Docker CLI — End-to-end Guide

Concise, practical, and command-focused guide covering Docker CLI from basics to advanced. Use this as a reference, classroom handout, or quick cheatsheet.

Table of Contents

---

1. Quick intro & conventions

- This guide focuses on `docker` CLI (commands typed in a terminal).
- Example format: `docker <object> <command> [FLAGS] [ARGUMENTS]`.
- Flags commonly used:
  - `-d` detach (run container in background)
  - `-it` interactive + TTY
  - `-p HOST:CONTAINER` port mapping
  - `-v HOST:CONTAINER` or `--mount` for volumes
  - `--name` friendly container/image name
  - `--rm` remove container after exit
- Use `--help` for command-level help: `docker run --help`.

2. Installation & Setup (CLI-focused)

- Install Docker Desktop (Windows/macOS) or Docker Engine (Linux). After install, verify:
  - `docker --version` — prints client version
  - `docker info` — shows daemon info, running state
- Log in to registries (Docker Hub or private):
  - `docker login` — enter username/password; `--username` to pass username

3. Basics — day-to-day commands

- `docker --help` — top-level help.
- `docker version` — client & server versions.
- `docker info` — detailed daemon info (containers, images, storage driver).
- `docker ps` — list running containers.
  - `docker ps -a` — list all containers (including stopped).
  - `docker ps --filter status=exited` — filter results.
- `docker images` (or `docker image ls`) — list images.
- `docker pull <image[:tag]>` — download image from registry.
- `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]` — create & start container.
  - Example: `docker run --name web -d -p 8080:80 nginx:latest`.
- `docker stop <container>` — stop container gracefully.
- `docker start <container>` — start a stopped container.
- `docker restart <container>` — stop and start.
- `docker rm <container>` — delete stopped container. Add `-f` to force.
- `docker rmi <image>` — remove image. Use `docker image prune` to remove dangling images.

4. Images & registries

- `docker pull <image>` — pull image.
- `docker pull alpine:3.18` — pull specific tag.
- `docker image ls` — list images.
- `docker image inspect <image>` — metadata of image (layers, config).
- `docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]` — tag local image for a registry.
- `docker push <registry>/<repo>:<tag>` — push image to registry (after `docker login`).
- `docker save -o file.tar <image>` — export image to tar.
- `docker load -i file.tar` — load image from tar.

- `docker export` / `docker import` — export container filesystem ↔ import as image.

5. Containers — lifecycle & management

- Create & run in one step: `docker run --name myapp -d image`.
- Run and get interactive shell: `docker run -it --rm ubuntu:24.04 /bin/bash`.
- Execute commands inside running container:
  - `docker exec -it <container> /bin/bash` — open shell.
  - `docker exec -it <container> ps aux` — run any command.
- Copy files between host and container:
  - `docker cp hostfile.txt <container>:/path/` and reverse.
- Commit container changes to image:
  - `docker commit <container> myrepo/myimage:tag`.
- Resource constraints when running:
  - `--memory=512m` — limit memory.
  - `--cpus=1.5` — limit CPU shares.
- Restart policies:
  - `--restart no|on-failure[:max-retries]|always|unless-stopped`.

6. Storage — volumes & bind mounts

- Short forms:
  - Bind mount: `-v /host/path:/container/path` (host path persisted).
  - Named volume: `-v name:/container/path` (managed by Docker).
- Create volume: `docker volume create myvol`.
- Inspect volume: `docker volume inspect myvol`.
- Remove volume: `docker volume rm myvol` (only if not used).
- Use `--mount` (recommended for long-form clarity):
  - `--mount type=volume,source=myvol,target=/data`
  - `--mount type=bind,source=/host/path,target=/app`
- Backups & restore patterns: `docker run --rm -v myvol:/data -v $(pwd):/backup alpine tar czf /backup/vol.tar.gz -C /data .`

7. Networking — built-in drivers & commands

- Common drivers: `bridge` (default), `host`, `none`, and `overlay` (multi-host with swarm).
- List networks: `docker network ls`.
- Inspect network: `docker network inspect bridge`.
- Create user-defined bridge: `docker network create mynet`.
- Connect container to network: `docker network connect mynet container`.

- Run with network: `docker run --network mynet --name svc1 nginx`.
- DNS: containers on same user-defined network can reach each other by container name.
- Port mapping: `-p host_port:container_port` — exposes container port.

8. Building images (Dockerfile) & advanced build features

- Build basic image:
  - `docker build -t myapp:1.0 .` — build image from `Dockerfile` in current folder.
- Dockerfile basics (short):
  - `FROM` — base image.
  - `WORKDIR` — set working directory.
  - `COPY` / `ADD` — copy files.
  - `RUN` — run commands during build.
  - `CMD` — default command at container start.
  - `ENTRYPOINT` — container entrypoint.
  - `EXPOSE` — documentation of port.
- Multistage builds (for smaller images):
  - Use `FROM golang:1.XX as builder` then `FROM alpine` and `COPY --from=builder /app /app`.
- Build cache control:
  - `--no-cache` to ignore cache.
  - Use `--target` to build a specific stage.
- BuildKit & buildx (advanced builds & multi-arch):
  - Enable BuildKit: `DOCKER_BUILDKIT=1 docker build .` or `docker buildx build`.
  - `docker buildx create --name mybuilder && docker buildx use mybuilder`.
  - Multi-arch build: `docker buildx build --platform linux/amd64,linux/arm64 -t repo/app:tag --push .`

9. Compose with `docker compose` (CLI plugin)

- Compose V2 integrated into `docker` as `docker compose` (not `docker-compose` Python tool in many installs).
- Basic usage:
  - `docker compose up` — create and start services from `docker-compose.yml`.
  - `docker compose up -d` — run detached.
  - `docker compose down` — stop & remove resources.

    - `docker compose logs -f` — follow service logs.
    - `docker compose exec SERVICE /bin/sh` — run command in a service container.
- Useful flags: `--build` (rebuild images), `--scale SERVICE=N`.
- Compose examples: define `services`, `volumes`, `networks`, and `depends_on`.

10. Debugging, logs & inspection

- Logs:
    - `docker logs <container>` — fetch container logs.
    - `docker logs -f --tail 200 <container>` — follow logs and show last 200 lines.
- Inspect container or image details:
    - `docker inspect <container|image|network>` — JSON output with metadata.
    - Use `--format` to extract fields: `docker inspect --format='{{.State.Running}}' <container>`.
- Check running processes inside container: `docker top <container>`.
- Events stream: `docker events` — real-time low-level events.
- Attach to container STDIN/STDOUT: `docker attach <container>` (use cautiously with PID 1).

11. Advanced CLI features & developer workflow

- Contexts (work with multiple endpoints/hosts):
    - `docker context ls` / `docker context create myctx --docker "host=ssh://user@host"` / `docker context use myctx`.
- Buildx for advanced builds (see section 8).
- `docker run --rm -it --privileged` for containers needing extra capabilities (use sparingly).
- CLI formatting & filtering:
    - `--format` (Go template) e.g. `docker ps --format "{{.ID}}  {{.Names}} {{.Status}}"`.
    - `--filter` e.g. `docker ps --filter "ancestor=nginx"`.
- Inspect layers and sizes: `docker image history <image>`.
- Image signing & content trust (experimental in some setups): `docker trust` commands if configured.
- Use `docker scan <image>` (integrated Snyk scanner in Docker Desktop) to find vulnerabilities.

12. Orchestration: Docker Swarm (brief)

- Initialize swarm: `docker swarm init`.
- Join worker: `docker swarm join --token <token> <manager-ip>:2377`.
- Deploy stack: `docker stack deploy -c docker-compose.yml mystack`.
- Service commands: `docker service ls`, `docker service ps <service>`, `docker service scale svc=3`.
- Note: Kubernetes is the dominant orchestration tool; Swarm is lightweight and simpler for demos.

13. Cleanup, pruning & housekeeping

- Remove dangling images: `docker image prune`.
- Remove unused images, containers, volumes, networks: `docker system prune`.
  - `docker system prune -a` — also removes unused images (careful).
- Remove unused volumes: `docker volume prune`.
- Clean build cache (BuildKit): `docker builder prune`.
- Disk usage summary: `docker system df`.

14. Best practices & tips

- Prefer small, single-responsibility images and multi-stage builds.
- Use named volumes for persistent data and avoid baking runtime state into images.
- Pin image versions (use explicit tags) — avoid `:latest` in production.
- Keep secrets out of Dockerfiles. Use environment variables, secret managers, or `docker secret` in swarm.
- Limit container resources (`--memory`, `--cpus`) in production.
- Use `.dockerignore` to speed builds and reduce context size.
- Avoid running unnecessary `apt-get`/package steps in runtime image — do them in builder stage.
- Rebuild only when necessary; leverage cache layers order for stable layers first.

15. Compact cheatsheet (one-line commands)

- `docker --version` — show Docker version.
- `docker info` — show Docker daemon info.
- `docker pull nginx:latest` — pull image.
- `docker run --name web -d -p 8080:80 nginx` — run nginx detached.
- `docker ps -a` — list all containers.
- `docker logs -f web` — follow logs.

- `docker exec -it web /bin/sh` — open shell inside running container.
- `docker build -t myapp:1.0 .` — build image from Dockerfile.
- `docker image ls` — list images.
- `docker image rm myimage` — remove image.
- `docker system prune -a` — aggressive cleanup.
- `docker compose up -d` — bring up compose services.

Appendix — Example workflows (short)

Developer quick start 1. `git clone repo && cd repo` 2. `docker build -t myapp:dev .` 3. `docker run --rm -it -p 3000:3000 --name myapp myapp:dev` 4. Edit source, rebuild, restart container.

Local multi-service with compose 1. `docker compose up --build` (starts DB + API + frontend) 2. `docker compose exec api pytest` (run tests inside service) 3. `docker compose down -v` (stop & remove volumes)