Sumário

[**Resumo e Principais Tópicos** 4](#_Toc201011170)

[**Contexto Inicial** 4](#_Toc201011171)

[**Problemas Encontrados** 4](#_Toc201011172)

[**Soluções Tradicionais e Seus Problemas** 4](#_Toc201011173)

[**Containers como Solução** 4](#_Toc201011174)

[**Dicas** 5](#_Toc201011175)

[**Resumo e Principais Tópicos** Responder às perguntas: 6](#_Toc201011176)

[**1. Por que Containers São Mais Leves que VMs?** 6](#_Toc201011177)

[**2. Como os Containers Garantem Isolamento?** 6](#_Toc201011178)

[**3. Como Funcionam sem Instalar um SO?** 6](#_Toc201011179)

[**4. Divisão de Recursos (CPU, Memória, etc.)** 7](#_Toc201011180)

[**Exemplo**: 7](#_Toc201011181)

[**Conclusão** 7](#_Toc201011182)

[Documentações: **Ubuntu | Docker Docs Aprenda mais sobre Máquinas Virtuais com o Artigo VirtualBox e Máquinas Virtuais** 8](#_Toc201011183)

[**Resumo: Docker e Virtualização com Containers** 9](#_Toc201011184)

[**O que é o Docker?** 9](#_Toc201011185)

[**Principais Benefícios do Docker** 9](#_Toc201011186)

[**Como o Docker Funciona?** 10](#_Toc201011187)

[**Casos de Uso do Docker** 10](#_Toc201011188)

[**Dicas Práticas** 10](#_Toc201011189)

[**Conclusão** 11](#_Toc201011190)

[**Fluxo de Execução do Comando**docker run 12](#_Toc201011191)

[**Resumo dos Comandos Docker Essenciais** 15](#_Toc201011192)

[**1. Gerenciamento de Containers** 15](#_Toc201011193)

[**2. Interação com Containers** 15](#_Toc201011194)

[**3. Ciclo de Vida de um Container** 16](#_Toc201011195)

[**4. Comparação:**stop**vs**pause 16](#_Toc201011196)

[**5. Fluxo Prático** 17](#_Toc201011197)

[**Para inspecionar containers**: 17](#_Toc201011198)

[**Resumo: Executando um Container Docker com Aplicação Web** 18](#_Toc201011199)

[**Principais Tópicos Abordados** 18](#_Toc201011200)

[**Dicas Importantes** 19](#_Toc201011201)

[**Conclusão**: A aula demonstrou como executar, gerenciar e expor uma aplicação web em um container Docker, destacando flags essenciais (-d, -P, -p) e boas práticas. 20](#_Toc201011202)

[**Resumo: O que são Imagens Docker e Como Funcionam?** 20](#_Toc201011203)

[**Principais Tópicos Abordados** 20](#_Toc201011204)

[**Dicas Importantes** 21](#_Toc201011205)

[**Conclusão**: Entender imagens e containers é essencial para dominar o Docker. Com esse conhecimento, você está pronto para criar e gerenciar seus próprios ambientes de forma eficiente! 🚀 21](#_Toc201011206)

[**Resumo: Criando sua Primeira Imagem Docker com Node.js** 22](#_Toc201011207)

[**Principais Tópicos Abordados** 22](#_Toc201011208)

[**Dicas Importantes** 23](#_Toc201011209)

[**Próximos Passos** 24](#_Toc201011210)

[**Conclusão**: Você criou sua primeira imagem Docker! Com esse fluxo, você pode empacotar qualquer aplicação e compartilhá-la de forma consistente. 🎉 24](#_Toc201011211)

[**Resumo dos Principais Tópicos** 25](#_Toc201011212)

[**Dicas** 26](#_Toc201011213)

[**Comandos Docker e Persistência de Dados** 27](#_Toc201011214)

[**Principais Tópicos** 27](#_Toc201011215)

[**Dicas Importantes** 28](#_Toc201011216)

[**Principais Comandos** 28](#_Toc201011217)

[**Bind Mounts no Docker** 29](#_Toc201011218)

[**Principais Tópicos** 29](#_Toc201011219)

[**Dicas Importantes** 30](#_Toc201011220)

[**Resumo: Volumes no Docker** 32](#_Toc201011221)

[**Principais Tópicos** 32](#_Toc201011222)

[**Dicas Importantes** 33](#_Toc201011223)

[**tmpfs no Docker - Armazenamento Temporário em Memória** 35](#_Toc201011224)

[**O que é tmpfs?** 35](#_Toc201011225)

[**Como Usar tmpfs?** 35](#_Toc201011226)

[**Exemplo Prático** 36](#_Toc201011227)

[**Comparação: Bind Mount vs Volume vs tmpfs** 37](#_Toc201011228)

[**Comandos Principais** 37](#_Toc201011229)

[**Dicas Importantes** 37](#_Toc201011230)

[**Comunicação entre Contêineres no Docker** 39](#_Toc201011231)

[**O Problema da Comunicação** 39](#_Toc201011232)

[**Solução: Redes Docker** 39](#_Toc201011233)

[**Como Melhorar a Comunicação?** 39](#_Toc201011234)

[**Passo a Passo: Testando Comunicação** 40](#_Toc201011235)

[**Próximos Passos** 41](#_Toc201011236)

[**Dicas Importantes** 41](#_Toc201011237)

# **Resumo e Principais Tópicos**

### **Contexto Inicial**

* Sistemas modernos são compostos por múltiplas aplicações e ferramentas que interagem entre si.
* Exemplo: Um sistema com **Nginx** (load balancer), uma aplicação **Java** e uma aplicação **C# (.NET)**.

### **Problemas Encontrados**

1. **Conflito de Portas**: Todas as aplicações precisam da **porta 80** simultaneamente.
2. **Gerenciamento de Versões**:
   * Difícil atualização/downgrade (ex: C# .NET 9, Java 17, Nginx 1.17.0).
   * Risco de quebrar dependências ao alterar versões.
3. **Alocação de Recursos**:
   * Como definir limites de CPU e memória para cada aplicação?
   * Ex: C# precisa de 100 millicores de CPU e 200MB de RAM.
4. **Manutenção a Longo Prazo**:
   * Complexidade em gerenciar portas, recursos e versões em um único ambiente.

### **Soluções Tradicionais e Seus Problemas**

1. **Máquinas Físicas Dedicadas**
   * **Vantagem**: Isolamento completo (sem conflitos de portas ou recursos).
   * **Desvantagem**: Custo proibitivo (uma máquina por aplicação é inviável em escala).
2. **Máquinas Virtuais (VMs)**
   * Usam **hypervisor** para virtualizar sistemas operacionais.
   * **Vantagem**: Isolamento e controle de recursos.
   * **Desvantagem**: **Overhead** (cada VM roda um SO completo, consumindo mais recursos).

### **Containers como Solução**

* **Diferença das VMs**:
  + Sem hypervisor ou SO virtualizado.
  + Isolamento direto no nível do kernel do SO hospedeiro.
* **Vantagens**:
  + **Leveza**: Menos consumo de recursos (não precisa de um SO completo por app).
  + **Isolamento**: Aplicações rodam independentemente (sem conflitos).
  + **Portabilidade**: Fácil gerenciamento de versões e dependências.
* **Perguntas-chave**:
  + Como o container isola as aplicações sem um SO dedicado?
  + Como os recursos (CPU, memória) são alocados?

**Conclusão e Próximos Passos**

* Containers oferecem uma solução **eficiente e escalável** para os problemas citados.
* No próximo vídeo:
  + Como containers garantem isolamento.
  + Como funcionam sem um SO completo.
  + Divisão de recursos entre containers.

### **Dicas**

1. **Para evitar conflitos de portas**: Use containers para isolar aplicações (cada uma pode "enxergar" a porta 80 internamente, mas mapear para portas diferentes no host).
2. **Gerenciamento de versões**: Utilize imagens de containers com versões específicas (ex: nginx:1.17.0, openjdk:17).
3. **Controle de recursos**: Defina limites de CPU/memória nos containers (ex: docker run --memory=200m --cpus=0.1).
4. **Escalabilidade**: Containers permitem deploy rápido e replicação (útil para load balancing).
5. **Ferramentas recomendadas**:
   * **Docker** (para desenvolvimento e testes).
   * **Kubernetes** (para orquestração em produção).

**Próximo passo**: Entender a arquitetura interna dos containers e como eles operam de forma leve e isolada. 🚀

# **Resumo e Principais Tópicos** Responder às perguntas:

1. Por que **containers são mais leves** que máquinas virtuais (VMs)?
2. Como garantem **isolamento**?
3. Como funcionam **sem instalar um sistema operacional**?
4. Como é feita a **divisão de recursos** do sistema?

## **1. Por que Containers São Mais Leves que VMs?**

* **Máquinas Virtuais (VMs)**:
  + Exigem um **SO completo** para cada aplicação.
  + Usam **hypervisor**, criando overhead (consumo extra de CPU, memória e armazenamento).
* **Containers**:
  + Rodam como **processos isolados** diretamente no SO hospedeiro.
  + **Não virtualizam um SO** → menos consumo de recursos.
  + São mais **rápidos** para iniciar/parar e ocupam **menos espaço**.

## **2. Como os Containers Garantem Isolamento?**

Utilizam **namespaces** do Linux para isolar diferentes aspectos do sistema:

* **PID namespace**: Isola processos (um container não enxerga os processos de outro).
* **NET namespace**: Isola interfaces de rede (cada container tem sua própria rede).
* **IPC namespace**: Isola comunicação entre processos (evita conflitos).
* **MNT namespace**: Isola o sistema de arquivos (cada container tem seu próprio filesystem).
* **UTS namespace**: Isola identificadores do sistema (hostname, kernel compartilhado).

👉 **Resultado**: Cada container age como um ambiente independente, mesmo rodando no mesmo host.

## **3. Como Funcionam sem Instalar um SO?**

* Containers **não precisam de um SO completo** porque:
  + Usam o **kernel do host** (via **UTS namespace**).
  + Apenas **bibliotecas e dependências** da aplicação são empacotadas.
* **Exemplo**:
  + Se o host usa **Linux**, o container compartilha o mesmo kernel, mas com isolamento.
  + Não é necessário instalar um **Windows/Linux** dentro do container.

## **4. Divisão de Recursos (CPU, Memória, etc.)**

Usam **cgroups (Control Groups)** para gerenciar recursos:

* **Limite de CPU**: Definir quantos núcleos ou porcentagem de CPU um container pode usar.
* **Limite de Memória**: Evitar que um container consuma toda a RAM.
* **Prioridade de E/S (I/O)**: Controlar acesso a disco/rede.

### **Exemplo**:

sh

docker run --cpus="0.5" --memory="512m" nginx # Limita a 0.5 CPU e 512MB RAM

### **Conclusão**

* **Containers vs. VMs**:

| **Aspecto** | **Containers** | **Máquinas Virtuais (VMs)** |
| --- | --- | --- |
| **Isolamento** | Namespaces + cgroups | Hypervisor + SO completo |
| **Consumo** | Leve (processos do host) | Pesado (SO virtualizado) |
| **Inicialização** | Segundos | Minutos |
| **Portabilidade** | Alta (imagens autocontidas) | Média (depende do SO da VM) |

* **Próximos passos**:
  + Instalação do **Docker** (Windows/Linux).
  + Uso prático de containers.

**Dicas**

1. **Para otimizar recursos**: Use --cpus e --memory no Docker para evitar consumo excessivo.
2. **Isolamento seguro**: Namespaces garantem que um container não afete outros ou o host.
3. **Sem SO redundante**: Containers são eficientes porque **compartilham o kernel do host**.
4. **Ferramentas complementares**:
   * **Docker** (para criação e execução de containers).
   * **Kubernetes** (para orquestração em grande escala).

**Próximo tema**: Instalação e primeiros passos com Docker! 🐳

# Documentações: [**Ubuntu | Docker Docs**](https://docs.docker.com/engine/install/ubuntu/)[**Aprenda mais sobre Máquinas Virtuais com o Artigo VirtualBox e Máquinas Virtuais**](https://www.alura.com.br/artigos/virtualbox-maquinas-virtuais)

[**Saiba como instalar e utilizar uma Máquina Virtual com o vídeo Máquina Virtual: O que é e como instalar**](https://cursos.alura.com.br/extra/alura-mais/maquina-virtual-o-que-e-e-como-instalar-c1376)

# **Resumo: Docker e Virtualização com Containers**

## **O que é o Docker?**

* Plataforma que implementa **virtualização em nível de sistema operacional** usando **containers**.
* Permite empacotar aplicações com **código, runtime, bibliotecas e dependências** em unidades isoladas e portáteis.

## **Principais Benefícios do Docker**

**1. Isolamento de Contextos**

* Cada container tem:
  + **Sistema de arquivos próprio** (isolado do host e de outros containers).
  + **Processos independentes** (via namespaces do Linux).
  + **Rede dedicada** (evita conflitos de portas).
* **Vantagem**:
  + Aplicações não interferem umas nas outras ou no sistema hospedeiro.
  + Maior segurança e estabilidade.

**2. Versionamento de Aplicações**

* **Imagens Docker**:
  + São **imutáveis** e autocontidas (tudo necessário para rodar a aplicação).
  + Construídas em **camadas** (otimiza reutilização e armazenamento).
* **Dockerfile**:
  + Arquivo de texto que define os passos para criar uma imagem.
  + Exemplo:

dockerfile

FROM python:3.8 # Imagem base

. /app # Copia código para o container

RUN pip install -r /app/requirements.txt # Instala dependências

CMD ["python", "/app/main.py"] # Comando de execução

* **Vantagens do versionamento**:
  + Consistência entre ambientes (desenvolvimento, teste, produção).
  + Facilidade para rollback (voltar para versões anteriores).

## **Como o Docker Funciona?**

1. **Imagens**: Modelo estático (como um "molde") do container.
2. **Containers**: Instâncias em execução de uma imagem.
3. **Docker Engine**:
   * Gerencia containers (inicia, para, monitora recursos).
   * Usa **namespaces** e **cgroups** para isolamento e controle de recursos.

**Comparativo: Docker vs. Virtualização Tradicional (VMs)**

| **Característica** | **Docker (Containers)** | **Máquinas Virtuais (VMs)** |
| --- | --- | --- |
| **Isolamento** | Processos do host (via namespaces) | SO completo virtualizado |
| **Consumo de Recursos** | Leve (compartilha kernel) | Pesado (cada VM tem um SO) |
| **Inicialização** | Segundos | Minutos |
| **Portabilidade** | Alta (imagens únicas) | Média (depende do SO da VM) |

## **Casos de Uso do Docker**

* **Desenvolvimento**: Ambientes consistentes para toda a equipe.
* **CI/CD**: Integração contínua com testes isolados.
* **Microserviços**: Isolamento e escalabilidade de serviços.
* **Deploy em Nuvem**: Facilidade para rodar em qualquer cloud (AWS, Azure, GCP).

### **Dicas Práticas**

1. **Otimize imagens**:
   * Use imagens oficiais (ex: python:3.8-slim).
   * Remova arquivos desnecessários com .dockerignore.
2. **Gerenciamento de recursos**:
   * Limite CPU/memória com --cpus e --memory.
3. **Orquestração**:
   * Use **Kubernetes** ou **Docker Swarm** para gerenciar múltiplos containers.

## **Conclusão**

O Docker revoluciona a entrega de software com:

* **Isolamento seguro** (sem overhead de VMs).
* **Versionamento confiável** (imagens imutáveis).
* **Portabilidade** ("rode em qualquer lugar").

**Próximos passos**:

* Instale o Docker ([Windows](https://docs.docker.com/desktop/install/windows-install/), [Linux](https://docs.docker.com/engine/install/ubuntu/)).
* Experimente comandos básicos (docker run, docker build).

📌 **Dúvidas?** Participe do fórum da Alura para discutir casos reais! 🚀

# **Fluxo de Execução do Comando**docker run

Quando você executa docker run [OPÇÕES] IMAGEM, o Docker segue estas etapas em ordem:

**1. Verificação da Imagem Local**

* O Docker verifica se a **imagem especificada** (ex: nginx:latest) já existe no cache local (docker images).
* Se **não existir**, ele parte para o próximo passo.

**2. da Imagem (Se Necessário)**

* Se a imagem não estiver localmente, o Docker a **baixa do registry padrão** (Docker Hub) ou de um registry customizado.
* Exemplo:

bash

docker pull nginx:latest # (Etapa implícita no `docker run` se a imagem não existir)

**3. Criação do Container**

* O Docker **cria um novo container** a partir da imagem, configurando:
  + **Filesystem isolado** (usando as camadas da imagem + uma camada gravável temporária).
  + **Namespaces** (isolamento de processos, rede, etc.).
  + **Cgroups** (limites de CPU/memória, se definidos).

**4. Configuração da Rede**

* O Docker **conecta o container a uma rede**:
  + Se nenhuma rede for especificada, usa a rede padrão (bridge).
  + Atribui um **IP interno** e mapeia portas (se usado -p HOST:CONTAINER).

**5. Configuração de Variáveis de Ambiente (Se Aplicável)**

* Se houver variáveis definidas (via -e ou no Dockerfile), elas são injetadas no container.

bash

docker run -e "VAR=valor" nginx

**6. Montagem de Volumes (Se Aplicável)**

* Se volumes ou bind mounts forem especificados (-v ou --mount), o Docker os vincula ao container.

bash

docker run -v /pasta/local:/pasta/container nginx

**7. Execução do Comando Principal**

* O Docker **inicia o processo principal** do container (definido por CMD ou ENTRYPOINT no Dockerfile).
* Se um comando customizado for passado no docker run, ele substitui o CMD padrão.

bash

docker run nginx echo "Olá, Docker!" # Substitui o CMD do Nginx

**8. Gerenciamento do Ciclo de Vida**

* O container entra em estado **"Running"**.
* Se o processo principal terminar, o container **para** (a menos que seja usado -d para rodar em segundo plano).

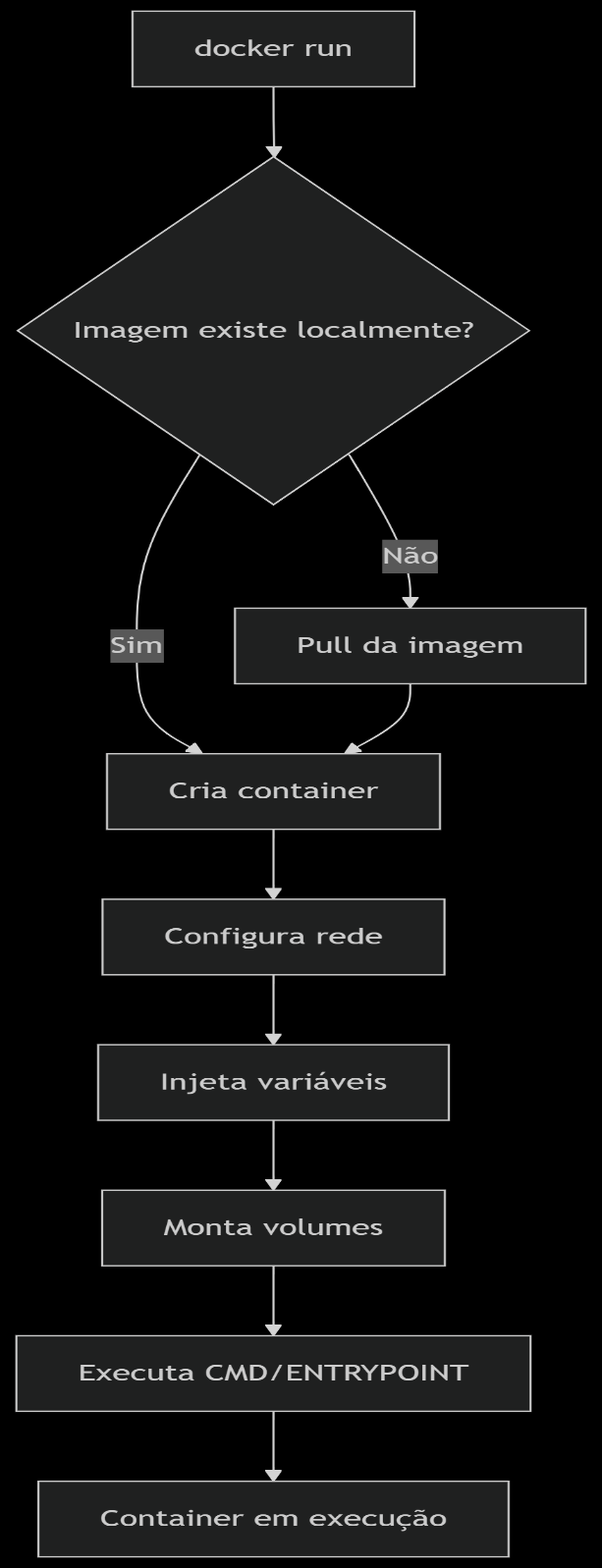
**Exemplo Completo**

bash

docker run -d --name meu-nginx -p 8080:80 -e "ENV=prod" -v ./dados:/usr/share/nginx/html nginx:latest

**Ordem de execução**:

1. Verifica se nginx:latest existe localmente → Se não, baixa do Docker Hub.
2. Cria o container meu-nginx com isolamento.
3. Conecta-o à rede bridge e mapeia a porta 8080 do host para 80 do container.
4. Injeta a variável ENV=prod.
5. Monta o volume ./dados no container.
6. Roda o comando padrão do Nginx (CMD ["nginx", "-g", "daemon off;"]).
7. Mantém o container ativo em segundo plano (-d).



# **Resumo dos Comandos Docker Essenciais**

## **1. Gerenciamento de Containers**

| **Comando** | **Descrição** | **Exemplo** |
| --- | --- | --- |
| docker ps | Lista containers **em execução** | docker ps |
| docker ps -a | Lista **todos** containers (incluindo parados) | docker ps -a |
| docker stop | Para um container (envia sinal SIGTERM) | docker stop <ID/NOME> |
| docker stop -t=0 | Para **imediatamente** (SIGKILL) | docker stop -t=0 <ID> |
| docker start | Reinicia um container parado | docker start <ID> |
| docker pause | Pausa um container (congela processos) | docker pause <ID> |
| docker unpause | Despausa um container | docker unpause <ID> |
| docker rm | Remove um container **parado** | docker rm <ID> |
| docker rm -f | Remove um container **em execução** (forçado) | docker rm -f <ID> |

## **2. Interação com Containers**

| **Comando** | **Descrição** | **Exemplo** |
| --- | --- | --- |
| docker exec -it | Executa um comando **dentro** do container (modo interativo) | docker exec -it <ID> bash |
| docker run -it | Cria e **já acessa** o container (terminal interativo) | docker run -it ubuntu bash |

👉 **Observação**:

* Se o **processo principal** (ex: bash) for encerrado, o container **para**.
* Use sleep ou processos em segundo plano (-d) para manter o container ativo.

## **3. Ciclo de Vida de um Container**

1. **Criação**:

bash

docker run -d --name meu-container ubuntu sleep 1d

* + Cria um container em segundo plano (-d) com o comando sleep 1d.

1. **Acesso**:

bash

docker exec -it meu-container bash

* + Acessa o terminal do container.

1. **Persistência de Dados**:
   * Arquivos criados **dentro** do container são **efêmeros** (perdidos ao remover o container).
   * Solução futura: **Volumes Docker** (persistência).
2. **Encerramento**:
   * Se o processo principal (sleep) terminar, o container **para**.
   * Se você sair do bash (Ctrl+D), o container **continua** (pois sleep ainda está ativo).

## **4. Comparação:**stop**vs**pause

| **Ação** | **Efeito** |
| --- | --- |
| docker stop | Encerra processos (SIGTERM/SIGKILL) → **Reinicia PID ao usar**start. |
| docker pause | Congela processos (não encerra) → **Mantém estado exato ao despausar**. |

## **5. Fluxo Prático**

Diagram

Code

Mermaid rendering failed.

**Dicas Importantes**

1. **Para containers efêmeros**:
   * Use docker run --rm para **auto-remover** o container ao parar.

bash

docker run --rm -it ubuntu bash

1. **Para evitar conflitos de nomes**:
   * Use --name para identificar containers facilmente.

bash

docker run --name meu-app -d nginx

### **Para inspecionar containers**:

* + Use docker inspect para ver detalhes (redes, volumes, etc.).

bash

docker inspect <ID>

**Próximos Passos**

* **Persistência de dados**: Como usar volumes e bind mounts.
* **Dockerfile**: Como criar imagens personalizadas.
* **Orquestração**: Introdução ao docker-compose.

📌 **Key Takeaway**: Containers são **leves, isolados e efêmeros** – gerencie-os com os comandos acima! 🐳

New chat

# **Resumo: Executando um Container Docker com Aplicação Web**

## **Principais Tópicos Abordados**

1. **Exemplo Prático com Aplicação Web**
   * Utilização da imagem não oficial dockersamples/static-site do Docker Hub para demonstração.
   * Diferença entre imagens oficiais (com verificação) e não oficiais.
2. **Execução do Container em Segundo Plano**
   * Uso da flag -d (detached) para rodar o container sem travar o terminal:

bash

docker run -d dockersamples/static-site

1. **Verificação do Container em Execução**
   * Comando docker ps mostra o container ativo, expondo as portas **80** e **443**.
2. **Problema de Acesso à Aplicação**
   * O container roda em um namespace de rede isolado, então localhost:80 não funciona diretamente.
3. **Mapeamento de Portas Automático**
   * Flag -P (maiúsculo) mapeia automaticamente as portas do container para portas aleatórias no host:

bash

docker run -d -P dockersamples/static-site

* + Verificação das portas mapeadas com docker port <ID\_CONTAINER>.

1. **Mapeamento de Portas Manual**
   * Flag -p (minúsculo) permite definir manualmente a porta do host e do container:

bash

docker run -d -p 8080:80 dockersamples/static-site

* + Acesso via http://localhost:8080.

1. **Remoção Forçada do Container**
   * Comando para parar e remover o container em uma única etapa:

bash

docker rm <ID\_CONTAINER> --force

## **Dicas Importantes**

✅ **Prefira imagens oficiais** sempre que possível para maior segurança e confiabilidade.

* O Docker Hub é um grande repositório de imagens que podemos utilizar;
* A base dos containers são as imagens;

✅ **Use**-d para executar containers em segundo plano e liberar o terminal.  
✅ **Mapeie portas** com -P (automático) ou -p HOST:CONTAINER (manual) para expor serviços.  
✅ **Verifique portas** com docker port <ID> ou docker ps para ver o mapeamento.  
✅ **Remova containers parados** com docker rm --force para limpar recursos não utilizados.  
✅ **Acesse a aplicação** no navegador usando a porta mapeada (ex: localhost:8080).

## **Conclusão**: A aula demonstrou como executar, gerenciar e expor uma aplicação web em um container Docker, destacando flags essenciais (-d, -P, -p) e boas práticas.

# **Resumo: O que são Imagens Docker e Como Funcionam?**

## **Principais Tópicos Abordados**

1. **Definição de Imagens Docker**
   * Uma imagem é um **conjunto de camadas imutáveis (read-only)** que, quando combinadas, formam a base para criar containers.
   * Cada camada tem um **ID único** e contém alterações específicas em relação à camada anterior.
2. **Estrutura das Imagens**
   * **Camadas (Layers)**: Arquivos empilhados que representam mudanças incrementais (instalação de pacotes, cópias de arquivos, configurações).
   * **Reutilização**: Camadas são compartilhadas entre imagens para otimizar espaço e desempenho.
3. **Comandos para Analisar Imagens**
   * docker images ou docker image ls: Lista imagens baixadas no sistema.
   * docker inspect <ID\_IMAGEM>: Exibe informações detalhadas (ID, data de criação, configurações).
   * docker history <ID\_IMAGEM>: Mostra as camadas que compõem a imagem e seus tamanhos.
4. **Como uma Imagem Vira um Container?**
   * Ao executar docker run, o Docker:
     1. Baixa as camadas necessárias (se não estiverem no host).
     2. Adiciona uma **camada temporária de leitura/escrita (read-write)** em cima da imagem (read-only).
   * **Dados escritos no container** são armazenados nessa camada temporária e **perdidos quando o container é removido**.
5. **Por que Containers São Leves?**
   * **Compartilhamento de camadas**: Múltiplos containers podem usar a mesma imagem base, adicionando apenas uma camada fina de escrita.
   * **Eficiência**: A imagem original (read-only) não é duplicada, apenas reutilizada.

## **Dicas Importantes**

✅ **Use**docker history para entender como uma imagem foi construída (ordem das camadas e comandos usados).  
✅ **Imagens são imutáveis**: Alterações em containers não afetam a imagem original.  
✅ **Containers efêmeros**: Dados não persistidos em volumes são perdidos ao remover o container.  
✅ **Otimize suas imagens**: Camadas menores e bem organizadas melhoram desempenho e reduz tamanho.

**Próximos Passos**

* **Criação de Imagens Personalizadas**: Aprenderemos a usar o Dockerfile para definir nossas próprias imagens.
* **Persistência de Dados**: Como usar volumes para armazenar informações permanentemente.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

## **Conclusão**: Entender imagens e containers é essencial para dominar o Docker. Com esse conhecimento, você está pronto para criar e gerenciar seus próprios ambientes de forma eficiente! 🚀

# **Resumo: Criando sua Primeira Imagem Docker com Node.js**

## **Principais Tópicos Abordados**

1. **Objetivo**
   * Criar uma imagem Docker personalizada para uma aplicação Node.js simples (exibe "Eu amo Docker!").
   * **Fluxo básico**:
     + Escrever um Dockerfile → Construir a imagem (docker build) → Executar o container (docker run).
2. **Arquivo**Dockerfile
   * **Base da imagem**: Usar a imagem oficial do Node.js (versão 14):

dockerfile

FROM node:14

* + **Diretório de trabalho**: Definir /app-node como pasta padrão no container:

dockerfile

WORKDIR /app-node

* + **Copiar arquivos**: Transferir o código do host para o container:

dockerfile

. .

* + **Instalar dependências**: Executar npm install durante a construção da imagem:

dockerfile

RUN npm install

* + **Ponto de entrada**: Iniciar a aplicação com npm start quando o container rodar:

dockerfile

ENTRYPOINT ["npm", "start"]

1. **Construindo a Imagem**
   * Comando para criar a imagem (a partir do diretório do projeto):

bash

docker build -t danielartini/app-node:1.0 .

* + - -t: Define o nome e a tag da imagem (ex: usuario/nome-da-imagem:versao).

1. **Executando o Container**
   * Mapear a porta do host (8081) para a porta do container (3000):

bash

docker run -d -p 8081:3000 danielartini/app-node:1.0

* + Acessar a aplicação no navegador: http://localhost:8081.

1. **Documentação e Melhores Práticas**
   * Referência à [documentação oficial do Docker](https://docs.docker.com/engine/reference/builder/) para sintaxe do Dockerfile.
   * Instruções úteis não usadas aqui: ENV (variáveis de ambiente), EXPOSE (expor portas), VOLUME (persistência de dados).

## **Dicas Importantes**

✅ **Use imagens oficiais** (ex: node:14) como base para confiabilidade.  
✅ **Defina**WORKDIR para evitar caminhos absolutos repetidos.  
✅ **Ordene instruções do**Dockerfile do menos para o mais frequente alterado (otimiza cache de build).  
✅ **Explicit portas**: Adicione EXPOSE 3000 no Dockerfile para documentar a porta usada pela aplicação.  
✅ **Versionamento**: Sempre use tags (ex: :1.0) para facilitar rollbacks e rastreabilidade.

**Problema Comum e Solução**

* **Como saber a porta da aplicação no container?**
  + Verifique o código-fonte (ex: app.listen(3000) no index.js) ou use EXPOSE no Dockerfile.
  + Se a aplicação não responder, verifique logs do container:

bash

docker logs <ID\_CONTAINER>

## **Próximos Passos**

* **Melhorar o**Dockerfile:
  + Adicionar EXPOSE 3000.
  + Usar variáveis de ambiente (ENV) para configurações flexíveis.
* **Persistir dados**: Aprender sobre volumes Docker para dados não efêmeros.

### **Conclusão**: Você criou sua primeira imagem Docker! Com esse fluxo, você pode empacotar qualquer aplicação e compartilhá-la de forma consistente. 🎉

<https://docs.docker.com/engine/reference/builder/>

Comandos:

docker build -t danielartine/app-node:1 .

docker images

docker run -d –p 8081:3000 danielartine/app-node:1.0

# **Resumo dos Principais Tópicos**

1. **Parando Containers em Massa**
   * Comando para parar todos os containers em execução:

bash

docker stop $(docker container ls -q)

* + A flag -q retorna apenas os IDs dos containers.

1. **Expondo Portas no Dockerfile**
   * A instrução EXPOSE documenta a porta em que a aplicação está rodando dentro do container.
   * Exemplo:

dockerfile

EXPOSE 3000

* + Facilita para outros usuários saberem qual porta mapear.

1. **Trabalhando com Variáveis de Ambiente**
   * **ARG vs. ENV**:
     + ARG: Define variáveis durante o **build** da imagem (ex.: ARG PORT\_BUILD=6000).
     + ENV: Define variáveis que persistem no **container em execução** (ex.: ENV PORT=$PORT\_BUILD).
   * Uso no index.js:

javascript

app.listen(process.env.PORT, ...)

1. **Build e Execução com Variáveis**
   * Build da imagem com nova versão:

bash

docker build -t danielartine/app-node:1.2 .

* + Execução com mapeamento de porta:

bash

docker run -p 9090:6000 -d danielartine/app-node:1.2

1. **Próximos Passos**
   * Deixar a imagem mais parametrizável e semântica.
   * Preparar para deploy no Docker Hub.

Nesta aula do curso "Docker: criando e gerenciando containers", o foco foi na criação e aprimoramento de imagens Docker, especificamente no uso do Dockerfile. Os principais pontos abordados foram:

1. **Documentação da Porta**: Aprendeu-se a usar a instrução EXPOSE no Dockerfile para documentar a porta em que a aplicação está rodando, facilitando o entendimento para outros usuários.
2. **Uso de Variáveis de Ambiente**: Foi introduzido o conceito de variáveis de ambiente, utilizando ARG para definir valores durante a construção da imagem e ENV para definir variáveis que estarão disponíveis dentro do container em execução.
3. **Criação de Imagens**: A aula demonstrou como construir novas versões da imagem com o comando docker build, incluindo a definição de portas e variáveis de ambiente.
4. **Execução de Containers**: Foi mostrado como executar containers com o comando docker run, incluindo o mapeamento de portas entre a máquina host e o container.
5. **Próximos Passos**: A aula concluiu com a promessa de abordar o deploy da imagem no Docker Hub nas próximas aulas.

Esses conceitos ajudam a tornar as imagens Docker mais robustas e fáceis de usar.

## **Dicas**

✅ **Use**EXPOSE para documentar portas no Dockerfile, facilitando o uso por outros desenvolvedores.  
✅ **Prefira variáveis de ambiente (**ENV**)** para configurações dinâmicas, como portas.  
✅ **Use**ARG para valores temporários durante o build da imagem.  
✅ **Combine**-p**com**EXPOSE no docker run para mapeamento explícito de portas.  
✅ **Teste sempre** após mudanças no Dockerfile para garantir que as variáveis e portas funcionam como esperado.

# **Comandos Docker e Persistência de Dados**

## **Principais Tópicos**

1. **Remoção de Containers e Imagens**
   * docker container rm $(docker container ls -aq) → Remove **todos** os containers (parados e em execução).
   * docker rmi $(docker image ls -aq) → Remove **todas** as imagens.
   * --force → Força a remoção de imagens com conflitos.
2. **Verificação de Containers e Imagens**
   * docker ps → Lista containers em execução.
   * docker ps -a → Lista **todos** os containers (incluindo parados).
   * docker images → Lista imagens disponíveis.
3. **Tamanho do Container**
   * docker ps -s → Mostra o tamanho real (camada de read-write) e virtual (imagem base + modificações).
   * docker history <imagem> → Exibe as camadas que compõem uma imagem.
4. **Persistência de Dados**
   * Containers têm uma **camada temporária (read-write)** que é perdida ao removê-los.
   * **3 formas de persistir dados**:
     + **Bind Mount** → Vincula um diretório do host ao container.
     + **Volume** → Armazenamento gerenciado pelo Docker.
     + **tmpfs mount** → Armazenamento temporário em memória (não persiste após reinício).

## **Dicas Importantes**

✔ Use --force se uma imagem não puder ser removida devido a conflitos.  
✔ O **tamanho virtual** de um container é a soma da imagem base + modificações.  
✔ Dados em containers **não são persistentes** por padrão – use **volumes** ou **bind mounts** para armazenamento duradouro.

### **Principais Comandos**

| **Comando** | **Descrição** |
| --- | --- |
| docker container rm $(docker container ls -aq) | Remove todos os containers |
| docker rmi $(docker image ls -aq) --force | Remove todas as imagens (forçado) |
| docker ps -s | Mostra tamanho do container |
| docker history <imagem> | Exibe camadas da imagem |
| docker run -it ubuntu bash | Inicia um container Ubuntu interativo |
| exit | Sai do terminal interativo do container |

**Próximo Passo**: Aprofundar em **volumes** e **bind mounts** para persistência de dados! 🚀

# **Bind Mounts no Docker**

## **Principais Tópicos**

1. **O que é um Bind Mount?**
   * **Ligação entre um diretório do host (sistema operacional) e um diretório dentro do container.**
   * Permite persistir dados mesmo após a remoção do container.
2. **Criando um Bind Mount**
   * **Método 1: Flag**-v**(mais simples)**

bash

docker run -it -v /caminho/no/host:/caminho/no/container ubuntu bash

* + - Exemplo:

bash

docker run -it -v /home/usuario/volume-docker:/app ubuntu bash

→ Tudo em /app no container será salvo em /home/usuario/volume-docker no host.

* + **Método 2: Flag**--mount**(recomendado pela Docker)**

bash

docker run -it --mount type=bind,source=/caminho/no/host,target=/caminho/no/container ubuntu bash

* + - Exemplo:

bash

docker run -it --mount type=bind,source=/home/usuario/volume-docker,target=/app ubuntu bash

* + - **Vantagem**: Mais explícito e semântico.
    - **Cuidado**: Se o caminho no host não existir, o Docker retornará um erro.

1. **Funcionamento Prático**
   * Arquivos criados no diretório montado (ex.: /app) **persistem no host**.
   * Se o container for removido e outro for criado com o mesmo bind mount, os dados **continuam disponíveis**.
   * **Exemplo**:

bash

touch /app/arquivo.txt # No container

→ O arquivo aparecerá em /home/usuario/volume-docker/arquivo.txt no host.

1. **Limitações do Bind Mount**
   * **Dependência do caminho no host**: Se o diretório for movido, renomeado ou excluído, o container não funcionará corretamente.
   * **Problemas de permissão**: O container pode não ter acesso ao diretório no host.
2. **Alternativa Recomendada: Volumes Docker**
   * **Gerenciados pelo Docker** (não dependem de caminhos absolutos no host).
   * Mais seguro e portável (será abordado no próximo tópico).

## **Dicas Importantes**

✔ **Prefira**--mount em vez de -v para maior clareza (recomendação oficial).  
✔ **Verifique o caminho no host** antes de usar bind mounts para evitar erros.  
✔ **Use volumes Docker** se precisar de portabilidade ou evitar dependência do filesystem do host.

**Principais Comandos**

| **Comando** | **Descrição** |
| --- | --- |
| docker run -it -v /host:/container ubuntu bash | Cria um bind mount com -v |
| docker run -it --mount type=bind,source=/host,target=/container ubuntu bash | Cria um bind mount com --mount |
| touch /caminho/no/container/arquivo.txt | Cria um arquivo persistente no bind mount |
| ls /caminho/no/host | Verifica os arquivos persistidos no host |

**Próximo Passo**: Explorar **volumes Docker** para uma solução mais robusta de persistência! 🚀

# **Resumo: Volumes no Docker**

## **Principais Tópicos**

1. **O que são Volumes Docker?**
   * **Área gerenciada pelo Docker** no sistema de arquivos do host.
   * **Mais seguro e recomendado para produção** (evita problemas de permissão e dependência de caminhos absolutos).
   * Dados são armazenados em /var/lib/docker/volumes/.
2. **Criando e Gerenciando Volumes**
   * **Criar um volume**:

bash

docker volume create meu-volume

* + **Listar volumes**:

bash

docker volume ls

* + **Inspecionar um volume**:

bash

docker volume inspect meu-volume

* + **Remover um volume**:

bash

docker volume rm meu-volume

* + **Remover volumes não utilizados**:

bash

docker volume prune

1. **Usando Volumes em Containers**
   * **Método 1: Flag**-v

bash

docker run -it -v meu-volume:/app ubuntu bash

* + - Se o volume não existir, **o Docker o cria automaticamente**.
  + **Método 2: Flag**--mount**(recomendado)**

bash

docker run -it --mount source=meu-volume,target=/app ubuntu bash

* + - Mais explícito e semântico.
    - **Se o volume não existir, o Docker o cria automaticamente**.

1. **Onde os Dados são Armazenados?**
   * **Local padrão**:

bash

/var/lib/docker/volumes/<nome-do-volume>/\_data/

* + **Como acessar**:

bash

sudo su

cd /var/lib/docker/volumes/meu-volume/\_data/

ls

1. **Vantagens dos Volumes sobre Bind Mounts**
   * **Gerenciamento pelo Docker** (não depende de caminhos no host).
   * **Mais seguro** (evita acesso acidental a arquivos do host).
   * **Portabilidade** (facilita migração entre ambientes).
   * **Backup e recuperação simplificados**.

## **Dicas Importantes**

✔ **Prefira**--mount para maior clareza (recomendação oficial).  
✔ **Não é necessário criar o volume antes** – o Docker cria automaticamente se ele não existir.  
✔ **Evite manipular**/var/lib/docker/volumes/**manualmente** – use comandos do Docker (volume rm, prune).  
✔ **Use volumes em produção** para maior segurança e gerenciamento simplificado.

**Alternativa: Usar**docker volume inspect

Se você quer descobrir o caminho exato de um volume **sem navegar manualmente**, use:

bash

**docker volume inspect nome-do-volume**

**Principais Comandos**

| **Comando** | **Descrição** |
| --- | --- |
| docker volume create <nome> | Cria um novo volume |
| docker volume ls | Lista volumes existentes |
| docker volume inspect <nome> | Mostra detalhes do volume |
| docker run -v <volume>:<caminho-no-container> | Monta um volume no container |
| docker run --mount source=<volume>,target=<caminho> | Método recomendado para montar volumes |
| docker volume rm <nome> | Remove um volume |
| docker volume prune | Remove volumes não utilizados |

**Próximo Passo**: Explorar **tmpfs mounts** para armazenamento temporário em memória! 🚀

# **tmpfs no Docker - Armazenamento Temporário em Memória**

## **O que é tmpfs?**

* **Armazenamento temporário em memória RAM** (não persiste após a parada do container).
* **Exclusivo para Linux** (não funciona em Windows/Mac sem configurações extras).
* **Dados sensíveis**: Ideal para informações que **não devem ser persistidas** no filesystem (ex.: tokens, senhas temporárias).

### **Como Usar tmpfs?**

**Método 1: Flag**--tmpfs

bash

docker run -it --tmpfs=/app ubuntu bash

* **Funcionamento**:
  + Cria a pasta /app no container **em memória RAM**.
  + Arquivos em /app são **perdidos** quando o container é removido.

**Método 2: Flag**--mount**(Recomendado)**

bash

docker run -it --mount type=tmpfs,destination=/app ubuntu bash

* **Vantagem**: Mais explícito e alinhado com a sintaxe de outros mounts.

**Características do tmpfs**

| **Feature** | **Detalhe** |
| --- | --- |
| **Persistência** | Dados são **apagados** quando o container é removido. |
| **Performance** | Extremamente rápido (armazenamento em RAM). |
| **Segurança** | Evita vazamento de dados sensíveis (não escreve em disco). |
| **Pastas verdes no**ls | Pastas com tmpfs aparecem em verde (indicando armazenamento temporário). |

## **Exemplo Prático**

1. **Criar container com tmpfs**:

bash

docker run -it --tmpfs=/app ubuntu bash

1. **Criar arquivo temporário**:

bash

touch /app/arquivo-temporario.txt

1. **Verificar conteúdo**:

bash

ls /app # Mostrará o arquivo.

1. **Parar container e recriar**:

bash

exit

docker run -it --tmpfs=/app ubuntu bash

ls /app # Vazio (dados perdidos).

**Quando Usar tmpfs?**

* **Dados efêmeros**: Cache, sessões temporárias, logs não críticos.
* **Segurança**: Evitar persistência de senhas, chaves API ou tokens.
* **Performance**: Aplicações que exigem acesso ultrarrápido a dados temporários.

## **Comparação: Bind Mount vs Volume vs tmpfs**

| **Tipo** | **Armazenamento** | **Persistência** | **Uso Recomendado** |
| --- | --- | --- | --- |
| **Bind Mount** | Diretório do host | Sim | Configurações ou desenvolvimento local. |
| **Volume** | Gerenciado pelo Docker | Sim | Produção (dados persistentes). |
| **tmpfs** | Memória RAM (temporário) | Não | Dados sensíveis ou temporários. |

## **Comandos Principais**

| **Comando** | **Descrição** |
| --- | --- |
| docker run --tmpfs=/caminho | Cria tmpfs no caminho especificado. |
| docker run --mount type=tmpfs,destination=/caminho | Método recomendado para tmpfs. |

## **Dicas Importantes**

⚠️ **Linux apenas**: tmpfs não funciona nativamente em Windows/Mac.  
🚀 **Performance**: Use para dados que precisam de acesso rápido e não requerem persistência.  
🔒 **Segurança**: Ideal para evitar vazamento de informações confidenciais.

**Próximo Passo**: Aplicar esses conceitos em um projeto real com múltiplas camadas de persistência! 🚀

# **Comunicação entre Contêineres no Docker**

## **O Problema da Comunicação**

Contêineres são **isolados por padrão**, mas sistemas complexos exigem comunicação entre eles (ex.: back-end ↔ banco de dados).  
**Como conectar contêineres de forma eficiente e estável?**

### **Solução: Redes Docker**

**1. Rede Padrão**bridge

* **Criada automaticamente** pelo Docker (docker0).
* Contêineres sem rede definida são **conectados a ela**.
* **Comunicação via IP**:
* **Comunicação via IP**:

bash

docker inspect <ID\_CONTÊINER> # Verifica o IP (ex.: 172.17.0.2)

ping 172.17.0.2 # Teste de comunicação (requer `iputils-ping` instalado)

* + **Problema**: IPs mudam se o contêiner for recriado → **Solução instável**.

**2. Listando Redes Existentes**

bash

docker network ls

Saída:

text

NETWORK ID NAME DRIVER SCOPE

80a1db0b6238 bridge bridge local # Rede padrão

df26e341d36e host host local # Usa a rede do host

6ef79c7aa3d6 none null local # Sem rede

### **Como Melhorar a Comunicação?**

**a) Usar Nomes (DNS Interno)**

* Contêineres na **mesma rede** podem se comunicar via **nome** (não apenas IP).
* Exemplo:

bash

docker run --name meu-container -it ubuntu bash

* + Outro contêiner pode acessá-lo via ping meu-container.

**b) Criar Redes Personalizadas**

* **Mais controle** sobre configurações de rede.
* **Melhor isolamento** e segurança.
* Comando:

bash

docker network create minha-rede

docker run --network minha-rede -it ubuntu bash

### **Passo a Passo: Testando Comunicação**

1. **Crie dois contêineres na rede padrão**bridge:

bash

docker run -it --name container1 ubuntu bash

docker run -it --name container2 ubuntu bash

1. **No**container1**, instale o**ping:

bash

apt-get update && apt-get install -y iputils-ping

1. **Teste a comunicação com**container2:

bash

ping container2 # Funciona se estiverem na mesma rede!

**Comparação: Tipos de Rede Docker**

| **Tipo** | **Descrição** | **Uso** |
| --- | --- | --- |
| bridge | Rede padrão (NAT interna) | Comunicação básica entre contêineres. |
| host | Usa a rede do host diretamente | Alta performance (sem isolamento). |
| none | Sem rede | Casos extremos de isolamento. |
| **Custom** | Rede criada pelo usuário | Controle total (recomendado para produção). |

## **Próximos Passos**

* **Redes personalizadas**: Como criar e gerenciar redes específicas para aplicações.
* **DNS automático**: Comunicação estável usando nomes de contêineres.
* **Segurança**: Isolar redes para serviços críticos (ex.: banco de dados).

## **Dicas Importantes**

🔹 **Evite depender de IPs** → Use **nomes de contêineres** ou redes customizadas.  
🔹 **Instale ferramentas de rede** (ping, curl) em imagens mínimas (ex.: ubuntu) para testes.  
🔹 **Produção**: Sempre use **redes personalizadas** para maior controle e segurança.

**Comando útil**:

bash

docker network inspect bridge # Detalhes da rede padrão

No próximo vídeo: **Como criar e gerenciar redes Docker personalizadas!** 🚀