**Модуль 7: ORM SQLAlchemy**

**Заняття 2: Міграції разом з Alembic. Підготовка проекту**

У цьому практичному прикладі ми створимо базу даних (БД) **Sqlite** з використанням ORM пакета [**sqlalchemy**](https://www.sqlalchemy.org/) (https://www.sqlalchemy.org/), а також використаємо пакет [**alembic**](https://alembic.sqlalchemy.org/en/latest/) (https://alembic.sqlalchemy.org/en/latest/) для створення та контролю міграцій.

Наше завдання отримати невеликий "TODO" проект, у якому будуть завдання — таблиця notes, у завдань будуть підпорядковані пункти з описом та контролем виконання — таблиця records. Також ми реалізуємо можливість додати до завдань теги — таблиця tags.

Створюємо папку проекту.

mkdir EX-09

cd EX-09

Створюємо віртуальне оточення poetry та активуємо його. Встановлюємо пакет **alembic** та **sqlalchemy**

poetry add alembic sqlalchemy

Після встановлення, в корені проекту необхідно ініціалізувати оточення **alembic** за допомогою команди:

alembic init alembic

Цей виклик створить папку з ім'ям alembic всередині вашого проекту, а також файл alembic.ini з налаштуваннями.

**Налаштування пакета alembic**

Починаємо налаштовувати наше оточення. Потрібно знайти у файлі alembic.ini рядок

sqlalchemy.url = driver://user:pass@localhost/dbname

І замінити його на шлях до нашої БД. Ми будемо використовувати файл mynotes.db в корені проекту.

sqlalchemy.url = sqlite:///mynotes.db

Наступним кроком буде зміна файлу env.py. Тому що ми хочемо використовувати автогенерацію SQL скриптів в міграціях alembic — нам необхідно повідомити про це оточення у файлі env.py, який розташований у папці alembic.

Відкриємо його і насамперед імпортуємо нашу декларативну базу з файлу model.py. Цей файл ми створимо трохи пізніше.

alembic/env.py

from logging.config import fileConfig

from sqlalchemy import engine\_from\_config

from sqlalchemy import pool

from alembic import context

from models import Base

Далі нам необхідно знайти рядок:

target\_metadata = None

і замість None, вказати наші метадані:

target\_metadata = Base.metadata

**Створення моделей**

Тепер створимо моделі у файлі models.py для нашого завдання. У ньому ми опишемо ORM класи з використанням sqlalchemy.

**models.py**

from datetime import datetime

from sqlalchemy import Column, Integer, String, Boolean

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import relationship

from sqlalchemy.sql.schema import ForeignKey, Table

from sqlalchemy.sql.sqltypes import DateTime

Base = declarative\_base()

# таблиця для зв'язку many-to-many між таблицями notes та tags

note\_m2m\_tag = Table(

"note\_m2m\_tag",

Base.metadata,

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

Column("note", Integer, ForeignKey("notes.id", ondelete="CASCADE")),

Column("tag", Integer, ForeignKey("tags.id", ondelete="CASCADE")),

)

# Таблиця notes, де зберігатимуться назви завдань

class Note(Base):

\_\_tablename\_\_ = "notes"

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

name = Column(String(50), nullable=False)

created = Column(DateTime, default=datetime.now())

records = relationship("Record", cascade="all, delete", backref="note")

tags = relationship("Tag", secondary=note\_m2m\_tag, backref="notes", passive\_deletes=True)

# Таблиця records, де зберігатимуться записи справ для конкретного завдання з таблиці notes - зв'язок one-to-many, поле note\_id

class Record(Base):

\_\_tablename\_\_ = "records"

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

description = Column(String(150), nullable=False)

done = Column(Boolean, default=False)

note\_id = Column(Integer, ForeignKey(Note.id, ondelete="CASCADE"))

# Таблиця tags, де зберігається набір тегів для списку справ.

class Tag(Base):

\_\_tablename\_\_ = "tags"

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

name = Column(String(25), nullable=False, unique=True)

Тут ми якраз використовуємо таблицю note\_m2m\_tag, щоб створити відношення *багато до багатьох* між таблицями notes та tags.

Зверніть увагу, що вказавши для таблиці records параметр ondelete="CASCADE",

note\_id = Column(Integer, ForeignKey(Note.id, ondelete="CASCADE"))

Ми говоримо про те, що при видаленні запису з таблиці notes ми видалимо автоматично всі пов'язані записи у таблиці records.

Аргумент ключового слова backref в конструкції relations дозволяє автоматично генерувати нове відношення, яке буде автоматично додано до зіставлення ORM для пов'язаного класу. Іншими словами, в таблиці Note ми створили відносини relationship для того, щоб отримувати записи records з таблиці Record. За допомогою таблиці secondary=note\_m2m\_tag здійснили зв'язок *багато до багатьох*, щоб отримувати доступ до тегів завдання.

INFO

Параметр passive\_deletes вказує на поведінку завантаження під час операцій видалення. Значення True вказує, що вивантажені дочірні елементи не повинні завантажуватись під час операції видалення батьківського елемента. Зазвичай при видаленні батьківського елемента завантажуються всі дочірні елементи, щоб їх можна було або помітити як видалені, або встановити їхній зовнішній ключ до батьківського елемента, що дорівнює NULL. Позначка цього прапора як True зазвичай має на увазі наявність правила ON DELETE <CASCADE|SET NULL>, яке оброблятиме оновлення/видалення дочірніх рядків на стороні бази даних.

**Створення міграцій**

Створивши файл із моделями, можна виконати міграцію для перевірки загальної працездатності застосунку. Для цього потрібно виконати наступну консольну команду в корені проекту.

alembic revision --autogenerate -m 'Init'

Приблизно, у разі успішного виконання, виведення в консолі повинно бути наступним:

PS ...> alembic revision --autogenerate -m 'Init'

INFO [alembic.runtime.migration] Context impl SQLiteImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.autogenerate.compare] Detected added table 'notes'

INFO [alembic.autogenerate.compare] Detected added table 'tags'

INFO [alembic.autogenerate.compare] Detected added table 'note\_m2m\_tag'

INFO [alembic.autogenerate.compare] Detected added table 'records'

Generating ...\alembic\versions\78d017764932\_init.py ... done

Після виконання команди, в корені проекту повинен з'явитися файл SQLite mynotes.db. Це наша БД, в якій поки що буде порожня таблиця alembic\_version з версіями міграцій **alembic**. Також у папці alembic\versions з'явиться .py файл зі скриптом міграції. Він автоматично згенерований і у вашому випадку повинен відрізнятися за Revision ID

**alembic/versions/78d017764932\_init.py**

"""Init

Revision ID: 78d017764932

Revises:

Create Date: 2022-02-23 15:02:47.133199

"""

from alembic import op

import sqlalchemy as sa

# revision identifiers, used by Alembic.

revision = '78d017764932'

down\_revision = None

branch\_labels = None

depends\_on = None

def upgrade():

# ### commands auto generated by Alembic - please adjust! ###

op.create\_table('notes',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('name', sa.String(length=50), nullable=False),

sa.Column('created', sa.DateTime(), nullable=True),

sa.PrimaryKeyConstraint('id')

)

op.create\_table('tags',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('name', sa.String(length=25), nullable=False),

sa.PrimaryKeyConstraint('id'),

sa.UniqueConstraint('name')

)

op.create\_table('note\_m2m\_tag',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('note', sa.Integer(), nullable=True),

sa.Column('tag', sa.Integer(), nullable=True),

sa.ForeignKeyConstraint(['note'], ['notes.id'], ondelete='CASCADE'),

sa.ForeignKeyConstraint(['tag'], ['tags.id'], ondelete='CASCADE'),

sa.PrimaryKeyConstraint('id')

)

op.create\_table('records',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('description', sa.String(length=150), nullable=False),

sa.Column('done', sa.Boolean(), nullable=True),

sa.Column('note\_id', sa.Integer(), nullable=True),

sa.ForeignKeyConstraint(['note\_id'], ['notes.id'], ondelete='CASCADE'),

sa.PrimaryKeyConstraint('id')

)

# ### end Alembic commands ###

def downgrade():

# ### commands auto generated by Alembic - please adjust! ###

op.drop\_table('records')

op.drop\_table('note\_m2m\_tag')

op.drop\_table('tags')

op.drop\_table('notes')

# ### end Alembic commands ###

Для застосування міграції з отриманого файлу та створення таблиць наших моделей у БД виконаємо команду:

alembic upgrade head

Виведення:

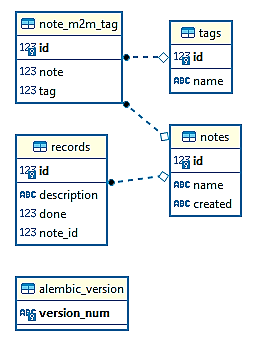
...> alembic upgrade head

INFO [alembic.runtime.migration] Context impl SQLiteImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.runtime.migration] Running upgrade -> 78d017764932, Init

Після виконання, міграція створить таблиці в нашій БД. І ми отримаємо наступну ER діаграму.



На цьому підготовчий етап завершений і потрібно заповнити нашу БД даними.

**Заповнюємо БД даними**

Створимо в корені проекту файл підключення до БД connect\_db.py:

**connect\_db.py**

from sqlalchemy.engine import create\_engine

from sqlalchemy.orm import sessionmaker

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

Session = sessionmaker(bind=engine)

session = Session()

Потім створюємо у корені проекту файл seeds.py з наступним вмістом:

**seeds.py**

from connect\_db import session

from models import Note, Record, Tag

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

tag1 = Tag(name="groceries")

tag2 = Tag(name="food")

note = Note(name="Go to the store")

note.tags = [tag1, tag2]

rec1 = Record(description="Buy bread", note=note)

rec2 = Record(description="Buy sausage 0.5 kg", note=note)

rec3 = Record(description="Buy tomatoes 1 kg", note=note)

session.add(note)

session.commit()

Виконаємо його. Після виконання у БД ми отримаємо завдання "Go to the store", яке містить три записи: "Buy bread", "Buy sausage 0.5 кг" та "Buy tomatoes 1 кг". Так само для завдання "Збігати в магазин" ми присвоїли два теги "groceries" та "food"

**Запускаємо приклад**

Перевіримо, що дані записалися в БД наступним запитом до неї.

SELECT nt.id, nt.name, rc.description, rc.done, t.name as tag

FROM notes AS nt

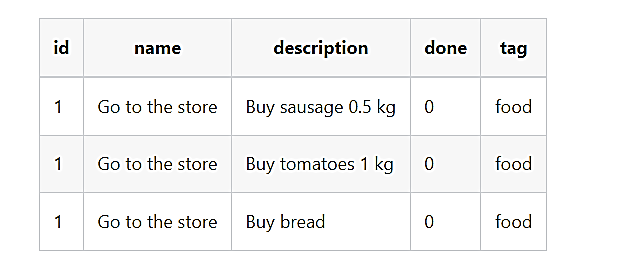
LEFT JOIN records AS rc ON rc.note\_id = nt.id

LEFT JOIN note\_m2m\_tag AS nmmt ON nmmt.note = nt.id

LEFT JOIN tags AS t ON t.id = nmmt.tag

WHERE t.name = 'food'

Ми повинні отримати результат у вигляді такої таблиці:



Створимо файл main.py і помістимо туди код виконання запиту до БД, але вже за допомогою SQLAlchemy.

**main.py**

from sqlalchemy import select

from connect\_db import session

from models import Note, Tag, Record, note\_m2m\_tag

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

q = session.execute(

select(Note.id, Note.name, Record.description, Record.done, Tag.name.label('tag'))

.join(Record)

.join(note\_m2m\_tag)

.join(Tag).filter(Tag.name == 'food')

).mappings().all()

print(q)

Виведення (метод mappings надає нам виведення у вигляді готового словника):

[

{'id': 1, 'name': 'Go to the store', 'description': 'Buy bread', 'done': False, 'tag': 'food'},

{'id': 1, 'name': 'Go to the store', 'description': 'Buy sausage 0.5 kg', 'done': False, 'tag': 'food'},

{'id': 1, 'name': 'Go to the store', 'description': 'Buy tomatoes 1 kg', 'done': False, 'tag': 'food'}

]

Як бачимо, результати збігаються. Наше завдання повністю вирішене. Успіхів у виконанні домашнього завдання!