***Curso de Docker***

**O que é Docker?**

N

## 🔹 1. O que é Docker?

Docker é uma plataforma de **virtualização baseada em containers** que permite empacotar uma aplicação e todas as suas dependências em um ambiente isolado.

O objetivo do Docker é permitir que um software rode de maneira **consistente e previsível** em diferentes ambientes, eliminando o problema do "funciona na minha máquina, mas não no servidor".

Ao contrário de máquinas virtuais tradicionais, que incluem um sistema operacional inteiro, **containers compartilham o kernel do sistema** e apenas empacotam as bibliotecas e dependências da aplicação. Isso torna os containers **leves, rápidos e eficientes**.

**🟢 Benefícios do Docker:**  
✅ Portabilidade → O mesmo container pode rodar em qualquer ambiente (Windows, Linux, Mac, servidores em nuvem, etc.).  
✅ Eficiência → Consome menos recursos do que máquinas virtuais, pois compartilha o kernel do host.  
✅ Escalabilidade → Facilita o deploy de múltiplas instâncias da mesma aplicação.  
✅ Automação → Integrado facilmente em pipelines de CI/CD para automatizar deploys.  
✅ Isolamento → Cada container roda separadamente, evitando conflitos entre aplicações.

## 🔹 2. Conceitos Fundamentais

### 🏗️ ****Containers vs Máquinas Virtuais****

| Característica | Containers (Docker) | Máquinas Virtuais (VMs) |
| --- | --- | --- |
| Isolamento | Compartilham o Kernel do sistema | Virtualizam um SO inteiro |
| Tamanho | Pequenos (MBs) | Grandes (GBs) |
| Desempenho | Rápido | Mais lento |
| Inicialização | Segundos | Minutos |
| Uso de recursos | Menos memória e CPU | Mais pesado |

Containers são **muito mais leves** que VMs porque **não precisam de um sistema operacional completo** dentro de cada instância.

### 📦 ****Imagens e Containers****

* **Imagem:** Um "blueprint" de um container, contendo o código-fonte, dependências e configurações necessárias.
* **Container:** Uma instância em execução de uma imagem.

As imagens podem ser armazenadas em repositórios como o **Docker Hub** ou um **Docker Registry privado**.

## 🔹 3. Como o Docker Funciona?

O Docker tem **três componentes principais**:

1️⃣ **Docker Engine** → Responsável por criar e gerenciar containers.  
2️⃣ **Docker CLI** → Interface de linha de comando para interagir com o Docker.  
3️⃣ **Docker Daemon** → Serviço em segundo plano que controla os containers.

**Fluxo de trabalho básico do Docker:**

1. O desenvolvedor escreve um **Dockerfile**, especificando como construir uma imagem.
2. O Docker constrói essa imagem com todas as dependências.
3. A imagem é armazenada em um **Docker Registry** (público ou privado).
4. Outros desenvolvedores ou servidores baixam essa imagem e criam um **container** a partir dela.

## 🔹 4. Dockerfile: Criando Imagens Personalizadas

O **Dockerfile** é um arquivo de texto contendo instruções para construir uma imagem Docker.

**Principais comandos de um Dockerfile:**

* FROM → Define a imagem base (exemplo: FROM ubuntu:latest).
* RUN → Executa comandos durante a construção da imagem.
* COPY → Copia arquivos para dentro do container.
* CMD e ENTRYPOINT → Define o comando principal que será executado quando o container iniciar.
* EXPOSE → Expõe portas do container.
* ENV → Define variáveis de ambiente.

## 🔹 5. Docker Compose: Orquestração de Múltiplos Containers

O **Docker Compose** permite definir e rodar múltiplos containers usando um único arquivo (docker-compose.yml).

💡 **Casos de uso do Docker Compose:**

* Criar ambientes de desenvolvimento completos.
* Rodar aplicações que dependem de múltiplos serviços (exemplo: banco de dados, backend e frontend).
* Automatizar a inicialização e configuração de serviços interdependentes.

**Principais elementos do** docker-compose.yml**:**

* services → Define os containers a serem executados.
* volumes → Define armazenamento persistente.
* networks → Configura a comunicação entre containers.
* depends\_on → Define dependências entre serviços.

## 🔹 6. Redes no Docker

O Docker permite criar redes isoladas para containers se comunicarem.

**Tipos de redes no Docker:**  
1️⃣ **Bridge (padrão)** → Containers na mesma rede podem se comunicar, mas não com o host.  
2️⃣ **Host** → O container compartilha a rede do sistema host diretamente.  
3️⃣ **Overlay** → Usado em clusters para comunicação entre múltiplos hosts.

💡 **Por que redes são importantes?**

* Permitem comunicação segura entre containers.
* Evitam exposição desnecessária de portas.
* Facilitam a organização de serviços distribuídos.

## 🔹 7. Persistência de Dados: Volumes e Bind Mounts

Por padrão, os dados dentro de um container **são temporários**. Para manter dados persistentes, o Docker usa **volumes** e **bind mounts**.

* **Volumes:** Criados e gerenciados pelo Docker. Melhor para dados persistentes.
* **Bind Mounts:** Usa um diretório do host dentro do container. Melhor para desenvolvimento.

**Exemplos de uso:**  
✅ **Bancos de dados:** Evita perda de dados quando o container é reiniciado.  
✅ **Logs:** Armazena logs de forma persistente.  
✅ **Configurações e arquivos compartilhados:** Útil para comunicação entre containers.

## 🔹 8. Segurança no Docker

Containers **não são isolados completamente** do sistema operacional host. Algumas boas práticas de segurança incluem:

🔒 **Práticas recomendadas:**

* **Evitar rodar containers como root** → Sempre use um usuário não privilegiado.
* **Usar imagens oficiais e seguras** → Evita pacotes maliciosos.
* **Restringir permissões de rede** → Não expor portas desnecessárias.
* **Usar escaneamento de vulnerabilidades** → Ferramentas como docker scan ajudam a identificar problemas.
* **Aplicar políticas de segurança** → Usar AppArmor, SELinux e seccomp para restringir ações dos containers.

## 🔹 9. Docker Swarm e Kubernetes

Para **orquestração de containers** em larga escala, existem duas principais soluções:

1️⃣ **Docker Swarm:**

* Orquestrador embutido no Docker.
* Fácil de configurar, mas menos poderoso que Kubernetes.

2️⃣ **Kubernetes:**

* Plataforma robusta para gerenciar containers em cluster.
* Suporte avançado para escalabilidade e alta disponibilidade.

Docker Swarm é mais simples para projetos pequenos, enquanto Kubernetes é mais usado para **grandes aplicações distribuídas**.

## 🔹 10. Boas Práticas com Docker

✅ **Mantenha as imagens pequenas** → Evite instalar pacotes desnecessários.  
✅ **Use multi-stage builds** → Reduz o tamanho da imagem final.  
✅ **Evite rodar containers como root** → Use um usuário seguro.  
✅ **Armazene logs e dados em volumes** → Evita perda de dados.  
✅ **Monitore os containers** → Ferramentas como Prometheus e Grafana ajudam a monitorar performance e segurança.