**Модуль 7: ORM SQLAlchemy**

**Заняття 1: Основи роботи з ORM SQLAlchemy**

**Причини використання та підключення**

**RDB** бази даних є стандартом для зберігання та пошуку інформації. Коли ми хочемо десь зберігати дані між запусками нашого застосунку, ми в 9/10 випадків скористаємося тією абл іншою реляційною базою даних. Майже всі популярні реляційні бази даних реалізують протокол спілкування за допомогою мови SQL. Отже, якщо потрібно зберігати якусь інформацію, ми неминуче приходимо до SQL запитів у коді застосунку. В результаті виходить, що частина логіки застосунку написана здебільшого мовою програмування, а частина (і часто чимала) написана на SQL. Це ускладнює підтримку такого коду, робить його менш читабельним і складнішим для сприйняття.

Ось для прикладу дві функції для створення таблиці проектів та завдань, які належать до цих проектів:

def create\_projects\_table(conn):

try:

c = conn.cursor()

c.execute(

"""CREATE TABLE IF NOT EXISTS projects (

id integer PRIMARY KEY,

name text NOT NULL,

begin\_date text,

end\_date text

);"""

)

except Error as e:

print(e)

def create\_tasks\_table(conn):

try:

c = conn.cursor()

c.execute(

"""CREATE TABLE IF NOT EXISTS tasks (

id integer PRIMARY KEY,

name text NOT NULL,

priority integer,

project\_id integer NOT NULL,

status\_id integer NOT NULL,

begin\_date text NOT NULL,

end\_date text NOT NULL,

FOREIGN KEY (project\_id) REFERENCES projects (id)

);"""

)

except Error as e:

print(e)

Цей код можна спростити, але що робити, коли нам доведеться додати колонку до нашої таблиці завдань, додати зв'язок з новою таблицею виконавців, поміняти тип якогось поля?

Всі ці проблеми змушують змінювати SQL запит, а потім всі запити, що стосуються цієї таблиці у коді. Такий підхід очевидно викликає лавину помилок і не дозволяє створювати скільки-небудь складні системи.

Але якщо винести завдання генерації самих SQL запитів в окремий пакет/модуль? Тоді ми можемо зосередитись на описі таблиць як Python класів і, якщо необхідно, щось змінити в структурі даних, що зберігаються, ми робитимемо це тільки в одному місці. У всіх інших місцях нехай механізм генерації SQL запитів зробить це за нас, спираючись на певний опис структури даних в одному місці.

Саме це завдання вирішують усі **ORM** (Object Relationship Mapper) пакети. Вони всі є генераторами SQL запитів, що спираються на визначену в одному місці структуру даних та приховують реальний SQL, дозволяючи розробнику зосередитись на використанні лише однієї мови у застосунку.

[**SQLAlchemy**](https://www.sqlalchemy.org/) (https://www.sqlalchemy.org/) — це найпопулярніший ORM-пакет, що спрощує і стандартизує роботу з SQL-сумісними базами даних.

**SQLAlchemy** — великий та досить складний інструмент. Він дуже гнучкий і дозволяє зробити все, що ви можете зробити, використовуючи "чистий" SQL. За таку "гнучкість" доводиться платити складністю структури та наявністю деякої кількості "магії" у цьому пакеті.

NOTE

Тут і далі під "магією" слід розуміти неочевидну, не інтуїтивну та місцями заплутану поведінку системи.

Для початку роботи з Sqlalchemy варто його встановити, виконавши команду в консолі

poetry add sqlalchemy

Робота з ORM починається зі створення об'єкта, що інкапсулює доступ до бази даних, в **SQLAlchemy** він називається **engine**.

from sqlalchemy import create\_engine

engine = create\_engine('sqlite:///:memory:', echo=True)

У цьому прикладі ми використовуємо SQLite базу даних у пам'яті, на диск нічого не записується.

Для початку потрібно розуміти, що **SQLAlchemy** містить кілька шарів (рівнів) організації. Найнижчий рівень організації — це ядро, **core**, на якому ви можете писати дуже близький до SQL коду Python код. Робота на **core** рівні не надто зручна і потрібна, в основному, тільки для дуже специфічних завдань.

Далі розглянемо як працювати на **core** рівні.

**Робота SQLAlchemy на core рівні**

Для цього потрібно створити спеціальний об'єкт-міст між базою даних та Python кодом. Завдання цього об'єкту синхронізувати базу даних та опис цієї бази в Python об'єктах.

from sqlalchemy import MetaData

metadata = MetaData()

Створимо дві таблиці: users, addresses.

from sqlalchemy import Table, Column, Integer, String, ForeignKey

users = Table('users', metadata,

Column('id', Integer, primary\_key=True),

Column('name', String),

Column('fullname', String),

)

addresses = Table('addresses', metadata,

Column('id', Integer, primary\_key=True),

Column('user\_id', Integer, ForeignKey('users.id')),

Column('email\_address', String, nullable=False)

)

Опис таблиць та полів — досить виразне і додаткових коментарів не потребує. Щоб створити описані вище таблиці у порожній базі даних, можна "попросити" наш metadata об'єкт зробити це:

metadata.create\_all(engine)

Тепер, щоб наприклад створити нового користувача в базі:

ins = users.insert().values(name='jack', fullname='Jack Jones')

print(str(ins)) # INSERT INTO users (name, fullname) VALUES (:name, :fullname)

Цей код створює вираз SQL, його можна виконати, коли потрібно, або модифікувати в майбутньому. Щоб виконати, потрібно створити з'єднання до бази даних і виконати вираз, використовуючи це з'єднання:

conn = engine.connect()

result = conn.execute(ins)

conn.close()

Для отримання даних із таблиці:

from sqlalchemy import create\_engine, Table, Column, Integer, String, ForeignKey

from sqlalchemy import MetaData

from sqlalchemy.sql import select

engine = create\_engine('sqlite:///:memory:', echo=True)

metadata = MetaData()

users = Table('users', metadata,

Column('id', Integer, primary\_key=True),

Column('name', String),

Column('fullname', String),

)

addresses = Table('addresses', metadata,

Column('id', Integer, primary\_key=True),

Column('user\_id', Integer, ForeignKey('users.id')),

Column('email\_address', String, nullable=False)

)

metadata.create\_all(engine)

with engine.connect() as conn:

ins = users.insert().values(name='jack', fullname='Jack Jones')

print(str(ins))

result = conn.execute(ins)

s = select(users)

result = conn.execute(s)

for row in result:

print(row) # (1, u'jack', u'Jack Jones')

Є методи для отримання одного рядка, кількох тощо.

Загальний підхід повинен бути вже зрозумілим, це дуже схоже на конструювання SQL запитів, тільки замість ключових слів на SQL, використовуються методи та/або функції з **SQLAlchemy.**

Детальна документація з **SQLAlchemy** core доступна за [посиланням](https://docs.sqlalchemy.org/en/14/core) (https://docs.sqlalchemy.org/en/14/orm/tutorial.html).

Це далеко не найзручніший, але дуже потужний набір інструментів, що надаються **SQLAlchemy**.

**Механізм сесій в SQLAlchemy.**

Щоб працювати з базою даних так, ніби немає жодного окремого сховища, просто об'єкти Python "на льоту" зберігаються і так само "на льоту" доступні для роботи, в **SQLAlchemy** є високорівневіший механізм ORM.

Цей механізм використовує сесії, які приховують створення з'єднань з базою та дають можливість виконувати кілька транзакцій одним запитом для економії ресурсів.

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

# Create an engine that stores data in the local directory's

# sqlalchemy\_example.db file.

engine = create\_engine('sqlite:///sqlalchemy\_example.db')

DBSession = sessionmaker(bind=engine)

session = DBSession()

У цьому прикладі ми створюємо клас DBSession, об'єкти якого є окремими сесіями доступу до бази даних. Кожна така сесія може зберігати набір транзакцій і виконувати їх тільки коли це дійсно потрібно. Таке "ледаче" виконання зменшує навантаження на базу та прискорює роботу застосунку.

Сесія в ORM — це об'єкт, за допомогою якого ви можете управляти, коли саме накопичені зміни будуть застосовані до бази. Для цього є метод commit. Є методи для додавання одного або кількох об'єктів до бази (add, add\_all).

**Створення БД за допомогою моделей SQLAlchemy**

Найпростіше це розглянути на конкретному прикладі. Давайте створимо дві таблиці в БД persons та addresses:

from sqlalchemy import create\_engine, Column, Integer, String, ForeignKey

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker, relationship

engine = create\_engine('sqlite:///sqlalchemy\_example.db')

DBSession = sessionmaker(bind=engine)

session = DBSession()

Base = declarative\_base()

class Person(Base):

\_\_tablename\_\_ = 'persons'

id = Column(Integer, primary\_key=True)

name = Column(String(250), nullable=False)

class Address(Base):

\_\_tablename\_\_ = 'addresses'

id = Column(Integer, primary\_key=True)

street\_name = Column(String(250))

street\_number = Column(String(250))

post\_code = Column(String(250), nullable=False)

person\_id = Column(Integer, ForeignKey('persons.id'))

person = relationship(Person)

Base.metadata.create\_all(engine)

Base.metadata.bind = engine

new\_person = Person(name="Bill")

session.add(new\_person)

session.commit()

new\_address = Address(post\_code='00000', person=new\_person)

session.add(new\_address)

session.commit()

for person in session.query(Person).all():

print(person.name)

У якості об'єкта, що сполучає стан бази та опис бази, в Python коді виступає Base, саме цей клас відповідає за "магію" синхронізації таблиць бази даних та їх описи в Python класах Person та Address.

**ORM**підхід виразніший. Наприклад, додавання нових записів до таблиці — це просто створення нових об'єктів класів Person та Address:

new\_person = Person(name="Bill")

session.add(new\_person)

session.commit()

new\_address = Address(post\_code='00000', person=new\_person)

session.add(new\_address)

session.commit()

Зверніть увагу, щоб зміни набули чинності та були записані в базу, обов'язково потрібно виконати commit, після того, як ми додали дані методом add.

Щоб отримати дані з бази, можна скористатися методом query:

for person in session.query(Person).all():

print(person.name) # Bill

Ви можете додавати фільтруючі умови, використовуючи метод filter, та інші SQL конструкції, використовуючи методи об'єкту Query (саме його повертає метод query). Конструювання запитів дуже спрощується за допомогою **ORM**, водночас це досить велика тема і рекомендуємо ознайомитися з документацією за[**посиланням**](https://docs.sqlalchemy.org/en/14/orm/tutorial.html) (https://docs.sqlalchemy.org/en/14/orm/tutorial.html)**.**

**Визначення відношень**

**SQLAlchemy** середовище ORM надає функції відношень один-до-багатьох, багато-до-багатьох між двома об'єктами.

Наприклад, для звичайного блогу користувачі та статті, очевидно, є відношеннями один до багатьох. Стаття належить користувачеві, і користувач може написати багато статей. Потім вони можуть використати асоціацію зовнішнього ключа. Визначимо два класи Article та User і пропишемо в них відношення:

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer(), primary\_key=True)

name = Column(String(20))

articles = relationship('Article', back\_populates='author')

class Article(Base):

\_\_tablename\_\_ = 'articles'

id = Column(Integer(), primary\_key=True)

title = Column(String(255))

content = Column(Text())

user\_id = Column(Integer(), ForeignKey('users.id'))

author = relationship('User', back\_populates='articles')

Клас Article зберігає у полі user\_id зовнішній ключ на таблицю класу User. Так само для query запитів ми створюємо два відношення: articles у класі User та author у класі Article. Параметр back\_populates пов'язує ці відношення між собою. Тепер ми можемо будувати двоспрямовані запити до таблиць.

Можна отримати доступ до інформації в таблиці статей через атрибут статей користувачів:

users= session.query(User).filter\_by(name='Peter Miller').all()

for user in users:

for article in user.articles:

print(article.title, user.name)

Або у зворотний бік:

article = session.query(Article).filter\_by(title='Our country’s saddest day').one()

print(article.title, article.author.name)

Відношення багато до багатьох. Наприклад, запис зазвичай має кілька тегів. Між тегами та записами існує відношення багатьох до багатьох. Відношення багато до багатьох не може бути визначено безпосередньо, і його необхідно розбити на два відношення один до багатьох. Для цього використовують допоміжну таблицю для зв'язку між цими таблицями. Ми зараз не повністю розписуватимемо цей процес, тому що розберемо його в наступному розділі при розборі міграцій.

Якщо з якоїсь причини нам потрібно визначити відношення один до одного, то воно будується на відношенні один до багатьох за допомогою параметра uselist

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer(), primary\_key=True)

name = Column(String(20))

userinfo = relationship('UserInfo', backref='user', uselist=False)

class UserInfo(Base):

\_\_tablename\_\_ = 'userinfo'

id = Column(Integer(), primary\_key=True)

telegram = Column(String(11))

phone = Column(String(11))

site = Column(String(64))

user\_id = Column(Integer(), ForeignKey('users.id'))

INFO

Параметр backref є аналогом раніше розглянутого параметра back\_populates. Відмінність у тому, що відношення з параметром backref достатньо оголосити в одному класі, щоб можна було б будувати двоспрямовані запити з query.

**Простий CRUD**

Давайте побудуємо простий [CRUD](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) ([https://en.wikipedia.org/wiki/Create, read, update and delete](https://en.wikipedia.org/wiki/Create,%20read,%20update%20and%20delete)) на попередньому прикладі. Визначимо файл моделей одночасно з підключенням до БД.

**rel\_one\_to\_many.py**

from sqlalchemy.engine import create\_engine

from sqlalchemy.orm import sessionmaker, declarative\_base, relationship

from sqlalchemy import Column, String, Integer, Text, ForeignKey

engine = create\_engine("sqlite:///test.db")

Session = sessionmaker(bind=engine)

session = Session()

Base = declarative\_base()

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer(), primary\_key=True)

name = Column(String(20))

articles = relationship('Article', back\_populates='author')

class Article(Base):

\_\_tablename\_\_ = 'articles'

id = Column(Integer(), primary\_key=True)

title = Column(String(255))

content = Column(Text())

user\_id = Column(Integer(), ForeignKey('users.id'))

author = relationship('User', back\_populates='articles')

Base.metadata.create\_all(engine)

Base.metadata.bind = engine

**Create (створення)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/crud#create-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F)

Щоб використовувати SQLAlchemy для додавання даних до БД, нам потрібно тільки створити екземпляр відповідного класу, викликати session.add і потім session.commit. Всі дані в БД.

from rel\_one\_to\_many import User, Article, session

user = User(name='Boris Johnson')

session.add(user)

session.commit()

article = Article(title='Our country’s saddest day', content='Lorem ipsum...', user\_id=user.id)

session.add(article)

session.commit()

**Read (читання)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/crud#read-%D1%87%D0%B8%D1%82%D0%B0%D0%BD%D0%BD%D1%8F)Якщо ми знаємо ідентифікатор користувача, ми можемо використовувати метод get. Метод для отримання значення поля може безпосередньо використовувати властивості класу:

from rel\_one\_to\_many import User, Article, session

user = session.query(User).get(1)

print(user.id, user.name)

Виведення:

1 Boris Johnson

Метод all використовують, щоб отримати всі результати запиту:

users = session.query(User).all()

for user in users:

print(user.id, user.name)

Є також методи first, scalar з one. Різниця між трьома:

* first — Повертає перший об'єкт запису, якщо він є.
* one — Запитує всі рядки і викликає виняток, якщо щось повертається, крім одного результату.
* scalar — Повертає перший елемент першого результату, None, якщо результату немає, або помилку, якщо їх більше ніж один результат.

Метод filter\_by використовується для фільтрації за певним полем, або його аналог метод filter трохи з іншим синтаксисом. Давайте відфільтруємо за полем:

user1 = session.query(User).filter\_by(name='Boris Johnson').first()

user2 = session.query(User).filter(User.name == 'Boris Johnson').scalar()

print(user1.id, user1.name)

print(user2.id, user2.name)

Виведення:

1 Boris Johnson

1 Boris Johnson

**Update (оновлення)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/crud#update-%D0%BE%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)Подібно до знову доданих даних, додавайте і фіксуйте після оновлення даних.

from rel\_one\_to\_many import User, Article, session

article = session.query(Article).get(1)

article.content = 'Very important content for the article'

session.add(article)

session.commit()

**Delete (видалення)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/crud#delete-%D0%B2%D0%B8%D0%B4%D0%B0%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)Видалення відбувається напряму — викликом методу delete для отриманого об'єкту:

from rel\_one\_to\_many import User, Article, session

article = session.query(Article).get(1)

session.delete(article)

session.commit()

**Міграція баз даних за допомогою Alembic**

В ідеальному світі один раз описана база даних ніколи не змінює свою структуру. Наш світ не є ідеальним і бази даних постійно змінюються. Це досить болісний процес, особливо, якщо врахувати, що в базі часто вже є дані і потрібно їх правильно і акуратно перенести на нову схему.

Щоб якось зберегти історію змін бази даних, вигадали механізм міграцій.

Зберігати історію змін потрібно переважно, щоб мати можливість відкотити зміни до працюючого стану. На жаль, це відбувається частіше, ніж хотілося б, і практично неминуче.

Щоб якось зменшити ймовірність помилок при зміні структури бази даних цей механізм спробували автоматизувати. **SQLAlchemy** не має свого інструмента міграцій. Для цього використовується окремий пакет —[**Alembic**](https://alembic.sqlalchemy.org/en/latest/) (https://alembic.sqlalchemy.org/en/latest/)**.**

Насправді,**Alembic**— дуже потужний інструмент і він не зав'язаний жорстко на **SQLAlchemy**, може бути інструментом міграцій для широкого набору різних пакетів. Найчастіше, звичайно, застосовується в парі з**SQLAlchemy.**

Міграції, за своєю суттю, — це просто набір скриптів, що описують перехід від схеми A у схему B, і назад. Головна заслуга **Alembic** — це автогенерація таких скриптів. Коли ви змінюєте модель, що описує таблицю в базі, та з ваших змін **Alembic**може згенерувати дві функції: як перейти з поточного стану бази в описаний вами Python код і назад. Другий дуже важливий механізм **Alembic**— це історія всіх змін. Alembic зберігає весь ланцюжок всіх змін і може автоматизувати перехід до будь-якої точки у цій історії. По суті, аналог GIT лише для баз даних.

Детальний розбір роботи з **Alembic**розбиратиметься далі в наступному розділі, але спочатку рекомендується ознайомитися з досить детальним описом та набором рецептів на [сторінці офіційної документації](сторінці%20офіційної%20документації)

(https://alembic.sqlalchemy.org/en/latest/).

**Асинхронні конектори та ORM**

Більшість ORM з'явилася порівняно давно, до появи підтримки asyncio у Python. Саме з цієї причини майже всі ORM синхронні та конектори до баз також синхронні. Починаючи з Python 3.6 стало очевидним, що Python буде розвиватися у бік розширення підтримки asyncio, і будуть потрібні асинхронні ORM. **SQLAlchemy** з версії 1.4 та 2.0 підтримує asyncio синтаксис та дозволяє асинхронно працювати із базою даних. У будь-якому випадку для цього **SQLAlchemy** потрібні конектори до бази даних, що підтримують asyncio. Основні відмінності від "звичайного", синхронного підходу, описані на [спеціальній сторінці](https://docs.sqlalchemy.org/en/latest/orm/extensions/asyncio.html) (https://docs.sqlalchemy.org/en/20/orm/extensions/asyncio.html).

Є сторонні проекти, наприклад [databases](https://github.com/encode/databases) (https://github.com/encode/databases), який перетворює синхронні конектори в окремі потоки і надає asyncio API. Як додатковий інструмент дає можливість працювати з**SQLAlchemy** на рівні **core**, підміняючи звичайний, синхронний engine.

Ви можете використовувати асинхронний режим роботи з SQLAlchemy, якщо ви використовуєте діалекти, сумісні з asyncio, такі, як asyncpg, aiomysql або aiopg.

Для цього вам потрібно створити асинхронний движок за допомогою функції create\_async\_engine. Функція приймає URL з'єднання з додатковим параметром +asyncpg, +aiomysql або +aiopg. Наприклад:

engine = create\_async\_engine("postgresql+asyncpg://user:password@host/database")

Потім ви можете використовувати асинхронний движок для отримання асинхронних з'єднань або транзакцій за допомогою методів AsyncEngine.connect() і AsyncEngine.begin(), які повертають асинхронні контекстні менеджери. Ви можете виконувати SQL-запити за допомогою методів AsyncConnection.execute() або AsyncConnection.stream(), які повертають об'єкти Result або AsyncResult відповідно.

Наприклад:

async with engine.connect() as conn:

result = await conn.execute(select(User))

rows = result.fetchall()

Якщо ви хочете використовувати ORM з асинхронним режимом, вам потрібно створити асинхронну сесію за допомогою класу AsyncSession, який також приймає асинхронний движок як параметр. Ви можете виконувати ORM-запити за допомогою методу AsyncSession.execute(), який повертає об'єкт AsyncResult.

Наприклад:

from sqlalchemy.ext.asyncio import create\_async\_engine, AsyncSession, AsyncEngine, async\_sessionmaker

engine: AsyncEngine = create\_async\_engine('postgresql+asyncpg://user:password@host/database', echo=True)

AsyncDBSession = async\_sessionmaker(engine, expire\_on\_commit=False, class\_=AsyncSession)

async with AsyncDBSession() as session:

result = await session.execute(select(User).where(User.name == "Alice"))

user = result.scalars().first()

Асинхронний режим роботи з SQLAlchemy дозволяє вам використовувати переваги asyncio для підвищення продуктивності та масштабованості вашого додатка. Однак він також вимагає від вас дотримання деяких правил і обмежень, таких як:

* Використання тільки одного **event loop** для всіх асинхронних операцій.
* Уникнення виклику синхронних методів і функцій всередині асинхронного коду.
* Використання методу run\_sync для виконання синхронних операцій, таких як створення і видалення таблиць.
* Використання методу scalar або scalars для вилучення об'єктів з результатів ORM-запитів.

INFO

Для більш детальної інформації щодо використання асинхронного режиму роботи з SQLAlchemy ви можете звернутися до офіційної документації з підтримки asyncio у SQLAlchemy, яка містить приклади коду та посилання на додаткові ресурси — Asynchronous I/O (asyncio) - SQLAlchemy 2.0 Documentation

Є й повністю асинхронні ORM, які спочатку розроблялися з підтримкою asyncio:

* [Gino](https://python-gino.org/) (https://python-gino.org/)
* [Pony](https://docs.ponyorm.org/) (https://docs.ponyorm.org/)
* [Tortoise](https://tortoise.github.io/) (https://tortoise.github.io/)
* [Peewee](http://docs.peewee-orm.com/en/latest/) (https://docs.peewee-orm.com/en/latest/)

**Новий інтерфейс запитів SQLAlchemy 2.0**

SQLAlchemy 2.0 являє собою значне оновлення фреймворка для роботи з базами даних у Python з безліччю нових можливостей і поліпшень. Однією з найважливіших змін є єдиний інтерфейс запитів для **Core** та **ORM**. Це дає змогу використовувати один і той самий конструктор select() для побудови SQL-запитів до об'єктів**ORM** або таблиць **Core**. Що робить код більш консистентним, читабельним і легко переношуваним між різними рівнями абстракції.

Ми розглянемо, як використовувати новий інтерфейс запитів API SQLAlchemy 2.0 для виконання різних типів запитів до бази даних, таких як вибірка, фільтрація, сортування, групування, об'єднання, агрегація тощо. Також подивимось, як працювати з результатами запитів і використовувати різні опції та модифікатори для налаштування поведінки запитів.

**Визначення моделей**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/new#%D0%B2%D0%B8%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B5%D0%B9)

Визначимо наші класи-моделі, які успадковуються від класу Base і відповідають таблицям у базі даних.

from sqlalchemy import create\_engine, Integer, String, ForeignKey, select, Text, and\_, desc, func

from sqlalchemy.orm import declarative\_base, sessionmaker, Mapped, mapped\_column, relationship

engine = create\_engine('sqlite:///:memory:', echo=False)

DBSession = sessionmaker(bind=engine)

session = DBSession()

Base = declarative\_base()

class User(Base):

\_\_tablename\_\_ = 'users'

id: Mapped[int] = mapped\_column(primary\_key=True)

fullname: Mapped[str] = mapped\_column(String)

class Post(Base):

\_\_tablename\_\_ = 'posts'

id: Mapped[int] = mapped\_column(Integer, primary\_key=True)

title: Mapped[str] = mapped\_column(String(150), nullable=False, index=True)

body: Mapped[str] = mapped\_column(Text, nullable=False)

user\_id: Mapped[str] = mapped\_column('user\_id', Integer, ForeignKey('users.id'))

user: Mapped['User'] = relationship(User)

Base.metadata.create\_all(engine)

У версії 2.0 тип стовпця можна визначити за допомогою Mapped. Якщо є якісь додаткові параметри, їх можна вказати у виклику mapped\_column().

Для створення таблиць у базі даних за нашими моделями ми використовуємо метод Base.metadata.create\_all(), який приймає engine як параметр. Цей метод створить таблиці users і posts у базі даних sqlite, згідно з нашими класами User та Post.

**Додавання користувачів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/new#%D0%B4%D0%BE%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D1%96%D0%B2)Додамо нових користувачів зі списку names у базу даних.

if \_\_name\_\_ == '\_\_main\_\_':

names = ['Crystal Najera', 'Shaun Beck', 'Kathrin Reinhardt']

for name in names:

user = User(fullname=name)

session.add(user)

session.commit()

**Запити до бази даних**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/new#%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D0%B8-%D0%B4%D0%BE-%D0%B1%D0%B0%D0%B7%D0%B8-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)Для виконання запитів до бази даних ми будемо використовувати функцію select(). Функція приймає одну або кілька сутностей (entities - англ.), які можуть бути класами-моделями **ORM** або об'єктами **Table Core.**

Функція select() повертає об'єкт Select, який є конструктором SQL-запитів.

Наприклад, щоб вибрати всі записи з таблиці users, ми можемо написати такий код:

stmt = select(User)

result = session.execute(stmt)

for user in result.scalars():

print(user.id, user.fullname)

І отримаємо таке виведення:

1 Crystal Najera

2 Shaun Beck

3 Kathrin Reinhardt

Метод session.execute() приймає об'єкт Select як параметр і виконує запит в базі даних. Він повертає об'єкт Result, що містить результати запиту.

Ми можемо отримати результати запиту з об'єкта result за допомогою різних методів і атрибутів:

* result.all() - повертає список усіх рядків результату.
* result.first() - повертає перший рядок результату або None, якщо результат порожній.
* result.one() - повертає єдиний рядок результату або викликає виняток NoResultFound або MultipleResultsFound за відсутності або наявності більше одного рядка.
* result.one\_or\_none() - повертає єдиний рядок результату або None за відсутності рядка або викликає виняток MultipleResultsFound за наявності більше одного рядка.

Об'єкт Session має два додаткові методи виконання, які роблять роботу з рядками (row) з одним значенням більш зручною у використанні:

* scalars() повертає об'єкт ScalarResult, що є ітератором за першим стовпцем кожного рядка результату
* scalar() повертає перше значення першого рядка результату.
* mappings() повертає об'єкт MappingsResult, що є ітератором за словниками ключ-значення для кожного рядка результату.

Ми також можемо використовувати функцію select() для вибірки певних стовпчиків із таблиці. Для цього ми можемо передати атрибути-колонки класу-моделі **ORM** або об'єкта **Table Core** як аргументи функції select().

Наприклад:

stmt = select(User.id, User.fullname)

result = session.execute(stmt)

users = []

for row in result:

print(row)

users.append(row)

Виведення:

(1, 'Crystal Najera')

(2, 'Shaun Beck')

(3, 'Kathrin Reinhardt')

**Фільтрація записів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/new#%D1%84%D1%96%D0%BB%D1%8C%D1%82%D1%80%D0%B0%D1%86%D1%96%D1%8F-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%96%D0%B2)Для фільтрації записів за умовою ми можемо використовувати метод where() об'єкта Select, що приймає один або кілька аргументів-критеріїв.

Критерії можуть бути виразами порівняння, логічними операторами або іншими об'єктами.

Наприклад:

stmt = select(User).where(User.fullname == "Shaun Beck")

result = session.execute(stmt).scalar\_one()

print(result.id, result.fullname)

Виведення:

2 Shaun Beck

Ми можемо використовувати різні оператори для порівняння колонок зі значеннями або іншими колонками, такі як як ==, !=, <, >, <=, >=, in\_, like, ilike тощо.

Наприклад:

stmt = select(User).where(User.fullname.like("%ha%"))

result = session.execute(stmt)

for user in result.scalars().all():

print(user.id, user.fullname)

Виведення:

2 Shaun Beck

3 Kathrin Reinhardt

Ми також можемо використовувати логічні оператори для комбінування кількох критеріїв в одну умову where, такі як and\_, or\_, not\_ тощо. Наприклад:

from sqlalchemy import and\_, or\_, not\_

...

stmt = select(User).where(and\_(User.fullname.like("%ha%"), User.fullname != 'Shaun Beck'))

result = session.execute(stmt)

for user in result.scalars():

print(user.id, user.fullname)

Виведення:

3 Kathrin Reinhardt

Ми також можемо використовувати метод where() кілька разів. Це аналог додавання декількох критеріїв в одну умову where, за допомогою оператора and\_. Наприклад:

stmt = select(User).where(User.fullname.like("%ha%")).where(User.fullname != 'Shaun Beck')

result = session.execute(stmt)

for user in result.scalars().all():

print(user.id, user.fullname)

Виведення:

3 Kathrin Reinhardt

**Сортування записів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/new#%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%96%D0%B2)Для сортування записів за певним критерієм ми можемо використовувати метод order\_by() об'єкта Select. Наприклад:

stmt = select(User).order\_by(User.fullname)

result = session.execute(stmt)

for user in result.scalars():

print(user.id, user.fullname)

Виведення:

1 Crystal Najera

3 Kathrin Reinhardt

2 Shaun Beck

Ми можемо використовувати метод desc() або asc() для вказівки напрямку сортування за зменшенням або зростанням відповідно.

Наприклад:

from sqlalchemy import desc

...

stmt = select(User).order\_by(desc(User.fullname))

result = session.execute(stmt)

for user in result.scalars():

print(user.id, user.fullname)

Виведення:

2 Shaun Beck

3 Kathrin Reinhardt

1 Crystal Najera

Ми також можемо використовувати метод order\_by() кілька разів для додавання декількох критеріїв сортування в один порядок.

**Запити з JOIN**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-07/module-07-01/new#%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D0%B8-%D0%B7-join)Для виконання складних запитів з використанням join, group by або інших операцій SQL, ми також можемо використовувати оператор select().

Наприклад, у нас є ще одна модель Post, яка пов'язана з моделлю User через зовнішній ключ user\_id, і ми хочемо отримати кількість постів для кожного користувача. Для цього, ми можемо написати:

from sqlalchemy import func # импортируем функцию func для агрегатных функций SQL

...

stmt = select(User.id, User.fullname)

result = session.execute(stmt)

users = []

for row in result:

users.append(row)

for user in users: # створюємо запис Post для кожного користувача

post = Post(title=f'Title {user[1]}', body=f'Body post user {user[1]}', user\_id=user[0])

session.add(post)

session.commit()

stmt = (

select(User.fullname, func.count(Post.id)) # створюємо об'єкт select із вибіркою імені користувача та кількості постів

.join(Post) # робимо join з моделлю Post за зовнішнім ключем user\_id

.group\_by(User.fullname) # групуємо результати за ім'ям користувача

)

results = session.execute(stmt).all() # виконуємо запит і отримуємо список кортежів

for name, count in results: # перебираємо результати

print(f"{name} has {count} posts") # виводимо ім'я користувача та кількість постів

Виведення:

Crystal Najera has 1 posts

Kathrin Reinhardt has 1 posts

Shaun Beck has 1 posts

У цьому розділі ми розібрали основи використання нового інтерфейсу запитів API SQLAlchemy версії 2.0. Для більш докладної інформації ви можете звернутися до офіційної документації SQLAlchemy або до керівництва з [міграції](https://docs.sqlalchemy.org/en/20/changelog/migration_20.html) (<https://docs.sqlalchemy.org/en/20/changelog/migration_20.html>) на SQLAlchemy 2.0.

ПРИМІТКА

Далі в конспекті ми будемо використовувати механізм сесій session.query. Для простих запитів session.query() простіший та інтуїтивно зрозумілий, особливо для тих, хто тільки починає працювати з SQLAlchemy. А також з метою підтримки наявного коду конспекту.