Dockerizing a Spring Boot Application

Choosing a JRE vs JDK base image for a Spring Boot JAR, .dockerignore, multi-stage builds, and container-aware JVM memory sizing -- via this project's own real learning-platform application.

Intermediate 35 min

Last updated: August 31, 2026
Read online: https://www.learnforgex.com/en/topics/dockerizing-a-spring-boot-application

---

"Docker Images and Dockerfiles" built and ran a Dockerfile on purpose before Java entered the picture — `FROM`, `WORKDIR`, `COPY`, `RUN`, `EXPOSE`, `CMD` / `ENTRYPOINT`, and `docker build` are all already familiar. Nothing about those instructions changes here — this lesson applies the exact same set to this project's own `learning-platform` Spring Boot application (`com.cdurgun:learning-platform:0.1.0-SNAPSHOT`, per its own `pom.xml`), and adds three things that specifically matter for a real Java application: choosing the right base image, `.dockerignore`, and multi-stage builds.

From a JAR to a Container Image

`mvn package` already produces a single, self-contained, runnable artifact — `target/learning-platform-0.1.0-SNAPSHOT.jar` — before Docker enters the picture at all. That JAR already bundles every dependency this project needs (Spring Boot's own repackaging, via `spring-boot-starter-parent`, is what makes it self-contained in the first place); nothing about that changes here. What a Dockerfile adds is a way to package that already-built JAR *together with a matching Java runtime* into one image, so running it no longer depends on whatever JDK happens to already be installed on a given machine.

Choosing a Java Base Image

A Java base image comes in two shapes worth distinguishing: a full **JDK** image, which includes the compiler and everything needed to *build* Java code, and a **JRE**-only image, which includes just enough to *run* an already-compiled JAR. Since `target/learning-platform-0.1.0-SNAPSHOT.jar` is already fully built before the Dockerfile even runs, the final image only ever needs to run it — a JRE image is enough, and deliberately smaller than a JDK one for exactly that reason.

```
FROM eclipse-temurin:21-jre
```

`eclipse-temurin` is the Eclipse Foundation's own distribution of OpenJDK, published as official, versioned Docker images — `21-jre` matching this project's own `<java.version>21</java.version>` from `pom.xml` (see "Best Practices" in "Docker Images and Dockerfiles" on preferring a small, purpose-built base image — the same reasoning applies here, JRE over JDK, once the JAR is already built).

A First Dockerfile for a Spring Boot JAR

Putting the base image together with the instructions "Docker Images and Dockerfiles" already covered produces a complete, working Dockerfile:

```
FROM eclipse-temurin:21-jre

WORKDIR /app

COPY target/learning-platform-0.1.0-SNAPSHOT.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

`COPY target/learning-platform-0.1.0-SNAPSHOT.jar app.jar` brings the already-built JAR into the image under a fixed, simple name, and `ENTRYPOINT ["java", "-jar", "app.jar"]` is exactly the fixed-command pattern "`CMD` vs `ENTRYPOINT`" recommended — this container's one job is always running this one JAR. This works, but it has a real prerequisite worth naming: `mvn package` has to be run *on the host machine*, before `docker build`, so the JAR already exists for `COPY` to find. "Multi-Stage Builds," later in this lesson, removes that prerequisite entirely.

`.dockerignore` — What Not to Send to the Build Context

"Building an Image: `docker build`" already established that `docker build`'s trailing `.` is the **build context** — the directory Docker reads the `Dockerfile` and every `COPY`-referenced file from. Without anything limiting it, that context includes everything in the project folder — `.git`'s full history, IDE configuration, documentation — none of which the image being built actually needs. A `.dockerignore` file, sitting next to the `Dockerfile`, excludes exactly the same way a `.gitignore` excludes files from a commit:

```
.git/
.idea/
*.iml
.mvn/
mvnw
mvnw.cmd
*.md
docs/
```

> Notice `target/` is deliberately **not** in this list yet — the single-stage Dockerfile above still needs `target/learning-platform-0.1.0-SNAPSHOT.jar` to exist in the build context for `COPY` to find. "Multi-Stage Builds" changes that, and once it does, `target/` belongs in `.dockerignore` too.

A smaller build context isn't just about disk usage — the entire context is read from disk (and, in some Docker setups, over a socket to a remote daemon) every time `docker build` runs, so a `.dockerignore` that excludes `.git` and other irrelevant directories keeps every build meaningfully faster, not just tidier.

Multi-Stage Builds

A **multi-stage build** uses more than one `FROM` in a single `Dockerfile`, where each `FROM` starts a fresh stage, and a later stage can selectively copy specific files out of an earlier one — discarding everything else that earlier stage produced or contained.

```
# Stage 1: build the JAR with Maven -- this stage's contents never reach the final image.
FROM maven:3.9-eclipse-temurin-21 AS builder

WORKDIR /build

COPY pom.xml .
COPY src ./src

RUN mvn -B package -DskipTests

# Stage 2: run it with just a JRE -- no Maven, no JDK, no source code.
FROM eclipse-temurin:21-jre

WORKDIR /app

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

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

The first stage, named `builder` via `FROM ... AS builder`, uses a Maven+JDK image to run `mvn package` *inside the container itself* — this project's `pom.xml` and `src/` go in, a JAR comes out, entirely inside Docker's build process, with no dependency on Maven or a JDK already being installed on the host machine at all. The second stage starts completely fresh from the same small `eclipse-temurin:21-jre` base as before, and `COPY --from=builder /build/target/learning-platform-0.1.0-SNAPSHOT.jar app.jar` reaches back into the first stage to grab only the one file it actually needs. Maven, the JDK, `pom.xml`, and the full `src/` tree — everything the `builder` stage contained — never becomes part of the final image at all.

> `COPY --from=builder` only works because the earlier stage was given a name (`AS builder`) — without it, a later stage has nothing to reference. It's easy to add a second `FROM` for a genuinely new reason later and forget the `AS <name>`, at which point `--from=builder` in a subsequent `COPY` silently breaks.

JVM and Container Resource Limits

Since Java 10, the JVM has been **container-aware** by default — running inside a container with a memory limit set via `docker run -m` (or, in "Docker Compose," a service's own memory limit), it reads that cgroup-imposed limit directly, rather than seeing the host machine's full memory the way older JVMs did. By default, the JVM sizes its heap as a percentage of that limit (`-XX:MaxRAMPercentage`, 25.0 by default) — leaving room for thread stacks, metaspace, and other non-heap memory the JVM itself needs.

```
docker run -m 512m learning-platform:0.1.0
```

25% of a 512MB limit is a fairly small heap for a real Spring Boot application — for a container whose *only* job is running this one JVM (true of every example in this course), it's common to raise that percentage explicitly:

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

This still lets the JVM size itself relative to whatever memory limit the container is actually given — via `-m`, or via "Docker Compose"'s own memory settings later in this course — rather than hardcoding a fixed `-Xmx` value that would need updating by hand every time the container's memory limit changes.

Putting It All Together

The complete build-run cycle, using the multi-stage Dockerfile above and this project's own PostgreSQL container from "Docker CLI Fundamentals":

```
#!/bin/sh
# Building this project's own image with the multi-stage Dockerfile -- no
# local Maven or JDK install required, Docker builds the JAR internally --
# and running it against the PostgreSQL container from "Docker CLI Fundamentals".

docker build -t learning-platform:0.1.0 .
# Output (trimmed):
# [+] Building 42.7s (14/14) FINISHED
# => [builder 1/4] FROM docker.io/library/maven:3.9-eclipse-temurin-21
# => [builder 4/4] RUN mvn -B package -DskipTests
# => [stage-1 3/3] COPY --from=builder /build/target/learning-platform-0.1.0-SNAPSHOT.jar
app.jar
# => naming to docker.io/library/learning-platform:0.1.0

docker run -d --name learning-platform-app \
  -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://host.docker.internal:5432/postgres \
  -e SPRING_DATASOURCE_PASSWORD=secret \
  learning-platform:0.1.0
# Output:
# 9b8c7d6e5f4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c

docker logs -f learning-platform-app
# Output (trimmed):
# Started LearningPlatformApplication in 3.1 seconds

curl http://localhost:8080/en
# Output (trimmed):
# <!doctype html>
# <html> ... the actual homepage this project renders ...

docker stop learning-platform-app
docker rm learning-platform-app
```

`SPRING_DATASOURCE_URL` and `SPRING_DATASOURCE_PASSWORD` are ordinary Spring Boot environment-variable overrides — the same relaxed-binding mechanism `application.yml` / `application-prod.yml` already rely on in this project, just supplied via `-e` instead of a properties file. `host.docker.internal` lets a container reach a service running on the host machine's own network — the full story of how containers actually reach each other over the network, including a cleaner alternative to `host.docker.internal`, is "Docker Networking," the first lesson of the next category.

Common Mistakes

- Copying a JAR that doesn't exist yet — a single-stage Dockerfile like the first one in this lesson requires `mvn package` to have already run on the host; forgetting that step produces a `COPY` failure, not a silent no-op.
- Using a full JDK base image for the final, running container when a JRE image would do — extra size and attack surface for a compiler and build tooling the running application never uses.

- Skipping `.dockerignore` and sending `.git`'s entire history (and everything else in the project folder) into the build context on every single build.
- Naming a build stage `AS builder` and then referencing it as something else (or forgetting the name entirely) in a later `COPY --from=...` — the name has to match exactly.
- Hardcoding a fixed `-Xmx` value instead of relying on the JVM's default container-aware sizing (or an explicit `-XX:MaxRAMPercentage`) — a fixed value silently stops matching reality the moment the container's own memory limit changes.

Best Practices

- Prefer a JRE base image for the final stage of a multi-stage build — the JDK, Maven, and source code that produced the JAR have no reason to exist in the image that only runs it.
- Always pair a Dockerfile with a `.dockerignore` — smaller build context, faster builds, and no risk of accidentally leaking `.git` history or local IDE configuration into an image.
- Reach for a multi-stage build by default for a real application image, not just this project's own — it removes the "you must already have Maven/a JDK installed to build this image" requirement entirely, which matters for anyone building the image on a machine that isn't a developer's own.
- Let the JVM's built-in container awareness do the sizing by default, and only reach for an explicit `-XX:MaxRAMPercentage` when the container's sole purpose is running this one JVM and the default 25% is clearly too conservative.

Summary, Cheat Sheet, and Glossary

Summary

- The same Dockerfile instructions from "Docker Images and Dockerfiles" apply unchanged to a real Spring Boot JAR — only the base image and a few specifics differ.
- A JRE base image (`eclipse-temurin:21-jre`) is enough for the final, running container — a full JDK is only needed to *build* the JAR, not to run it.
- `.dockerignore` excludes files from the build context the same way `.gitignore` excludes them from a commit — smaller context, faster builds, no accidental leaks.
- A multi-stage build uses multiple `FROM` instructions in one Dockerfile, builds the JAR with Maven+JDK in an earlier, named stage, and `COPY --from=<stage>` copies only the finished JAR into a small, final JRE-based stage.
- The JVM has been container-aware since Java 10 — it reads the container's actual memory limit and sizes its heap as a percentage of it (`-XX:MaxRAMPercentage`, 25.0 by default) automatically, no manual `-Xmx` tuning required by default.

Cheat Sheet

```
FROM maven:3.9-eclipse-temurin-21 AS builder   # build stage: has Maven + JDK
WORKDIR /build
COPY pom.xml .
COPY src ./src
RUN mvn -B package -DskipTests

FROM eclipse-temurin:21-jre                     # final stage: JRE only
WORKDIR /app
COPY --from=builder /build/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

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

Glossary

- **JRE image**: a base image with only what's needed to *run* already-compiled Java bytecode — no compiler.
- **JDK image**: a base image that additionally includes the Java compiler and full toolchain, needed to *build* Java code from source.
- **`.dockerignore`**: a file next to the `Dockerfile` that excludes paths from the build context, the same way `.gitignore` excludes them from a commit.
- **Multi-stage build**: a `Dockerfile` with more than one `FROM`, where a later stage can `COPY --from=<earlier-stage>` specific files while discarding everything else that stage contained.
- **Container-aware JVM**: the JVM's default (since Java 10) behavior of reading the container's own memory limit via cgroups and sizing its heap as a percentage of it, instead of the host machine's total memory.