Déploiement de notre API dans un conteneur Docker

Qu'est-ce que Docker ?

Docker est un outil **puissant et très populaire** dans l'écosystème tech moderne. Il permet d'**emballer une application et toutes ses dépendances** dans un environnement léger, appelé **conteneur** (*container*), qui peut ensuite être exécuté **partout de la même façon** : sur ton ordinateur, sur un serveur cloud, ou sur une machine distante.

> En résumé : Docker permet de dire **"ça marche chez moi" et que ça marche aussi chez tout le monde**.

Pourquoi Docker est-il si important ?

- **Standardisation** : On évite les problèmes de compatibilité entre différents environnements.
- **Portabilité** : Le même conteneur peut être exécuté localement, sur AWS, Azure, GCP, etc.
- **Isolation** : Chaque conteneur fonctionne de manière indépendante, sans conflit avec les autres applications.
- **Gain de temps** : Fini les longues installations manuelles de dépendances.

Installer Docker

Tu peux installer Docker en suivant les instructions officielles en fonction de ton système d'exploitation (Windows, macOS ou Linux) : https://docs.docker.com/get-docker/

> Prends le temps de bien suivre les étapes. L'installation est généralement rapide.

Vérifier que Docker est bien installé

Une fois l'installation terminée, ouvre un terminal et tape la commande suivante :

```
docker --version
```

Si Docker est bien installé, tu verras s'afficher la version actuelle, comme ceci :

```
Docker version 25.0.3, build 4debf41
```

Étapes du déploiement avec Docker

Voici les **3 grandes étapes** que nous allons suivre pour créer une version Dockerisée de notre API :

1- Créer un fichier `Dockerfile`

Ce fichier décrit **comment construire l'image Docker** à partir du code de ton API :

- Quelle image de base utiliser (Python)
- Comment installer les dépendances
- Quelle commande lancer pour démarrer l'API

Crée un fichier `Dockerfile` dans le dossier api/ et écris ceci dedans :

```
# Utilise une image légère de Python 3.12 comme base
FROM python:3.12-slim

# Définit le répertoire de travail dans le conteneur
WORKDIR /app

# Copie le fichier des dépendances dans le conteneur
COPY requirements.txt .

# Installe les dépendances Python sans mise en cache
RUN pip install --no-cache-dir --upgrade -r requirements.txt

# Copie tous les fichiers .py et le fichier .db dans le conteneur
COPY . .

# Lance le serveur Uvicorn pour exécuter l'API FastAPI
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

Un **Dockerfile** est un fichier texte qui explique à Docker **comment construire une image** contenant ton application (ici, ton API FastAPI).

C'est comme **une recette de cuisine** : chaque ligne du fichier indique une étape pour préparer l'environnement nécessaire à ton application.

```
FROM python:3.12-slim
```

- Cette ligne dit à Docker de **partir d'une image Python légère** (basée sur Python 3.12). C'est comme une base toute prête qui contient Python mais très peu de choses en plus (d'où le mot *slim*).

```
WORKDIR /app
```

- On dit à Docker de **travailler dans le dossier** `/app` dans le conteneur. Tous les fichiers qu'on ajoutera et toutes les commandes qu'on exécutera après se feront dans ce dossier.

```
COPY requirements.txt .
```

- On **copie le fichier** `requirements.txt` de ton projet local vers le dossier `/app` dans le conteneur.
  Ce fichier contient la liste des bibliothèques Python nécessaires (comme `fastapi` et `sqlalchemy`).

```
RUN pip install --no-cache-dir --upgrade -r requirements.txt
```

- On **installe toutes les bibliothèques Python** listées dans `requirements.txt`.
  L'option `--no-cache-dir` évite de stocker les fichiers d'installation (pour que le conteneur soit plus léger).

```
COPY . .
```

- On **copie tous les fichiers de ton dossier** `api/` vers le dossier `/app` du conteneur. Cela inclut tes fichiers `.py` (comme `main.py`, `crud.py`, etc.) et ta base de données `.db`.

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

- C'est la **commande de démarrage de l'API**.
  On utilise `uvicorn`, le serveur qui exécute FastAPI. Il va chercher l'application dans `main.py`, dans la variable `app`, et la rend accessible depuis l'extérieur (grâce à `--host 0.0.0.0`).

Avec ce Dockerfile, tu peux créer une **image Docker contenant ton API** prête à être exécutée.

Mais avant de construire une image Docker, il est **essentiel** de créer un fichier `.dockerignore`.

Voici un exemple **simple et efficace** de `.dockerignore` pour notre projet FastAPI :

```
# Fichiers Python compilés
__pycache__/
*.pyc
*.pyo

# Fichiers de logs
*.log

# Fichiers de configuration d'environnement local
.env
```

```
# Répertoire Git
.git
.gitignore

# Fichiers système
.DS_Store

# Cache pip ou venv
.venv/
venv/
pip-wheel-metadata/
```

Pourquoi un `.dockerignore` est important ?

Le fichier `.dockerignore` fonctionne **exactement comme un `.gitignore`**, mais pour **Docker**. Il indique quels fichiers **ne doivent pas être copiés** dans l'image Docker au moment de la construction.

Sans `.dockerignore` : Docker copierait **tout le dossier** (y compris `.git`, `.venv`, fichiers inutiles, etc.), ce qui rend l'image :

- plus **lourde**
- plus **lente à construire**
- plus **vulnérable** (ex. : si tu oublies un `.env` avec une clé secrète)

En résumé

Le fichier `.dockerignore` :

- **Réduit la taille** de l'image Docker
- **Accélère le build**
- **Protège les données sensibles**
- Évite de copier du bazar inutile

2- Construire une image Docker

On utilise la commande :

```
docker build -t nom_de_ton_image .
```

Cette commande sert à **construire une image Docker** à partir de ton `Dockerfile`. Voyons cette commande en détail :

Élément	Explication
`docker`	Lance l'outil Docker depuis ton terminal

Élément	Explication
`build`	Indique que tu veux **construire une image** Docker
`-t nom_de_ton_image`	Donne un **nom (tag)** à ton image, ici `nom_de_ton_image`
`.`	Spécifie le **contexte de construction**, ici le dossier courant (`.`) où se trouve le `Dockerfile`

Concrètement, que fait Docker ici ?

1. Il lit ton `Dockerfile`
2. Il suit **chaque instruction** (`FROM`, `COPY`, `RUN`, `CMD`, etc.)
3. Il crée une **image Docker complète**, prête à être lancée
4. Il la nomme `nom_de_ton_image` pour que tu puisses la réutiliser ensuite

Tu peux voir l'image créée avec :

```
docker images
```

Exemple de résultat :

```
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
movieslensapicontainerimage   latest    fb50d410d93f   2 minutes ago
263MB
```

En résumé, cette commande te permet de transformer ton code et tes dépendances en **une image Docker réutilisable**. C'est comme faire une boîte scellée qui contient ton API, prête à être envoyée ou exécutée **n'importe où** !

3- Lancer un conteneur

Une fois l'image créée, tu peux **démarrer l'API dans un conteneur** avec :

```
docker run -d -p 80:80 --name donne_un_nom_au_conteneur nom_de_ton_image
```

Décomposons la commande :

- `docker run` : Lance un conteneur basé sur une image existante.
- `-d` : Le mode **détaché**, qui permet d'exécuter le conteneur en arrière-plan.

- `-p 80:80` : Fait correspondre le port **80 de ton conteneur** au port **80 de ta machine locale**. Cela signifie que l'API sera accessible via `http://localhost:80`.
- `--name donne_un_nom_au_conteneur` : Cette option te permet de donner un nom personnalisé à ton conteneur. Dans cet exemple, le conteneur sera nommé `donne_un_nom_au_conteneur`. Cela peut être utile pour identifier facilement ton conteneur dans une liste ou pour l'arrêter et le gérer plus facilement.
- `nom_de_ton_image` : Le nom de l'image Docker que tu viens de construire.

Vérification :

Tu peux maintenant vérifier que ton conteneur fonctionne en accédant à `http://localhost:80` et refaire des tests des endpoins à travers l'interface Swagger : `http://localhost/docs`

Si tu veux vérifier les conteneurs en cours d'exécution, tu peux aussi utiliser la commande :

```
docker ps
```

Voici un exemple de résultat de cette commande :

```
CONTAINER ID   IMAGE                            COMMAND
CREATED          STATUS          PORTS                  NAMES
5ca9b17d9b05   movieslensapicontainerimage   "uvicorn main:app --…"   6
minutes ago   Up 6 minutes   0.0.0.0:80->80/tcp   movielensapicontainer
```

Cela te permettra de voir le nom de ton conteneur dans la liste.

Quelques commandes utiles pour nettoyer l'environnement Docker

Voici quelques commandes utiles pour nettoyer ton environnement Docker après avoir testé ton application :

1. **Arrêter un conteneur en cours d'exécution** Pour arrêter un conteneur qui fonctionne actuellement, utilise la commande suivante (remplace `movielens-api-container` par le nom de ton conteneur) :

   ```
   docker stop movielens-api-container
   ```

2. **Supprimer un conteneur** Après avoir arrêté ton conteneur, tu peux le supprimer avec la commande suivante :

   ```
   docker rm movielens-api-container
   ```

   Cela supprimera le conteneur spécifique.

3. **Supprimer une image Docker** Pour supprimer une image Docker, utilise cette commande en remplaçant `movieslensapicontainerimage` par le nom de ton image :

   ```
   docker rmi movieslensapicontainerimage
   ```

   Assure-toi d'abord que tous les conteneurs basés sur cette image sont arrêtés et supprimés avant de la supprimer.

4. **Supprimer toutes les images** Pour supprimer toutes les images Docker présentes sur ton système (attention, cela supprimera toutes les images et tu devras les reconstruire plus tard si nécessaire) :

   ```
   docker rmi $(docker images -q)
   ```

   Cette commande supprimera toutes les images présentes sur ton système.

5. **Supprimer tous les conteneurs (arrêtés et en cours d'exécution)** Pour supprimer tous les conteneurs (en cours d'exécution ou arrêtés), utilise la commande suivante :

   ```
   docker rm $(docker ps -a -q)
   ```

   Cela supprimera tous les conteneurs, qu'ils soient actifs ou non.

6. **Supprimer tous les conteneurs et images** Si tu souhaites tout supprimer en une seule commande (tous les conteneurs et toutes les images), tu peux combiner les deux commandes comme suit :

   ```
   docker system prune -a
   ```

   Cela supprimera tous les conteneurs, images et volumes inutilisés. C'est une commande puissante à utiliser avec précaution, car elle supprimera toutes les ressources non utilisées par Docker.

En résumé : Après avoir terminé tes tests ou ton déploiement, il est important de nettoyer ton environnement Docker (au cas où tu ne comptes plus utiliser les images/conteneurs créés) pour libérer de l'espace et éviter l'accumulation de ressources inutiles. Les commandes ci-dessus te permettent de stopper et supprimer les conteneurs et les images Docker.