Integrando Banco de Dados a API

> https://fastapidozero.dunossauro.com/04/

Objetivos dessa aula:

- Integrando SQLAlchemy à nossa aplicação FastAPI
- Utilizando a função Depends para gerenciar dependências
- Modificando endpoints para interagir com o banco de dados
- Testando os novos endpoints com Pytest e fixtures

Integrando SQLAlchemy à Nossa Aplicação FastAPI

Para aqueles que não estão familiarizados, o SQLAlchemy é uma biblioteca Python que facilita a interação com um banco de dados SQL. Ele faz isso oferecendo uma forma de trabalhar com bancos de dados que aproveita a facilidade e o poder do Python, ao mesmo tempo em que mantém a eficiência e a flexibilidade dos bancos de dados SQL.

Sessão

Uma peça chave do SQLAlchemy é o conceito de uma "sessão". Se você é novo no mundo dos bancos de dados, pode pensar na sessão como um carrinho de compras virtual: conforme você navega pelo site (ou, neste caso, conforme seu código executa), você pode adicionar ou remover itens desse carrinho. No entanto, nenhuma alteração é realmente feita até que você decida finalizar a compra. No contexto do SQLAlchemy, "finalizar a compra" é equivalente a fazer o commit das suas alterações.

1. **Mapa de Identidade: Imagine que você esteja comprando frutas em uma loja online. Cada fruta que você adiciona ao seu carrinho recebe um código de barras único, para que a loja saiba exatamente qual fruta você quer. O Mapa de Identidade no SQLAlchemy é esse sistema de código de barras: ele garante que cada objeto na sessão seja único e facilmente identificável.**
2. **Repositório: A sessão também atua como um repositório. Isso significa que ela é como um porteiro: ela controla todas as comunicações entre o seu código Python e o banco de dados. Todos os comandos que você deseja enviar para o banco de dados devem passar pela sessão.**
3. **Unidade de Trabalho: Finalmente, a sessão age como uma unidade de trabalho. Isso significa que ela mantém o controle de todas as alterações que você quer fazer no banco de dados. Se você adicionar uma fruta ao seu carrinho e depois mudar de ideia e remover, a sessão lembrará de ambas as ações. Então, quando você finalmente decidir finalizar a compra, ela enviará todas as suas alterações para o banco de dados de uma só vez.**

Criando a nossa sessão

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import Session

from fast_zero.settings import Settings

engine = create_engine(Settings().DATABASE_URL)


def get_session():
    with Session(engine) as session:
        yield session
```

Gerenciando Dependências com FastAPI

Assim como a sessão SQLAlchemy, que implementa vários padrões arquiteturais importantes, FastAPI também usa um conceito de padrão arquitetural chamado "Injeção de Dependência".

FastAPI fornece a função `Depends` para ajudar a declarar e gerenciar essas dependências. É uma maneira declarativa de dizer ao FastAPI: "Antes de executar esta função, execute primeiro essa outra função e passe-me o resultado". Isso é especialmente útil quando temos operações que precisam ser realizadas antes de cada request, como abrir uma sessão de banco de dados.

Implementando o banco nos endpoints

```python
from fastapi import Depends, FastAPI, HTTPException
from sqlalchemy import select
from sqlalchemy.orm import Session

# ...

@app.post('/users/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema, session: Session = Depends(get_session)):
    db_user = session.scalar(
        select(User).where(User.username == user.username)
    )
    if db_user:
        raise HTTPException(
            status_code=400, detail='Username already registered'
        )
    # ...
```

Criando uma estrutura para usar a sessão de testes

```
@pytest.fixture
def client(session):
    def get_session_override():
        return session

    with TestClient(app) as client:
        app.dependency_overrides[get_session] = get_session_override
        yield client

    app.dependency_overrides.clear()
```

Alterando nosso teste

```
def test_create_user(client):
    response = client.post(
        '/users',
        json={
            'username': 'alice',
            'email': 'alice@example.com',
            'password': 'secret',
        },
    )
    assert response.status_code == 201
    assert response.json() == {
        'username': 'alice',
        'email': 'alice@example.com',
        'id': 1,
    }
```

Erros!

A fixture precisa de algumas pequenas adaptações para rodar em threads diferentes:

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

    Session = sessionmaker(bind=engine)

    yield Session()

    Base.metadata.drop_all(engine)
```

Implementação dos outros endpoints

...

Commit!

```
git add .
git commit -m "Atualizando endpoints para usar o banco de dados real"
git push
```