

Docker Compose

IUT Sénart-Fontainebleau

2025

Plan du cours

1. Introduction à Docker Compose

- Problématique multi-conteneurs
- Avantages de Docker Compose
- Structure d'un fichier docker-compose.yml

2. Démonstration en direct

- Application web avec base de données
- Configuration et déploiement
- Gestion des services

1. Pourquoi Docker Compose ?

Cas d'usage courant

- Application web (Frontend)
- API (Backend)
- Base de données
- Cache Redis
- Proxy reverse

Sans Docker Compose

```
docker run --name db -e POSTGRES_PASSWORD=secret postgres
docker run --name backend --link db:db myapp
docker run --name frontend -p 80:80 nginx
```

Problèmes sans Docker Compose

Gestion manuelle

- Ordre de démarrage
- Variables d'environnement
- Configuration réseau
- Volumes persistants
- Redémarrage automatique

Risques

- Erreurs humaines
- Configuration incohérente
- Perte de temps
- Documentation complexe

Docker Compose : La solution

Avantages

- Un seul fichier de configuration
- Gestion des dépendances
- Environnement reproductible
- Commandes simplifiées
- Documentation as Code

```
version: '3'
services:
  web:
    build: .
    ports:
      - "80:80"
  db:
    image: postgres
```

Structure du fichier docker-compose.yml

Éléments principaux

```
version: '3'           # Version de la syntaxe
services:              # Définition des conteneurs
networks:              # Configuration réseau
volumes:               # Stockage persistant
configs:               # Configuration applicative
secrets:               # Données sensibles
```

2. Démonstration en direct

Préparation de l'environnement

1. Structure du projet

```
mkdir demo-docker-compose  
cd demo-docker-compose
```

2. Création des dossiers

```
mkdir api frontend  
touch docker-compose.yml .env
```

Étape 1: Configuration de base

1. Création du fichier .env

```
# .env
POSTGRES_USER=demouser
POSTGRES_PASSWORD=secret123
POSTGRES_DB=demodb
NODE_ENV=development
```


2. Premier docker-compose.yml

```
version: '3.8'
services:
  db:
    image: postgres:13
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
```

3. Test initial

```
docker compose up db -d
docker compose ps
docker compose logs db
```

Étape 2: API Node.js

1. Structure de l'API

```
cd api  
npm init -y  
npm install express pg cors
```

2. API simple (index.js)

```
const express = require('express');
const cors = require('cors');

const { Pool } = require('pg');
const app = express();
app.use(cors());

const pool = new Pool({
  host: 'db',
  user: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD,
  database: process.env.POSTGRES_DB
});

app.get('/health', (req, res) => {
  res.json({ status: 'ok' });
});

app.listen(5000, () => console.log('API running'));
```

3. Création du Dockerfile API

```
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "index.js"]
```

4. Ajout au docker-compose.yml

```
api:
  build: ./api
  environment:
    - POSTGRES_USER=${POSTGRES_USER}
    - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    - POSTGRES_DB=${POSTGRES_DB}
  depends_on:
    - db
```

Étape 3: Frontend Vite

1. Création du frontend

```
npm create vite@latest .  
# Suivre les instructions pour sélectionner React et TypeScript/JavaScript  
  
npm install  
  
# ajouter le fichier  
import HealthCheck from './HealthCheck'  
<HealthCheck />
```

2. Dockerfile Frontend

```
FROM node:18-alpine as build  
WORKDIR /app  
COPY package*.json ./  
RUN npm install  
COPY . .  
RUN npm run build  
  
# Production stage  
FROM nginx:alpine  
COPY --from=build /app/dist /usr/share/nginx/html  
COPY nginx.conf /etc/nginx/conf.d/default.conf  
EXPOSE 80  
CMD ["nginx", "-g", "daemon off;"]
```

3. Ajout au docker-compose.yml

```
frontend:  
  build: ./frontend  
  environment:  
    - VITE_API_URL=http://localhost:5000  
  ports:  
    - "80:80"  
  depends_on:  
    - api
```

4. Configuration Vite (vite.config.js)

```
export default defineConfig({  
  plugins: [react()],  
  server: {  
    host: true, // Nécessaire pour Docker  
    port: 5173,  
    watch: {  
      usePolling: true // Pour le hot reload dans Docker  
    }  
  }  
})
```


Étape 4: Nginx Reverse Proxy

1. Configuration Nginx

```
cd frontend  
touch nginx.conf
```

2. nginx.conf basique

```
server {  
    listen 80;  
    server_name localhost;  
  
    # Racine du site React  
    root /usr/share/nginx/html;  
    index index.html;  
  
    # Configuration pour React Router  
    location / {  
        try_files $uri $uri/ /index.html;  
    }  
  
    # Proxy vers le backend  
    location /api/ {  
        proxy_pass http://api:5000/;  
        proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade;  
        proxy_set_header Connection 'upgrade';  
        proxy_set_header Host $host;  
        proxy_cache_bypass $http_upgrade;  
        proxy_set_header X-Real-IP $remote_addr;  
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    }  
  
    # Configuration pour les assets statiques  
    location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {  
        expires max;  
        log_not_found off;  
    }  
}
```

Démonstration du déploiement

1. Lancement complet

```
docker compose up -d
```

2. Vérification des services

```
docker compose ps  
docker compose logs
```

Démonstration : Structure du projet

```
./mon-app/  
├── docker-compose.yml  
├── frontend/  
│   ├── Dockerfile  
│   └── src/  
├── api/  
│   ├── Dockerfile  
│   └── src/  
└── .env
```

Commandes essentielles

Démarrage

```
# Démarrer tous les services  
docker compose up
```

```
# Mode détaché  
docker compose up -d
```

Commandes essentielles

Gestion

```
# Voir les logs  
docker compose logs
```

```
# État des services  
docker compose ps
```

```
# Arrêter les services  
docker compose down
```

Variables d'environnement

Fichier .env

```
POSTGRES_USER=myapp  
POSTGRES_PASSWORD=secret  
POSTGRES_DB=myapp  
API_PORT=3000
```

Utilisation dans docker-compose.yml

```
services:  
  api:  
    environment:  
      - DB_USER=${POSTGRES_USER}  
      - DB_PASSWORD=${POSTGRES_PASSWORD}
```

Réseaux Docker Compose

Configuration réseau

```
services:
  frontend:
    networks:
      - frontend-net

  api:
    networks:
      - frontend-net
      - backend-net

networks:
  frontend-net:
  backend-net:
    internal: true
```


Volumes et persistance

Types de volumes

```
services:
  db:
    volumes:
      # Volume nommé
      - db-data:/var/lib/postgresql/data

      # Bind mount
      - ./config:/etc/postgresql

      # Volume anonyme
      - /var/lib/postgresql/data

volumes:
  db-data:
```

Dépendances entre services

Configuration

```
services:
  api:
    depends_on:
      db:
        condition: service_healthy

  db:
    healthcheck:
      test: ["CMD", "pg_isready"]
      interval: 10s
      timeout: 5s
      retries: 5
```

Mise à l'échelle

Scale services

```
# Lancer 3 instances de l'API
docker compose up --scale api=3

# Configuration load balancer
services:
  api:
    deploy:
      replicas: 3
```

Surveillance et debugging

Logs

```
# Suivre les logs en temps réel  
docker compose logs -f api
```

```
# Logs depuis un timestamp  
docker compose logs --since 30m
```

Shell dans un conteneur

```
docker compose exec api sh
```

Proxy Reverse avec Docker Compose

Pourquoi un proxy reverse ?

- Répartition de charge
- SSL/TLS termination
- Sécurité et filtrage
- Routage des requêtes
- Mise en cache

Exemple de configuration Nginx

```
# nginx.conf
http {
    upstream frontend {
        server frontend:3000;
    }

    upstream api {
        server api:5000;
    }

    server {
        listen 80;
        server_name example.com;

        location / {
            proxy_pass http://frontend;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        location /api {
            proxy_pass http://api;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

Bonnes pratiques

Organisation

1. Séparer les environnements
2. Utiliser les variables d'environnement
3. Gérer les secrets correctement
4. Optimiser les builds
5. Monitorer les services

Sécurité

1. Limiter les ports exposés
2. Utiliser des réseaux internes
3. Mettre à jour les images
4. Scan des vulnérabilités

Questions ?

N'hésitez pas à poser vos questions !

