

ПОДКЛЮЧЕНИЕ К БАЗАМ ДАННЫХ

ВВЕДЕНИЕ

Задачи урока

- Научиться подключаться к базе данных
- Обернуть метод подключения в класс

create_engine

это функция, которая предоставляется ORM-фреймворком, таким как SQLAlchemy, для создания объекта-движка базы данных. Она играет важную роль в установлении соединения между вашим приложением и базой данных

Цель **create_engine**

передать конфигурацию базы данных ORM-фреймворку и создать объект движка, который будет использоваться для управления соединением с базой данных

Пример использования **create_engine** с SQLAlchemy в языке Python

```
...
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
connection_string = f"postgresql://postgres:password@localhost:5432/synergy"
engine = create_engine(connection_string)
```

В приведенном выше примере мы создаем объект движка для базы данных SQLite, используя `create_engine` и передавая ему строку подключения в качестве параметра

Строка подключения может содержать информацию

Тип базы
данных

Расположение
базы данных

Учетные данные и другие параметры,
необходимые для установления соединения

После создания объекта движка, можно использовать его для

- Управления соединением с базой данных
- Создания таблиц
- Выполнения SQL-запросов и других операций

`create_engine` также поддерживает различные параметры, которые можно передать для настройки поведения движка. Например, вы можете указать пул соединений, отвечающий за управление пулом активных соединений с базой данных, или указать параметры кодировки, используемые при работе с данными в базе данных

Определение объекта движка базы данных с помощью `create_engine` является важным шагом при работе с ORM, поскольку оно обеспечивает взаимодействие с базой данных и облегчает выполнение операций CRUD (Create, Retrieve, Update, Delete) и других действий с данными в вашем приложении

sessionmaker

это класс, предоставляемый ORM-фреймворком, таким как SQLAlchemy, для создания объекта сессии. Сессия представляет собой основной интерфейс для взаимодействия с базой данных в ORM

Для работы с ORM, необходимо создать сессию, которая будет использоваться для выполнения операций CRUD (Create, Retrieve, Update, Delete) и других операций с базой данных

Пример использования sessionmaker с SQLAlchemy в языке Python

```
...
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
connection_string = f"postgresql://postgres:password@localhost:5432/synergy"
engine = create_engine(connection_string)
Session = sessionmaker(bind=engine)
Session = Session()
```

В приведенном выше примере, мы создаем объект движка для базы данных **SQLite**, используя `create_engine`. Затем мы создаем объект сессии, используя `sessionmaker` и передавая ему объект движка с помощью параметра `bind`. Это устанавливает связь между сессией и движком базы данных.

После создания объекта сессии, мы можем использовать его для выполнения различных операций с базой данных, **например:**

Создавать новые объекты

Изменять объекты

Удалять объекты

Выполнять запросы

Получать результаты

Преимущества sessionmaker



Обеспечивает единое место для создания сессий



Позволяет управлять параметрами сессии, такими как уровень изоляции и поведение транзакций



Позволяет настраивать пул соединений и другие аспекты работы с базой данных

Использование `sessionmaker` упрощает разработку с помощью ORM, обеспечивая удобный интерфейс для работы с базой данных и обеспечивая согласованность данных во время сеанса работы с базой данных

Так же все это лучше обернуть в класс не забывая про ООП

```
...
class Connection:

    def __init__(self, sql_type, user, password, server, port =
None, **args) -> None:
        self.user = user
        self.password = password
        self.server = server
        self.port = port
        self.sql_type = sql_type
        self.args = args

    @property
    def engine(self):
        if self.sql_type == "MSSQL":
            return f"mssql+pymssql://{{self.user}}:{{self.password}}@{{self.server}}/{{self.args['db_name']}}"
        if self.sql_type == "PostgresSQL":
            return f"postgresql://{{self.user}}:{{self.password}}@{{self.server}}:{{str(self.port)}}/{{self.args['db_name']}}"
        else: print("Соединение не поддерживается")

    @property
    def async_engine(self):
        if self.sql_type == "MSSQL":
            return f"mssql+aiodbc://{{self.user}}:{{self.password}}@{{self.server}}/{{self.args['db_name']}}"
        if self.sql_type == "PostgresSQL":
            return f"postgresql+asyncpg://{{self.user}}:{{self.password}}@{{self.server}}:{{str(self.port)}}/{{self.args['db_name']}}"
        else: print("Соединение не поддерживается")
```

Мы создали класс, который в зависимости от типа подключения возвращает нам строку подключения

```
...
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession

from orm.connection import Connection

class SessionBuilder:

    def __init__(self, connection: Connection) -> None:
        self.engine = create_engine(connection.engine)
        self.async_engine =
create_async_engine(connection.async_engine)

    def buid(self):
        Session = sessionmaker(bind=self.engine)
        return Session()

    def async_buid(self):
        with sessionmaker(bind=self.session_dwh.async_engine,
class_=AsyncSession, expire_on_commit=False)() as session:
            return session
```

И класс который создает сессию

Пример использования

```
...
from orm.connection import Connection
from orm.session_builder import SessionBuilder

session = SessionBuilder(
    Connection(
        server='localhost',
        port=5432,
        user='postgres',
        password='password',
        db_name='synergy',
        sql_type='PostgreSQL'
    )
).buid()
```

ИТОГИ

- ✓ Для подключения используется `create_engine` со строкой подключения
- ✓ Все методы можно описать в классы и использовать классовую структуру в своем проекте