The Practical Project: Spring Boot + PostgreSQL with Docker

Every piece this course covered -- multi-stage Dockerfile, non-root user, HEALTHCHECK, Docker Compose, named volume, automatic networking -- brought together in one small, complete Spring Boot + PostgreSQL task tracker application. The closing lesson of the course.

Advanced 45 min

Last updated: August 31, 2026
Read online: https://www.learnforgex.com/en/topics/docker-practical-project

---

Every lesson in this course built up one piece: images and containers, the CLI, Dockerfiles, dockerizing a real Spring Boot JAR, networking, volumes, Compose, and production hardening. This final lesson doesn't teach anything new — it's a complete, standalone Spring Boot + PostgreSQL application, small enough to hold in your head all at once, that puts every one of those pieces to work together, end to end. Unlike the earlier lessons' Dockerfiles and Compose files, which containerized this platform's own, much larger `learning-platform` application, this one is a project meant to actually be built and run — a task tracker, deliberately minimal, exercising nothing this course hasn't already covered.

What We're Building

A **task tracker**: a REST API backed by PostgreSQL, with exactly two operations — list every task, and create a new one. Nothing about authentication, pagination, or task editing is included; per this platform's own example-writing principle, an exercise like this should use "the least code needed to make the concept clear," and the concept here is containerization, not application design.

```
GET  /tasks   -> list every task
POST /tasks   -> create a new task
```

The Application Itself

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

// A complete, minimal Spring Boot + PostgreSQL application -- everything
// this lesson's Dockerfile and docker-compose.yml actually run. In a real
// project, Task/TaskRepository/TaskController would each live in their own
// file; they're combined here into one compilable teaching example, the
// same convention this course's other multi-class examples already use.
@SpringBootApplication
public class TaskTrackerApplication {

    public static void main(String[] args) {
        SpringApplication.run(TaskTrackerApplication.class, args);
    }

    @Entity
    static class Task {
        @Id
        @GeneratedValue
        private Long id;
        private String title;
        private boolean done;

        public Long getId() {
            return id;
        }

        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        public boolean isDone() {
            return done;
        }

        public void setDone(boolean done) {
            this.done = done;
        }
    }

    interface TaskRepository extends JpaRepository<Task, Long> {
    }

    @RestController
    static class TaskController {

        private final TaskRepository taskRepository;

        TaskController(TaskRepository taskRepository) {
```

```
            this.taskRepository = taskRepository;
        }

        @GetMapping("/tasks")
        List<Task> listTasks() {
            return taskRepository.findAll();
        }

        @PostMapping("/tasks")
        Task createTask(@RequestBody Task task) {
            return taskRepository.save(task);
        }
    }
}
```

In a real, multi-file project, `Task`, `TaskRepository`, and `TaskController` would each live in their own file, following ordinary Spring Boot convention — they're combined into one file here as a single, self-contained teaching example, the same convention this course's other multi-class examples already follow. Three things are worth naming explicitly: `@Entity` maps `Task` onto a database table exactly the way "Entities and the Repository Abstraction" (in the Spring Data JPA course) already covered; `TaskRepository extends JpaRepository<Task, Long>` gets `findAll()` and `save()` for free, with no implementation written; and `SPRING_JPA_HIBERNATE_DDL_AUTO=update`, set in the Compose file below, is a deliberate demo-only shortcut — this platform's own real application uses Flyway migrations with `ddl-auto: validate` instead (see "JPA, Hibernate, and Spring Data JPA" in the Spring Data JPA course), which is what a real project should reach for.

Writing the Dockerfile

Everything here is a direct application of "Dockerizing a Spring Boot Application" and "Production Docker for Java Applications" — nothing new, just assembled:

```
# 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/task-tracker-0.1.0.jar app.jar

USER appuser

EXPOSE 8080

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

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

A multi-stage build ("Multi-Stage Builds") with dependency downloads cached in their own layer ("Ordering Instructions for Faster Rebuilds (Layer Caching)"), a non-root `appuser` ("Running as a Non-Root User"), and a `HEALTHCHECK` against this app's own real `/tasks` endpoint ("Health Checks: `HEALTHCHECK` in a Dockerfile") — the same pattern used throughout the last two lessons, aimed at this small application instead of `learning-platform` itself.

Writing `docker-compose.yml`

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - task-tracker-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
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
    ports:
      - "8080:8080"

volumes:
  task-tracker-db-data:
```

`db` and `app` are the same two-service shape "Docker Compose" already built — a named volume for PostgreSQL's data ("Docker Volumes"), automatic name-based networking so `app` reaches PostgreSQL at `db:5432` with no manual `docker network create` ("Container-to-Container Communication by Name"), and `depends_on: condition: service_healthy` so `app` actually waits for `db` to be ready, not just started ("Making `depends_on` Actually Wait: Compose Health Conditions").

Running It End to End

```
docker compose up -d
docker compose ps
```

Once `docker compose ps` shows both services `healthy`, the API is reachable exactly the way `curl` reached this platform's own homepage throughout this course:

```
curl -X POST http://localhost:8080/tasks -H "Content-Type: application/json" -d '{"title":
"Finish the Docker course", "done": false}'
curl http://localhost:8080/tasks
```

A created task comes back immediately from `GET /tasks` — confirmation that `app` actually reached `db` over the network this Compose file set up automatically.

Verifying Data Persistence

The real proof that this setup is correct isn't that it works once — it's that the data survives exactly the events "Docker Volumes" and "Docker Compose" said it should, and doesn't survive the one event it deliberately shouldn't:

```
#!/bin/sh
# Bringing the whole thing up, creating real data through the REST API,
# confirming it's actually stored in PostgreSQL, and confirming it survives
# a restart of just the app container.

docker compose up -d
# Output (trimmed):
# => Container task-tracker-db-1    Started
# => Container task-tracker-db-1    Healthy
# => Container task-tracker-app-1   Started

docker compose ps
# Output:
# NAME                    IMAGE               STATUS                    PORTS
# task-tracker-app-1      task-tracker-app    Up 10 seconds (healthy)   0.0.0.0:8080->8080
/tcp
# task-tracker-db-1       postgres:16         Up 15 seconds (healthy)

curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Finish the Docker course", "done": false}'
# Output:
# {"id":1,"title":"Finish the Docker course","done":false}

curl http://localhost:8080/tasks
# Output:
# [{"id":1,"title":"Finish the Docker course","done":false}]

# Restart just the app container -- the database keeps running, untouched.
docker compose restart app

curl http://localhost:8080/tasks
# Output:
# [{"id":1,"title":"Finish the Docker course","done":false}]

# The task survived the app restart, because it was never stored in the app
# container to begin with -- it lives in PostgreSQL, in the named volume.

# The real test, per "Docker Volumes": tear down BOTH containers entirely,
# not just restart one, and confirm the data is still there afterward.
docker compose down
# Output (trimmed):
# => Container task-tracker-app-1   Removed
# => Container task-tracker-db-1    Removed
# => Network task-tracker_default   Removed

docker compose up -d
# Output (trimmed):
# => Container task-tracker-db-1    Started
# => Container task-tracker-app-1   Started

curl http://localhost:8080/tasks
# Output:
# [{"id":1,"title":"Finish the Docker course","done":false}]

# Still there -- docker compose down (without -v) never touched the named
# volume, exactly as "Docker Compose" described.
```

A task created before `docker compose down` is still there after `docker compose up -d` brings everything back — because `docker compose down`, without `-v`, never touches the named volume it created, exactly as "Docker Compose" described. Running `docker compose down -v` instead, at any point, would be the one command in this entire setup that actually deletes that data — worth trying once, deliberately, to see the difference for yourself.

What This Project Demonstrates

Every file here is small enough to read in full, and every line in it traces back to a specific earlier lesson:

```
TaskTrackerApplication.java  -> a real Spring Data JPA entity + repository + controller
TaskTrackerDockerfile        -> Dockerizing a Spring Boot Application, Production Docker
TaskTrackerCompose.yml       -> Docker Networking, Docker Volumes, Docker Compose
```

Nothing about a larger, real-world Spring Boot application changes this picture in kind — a bigger `pom.xml`, more entities, more endpoints, even this platform's own `learning-platform` itself, all containerize with exactly this same shape: a multi-stage Dockerfile, a named volume for the database, and a `docker-compose.yml` tying it together.

Common Mistakes

- Treating `SPRING_JPA_HIBERNATE_DDL_AUTO=update` as something a real, production Spring Boot project should use — it's a deliberate shortcut for this small demo only; a real project uses Flyway migrations and `ddl-auto: validate` instead (see "The Application Itself").
- Skipping the `docker compose down` / `docker compose up -d` cycle in "Verifying Data Persistence" and only restarting the `app` service — that only proves the app container is stateless, not that the volume itself survives a real teardown.
- Forgetting that `POSTGRES_PASSWORD` and `SPRING_DATASOURCE_PASSWORD` in this lesson's Compose file are illustrative, matching "Docker Compose"'s own note that a real deployment moves values like these to a separate, uncommitted `.env` file or a proper secrets mechanism.

Best Practices

- Build and run this exact project once, by hand, rather than only reading it — every command here has already been run and verified individually in an earlier lesson, but running the whole thing together end to end is what actually cements the full picture.
- When containerizing a real application later, reach for this same shape by default: a multi-stage Dockerfile with cached dependency downloads and a non-root user, a named

volume for anything stateful, and a Compose file with real health conditions — not because this course said so, but because each piece was chosen to solve a real, specific problem covered along the way.

- Keep treating `docker compose down -v` as a deliberate, separate decision, never a habit — the fact that this lesson's own demo relies on plain `docker compose down` leaving data intact is exactly why.

Summary, Cheat Sheet, and Glossary

Summary

- This lesson's task tracker is a small, complete Spring Boot + PostgreSQL application, containerized using nothing beyond what this course already covered.
- Its Dockerfile applies "Dockerizing a Spring Boot Application" and "Production Docker for Java Applications" directly: multi-stage build, cached dependency layer, non-root user, `HEALTHCHECK`.
- Its `docker-compose.yml` applies "Docker Networking," "Docker Volumes," and "Docker Compose" directly: automatic name-based service discovery, a named volume for PostgreSQL's data, and a health-based `depends_on`.
- Data created through the API survives a full `docker compose down` / `docker compose up -d` cycle — proof the named volume, not either container, is where the data actually lives.
- The same shape — multi-stage Dockerfile, named volume, Compose with health conditions — scales unchanged to a much larger real application, this platform's own `learning-platform` included.

Cheat Sheet

```
docker compose up -d              # build and start everything
docker compose ps                 # confirm both services are healthy
curl -X POST http://localhost:8080/tasks -H "Content-Type: application/json" -d '{"title":
"...", "done": false}'
curl http://localhost:8080/tasks  # list tasks
docker compose down               # tear down containers; volume survives
docker compose up -d              # bring it back; data is still there
```

Glossary

- **Task tracker**: this lesson's own minimal, complete Spring Boot + PostgreSQL demo application — two endpoints, one entity, deliberately nothing more.
- **End-to-end verification**: confirming a system works by actually exercising it — creating real data through its real API, then proving that data survives the specific events it's supposed to survive.