Docker Images and Dockerfiles

What a Dockerfile is, the FROM, WORKDIR, COPY, RUN, EXPOSE instructions, the difference between CMD and ENTRYPOINT, and building an image with docker build -- via a generic, minimal web server example.

Beginner 30 min

Last updated: August 31, 2026
Read online: https://www.learnforgex.com/en/topics/docker-images-and-dockerfiles

---

"Docker CLI Fundamentals" pulled and ran an image someone else had already built — `postgres:16` came ready-made from Docker Hub. This lesson is about building your own: writing a `Dockerfile`, the plain-text recipe Docker reads to produce a new image, and running `docker build` to actually turn that recipe into one. The example here is deliberately generic — a tiny web server, nothing Java-specific yet — so the Dockerfile instructions themselves are what stays in focus; "Dockerizing a Spring Boot Application," the next lesson in this category, applies exactly the same instructions to a real Spring Boot JAR.

What Is a Dockerfile?

A `Dockerfile` is a plain-text file, conventionally named exactly `Dockerfile` with no extension, containing a sequence of instructions Docker executes in order to produce an image. Each instruction — `FROM`, `COPY`, `RUN`, and the rest covered below — adds one new, cached layer on top of the previous one, which is why an image built from a `Dockerfile` behaves like the frozen template described in "Images vs. Containers": once built, it's a fixed, reproducible artifact, not something `docker build` re-derives differently each time it's re-run against unchanged inputs.

`FROM` — Choosing a Base Image

Every `Dockerfile` starts with `FROM`, naming the image everything else builds on top of — exactly the same `<repository>:<tag>` syntax "Pulling an Image: `docker pull`" already covered.

```
FROM alpine:3.20
```

Alpine Linux is a common base for exactly this reason: it's a real, minimal Linux distribution, only a few megabytes, with a package manager (`apk`) available for anything more it needs — a deliberately small starting point instead of a general-purpose OS image with tools this particular image will never use.

`WORKDIR` — Setting the Working Directory

`WORKDIR` sets the directory every subsequent instruction runs relative to, creating it inside the image if it doesn't already exist.

```
WORKDIR /app
```

Without an explicit `WORKDIR`, later instructions like `COPY` and `RUN` operate relative to the image's filesystem root — technically valid, but it mixes an application's files in with the base image's own system directories. Setting `WORKDIR /app` once near the top keeps everything that follows unambiguous and tidy.

`COPY` — Bringing Files Into the Image

`COPY` brings a file (or directory) from the **build context** — the folder `docker build` is run from — into the image being built.

```
COPY index.html .
```

The `.` here is shorthand for "the current `WORKDIR`" — with `WORKDIR /app` already set, this places the file at `/app/index.html` inside the image. `COPY` only ever reads from the build context on the host machine; it cannot reach outside it, which is exactly why `docker build` is always run from the folder containing everything the image actually needs.

`RUN` — Executing Commands While Building

`RUN` executes a command *while the image is being built*, and whatever that command changes on disk becomes a permanent part of the resulting image.

```
RUN apk add --no-cache python3
```

This installs Python 3 into the image using Alpine's package manager — `--no-cache` avoids leaving the package manager's own download cache behind, keeping the resulting image smaller (see "Best Practices"). This is the mechanism that separates a base image like `alpine:3.20` from a finished, purpose-built one: `RUN` is where whatever that specific image needs gets installed, once, at build time — not repeated every time a container starts from it.

`EXPOSE` — Documenting a Container's Port

`EXPOSE` declares which port the application inside the container listens on.

```
EXPOSE 8080
```

> `EXPOSE` is documentation, not configuration — it does not actually publish the port to the host machine on its own. Reaching a containerized application from outside still requires `-p` on `docker run` (`-p 8080:8080`, already covered in "Running a Container: `docker run`"), which "Docker Networking," later in this course, explains in full. `EXPOSE` mainly helps a human (or another tool) reading the `Dockerfile` know what the image expects.

`CMD` vs `ENTRYPOINT`

Both `CMD` and `ENTRYPOINT` specify what runs when a container starts — the difference is what happens when `docker run` is given extra arguments.

```
FROM alpine:3.20

CMD ["echo", "Hello from CMD"]
```

```
docker run my-image
# Output: Hello from CMD

docker run my-image echo "Overridden"
# Output: Overridden
```

With only `CMD`, any command given to `docker run` **replaces** it entirely. `ENTRYPOINT` behaves differently — it's the fixed part that always runs, and `CMD` (when both are present) supplies its *default arguments*, which `docker run` can still override without touching the entrypoint itself:

```
FROM alpine:3.20

ENTRYPOINT ["echo"]
CMD ["Hello from CMD"]
```

```
docker run my-image
# Output: Hello from CMD

docker run my-image "Overridden"
# Output: Overridden
```

In the second version, `echo` always runs — `docker run my-image "Overridden"` only replaces the default argument, producing `echo "Overridden"`, not a completely different command. This `ENTRYPOINT` + default-`CMD` pairing is the pattern most real-world images use, including the official `postgres` image already run in "Docker CLI Fundamentals" — its entrypoint always starts PostgreSQL; the arguments after it can be overridden without losing that.

Building an Image: `docker build`

`docker build` reads a `Dockerfile` and produces an image from it.

```
docker build -t minimal-web-server:1.0 .
```

```
[+] Building 4.2s (9/9) FINISHED
 => [1/4] FROM docker.io/library/alpine:3.20
 => [2/4] RUN apk add --no-cache python3
 => [3/4] WORKDIR /app
 => [4/4] COPY index.html .
 => exporting to image
 => naming to docker.io/library/minimal-web-server:1.0
```

`-t minimal-web-server:1.0` tags the resulting image with a repository name and version — the exact same `<repository>:<tag>` shape as `postgres:16`, now applied to an image built locally instead of pulled from Docker Hub. The trailing `.` is the **build context**: the directory `docker build` reads the `Dockerfile` and any `COPY`-referenced files from — it's the same reason `COPY index.html .` in the `Dockerfile` above works, since `index.html` sits right next to the `Dockerfile` in that directory.

Putting It All Together: A Minimal Dockerfile

The complete, working `Dockerfile` this lesson has been building toward:

```
FROM alpine:3.20

RUN apk add --no-cache python3

WORKDIR /app

COPY index.html .

EXPOSE 8080

CMD ["python3", "-m", "http.server", "8080"]
```

```
<!doctype html>
<html>
  <head>
    <title>Minimal Web Server</title>
  </head>
  <body>
    <h1>Served from inside a container.</h1>
  </body>
</html>
```

And the full build-run-verify-cleanup cycle, using exactly the CLI commands "Docker CLI Fundamentals" already covered:

```
#!/bin/sh
# Building the minimal web server image from its Dockerfile, running it,
# confirming it actually serves the page, and cleaning up afterward.

docker build -t minimal-web-server:1.0 .
# Output (trimmed):
# [+] Building 4.2s (9/9) FINISHED
# => [1/4] FROM docker.io/library/alpine:3.20
# => [2/4] RUN apk add --no-cache python3
# => [3/4] WORKDIR /app
# => [4/4] COPY index.html .
# => exporting to image
# => naming to docker.io/library/minimal-web-server:1.0

docker run -d --name minimal-web-server -p 8080:8080 minimal-web-server:1.0
# Output:
# 3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d

curl http://localhost:8080
# Output:
# <!doctype html>
# <html>
#   <head>
#     <title>Minimal Web Server</title>
#   </head>
#   <body>
#     <h1>Served from inside a container.</h1>
#   </body>
# </html>

# Confirmed -- the container is actually serving the file COPY placed inside
# the image. Clean up.
docker stop minimal-web-server
docker rm minimal-web-server
```

Every instruction in this `Dockerfile` was introduced above individually — `FROM` picks the base, `RUN` installs Python, `WORKDIR` and `COPY` place the file being served, `EXPOSE` documents the port, and `CMD` starts the server. Nothing here is Java-specific yet, which is exactly the point: "Dockerizing a Spring Boot Application" reuses this same instruction set, unchanged, just aimed at a Spring Boot JAR and a JRE base image instead of Python and a static file.

Common Mistakes

- Treating `EXPOSE` as if it publishes a port on its own — it doesn't; `-p` on `docker run` is what actually makes a containerized service reachable from outside (see "`EXPOSE`" above).
- Using `CMD` when the intent is really "this container always does this one thing" — `ENTRYPOINT` with a default `CMD` communicates that intent and behaves correctly when `docker run` is given extra arguments.
- Running `docker build` from the wrong directory, so a `COPY` instruction can't find the file it expects — the build context is always the directory passed to `docker build` (usually `.`), not the location of the `Dockerfile` itself if they differ.
- Skipping `WORKDIR` and letting files land at the image's filesystem root, mixed in with the base image's own system directories.

Best Practices

- Prefer a small, purpose-built base image (`alpine:3.20` here) over a large general-purpose one — every unnecessary tool in the base image is size and attack surface the final image doesn't need.
- Use `--no-cache` (Alpine) or the equivalent for your package manager on `RUN` install commands — leftover package caches only make the resulting image bigger for no benefit.
- Set `WORKDIR` explicitly near the top of the `Dockerfile`, before any `COPY` or `RUN` that depends on it, rather than relying on the filesystem root.
- Reach for `ENTRYPOINT` with a default `CMD` for images meant to always run one specific program — it documents intent more precisely than `CMD` alone.

Summary, Cheat Sheet, and Glossary

Summary

- A `Dockerfile` is a plain-text recipe of instructions Docker executes, in order, to produce an image — each instruction adds one layer.
- `FROM` picks the base image; `WORKDIR` sets the directory later instructions run relative to; `COPY` brings files from the build context into the image; `RUN` executes a command at build time and keeps its filesystem changes.
- `EXPOSE` documents which port the containerized app listens on — it does not publish that port by itself; `-p` on `docker run` does.
- `CMD` alone is fully replaced by any command given to `docker run`; `ENTRYPOINT` stays fixed and `CMD` becomes its overridable default arguments when both are present.

- `docker build -t <name>:<tag> <context>` reads a `Dockerfile` and produces a tagged image from it, using the given directory as the build context.

Cheat Sheet

```
FROM <base-image>:<tag>      # the image everything else builds on
WORKDIR <path>                # directory later instructions run relative to
COPY <src> <dest>             # bring a file from the build context into the image
RUN <command>                 # run a command at build time, keep its filesystem changes
EXPOSE <port>                 # document which port the app listens on
CMD ["executable", "arg"]     # default command; fully replaced by docker run arguments
ENTRYPOINT ["executable"]     # fixed command; CMD becomes its default (overridable)
arguments
```

```
docker build -t <name>:<tag> .   # build an image from the Dockerfile in the current
directory
```

Glossary

- **Dockerfile**: a plain-text file of ordered instructions Docker executes to build an image.
- **Layer**: the filesystem change produced by one Dockerfile instruction, cached and stacked on top of the previous one.
- **Build context**: the directory passed to `docker build` — the only place `COPY` (and similar instructions) can read files from.
- **Entrypoint**: the fixed command an `ENTRYPOINT`-configured container always runs, regardless of arguments passed to `docker run`.