Docker

Run, inspect and manage containers and images.

01 Install & verify

```
brew install --cask docker
winget install -e --id Docker.DockerDesktop
docker version    # Client and Server
docker info
docker run --rm hello-world
```

Docker Desktop includes the CLI, daemon and Compose. On Linux, install Docker Engine.

02 Run a container

```
docker run nginx                # foreground
docker run -d --name web -p 8080:80 nginx
docker run -it ubuntu bash      # shell
docker run --rm hello-world     # then remove
docker run -w /app node node server.js
```

Arguments after the image replace its default command. Name containers so later commands can use the name.

03 docker run options

`-d`	Run in the background
`-it`	Interactive terminal
`--name web`	Name the container
`-p 8080:80`	Publish host:container port
`-v data:/data`	Mount a volume or host path
`-e KEY=value`	Set a variable; --env-file .env
`--rm`	Remove the container on exit
`--restart`	always, unless-stopped or on-failure:3

-m 512m and --cpus="1.5" cap memory and CPU.

04 List, stop & remove

```
docker ps                     # running
docker ps -a                  # all
docker stop web               # graceful
docker start web
docker rm web
docker rm -f web              # force
docker stop $(docker ps -q)   # stop all
```

stop -t 30 sets the timeout; kill stops at once. docker container prune removes all stopped ones.

05 Logs & shell access

```
docker logs web
docker logs -f --tail 50 web   # follow
docker logs --since 30m web
docker exec -it web sh         # shell
docker exec -u root web whoami
docker exec -d web npm run migrate
```

Use sh on Alpine images, bash on Debian or Ubuntu. Container will not start? Try docker run -it --entrypoint sh myapp.

06 Inspect, stats & copy

```
docker inspect web
docker inspect -f '{{.State.Status}}' web
docker stats --no-stream   # CPU, memory
docker top web             # processes
docker port web            # port mappings
docker cp web:/app/logs ./logs
docker cp ./config.yml web:/app/config.yml
```

docker cp works on a running container, so you can grab logs or inject config without rebuilding.

07 Build an image

```
docker build -t myapp .
docker build -t myapp:v1 .
docker build -f Dockerfile.dev -t app:dev .
docker build --build-arg VERSION=1.0 .
docker build --no-cache --pull -t myapp .
docker build --target production -t myapp .
```

The final . is the build context. Add --progress=plain to see full output when a step fails.

08 Images & registries

```
docker images                  # list
docker pull nginx:alpine
docker login                   # Docker Hub
docker tag myapp username/myapp:v1
docker push username/myapp:v1
docker rmi myapp:v1
docker save -o myapp.tar myapp:v1
```

Use version tags, not latest. For a private registry, put its host in the tag: myregistry.azurecr.io/myapp. docker load -i myapp.tar restores a saved image.

09 Volumes & bind mounts

```
docker volume create dbdata
docker volume ls
docker run -v dbdata:/var/lib/mysql mysql
docker run -v $(pwd):/app node   # bind
docker run -v dbdata:/data:ro nginx
docker volume rm dbdata
```

Named volumes keep data after the container is gone. Use bind mounts in development; --mount source=dbdata,target=/data is the explicit form.

Docker

Build images, clean up and run multi-container apps.

10 Dockerfile basics

```
FROM node:22-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

Install dependencies first so that layer stays cached.

11 Dockerfile instructions

FROM	Base image; pin a tag, not latest
WORKDIR	Directory for the steps that follow
RUN	Run a command at build time
COPY	Copy files from the build context
ENV	Variable kept in the container
ARG	Build-time only variable
EXPOSE	Documents a port; -p publishes it

Chain related commands in one RUN with && to cut layers.

12 User, entrypoint & health

```
RUN addgroup -S app && adduser -S app -G app
USER app
ENTRYPOINT ["node"]
CMD ["server.js"]
HEALTHCHECK --interval=30s --retries=3 CMD \
  curl -f http://localhost:3000 || exit 1
```

CMD gives default arguments to ENTRYPOINT: docker run myapp worker.js runs node worker.js. Run as a non-root USER.

13 Multi-stage builds

```
FROM node:22-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
FROM node:22-alpine AS production
WORKDIR /app
COPY --from=build /app/dist ./dist
COPY --from=build /app/node_modules \
  ./node_modules
CMD ["node", "dist/server.js"]
```

Only the final stage ships; build tools stay behind.

14 .dockerignore

```
# Same syntax as .gitignore
node_modules
.git
.env*
dist
```

Keeps the build context small and secrets out of the image.

15 Disk cleanup

```
docker system df          # disk usage
docker image prune -a     # unused images
docker volume prune       # unused volumes
docker builder prune      # build cache
docker system prune
docker system prune -af --volumes
```

Plain system prune keeps volumes and tagged images; -a and --volumes remove unused ones too.

16 Networks

```
docker network create mynet
docker run -d --network mynet --name db \
  postgres
docker network connect mynet web
```

Containers on a custom network reach each other by name.

17 compose.yaml

```
services:
  app:
    build: .
    ports: ["3000:3000"]
    depends_on:
      db:
        condition: service_healthy
  db:
    image: postgres:17-alpine
    volumes:
      - db-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready"]
volumes:
  db-data:
```

app waits until db passes its healthcheck.

18 Compose commands

```
docker compose up -d     # --build rebuilds
docker compose ps
docker compose logs -f app
docker compose exec app sh
docker compose down      # stop and remove
```

down -v deletes named volumes and their data.