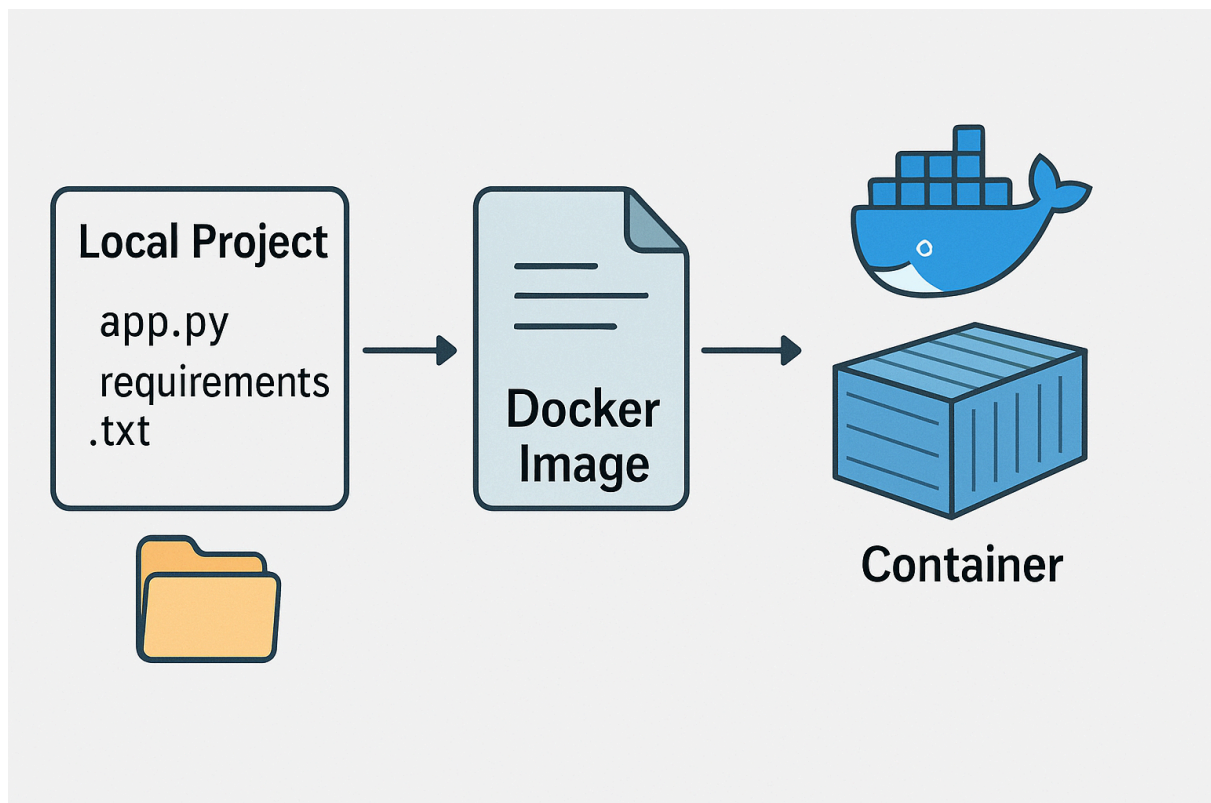




## O que é o Docker?

Imagine que você tem um projeto (um site, um app, uma API...). Para rodar esse projeto, você precisa instalar um monte de coisas no seu computador: Python, bibliotecas, banco de dados, configurações certas, etc. Agora pense: e se você quisesse passar esse projeto para outra pessoa? Ou rodar em outro computador, ou até em um servidor? Isso pode dar problema, porque cada ambiente pode ser diferente...

É aí que o Docker entra: ele cria um ambiente fechado, isolado e controlado para o seu projeto rodar — como se fosse uma caixinha com tudo que ele precisa lá dentro.



---

## O que é um *Container*?

Um container é essa "caixinha" que o Docker monta para rodar seu projeto. Dentro dela, vai estar:

- O seu código
- O sistema operacional (Linux leve)
- Todas as dependências que o projeto precisa
- Tudo configuradinho



**Pensa num container como um potinho de comida:**

**Você pode cozinhar em qualquer lugar, colocar tudo dentro do potinho e levar pra onde quiser. A comida vai estar igual, seja no seu micro-ondas, seja no de outra pessoa.**

---

## O que é uma *Imagem Docker*?

Uma imagem Docker é como uma receita para criar esse potinho.

- Ela define o que vai estar dentro do container.
- Você pode criar a sua própria imagem (com um arquivo chamado `Dockerfile`) ou usar imagens prontas (como do Python, Node.js, etc).



**Pensa na imagem como o molde do potinho. O Docker usa esse molde para construir um container toda vez que você quiser rodar seu projeto.**

---

## E o meu projeto local, como entra nisso?

Vamos dizer que você tem um projeto local, com arquivos `.py`, por exemplo. Para rodá-lo com Docker, você:

1. Cria uma imagem Docker dizendo:
  - "Quero que instale Python 3.10"
  - "Quero que copie meu projeto pra dentro do container"
  - "Quero que rode `python app.py`"

2. Roda um container com base nessa imagem.

Pronto! Agora seu projeto está rodando dentro do container, e você pode:

- Testar localmente com o ambiente isolado
- Compartilhar a imagem com alguém
- Subir pro servidor (com o mesmo comportamento)

### Por que isso é útil?

- Acaba com o "na minha máquina funciona"
- Padroniza o ambiente entre o time
- Facilita o deploy (subir para produção)
- Roda em qualquer lugar que tenha Docker

---

## O que é o **.dockerignore**?

O arquivo **.dockerignore** serve para dizer ao Docker **quais arquivos ou pastas ele deve ignorar quando estiver construindo a imagem** do seu projeto.

Imagina que você tem um projeto com essas pastas e arquivos:

```
task-manager/  
├─ app.py  
├─ requirements.txt  
├─ .env  
├─ node_modules/  
├─ __pycache__/  
└─ .git/
```

Quando o Docker for construir a imagem, por padrão ele **vai copiar tudo isso** pro container.

😱 Mas pera... você **não precisa de tudo isso** lá dentro!

- `.git/`: é só pro controle de versão, não precisa no container.
- `__pycache__`/: são arquivos temporários do Python.
- `node_modules/`: pode ser muito pesado e inútil se o container for de Python.
- `.env`: talvez tenha senhas sensíveis que você não quer enviar para produção!

## ✅ A solução: usar o `.dockerignore`

Você cria um arquivo chamado `.dockerignore` e coloca os nomes dos arquivos e pastas que quer que o Docker ignore. Exemplo:

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

Com isso, o Docker **não vai copiar esses arquivos** para dentro da imagem. Isso deixa sua imagem:

- Menor e mais leve
  - Mais rápida de construir
  - Mais segura (evita mandar dados sensíveis sem querer)
-

## O que é o `docker-compose.yml`?

Quando seu projeto precisa **de mais de um container**, ou quando você quer organizar bem como o container é rodado (portas, volumes, variáveis...), escrever tudo no terminal pode virar uma bagunça. 😓

O `docker-compose.yml` resolve isso! Ele é um **arquivo de configuração** onde você descreve tudo que o Docker precisa para rodar seu projeto com um só comando.

### Pra que serve?

- Rodar **vários containers ao mesmo tempo** (ex: app + banco de dados)
- Definir **ambientes** de forma clara e legível
- Subir seu projeto com um simples:

```
docker-compose up
```

### Exemplo prático

Imagina que seu projeto tem:

- Um app em Python (Flask)
- Um banco de dados PostgreSQL

O `docker-compose.yml` poderia ser assim:

```
version: '3.8'

services:
  web:
    build: .
    ports:
      - "5000:5000"
```

```
volumes:
  - ./app
depends_on:
  - db

db:
  image: postgres:13
  environment:
    POSTGRES_USER: usuario
    POSTGRES_PASSWORD: senha
    POSTGRES_DB: meu_banco
```

## O que está acontecendo aqui?

- **services:** → define os containers do projeto
- **web:** → é o serviço do seu app
  - **build: .** → vai usar o **Dockerfile** na pasta atual
  - **ports: "5000:5000"** → expõe a porta 5000
  - **depends\_on: db** → só sobe o app depois do banco estar pronto
- **db:** → é o serviço do banco
  - usa uma imagem pronta do PostgreSQL
  - define as variáveis de ambiente do banco



## Vantagens de usar o **docker-compose**

- **Organização:** tudo num arquivo só
- **Reusável:** serve pro time todo
- **Escalável:** fácil adicionar novos serviços
- **Automação:** perfeito pra rodar local ou em servidores