

Conteneurs — Application à Java

Construire des images Docker Java prêtes pour l'orchestration

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

2026-10-14

Table des matières

1 Objectifs	1
2 Exemples pratiques	1
3 Image Java simple (Maven embarqué)	2
4 Image Java optimisée (multi-stage build) .	2
5 Prêt pour l'orchestration (Production-Ready)	3
6 Affichage des images créées	3
7 Accélérer les builds (Cache BuildKit)	3
8 Construction personnalisée (SDKMAN) — Avancé	3
9 Image minimale avec jlink (JPMS) — Expert	4
10 jlink-alpine — Minimalité et compromis .	4
11 Exécutable natif avec GraalVM — Culture	4
12 Comparaison des stratégies	5
13 Conclusion	5

1 Objectifs

Comment **transformer une application Java en image Docker exploitable en production**.

À l'issue, vous devez comprendre :

- L'impact de l'ordre des instructions sur le **cache de build**.

- Pourquoi séparer l'environnement de build du runtime (**Multi-stage**).
- Comment rendre une image **orchestrable** (sécurité non-root, signaux PID 1).

! Important

Ce chapitre ne vise pas à maîtriser toutes les techniques, mais à comprendre **les trajectoires possibles**.

2 Exemples pratiques

Les exemples suivants sont accessibles dans le dépôt :

☒ : ebpro/sample-containers-java-helloworld

Détails

- **URL:** <https://github.com/ebpro/sample-containers-java-helloworld>
- **Branch:** develop
- **Local Path:** `/home/jovyan/work/github/ebpro/sample-containers-java-helloworld`
- **Latest Commit:** a684804 - chore: add _quarto-utils submodule + .devcontainer (SOTA pattern)
- **Timestamp:** 2026-10-14T14:52:33Z

Clone Command:

```
git clone -b develop https://
github.com/ebpro/sample-containers-
java-helloworld
```

3 Image Java simple (Maven embarqué)

La méthode la plus directe consiste à utiliser une image Maven officielle. C'est l'approche « tout-en-un » où l'on compile et exécute au même endroit.

```
docker image build \
  --tag javahello:mavenimage \
  --file Dockerfile.10.maven .
```

```
docker container run --
label ebpro-render=true --rm
javahello:mavenimage
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:
+UseContainerSupport
-XX:MaxRAMPercentage=75.0
14:53:33.553 INFO
fr.univtln.bruno.demos.docker.App -
Java Vendor: Eclipse Adoptium |
Version: 25.0.4.1
14:53:33.560 INFO
fr.univtln.bruno.demos.docker.App
- Démarrage de l'application
(Iterations: 10000)...
14:53:33.653 INFO
fr.univtln.bruno.demos.docker.App
- Traitement terminé. Éléments
filtrés : 8869 | Temps : 93 ms
Fin du programme. (Éléments traités:
8869)
```

```
ANALYSE DE L'IMAGE :
javahello:mavenimage
• Sécurité : ☐ appuser
• Surface : ☒ Shell présent
• Structure : ☒ Polluants (Maven
ou /src)
• Runtime : ☒ Standard (JRE)
• Java Opts : ☒ Manquante
• Poids : 509MB
```

! Analyse critique

- **Sécurité** : L'image contient le code source, Maven et le JDK complet. C'est une surface d'attaque inutile.
- **Poids** : L'image dépasse souvent les 800MB pour une application simple.

4 Image Java optimisée (multi-stage build)

Le **Multi-stage build** est la stratégie de référence. On utilise une image « lourde » pour compiler, puis on ne transfère que le `.jar` et les dépendances (`lib/*.jar`) vers une image JRE légère pour l'exécution.

```
docker image build \
  --tag javahello:mavenimagestage \
  --file Dockerfile.20.stage .
```

```
ANALYSE DE L'IMAGE :
javahello:mavenimagestage
• Sécurité : ☐ appuser
• Surface : ☒ Shell présent
• Structure : ☐ Artefact pur (Multi-
stage OK)
• Runtime : ☒ Standard (JRE)
```

- Java Opts : ✕ Manquante
- Poids : 304MB

5 Prêt pour l'orchestration (Production-Ready)

Pour être orchestrée (Kubernetes, Compose), une image doit respecter des règles d'hygiène :

1. **Non-Root** : Vos Dockerfiles utilisent un `appuser`. Cela empêche une faille applicative de compromettre l'hôte.
2. **Signaux (Exec Form)** : L'usage de `ENTRYPOINT ["/script.sh"]` permet à la JVM de recevoir le `SIGTERM` pour un arrêt propre.
3. **Mémoire** : L'usage de `-XX:MaxRAMPercentage` permet à la JVM d'être « Container Aware ».

6 Affichage des images créées

```
docker image ls \
--filter "reference=javahello" \
--format "table {{.Repository}}\t{{.Tag}}\t{{.ImageID}}\t{{.CreatedAt}}"
```

REPOSITORY	TAG	IMAGE ID	SIZE	CREATED
javahello		8fb3cd22dd58	319MB	8 seconds ago
javahello		592bfa090487	533MB	About a minute ago

7 Accélérer les builds (Cache BuildKit)

Docker permet de monter des caches persistants pour le répertoire `~/.m2`. Cela évite de retélécharger les dépendances à chaque modification du code source.

```
docker image build \
--tag javahello:dockercache \
--file Dockerfile.30.cache .
```

ANALYSE DE L'IMAGE : javahello:dockercache

- Sécurité : ☐ appuser
- Surface : ☒ Shell présent
- Structure : ☐ Artefact pur (Multi-stage OK)
- Runtime : ☒ Standard (JRE)
- Java Opts : ✕ Manquante
- Poids : 304MB

i Note

L'instruction `--mount=type=cache,target=/root/.m2` est une fonctionnalité BuildKit qui transforme radicalement l'expérience de CI/CD.

8 Construction personnalisée (SDKMAN) — Avancé

L'usage de SDKMAN permet un contrôle granulaire sur les versions exactes du JDK et des outils de build, indépendamment des images Maven officielles.

```
docker image build \
--tag javahello>manual \
--file Dockerfile.40>manual .
```

ANALYSE DE L'IMAGE : javahello>manual

- Sécurité : ☐ appuser
- Surface : ☒ Shell présent
- Structure : ☐ Artefact pur (Multi-

```
stage OK)
  • Runtime      : ☹ Standard (JRE)
  • Java Opts    : ✗ Manquante
  • Poids       : 304MB
```

9 Image minimale avec jlink (JPMS) — Expert

`jlink` permet de créer un runtime Java sur mesure contenant uniquement les modules nécessaires.

⚠ Avertissement

Nécessite une application modularisée (JPMS) et une analyse fine des dépendances.

```
ANALYSE DE L'IMAGE : javahello:jlink
  • Sécurité      : ☐ appuser
  • Surface      : ▲ Shell présent
  • Structure    : ☐ Artefact pur (Multi-stage OK)
  • Runtime      : ☹ Standard (JRE)
  • Java Opts    : ✗ Manquante
  • Poids       : 121MB
```

10 jlink-alpine — Minimalité et compromis

`jlink-alpine` propose un runtime Java construit sur Alpine (musl) contenant uniquement les modules nécessaires. C'est une excellente option pédagogique pour montrer l'impact du runtime sur la taille d'image, mais elle comporte des limites pratiques.

⚠ Avertissement

Alpine utilise `musl` (libc alternative). Les composants natifs (JNI), certaines bibliothèques précompilées ou des outils attendus par `glibc` peuvent ne pas fonctionner. Testez systématiquement avant production.

• Avantages:

- ▶ Image très compacte (gain de plusieurs dizaines à centaines de MB).
- ▶ Idéale pour démonstrations et environnements contrôlés.

• Inconvénients:

- ▶ Compatibilité native réduite (JNI, agents natifs, certains drivers).
- ▶ Parfois plus difficile à déboguer (outils absents dans l'image).

Commandes utiles (exemples étudiants) :

Recommandation : pour un déploiement robuste, préférez `jlink` (Debian/distroless) sauf si vous maîtrisez l'ensemble des dépendances natives.

11 Exécutable natif avec GraalVM — Culture

GraalVM compile Java en binaire natif.

- **Avantage** : Démarrage instantané (< 100ms) et RAM dérisoire.
- **Inconvénient** : Build complexe et perte de certaines capacités dynamiques de la JVM.

```
ANALYSE DE L'IMAGE :
javahello:graalvm
  • Sécurité      : ☐ nonroot:nonroot
```

- Surface : ☐ Minimaliste / Distroless
- Structure : ☐ Artefact pur (Multi-stage OK)
- Runtime : ☐ Natif (GraalVM app)
- Java Opts : ☐ N/A (Binaire natif)
- Poids : 20MB

12 Comparaison des stratégies

Image Tag	Build Cold	Size	Peak RAM	Surface d'Attaque
ma- veni- mage	28s	928MB	86.01 MiB	Maxi- male
mave- nima- ges- tage	24s	406MB	71.97 MiB	Moyenne
jlink- alpine	30s	163MB	55.83 MiB	Faible
graalvm	57s	46MB	5.02 MiB	Mini- male

13 Conclusion

! Points clés pour la mise en production

- **Multi-stage build** : Séparation build / runtime.
- **Utilisateur non-root** : Sécurité de l'hôte.
- **Gestion des ressources** : Paramétrage JVM pour les limites de conteneur.
- **Logs** : Toujours sur `stdout/stderr` (laissé à la charge de l'orchestrateur).

Prochaine étape : **orchestrer ces images avec Docker Compose.**