Pratique : Images Docker

Créer et optimiser des images Docker

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 EXERCICE 1 — Image Java simple 2
1.1 Structure du projet 2
2 Exercice 2 — `ARG`, `ENV` (adapté au projet `hello-java`) 4
3 EXERCICE 3 — Multi-stage build + ENTRYPOINT/CMD 5

! Corrections

La correction des exercices est disponible dans le dépôt :

: ebpro/sample-containers-exercices-images

Détails

- **URL**: https://github.com/ebpro/sample-containers-exercices-images
- **Branch**: develop
- **Local Path**: `/home/jovyan/work/github/ebpro/sample-containers-exercices-images`
- **Latest Commit**: 58e9ae6 - chore: add _quarto-utils submodule + .devcontainer (SOTA pattern)
- **Timestamp**: 2026-11-01T16:26:06Z

Clone Command:

```
git clone -b develop https://github.com/ebpro/sample-containers-exercices-images
```

1 EXERCICE 1 — Image Java simple

Créer une image Docker contenant une application Java simple construite avec Maven Wrapper.

1.1 Structure du projet

Créez l'arborescence suivante :

```
hello-java/
├── src/
│   └── main/
│       └── java/
│           └── app/
│               └── App1.java
├── pom.xml
```

Etudier le code de l'application Java `App1.java` :

```
```

Etudier le fichier `pom.xml`, regarder :

- les coordonnées du projet (groupId, artifactId, version) : identifier ton application et sa version.
- la section properties est très importante, car elle regroupe les versions de Java et des bibliothèques ; le projet utilise Java 21.
  - la propriété `main.class` définit la classe principale de l'application (avec la méthode `main`) qui sera définit dans le manifeste du JAR exécutable.
- La partie dependencies définit les bibliothèques externes dont le projet a besoin, comme JUnit pour les tests ou les outils de logging pour afficher des messages.
- build/plugins, définissent comment Maven va compiler le code et lancer les tests automatiquement.
- Enfin, la section profiles (notamment prod). Un profil Maven est une partie optionnelle de la configuration activé manuellement (ici avec `-Pprod`). Ce profile sert à créer une version prête à être distribuée, avec un JAR exécutable (dans `target`) et les dépendances copiées dans le dossier (`libe`).

```
```

Générer le Maven Wrapper avec la commande :

```
mvn wrapper:wrapper
```

Tester que l'application fonctionne localement :

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Maven build in CI"
else
  ./mvnw package -Pprod
  java -jar target/hello-java-*.jar app.App1
fi
```

```
⏭ Skipping Maven build in CI
```

! A FAIRE

1. Ecrire un Dockerfile pour construire l'image Docker. Vous utiliserez l'image de base `eclipse-temurin:21-jdk-alpine`. Vous devrez copier les fichiers, construire l'application avec Maven Wrapper, et définir la commande de démarrage.
2. Construire l'image Docker
3. Lancer un conteneur pour exécuter l'application Java.

Solution Dockerfile

Solution Commandes

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker build -t hello-java:0.0.1 .
fi
```

```
⏭ Skipping Docker step in CI
```

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker run --label teaching-demo=true --rm hello-java:0.0.1
fi
```

```
⏭ Skipping Docker step in CI
```

2 Exercice 2 — `ARG`, `ENV` (adapté au projet `hello-java`)

Ajouter au projet la classe `app.App2` qui lit un fichier pour connaitre le commit. Ce fichier sera créé à la construction de l'image avec un `build-arg`. L'application lit aussi la variable d'environnement `LOG_LEVEL` (pour contrôler le niveau de logs au runtime).

```
```

! A FAIRE

Créer un `Dockerfile.arg_env` qui accepte un `build-arg` et crée le fichier `/app/GIT_COMMIT` avec cette valeur.

Depuis le dossier `hello-java/`, construisez l'image en passant la valeur du commit au build :

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker build \
    -f Dockerfile.arg_env \
    --build-arg GIT_COMMIT=abc123 \
```

```
    -t hello-java:env \
    .
fi
```

```
⏭ Skipping Docker step in CI
```

1. Lancez le conteneur pour vérifier que le commit est bien passé au build :

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker run --label teaching-demo=true --rm hello-java:env
fi
```

```
⏭ Skipping Docker step in CI
```

2. Lancez le conteneur en surchargeant avec une variable d'environnement le niveau de logs (runtime) :

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
 docker run --label teaching-demo=true --rm --env LOG_LEVEL=DEBUG hello-java:env
fi
```

```
⏭ Skipping Docker step in CI
```

3. Que peut-on faire si on ajoute dans le Dockerfile une instruction `ENV COMMIT=${GIT_COMMIT}` ?

Solution Dockerfile

```
```

3 EXERCICE 3 — Multi-stage build + ENTRYPOINT/CMD

Compléter l'application Java :

```
```

! A FAIRE

Créer une image optimisée en multi-stage et utiliser `ENTRYPOINT` + `CMD`.

1. Ecrire un Dockerfile `Dockerfile.multistage` qui utilise une construction en deux étapes :
   - une étape de build utilisant l'image `eclipse-temurin:21-jdk` pour compiler l'application Java
   - une étape runtime utilisant l'image `eclipse-temurin:21-jre` pour exécuter cette nouvelle classe de l'application Java
2. Utiliser `ENTRYPOINT` pour définir la commande principale et `CMD` pour passer un argument optionnel (le nom à saluer).

Tester votre solution avec les commandes suivantes :

Construire l'image Docker

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker build \
  --file Dockerfile.multistage \
  -t hello-java:0.0.1-multi \
  .
fi
```

```
⏭ Skipping Docker step in CI
```

4. Lancer le conteneur sans argument (doit afficher « Hello World! »)

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker run --label teaching-demo=true --rm hello-java:0.0.1-multi
fi
```

```
⏭ Skipping Docker step in CI
```

5. Lancer le conteneur avec un argument (doit afficher « Hello Pierre! »)

```
if [ -n "$CI" ]; then
  echo "⏭ Skipping Docker step in CI"
else
  docker run --label teaching-demo=true --rm -e NAME=Pierre hello-java:0.0.1-
multi
fi
```

```
⏭ Skipping Docker step in CI
```

6. comparer la taille de l'image optimisée

```
docker image ls --filter "reference=hello-java*"
```

```
                                                    i Info →   U  In Use
IMAGE   ID             DISK USAGE   CONTENT SIZE   EXTRA
```

! Extensions (Bonus)

1. (Bonus 1) Modifier l'application Java pour ajouter une variable d'environnement `GREETING` personnalisable pour le message de salutation.
2. (Bonus 2) Optimiser le Dockerfile pour le pas télécharger inutilement des dépendances Maven à chaque build quand seule le code source change.
3. (Bonus 3) Utiliser `--mount=type=cache` associé à `RUN` pour cacher le répertoire `.m2` lors du build Maven.
4. (Bonus 4) Utiliser des images de base plus légères (ex: `eclipse-temurin:21-jre-alpine`).

Solution Dockerfile