Docker Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What a container actually is 3
Installing and verifying Docker 4
Images, containers, and registries 5
Running and inspecting long-lived containers 7
Building a real app to containerize: the Notes API 8
Writing Dockerfiles, `.dockerignore`, and layer caching 10
Data persistence: volumes vs bind mounts 17
Container networking and DNS service discovery 18
Multi-container apps with Docker Compose 20
Debugging runtime failures and managing disk space 25
Tagging, pushing, and shipping to production 29

Preface

This is the Docker Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

Docker solves one of the oldest headaches in software development: an app that runs on your laptop but breaks on a teammate's machine or on a production Linux server because of a different runtime version, missing system library, or mismatched database setup.

This book takes you from running your first container to packaging, running, debugging, and shipping a real multi-container application: a Node.js Notes API backed by PostgreSQL 17. Along the way we measure real image sizes, prove layer caching, test non-root security, verify data persistence across container recreation, trigger real exit code `1` and `137` (OOM) failures, and inspect how Docker networking works under the hood.

Every command, size, timing, and output in this book comes from a real run on my Mac (using OrbStack's Linux VM) in November 2026, on these versions:

Software	Version
Docker Engine & CLI	29.8.2 (`linux/arm64`)
Docker Compose	v5.5.0
Base images	`alpine:3.21`, `nginx:1.27-alpine`, `node:22`, `node:22-alpine`, `postgres:17-alpine`
Node.js (inside container)	22.x

What a container actually is

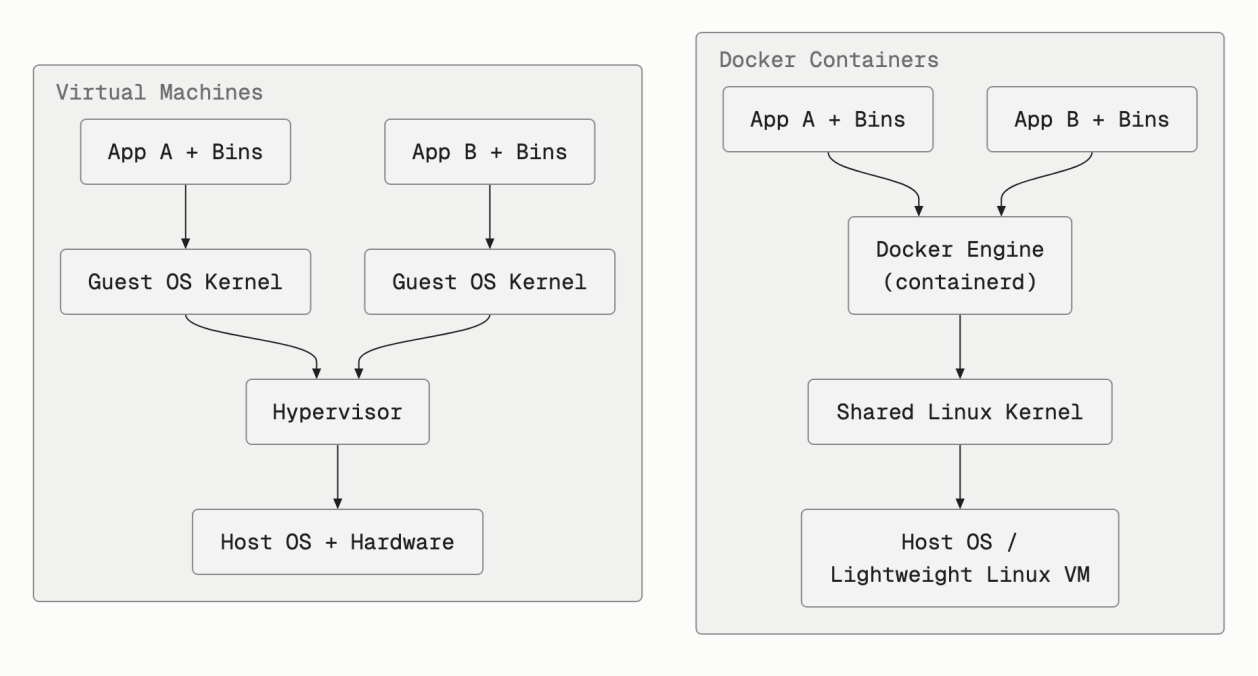
Before typing a single `docker` command, it helps to clear up the most common misconception: **a container is not a tiny virtual machine.**

A virtual machine emulates hardware and boots an entire guest operating system with its own kernel, init system, and gigabytes of RAM overhead. Starting a VM takes tens of seconds.

A Linux container is just a regular Linux process (or small group of processes) running on the host's Linux kernel, wrapped in two kernel features:

- **Namespaces** isolate what the process can *see*: its own process tree (`pid`), filesystem mount table (`mnt`), network interfaces and IP address (`net`), hostname (`uts`), and user IDs (`user`). Inside a container, `node server.js` thinks it is PID 1 on its own machine.
- **Control groups (cgroups)** limit what the process can *use*: maximum memory (`--memory 256m`), CPU shares (`--cpus 1`), and process counts.

Because every container shares the host Linux kernel, containers start in milliseconds and use only a few megabytes of RAM beyond the application itself.



On macOS and Windows, Docker cannot use the host kernel directly because macOS uses XNU and Windows uses NT. Tools like OrbStack (macOS) or WSL 2 (Windows) run one lightweight Linux VM in the background, and all your containers share that single Linux kernel.

Dimension	Virtual Machine	Docker Container
Isolation unit	Full hardware + guest OS kernel	Linux namespaces + cgroups on shared kernel
Startup time	10–60 seconds	Under 1 second
Disk footprint	Gigabytes per VM	Tens to hundreds of megabytes (shared layers)
RAM overhead	Fixed OS allocation per VM	Only the process memory itself
Portability	Heavy VM images	Lightweight OCI images pushed to any registry

Installing and verifying Docker

macOS

On macOS, I use **OrbStack** instead of Docker Desktop. It boots in about two seconds, uses far less RAM and battery, and provides the same `docker` and `docker compose` CLI commands.

Install it with Homebrew or download it from `orbstack.dev`, then start the daemon:

```
brew install --cask orbstack
orb start
```

Ubuntu Linux

On a Linux server (like Ubuntu 24.04), install Docker Engine directly from Docker's official repository so you get current `docker` and `docker compose` (v2/v5 plugin) releases rather than outdated distro packages:

```
curl -fsSL https://get.docker.com | sudo sh
sudo usermod -aG docker "$USER"
```

Log out and back in after `usermod` so your user can run `docker` without `sudo`. Keep in mind that anyone in the `docker` group has root-equivalent access to the host (because they can mount `/` into a container), so only add trusted admin users.

Verifying your installation

Check both the Docker client/server and the Compose plugin:

```
docker version --format 'Client: {{.Client.Version}} | Server: {{.Server.Version}} ({{.Server.Os}}/{{.
    ↪ Server.Arch}})'
docker compose version
```

```
Client: 29.8.2 | Server: 29.8.2 (linux/arm64)
Docker Compose version v5.5.0
```

Notice we use `docker compose` (with a space), not the legacy `docker-compose` (with a hyphen). The standalone Python `docker-compose` v1 binary has been deprecated for years.

Images, containers, and registries

Three words show up everywhere in Docker. Keep them distinct:

- **Image:** a read-only, immutable template built from stacked filesystem layers plus metadata (default command, environment variables, exposed ports, user). It works like a class or a snapshot.
- **Container:** a running (or stopped) instance of an image, with one thin **read-write layer** mounted on top. It works like an object created from that class.
- **Registry:** a server that stores and distributes images by name and tag. Docker Hub (`docker.io`) is the default, while GitHub Container Registry (`ghcr.io`) and Amazon ECR are common for private apps.

Tags, digests, and why `latest` is a trap

When you pull `alpine:3.21`, `alpine` is the repository name and `3.21` is a **tag**. Tags are mutable pointers: `node:22` moves forward whenever a new 22.x patch releases, and `latest` simply means "whatever the maintainer most recently tagged as `latest`".

In a Dockerfile or `compose.yaml`:

- Never use `:latest` in production. A routine rebuild six months from now can silently pull a breaking major version.
- Pin at least a major or minor version (`node:22-alpine`, `postgres:17-alpine`, `nginx:1.27-alpine`).
- For strict reproducibility, pin the immutable SHA-256 **digest** (`alpine:3.21@sha256:bc214871...`).

Your first container and the ephemeral filesystem

Let's run a one-shot container from `alpine:3.21`. The `--rm` flag tells Docker to delete the container as soon as its command exits so stopped containers don't pile up:

```
docker run --rm --name docker-handbook-hello alpine:3.21 \
  sh -c 'cat /etc/os-release | head -n 2; uname -m; echo "hello from alpine"'
```

```
Unable to find image 'alpine:3.21' locally
3.21: Pulling from library/alpine
Digest: sha256:bc21487101ae47172509130ee1092cc273a831968d64b617776d1f69c81e42ab
Status: Downloaded newer image for alpine:3.21
NAME="Alpine Linux"
ID=alpine
aarch64
hello from alpine
```

Docker didn't have `alpine:3.21` cached locally, so it pulled the layers from Docker Hub once, started the container, ran our shell command, printed the output, and removed the container. Running the same command a second time starts instantaneously from the local cache.

Now let's prove one of Docker's golden rules: **anything written inside a container's writable layer disappears when the container is removed.**

```
docker run --rm --name docker-handbook-ephem1 alpine:3.21 \
  sh -c 'echo "temp" > /tmp/ephem.txt && cat /tmp/ephem.txt'
docker run --rm --name docker-handbook-ephem2 alpine:3.21 \
  sh -c 'ls /tmp/ephem.txt'
```

```
temp
ls: /tmp/ephem.txt: No such file or directory
```

Each `docker run` starts from the same read-only `alpine:3.21` image with a brand-new empty writable layer. That ephemeral behavior is a feature for application code (every restart is clean and predictable), and it is why databases and uploads must live in **volumes**, which we cover later.

Running and inspecting long-lived containers

One-shot commands exit immediately. Web servers, APIs, and databases run in the foreground inside the container while you detach them into the background with `-d`.

Let's start an Nginx web server on localhost port `8088`:

```
docker run -d --name docker-handbook-web -p 127.0.0.1:8088:80 nginx:1.27-alpine
```

Look closely at `-p 127.0.0.1:8088:80`:

- `80` (right side) is the port Nginx listens on **inside** the container.
- `8088` (middle) is the port on your **host** machine.
- `127.0.0.1` (left side) binds the port to loopback only. On a public Linux VPS, writing `-p 8088:80` without `127.0.0.1` binds `0.0.0.0:8088` and punches straight through `ufw` firewall rules via Docker's `iptables` rules! Always prefix `127.0.0.1:` for ports that sit behind Nginx/Caddy or are only for local development.

Check running containers with `docker ps`:

```
docker ps --filter name=docker-handbook-web \
  --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}'
```

```
NAMES                 IMAGE               STATUS        PORTS
docker-handbook-web   nginx:1.27-alpine   Up 1 second   127.0.0.1:8088->80/tcp
```

Send an HTTP request with `curl`:

```
curl -sI http://127.0.0.1:8088 | head -n 4
```

```
HTTP/1.1 200 OK
Server: nginx/1.27.5
Date: Sat, 28 Nov 2026 21:25:16 GMT
Content-Type: text/html
```

Now inspect the running container from three angles: its logs (`docker logs`), running a command inside it (`docker exec`), and its live CPU/memory footprint (`docker stats`):

```
docker logs --tail 3 docker-handbook-web
```

```
2026/11/28 21:25:15 [notice] 1#1: start worker process 41
2026/11/28 21:25:15 [notice] 1#1: start worker process 42
192.168.215.1 - - [28/Nov/2026:21:25:16 +0000] "HEAD / HTTP/1.1" 200 0 "-" "curl/8.7.1" "-"
```

```
docker exec docker-handbook-web nginx -v
docker stats --no-stream --format 'table {{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}' docker-handbook-web
```

```
nginx version: nginx/1.27.5
NAME                  CPU %     MEM USAGE / LIMIT
docker-handbook-web   0.00%     11.68MiB / 15.66GiB
```

11.68 MiB of RAM for a complete web server. Notice `docker exec -it <name> sh` (or `bash` on Debian/Ubuntu images) opens an interactive shell inside a running container without starting a new container.

Clean it up when you're done:

```
docker rm -f docker-handbook-web
```

Everyday Docker CLI reference

Command	What it does
`docker run --rm -it alpine:3.21 sh`	Start an interactive container and remove it on exit
`docker ps` / `docker ps -a`	List running containers / list all containers including stopped
`docker logs -f --tail 50 <name>`	Follow the last 50 log lines (`stdout` and `stderr`)
`docker exec -it <name> sh`	Run an interactive shell inside a running container
`docker stats --no-stream`	Snapshot CPU, RAM, network, and disk I/O per container
`docker inspect <name>`	Print full JSON config, mounts, IP addresses, and exit state
`docker stop <name>`	Send `SIGTERM`, wait 10s, then `SIGKILL`
`docker rm -f <name>`	Force-stop and remove a container
`docker images`	List locally cached images and their sizes
`docker system df`	Show disk usage across images, containers, volumes, and build cache

Building a real app to containerize: the Notes API

To learn Dockerfiles, layer caching, multi-stage builds, volumes, networking, and Compose properly, we need a real application with a real database.

Let's build a small Node.js 22 HTTP API (`notes-api`) that connects to PostgreSQL, exposes `GET /health`, `GET /notes`, and `POST /notes`, fails fast when `DATABASE_URL` is missing, and shuts down cleanly on `SIGTERM`.

Create `package.json`:

```
{
  "name": "notes-api",
```

```
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "pg": "^8.16.0"
  }
}
```

Generate a lockfile (`npm install --package-lock-only`) so our Dockerfile can use deterministic `npm ci` builds.

Now create `server.js`:

```
import http from 'node:http'
import { Pool } from 'pg'

const port = Number(process.env.PORT || 3000)
if (!process.env.DATABASE_URL) {
  console.error('FATAL: DATABASE_URL is required')
  process.exit(1)
}

const pool = new Pool({ connectionString: process.env.DATABASE_URL, max: 10 })

async function init() {
  await pool.query(`
    CREATE TABLE IF NOT EXISTS notes (
      id SERIAL PRIMARY KEY,
      title TEXT NOT NULL,
      created_at TIMESTAMPTZ NOT NULL DEFAULT now()
    )
  `)
}

const server = http.createServer(async (req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`)
  try {
    if (req.method === 'GET' && url.pathname === '/health') {
      await pool.query('SELECT 1')
      res.writeHead(200, { 'content-type': 'application/json' })
      return res.end(JSON.stringify({ status: 'ok', user: process.getuid() }) + '\n')
    }
    if (req.method === 'GET' && url.pathname === '/notes') {
      const { rows } = await pool.query(
        'SELECT id, title, created_at FROM notes ORDER BY id DESC LIMIT 20',
      )
      res.writeHead(200, { 'content-type': 'application/json' })
      return res.end(JSON.stringify({ count: rows.length, notes: rows }, null, 2) + '\n')
    }
    if (req.method === 'POST' && url.pathname === '/notes') {
      let body = ''
      for await (const chunk of req) body += chunk
      const { title } = JSON.parse(body || '{}')
      if (!title || !title.trim()) {
        res.writeHead(400, { 'content-type': 'application/json' })
        return res.end(JSON.stringify({ error: 'title is required' }) + '\n')
      }
      const { rows } = await pool.query(
        'INSERT INTO notes (title) VALUES ($1) RETURNING id, title, created_at',
        [title.trim()],
      )
      res.writeHead(201, { 'content-type': 'application/json' })
      return res.end(JSON.stringify(rows[0], null, 2) + '\n')
```

```
    }
    res.writeHead(404, { 'content-type': 'application/json' })
    res.end(JSON.stringify({ error: 'not found' }) + '\n')
  } catch (err) {
    res.writeHead(500, { 'content-type': 'application/json' })
    res.end(JSON.stringify({ error: err.message }) + '\n')
  }
})

await init()
server.listen(port, '0.0.0.0', () => {
  console.log(`notes-api listening on :${port} (uid=${process.getuid()})`)
})

for (const sig of ['SIGTERM', 'SIGINT']) {
  process.on(sig, () => {
    console.log(`Received ${sig}, shutting down...`)
    server.close(async () => {
      await pool.end()
      process.exit(0)
    })
  })
}
```

Notice three container-friendly design choices in `server.js`:

1. **Bind `0.0.0.0`, not `127.0.0.1`.** Inside a container, `127.0.0.1` is the container's own loopback interface, unreachable from Docker's port publisher on the host. Always listen on `0.0.0.0` inside the container, and restrict host exposure in `-p 127.0.0.1:3040:3000`.
2. **Log to `stdout` / `stderr`.** Don't write log files to `/var/log/app.log` inside a container. Docker captures `stdout` and `stderr` automatically for `docker logs` and log shippers.
3. **Handle `SIGTERM`.** When you run `docker stop` or deploy a new release, Docker sends `SIGTERM` to PID 1 and waits 10 seconds before killing the process with `SIGKILL`. Handling `SIGTERM` lets in-flight HTTP requests finish and closes database pools cleanly.

Writing Dockerfiles, `.dockerignore`, and layer caching

A `Dockerfile` is a recipe of instructions. Every instruction that changes the filesystem (`FROM`, `RUN`, `COPY`, `ADD`) creates a new read-only **layer** stacked on top of the previous one.

The naive Dockerfile (and what goes wrong)

Here is the first Dockerfile most beginners write (`Dockerfile.fat`):

```
FROM node:22
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "server.js"]
```

It works, but it has four serious problems:

1. **`FROM node:22`** pulls the full Debian base image with compilers, Python, git, and hundreds of tools your runtime doesn't need.
2. **`COPY . .` before `RUN npm install`** busts the layer cache on every single code edit. Change one comment in `server.js` and Docker has to re-run `npm install` from scratch.
3. **No `.dockerignore`** means `docker build` sends your local `node_modules/`, `.git/`, and `.env` secrets to the Docker daemon on every build.
4. **Runs as `root`** (`uid=0`) by default.

Controlling the build context with `.dockerignore`

When you run `docker build .`, the `.` is the **build context** sent to BuildKit. Always add a `.dockerignore` file next to your `Dockerfile`:

```
node_modules
.git
.env
*.log
```

This keeps builds fast, prevents macOS-compiled native binaries in your local `node_modules` from overwriting Linux binaries inside the image, and ensures `.env` secrets never leak into an image layer.

Multi-stage production Dockerfile

Now compare `Dockerfile.fat` with a clean multi-stage production `Dockerfile`:

```
# Stage 1: install production dependencies cleanly
FROM node:22-alpine AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

# Stage 2: minimal runtime image
FROM node:22-alpine
WORKDIR /app
ENV NODE_ENV=production
COPY --from=deps /app/node_modules ./node_modules
COPY package.json server.js ./
USER node
EXPOSE 3000
HEALTHCHECK --interval=10s --timeout=3s --start-period=5s --retries=3 \
  CMD wget -qO- http://127.0.0.1:3000/health || exit 1
CMD ["node", "server.js"]
```

Look at why each line is in that order:

1. **`COPY package.json package-lock.json ./` comes before `RUN npm ci --omit=dev`.** Dependencies change rarely, while source code changes constantly.

2. **Stage 2 copies only `node_modules`, `package.json`, and `server.js`.** If you had a TypeScript or Vite build step in Stage 1 (`npm run build`), Stage 2 would copy only `dist/` and `node_modules/`, leaving `devDependencies`, `tsc`, and build caches behind in the discarded builder stage.
3. `USER node` switches to the built-in unprivileged user (`uid=1000`) before `CMD` runs.
4. `CMD ["node", "server.js"]` uses the JSON exec form (not shell form `CMD node server.js`). Shell form wraps your app in `/bin/sh -c`, which swallows `SIGTERM` signals and causes every `docker stop` to hang for 10 seconds before `SIGKILL`.

Dockerfile instructions that trip people up

Most Dockerfiles use fewer than ten instructions, yet four pairs cause almost all confusion:

Pair	Difference	Rule of thumb
`COPY` vs `ADD`	`COPY` copies local files. `ADD` also auto-extracts tarballs and fetches URLs	Always use `COPY` unless you explicitly want tar extraction
`ENV` vs `ARG`	`ARG` exists only at `docker build` time. `ENV` stays in the image at runtime	Use `ARG` for build flags (like `NODE_VERSION`), never for secrets
`RUN` vs `CMD`	`RUN` executes during `docker build` and commits a new layer. `CMD` sets the default command at `docker run`	Put installs and compilation in `RUN`, and your server start command in `CMD`
`ENTRYPOINT` vs `CMD`	`ENTRYPOINT` sets the fixed executable. `CMD` provides default arguments that `docker run <image> <args>` overrides	Use `CMD` for normal app containers, `ENTRYPOINT` for CLI tool images

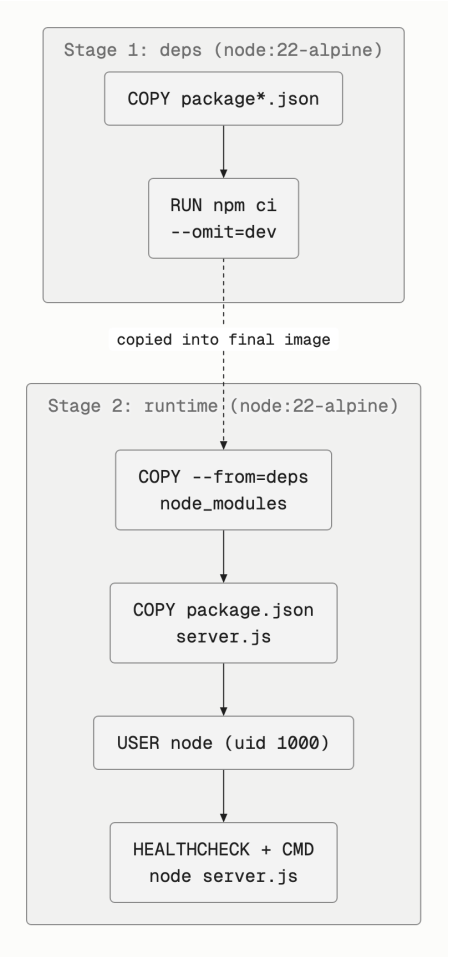
Watch how overriding `CMD` vs `--entrypoint` behaves on `alpine:3.21`:

```
docker run --rm alpine:3.21 echo "overriding default cmd"
docker run --rm --entrypoint /bin/sh alpine:3.21 -c 'echo "pid=$$ user=$(id -un)"'
```

```
overriding default cmd
pid=1 user=root
```

Notice `pid=1` in that output. Inside a Linux container, the process started by `CMD` or `ENTRYPOINT` runs as **PID 1**. Linux treats PID 1 specially: it ignores `SIGTERM` and `SIGINT` unless the process explicitly registers signal handlers (which we did in `server.js`), and it is responsible for reaping zombie child processes. If your app

spawns child processes or doesn't handle `SIGTERM`, pass `--init` to `docker run` (or `init: true` in `compose.yaml`) so Docker injects the tiny `tini` init process as PID 1.



Measuring real image sizes and layer caching

Let's build both images on the same Notes API and compare their sizes:

```
docker build -f Dockerfile.fat -t docker-handbook-notes:fat .
docker build -f Dockerfile -t docker-handbook-notes:prod .
docker images --filter reference='docker-handbook-notes' \
  --format 'table {{.Repository}}:{{.Tag}}\t{{.Size}}'
```

```
REPOSITORY:TAG                SIZE
docker-handbook-notes:prod    234MB
docker-handbook-notes:fat     1.61GB
```

From **1.61 GB** down to **234 MB**, nearly **7× smaller**, with zero changes to the application code.

Now test layer caching by adding a comment to `server.js` and rebuilding `docker-handbook-notes:prod`:

```
printf '\n// cache test comment\n' >> server.js
/usr/bin/time -p docker build -f Dockerfile -t docker-handbook-notes:prod .
```

```
#6 CACHED
#7 CACHED
#8 CACHED
#9 CACHED
#10 [stage-1 4/4] COPY package.json server.js ./
#10 DONE 0.2s
#11 naming to docker.io/library/docker-handbook-notes:prod done
real 0.77
```

0.77 seconds total! Steps 6 through 9 (including `npm ci --omit=dev`) were `CACHED` and skipped completely. Only the tiny 8 KB `COPY package.json server.js ./` layer was rebuilt.

Inspecting layers and verifying non-root execution

Use `docker history` to see what each layer contributed to your final image:

```
docker history --format 'table {{.CreatedBy}}\t{{.Size}}' docker-handbook-notes:prod | head -n 9
```

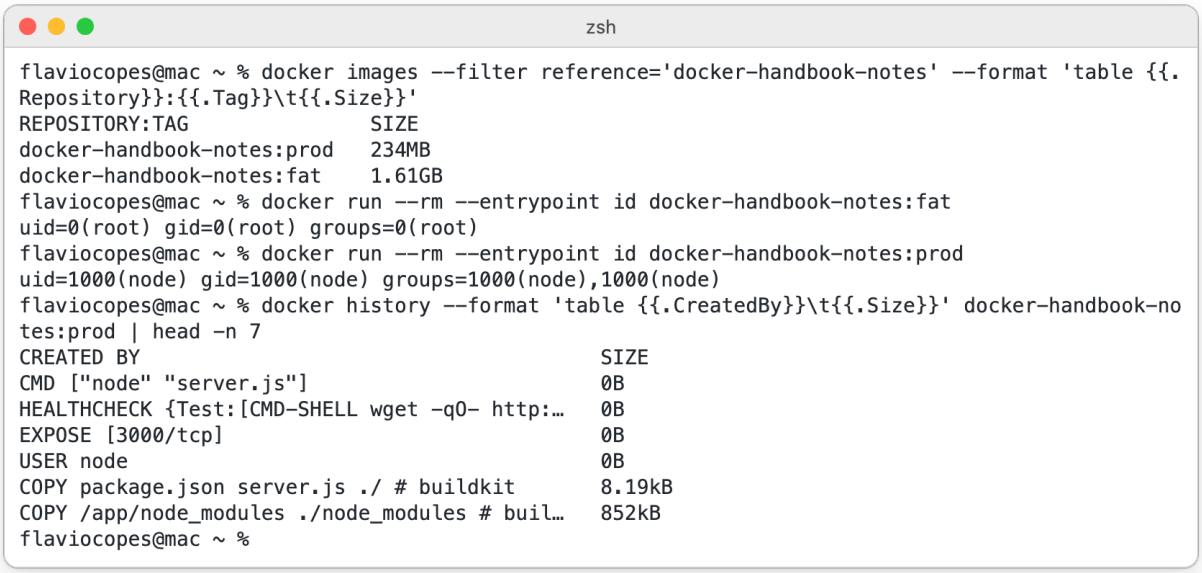
```
CREATED BY                                      SIZE
CMD ["node" "server.js"]                        0B
HEALTHCHECK {Test:[CMD-SHELL wget -qO- http…:   0B
EXPOSE [3000/tcp]                               0B
USER node                                       0B
COPY package.json server.js ./ # buildkit       8.19kB
COPY /app/node_modules ./node_modules # …buil   852kB
ENV NODE_ENV=production                         0B
WORKDIR /app                                    0B
```

And verify which OS user each image runs as by overriding the entrypoint with `id`:

```
docker run --rm --entrypoint id docker-handbook-notes:fat
docker run --rm --entrypoint id docker-handbook-notes:prod
```

```
uid=0(root) gid=0(root) groups=0(root)
uid=1000(node) gid=1000(node) groups=1000(node),1000(node)
```

If an attacker ever exploits a bug in your Node.js process inside `prod`, they land as unprivileged `uid=1000(node)` with no permission to install packages or overwrite system binaries.



Choosing a base image

Base image	Typical size	Pros	Trade-offs
`node:22` (Debian full)	~1.1–1.6 GB	Has `gcc`, `python3`, `git`, `curl` preinstalled	Huge image, large attack surface
`node:22-slim` (Debian slim)	~260 MB	Standard `glibc`, compatible with all native addons	Missing build tools (install in builder stage)
`node:22-alpine` (Alpine)	~180–235 MB	Smallest Node image, fast pulls	Uses `musl` libc; some prebuilt native npm binaries expect `glibc`
`gcr.io/distroless/nodejs22-debian12`	~150 MB	No shell, no package manager, minimal CVE surface	Harder to debug (`docker exec sh` won't exist)

Start with `-alpine` for pure JavaScript/TypeScript apps, and switch to `-slim` whenever a native package (like `sharp`, `canvas`, or `playwright`) expects `glibc`.

Multi-stage patterns for TypeScript and static frontends

Our Notes API ran plain JavaScript, so Stage 1 only installed production `node_modules`. In real projects you often have a compile step (`tsc`, Vite, Astro, or Next.js). Multi-stage builds shine even more there because your final runtime image never needs TypeScript, Tailwind, Vite, or `devDependencies`.

Here is a clean three-stage pattern for a TypeScript API:

```
# Stage 1: build TypeScript with all devDependencies
```

```
FROM node:22-alpine AS builder
WORKDIR /app
COPY package.json package-lock.json tsconfig.json ./
RUN npm ci
COPY src ./src
RUN npm run build

# Stage 2: install production-only dependencies
FROM node:22-alpine AS prod-deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

# Stage 3: minimal runtime image
FROM node:22-alpine
WORKDIR /app
ENV NODE_ENV=production
COPY --from=prod-deps /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY package.json ./
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

And for a static frontend (React, Vue, or Astro static output), the final stage doesn't even need Node.js! Build the static files in `node:22-alpine`, then copy `dist/` into `nginx:1.27-alpine`:

```
FROM node:22-alpine AS builder
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM nginx:1.27-alpine
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80
```

That turns a 600 MB Node.js build environment into a ~48 MB static Nginx image that boots in under 100 milliseconds.

Speeding up builds with BuildKit cache mounts

Layer caching skips `npm ci` completely when `package-lock.json` hasn't changed. What about the moment you add one new npm package? `package-lock.json` changes, so Docker invalidates the `RUN npm ci` layer and re-downloads every single package from npmjs.org.

BuildKit (enabled by default in Docker 29) lets you attach a persistent package cache to a `RUN` instruction without baking that cache into the final image:

```
FROM node:22-alpine AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN --mount=type=cache,target=/root/.npm \
    npm ci --omit=dev
```

Because `/root/.npm` is a BuildKit cache mount, npm reuses tarballs from earlier builds and only downloads the one newly added package. The cache directory stays outside the image layers, so your final image stays just as small. The same pattern works for `pip` (`/root/.cache/pip`), `apt` (`/var/cache/apt`), `cargo` (`/usr/local/cargo/registry`), and Go (`/go/pkg/mod`).

Data persistence: volumes vs bind mounts

Earlier we proved that a container's writable layer vanishes when the container is removed. To keep data alive across restarts, upgrades, and rebuilds, Docker gives you three mount types:

Mount type	Managed by	Best for	Example
Named volume	Docker (`/var/lib/docker/volumes/`)	Databases (`postgres`, `mysql`, `redis`), persistent uploads	`pgdata:/var/lib/postgresql/data`
Bind mount	Your host filesystem	Local development source code, config files (`nginx.conf`)	`./src:/app/src:ro`
`tmpfs` mount	Host RAM (Linux)	Temporary secrets, scratch caches never written to disk	`--tmpfs /tmp:rw,size=64m`

Two practical rules save hours of debugging:

- **Use named volumes for databases.** On macOS, bind-mounting a PostgreSQL data directory across the VM boundary is both slower and prone to file-locking quirks. Named volumes live natively inside the Linux VM filesystem and run at full speed.
- **Mount read-only (`:ro`) whenever a container doesn't need to write.** Configuration files like `./nginx.conf:/etc/nginx/conf.d/default.conf:ro` should always be read-only.

Verifying a read-only bind mount

Let's mount a local folder containing `index.html` into `nginx:1.27-alpine` with `:ro` and try writing to it from inside the container:

```
mkdir -p /tmp/docb/static
echo "<h1>Served from host bind mount</h1>" > /tmp/docb/static/index.html
docker run -d --name docker-handbook-ro -p 127.0.0.1:8089:80 \
  -v /tmp/docb/static:/usr/share/nginx/html:ro nginx:1.27-alpine
curl -s http://127.0.0.1:8089
docker exec docker-handbook-ro sh -c 'touch /usr/share/nginx/html/hack.txt'
```

```
<h1>Served from host bind mount</h1>
touch: /usr/share/nginx/html/hack.txt: Read-only file system
```

Nginx served our host file immediately, and the kernel blocked the write inside the container even though the process ran as root. Whenever you mount TLS certificates, static assets, or config files into a container, append `:ro`.

Backing up a named database volume

Never copy raw PostgreSQL files straight out of `/var/lib/docker/volumes/` while the database is running, because you can catch half-written pages. Instead, stream a consistent logical dump directly from the running `db` container to the host:

```
docker compose -p docker-handbook exec -T db \
  pg_dump -U notes notes | gzip > backup-$(date +%Y-%m-%d).sql.gz
```

Notice the `-T` flag on `exec`: it disables pseudo-TTY allocation so binary `gzip` output isn't corrupted by terminal line endings, and it lets the command run cleanly inside cron jobs and CI pipelines.

To restore that dump into a fresh database container:

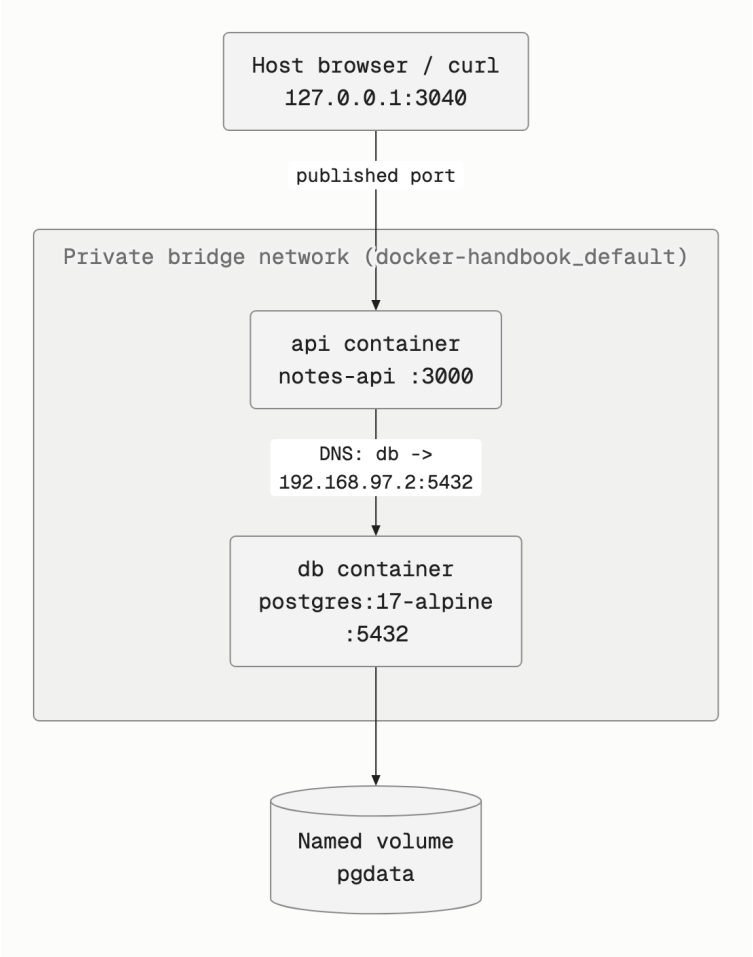
```
gunzip -c backup-2026-11-28.sql.gz | \
  docker compose -p docker-handbook exec -T db psql -U notes -d notes
```

Container networking and DNS service discovery

Every container needs a way to talk to sibling services (like `api` talking to `db`) without hardcoding fragile IP addresses that change on every restart.

Docker solves this with **user-defined bridge networks**. When two containers join the same user-defined network (which Docker Compose creates automatically for every project):

1. Docker runs an embedded DNS server inside the network (`127.0.0.11`).
2. Containers reach each other by **service name** (`postgres://notes:notes-dev-password@db:5432/notes`).
3. Ports that are **not** listed under `ports:` stay completely private to that bridge network. Notice in our Compose file below that `db` has no `ports:` mapping at all: `api` reaches `db:5432` privately, while nothing on the host or public internet can touch PostgreSQL directly.



Network isolation between tiers

Because Docker scopes DNS and packet routing per bridge network, a container attached to a different network cannot even resolve `db`, let alone open a TCP connection to it.

Let's verify that by creating a separate `docker-handbook-front` network and asking `nslookup` for `db`:

```
docker network create docker-handbook-front
docker run --rm --network docker-handbook-front alpine:3.21 nslookup db
```

```
Server:     127.0.0.11
Address:    127.0.0.11:53

** server can't find db: NXDOMAIN
```

In larger Compose stacks, you can declare two networks (`public` and `internal: true backend`): attach your reverse proxy and API to `public`, and attach your API and database to `backend`. The reverse proxy can reach the API, the API can reach PostgreSQL, and the database has zero direct network path to the public tier or the outside internet.

What if a container during local development needs to reach a service running directly on your Mac (for example, a local Ollama or Stripe CLI listener)? Use the special hostname `host.docker.internal`, which Docker Desktop and OrbStack resolve to your host machine automatically. On Linux, add `extra_hosts: ["host.docker.internal:host-gateway"]` to your service in `compose.yaml`.

Multi-container apps with Docker Compose

Typing long `docker run` commands with networks, volumes, health checks, and environment variables gets error-prone fast. **Docker Compose** puts your entire local or single-server stack in one declarative `compose.yaml` file.

Here is the `compose.yaml` for our Notes API + PostgreSQL 17 stack:

```
services:
  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_USER: notes
      POSTGRES_PASSWORD: notes-dev-password
      POSTGRES_DB: notes
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U notes -d notes"]
      interval: 5s
      timeout: 3s
      retries: 10
    restart: unless-stopped

  api:
    build: .
    environment:
      PORT: "3000"
      DATABASE_URL: postgres://notes:notes-dev-password@db:5432/notes
    ports:
      - "127.0.0.1:3040:3000"
    depends_on:
      db:
        condition: service_healthy
    restart: unless-stopped

volumes:
  pgdata:
```

Pay special attention to `depends_on` with `condition: service_healthy`. A plain `depends_on: [db]` only waits for the `db` container to *start*, not for PostgreSQL to finish initializing and accept TCP connections. Combining `healthcheck` (`pg_isready`) with `condition: service_healthy` guarantees `api` only boots once PostgreSQL is actually ready.

Starting the stack and testing end to end

Bring up the entire stack with one command:

```
docker compose -p docker-handbook up -d --build
```

Docker Compose builds `api`, creates the private `docker-handbook_default` bridge network and `docker-handbook_pgdata` named volume, starts `db`, waits until `db` reports `Healthy`, and then starts `api`.

Check service status:

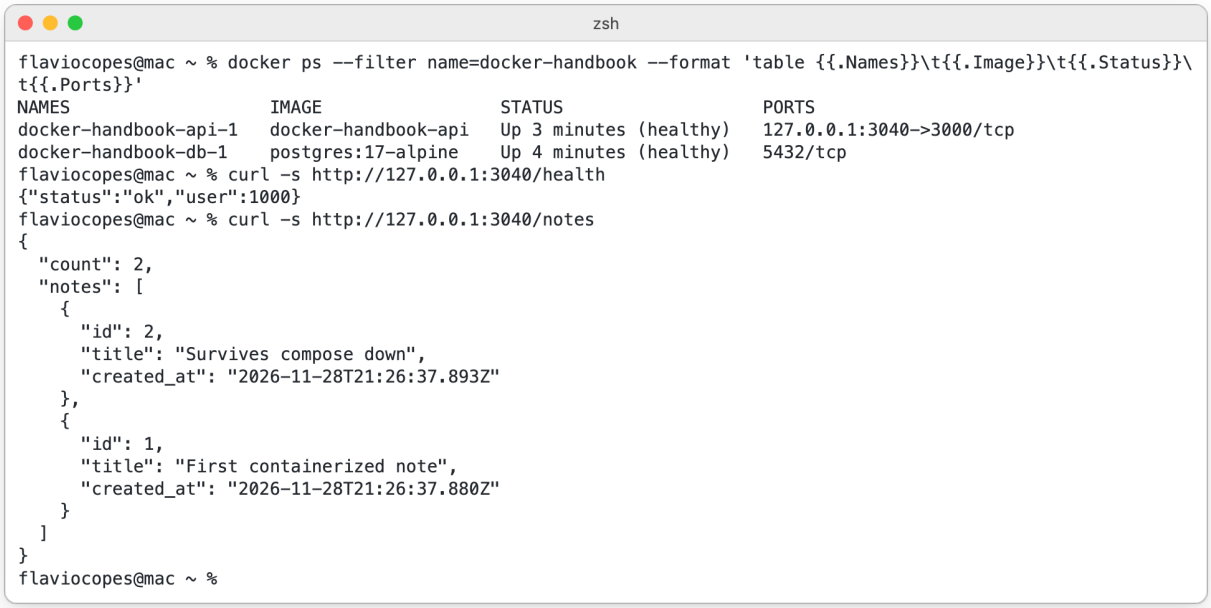
```
docker compose -p docker-handbook ps
```

```
NAME                    IMAGE                SERVICE   STATUS
    ↪ PORTS
docker-handbook-api-1   docker-handbook-api   api       Up Less than a second (health: starting)
    ↪ 127.0.0.1:3040->3000/tcp
docker-handbook-db-1    postgres:17-alpine    db        Up 6 seconds (healthy)
    ↪ 5432/tcp
```

Verify the health endpoint (note `user: 1000`, confirming non-root `node`), insert two notes, and read them back:

```
curl -s http://127.0.0.1:3040/health
curl -s -X POST http://127.0.0.1:3040/notes \
  -H 'content-type: application/json' \
  -d '{"title":"First containerized note"}'
curl -s -X POST http://127.0.0.1:3040/notes \
  -H 'content-type: application/json' \
  -d '{"title":"Survives compose down"}'
curl -s http://127.0.0.1:3040/notes
```

```
{"status":"ok","user":1000}
{
  "count": 2,
  "notes": [
    {
      "id": 2,
      "title": "Survives compose down",
      "created_at": "2026-11-28T21:26:37.893Z"
    },
    {
      "id": 1,
      "title": "First containerized note",
      "created_at": "2026-11-28T21:26:37.880Z"
    }
  ]
}
```



Proving data persistence and DNS discovery

Now tear down every container and network with `docker compose down` (without `-v`!), bring them right back up, and query `/notes` again:

```
docker compose -p docker-handbook down
docker compose -p docker-handbook up -d
curl -s http://127.0.0.1:3040/notes
```


```
{
  "count": 2,
  "notes": [
    {
      "id": 2,
      "title": "Survives compose down",
      "created_at": "2026-11-28T21:26:37.893Z"
    },
    {
      "id": 1,
      "title": "First containerized note",
      "created_at": "2026-11-28T21:26:37.880Z"
    }
  ]
}
```


Both notes survived container destruction and recreation because PostgreSQL stored them in the `pgdata` named volume. (Only `docker compose down -v` deletes named volumes, so be careful with `-v`!)

We can also verify Docker's embedded DNS resolution right from inside the `api` container:

```
docker network ls --filter name=docker-handbook --format 'table {{.Name}}\t{{.Driver}}'
```

```
docker compose -p docker-handbook exec -T api \
  node -e "import('node:dns/promises').then(d => d.lookup('db')).then(console.log)"


NAME                        DRIVER
docker-handbook_default   bridge
{ address: '192.168.97.2', family: 4 }
```

A fast development loop with Compose

Rebuilding an image on every file save slows down local development. Keep your production `compose.yaml` clean, and add a `compose.override.yaml` (which Compose merges automatically during local dev) or use `develop.watch`:

```
services:
  api:
    command: ["node", "--watch", "server.js"]
    volumes:
      - ./server.js:/app/server.js:ro
```

With Node 22's built-in `--watch` flag and a read-only bind mount for `server.js`, saving a file on your Mac restarts the server inside the container in milliseconds while keeping `node_modules` inside the Linux container where it belongs.

Environment files, interpolation, and Compose profiles

Hardcoding passwords like `notes-dev-password` in `compose.yaml` is fine for a throwaway tutorial, never for a real repo. Docker Compose automatically reads a `.env` file in the same directory and supports shell-style defaults and required-variable checks:

```
services:
  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_USER: ${POSTGRES_USER:-notes}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in .env}
      POSTGRES_DB: ${POSTGRES_DB:-notes}
  api:
    image: ghcr.io/youruser/notes-api:${APP_TAG:-local}
    ports:
      - "127.0.0.1:${APP_PORT:-3040}:3000"
```

Two operators to remember:

- `${VAR:-default}` uses `default` when `VAR` is unset or empty.
- `${VAR:?error message}` aborts `docker compose up` immediately with a clear error message if a required secret is missing.

Commit a `.env.example` with placeholder values, add `.env` to both `.gitignore` and `.dockerignore`, and run `docker compose config` whenever you want to inspect

the fully interpolated YAML before starting containers.

For optional developer tools (like a database GUI or a local mail catcher) that shouldn't start on every `docker compose up`, attach a `profiles:` key:

```
mailpit:
  image: axllent/mailpit:v1.21
  profiles: ["debug"]
  ports:
    - "127.0.0.1:8025:8025"
```

Plain `docker compose up -d` skips `mailpit`. Running `docker compose --profile debug up -d` brings it up only when you need it.

Restart policies compared

In `compose.yaml` we set `restart: unless-stopped` on both `db` and `api`. Docker supports four restart policies, and picking the right one prevents both silent outages and infinite crash loops on one-off jobs:

Policy	Restarts on crash?	Restarts after server reboot?	Best for
`no` (default)	No	No	One-off migration scripts, batch jobs, local experiments
`on-failure:5`	Up to 5 times on non-zero exit	Only if it was running and exited non-zero	Background workers that should stop retrying after repeated crashes
`always`	Yes	Always (even if you manually stopped it)	Critical daemons that must always be up
`unless-stopped`	Yes	Yes, unless you explicitly ran `docker stop`	Web servers, APIs, databases (the best production default)

Use `unless-stopped` for long-lived services so a VPS reboot brings your stack right back up, while still letting you stop a container manually during maintenance without Docker immediately restarting it behind your back.

Everyday Docker Compose CLI reference

Once you move from single `docker run` commands to `compose.yaml`, these ten commands cover almost every daily workflow:

Command	What it does
`docker compose up -d --build`	Build any changed Dockerfiles and start all services in the background
`docker compose ps`	Show container status, healthcheck state, and published ports
`docker compose logs -f --tail 50 api`	Stream the last 50 log lines for the `api` service
`docker compose exec api sh`	Open a shell inside the running `api` container
`docker compose exec -T db psql -U notes`	Pipe SQL or run non-interactive commands inside `db`
`docker compose run --rm api node migrate.js`	Run a one-off command in a fresh container and remove it on exit
`docker compose up -d --no-deps api`	Recreate only `api` without touching `db`
`docker compose config`	Print the fully merged, `.env`-interpolated YAML config
`docker compose down`	Stop and remove containers and networks (keeps named volumes safe)
`docker compose down -v`	Stop and remove containers, networks, **and** named volumes (destructive)

Debugging runtime failures and managing disk space

When a container crashes or enters a restart loop, don't guess. Three commands tell you almost everything: `docker ps -a`, `docker logs`, and `docker inspect`.

Exit code 1: application startup error

Let's start our `notes-api` container without `DATABASE_URL` and inspect what happened:

```
docker run --name docker-handbook-fail1 docker-handbook-notes:prod
docker inspect --format 'Status={{.State.Status}} ExitCode={{.State.ExitCode}} OOMKilled={{.State.
    ↪ OOMKilled}}' docker-handbook-fail1


FATAL: DATABASE_URL is required
Status=exited ExitCode=1 OOMKilled=false
```

Clean, immediate diagnosis: the container exited with code `1`, it was not killed for memory, and `stderr` told us the exact missing environment variable.

Exit code 137: Out-of-Memory (`OOMKilled`)

Now let's cap a container to 32 MB of RAM and allocate memory until the Linux OOM killer terminates it:

```
docker run --name docker-handbook-oom --memory 32m --memory-swap 32m \
  node:22-alpine node -e "const a=[]; while(true) a.push(Buffer.alloc(1024*1024))"
docker inspect --format 'Status={{.State.Status}} ExitCode={{.State.ExitCode}} OOMKilled={{.State.
    ↪ OOMKilled}}' docker-handbook-oom


Status=exited ExitCode=137 OOMKilled=true
```

When a production container mysteriously dies with **no stack trace in `docker logs`**, check `ExitCode` and `OOMKilled` first. `137` (`128 + 9`, `SIGKILL`) with `OOMKilled=true` means the kernel killed the process for exceeding its cgroup memory limit.

Common container exit codes

Exit code	Meaning	First thing to check
`0`	Clean exit	Normal for one-shot jobs; unexpected for a server (did it daemonize instead of staying in foreground?)
`1`	Application error	`docker logs <name>` for stack trace or missing env var
`126`	Command cannot execute	File permissions (`chmod +x`) or wrong CPU architecture (`amd64` vs `arm64`)
`127`	Command not found	Typo in `CMD`/`ENTRYPOINT` or binary not installed in Alpine/slim
`137`	`SIGKILL` (`128 + 9`)	Check `OOMKilled` in `docker inspect`, or app ignored `SIGTERM` during `docker stop`
`143`	`SIGTERM` (`128 + 15`)	Graceful stop signal received (`docker stop`)

Resource limits, log rotation, and container hardening

Let's check the live resource usage of our running Notes API and PostgreSQL 17 containers:

```
docker stats --no-stream \
  --format 'table {{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}\t{{.PIDs}}' \
  docker-handbook-api-1 docker-handbook-db-1


NAME                    CPU %     MEM USAGE / LIMIT     PIDS
docker-handbook-api-1   0.00%     16.25MiB / 15.66GiB   11
docker-handbook-db-1    0.44%     30MiB / 15.66GiB      12
```

Notice the right side of `MEM USAGE / LIMIT`: **15.66 GiB**! By default, Docker puts no memory or CPU cap on a container, which means a single memory leak in a background worker can eat every byte of RAM on a VPS and trigger the Linux OOM killer on your database.

In production `compose.yaml`, always set explicit memory limits, log rotation, and container hardening options:

```
services:
  api:
    image: ghcr.io/youruser/notes-api:1.0.0
    init: true
    read_only: true
    tmpfs:
      - /tmp:size=64m
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true
    mem_limit: 256m
    cpus: 1.0
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
```

Every line there earns its keep:

- `read_only: true` + `tmpfs: [/tmp]` stops an attacker from dropping scripts or binaries onto the container filesystem.
- `cap_drop: [ALL]` and `no-new-privileges:true` strip Linux kernel capabilities a normal Node.js HTTP server never needs.
- `mem_limit: 256m` and `cpus: 1.0` keep one misbehaving container from starving the rest of the box.
- `max-size: "10m"` and `max-file: "3"` cap Docker's JSON log files at 30 MB per container so verbose errors never fill the server disk.

Common Docker pitfalls and how to fix them

Even experienced developers hit the same six Docker issues over and over. Here is how to recognize and fix each one in under a minute:

1. **`ECONNREFUSED 127.0.0.1:5432` from your app container.** Inside a container, `127.0.0.1` is the container itself, not your Mac and not the Postgres container. Point `DATABASE_URL` to the Compose service name (`postgres://...@db:5432/notes`), never `localhost`.
2. **`bind: address already in use` on `docker compose up`.** Another process on your host is already listening on the published host port. Run `lsof -i :3040` on macOS or `ss -tulpn | grep 3040` on Linux, stop the conflicting process, or map a different host port (`127.0.0.1:3041:3000`).
3. **`exec format error` when deploying a Mac-built image to Linux.** Apple Silicon builds `linux/arm64` images by default, while most x86_64 VPS servers require `linux/amd64`. Build with `docker buildx build --platform linux/amd64` or build on a native `ubuntu-latest` GitHub Actions runner.

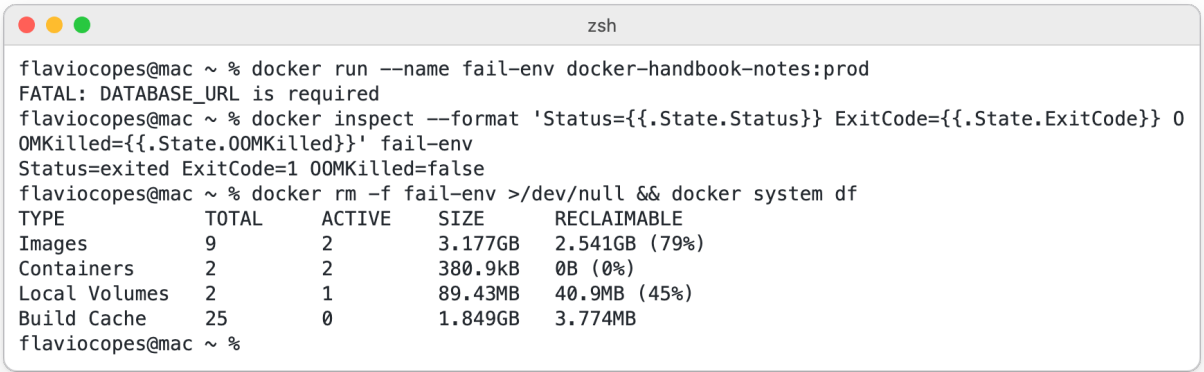
4. **`node_modules` vanishes when you bind-mount `.:/app` in development.** A bind mount replaces the entire `/app` directory inside the container with your host directory. Either mount only source folders (`./src:/app/src:ro`, `./server.js:/app/server.js:ro`) or add an anonymous volume (`- /app/node_modules`) to shield the container's Linux `node_modules`.
5. **Every `docker stop` takes exactly 10 seconds.** Your `Dockerfile` used shell-form `CMD node server.js` instead of exec-form `CMD ["node", "server.js"]`, or your app didn't register a `SIGTERM` handler. Docker waits 10 seconds for a clean exit before sending `SIGKILL`.
6. **File permission errors (`EACCES`) on mounted volumes.** On Linux hosts, if your container runs as `USER node` (`uid=1000`) and a mounted directory is owned by `root:root`, writes fail. Set ownership on the host directory (`chown -R 1000:1000 ./uploads`) or use a Docker named volume initialized with the right user.

Inspecting and reclaiming Docker disk space

Images, stopped containers, dangling volumes, and BuildKit cache grow quietly over time. Check your usage with `docker system df`:

```
docker system df


TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          9         2         3.177GB   2.541GB (79%)
Containers      2         2         380.9kB   0B (0%)
Local Volumes   2         1         89.03MB   40.9MB (45%)
Build Cache     25        0         1.849GB   3.774MB
```



Just building our test images left **2.54 GB** of reclaimable images and **1.85 GB** of build cache. Safe cleanup commands:

- `docker container prune` removes stopped containers.
- `docker image prune` removes untagged dangling images (add `-a` to remove all unused images).
- `docker builder prune` clears BuildKit cache.
- `docker system prune` cleans stopped containers, unused networks, dangling

images, and build cache (it does **not** touch named volumes unless you pass `--volumes`).

Tagging, pushing, and shipping to production

Once your image builds cleanly and passes local tests, shipping it to a server follows the same flow every time: tag, push to a registry, pull on the server, and roll the container.

Tagging with immutable versions

Tag each build with the Git commit SHA (and optionally a semantic version) instead of overwriting `:latest`:

```
GIT_SHA=$(git rev-parse --short HEAD)
docker build -t ghcr.io/youruser/notes-api:$GIT_SHA -t ghcr.io/youruser/notes-api:1.0.0 .
echo "$GITHUB_TOKEN" | docker login ghcr.io -u youruser --password-stdin
docker push ghcr.io/youruser/notes-api:$GIT_SHA
docker push ghcr.io/youruser/notes-api:1.0.0
```

Tagging with `$GIT_SHA` makes rollbacks trivial: if release `a1b2c3d` misbehaves, update `image:` back to the previous SHA and run `docker compose up -d`.

Building for `linux/amd64` from Apple Silicon Macs

If you develop on an Apple Silicon Mac (`linux/arm64`) and deploy to an x86_64 Linux VPS (`linux/amd64`), a plain `docker build` produces an ARM64 image that will fail on the server with `exec format error`.

Use `docker buildx build` with `--platform`:

```
docker buildx build \
  --platform linux/amd64,linux/arm64 \
  -t ghcr.io/youruser/notes-api:1.0.0 \
  --push .
```

Better yet, build and push your production images inside GitHub Actions (which runs native `linux/amd64` runners) so your laptop never has to cross-compile under QEMU emulation.

Secrets hygiene in Docker

Never put API keys, database passwords, or private tokens in `ENV` or `ARG` inside a `Dockerfile`. Both `ENV` and `ARG` are baked into the image's JSON metadata and readable by anyone who runs `docker history` or `docker inspect`!

Instead:

- Pass runtime secrets via `env_file` (with `chmod 600` on the server, never committed to Git) or Docker Compose `secrets:` (mounted as files under `/run/secrets/`).

- If you need a private npm token during `docker build`, use BuildKit secret mounts (`RUN --mount=type=secret,id=npmrc,target=/root/.npmrc npm ci`) so the token never lands in any image layer.

Building and pushing images in GitHub Actions

Building production images on a laptop works for side projects, but team projects should build and push images in CI so every image is traceable to a Git commit and built on native `linux/amd64` runners.

Here is a minimal GitHub Actions job that builds your multi-stage `Dockerfile`, reuses BuildKit layer cache inside GitHub Actions (`type=gha`), and pushes to GitHub Container Registry (`ghcr.io`):

```
name: publish-image
on:
  push:
    branches: [main]

permissions:
  contents: read
  packages: write

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: docker/setup-buildx-action@v3
      - uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: |
            ghcr.io/${{ github.repository }}:sha-${{ github.sha }}
            ghcr.io/${{ github.repository }}:main
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Notice there are no manual Docker Hub passwords to rotate: `GITHUB_TOKEN` with `packages: write` authenticates straight to `ghcr.io`.

Putting Caddy in front of Compose for automatic HTTPS

When you deploy `compose.yaml` to a Linux VPS, you want HTTPS on ports `80` and `443` without hand-managing TLS certificates. Adding a `caddy:2.9-alpine` service

lets Caddy terminate TLS automatically and proxy traffic to `api:3000` over the private bridge network:

```
services:
  caddy:
    image: caddy:2.9-alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile:ro
      - caddy_data:/data
      - caddy_config:/config
    depends_on:
      api:
        condition: service_healthy
    restart: unless-stopped

  api:
    image: ghcr.io/youruser/notes-api:${APP_TAG}
    expose:
      - "3000"
    environment:
      PORT: "3000"
      DATABASE_URL: postgres://notes:${POSTGRES_PASSWORD}@db:5432/notes
    depends_on:
      db:
        condition: service_healthy
    restart: unless-stopped
```

With a three-line `Caddyfile`:

```
notes.example.com {
    reverse_proxy api:3000
}
```

Notice that `api` no longer has a `ports:` block at all, only `expose: ["3000"]`. Neither `api` nor `db` is reachable directly from the internet: every public request enters through Caddy over HTTPS and flows over Docker's private bridge network.

Keeping base images patched and scanning for CVEs

Even if your application code never changes, the Linux packages inside `node:22-alpine` or `postgres:17-alpine` receive security patches every few weeks. An image you built six months ago still contains the old OpenSSL and glibc/musl libraries from the day you built it.

Three low-effort habits keep your container supply chain clean:

1. **Rebuild on a schedule.** Add a weekly `schedule:` trigger to your GitHub Actions build workflow (or use Dependabot/Renovate for `docker` ecosystems) so base image patches flow into production automatically.

2. **Scan images before pushing.** Run a vulnerability scanner like Trivy (`trivy image --severity HIGH,CRITICAL ghcr.io/youruser/notes-api:1.0.0`) or `docker scout quickview` in CI to catch critical OS or npm CVEs before they reach your server.
3. **Keep images minimal.** Every package you leave out of the final stage (compilers, `curl`, `git`, `devDependencies`) is one fewer package that can ever appear in a CVE report. That is another reason our multi-stage `234 MB` Alpine image beats the `1.61 GB` Debian image by a mile.

Deploying Docker Compose to a Linux VPS

For most small and medium web applications, you don't need Kubernetes. A single Linux VPS running Caddy (for automatic HTTPS) in front of Docker Compose handles thousands of requests per second with a fraction of the operational complexity.

A clean production deployment flow looks like this:

1. CI builds the image on merge to `main`, tags it with the Git SHA (`ghcr.io/youruser/notes-api:a1b2c3d`), and pushes it to GitHub Container Registry.
2. On the VPS, update `APP_TAG=a1b2c3d` in `.env` and pull the new image:

```
docker compose pull api
```

3. Run database migrations as a one-shot container **before** swapping the running app container:

```
docker compose run --rm api node migrate.js
```

4. Recreate only the `api` container (leaving `db` untouched):

```
docker compose up -d --no-deps api
```

5. Verify `/health` returns `200 OK` and check `docker compose logs --tail 30 api`. If anything looks wrong, change `APP_TAG` back to the previous SHA and run `docker compose up -d --no-deps api` to roll back in seconds.

Stay on Docker Compose until you genuinely need multi-node scheduling, automated cross-region failover, or dozens of independent engineering teams deploying microservices to a shared cluster. Only then does Kubernetes pay for its complexity.

Docker Compose vs PaaS vs Kubernetes

Dimension	Docker Compose on a VPS	Managed PaaS (Fly.io, Railway, Render)	Kubernetes (EKS, GKE, k3s)
Best team size	Solo developer to ~10 engineers	Solo developer to ~30 engineers	Multiple platform and product teams
Operational effort	Low (one Linux host + Caddy + cron backups)	Lowest (managed deploys, TLS, logs)	High (control plane, upgrades, RBAC, ingress)
Cost predictability	Flat monthly VPS bill ($5–$40/mo)	Usage-based, grows with scale	Cluster control plane + worker nodes
Multi-node failover	Manual (or move to Swarm / managed DB)	Built-in	Built-in
Local parity	Exact same `compose.yaml` locally and in prod	Same `Dockerfile`, different orchestration	Needs local cluster (`kind`, `k3d`)

Most projects start fastest and stay cheapest on either a managed PaaS (using the same multi-stage `Dockerfile` you wrote in this book) or a single Linux VPS running Docker Compose behind Caddy. Because your app is packaged as a standard OCI container image, moving from one to the other later requires zero code changes.

Production readiness checklist

Before deploying a Docker Compose stack to a Linux server, check every item on this list:

- **Pinned base images:** No `:latest` tags in `Dockerfile` or `compose.yaml`.
- **Multi-stage + `.dockerignore`:** Small runtime image (`-alpine` or `-slim`), zero `.git`, `.env`, or dev dependencies inside.
- **Non-root `USER`:** Application runs as an unprivileged UID (`USER node`), verified with `docker run --rm --entrypoint id`.
- **Health checks and restart policies:** Both database and app define `healthcheck`, `depends_on: condition: service_healthy`, and `restart: unless-stopped`.
- **Private networking:** Database ports stay private on the bridge network, and app ports behind a reverse proxy bind `127.0.0.1:<port>:<port>`.
- **Resource limits and log rotation:** Configure `mem_limit`, `cpus`, and `json-file` log rotation (`max-size: "10m"`, `max-file: "3"`) so one runaway container or verbose log stream cannot crash the host or fill the disk.
- **Graceful shutdown:** Exec-form `CMD ["node", "server.js"]` with a `SIGTERM` handler that closes the HTTP server and database pool.
- **Volume backups:** Named volumes are backed up on a schedule (for Postgres,

`docker compose exec -T db pg_dump -U notes notes | gzip > backup.sql.gz`) and restore-tested.