Configurando o Banco de Dados e gerenciando Migrações com Alembic

> https://fastapidozero.dunossauro.com/4.0/04/

Objetivos dessa aula:

- Introdução ao SQLAlchemy e Alembic
- Instalando SQLAlchemy e Alembic
- Configurando e criando o banco de dados
- Criando e localizando tabelas utilizando SQLAlchemy
- Testando a criação de tabelas
- Eventos do SQLAlchemy
- Gerenciando migrações do banco de dados com Alembic

Uma introdução ao SQLAlchemy

SQLalchemy

O SQLAlchemy é um ORM. Ele permite que você trabalhe com bancos de dados SQL de maneira mais natural aos programadores Python. Em vez de escrever consultas SQL cruas, você pode usar métodos e atributos Python para manipular seus registros de banco de dados.

ORM significa Mapeamento Objeto-Relacional. É uma técnica de programação que vincula (ou mapeia) objetos a registros de banco de dados. Em outras palavras, um ORM permite que você interaja com seu banco de dados, como se você estivesse trabalhando com objetos Python.

Mas por que usaríamos um ORM?

- Abstração de banco de dados: ORMs permitem que você mude de um tipo de banco de dados para outro com poucas alterações no código.
- Segurança: ORMs geralmente lidam com escapar de consultas e prevenir injeções SQL, um tipo comum de vulnerabilidade de segurança.
- Eficiência no desenvolvimento: ORMs podem gerar automaticamente esquemas, realizar migrações e outras tarefas que seriam demoradas para fazer manualmente.

Instalação do SQLalchemy

```
poetry add sqlalchemy
```

Definindo nosso modelo de "user" com SQLalchemy

no arquivo `fast_zero/models.py` vamos criar

```python
from datetime import datetime
from sqlalchemy.orm import Mapped, registry

table_registry = registry()


@table_registry.mapped_as_dataclass
class User:
    __tablename__ = 'users'

    id: Mapped[int]
    username: Mapped[str]
    password: Mapped[str]
    email: Mapped[str]
    created_at: Mapped[datetime]
```

Restrições em colunas

```python
@table_registry.mapped_as_dataclass
class User:
    __tablename__ = 'users'

    id: Mapped[int] = mapped_column(init=False, primary_key=True)
    username: Mapped[str] = mapped_column(unique=True)
    password: Mapped[str]
    email: Mapped[str] = mapped_column(unique=True)
    created_at: Mapped[datetime] = mapped_column(
        init=False, server_default=func.now()
    )
```

Criando um teste para esse modelo

Vamos criar um arquivo novo para testes de banco de dados: `tests/test_db.py`

```python
from fast_zero.models import User

def test_create_user():
    user = User(username='test', email='test@test.com', password='secret')

    assert user.password == 'secret'
```

> Aqui temos uma bomba!

O que esse teste testa?

Aparentemente ele testa se uma classe pode ser instanciada **ou seja, NADA**.

Precisamos garantir algumas coisas:

1. Se é possível criar essa tabela
    - `Metadata`!
2. Se é possível buscar um `User` usando ela como base
    - `Session`!

Só que para isso precisamos conhecer alguns outros componentes importantes.

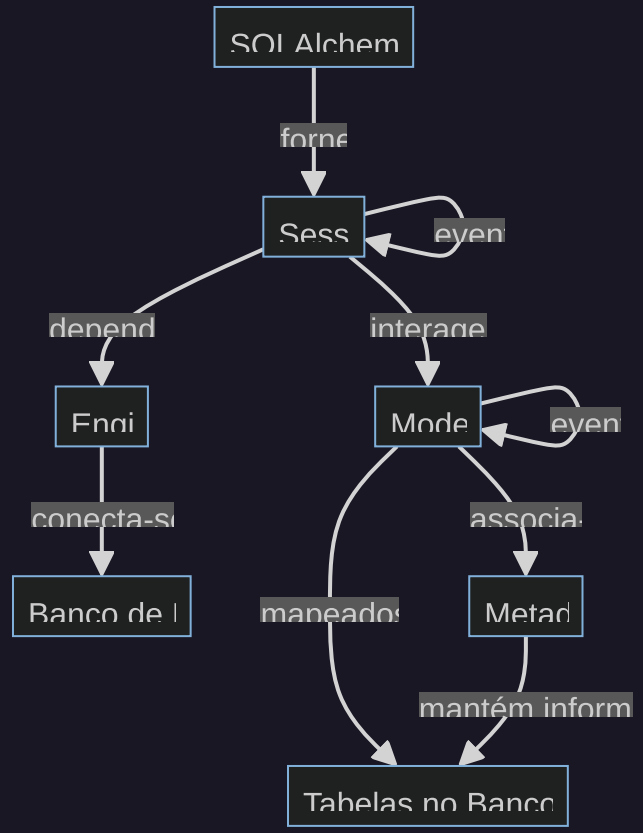
Outros componentes importantes

Engine

A 'Engine' do SQLAlchemy é o ponto de contato com o banco de dados, estabelecendo e gerenciando as conexões. Ela é instanciada através da função `create_engine()`, que recebe as credenciais do banco de dados, o endereço de conexão (URI) e configura o pool de conexões.

Session

Quanto à persistência de dados e consultas ao banco de dados utilizando o ORM, a Session é a principal interface. Ela atua como um intermediário entre o aplicativo Python e o banco de dados, mediada pela Engine. A Session é encarregada de todas as transações, fornecendo uma API para conduzi-las.


SQLAlchem
forne
Sess
even
depend
interage
Engi
Mode
even
conecta-s
associa
Banco de
maneados
Metad
mantém inform
Tabelas no Banco

Escrevendo testes para esse modelo

A primeira coisa que temos que montar é uma fixture da sessão do banco em `tests/conftest.py`

```python
import pytest
from sqlalchemy import create_engine, select
from sqlalchemy.orm import sessionmaker

from fast_zero.models import table_registry


@pytest.fixture
def session():
    engine = create_engine('sqlite:///:memory:')
    table_registry.metadata.create_all(engine)

    with Session(engine) as session:
        yield session

    table_registry.metadata.drop_all(engine)
```

Eu sei, esse código é um pouco complexo de mais [0]

1. `create_engine('sqlite:///:memory:')`: cria um mecanismo de banco de dados SQLite em memória usando SQLAlchemy. Este mecanismo será usado para criar uma sessão de banco de dados para nossos testes.
2. `table_registry.metadata.create_all(engine)`: cria todas as tabelas no banco de dados de teste antes de cada teste que usa a fixture `session`.
3. `with Session(engine) as session`: cria uma sessão `Session` para que os testes possam se comunicar com o banco de dadosvia `engine`.

Eu sei, esse código é um pouco complexo de mais [1]

4. `yield session`: fornece uma instância de Session que será injetada em cada teste que solicita a fixture `session`. Essa sessão será usada para interagir com o banco de dados de teste.

5. `table_registry.metadata.drop_all(engine)`: após cada teste que usa a fixture `session`, todas as tabelas do banco de dados de teste são eliminadas, garantindo que cada teste seja executado contra um banco de dados limpo.

Agora nosso teste

```python
from sqlalchemy import select
from fast_zero.models import User


def test_create_user(session):
    new_user = User(username='alice', password='secret', email='teste@test')
    session.add(new_user)
    session.commit()

    user = session.scalar(select(User).where(User.username == 'alice'))

    assert user.username == 'alice'
```

> task test

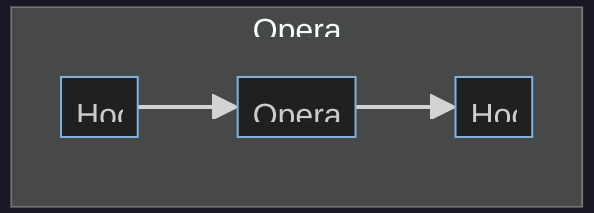
Um sucesso, mas nem tanto

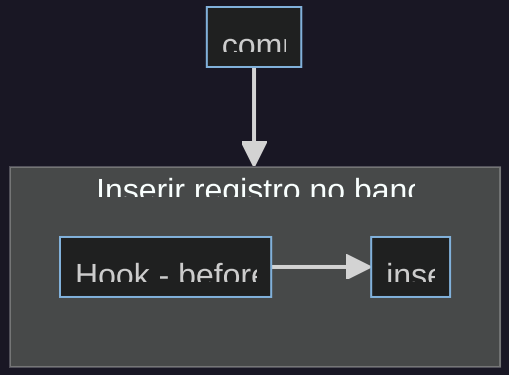
Temos um problema nesse teste que pode tornar ele complicado de validar. Validamos somente o nome `alice`, mas não validamos o objeto todo. Isso é um tanto quanto complicado. Pois para validar o objeto inteiro, precisamos saber a que horas ele foi criado, por conta do campo `init=False`.

Ele inviabiliza o envido de um dado determinístico ao objeto.

Eventos do ORM

Para atuar em cenários assim, podemos "roubar nos testes" usando eventos do SQLAlchemy.





A ideia por trás dos eventos é fazer alguma operação antes ou depois de alguma operação.

Exemplo de evento

Chamamos o objeto `event` do SQLalchemy para "ouvir" uma operação:

```
from sqlalchemy import event


def hook(mapper, connection, target):
    ...


event.listen(User, 'before_insert', hook)
```

Nesse caso, estamos ouvindo `before_insert`. O que significa que ele executará a função `hook` antes de inserir no banco de fato.

Nosso evento para manipular o campo de data

```python
from contextlib import contextmanager
from datetime import datetime
from sqlalchemy import create_engine, event

# ...

@contextmanager
def _mock_db_time(*, model, time=datetime(2024, 1, 1)):

    def fake_time_hook(mapper, connection, target):
        if hasattr(target, 'created_at'):
            target.created_at = time

    event.listen(model, 'before_insert', fake_time_hook)

    yield time

    event.remove(model, 'before_insert', fake_time_hook)
```

O que de fato esse evento vai fazer?

```
@contextmanager
def _mock_db_time(*, model, time=datetime(2024, 1, 1)):

    def fake_time_hook(mapper, connection, target):
        if hasattr(target, 'created_at'):
            target.created_at = time

    event.listen(model, 'before_insert', fake_time_hook)

    yield time

    event.remove(model, 'before_insert', fake_time_hook)
```

Antes de executar o insert a função `fake_time_hook` vai alterar o `created_at` para o valor default do parâmetro `time`. Fazendo que o ele não use o valor padrão do datetime do db.

O `contextmanager` faz com que a função possa ser usada com o bloco `with`.

Transformando em uma fixture

Agora que temos a função gerenciadora de contexto, para evitar o sistema de importação durante os testes, podemos criar uma fixture para ele.

De forma bem simples, somente retornando a função `_mock_db_time`:

```
@pytest.fixture
def mock_db_time():
    return _mock_db_time
```

Dessa forma podemos fazer a chamada direta no teste.

Fazendo o teste do objeto completo

```python
from dataclasses import asdict
from sqlalchemy import select
from fast_zero.models import User

def test_create_user(session, mock_db_time):
    with mock_db_time(model=User) as time:
        new_user = User(
            username='alice', password='secret', email='teste@test'
        )
        session.add(new_user)
        session.commit()

        user = session.scalar(select(User).where(User.username == 'alice'))

    assert asdict(user) == {
        'id': 1,
        'username': 'alice',
        'password': 'secret',
        'email': 'teste@test',
        'created_at': time,
    }
```

A validação completa

Dessa forma todos os campos, até os que são manipulados diretamente pelo ORM podem ser testados.

```
    assert asdict(user) ==
        'id': 1,
        'username': 'alice',
        'password': 'secret',
        'email': 'teste@test',
        'created_at': time,
    }
```

Configurações de ambiente e as 12 fatores

Uma boa prática no desenvolvimento de aplicações é separar as configurações do código.

Configurações, como credenciais de banco de dados, são propensas a mudanças entre ambientes diferentes (como desenvolvimento, teste e produção).

Misturá-las com o código pode tornar o processo de mudança entre esses ambientes complicado e propenso a erros.

```
poetry add pydantic-settings
```

Configuração do ambiente do banco de dados

```python
#fast_zero/settings.py
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file='.env', env_file_encoding='utf-8'
    )

    DATABASE_URL: str
```

`.env`

Esse configuração permite que usemos arquivos `.env` para não inserir dados do banco no código fonte

```
DATABASE_URL="sqlite:///database.db"
```

Não podemos esquecer de adicionar essa base de dados no `.gitignore`

```
echo 'database.db' >> .gitignore
```

Migrações

Antes de avançarmos, é importante entender o que são migrações de banco de dados e por que são úteis.

- Banco de dados evolutivo
- O banco acompanha as alterações do código
- Reverter alterações no schema do banco

Instalação e configuração do alembic

```
poetry add alembic

alembic init migrations
```

Isso criará uma estrutura de pastas nova

```
.
├── .env
├── alembic.ini     <-
├── fast_zero
│   ├── __init__.py
│   ├── app.py
│   ├── models.py
│   └── schemas.py
├── migrations      <-
│   ├── env.py
│   ├── README
│   ├── script.py.mako
│   └── versions
├── poetry.lock
├── pyproject.toml
├── README.md
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── test_app.py
    └── test_db.py
```

Configurando a migração automática

Vamos fazer algumas alterações no arquivo `migrations/env.py` para que nossa configurações de banco de dados sejam passadas ao alembic:

1. Importar as `Settings` do nosso arquivo `settings.py` e a `table_registry` dos nossos modelos.
2. Configurar a URL do SQLAlchemy para ser a mesma que definimos em `Settings`.
3. Verificar a existência do arquivo de configuração do Alembic e, se presente, lê-lo.
4. Definir os metadados de destino como `table_registry.metadata`, que é o que o Alembic utilizará para gerar automaticamente as migrações.

```python
from alembic import context
from fast_zero.settings import Settings
from fast_zero.models import table_registry

config = context.config
config.set_main_option('sqlalchemy.url', Settings().DATABASE_URL)

if config.config_file_name is not None:
    fileConfig(config.config_file_name)

target_metadata = table_registry.metadata
```

Gerando a migração

```
alembic revision --autogenerate -m "create users table"
```

Aplicando a migração

```
alembic upgrade head
```

Dando uma olhada no banco de dados

Pra isso poderíamos usar uma ferramenta gráfica ou usando a CLI do sqlite:

```
python -m sqlite3 database.db

# Abrirá o shell do sqlite3
sqlite>
```

```
select * from alembic_version;
select * from users;
```

Exercícios

1. Fazer uma alteração no modelo (tabela `User`) e adicionar um campo chamado `updated_at`:
    - Esse campo deve ser mapeado para o tipo `datetime`
    - Esse campo não deve ser inicializado por padrão `init=False`
    - O valor padrão deve ser `now`
    - Toda vez que a tabela for atualizada esse campo deve ser atualizado:

```
mapped_column(onupdate=func.now())
```

Exercícios + Quiz

2. Altere o evento de testes (`mock_db_time`) para ser contemplado no mock o campo `updated_at` na validação do teste.
3. Criar uma nova migração autogerada com alembic
4. Aplicar essa migração ao banco de dados

> Obviamente, não esqueça de responder ao quiz da aula

commit

```
git add .
git commit -m "Adicionada a primeira migração com Alembic. Criada tabela de usuários."
git push
```