

Exercice Docker & Docker Compose

Application de Chat IA avec Sauvegarde

Contexte

Vous êtes développeur DevOps dans une entreprise qui souhaite déployer une application de chat utilisant l'intelligence artificielle. L'application est déjà développée, mais elle doit être conteneurisée avec Docker pour faciliter son déploiement.

L'application est composée de **4 services** qui doivent communiquer entre eux :

1. **PostgreSQL** - Base de données pour stocker les conversations
2. **Ollama** - Serveur d'IA pour générer les réponses
3. **Backend** - API Flask (Python) qui gère la logique métier
4. **Frontend** - Interface web (HTML/CSS/JS) pour les utilisateurs

Objectifs de l'exercice

Votre mission est de :

1. **Créer les Dockerfiles** pour containeriser le backend et le frontend
2. **Créer le fichier docker-compose.yml** pour orchestrer tous les services
3. **Configurer les réseaux** pour permettre la communication entre les conteneurs
4. **Configurer les volumes** pour persister les données
5. **Gérer les dépendances** entre les services

Structure du projet fournie

```
ai-chat-app/
├── backend/
│   ├── app.py          ✓ Fourni
│   ├── requirements.txt ✓ Fourni
│   └── Dockerfile       ✗ À CRÉER
├── frontend/
│   ├── index.html      ✓ Fourni
│   ├── style.css       ✓ Fourni
│   ├── script.js       ✓ Fourni
│   └── Dockerfile       ✗ À CRÉER
└── docker-compose.yml  ✗ À CRÉER
```

Partie 1 : Dockerfile du Backend

Informations techniques :

- **Langage** : Python 3.11
- **Framework** : Flask
- **Port** : 5000
- **Fichier principal** : `app.py`
- **Dépendances** : Listées dans `requirements.txt`

Votre mission :

Créez le fichier `backend/Dockerfile` qui doit :

1. Partir d'une image Python 3.11 (utilisez `python:3.11-slim`)
2. Définir `/app` comme répertoire de travail
3. Copier le fichier `requirements.txt` et installer les dépendances
4. Copier tous les fichiers du backend dans le conteneur
5. Exposer le port 5000
6. Lancer l'application avec la commande : `python app.py`

Questions :

- Pourquoi copie-t-on `requirements.txt` avant les autres fichiers ?
- Quel est l'avantage d'utiliser l'image `slim` ?

Partie 2 : Dockerfile du Frontend

Informations techniques :

- **Serveur web** : Nginx
- **Port** : 80
- **Fichiers** : `index.html`, `style.css`, `script.js`

Votre mission :

Créez le fichier `frontend/Dockerfile` qui doit :

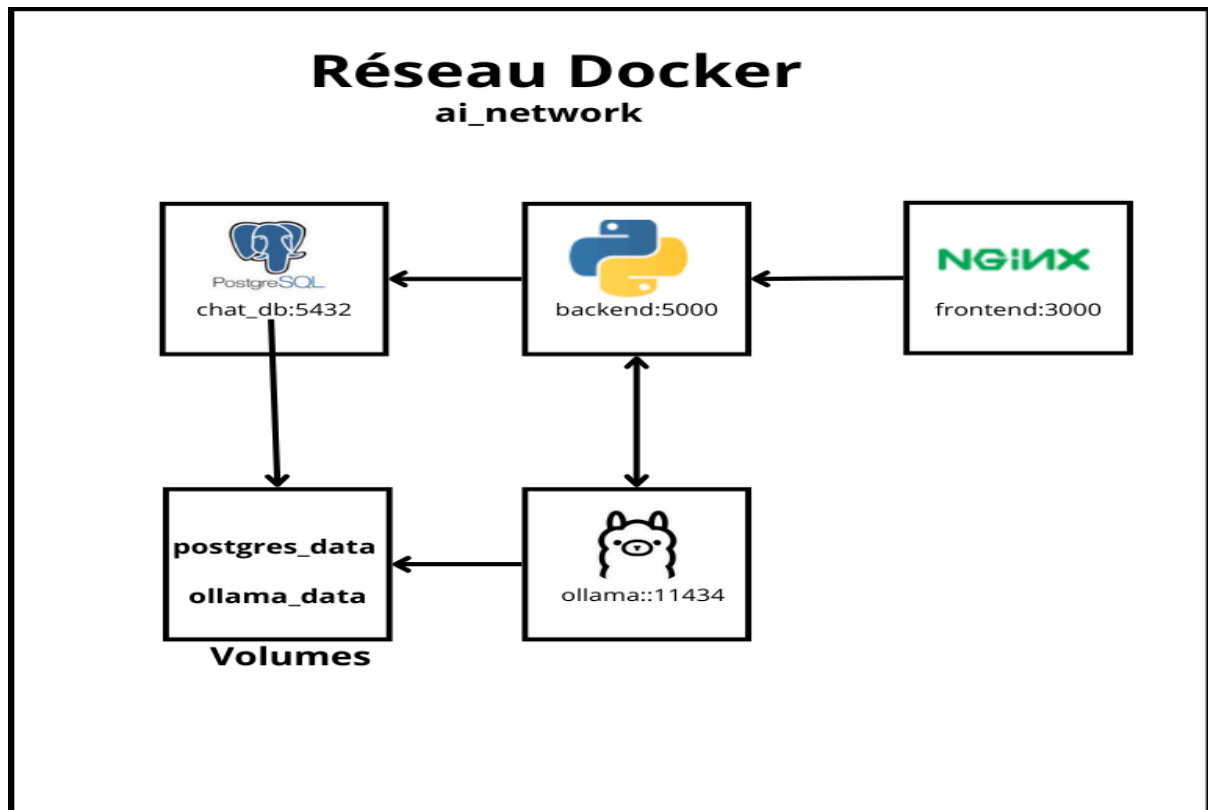
1. Partir d'une image Nginx Alpine (`nginx:alpine`)
2. Copier les 3 fichiers HTML/CSS/JS dans `/usr/share/nginx/html/`
3. Exposer le port 80
4. Démarrer Nginx avec la commande : `nginx -g daemon off;`

Questions :

- Pourquoi utiliser l'image Alpine ?
- Où Nginx sert-il les fichiers statiques par défaut ?

Partie 3 : Docker Compose

Architecture à mettre en place :



Votre mission :

Créez le fichier `docker-compose.yml` qui définit **4 services** :

Service 1 : **postgres**

- Image : `postgres:15-alpine`
- Nom du conteneur : `chat_db`
- Variables d'environnement :
 - `POSTGRES_USER: chatuser`
 - `POSTGRES_PASSWORD: chatpass`
 - `POSTGRES_DB: chatdb`
- Port : Mapper 5432:5432
- Volume : Monter `postgres_data` sur `/var/lib/postgresql/data`

- **Health check** : Utiliser la commande `pg_isready -U chatuser`
 - Intervalle : 10s
 - Timeout : 5s
 - Retries : 5

Service 2 : `ollama`

- **Image** : `ollama/ollama:latest`
- **Nom du conteneur** : `ollama`
- **Port** : Mapper 11434:11434
- **Volume** : Monter `ollama_data` sur `/root/.ollama`

Service 3 : `backend`

- **Build** : Construire depuis `./backend`
- **Nom du conteneur** : `chat_backend`
- **Port** : Mapper 5000:5000
- **Variables d'environnement** :
 - `OLLAMA_HOST`: `http://ollama:11434`
 - `DATABASE_URL` :
`postgresql://chatuser:chatpass@postgres:5432/chatdb`
- **Dépendances** :
 - Doit attendre que `postgres` soit healthy
 - Doit attendre que `ollama` soit démarré

Service 4 : `frontend`

- **Build** : Construire depuis `./frontend`
- **Nom du conteneur** : `chat_frontend`
- **Port** : Mapper 3000:80
- **Dépendances** : Doit attendre que `backend` soit démarré

Configuration globale :

- **Version** : 3.8
- **Réseau** : Créer un réseau bridge nommé `ai_network`
- **Volumes nommés** : `postgres_data` et `ollama_data`
- **Restart policy** : `unless-stopped` pour tous les services

Points d'attention

Réseau

- Tous les services doivent être sur le même réseau `ai_network`
- Les conteneurs peuvent communiquer entre eux via leur nom de service

- Exemple : le backend contacte postgres via `postgres:5432`

Volumes

- Les volumes permettent de persister les données entre les redémarrages
- Sans volumes, toutes les données sont perdues à l'arrêt du conteneur

Dépendances

- L'ordre de démarrage est crucial
- PostgreSQL doit être prêt (healthy) avant que le backend démarre
- Le frontend dépend du backend

Variables d'environnement

- Permettent de configurer les services sans modifier le code
- Le backend utilise ces variables pour se connecter aux autres services

Critères de validation

Votre solution est correcte si :

1. Build réussi :

`docker-compose build` # Pas d'erreurs

2. Démarrage réussi :

`docker-compose up -d`
`docker-compose ps` # Les 4 conteneurs sont "Up"

3. Connectivité :

- Le backend peut contacter la base de données
`docker-compose logs backend | grep "CREATE TABLE"`
- Le backend peut contacter Ollama
`curl http://localhost:5000/api/health`

4. Application fonctionnelle :

- Ouvrir `http://localhost:3000`
- Créer une nouvelle conversation
- Télécharger Llama 2 (recommandé pour débiter, ~4GB)
`docker exec -it ollama ollama pull llama2`
- Envoyer un message
- Le message est sauvegardé (visible après refresh)

5. Persistance des données :

- Arrêter et redémarrer
docker-compose down
docker-compose up -d

Les conversations doivent toujours être présentes

Commandes utiles

Construction et démarrage

docker-compose build # Construire les images
docker-compose up -d # Démarrer en arrière-plan
docker-compose ps # Voir l'état des conteneurs
docker-compose logs -f # Voir les logs en temps réel

Télécharger le modèle IA

docker exec -it ollama ollama pull llama2

Débogage

docker-compose logs backend # Logs du backend
docker-compose logs postgres # Logs de la BDD
docker exec -it chat_db psql -U chatuser -d chatdb # Se connecter à PostgreSQL

Nettoyage

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

Questions de réflexion

1. Pourquoi utiliser un health check pour PostgreSQL ?
 - Qu'est-ce qui pourrait mal se passer sans ?
2. Que se passe-t-il si on inverse l'ordre des dépendances ?
 - Essayez de démarrer le backend avant PostgreSQL
3. Pourquoi créer un réseau personnalisé ?
 - Quelle est la différence avec le réseau par défaut ?
4. Les volumes sont-ils vraiment nécessaires ?
 - Testez sans volumes et redémarrez les conteneurs

5. Comment sécuriser cette application pour la production ?

- Quels sont les points faibles actuels ?

Bonus (optionnel)

Si vous terminez rapidement, essayez de :

1. **Ajouter un monitoring** : Service pour visualiser les conteneurs
2. **Docker secrets** : Sécuriser les mots de passe PostgreSQL
3. **Scaling** : Lancer plusieurs instances du backend

Livrable attendu

À la fin de l'exercice, vous devez fournir :

1. `backend/Dockerfile`
2. `frontend/Dockerfile`
3. `docker-compose.yml`
4. Un fichier `REPONSES.md` avec vos réponses aux questions

Bon courage !