Production Docker for Java Applications

Faster rebuilds with layer caching, a non-root user, a Dockerfile HEALTHCHECK, waiting for real dependency readiness with Compose's depends_on: condition: service_healthy, and production configuration/secrets.

Advanced 40 min

Last updated: August 31, 2026
Read online: https://www.learnforgex.com/en/topics/production-docker-for-java-applications

---

Everything so far in this category has produced a Dockerfile and a `docker-compose.yml` that genuinely work — but "working on a developer's own machine" and "ready to actually deploy" aren't the same bar. "Docker Compose" left one gap open on purpose: `depends_on` waiting for a container to *start*, not for the service inside it to actually be *ready*. This lesson closes that gap, and adds the handful of things that specifically separate a development-grade Docker setup from a production-grade one: a smaller image, faster rebuilds, a non-root user, real health checks, and where configuration and secrets are meant to actually live.

Smaller Images, Revisited

"Choosing a Java Base Image" already chose `eclipse-temurin:21-jre` over a full JDK image for exactly this reason — every unnecessary byte in a base image is size an attacker (or just a slow deploy pipeline) doesn't need to deal with. `eclipse-temurin` also publishes Alpine-based variants (`21-jre-alpine`), smaller still than the Ubuntu-based default tag used so far in this course — worth knowing exists, though this lesson sticks with the Ubuntu-based tag specifically because it makes `apt-get` , used below for `curl` , available without adding an extra, separate package manager to reason about.

Ordering Instructions for Faster Rebuilds (Layer Caching)

"Multi-Stage Builds" already introduced the `builder` stage that runs `mvn package` inside Docker itself — but the order instructions appear in *within* that stage matters for a reason not yet covered: Docker caches each layer, and reuses a cached layer instead of re-running it, as long as nothing that layer depends on has changed.

```
COPY pom.xml .
RUN mvn -B dependency:go-offline

COPY src ./src
RUN mvn -B package -DskipTests
```

`mvn dependency:go-offline` downloads every dependency `pom.xml` declares, without touching `src` at all — copying `pom.xml` *before* `src`, and running that download in its own `RUN` step, means Docker only re-downloads dependencies when `pom.xml` itself actually changes. Editing a single Java file and rebuilding no longer re-downloads this project's entire dependency tree — only the `COPY src ./src` layer and everything after it re-runs, since Docker's cache invalidates a layer, and every layer after it, the moment anything that layer depends on changes.

Running as a Non-Root User

By default, a process inside a container runs as **root** unless a Dockerfile says otherwise — the exact same root as the container's own filesystem, which is more privilege than a Spring Boot application actually needs to do its job.

```
RUN groupadd -r appuser && useradd -r -g appuser appuser

WORKDIR /app

COPY --from=builder --chown=appuser:appuser /build/target/learning-platform-0.1.0-SNAPSHOT.
jar app.jar

USER appuser
```

`groupadd` / `useradd` create an ordinary, unprivileged Linux user inside the image (available here because the Ubuntu-based `eclipse-temurin:21-jre` tag has them, per "Smaller Images, Revisited"); `COPY --chown` gives that new user ownership of the JAR it needs to read; and `USER appuser` switches every instruction after it — including the final `ENTRYPOINT`'s `java` process — to run as that user instead of root. This is defense in depth: it doesn't prevent every possible vulnerability, but it means a compromise of the running Java process doesn't automatically hand over root inside the container.

Health Checks: `HEALTHCHECK` in a Dockerfile

`HEALTHCHECK` tells Docker how to actually ask a running container "are you working?" — periodically running a command inside the container and tracking whether it succeeds.

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=10s \
  CMD curl -f http://localhost:8080/en || exit 1
```

> The standard, purpose-built way to answer "is this Spring Boot application healthy?" is Spring Boot Actuator's `/actuator/health` endpoint — but this project doesn't currently depend on `spring-boot-starter-actuator`. The `HEALTHCHECK` above curls this project's own real homepage route (`/en`, the same one used throughout this course to verify a running container) as a pragmatic stand-in: a successful response is at least proof the application started and is serving requests, even without a dedicated health endpoint.

`docker ps` shows a container's health status (`healthy` / `unhealthy` / `starting`) once a `HEALTHCHECK` is configured — visible feedback that simply "the process hasn't crashed" doesn't provide on its own.

Making `depends_on` Actually Wait: Compose Health Conditions

"`depends_on` — Ordering Startup" warned that plain `depends_on` only waits for a container to *start*. Compose's long-form `depends_on` closes exactly that gap, by waiting on a service's actual `healthcheck` result instead of just its process starting:

```
db:
  image: postgres:16
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 5s
    timeout: 3s
    retries: 5

app:
  build: .
  depends_on:
    db:
      condition: service_healthy
```

`pg_isready` is a real utility already installed inside the official `postgres` image, built specifically to answer "is PostgreSQL actually ready to accept connections?" — not a script written for this lesson. `condition: service_healthy` tells Compose not to start `app` until `db`'s `healthcheck` reports success, closing the exact gap "Docker Compose" left open on purpose.

Environment-Based Configuration for Production

This project's own `application-prod.yml` already reads every genuinely sensitive value — `DB_URL`, `DB_USERNAME`, `DB_PASSWORD` — from environment variables (`${DB_URL}` and the rest) rather than hardcoding them, with its own comment stating the reasoning directly: *"Üretimde tüm gizli bilgiler ortam değişkenlerinden okunur, repoya asla yazılmaz"* ("in production, every secret is read from environment variables, never written into the repo"). A container built from this project's Dockerfile is activated with that same profile via the ordinary Spring Boot mechanism:

```
docker run -e SPRING_PROFILES_ACTIVE=prod -e DB_URL=... -e DB_USERNAME=... -e
DB_PASSWORD=... learning-platform:1.0
```

Nothing here is Docker-specific — `-e` is just how a container receives the environment variables `application-prod.yml` already expected to exist, the same relaxed-binding mechanism "Dockerizing a Spring Boot Application" already used for `SPRING_DATASOURCE_URL`.

Basic Image Security Considerations

A handful of habits, several already established earlier in this course, add up to meaningfully safer images:

- **Pin exact tags** (`postgres:16`, `eclipse-temurin:21-jre`, never `:latest`) — "Best Practices" in "Docker CLI Fundamentals" already covered why; it applies just as much to production.
- **Never bake a real secret into an image** with `ENV` or a hardcoded value in a Dockerfile — anything set that way becomes part of the image's own layers, readable by anyone who can pull or inspect it, permanently, even after that value is later "changed." Secrets belong in environment variables supplied at `docker run` / `docker compose up` time (see "Environment-Based Configuration for Production"), never in the image itself.
- **Keep the base image, and what's installed on top of it, minimal** — every package `RUN apt-get install`s (`curl` above, for the `HEALTHCHECK`) is something that has to be justified, not added by habit.
- **Run as a non-root user** (see above) — a small, real reduction in what a compromised process can do inside the container.

Putting It All Together

The complete production-oriented Dockerfile and Compose file for this project — layer-cached dependency downloads, a non-root user, a `HEALTHCHECK`, and a Compose `healthcheck` condition that makes `depends_on` actually wait:

```
# Stage 1: build the JAR, with dependency downloads cached in their own layer.
FROM maven:3.9-eclipse-temurin-21 AS builder

WORKDIR /build

COPY pom.xml .
RUN mvn -B dependency:go-offline

COPY src ./src
RUN mvn -B package -DskipTests

# Stage 2: a small, non-root, health-checked runtime image.
FROM eclipse-temurin:21-jre

RUN apt-get update \
    && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*

RUN groupadd -r appuser && useradd -r -g appuser appuser

WORKDIR /app

COPY --from=builder --chown=appuser:appuser /build/target/learning-platform-0.1.0-SNAPSHOT.
jar app.jar

USER appuser

EXPOSE 8080

HEALTHCHECK --interval=30s --timeout=3s --start-period=10s \
  CMD curl -f http://localhost:8080/en || exit 1

ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75.0", "-jar", "app.jar"]
```

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 3s
      retries: 5

  app:
    build: .
    depends_on:
      db:
        condition: service_healthy
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/postgres
      SPRING_DATASOURCE_PASSWORD: secret
    ports:
      - "8080:8080"

volumes:
  db-data:
```

Nothing here replaces the mechanisms this course already covered — it's the same multi-stage build from "Dockerizing a Spring Boot Application," the same `docker-compose.yml` shape from "Docker Compose," with the handful of production-specific additions this lesson introduced layered on top, each addressing a real gap a prior lesson left open on purpose.

Common Mistakes

- Leaving a container running as root because it "just works" — it does, but it's more privilege than a Spring Boot process needs, and the entire point of "Running as a Non-Root User" is that it costs almost nothing to avoid.
- Writing a real password or API key into a Dockerfile with `ENV`, assuming it can be "removed later" — it can't; it's baked into that image's layers permanently the moment it's built (see "Basic Image Security Considerations").
- Adding a `HEALTHCHECK` to a Dockerfile but never using it in Compose's `depends_on: condition: service_healthy` — the check runs, but nothing actually waits on it, which quietly defeats the entire point.
- Reordering a multi-stage Dockerfile so `COPY src ./src` happens before the dependency-download step — this silently undoes the layer-caching benefit "Ordering Instructions for Faster Rebuilds (Layer Caching)" described, without producing any visible error.

Best Practices

- Order Dockerfile instructions from least- to most-frequently-changing — dependency manifests and downloads first, application source last — so ordinary code changes invalidate the smallest possible number of cached layers.
- Add a `USER` instruction to every production-facing Dockerfile, right after the files that user needs to read are copied in with correct ownership.
- Pair every Compose `healthcheck` with a `depends_on: condition: service_healthy` on whatever actually depends on it — a health check nothing waits on only provides visibility, not correctness.
- Treat any value that would be embarrassing or dangerous to leak (a password, an API key, a private URL) as something that belongs in an environment variable supplied at runtime, never in a Dockerfile itself.

Summary, Cheat Sheet, and Glossary

Summary

- Copying dependency manifests and downloading dependencies before copying application source lets Docker's layer cache skip re-downloading dependencies on ordinary code changes.
- A `USER` instruction, paired with `COPY --chown`, runs the containerized application as an unprivileged user instead of root by default.
- A Dockerfile `HEALTHCHECK` lets Docker (and `docker ps`) actually track whether a running container is working, not just whether its process is still alive.
- Compose's long-form `depends_on: condition: service_healthy`, paired with a service's own `healthcheck`, is what actually closes the "container started but isn't ready yet" gap `depends_on` alone leaves open.
- Real secrets belong in environment variables supplied at `docker run` / `docker compose up` time — an image's own layers are effectively permanent and readable by anyone who can pull it.

Cheat Sheet

```
COPY pom.xml .
RUN mvn -B dependency:go-offline    # cached separately from source changes
COPY src ./src
RUN mvn -B package -DskipTests

RUN groupadd -r appuser && useradd -r -g appuser appuser
COPY --from=builder --chown=appuser:appuser /build/target/*.jar app.jar
USER appuser

HEALTHCHECK --interval=30s --timeout=3s --start-period=10s \
  CMD curl -f http://localhost:8080/en || exit 1
```

```
depends_on:
  db:
    condition: service_healthy   # waits for db's healthcheck, not just its start
```

Glossary

- **Layer caching**: Docker's reuse of a previously built layer, instead of re-running its instruction, as long as nothing that layer depends on has changed.
- **`USER`**: a Dockerfile instruction that switches every following instruction, and the final running process, to a specific (ideally non-root) user.
- **`HEALTHCHECK`**: a Dockerfile instruction defining a command Docker periodically runs inside a container to determine whether it's actually working, not just running.
- **`condition: service_healthy`**: a Compose `depends_on` setting that waits for a dependency's `healthcheck` to succeed, instead of just waiting for its container to start.