222:22'
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'http://gitlab.local'
        gitlab_rails['gitlab_shell_ssh_port'] = 2222
        postgresql['enable'] = false
```



Configuration GitLab Expert

Scripts de configuration automatique

```
# Script de déploiement GitLab
# 2. Créer le fichier deploy-gitlab.sh
#!/bin/bash

echo "🚀 Déploiement GitLab DevOps Stack..."

# Ajout de l'hostname local
echo "127.0.0.1 gitlab.local" | sudo tee -a /etc/hosts

# Démarrage de la stack
docker compose up -d

echo "⌚ GitLab démarre... (peut prendre 5-10 minutes)"
echo "📊 Monitoring du démarrage:"
```



Configuration GitLab Runner

Registration automatique du Runner

```
# Script de configuration du Runner
# 2. Créer le fichier setup-runner.sh
#!/bin/bash

echo "🏃 Configuration GitLab Runner..."

# Récupérer le token de registration
echo "1. Aller sur http://gitlab.local/admin/runners"
echo "2. Copier le registration token"
echo "3. Exécuter la commande suivante:"

echo ""
echo "docker compose exec gitlab-runner gitlab-runner register \\\"
echo "  --non-interactive \\\"
echo "  --url http://gitlab.local \\\""
```



Test Stack DevOps Complète

```
# Déploiement complet
./deploy-gitlab.sh

# Attendre le démarrage complet
sleep 300

# Vérifications
echo "🧪 Tests de la stack DevOps..."

# Test GitLab
curl -I http://gitlab.local && echo "✅ GitLab accessible"

# Vérifier PostgreSQL
docker compose exec postgresql pg_isready -U gitlab && echo "✅ PostgreSQL OK"
```

✓ **Résultat :** Plateforme DevOps complète avec CI/CD intégré



[Revenir au sommaire](#)

Récapitulatif Exercices Compose



Compétences acquises avec images officielles

🟢 Niveau Simple :

- WordPress + MySQL (images officielles)
- Volumes et persistance
- Variables d'environnement
- Commandes de base

🟡 Niveau Intermédiaire :

- NGINX + Node.js + Redis
- Réseaux personnalisés
- Configuration de proxy
- Monitoring des services



[Revenir au sommaire](#)

Récapitulatif Compose (suite)



🔴 Niveau Avancé :

- Stack Prometheus + Grafana (monitoring)
- Node Exporter + AlertManager
- Configuration avancée
- Health checks

🔵 Niveau Expert :

- GitLab CE + PostgreSQL + Runner
- Plateforme CI/CD complète
- Infrastructure DevOps
- Pipelines automatisés

 **Docker Compose maîtrisé avec images réelles !**

Prochaine étape : Ansible pour automatiser le déploiement !



[Revenir au sommaire](#)

Dockerfile & Images Docker

Dockerfile & Images Docker

Créer vos propres images personnalisées

Un **Dockerfile** est un fichier de recette qui automatise la création d'images Docker. Maîtrisons la création d'images optimisées pour la production.



[Revenir au sommaire](#)

Qu'est-ce qu'un Dockerfile ?



La recette de cuisine pour votre application

Un **Dockerfile** est un fichier texte contenant des instructions :

- **Recette** : Liste d'étapes pour construire votre application
- **Instructions** : Commandes automatisées (installer, copier, configurer...)
- **Reproductible** : Même résultat sur n'importe quel serveur
- **Packager** : Transforme votre code en image Docker prête à l'emploi

Analogie : C'est comme une recette de cuisine détaillée que n'importe qui peut suivre pour obtenir le même plat !



Qu'est-ce qu'une Image Docker ?

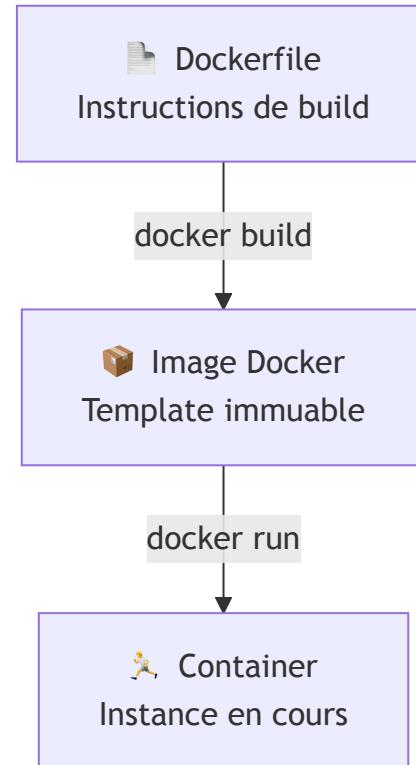
Le modèle prêt à utiliser

Une **Image Docker** est un template immuable :

-  **Template** : Modèle figé de votre application
-  **Couches** : Empilage d'instructions du Dockerfile
-  **Stockage** : Sauvegardée et réutilisable
-  **Base** : Sert à créer des containers

Analogie : C'est comme un moule à gâteau - une fois créé, vous pouvez faire autant de gâteaux identiques que vous voulez !

Relation Image ↔ Container



Qu'est-ce qu'une Couche (Layer) ? 🍽️

L'empilement intelligent

Chaque instruction Dockerfile crée une **couche** :

- 🍽️ **Empilement** : Chaque RUN, COPY, ADD = une nouvelle couche
- 💾 **Cache** : Les couches non modifiées sont réutilisées
- ⚡ **Performance** : Builds plus rapides grâce au cache
- 🔧 **Taille** : Moins de couches = image plus légère

Analogie : C'est comme un mille-feuille - chaque instruction ajoute une couche, et on peut réutiliser les couches du bas !



Dockerfile moderne - Structure type



```
# 1. Image de base optimisée
FROM node:20-alpine

# 2. Métadonnées
LABEL maintainer="dev@myapp.com" version="1.0.0"

# 3. Variables d'environnement
ENV NODE_ENV=production \
    PORT=3000

# 4. Répertoire de travail
WORKDIR /app

# 5. Dépendances (ordre optimal pour le cache)
COPY package*.json ./
```



Pourquoi FROM ?

La fondation de votre application

FROM définit l'image de base sur laquelle construire :

-  **Fondation** : Le système d'exploitation de base
-  **Outils** : Environnement et outils pré-installés
-  **Spécialisée** : Choisir selon votre technologie
-  **Optimisée** : Images Alpine = plus légères et sécurisées

Analogie : C'est comme choisir un terrain avec ou sans maison dessus pour construire !



Instructions essentielles

FROM - Images de base recommandées 2025

```
FROM node:20-alpine          # Node.js optimisé
FROM python:3.12-slim        # Python production-ready
FROM openjdk:21-jre-slim     # Java moderne
FROM nginx:1.25-alpine       # Serveur web performant
FROM postgres:16-alpine      # Base de données légère
```

Évitez ubuntu:latest - préférez des images spécialisées et taguées !



Pourquoi COPY vs ADD ?

La différence importante

COPY et ADD transfèrent des fichiers, mais différemment :

-  **COPY** : Simple transfert de fichiers (recommandé)
-  **ADD** : Transfert + fonctions spéciales (archives, URLs)
-  **Clarté** : COPY est plus explicite et prévisible
-  **Sécurité** : COPY évite les surprises

Analogie : COPY = photocopieuse simple, ADD = photocopieuse avec scanner et fax intégrés !



COPY vs ADD – Bonnes pratiques

COPY (recommandé dans 95% des cas)

```
# ✓ Ordre optimal pour le cache Docker
COPY package*.json ./          # Dépendances d'abord
RUN npm install
COPY . .                      # Code source après

# ✓ Copie avec permissions
COPY --chown=appuser:appgroup . .
```

ADD (cas spéciaux uniquement)

```
# Pour extraire des archives automatiquement
ADD release.tar.gz /app/
```



Pourquoi optimiser RUN ? ⚡

L'importance des couches

Chaque RUN crée une nouvelle couche :

- 🧩 **Multiplication** : Plus de RUN = plus de couches = image plus lourde
- 💾 **Cache** : Grouper les commandes optimise le cache
- 🖌️ **Nettoyage** : Supprimer les fichiers temporaires dans la même couche
- ⚡ **Performance** : Images plus légères = déploiements plus rapides

Analogie : C'est comme ranger sa chambre - mieux vaut tout faire d'un coup que laisser traîner !



RUN - Optimisation des couches ⚡

Mauvais exemple ✗

```
RUN apt-get update  
RUN apt-get install -y curl  
RUN apt-get install -y git  
RUN rm -rf /var/lib/apt/lists/*
```

Bon exemple ✓

```
RUN apt-get update && \  
    apt-get install -y curl git && \  
    rm -rf /var/lib/apt/lists/* && \  
    apt-get clean
```

Une seule couche = image plus légère !



Pourquoi ENV et ARG ?

La configuration flexible

ENV et ARG permettent la personnalisation :

-  **ARG** : Variables temporaires pour le build uniquement
-  **ENV** : Variables persistantes dans le container
-  **Flexibilité** : Même Dockerfile pour différents environnements
-  **Réutilisabilité** : Paramétrier sans modifier le code

Analogie : ARG = note temporaire pour le cuisinier, ENV = réglage permanent du four !



ENV et ARG - Configuration



```
# ARG : Variables de build uniquement
ARG BUILD_VERSION=1.0.0
ARG NODE_ENV=production

# ENV : Variables disponibles au runtime
ENV VERSION=$BUILD_VERSION \
    NODE_ENV=$NODE_ENV \
    PORT=3000 \
    DATABASE_URL=""

# Configuration multi-environnements
ENV TZ=Europe/Paris \
    LANG=en_US.UTF-8
```



Pourquoi USER non-root ?

La sécurité avant tout

Utiliser USER pour la sécurité :

-  **Principe** : Moindre privilège = meilleure sécurité
-  **Root = Danger** : Accès total au système en cas de faille
-  **Utilisateur limité** : Accès restreint aux ressources
-  **Production** : Obligation pour la sécurité en production

Analogie : C'est comme donner un badge visiteur au lieu des clés de la maison !



Sécurité avec USER



Toujours utiliser un utilisateur non-root

```
# Alpine Linux
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

```
# Debian/Ubuntu
RUN groupadd -r appuser && useradd -r -g appuser appuser
USER appuser
```

Jamais de USER root en production !



[Revenir au sommaire](#)

Pourquoi CMD vs ENTRYPOINT ? 🚀

Les deux façons de démarrer

CMD et ENTRYPOINT définissent le démarrage :

- ⚙️ **CMD** : Commande par défaut, surchargeable facilement
- 🔒 **ENTRYPOINT** : Point d'entrée fixe, plus difficile à modifier
- 🚀 **Flexibilité** : CMD pour des containers polyvalents
- 🛡️ **Sécurité** : ENTRYPOINT pour forcer un comportement

Analogie : CMD = suggestion de menu, ENTRYPOINT = plat du jour imposé !



CMD vs ENTRYPOINT 🚀

CMD - Peut être surchargé

```
CMD ["npm", "start"]          # Défaut  
CMD ["python", "app.py"]       # Surchargeable avec docker run
```

ENTRYPOINT - Point d'entrée fixe

```
ENTRYPOINT ["../docker-entrypoint.sh"]  
CMD ["--help"]                  # Arguments par défaut  
  
# Ou combinaison  
ENTRYPOINT ["java", "-jar", "app.jar"]  
CMD ["--spring.profiles.active=prod"]
```



Qu'est-ce qu'un Multi-stage Build ? 🏭

L'art de l'optimisation

Le **multi-stage build** sépare construction et production :

- 🏗 **Stage Build** : Image lourde avec tous les outils de développement
- 🚀 **Stage Production** : Image légère avec seulement l'application
- 📏 **Taille** : Réduction drastique (de 1GB à 200MB possible)
- 🔒 **Sécurité** : Pas d'outils de build en production

Analogie : C'est comme construire dans un atelier et ne livrer que le produit fini !

Multi-stage builds 🏭

Optimisation drastique : de 1GB à 200MB

```
# Stage 1: Build (image lourde avec outils)
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build && npm prune --production

# Stage 2: Production (image minimale)
FROM node:20-alpine AS production
WORKDIR /app

# Copie sélective depuis le stage précédent
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
```



Pourquoi HEALTHCHECK ? 🧑

Le monitoring automatique

HEALTHCHECK surveille la santé du container :

- 🧑 **Surveillance** : Vérification automatique de l'état
- ⚡ **Auto-repair** : Redémarrage automatique si problème
- 📊 **Monitoring** : Intégration avec les orchestrateurs
- ⚡ **Réactivité** : Détection rapide des pannes

Analogie : C'est comme un détecteur de fumée qui appelle automatiquement les pompiers !



HEALTHCHECK - Monitoring intégré



```
# HTTP healthcheck
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:3000/health || exit 1

# Avec wget (si curl indisponible)
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
    CMD wget --no-verbose --tries=1 --spider http://localhost:8080/ping || exit 1
```

Les containers avec healthcheck redémarrent automatiquement !



[Revenir au sommaire](#)

Dockerfile optimal - Template 2025 ✓

```
FROM node:20-alpine

LABEL maintainer="dev@example.com" \
      version="1.0.0" \
      description="Production-ready Node.js app"

ENV NODE_ENV=production \
    PORT=3000 \
    LOG_LEVEL=info

WORKDIR /app

# Optimisation cache : dépendances d'abord
COPY package*.json ./
RUN npm ci --only=production && \
```



Construction et analyse



Commandes de build avancées

```
# Build optimisé avec cache  
docker build --no-cache -t mon-app:latest .  
  
# Build avec arguments  
docker build --build-arg NODE_ENV=production -t mon-app:prod .  
  
# Multi-plateforme (ARM + x86)  
docker buildx build --platform linux/amd64,linux/arm64 -t mon-app:multi .  
  
# Analyse des couches  
docker history mon-app:latest  
  
# Inspection complète  
docker inspect mon-app:latest
```



Qu'est-ce qu'un `.dockerignore` ?

Le filtre intelligent

Le `.dockerignore` exclut les fichiers inutiles :

-  **Exclusion** : Évite de copier des fichiers non nécessaires
-  **Performance** : Builds plus rapides
-  **Taille** : Images plus légères
-  **Sécurité** : Évite de copier des secrets par accident

Analogie : C'est comme une liste de ce qu'il ne faut PAS mettre dans sa valise !



.dockerignore - Performance ⚡

Exclude les fichiers inutiles

```
# .dockerignore
node_modules
npm-debug.log
.git
.gitignore
README.md
.env
.nyc_output
coverage
.vscode
*.log
```

Un .dockerignore optimal = builds plus rapides !



Erreurs courantes à éviter ✗

Anti-patterns

```
# ✗ Image sans version
FROM ubuntu:latest

# ✗ Installation inutile
RUN apt-get update && apt-get install -y vim nano

# ✗ Copie inefficace
COPY . .
RUN npm install

# ✗ Pas de nettoyage
RUN apt-get install -y curl
# (laisse les caches)

# ✗ Reste en root
```



Bonnes pratiques résumées



Checklist pour un Dockerfile professionnel

- ✓ **Image de base** : Alpine, slim, ou spécialisée avec version ✓ **Ordre des COPY** : Dépendances avant code source
- ✓ **RUN optimisé** : Une seule couche avec nettoyage ✓ **USER non-root** : Sécurité obligatoire ✓
- HEALTHCHECK** : Monitoring automatique ✓ **.dockerignore** : Exclusions optimisées ✓ **Multi-stage** : Images de production minimales



Exemples par stack technique



Python Flask

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
RUN adduser --disabled-password appuser
USER appuser
EXPOSE 5000
CMD ["python", "app.py"]
```



Exemple Java Spring Boot

Java Spring Boot

```
FROM openjdk:21-jre-slim
WORKDIR /app
COPY target/*.jar app.jar
RUN addgroup --system spring && adduser --system --group spring
USER spring
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```



Exercices Dockerfile



3 niveaux progressifs simples

Apprenez Docker étape par étape !



Exercices Express (Mise en jambes)

3 exercices rapides pour créer ses premières images

Avant les exercices principaux, des Dockerfiles simples pour s'échauffer !



[Revenir au sommaire](#)

Exercice Express 1 : Custom Nginx Page

Personnaliser une page nginx (20 min)

Ce qu'on apprend : FROM, COPY, instructions de base

```
# 1. Créer une page personnalisée
mkdir my-nginx && cd my-nginx

# 2. Créer le fichier HTML : index.html
<!DOCTYPE html>
<html>
<head><title>Mon Docker Custom</title></head>
<body style="font-family: Arial; text-align: center; padding: 50px; background: #e3f2fd;">
    <h1>🌐 Ma Première Image Custom</h1>
    <p>J'ai créé cette page avec Dockerfile !</p>
    <p>Version: <strong>1.0</strong></p>
</body>
</html>

# 2. Créer le Dockerfile
```

Mission : Voir votre page personnalisée !



[Revenir au sommaire](#)

Exercice Express 2 : App Node.js Simple

Containeriser une app web basique (25 min)

Ce qu'on apprend : WORKDIR, RUN, CMD, workflow complet

```
# 1. Créer l'app Node.js
mkdir my-app && cd my-app

# 2. Créer le fichier package.json
{
  "name": "docker-app",
  "version": "1.0.0",
  "main": "server.js",
  "dependencies": {
    "express": "^4.18.0"
  }
}

# 3. Créer le fichier server.js
const express = require('express');
```

Mission : Voir votre app web fonctionner !



[Revenir au sommaire](#)

Exercice Express 3 : Multi-stage Optimisé

Optimiser avec un build en 2 étapes (30 min)

Ce qu'on apprend : Multi-stage build, optimisation de taille

```
# 1. Préparer le projet
mkdir optimized-app && cd optimized-app

# App simple qui génère du contenu statique
const fs = require('fs');

const html = `
<!DOCTYPE html>
<html>
<head><title>App Optimisée</title></head>
<body style="font-family: Arial; text-align: center; padding: 50px; background: #f3e5f5;">
    <h1>⚡ Image Multi-Stage</h1>
    <p>Cette image a été optimisée !</p>
    <p>Générée à: ${new Date().toLocaleString()}</p>
</body>
```

Mission : Comparer les tailles d'images !



[Revenir au sommaire](#)



Exercices Principaux Détailés



[Revenir au sommaire](#)

Exercice Niveau Simple

Personnaliser une page web

Objectif : Customiser une image nginx avec votre propre page

Ce qu'on apprend :

- FROM : Choisir une image de base
- COPY : Ajouter nos fichiers
- ENV : Variables d'environnement

Consignes :

1. Partir de nginx:alpine
2. Ajouter votre page web personnalisée
3. Tester le résultat



[Revenir au sommaire](#)

Correction Niveau Simple

```
# 1. Créer le projet
mkdir mon-site
cd mon-site

# 2. Créer une page web simple
# 2. Créer le fichier HTML : index.html
<!DOCTYPE html>
<html>
<head>
    <title>Mon Site Docker</title>
    <style>
        body {
            font-family: Arial;
            text-align: center;
            padding: 50px;
```



Dockerfile Simple

```
# 3. Créer le Dockerfile
# Image de base
FROM nginx:alpine

# Informations sur l'image
LABEL maintainer="moi@formation.fr"
LABEL description="Mon premier site personnalisé"

# Variables d'environnement
ENV SITE_NAME="Mon Site Docker"
ENV VERSION="1.0"

# Copier ma page web dans nginx
COPY index.html /usr/share/nginx/html/
```

✓ **Résultat :** Votre première image Docker personnalisée !



[Revenir au sommaire](#)

Exercice Niveau Intermédiaire

Ajouter des outils utiles

Objectif : Créez une image avec quelques outils pratiques

Ce qu'on apprend :

- RUN : Installer des packages
- WORKDIR : Définir le répertoire de travail
- CMD : Commande par défaut

Outils ajoutés :

- curl : Pour tester des URLs
- nano : Éditeur de texte
- htop : Voir les processus



[Revenir au sommaire](#)

Correction Niveau Intermédiaire

```
# 1. Créer le projet
mkdir outils-docker
cd outils-docker

# 2. Script d'aide simple
#!/bin/sh
echo "✖ Outils disponibles:"
echo " curl - Tester des URLs"
echo " nano - Éditer des fichiers"
echo " htop - Voir les processus"
echo ""
echo "Exemples:"
echo " curl https://httpbin.org/json"
echo " nano test.txt"
echo " htop"
```



Dockerfile Intermédiaire

```
# 3. Dockerfile avec outils
# Image de base légère
FROM alpine:latest

# Infos
LABEL description="Image avec outils utiles"
LABEL version="2.0"

# Installer les outils
RUN apk update && apk add --no-cache \
    curl \
    nano \
    htop \
    bash
```

✓ **Résultat :** Image avec outils pratiques pour tester et débugger



[Revenir au sommaire](#)

Exercice Niveau Avancé

Multi-stage simple

Objectif : Optimiser la taille avec un build en 2 étapes

Ce qu'on apprend :

- Multi-stage build
- COPY --from=
- Optimisation des images

Concept :

- Étape 1 : Préparer les fichiers
- Étape 2 : Image finale légère



[Revenir au sommaire](#)

Correction Multi-stage

```
# 1. Créer le projet
mkdir site-optimise
cd site-optimise

# 2. Créer plusieurs pages
<!DOCTYPE html>
<html>
<head>
    <title>Site Optimisé</title>
    <style>
        body { font-family: Arial; padding: 20px; background: #f0f8ff; }
        .container { max-width: 600px; margin: 0 auto; background: white; padding: 20px; border-radius: 10px; }
    </style>
</head>
<body>
```



Dockerfile Multi-stage

```
# 3. Dockerfile optimisé
# =====
# Étape 1: Préparation
# =====
FROM alpine:latest AS builder

# Installer des outils pour préparer
RUN apk add --no-cache curl

# Copier les fichiers sources
WORKDIR /src
COPY *.html ./

# Simuler une optimisation (minification)
RUN mkdir /dist && \
```

✓ **Résultat :** Image optimisée plus petite grâce au multi-stage !



[Revenir au sommaire](#)

Récapitulatif Dockerfile



Ce qu'on a appris simplement

● Niveau Simple :

- FROM : Choisir une image de base
- COPY : Ajouter nos fichiers
- ENV : Variables d'environnement
- LABEL : Métadonnées

🟡 Niveau Intermédiaire :

- RUN : Installer des packages
- WORKDIR : Répertoire de travail
- CMD : Commande par défaut
- Scripts d'aide basiques



[Revenir au sommaire](#)

Récapitulatif Dockerfile (suite)



🔴 Niveau Avancé :

- Multi-stage build (2 étapes)
- COPY --from= : Copier depuis une étape
- Optimisation de taille
- Comparaison d'images

🎯 Progression logique maîtrisée !

Prochaine étape : Docker Compose pour orchestrer plusieurs containers !



[Revenir au sommaire](#)

💡 Points clés à retenir

Instructions Dockerfile essentielles

```
FROM image:tag          # Image de base
LABEL key="value"        # Métadonnées
ENV VAR=value           # Variables d'environnement
RUN commande            # Exécuter pendant le build
COPY source dest         # Copier fichiers
WORKDIR /path            # Répertoire de travail
CMD ["commande"]         # Commande par défaut
```

Bonnes pratiques simples

1. **Images de base légères** (alpine)
2. **Une seule responsabilité** par image
3. **Multi-stage** pour optimiser
4. **Labels** pour la documentation

🚀 **Docker maîtrisé progressivement !**



Ansible - Automatisation Infrastructure

Ansible - Automatisation Infrastructure

Infrastructure as Code moderne et simple

Ansible est l'outil d'automatisation de référence : **sans agent, idempotent, déclaratif.**



[Revenir au sommaire](#)

Ansible - Perfect pour Docker

Pourquoi Ansible + Docker ?

Parfait pour orchestrer Docker et automatiser l'infrastructure.

Exemple concret : Automatiser le déploiement d'une app Dockerisée sur plusieurs serveurs en une seule commande, sans se connecter manuellement à chaque machine, pour installer Docker, copier le code, builder l'image, lancer le conteneur, et gérer les mises à jour ou redémarrages.

Ansible 2025 : Support natif containers, modules cloud avancés, collections étendues



Pourquoi Ansible ?

Les avantages clés

-  **Simple** : Configuration YAML lisible
-  **Idempotent** : Même résultat à chaque exécution
-  **Sans agent** : SSH/WinRM/API uniquement



[Revenir au sommaire](#)

Pourquoi Ansible ? (suite)



Plus d'avantages

- ✓ **Scalable** : De 1 à 10,000+ serveurs
- 🔒 **Sécurisé** : Utilise vos connexions existantes



[Revenir au sommaire](#)

Installation et setup 2025

Installation rapide

```
# Installation via pip (recommandé)
python3 -m pip install --user ansible
# Collections essentielles
# community.general : collection avec plein de modules pour gérer fichiers, paquets, services, réseau,
# ansible.posix : modules spécifiques POSIX/Linux comme gestion utilisateurs, groupes, permissions, tâches
# ansible.windows : modules spécifiques Windows comme gestion services, tâches planifiées, partage de ...
ansible-galaxy collection install community.general ansible.posix
# Vérification
ansible --version
```

Si vous avez : Nothing to do. All requested collections are already installed. If you want to reinstall them, consider using --force .

Cela veut dire que vous avez déjà installé les collections ou que vous avez déjà installé ansible qui les comporte maintenant par défaut.



[Revenir au sommaire](#)

Installation Windows

```
# Si Windows et pas de python :  
# Installer python via chocolatey  
choco install python  
# Installer ansible via pip  
python3 -m pip install --user ansible  
# Vérification  
ansible --version
```



Solution environnement virtuel

Si vous avez un problème d'installation, voici une solution :

Sur python 3 vous devez mettre en place un venv :

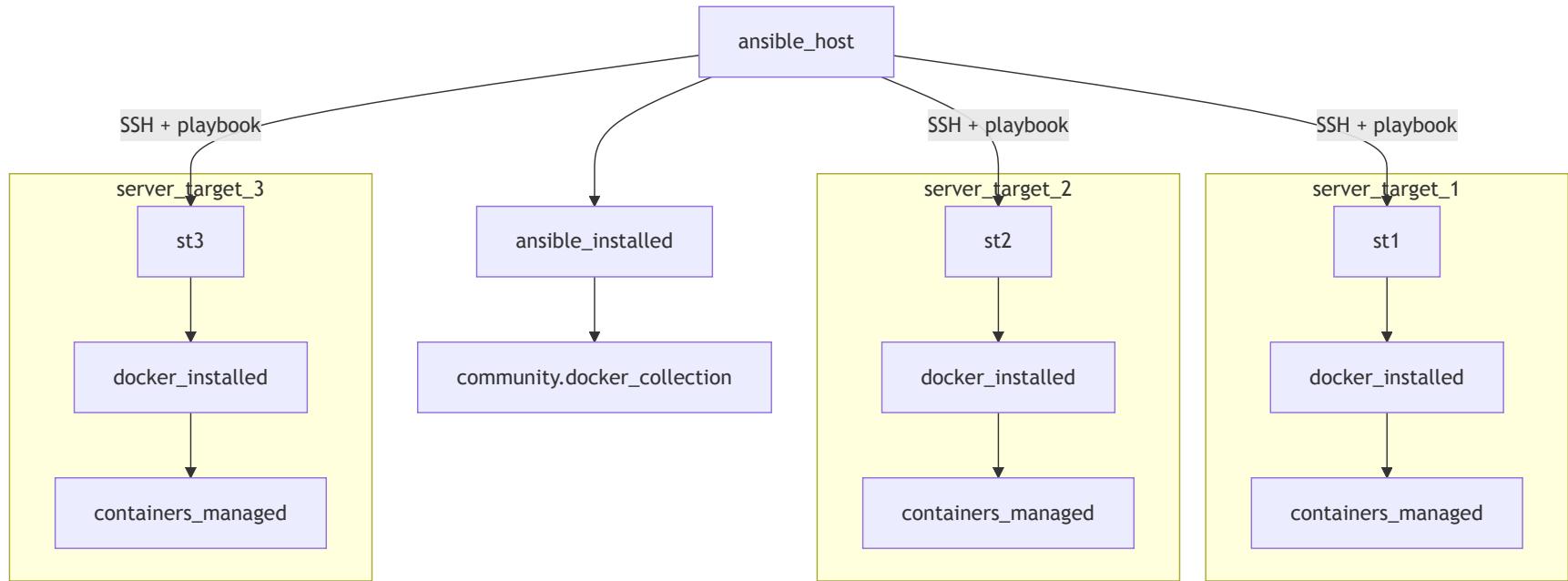
```
python3 -m venv .venv  
source .venv/bin/activate  
pip3 install ansible
```

Cela créer un environnement virtuel avec ansible installé dedans. (mode sandbox par défaut de python 3 pour ne pas polluer votre environnement global)



[Revenir au sommaire](#)

Pour bien comprendre schématiquement le fonctionnement d'ansible, voici un schéma :



Vous pouvez avoir plusieurs buts à l'utilisation d'ansible :

- Vous voulez automatiser des tâches répétitives (installation de paquets, configuration de services, etc.)
- Vous voulez déployer des applications sur plusieurs serveurs
- Vous voulez configurer des serveurs de manière uniforme
- Vous voulez gérer des secrets de manière sécurisée



[Revenir au sommaire](#)

Intégration d'Ansible dans un workflow CI/CD

Option 1 – Utilisation avec GitHub Actions

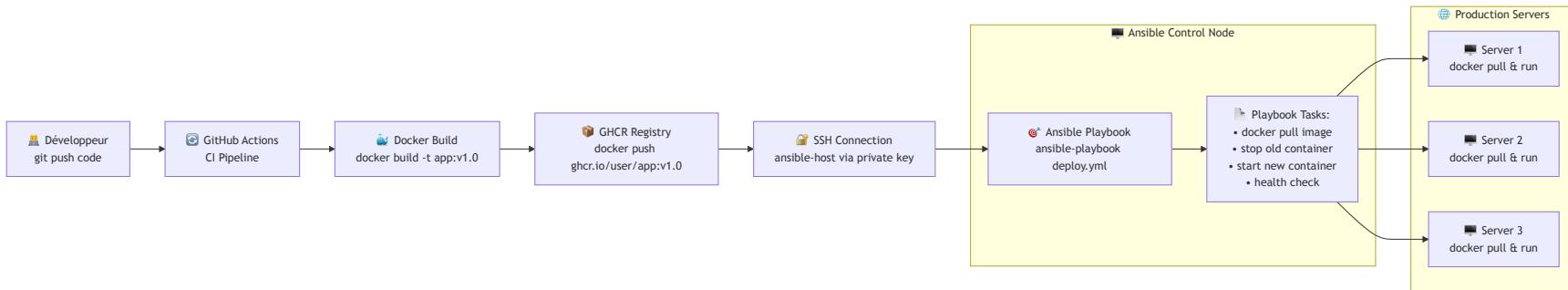
Il est tout à fait possible d'intégrer Ansible dans un pipeline GitHub Actions afin d'automatiser le déploiement à chaque git push.

Deux approches sont possibles :

- Exécution distante : GitHub Actions peut se connecter en SSH à un serveur (via une clé privée stockée dans les secrets) pour exécuter un ansible-playbook depuis ce serveur.
- Runner auto-hébergé : Vous pouvez utiliser un runner GitHub hébergé sur un serveur interne, avec Ansible installé, afin de lancer localement les déploiements via les playbooks.

 Attention : dans ce modèle, Ansible ne copie pas directement le code applicatif. Le code est packagé dans une image Docker (via GitHub Actions), poussée vers un registry (comme GHCR), puis Ansible se charge uniquement de récupérer et d'exécuter cette image sur les serveurs cibles (docker pull, docker_container...).



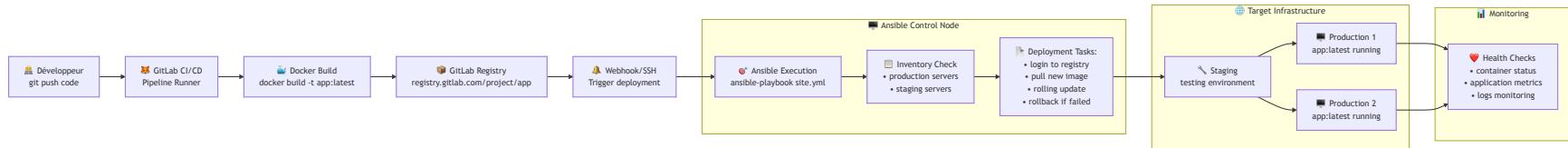


Option 2 – Intégration avec GitLab CI/CD

GitLab permet un fonctionnement similaire :

- Le pipeline CI build l'image Docker de votre application.
- Cette image est poussée automatiquement vers le GitLab Container Registry.
- Ensuite, plusieurs options sont possibles :
 - Exécution Ansible depuis le pipeline, si le runner dispose d'Ansible.
 - Connexion SSH à un hôte distant pour exécuter un playbook.
 - Déclenchement via webhook d'un outil externe tel que Jenkins, qui se chargera lui-même de lancer Ansible.





Qu'est-ce qu'un Inventaire ?

Le carnet d'adresses de vos serveurs

L'inventaire est un fichier qui liste tous vos serveurs et leurs informations :

-  **Adresses IP** : Où sont vos serveurs
-  **Groupes** : Organiser par fonction (web, base de données...)
-  **Variables** : Configuration spécifique à chaque serveur
-  **Connexion** : Comment se connecter (utilisateur, clés SSH...)

Analogie : C'est comme un carnet d'adresses pour vos serveurs !



Inventaire : Définir vos serveurs



```
# inventory/hosts.yml - Fichier d'inventaire principal
all: # Groupe racine qui contient TOUS les serveurs
  vars: # Variables globales applicables à tous les serveurs
    ansible_user: ubuntu # Utilisateur par défaut pour se connecter via SSH
    ansible_python_interpreter: /usr/bin/python3 # Chemin vers Python sur les serveurs cibles

children: # Sous-groupes organisés par fonction
  docker_hosts: # Groupe des serveurs qui hébergent Docker
    hosts: # Liste des serveurs dans ce groupe
      docker-01: {ansible_host: 10.0.1.10} # Nom logique : adresse IP réelle
      docker-02: {ansible_host: 10.0.1.11} # Deuxième serveur Docker

  databases: # Groupe des serveurs de base de données
    hosts: # Serveurs de BDD
      db-01: {ansible_host: 10.0.1.20} # Serveur de base de données principal
```



Un inventaire déjà fait pour vous en local pour s'entraîner

```
all:  
  children:  
    local:  
      hosts:  
        localhost:  
          ansible_connection: local  
          ansible_python_interpreter: /usr/bin/python3
```

```
ansible-inventory --graph
```

Vous aurez un graphique qui ressemblera à ça :

```
@all:  
|--ungrouped: # logique, vous n'avez rien qui n'est pas dans un groupe, donc ungrouped est vide  
|--@local:  
|  |--localhost
```



Qu'est-ce qu'un Playbook ? 🎭

La recette de cuisine pour vos serveurs

Un **playbook** est un fichier qui décrit les actions à effectuer :

- **Recette** : Suite d'étapes ordonnées
- **Cibles** : Sur quels serveurs agir
- **Tâches** : Quoi installer, configurer, démarrer
- **Rôles** : Qui fait quoi (utilisateur admin ou normal)

Analogie : C'est comme une recette de cuisine, mais pour configurer vos serveurs !

Premier playbook 🎭

```
- name: Configuration serveurs Docker
hosts: local
become: true
vars:
  docker_compose_version: '2.24.0'
  ansible_ssh_public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"

tasks:
  - name: Installation Docker
    apt:
      name:
        - docker.io
      state: present
      update_cache: true
```



Exécution du playbook

```
# Exécution  
ansible-playbook -i inventory/hosts.yml deploy.yml
```



[Revenir au sommaire](#)

Qu'est-ce qu'un Module ?

Les outils prêts à l'emploi

Un **module** est une fonction prête à utiliser dans Ansible :

-  **Outil spécialisé** : Une action précise (installer, copier, démarrer...)
-  **Paramètres** : Options pour personnaliser l'action
-  **Idempotent** : Peut être exécuté plusieurs fois sans problème
-  **Bibliothèque** : Des centaines de modules disponibles

Analogie : C'est comme avoir une boîte à outils avec chaque outil pour une tâche précise !



Modules essentiels

Les modules indispensables pour Docker/Infrastructure

```
# Gestion des packages
- name: Installation packages
  apt:
    name: [nginx, git, docker.io, python3-pip]
    state: present
```



Modules : Fichiers et templates

```
# Gestion des fichiers/templates
- name: Configuration nginx
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
    backup: true
  notify: restart nginx
```



Modules : Commandes et scripts

```
# Commandes et scripts
- name: Build application
  shell: |
    cd /opt/app
    docker build -t myapp:{{ app_version }} .
  changed_when: true
```



Modules : Containers Docker

```
# Gestion des containers Docker
- name: Lancer container webapp
  community.docker.docker_container:
    name: webapp
    image: 'myapp:{{ app_version }}'
    ports:
      - '80:8080'
    state: started
    restart_policy: always
```



Qu'est-ce qu'une Variable ? 🔧

Les données personnalisables

Une **variable** permet de personnaliser vos playbooks :

-  **Données** : Valeurs réutilisables (version, nom, chemin...)
-  **Flexibilité** : Même playbook pour différents environnements
-  **Réutilisabilité** : Évite la duplication de code
-  **Environnements** : Dev, test, production avec des valeurs différentes

Analogie : C'est comme des champs à remplir dans un formulaire !



Variables et templates



Configuration dynamique

```
# group_vars/all.yml
app_version: 'v1.2.0'
db_password: '{{ vault_db_password }}'
nginx_worker_processes: '{{ ansible_processor_vcpus }}'
```



Variables par environnement

```
# Variables par environnement
environments:
  dev:
    domain: 'dev.myapp.com'
    replicas: 1
  prod:
    domain: 'myapp.com'
    replicas: 3
```



Qu'est-ce qu'un Template ?



Les fichiers configurables automatiquement

Un **template** est un fichier modèle avec des variables :

- **Modèle** : Fichier avec des zones à remplir automatiquement
- **Variables** : Placeholders remplacés par des vraies valeurs
- **Génération** : Crée des fichiers personnalisés pour chaque serveur
- **Configuration** : Fichiers de config adaptés à chaque environnement

Analogie : C'est comme un document Word avec des champs à compléter automatiquement !



Template : Exemple concret



De la théorie à la pratique

Situation : Vous devez configurer une application web sur 10 serveurs différents, chacun avec son IP et sa configuration spécifique.

Sans template : 10 fichiers de config différents à maintenir manuellement 😱

Avec template : 1 seul fichier modèle + variables = 10 configs générées automatiquement ! 🎉



Template : Exemple simple



Fichier de configuration d'application

```
{# templates/app.conf.j2 - Le template (modèle) #}
# Configuration pour {{ inventory_hostname }}
server_name={{ inventory_hostname }}
server_ip={{ ansible_default_ipv4.address }}
database_host={{ db_host }}
database_port={{ db_port | default(5432) }}
debug_mode={{ debug | default('false') }}

# Génération conditionnelle
{% if environment == 'production' %}
log_level=ERROR
{% else %}
log_level=DEBUG
{% endif %}
```



Template : Résultat généré



Ce que produit le template sur le serveur web-01

```
# Configuration pour web-01
server_name=web-01
server_ip=10.0.1.31
database_host=db-01.mondomaine.com
database_port=5432
debug_mode=false

# Génération conditionnelle
log_level=ERROR
```

👉 Magie : Même template → Configs différentes selon le serveur !



Template nginx avancé

```
{# templates/nginx.conf.j2 #}
worker_processes {{ nginx_worker_processes }};
# On peut utiliser des boucles pour générer des fichiers de config dynamiquement
upstream app {
    {% for i in range(environments[env].replicas) %}
        server app-{{ i+1 }}:8080;
    {% endfor %}
}
# Donc dans ce cas :
# On peut aussi utiliser des variables pour générer des fichiers de config dynamiquement
```



Template nginx (suite)

```
server {  
    server_name {{ environments[env].domain }};  
  
    location / {  
        proxy_pass http://app;  
    }  
}
```



Template : Syntaxe Jinja2



Les éléments essentiels à retenir

```
{# Ceci est un commentaire (ne sera pas dans le fichier final) #-}

{{ variable }}                      # Affiche la valeur d'une variable
{{ variable | default('valeur') }} # Valeur par défaut si variable vide
{{ ansible_hostname }}            # Variable automatique d'Ansible

{% if condition %}                  # Structure conditionnelle
    contenu si vrai
{% else %}
    contenu si faux
{% endif %}

{% for item in liste %}           # Boucle
    traiter {{ item }}
{% endfor %}
```

Astuce : Les templates utilisent l'extension .j2 (pour Jinja2)



Qu'est-ce qu'un Handler ? 🎯

Les actions déclenchées automatiquement

Un **handler** est une tâche qui se déclenche uniquement si nécessaire :

- ⚡ **Réaction** : Se déclenche quand quelque chose change
- ⚡ **Efficacité** : Évite les redémarrages inutiles
- ⚡ **Précision** : Action ciblée (redémarrer service, recharger config...)
- ⚡ **Idempotence** : Ne s'exécute que si vraiment nécessaire

Analogie : C'est comme un système d'alarme qui ne sonne que s'il y a un problème !



Handlers et conditions 🎯

Réactivité et logique

```
tasks:  
  - name: Configuration Docker daemon  
    template:  
      src: daemon.json.j2  
      dest: /etc/docker/daemon.json  
    notify: restart docker  
    when: configure_docker_daemon | default(false)
```



Handlers : Déploiement multi-réplicas

```
- name: Déploiement app selon environnement
  community.docker.docker_container:
    name: 'webapp-{{ item }}'
    image: 'myapp:{{ app_version }}'
    ports:
      - '{{ 8080 + item }}:8080'
    env:
      ENV: '{{ env }}'
      REPLICA: '{{ item }}'
  loop: '{{ range(1, environments[env].replicas + 1) | list }}'
```



Handlers : Définition

```
handlers:  
  - name: restart docker  
    systemd:  
      name: docker  
      state: restarted  
  
  - name: reload nginx  
    systemd:  
      name: nginx  
      state: reloaded
```



Qu'est-ce qu'un Rôle ?

Les modules réutilisables

Un **rôle** est un ensemble organisé de tâches réutilisables :

-  **Organisation** : Structure claire (tâches, variables, templates...)
-  **Réutilisabilité** : Utilisable dans plusieurs playbooks
-  **Bibliothèque** : Partageable avec d'autres équipes
-  **Modularité** : Combine plusieurs rôles pour une solution complète

Analogie : C'est comme une application mobile que vous installez pour une fonction précise !



Rôles : Réutilisabilité



Structure modulaire

```
# roles/docker/tasks/main.yml
---
- name: Installation Docker
  apt:
    name: [docker.io, docker-compose-plugin]
    state: present

- name: Configuration Docker daemon
  template:
    src: daemon.json.j2
    dest: /etc/docker/daemon.json
  notify: restart docker
```



Utilisation des rôles

```
# Utilisation dans un playbook
---
- name: Setup infrastructure
  hosts: all
  become: true

  roles:
    - docker
    - nginx
    - {role: app, app_version: 'v2.0.0'}
```



Ansible + Docker : Stack complète 🐳

Déploiement d'application containerisée

```
---
- name: Déploiement stack web complète
  hosts: docker_hosts
  become: true
  vars:
    app_name: 'webapp'
    app_version: "{{ lookup('env', 'CI_COMMIT_SHA') | default('latest') }}"
```



Stack Docker : Préparation

```
tasks:
  - name: Création des répertoires
    file:
      path: '/opt/{{ app_name }}/{{ item }}'
      state: directory
    loop: [data, logs, config]

  - name: Configuration docker-compose
    template:
      src: docker-compose.yml.j2
      dest: '/opt/{{ app_name }}/docker-compose.yml'
```



Stack Docker : Déploiement

```
- name: Déploiement de l'application
  community.docker.docker_compose:
    project_src: '/opt/{{ app_name }}'
    pull: true
    state: present

- name: Vérification santé des services
  uri:
    url: 'http://{{ inventory_hostname }}/health'
    method: GET
  retries: 5
  delay: 10
```



Qu'est-ce qu'une Collection ?

Les extensions spécialisées

Une **collection** est un pack d'extensions pour Ansible :

-  **Pack** : Ensemble de modules spécialisés
-  **Domaine** : Cloud (AWS, Azure), containers (Docker), orchestration (Kubernetes)
-  **Évolution** : Mises à jour indépendantes d'Ansible
-  **Spécialisation** : Outils experts pour des technologies précises

Analogie : C'est comme des extensions dans votre navigateur pour des fonctions spéciales !



Collections et écosystème

Extensions essentielles

```
# Collections Docker  
ansible-galaxy collection install community.docker
```

```
# Collections Cloud  
ansible-galaxy collection install amazon.aws  
ansible-galaxy collection install azure.azcollection
```

```
# Collections Kubernetes  
ansible-galaxy collection install kubernetes.core
```



Utilisation avec collections

```
# Utilisation avec collections
- name: Gestion infrastructure cloud + containers
  hosts: localhost
  tasks:
    - name: Création instance AWS
      amazon.aws.ec2_instance:
        name: 'docker-host'
        image_id: ami-0abcdef1234567890
        instance_type: t3.medium
```



Collections : Attente et déploiement

```
- name: Attente démarrage
  wait_for:
    host: '{{ item.public_ip_address }}'
    port: 22

- name: Déploiement containers
  community.docker.docker_container:
    name: myapp
    image: nginx:alpine
    delegate_to: '{{ item.public_ip_address }}
```



Qu'est-ce qu'Ansible Vault ?

Le coffre-fort pour vos secrets

Ansible Vault chiffre vos données sensibles :

-  **Chiffrement** : Mots de passe, clés API, certificats
-  **Mot de passe maître** : Un seul mot de passe pour tout déchiffrer
-  **Fichiers sécurisés** : Stockage chiffré dans votre projet
-  **Partage sécurisé** : Équipe peut utiliser sans voir les secrets

Analogie : C'est comme un coffre-fort numérique pour vos mots de passe !



Ansible Vault : Secrets 🔒

Gestion sécurisée des secrets

```
# Créer un fichier chiffré  
ansible-vault create secrets.yml  
ansible-vault edit secrets.yml  
  
# Utilisation  
ansible-playbook -i inventory deploy.yml --ask-vault-pass
```



[Revenir au sommaire](#)

Vault : Utilisation des secrets

```
# secrets.yml (chiffré)
vault_db_password: 'super_secret_password'
vault_api_key: '1234567890abcdef'

# Utilisation dans les playbooks
database:
    password: '{{ vault_db_password }}'

api:
    key: '{{ vault_api_key }}'
```



Optimisation et bonnes pratiques 🚀

Configuration production

```
# ansible.cfg
[defaults]
# Pour ne pas vérifier les clés SSH
host_key_checking = False
# Pour afficher le temps d'exécution de chaque tâche
callback_whitelist = timer, profile_tasks
# Pour afficher les résultats de chaque tâche
stdout_callback = yaml
# Pour configurer le nombre de forks
forks = 20
# Pour configurer le timeout
timeout = 60
```



Structure projet

```
ansible-project/
├── ansible.cfg
├── inventory/
│   ├── production.yml
│   └── staging.yml
├── group_vars/all.yml
├── playbooks/
│   ├── site.yml
│   └── deploy.yml
├── roles/
└── secrets.yml (vault)
```



Qu'est-ce qu'un Tag ?



Les étiquettes pour l'exécution sélective

Un **tag** permet d'exécuter seulement certaines parties :

- **Étiquette** : Marquer des tâches par fonction
- **Sélectif** : Exécuter seulement l'installation, ou la config...
- **Rapidité** : Éviter de rejouer tout le playbook
- **Maintenance** : Corrections ciblées sans tout recommencer

Analogie : C'est comme des filtres dans une liste de courses !



Tags et exécution sélective

```
- name: Installation Docker
  apt:
    name: docker.io
    tags: [install, docker]

- name: Configuration
  template:
    src: config.j2
    dest: /etc/app.conf
  tags: [config]
```



Exécution avec tags

```
# Exécution sélective
ansible-playbook site.yml --tags "docker,config"
ansible-playbook site.yml --skip-tags "install"
```



[Revenir au sommaire](#)

Exercices Ansible



3 niveaux DevOps progressifs

Automatisez vos déploiements Docker avec Ansible !

[Revenir au sommaire](#)

Exercice Niveau Simple

Installation Docker avec Ansible

Objectif : Utiliser Ansible pour installer Docker localement

Consignes :

1. Créer un inventaire local
2. Playbook d'installation Docker
3. Vérifier l'installation
4. Préparer l'environnement pour les containers



[Revenir au sommaire](#)

Correction Niveau Simple - Inventaire

```
# 1. Créer le projet Ansible
mkdir ansible-docker
cd ansible-docker

# 2. Inventaire local
all:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python_interpreter: /usr/bin/python3
  vars:
    ansible_user: "{{ ansible_env.USER }}"
EOF
```



Correction Niveau Simple - Playbook

```
- name: Installation Docker
hosts: localhost
become: true
vars:
  docker_packages:
    - docker.io
    - docker-compose-plugin
    - python3-docker

tasks:
  - name: Installation Docker
    apt:
      name:
        - docker.io
      state: present
```



Correction Niveau Simple - Exécution

```
# 5. Exécuter l'installation  
ansible-playbook -i inventory.yml install-docker.yml
```

 **Résultat :** Docker installé et configuré automatiquement

Vous pouvez vérifier dans votre container avec un simple `docker ps` ou tout autre commande docker.



[Revenir au sommaire](#)

🟡 Exercice Niveau Intermédiaire

Déploiement Dockerfile avec Ansible

Objectif : Utiliser Ansible pour déployer l'image créée dans les exercices Dockerfile

Consignes :

1. Créer un rôle Ansible pour le déploiement
2. Copier et builder un Dockerfile
3. Lancer le container avec configuration
4. Gérer le cycle de vie (start/stop/update)



[Revenir au sommaire](#)

Correction Niveau Intermédiaire - Structure

```
# 1. Créer la structure de rôle
mkdir -p roles/webapp/{tasks,files,templates,vars,handlers}

# 2. Copier le Dockerfile de l'exercice précédent
mkdir -p roles/webapp/files/app

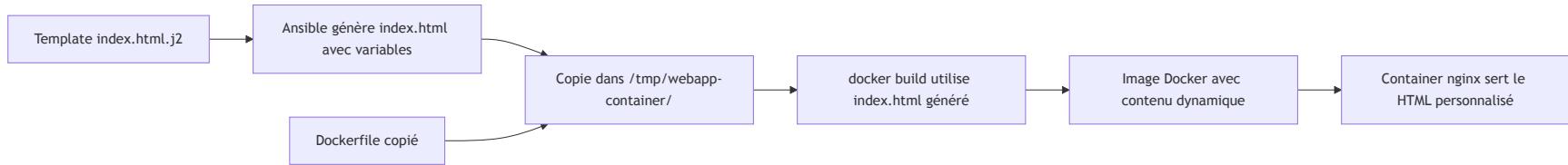
FROM nginx:alpine

# Copier notre page
COPY index.html /usr/share/nginx/html/

# Healthcheck
HEALTHCHECK --interval=30s --timeout=3s \
CMD wget --no-verbose --tries=1 --spider http://localhost/ || exit 1
```



Niveau Intermédiaire - Template



Au lieu d'avoir une page web statique, vous avez une page qui affiche automatiquement :

- "Environnement: development" ou "production"
- "Version: 2.0.0"
- "Déployé le: 2025-01-09 à 14:30:25"
- Des couleurs différentes selon l'environnement

Correction Niveau Intermédiaire - Template

3. Template de page web avec variables Ansible

```
<!DOCTYPE html>
<html>
<head>
    <title>{{ app_name | default('WebApp Ansible') }}</title>
    <style>
        body {
            font-family: Arial;
            text-align: center;
            padding: 50px;
            background: {{ bg_color | default('#f0f8ff') }};
        }
        .container {
            background: white;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Welcome to {{ app_name }}!</h1>
        <p>This is a simple web application built using Ansible. The application name is <b>{{ app_name }}</b>. The background color is <b>{{ bg_color }}</b>.</p>
        <ul>
            <li>Home</li>
            <li>About</li>
            <li>Contact</li>
        </ul>
    </div>
</body>
</html>
```



Correction Niveau Intermédiaire - Variables

4. Variables du rôle

```
app_name: "Ma WebApp Ansible"
app_version: "2.0.0"
app_port: 8080
docker_image: "webapp-ansible"
docker_container: "webapp-container"
environment: "{{ env | default('development') }}"

# Configuration des couleurs par environnement
env_colors:
  development: "#e3f2fd"
  staging: "#fff3e0"
  production: "#e8f5e8"
```



Correction Niveau Intermédiaire - Tâches

```
# 5. Tâches principales du rôle

---

- name: Créer le répertoire de travail
  file:
    path: "/tmp/{{ docker_container }}"
    state: directory
    mode: '0755'

- name: Copier le Dockerfile
  copy:
    src: "app/"
    dest: "/tmp/{{ docker_container }}/"
    mode: '0644'
```



Correction Niveau Intermédiaire - Handlers

```
# 6. Handlers pour les notifications

---
- name: Rebuild image
  debug:
    msg: "Image sera reconstruite avec les nouveaux fichiers"

- name: Container deployed
  debug:
    msg: "Container {{ docker_container }} déployé sur le port {{ app_port }}"

- name: Display access info
  debug:
    msg: "Application accessible sur http://localhost:{{ app_port }}"
```



Correction Niveau Intermédiaire - Playbook

```
# 7. Playbook principal

---

- name: Déploiement WebApp avec Dockerfile
  hosts: localhost
  vars:
    env: "{{ target_env | default('development') }}"

  tasks:
    - name: Déployer l'application web
      include_role:
        name: webapp
      notify: Display access info

  handlers:
```



Correction Niveau Intermédiaire - Tests

```
# 8. Scripts de déploiement par environnement

#!/bin/bash

case "$1" in
  dev)
    echo "🚀 Déploiement en développement..."
    ansible-playbook -i inventory.yml deploy-webapp.yml -e target_env=development
    ;;
  staging)
    echo "🚀 Déploiement en staging..."
    ansible-playbook -i inventory.yml deploy-webapp.yml -e target_env=staging -e app_port=8081
    ;;
  prod)
    echo "🚀 Déploiement en production..."
```

✓ **Résultat :** Application web déployée avec Ansible et Dockerfile



Exercice Niveau Avancé

Evolution vers une Stack Production

Objectif : Faire évoluer notre webapp simple vers une vraie stack production

Le problème : Notre webapp du niveau intermédiaire est trop simple pour la production :

- Pas de base de données
- Pas de proxy/load balancer
- Pas de monitoring
- Configuration manuelle

L'objectif : Créer une stack complète avec Ansible !



Ce qu'on va construire

Architecture cible



[Revenir au sommaire](#)

Étape 1 - Structure du rôle

D'abord, on organise notre nouveau rôle pour la stack :

```
# 1. Créer le rôle pour la stack
mkdir -p roles/docker-stack/{tasks,files,templates,vars,handlers,meta}

# 2. Métadonnées du rôle (dépendance du rôle webapp)

---
dependencies:
  - role: webapp
galaxy_info:
  author: DevOps Team
  description: Stack Docker Compose production avec Nginx + WebApp + MySQL
  min_ansible_version: 2.9
EOF
```

 **Logic** : Notre nouvelle stack utilise le rôle webapp qu'on a créé avant !



Étape 2 - Variables de base

Configuration de base de notre stack :

```
# 3. Variables principales

# Configuration de la stack
stack_name: "production-stack"
stack_directory: "/opt/{{ stack_name }}"

# Configuration application
app_port: 80
app_image: "webapp-ansible" # L'image qu'on a créée avant
app_version: "latest"
EOF
```



[Revenir au sommaire](#)

Étape 3 - Variables base de données

Configuration MySQL sécurisée :

```
# Ajouter à vars/main.yml

# Configuration base de données
mysql_root_password: "{{ vault_mysql_root_password | default('production123') }}"
mysql_database: "webapp"
mysql_user: "app_user"
mysql_password: "{{ vault_mysql_password | default('apppass123') }}"
EOF
```



[Revenir au sommaire](#)

Étape 4 - Variables environnements

Configuration par environnement (dev/staging/prod) :

```
# Ajouter à vars/main.yml

# Environnements et ressources
environments:
  production:
    replicas: 2
    memory_limit: "512m"
    cpu_limit: "0.5"
  staging:
    replicas: 1
    memory_limit: "256m"
    cpu_limit: "0.25"

# Configuration monitoring
```



[Revenir au sommaire](#)

Étape 5 - Docker Compose : Base

Création du template docker-compose principal :

```
# 4. Template docker-compose.yml - Base

version: '3.8'

services:
  proxy:
    image: nginx:alpine
    container_name: {{ stack_name }}-proxy
    ports:
      - "{{ app_port }}:80"
    volumes:
      - ./nginx-proxy/nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - app
  networks:
```



Étape 6 - Docker Compose : Application

Service application dans le docker-compose :

```
# Continuer le template docker-compose.yml

app:
  image: {{ app_image }}:{{ app_version }}
  container_name: {{ stack_name }}-app
  networks:
    - frontend
    - backend
  depends_on:
    - database
  restart: unless-stopped
  deploy:
    resources:
      limits:
```



[Revenir au sommaire](#)

Étape 7 - Docker Compose : Base de données

Service MySQL dans le docker-compose :

```
# Continuer le template docker-compose.yml

database:
  image: mysql:8.0
  container_name: {{ stack_name }}-db
  environment:
    MYSQL_ROOT_PASSWORD: {{ mysql_root_password }}
    MYSQL_DATABASE: {{ mysql_database }}
    MYSQL_USER: {{ mysql_user }}
    MYSQL_PASSWORD: {{ mysql_password }}
  volumes:
    - db_data:/var/lib/mysql
    - ./backups:/backups
  networks:
```



[Revenir au sommaire](#)

Étape 8 - Docker Compose : Réseaux

Volumes et réseaux du docker-compose :

```
# Finir le template docker-compose.yml

volumes:
  db_data:

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
EOF
```



[Revenir au sommaire](#)

Étape 9 - Configuration Nginx : Base

Template nginx pour le proxy :

```
# 5. Template nginx proxy - Configuration de base

events {
    worker_connections 1024;
}

http {
    upstream app {
        server app:80;
    }

    # Logs
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
}

EOF
```



[Revenir au sommaire](#)

Étape 10 - Configuration Nginx : Virtual Host

Configuration du serveur web :

```
# Continuer le template nginx
# chemin : roles/docker-stack/templates/nginx.conf.j2

server {
    listen 80;
    server_name {{ ansible_fqdn | default('localhost') }};

    # Health check endpoint
    location /health {
        access_log off;
        return 200 "healthy\n";
        add_header Content-Type text/plain;
    }

    # Application proxy
}
```



[Revenir au sommaire](#)

Étape 11 - Tâches : Préparation

Tâches de préparation des répertoires :

```
# 6. Tâches de déploiement - Partie 1 : Préparation
# chemin : roles/docker-stack/tasks/main.yml
---
- name: Créer le répertoire de la stack
  file:
    path: "{{ stack_directory }}"
    state: directory
    mode: '0755'
    owner: "{{ ansible_user }}"
    group: "{{ ansible_user }}"
- name: Créer les sous-répertoires
  file:
    path: "{{ stack_directory }}/{{ item }}"
    state: directory
```



Étape 12 - Tâches : Génération des fichiers

Génération des templates :

```
# Continuer tasks/main.yml - Partie 2 : Templates
# chemin : roles/docker-stack/tasks/main.yml

- name: Générer docker-compose.yml
  template:
    src: docker-compose.yml.j2
    dest: "{{ stack_directory }}/docker-compose.yml"
    mode: '0644'
  notify:
    - Restart stack

- name: Générer configuration nginx
  template:
    src: nginx.conf.j2
    dest: "{{ stack_directory }}/nginx-proxy/nginx.conf"
```



Étape 13 – Tâches : Déploiement

Démarrage de la stack et vérifications :

```
# Continuer tasks/main.yml - Partie 3 : Déploiement
# chemin : roles/docker-stack/tasks/main.yml

- name: Démarrer la stack Docker Compose
  docker_compose:
    project_src: "{{ stack_directory }}"
    state: present
    restarted: "{{ force_restart | default(false) }}"
  register: stack_result

- name: Configurer la crontab pour les backups
  cron:
    name: "Backup {{ stack_name }}"
    minute: "0"
    hour: "2"
```



[Revenir au sommaire](#)

Étape 14 - Script de backup

Script automatisé de sauvegarde :

```
# 7. Template script de backup
# chemin : roles/docker-stack/templates/backup.sh.j2
#!/bin/bash

BACKUP_DIR="{{ stack_directory }}/backups"
DATE=$(date +%Y%m%d_%H%M%S)
STACK_NAME="{{ stack_name }}"

echo "📁 Backup de la stack $STACK_NAME - $DATE"

# Backup base de données
docker compose -f {{ stack_directory }}/docker-compose.yml exec -T database \
    mysqldump -u root -p{{ mysql_root_password }} {{ mysql_database }} \
    > "$BACKUP_DIR/db_backup_$DATE.sql"
EOF
```



Étape 15 – Script de backup (suite)

Sauvegarde config et nettoyage :

```
# Continuer le script de backup
# chemin : roles/docker-stack/templates/backup.sh.j2

# Backup configuration
tar -czf "$BACKUP_DIR/config_backup_${DATE}.tar.gz" \
    -C {{ stack_directory }} \
    docker-compose.yml nginx-proxy/ scripts/

# Nettoyage des anciens backups (garder 7 jours)
find "$BACKUP_DIR" -name "*backup_*.sql" -mtime +7 -delete
find "$BACKUP_DIR" -name "*backup_*.tar.gz" -mtime +7 -delete

echo "✅ Backup terminé dans $BACKUP_DIR"
ls -la "$BACKUP_DIR"/${DATE}
EOF
```



Étape 16 - Script de monitoring

Script de surveillance de la stack :

```
# 8. Template script de monitoring
# chemin : roles/docker-stack/templates/monitor.sh.j2
#!/bin/bash

echo "📊 Monitoring de la stack {{ stack_name }}"
echo "====="

# Status des containers
docker compose -f {{ stack_directory }}/docker-compose.yml ps

echo ""
echo "🌐 Health checks:"
docker compose -f {{ stack_directory }}/docker-compose.yml ps --format "table {{.Name}}\t{{.Status}}"

echo ""
```



Étape 17 - Handlers

Gestionnaires de redémarrage :

```
# 9. Handlers pour les redémarrages
# chemin : roles/docker-stack/handlers/main.yml
---
- name: Restart stack
  docker_compose:
    project_src: "{{ stack_directory }}"
    restarted: true

- name: Restart proxy
  docker_compose:
    project_src: "{{ stack_directory }}"
    services:
      - proxy
    restarted: true
EOF
```



[Revenir au sommaire](#)

Étape 18 - Playbook principal

Orchestration complète :

```
# 10. Playbook principal
# chemin : deploy-stack.yml
---
- name: Déploiement Stack Docker Compose Production
  hosts: localhost
  vars:
    target_env: "{{ env | default('production') }}"
    force_restart: "{{ restart | default(false) }}"
  pre_tasks:
    - name: Vérifier les prérequis
      assert:
        that:
          - target_env in ['production', 'staging']
      fail_msg: "Environnement doit être 'production' ou 'staging'"
```



Étape 19 – Playbook (suite)

Informations de déploiement :

```
# Continuer le playbook principal
# chemin : deploy-stack.yml

post_tasks:
  - name: Afficher les informations de déploiement
    debug:
      msg: |
        ✓ Stack {{ stack_name }} déployée en {{ target_env }} !
        🌐 Application: http://localhost:{{ app_port }}
        🏫 Health: http://localhost:{{ app_port }}/health
        🔧 Monitoring: {{ stack_directory }}/scripts/monitor.sh
        📁 Backup: {{ stack_directory }}/scripts/backup.sh

EOF
```



Étape 20 - Script de déploiement

Script d'orchestration finale :

```
# 11. Script de déploiement avancé
# chemin : deploy-production.sh
#!/bin/bash

echo "🚀 Déploiement de la stack de production..."

case "$1" in
deploy)
    ansible-playbook -i inventory.yml deploy-stack.yml -e env=production
;;
update)
    ansible-playbook -i inventory.yml deploy-stack.yml -e env=production -e restart=true
;;
staging)
    ansible-playbook -i inventory.yml deploy-stack.yml -e env=staging -e app_port=8080
```



Étape 21 - Test final

Déploiement et vérification :

```
# Déploiement final
echo "⌚ Lancement du déploiement production..."
./deploy-production.sh deploy

echo "✅ Stack complète déployée avec Ansible !"

# Tests post-déploiement
echo "⚡ Tests de la stack..."
curl http://localhost/health
docker compose -f /opt/production-stack/docker-compose.yml ps
```

✅ **Résultat** : Stack production complète avec Nginx + WebApp + MySQL + Monitoring/Backup !



[Revenir au sommaire](#)

Récapitulatif Exercices Ansible



Compétences acquises

● Niveau Simple :

- Inventaires Ansible
- Playbooks d'installation
- Modules de base
- Vérification automatique

🟡 Niveau Intermédiaire :

- Rôles Ansible
- Templates Jinja2
- Variables et handlers
- Intégration Docker



[Revenir au sommaire](#)

Récapitulatif Ansible (suite)

Niveau Avancé :

- Déploiement Docker Compose
- Gestion de configuration
- Scripts de maintenance
- Monitoring automatisé

Formation Docker & Ansible complète !

Vous maîtrisez maintenant l'automatisation complète avec Docker et Ansible !



[Revenir au sommaire](#)

QCM : Maîtrise d'Ansible

[Revenir au sommaire](#)

QCM sur la maîtrise d'Ansible

1. Quel est le principe fondamental d'Ansible ?

- Ansible nécessite des agents sur tous les serveurs cibles
- Ansible fonctionne en mode "push" sans agent
- Ansible utilise uniquement le protocole HTTP
- Ansible remplace complètement SSH

2. Quels sont les composants principaux d'Ansible ?

- Control Node, Managed Nodes, Playbooks
- Master, Workers, Registry
- Client, Server, Database
- Controller, Executors, Storage



QCM Ansible (suite)

3. Que signifie "idempotent" dans le contexte Ansible ?

- Les tâches s'exécutent toujours plus rapidement à la deuxième fois
- Exécuter un playbook plusieurs fois produit le même résultat
- Les erreurs sont automatiquement corrigées
- Les tâches sont exécutées en parallèle

4. Dans quel format sont écrits les playbooks Ansible ?

- JSON
- XML
- YAML
- TOML

5. Qu'est-ce qu'un inventaire Ansible ?

- La liste des playbooks disponibles
- La liste des serveurs et groupes gérés par Ansible
- L'historique des exécutions



QCM Ansible (suite 2)

6. À quoi sert Ansible Vault ?

- Stocker les playbooks de manière sécurisée
- Chiffrer les données sensibles comme les mots de passe
- Sauvegarder l'inventaire
- Gérer les versions des playbooks

7. Quelle est la différence entre un module et un rôle ?

- Un module est réutilisable, un rôle ne l'est pas
- Un module exécute une tâche spécifique, un rôle est un ensemble de tâches organisées
- Un rôle est plus rapide qu'un module
- Il n'y a pas de différence

8. Quelle commande exécute une tâche ad-hoc sur tous les serveurs web ?

- ansible webservers -m ping
- ansible-playbook -i webservers ping.yml
- ansible all -m webservers -a ping



QCM Ansible (suite 3)

9. Quelle est la structure standard d'un rôle Ansible ?

- tasks/, handlers/, vars/, files/
- src/, build/, test/, deploy/
- main/, config/, scripts/, docs/
- playbooks/, inventories/, modules/, plugins/

10. Comment ignorer les erreurs pour une tâche spécifique ?

- ignore_errors: true
- failed_when: false
- error_handling: ignore
- skip_errors: yes

11. À quoi servent les tags dans Ansible ?

- Identifier les versions des playbooks
- Exécuter seulement certaines tâches d'un playbook
- Catégoriser les serveurs dans l'inventaire



[Revenir au sommaire](#)

QCM Ansible (suite 4)

12. Comment implémenter un déploiement blue-green avec Ansible ?

- Utiliser des groupes d'inventaire distincts et des variables conditionnelles
- Créer deux playbooks séparés
- Utiliser uniquement des rôles
- Impossible avec Ansible seul

13. Quelle n'est PAS une bonne pratique de sécurité avec Ansible ?

- Utiliser Ansible Vault pour les secrets
- Stocker les clés SSH dans les playbooks
- Limiter les priviléges avec `become_user`
- Utiliser des connexions SSH avec clés



Réponses (1-5)

1. Ansible fonctionne en mode "push" sans agent
2. Control Node, Managed Nodes, Playbooks
3. Exécuter un playbook plusieurs fois produit le même résultat
4. YAML
5. La liste des serveurs et groupes gérés par Ansible



Réponses (6-10)

6. Chiffrer les données sensibles comme les mots de passe
7. Un module exécute une tâche spécifique, un rôle est un ensemble de tâches organisées
8. ansible webservers -m ping
9. tasks/, handlers/, vars/, files/
10. ignore_errors: true



Réponses (11-13)

11. Exécuter seulement certaines tâches d'un playbook
12. Utiliser des groupes d'inventaire distincts et des variables conditionnelles
13. Stocker les clés SSH dans les playbooks



[Revenir au sommaire](#)



Questions Pratiques

Question 7 : Modules vs Rôles

Quelle est la différence entre un module et un rôle ?

- A) Un module est réutilisable, un rôle ne l'est pas
- B) Un module exécute une tâche spécifique, un rôle est un ensemble de tâches organisées
- C) Un rôle est plus rapide qu'un module
- D) Il n'y a pas de différence



Question 8 : Commandes ad-hoc ⚡

Quelle commande exécute une tâche ad-hoc sur tous les serveurs web ?

- A) ansible webservers -m ping
- B) ansible-playbook -i webservers ping.yml
- C) ansible all -m webservers -a ping
- D) ansible run webservers ping



Question 9 : Structure de rôle

Quelle est la structure standard d'un rôle Ansible ?

- A) tasks/, handlers/, vars/, files/
- B) src/, build/, test/, deploy/
- C) main/, config/, scripts/, docs/
- D) playbooks/, inventories/, modules/, plugins/



Question 10 : Gestion des erreurs 🚨

Comment ignorer les erreurs pour une tâche spécifique ?

- A) ignore_errors: true
- B) failed_when: false
- C) error_handling: ignore
- D) skip_errors: yes



Question 11 : Tags

À quoi servent les tags dans Ansible ?

- A) Identifier les versions des playbooks
- B) Exécuter seulement certaines tâches d'un playbook
- C) Catégoriser les serveurs dans l'inventaire
- D) Marquer les erreurs dans les logs





Scénarios Avancés

Question 12 : Déploiement Blue-Green

Comment implémenter un déploiement blue-green avec Ansible ?

- A) Utiliser des groupes d'inventaire distincts et des variables conditionnelles
- B) Créer deux playbooks séparés
- C) Utiliser uniquement des rôles
- D) Impossible avec Ansible seul



Question 13 : Sécurité

Quelle n'est PAS une bonne pratique de sécurité avec Ansible ?

- A) Utiliser Ansible Vault pour les secrets
- B) Stocker les clés SSH dans les playbooks
- C) Limiter les priviléges avec `become_user`
- D) Utiliser des connexions SSH avec clés



Correction et Barème

Réponses Correctes (Fondamentaux)

Questions 1-6

1. **B** - Mode push sans agent
2. **A** - Control Node, Managed Nodes, Playbooks
3. **B** - Même résultat à chaque exécution
4. **C** - Format YAML
5. **B** - Liste des serveurs gérés
6. **B** - Chiffrement des données sensibles



Correction (suite)

Réponses Correctes (Concepts)

Questions 7-11

7. **B** - Module = tâche spécifique, Rôle = ensemble organisé
8. **A** - ansible webservers -m ping
9. **A** - Structure standard des rôles
10. **A** - ignore_errors: true
11. **B** - Exécution sélective de tâches

Questions 12-13 (Avancé)

12. **A** - Groupes d'inventaire + variables
13. **B** - Ne jamais stocker les clés dans les playbooks



Score d'évaluation

Barème de notation

- **12-13 bonnes réponses :** 🏆 Expert Ansible !
- **10-11 bonnes réponses :** 🥈 Niveau avancé
- **8-9 bonnes réponses :** 🥉 Bon niveau
- **6-7 bonnes réponses :** 🥇 Niveau intermédiaire
- **< 6 bonnes réponses :** 📚 Reprenez les bases



[Revenir au sommaire](#)

💡 Explications Détaillées

Points clés à retenir

Question 3 - Idempotence : L'idempotence est cruciale : si l'état désiré est déjà atteint, Ansible ne fait rien. Cela permet d'exécuter le même playbook plusieurs fois sans effet de bord.

Question 7 - Modules vs Rôles :



[Revenir au sommaire](#)



Exemples de code

```
# Module = action unique
- name: Installer nginx
  apt:
    name: nginx
    state: present
```



💡 Exemples de code (suite)

```
# Rôle = collection organisée
- hosts: webservers
  roles:
    - webserver # Contient installation + config + service
```



Sécurité avec Ansible

```
# ❌ Mauvais - clé dans le playbook  
ssh_key: -----BEGIN RSA PRIVATE KEY-----
```



[Revenir au sommaire](#)

💡 Sécurité (suite)

```
# ✅ Bon - référence sécurisée  
ssh_key_path: "{{ vault_ssh_key_path }}"
```





Bonnes pratiques

À retenir absolument

- Ansible est agentless et idempotent
- YAML est le format standard
- Utilisez Vault pour les secrets
- Les rôles permettent la réutilisabilité
- Les tags facilitent l'exécution sélective





Prochaines étapes