Conteneurisation et Orchestration

Partie 4 : Dockerfile

Qu'est-ce qu'un Dockerfile ?

Un `Dockerfile` est un fichier texte qui contient une série d'instructions pour construire une image Docker. C'est la **recette** de votre image.

- **Automatisation** : Décrit les étapes pour assembler l'image (OS, dépendances, code...).
- **Reproductibilité** : Garantit que l'image est toujours construite de la même manière.
- **Portabilité** : Le `Dockerfile` peut être partagé pour construire l'image sur n'importe quelle machine.

La Structure d'un Dockerfile

Un `Dockerfile` est une séquence d'instructions. Chaque instruction crée une nouvelle **couche** dans l'image.

Instruction	Rôle
`FROM`	Définit l'**image de base** (ex: `python:3.12-slim`). **Obligatoire**.
`WORKDIR`	Définit le **répertoire de travail** pour les instructions suivantes.
`COPY`	**Copie** les fichiers et dossiers de l'hôte vers l'image.
`RUN`	**Exécute** une commande (ex: `pip install`, `apt-get update`).

Instruction	Rôle
`CMD`	Définit la **commande par défaut** à exécuter au lancement du conteneur.
`ENTRYPOINT`	Configure le conteneur pour qu'il s'exécute comme un exécutable.
`ENV`	Définit des **variables d'environnement**.
`EXPOSE`	**Informe** Docker que le conteneur écoute sur un port spécifique.

Exemple : Dockerfile pour une App Python

```
# 1. Définir l'image de base
FROM python:3.12-slim

# 2. Définir le répertoire de travail
WORKDIR /app

# 3. Copier les dépendances et les installer
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 4. Copier le reste du code de l'application
COPY . .

# 5. Exposer le port de l'application
EXPOSE 5000

# 6. Définir la commande pour lancer l'application
CMD ["python", "app.py"]
```

Bonnes Pratiques (1/3)

- **Utiliser des images de base minimalistes** :
  - `alpine`, `slim`, `distroless`.
  - Réduit la taille de l'image et la surface d'attaque.
- **Minimiser le nombre de couches** :
  - Regrouper les commandes `RUN` avec `&&`.
  - Chaque instruction crée une couche, plus il y en a, plus l'image est lourde.

```
# Mauvais
RUN apt-get update
RUN apt-get install -y curl

# Bon
RUN apt-get update && apt-get install -y curl
```

Bonnes Pratiques (2/3)

- **Utiliser le cache de build efficacement** :
  - Placez les instructions qui changent le moins souvent **en premier**.
  - Copiez d'abord les fichiers de dépendances (`requirements.txt` , `package.json`), installez-les, PUIS copiez le reste du code.

```
# Copier d'abord les dépendances pour profiter du cache
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copier le code source ensuite
COPY . .
```

Bonnes Pratiques (3/3)

- **Utiliser un fichier `.dockerignore` :**
  - Exclut les fichiers et dossiers inutiles du contexte de build (`.git` , `node_modules` , `.env`).
  - Accélère le build et réduit la taille de l'image.
- **Ne pas exécuter en tant que `root` :**
  - Créez un utilisateur non-privilégié et utilisez l'instruction `USER` pour plus de sécurité.

```
RUN addgroup -S appuser && adduser -S appuser -G appuser
USER appuser
```

Optimisation : Les Builds Multi-Stage

Le **multi-stage build** est une technique pour créer des images légères en séparant la phase de **construction** de la phase d'**exécution**.

- **Étape 1 (`builder`)** : Une image avec tous les outils de build (compilateur, SDK...). On y compile l'application.
- **Étape 2 (finale)** : Une image légère (ex: `alpine`) où l'on copie **uniquement l'artefact compilé** de l'étape 1.

Résultat : L'image finale ne contient que le strict nécessaire pour exécuter l'application.

Exemple de Build Multi-Stage (Python avec UV et pyproject.toml)

```dockerfile
# --- Étape 1: Créer l'environnement virtuel ---
FROM python:3.12-slim AS builder

# Installer uv
RUN pip install uv

# Créer un environnement virtuel
WORKDIR /app
RUN uv venv

# Copier le fichier de dépendances et les installer
COPY pyproject.toml ./
RUN uv pip install -p . --system

# --- Étape 2: Construire l'image finale ---
FROM python:3.12-slim

# Copier l'environnement virtuel de l'étape de build
COPY --from=builder /app/.venv /.venv

# Ajouter l'environnement virtuel au PATH
ENV PATH="/.venv/bin:$PATH"

# Copier le code de l'application
WORKDIR /app
COPY . .

# Lancer l'application
CMD ["python", "app.py"]
```

Conclusion sur le Dockerfile

- Le `Dockerfile` est le **plan de construction** de vos applications conteneurisées.
- Une bonne rédaction est essentielle pour obtenir des images **légères, rapides et sécurisées**.
- Maîtriser les **bonnes pratiques** (cache, multi-stage, `.dockerignore`) est une compétence clé pour tout développeur moderne.