Integrando Banco de Dados a API

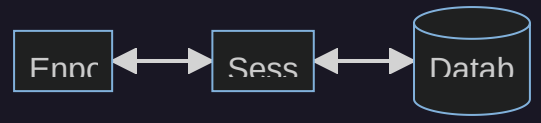
> https://fastapidozero.dunossauro.com/04/

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

A peça principal da nossa integração é a sessão do ORM. Ela precisa ser visível aos endpoints para que eles possam se comunicar com o banco.

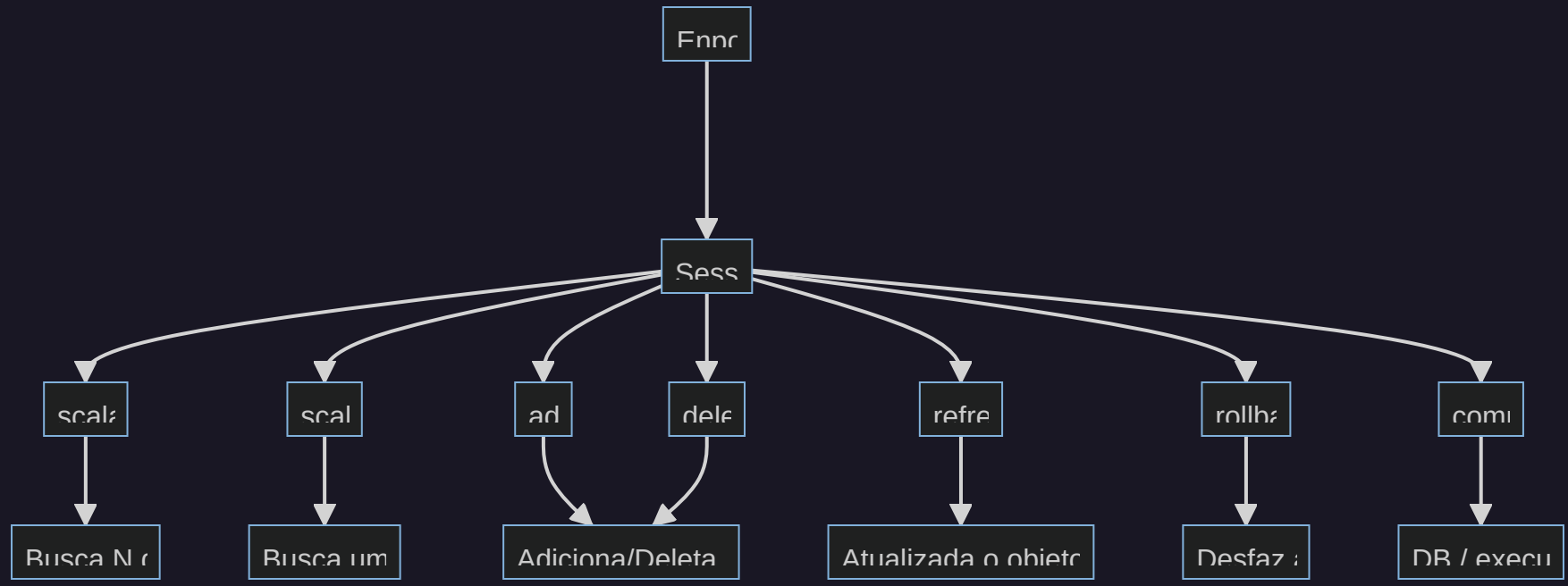


Sessão

No sentido mais geral, o `Session` estabelece todas as conversas com o banco de dados e representa uma “zona de retenção” para todos os objetos que você carregou ou associou a ele durante sua vida útil. Ele fornece o interface onde são feitas SELECT e outras consultas que retornarão e modificarão Objetos mapeados por ORM. Os próprios objetos ORM são mantidos dentro do Session, dentro de uma estrutura chamada mapa de identidade - um conjunto de dados estrutura que mantém cópias únicas de cada objeto, onde “único” significa “apenas um objeto com uma chave primária específica”.

1. **Repositório: A sessão atua como um repositório. A ideia de um repositório é abstrair qualquer interação envolvendo persistência de dados.**
2. **Unidade de Trabalho: Quando a sessão é aberta, todos os dados inseridos, modificados ou deletados não são feitos de imediato no banco de dados. Fazemos todas as modificações que queremos e executamos uma única ação.**
3. **Mapeamento de Identidade: É criado um cache para as entidades que já estão carregadas na sessão para evitar conexões desnecessárias.**

De uma forma visual



O básico sobre uma sessão

```python
from fast_zero.settings import Settings
from sqlalchemy import create_engine
from sqlalchemy.orm import Session

# Cria o pool de conexões
engine = create_engine(Settings().DATABASE_URL)

session = Session(engine)  # Cria a sessão

session.add(obj)      # Adiciona no banco
session.delete(obj)   # Remove do banco
session.refresh(obj)  # Atualiza o objeto com a sessão

session.scalars(query)  # Lista N objetos
session.scalar(query)   # Lista 1 objeto

session.commit()    # Executa as UTs no banco
session.rollback()  # Desfaz as UTs

session.begin()  # inicia a sessão
session.close()  # Fecha a sessão
```

Entendendo o enpoint de cadastro

Precisamos executar algumas operações para efetuar um cadastro:

1. O `email` não pode existir na base de dados
2. Se existir, devemos dizer que já está cadastrado com um erro
3. Caso não exista, deve ser inserido na base de dados

Abrindo mais as operações!

Precisamos executar algumas operações para efetuar um cadastro:

1. O `email` não pode existir na base de dados
    - Fazer uma busca procurando o `email` fornecido
        - `selecionar` na tabela de `Users` por email
        - Fazer isso de forma escalar e buscando por 1
2. Se existir, devemos dizer que já está cadastrado com um erro
    - Retornar `HTTPException`
3. Caso não exista, deve ser inserido na base de dados
    - Pedir para adicionar na sessão (`add`)
    - Fazer a persistência desse dado (`commit`)

Vamos fazer isso parte por parte!!

O `email` não pode existir na base de dados

```python
from sqlalchemy import create_engine, select
from sqlalchemy.orm import Session
from fast_zero.models import User
from fast_zero.settings import Settings
# ...

@app.post('/users/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema):
    engine = create_engine(Settings().DATABASE_URL)

    session = Session(engine)

    db_user = session.scalar(
        select(User).where(User.email == user.email)
    )

    if db_user: return 'ERRROOOO'
```

Se existir, devemos dizer que já está cadastrado com um erro

```python
@app.post('/users/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema):
    # ...

    raise HTTPException(
            status_code=400, detail='Username already registered'
    )
```

Caso não exista, deve ser inserido na base de dados

```python
@app.post('/users/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema):
    # ...

    db_user = User(
        username=user.username, password=user.password, email=user.email
    )
    session.add(db_user)
    session.commit()
    session.refresh(db_user)

    return db_user
```

> **Não esquecer de testar no swagger e mostar o banco!**

Não se repita (DRY)

> **Não acople e TESTE!**

Reutilizando a sessão

Uma das formas de reutilizar, seria cria uma função para obtermos a sessão

```python
# fast_zero/database.py
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
```

Usando a função!

Com isso, podemos somente chamar a nossa função e obter a nossa sessão. Evitando a repetição do código da sessão em todos os endpoints:

```python
from fast_zero.database import get_session
# ...

@app.post('/users/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema):
    session = get_session()

    db_user = session.scalar(
        select(User).where(User.email == user.email)
    )
    # ...
```

Acoplamento

Embora esteja bom, não tenhamos muita coisa que fuja da nossa lógica, somente a invocação de `get_session`. A chamada está acoplada. Isso traz dois problemas:

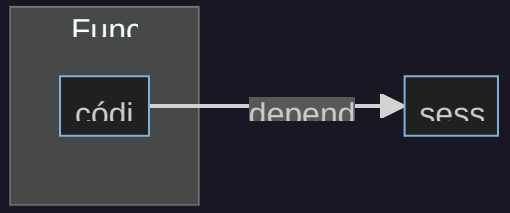
1. **Encapsulamento**: é complicado de escrever testes!
2. **Dependência**: o enpoint tem que conhecer a chamada da sessão

Mas, nem tudo está perdido!

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.




```python
def endpoint(
    user: UserSchema,
    session = Depends(get_session)
):

    session...
```

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

```
# tests/conftest.py
@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
```