

docker//**field-manual**

CONTAINER REFERENCE

by @contenintech

FOUNDATIONS

- 01 · Docker Architecture
- 02 · Images vs Containers

BUILDING IMAGES

- 03 · Dockerfile Instructions
- 04 · Building & Layer Caching
- 05 · Multi-Stage Builds

RUNNING CONTAINERS

- 06 · Running & Managing Containers
- 07 · Volumes & Persistent Data
- 08 · Networking
- 09 · Environment Variables & Secrets

MULTI-CONTAINER APPS

- 10 · Docker Compose

OPERATIONS

- 11 · Registries & Docker Hub
- 12 · Resource Limits & Health Checks
- 13 · Security Best Practices
- 14 · Debugging & Troubleshooting

REFERENCE

- 15 · Essential Command Reference

```
~/app (main) $ docker build -t app .
```

The **Docker** Field Manual

Everything an SDE needs to package, run, and ship software in containers — architecture, Dockerfiles, images, volumes, networking, Compose, and the operational habits that keep containers safe and debuggable in production.

01 **Docker** Architecture

The pieces that work together every time you run a container.

Client-Server Model

CONCEPT

The docker command you type is a client that talks to the **Docker daemon** (dockerd), a background service that does the actual work of building images and running containers — usually over a local Unix socket, but it can be remote too.

docker CLI (client)

→

Docker daemon (dockerd)

→

containerd

→

runc (creates the container)

Under the hood, the daemon delegates to **containerd** (manages container lifecycle) which delegates to **runc** (the low-level tool that actually creates a container using Linux namespaces and cgroups, per the OS-level virtualization chapter).

COMPONENT	ROLE
Docker Daemon	Background service managing images, containers, networks, volumes.
Docker Client	The CLI you type commands into.
Docker Registry	Stores and distributes images (Docker Hub, ECR, GCR, private registries).
Image	A read-only template — filesystem + metadata — used to create containers.
Container	A running (or stopped) instance of an image.

02Images vs Containers

The core distinction the rest of Docker builds on.

	IMAGE	CONTAINER
What it is	A read-only, layered snapshot of a filesystem plus metadata	A running process using that image, with its own writable layer
Analogy	A class / a recipe	An instance of that class / a dish made from the recipe
Mutability	Immutable — never changes once built	Mutable while running; changes are lost when removed unless committed or persisted via a volume
Lifecycle	Built once, reused many times	Created, started, stopped, removed

Image Layers

CONCEPT

An image is built as a stack of read-only layers, each corresponding to one Dockerfile instruction. Layers are cached and shared across images — if two images share the same base layers, Docker stores that data only once. A running container adds one thin writable layer on top, using copy-on-write: reads fall through to the read-only layers, writes only touch the top layer.

03Dockerfile Instructions

The recipe describing how to build an image, instruction by instruction.

A complete example

EXAMPLE

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --production

COPY . .

ENV NODE_ENV=production
EXPOSE 3000

USER node
HEALTHCHECK --interval=30s CMD wget -q --spider http://localhost:3000/health || exit 1

CMD ["node", "server.js"]
```

INSTRUCTION	WHAT IT DOES
FROM	Sets the base image everything else builds on top of — the only mandatory instruction.
WORKDIR	Sets the working directory for subsequent instructions (and the container's default directory).
COPY	Copies files from the build context into the image.
ADD	Like COPY, but also auto-extracts tar files and can fetch remote URLs — prefer COPY unless you need those extras.
RUN	Executes a command at build time, creating a new layer (e.g. installing packages).
ENV	Sets an environment variable available at build time and in the running container.
ARG	Defines a build-time-only variable, passed via --build-arg — not present in the final container.
EXPOSE	Documents which port the app listens on — informational only, doesn't actually publish the port.
VOLUME	Marks a directory as a mount point for external/persistent storage.
USER	Sets which user subsequent instructions and the container process run as — avoid running as root.
CMD	The default command run when the container starts — overridable at docker run.
ENTRYPOINT	The fixed command that always runs; CMD's arguments (if any) are passed to it — harder to override than CMD alone.
LABEL	Attaches arbitrary key-value metadata to the image.
HEALTHCHECK	Defines a command Docker runs periodically to decide if the container is healthy.

CMD vs ENTRYPOINT

CONCEPT

Both define what runs when the container starts, but they combine differently:

```
-- CMD alone: fully replaced by any command passed to `docker run`
CMD ["node", "server.js"]
-- `docker run myimage npm test` replaces the whole CMD with `npm test`

-- ENTRYPOINT + CMD: CMD becomes default *arguments* to ENTRYPOINT
ENTRYPOINT ["node"]
CMD ["server.js"]
-- `docker run myimage app.js` runs `node app.js` instead of `node server.js`
```

04 Building & Layer Caching

Turning a Dockerfile into a usable image efficiently.

docker build

BUILD

```
docker build -t myapp:1.0 .           # build, tag it myapp:1.0, context = current dir
docker build -t myapp:latest -f Dockerfile.prod .  # use a specific Dockerfile
docker build --build-arg VERSION=2.1 .    # pass a build-time ARG
docker build --no-cache .                 # ignore layer cache entirely
```

Layer caching

CONCEPT

Docker caches each layer and reuses it on the next build if nothing that layer depends on has changed. The moment one instruction's inputs change, that layer and every layer after it must rebuild — which is why **instruction order matters**.

```
-- GOOD: dependency files copied and installed before the rest of the source
COPY package*.json ./
RUN npm ci
COPY . .
-- changing application code doesn't invalidate the (slow) npm ci layer

-- BAD: copying everything first busts the cache on every code change
COPY . .
RUN npm ci
```

Rule of thumb: order instructions from least-frequently-changing to most-frequently-changing.

.dockerignore

CONFIG

Excludes files from the build context, keeping images smaller and builds faster — same syntax as `.gitignore`.

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

05 Multi-Stage Builds

Keeping build tools out of the final image.

Why multi-stage builds

CONCEPT

Compiling an app often needs a heavy toolchain (compilers, dev dependencies) that the final running image doesn't need at all. Multi-stage builds let you use one image to build the app, then copy only the compiled output into a clean, minimal final image — dramatically reducing final image size and attack surface.

```
# --- build stage ---
FROM golang:1.22 AS builder
WORKDIR /src
COPY . .
RUN go build -o /app

# --- final stage ---
FROM alpine:3.19
COPY --from=builder /app /app
ENTRYPOINT ["/app"]
```

The final image contains only alpine + the compiled binary — no Go toolchain, no source code, often shrinking a 1GB+ build image down to a few MB.

06 Running & Managing Containers

The day-to-day commands for working with running containers.

docker run

RUN

docker run nginx	# run in the foreground
docker run -d nginx	# detached (background)
docker run -d -p 8080:80 nginx	# map host:8080 -> container:80
docker run -d --name web nginx	# give it a name
docker run -it ubuntu bash	# interactive terminal
docker run --rm alpine echo hi	# auto-remove when it exits
docker run -e API_KEY=xyz myapp	# set an environment variable

Managing the lifecycle

COMMANDS

docker ps	# running containers
docker ps -a	# all containers, including stopped
docker stop <id/name>	# graceful stop (SIGTERM, then SIGKILL after a timeout)
docker start <id/name>	# start a stopped container
docker restart <id/name>	
docker rm <id/name>	# remove a stopped container
docker rm -f <id/name>	# force-remove a running container

Inspecting & interacting

COMMANDS

```
docker logs <name>           # view stdout/stderr
docker logs -f <name>        # follow logs live
docker exec -it <name> bash   # open a shell inside a running container
docker inspect <name>        # full JSON metadata (IP, mounts, env, etc.)
docker stats                  # live CPU/memory usage per container
docker top <name>             # processes running inside the container
```

07 Volumes & Persistent Data

Containers are ephemeral by default — volumes are how data survives.

TYPE	DESCRIPTION	USE CASE
Volume	Managed entirely by Docker, stored under Docker's own directory	Databases, persistent app data — the recommended default
Bind mount	Maps a specific host path directly into the container	Local development — live-editing source code
tmpfs mount	Stored only in host memory, never written to disk	Sensitive temporary data

```
docker volume create mydata
docker run -d -v mydata:/var/lib/mysql mysql      # named volume
docker run -d -v $(pwd):/app node                  # bind mount current directory
docker run -d --mount type=volume,src=mydata,dst=/data myapp # explicit --mount syntax

docker volume ls
docker volume inspect mydata
docker volume rm mydata
```

Without a volume, anything a container writes disappears the moment it's removed — this is intentional, and part of why containers are treated as disposable/replaceable.

08 Networking

How containers talk to the host, the internet, and each other.

NETWORK DRIVER	BEHAVIOR
bridge	The default — containers get a private internal IP and reach the outside world via NAT.
host	Container shares the host's network stack directly — no port mapping needed, but no isolation either.
none	No networking at all — fully isolated.
overlay	Connects containers across multiple Docker hosts — used in Swarm/multi-host setups.

```
docker run -p 8080:80 nginx          # host:8080 -> container:80
docker run -p 127.0.0.1:8080:80 nginx # bind only to localhost
docker run -P nginx                  # publish all EXPOSEd ports to random host ports
```

Containers on the same user-defined bridge network can reach each other by container name — Docker runs an internal DNS server that resolves names automatically, unlike the default bridge network.

```
docker network create mynet
docker run -d --name db --network mynet postgres
docker run -d --name api --network mynet myapi
-- inside "api", connecting to "db:5432" just works — no IP addresses needed
```

09 Environment Variables & Secrets

Configuring container behavior without baking values into the image.

```
docker run -e NODE_ENV=production myapp
docker run --env-file .env myapp          # load many vars from a file
```

```
# .env
DATABASE_URL=postgres://localhost/mydb
API_KEY=supersecret
```

Secrets warning: environment variables are visible via `docker inspect` and process listings — for real secrets in production, prefer a dedicated secrets manager (Docker Swarm secrets, Kubernetes Secrets, Vault, cloud secret managers) over plain env vars.

10 Docker Compose

Defining and running a multi-container application as one unit.

```
services:
  web:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
    depends_on:
      - db
    volumes:
      - ./app

  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - dbdata:/var/lib/postgresql/data

volumes:
  dbdata:
```

One file declares every service, its image/build, ports, environment, volumes, and dependencies — Compose creates a shared network so services can reach each other by name automatically.

Compose commands

COMMANDS

```
docker compose up           # start everything, foreground
docker compose up -d        # start everything, detached
docker compose up --build   # rebuild images before starting
docker compose down         # stop and remove containers/networks
docker compose down -v      # also remove named volumes
docker compose logs -f web  # follow logs for one service
docker compose exec web bash # shell into a running service
docker compose ps           # status of each service
```

11 Registries & Docker Hub

Storing and sharing images.

```
docker login                # authenticate to Docker Hub
docker login myregistry.example.com # authenticate to a private registry

docker tag myapp:latest username/myapp:1.0 # tag before pushing
docker push username/myapp:1.0
docker pull username/myapp:1.0

docker tag myapp:latest myregistry.example.com/myapp:1.0
docker push myregistry.example.com/myapp:1.0
```

An image name has an implicit structure: [registry/]repository[:tag] — no registry prefix means Docker Hub by default.

12Resource Limits & Health Checks

Keeping one container from starving the whole host, and knowing when it's actually working.

Resource limits

CONFIG

```
docker run -d --memory=512m --cpus=1.5 myapp
# hard memory cap of 512MB, up to 1.5 CPU cores
```

These limits are enforced via the same cgroups mechanism covered in the OS containers chapter — exceeding the memory limit gets the container's process killed (OOM-killed), not gracefully throttled.

Health checks

CONFIG

```
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1
```

Docker runs this command periodically and marks the container healthy/unhealthy in `docker ps` — orchestrators like Kubernetes/Swarm use this signal to decide whether to restart or stop routing traffic to a container.

13Security Best Practices

Containers share the host kernel — these habits keep that isolation meaningful.

PRACTICE	WHY
Don't run as root	Use USER in the Dockerfile — limits the blast radius if the container is compromised.
Use minimal base images	Alpine or distroless images have a smaller attack surface and fewer CVEs than a full OS image.
Pin image versions	Avoid :latest in production — it's not reproducible and can silently change.
Scan images for vulnerabilities	Tools like <code>docker scout</code> , Trivy, or Snyk catch known CVEs in dependencies/base images.
Don't bake secrets into images	Anything in a layer is recoverable even if deleted in a later layer — use secrets managers or build-time secrets mounts instead.
Read-only filesystem where possible	<code>docker run --read-only</code> prevents a compromised process from modifying the container's own filesystem.
Drop unnecessary capabilities	<code>--cap-drop=ALL --cap-add=NET_BIND_SERVICE</code> grants only what's actually needed.

14Debugging & Troubleshooting

What to check when a container won't start or behaves unexpectedly.

```
docker logs <name>                # check what the app actually printed
docker inspect <name> --format '{{.State.Status}}' # quick status check
docker events                      # live stream of Docker daemon events
docker system df                   # disk space used by images/containers/volumes
docker system prune                # clean up unused containers/images/networks
docker system prune -a --volumes  # aggressive cleanup, including unused volumes

# a container that exits immediately: run it interactively to see why
docker run -it --entrypoint sh myapp
```

Common gotcha: a container that exits right after starting usually means its main process (PID 1) exited — often because the process was meant to run in the foreground but the image backgrounds it, or a startup script failed silently. Check `docker logs` first, always.

15 Essential Command Reference

The full toolbox in one place.

Images

[REFERENCE](#)

```
docker images          # list local images
docker rmi <image>     # remove an image
docker history <image> # show layer history
docker save -o out.tar myapp # export image to a tar file
docker load -i out.tar  # import an image from a tar file
```

Containers

[REFERENCE](#)

```
docker cp <name>:/app/file.txt . # copy a file out of a container
docker diff <name>                # show filesystem changes vs the image
docker commit <name> newimage:tag # snapshot a container's state as a new image
docker pause / unpause <name>    # freeze / resume all processes in a container
```

Cleanup

[REFERENCE](#)

```
docker container prune # remove all stopped containers
docker image prune     # remove dangling (untagged) images
docker volume prune    # remove unused volumes
docker network prune   # remove unused networks
```