Sumário

[**Resumo e Principais Tópicos** 2](#_Toc200741276)

[**Contexto Inicial** 2](#_Toc200741277)

[**Problemas Encontrados** 2](#_Toc200741278)

[**Soluções Tradicionais e Seus Problemas** 2](#_Toc200741279)

[**Containers como Solução** 2](#_Toc200741280)

[**Dicas** 3](#_Toc200741281)

[**Resumo e Principais Tópicos** Responder às perguntas: 4](#_Toc200741282)

[**1. Por que Containers São Mais Leves que VMs?** 4](#_Toc200741283)

[**2. Como os Containers Garantem Isolamento?** 4](#_Toc200741284)

[**3. Como Funcionam sem Instalar um SO?** 4](#_Toc200741285)

[**4. Divisão de Recursos (CPU, Memória, etc.)** 5](#_Toc200741286)

[**Exemplo**: 5](#_Toc200741287)

[**Conclusão** 5](#_Toc200741288)

[Documentações: **Ubuntu | Docker Docs Aprenda mais sobre Máquinas Virtuais com o Artigo VirtualBox e Máquinas Virtuais** 6](#_Toc200741289)

[**Resumo: Docker e Virtualização com Containers** 7](#_Toc200741290)

[**O que é o Docker?** 7](#_Toc200741291)

[**Principais Benefícios do Docker** 7](#_Toc200741292)

[**Como o Docker Funciona?** 8](#_Toc200741293)

[**Casos de Uso do Docker** 8](#_Toc200741294)

[**Dicas Práticas** 8](#_Toc200741295)

[**Conclusão** 9](#_Toc200741296)

[**Fluxo de Execução do Comando**docker run 10](#_Toc200741297)

[**Resumo dos Comandos Docker Essenciais** 13](#_Toc200741298)

[**1. Gerenciamento de Containers** 13](#_Toc200741299)

[**2. Interação com Containers** 13](#_Toc200741300)

[**3. Ciclo de Vida de um Container** 14](#_Toc200741301)

[**4. Comparação:**stop**vs**pause 14](#_Toc200741302)

[**5. Fluxo Prático** 15](#_Toc200741303)

[**Resumo: Executando um Container Docker com Aplicação Web** 16](#_Toc200741304)

[**Principais Tópicos Abordados** 16](#_Toc200741305)

[**Dicas Importantes** 17](#_Toc200741306)

[**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. 18](#_Toc200741307)

# **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

Copy

Download

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

Copy

Download

FROM python:3.8 # Imagem base

COPY . /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. Download 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

Copy

Download

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

Copy

Download

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

Copy

Download

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

Copy

Download

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

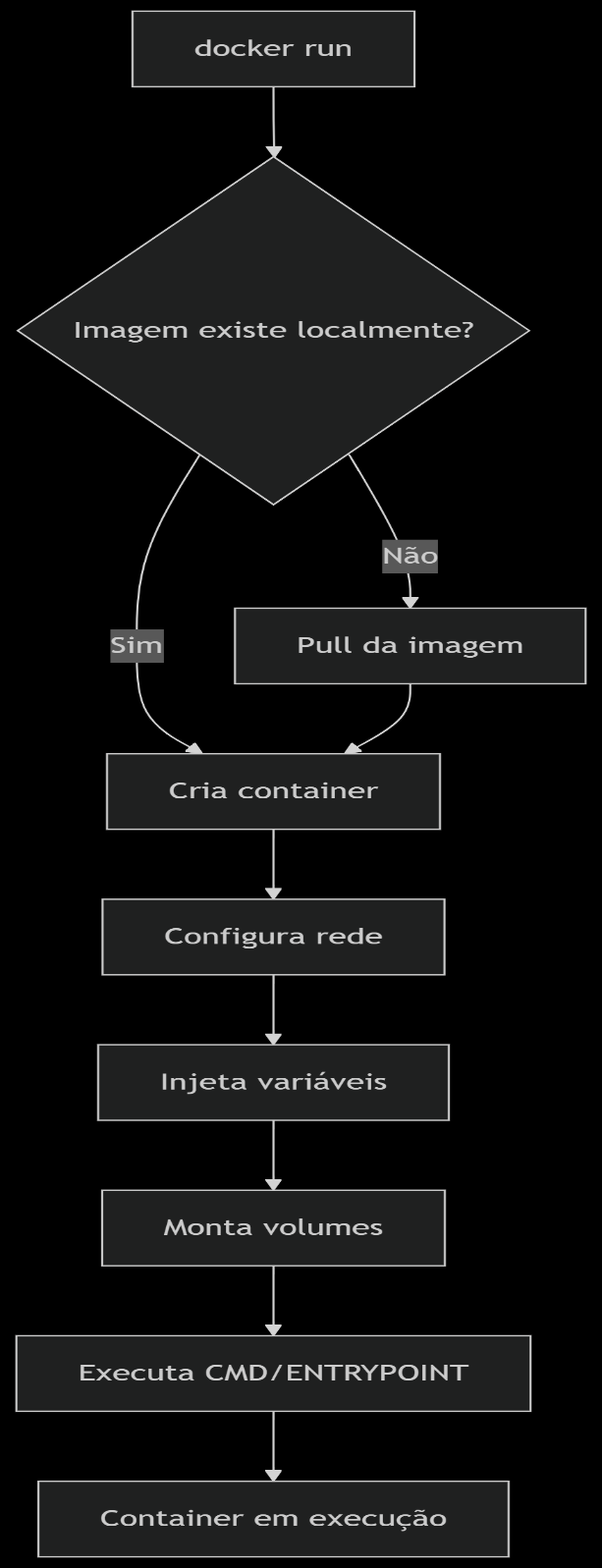
Copy

Download

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

Copy

Download

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

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

1. **Acesso**:

bash

Copy

Download

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

Download

Mermaid rendering failed.

**Dicas Importantes**

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

bash

Copy

Download

docker run --rm -it ubuntu bash

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

bash

Copy

Download

docker run --name meu-app -d nginx

1. **Para inspecionar containers**:
   * Use docker inspect para ver detalhes (redes, volumes, etc.).

bash

Copy

Download

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

Copy

Download

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

Copy

Download

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

Copy

Download

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

Copy

Download

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! 🚀