Containerising the Node App with Docker

Same app, now packaged as a Docker image: one file that contains the app *and* everything it needs to run, on any machine.

- Part 1: a simple Dockerfile that works.
- Part 2: a multi-stage Dockerfile that is much smaller, and how to measure the difference.

---

0. Vocabulary and setup

Term	Meaning
Dockerfile	Recipe: step-by-step instructions to build an image
Image	The built result: read-only package of OS files + app
Container	A running instance of an image
Layer	Each instruction in a Dockerfile adds a cached layer to the image

Install Docker:

- Windows / macOS: Docker Desktop. Start it after installing.
- Linux: Docker Engine, following https://docs.docker.com/engine/install/ for your distro.

Check it:

```
docker --version
docker run --rm hello-world     # downloads a tiny image and runs it
```

All commands below run from the `node-auth-app` folder.

Part 1: A simple Dockerfile

1. Tell Docker what to ignore: `.dockerignore`

When you build, Docker sends the folder (the build context) to the builder. Exclude what the image doesn't need:

```
node_modules
dist
test-results
playwright-report
.git
*.md
```

```
Dockerfile*
.dockerignore
```

> Why `node_modules` matters most: it's ~80 MB, and it was installed for *your* OS. The container runs Linux, so it must install its own.

2. Write the `Dockerfile`

```
FROM node:26

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm ci

COPY . .

EXPOSE 5173

CMD ["npm", "run", "dev", "--", "--host", "0.0.0.0"]
```

Line	What it does
`FROM node:26`	Start from the official Node 26 image (Debian Linux + Node + npm).
`WORKDIR /app`	Create `/app` and run the next commands there.
`COPY package.json package-lock.json ./`	Copy **only** the dependency list first.
`RUN npm ci`	Install the exact versions from `package-lock.json`. Runs at build time.
`COPY . .`	Copy the rest of the source code.
`EXPOSE 5173`	Documents the port the app listens on. Doesn't open it by itself.
`CMD [...]`	Command run when a container starts: the Vite dev server.

Two details worth understanding:

- Why copy `package*.json` before the code? Layers are cached. If you only change `src/main.js`, Docker reuses the cached `npm ci` layer, and rebuilds take seconds instead of minutes.
- Why `--host 0.0.0.0`? By default Vite only listens to `localhost` *inside* the container, which your browser can't reach. `0.0.0.0` means "accept connections from outside".

> `npm ci` needs `package-lock.json`. It's created by `npm install`. Keep it in the project.

3. Build the image

```
docker build -t node-auth-app:simple .
```

- `build`: create an image from the `Dockerfile` in this folder.
- `-t node-auth-app:simple`: name the image `node-auth-app`, with the tag `simple` (like a version label).
- `.`: the build context is the current folder.

The first build downloads the Node image and takes a minute. Run it again: almost instant, every step says `CACHED`.

4. Run a container

```
docker run --rm -p 5173:5173 node-auth-app:simple
```

- `run`: start a container from the image.
- `--rm`: delete the container when it stops (keeps things clean).
- `-p 5173:5173`: port mapping, `host:container`. Traffic to port 5173 on your machine goes to port 5173 in the container.

Open http://localhost:5173, register and log in to confirm it works. Stop with `Ctrl+C`.

5. Confirm with the Playwright tests

Run the container in the background, then run the e2e tests against it:

```
docker run --rm -d -p 5173:5173 --name auth-app node-auth-app:simple
npx playwright test
docker stop auth-app
```

- `-d`: detached, runs in the background and gives you the terminal back.
- `--name auth-app`: a name, so you can refer to the container later.
- Playwright finds something already running on port 5173 (`reuseExistingServer: true`) and tests the container instead of starting its own server. Expect `2 passed`.

Useful commands

Command	Purpose
`docker ps`	List running containers
`docker logs auth-app`	Show a container's output
`docker stop auth-app`	Stop a container

Command	Purpose
`docker images`	List images and their sizes
`docker rmi node-auth-app:simple`	Delete an image

Part 2: Multi-stage build and image size

6. The problem

Check the size of what you just built:

```
docker images node-auth-app
```

Look at the DISK USAGE (or SIZE) column: expect over 1 GB. For an app whose code is a few KB. Where does it go?

- A full Debian Linux with compilers and tools (the `node:26` base).
- Node and npm.
- `node_modules` (~80 MB): Vite, Playwright...
- A development server.

But this app is frontend-only. In production it's just static HTML, CSS and JS. Let's see how small:

```
npm run build
```

```
dist/index.html                 0.46 kB
dist/assets/index-....css       0.80 kB
dist/assets/index-....js        3.74 kB
```

The whole app is ~20 KB in `dist/`. Node, npm and `node_modules` are only needed to produce these files, not to serve them.

7. Quick win: a smaller base image

Change the first line of `Dockerfile`:

```
FROM node:26-alpine
```

Alpine is a minimal Linux (a few MB, vs over 100 MB for Debian). Build with a new tag so you can compare:

```
docker build -t node-auth-app:alpine .
```

Then change the line back to `FROM node:26`. Better, but it still ships Node, npm, `node_modules` and a dev server.

8. The multi-stage Dockerfile: `Dockerfile.multistage`

A multi-stage build uses several `FROM` lines. Each starts a new stage. Only the last stage becomes the image; earlier ones are thrown away, except for files you explicitly copy out.

```
# ---------- Stage 1: build the static files ----------
FROM node:26-alpine AS builder

WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

# ---------- Stage 2: serve them with nginx ----------
FROM nginx:stable-alpine

COPY --from=builder /app/dist /usr/share/nginx/html

EXPOSE 80
```

Line	What it does
`FROM node:26-alpine AS builder`	Stage 1, named `builder`. Has Node to run the build.
`RUN npm run build`	Produces `/app/dist` inside stage 1.
`FROM nginx:stable-alpine`	Stage 2 starts from scratch with nginx, a small, fast web server. Nothing from stage 1 is included.
`COPY --from=builder /app/dist ...`	Copy only the built files out of stage 1, into the folder nginx serves.
`EXPOSE 80`	nginx listens on port 80. Its image already has a `CMD`, so we don't need one.

Build it. `-f` picks a Dockerfile other than the default:

```
docker build -f Dockerfile.multistage -t node-auth-app:multistage .
```

9. Run and confirm

```
docker run --rm -d -p 5173:80 --name auth-app node-auth-app:multistage
npx playwright test
docker stop auth-app
```

Note `-p 5173:80`: your port 5173 now maps to nginx's port 80 in the container. The tests don't change at all, and should still give `2 passed`. You can also open http://localhost:5173 in your browser.

10. Compare the sizes

```
docker images node-auth-app
```

Newer Docker versions show two columns:

- DISK USAGE: unpacked size on your machine.
- CONTENT SIZE: compressed size, what gets downloaded when someone pulls the image.

Compare the same column across tags. For a clean table:

```
docker image ls node-auth-app --format "table {{.Tag}}\t{{.Size}}"
```

Fill in your results:

Tag	Base image	Size (yours)	Rough expectation (unpacked)
`simple`	`node:26`		> 1 GB
`alpine`	`node:26-alpine`		~ 250–300 MB
`multistage`	`nginx:stable-alpine`		~ 50 MB

To see which step adds the most, list an image's layers:

```
docker history node-auth-app:simple
docker history node-auth-app:multistage
```

Look at the `SIZE` column: in `simple`, the base image layers and `RUN npm ci` dominate. In `multistage`, the `COPY --from=builder` layer is just a few KB.

Why smaller is better: faster to download and deploy, less disk and storage cost, and fewer installed packages means fewer potential security vulnerabilities.

> Dev vs production: the `simple` image runs a development server (hot reload, unminified code). The `multistage` image serves the production build. Use the first while developing, the second to deploy.

11. Exercises

1. Cache in action. Change a text in `src/main.js`, rebuild `:simple`. Which steps say `CACHED`? Now add a dependency to `package.json` and rebuild. What changed?
2. Break the ignore file. Remove `node_modules` from `.dockerignore`, run `npm install` locally, rebuild. Does the build get slower? Why is it risky to copy your own `node_modules` into a Linux container? Restore the line afterwards.
3. Inspect a container. While `multistage` runs, open a shell inside it: `docker exec -it auth-app sh`, then `ls /usr/share/nginx/html`. Is there any Node or `node_modules` in there?
4. Even smaller? Try `nginx:stable-alpine-slim` in stage 2 and compare sizes.