
FIAP

Banco de Dados e Alembic

ORM

Quando falamos de manipular os dados dentro de um banco, é muito comum utilizarmos o padrão (caso prefira o termo em inglês, pattern) **ORM**.
ORM é a sigla para **Object-Relational Mapping** que traduzindo literalmente significa Mapeamento objetos-relacional. E o que isso significa em termos práticos?
Significa que no seu código você cria classes que são a representação de uma tabela no seu banco de dados e cada atributo da sua classe representa uma coluna na sua tabela, com seu nome e tipo definidos.
E o **ORM** vai fazer todo o trabalho de executar os comandos no banco de dados, sem você precisar se preocupar com isso. Ou seja, para criar uma tabela, inserir um registro, buscar registros, etc. é tudo feito pelo **ORM**

SQLAlchemy

No mundo Python, um dos ORMs mais famosos e com muita aderência pela comunidade é o SQLAlchemy. Ele tem um bom suporte a diversos tipos de banco de dados (PostgreSQL, MySQL, Oracle, Microsoft SQL) e tem muitas funcionalidades que o tornam tão relevante.

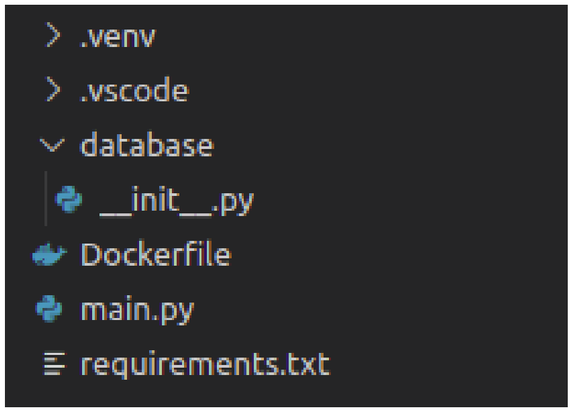
SQLAlchemy

Para começar a utilizá-lo, instale ele no seu ambiente:

```
pip install SQLAlchemy
```

SQLAlchemy

Uma vez instalado, agora vamos começar a criar nossa estrutura. Crie uma basta chamada `database`. Dentro dela, crie um arquivo `__init__.py`. Vai ficar com essa estrutura:



SQLAlchemy

Dentro do arquivo `__init__.py`, iremos colocar o seguinte código:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = 'sqlite://./fastapi.db'
engine = create_engine(SQLALCHEMY_DATABASE_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

SQLAlchemy

Explicando as linhas:

1-3 -> Faz os imports necessários para criar minha conexão com o banco de dados
5 -> Define a localização do meu banco de dados. Nesse caso, ele vai ser um banco de dados SQLite.
7-9 -> A engine nada mais é que o motor que vai fazer o trabalho de "traduzir" as instruções para o banco de dados.
10 -> Define uma sessão do banco de dados. Lembre-se que isso é apenas definição e não a inicialização de uma sessão
12 -> Define base que será utilizada nos meus modelos

SQLAlchemy

Agora, vamos definir os modelos. Crie um arquivo dentro da pasta `database` chamado `models.py`.

```
from sqlalchemy import Boolean, Column, Integer, String

from database import Base


class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)
```

SQLAlchemy

Primeiro definiremos nosso modelo `User`. Importamos os tipos de dados que utilizaremos nos modelos. A classe `Column` é uma representação de uma coluna na tabela.
Depois, importamos o `Base`, que já definimos que será a base dos nosso modelos.
Ao criar a classe `User`, note que nela temos a propriedade `__tablename__`, que é o nome da tabela que será utilizada no banco de dados. E cada atributo depois dele, defina qual será a coluna na tabela, com seu respectivo nome e tipo

SQLAlchemy

Agora criaremos uma nova tabela chamada `Item`, que segue os mesmos principios para criar a tabela `User`

```
class Item(Base):
    __tablename__ = 'item'

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String)
    description = Column(String)
```

Chaves estrangeiras

A ideia é que a tabela `User` tenha um relacionamento com a tabela `Item`, onde o usuário tenha seus itens. E como fazemos isso?
Para isso importaremos mais algumas classes e mudaremos nosso modelo `User` para ficar da seguinte forma:

```
from sqlalchemy import Boolean, Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

    items = relationship("Item", back_populates="owner")
```

Chaves estrangeiras

Uma vez que a propriedade `items` for acessada atráves de uma instância da classe `User`, implicitamente uma busca na tabela `Item` será realizada, filtrando pelo id do usuário. Apenas como demonstração, será feita uma query conforme a seguir:

```
select * from item where owner_id = user.id;
```

Chaves estrangeiras

Agora, na classe `Item` devemos fazer a referência a tabela de `User` para que tudo funcione conforme esperado.

```
class Item(Base):
    __tablename__ = 'item'

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, index=True)
    description = Column(String, index=True)
    owner_id = Column(Integer, ForeignKey("user.id"))

    owner = relationship("User", back_populates="items")
```

Chaves estrangeiras

A propriedade `owner_id` faz referência a tabela `User`. Note que usamos a síntaxe `tabela.campo`.
E assim que a propriedade `owner` for acessada, irá fazer implicitamente fazer uma busca na tabela `User` e trazer uma instância de `User`.
Apenas como demonstração, será feita uma query conforme a seguir:

```
select * from user where id = item.owner_id;
```

Alembic

Para ajudar no gerenciamento de versão do banco de dados, usamos uma biblioteca chamada Alembic. Com ele, você pode adicionar tabelas, criar colunas, migrar dados, etc. sem se preocupar com qual opção de banco de dados está sendo utilizada.
Para utilizá-la, basta instalar da seguinte forma:

```
pip install alembic
```

Alembic

Inicialmente, deve-se criar a configuração inicial. Para isso, basta rodar o comando:

```
alembic init <nome da pasta>
```

Geralmente, se cria uma pasta chamada `alembic` para deixar bem explicito que esta biblioteca é utilizada no projeto. Então o comando ficaria:

```
alembic init alembic
```

Alembic

Ao olhar a estrutura de pastas, teremos a seguinte estrutura:

```
alembic/
├── versions/
├── env.py
├── README
└── script.py.mako
```

Alembic

Dentro da pasta `alembic`, existe o arquivo `env.py`. Iremos fazer uma modificação nele para o alembic entender qual banco de dados e quais modelos ele deve gerenciar.

Procure pela linha que tem começa com o seguinte comentário

```
# add your model's Metadata object here
```

Então deixe o conteúdo da seguinte forma:

```
# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
from database import SQLALCHEMY_DATABASE_URL
config.set_main_option("sqlalchemy.url", SQLALCHEMY_DATABASE_URL)

from database.models import Base
target_metadata = [Base.metadata]
```

IMPORTANTE: Leve em consideração o que foi feito nos slides sobre utilizar o `FastAPI` e `ORM`

Alembic

Caso queira utilizar outros bancos de dados, como o **PostgreSQL**, devemos instalar sua biblioteca:
`pip install "psycopg[binary]"`

E então, nossa constante `SQLALCHEMY_DATABASE_URL` deve ficar da seguinte forma:

```
SQLALCHEMY_DATABASE_URL = "postgresql://{user}:{password}@{server}/{database}"
```

Onde devemos substituir o que está entre chaves para seus respectivos valores. Ou seja:
user -> Usuário que irá conectar no banco de dados
password -> Senha do usuário que irá conectar no banco de dados
server -> Servidor onde está hospedado o banco de dados
database -> Nome do banco de dados a ser utilizado

Alembic

Após isso, crie sua primeira migração. Migração é o termo utilizado para determinar quais alterações vão ocorrer no seu banco de dados na versão que você está criando. Nesse caso, a primeira migração vai ser a criação de toda a estrutura de dados atual.

Para cria-lá, rode o comando:

```
alembic revision –autogenerate --m "Minha estrutura inicial"
```

Alembic

Olhando novamente a estrutura de pastas, veremos que dentro da pasta `versions`, dentro do `alembic` será criado um arquivo como no exemplo:

```
alembic/
├── versions/
│   └── a9a2386fe91c_minha_estrutura_inicial.py
├── env.py
├── README
└── script.py.mako
```

Ele coloca um `hash` na frente do nome do arquivo para identificar unicamente cada alteração feita na estrutura do banco de dados

Alembic

Uma vez criada a primeira migração, para rodar no banco de dados as alterações dentro dela, execute:

```
alembic upgrade head
```

Também é possível rodar colocando o `hash` conforme citado anteriormente

```
alembic upgrade a9a2386fe91c
```

Alembic

Toda vez que precisar migrar algo no seu banco de dados, existem duas opções no comando `alembic`:

`alembic upgrade` -> Sempre atualiza o banco de dados para a versão informada.
`alembic downgrade` -> Regride a versão do banco de dados para a versão desejada

Para mais detalhes sobre esse gerenciamento, pode ser consultado na sua documentação

Alembic

Caso queira consultar todas as migrações numa ordem cronológica, podemos utilizar o seguinte comando:

`alembic history`

A saída será a seguinte:

```
<base> -> a9a2386fe91c (head), Minha estrutura inicial
```

Caso queira mais detalhes, só rodar o mesmo comando passando o argumento `--verbose`

`alembic history --verbose`

```
Rev: a9a2386fe91c (head)
Parent: <base>
Path: /mnt/6769bdb4-48cc-4fb6-af4f-4b745e2d4ae4/code/Python/mastering/fastapi/alembic/versions/a9a2386fe91c_minha_estrutura_inicial.py

    Minha estrutura inicial

    Revision ID: a9a2386fe91c
    Revises:
    Create Date: 2023-07-04 20:35:07.914665
```


FIAP