Dockerizando a nossa aplicação e introduzindo o PostgreSQL

> https://fastapidozero.dunossauro.com/10/

Objetivos dessa aula

- Entender como criar uma imagem Docker para a nossa aplicação
- Aprender a rodar a aplicação utilizando Docker
- Introduzir o conceito de Docker Compose para múltiplos contêineres
- Aprender o que é um Dockerfile e sua estrutura
- Entender os benefícios e motivos da mudança para o PostgreSQL

Parte 1

> Docker e Postgres

Docker

Docker é uma ferramenta para criar containers.

Containers são formas de **isolar as dependências**. Da mesma forma que fazemos com o ambiente virtual. Que isolam as dependências do python.

No caso dos containers Docker, estamos falando de isolamento de ferramentas do sistema operacional. Isolamos programas e ambientes de forma completa

> Cruso gratuito sobre docker da linuxtips

PostgreSQL

Um Banco de Dados Relacional de código aberto.

- `Escalabilidade`: SQLite não é ideal para aplicações em larga escala ou com grande volume de dados. PostgreSQL foi projetado para lidar com uma grande quantidade de dados e requisições.
- `Concorrência`: diferentemente do SQLite, que tem limitações para gravações simultâneas, o PostgreSQL suporta múltiplas operações simultâneas.
- `Funcionalidades avançadas`: PostgreSQL vem com várias extensões e funcionalidades que o SQLite pode não oferecer.

Nota importante

> Embora para o escopo da nossa aplicação e os objetivos de aprendizado o SQLite pudesse ser suficiente, é sempre bom nos prepararmos para cenários de produção real. A adoção de PostgreSQL nos dá uma prévia das práticas do mundo real e garante que nossa aplicação possa escalar sem grandes modificações de infraestrutura.

Executando o postgres com docker

```
docker run \
    --name app_database \
    -e POSTGRES_USER=app_user \
    -e POSTGRES_DB=app_db \
    -e POSTGRES_PASSWORD=app_password \
    -p 5432:5432 \
    postgres
```

Essa instrução vai iniciar um container do postgres no nosso pc e disponibilizando ele na porta `5432`.

Conectando nossa aplicação ao postgres

Precisamos instalar o driver para o postgres na nossa aplicação:

```
poetry add "psycopg[binary]"
```

Também precisamos alterar a URL do banco de dados

```
# env
DATABASE_URL="postgresql+psycopg://app_user:app_password@127.0.0.1:5432/app_db"
```

Subindo a aplicação

```
task run
```

Erro na migração

Ao fazer uma chamada que depende do banco de dados, vamos obter um:

`Internal Server Error`

Se olharmos o shell

```
sqlalchemy.exc.ProgrammingError: (psycopg.errors.UndefinedTable) relation "users" does not exist
LINE 2: FROM users
```

A tabela `users` não existe na nossa aplicação.

Executando as migrações no novo banco

```
alembic upgrade head
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 74f39286e2f6, create users table
INFO  [alembic.runtime.migration] Running upgrade 74f39286e2f6 -> 3a79a86c9e4a, create todos table
```

Com isso, tudo deve funcionar como esperado!

Os testes envolvendo o postgres

Se executarmos os testes, eles vão continuar passando, pois os dados estão fixos na fixture:

```
@pytest.fixture
def session():
    engine = create_engine(
        'sqlite:///:memory:',
        connect_args={'check_same_thread': False},
        poolclass=StaticPool,
    )
    # ...
```

Usando as variáveis de ambiente no teste

```python
from fast_zero.settings import Settings
# ...

@pytest.fixture
def session():
    engine = create_engine(Settings().DATABASE_URL)
    table_registry.metadata.create_all(engine)
```

Porém, agora temos um novo problema :)

O container do banco de dados precisa estar rodando

```
E           sqlalchemy.exc.OperationalError: (psycopg.OperationalError) connection failed:
E    connection to server at "127.0.0.1", port 5432 failed: Connection refused
E               Is the server running on that host and accepting TCP/IP connections?
E           (Background on this error at: https://sqlalche.me/e/20/e3q8)

OperationalError
```

Testando com Docker

Existe uma biblioteca python que gerencia as dependências de containers externos para que a aplicação seja executada. O `TestContainers`

Para instalar:

```
poetry add --group dev testcontainers
```

Alterando a fixture para usar o TestContainer

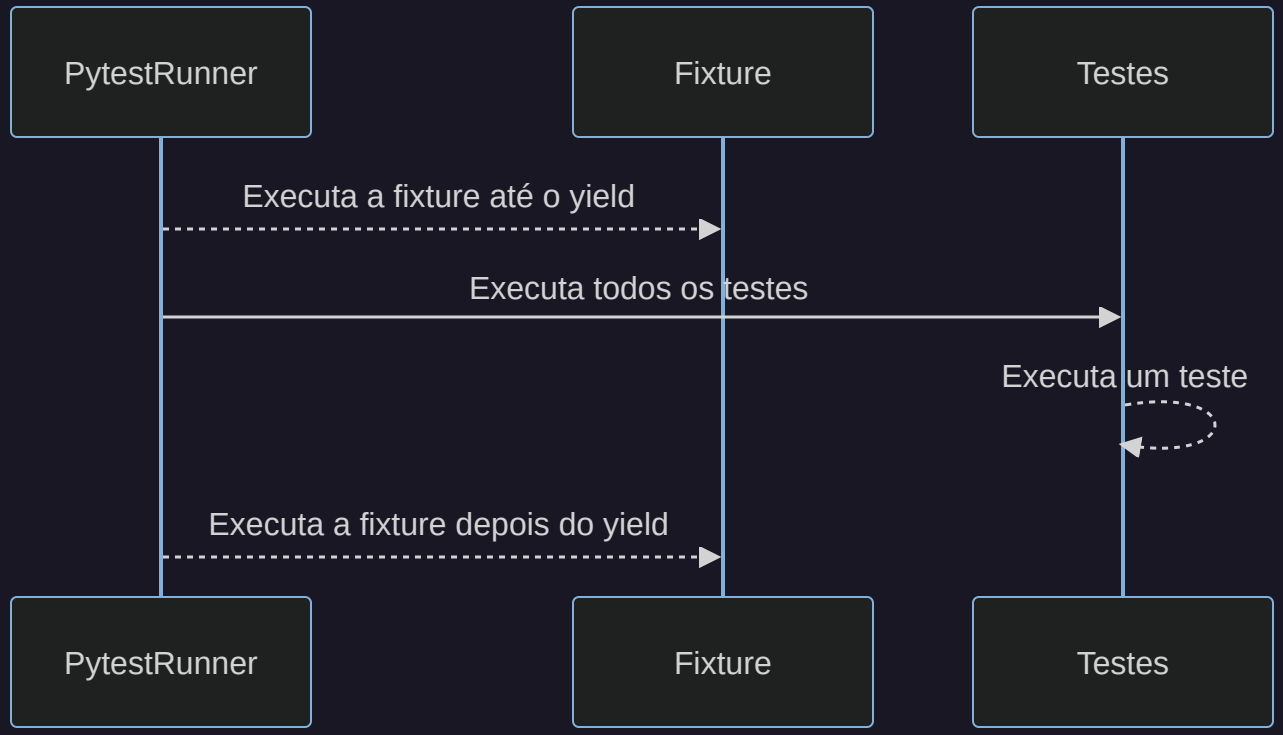
```
from testcontainers.postgres import PostgresContainer
# ...
@pytest.fixture
def session():
    with PostgresContainer('postgres:16', driver='psycopg') as postgres:
        engine = create_engine(postgres.get_connection_url())
        table_registry.metadata.create_all(engine)

        with Session(engine) as session:
            yield session
            session.rollback()

        table_registry.metadata.drop_all(engine)
```

Assim os testes podem iniciar um container novo a cada vez que a fixture for chamada

O escopo da fixture



Diferentes escopos

- `function`: executada em todas as funções de teste;
- `class`: executada uma vez por classe de teste;
- `module`: executada uma vez por módulo;
- `package`: executada uma vez por pacote;
- `session`: executava uma vez por execução dos testes;

Para resolver o problema com a lentidão dos testes, iremos criar uma fixture para iniciar o container do banco de dados com o escopo `session`.

Criando uma nova fixture

Para criar um única imagem docker, podemos iniciar ela de forma isolada na aplicação

```
@pytest.fixture(scope='session')
def engine():
    with PostgresContainer('postgres:16', driver='psycopg') as postgres:

        _engine = create_engine(postgres.get_connection_url())

        with _engine.begin():
            yield _engine
```

Alterando a fixture de session

```python
@pytest.fixture
def session(engine):
    table_registry.metadata.create_all(engine)

    with Session(engine) as session:
        yield session
        session.rollback()

    table_registry.metadata.drop_all(engine)
```

Agora a session faz tudo que precisa fazer, mas sem a responsabilidade do iniciar a conexão

Parte 2

> Criando a imagem do nosso projeto

Criando a imagem do nosso projeto

```
FROM python:3.13-slim
ENV POETRY_VIRTUALENVS_CREATE=false

WORKDIR app/
COPY . .

RUN pip install poetry

RUN poetry config installer.max-workers 10
RUN poetry install --no-interaction --no-ansi

EXPOSE 8000
CMD poetry run uvicorn --host 0.0.0.0 fast_zero.app:app
```

Rodando o código

```
# Criar a imagem
docker build -t "fast_zero" .
# Iniciar a imagem
docker run -it --name fastzeroapp -p 8000:8000 fast_zero:latest
```

> Podemos acessar para ver nossa aplicação http://127.0.0.1:8000/docs

Parte 3

> Docker compose

Docker compose

A ideia do docker compose é criar um único arquivo `yaml` que reúna todos os containers necessários para executar a aplicação.

Dessa forma podemos gerenciar todos os os containers com um único comando o `docker compose`.

O banco de dados

```
services:
  fastzero_database:
    image: postgres
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: app_user
      POSTGRES_DB: app_db
      POSTGRES_PASSWORD: app_password
    ports:
      - "5432:5432"

volumes:
  pgdata:
```

A aplicação

```
  fastzero_app:
    image: fastzero_app
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - fastzero_database
    environment:
      DATABASE_URL: postgresql+psycopg://app_user:app_password@fastzero_database:5432/app_db
```

Rodando tudo

```
docker compose up
```

Funciona?

Não ... migrações ...

Entrypoint

A ideia do entrypoint é alterar o comando `CMD` para executar um script bash.

```
#!/bin/sh

# Executa as migrações do banco de dados
poetry run alembic upgrade head

# Inicia a aplicação
poetry run uvicorn --host 0.0.0.0 --port 8000 fast_zero.app:app
```

Assim que o container for inciado, ele executará esse script.

Entrypoint no compose

```
  fastzero_app:
    image: fastzero_app
    entrypoint: ./entrypoint.sh
    build: .
```

Refazendo o container

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

Não esqueça de responder ao quiz

Quiz