Integrando banco de dados à API

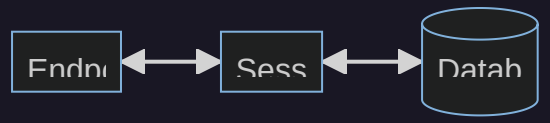
> https://fastapidozero.dunossauro.com/4.0/05/

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

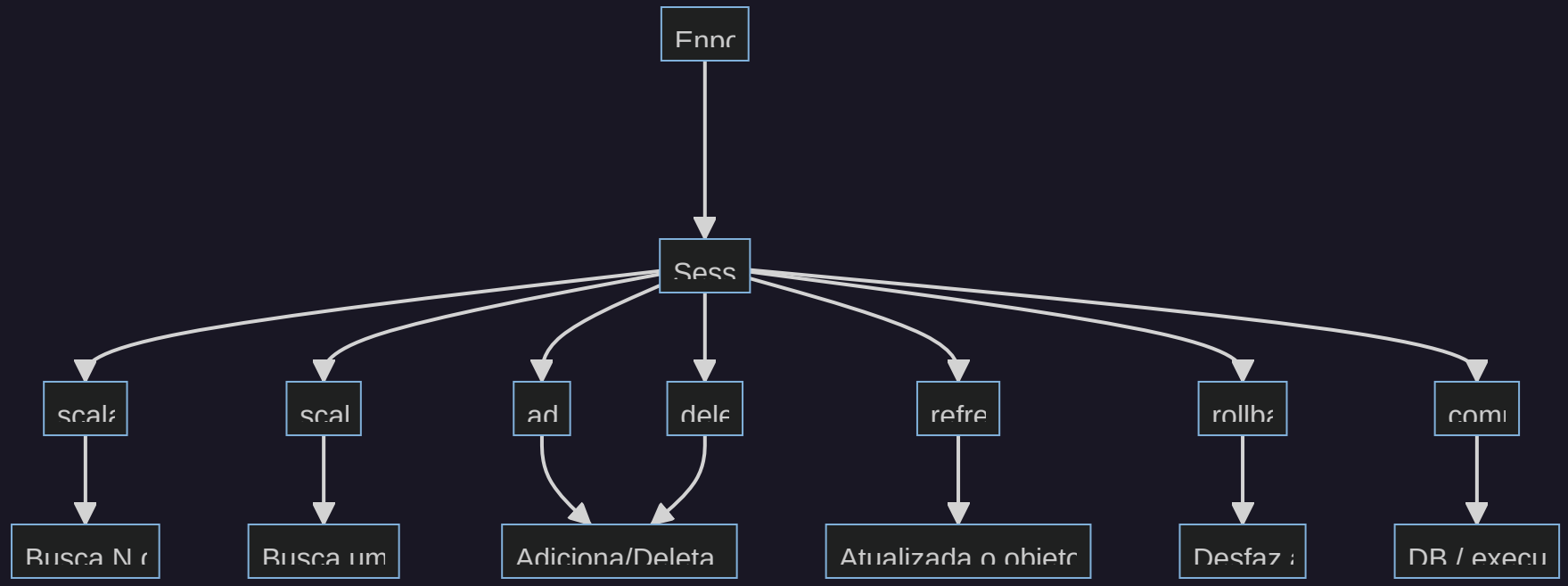
A peça principal da nossa integração é a **sessão** do ORM. Ela precisa ser visível aos endpoints para que eles possam se comunicar com o banco.



Padrões da sessão

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
```

Entendendo o enpoint de cadastro

Precisamos executar algumas operações para efetuar um cadastro:

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

Abrindo mais as operações!

Precisamos executar algumas operações para efetuar um cadastro:

1. Os dados unique não podem ser "readicionados"
    - Checar se username e email já não existem
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!!

Checando valores únicos

```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=HTTPStatus.CREATED)
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) | (User.username == user.username)
        )
    )

    if db_user: return 'ERRROOOO'
```

Caso exista

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

    if db_user:
        if db_user.username == user.username:
            raise HTTPException(
                status_code=HTTPStatus.CONFLICT,
                detail='Username already exists',
            )
        elif db_user.email == user.email:
            raise HTTPException(
                status_code=HTTPStatus.CONFLICT,
                detail='Email already exists',
            )
```

Caso não exista, deve ser inserido na base de dados

```python
@app.post('/users/', response_model=UserPublic, status_code=HTTPStatus.CREATED)
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!

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

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

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

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

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

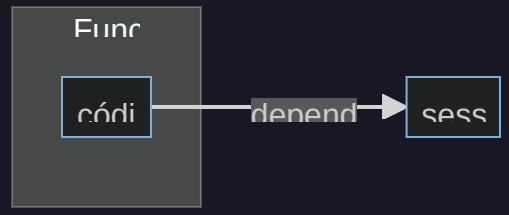
Mas, nem tudo está perdido!

Gerenciando Dependências com FastAPI

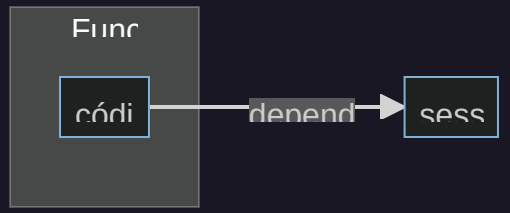
Assim como a sessão SQLAlchemy, que implementa vários padrões arquiteturais importantes, FastAPI também usa um conceito de padrão chamado "Injeção de Dependência". Por meio do objeto `Depends`.

É uma maneira declarativa de dizer ao FastAPI:

"Antes de executar esta função, execute primeiro essa outra função e passe o resultado para o parâmetro".



No código




```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=HTTPStatus.CREATED)
def create_user(user: UserSchema, session: Session = Depends(get_session)):
    db_user = session.scalar(
        select(User).where(
            (User.username == user.username) | (User.email == user.email)
        )
    )

    if db_user:
        # ...
```

Criando uma estrutura para usar a sessão de testes

```python
# 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 == HTTPStatus.CREATED
    assert response.json() == {
        'username': 'alice',
        'email': 'alice@example.com',
        'id': 1,
    }
```

> **Executar esse teste!**

Erros!

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

    table_registry.metadata.create_all(engine)

        with Session(engine) as session:
        yield session

    table_registry.metadata.drop_all(engine)
```

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

Endpoint de /GET

```python
@app.get('/users/', response_model=UserList)
def read_users(
    skip: int = 0, limit: int = 100, session: Session = Depends(get_session)
):
    users = session.scalars(select(User).offset(skip).limit(limit)).all()
    return {'users': users}
```

- `skip` e `limit`: são parâmetros de query
- Entrar no swagger: http://127.0.0.1:8000/docs

Testando o /GET

```
def test_read_users(client):
    response = client.get('/users')
    assert response.status_code == HTTPStatus.OK
    assert response.json() == {'users': []}
```

Uma nova fixture

```python
from fast_zero.models import User, table_registry

# ...

@pytest.fixture
def user(session):
    user = User(username='Teste', email='teste@test.com', password='testtest')
    session.add(user)
    session.commit()
    session.refresh(user)

    return user
```

Um novo teste para /GET

```python
from fast_zero.schemas import UserPublic
# ...

def test_read_users_with_users(client, user):
    user_schema = UserPublic.model_validate(user).model_dump()
    response = client.get('/users/')
    assert response.json() == {'users': [user_schema]}
```

`Schema.model_validate(obj)` faz a conversão de um objeto qualquer para um modelo do pydantic

Conversão do SQLA com Pydantic

```python
from pydantic import BaseModel, ConfigDict, EmailStr
# ...
class UserPublic(BaseModel):
    id: int
    username: str
    email: EmailStr
    model_config = ConfigDict(from_attributes=True)
```

`model_config` recebe uma configuração adicional com `ConfigDict`. Dizemos para tentar encontrar os atributos de `UserPublic` no objeto passado em `model_validate`.

Endpoints /PUT e /DELETE

A primeira coisa que esses endpoints devem fazer é verificar se o registro existe:

```
def put_ou_delete():
    db_user = session.scalar(select(User).where(User.id == user_id))

    if not db_user:
        raise HTTPException(
            status_code=HTTPStatus.NOT_FOUND, detail='User not found'
        )
```

Enpoint de /PUT

```python
@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(
    user_id: int, user: UserSchema, session: Session = Depends(get_session)
):
    # Validação

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

    return db_user
```

Enpoint de /DELETE

```python
@app.delete('/users/{user_id}', response_model=Message)
def delete_user(user_id: int, session: Session = Depends(get_session)):
    # Validação
    session.delete(db_user)
    session.commit()

    return {'message': 'User deleted'}
```

Teste para o /PUT

```
def test_update_user(client, user):
    response = client.put(
        '/users/1',
        json={
            'username': 'bob',
            'email': 'bob@example.com',
            'password': 'mynewpassword',
        },
    )
    assert response.status_code == HTTPStatus.OK
    assert response.json() == {
        'username': 'bob',
        'email': 'bob@example.com',
        'id': 1,
    }
```

Teste para o /DELETE

```
def test_delete_user(client, user):
    response = client.delete('/users/1')
    assert response.status_code == HTTPStatus.OK
    assert response.json() == {'message': 'User deleted'}
```

Um caso não pensado

E se os campos marcados como `unique=True` forem sobrescritos em um update?

Por exemplo:

- `User(username='fausto', ...)` entra na base
- `User(username='faustino', ...)` entra na base
- `User(username='faustino', ...)` que mudar seu username para `fausto`

Isso vai dar um erro de `integridade` no banco de dados!

Vamos testar esse cenário?

```python
def test_update_integrity_error(client, user):
    # Inserindo fausto
    client.post(
        '/users',
        json={
            'username': 'fausto',
            'email': 'fausto@example.com',
            'password': 'secret',
        },
    )

    # Alterando o user das fixture para fausto
    response_update = client.put(
        f'/users/{user.id}',
        json={
            'username': 'fausto',
            'email': 'bob@example.com',
            'password': 'mynewpassword',
        },
    )
```

Executando os testes

```
FAILED tests/test_app.py::test_update_integrity_error - sqlalchemy.exc.IntegrityError:
(sqlite3.IntegrityError) UNIQUE constraint failed: users.username
[SQL: UPDATE users SET username=?, password=?, email=? WHERE users.id = ?]
[parameters: ('fausto', 'mynewpassword', 'bob@example.com', 1)]
(Background on this error at: https://sqlalche.me/e/20/gkpj)
!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!
```

O erro le disse que temos um problema de integridade: falha na restrição `UNIQUE`: users.username.

Isso acontece, pois temos a restrição `UNIQUE` no campo username da tabela users. Quando adicionamos o mesmo nome a um registro que já existia, causamos um erro de integridade.

Resolvendo o problema no endpoint

```python
from sqlalchemy.exc import IntegrityError
# ...
@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(...):
    # Validação!
    try: # Tente inserir
        db_user.username = user.username
        db_user.password = user.password
        db_user.email = user.email
        session.commit()
        session.refresh(db_user)

        return db_user

    except IntegrityError: # Se esse erro, conflito!
        raise HTTPException(
            status_code=HTTPStatus.CONFLICT,
            detail='Username or Email already exists'
        )
```

Ajustando o teste dos faustos

Vamos confirmar se temos mesmo um conflito:

```
def test_update_integrity_error(client, user):
    # ...

    assert response_update.status_code == HTTPStatus.CONFLICT
    assert response_update.json() == {
        'detail': 'Username or Email already exists'
    }
```

Agora tudo foi coberto com sucesso :)

Exercícios:

1. Escrever um teste para o endpoint de POST (create_user) que contemple o cenário onde o username já foi registrado. Validando o erro `409`
2. Escrever um teste para o endpoint de POST (create_user) que contemple o cenário onde o e-mail já foi registrado. Validando o erro `409`
3. Atualizar os testes criados nos exercícios 1 e 2 da aula 03 para suportarem o banco de dados
4. Implementar o banco de dados para o endpoint de listagem por id, criado no exercício 3 da aula 03

Quiz

> https://fastapidozero.dunossauro.com/4.0/quizes/aula_05/

Commit!

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