# Configurando o Banco de Dados e Gerenciando Migrações com Alembic

> https://fastapidozero.dunossauro.com/03/

# 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
- 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 sqlalchemy.orm import DeclarativeBase
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import mapped_column


class Base(DeclarativeBase):
    pass


class User(Base):
    __tablename__ = 'users'

    id: Mapped[int] = mapped_column(primary_key=True)
    username: Mapped[str]
    password: Mapped[str]
    email: Mapped[str]
```

## Criando um teste para esse modelo

```python
from fast_zero.models import User

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

    assert user.password == 'secrete'
```

> **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.**

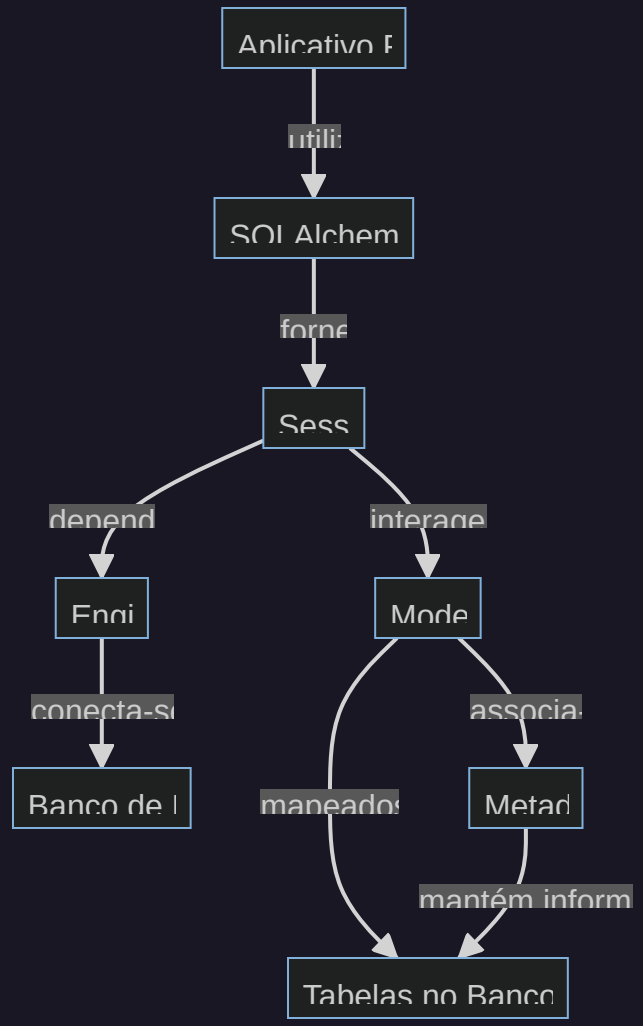
# Outros componentes importantes

## 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.


Aplicativo F
utili
SQLAlchem
forne
Sess
depend
interage
Engi
Mode
conecta-s
associa
Banco de I
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

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

from fast_zero.models import Base


@pytest.fixture
def session():
    engine = create_engine('sqlite:///:memory:')
    Session = sessionmaker(bind=engine)
    Base.metadata.create_all(engine)
    yield Session()
    Base.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. `Session = sessionmaker(bind=engine)`: cria uma fábrica de sessões para criar sessões de banco de dados para nossos testes.
3. `Base.metadata.create_all(engine)`: cria todas as tabelas no banco de dados de teste antes de cada teste que usa a fixture `session`.

## 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. `Base.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'
```

# 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

```
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.**

**As migrações são uma maneira de fazer alterações ou atualizações no banco de dados, como adicionar uma tabela ou uma coluna a uma tabela, ou alterar o tipo de dados de uma coluna. Elas são extremamente úteis, pois nos permitem manter o controle de todas as alterações feitas no esquema do banco de dados ao longo do tempo. Elas também nos permitem reverter para uma versão anterior do esquema do banco de dados, se necessário.**

# 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

**Com o Alembic devidamente instalado e iniciado, agora é o momento de gerar nossa primeira migração. Mas, antes disso, precisamos garantir que o Alembic consiga acessar nossas configurações e modelos corretamente. Para isso, vamos fazer algumas alterações no arquivo `migrations/env.py`.**

**Neste arquivo, precisamos:**

1. Importar as `Settings` do nosso arquivo `settings.py` e a `Base` 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 `Base.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 Base

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 = Base.metadata
```

# Gerando a migração

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

# Aplicando a migração

```
alembic upgrade head
```

# commit

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