



DevOps & Docker - Jour 2 Matin

Master 1 Développement Fullstack



Rappel Jour 1

Volumes, Réseaux et le problème à résoudre

Ce que vous avez appris hier

Fondamentaux Docker

- Image vs Conteneur
- Port mapping
- Isolation et éphémérité
- Construire une image avec Dockerfile

Persistante

- Named volumes
- Bind mounts
- Cycle de vie des volumes

Communication

- Réseaux Docker
- DNS interne
- Communication inter-conteneurs
- Réseau custom

TP de fin de journée

- PostgreSQL avec volume persistant
- Adminer pour l'administration
- Réseau custom
- Test de persistante

Le TP d'hier : PostgreSQL + Adminer

Vous avez créé cette stack avec plusieurs commandes :

```
# 1. Créer le réseau
docker network create demo-network

# 2. Créer le volume
docker volume create postgres-data

# 3. Lancer PostgreSQL
docker run -d --name postgres \
--network demo-network \
-v postgres-data:/var/lib/postgresql \
-e POSTGRES_PASSWORD=secret \
-e PGDATA=/var/lib/postgresql/18/docker \
postgres:18-alpine

# 4. Lancer Adminer
docker run -d --name adminer \
--network demo-network \
```

Le problème

4 commandes différentes à mémoriser et exécuter dans le bon ordre

Difficile à partager avec l'équipe (documentation, scripts, erreurs...)

Pas reproductible facilement sur une autre machine

Gestion manuelle des dépendances (ordre de démarrage)

Et si on devait créer une stack avec...

- Frontend (React)
- Backend (Node.js)
- Base de données (PostgreSQL)
- Cache (Redis)
- Reverse proxy (Nginx)

Combien de commandes Docker run ?

5 services × 4-5 options = 20+ lignes de commandes !

La solution : Docker Compose

Un seul fichier YAML déclaratif : `docker-compose.yml`

Une seule commande : `docker compose up -d`

Infrastructure as Code : versionné, partageable, reproductible



Chapitre 1

Introduction à Docker Compose

Qu'est-ce que Docker Compose ?

Docker Compose est un outil pour définir et gérer des applications multi-conteneurs avec Docker.

- Fichier **YAML déclaratif** (`docker-compose.yml`)
- Définit services, volumes, réseaux dans un seul fichier
- Gère le cycle de vie de tous les conteneurs en une seule commande
- **Infrastructure as Code** (IaC)

Docker CLI vs Docker Compose

Avec Docker CLI

```
# Créer réseau
docker network create app-net

# Créer volume
docker volume create db-data

# Lancer DB
docker run -d --name db --network app-net -v db-data:/var/lib/postgresql \
-e POSTGRES_PASSWORD=secret -e PGDATA=/var/lib/postgresql/18/docker \
postgres:18-alpine

# Lancer app
docker run -d --name app --network app-net -p 3000:3000 \
-e DATABASE_URL=... myapp:latest
```

Avec Docker Compose

```
# docker-compose.yml
services:
  db:
    image: postgres:18-alpine
    volumes:
      - db-data:/var/lib/postgresql
    environment:
      POSTGRES_PASSWORD: secret
      PGDATA: /var/lib/postgresql/18/docker

  app:
    image: myapp:latest
    ports:
      - "3000:3000"
    environment:
      DATABASE_URL: postgres://db:5432

volumes:
  db-data:
```

Avantages de Docker Compose

Simplicité

- Un seul fichier
- Syntaxe déclarative claire
- Facile à lire et comprendre

Reproductibilité

- Même config partout
- Versionné dans Git
- Partageable avec l'équipe

Automatisation

- Réseau créé automatiquement
- Gestion des dépendances
- Ordre de démarrage

Multi-environnement

- Fichiers `.env`
- Override files
- Config par environnement

Quand utiliser Docker Compose ?

Utilisez Docker Compose pour :

- Développement local d'applications multi-services
- Tests d'intégration
- Prototypes et démos
- Environnements staging/QA
- Petites applications en production (1 serveur)

✖ N'utilisez PAS Docker Compose pour :

- Production à grande échelle → **Kubernetes**
- Scaling horizontal automatique → **Kubernetes**
- Haute disponibilité multi-machines → **Kubernetes**
- Orchestration complexe avec failover → **Kubernetes / Docker Swarm**

En entreprise

90% des cas d'usage sont couverts par Compose

Vous utiliserez Docker Compose quotidiennement en dev

Kubernetes ? Géré par les DevOps en production



Chapitre 2

Syntaxe Docker Compose

Structure d'un fichier docker-compose.yml

```
version: '3.9' # Version du format (optionnel depuis Compose v2)

services:          # Définition des conteneurs
  service1:
    # Configuration du service 1
  service2:
    # Configuration du service 2

volumes:          # Définition des volumes (optionnel)
  volume1:

networks:         # Définition des réseaux (optionnel)
  network1:
```

Exemple minimal : Un service web

```
version: '3.9'

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

Lancer le service :

```
docker compose up -d
```

Arrêter le service :

```
docker compose down
```

Les sections principales

services

- Définit les conteneurs
- Configuration par service
- Image ou build

volumes

- Named volumes
- Persistance des données
- Partagés entre services

networks

- Réseaux personnalisés
- Isolation
- Communication inter-services

secrets (avancé)

- Gestion sécurisée des secrets
- Production uniquement

Indentation et syntaxe YAML

⚠ YAML est sensible à l'indentation !

- Utilisez **2 espaces** (pas de tabulations)
- Respect strict de la hiérarchie
- Les tirets  pour les listes

```
services:  
  web:                      # Service web  
    image: nginx:alpine     # 2 espaces d'indentation  
    ports:  
      - "8080:80"          # 4 espaces + tiret pour liste  
    environment:  
      - NODE_ENV=prod       # 2 espaces  
                                # 4 espaces + tiret
```



Chapitre 3

Configuration des services

Service : Utiliser une image existante

```
services:  
  postgres:  
    image: postgres:18-alpine  
    container_name: my-postgres  
    ports:  
      - "5432:5432"  
    environment:  
      POSTGRES_PASSWORD: secret  
      POSTGRES_USER: admin  
      POSTGRES_DB: mydb  
      PGDATA: /var/lib/postgresql/18/docker  
    restart: unless-stopped
```

Port mapping

```
services:  
  web:  
    ports:  
      - "8080:80"      # host:container  
      - "443:443"
```

Format : "PORT_HOST:PORT_CONTAINER"

- Le port host est exposé publiquement
- Le port container est interne au conteneur

```
services:  
  web:  
    ports:  
      - "127.0.0.1:8080:80"  # Localhost
```

Bind sur localhost uniquement

- Plus sécurisé
- Non accessible depuis l'extérieur
- Utile en production avec reverse proxy

Restart policies

```
services:  
  backend:  
    image: node:20-alpine  
    restart: unless-stopped
```

Policy	Comportement
no	Ne redémarre jamais (défaut)
always	Redémarre toujours
on-failure	Redémarre uniquement en cas d'erreur
unless-stopped	Redémarre sauf si explicitement arrêté

Service : Construire une image

```
services:  
  backend:  
    build:  
      context: ./backend  
      dockerfile: Dockerfile  
    ports:  
      - "3000:3000"  
    environment:  
      NODE_ENV: development
```

- `context` : dossier contenant le Dockerfile
- `dockerfile` : nom du Dockerfile (par défaut : `Dockerfile`)

Commande de build :

```
docker compose build          # Construit toutes les images  
docker compose build backend # Construit une image spécifique
```

Build : Options avancées

```
services:  
  backend:  
    build:  
      context: ./backend  
      dockerfile: Dockerfile.dev  
      args:  
        - NODE_VERSION=20  
        - BUILD_ENV=development  
      target: development  
    image: myapp:dev
```

- `args` : arguments de build (`ARG` dans le Dockerfile)
- `target` : stage spécifique dans multi-stage build
- `image` : nom de l'image construite

Environment variables

Méthode 1 : Liste

```
services:  
  backend:  
    environment:  
      - NODE_ENV=production  
      - PORT=3000  
      - DEBUG=false
```

Méthode 2 : Map

```
services:  
  backend:  
    environment:  
      NODE_ENV: production
```

Méthode 3 : Fichier

```
services:  
  backend:  
    env_file:  
      - .env  
      - .env.local
```

Fichier `.env` :

```
NODE_ENV=production  
PORT=3000  
DATABASE_URL=postgres://...
```

Container name vs Service name

```
services:  
  database:  
    image: postgres:18-alpine  
    container_name: my-db          # ← Container name (optionnel)
```

Service name (`database`)

- Utilisé pour la communication inter-services
- DNS interne : `database:5432`
- Obligatoire

Container name (`my-db`)

- Nom du conteneur Docker
- Visible dans `docker ps`
- Optionnel



Chapitre 4

Volumes et persistance

Rappel : Pourquoi les volumes ?

Sans volume : Les données sont éphémères

- Suppression du conteneur = perte des données
- Restart = données perdues

Avec volume : Les données persistent

- Indépendant du cycle de vie du conteneur
- Survit aux redémarrages
- Partageable entre conteneurs

Named volumes

```
services:  
  postgres:  
    image: postgres:18-alpine  
    environment:  
      PGDATA: /var/lib/postgresql/18/docker  
    volumes:  
      - postgres-data:/var/lib/postgresql  
  
volumes:  
  postgres-data:
```

Caractéristiques :

- Géré par Docker
- Portable
- Sauvegardé automatiquement

- Performance optimale

Usage : Production, données persistantes

Bind mounts

```
services:  
  backend:  
    image: node:20-alpine  
    volumes:  
      - ./backend/src:/app/src  
      - ./backend/package.json:/app/package.json
```

Caractéristiques :

- Lié au filesystem hôte
- Chemin absolu ou relatif
- Modifications en temps réel

Usage : Développement, hot-reload

Exemples de volumes

```
services:  
  postgres:  
    environment:  
      PGDATA: /var/lib/postgresql/18/docker  
    volumes:  
      - postgres-data:/var/lib/postgresql  
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql:ro  
  
volumes:  
  postgres-data:
```

:ro = read-only (lecture seule)

Volume : Options avancées

```
services:  
  backend:  
    volumes:  
      # Named volume  
      - app-data:/app/data  
      # Bind mount  
      - ./src:/app/src  
      # Bind mount read-only  
      - ./config.yml:/app/config.yml:ro  
      # Volume anonyme (cache)  
      - /app/node_modules  
  
volumes:  
  app-data:  
    driver: local  
    driver_opts:  
      type: none  
      device: /mnt/data  
      o: bind
```

Volume pour hot-reload en dev

```
services:  
  frontend:  
    build: ./frontend  
    volumes:  
      - ./frontend/src:/app/src          # Code source  
      - ./frontend/public:/app/public    # Assets  
      - /app/node_modules                # Exclude node_modules  
  
  backend:  
    build: ./backend  
    volumes:  
      - ./backend/src:/app/src  
      - /app/node_modules  
    command: npm run dev
```

Pattern classique : Bind mount le code source + volume anonyme pour node_modules

Named volumes : Cycle de vie

```
# Créer et démarrer  
docker compose up -d  
  
# Arrêter (volumes conservés)  
docker compose down  
  
# Arrêter + supprimer volumes  
docker compose down -v
```

Attention : `-v` supprime TOUTES les données !

Gestion manuelle

```
# Lister les volumes  
docker volume ls  
  
# Inspecter un volume  
docker volume inspect postgres-data  
  
# Supprimer un volume  
docker volume rm postgres-data  
  
# Nettoyer tous les volumes inutilisés  
docker volume prune
```

Volume partagé entre services

```
services:  
  app:  
    image: myapp:latest  
    volumes:  
      - shared-data:/data  
  
  backup:  
    image: backup-tool:latest  
    volumes:  
      - shared-data:/data:ro      # Read-only  
    command: backup /data  
  
volumes:  
  shared-data:
```

Use case : Application écrit des fichiers, service de backup les lit



Chapitre 5

Réseaux et communication

Réseau par défaut

```
services:  
  backend:  
    image: node:20-alpine  
  
  postgres:  
    image: postgres:18-alpine
```

Par défaut, Compose crée automatiquement un réseau bridge.

- Tous les services sont sur le même réseau
- Communication via le **service name**
- DNS interne automatique

Le backend peut joindre PostgreSQL via `postgres:5432`

DNS interne automatique

```
services:  
  backend:  
    environment:  
      DATABASE_URL: postgres://user:pass@postgres:5432/mydb  
      REDIS_URL: redis://redis:6379  
  
  postgres:  
    image: postgres:18-alpine  
  
  redis:  
    image: redis:alpine
```

- `postgres` résout vers l'IP du conteneur PostgreSQL
- `redis` résout vers l'IP du conteneur Redis
- Pas besoin de connaître les IPs !

Réseau custom

```
services:  
  backend:  
    image: node:20-alpine  
  networks:  
    - app-network  
  
  postgres:  
    image: postgres:18-alpine  
  networks:  
    - app-network  
  
networks:  
  app-network:  
    driver: bridge
```

Pourquoi un réseau custom ?

- Isolation des services
- Contrôle fin de la communication
- Nommage explicite

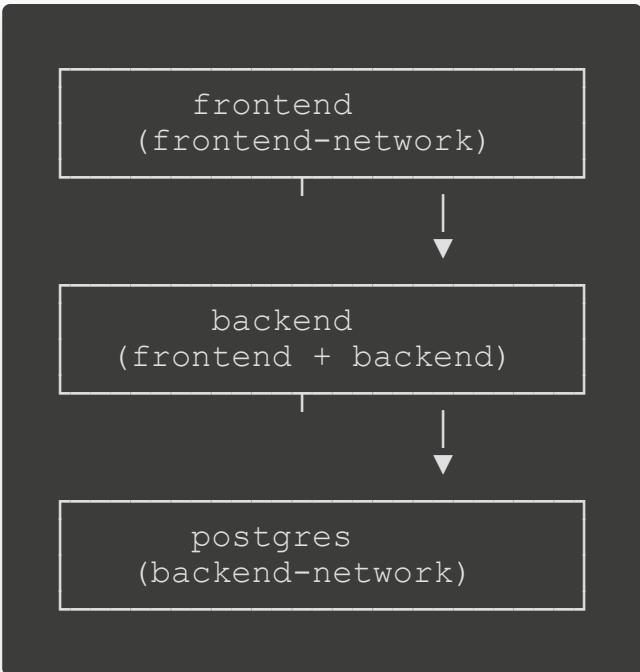
Réseaux multiples

```
services:  
  frontend:  
    networks:  
      - frontend-network  
  
  backend:  
    networks:  
      - frontend-network  
      - backend-network  
  
  postgres:  
    networks:  
      - backend-network  
  
networks:  
  frontend-network:  
  backend-network:
```

Isolation :

- Frontend ne peut PAS parler directement à PostgreSQL
- Backend peut parler aux deux

Schéma d'architecture



Port mapping : Expose vs Ports

```
services:  
  backend:  
    expose:  
      - "3000"  
  # Accessible seulement depuis le réseau interne
```

expose

- Communication inter-conteneurs uniquement
- Plus sécurisé
- Usage : services internes

```
services:  
  backend:  
    ports:  
      - "3000:3000"  
  # Accessible depuis l'hôte ET le réseau interne
```

ports

- Accessible publiquement
- Port mapping host:container
- Usage : services exposés (frontend, API)

Réseau host (avancé)

```
services:  
  monitoring:  
    image: prometheus:latest  
    network_mode: host
```

Mode host :

- Le conteneur utilise le réseau de l'hôte directement
- Pas d'isolation réseau
- Performance maximale

Attention : À utiliser avec précaution, perte d'isolation !

Extra hosts : Accéder à l'hôte (Mac/Windows)

Sur Mac/Windows, `network_mode: host` ne fonctionne pas bien.

Alternative : Utiliser `extra_hosts` et `host.docker.internal`

```
services:  
  frontend:  
    build: ./frontend  
    extra_hosts:  
      - "host.docker.internal:host-gateway"  
  environment:  
    # Accéder à un backend qui tourne sur votre Mac/PC  
    VITE_API_URL: http://host.docker.internal:3001
```

- Le frontend est conteneurisé
- Le backend tourne localement sur votre machine (hors Docker)

Extra hosts : DNS personnalisé

```
services:  
  app:  
    extra_hosts:  
      - "api.example.com:192.168.1.100"  
      - "database:172.17.0.5"  
      - "host.docker.internal:host-gateway"
```

Ce que ça fait :

- Ajoute des entrées dans `/etc/hosts` du conteneur
- Permet de résoudre des noms d'hôtes personnalisés
- Utile pour pointer vers des services externes

Vérifier dans le conteneur :

```
docker compose exec app cat /etc/hosts
# 192.168.1.100    api.example.com
# 172.17.0.5        database
```

Réseau externe

```
services:  
  app:  
    networks:  
      - existing-network  
  
networks:  
  existing-network:  
    external: true
```

Réseau externe :

- Réseau créé en dehors de Compose
- Partagé entre plusieurs stacks
- Utile pour connecter plusieurs `docker-compose.yml`



Chapitre 6

Variables d'environnement

Méthode 1 : Directement dans le fichier

```
services:  
  backend:  
    image: node:20-alpine  
    environment:  
      NODE_ENV: production  
      PORT: 3000  
      DATABASE_URL: postgres://db:5432
```

Avantages :

- Simple et lisible
- Adapté pour les valeurs non sensibles

Inconvénients :

- Valeurs en clair dans Git
- Pas de secrets !

Méthode 2 : Fichier .env

```
services:  
  backend:  
    environment:  
      NODE_ENV: ${NODE_ENV}  
      PORT: ${PORT}  
      DATABASE_URL: ${DATABASE_URL}
```

Fichier `.env` :

```
NODE_ENV=production  
PORT=3000  
DATABASE_URL=postgres://db:5432
```

Avantages :

- Séparation config/code
- Fichier .env dans .gitignore

Fichier .env : Bonnes pratiques

Fichier .env (ne jamais commit !)

```
POSTGRES_PASSWORD=secret123
API_KEY=abc123xyz
JWT_SECRET=supersecret
STRIPE_KEY=sk_live_...
```

Fichier .env.example (à commit)

```
POSTGRES_PASSWORD=
API_KEY=
JWT_SECRET=
STRIPE_KEY=
```

Dans .gitignore :

```
.env
.env.local
docker-compose.override.yml
```

Substitution de variables

```
services:  
  postgres:  
    image: postgres:${ POSTGRES_VERSION:-18-alpine }  
    ports:  
      - "${ POSTGRES_PORT:-5432 }:5432"  
    environment:  
      POSTGRES_PASSWORD: ${ POSTGRES_PASSWORD }
```

Syntaxe :

- `${VAR}` : Variable obligatoire
- `${VAR:-default}` : Variable avec valeur par défaut
- `${VAR-default}` : Variable optionnelle

Fichier `.env` :

```
POSTGRES_VERSION=18-alpine  
POSTGRES_PORT=5432  
POSTGRES_PASSWORD=secret
```

Env file

```
services:  
  backend:  
    env_file:  
      - .env  
      - .env.local  
      - .env.production
```

Ordre de priorité :

- 1 Variables définies dans `environment`
- 2 Variables du dernier `env_file`
- 3 Variables du premier `env_file`

Le dernier fichier écrase les précédents !

Environnements multiples

Structure de projet :

```
project/
└── docker-compose.yml
└── .env.dev
└── .env.staging
└── .env.prod
└── .gitignore
```

Lancer avec un environnement spécifique :

```
# Development
docker compose --env-file .env.dev up -d
# Staging
docker compose --env-file .env.staging up -d
# Production
docker compose --env-file .env.prod up -d
```

Override files

```
# docker-compose.yml (base)
services:
  backend:
    image: myapp:latest
    environment:
      NODE_ENV: production
```

```
# docker-compose.override.yml (développement)
services:
  backend:
    build: ./backend
    volumes:
      - ./backend/src:/app/src
    environment:
      NODE_ENV: development
    command: npm run dev
```

Par défaut, Compose charge :

- 1 docker-compose.yml
- 2 docker-compose.override.yml (si existe)

Override files : Cas d'usage

```
# Development (utilise override automatiquement)
docker compose up -d

# Production (ignore override)
docker compose -f docker-compose.yml up -d

# Staging (fichier custom)
docker compose -f docker-compose.yml -f docker-compose.staging.yml up -d
```

Structure recommandée :

```
project/
  └── docker-compose.yml          # Base (production)
      ├── docker-compose.override.yml # Dev (dans .gitignore)
      ├── docker-compose.dev.yml    # Dev (versionné)
      ├── docker-compose.staging.yml # Staging
      └── docker-compose.prod.yml   # Production
```

Exemple complet : Multi-environnement

```
# docker-compose.prod.yml
services:
  backend:
    image: myapp:1.0.0
    restart: always
    environment:
      NODE_ENV: production
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
```

Secrets : Ne jamais faire ça !

✖ Mauvaise pratique

```
services:  
  backend:  
    environment:  
      DATABASE_PASSWORD: secret123      # JAMAIS ÇA !  
      API_KEY: sk_live_abcl23          # JAMAIS ÇA !
```

Problèmes :

- Secrets en clair dans Git
- Visible dans `docker inspect`
- Accessible via logs



Chapitre 7

Dépendances et Health checks

Dépendances : depends_on

```
services:  
  backend:  
    image: node:20-alpine  
    depends_on:  
      - postgres  
      - redis  
  
  postgres:  
    image: postgres:18-alpine  
  
  redis:  
    image: redis:alpine
```

Comportement :

- PostgreSQL et Redis démarrent **avant** backend
- Garantit seulement l'ordre de démarrage
- **Ne garantit PAS** que le service soit prêt

Le problème de depends_on

```
services:  
  backend:  
    depends_on:  
      - postgres  
    command: npm start  
  
  postgres:  
    image: postgres:18-alpine
```

Problème :

- 1 PostgreSQL démarre
- 2 Backend démarre immédiatement après
- 3 PostgreSQL n'est pas encore prêt à accepter des connexions
- 4 Backend crash avec "Connection refused"

Solution : Health checks

```
services:  
  postgres:  
    image: postgres:18-alpine  
    healthcheck:  
      test: ["CMD-SHELL", "pg_isready -U postgres"]  
      interval: 10s  
      timeout: 5s  
      retries: 5  
      start_period: 30s
```

Options :

- `test` : Commande pour tester la santé
- `interval` : Intervalle entre les tests (10s)
- `timeout` : Timeout du test (5s)
- `retries` : Nombre de tentatives avant de marquer unhealthy (5)

depends_on avec condition

```
services:  
  backend:  
    depends_on:  
      postgres:  
        condition: service_healthy  
      redis:  
        condition: service_healthy  
  
  postgres:  
    image: postgres:18-alpine  
    healthcheck:  
      test: ["CMD-SHELL", "pg_isready -U postgres"]  
      interval: 10s  
      timeout: 5s  
      retries: 5  
  
  redis:  
    image: redis:alpine  
    healthcheck:  
      test: ["CMD", "redis-cli", "ping"]  
      interval: 5s
```

Health check : Exemples par technologie

```
# API avec curl
healthcheck:
  test: ["CMD-SHELL", "curl -f http://localhost:3000/health || exit 1"]
  interval: 30s
  timeout: 10s
  retries: 3
```

Voir l'état de santé

```
# Voir le statut des services
docker compose ps

# Output :
# NAME           STATUS          HEALTH
# postgres        Up 2 minutes (healthy)
# backend         Up 1 minute (healthy)
```

```
# Voir les détails d'un service
docker inspect postgres | grep -A 10 Health

# Voir les logs du health check
docker compose logs postgres
```

depends_on : Conditions disponibles

```
services:  
  app:  
    depends_on:  
      db:  
        condition: service_healthy          # Attend que le service soit healthy  
      cache:  
        condition: service_started         # Attend juste que le service démarre  
      queue:  
        condition: service_completed_successfully  # Attend la fin avec succès
```

Condition	Comportement
service_started	Attend que le service démarre (défaut)
service_healthy	Attend que le health check soit OK
service_completed_successfully	Attend la fin du service (exit 0)

Exemple complet : Stack avec dépendances

```
services:
  frontend:
    build: ./frontend
    depends_on:
      - backend:
          condition: service_healthy

  backend:
    build: ./backend
    depends_on:
      - postgres:
          condition: service_healthy
      - redis:
          condition: service_healthy
  healthcheck:
    test: ["CMD", "wget", "--spider", "-q", "http://localhost:3000/health"]
    interval: 30s
    timeout: 10s
    retries: 3
```

Ordre de démarrage

1. postgres démarre
↓ (attend healthy)
2. redis démarre
↓ (attend healthy)
3. backend démarre
↓ (attend healthy)
4. frontend démarre

Garantie : Chaque service attend que ses dépendances soient réellement prêtes !



Chapitre 8

Commandes Docker Compose

Cycle de vie

```
# Démarrer les services
docker compose up

# Démarrer en arrière-plan
docker compose up -d

# Arrêter les services
docker compose stop

# Redémarrer
docker compose restart

# Arrêter et supprimer
docker compose down

# Supprimer + volumes
docker compose down -v
```

Différences :

- `stop` : Arrête les conteneurs (conserve tout)
- `down` : Arrête + supprime conteneurs + réseaux
- `down -v` : + supprime les volumes

Attention : `down -v` supprime toutes les données !

Observation et monitoring

```
# 100 dernières lignes
docker compose logs --tail=100 backend
```

Exécution de commandes

```
# Ouvrir un shell dans un conteneur
docker compose exec backend sh

# Avec bash si disponible
docker compose exec backend bash

# Exécuter une commande
docker compose exec backend npm test

# Exécuter une commande en tant que root
docker compose exec -u root backend apt update
```

exec = exécute dans un conteneur déjà en cours d'exécution

Run vs Exec

```
# exec : Dans un conteneur en cours
docker compose exec backend npm test
```

```
# run : Crée un nouveau conteneur temporaire
docker compose run backend npm test
docker compose run --rm backend npm test      # Supprime après exécution
```

Commande

Usage

exec

Commande dans un conteneur existant

run

Crée un nouveau conteneur temporaire

Build

```
# Construire toutes les images
docker compose build

# Construire une image spécifique
docker compose build backend

# Construire sans cache
docker compose build --no-cache

# Construire et démarrer
docker compose up --build

# Reconstruire si Dockerfile modifié
docker compose up --build -d
```

Scaling

```
# Lancer 3 instances du backend
docker compose up -d --scale backend=3

# Voir les instances
docker compose ps
```

Attention : Ne fonctionne que si vous n'avez pas spécifié `container_name` et si les ports ne sont pas mappés de manière fixe !

```
services:
  backend:
    image: myapp:latest
    # container_name: backend  # ✗ Retirer ça pour scaler
```

Gestion des ressources

```
# Voir les processus  
docker compose top
```

```
# Voir les statistiques  
docker compose stats
```

```
# Voir l'utilisation disque  
docker compose images
```

```
# Mettre en pause  
docker compose pause backend
```

```
# Reprendre  
docker compose unpause backend
```

Nettoyage

```
# Supprimer les conteneurs arrêtés  
docker compose rm  
  
# Forcer la suppression  
docker compose rm -f  
  
# Supprimer tout (conteneurs + réseaux + volumes)  
docker compose down -v  
  
# Nettoyer le système Docker  
docker system prune -a --volumes
```

Attention : `system prune -a --volumes` supprime TOUT ce qui n'est pas utilisé !

Configuration et validation

```
# Voir la configuration résolue  
docker compose config  
  
# Valider le fichier sans lancer  
docker compose config --quiet  
  
# Voir les services  
docker compose config --services  
  
# Voir les volumes  
docker compose config --volumes
```

Tip : Utilisez `config` pour débugger les variables d'environnement et les override files !

Pull et push d'images

```
# Télécharger toutes les images
docker compose pull

# Télécharger une image spécifique
docker compose pull postgres

# Pousser les images vers un registry
docker compose push
```

Commandes avancées

```
# Créer les services sans les démarrer
docker compose create

# Démarrer des services créés
docker compose start

# Voir les événements
docker compose events

# Copier un fichier
docker compose cp backend:/app/logs/app.log ./local.log

# Port d'un service
docker compose port backend 3000
```

Commandes utiles au quotidien

```
# Redémarrer un service spécifique
docker compose restart backend

# Voir les logs en temps réel de plusieurs services
docker compose logs -f backend frontend

# Rebuild et redémarrer un service
docker compose up -d --build backend

# Supprimer tout et recommencer
docker compose down -v && docker compose up --build -d

# Debug : Voir la config finale
docker compose config
```

Commandes essentielles

Commande	Description
<code>docker compose up -d</code>	Démarrer en arrière-plan
<code>docker compose down</code>	Arrêter et supprimer
<code>docker compose ps</code>	Voir les services
<code>docker compose logs -f</code>	Suivre les logs
<code>docker compose exec <service> sh</code>	Ouvrir un shell
<code>docker compose restart <service></code>	Redémarrer un service
<code>docker compose build</code>	Construire les images

Pause - 15 minutes



À votre retour : Atelier pratique

Transformer le TP PostgreSQL + Adminer de hier en Docker Compose



Atelier 3 - Guidé

PostgreSQL + Adminer en Docker Compose

Objectif de l'atelier

Transformer le TP de hier en Docker Compose

Hier, vous avez créé manuellement :

```
docker network create demo-network
docker volume create postgres-data
docker run -d --name postgres ...
docker run -d --name adminer ...
```

Aujourd'hui : Un seul fichier `docker-compose.yml` et une seule commande !

Étape 1 : Structure du projet

```
mkdir tp-compose-postgres-adminer
cd tp-compose-postgres-adminer
touch docker-compose.yml
```

Structure finale :

```
tp-compose-postgres-adminer/
└── docker-compose.yml
└── init.sql (optionnel)
└── .env (optionnel)
```

Étape 2 : Service PostgreSQL

```
version: '3.9'

services:
  postgres:
    image: postgres:18-alpine
    container_name: demo-postgres
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret123
      POSTGRES_DB: demo_db
      PGDATA: /var/lib/postgresql/18/docker
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U admin"]
      interval: 10s
      timeout: 5s
      retries: 5
    restart: unless-stopped
```

Étape 3 : Service Adminer

```
services:  
  postgres:  
    # ... (configuration précédente)  
  
  adminer:  
    image: adminer:latest  
    container_name: demo-adminer  
    ports:  
      - "8080:8080"  
    depends_on:  
      postgres:  
        condition: service_healthy  
    restart: unless-stopped
```

Note : Pas besoin de déclarer explicitement le réseau, Compose en crée un automatiquement !

Fichier complet : docker-compose.yml

```
version: '3.9'

services:
  postgres:
    image: postgres:18-alpine
    container_name: demo-postgres
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret123
      POSTGRES_DB: demo_db
      PGDATA: /var/lib/postgresql/18/docker
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U admin"]
      interval: 10s
      timeout: 5s
```

Étape 4 : Démarrer la stack

```
# Démarre les services
docker compose up -d

# Vérifie que tout fonctionne
docker compose ps

# Affiche les logs
docker compose logs -f
```

Ouvrez Adminer : <http://localhost:8080>

Connexion :

- Système : PostgreSQL
- Serveur : `postgres`
- Utilisateur : `admin`
- Mot de passe : `secret123`

Exercice pratique : Tester la persistance

1. Créez une table dans Adminer

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com');
```

2. Arrêtez la stack

Étape 5 : Variables d'environnement (optionnel)

Créez un fichier `.env` :

```
POSTGRES_USER=admin
POSTGRES_PASSWORD=secret123
POSTGRES_DB=demo_db
POSTGRES_PORT=5432
ADMINER_PORT=8080
```

Modifiez `docker-compose.yml` :

```
services:
  postgres:
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

Étape 6 : Init script SQL (bonus)

Créez `init.sql` :

```
CREATE TABLE IF NOT EXISTS products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO products (name, price) VALUES
    ('Laptop', 999.99),
    ('Mouse', 29.99),
    ('Keyboard', 79.99);
```

Modifiez `docker-compose.yml` :

```
postgres:
  environment:
```

Étape 7 : Explorer les commandes

```
# Voir les logs de PostgreSQL
docker compose logs postgres

# Voir les logs d'Adminer
docker compose logs adminer

# Ouvrir un shell dans PostgreSQL
docker compose exec postgres psql -U admin -d demo_db

# Lister les réseaux créés
docker network ls | grep tp-compose

# Lister les volumes créés
docker volume ls | grep tp-compose
```

Commandes SQL dans le conteneur

```
# Se connecter à PostgreSQL
docker compose exec postgres psql -U admin -d demo_db

# Dans le shell psql :
\dt                      # Lister les tables
\d users                  # Décrire la table users
SELECT * FROM users;     # Requête
\q                       # Quitter
```

Étape 8 : Nettoyage

```
# Arrêter et supprimer les conteneurs + réseaux  
docker compose down  
  
# Arrêter et supprimer les conteneurs + réseaux + volumes  
docker compose down -v
```

Attention : `-v` supprime les volumes et donc toutes les données !

Comparaison : Avant / Après

Avant (Docker CLI)

```
# 6+ commandes
docker network create ...
docker volume create ...
docker run -d --name postgres \
--network ... \
-v ... \
-e ... \
postgres:18-alpine
docker run -d --name adminer \
--network ... \
-p 8080:8080 \
adminer
```

Après (Docker Compose)

```
# 1 commande
docker compose up -d
```

Fichier versionné, réproductible, partageable !

Points clés de l'atelier

Ce que vous avez appris :

- Créer un fichier docker-compose.yml
- Définir des services avec images
- Configurer des volumes persistants
- Utiliser les health checks
- Gérer les dépendances

Bonnes pratiques appliquées :

- Named volumes pour les données
- Health checks avant démarrage
- Restart policies
- Variables d'environnement
- Init scripts SQL

Exercices bonus (si temps)

1. Ajouter un réseau custom

```
networks:  
  demo-network:  
    driver: bridge
```

2. Ajouter un service pgAdmin au lieu d'Adminer

```
pgadmin:  
  image: dpage/pgadmin4  
  environment:  
    PGADMIN_DEFAULT_EMAIL: admin@admin.com  
    PGADMIN_DEFAULT_PASSWORD: admin  
  ports:  
    - "5050:80"
```

3. Configurer des limites de ressources



Projet autonome

Créer le docker-compose de demo-projet-devops

Objectif du projet

Créer le `docker-compose.yml` pour votre projet fullstack

Architecture :

- **Frontend** : React/TypeScript (Vite)
- **Tasks Service** : Backend API (Express + PostgreSQL)
- **Stats Service** : Backend API (Express, appelle Tasks Service)
- **PostgreSQL** : Base de données

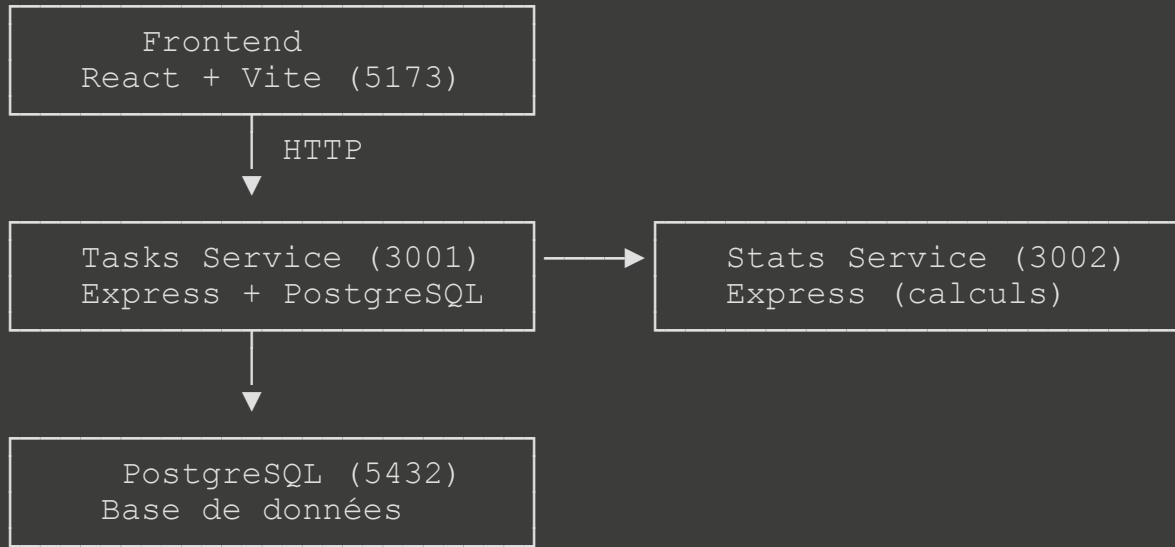
Temps estimé : 1h en autonomie

À la fin, vous aurez une stack complète qui démarre avec une seule commande !

Structure du projet demo-projet-devops

```
demo-projet-devops/
└── frontend/
    ├── Dockerfile
    ├── package.json
    └── src/
└── tasks-service/
    ├── Dockerfile
    ├── package.json
    └── init.sql
    └── src/
└── stats-service/
    ├── Dockerfile
    ├── package.json
    └── src/
└── docker-compose.yml  ← À créer !
```

Schéma de l'architecture



Étape 1 : Service PostgreSQL

Indications :

- Image : `postgres:18-alpine`
- Variables d'environnement : `POSTGRES_USER` , `POSTGRES_PASSWORD` , `POSTGRES_DB` ,
`PGDATA=/var/lib/postgresql/18/docker`
- Port : `5432:5432`
- Volume : Pour persister `/var/lib/postgresql`
- Init script : Montez `./tasks-service/init.sql` dans `/docker-entrypoint-initdb.d/`
- Health check : `pg_isready -U <user>`
- Restart : `unless-stopped`

Nom du service : `postgres`

Étape 2 : Service Tasks

Indications :

- Build depuis `./tasks-service`
- Port : `3001:3001`
- Variables d'environnement :
 - `PORT=3001`
 - `NODE_ENV=development`
 - `DATABASE_URL=postgres://user:pass@postgres:5432/db`
- Volume : Bind mount `./tasks-service/src` vers `/app/src` (hot-reload)
- Volume : Exclude `/app/node_modules`
- Depends on : `postgres` (avec condition `service_healthy`)
- Command : `npm run dev`
- Restart : `unless-stopped`

Étape 3 : Service Stats

Indications :

- Build depuis `./stats-service`
- Port : `3002:3002`
- Variables d'environnement :
 - `PORT=3002`
 - `NODE_ENV=development`
 - `TASKS_SERVICE_URL=http://tasks-service:3001`
- Volume : Bind mount `./stats-service/src` vers `/app/src`
- Volume : Exclude `/app/node_modules`
- Depends on : `tasks-service`
- Command : `npm run dev`
- Restart : `unless-stopped`

Étape 4 : Service Frontend

Indications :

- Build depuis `./frontend`
- Port : `5173:5173` (Vite dev server)
- Variables d'environnement :
 - `VITE_TASKS_API_URL=http://localhost:3001/api/tasks`
 - `VITE_STATS_API_URL=http://localhost:3002/api/stats`
- Volume : Bind mount `./frontend/src` et `./frontend/public`
- Volume : Exclude `/app/node_modules`
- Depends on : `tasks-service` , `stats-service`
- Command : `npm run dev`
- Restart : `unless-stopped`

Nom du service : `frontend`

Étape 5 : Réseau et volumes

```
networks:  
  app-network:  
    driver: bridge  
    name: task-manager-network  
  
volumes:  
  postgres-data:  
    name: task-manager-postgres-data
```

N'oubliez pas :

- Tous les services doivent être sur le réseau `app-network`
- Le volume `postgres-data` doit être monté sur PostgreSQL

Étape 6 : Fichier .env

Créez un fichier `.env` à la racine :

```
# PostgreSQL
POSTGRES_USER=taskuser
POSTGRES_PASSWORD=taskpass
POSTGRES_DB=taskdb

# Ports
POSTGRES_PORT=5432
TASKS_PORT=3001
STATS_PORT=3002
FRONTEND_PORT=5173

# Environment
NODE_ENV=development
```

Étape 7 : Dockerfiles pour les services

Vous devez créer 3 Dockerfiles (un par service).

Exemple pour tasks-service (Dockerfile) :

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 3001

CMD ["npm", "run", "dev"]
```

Structure finale attendue

```
demo-projet-devops/
└── docker-compose.yml
    ├── .env
    ├── .gitignore
    └── frontend/
        ├── Dockerfile
        └── ...
    └── tasks-service/
        ├── Dockerfile
        ├── init.sql
        └── ...
    └── stats-service/
        ├── Dockerfile
        └── ...
```

Étape 8 : Tester votre stack

```
# Démarrer la stack
docker compose up --build -d

# Vérifier les services
docker compose ps

# Vérifier les logs
docker compose logs -f

# Tester les services
# Frontend : http://localhost:5173
# Tasks API : http://localhost:3001/api/tasks
# Stats API : http://localhost:3002/api/stats
```

Debug : Commandes utiles

Si un service ne démarre pas :

```
# Voir les logs d'un service spécifique
docker compose logs tasks-service

# Voir les logs en temps réel
docker compose logs -f tasks-service

# Ouvrir un shell dans un conteneur
docker compose exec tasks-service sh

# Vérifier la connectivité réseau
docker compose exec tasks-service ping postgres

# Voir la configuration résolue
docker compose config
```

Étape 9 : Tester le hot-reload

Frontend :

- 1 Modifiez un fichier dans `frontend/src/`
- 2 Le navigateur doit se recharger automatiquement

Backend (tasks-service) :

- 1 Modifiez un fichier dans `tasks-service/src/`
- 2 Le serveur doit redémarrer automatiquement
- 3 Vérifiez les logs : `docker compose logs -f tasks-service`

Le hot-reload fonctionne grâce aux bind mounts !

Étape 10 : Tester la persistance

1. Créez une tâche via le frontend

2. Arrêtez la stack

```
docker compose down
```

3. Redémarrez la stack

```
docker compose up -d
```

4. Vérifiez que la tâche est toujours là

Les données persistent grâce au volume `postgres-data` !

Critères de réussite

Votre stack doit :

- Démarrer avec `docker compose up -d`
- Tous les services doivent être "healthy" ou "running"
- Le frontend doit être accessible sur <http://localhost:5173>
- Les APIs doivent répondre (3001 et 3002)
- Les données PostgreSQL doivent persister après `docker compose down`
- Le hot-reload doit fonctionner (modification de code → recharge automatique)

Bonus :

- Health checks sur tous les services
- Fichier `.env` pour les variables sensibles
- Réseau custom avec nom explicite

Conseils et astuces

Ordre de création recommandé :

- 1 Service PostgreSQL (le plus simple)
- 2 Service Tasks (dépend de PostgreSQL)
- 3 Service Stats (dépend de Tasks)
- 4 Service Frontend (dépend de tous)

Tests intermédiaires :

- Testez chaque service individuellement avant d'ajouter le suivant
- Vérifiez les logs à chaque ajout

Variables d'environnement :

- Utilisez `${VAR_NAME}` dans le docker-compose.yml
- Définissez les valeurs dans `.env`

Erreurs courantes

Problème : Service ne démarre pas

- Vérifier les logs
- Vérifier le Dockerfile
- Vérifier les variables d'env

Problème : Connection refused

- Vérifier le réseau
- Vérifier le nom du service
- Vérifier les health checks

Problème : Hot-reload ne fonctionne pas

- Vérifier les bind mounts
- Vérifier les chemins
- Vérifier l'exclusion node_modules

Problème : Données perdues

- Vérifier le volume
- Ne pas utiliser `-v` dans down
- Vérifier le chemin de montage

Ressources

Documentation officielle :

- [Docker Compose Docs](#)
- [Compose file reference](#)

Exemples de configuration :

- Variables : `${VAR:-default}`
- Depends on : `condition: service_healthy`
- Health checks : `test , interval , timeout , retries`

Si vous êtes bloqué :

- Regardez les logs : `docker compose logs -f`
- Vérifiez les réseaux : `docker network inspect <network_name>`
- Vérifiez les volumes : `docker volume inspect <volume_name>`

Exercices bonus (si temps)

1. Ajouter un service de cache Redis

```
redis:  
  image: redis:alpine  
  ports:  
    - "6379:6379"
```

2. Ajouter un reverse proxy Nginx

- Routage : `/` → frontend, `/api` → backend

3. Configurer des health checks sur tous les services

4. Créer un fichier `docker-compose.prod.yml`

- Sans bind mounts
- Avec limites de ressources

À vous de jouer !

Bon courage !

N'hésitez pas à demander de l'aide si besoin