

# Containers e Docker para Sistemas Distribuídos

Parte 1: Fundamentos Teóricos | Parte 2: Demo ao Vivo

Pedro Ximenes

Universidade Católica de Pernambuco (UNICAP)

22 de outubro de 2026

## Sumário I

- 1 Por que containers existem?
  - O problema que containers resolvem
- 2 Evolução: do bare-metal ao container
  - Três gerações de deploy
- 3 Conceitos Fundamentais
  - Imagem Docker
  - Container Docker
  - Volumes
  - Redes Docker
  - Registry
- 4 Dockerfile
  - Escrevendo seu primeiro Dockerfile
- 5 Docker Compose
  - Orquestrando múltiplos containers
- 6 Troubleshooting
  - Resolvendo problemas comuns

## Sumário II

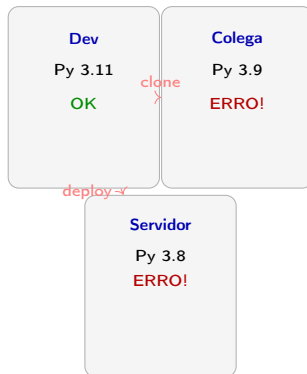
- 7 Demo 1 — Primeiro container
  - Hello-world e exploração
- 8 Demo 2 — Construindo uma imagem
  - Dockerfile e docker build
- 9 Demo 3 — Docker Compose
  - Subindo múltiplos serviços
- 10 Conexão com o Projeto
  - O que fazer esta semana

## O problema clássico de ambiente

### Um cenário real

Imagine que você desenvolveu uma API Python. Ela funciona perfeitamente no seu laptop. Quando o colega clona o repositório, a API quebra com erro de versão de biblioteca.

- Seu laptop: Python 3.11, Flask 3.0, SQLAlchemy 2.0.
- Laptop do colega: Python 3.9, Flask 2.2, SQLAlchemy 1.4.
- Servidor: Ubuntu 20.04, Python 3.8...



### O resultado

Cada ambiente é diferente. Bugs aparecem só em certos ambientes. O deploy vira um evento de alto risco.

## A solução: empacotar o ambiente junto com o código

### A ideia central de containers

Em vez de distribuir apenas o código e torcer para que o ambiente seja compatível, você empacota **código + todas as dependências + configuração de ambiente** em uma unidade chamada **container**.

- O container leva **consigo** o sistema de arquivos, as bibliotecas, variáveis de ambiente e o comando de inicialização.
- Qualquer máquina com o runtime de containers executa esse pacote de forma **idêntica**.
- Não importa se é Windows, macOS, Linux, servidor ou nuvem.

### Analogia prática

Pense em um container como uma **marmita com tudo incluso**: a comida (código), os utensílios (bibliotecas) e o modo de preparo (configuração). O recipiente é padronizado — qualquer microondas (Docker) sabe como aquecer.

## Por que containers importam para Sistemas Distribuídos?

### Sistemas distribuídos são compostos de múltiplos serviços

Em sistemas como Netflix, Uber ou iFood, existem **dezenas a centenas de serviços** rodando simultaneamente — cada um com suas próprias dependências.

- Containers permitem que cada serviço tenha seu **próprio ambiente isolado** sem interferir nos outros.
- O mesmo código que o desenvolvedor testou é **exatamente** o que vai para produção.
- Iniciar 20 serviços em containers leva **segundos**. O mesmo em VMs levaria minutos.

### Três propriedades fundamentais

- **Portabilidade:** a mesma imagem roda em qualquer máquina com Docker.
- **Isolamento:** cada serviço tem seu próprio espaço de processo e rede.
- **Reprodutibilidade:** ambiente de desenvolvimento idêntico ao de entrega.

### Conexão com o projeto

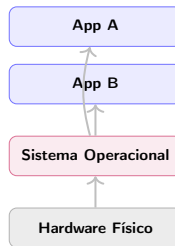
Seu middleware distribuído será avaliado com `docker compose up`. "Roda só no meu computador" não é aceito.

# Geração 1 — Bare-metal: aplicação direto no hardware

## Como funciona

A aplicação roda diretamente no sistema operacional do servidor físico, sem nenhuma camada de virtualização.

- **Vantagem:** máxima performance — sem overhead.
- **Problema 1:** duas aplicações com dependências conflitantes no mesmo servidor são difíceis de manter.
- **Problema 2:** adicionar um novo servidor requer configurar tudo do zero manualmente.
- **Problema 3:** falha em uma aplicação pode afetar todas as outras no mesmo servidor.



Apps compartilham  
o mesmo Python/Java

## Uso hoje

Ainda usado para workloads críticos (HPC, bancos de dados), mas raro para aplicações web modernas.

## Geração 2 — Máquina Virtual: isolamento via hypervisor

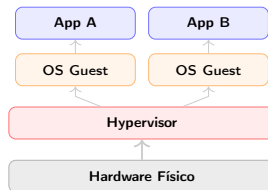
### Como funciona

Um **hypervisor** (VMware, VirtualBox, KVM) simula hardware completo. Cada VM roda seu próprio sistema operacional guest sobre o hardware virtual.

- **Vantagem 1:** isolamento forte — cada VM é independente.
- **Vantagem 2:** fácil provisionar novos ambientes copiando imagens de VM.
- **Problema 1:** cada VM carrega um OS completo. Uma VM simples consome **1–4 GB de RAM** só para o sistema operacional.
- **Problema 2:** startup lento (1–5 minutos).
- **Problema 3:** overhead de CPU para virtualizar o hardware.

### Uso hoje

VMs ainda dominam infraestrutura de nuvem (EC2, Azure VM). Containers são preferidos para microserviços por serem mais leves e rápidos.



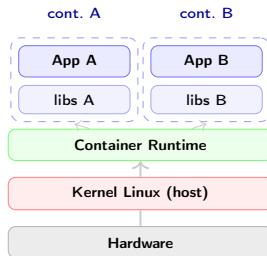
Cada VM: OS completo  
(1–4 GB RAM extra)

## Geração 3 — Container: isolamento leve via kernel do host

### Como funciona

Containers **não virtualizam hardware**. Eles usam recursos do próprio kernel Linux (**namespaces** e **cgroups**) para criar ambientes isolados dentro do mesmo OS.

- **Vantagem 1:** leveza — o container carrega apenas a aplicação e suas bibliotecas. Sem OS duplicado.
- **Vantagem 2:** startup em milissegundos a poucos segundos.
- **Vantagem 3:** alta densidade — dezenas de containers por máquina.
- **Limite:** todos os containers compartilham o **mesmo kernel**. Isolamento menor que VMs.



### Comparação de startup

VM: 2–5 minutos | Container: 1–5 segundos

## Comparativo: bare-metal vs. VM vs. container

| Aspecto             | Bare-metal | VM             | Container     |
|---------------------|------------|----------------|---------------|
| Isolamento          | Nenhum     | Forte          | Suficiente    |
| Overhead de memória | Mínimo     | Alto (OS dup.) | Mínimo        |
| Tempo de startup    | Minutos    | 1–5 min        | Segundos      |
| Portabilidade       | Baixa      | Média          | Alta          |
| Densidade no host   | Baixa      | Média          | Alta          |
| Uso típico hoje     | HPC/BD     | IaaS/Nuvem     | Microserviços |

## Uso combinado na prática

VMs e containers são usados juntos: uma VM (ex.: EC2 na AWS) roda Docker e hospeda dezenas de containers.

# Namespaces e cgroups: a base técnica dos containers

## Namespaces — “mundos isolados”

O kernel Linux cria **visões isoladas** de recursos do sistema:

- **pid**: container vê apenas seus processos. Primeiro processo = PID 1 *dentro* do container.
- **net**: interface de rede própria (eth0 virtual).
- **mnt**: sistema de arquivos próprio e isolado.
- **uts**: hostname próprio.

## cgroups — “orçamento de recursos”

**Control Groups** limitam o consumo de recursos:

- CPU: no máx. 50% de um núcleo.
- Memória: no máx. 256 MB.
- I/O: limite de leitura/escrita.

Sem cgroups, um container consome 100% dos recursos e trava os demais.

## Atenção

Containers **não são VMs**. Se o kernel do host falha, todos os containers caem juntos.

# O que é uma Imagem Docker?

## Definição simples

Uma **imagem Docker** é um **arquivo** que contém tudo necessário para executar uma aplicação:

- Código da aplicação (Python, Java, Go, etc.)
- Todas as bibliotecas e ferramentas necessárias.
- Sistema de arquivos mínimo (uma base de sistema operacional ou imagem 'scratch').
- Configuração e comando de inicialização.

- Imagens são **imutáveis** (somente leitura).
- São **construídas** a partir de um Dockerfile.
- São **distribuídas** via registries (Docker Hub).
- Um **container** é uma imagem *em execução*.

## Analogia com POO

**Imagem**  $\approx$  Classe

**Container**  $\approx$  Objeto (instância)

Você pode criar 10 containers a partir da mesma imagem, assim como 10 objetos da mesma classe.

# Imagens são construídas em camadas

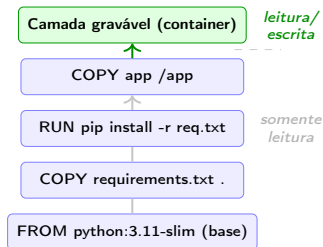
## Por que camadas?

Cada instrução do Dockerfile gera uma camada:

- **Imutáveis:** uma vez criadas, não mudam.
- **Compartilhadas:** se duas imagens usam `python:3.11-slim`, a camada base é armazenada **apenas uma vez** no disco.
- **Incrementais:** um rebuild só reconstrói as camadas que mudaram.

## Camada gravável do container

Ao iniciar um container, o Docker adiciona uma fina camada **gravável** no topo. Ao **remover o container**, essa camada é descartada. Use volumes para dados persistentes.



## Tags: como identificar e versionar imagens

### Anatomia de uma referência de imagem

`[registry/] [usuario/]nome:tag`

- `python:3.11-slim` — imagem oficial Python, Debian slim.
- `postgres:15.3` — PostgreSQL versão 15.3.
- `ghcr.io/meugrupo/api:v1.2.0` — registry privado no GitHub.

### Nunca use `:latest` no projeto

A tag `:latest` aponta para a versão mais recente no momento do pull. Se a mantenedora publicar uma nova versão com breaking changes, seu projeto quebra na próxima vez que alguém executar `docker compose up`.

- Use `postgres:15.3` em vez de `postgres:latest`.
- Use `python:3.11-slim` em vez de `python:latest`.

# O que é um Container Docker?

## Definição

Um **container** é um **processo em execução**, criado a partir de uma imagem, isolado dos demais processos do sistema.

- Tem seu **próprio sistema de arquivos** (as camadas da imagem).
- Tem sua **própria interface de rede** (endereço IP virtual).
- Enxerga **apenas seus próprios processos**.
- Inicia em **segundos** — não precisa inicializar um OS.

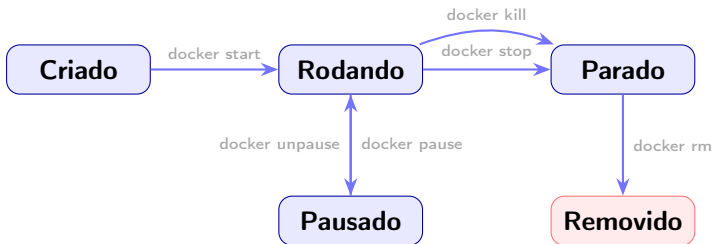
## Quando usar

- Cada serviço do middleware em um container.
- Banco de dados em um container.
- Broker de mensagens em um container.

## Dado importante

Dados escritos **dentro** do container sem volume são **perdidos** ao removê-lo. Não salve dados na camada gravável.

## Ciclo de vida de um container



- `docker run` = cria e inicia em uma única etapa.
- `docker stop` envia SIGTERM (encerramento gracioso), depois SIGKILL.
- Container **parado** ainda ocupa espaço em disco. Use `docker rm` para removê-lo ou `docker run -rm` para remoção automática.

# Primeiros comandos Docker

```
# Rodar o "Hello World" do Docker
docker run hello-world

# Rodar Ubuntu interativamente (sair: exit)
docker run -it ubuntu bash

# Servidor nginx em background (-d = detached)
docker run -d --name meu-nginx -p 8080:80 nginx

# Listar containers em execucao
docker ps

# Listar todos (inclusive parados)
docker ps -a

# Seguir logs em tempo real
docker logs -f meu-nginx

# Abrir shell dentro do container
docker exec -it meu-nginx bash

# Parar e remover
docker stop meu-nginx
docker rm meu-nginx

# Ver imagens baixadas localmente
docker images
```

# Volumes Docker: persistência de dados

## Por que preciso de volumes?

Imagine um container de PostgreSQL. Você insere dados, para o container e sobe de novo. **Sem volume, todos os dados são perdidos.**

## Named Volume (produção)

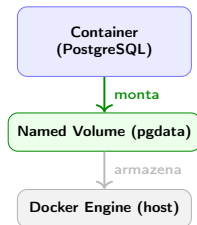
- Gerenciado pelo próprio Docker.
- Sobrevive ao container.
- Use para: banco de dados, estado de broker.

```
-v pgdata:/var/lib/postgresql/data
```

## Bind Mount (desenvolvimento)

- Pasta do host mapeada no container.
- Mudanças refletidas em tempo real.
- **Não use em produção.**

```
-v $(pwd)/src:/app/src
```



Dados persistem  
após container removido

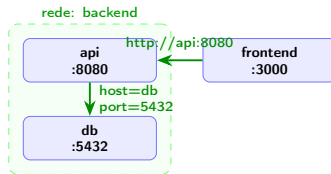
# Redes Docker: como containers se comunicam

## O problema sem redes customizadas

Por padrão, containers não se enxergam pelo nome. Você teria que descobrir o IP de cada container, que muda a cada reinicialização.

## Tipos de rede

- **bridge customizada** (recomendado): rede privada virtual. Containers se resolvem pelo **nome do serviço**.
- **host**: container usa a rede da máquina diretamente. Sem isolamento de rede.
- **none**: sem interface de rede. Isolamento total.
- **overlay**: comunicação entre containers em hosts diferentes (Swarm/Kubernetes).



## Armadilha!

localhost dentro de um container aponta para o **próprio container**, não para outros.

## DNS interno do Docker

Na rede bridge customizada:

`http://api:8000/health`

`postgresql://db:5432/mydb`

# Registry: de onde as imagens vêm?

## O que é um registry?

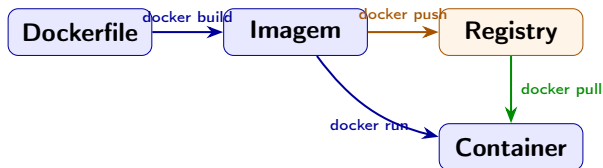
Um **registry** é um servidor que armazena e distribui imagens Docker — funcionando como um “GitHub para imagens”.

## Registries gratuitos

- **Docker Hub** ([docker.io](https://docker.io)): padrão. postgres, redis, nginx vem daqui.
- **GHCR**: GitHub Container Registry. Integrado com repositórios do GitHub.

## Fluxo básico

- 1 `docker pull postgres:15.3`
- 2 `docker build -t api:1.0 .`
- 3 `docker push usuario/api:1.0`



# Estrutura completa de um Dockerfile

```
# 1. Imagem base (OS minimo + runtime)
FROM python:3.11-slim

# 2. Metadado (opcional)
LABEL maintainer="time@projeto.io"

# 3. Diretorio de trabalho no container
WORKDIR /app

# 4. Copia APENAS as dependencias primeiro
# (cache inteligente: ver proximo slide)
COPY requirements.txt .

# 5. Instala as dependencias
RUN pip install --no-cache-dir \
    -r requirements.txt

# 6. Copia o restante do codigo
COPY . .

# 7. Documenta a porta (nao expoe ao host!)
EXPOSE 8000

# 8. Executa como usuario sem privilegios
USER nobody

# 9. Comando que inicia a aplicacao
CMD ["python", "app.py"]
```

## Instruções principais

- FROM: imagem base.
- WORKDIR: cria e entra no dir.
- COPY: copia arquivos.
- RUN: executa no build.
- EXPOSE: documenta porta.
- USER: define usuário.
- CMD: comando padrão.

## A ordem importa!

Dependências **antes** do código-fonte. Aproveita o cache do Docker. Veja o próximo slide.

# Build cache: antes e depois

## Ruim: pip install toda vez

```
FROM python:3.11-slim
WORKDIR /app

# Copia tudo de uma vez
COPY . .

# Se app.py mudar ->
# reinstala TODAS as libs!
RUN pip install \
    -r requirements.txt

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

cache invalidado a cada mudança de código

## Bom: cache inteligente

```
FROM python:3.11-slim
WORKDIR /app

# 1. So as dependencias
COPY requirements.txt .

# 2. Instala (fica no cache!)
RUN pip install --no-cache-dir \
    -r requirements.txt

# 3. Código por ultimo
COPY . .

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

cache reutilizado, build rapido

## Variáveis de ambiente: ENV e ARG

### ENV — variáveis em runtime

```
ENV PORT=8000
ENV LOG_LEVEL=info
ENV APP_ENV=production
```

Ficam disponíveis **dentro do container** em execução. Lidas com `os.environ["PORT"]` (Python) ou `System.getenv("PORT")` (Java).

### ARG — variáveis só no build

```
ARG APP_VERSION=1.0.0
RUN echo "Compilando versao $APP_VERSION"
```

Não ficam na imagem final. Use para versões, tokens de CI.

### Segurança importante

Variáveis ENV com senhas ficam **visíveis na imagem** (`docker inspect`). Não coloque `DATABASE_PASSWORD` no Dockerfile. Injete em runtime via `docker run -e` ou no Compose.

# Boas práticas essenciais de Dockerfile

## Use imagens base menores

```
# ~900 MB - evite em producao
FROM python:3.11

# ~130 MB - RECOMENDADA
FROM python:3.11-slim

# ~50 MB - cuidado com libs C
FROM python:3.11-alpine
```

## Não rode como root

```
RUN addgroup --system app \
    && adduser --system \
        --ingroup app appuser
USER appuser
```

## Consolide comandos RUN

```
# Ruim: 3 camadas
RUN apt-get update
RUN apt-get install -y curl
RUN rm -rf /var/lib/apt/lists/*

# Bom: 1 camada, apt limpo
RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
```

## .dockerignore sempre

```
.git
__pycache__
*.pyc
.env
tests/
*.log
node_modules/
```

Exclui do contexto de build: evita vazar segredos e acelera o build.

# Por que o Docker Compose existe?

## O problema com `docker run` manual

Uma aplicação de microserviços simples pode ter: API + banco + broker de mensagens. Iniciar cada container com `docker run` requer lembrar: portas, variáveis de ambiente, volumes, redes, ordem de dependências...

## A solução

O **Docker Compose** descreve toda a infraestrutura em um único arquivo YAML. Com um comando, tudo sobe:

```
docker compose up
```

## O que o Compose gerencia

- Quais containers subir e em qual ordem.
- Variáveis de ambiente de cada serviço.
- Mapeamento de portas entre host e containers.
- Volumes e redes compartilhadas.
- Dependências entre serviços.

# Estrutura do docker-compose.yml — parte 1

```
version: "3.9"

services:

  api:
    build: ./api          # constroi a partir do Dockerfile em ./api
    ports:
      - "8000:8000"       # host:container
    environment:
      - DATABASE_URL=postgresql://user:pass@db:5432/mydb
      - LOG_LEVEL=info
    depends_on:
      db:
        condition: service_healthy # espera db estar saudavel
    networks:
      - app-net

  db:
    image: postgres:15.3
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
      POSTGRES_DB: mydb
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user"]
      interval: 5s
      retries: 5
    networks:
      - app-net
```

## Estrutura do docker-compose.yml — parte 2

```
broker:
  image: rabbitmq:3-management
  ports:
    - "5672:5672"      # porta AMQP (mensagens)
    - "15672:15672"    # painel web de gerenciamento
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "ping"]
    interval: 10s
    retries: 5
    start_period: 30s  # RabbitMQ demora para iniciar
  networks:
    - app-net

volumes:
  pgdata:      # named volume gerenciado pelo Docker

networks:
  app-net:     # rede bridge customizada
```

### Comandos essenciais

```
docker compose up -d   sobe em background
docker compose down    para e remove containers
docker compose logs -f api  segue logs do servico api
docker compose ps      status de todos os servicos
```

## depends\_on: a armadilha da inicialização

### O problema

`depends_on` garante a **ordem de criação** dos containers, não que o serviço esteja **pronto para aceitar conexões**. O container do PostgreSQL pode estar rodando, mas o banco ainda inicializando internamente. Sua API vai falhar.

### Solução: `condition: service_healthy`

- O Compose espera até o **healthcheck** do serviço passar antes de iniciar o dependente.
- Para isso, o serviço dependido precisa ter um healthcheck definido.

### Três condições disponíveis

- `service_started`: só aguarda o container iniciar (comportamento fraco, legado).
- `service_healthy`: aguarda o healthcheck passar. **Use sempre.**
- `service_completed_successfully`: aguarda o container encerrar com sucesso (útil para scripts de migração de banco antes da API subir).

# Variáveis de ambiente no Compose

```
# Forma 1: inline
services:
  api:
    environment:
      - DATABASE_URL=postgres://...
      - LOG_LEVEL=info
```

```
# Forma 2: arquivo .env
services:
  api:
    env_file:
      - .env
```

```
# Forma 3: repassa do shell
services:
  api:
    environment:
      - SECRET_KEY=${SECRET_KEY}
```

## Nunca versiona senhas

Adicione `.env` ao `.gitignore`. Crie `.env.example` sem valores reais para documentar quais variáveis são necessárias.

## Problema 1: conflito de porta

### Erro típico

```
Error: Ports are not available:  
listen tcp 0.0.0.0:5432: bind: address already in use
```

### Causa e diagnóstico (Windows PowerShell)

Outra aplicação já está usando essa porta.

```
netstat -ano | findstr :5432  
# Mostra qual processo esta usando a porta
```

### Solução: mude a porta do HOST

```
ports:  
- "15432:5432" # host:container  
# Acesse pelo host na porta 15432  
# Dentro da rede Docker, continua sendo db:5432
```

## Problema 2: localhost dentro de container

### Erro típico

Connection refused: localhost:5432  
(mesmo o banco estando rodando em outro container)

### Causa

Dentro de um container, localhost = o **próprio container**. Outros containers não são acessíveis via localhost.

### Errado

```
DATABASE_URL=postgresql://  
user:pass@localhost:5432/db
```

### Correto

```
DATABASE_URL=postgresql://  
user:pass@db:5432/db  
# "db" = nome do serviço  
# no docker-compose.yml
```

# Checklist de diagnóstico: quando algo não sobe

## Siga esta ordem antes de pedir ajuda

- 1 O container está rodando?  
`docker compose ps`
- 2 Qual é o erro?  
`docker compose logs [nome-do-servico]`
- 3 O healthcheck passou? (coluna STATUS deve ser "healthy")
- 4 As variáveis chegaram?  
`docker exec [container] env`
- 5 Um container consegue alcançar outro?  
`docker exec [container] ping db`
- 6 Último recurso — limpar e rebuildar:  
`docker compose down -v`  
`docker compose build --no-cache`  
`docker compose up`

## Aviso

`docker compose down -v` remove volumes — inclui dados do banco. Nunca execute em produção sem backup.

# Parte 2

## Demo ao Vivo

O professor irá executar os comandos no terminal.

Acompanhe, faça perguntas em qualquer momento.

# [DEMO 1] — Verificando a instalação e rodando hello-world

## O que faremos

- 1 Verificar que o Docker está instalado e funcionando.
- 2 Rodar o container de exemplo `hello-world`.
- 3 Entender o que aconteceu nos bastidores.

## Comandos a executar

- 1 `docker -version`
- 2 `docker run hello-world`
- 3 `docker ps -a` — o container já parou
- 4 `docker images` — a imagem está no cache local

## Perguntas para reflexão

De onde veio a imagem `hello-world`? Por que o container parou imediatamente? Por que a segunda execução seria mais rápida (sem baixar novamente)?

## [DEMO 2] — Servidor web em container

### O que faremos

Subir um servidor Nginx em container, acessar no navegador sem instalar nada, e ver os logs em tempo real.

### Comandos a executar

- 1 `docker run -d -name web -p 8080:80 nginx`
- 2 Abrir `http://localhost:8080` no navegador
- 3 `docker logs -f web` (recarregar a página para gerar logs)
- 4 `docker inspect web` — JSON completo do container
- 5 `docker stop web` e `docker rm web`
- 6 Recarregar o navegador — conexão recusada

### Observe

O Nginx rodou **sem instalar nada** no sistema operacional do host. Todos os arquivos do servidor estão dentro do container. Ao remover o container, o servidor desaparece completamente.

## [DEMO 3] — Construindo nossa primeira imagem

### O que faremos

Construir a imagem de uma API Python minimalista que retorna "Hello from container!" (pasta: demo/01-api-simples/).

### Arquivos do demo

- `app.py` — API Flask com um endpoint simples.
- `requirements.txt` — dependência flask.
- `Dockerfile` — instruções de build.

### Comandos a executar

- 1 `cd demo/01-api-simples`
- 2 `docker build -t minha-api:1.0 .`  
(observar as camadas sendo criadas)
- 3 `docker run -p 8000:8000 minha-api:1.0`
- 4 Abrir `http://localhost:8000` no navegador
- 5 Modificar `app.py`, rebuildar e observar o **cache em ação**

## [DEMO 4] — Explorando o interior do container

### O que faremos

Entrar dentro de um container em execução com `docker exec` e explorar seu sistema de arquivos e processos.

### Comandos a executar

- ❶ `docker run -d -name api minha-api:1.0`
- ❷ `docker exec -it api sh` (*shell dentro do container*)
- ❸ Dentro do container:
  - `ls /app` — ver os arquivos da aplicação
  - `ps aux` — ver apenas os processos do container
  - `hostname` — é o ID do container
  - `exit`
- ❹ `docker stats` — uso de CPU/RAM em tempo real
- ❺ `docker stop api` e `docker rm api`

### Observe

O `ps aux` dentro do container mostra **apenas os processos do container**, não os do sistema operacional completo.

## [DEMO 5] — API + banco com Docker Compose

### O que faremos

Subir uma aplicação com 2 serviços: **api** (Python) + **db** (PostgreSQL) com volume persistente. (pasta: demo/02-api-postgres/)

### Arquivos do demo

- api/app.py, api/Dockerfile
- docker-compose.yml — orquestra os 2 serviços
- .env.example — documenta as variáveis necessárias

### Comandos a executar

- 1 `cd demo/02-api-postgres`
- 2 `cp .env.example .env`
- 3 `docker compose up`
- 4 Acessar `http://localhost:8000/items`
- 5 `docker compose ps` (em outro terminal)
- 6 `docker compose logs -f api`

## [DEMO 6] — Persistência de dados com volumes

### O que demonstraremos

Os dados gravados no banco **sobrevivem** ao remover e recriar os containers.

### Sequência a demonstrar

- 1 Inserir dados via POST `/items`
- 2 `docker compose down` — para e remove os containers
- 3 `docker compose up` — sobe tudo novamente
- 4 GET `/items` — dados ainda existem!
- 5 `docker compose down -v` — remove volume também
- 6 `docker compose up` — dados sumiram

### Lição essencial

`docker compose down` **não remove volumes** por padrão.

`docker compose down -v` remove tudo, inclusive os dados.

**Nunca use `-v` em produção sem backup.**

## [DEMO 7] — A armadilha do localhost ao vivo

### O que demonstraremos

Mostrar o erro exato que acontece ao usar `localhost` para referenciar outro container.

### Sequência a demonstrar

- 1 Editar `app.py`: trocar `db:5432` por `localhost:5432`
- 2 `docker compose up -build`
- 3 A API falha: `Connection refused: localhost:5432`
- 4 Reverter para `db:5432` e subir de novo — funciona

### Mensagem principal

Dentro de containers, use **sempre** o nome do serviço como hostname. `localhost` aponta para o próprio container, não para outros serviços.

## O que os grupos devem preparar para a Aula Prática (26/10)

### Próxima aula — Prática: Containers e Docker (26/10)

- ☐ Grupo formado (até 5 integrantes) e informado ao professor.
- ☐ Domínio da aplicação escolhido.
- ☐ Repositório criado no GitHub/GitLab com o professor como colaborador.
- ☐ **Pelo menos 2 serviços subindo** com docker compose up.
- ☐ README.md com nome do grupo, domínio e instruções de execução.
- ☐ Esboço do diagrama de arquitetura (pode ser no papel ou draw.io).

### Ponto de partida sugerido

Use o demo/02-api-postgres como base. Adapte para o domínio do grupo, adicione um segundo serviço com responsabilidade diferente e defina o esquema do banco.

### Atenção ao cronograma

O projeto foi apresentado hoje. As apresentações finais serão em **19/11 e 23/11**. Não deixem para a última hora!