

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

2026-09-30

Table des matières

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

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/notebook-containers-intro-sample-java-helloworld

Détails

- **URL:** <https://github.com/ebpro/notebook-containers-intro-sample-java-helloworld>
- **Branch:** develop
- **Local Path:** `/home/jovyan/work/examples/github/ebpro/notebook-containers-intro-sample-java-helloworld`

- **Latest Commit:** a21605e - fix: native-image heap cap must use JAVA_TOOL_OPTIONS (real JVM var)
- **Timestamp:** 2026-09-30T07:03:15Z

Clone Command:

```
git clone -b develop https://github.com/ebpro/notebook-containers-intro-sample-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.

```
1 # =====
2 # Stage: Build & Run (Single-stage Maven)
3 # Purpose: Baseline development environment for Java 25
4 # =====
5 FROM maven:3.9.16-eclipse-temurin-25-noble
6
7 # -----
8 # Build Environment Configuration
9 # -----
10 WORKDIR /app
11
12 ENV HOME=/home/appuser
13 ENV MAVEN_CONFIG=/home/appuser/.m2
14 ENV MAVEN_OPTS="-Dmaven.repo.local=/home/appuser/.m2"
15
16 LABEL maintainer="Emmanuel Bruno <emmanuel.bruno@univ-tln.fr>"
17 LABEL description="Java Hello
```

```
World Application - Full Maven baseline"
18
19 # -----
20 # Security: Unprivileged user creation
21 # -----
22 RUN groupadd -r appgroup \
23 && useradd -r -g appgroup -m -d /home/appuser appuser \
24 && mkdir -p /home/appuser/.m2 \
25 && chown -R appuser:appgroup /home/appuser \
26 && chown -R appuser:appgroup /app
27
28 # -----
29 # Layer 1: Dependency Caching
30 # -----
31 # Copying only the wrapper and POM allows Docker to cache dependencies independently of source code changes.
32
33 COPY --chown=appuser:appgroup .mvn/ .mvn
34 COPY --chown=appuser:appgroup mvnw pom.xml ./
35
36 USER appuser
37
38 # Pre-resolve project dependencies to optimize build time
39 RUN ./mvnw dependency:resolve
40
41 # -----
42 # Layer 2: Source Compilation
43 # -----
44 COPY --chown=appuser:appgroup src ./src
45
46 # Build artifact using the 'jvm' profile (configured for thin JAR)
```

```

47 RUN ./mvnw clean package -
DskipTests -Pjvm
48
49 # -----
50 # Execution
51 # -----
52
53 # Configure JVM container
ergonomics for resource management
54 ENV JAVA_TOOL_OPTIONS="-XX:
+UseContainerSupport
-XX:MaxRAMPercentage=75.0"
55
56 # Explicitly invoke the JAR. The
'jvm' profile's Manifest
57 # automatically links to
external dependencies in the /libs
folder.
58 ENTRYPOINT ["java", "-jar",
"/app/target/hello-world-0.1.0-
SNAPSHOT.jar"]

```

```

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

```

```

docker container run --rm
javahello:mavenimage

```

```

Picked up JAVA_TOOL_OPTIONS: -XX:
+UseContainerSupport
-XX:MaxRAMPercentage=75.0
07:04:08.564 INFO
fr.univtln.bruno.demos.docker.App -
Java Vendor: Eclipse Adoptium |
Version: 25.0.4.1
07:04:08.572 INFO
fr.univtln.bruno.demos.docker.App
- Démarrage de l'application
(Iterations: 10000)...

```

```

07:04:08.655 INFO
fr.univtln.bruno.demos.docker.App
- Traitement terminé. Éléments
filtrés : 8926 | Temps : 82 ms
Fin du programme. (Éléments traités:
8926)

```

```

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.

```

1 # =====
2 # Stage 1: Build environment
3 # =====
4 # Uses the full Maven SDK to
compile the source code into a thin
JAR
5 FROM maven:3.9.16-eclipse-

```

```

temurin-25-noble AS stage-build
6   WORKDIR /app
7
8   # Cache dependency layer:
   Copying only the wrapper and POM
9   # allows Docker to skip
   downloading dependencies if they
   haven't changed.
10  COPY .mvn/ .mvn
11  COPY mvnw pom.xml ./
12  RUN ./mvnw dependency:go-offline
13
14  # Copy source code and perform a
   clean build
15  COPY src ./src
16  # Build using the 'jvm' profile
   (configured to output a thin JAR)
17  RUN ./mvnw clean package -
   DskipTests -Pjvm
18
19  # =====
20  # Stage 2: Runtime environment
21  # =====
22  # Uses the JRE-only base image to
   minimize the security attack surface
23  FROM eclipse-temurin:25-jre-
   noble
24
25  LABEL maintainer="Emmanuel Bruno
   <emmanuel.bruno@univ-tln.fr>"
26  LABEL description="Java Hello
   World Application - multi-stage"
27
28  # Security: Create an
   unprivileged user/group to run the
   application
29  RUN groupadd -r appgroup &&
   useradd -r -g appgroup appuser && \
30      mkdir -p /app && chown -R
   appuser:appgroup /app
31
32  # Copy artifacts from the builder

```

```

stage:
33  # Only the thin JAR and external
   dependencies are required at runtime.
34  COPY --from=stage-build --
   chown=appuser:appgroup /app/target/
   libs /app/libs
35  COPY --from=stage-build --
   chown=appuser:appgroup /app/target/
   hello-world-*.jar /app/app.jar
36
37  USER appuser
38
39  # Configure JVM ergonomics for
   containerized environments:
40  # JAVA_TOOL_OPTIONS is natively
   parsed by the JVM at startup,
41  # ensuring memory limits are
   respected without needing shell
   script injection.
42  ENV JAVA_TOOL_OPTIONS="-XX:
   +UseContainerSupport
   -XX:MaxRAMPercentage=75.0"
43
44  # Execute the application
45  ENTRYPOINT ["java", "-jar", "/
   app/app.jar"]

```

```

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     : 298MB

```

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}}
```

REPOSITORY	TAG	IMAGE
ID	SIZE	CREATED
javahello		mavenimagestage
c0415d4244c7	312MB	8 seconds ago
javahello		mavenimage
f21f79c68234	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.

```
1 # =====
2 # Stage 1: Build Environment
3 # Purpose: Leverage BuildKit
  caching to accelerate Maven builds
```

```
4 # =====
5 FROM maven:3.9.16-eclipse-
  temurin-25-noble AS stage-build
6
7 WORKDIR /app
8
9 # -----
10 # Layer 1: Maven Wrapper and
  Project Metadata
11 # -----
12 # Copying the wrapper and POM
  first allows Docker to cache the
13 # dependency resolution layer,
  skipping downloads if the project
14 # structure hasn't changed.
15 COPY .mvn/ .mvn/
16 COPY mvnw pom.xml ./
17 RUN chmod +x mvnw
18
19 # -----}
20 # Layer 2: Dependency Caching
  (BuildKit)
21 # -----
22 # Using BuildKit's cache mount
  persists the local M2 repository
23 # across builds, significantly
  reducing network overhead.
24 RUN --mount=type=cache,target=/
  root/.m2 \
25     ./mvnw --batch-mode
  dependency:go-offline
26
27 # -----
28 # Layer 3: Application Build
29 # -----
30 COPY src ./src
31
32 # Compile the project using the
  'jvm' profile.
33 # The persistent cache ensures
  that even if sources change,
34 # dependencies do not need to be
```

```

re-downloaded.
35  RUN --mount=type=cache,target=/
    root/.m2 \
36      ./mvnw --batch-mode clean
package -DskipTests -Pjvm
37
38
39  # =====
40  # Stage 2: Runtime Environment
41  # =====
42  # Using a JRE-only base image
    minimizes the attack surface
43  # and keeps the final image size
    optimized for production.
44  FROM eclipse-temurin:25-jre-
    noble
45
46  LABEL maintainer="Emmanuel Bruno
    <emmanuel.bruno@univ-tln.fr>"
47  LABEL description="Java Hello
    World Application - BuildKit cache
    optimized"
48
49  # Security: Create an
    unprivileged user/group to adhere to
    the principle of least privilege
50  RUN groupadd -r appgroup &&
    useradd -r -g appgroup appuser && \
51      mkdir -p /app && chown -R
    appuser:appgroup /app
52
53  # -----
54  # Deployment: Copy artifacts
    from builder stage
55  # -----
56  # We only copy the thin JAR
    and external dependencies required at
    runtime.
57  COPY --from=stage-build --
    chown=appuser:appgroup /app/target/
    libs /app/libs
58  COPY --from=stage-build --

```

```

chown=appuser:appgroup /app/target/
hello-world-*.jar /app/app.jar
59
60  USER appuser
61
62  # -----
63  # Runtime Execution
64  # -----
65  # Configure JVM container
    ergonomics. JAVA_TOOL_OPTIONS is
    parsed
66  # natively by the JVM, avoiding
    the need for shell script wrappers.
67  ENV JAVA_TOOL_OPTIONS="-XX:
    +UseContainerSupport
    -XX:MaxRAMPercentage=75.0"
68
69  # Execute the application. The
    JAR manifest is configured to
70  # resolve external dependencies
    from the /app/libs/ directory.
71  ENTRYPOINT ["java", "-jar", "/
    app/app.jar"]

```

```

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     : 298MB

```

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.

```
1  # =====
2  # Stage 1: Build Environment
3  # Purpose: Demonstrate manual
4  #           environment provisioning and
5  #           fixed-UID/GID file
6  #           ownership for high-security CI.
7  # =====
8  FROM ubuntu:noble AS stage-build
9
10 # Toolchain versioning for
11 # reproducible builds
12 ARG JAVA_VERSION="25.0.3-tem"
13 ARG MAVEN_VERSION="3.9.16"
14 ARG BUILDER_UID=2000
15 ARG BUILDER_GID=2000
16
17 LABEL maintainer="Emmanuel Bruno
18 <emmanuel.bruno@univ-tln.fr>"
19 LABEL description="Java Hello
20 World Application - SDKMAN build"
```

```
21 apt-get install --yes --quiet
22 --no-install-recommends \
23 ca-certificates curl unzip
24 zip bash && \
25 rm -rf /var/lib/apt/lists/*
26
27 # -----
28 # Security: Create dedicated
29 # builder user with deterministic UID/
30 # GID
31 # -----
32 RUN groupadd -g ${BUILDER_GID}
33 builder \
34 && useradd -m -u
35 ${BUILDER_UID} -g builder -s /bin/
36 bash builder
37
38 USER builder
39 ENV HOME=/home/builder
40 ENV SDKMAN_DIR="$HOME/.sdkman"
41 ENV PATH="$SDKMAN_DIR/bin:
42 $SDKMAN_DIR/candidates/java/current/
43 bin:$SDKMAN_DIR/candidates/maven/
44 current/bin:$PATH"
45
46 SHELL ["/bin/bash", "-c"]
47
48 # -----
49 # Provisioning: Install Java/
50 # Maven via SDKMAN
51 # -----
52 RUN curl -s "https://get.sdkman.
53 io" | bash && \
54 source "$SDKMAN_DIR/bin/
55 sdkman-init.sh" && \
56 sdk install java
57 "$JAVA_VERSION" && \
58 sdk install maven
59 "$MAVEN_VERSION" && \
60 rm -rf "$SDKMAN_DIR/archives/
61 *" "$SDKMAN_DIR/tmp/*"
```

```

47 WORKDIR /app
48
49 # -----
50 # Layer: Dependency Resolution
  (BuildKit Cached)
51 # -----
52 COPY --chown=builder:builder
  mvnw ./
53 COPY
  --chown=builder:builder .mvn .mvn
54 COPY --chown=builder:builder
  pom.xml ./
55
56 RUN --mount=type=cache,target=/
  home/builder/.m2,uid=${BUILDER_UID},gid=$
  \
57     source "$SDKMAN_DIR/bin/
  sdkman-init.sh" && \
58     ./mvnw --batch-mode
  dependency:resolve
59
60 # -----
61 # Layer: Build
62 # -----
63 COPY --chown=builder:builder
  src ./src
64
65 RUN --mount=type=cache,target=/
  home/builder/.m2,uid=${BUILDER_UID},gid=$
  \
66     source "$SDKMAN_DIR/bin/
  sdkman-init.sh" && \
67     ./mvnw --batch-mode -Pjvm -
  DskipTests clean verify
68
69 # =====
70 # Stage 2: Runtime Environment
71 # =====
72 # Distribute using a slim JRE-
  only image. Note: We do not require
73 # the UID to match the builder;
  we only need a non-root user.
74 FROM eclipse-temurin:25-jre-
  noble AS stage-runtime
75
76 LABEL maintainer="Emmanuel Bruno
  <emmanuel.bruno@univ-tln.fr>"
77 LABEL description="Java Hello
  World Application - multi-stage
  runtime"
78
79 RUN groupadd -r appgroup &&
  useradd -r -g appgroup appuser && \
80     mkdir -p /app && chown -R
  appuser:appgroup /app
81
82 # Deploy minimal runtime
  artifacts from the build stage
83 COPY --from=stage-build --
  chown=appuser:appgroup /app/target/
  hello-world-*-SNAPSHOT.jar /app/
  app.jar
84 COPY --from=stage-build --
  chown=appuser:appgroup /app/target/
  libs /app/libs
85
86 USER appuser
87 ENV HOME=/home/appuser
88
89 # JVM Container Ergonomics:
  JAVA_TOOL_OPTIONS is natively parsed
  by the
90 # JVM, enabling container-aware
  resource management.
91 ENV JAVA_TOOL_OPTIONS="-XX:
  +UseContainerSupport
  -XX:MaxRAMPercentage=75.0"
92
93 ENTRYPOINT ["java", "-jar", "/
  app/app.jar"]

```

```
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 : 298MB

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.

```
1  # =====
2  # Stage 1: Build Environment
3  # Purpose: Leverage Maven to
4  # compile code and "slice" a
5  # custom JRE using the
6  # jlink toolchain.
7  # =====
8  FROM maven:3.9.16-eclipse-
9  temurin-25-noble AS builder
10
11 WORKDIR /workspace
12
13 # -----
14 # Layer 1: Build Infrastructure
15 # -----
```

```
13 COPY .mvn/ .mvn/
14 COPY mvnw pom.xml ./
15 RUN chmod +x mvnw
16
17 # -----
18 # Layer 2: Dependency Caching
19 # -----
20 # Warm the Maven cache to speed
21 # up subsequent builds
22 RUN --mount=type=cache,target=/
23 root/.m2 \
24     ./mvnw -q dependency:go-
25 offline
26
27 # -----
28 # Layer 3: Application Build &
29 # JRE Slicing
30 # -----
31 COPY src ./src
32
33 # The 'jlink' profile triggers
34 # the maven-jlink-plugin, which
35 # inspects the module-info and
36 # builds a minimal, custom runtime
37 # image that contains only the
38 # required JDK modules.
39 RUN --mount=type=cache,target=/
40 root/.m2 \
41     ./mvnw clean verify -Pjlink
42 -DskipTests
43
44 # =====
45 # Stage 2: Runtime Environment
46 # (Ubuntu Noble)
47 # Purpose: Deploy the custom,
48 # modularized Java runtime.
49 # =====
50 FROM ubuntu:noble
51
52 LABEL maintainer="Emmanuel Bruno
53 <emmanuel.bruno@univ-tln.fr>"
```

```

43 LABEL description="Java Hello
World Application with Custom JRE via
jlink"
44
45 # Security: Create a non-
privileged user to run the
application
46 RUN groupadd -r appgroup &&
useradd -r -g appgroup appuser && \
47     mkdir -p /jre && chown -R
appuser:appgroup /jre
48
49 # -----
50 # Deployment: Copy the custom
runtime
51 # -----
52 # The custom JRE includes the
application logic, dependencies,
53 # and the native launcher ('app')
generated during the build.
54 COPY --from=builder --
chown=appuser:appgroup /workspace/
target/maven-jlink/classifiers/
runtime-image/ /jre/
55
56 USER appuser
57
58 # JVM Container Ergonomics:
JAVA_TOOL_OPTIONS is natively parsed
by
59 # the JRE, ensuring resource
limits are applied to the custom
runtime.
60 ENV JAVA_TOOL_OPTIONS="-XX:
+UseContainerSupport
-XX:MaxRAMPercentage=75.0"
61
62 # Execution: Invoke the native
binary created by jlink.
63 # No 'java -jar' command is
needed, as the application is now a
64 # self-contained modular

```

executable.

```
65 ENTRYPOINT ["/jre/bin/app"]
```

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.

```
1  # syntax=docker/dockerfile:1
2
3  # Layer 1 – GraalVM Native Image
  toolchain
4  # =====
5  # GraalVM + native-image (the
  "native" toolchain, installed on top
  of the JDK)
6  ARG GRAALVM_VERSION=25
7  FROM ghcr.io/graalvm/native-
  image-community:${GRAALVM_VERSION}
  AS builder
8  WORKDIR /workspace
9
10 # Maven wrapper + POM first
  (re-uses the /root/.m2 cache for
  dependency download)
11 COPY .mvn/ .mvn/
12 COPY mvnw pom.xml ./
13 RUN chmod +x mvnw
14 RUN --mount=type=cache,target=/
  root/.m2 \
15     ./mvnw -q -Pnative
  dependency:go-offline
16
```

```
17  # Cap the native-image build heap
  so the CI render stays within the
  runner
18  # memory budget (large pool: 24
  Gi cgroup limit). JAVA_TOOL_OPTIONS
  is read by
19  # every JVM in the build (maven +
  native-image analysis); -Xmx8G leaves
  headroom
20  # for the native compiler + GC +
  runtime within the limit.
21  ENV JAVA_TOOL_OPTIONS=-Xmx8G
22  COPY src ./src
23  RUN --mount=type=cache,target=/
  root/.m2 \
24     ./mvnw -Pnative -DskipTests
  package
25
26  # Layer 2 – minimal runtime
27  FROM gcr.io/distroless/base-
  debian12:nonroot AS runtime
28  WORKDIR /workspace
29
30  # Copy the compiled native
  executable from the builder stage
31  COPY --from=builder /workspace/
  target/*-runner /app
32
33  # The native executable is self-
  contained (no JVM needed at runtime).
34  # Run as the nonroot user (uid
  65532) provided by the distroless
  base image.
35  USER nonroot:nonroot
36  ENTRYPOINT ["/app"]
37  EXPOSE 8080
38
39  # -----
40  # Layer 3 – optional: multi-stage
  variant (for the "advanced" lesson)
41  # -----
42  # In a real project you'd split
```

```

build + runtime more aggressively.
Here we show
43 # the same idea with a single
GaalVM image used for both stages,
to keep the
44 # lesson focused on the toolchain
rather than image optimisation.
45 FROM ghcr.io/graalvm/native-
image-community:${GRAALVM_VERSION}
AS build-and-run
46 WORKDIR /workspace
47 COPY .mvn/ .mvn/
48 COPY mvnw pom.xml ./
49 RUN chmod +x mvnw
50 COPY src ./src
51 RUN --mount=type=cache,target=/
root/.m2 \
52 ./mvnw -Pnative -DskipTests
package
53 CMD ["./target/*-runner"]

```

```

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	Sur-face d'Attaque
ma- veni- mage	28s	928MB	86.01 MiB	Maxi- male

Image Tag	Build Cold	Size	Peak RAM	Sur-face d'Attaque
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.**