Docker Course

Flavio Copes

Updated August 3, 2026

Contents

Learn containers by packaging a Node.js API, connecting it to PostgreSQL, building a Compose workflow, and preparing a secure production image.

What you'll learn: Build, run, inspect, persist, connect, and ship a containerized Node.js application with Docker and Docker Compose.

Container foundations

Understand images and containers, install Docker, and inspect a running workload.

What a container is

Build a useful mental model of images, containers, isolation, and the host kernel before running Docker commands.

A container is an isolated process with its own view of files, environment variables, users, and networking. It is not a small virtual machine.

The image supplies the filesystem and startup instructions. Docker creates a writable container from that image and asks the host operating system to isolate the process. Containers start quickly because they share the host kernel instead of booting another operating system.

The startup command becomes the container's main process. When that process exits, the container stops, even if files remain in its writable layer. A container is not a background machine that keeps running independently of its application.

Isolation is useful, but it is not a security boundary you can ignore. The process still shares a kernel with the host. Run with the fewest privileges you can, keep durable data outside the writable layer, and treat the image as the repeatable input from which disposable containers are created.

Write down the application, runtime, system packages, and startup command for a small Node.js API. Those are the ingredients we will put in an image.

Install and verify Docker

Install a current Docker toolchain for your operating system and verify the client can reach the Docker engine.

Install Docker Desktop on macOS or Windows, or Docker Engine from Docker's repository on Linux. Use the current official instructions for your operating system.

The `docker` command is a client. Most commands ask a long-running Docker engine to create images, containers, networks, and volumes. A working client with an unreachable engine is not a working installation.

```
docker version
docker info
docker run --rm hello-world
```

These commands test different parts of the path. `docker version` proves that the client can negotiate with the server. `docker info` asks the selected engine for its configuration. `hello-world` also exercises image lookup, download, container creation, process startup, logs, and cleanup.

If the client exists but the server section is missing, check `docker context show` before reinstalling anything. A context points the client at a particular engine. On Docker Desktop, also verify that the application is running. This distinction prevents a daemon connection problem from being mistaken for a broken CLI.

Run all three commands. Read the client and server sections of `docker version` instead of checking only that the executable exists.

Run and inspect a container

Start a web server container, publish its port, inspect its state, read its logs, and remove it deliberately.

Run a known image before building your own. This separates Docker basics from application build problems.

`--name` gives the container a stable local name, `-d` runs it in the background, and `-p 8080:80` forwards host port 8080 to container port 80. Port publishing is explicit: a process listening inside a container is not automatically reachable from the host.

```
docker run --name course-web -d -p 8080:80 nginx:alpine
docker ps
docker logs course-web
docker inspect course-web
docker rm -f course-web
```

The lifecycle has separate steps even when `docker run` combines them: create a container, start its main process, stop it, and remove it. A stopped container can be inspected and started again, but its writable layer remains tied to that particular container. Removing it does not remove the image or an independently managed volume.

Use `docker inspect` for configured and runtime state, and `docker logs` for what the main process wrote to standard output and standard error. If the main process exits, start with its exit code and logs. An interactive shell is secondary evidence and is unavailable when the container is stopped.

Open http://localhost:8080, find the image name and port mapping in `docker ps`, then remove the container.

Connect images, containers, and registries

Trace an image from a registry to the local cache and into one or more containers without confusing their lifecycles.

An image name such as `nginx:alpine` combines a repository and tag. Docker pulls missing image layers from a registry, normally Docker Hub unless the name says otherwise.

A tag is a readable pointer, not proof that content never changes. An image digest identifies exact content. Multiple containers can share the same read-only image layers while each gets its own writable layer.

```
docker pull node:22-alpine
docker image ls node
docker run --rm node:22-alpine node --version
```

Tags make images convenient for humans, but deployment records should include the digest returned by the registry. A maintainer can move a tag to different content; a digest lets another machine pull the exact bytes you tested. This matters most for broad tags such as `latest` or `22-alpine`.

An image name can also resolve to a multi-platform manifest. Docker selects an architecture-specific image for the current machine. Before production, use `docker image inspect` or the registry UI to confirm both the digest and supported platform rather than assuming an image built on a laptop will run everywhere.

Pull the image, run two short-lived containers from it, and compare the image ID shown by `docker`

`image ls`.

Check your understanding: container foundations

Check the container model, Docker installation, images, containers, and the commands used to inspect a running workload.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build images

Create the Notes API, write its Dockerfile, and use build context and caching well.

Create the Notes API

Create the small Node.js HTTP service that will become the running project for the rest of the Docker course.

The course project is a tiny Notes API. Keeping the application small lets us see which behavior comes from Node.js and which comes from Docker.

The server reads its port from the environment and binds to `0.0.0.0`. Binding only to localhost inside a container would make the process unreachable through Docker's virtual network.

```
import { createServer } from 'node:http'

const notes = [{ id: 1, text: 'Learn containers' }]
const port = Number(process.env.PORT ?? 3000)

createServer((request, response) => {
  response.setHeader('content-type', 'application/json')
  response.end(JSON.stringify({ notes }))
}).listen(port, '0.0.0.0')
```

Keep the first version observable and easy to stop. Reject an invalid port during startup, log fatal startup errors to standard error, and let the process respond to termination instead of hiding work in detached child processes. Docker can only judge the lifecycle of the main process it started.

The host binding is part of the container contract. `0.0.0.0` accepts traffic arriving on the container's network interface; `127.0.0.1` would accept traffic only from inside that container. Test the host process first, then use the same `curl` request after containerizing it. The changed boundary is then the only new variable.

Create `server.js`, add `"type": "module"` to package.json, and run the server directly with Node before involving Docker.

Write the first Dockerfile

Describe the Notes API image with a base image, working directory, copied files, documented port, and startup command.

A Dockerfile is an ordered build recipe. Each instruction creates image metadata or a filesystem layer that later instructions can use.

`FROM` chooses the starting filesystem, `WORKDIR` changes the directory for later steps, `COPY` moves files from the build context, and `CMD` supplies the default process. `EXPOSE` documents a port but does not publish it.

```
FROM node:22-alpine
WORKDIR /app
COPY package.json server.js ./
ENV PORT=3000
EXPOSE 3000
CMD ["node", "server.js"]
```

Prefer the JSON form of `CMD`. Docker starts Node directly, so termination signals reach the application instead of first passing through an extra shell. This helps the process shut down cleanly during a deploy.

The Dockerfile separates build-time facts from runtime choices. `COPY` and `RUN` shape the immutable image; environment variables and port mappings can vary per container. `EXPOSE 3000` is documentation for humans and tools, not a firewall rule and not the equivalent of `-p`. Also choose base-image versions deliberately: a tag is convenient, while a recorded digest gives a reproducible release input.

Save the file as `Dockerfile`, then build it with `docker build -t notes-api:dev .`.

Control the build context

Keep secrets, dependencies, Git history, and unrelated files out of image builds with a small context and .dockerignore.

The final dot in `docker build ... .` selects the build context. Dockerfile `COPY` instructions can only read files from that context.

Sending a large context slows builds and can accidentally make sensitive files available to `COPY`. A `.dockerignore` file excludes paths before the context is sent to the builder.

```
node_modules
.git
.env
coverage
dist
*.log
```

Think of the context as a security and correctness boundary, not only a performance setting. A file excluded by `.dockerignore` cannot be copied accidentally. A file that enters an image layer can remain recoverable from image history even if a later step deletes it, so never copy credentials and then try to clean them up.

When a build genuinely needs a credential, use a BuildKit secret mount so the secret is available only to that build step and is not committed to a layer. Run a plain-progress build and read the context transfer line:

```
docker build --progress=plain -t notes-api:dev .
```

If the context is unexpectedly large, inspect `.dockerignore` and the directory passed as the final

build argument before optimizing individual instructions.

Create `.dockerignore`, add a temporary large file, and compare the build context size before and after ignoring it.

Use layer caching deliberately

Order Dockerfile instructions so dependency installation is reused until its real inputs change.

Docker can reuse an unchanged build step and the layers before it. A change invalidates that step and every step that follows.

Copy dependency manifests and install dependencies before copying frequently changing application code. A source edit then keeps the expensive dependency layer cached, while a package change correctly rebuilds it.

```
FROM node:22-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY server.js ./
CMD ["node", "server.js"]
```

Caching must remain correct before it becomes fast. Every dependency installation step needs all of its real inputs, including the lockfile. If a script, configuration file, or package-manager setting affects installation, copy it before the install step too. Otherwise Docker may reuse a layer whose inputs were incomplete.

Use `--no-cache` to diagnose stale build layers, not as a normal workflow. It forces instructions to run again but does not by itself pull a newer base image; use `--pull` when freshness of `FROM` matters. Build cache mounts can speed package downloads, but the build must still succeed when that cache is empty because cache contents are disposable optimization data.

Add one dependency, rebuild twice, then edit only `server.js` and rebuild again. Read which steps say `CACHED`.

Check your understanding: build images

Check Dockerfile instructions, build context, layer caching, ignore rules, startup commands, and configuration boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Data and networking

Persist data, mount files, publish ports, and connect containers by name.

Separate data from containers

Understand the writable container layer and move durable application state into storage with an independent lifecycle.

A container can write files, but those files live in that container's writable layer. Removing the container removes that layer.

Treat containers as replaceable. Put durable database or upload data in a volume, and put configuration in environment variables or mounted configuration files. This lets you update the image without treating one old container as precious.

```
docker volume create notes-data
docker volume inspect notes-data
```

A named volume has its own lifecycle, but persistence is not a backup. Accidental deletion, application bugs, and database corruption can all survive a container replacement. Define how the volume is backed up, where the backup is stored, and how a restore is tested before calling the data durable.

Use the writable container layer only for disposable runtime changes. Use a volume for state that must survive replacement, and a temporary filesystem for scratch data that should disappear. Do not edit Docker's internal volume directory on the host; access the data through a container or a supported backup tool so ownership and filesystem assumptions remain consistent.

Create and inspect a named volume. Identify its Docker-managed name without editing its host storage directly.

Choose volumes or bind mounts

Use named volumes for Docker-managed data and bind mounts when the host must directly provide or edit files.

Volumes and bind mounts both place external storage at a container path, but they serve different workflows.

A named volume is managed by Docker and works well for databases. A bind mount maps an explicit host path and works well for source code during development. A mount hides image files already present at its target path.

```
docker run --rm -v notes-data:/data alpine sh -c "echo hello > /data/note.txt"
docker run --rm -v notes-data:/data alpine cat /data/note.txt
docker run --rm --mount type=bind,src="$PWD",dst=/work,readonly alpine ls /work
```

Bind mounts inherit the host path's existence, permissions, and contents. They can make a development loop fast, but they also couple the container to one machine and can hide files placed at the mount target by the image. A writable bind mount lets the container change host files, so use `readonly` unless writes are part of the design.

Volumes are more portable across Docker workflows because Docker owns their location, but applications must still handle ownership and backups. Before choosing, ask who owns the data, whether a human must edit it directly, whether the container may write it, and how it will be restored. Those questions lead to a better answer than treating the two mount types as interchangeable syntax.

Persist one file in a volume, then mount the project directory read-only into a different container.

Publish ports deliberately

Distinguish a process listening inside a container from a host port published to local or external clients.

The Notes API listens on port 3000 inside its container. Clients on the host need a published port

to reach it.

`-p 127.0.0.1:8080:3000` maps host port 8080 to container port 3000 and limits host access to the loopback interface. Omitting the host IP usually publishes on all host interfaces.

```
docker run --name notes -d -p 127.0.0.1:8080:3000 notes-api:dev
docker port notes
curl http://localhost:8080
```

`EXPOSE` in an image does not publish anything. Publishing is a runtime decision that creates a path from a host interface to a container port. The host and container port numbers do not need to match, and two containers can use the same internal port as long as their host mappings differ.

The host address controls exposure. Loopback is a safer development default; binding to all interfaces may make the service reachable from the local network or beyond, depending on firewall and routing rules. Databases used only by other containers usually need no published port at all. Confirm the actual mapping with `docker port` rather than inferring it from the Dockerfile.

Run the API with the loopback-only mapping, inspect it, then try a different host port without changing the image.

Connect containers by name

Place services on a user-defined network and use Docker DNS instead of hard-coded container IP addresses.

Container IP addresses are implementation details and can change when containers are replaced. Stable service names are a better connection contract.

A user-defined Docker network includes DNS-based name resolution. Containers on that network can use another container's name as a hostname. They connect to the other container's internal port, not its published host port.

```
docker network create notes-net
docker run --name database --network notes-net -d postgres:17-alpine
docker run --rm --network notes-net alpine ping -c 1 database
```

Inside the API container, `localhost` means the API container itself. To reach PostgreSQL, use `database:5432`: the service name and the port on which PostgreSQL listens inside its container. Publishing `5432` on the host is unnecessary for this connection and expands the exposed surface.

Docker DNS follows container and network membership, so replacement containers can receive new IP addresses without changing application configuration. Name resolution is discovery, not authentication or encryption. Still use database credentials, restrict which services join each network, and add transport security when the threat model requires it.

Create the network and confirm the temporary client resolves `database`. Remove the practice containers after the check.

Check your understanding: data and networking

Check writable layers, volumes, bind mounts, port publishing, container networks, and service-name DNS.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Compose applications

Run the API and PostgreSQL together with health checks and a fast development loop.

Describe services with Compose

Replace a collection of long docker run commands with one versioned description of the application services.

Docker Compose describes related containers, networks, volumes, and configuration in `compose.yaml`.

A service is a repeatable container configuration. Compose gives services a shared default network and uses service names for DNS. It does not merge the services into one container.

```
services:
  api:
    build: .
    ports:
      - "8080:3000"
  database:
    image: postgres:17-alpine
    volumes:
      - database-data:/var/lib/postgresql/data

volumes:
  database-data:
```

Treat the file as desired configuration. `docker compose up` compares it with the current project and creates or replaces resources as needed. The project name also namespaces generated container, network, and volume names, which is why the same file can run as separate projects.

Always read `docker compose config` before a risky change. It resolves interpolation and merged files, showing the configuration Docker will actually use without starting containers. In production-like environments, pin external image versions and review the rendered output for unintended host ports, bind mounts, or missing variables. Compose records how services should run; it does not make application data migrations or secret distribution safe automatically.

Create the file and run `docker compose config` to see the normalized configuration before starting anything.

Configure the database service

Supply PostgreSQL configuration without baking environment-specific values into the application image.

The official PostgreSQL image needs initial database credentials. The API needs a connection URL using the Compose service name.

Environment variables configure a container at runtime. Keep real secrets out of the Compose file committed to Git; use a local ignored environment file or a platform secret store for deployments.

```
services:
  api:
    environment:
      DATABASE_URL: postgres://notes:development-only@database:5432/notes
  database:
    environment:
      POSTGRES_USER: notes
      POSTGRES_PASSWORD: development-only
      POSTGRES_DB: notes
```

Initialization variables affect the official image only when the database directory is empty. Once a named volume contains a database, changing `POSTGRES_USER`, `POSTGRES_PASSWORD`, or `POSTGRES_DB` does not rewrite existing roles and data. Make later changes through database administration or migrations and test them against a copy of real data.

A connection URL is convenient but easy to expose in logs or process inspection. Keep production credentials in a platform secret store, give the application only the permissions it needs, and rotate credentials independently of rebuilding the image. Development defaults must be obviously local and must never become the production fallback.

Add development configuration, start only the database, and inspect its logs with `docker compose logs database`.

Add health and startup handling

Distinguish process startup from service readiness and make the API tolerate a database that is still initializing.

A running database container may still be replaying logs or creating its first database. Process state alone does not mean the service is ready.

A health check tests useful behavior. Compose can wait for a dependency to become healthy, but the application should still retry temporary connection failures because services can restart after initial startup.

```
database:
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U notes -d notes"]
    interval: 5s
    timeout: 3s
    retries: 10
api:
  depends_on:
    database:
      condition: service_healthy
```

The health command runs inside the database container, so it must use tools available in that image and test the same behavior the application needs. Keep it fast, read-only, and specific. A check that always succeeds proves nothing; a heavyweight check can create its own outage.

Use `docker inspect` to see recent probe results when a service remains unhealthy:

```
docker inspect database --format '{{json .State.Health}}'
```

Startup ordering is only an initial convenience. Connections can fail after both services are healthy, so the API still needs bounded retries with backoff, useful logs, and a clear terminal failure. Health checks should expose a problem, not replace application resilience.

Add the health check, recreate the stack, and watch `docker compose ps` while PostgreSQL changes from starting to healthy.

Build a fast development loop

Use targeted rebuilds, logs, exec, and cleanup commands without destroying useful development data by accident.

Compose is most useful when everyday commands are predictable enough to become muscle memory.

`up --build` reconciles the running project with its configuration, `logs -f` follows output, and `exec` runs a command in an existing service. `down` removes project containers and networks; `down -v` also removes named volumes and their data.

```
docker compose up --build -d
docker compose logs -f api
docker compose exec api node --version
docker compose ps
docker compose down
```

Know which artifact each edit invalidates. A bind-mounted source change may need only an application reload. A Dockerfile or dependency change needs an image rebuild. A configuration change may need a container recreation even when the image is unchanged. Use `docker compose up -d --build api` to target the service, then inspect `docker compose ps` instead of assuming it was replaced.

Treat `down -v` as destructive: it removes the project's named volumes and cannot be undone without a backup. Ordinary `down` is the safer reset when you want to preserve database state. When debugging, change one layer at a time and keep the last working image tag so you can distinguish an application regression from damaged development data.

Change the API response, rebuild only the API service, and confirm the database data survives ordinary `down` and `up` commands.

Check your understanding: compose applications

Check Compose services, networks, volumes, environment configuration, health checks, and repeatable development commands.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Ship and operate

Build a focused image, reduce privileges, diagnose failures, and publish a release.

Build a production image

Use a multi-stage build to separate compilation tools from the smaller runtime filesystem delivered to production.

A build stage may need TypeScript, test tools, and a compiler. The running application normally needs only production dependencies and compiled output.

Each `FROM` begins a stage. `COPY --from` selects the artifacts that cross into the runtime image. This reduces size and prevents build-only files from becoming part of the deployed filesystem.

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

FROM node:22-alpine AS runtime
WORKDIR /app
ENV NODE_ENV=production
COPY package*.json ./
RUN npm ci --omit=dev
COPY --from=build /app/dist ./dist
CMD ["node", "dist/server.js"]
```

The stage boundary is an explicit allowlist. Only paths named by `COPY --from` enter the runtime stage. Inspect the final image rather than assuming the boundary worked: list its filesystem, review `docker history`, and run the same smoke request you used in development.

Multi-stage builds can also reveal portability mistakes. Native modules compiled against one operating system or CPU architecture must match the runtime stage. Keep build and runtime families compatible, build for the production platform, and test that exact image. A smaller runtime image reduces unused tools and attack surface, but it still needs vulnerability scanning, a non-root user, and a repeatable way to rebuild when the base image changes.

Adapt the pattern to a TypeScript version of the Notes API and inspect the final image history.

Run as a non-root user

Reduce the impact of an application compromise by giving the container process only the filesystem access it needs.

Many base images start as root inside the container. Isolation helps, but unnecessary root privileges still increase risk.

Use the non-root user supplied by the base image or create one. Make application files readable by that user and writable only where the program truly needs to write. Do not solve permission errors by making everything world-writable.

```
COPY --chown=node:node --from=build /app/dist ./dist
USER node
CMD ["node", "dist/server.js"]
```

The user change must match filesystem ownership. Copying files as root and switching users later

is fine for read-only application code, but writable paths such as an uploads or cache directory need narrow, explicit ownership. Test both a required write and a forbidden write instead of checking only the numeric UID.

Non-root is one layer of defense. Also avoid unnecessary Linux capabilities, mount filesystems read-only where practical, and do not mount the Docker socket into an application container. Root inside a container is not automatically root on the host, but a kernel or runtime mistake has a much larger impact when the process begins with broad privileges.

Run the image and print its identity with `docker run --rm notes-api:prod id`.

Inspect runtime failures

Use container state, exit codes, logs, resource statistics, and an interactive shell to diagnose a failing workload.

A container stops when its main process exits. Restarting blindly can hide the evidence that explains why it exited.

Start with `ps -a`, inspect the exit code, and read logs. Use `stats` for live resource pressure and `exec` only while the container is running. Keep application logs on standard output and standard error so the runtime can collect them.

```
docker ps -a
docker inspect notes --format "{{.State.ExitCode}} {{.State.Error}}"
docker logs --tail 100 notes
docker stats --no-stream notes
docker exec -it notes sh
```

Container state gives stronger clues than the word "crashed." Inspect whether the process was killed for memory pressure, how many times a restart policy retried it, and the timestamps around the exit:

```
docker inspect notes --format '{{.State.Status}} exit={{.State.ExitCode}} oom={{.State.OOMKil
```

An exit code points to the process result, while `.State.Error` can reveal a runtime failure before the process started, such as a missing executable. Connectivity failures usually appear after startup and should be tested from the same network namespace. Preserve the failing container until you have captured evidence, change one variable, and rerun the smallest command that can disprove your current hypothesis.

Start the API with an intentionally invalid command, diagnose the exit, then restore the correct command.

Tag, push, and deploy the image

Publish an immutable application version to a registry and identify what a production container platform must configure around it.

A deployment starts with an image that another Docker engine or container platform can pull. Give releases explicit version tags rather than relying only on `latest`.

Authenticate to your registry, tag the tested image with the registry repository, and push it. Production still needs secrets, ports, persistent storage, health checks, resource limits, logs, backups, and a rollback plan. The image alone is not the whole system.

```
docker tag notes-api:prod YOUR_NAME/notes-api:1.0.0
docker login
docker push YOUR_NAME/notes-api:1.0.0
```

Record the digest produced by the push and deploy that immutable reference when the platform supports it. Also confirm the image platform matches production. A successful pull proves distribution worked; it does not prove the configured service can reach its database, write its volume, or pass its health check.

Make the release record include the image digest, configuration version, migration state, and smoke-test result. Keep the previous digest available for rollback. Rolling the image back does not roll back a database schema or external side effect, so migrations should remain compatible across the rollback window. After deployment, request a real application path and verify logs and health from the production environment before declaring the release complete.

Use a registry repository you control, push one versioned tag, and pull it into a clean local environment or another machine.

Check your understanding: ship and operate

Check production image design, non-root users, runtime inspection, tagging, registries, and deployment boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/docker/.