Docker na Prática

Comandos, conceitos e exemplos funcionais (CLI, Dockerfile e Compose)

Objetivos

O que você vai dominar ao final

Você vai aprender

- Como o Docker funciona (cliente, daemon, objetos e registries)
- Comandos essenciais e flags mais usadas no dia a dia
- Criação de imagens com Dockerfile (build, cache, layers)
- Persistência com volumes e bind mounts
- Redes, isolamento e comunicação entre containers
- Docker Compose para ambientes multi-serviço
- Limpeza, troubleshooting e boas práticas de segurança

Formato sugerido

- Use os slides como referência rápida + exemplos para copiar/colar
- Para aprender mais rápido: execute os exemplos em um projeto de teste
- Ao final, há um “cheat sheet” e labs guiados (passo a passo)

Fundamentos

Conceitos e arquitetura para entender o “porquê” dos comandos

O que é Docker?

Containerização para empacotar e executar apps com consistência

Por que usar

- Ambiente reprodutível (dev, CI e produção mais parecidos)
- Isolamento de dependências (sem “na minha máquina funciona”)
- Start rápido e bom aproveitamento de recursos
- Distribuição via imagens versionadas (tags/digests)

Quando faz mais sentido

- APIs e serviços web
- Jobs e ferramentas de build/teste
- Bancos e dependências locais para desenvolvimento
- Ambientes multi-serviço (Compose)

Conceitos-chave

Vocabulário do Docker (para ler a CLI sem dor)

Imagem

- Template imutável (camadas)
- Identificada por nome:tag ou digest
- Base para criar containers

Container

- Instância em execução de uma imagem
- Isolada via namespaces/cgroups
- Ciclo de vida: create → start → stop → rm

Registry

- Repositório de imagens
- Ex.: Docker Hub
- pull/push para distribuir versões

Volume

- Persistência gerenciada pelo Docker
- Compartilhável entre containers
- Ideal p/ dados de banco

Bind mount

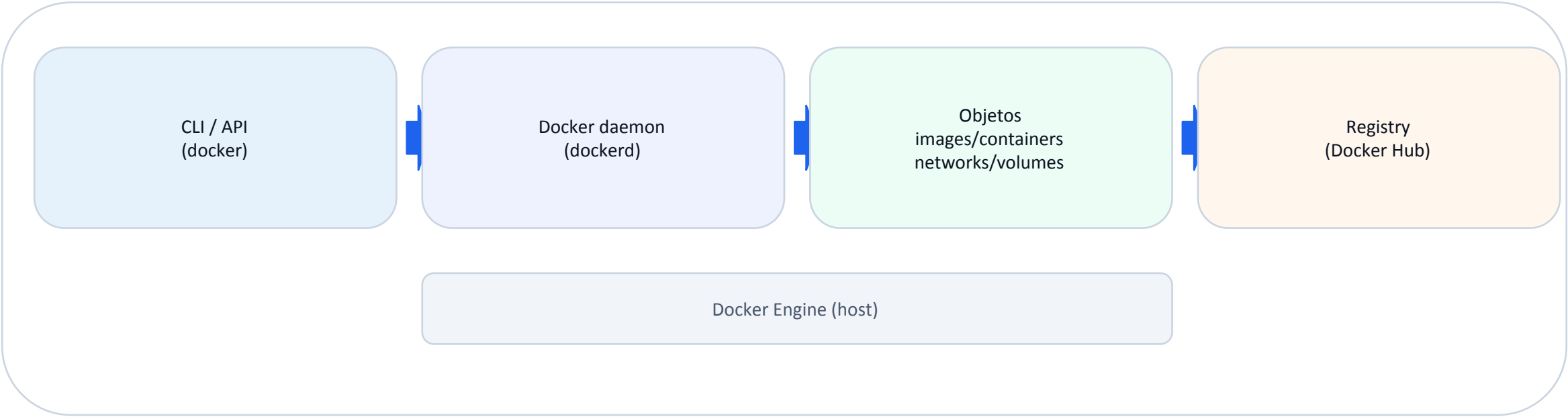
- Pasta do host montada no container
- Ótimo para dev (hot reload)
- Cuidado com permissões

Network

- Comunicação e DNS entre containers
- Bridge padrão no host
- Compose cria rede por projeto

Arquitetura do Docker

Como a CLI fala com o daemon e manipula objetos



Leitura rápida

- O comando `docker ...` é o cliente (CLI) que chama a API do daemon (dockerd).
- O daemon cria e gerencia objetos: imagens, containers, redes e volumes.
- Registries armazenam imagens para pull/push (ex.: Docker Hub, GHCR, ECR).

Instalação e Primeiros Passos

Garantindo que tudo funciona antes de construir imagens e rodar serviços

Instalação (alto nível)

Docker Desktop (Windows/macOS) ou Docker Engine (Linux)

Docker Desktop

- Inclui UI + Engine + Compose
- Ideal para dev em Windows/macOS
- Integrações: Kubernetes (opcional), extensions, etc.

Dica: prefira seguir a documentação oficial do seu SO (passos mudam com o tempo).

Docker Engine (Linux)

- Instala o daemon (dockerd) e CLI no host
- Boa opção para servidores e VMs
- Gerencie permissões (grupo docker) com cuidado

Verificando a instalação

Comandos mínimos para checar versão, daemon e um container de teste

Terminal (bash)

```
# Versões do cliente e do daemon
docker version

# Informações do host/engine
docker info

# Primeiro container (baixa a imagem e roda)
docker run --rm hello-world
```

O que observar

- Se `hello-world` rodar, a comunicação CLI ↔ daemon está OK.
- Se houver erro de permissão no Linux, verifique grupo `docker` ou use `sudo`.
- Se a imagem não baixar, cheque proxy/firewall e DNS.

Imagens e Dockerfile

Baixar, inspecionar, construir e versionar imagens (com exemplos funcionais)

Comandos de imagem (essenciais)

Do download à inspeção

Terminal (bash)

```bash
# Baixar uma imagem
docker pull nginx:alpine

# Listar imagens locais
docker images

# Ver detalhes (JSON)
docker image inspect nginx:alpine

# Ver histórico de camadas
docker history nginx:alpine
```

Boas práticas rápidas

- Use tags explícitas (evite depender apenas de `latest`).
- Quando precisar reprodutibilidade forte, use digest (ex.: `@sha256:...`).
- Inspecione a imagem para descobrir portas expostas, entrypoint, labels, etc.

Dockerfile (exemplo funcional)

API Python simples (Flask)

Dockerfile

```dockerfile
# Dockerfile
FROM python:3.13-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

ENV PORT=8000
EXPOSE 8000

CMD ["python", "app.py"]
```

Arquivos da aplicação

```
# requirements.txt
flask==3.0.3

# app.py
from flask import Flask

app = Flask(__name__)

@app.get('/')
def home():
    return {'status': 'ok'}

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Build e execução:

Terminal

```
docker build -t minha-api:1.0 .

docker run --rm -p 8000:8000 --name api minha-api:1.0

# Teste
curl http://localhost:8000/
```

Build, tag e push

Criando e publicando versões

Terminal (bash)

```bash
# Build (diretório atual)
docker build -t usuario/minha-api:1.0 .

# Tag extra (ex.: latest ou release)
docker tag usuario/minha-api:1.0 usuario/minha-api:latest

# Login e push
docker login
docker push usuario/minha-api:1.0
docker push usuario/minha-api:latest
```

Dicas de versionamento

- Use tags semânticas (ex.: 1.2.0) e automatize no CI.
- Para deploys críticos, prefira referenciar imagens por digest.
- Atenção a credenciais: use tokens e evite commit de secrets no repositório.

Camadas (layers) e cache de build

Por que a ordem do Dockerfile importa

Como o cache funciona

- Cada instrução no Dockerfile gera uma camada (quando aplicável).
- Se uma camada muda, as seguintes precisam ser reconstruídas.
- Organize o Dockerfile para maximizar cache (deps antes do código).

Exemplos de otimização

- Copie `requirements.txt` e instale deps antes de copiar o resto do código.
- Use `.dockerignore` para não enviar arquivos inúteis ao build context.

Comparação (exemplo)

```
# Melhor para cache
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .

# Pior para cache
COPY . .
RUN pip install -r requirements.txt
```

Containers: execução e gestão

Comandos do dia a dia: run, ps, logs, exec, stop, rm

`docker run` (flags essenciais)

O comando mais importante

Terminal (bash)

```bash
# Rodar em foreground
docker run nginx:alpine

# Rodar em background, nomear e publicar porta
docker run -d --name web -p 8080:80 nginx:alpine

# Variáveis de ambiente
docker run --rm -e ENV=dev alpine:3.20 env

# Montar volume (persistência)
docker run -d --name db -v pgdata:/var/lib/postgresql/data postgres:16

# Remover ao sair
docker run --rm alpine:3.20 echo "oi"
```

Checklist rápido

- Preciso expor portas? Use `-p host:container`.
- Preciso persistir dados? Use `-v volume:/caminho` ou bind mount.
- Vai ficar rodando? Use `-d` e observe com `docker logs`.

Ciclo de vida e observabilidade

ps, logs, exec, inspect

Terminal (bash)

```bash
# Listar containers
docker ps
docker ps -a

# Logs (seguindo)
docker logs -f web

# Entrar no container
docker exec -it web sh

# Ver config/estado
docker inspect web

# Parar/remover
docker stop web
```

Dicas de logs

Dicas de exec/inspect

Recursos, restart e healthchecks

Deixando containers mais “operáveis”

CLI: limites e restart

```
# Limitar CPU e memória
docker run -d --name api \n+  --cpus=1.5 --memory=512m \n+  -p
8000:8000 minha-api:1.0

# Política de restart
docker run -d --restart=unless-stopped nginx:alpine
```

Dockerfile: HEALTHCHECK

```
# Dockerfile: HEALTHCHECK
FROM nginx:alpine

HEALTHCHECK --interval=30s --timeout=3s \n+  CMD wget
-qO- http://localhost:80/ || exit 1
```

Copiar arquivos e snapshots

docker cp e docker commit (uso com cautela)

`docker cp`

- Copia arquivos entre host ↔ container.
- Útil para extrair logs, dumps, artefatos de build.

`docker commit`

- Cria uma imagem a partir do estado atual de um container.
- Útil para debug/inspeção, mas não substitui Dockerfile.

Exemplos

```
# Copiar do container para o host
docker cp web:/etc/nginx/nginx.conf ./nginx.conf

# Snapshot (evite em pipelines)
docker commit web web-debug:temp
docker image ls web-debug:temp
```

Persistência: Volumes e Mounts

Como manter dados fora do container e compartilhar arquivos com o host

Volumes, bind mounts e tmpfs

Escolha o tipo certo para cada caso

Volume (gerenciado)

- Armazenamento gerenciado pelo Docker (portável).
- Bom para dados de banco e caches persistentes.

Bind mount (host)

- Monta um caminho do host dentro do container.
- Ideal para desenvolvimento e live reload.

Exemplos

```
# Volume
docker volume create pgdata
docker run -d --name db -v pgdata:/var/lib/postgresql/data postgres:16

# Bind mount (dev)
docker run --rm -it -v "$PWD":/work -w /work node:22-alpine node -v

# tmpfs (memória)
docker run --rm --tmpfs /tmp:rw,size=64m alpine:3.20 sh -lc "df -h /tmp"
```

Gerenciamento de volumes

criar, inspecionar e limpar com segurança

Terminal

```
docker volume ls

docker volume inspect pgdata

# Remover volumes não usados
docker volume prune

# Remover volume específico (garanta que não está em uso)
docker volume rm pgdata
```

Dica importante

- `prune` remove recursos “não referenciados”. Confirme antes em ambientes compartilhados.
- Para backup, você pode anexar o volume a um container temporário e copiar os dados.

Redes (Networking)

Como containers se enxergam e como expor serviços para o host

Redes: conceitos e comandos

bridge, host, none + redes custom

Conceitos rápidos

- A rede `bridge` é o padrão no Docker Engine.
- Containers na mesma rede conseguem se resolver por DNS (nome do container).

Comandos úteis

- `docker network ls` / `inspect`
- `docker network create minha-rede`
- `docker network connect` / `disconnect`

Exemplo

```
# Rede custom + comunicação por DNS
docker network create appnet

docker run -d --name redis --network appnet redis:7-alpine

docker run --rm -it --network appnet alpine:3.20 \
+  sh -lc "apk add --no-cache redis && redis-cli -h redis ping"
```

Docker Compose

Multi-serviço com um único arquivo: build, rede, volumes e variáveis

Por que usar Compose?

Ambientes locais e stacks de serviços repetíveis

O que ele resolve

- Define serviços, redes e volumes em um arquivo
- Sobe/derruba o ambiente inteiro com um comando

Comandos essenciais

- `docker compose up -d` (subir)
- `docker compose logs -f` (logs)
- `docker compose exec` (shell/commands)

Terminal

```
# Subir serviços
docker compose up -d

# Ver status
docker compose ps

# Logs
docker compose logs -f

# Executar comando dentro do service
```

Exemplo de Compose (funcional)

Web + Postgres com volume e variáveis

docker-compose.yml

```yaml
# docker-compose.yml
services:
  web:
    build: .
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@db:5432/app
    depends_on:
      - db

  db:
    image: postgres:16
    environment:
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=app
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata: {}
```

Executar

- No diretório com Dockerfile + app
- `docker compose up -d --build`
- Acesse: http://localhost:8000/

Terminal

```
docker compose up -d --build

docker compose ps

# abrir shell
docker compose exec web sh

# derrubar (mantém volume)
docker compose down

# derrubar e remover volumes
docker compose down -v
```

Compose: recursos úteis

env files, project name, overrides, profiles

Configuração por ambiente

- Use `.env` para variáveis (ex.: senhas dev).
- Use múltiplos arquivos: `-f docker-compose.yml -f docker-compose.dev.yml`.

Organização por projeto

- Compose cria recursos com nome do projeto (rede, containers).
- Você pode definir `--project-name` ou `COMPOSE_PROJECT_NAME`.

Terminal

```
# Exemplo: sobrescrever config
docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d

# Definir nome do projeto
docker compose --project-name loja up -d

# Usar .env (carregado automaticamente no diretório)
# Ex.: DATABASE_URL=...
```

Registries e Distribuição

Como publicar, versionar e consumir imagens com segurança

Tags vs digests

Como referenciar imagens de forma segura

Tags

- Fáceis de ler (ex.: `minha-api:1.2.0`).
- Podem mudar com o tempo (tag pode ser “movida”).

Digests

- Imutáveis: apontam para um conteúdo específico (`@sha256:...`).
- Mais reprodutíveis para deploys críticos.

Exemplo

```
# Pull por tag
docker pull nginx:alpine

# Inspect para ver digest
docker image inspect nginx:alpine --format '{{index .RepoDigests 0}}'

# Pull por digest (exemplo)
docker pull nginx@sha256:SEU_DIGEST_AQUI
```

Troubleshooting e Limpeza

Diagnóstico rápido e como evitar encher o disco

Diagnóstico rápido

O que checar quando “não funciona”

Checklist (containers)

- `docker ps -a` (status / exit code)
- `docker logs <nome>` (erro real)
- `docker inspect <nome>` (ports, envs, mounts)
- `docker exec -it <nome> sh` (debug pontual)

Problemas comuns

- Porta já em uso no host → troque o lado esquerdo do `-p`
- Permissões em bind mount → ajuste owner/UID/GID
- DNS/Proxy → configure Docker Desktop/daemon
- Imagem antiga em cache → use `--no-cache` no build

Comandos úteis

```
# Ver eventos do daemon
docker events --since 10m

# Checar uso de disco
docker system df

# Ver recursos do container
docker stats
```

Limpeza e manutenção

Removendo recursos não usados (com cuidado)

Terminal

```
# Espaço ocupado
docker system df

# Remover containers parados, networks não usadas, imagens sem tag e cache de build
docker system prune

# Remover também volumes não usados (cuidado!)
docker system prune --volumes

# Remover imagens específicas
docker rmi minha-api:1.0
```

Cuidado

- Em máquina compartilhada, `prune` pode apagar recursos de outras pessoas/projetos.
- Antes de limpar volumes, faça backup se houver dados importantes.

Segurança e Boas Práticas

O que melhora confiabilidade, tamanho e segurança das imagens

Segurança: baseline prático

Medidas com bom custo/benefício

No Dockerfile

- Evite rodar como root: defina `USER` quando possível.
- Minimize pacotes e atualize base images com frequência.

No runtime

- Use `--read-only` e mounts explícitos quando aplicável.
- Limite recursos (CPU/memória) e defina healthchecks.

Exemplo

```
# Runtime mais restrito (exemplo)
docker run --rm -p 8000:8000 \
+  --read-only \
+  --tmpfs /tmp:rw,size=64m \
+  minha-api:1.0
```

Boas práticas de Dockerfile

Imagens menores, builds mais rápidos

Checklist

- Use multi-stage para separar build de runtime
- Aproveite cache: deps antes do código
- Fixe versões importantes (quando fizer sentido)
- Remova caches temporários e use `--no-cache-dir` (Python)
- Prefira base images menores (ex.: alpine/slim) com critério

Exemplo multi-stage

```dockerfile
# Multi-stage (exemplo Node)
FROM node:22-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
```

Labs (mão na massa)

Do zero ao Compose: exercícios guiados para fixar

Lab 1 — Containerize uma API

Objetivo: build + run + logs

Tarefas

- Crie uma pasta com `app.py` e `requirements.txt` (exemplo do slide de Dockerfile).
- Escreva um Dockerfile e gere a imagem `minha-api:1.0`.
- Rode com `-p 8000:8000` e teste com `curl`.
- Veja logs e entre no container com `docker exec`.

Roteiro (copiar/colar)

```
docker build -t minha-api:1.0 .

docker run -d --name api -p 8000:8000 minha-api:1.0

docker logs -f api

docker exec -it api sh

curl http://localhost:8000/

docker stop api && docker rm api
```

Lab 2 — Compose (web + banco)

Objetivo: stack multi-serviço com volume

Tarefas

- Crie `docker-compose.yml` com `web` (build) e `db` (postgres).
- Configure `DATABASE_URL` apontando para `db` (hostname = nome do service).
- Suba com `docker compose up -d --build`.
- Verifique rede/DNS: `docker compose exec web ping db` (se tiver ping).
- Derrube com `docker compose down` e depois `down -v` para limpar.

Roteiro (copiar/colar)

```
docker compose up -d --build

docker compose ps

docker compose logs -f

# Executar comando dentro do web
docker compose exec web sh

# Limpar
docker compose down
docker compose down -v
```

Cheat sheet (comandos do dia a dia)

Um resumo para consulta rápida

CLI — básico

```
# IMAGENS
docker pull <img>:<tag>
docker images
docker rmi <img>

# CONTAINERS
docker run ... <img>
docker ps -a
docker logs -f <ctr>
docker exec -it <ctr> sh
docker stop <ctr>
docker rm <ctr>

# VOLUMES
docker volume ls
docker volume inspect <vol>
docker volume prune
```

Rede • Compose • Limpeza

```
# REDES
docker network ls
docker network create <net>
docker network connect <net> <ctr>

# COMPOSE
docker compose up -d --build
docker compose ps
docker compose logs -f
docker compose exec <svc> sh
docker compose down
docker compose down -v

# LIMPEZA
docker system df
docker system prune
docker system prune --volumes
```

Referências oficiais

Documentação para aprofundar

Links (oficiais)

- Docker CLI reference: docs.docker.com/reference/cli/docker/
- Dockerfile reference: docs.docker.com/reference/dockerfile/
- Compose file reference: docs.docker.com/reference/compose-file/
- Storage (volumes/mounts): docs.docker.com/storage/
- Networking: docs.docker.com/network/
- Docker Scout: docs.docker.com/scout/

Próximos passos: pegue um projeto pequeno, crie o Dockerfile, depois compose com DB/cache.