

Objetivos:

- I. Docker Compose;
- II. Estrutura do projeto e organização dos serviços;
- III. Estrutura do arquivo docker-compose.yml;
- IV. Comandos com Docker Compose;
- V. Persistência de dados com volumes no Docker.

Atenção: Para reproduzir os exemplos e exercícios, utilize o seguinte repositório:
<https://github.com/arleysouza/docker-compose>.

I. Docker Compose

Ao desenvolver aplicações modernas, é comum depender de múltiplos serviços que precisam ser executados em conjunto, como servidores web, redes, bancos de dados, mensagerias, entre outros. Gerenciar manualmente vários containers que se comunicam entre si pode tornar-se uma tarefa trabalhosa e sujeita a erros. Para facilitar esse processo, o Docker disponibiliza uma ferramenta chamada Docker Compose.

O Docker Compose é uma ferramenta que permite definir e gerenciar múltiplos containers Docker por meio de um único arquivo de configuração escrito em YAML (*YAML Ain't Markup Language* - em português, YAML Não é Linguagem de Marcação). Com essa ferramenta, é possível descrever toda a arquitetura da aplicação - como serviços, volumes, redes e variáveis de ambiente - em um único documento, geralmente chamado `docker-compose.yml` ou `compose.yml`. É importante notar que tanto `.yaml` quanto `.yml` são extensões válidas para arquivos YAML, sendo que a escolha entre elas é uma questão de preferência.

Ao invés de executar comandos longos e manuais como `docker run`, o desenvolvedor pode simplesmente utilizar comandos como `docker compose up` e `docker compose down` para iniciar e parar todos os containers da aplicação de maneira coordenada.

Benefícios do uso do Docker Compose:

- Automação: elimina a necessidade de iniciar manualmente cada container;
- Organização: centraliza a definição dos serviços em um único arquivo;
- Reprodutibilidade: torna o ambiente de desenvolvimento e testes mais previsível;
- Escalabilidade local: facilita a replicação de ambientes mais próximos da produção;
- Integração com volumes e redes: permite definir volumes persistentes e redes privadas entre os containers.

II. Estrutura do projeto e organização dos serviços

A aplicação usada como exemplo é constituída por backend Node.js e banco de dados PostgreSQL, os serviços serão mantidos em containers distintos.

O projeto possui a seguinte estrutura de diretórios:

```
app/
├── db/
│   └── init.sql
├── server/
│   ├── src/
│   │   ├── config/
│   │   ├── controllers/
│   │   └── routes/
│   └── index.ts
├── .dockerignore
├── Dockerfile
├── package-lock.json
├── package.json
├── tsconfig.json
└── compose.yml
```

- O arquivo `compose.yml` deve estar localizado na raiz do projeto;
- Cada subprojeto que será transformado em imagem Docker deve conter um `Dockerfile`, com os passos necessários para construir a imagem e subir o container correspondente.

III. Estrutura do arquivo `docker-compose.yml`

O arquivo `docker-compose.yml` é essencial para orquestrar múltiplos containers de forma coordenada. Ele permite definir serviços, redes, volumes e variáveis de ambiente de maneira organizada e reproduzível.

A seguir, apresentamos o arquivo `compose.yml` utilizado no estudo de caso desta aula, que demonstra a execução conjunta de uma aplicação backend e um banco de dados PostgreSQL.

Exemplo de `compose.yml`:

```
services:
  db:
    image: postgres:17-alpine
    container_name: db-container
    restart: always
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: 123
      POSTGRES_DB: docker-aula
    ports:
      # Mapeia a porta externa 5433 (host) para a 5432 (container)
      - "5433:5432"
    volumes:
      - meu-volume:/var/lib/postgresql/data
      - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql:ro
    networks:
```

```
- minha-rede

server:
  build:
    context: ./server
  container_name: server-container
  ports:
    - "3001:3000"
  environment:
    PORT: 3000
    DB_HOST: db # o nome do serviço "db" funciona como hostname
    DB_USER: postgres
    DB_PASSWORD: 123
    DB_NAME: docker-aula
    DB_PORT: 5432 # Dentro do container, a porta do Postgres é 5432
    NODE_ENV: production
  depends_on:
    - db
  networks:
    - minha-rede

volumes:
  meu-volume:

networks:
  minha-rede:
    driver: bridge
```

Explicações:

services: define os containers que compõem a aplicação:



- **db:** serviço do banco de dados PostgreSQL;
- **server:** aplicação backend que se conecta ao banco.

Serviço db

- **image:** especifica a imagem a ser utilizada (`postgres:17-alpine`);
- **container_name:** define um nome fixo para o container;
- **restart:** reinicia o container automaticamente em caso de falha;
- **environment:** variáveis de ambiente exigidas pelo PostgreSQL;
- **ports:** mapeia a porta `5432` do container para a porta `5433` do host;
- **volumes:** monta dois volumes:
 - Monta um volume persistente para o banco de dados (`meu-volume` recebe o caminho `/var/lib/postgresql/data`);

[Containers](#) / db-container

db-container

<  768264318c45  postgres:17-alpine
5433:5432



Logs Inspect Bind mounts Exec **Files** Stats

Name ↑	Note
var	
> cache	
> empty	
lib	
> misc	
postgresql	
> data	VOLUME

- Monta o script de inicialização no diretório apropriado (`init.sql` montado como leitura `ro` – read only).

[Containers](#) / db-container

db-container

<  768264318c45  postgres:17-alpine
5433:5432

Logs Inspect Bind mounts **Exec** Files

```
/ # cd docker-entrypoint-initdb.d/
/docker-entrypoint-initdb.d # ls
init.sql
/docker-entrypoint-initdb.d # more init.sql
CREATE TABLE IF NOT EXISTS users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL
);
```

- **networks**: conecta o container à rede personalizada (`minha-rede`).

Serviço server

- **build.context**: aponta para a pasta onde está o `Dockerfile`;
- **container_name**: nome fixo para o container;
- **ports**: mapeia a porta `3000` do container para a `3001` do host;
- **environment**: define variáveis de ambiente, inclusive dados de conexão com o banco;
- **depends_on**: garante que o serviço `db` seja iniciado antes;
- **networks**: conecta o serviço à mesma rede do banco.

volumes

Define o volume nomeado `meu-volume`, responsável por persistir os dados do banco.

networks

Define a rede nomeada **minha-rede** com driver do tipo **bridge**.

Execução de comandos SQL do `init.sql`

Quando o volume **meu-volume** é criado pela primeira vez, o PostgreSQL detecta que ele está vazio e, então:

1. Cria o banco (**POSTGRES_DB**);
2. Cria o usuário e senha (**POSTGRES_USER** e **POSTGRES_PASSWORD**);
3. Executa os scripts presentes em `/docker-entrypoint-initdb.d/` - nesse caso, o `init.sql`.

Em execuções posteriores, se o volume persistente já existir, o PostgreSQL não executará novamente o script `init.sql`, preservando os dados existentes.

IV. Comandos com Docker Compose

Verificar a versão instalada:

```
docker compose version
```

Listar e remover volumes:

```
docker volume ls
docker volume rm <nome_do_volume>
```

Listar e remover redes:

```
docker network ls
docker network rm <nome_da_rede>
```

Execute os comandos a seguir no diretório onde se encontra o `compose.yml`.

Subir os containers e construir as imagens:

```
docker compose up --build -d
```

Apenas subir os containers:

```
docker compose up
```

Este comando realiza:

1. Build da imagem
 - Recria as imagens para os serviços com instrução **build:**.
2. Criação de rede, volume e container
 - Cria os recursos definidos no `compose.yml`, se ainda não existirem.
3. Execução dos serviços em background
 - O `-d` mantém os containers executando em segundo plano.

Verificar se os containers estão em execução:

```
docker compose ps
```

Parar os containers (sem remover volumes e redes):

```
docker compose down
```

Parar os containers e remover recursos órfãos:

```
docker compose down --volumes --remove-orphans
```

Visualizar logs de um container específico:

```
docker logs <nome_do_container>
# Acompanhar logs em tempo real de um serviço
docker logs -f <nome_do_container>
docker compose logs -f <nome_do_serviço>
# Acompanhar logs em tempo real de todos os serviços
docker compose logs -f
```

V. Persistência de dados com volumes no Docker

Por padrão, os containers Docker são efêmeros. Isso significa que qualquer dado escrito dentro do sistema de arquivos do container é perdido assim que ele é destruído ou reiniciado. Em aplicações que utilizam bancos de dados ou manipulam arquivos, essa característica é problemática. Para contornar essa limitação, o Docker oferece uma funcionalidade chamada volume, que permite a persistência e o compartilhamento de dados entre containers, de forma segura e eficiente.

Volumes são áreas de armazenamento gerenciadas pelo próprio Docker e que existem fora do ciclo de vida dos containers. Eles são mantidos em diretórios internos, como `/var/lib/docker/volumes/`, e não dependem de caminhos específicos no sistema de arquivos do host. Por serem gerenciados pelo Docker, os volumes oferecem maior controle e abstração na manipulação de dados persistentes.

Vantagens do uso de volumes:

- Persistência de dados após remoção do container;
- Compartilhamento de dados entre containers;
- Maior segurança e isolamento do sistema de arquivos do host;
- Integração com ferramentas de backup, restauração e monitoramento.

Para visualizar os detalhes de um volume:

```
docker volume inspect <nome_do_volume>
```

```
C:\>docker volume ls
DRIVER      VOLUME NAME
local       app_meu-volume

C:\>docker volume inspect app_meu-volume
[
  {
    "CreatedAt": "2025-07-27T16:26:02Z",
    "Driver": "local",
    "Labels": {
      "com.docker.compose.config-hash": "e0faf2a0ad773922bb0c9d2139f1db356eee997a0034ef43fe0a8d714c6f6807",
      "com.docker.compose.project": "app",
      "com.docker.compose.version": "2.38.2",
      "com.docker.compose.volume": "meu-volume"
    },
    "Mountpoint": "/var/lib/docker/volumes/app_meu-volume/_data",
    "Name": "app_meu-volume",
    "Options": null,
    "Scope": "local"
  }
]
```

VI. Exercícios

Veja o vídeo se tiver dúvidas - <https://youtu.be/7in8yJSzPzU>

Exercício 1 – Adicionar serviço no `compose.yml`

Objetivo: Adicionar o serviço `frontum` no arquivo `compose.yml` para criar uma imagem baseada no projeto da pasta `front-um/` e expô-la na porta 3002 do host.

Tarefa:

- No arquivo `compose.yml`, adicione o serviço do projeto disponível na pasta `front-um/`:
 - O serviço deve estar disponível na porta 3002 do host;
 - Defina o nome do container como `um-container`.
- No CMD do Windows, liste os serviços que estão rodando. Os containers `db-container` e `server-container` devem estar em execução;
- Construa e execute o container `um-container` e teste no navegador <http://localhost:3002>.

Imagem `nginx:alpine` usada no `Dockerfile`

A imagem `nginx` é uma das mais populares no Docker Hub e fornece um servidor web Nginx pronto para uso em containers. O Nginx é conhecido por sua alta performance, baixo consumo de recursos e capacidade de atuar como:

- Servidor web: para hospedar sites estáticos ou dinâmicos;
- Proxy reverso: encaminhando requisições para outras aplicações;
- Balanceador de carga: distribuindo tráfego entre múltiplos servidores.

Resposta:

- Adicionar o serviço no arquivo `compose.yml`

```
frontum:
  build: ./front-um
  container_name: um-container
  ports:
    - "3002:80"
  networks:
    - minha-rede
  depends_on:
    - server
```

- b) `docker ps`
- c) `docker compose build frontum`
`docker compose up -d frontum`
`docker ps`

Exercício 2 – Adicionar serviço no `compose.yml`

Objetivo: Adicionar o serviço `frontdois` no arquivo `compose.yml`, utilizando a pasta `front-dois/`, que contém um projeto React com TypeScript baseado em Vite, e disponibilizá-lo na porta 3003 do host.

Tarefa:

- a) No arquivo `compose.yml`, adicione o serviço do projeto disponível na pasta `front-dois/`:
 - O serviço deve estar disponível na porta 3003 do host;
 - Defina o nome do container como `dois-container`.
- b) No CMD do Windows, liste os serviços que estão rodando. Os containers `db-container`, `um-container` e `server-container` devem estar em execução;
- c) Construa e execute o container `dois-container` e teste no navegador <http://localhost:3003>.

Observações:

Para que o projeto React com Vite seja acessível na porta 3003 do host, as seguintes condições devem ser atendidas:

- O Dockerfile do projeto já está configurado para build da aplicação;
- É necessário expor corretamente a porta 3003 no arquivo `compose.yml` (tarefa a ser realizada neste exercício);
- O arquivo `vite.config.ts` já está configurado para aceitar conexões externas (por exemplo, utilizando `host: true` na configuração do servidor).