Rappels : Images, Conteneurs, et leur Interrelation

Définition d'une image Docker

Une **image Docker** est un modèle (ou blueprint) d'un conteneur. Elle contient tout le nécessaire pour exécuter une application, c'est-à-dire :

- Le système d'exploitation de base (souvent une version minimaliste de Linux ou Windows)
- Les fichiers nécessaires à l'application (par exemple, le code source, les binaires, les bibliothèques, etc.)
- Les configurations (par exemple, variables d'environnement, ports exposés)
- Les métadonnées nécessaires pour son exécution

Les images sont **en lecture seule**, c'est-à-dire qu'elles ne peuvent pas être modifiées directement une fois créées. Lorsqu'un conteneur est créé à partir d'une image, Docker utilise l'image pour fournir un environnement d'exécution isolé.

Création d'une image :

- Vous pouvez créer une image via un fichier `Dockerfile` où vous spécifiez les instructions pour configurer l'image.
- Exemple de `Dockerfile` pour une application Node.js :

```
FROM node:14
WORKDIR /app
COPY . .
RUN npm install
CMD ["npm", "start"]
```

Définition d'un conteneur Docker

Un **conteneur Docker** est une instance en cours d'exécution d'une image Docker. Il représente l'environnement d'exécution isolé où l'application peut fonctionner. Chaque conteneur dispose de son propre système de fichiers, processus, et ressources, tout en étant basé sur une image.

- **Isolation** : Les conteneurs sont isolés les uns des autres. Chaque conteneur fonctionne dans son propre espace de nommage (namespace) pour les processus, les réseaux et les systèmes de fichiers.
- **Légèreté** : Un conteneur utilise une image comme référence, mais contrairement aux machines virtuelles, il ne nécessite pas d'OS complet. Cela permet de les rendre très légers et rapides à démarrer.
- **Portabilité** : Un conteneur fonctionne de manière identique quel que soit l'environnement (local, cloud, etc.), car il contient tout ce qui est nécessaire pour faire fonctionner l'application.

Exemple de création et d'exécution d'un conteneur :

```
docker run -d --name my-container my-image
```

Comment une image devient un conteneur : De manière avancée

Lorsqu'une image devient un conteneur, plusieurs étapes se déroulent en arrière-plan pour créer un environnement d'exécution isolé pour l'application.

1. **Lecture de l'image** :
   Lorsque vous exécutez une commande comme `docker run`, Docker prend l'image spécifiée et la lit pour créer un conteneur. L'image elle-même est en lecture seule.

2. **Création d'un système de fichiers en couche** :
   Un conteneur utilise une architecture de **couches** pour son système de fichiers. Docker emploie un **système de fichiers union** où chaque image et chaque modification (par exemple, l'exécution d'une commande `RUN` dans un `Dockerfile`) génère une nouvelle couche.

   - **Lecture seule** : Les couches d'image sont immuables, ce qui permet de garantir que les conteneurs ne modifient pas l'image sous-jacente.
   - **Écriture sur la couche supérieure** : Quand un conteneur s'exécute, une nouvelle couche d'écriture (appelée "couche de conteneur") est ajoutée au-dessus des couches de l'image. Toutes les modifications (fichiers créés, modifiés ou supprimés) dans le conteneur se font dans cette couche.

3. **Initialisation du conteneur** :

   - **PID Namespace** : Un conteneur Docker utilise son propre espace de processus (PID) afin d'empêcher les autres conteneurs de voir ou d'interférer avec ses processus.
   - **Réseau et IPC** : Le conteneur obtient son propre espace réseau (avec une interface réseau virtuelle) et un espace IPC (Inter Process Communication) isolé.
   - **Volume et fichier temporaire** : Docker utilise des volumes pour persister les données en dehors du conteneur. Si un conteneur crée ou modifie des fichiers, ces modifications sont conservées dans une couche d'écriture ou dans des volumes externes.

4. **Exécution des instructions CMD ou ENTRYPOINT** :

   - Docker démarre le processus spécifié dans l'instruction `CMD` ou `ENTRYPOINT` dans le `Dockerfile`. Cela représente le processus principal du conteneur (par exemple, un serveur web, une application Node.js, etc.).
   - Exemple :

     ```
     docker run -d --name my-app my-image npm start
     ```

   - Dans cet exemple, Docker démarre le conteneur et exécute `npm start` pour lancer l'application.

5. **Gestion de l'environnement** :
   Docker peut également injecter des variables d'environnement dans le conteneur au moment de l'exécution (par exemple via l'option `-e` dans `docker run`), permettant à l'application dans le conteneur de s'adapter à des environnements différents sans modification du code.

6. **Network & Port Binding** :

   - **Réseau isolé** : Un conteneur dispose de son propre réseau (soit en mode `bridge`, `host`, ou `overlay`), ce qui lui permet de communiquer avec d'autres conteneurs ou avec l'extérieur (selon les paramètres de configuration).
   - **Port mapping** : Si le conteneur doit exposer des ports vers l'extérieur, Docker lie les ports de l'hôte aux ports du conteneur à l'aide de la commande `-p`.

7. **Nettoyage après arrêt** :

   - Une fois le conteneur arrêté, ses couches sont laissées intactes. Si un conteneur est supprimé avec `docker rm`, toutes ses couches d'écriture et ses volumes sont supprimés (sauf s'ils sont liés à d'autres conteneurs ou volumes).

Illustration avec un Exemple Pratique

1. **Créer une image** :

   - Dockerfile pour une application simple Python :

     ```
     FROM python:3.8-slim
     WORKDIR /app
     COPY . .
     RUN pip install -r requirements.txt
     CMD ["python", "app.py"]
     ```

2. **Construire l'image** :

   ```
   docker build -t python-app .
   ```

3. **Créer un conteneur à partir de l'image** :

   ```
   docker run -d --name python-container python-app
   ```

Dans cet exemple, l'image `python-app` devient un conteneur avec son propre environnement isolé, où le fichier `app.py` sera exécuté à l'intérieur du conteneur.