Lab : Déploiement d'une Application Multi-Tiers avec Docker Compose

Objectifs

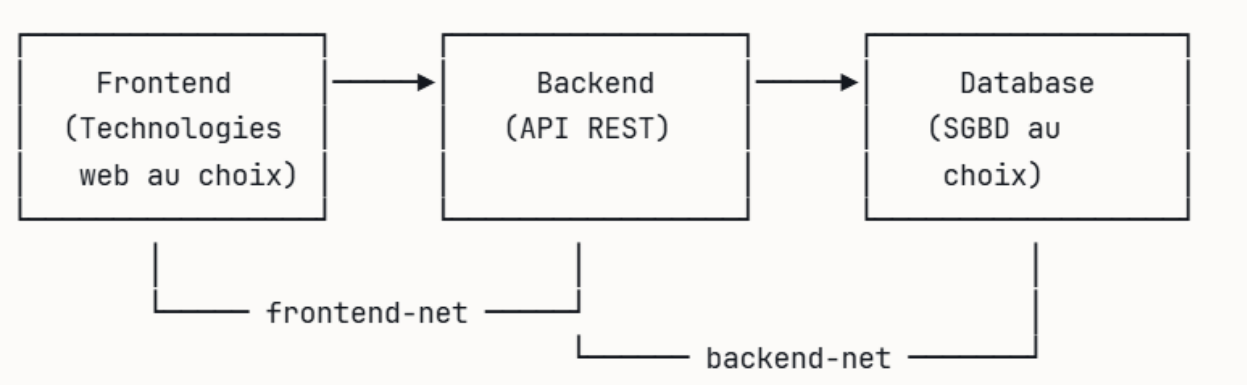
À l'issue de ce laboratoire, l'étudiant sera capable de :

1. Concevoir et développer une application web full-stack simple
2. Rédiger des Dockerfiles optimisés avec approche multi-stage build
3. Orchestrer une architecture multi-conteneurs avec Docker Compose
4. Implémenter une segmentation réseau pour isoler les composants
5. Configurer la persistance des données avec les volumes Docker
6. Paramétrer les applications via variables d'environnement dynamiques
7. Appliquer les bonnes pratiques DevOps (health checks, limites de ressources, sécurité)

Contexte du Projet

Vous devez concevoir, développer et déployer une application web complète suivant une architecture trois-tiers. L'application doit permettre une interaction CRUD (Create, Read, Update, Delete) basique avec une base de données.

Architecture Cible



Principe de sécurité : Le frontend ne doit JAMAIS communiquer directement avec la base de données.

Partie 1 : Conception de l'Application

1.1 Choix Technologiques

Vous êtes libre de choisir les technologies pour chaque composant, dans la limite des contraintes suivantes :

Frontend (au choix parmi)

- React / Vue.js / Angular
- HTML/CSS/JavaScript vanilla avec un serveur statique
- Tout autre framework frontend moderne

Contraintes obligatoires :

- Interface utilisateur permettant d'afficher, ajouter, modifier ou supprimer des données
- Communication avec le backend via appels HTTP (fetch, axios, etc.)
- **L'URL du backend doit être configurable via variable d'environnement**

Backend (au choix parmi)

- Node.js (Express, Fastify, NestJS)
- Python (Flask, FastAPI, Django)
- Java (Spring Boot)
- Go (Gin, Echo)
- Tout autre langage/framework capable de créer une API REST

Contraintes obligatoires :

- API RESTful avec au minimum 3 endpoints :
  - `GET /health` : Health check
  - `GET /api/[resource]` : Récupération des données
  - `POST /api/[resource]` : Ajout de données
- **Toutes les connexions à la base de données doivent utiliser des variables d'environnement**

Base de Données (au choix parmi)

- PostgreSQL
- MySQL/MariaDB
- MongoDB
- Tout autre SGBD supporté par Docker

Contraintes obligatoires :

- Créer un schéma/collection avec au minimum 3 champs

- Script d'initialisation pour créer la structure et insérer des données de test
- Configuration via variables d'environnement (utilisateur, mot de passe, nom de la base)

1.2 Thématique de l'Application (Suggestions)

Choisissez une thématique simple pour votre application :

- **Gestionnaire de tâches** : titre, description, statut
- **Catalogue de livres** : titre, auteur, année de publication
- **Liste de contacts** : nom, email, téléphone
- **Inventaire de produits** : nom, prix, quantité en stock
- **Journal de bord** : date, titre, contenu
- **Autre thématique** de votre choix (validation recommandée auprès de l'enseignant)

Partie 2 : Dockerisation du Backend

2.1 Dockerfile Backend - Multi-Stage Build

Objectif : Créer un Dockerfile optimisé utilisant la technique du multi-stage build.

Spécifications Techniques Obligatoires

Stage 1 - Builder :

- Utiliser une image de base appropriée pour votre langage
- Installer toutes les dépendances nécessaires
- Compiler/préparer l'application si nécessaire
- Optimiser le cache des layers Docker

Stage 2 - Production :

- Utiliser une image de base légère (alpine de préférence)
- Copier uniquement les artefacts nécessaires depuis le stage builder
- **Créer un utilisateur non-root** et l'utiliser pour exécuter l'application
- **Définir les variables d'environnement** :
    - Port d'écoute de l'application
    - Variables de connexion à la base de données (à remplir via Docker Compose)
    - Toute autre configuration nécessaire
- **Exposer le port** de l'application
- **Implémenter un HEALTHCHECK** :
    - Intervalle : 30 secondes
    - Timeout : 10 secondes
    - Retries : 3
    - Start period : 40 secondes

- Commande : requête HTTP sur l'endpoint `/health`

Questions de Réflexion

1. Pourquoi utiliser un multi-stage build plutôt qu'un Dockerfile simple ?
2. Quels sont les avantages en termes de taille d'image ?
3. Comment cela améliore-t-il la sécurité ?
4. Que contient chaque stage et pourquoi ?

2.2 Fichier .dockerignore

Créez un fichier `.dockerignore` approprié pour exclure :

- Dossiers de dépendances (node_modules, venv, target, etc.)
- Fichiers de configuration locale (.env, .env.local)
- Fichiers Git (.git, .gitignore)
- Documentation (*.md, docs/)
- Fichiers temporaires et de cache

Partie 3 : Dockerisation du Frontend

3.1 Dockerfile Frontend - Build Simple

Objectif : Créer un Dockerfile optimisé pour servir une application frontend.

Spécifications Techniques Obligatoires

- Utiliser une image de base appropriée
- Builder l'application en mode production (si applicable)
- Utiliser un serveur web léger pour servir les fichiers statiques :
  - `nginx:alpine` (recommandé)
  - `serve` pour Node.js
  - `http-server` ou autre serveur statique
- **Définir une variable d'environnement** pour l'URL du backend :
  - Nom suggéré : `API_URL`, `BACKEND_URL`, `REACT_APP_API_URL`, etc.
  - Valeur à définir via Docker Compose en utilisant le **nom du service backend**
- **Créer un utilisateur non-root**
- **Exposer le port** approprié (80 recommandé)
- **Implémenter un HEALTHCHECK** :
  - Intervalle : 30 secondes
  - Commande : requête HTTP sur la racine `/`

Considérations Importantes

- Si vous utilisez un framework avec variables d'environnement au build-time (React, Vue, Angular), réfléchissez à comment les rendre dynamiques
- Le frontend doit pouvoir résoudre le backend via le nom du service Docker Compose
- Prévoir la configuration CORS appropriée

3.2 Fichier .dockerignore

Créez un fichier `.dockerignore` adapté à votre frontend.

Partie 4 : Configuration Docker Compose

4.1 Structure Globale

Créez un fichier `docker-compose.yml` définissant trois services qui fonctionnent ensemble.

Service `database`

Configuration requise :

- Image officielle de votre SGBD choisi (version spécifique, pas `latest`)
- Variables d'environnement pour :
  - Nom de la base de données
  - Utilisateur
  - Mot de passe
  - (Selon le SGBD : port, root password, etc.)
- **Volume nommé** pour la persistance :
  - Monter sur le chemin de données du SGBD
  - Nom du volume : `[nom-base]-data`
- **Réseau** : `backend-net` uniquement
- **Limites de ressources** :

CPU : 0.5 core maximum
Mémoire : 512M maximum

- **HEALTHCHECK** utilisant la commande native du SGBD
- **Restart policy** : `unless-stopped`
- **Script d'initialisation** :
  - Créer un fichier SQL/script d'initialisation
  - Le monter dans le conteneur (bind mount)
  - Utiliser le mécanisme d'initialisation du SGBD (ex: `/docker-entrypoint-initdb.d/`)

Service `backend`

Configuration requise :

- **Build** depuis le répertoire `./backend`
- **Variables d'environnement** :
  - Host de la base : **utiliser le nom du service database**
  - Port de la base
  - Nom de la base de données
  - Utilisateur et mot de passe
  - Port d'écoute de l'application
  - Toute autre configuration nécessaire
- **Dépendances** :
  - `depends_on` : database avec condition `service_healthy`
- **Réseaux** : `frontend-net` ET `backend-net`
- **Limites de ressources** :

CPU : 0.5 core maximum
Mémoire : 256M maximum

- **Restart policy** : `unless-stopped`
- **HEALTHCHECK** : déjà défini dans le Dockerfile

Service `frontend`

Configuration requise :

- **Build** depuis le répertoire `./frontend`
  - Avec argument de build ou variable d'environnement pour l'URL du backend
  - L'URL doit pointer vers le **nom du service backend** (pas localhost)
- **Port mapping** : exposer sur le port 8080 de l'hôte
- **Dépendances** :
  - `depends_on` : backend avec condition `service_healthy`
- **Réseau** : `frontend-net` uniquement
- **Limites de ressources** :

CPU : 0.25 core maximum
Mémoire : 128M maximum

- **Restart policy** : `unless-stopped`
- **HEALTHCHECK** : déjà défini dans le Dockerfile

4.2 Configuration des Réseaux

Définir deux réseaux bridge personnalisés :

1. **frontend-net**
   - Connecte : frontend ↔ backend
   - Isolé de la base de données

2. **backend-net**
    - Connecte : backend ↔ database
    - Isolé du frontend

Objectif de sécurité : Implémenter le principe de moindre privilège - chaque service n'a accès qu'aux services dont il a besoin.

4.3 Configuration des Volumes

Définir un volume nommé pour la persistance :

- Nom explicite (ex: `postgres-data`, `mysql-data`, `mongo-data`)
- Driver : local (par défaut)

4.4 Variables d'Environnement Centralisées

Option 1 - Fichier .env (recommandé) : Créer un fichier `.env` à la racine contenant toutes les variables :

env
# À compléter avec vos propres variables
DB_NAME=...
DB_USER=...
DB_PASSWORD=...
...

Option 2 - Directement dans docker-compose.yml : Définir les variables dans chaque service.

Important : Ne JAMAIS commiter de mots de passe en dur. Utiliser `.env` et l'ajouter à `.gitignore`.

Tests de Validation

Effectuez et documentez les tests suivants :

Test 1 : Connectivité Base de Données
- Vérifier que le backend peut se connecter à la database
- Exécuter une requête simple depuis le conteneur backend
- Documenter la commande et le résultat

Test 2 : API Backend
- Tester l'endpoint health check
- Tester les endpoints de votre API
- Utiliser `curl`, Postman, ou autre outil
- Documenter les requêtes et réponses

Test 3 : Frontend
- Accéder à l'interface via le navigateur

- Vérifier que l'application charge correctement
- Tester les fonctionnalités CRUD
- Capturer des screenshots

Test 4 : Isolation Réseau
- ****Vérifier** que le frontend ne peut PAS contacter directement la database**
- Commande suggérée : essayer de pinger/curl depuis le conteneur frontend
- Documenter que la connexion échoue (preuve de l'isolation)

Test 5 : Persistance des Données
- Ajouter des données via l'interface
- Arrêter tous les conteneurs (`docker-compose down`)
- Redémarrer (`docker-compose up -d`)
- ****Vérifier** que les données sont toujours présentes**
- Si les données sont perdues, le volume n'est pas correctement configuré

Test 6 : Limites de Ressources
- Utiliser `docker stats` pour vérifier les limites
- Documenter que les conteneurs respectent les contraintes CPU/mémoire

Test 7 : Health Checks
- Vérifier que tous les conteneurs passent à l'état "healthy"
- Documenter le temps nécessaire pour atteindre cet état
- Tester le comportement en cas de panne (arrêter la database, observer les health checks)

Contenu du README.md (Obligatoire)

Votre documentation doit inclure les sections suivantes :

Tests et Validation

- Liste des tests effectués (référence Partie 6.2)
- Commandes utilisées
- Résultats obtenus
- Screenshots/captures d'écran démontrant le bon fonctionnement

Commandes Utiles

- Résumé des commandes principales
- Troubleshooting courant

Captures d'Écran Obligatoires

Incluez dans votre README ou dans un dossier `/screenshots/` :

1. `docker-compose ps` montrant tous les services "Up" et "healthy"
2. `docker network ls` montrant vos deux réseaux
3. `docker volume ls` montrant votre volume de données
4. `docker stats` montrant les limites de ressources appliquées

5. Interface frontend fonctionnelle affichant des données
6. Test d'ajout de données via l'interface
7. Logs démontrant la connexion backend → database
8. Test de l'isolation réseau (échec de connexion frontend → database)
9. Health check status dans `docker inspect`
10. Comparaison de taille d'image (backend avec/sans multi-stage si possible)