Programação de Sistemas para Internet

Prof. Romerito Campos

Plano de Aula

- Objetivo: *Configurar o acesso ao banco de dados utilizando SQLAlchemy*

Conteúdos

- Instalação e Configurações
- Modelos
- Engine
- Sessão

Instalação

Instalação

- O SQLAlchemy é uma caixa de ferramentas para SQL e um Mapeador Objeto-Relacional em python.
- É independente de framework.
- Portanto, pode ser usado isoladamente tanto scripts quanto em frameworks como Flask.
  - O flask possui uma extensão para SQLAlchemy
- Instalação:

```
pip install sqlalchemy
```

Configuração

Configuração

- A configuração do SQLAlchemy depende da criação de uma conexão com o banco de dados (por exemplo sqlite, MySQL e etc).
- Para tanto, o SQLAlchemy define uma função em seu **core** chamada `create_engine`

```
from sqlalchemy import create_engine
```

- Esta função recebe uma string que pode ser montada com base no banco de dados.

Configuração

- Nos exemplos deste material utilizei o SQLITE.
- Para criar uma conexão com o SQLITE e salvar os dados em arquivo basta:

```
from sqlalchemy import create_engine

engine = create_engine("sqlite:///database.db")
```

- A partir do momento que o objeto `engine` tem a conexão com o banco, poderemos criar as tabelas.

Configuração

- O objeto `engine` é o ponto inicial para uma aplicação com SQLAlchemy e também o meio pelo qual a robusta API do ORM pode acessar o banco.
- A engine é configurada com base em dialetos. O SQLITE é um dialeto. O MYSQL é outro. Neste link, há todos os dialetos.
- Na prática, usamos a função `create_engine` como uma fábrica de conexões.

Configuração

- Exemplo com MYSQL

```
# conexão com MYSQL
engine = create_engine("mysql://root:romerito@localhost/banco")
```

- Observe o formato da String que pode ser decomposta em:
  - Sistema de Gerenciamento do Banco: `mysql`
  - Usuário: `root` e senha: `romerito`
  - Endereço do banco: `localhost`
  - Nome do banco: `banco`

Executando SQL

Executanto SQL

- Através da `engine`, podemos obter um objeto que é capaz de executar instruções em SQL no banco.

```
# continuação do código anterior
connection = engine.connect()
```

- Com o objeto `connection`, podemos **aplicar instruções** SQL no banco.
- Obviamente, precisamos criar as tabelas e isso será feito com este objeto. Esta será a primeira instrução a ser aplicada.

Executando SQL

- O exemplo 01 mostra como usar a engine para conecar com o banco e aplicar SQL. O código abaixo mostra como criar uma tabela no Banco:

```
from sqlalchemy import create_engine, text
engine = create_engine('sqlite:///database.db')
connection = engine.connect() # uma das formas
sql = text("CREATE TABLE users IF NOT EXISTS(id INTEGER PRIMARY KEY)")
connection.execute(sql)
```

- O trecho acima é um resumo do que foi falado até aqui.

- Para executar instruções, primeiro devemos criar uma instrução. Por exemplo: `CREATE TABLE users`.
  - Para isso vamos usar uma funação do SQLAlchemy chamada `text`
  - O código abaixo prepara o SQL a ser executado.

```
from sqlalchemy import text
# código da engine e connection
sql = text("""CREATE TABLE users IF NOT EXISTS""")
```

- Com o objeto `connection`, podemos executar a função `execute(sql)` e ter a construção da tabela.

Executando SQL

- Da mesma forma que podemos usar `connection` para criar as tabelas, podemos também realizar outras operações como `INSERT`.
- No exemplo 01, temos a operação de inserção de dados.

```
# prepara a inserção
insert = text("""INSERT INTO users(nome) VALUES(:nome)""")
connection.execute(insert, {'nome':'jose'})
```

- O trecho acima está resumido. Perceba que `:nome` é *alias* para o verdadeiro valor. O dicionário na funação `execute` recebe o nome `josé` que será adicionado no lugar do *alias* .

Executando SQL

- Neste primeiro exemplo, utilizamos a `engine` que conecta com o banco de dados.
- Criamos um objeto `connection` que permite executar instruções SQL como criar tabelas e realizar inserções de dados
- Estas operações foram realizadas com SQL bruto (***raw SQL***)
- Veremos adiante que há recursos mais interessantes para interagir com banco.

Sessão

Sessão

- O exemplo 02 modifica o exemplo 01 e incorpora o uso de Sessões.
- De acordo com a documentação do SQLAlchemy, as sessões são usadas para estabelecer uma conversação com o banco de dados.
- Ela representa uma área onde objetos são carregados e mantidos durante a sessão do usuário (até que ela seja encerrada).
- Por exemplo, se você salva um dado de um objeto utilizando uma sessão e modifica este dado, a sessão detecta esta alteração.

Sesssão

- Parece complexo do ponto de vista conceitual e realmente é. Entretanto, na prática o uso de sessões facilita bastante a codificação.
- Podemos executar instruções SQL bruta a partir das sessões da mesma forma que fizemos com a conexão obtida da engine no exemplo 01.

Sessão

- Retomando o exemplo 02, vamos focar apenas no que é diferente em relação ao exemplo 01.
- A primeira coisa que fazemos é importar a classe `Session`

```
from sqlalchmey.orm import Session
```

- O detalhe da importação é que a classe `Session` pertence ao módulo `ORM`. Lembre-se da definição do que é o SQLAlchemy.

- Vamos, em seguida, criar um objeto `session`.

```
session = Session(bind=engine)
```

- Observe que continuamos utilizando a `engine` que é o ponto de acesso ao banco de dados. Vinculamos a engine a Sessão que estamos criando.
- O objeto session vai utilizar o banco que foi definido para a engine.
- No exemplo 02, vamos criar o banco de dados utilizando SQL bruto.

- O código abaixo cria o banco de dados com uma tabela de usuários:

```
# trecho de app.py do Exemplo 02
from sqlalchemy import text
sql = text("""CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nome TEXT NOT NULL)""")

# neste momento o banco é criado com a tabela de usuários
session.execute(sql)
```

- Observe que a função `execute` fara o trabalho de criação do banco utilizando a engine, mas isso é feito indiretamente e internamente.

- O próximo passo do exemplo é executar alguma instrução de registro de dados de usuários no banco
- Novamente, será utilizado SQL bruto conforme o código abaixo:

```
# trecho de app.py
insert = text("""INSERT INTO users(nome) VALUES(:nome)""")

session.execute(insert, {'nome':'zefa'})
```

- Observe a semelhante entre a operação com sessões e a operação do Exemplo 01 com a engine.
- Ainda não chegamos ao potencial do usso de sessões.

Sessões e Modelos

Sessões e Modelos

- No exemplo 03, alcançamos o mesmo resultado do Exemplo 02. Entretanto, incluímos um novo recurso que é o uso de Modelos.
- O SQLAlchemy, na sua definição, indica que ele é um Mapeador Objeto-Relacional para python.
- O uso de Sessões com Modelos é a concretização dessa definição, além - **é claro** - de inúmeros outros recursos.
- Portanto, não faz muito sentido usar SQL Bruto como nos exemplos anteriores, se podemos utilizar algo diretamente em Orientação a objetos para manipular o banco de dados.

- A novidade do Exemplo 03 é incluir um novo arquivo com a definição de um modelo de usuários

```
# trecho de app.py
from models import Base, User
```

- A classe `Base` é uma classe definda no arquivo `models.py` cujo objetivo é servir de classe modelo para as nossas classe.
  - A classe `Base` é fudamental pois ela vair permitir o mapeamento declarativo das tabelas e suas colunas de modo que o SQLAlchemy conheças essas definições
  - A classe `User` terá como modelo a classe Base

- O código do arquivo `models.py` define a classe `Base` e a classe `User`

```python
# código models.py
from sqlalchemy.orm import DeclarativeBase
from sqlalchemy.orm import Mapped, mapped_column

class Base(DeclarativeBase):
    pass

class User(Base):
    __tablename__ = 'users'
    id: Mapped[int] = mapped_column(primary_key=True)
    nome:Mapped[str]
```

- A classe `Base` herda da classe `DeclarativeBase`, que o SQLAlchemy utiliza como definição de Modelo padrão.
- A classe `User` utiliza o mapeamento declarativo. É definido o nome da tabela e dois atributos, que são:
  - id (chave primária)
  - nome do usuário
- `id: Mapped[int]`: indica que o `id` será mapeado para inteiro no banco escolhido (INTEGER para SQLITE).
- `mapped_column(primary_key=True)` é um complemento para a coluna `id`. Neste caso, indica que id é a chave primária
- `nome: Mapped[str]` é mapeado para `TEXT` no sqlite

- Esta é a primeira parte da configuração de uso de Modelos com Sessões.
- Uma vez que temos os modelos e o objeto `session` definido no arqiuvo `app.py` podemos criar o banco.
- A criação do banco utiliza o registro de metadados acessível através da classe `Base`.

```
# trecho de app.py
Base.metadata.create_all(bind=engine)
```

- Observe que utilizamos a engine vinculada a criação das tabelas no banco.

- A última é etapa é utilizar os modelos para criar objetos e persistir os dados no banco.
- O trecho de código abaixo cria um objeto e o adiciona ao banco.

```
#trecho de app.py
usuario = User(nome='jose')
# o objeto session já criado anteriormente
session.add (usuario)
session.commit()
```

- Observe o exemplo 03 na íntegra e perceba que não se trabalha com SQL Bruto. Isso é abstraído pelo SQLAlchemy e sua implementação do ORM.

- Se você acompanhou até aqui, compare os exemplos 2 e 3.
- O exemplo 02 faz a criação de tabela e inserção de usuários. Isso é feito utilizando o objeto `session` e executando SQL definido diretamente no código
- O exemplo 3 cria a tabela e insere dados sem trabalhar com SQL Bruto.
- O objetivo do ORM é permite que se trabalhe exclusivamente com orientação a objetos e mesmo assim seja possível e - ***fácil*** - de manipular o banco de dados.
- Os modelos podem ser mais detalhados que este exemplo.