What Is Docker?

What Docker is, how containers differ from virtual machines, the difference between an image and a container, the Docker Engine, and Docker Hub. The 1st lesson in both the Docker course and its Docker Fundamentals category.

Beginner 15 min

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

---

This is the first lesson of the new Docker course, and the first lesson of its "Docker Fundamentals" category. You already know Java and Spring Boot, and this platform's own PostgreSQL course already covered how a relational database stores and serves data. None of that has explained yet how the running application itself — the actual JVM process, with its own dependencies, its own JDK version, talking to a specific database over a specific network — gets from "code on a developer's machine" to "running the same way everywhere else." That is exactly the gap Docker fills, and this lesson builds the mental model everything else in this course depends on: what a container actually is, what an image actually is, and why this problem needed solving in the first place.

What Is Docker?

Docker is a platform for packaging an application together with everything it needs to run — its dependencies, its runtime, its configuration — into a single, portable unit called a **container**, and then running that unit consistently on any machine that has Docker installed. A Spring Boot application built on a developer's laptop and a copy of the exact same application running on a production server are, without Docker, two different environments that merely try to resemble each other: different JDK patch versions, different installed libraries, different operating systems. With Docker, they can be the *same* environment, packaged once and run unchanged everywhere.

Why Does It Exist?

"It works on my machine" is the specific, recurring problem Docker exists to solve. Before containers were common, deploying a Java application meant relying on the target machine already having the right JDK version installed, the right environment variables set, and no conflicting version of some other dependency already present — and any mismatch between a developer's machine, a test server, and production could turn into a bug that only reproduces in one of those three places. Docker's answer is to stop relying on the

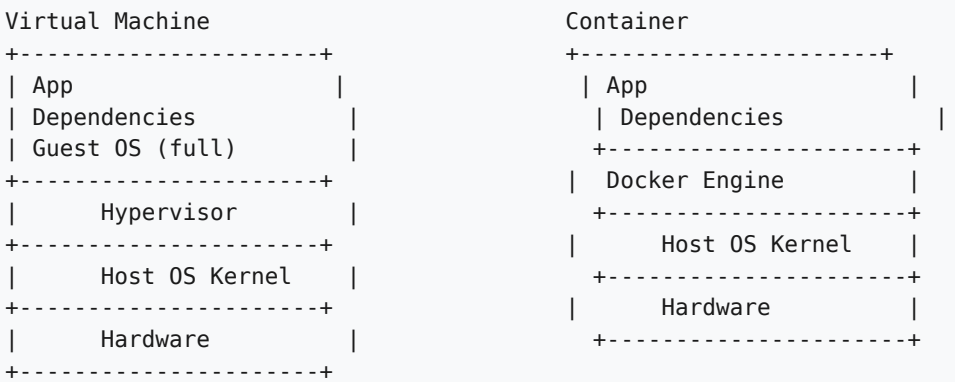
target machine's own installed software at all: the application ships together with its entire runtime environment, as one unit, so the only thing every machine needs in common is Docker itself.

History

Containers as a general idea predate Docker by decades — Unix `chroot` (1979) and later Linux namespaces and cgroups (added to the Linux kernel through the mid-2000s) already let a process be isolated from the rest of a system. Docker, Inc. (originally a company called dotCloud) released the Docker Engine as an open-source project in 2013, and its real contribution wasn't inventing container isolation from scratch — it was making that existing Linux kernel capability usable through a simple command-line tool, a standard image format, and Docker Hub, a public place to share those images. That combination is what took containers from a specialist Linux technique to a default part of how software gets built and shipped, Java and Spring Boot applications very much included.

Containers vs. Virtual Machines

A virtual machine and a container solve a similar-sounding problem — isolating one application's environment from another's — in fundamentally different ways, and the difference is worth seeing side by side.



A virtual machine runs a complete guest operating system — its own kernel, its own copy of everything — on top of a hypervisor, which is itself software that emulates hardware. A container runs directly on the host machine's existing kernel, isolated from other containers by the same Linux namespaces and cgroups mentioned in "History," with no second operating system underneath it. The practical consequence is size and speed: a VM image is typically gigabytes and takes on the order of minutes to boot a full OS; a container image can be tens of megabytes and starts in roughly the time it takes the application process itself to start, because there is no OS to boot — only the application process, starting directly.

Images vs. Containers

These two words are used precisely in Docker, and conflating them is one of the most common early mistakes (see "Common Mistakes"). An **image** is a read-only, packaged template — application code, a runtime, installed dependencies, and configuration, frozen into a single artifact. A **container** is a running (or stopped) instance created *from* an image, in exactly the same relationship a Java class has to its instances: one `Book` class, many `Book` objects, each with its own state, all built from the same blueprint. One Spring Boot image can be the source of several running containers at once — three containers all started from the same image, each an independent, isolated process, none of them sharing runtime state with the others.

```
image: my-app:1.0            (the blueprint — built once)
     |
     +--> container A  (running)
     +--> container B  (running)
     +--> container C  (stopped)
```

Starting a container from an image doesn't consume or modify that image — the same image can be used to start any number of containers, independently, and the image itself stays exactly as it was built. Later lessons in this category cover exactly how an image gets built ("Docker Images and Dockerfiles") and exactly how a container is started, inspected, and stopped from the command line ("Docker CLI Fundamentals").

The Docker Engine

The **Docker Engine** is the background service (a daemon, conventionally named `dockerd`) that actually does the work: building images, starting and stopping containers, managing the isolated networking and storage each container gets. Everything a developer types at the command line — `docker run`, `docker build`, and the rest, covered in full in "Docker CLI Fundamentals" — is a client command that talks to this daemon and asks it to do something; the CLI itself doesn't run containers directly, it's a thin client to the engine that does.

Docker Hub and Image Registries

An image, once built, needs somewhere to live so it can be pulled down and run on a different machine — that's what an **image registry** is, and **Docker Hub** is the default, publicly hosted one Docker itself points to out of the box. A registry stores images under a name and tag (`postgres:16`, `eclipse-temurin:21-jre`) the same way a package repository like Maven Central stores JAR artifacts under a group, artifact, and version — and just as this project's own `pom.xml` pulls dependencies from Maven Central by coordinate rather than shipping them by hand, a `docker pull` fetches an image from a registry by name and tag

rather than building it locally from scratch every time. A team can also run its own private registry instead of, or alongside, Docker Hub — the mechanism is the same either way, only the location differs.

Where Docker Fits Around This Project

Right now, running this project locally means having a JDK installed, having Maven resolve dependencies, and having a PostgreSQL server already running and reachable — three separate pieces of setup a new developer has to get right independently, on their own machine, before `mvn spring-boot:run` (or the workarounds this sandbox itself has needed, see `docs/known-constraints.md`) will even start successfully. Nothing about this project's own code changes because of Docker — the JAR Maven already produces is still the JAR that runs. What Docker adds is a way to package that JAR together with a matching JDK into one image ("Dockerizing a Spring Boot Application," later in this category), and to describe "this app plus a PostgreSQL instance, networked together" as a single, reproducible unit ("Docker Compose," in the next category) — so that getting this project running looks the same on any machine with Docker installed, instead of depending on what happens to already be installed on it.

Best Practices

- Keep "image" and "container" as precisely separate concepts from the start (see "Images vs. Containers") — nearly every later topic in this course assumes that distinction is already solid.
- When comparing Docker to a virtual machine, reach for the shared-kernel fact first (see "Containers vs. Virtual Machines") — it's the one architectural difference that explains both the size and the startup-time gap.
- Think of Docker Hub the same way this project's own `pom.xml` already treats Maven Central — a shared, named source of pre-built artifacts, not something to reinvent locally.
- Remember that the Docker Engine, not the `docker` command itself, is what actually runs containers — the CLI is a client talking to a background service.

Common Mistakes

- Using "image" and "container" interchangeably — an image is the frozen blueprint; a container is one running (or stopped) instance created from it, and a single image can back many containers at once.
- Assuming a container is "a lightweight virtual machine" — it isn't a VM at all; it shares the host's kernel directly instead of running its own guest OS under a hypervisor.
- Expecting Docker to change what this project's Java code does — it doesn't; the same JAR runs either way, Docker only changes how consistently the environment around it is packaged and shipped.

- Assuming Docker Hub is the only possible registry — it's the default, but private and self-hosted registries work through the identical pull/push mechanism.

Summary, Cheat Sheet, and Glossary

Summary

- Docker packages an application with its entire runtime environment into a container, so it runs the same way on any machine that has Docker installed — solving the "it works on my machine" problem directly.
- A container shares the host machine's kernel instead of running a full guest operating system, which is why containers are dramatically smaller and faster to start than virtual machines.
- An image is a read-only, built-once blueprint; a container is a running (or stopped) instance created from that image — one image can back many independent containers.
- The Docker Engine (`dockerd`) is the background service that actually builds images and runs containers; the `docker` command line is a client that talks to it.
- Docker Hub is a public image registry — the same role Maven Central plays for this project's own JAR dependencies, but for container images.

Cheat Sheet

```
image        read-only blueprint, built once      (e.g. my-app:1.0)
container    a running/stopped instance of an image
dockerd      the Docker Engine daemon that does the actual work
docker CLI   the client you type commands into; talks to dockerd
registry     where images live and get pulled from (Docker Hub = default)
```

Glossary

- **Container**: an isolated, running (or stopped) instance created from an image, sharing the host machine's kernel directly.
- **Image**: a read-only, packaged template — code, runtime, dependencies, and configuration — that containers are created from.
- **Docker Engine (`dockerd`)**: the background daemon that builds images and runs containers; the actual worker behind the `docker` command.
- **Image registry**: a service that stores and serves images by name and tag; Docker Hub is the default public one.
- **Docker Hub**: Docker's own public image registry, the default source `docker pull` reaches out to.