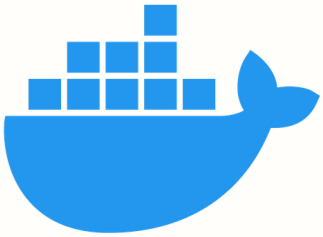
GrowthSchool

DOCKER

COMPLETE NOTES



BUILD - SHIP - RUN

Images, containers, Dockerfiles, Compose and production packaging

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn Docker in order. Every page has explanation, real commands, a diagram or example, common mistakes, a student lab and a revision snapshot. We package one ecommerce "Shop API" throughout.

HOW TO USE THESE NOTES

Type every command in a real terminal. Break an image on purpose and fix it - that is how the concepts stick. Use pages 25-29 for fast revision before interviews.

1. Containers vs Virtual Machines

GrowthSchool

IN SIMPLE TERMS
Docker packages an app and everything it needs into a container. A container shares the host's operating system, so it is far smaller and faster to start than a full virtual machine.

* The Old Way (VMs)

- Each VM ships a full guest OS on top of a hypervisor - heavy, slow to boot, large images.
- Great isolation, but you pay for a whole OS per application instance.

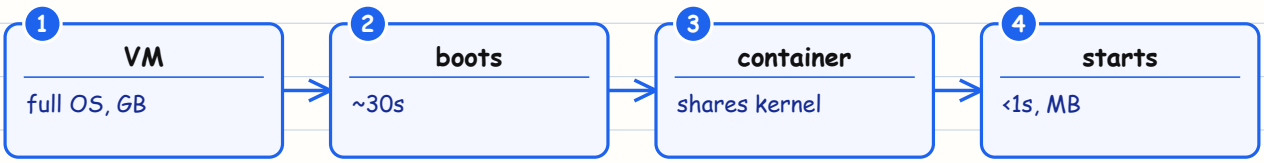
* The Container Way

- Containers share the host OS kernel and isolate processes, filesystem and network.
- They start in milliseconds, are megabytes not gigabytes, and pack far more density per host.

* When Each Fits

- Containers for app packaging and scaling. VMs for strong OS-level isolation or different kernels.

Container vs VM



REAL-WORLD EXAMPLE

```
# same app, radically different footprint
# VM image: ~1-4 GB, boots in ~30s
# container image: ~20-150 MB, starts in <1s
docker run --rm hello-world
```

COMMON MISTAKE
Thinking a container is a lightweight VM. It is an isolated process, not a machine.
It has no init/systemd by default and should run one main process.

REVISION SNAPSHOT

ASK YOURSELF	Why does a container start far faster than a VM?
DO THIS	Run hello-world, then docker info to see the host kernel it shares.

2. Docker Architecture

GrowthSchool

IN SIMPLE TERMS
Docker's architecture has three parts: the command you type (client), the background service that does the work (daemon), and the registry that stores images.

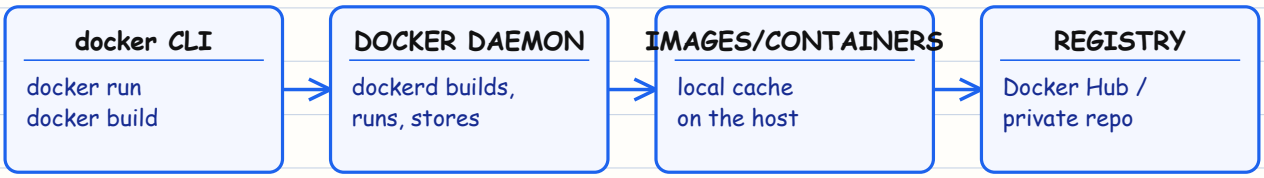
* Client and Daemon

- The docker CLI is a thin client that talks to the Docker daemon (dockerd) over an API.
- The daemon does the real work: building images, running containers, managing storage and networks.

* Registries

- A registry stores and distributes images. Docker Hub is the default public registry.
- docker pull downloads images; docker push uploads them to a registry you can authenticate to.

Docker Architecture: client -> daemon -> registry



REAL-WORLD EXAMPLE

```
docker version        # client + daemon versions
docker info           # daemon status, storage driver
docker pull nginx:1.27
docker images
```

REVISION SNAPSHOT

ASK YOURSELF What are the three pieces: client, daemon, registry - and who does the work?

DO THIS Run docker info and identify the storage driver and running container count.

3. Images, Layers & Union Filesystem

GrowthSchool

IN SIMPLE TERMS
An image is a read-only template built from stacked layers; because layers are cached and shared, rebuilds and downloads are fast.

* What an Image Is

- An image is a read-only stack of layers plus metadata describing how to run it.
- Each Dockerfile instruction that changes the filesystem creates a new cached layer.

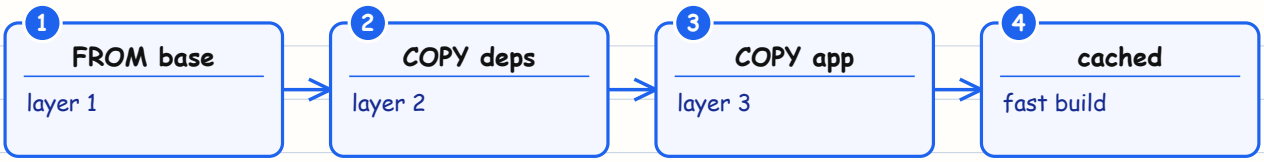
* Layers & Caching

- Layers are shared between images and cached between builds, making rebuilds fast.
- Order matters: put rarely-changing steps first so the cache is reused as often as possible.

* Tags

- A tag names a version, e.g. shop-api:1.4.2. "latest" is just a default tag, not "newest".

Image layers



REAL-WORLD EXAMPLE

```
docker history shop-api:1.0    # see the layers
docker image inspect shop-api:1.0
# a tag is a pointer to an image id:
docker tag shop-api:1.0 registry.example.com/shop-api:1.0
```

COMMON MISTAKE
Relying on the latest tag in production. It is mutable and ambiguous; two hosts may run different code. Always pin an explicit version or digest.

REVISION SNAPSHOT

ASK YOURSELF	Why does instruction order in a Dockerfile affect build speed?
DO THIS	Run docker history on any image and explain what each layer came from.

4. Container Lifecycle

GrowthSchool

IN SIMPLE TERMS
The container lifecycle is the stages a container moves through: created, running, stopped, and finally removed.

* States

- A container is created from an image, then running, then it can be paused, stopped, or removed.
- A stopped (exited) container keeps its writable layer until you docker rm it.

* The Writable Layer

- Each container adds a thin writable layer on top of the image; changes there are lost on rm.
- Persist important data in volumes, not the container layer.

Image -> Container Lifecycle

REAL-WORLD EXAMPLE

```
docker run -d --name api shop-api:1.0
docker stop api
docker start api
docker rm -f api        # force remove (stop + rm)
docker ps -a           # include stopped containers
```

COMMON MISTAKE
Assuming files written inside a running container survive removal. They do not - the writable layer is discarded. Use a volume for anything that must persist.

REVISION SNAPSHOT

ASK YOURSELF	What happens to container data when you docker rm it?
DO THIS	Create a file inside a container, remove it, recreate, and prove the file is gone.

5. Registries & Docker Hub

GrowthSchool

IN SIMPLE TERMS
A registry is a store for images. Docker Hub is the default public one; you pull images from it and push your own to it.

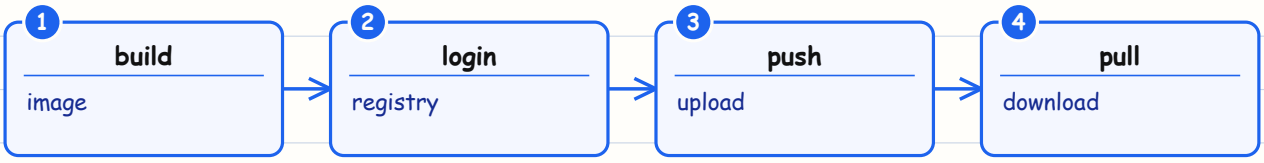
* Pull and Push

- docker pull fetches an image; docker push uploads to a registry after docker login.
- Image names encode the registry: registry.example.com/team/shop-api:1.0.

* Public vs Private

- Docker Hub hosts public and private repos. Companies also run private registries (see page 21).
- Use official or verified base images and pin versions to reduce supply-chain risk.

Registry flow



REAL-WORLD EXAMPLE

```
docker login
docker tag shop-api:1.0 myuser/shop-api:1.0
docker push myuser/shop-api:1.0
docker pull myuser/shop-api:1.0
```

STUDENT LAB
Create a free Docker Hub repo, push a small image you built, then pull it on a clean machine or after docker image rm to prove it round-trips.

REVISION SNAPSHOT

ASK YOURSELF	How does an image name encode which registry it lives in?
DO THIS	Push one image to a repo and pull it back after deleting the local copy.

6. Running Containers

GrowthSchool

IN SIMPLE TERMS
Running a container means starting an app from an image, usually publishing a port and passing in some settings.

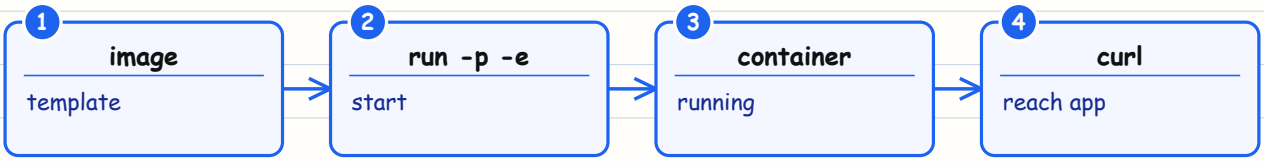
* Core Flags

- -d run detached; --name give a stable name; -p publish a port; -e set an env var; --rm auto-clean.
- -it gives an interactive terminal, useful for shells and debugging.

* Ports

- Containers are isolated: publish a port with -p host:container to reach the app from the host.
- Example: -p 8080:8080 maps host 8080 to the container app port 8080.

Run a container



REAL-WORLD EXAMPLE

```
docker run -d --name api -p 8080:8080 \
  -e APP_ENV=production shop-api:1.0
curl http://localhost:8080/health
docker run --rm -it ubuntu:24.04 bash   # throwaway shell
```

COMMON MISTAKE
Forgetting -p and wondering why localhost:8080 refuses the connection. Without publishing, the port is only reachable inside Docker networks, not from your host.

REVISION SNAPSHOT

ASK YOURSELF	What does -p 8080:8080 actually connect?
DO THIS	Run a web image, publish its port, and load it in your browser.

7. Dockerfile Basics

GrowthSchool

IN SIMPLE TERMS
A Dockerfile is a recipe of instructions (FROM, COPY, RUN, CMD) that tells Docker how to build your image step by step.

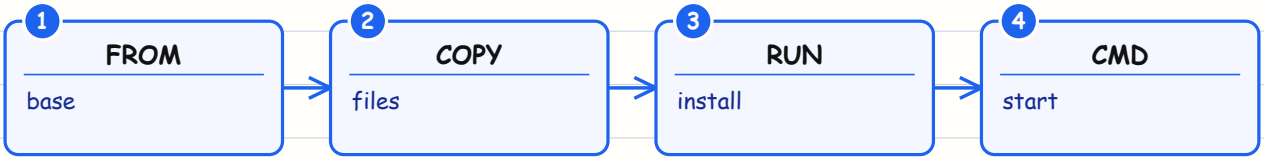
* The Essential Instructions

- FROM sets the base image. WORKDIR sets the working directory. COPY adds files from the build context.
- RUN executes build-time commands. CMD sets the default command when the container starts.

* Build and Run

- docker build -t name:tag . reads the Dockerfile and build context in the current folder.
- Then docker run name:tag starts a container from the image you built.

Dockerfile basics



REAL-WORLD EXAMPLE

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
EXPOSE 8080
CMD ["node", "server.js"]
```

COMMON MISTAKE
Doing COPY . . before installing dependencies. Any code change then busts the dependency cache and reinstalls everything. Copy manifests + install first, then copy the rest.

REVISION SNAPSHOT

ASK YOURSELF	Why copy package files and install BEFORE copying the whole app?
DO THIS	Write a minimal Dockerfile for a small app and build it successfully.

8. Dockerfile Deep Dive

GrowthSchool

IN SIMPLE TERMS
The deeper Dockerfile instructions control how your container starts and behaves - things like ENTRYPOINT, ENV, USER, and HEALTHCHECK.

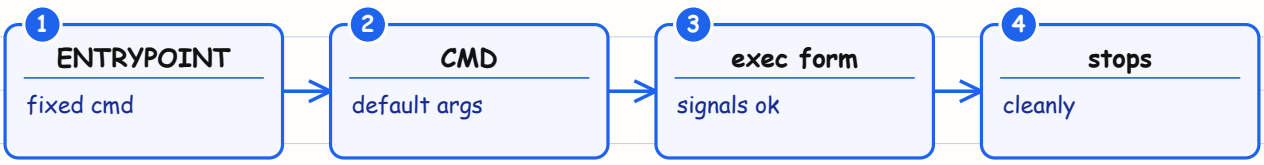
* CMD vs ENTRYPOINT

- ENTRYPOINT sets the fixed executable; CMD provides default arguments you can override at runtime.
- Use exec form ["cmd","arg"] so signals reach your process and it stops cleanly.

* More Instructions

- ARG = build-time variable; ENV = runtime variable; USER = drop root; HEALTHCHECK = liveness signal.
- EXPOSE documents a port (it does not publish it); LABEL adds metadata.

CMD vs ENTRYPOINT



REAL-WORLD EXAMPLE

```
FROM python:3.12-slim
ENV PYTHONUNBUFFERED=1
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
USER 1000
HEALTHCHECK CMD curl -f http://localhost:8080/health || exit 1
ENTRYPOINT ["gunicorn", "app:app", "-b", "0.0.0.0:8080"]
```

COMMON MISTAKE
Using shell form CMD gunicorn ... so the app runs as a child of /bin/sh. Then SIGTERM hits the shell, not your app, and stops become slow, ungraceful kills. Prefer exec form.

REVISION SNAPSHOT

ASK YOURSELF	When do you use ENTRYPOINT vs CMD, and why the exec form?
DO THIS	Add a non-root USER and a HEALTHCHECK to your Dockerfile.

9. Build Context & .dockerignore

GrowthSchool

IN SIMPLE TERMS
The build context is the folder sent to Docker when building; a .dockerignore file keeps unwanted files (and secrets) out of it.

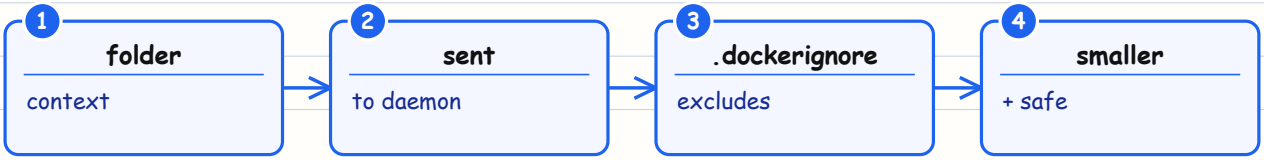
* The Build Context

- docker build sends the whole folder (the context) to the daemon. Big contexts = slow builds.
- Only files in the context can be COPYed. Keep it lean.

* .dockerignore

- Like .gitignore, it excludes files from the context: node_modules, .git, .env, dist, logs.
- This speeds builds, shrinks images, and stops secrets leaking into image layers.

Build context



REAL-WORLD EXAMPLE

```
# .dockerignore
.git
node_modules
.env
dist
*.log
Dockerfile
```

COMMON MISTAKE
Shipping node_modules or a .env into the image because there is no .dockerignore. This bloats the image and can bake secrets into a layer that anyone with the image can extract.

REVISION SNAPSHOT

ASK YOURSELF	Why can a missing .dockerignore leak secrets into an image?
DO THIS	Add a .dockerignore and compare build time and image size before/after.

10. Environment & Configuration

GrowthSchool

IN SIMPLE TERMS
Configuration means passing settings into a container at run time (environment variables) instead of baking them into the image.

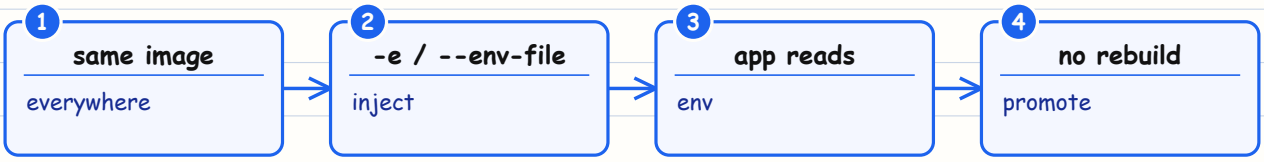
* Passing Config

- Use -e KEY=value or --env-file to inject configuration; read it from the environment in the app.
- Keep the same image across environments; only the injected config changes.

* Secrets Care

- Do not bake secrets into the image or commit .env. Inject at runtime or use a secrets manager.
- Anyone who can pull the image can read anything baked into its layers.

Config at runtime



REAL-WORLD EXAMPLE

```
docker run --env-file ./prod.env shop-api:1.0
docker run -e DB_HOST=db -e DB_PORT=5432 shop-api:1.0
# 12-factor: config comes from the environment, not the image
```

COMMON MISTAKE
Building separate images per environment (shop-api-dev, shop-api-prod). Build once, promote the same artifact, and vary only the injected configuration.

REVISION SNAPSHOT

ASK YOURSELF	Why build one image and inject config, instead of one image per environment?
DO THIS	Run the same image twice with different --env-file values.

11. Volumes & Bind Mounts

GrowthSchool

IN SIMPLE TERMS
A volume is Docker-managed storage that outlives a container; a bind mount maps a folder from your computer straight into the container.

* Why Persistence

- The container writable layer is disposable. Databases and uploads must live in a volume.
- Named volumes are managed by Docker; bind mounts map a host path into the container.

* Choosing

- Named volume for production data (portable, backed up). Bind mount for local dev live-reload.
- tmpfs keeps sensitive data in memory only, never on disk.

NAMED VOLUME	BIND MOUNT
managed by Docker in /var/lib/docker	maps a host path into the container
best for databases + app state	great for local dev live-reload
survives container removal	tied to host filesystem layout
-v shopdata:/var/lib/postgresql/data	-v $(pwd)/src:/app/src

REAL-WORLD EXAMPLE

```
docker volume create shopdata
docker run -d --name db -v shopdata:/var/lib/postgresql/data postgres:16
# dev live-reload with a bind mount:
docker run -v ${PWD}/src:/app/src shop-api:dev
```

COMMON MISTAKE
Storing database files in the container layer, then losing all data on docker rm. Always mount a named volume for stateful services.

REVISION SNAPSHOT

ASK YOURSELF	When do you pick a named volume vs a bind mount?
DO THIS	Put a database on a named volume, remove the container, recreate, verify data persists.

12. Docker Networking

GrowthSchool

IN SIMPLE TERMS
Docker networking is how containers get IP addresses, reach the internet, and talk to each other - by name on a shared network.

* Default Bridge

- By default containers join a bridge network with private IPs and reach the outside via NAT.
- Publish ports with -p to accept traffic from the host.

* User-Defined Networks

- Create a user-defined bridge so containers resolve each other by name via built-in DNS.
- This is how Compose lets the api reach the db as the hostname "db".

NETWORK DRIVERS

bridge : default, private subnet per host
host : share host network, no isolation
none : no networking at all
user-defined bridge : DNS by container name

KEY IDEAS

containers on one user network resolve each other by name (built-in DNS)
publish ports with -p host:container
never rely on changing container IPs

REAL-WORLD EXAMPLE

```
docker network create shopnet
docker run -d --name db --network shopnet postgres:16
docker run -d --name api --network shopnet -e DB_HOST=db shop-api:1.0
# api reaches the database at hostname "db"
```

COMMON MISTAKE
Hard-coding a container IP address in config. IPs change on restart. Use the container/service name on a user-defined network and let Docker DNS resolve it.

REVISION SNAPSHOT

ASK YOURSELF	Why do you get name-based DNS on a user-defined bridge but not the default one?
DO THIS	Put two containers on one network and curl one from the other by name.

13. Docker Compose Basics

GrowthSchool

IN SIMPLE TERMS
Docker Compose lets you define and run several containers together with one YAML file and one command.

* One File, Many Services

- A compose.yaml describes services, networks and volumes declaratively so you run them together.
- docker compose up -d starts everything; docker compose down stops and cleans it up.

* Why Compose

- It replaces long, error-prone docker run commands and documents your local stack in Git.
- Services share a network automatically and can reference each other by service name.

REAL-WORLD EXAMPLE

```
services:
  api:
    build: .
    ports: ["8080:8080"]
    environment: { DB_HOST: db }
  db:
    image: postgres:16
    volumes: ["shopdata:/var/lib/postgresql/data"]
volumes: { shopdata: {} }
```

REVISION SNAPSHOT

ASK YOURSELF	What three things does a compose file usually declare?
DO THIS	Write a two-service compose file and bring it up with docker compose up.

14. Compose for the Shop API

GrowthSchool

IN SIMPLE TERMS
This page builds a real multi-service Compose stack (web, API, database) for the Shop API on one network.

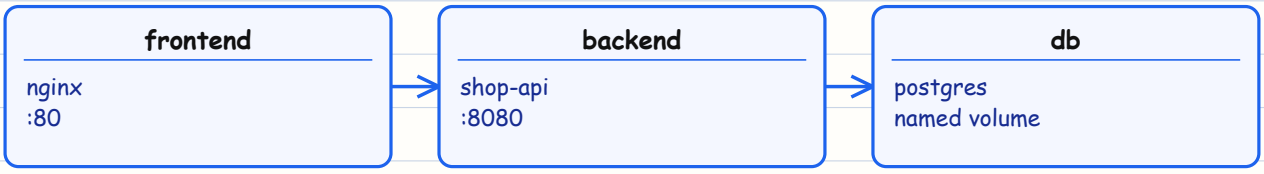
* The Full Local Stack

- Frontend (nginx), backend (shop-api) and database (postgres) on one network with a data volume.
- depends_on plus a healthcheck ensures the API waits for the database to be ready.

* Everyday Commands

- compose up -d --build to rebuild and start; compose logs -f api to tail; compose ps to see status.

Shop API with Compose: one network, three services



REAL-WORLD EXAMPLE

```
docker compose up -d --build
docker compose ps
docker compose logs -f api
docker compose exec db psql -U shop
docker compose down -v      # also remove volumes
```

COMMON MISTAKE
Assuming depends_on waits for the DB to be READY. It only waits for it to START. Add a healthcheck and a condition, or retry the DB connection in the app.

REVISION SNAPSHOT

ASK YOURSELF	Does depends_on guarantee the database is ready to accept queries?
DO THIS	Bring up the three-service Shop API stack and hit the API through nginx.

15. Multi-Stage Builds

GrowthSchool

IN SIMPLE TERMS
A multi-stage build compiles your app in a big "builder" image, then copies only the finished output into a tiny final image.

* The Problem

- Build tools (compilers, dev dependencies) make images huge and expand the attack surface.
- You need them to build, but not to run.

* The Solution

- Use multiple FROM stages: build in a fat stage, then COPY only the artifacts into a slim runtime.
- The final image contains just what runs, often 5-10x smaller.

Multi-Stage Build: fat builder -> slim runtime

REAL-WORLD EXAMPLE

```
# stage 1: build
FROM node:20 AS build
WORKDIR /app
COPY . . && RUN npm ci && npm run build
# stage 2: runtime
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
```

COMMON MISTAKE
Shipping the build stage as the final image, dragging along compilers and dev deps. Always end on a minimal runtime stage and copy only built artifacts into it.

REVISION SNAPSHOT

ASK YOURSELF	What do you COPY --from the build stage into the runtime stage?
DO THIS	Convert a single-stage Dockerfile to multi-stage and compare image sizes.

16. Image Optimization

GrowthSchool

IN SIMPLE TERMS
Image optimization is the set of techniques - small base, fewer layers, good caching - that make images smaller and faster.

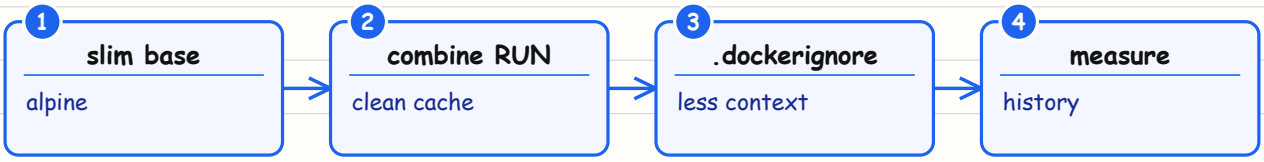
* Smaller & Faster

- Pick a slim/alpine base, combine RUN steps, clean package caches, and use .dockerignore.
- Order layers from least- to most-frequently changed to maximise cache hits.

* Measure It

- Use docker history and docker images to find fat layers; use dive-style analysis to inspect them.

Shrink the image



REAL-WORLD EXAMPLE

```
# combine + clean in one layer (Debian example)
RUN apt-get update && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*
docker images shop-api        # check final size
docker history shop-api:1.0   # find the biggest layers
```

COMMON MISTAKE
Running apt-get update in one RUN and install in another. The cache can serve a stale package index. Combine update + install + cleanup in a single RUN.

REVISION SNAPSHOT

ASK YOURSELF	Name three concrete ways to shrink an image.
DO THIS	Reduce one of your images by at least 30% and record what changed.

17. Image & Supply-Chain Security

GrowthSchool

IN SIMPLE TERMS
Image security means shrinking the attack surface: minimal trusted bases, scanning for vulnerabilities, and never running as root.

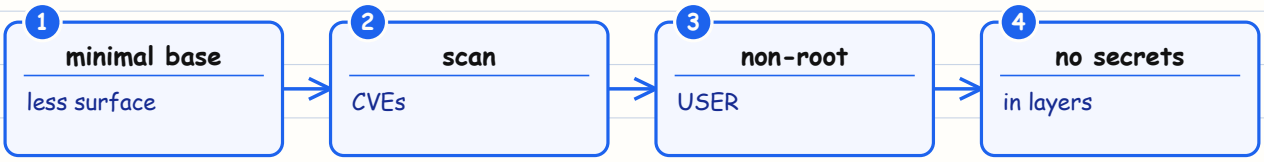
* Reduce Risk

- Use minimal, trusted base images; pin versions/digests; run as non-root; scan for CVEs.
- Fewer packages means fewer vulnerabilities and a smaller attack surface.

* Scan & Verify

- Scan images in CI and fail on critical CVEs; sign images so consumers can verify provenance.
- Never bake credentials into layers - they are trivially extractable.

Secure the image



REAL-WORLD EXAMPLE

```
docker scout cves shop-api:1.0     # or trivy image shop-api:1.0
# Dockerfile hardening:
USER 1000
FROM gcr.io/distroless/base-debian12   # no shell, tiny surface
```

COMMON MISTAKE
Running containers as root by default. If the app is compromised, root inside the container is a far bigger problem. Add a non-root USER.

REVISION SNAPSHOT

ASK YOURSELF	Why does running as non-root and using a minimal base reduce risk?
DO THIS	Scan one of your images and remediate at least one high/critical finding.

18. Debugging Containers

GrowthSchool

IN SIMPLE TERMS
Debugging containers is inspecting a misbehaving container using its logs, a shell inside it, and its full configuration.

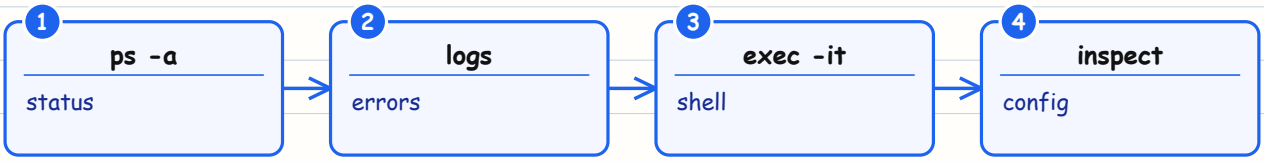
* First Four Commands

- docker logs shows stdout/stderr; docker exec opens a shell; docker inspect dumps full config.
- docker stats shows live CPU/memory/network per container.

* A Method

- Is it running? (ps) -> what did it say? (logs) -> get inside (exec) -> why configured so? (inspect).

Debug a container



REAL-WORLD EXAMPLE

```
docker ps -a
docker logs -f --tail 100 api
docker exec -it api sh
docker inspect api | less
docker stats api
```

COMMON MISTAKE
Deleting and recreating a crashing container repeatedly instead of reading its logs. The answer is almost always in docker logs or the exit code from docker ps -a.

REVISION SNAPSHOT

ASK YOURSELF	What is your four-step method for a misbehaving container?
DO THIS	Deliberately break an env var, then diagnose it using logs + inspect.

19. Resource Limits

GrowthSchool

IN SIMPLE TERMS
Resource limits cap how much CPU and memory a container may use, so one container cannot starve the others or the host.

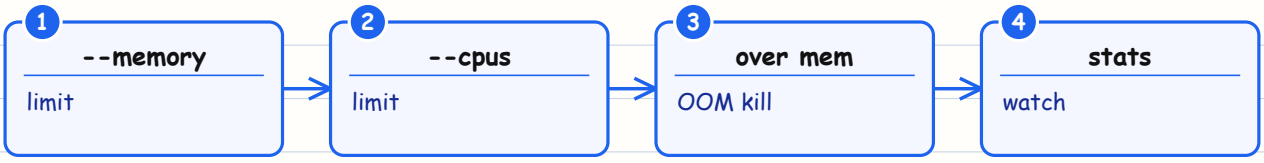
* Cap CPU & Memory

- Use --memory and --cpus to prevent one container from starving the host or its neighbours.
- Without limits, a leak in one container can take down everything else on the host.

* OOM Behaviour

- If a container exceeds its memory limit, the kernel OOM-kills it. Size limits from real usage.

Cap resources



REAL-WORLD EXAMPLE

```
docker run -d --name api \
  --memory=512m --cpus=1.0 shop-api:1.0
docker stats api          # watch actual usage
# in compose: deploy.resources.limits.{cpus,memory}
```

COMMON MISTAKE
Setting a memory limit far below real usage, causing constant OOM kills and restart loops. Measure typical + peak usage first, then set limits with headroom.

REVISION SNAPSHOT

ASK YOURSELF	What happens when a container exceeds its --memory limit?
DO THIS	Run a container with a tight memory limit and observe an OOM kill.

20. Healthchecks & Restart Policy

GrowthSchool

IN SIMPLE TERMS
A healthcheck lets Docker test if a container is truly working; a restart policy tells Docker whether to restart it if it crashes.

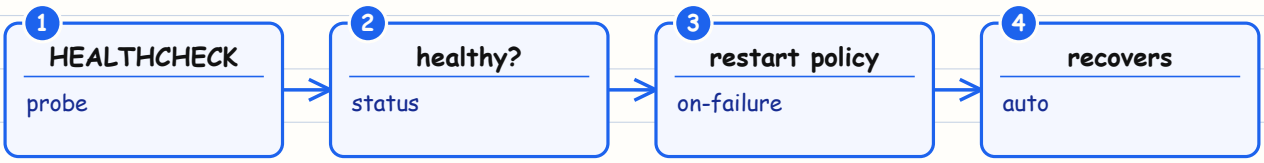
* Healthchecks

- A HEALTHCHECK lets Docker mark a container healthy/unhealthy based on a probe command.
- Orchestrators and Compose can wait for healthy before routing traffic or starting dependents.

* Restart Policies

- --restart=on-failure retries crashed containers; unless-stopped survives daemon restarts.
- Restart policy is basic self-healing for single hosts; Kubernetes does this at cluster scale.

Self-healing



REAL-WORLD EXAMPLE

```
docker run -d --restart=unless-stopped \
  --health-cmd="curl -f http://localhost:8080/health || exit 1" \
  --health-interval=10s shop-api:1.0
docker inspect --format "{{.State.Health.Status}}" api
```

COMMON MISTAKE
Using --restart=always for a container that crashes instantly on a config error - it just restart-loops forever. Fix the root cause; use on-failure with a sane backoff.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a healthcheck and a restart policy?
DO THIS	Add a healthcheck to the Shop API and watch its status change.

21. Private Registries & Auth

GrowthSchool

IN SIMPLE TERMS
A private registry stores your company's images securely; you log in with credentials before pushing or pulling.

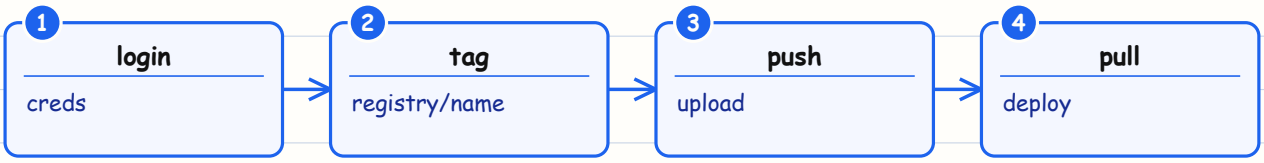
* Why Private

- Companies push internal images to a private registry (Docker Hub private, GHCR, ECR, GCR, ACR).
- Access is controlled with credentials or cloud IAM, keeping proprietary code off public hubs.

* Login & Tag

- Tag images with the registry hostname, docker login, then push. CI usually uses a robot token.

Private registry



PROVIDER-SPECIFIC
Registry auth differs per provider (ECR, GCR, ACR, GHCR). The docker tag/ push flow is the same; only the login step changes.

REAL-WORLD EXAMPLE

```
docker login registry.example.com
docker tag shop-api:1.0 registry.example.com/shop/shop-api:1.0
docker push registry.example.com/shop/shop-api:1.0
# cloud examples (provider-specific auth):
# aws ecr get-login-password | docker login --username AWS ...
```

REVISION SNAPSHOT

ASK YOURSELF	What must a private image name include that a public one may omit?
DO THIS	Push an image to any private registry and pull it from another machine.

22. Docker in CI/CD

GrowthSchool

IN SIMPLE TERMS
Using Docker in CI/CD means your pipeline builds one image, tags it, scans it, and ships that same image to every environment.

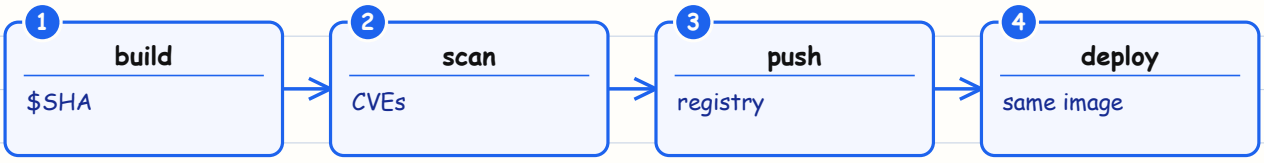
* Build Once, Deploy Many

- CI builds a single immutable image tagged with the commit SHA, then pushes it to a registry.
- The exact same artifact is promoted through staging and production - no rebuilds.

* Speed & Safety

- Cache layers between CI runs; scan the image; only push on green tests.
- Use build args for non-secret build config; use secret stores for credentials.

Build once, ship



REAL-WORLD EXAMPLE

```
docker build -t registry/shop-api:$GIT_SHA .
docker scout cves registry/shop-api:$GIT_SHA
docker push registry/shop-api:$GIT_SHA
# deploy step references the SAME immutable tag
```

COMMON MISTAKE
Rebuilding the image separately for each environment. That risks "works in staging, breaks in prod". Build one artifact, tag by SHA, and promote it unchanged.

REVISION SNAPSHOT

ASK YOURSELF	Why tag CI images by commit SHA instead of latest?
DO THIS	Build and push an image tagged by a fake commit SHA in a script.

23. Data & Persistence Patterns

GrowthSchool

IN SIMPLE TERMS
Persistence patterns are the rules for keeping data safe: keep apps stateless and put data in volumes or managed services.

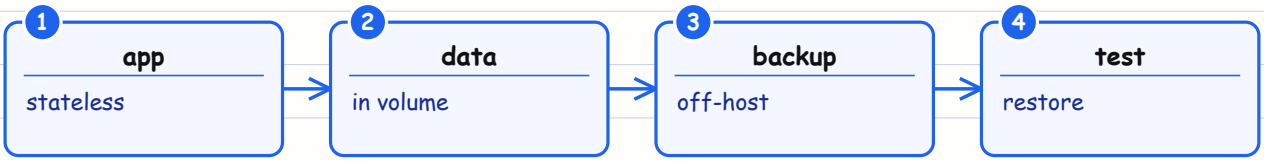
* Stateless vs Stateful

- Keep app containers stateless so they are disposable and horizontally scalable.
- Push state (DB, files, cache) into volumes or managed external services.

* Backups

- A volume is not a backup. Snapshot or dump volume data on a schedule and test restores.
- For heavy production databases, prefer a managed database over self-hosting in a container.

Keep state safe



REAL-WORLD EXAMPLE

```
# back up a postgres volume via a throwaway container
docker run --rm -v shopdata:/data -v ${PWD}:/backup alpine \
  tar czf /backup/shopdata.tgz -C /data .
# restore reverses the tar into the volume
```

COMMON MISTAKE
Treating a Docker volume as a durable backup. Disks fail. Export volume data off-host and periodically test that you can actually restore it.

REVISION SNAPSHOT

ASK YOURSELF	Why keep application containers stateless?
DO THIS	Back up a named volume to a tar file, then restore it into a fresh volume.

24. Best Practices

GrowthSchool

IN SIMPLE TERMS
Best practices are the proven habits - one process per container, small pinned base, non-root, no secrets baked in - that keep images clean.

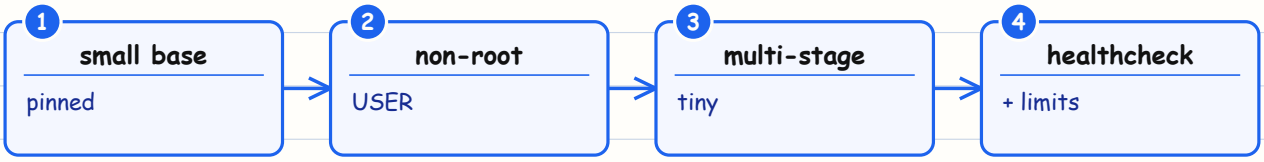
* Images

- One process per container; small trusted base; pin versions; non-root; .dockerignore; multi-stage.
- Log to stdout/stderr so a collector can pick logs up centrally.

* Operations

- Set resource limits and healthchecks; store data in volumes; never bake secrets into images.
- Keep the same image across environments and inject config at runtime.

Best-practice image



CHECKLIST
Small base - pinned tag - non-root USER - .dockerignore - multi-stage - healthcheck - resource limits - stdout logs - config via env - no secrets in layers.

COMMON MISTAKE
Cramming multiple services (nginx + app + cron) into one container. You lose independent scaling, restarts and clean logs. One concern per container.

REVISION SNAPSHOT

ASK YOURSELF	Recite the image checklist from memory.
DO THIS	Audit one of your Dockerfiles against every item in the checklist.

25. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm way to fix common Docker errors like port clashes, out-of-space, and build failures.

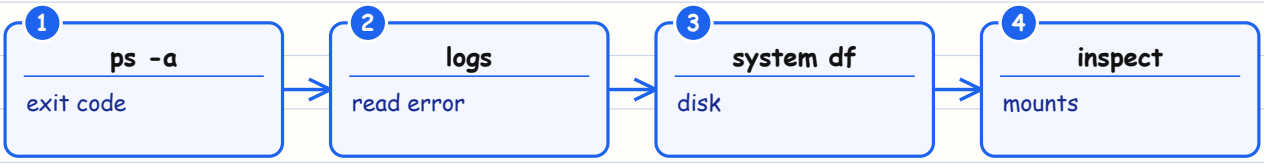
* Common Errors

- port is already allocated: another process/container uses that host port - change -p or free it.
- no space left on device: prune dangling images/volumes with docker system prune.

* More Fixes

- Cannot connect to the Docker daemon: the daemon is not running or you lack permissions.
- COPY failed / file not found: the file is outside the build context or .dockerignore excludes it.

Fix common errors



REAL-WORLD EXAMPLE

```
docker ps -a                  # find exit codes
docker logs <container>       # read the error
docker system df              # what is using disk
docker system prune -a        # reclaim space (careful!)
docker inspect <container>    # full config + mounts
```

COMMON MISTAKE
Running docker system prune -a on a shared/production host without thinking. It deletes all unused images and can remove data you still need. Understand the flags first.

REVISION SNAPSHOT

ASK YOURSELF	What causes "port is already allocated" and how do you fix it?
DO THIS	Reproduce a build "file not found" by excluding a file in .dockerignore.

26. Shop API Docker Project

GrowthSchool

IN SIMPLE TERMS
This page builds the complete Shop API Docker project end to end, from Dockerfile to a running Compose stack.

* Build the Image

- Write a multi-stage Dockerfile with a non-root user, healthcheck and pinned base image.
- Add a .dockerignore and build a SHA-tagged image.

* Run the Stack

- Use Compose to run api + postgres on a network with a named volume for the database.
- Confirm the API is healthy and can read/write the database.

REAL-WORLD EXAMPLE

```
docker build -t shop-api:$(git rev-parse --short HEAD) .
docker compose up -d --build
curl -f http://localhost:8080/health
docker compose exec db psql -U shop -c "\dt"
```

STUDENT LAB
Deliverables: a multi-stage image under 200 MB, running as non-root, with a passing healthcheck,
plus a compose stack whose database survives docker compose down (without -v).

REVISION SNAPSHOT

ASK YOURSELF	Does your Shop API image run as non-root with a healthcheck?
DO THIS	Complete the build + compose deliverables and verify data persistence.

27. Docker Cheat Sheet

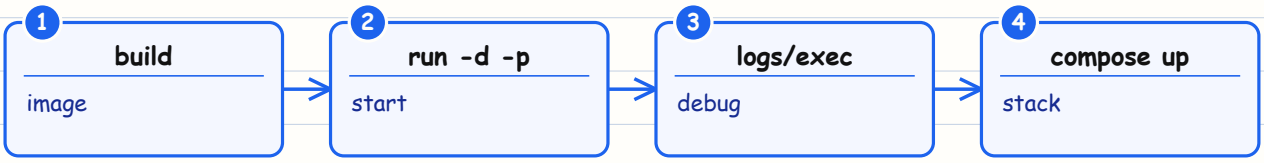
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Docker commands you will use every day.

* Daily Drivers

- build, run -d -p -e, ps -a, logs -f, exec -it, inspect, stats, compose up/down, image prune.

Everyday Docker



DEBUG LOOP
ps -a -> logs -> exec -> inspect. These four answer "why is my container misbehaving?" in almost every case.

ONE-PAGE COMMAND REFERENCE

```
docker build -t app:1.0 .          docker logs -f api
docker run -d -p 80:80 app:1.0     docker exec -it api sh
docker ps -a                       docker inspect api
docker stop/start/rm api           docker stats
docker images / rmi                docker volume ls
docker compose up -d --build       docker network ls
docker compose logs -f api         docker system prune
docker push/pull registry/img      docker history app:1.0
```

STUDENT LAB
From memory: build an image, run it with a published port and an env var, tail its logs, shell into it, then remove it. Repeat until fluent.

REVISION SNAPSHOT

ASK YOURSELF	Which four commands debug almost any container issue?
DO THIS	Recall ten Docker commands from memory without looking.

28. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by packaging and running the Shop API the professional way.

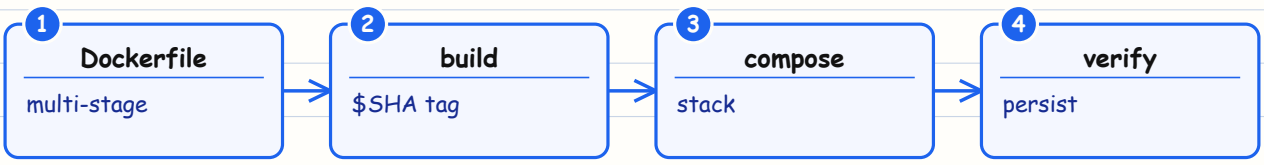
* Package the Shop API End to End

- Multi-stage Dockerfile, non-root, healthcheck, .dockerignore, pinned base, SHA tag.
- Compose stack: nginx -> shop-api -> postgres, one network, named volume for data.

* Prove It Works

- Publish the image to a registry, then deploy the same tag on a second machine.
- Show resource limits, a passing healthcheck and persistent data across restarts.

Capstone steps



REAL-WORLD EXAMPLE

```
docker build -t registry/shop-api:$SHA .
docker push registry/shop-api:$SHA
docker compose up -d
docker compose down && docker compose up -d   # data persists
```

STUDENT LAB
Stretch: add a CI script that builds, scans and pushes the image only when tests pass, then writes the image digest to a file for the deploy step to consume.

REVISION SNAPSHOT

ASK YOURSELF	Is your final image small, non-root, healthchecked and reproducible?
DO THIS	Complete every capstone deliverable and document the final image size.

29. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Image vs container; layer caching; CMD vs ENTRYPOINT; volume vs bind mount; bridge vs host network.
- Why multi-stage builds, why non-root, and why one process per container.

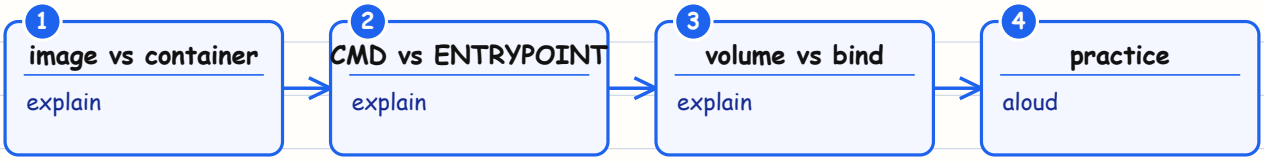
* Common Questions

- How do you shrink an image? Where does container data go? How do containers talk to each other?
- How do you pass secrets safely? What does docker build cache and how do you bust it well?

* Your Next Step

- Containerise a real app of your own, then optimise the image and write a compose stack for it.

Revise out loud



REAL-WORLD EXAMPLE

```
# 30-second self test
docker history <img>   # can you explain every layer?
docker inspect <ctr>   # where are its mounts + network?
# CMD vs ENTRYPOINT? volume vs bind mount? bridge vs host?
```

REVISION SNAPSHOT

ASK YOURSELF	Can you explain image vs container to a beginner in 30 seconds?
DO THIS	Answer every "Common Question" above out loud without notes.