А. И. ЕЛИСЕЕВ, Д. В. ПОЛЯКОВ

РАБОТА С БАЗАМИ ДАННЫХ НА ЯЗЫКЕ Python С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ SQLAlchemy 2.0

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет»

А. И. ЕЛИСЕЕВ, Д. В. ПОЛЯКОВ

РАБОТА С БАЗАМИ ДАННЫХ НА ЯЗЫКЕ Python С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ SQLAlchemy 2.0

Утверждено Ученым советом университета
в качестве учебного пособия для студентов 2, 3 курсов,
обучающихся по направлению подготовки 09.03.02
«Информационные системы и технологии»

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

УДК 004(075.8)
ББК *з*973-018.3я73
Е51

Рецензенты:

Кандидат технических наук, доцент, доцент Института новых технологий и искусственного интеллекта ФГБОУ ВО «ТГУ им. Г. Р. Державина»
И. А. Зауголков

Кандидат технических наук, доцент кафедры «Мехатроника и технологические измерения» ФГБОУ ВО «ТГТУ»
А. С. Егоров

Елисеев, А. И.

Е51 Работа с базами данных на языке Python с использованием библиотеки SQLAlchemy 2.0 [Электронный ресурс] : учебное пособие / А. И. Елисеев, Д. В. Поляков. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2024. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 1,2 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2820-4

Представляет собой руководство по использованию библиотеки SQLAlchemy 2.0, охватывающее основные концепции и практические примеры для эффективного управления базами данных в приложениях на Python.

Предназначено для студентов 2, 3 курсов направления подготовки 09.03.02 «Информационные системы и технологии».

УДК 004(075.8)
ББК *з*973-018.3я73

ISBN 978-5-8265-2820-4

ВВЕДЕНИЕ

Приложение, написанное на языке программирования Python, может взаимодействовать с различными системами управления базами данных (СУБД), такими как SQLite, MySQL, PostgreSQL и др. При работе с каждой из этих СУБД необходимо формировать запросы на языке SQL, что требует знания диалекта каждой системы. Синтаксис SQL-запросов может частично отличаться в зависимости от используемой СУБД, что, в свою очередь, может создать сложности при переходе с одной базы данных на другую. В таких случаях потребуется внести значительные изменения в код, чтобы адаптировать его к новому диалекту SQL и настройкам подключения.

Для решения этих проблем используют специальные ORM-библиотеки, (Object Relational Mapper/Mapping). Эти библиотеки позволяют абстрагироваться от конкретной базы данных и работать с данными как с объектами стандартных классов Python. Это значительно упрощает процесс разработки и делает код более универсальным и переносимым.

ПОДГОТОВКА К РАБОТЕ С SQLAlchemy

Установка базы данных SQLite

SQLite – это библиотека, написанная на языке Си, реализующая небольшой, быстрый и самодостаточный движок реляционной базы данных. Эта база данных примечательная тем, что для ее работы не требуется отдельный серверный процесс.

Библиотека SQLite поставляется в комплекте с интерпретатором Python, поэтому поддержка этой базы данных доступна для использования в Python

и SQLAlchemy без установки дополнительного программного обеспечения или выполнения каких-либо настроек.

Если необходим инструмент, который можно использовать для проверки и управления базами данных SQLite вне Python и SQLAlchemy, можно загрузить оболочку командной строки sqlite3 для нужной операционной системы с официального сайта SQLite.

Установка базы данных MySQL

MySQL – это реляционная база данных с открытым исходным кодом, принадлежащая компании Oracle. В отличие от SQLite, эта база данных включает в себя серверный и клиентский компоненты, и оба этих компонента необходимо установить.

Сервер MySQL

Если есть доступ к работающему серверу MySQL, то достаточно создать новую базу данных, пользователя и перейти к настройке клиента. Инструкции в этом разделе демонстрируют, как установить сервер вместе с популярным приложением для управления MySQL – phpMyAdmin.

Если сервер MySQL не установлен, самый простой способ запустить его – использовать Docker и Docker Compose. Скопируйте следующие директивы в файл с именем docker-compose.yml в директории проекта:

```
version: '3'

services:
  db:
    image: mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: password
```

```
    ports:
      - "3306:3306"
    volumes:
      - db-data:/var/lib/mysql
  admin:
    image: phpmyadmin
    restart: always
    environment:
      - PMA_ARBITRARY=1
    ports:
      - 8080:80
volumes:
  db-data:
```

Файл конфигурации Docker Compose запускает службы db и admin. Служба db в свою очередь запускает сервер MySQL на порту 3306. Служба admin запускает phpMyAdmin на порту 8080. Хранилище базы данных настроено в виде отдельного тома с именем db-data. Благодаря этому можно будет обновить контейнер базы данных без потери данных.

В строке MYSQL_ROOT_PASSWORD задается пароль администратора для сервера MySQL. Эту строку необходимо предварительно отредактировать.

Команда для запуска сервера MySQL:

```
docker-compose up -d
```

При первом запуске этой команды Docker будет загружать образы контейнеров MySQL и phpMyAdmin из репозитория Docker Hub. После загрузки образов запуск контейнеров займет несколько секунд, и после этого MySQL будет готов к работе.

Для доступа к инструменту управления базами данных phpMyAdmin необходимо перейти в браузере по адресу http://localhost:8080. Чтобы войти в систему, необходимо ввести следующие учетные данные:

– Server: db;

– Username: root;

– Password: пароль, который был определен в файле docker-compose.yml.

После того как вы войдете в интерфейс phpMyAdmin, перейдите на вкладку Databases. В самом верху вы увидите раздел Create Database. Введите имя новой базы данных, например db, и нажмите кнопку Create.

Хорошей практикой при создании новой базы данных является также определение пользователя, специально назначенного для нее. Использование пользователя root для повседневных операций слишком рискованно, поскольку эта учетная запись должна использоваться только для важных административных задач.

Перейдите на вкладку Privileges новой базы данных. В нижней части страницы есть раздел New со ссылкой Add user account. Щелкните ее, чтобы создать нового пользователя.

Выберите любое имя пользователя, например: user. Оставьте для хоста значение «%», затем введите пароль для нового пользователя.

Убедитесь, что опция Grant all privileges on database db включена, а затем прокрутите страницу до самого низа и нажмите кнопку Go, чтобы создать пользователя. У этого пользователя будет полный доступ к базе данных, но он не сможет получить доступ к другим базам данных или создавать их, что является хорошим принципом обеспечения безопасности.

С этого момента вы можете входить в phpMyAdmin, используя только что созданного пользователя.

Чтобы остановить сервер MySQL, необходимо из директории, в которой находится файл docker-compose.yml, выполнить следующую команду:

```
docker-compose down
```

Чтобы снова запустить сервер, повторите команду up:

```
docker-compose up -d
```

Остановка и перезапуск сервера не приводят к потере данных.

Клиент MySQL

Для доступа к базе данных MySQL необходимо установить Python-клиент, который иногда называют также драйвером. Существует несколько драйверов MySQL для Python, которые могут быть использованы.

Используем драйвер pymysql, который необходимо установить в виртуальную среду Python следующим образом:

```
poetry add pymysql cryptography
```

Установленный пакет cryptography является дополнительной зависимостью pymysql, необходимой для выполнения аутентификации в базе данных MySQL.

Настройки подключения для базы данных будут следующими:

- имя хоста: localhost;
- порт: 3306;
- база данных: db;
- имя пользователя: user;
- пароль: пароль, который вы выбрали для пользователя;
- драйвер Python: pymysql.

Установка базы данных PostgreSQL

PostgreSQL (часто сокращенно Postgres) – еще одна крупная система управления реляционными базами данных с открытым исходным кодом,

похожая на MySQL в том смысле, что для нее также требуется отдельный сервер и клиент.

Сервер PostgreSQL

Если есть доступ к работающему серверу PostgreSQL, то достаточно создать новую базу данных, пользователя и перейти к настройке клиента. Инструкции в этом разделе демонстрируют, как установить сервер вместе с популярным приложением для управления PostgreSQL – PgAdmin.

Скопируйте следующий файл конфигурации Docker Compose в файл с именем docker-compose.yml в каталоге проекта:

```
version: '3'

services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    volumes:
      - db-data:/var/lib/postgresql/data
  admin:
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: password
    ports:
      - 8080:80
    volumes:
      - admin-data:/var/lib/pgadmin
volumes:
  db-data:
  admin-data:
```

Этот файл конфигурации определяет службу db с сервером PostgreSQL, работающим на порту 5432, и вторую службу admin, запускающую pgAdmin на порту 8080. Обеим службам требуется хранилище, поэтому для них также создаются два тома. Использование отдельных томов для хранения позволяет сохранять данные при остановке и перезапуске контейнеров.

В приведенной выше конфигурации есть три строки, которые необходимо отредактировать:

– Измените POSTGRES_PASSWORD на пароль администратора PostgreSQL.

– Измените PGADMIN_DEFAULT_EMAIL на собственный адрес электронной почты (используется только для входа в систему).

– Измените PGADMIN_DEFAULT_PASSWORD на пароль администратора pgAdmin.

После того как файл docker-compose.yml готов, можно запустить сервисы следующей командой:

```
docker-compose up -d
```

При первом запуске этой команды Docker должен будет загрузить образы для PostgreSQL и pgAdmin, так что это может занять некоторое время. После загрузки образов запуск служб займет всего несколько секунд.

После выполнения вышеуказанной команды подключитесь к pgAdmin, набрав http://localhost:8080 в адресной строке браузера.

Войти в интерфейс pgAdmin можно с помощью электронной почты и пароля, которые были определены для параметров PGADMIN_DEFAULT_EMAIL и PGADMIN_DEFAULT_PASSWORD в файле конфигурации docker-compose.yml.

Первая задача – задать в pgAdmin сервер PostgreSQL. Для этого нажмите на иконку Add New Server. На вкладке General введите имя сервера, например db, в поле Name.

Затем на вкладке Connection установите для параметра Host name значение db, которое является именем службы PostgreSQL, определенным в конфигурации Docker Compose. Оставьте настройки Port и Maintenance Database со значениями по умолчанию. Измените Username на postgres, а в поле Password задайте пароль, который был определен в параметре POSTGRES_PASSWORD. Вы можете включить опцию Save password?, если не хотите повторно вводить пароль в будущем.

После нажатия кнопки Save pgAdmin добавит сервер в левую боковую панель и начнет показывать статистику его использования.

Следующим шагом является создание новой базы данных, которую можно будет использовать для выполнения примеров, описанных в пособии. Как и в случае с MySQL, хорошей практикой является создание отдельного пользователя для каждой базы данных. Чтобы создать пользователя, щелкните правой кнопкой мыши на имени базы данных в боковой панели и выберите Create, а затем Login/Group Role... .

На вкладке General введите имя нового пользователя, например user. Переключитесь на вкладку Definition и введите пароль для пользователя. Затем переключитесь на вкладку Privileges. У этого пользователя должны быть включены опции Can login? и Inherit rights from the parent roles? . Для повышения безопасности лучше отключить все остальные привилегии.

Нажмите кнопку Save, чтобы добавить пользователя.

Затем снова щелкните правой кнопкой мыши на базе данных слева, выберите Create, а затем Database... .

Дайте новой базе данных имя, например, db. Владельцем базы данных должен быть пользователь postgres, то есть администратор.

Нажмите Save, чтобы создать новую базу данных.

Следующим шагом будет настройка привилегий пользователя user, чтобы он имел полный доступ к новой базе данных. В левой боковой панели разверните древовидное представление, начиная с сервера db, далее Databases, раскройте ветку db, Schemas и, наконец, public.

Щелкните правой кнопкой мыши на публичной схеме и выберите Properties... . Затем выберите вкладку Security. Нажмите «+» в таблице Privileges, чтобы добавить новую запись. В разделе Grantee выберите пользователя user. В столбце Privileges отметьте опцию ALL, чтобы предоставить пользователю полный доступ к схеме. Нажмите Save, чтобы сохранить новые привилегии.

Чтобы остановить сервер PostgreSQL, необходимо из директории, в которой находится файл docker-compose.yml, выполнить следующую команду:

```
docker-compose down
```

Чтобы снова запустить сервер, повторите команду up:

```
docker-compose up -d
```

Остановка и перезапуск сервера не приводят к потере данных.

Клиент PostgreSQL

Последний шаг – установка драйвера PostgreSQL для Python. SQLAlchemy поддерживает несколько драйверов PostgreSQL. Очень популярным драйвером является psycopg2, который можно установить с помощью этой команды:

```
poetry add psycopg2-binary
```

Чтобы подключиться к базе данных из Python, необходимо определить детали подключения. Настройки подключения для базы данных будут следующими:

– имя хоста: localhost;
– порт: 5432;
– база данных: db;
– имя пользователя: user;
– пароль: пароль, который вы выбрали для пользователя;
– Python-драйвер: psycopg2.

URL-адреса подключения к базе данных

При использовании SQLAlchemy база данных, к которой нужно подключиться, представлена URL, имеющим следующую структуру:

```
{dialect}{+driver}://{username}:{password}@{hostname}:{port}/{
database}
```

URL для MySQL и PostgreSQL строятся с использованием mysql или postgresql в качестве диалекта соответственно, плюс данные подключения к базе данных.

В следующих примерах предполагается, что пароль пользователя – password:

```
# MySQL с pymysql

url = 'mysql+pymysql://user:password@localhost:3306/db'

# PostgreSQL с psycopg2

url = 'postgresql+psycopg2://user:password@localhost:5432/db'
```

URL-адреса баз данных SQLite немного отличаются. Для этой базы данных название диалекта – sqlite, а драйвер можно не указывать. Имя поль-

зователя, пароль, имя хоста и порт также опускаются, поскольку они не имеют никакого значения. Наконец, вместо имени базы данных указывается путь к файлу базы данных.

В следующих примерах показаны некоторые возможные URL-адреса базы данных SQLite с именем db.sqlite:

```
# файл базы данных в текущем каталоге
url = 'sqlite:///db.sqlite'

# файл базы данных в каталоге /home/user/app
url = 'sqlite:////home/user/app/db.sqlite'

# файл базы данных в каталоге C:\users\user\app (Microsoft
Windows)
url = 'sqlite:///c:\\users\\user\\app\\db.sqlite'
```

В первом примере для файла базы данных используется относительное местоположение (текущий каталог). В этом URL первые два слэша являются частью префикса sqlite://, а третий слэш – это тот, что идет после имени пользователя, пароля, имени хоста и порта, только в данном случае нужно включить только один разделитель.

Во втором примере после диалекта и драйвера указаны четыре прямых слэша. Первые три слэша имеют то же назначение, что и в первом примере. Четвертый слэш – это начало абсолютного пути к файлу базы данных SQLite, который в данном примере имеет вид /home/user/app/db.sqlite.

Третий и последний пример показывает, как можно указать абсолютный путь при использовании операционной системы Microsoft Windows. Абсолютный путь начинается с буквы дискового накопителя, и в нем используются обратные слэши в качестве разделителей компонентов пути. В строках Python символ обратного слэши нужно экранировать, используя второй обратный слэш.

База данных SQLite предоставляет одну дополнительную возможность: in-memory вариант базы данных. Эта особенность может быть полезна для временных баз данных, например, в модульных тестах. При использовании базы данных in-memory все данные хранятся в памяти процесса без записи на диск.

Во всех примерах, приведенных в пособии, URL базы данных будет задан извне через переменную окружения с именем DATABASE_URL. Чтобы не устанавливать эту переменную в каждом сеансе работы с оболочкой, создайте файл с именем .env, откройте его в текстовом редакторе и задайте в нем URL базы данных, который необходимо использовать, например следующим образом:

```
DATABASE_URL = postgresql://user:password@localhost:5432/db
```

Приведенный выше пример настраивает базу данных PostgreSQL.

Пакет python-dotenv позволяет приложению считывать переменные из файла .env. Установите его:

```
poetry add python-dotenv
```

Ниже приведен пример того, как считать переменную DATABASE_URL. Скопируйте этот код в файл с именем db.py в директории проекта.

```
import os
from dotenv import load_dotenv

load_dotenv()
print('Database URL:', os.environ['DATABASE_URL'])
```

БИБЛИОТЕКА SQLAlchemy

Одной из самых популярных ORM-библиотек для языка Python является SQLAlchemy. Эта библиотека предоставляет мощные инструменты для работы с базами данных, позволяя разработчикам писать код, который будет работать с различными СУБД. Однако при использовании SQLAlchemy важно учитывать версию Python, так как SQLAlchemy 2.0 поддерживает только версии Python 3.7 и выше.

Команда для установки выглядит следующим образом:

```
poetry add sqlalchemy
```

Библиотека SQLAlchemy разделена на два модуля: Core и ORM.

Модуль Core содержит логику интеграции базы данных для всех поддерживаемых диалектов базы данных, набор классов для описания таблиц базы данных и довольно сложную систему для генерации операторов SQL с использованием языковых конструкций Python.

Модуль ORM вводит уровень абстракции между приложением Python и базой данных, который позволяет автоматически генерировать операции над базой данных из действий, выполняемых над объектами Python.

В приложении может выбрать использование исключительно SQLAlchemy Core или можно комбинировать элементы из Core и ORM.

Движок базы данных

SQLAlchemy использует объекты engine для управления соединениями с базой данных, как для Core, так и для ORM-приложений.

Функция create_engine() создает движок, принимая URL базы данных. URL базы данных содержит необходимую информацию для подключения,

включая тип базы данных, имя пользователя, пароль, адрес сервера и имя самой базы данных.

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine

load_dotenv()
engine = create_engine(os.environ['DATABASE_URL'])
```

URL базы данных может быть определен в переменной окружения:

```
#.env
DATABASE_URL = postgresql://user:password@localhost:5432/db
```

Функции create_engine() можно передать дополнительные аргументы-ключи:

– echo=True: включает запись в журнал каждого SQL-оператора, отправленного базе данных. Опция полезна при отладке.

– pool_size=<N>: определяет размер пула соединений, который поддерживает SQLAlchemy (по умолчанию – до 5 одновременных соединений).

– max_overflow=<N>: максимальное количество соединений сверх размера пула, которое может быть создано во время увеличения нагрузки (по умолчанию 10).

– future=True: использование API версии 2.0.

Модели

При использовании модуля ORM таблицы базы данных определяются в приложении как классы Python. Нужно создать родительский класс для всех классов Python, в котором можно задать параметры, общие для всех таблиц.

Этот родительский класс (в терминах SQLAlchemy – декларативный базовый класс), часто называют Base или Model. Коллекция подклассов класса Base представляет структуру или схему базы данных. Подклассы в приложении обычно называется «моделями». Класс Base должен наследоваться от класса DeclarativeBase.

Пример:

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase): pass

load_dotenv()
engine = create_engine(os.environ['DATABASE_URL'])
```

Пример реализация модели для таблицы с фильмами:

```
from sqlalchemy import String, Date
from sqlalchemy.orm import Mapped, mapped_column
from db import Base
import datetime

class Movie(Base):
    __tablename__ = "movies"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director: Mapped[str] = mapped_column(String(64))
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    genre: Mapped[str] = mapped_column(String(32))
```

```
    rating: Mapped[float] = mapped_column()

    def __repr__(self):
        return f'Movie({self.id}, "{self.title}")'
```

Атрибут __tablename__ определяет имя таблицы базы данных, которую представляет класс. Очень распространенным соглашением об именовании таблиц баз данных является использование формы множественного числа сущности в нижнем регистре.

Атрибуты, определенные в классе, представляют столбцы таблицы.

Для определения каждого столбца используется объявление типа Mapped[type], причем type – это тип Python, например int, str или datetime.

Если столбцу необходимо задать дополнительные параметры, используется конструктор mapped_column(), в котором определяются эти параметры. Например, для столбцов типа str добавляется максимальная длина с помощью дополнительной опции String().

В модели Movie используется опция для определения столбца id в качестве первичного ключа primary_key. Это означает, что значения в этом столбце должны однозначно идентифицировать каждый фильм, хранящийся в таблице. Без дополнительных настроек поле primary_key определяет целочисленные столбцы первичных ключей с автоинкрементом, начиная с 1.

Для создания экземпляра класса модели используется стандартный конструктор класса, принимающий значения атрибутов модели в качестве аргументов.

Пример:

```
movie = Movie(title="Blade Runner", director="Ridley Scott")
```

В примере все атрибуты movie, кроме title и director, будут установлены в None, поскольку им не было присвоено значение.

Несмотря на то, что этот объект является экземпляром модели, в данный момент это обычный объект Python, который не хранится и не связан ни с какой базой данных.

Метаданные

SQLAlchemy хранит определения всех таблиц, составляющих базу данных, в объекте класса MetaData. Для удобства SQLAlchemy инициализирует декларативный базовый класс с атрибутом metadata, который имеет объект MetaData по умолчанию. Для класса Base экземпляр метаданных доступен в виде Base.metadata. При определении класса модели, такой как Movie, SQLAlchemy создает соответствующее определение таблицы в этом атрибуте.

У объекта MetaData есть метод create_all(), который позволяет создавать все таблицы базы данных, связанные с определенными моделями.

Пример использования:

```
Base.metadata.create_all(engine)
```

Метод create_all() выполнит SQL-запросы к базе данных, представленной движком, для создания таблиц базы данных, на которые ссылаются модели. Этот вызов создаст таблицу movies, которая определяется моделью Movie. Важным ограничением метода create_all() является то, что он создает только те таблицы, которые еще не существуют в базе данных. При изменении класса модели этот метод нельзя использовать для переноса изменений в соответствующую таблицу базы данных.

Обходной путь, который можно использовать для модификации существующей таблицы, – это удаление старой версии таблицы из базы данных

перед повторным вызовом create_all(). В объекте MetaData есть метод drop_all(), который позволяет удалить все таблицы из базы данных.

Следующий пример обновляет все таблицы до их последних определений:

```
Base.metadata.drop_all(engine)
Base.metadata.create_all(engine)
```

Обновление базы данных таким способом целесообразно только для проведения небольших тестов или при создании прототипов, поскольку вызов drop_all() удаляет не только таблицы, но и все хранящиеся в них данные.

Рекомендуемое решение – использовать пакет Alembic для управления обновлениями базы данных гораздо более эффективным способом с помощью скриптов миграции.

Сессии

Объект сессии session хранит список новых, прочитанных, измененных и удаленных экземпляров модели. Изменения, накапливающиеся в сессии, передаются в базу данных в контексте транзакции базы данных при операции flush. Flush – операция, которая в большинстве случаев автоматически выполняется SQLAlchemy, когда это необходимо. Операция flush записывает изменения в базу данных, но сохраняет транзакцию базы данных открытой.

Когда над сессией выполняется операция commit, соответствующая транзакция базы данных также коммитится, в результате чего все изменения будут окончательно записаны в базу данных.

Транзакции базы данных – одно из важнейших преимуществ реляционных баз данных, предназначенных для поддержания целостности данных. Изменения, которые фиксируются в рамках транзакции, записываются как

одна атомарная операция, поэтому ошибки или неожиданные прерывания выполнения запросов никогда не приведут к частичной или неполной записи данных.

Если во время использования активной сессии произойдет ошибка или сбой, операция отката (rollback) сессии приведет к откату транзакции, и все изменения, сделанные до этого момента в этой сессии, будут отменены.

Пример:

```
from sqlalchemy.orm import Session

with Session(engine) as session:
    try:
        session.add(movie)
        session.commit()
    except:
        session.rollback()
        raise
    print(movie)
```

Лучший способ управления сеансом базы данных – это создание его как менеджера контекста. Менеджер контекста гарантирует, что сессия будет правильно закрыта и утилизирована в конце использования. Объекты сессий предназначены для накопления изменений до тех пор, пока они не будут зафиксированы (commit) или отменены (rollback). Метод add() используется для вставки нового объекта в сессию. Блок try/except гарантирует, что сессия всегда будет зафиксирована или произойдет откат.

Как упоминалось ранее, SQLAlchemy настраивает целочисленные столбцы первичных ключей на автоинкремент по умолчанию. Когда сессия будет очищена, что обычно происходит во время вызова commit(), база данных присвоит следующее доступное число атрибуту id нового элемента, или 1, если это первая добавленная запись. Любые другие атрибуты объекта

модели, которые не были определены, будут записаны в базу данных со значением NULL.

SQLAlchemy предоставляет более лаконичные способы работы с сессиями. В типовом веб-приложении будет много участков кода, в которых нужно создавать сессии, и может быть неудобно постоянно передавать в них движок и другие параметры. Функция фабрики sessionmaker позволяет создать собственный класс Session, в котором будут учтены все параметры:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(engine)

with Session() as session:
    # ...
```

Необходимость оборачивать всю логику базы данных в блок try/except также может стать очень утомительной. В следующем примере внутренний менеджер контекста, запускаемый методом begin(), заменяет обработку исключений:

```
with Session() as session:
    with session.begin():
        session.add(movie)
    print(movie)
```

Менеджер контекста, созданный с помощью session.begin(), реализует внутреннюю логику try/except. Он автоматически коммитит сессию в конце, и, если возникают какие-либо исключения, откатывает сессию назад. Во всех случаях сессия корректно закрывается внешним менеджером контекста.

Запросы

Если вы уже использовали SQLAlchemy в прошлом, имейте в виду, что начиная с версии 1.4 в SQLAlchemy были внесены значительные измене-

ния в построение ORM-запросов. В документации SQLAlchemy новый стиль запросов называется «стиль запросов 2.0», но этот стиль запросов можно использовать и в релизах 1.4, если объекты движка и сессии создаются с опцией future=True.

Определение запросов

В реляционных базах данных используют ключевое слово SELECT для реализации запросов. SQLAlchemy предоставляет функцию select() с аналогичной функциональностью. Самый простой запрос – тот, который возвращает все элементы в таблице.

Запрос, который извлекает все фильм, хранящиеся в базе данных:

```
from sqlalchemy import select

query = select(Movie)
```

Функция `select()` принимает в качестве аргументов элементы, которые необходимо извлечь. Если в качестве аргумента указан класс модели, SQLAlchemy ORM извлекает все атрибуты модели и возвращает объекты Python.

Иногда полезно посмотреть, какой SQL-код будет отправлен в базу данных при выполнении этого запроса. Код SQL, связанный с объектом запроса SQLAlchemy, можно увидеть при выводе запроса на печать:

```
print(query)

SELECT movies.id, movies.title, movies.director, mov-
ies.release_date, movies.duration, movies.genre, movies.rating
FROM movies
```

Можно увидеть, как класс Movie, переданный в функцию select(), был преобразован в оператор SELECT, который извлекает все атрибуты таблицы.

Выполнение запросов

После создания объекта запроса его нужно передать сессии, которая отправит его драйверу базы данных для выполнения через соединение, поддерживаемое движком. Самый универсальный способ сделать это – использовать метод execute() сессии:

```
result = session.execute(query)
list(result)
```

```
> [(Movie(1, "The Shawshank Redemption"),), (Movie(2, "The Godfa-
ther"),), ..., (Movie(46, "Avengers: Infinity War"),)]
```

Метод execute() возвращает объект с результатами. Это итерируемый объект, который извлекает результаты запроса.

Метод all() у объекта, возвращаемого функцией execute(), даст тот же эффект, что и преобразование, выполняемое функцией list():

```
session.execute(query).all()
```

```
> [(Movie(1, "The Shawshank Redemption"),), (Movie(2, "The Godfa-
ther"),), ..., (Movie(46, "Avengers: Infinity War"),)]
```

Методы, позволяющие получить первый результат запроса:

– first(): возвращает первую строку результата или None, если результатов нет. Если в наборе результатов есть еще строки, они отбрасываются.

– one(): возвращает первый и единственный результат. Если имеется ноль или более одной строки результатов, возникает исключение.

– one_or_none(): возвращает первый и единственный результат или None, если результатов нет. Если есть две или более строк результатов, то возникает исключение.

Если известно, что будет получено одно значение в строке, то для выполнения запроса можно использовать метод scalars():

```
session.scalars(query).all()
```

```
> [Movie(1, "The Shawshank Redemption"), Movie(2, "The Godfather"),
..., Movie(46, "Avengers: Infinity War")]
```

При использовании `scalars()` возвращается другой объект с результатами, который выполняет итерирование только по первому значению в каждой строке результатов. Если запрос вернул несколько значений в строке, то дополнительные значения в каждой строке отбрасываются.

Методы all(), first(), one() и one_or_none() также доступны для объекта, возвращаемого функцией scalars().

Есть несколько дополнительных методов выполнения запроса, которые объединяют scalars() с first(), one() и one_or_none():

– scalar(query): то же самое, что scalars(query).first(), возвращает первое значение первой строки результатов или None, если у запроса нет результатов.

– scalar_one(query): то же самое, что scalars(query).one(), возвращает первое значение единственной строки результатов, либо вызывает исключение, если результатов ноль или более одного.

– scalar_one_or_none(query): то же самое, что scalars(query).one_or_none(), возвращает первое значение единственной строки результатов или None, если результатов нет. При наличии двух или более результатов возникает исключение.

```
result = session.scalar(query)
print(result)
```

```
> Movie(1, "The Shawshank Redemption")
```

Фильтры

Запрос, содержащий только оператор select(), возвращает все доступные элементы. Существует множество ситуаций, в которых необходимо получить подмножество всех элементов, соответствующих каким-то критериям.

Фильтр можно добавить к объекту запроса с помощью условия where():

```
query = select(Movie).where(Movie.director == "Ridley Scott"
session.scalars(query).all()

> [Movie(27, "Gladiator"), Movie(35, "Alien")]
```

В SQLAlchemy реализовано очень сложное решение для определения фильтров, которое сочетает атрибуты классов модели со стандартными операторами Python, такими как ==.

В следующем примере оператор >= используется в запросе, который извлекает все фильмы, выпущенные после 2010 года:

```
query = select(Movie).where(Movie.release_date >= "2010-01-01")
```

Оператор where() может быть указан несколько раз для задания нескольких условий:

```
query = (select(Movie)
        .where(Movie.director == "Ridley Scott")
        .where(Movie.release_date >= "2000-01-01"))
```

Два или более фильтров также могут быть заданы в качестве нескольких аргументов в одной функции where():

```
query = select(Movie).where(Movie.director == "Ridley Scott",
Movie.release_date >= "2000-01-01")
```

При объединении нескольких фильтров, как показано выше, к ним применяется логический оператор AND.

Иногда в запросе может потребоваться объединить фильтры с помощью оператора OR, который в SQLAlchemy представлен в виде функции or_():

```
from sqlalchemy import or_
query = select(Movie).where(or_(Movie.release_date < "1969-12-31",
Movie.release_date > "2010-01-01"))
```

Логический оператор AND также может быть задан явно с помощью функции and_(). Унарный оператор NOT доступен в виде функции not_(). Еще один очень полезный фильтр – оператор LIKE, который можно использовать для реализации простой функции поиска:

```
query = select(Movie).where(Movie.director.like('%David%'))
```

Метод like() принимает строку шаблона поиска и возвращает все результаты, соответствующие этому шаблону. Шаблон задает текст для поиска, расширенный символом % в качестве подстановочного знака, который соответствует нулю символов, одному или нескольким символам.

Примеры шаблонов для фильтра like():

– Pattern%: элементы, начинающиеся с Pattern.

– %Pattern: элементы, которые заканчиваются на Pattern.

– % Pattern: элементы, которые заканчиваются пробелом, а затем Pattern.

– P__%: элементы, начинающиеся с буквы P, за которой следуют еще как минимум два символа.

– _: элемент длиной в один символ.

Функция like() чувствительна к регистру. Для поиска без учета регистра можно использовать функцию ilike().

Диапазон элементов можно запросить с помощью условия where(), определяющего два условия для нижней и верхней границ соответственно. Для тех же целей можно использовать метод between():

```
query = select(Movie).where(Movie.rating.between(9, 10))
```

Вот как выглядит запрос, преобразованный в SQL:

```
SELECT movies.id, movies.title, movies.director, mov-
ies.release_date, movies.duration, movies.genre, movies.rating
FROM movies
WHERE movies.rating BETWEEN :rating_1 AND :rating_2
```

Видно, что литеральные значения, определенные в фильтрах запроса, не вставляются в созданный SQL-запрос. Вместо этого они заменяются аргументами-заместителями, такими как :rating_1 и :rating_2. Это хорошо известная практика безопасности, которая предотвращает атаки SQL-инъекций, и SQLAlchemy реализует ее автоматически.

Сортировка результатов

Приведенные выше запросы возвращают запрашиваемые данные в порядке, выбранном сервером базы данных. Однако реляционные базы данных способны очень эффективно сортировать результаты, предоставляя их в том порядке, который нужен в приложении.

Метод order_by() может быть добавлен в запрос для указания желаемого порядка:

```
query = select(Movie).order_by(Movie.title)
session.scalars(query).all()
```

```
> [Movie(5, "12 Angry Men"), Movie(35, "Alien"), ..., Movie(31,
"Whiplash")]
```

Также можно отсортировать данные в обратном порядке, вызвав метод desc() для атрибута столбца, указанного в предложении order_by():

```
query = select(Movie).order_by(Movie.title.desc())
session.scalars(query).all()

> [Movie(31, "Whiplash"), Movie(20, "The Usual Suspects"), ...,
Movie(5, "12 Angry Men")]
```

Метод order_by() принимает несколько аргументов, каждый из которых добавляет новый уровень сортировки:

```
query = select(Movie).order_by(Movie.release_date.desc(), Mov-
ie.title.asc())
```

Обратите внимание на метод asc(), который используется для указания возрастающего порядка сортировки. Порядок по возрастанию используется по умолчанию, поэтому нет необходимости включать этот метод, но иногда он может сделать запрос более понятным, если порядок указан явно.

Доступ к отдельным столбцам

Во всех приведенных примерах запросов запрашивались целые строки из таблицы movies, которые SQLAlchemy ORM отображает на экземпляры класса Movie. Хотя такой способ запроса ORM-сущностей очень распространен, функция select() очень гибкая и может работать и с более детальными запросами.

В следующем запросе будут получены только названия фильмов:

```
query = select(Movie.title)
session.scalars(query).all()

> ['The Shawshank Redemption', 'The Godfather' ..., 'Avengers: In-
finity War']
```

Следующий запрос позволяет получить название и режиссера каждого фильма:

```
query = select(Movie.title, Movie.director)
session.execute(query).all()
```

```
> [('The Shawshank Redemption', 'Frank Darabont'), ('The Godfa-
ther', 'Francis Ford Coppola'), ..., ('Avengers: Infinity War',
'Anthony Russo')]
```

Функции агрегирования

Функция select() также может работать с функциями SQL, которые на лету преобразуют данные. Очень полезной функцией является count(), которая подсчитывает количество строк в результате запроса:

```
from sqlalchemy import func

query = select(func.count(Movie.id))
result = session.scalar(query)
result
```

```
> 46
```

Функция count() сокращает список результатов до одного значения, и поэтому для его получения используется метод scalar(). В данном примере использование Movie.id в качестве аргумента для подсчета результатов является произвольным. Можно указать любой атрибут столбца класса Movie, и результат будет тем же, поскольку сами данные не имеют значения.

Существует альтернативная форма приведенного выше запроса, которая не требует выбора произвольного столбца для подсчета результатов:

```
query = select(func.count()).select_from(Movie)
result = session.scalar(query)
result
```

```
> 46
```

Во второй форме функции count() не передаются аргументы, указывающие на то, что требуется подсчет результатов, и не указывается, какие данные считать. При использовании этого формата необходимо добавить метод select_from(), чтобы определить таблицу для использования в запросе, поскольку SQLAlchemy не может автоматически определить ее по аргументам, переданным функции select().

Еще одна пара полезных функций SQL – min() и max(). В следующем примере возвращается первое и последнее значение дат, в которые были выпущены фильмы, хранящиеся в базе данных:

```
query = select(func.min(Movie.release_date),
func.max(Movie.release_date))
result = session.execute(query)
result.first()
```

```
> (datetime.date(1936, 2, 25), datetime.date(2018, 4, 27))
```

Функции min() и max() сокращают список результатов до одной строки, поэтому нет смысла использовать all().

Если заранее известно, что результат будет один, удобнее использовать методы first() или one(), причем последний метод вызывает исключение для запросов, возвращающих что-либо, кроме одного результата.

Группировка результатов

Запрос списка режиссеров из таблицы movies:

```
query = select(Movie.director).order_by(Movie.director)
session.scalars(query).all()
```

Если запрос к базе данных возвращает дублирующиеся результаты, метод distinct() позволит объединить идентичные записи:

```
query = select(Movie.director).order_by(Movie.director).distinct()
session.scalars(query).all()
```

К сожалению, distinct() не работает при использовании агрегатной функции count(), поскольку база данных выполняет вызов count() до вызова distinct(). Если нужно подсчитать уникальные результаты, существует метод distinct(), который можно вызвать на подсчитываемом элементе внутри функции count():

```
query = select(func.count(Movie.director.distinct()))
session.scalar(query)
```

Метод group_by() используется для группировки результатов и обладает большей гибкостью. Приведенный выше запрос, возвращающий список режиссеров, также может быть создан с помощью group_by() следующим образом:

```
query = (select(Movie.director)
        .group_by(Movie.director)
        .order_by(Movie.director))
```

Результаты те же, но при использовании функции group_by() в запрос можно добавить дополнительные столбцы, если они агрегированы в одно значение для каждой группы с помощью функции.

Следующий пример позволяет получить список режиссеров с указанием первого и последнего годов их работ, а также количество выпущенных ими фильмов:

```
query = (select( Movie.director, func.min(Movie.release_date),
func.max(Movie.release_date), func.count())
        .group_by(Movie.director)
        .order_by(Movie.director))
session.execute(query).all()
```

При такой группировке база данных использует такие функции, как min(), max() и count(), чтобы уменьшить количество различных значений в объединяемых группах.

Запрос, использующий group_by(), может иметь результирующие значения, которые либо явно указаны в вызове group_by(), либо агрегированы с помощью функции. Наличие любых других результирующих значений приведет к ошибке, поскольку будет невозможно включить несколько строк значений в сгруппированную результирующую строку.

Ранее вы видели, что метод where() можно использовать для фильтрации результатов, возвращаемых запросом. Условия, заданные в where(), оцениваются до группировки результатов, поэтому это условие нельзя использовать для фильтрации сгруппированных результатов. Подобно where(), функция having() используется для фильтрации сгруппированных и агрегированных результатов.

Запрос, получающий список режиссеров, у которых есть три фильма или больше в базе данных, вместе с их фактическим числом:

```
query = (select( Movie.director, func.count() )
        .group_by(Movie.director)
        .having(func.count() >= 2)
        .order_by(Movie.director))
session.execute(query).all()
```

```
> [('Charles Chaplin', 2), ('Christopher Nolan', 5), ('Francis Ford
Coppola', 2), ('Frank Darabont', 2), ('Martin Scorsese', 2),
('Quentin Tarantino', 2), ('Ridley Scott', 2), ('Robert Zemeckis',
2), ('Stanley Kubrick', 2), ('Steven Spielberg', 2)]
```

Можно заметить, что в этом примере функция count() появляется дважды. Сначала она используется в части select() для включения в результаты, а затем в методе having для отфильтровывания записей по этому значению.

Чтобы результат подсчета количества фильмов по каждому режисссеру был записан только один раз, можно использовать метод label() для привязки метки к подсчету, а затем использовать эту метку в тех местах, где она необходима:

```
num_movies = func.count().label(None)
query = (select( Movie.director,
             num_movies
            )
            .group_by(Movie.director)
            .having(num_movies >= 2)
            .order_by(Movie.director))
```

Аргумент метода label() – это имя для метки, которое автоматически генерируется SQLAlchemy, если указано None, что гарантирует выбор уникального имени.

Позволить SQLAlchemy автоматически выбрать имя – нормальная практика, поскольку важно то, что экземпляр метки назначен переменной num_movies. Но в любом случае, если вы предпочитаете указать имя для метки, это также можно сделать:

```
num_movies = func.count().label('num_movies')
```

Пагинация

Для запросов, возвращающих большой список результатов, общепринятой практикой является ограничение количества возвращаемых результатов некоторым максимальным числом.

Метод limit(), добавленный к запросу, устанавливает максимальное количество получаемых результатов:

```
query = select(Movie).order_by(Movie.title).limit(3)
session.scalars(query).all()
```

```
> [Movie(5, "12 Angry Men"), Movie(35, "Alien"), Movie(22, "Ameri-
can History X")]
```

Распространенной схемой при выдаче потенциально больших результатов является предоставление результатов в виде разбиения на страницы. В веб-приложениях пользователям часто предоставляется возможность перемещаться вперед и назад по результатам с шагом страницы заданного разме-

ра. Реализация пагинации результатов запроса включает установку размера страницы с помощью метода limit() и указание позиции, с которой следует начать получение результатов.

Самый простой подход к выбору начальной позиции – добавить метод offset() к запросу: этот метод устанавливает начальный индекс для получаемых результатов. Запрос для извлечения второй страницы из трех результатов будет выполнен следующим образом:

```
query = select(Movie).order_by(Movie.title).limit(3).offset(3)
session.scalars(query).all()
```

```
> [Movie(36, "Apocalypse Now"), Movie(46, "Avengers: Infinity
War"), Movie(24, "Back to the Future")]
```

Использование offset() для пагинации часто таит в себе проблемы:

– в большинстве баз данных он не имеет эффективной реализации;

– может возвращать неожиданные результаты для наборов данных, которые часто меняются.

Если добавляется новая запись в таблицу после того, как пользователь просмотрел вторую страницу результатов, то при запросе третьей страницы все позиции смещаются на одну позицию вниз, и просмотренный элемент снова появляется в выдаче. Если запись удаляется, все позиции после удаленного элемента смещаются на одну позицию вверх, и элемент может быть пропущен в выдаче.

Более надежный способ указать, откуда базе данных необходимо начать возвращать результаты, – использовать последний возвращенный элемент в качестве отсчета. Чтобы запросить вторую страницу результатов с помощью этого метода, надо сформировать запрос следующим образом:

```
query = select(Movie).order_by(Movie.title).where(Movie.title >
"American History X").limit(3)
```

Этот запрос возвращает те же результаты, что и предыдущий пример, но имеет явное преимущество: добавленные или удаленные элементы не приведут к повторению или пропуску результатов.

Недостаток в том, что при навигации по списку результатов назад запрос становится немного сложнее. После просмотра второй страницы результатов запрос для получения первой страницы результатов будет выглядеть так:

```
query = (se-
lect(Movie).order_by(Movie.title.desc()).where(Movie.title < "Apoc-
alypse Now").limit(3))
```

Получение элемента по его первичному ключу

Особенно полезным является запрос, который извлекает элемент из таблицы базы данных, соответствующий заданному значению первичного ключа. Этот запрос может вернуть один результат, если искомый элемент найден, или вообще не вернуть никаких результатов, если заданное значение ключа не существует. Запрос никогда не вернет больше одного результата, поскольку первичные ключи по определению уникальны:

```
query = select(Movie).where(Movie.id == 23)
session.scalar(query)
```

В следующем примере показано, как выполнить тот же запрос, что и выше, с помощью метода get() объекта сеанса:

```
session.get(Movie, 23)
```

Как и в случае с более длинной формой записи выше, если заданное значение первичного ключа не существует в таблице базы данных, возвращаемое значение будет равно None.

Ограничения

Хорошая практика проектирования базы данных – назначать ограничения столбцам.

Вспомним определение модели Movie:

```
class Movie(Base):
    __tablename__ = "movies"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director: Mapped[str] = mapped_column(String(64))
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    genre: Mapped[str] = mapped_column(String(32))
    rating: Mapped[float] = mapped_column()

    def __repr__(self):
        return f'Movie({self.id}, "{self.title}")'
```

Модель Movie уже имеет ограничение PRIMARY KEY, которое включено для столбца id с помощью параметра primary_key=True. Столбец id называется первичным ключом таблицы movies. Два других часто используемых ограничения – UNIQUE и NOT NULL.

Столбец с ограничением UNIQUE не допускает дублирования значений. Чтобы добавить это ограничение к столбцу, нужно задать параметр unique=True.

Ограничение NOT NULL предотвращает появление в столбце пустого или неопределенного значения. Столбцы, не имеющие ограничения NOT NULL, считаются необязательными.

Столбцы, определенные с помощью синтаксиса типизации Mapped[type], по умолчанию получают ограничение NOT NULL, и для создания столбца, которому разрешено иметь значения NULL, подсказку типа следует изменить на Mapped[Optional[type]] или Mapped[type | None].

Удаление

Новые объекты добавляются в базу данных с помощью метода add() сеанса. Сохранение нового объекта в базе данных происходит в следующей операции commit(). У сеанса также есть метод delete():

```
movie = session.get(Movie, 23)
session.delete(movie)
session.commit()
```

После коммита сеанса с удаленными объектами эти объекты фактически удаляются и больше не могут быть извлечены:

```
movie = session.get(Movie, 23)
print(movie)

> None
```

ОТНОШЕНИЯ «ОДИН КО МНОГИМ»

Реляционные базы данных позволяют создавать связи между сущностями, хранящимися в разных таблицах, посредством отношений.

Существует два основных типа отношений:

– «один ко многим»;

– «многие ко многим».

В литературе по реляционным базам данных также упоминаются два дополнительных типа отношений: многие к одному и один к одному, которые являются особыми формами отношения «один ко многим».

Отношение «один ко многим» описывает ситуацию, в которой две таблицы A и B связаны таким образом, что запись в таблице A может быть связана с любым количеством записей в таблице B, но каждая запись в таблице

B связана не более чем с одной записью из A. B этом сценарии таблица A является стороной «один», отношения а таблица B является стороной «многие».

Первым шагом при определении связи является создание таблиц базы данных (или моделей, при использовании SQLAlchemy ORM) для двух вовлеченных сущностей:

```
class Director(Base):
    __tablename__ = "directors"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), unique=True)


    movies: Mapped[list["Movie"]] = relationship("Movie",
back_populates="director")


from sqlalchemy import ForeignKey

class Movie(Base):
    __tablename__ = "movies"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director_id: Mapped[int] =
mapped_column(ForeignKey("directors.id"), index=True)
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    genre: Mapped[str] = mapped_column(String(32))
    rating: Mapped[float] = mapped_column()

    director: Mapped[Director] = relationship("Director",
back_populates="movies")
```

Новый столбец director_id представляет собой целое число, соответствующее типу первичного ключа новой модели Director. Столбцы, ссылающиеся на первичные ключи другой таблицы, называются внешними ключами и им присваивается ограничение внешнего ключа. Хорошим соглашением об именовании является присвоение столбцам внешнего ключа имени сущности с добавлением суффикса _id. Класс ForeignKey применяет ограничение внешнего ключа к столбцу.

Аргумент, переданный классу, указывает, что является целью первичного ключа, который может быть задан как объект столбца (например, Director.id) или как строка в формате <tablename>.<column>:

```
director_id: Mapped[int] =
mapped_column(ForeignKey("directors.id"), index=True)
```

Новый столбец director_id имеет индекс для более эффективного использования этого столбца. Некоторые базы данных автоматически индексируют столбцы внешнего ключа, но другие этого не делают, поэтому лучше явно добавлять индекс. Столбец также имеет ограничение NOT NULL, косвенно из-за подсказки столбца, не включающей Optional. Указание того, что внешний ключ не может быть пустым, гарантирует, что фильмы без связи с режиссерами не будут возможны.

ORM SQLAlchemy обеспечивает высокоуровневую поддержку отношений, выполняя большую часть работы по использованию внешних ключей. Чтобы получить доступ к этим функциям, двум классам моделей, участвующим в отношении, нужны атрибуты отношения, которые представляют это отношение:

```
from sqlalchemy.orm import relationship

class Movie(Base):
    # ...
    director: Mapped[Director] = relationship("Director",
back_populates="movies")
    # ...

class Director(Base):
    # ...
    movies: Mapped[list["Movie"]] = relationship("Movie",
back_populates="director")
    # ...
```

Класс Movie теперь имеет атрибут director, который представляет отношение со стороны «многие». Этот атрибут не является столбцом, который

физически хранится в базе данных; это высокоуровневая замена director_id, которая загружает связанный объект модели.

Класс Director имеет новый атрибут movies, представляющий ту же связь, но со стороны «один». С этой стороны режиссер может иметь много связанных фильмов, поэтому этот атрибут представляет собой список, который автоматически заполняется соответствующими экземплярами фильмов.

Запросы отношений «один ко многим»

Рассмотрим следующий запрос:

```
movie = session.scalar(select(Movie).where(Movie.title == "Inter-
stellar"))
```

```
> Movie(18, "Interstellar")
```

Так как вместо столбца строки есть объект отношения, то этот новый атрибут возвращает экземпляр модели, представляющий связанную сущность:

```
movie.director
```

```
> Director(4, "Christopher Nolan")
```

Имя режиссера хранится в атрибуте name модели, и это имя доступно для обращения к нему:

```
movie.director.name
```

```
> 'Christopher Nolan'
```

Проверка отношений в другую сторону:

```
movie.director.movies
```

```
> [Movie(4, "The Dark Knight"), Movie(9, "Inception"), Movie(18,
"Interstellar"), ...]
```

Один из запросов, показанных ранее, возвращал названия фильмов и имена режиссеров в одном и том же результате. Когда обе сущности определены в одной таблице, это сделать легко. Но теперь для этого требуется объединить информацию из двух таблиц – выполнить операцию соединения (join).

Запрос, который реализует соединение:

```
query = select(Movie.title, Director.name).join(Movie.director)
session.execute(query).all()
```

```
> [('The Shawshank Redemption', 'Frank Darabont'), ('The Godfa-
ther', 'Francis Ford Coppola'), ...]
```

В этом запросе оператор select() использует два атрибута из разных таблиц. Оператор join() может быть использован у любого атрибута связи, и SQLAlchemy вычисляет результат исходя из него. Поскольку два объекта связаны через параметры back_populates, в общем случае неважно, какой из двух задан в join().

В приведенном выше примере запроса передача Movie.title в join() означает, что Movie будет находиться с левой стороны соединения, а Director – с правой.

Реализуем один из предыдущих запросов, который вернул режиссеров в алфавитном порядке вместе с количеством фильмов, произведенных каждым из них:

```
from sqlalchemy import func

query = (select(
        Director,
        func.count(Movie.id))
        .join(Director.movies)
        .group_by(Director)
        .order_by(Director.name))
session.execute(query).all()
```

```
> [(Director(21, "Alfred Hitchcock"), 1), (Director(33, "Anthony
Russo"), 1), ..., (Director(18, "Tony Kaye"), 1)]
```

Обратите внимание на Director, указанного в операторе select(), и еще раз в group_by(). Когда group_by() получает в качестве аргумента класс модели, а не один атрибут, группировка выполняется по всем атрибутам модели вместе взятым. Такая группировка транслируется в следующий SQL:

```
print(query)

SELECT directors.id, directors.name, count(movies.id) AS count_1
FROM directors JOIN movies ON directors.id = movies.director_id
GROUP BY directors.id, directors.name ORDER BY directors.name
```

Отношения Lazy и Eager

Выполним следующий запрос:

```
from db import Session
from models import Movie, Director
from sqlalchemy import select

session = Session()
director = session.scalar(select(Director).where(Director.name ==
"Alfred Hitchcock"))
```

Сразу после выполнения запроса с помощью scalar() в терминале появятся (при заданном параметре echo=True движка) записи:

```
2024-08-31 22:29:30,135 INFO sqlalchemy.engine.Engine select
pg_catalog.version()
2024-08-31 22:29:30,136 INFO sqlalchemy.engine.Engine [raw sql] {}
2024-08-31 22:29:30,190 INFO sqlalchemy.engine.Engine select cur-
rent_schema()
2024-08-31 22:29:30,190 INFO sqlalchemy.engine.Engine [raw sql] {}
2024-08-31 22:29:30,215 INFO sqlalchemy.engine.Engine show stand-
ard_conforming_strings
2024-08-31 22:29:30,215 INFO sqlalchemy.engine.Engine [raw sql] {}
2024-08-31 22:29:30,277 INFO sqlalchemy.engine.Engine BEGIN (im-
plicit)
2024-08-31 22:29:30,293 INFO sqlalchemy.engine.Engine SELECT direc-
tors.id, directors.name
FROM directors
WHERE directors.name = %(name_1)s
2024-08-31 22:29:30,294 INFO sqlalchemy.engine.Engine [generated in
0.00047s] {'name_1': 'Alfred Hitchcock'}
```

При попытке получения доступа к director или любому из его прямых атрибутов, таких как director.name, дополнительной активности базы данных не будет, поскольку все атрибуты были загружены из запроса и кэшированы в сеансе базы данных.

Но посмотрим, что происходит, когда мы пытаемся получить доступ к отношению director.movies:

```
> 2024-08-31 22:32:42,470 INFO sqlalchemy.engine.Engine SELECT mov-
ies.id AS movies_id, movies.title AS movies_title, mov-
ies.director_id AS movies_director_id, movies.release_date AS mov-
ies_release_date, movies.duration AS movies_duration, movies.genre
AS movies_genre, movies.rating AS movies_rating
FROM movies
WHERE %(param_1)s = movies.director_id
2024-08-31 22:32:42,470 INFO sqlalchemy.engine.Engine [generated in
0.00041s] {'param_1': 21}
[Movie(26, "Psycho")]
```

SQLAlchemy выполняет запрос к базе данных для получения списка фильмов, связанных с режиссером. В такой ситуации говорят, что SQLAlchemy использует «ленивую» (Lazy) загрузку отношений. Благодаря этому к атрибутам отношений можно обращаться как к обычным атрибутами модели.

У ленивой загрузки есть недостаток. Тот факт, что запросы к базе данных выполняются неявно, может привести к тому, что приложение станет неэффективным из-за слишком большого количества неявных запросов, появляющихся на сервере базы данных.

Пример:

```
query = select(Movie)

for movie in session.scalars(query):
    print(movie.title, movie.director.name)
```

Точное количество запросов, требуемых этим циклом, равно одному для начального запроса, сохраненного в переменной query, плюс один дополнительный запрос ленивой загрузки на каждого режиссера.

Загрузчики отношений

SQLAlchemy предлагает несколько вариантов настройки отношений и делает их более полезными и эффективными в зависимости от того, как они будут использоваться. SQLAlchemy использует загрузчик отношений (Relationship Loaders) для добавления одного или нескольких связанных объектов в сеанс. Загрузчик по умолчанию, который вы видели в примере выше, называется загрузчиком select.

Другой доступный загрузчик – joined. Этот загрузчик считывает связанные объекты из базы данных одновременно с извлечением родительского объекта, расширяя основной запрос соединением join.

Загрузчик select – это «ленивый» загрузчик, поскольку запрос в базу данных для связанных объектов откладывается до тех пор, пока атрибут отношения не будет вызван в первый раз.

Загрузчик joined – это «жадный» загрузчик, поскольку данные отношения запрашиваются одновременно с родительским объектом, независимо от того, нужно ли это приложению или нет. При использовании загрузчика joined цикл в прошлом примере не произведет никаких дополнительных запросов помимо первоначального.

Чтобы включить этот загрузчик, нужно добавить метод options() к запросу следующим образом:

```
from sqlalchemy.orm import joinedload

query = select(Movie).options(joinedload(Movie.director))
```

Вместо выбора загрузчика явно в каждом запросе, можно также изменить загрузчик по умолчанию, который используется отношением. Это делается путем передачи желаемого загрузчика в аргументе lazy:

```
class Movie(Base):
    # ...
    director: Mapped['Director'] = relationship(lazy='joined',
back_populates='movies')
    # ...
```

Join-загрузчик полезен, когда вы точно знаете, что понадобится доступ к связанным объектам. Он также вряд ли будет хорошо работать для сложных запросов или отношений со многими элементами, потому что стоимость добавления join в этих случаях может быть значительной.

Ниже приведен полный список загрузчиков, которые можно использовать.

– select: «лениво» выполняет оператор select(), когда атрибут отношения запрашивается в первый раз. Поведение по умолчанию. В качестве параметра запроса этот загрузчик можно включить с помощью функции lazyload().

– immediately: загружает связанные сущности одновременно с загрузкой родителя с помощью отдельного оператора select(). Единственное отличие между select и immediately заключается в том, что последний выполняет все запросы отношений заранее, а не по требованию. В качестве параметра этот загрузчик включается с помощью функции immediatelyload().

– join: загружает связанные сущности одновременно с загрузкой родителя, расширяя запрос родителя с помощью соединения со связанной таблицей. Функция joinload() включить ее как параметр в запросе.

– subquery: загружает связанные сущности сразу после родителя в одном дополнительном запросе, который объединяет копию исходного запроса

(преобразованного в подзапрос) со связанной таблицей. Вариантом этого загрузчика является функция subqueryload().

– selectin: загружает связанные сущности сразу после родительской в одном дополнительном запросе, который указывает список первичных ключей для загрузки с помощью оператора IN. В качестве опции этот загрузчик включается с помощью опции selectinload().

– write_only: отключает загрузку отношения. Этот загрузчик доступен только в SQLAlchemy 2.0 и более поздних версиях и может использоваться только в отношении, поэтому у него нет версии с опцией.

– noload: похожая опция на write_only, но менее гибкая. Рекомендуется вместо write_only при использовании SQLAlchemy 1.4. Версия с опцией этого загрузчика включается с помощью функции noload().

– raise и raise_on_sql: два немного разных режима, в которых возникает исключение, если отношение необходимо загрузить неявно. Эти режимы можно использовать для обнаружения кода, который запускает неявные операции ввода-вывода, когда они нежелательны. Функция raiseload() – это версия с опцией запроса.

Удаление связанных объектов с помощью каскадирования

Удаление со стороны «многие» отношения «один ко многим» работает таким же образом, как и удаление отдельной сущности:

```
movie = session.get(Movie, 24)

> Movie(24, "Back to the Future")

director = movie.director

> Director(8, "Robert Zemeckis")

session.delete(movie)
session.commit()
```

Попробуем удалить режиссера:

```
session.delete(d)
session.commit()
```

```
...
sqlalchemy.exc.IntegrityError: (psycopg2.errors.NotNullViolation)
null value in column "director_id" of relation "movies" violates
not-null constraint
...
```

У режиссера есть несколько фильмов, все из которых все еще существуют и имеют свой внешний ключ director_id, указывающий на эту запись. Если режиссер удален, то внешние ключи этих режиссеров станут недействительными. SQLAlchemy распознает это событие и пытается установить внешние ключи, которые становятся недействительными, в значение NULL перед удалением режиссера. Однако столбец director_id не определен как необязательный, поэтому попытка установить его в NULL не удается и выдается указанное выше сообщение об ошибке.

Когда операция, например, commit, не удается, сеанс переходит в состояние ошибки и не может быть использован до тех пор, пока не будет выполнен откат. Это ситуация не вызывает никаких серьезных проблем при использовании менеджера контекста, поскольку сеансы всегда откатываются при ошибках и закрываются при выходе, но при управлении жизненным циклом сеанса вручную в командной строке Python необходимо явно выполнить откат:

```
session.rollback()
```

Автоматические операции, такие как попытка очистить внешние ключи элемента, который должен быть удален, называются каскадами и не ограничиваются удалениями.

Есть несколько других ситуаций, в которых SQLAlchemy также применяет изменение к дочерним объектам в результате действия, выполненного над родительским объектом.

Желаемое каскадное поведение определяется в каждом объекте отношения, и во многих случаях наилучшей конфигурацией является конфигурация по умолчанию.

Чтобы изменить параметры каскадирования, в аргументе cascade вызова relation() указывается строка с параметрами, разделенными запятыми.

Две наиболее часто используемые каскадные конфигурации:

– save-update, merge: консервативное каскадное поведение, которое является поведением по умолчанию, рекомендуемым для большинства отношений. Поскольку это поведение по умолчанию, нет необходимости задавать параметры явно. При такой конфигурации дочерние объекты автоматически включаются в сеанс, если был добавлен родительский объект. Поведение, которое устанавливает внешние ключи, становящиеся недействительными при удалении, в NULL, на самом деле определяется отсутствием параметра delete в этом списке.

– all, delete-orphan: альтернативная, более агрессивная каскадная конфигурация, которая делает так, что большинство операций, выполняемых над родителем, применяются к его дочерним объектам и, в частности, конфигурация удаляет дочерние объекты вместе с их родителем. Параметр all вызывает некоторую путаницу, поскольку, несмотря на свое название, он охватывает все каскады, кроме delete-orphan, который приводит к тому, что дочерние объекты также удаляются, когда они удаляются из своих отношений и становятся «сиротскими», даже если их родительский объект остается в базе данных.

Со следующим изменением конфигурации при удалении режиссера все связанные с ним фильмы также будут удалены:

```
class Director(Base):
    ...
    movies: Mapped[list['Movie']] = relationship(cascade='all, de-
lete-orphan', back_populates='manufacturer')
    ...
```

Отсоединение связанных объектов

Иногда необходимо удалить связь между двумя объектами, не удаляя сами объекты. Эту операцию можно рассматривать как операцию «отсоединения», которая разрывает связь между двумя объектами.

Для связи «один ко многим» есть два способа отсоединения двух связанных объектов в зависимости от стороны, с которой это действие совершается. При выполнении действия со стороны «один» объект связи представляет все связанные объекты на стороне «многие» в формате, похожем на список. В этом случае метод remove() объекта связи можно использовать для удаления элемента:

```
movie = session.get(Movie, 1)

> Movie(1, "The Shawshank Redemption")

director = movie.director

> Director(1, "Frank Darabont")

director.movies.remove(movie)
session.commit()
```

Что произойдет, если параметр delete-orphan не будет использован? Тогда SQLAlchemy установит внешний ключ director_id в Movie в None, чтобы разорвать связь с родителем, но этот столбец не может не принимать значения, поэтому результатом будет то, что операция коммита завершится неудачей, и связь не будет удалена.

Для связи «один ко многим», в которой допустимо иметь объекты со стороны «многие» в сиротском состоянии, столбец внешнего ключа должен быть настроен как допускающий значение Null, путем добавления подсказки типа Optional, так как это предотвратит ошибку.

Отсоединение связи «один ко многим» также работает со стороны «многие»:

```
movie = session.get(Movie, 2)

> Movie(2, "The Godfather")

movie.director = None
session.commit()
```

МИГРАЦИИ

Alembic – инструмент миграции базы данных, который является частью семейства SQLAlchemy.

Установка пакета:

```
poetry add alembic
```

Первым шагом для включения миграции баз данных является создание репозитория миграции с помощью команды alembic init:

```
alembic init migrations
```

Аргумент migrations – это имя подкаталога, который создается в каталоге проекта, где будут храниться скрипты миграции базы данных. В дополнение к этому подкаталогу в каталоге проекта создается файл alembic.ini. Для проекта, находящегося под управлением Git, файл alembic.ini и все содержимое подкаталога migrations следует рассматривать как исходный код и поддерживать вместе с исходными файлами приложения.

Файлы, созданные командой alembic init, изначально не имеют информации о том, какую базу данных использует проект. Чтобы указать Alembic на базу данных проекта, необходимо внести несколько простых изменений в конфигурацию. Единого способа сделать это не существует, но для проекта наиболее удобным вариантом будет редактирование файла env.py, расположенного внутри migrations.

В верхней части файла импортируем engine и Base:

```
from db import Base, engine
import models
```

Затем надо найти строку:

```
target_metadata = None
```

и заменить ее следующим кодом:

```
target_metadata = Base.metadata
config.set_main_option("sqlalchemy.url", en-
gine.url.render_as_string(hide_password=False))
```

Переменная target_metadata – это переменная, в которой Alembic ожидает экземпляр метаданных, используемый приложением. Для SQLAlchemy ORM этот экземпляр существует как атрибут метаданных декларативного базового класса Base.

Вторая строка вставляет значение для параметра sqlalchemy.url в объект конфигурации Alembic. Этот параметр определяет URL базы данных. Поскольку приложение создает экземпляр движка с этим URL, наиболее удобным вариантом (хотя, возможно, не самым эффективным) является получение URL из этого объекта напрямую.

Создание скрипта миграции

Alembic использует концепцию скриптов миграции для отслеживания изменений, вносимых в базу данных. Скрипт миграции содержит код Python,

который вносит изменения в действующую базу данных, не удаляя никаких таблиц или данных.

Alembic может автоматически сгенерировать сценарий миграции, сравнивая классы модели с соответствующими им таблицами базы данных. Например, если таблица, на которую ссылается класс модели, не существует в базе данных, Alembic решает, что это новая таблица, которую необходимо создать в базе данных, сопоставляя определения в классе модели.

Если на существующую в базе данных таблицу не ссылается ни один класс модели в приложении, то Alembic решает, что таблицу необходимо удалить, и генерирует код для этого в сценарии миграции.

Целью сгенерированного сценария миграции всегда является внесение любых необходимых изменений в базу данных, чтобы она отражала состояние моделей.

Для генерации начальной миграции необходимо, чтобы база данных была полностью пустой, поскольку это заставит Alembic сгенерировать миграцию, которая создаст таблицы.

```
alembic revision --autogenerate -m "add movies, directors"
```

Alembic выведет в терминал некоторые логи, указывающие на то, что он обнаружил новые таблицы и новые индексы.

Затем он покажет имя сгенерированного скрипта миграции, который будет иметь формат {code}_add_movies_directors.py, где {code} – уникальный код, идентифицирующий миграцию.

Сгенерированный скрипт миграции – это модуль Python, который имеет две функции upgrade() и downgrade(). Функция upgrade() применяет изменения, чтобы привести базу данных в соответствие с моделями, а функция downgrade() отменяет эти изменения.

Каждый скрипт миграции будет иметь эти две функции, что позволит Alembic выполнить цепочку upgrade или цепочку downgrade, вызывая соответствующие функции в правильном порядке.

Пример полученного скрипта миграции:

```
"""add movies, directors

Revision ID: 034e4b303867
Revises:
Create Date: 2024-09-03 21:29:39.008794

"""
from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision = '034e4b303867'
down_revision = None
branch_labels = None
depends_on = None


def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('directors',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('name', sa.String(length=64), nullable=False),
    sa.PrimaryKeyConstraint('id'),
    sa.UniqueConstraint('name')
    )
    op.create_table('movies',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('title', sa.String(length=128), nullable=False),
    sa.Column('director_id', sa.Integer(), nullable=False),
    sa.Column('release_date', sa.Date(), nullable=False),
    sa.Column('duration', sa.Integer(), nullable=False),
    sa.Column('genre', sa.String(length=32), nullable=False),
    sa.Column('rating', sa.Float(), nullable=False),
    sa.ForeignKeyConstraint(['director_id'], ['directors.id'], ),
    sa.PrimaryKeyConstraint('id'),
    sa.UniqueConstraint('title')
    )
    op.create_index(op.f('ix_movies_director_id'), 'movies', ['director_id'], unique=False)
    # ### end Alembic commands ###


def downgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_index(op.f('ix_movies_director_id'), table_name='movies')
    op.drop_table('movies')
    op.drop_table('directors')
    # ### end Alembic commands ###
```

Обновление базы данных

Команда alembic revision создает скрипт миграции, но не выполняет его. Следующая команда запускает скрипт миграции:

```
alembic upgrade head
```

Команда alembic имеет много подкоманд:

– downgrade для отмены миграции базы данных;

– current для отображения того, на какой миграции находится база данных;

– history для просмотра списка миграций и др.

АСИНХРОННОСТЬ

Начиная с версии 1.4, SQLAlchemy включает поддержку асинхронного кода с пакетом asyncio как для модулей Core, так и для ORM. Асинхронные приложения должны избегать долго выполняющихся синхронных функций, поскольку они блокируют и мешают параллелизму.

В результате этого ограничения веб-приложению, использующему SQLAlchemy, требуется асинхронный код во всех слоях:

– веб-сервер;

– веб-фреймворк;

– логика маршрутизации;

– сессия SQLAlchemy;

– движок SQLAlchemy;

– драйвер базы данных.

Еще одно важное отличие связано с неявной активностью базы данных. SQLAlchemy ORM – это высокоуровневый фреймворк базы данных, который иногда самостоятельно выполняет запросы к базе данных. Пример – атрибуты отношений, определенные с помощью ленивого загрузчика, который неявно запускает выполнение запроса к базе данных для получения результатов при первом обращении к атрибуту. Такое неявное поведение не может существовать в асинхронном приложении из-за ограничений раскраски функций асинхронной модели. Вся асинхронная активность, связанная с базой данных, должна происходить внутри функций, которые являются асинхронными.

Асинхронные драйверы баз данных

Для PostgreSQL драйвер asyncpg является единственным асинхронным вариантом.

Установка драйвера:

```
poetry add asyncpg
```

В URL-адрес подключения к базе данных добавляется asyncpg в диалектную часть.

```
DATABASE_URL=postgresql+asyncpg://username:password@localhost:5432/
db
```

Движки, метаданные и сеансы

SQLAlchemy поставляется с асинхронным расширением, которое предоставляет альтернативные объекты движка и сеанса. Они имеют те же интерфейсы, что и обычные версии, которые использовались в примерах ранее:

```
import os
from dotenv import load_dotenv
from sqlalchemy import MetaData
from sqlalchemy.ext.asyncio import create_async_engine,
async_sessionmaker
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    pass

load_dotenv()

engine = create_async_engine(os.environ['DATABASE_URL'])
Session = async_sessionmaker(engine, expire_on_commit=False)
```

Важная особенность – экземпляр MetaData не имеет асинхронной версии. Поэтому у функций create_all() и drop_all() нет версии с await. SQLAlchemy предоставляет метод run_sync(), который можно использовать для запуска синхронного кода:

```
async with engine.begin() as connection:
    await connection.run_sync(Base.metadata.drop_all)
    await connection.run_sync(Base.metadata.create_all)
```

Загрузчики отношений

По большей части определения моделей не нужно менять для асинхронного приложения. Единственное, что нужно тщательно проверить, – это конфигурация загрузчиков отношений. Многие атрибуты relationship() в классах моделей используют механизм ленивой загрузки, который запрашивает отношение из базы данных при первом доступе к атрибуту. Для изменения этого поведения можно использовать аргумент lazy, предложение запроса options() и подсказку WriteOnlyMapped.

Ленивое поведение по умолчанию, которое выглядит как lazy='select' или options(lazyload(...)), несовместимо с асинхронными приложениями, поэтому необходимо изменить его на загрузчик с более предсказуемым поведе-

нием. Безопасным выбором будет изменить все отношения ленивой загрузки на lazy='raise', чтобы они вызывали ошибку, если SQLAlchemy потребуется лениво загрузить данные. При этом приложение может явно передать один из жадных загрузчиков в параметре options() в качестве явного переопределения.

Другой вариант – выбрать соответствующие загрузчики для всех отношений, чтобы избежать любой возможности ленивой загрузки:

– Отношения «один»: загрузчик join-eagy. Если отношение не является необязательным, будет добавлена опция innerjoin=True (используется внутренний join, который часто более эффективен, чем левый join, который использует этот загрузчик).

– Отношения «многие» с write_only loader не будут изменены, так как они совместимы с асинхронным кодом.

– Оставшиеся отношения «многие» будут использовать жадный загрузчик selectin, который часто работает лучше, чем join, когда в отношении есть несколько элементов.

Пример:

```
class Movie(Base):
    ...
    director: Mapped['Director'] = relationship(lazy='joined', in-
nerjoin=True, back_populates='movies')
    ...


class Director(Base):
    ...
    movies: Mapped[list['Movie']] = relationship(lazy='selectin',
cascade='all, delete-orphan', back_populates='directors')
    ...
```

Конфигурация Alembic

Миграции баз данных требуют минимальных изменений при переходе на асинхронную модель разработки. Alembic использует концепцию шабло-

нов для генерации содержимого репозиториев миграции, которые он создает с помощью команды init, в частности файлов env.py и alembic.ini

Alembic поставляется с асинхронным шаблоном, который можно использовать при инициализации репозитория миграции.

Команда инициализации:

```
alembic init -t async migrations
```

Полученный файл env.py в подкаталоге migrations будет иметь несколько незначительных отличий от файла, основанного на шаблоне по умолчанию. Изменения аналогичны тем, что нужны для синхронной версии.

Генерация начальной миграции базы данных:

```
alembic revision --autogenerate -m "initial migration"
```

Обновление базы данных:

```
alembic upgrade head
```

Асинхронные запросы

Сеансы базы данных должны использовать асинхронные менеджеры контекста, поэтому операторы with необходимо изменить на async with:

```
async with Session() as session:
    async with session.begin():
    ...
```

Запросы и коммиты теперь выполняются асинхронно, поэтому для них следует использовать await:

```
session.execute()
session.scalar()
session.commit()
```

ОТНОШЕНИЯ «МНОГИЕ КО МНОГИМ»

В отношениях «один ко многим» на стороне «многие» добавляется внешний ключ, указывающий на сторону «один», и этого достаточно для создания отношения. Для полного представления отношений «многие ко многим» необходимо два отношения «один ко многим». Поскольку невозможно построить прямое отношение «многие ко многим» между двумя таблицами, добавляется третья таблица, называемая таблицей ассоциаций или соединения. Затем каждая сторона устанавливает отношения «один ко многим» с этой таблицей. Это означает, что к ней добавляются внешние ключи, ссылающиеся на две таблицы, участвующие в отношениях.

Таблица movies_genres – это таблица ассоциаций, в которой записаны внешние ключи каждой пары связанных фильмов и жанров. Общим соглашением об именовании таблиц ассоциаций является использование имен двух сущностей, которые являются частью отношений.

Например, для фильма, у которого три жанра, в таблице соединений будет три записи, в которых внешний ключ movie_id связан с фильмом, а genre_id – с один из жанров. А с другой стороны, для жанра, в котором было создано пять фильмов, будет столько же записей, в которых genre_id будет связан с жанром, каждая из которых будет в свою очередь связана с одним из пяти фильмов.

Управление внешними ключами в таблице ассоциаций отношения «многие ко многим» может показаться чересчур сложным, но SQLAlchemy выполняет большую часть работы.

Первым шагом в реализации отношения является создание таблицы ассоциации. Эта таблица будет управляться SQLAlchemy, поэтому все поведение, предоставляемое классом Declarative Base из SQLAlchemy, не нужно.

Вместо этого таблицу можно создать с помощью класса Table из SQLAlchemy Core. Чтобы было удобно ссылаться на эту таблицу в моделях Mbovie и Genre, таблицу ассоциации следует добавить над классами моделей в models.py:

```
    MovieGenre = Table(
        "movies_genres",
        Base.metadata,
        Column("movie_id", ForeignKey("movies.id"), primary_key=True,
nullable=False),
        Column("genre_id", ForeignKey("genres.id"), primary_key=True,
nullable=False),
    )
```

Конструктор Table принимает в качестве первого аргумента имя таблицы, за которым следует экземпляр метаданных базы данных. Остальные аргументы – это два экземпляра Column для внешних ключей. Поскольку таблицы используют конструктор Column() вместо подсказок, добавляется опция nullable=False, которая делает значения для этих столбцов обязательными.

В отличие от всех остальных таблиц, полученных из моделей, эта таблица не имеет первичного ключа id, а вместо этого объявляет два внешних ключа первичными ключами. Когда несколько столбцов объявляются первичными ключами, SQLAlchemy создает составной первичный ключ. По определению, первичные ключи должны быть уникальными, поэтому для данного отношения таблица не допустит двух строк с одинаковыми значениями внешних ключей.

Обновленная модель Movie и новая модель Genre показаны ниже:

```
    class Movie(Base):
        __tablename__ = "movies"

        id: Mapped[int] = mapped_column(primary_key=True)
```

```
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director_id: Mapped[int] = mapped_column(ForeignKey("directors.id"),
index=True)
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    rating: Mapped[float] = mapped_column()

    director: Mapped[Director] = relationship("Director",
back_populates="movies")

    genres: Mapped[list["Genre"]] = relationship(
        "Genre", secondary=MovieGenre, back_populates="movies"
    )

    def __repr__(self):
        return f'Movie({self.id}, "{self.title}")'


class Genre(Base):
    __tablename__ = "genres"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(32), unique=True)

    movies: Mapped[list["Movie"]] = relationship(
        "Movie", secondary=MovieGenre, back_populates="genres"
    )

    def __repr__(self):
        return f'Genre({self.id}, "{self.name}")'
```

В модели Movie произошли два изменения. Колонка с жанром была удалена, поскольку теперь жанры будут храниться в отдельной таблице.

Также, подобно атрибуту director, представляющему отношение «один ко многим», добавлен атрибут отношения genres для доступа к сущностям на другой стороне отношения «многие ко многим» с семантикой, похожей на список.

Аргумент secondary отношения relationship() сообщает SQLAlchemy, что это отношение поддерживается вторичной таблицей (таблицей ассоциации). Обратите внимание, что на таблицу ассоциации ссылаются непосредственно по ее имени, поэтому она должна быть определена выше классов модели. Опция secondary определяет отношение в режиме «многие ко многим», при этом SQLAlchemy автоматически добавляет и удаляет элементы из таблицы ассоциации по мере необходимости.

Новая модель Genre очень похожа на модель Director, только с первичным ключом и атрибутом name. Отношение movies в этой модели представляет собой обратный вид отношения «многие ко многим» и также инициализируется с таблицей ассоциации в аргументе secondary.

Как и раньше, опции back_populates в этих двух отношениях ссылаются друг на друга. Поэтому для SQLAlchemy очевидно, что это две стороны одного и того же отношения.

Запросы отношений «многие ко многим»

Объекты отношений movies и genres, добавленные в модели Genre и Movie соответственно, упрощают использование отношения «многие ко многим» в запросах.

Вот как можно получить фильм и его жанры:

```
movie = session.scalar( select(Movie).where(Movie.title ==
'Interstellar'))

movie

> Movie(18, "Interstellar")

movie.genres

> [Genre(8, "Sci-Fi"), Genre(5, "Adventure")]
```

Отношение genres использует ленивый загрузчик по умолчанию, поэтому оно неявно выполняет запрос для получения списка жанров при первом обращении к атрибуту.

Аналогично, по жанру можно определить фильмы:

```
genre = session.scalar(select(Genre).where(Genre.name == "Sci-Fi"))
genre

> Genre(8, "Sci-Fi")

genre.movies

> genre.movies
[Movie(9, "Inception"), Movie(10, "The Matrix"), Movie(18, "Interstellar"), Movie(23, "Terminator 2:
Judgment Day"), Movie(24, "Back to the Future"), Movie(35, "Alien")]
```

Переходя к более сложным задачам, приведем запрос, который возвращает все фильмы, имеющие несколько жанров, а также количество жанров в каждом из них:

```
genre_count = func.count(Genre.id).label(None)
query = (select(Movie, genre_count).join(Movie.genres).group_by(Movie).having(genre_count >=
2).order_by(Movie.title))
session.execute(query).all()

> [(Movie(35, "Alien"), 2), (Movie(22, "American History X"), 2), ... (Movie(31, "Whiplash"), 2)]
```

В этом запросе используются приемы, аналогичные тем, которые были рассмотрены при работе с отношениями «один ко многим». Запрос возвращает два значения в каждой строке результатов: фильм и количество жанров. Значение числа жанров создается с меткой и хранится в переменной genre_count, чтобы ее можно было использовать в выражениях select() и having() без повторений.

Для подсчета жанров необходимо соединить фильмы с жанрами. Группировка результатов по фильмам сворачивает результаты до одного фильма в строке. Для получения второго результата запускается агрегатная функция count() и заменяет список жанров на их количество. Выражение having() служит для фильтрации сгруппированных результатов и оставляет только те из них, в которых присутствует два или более жанров.

Метод join() в этом запросе интересен тем, что отношения «многие ко многим» не могут быть запрошены с помощью одного join. На самом деле в SQL невозможно напрямую соединить таблицы movies и genres, поскольку в них нет общих атрибутов, которые можно было бы использовать.

В реальности отношения «многие ко многим» требуют двухэтапного соединения. Сначала таблица movies соединяется с таблицей movies_genres, а затем movies_genres соединяется с genres.

Если интересно узнать о внутреннем устройстве, ниже вы можете увидеть SQL, сгенерированный этим запросом, включая два соединения, необходимые отношениям «многие ко многим»:

```
print(query)
```

```
> SELECT movies.id, movies.title, movies.director_id, movies.release_date, movies.duration,
movies.rating, count(genres.id) AS count_1
FROM movies JOIN movies_genres AS movies_genres_1 ON movies.id = movies_genres_1.movie_id
JOIN genres ON genres.id = movies_genres_1.genre_id GROUP BY movies.id, movies.title,
movies.director_id, movies.release_date, movies.duration, movies.rating
HAVING count(genres.id) >= :param_1 ORDER BY movies.title
```

Отношения «один ко многим» и «многие ко многим» можно использовать вместе. Следующий запрос позволяет получить список режиссеров, работающих в жанре Sci-Fi:

```
query = (select(Director).join(Director.movies).join(Movie.genres).where(Genre.name ==
"Sci-Fi").order_by(Director.name).distinct())
session.scalars(query).all()

> [Director(4, "Christopher Nolan"), Director(19, "James Cameron"), Director(9, "Lana Wachowski"),
Director(22, "Ridley Scott"), Director(8, "Robert Zemeckis")]
```

Цель этого запроса – вернуть список режиссеров, поэтому это единственная модель, которая добавляется в оператор select(). Но запрос требует доступа к жанрам, а жанры не имеют прямой связи с режиссерами. Единственным решением является навигация по доступным отношениям до тех пор, пока не будет достигнута связь. В данном случае сначала режиссеры соединяются со своими фильмами, а затем фильмы соединяются со своими жанрами. В результате этой цепочки соединений запрос имеет доступ ко всем допустимым тройкам (режиссер, фильм, жанр) и может добавлять фильтры по любой из них.

Выражение distinct() необходимо в этом случае, потому что многие из тех триплетов, у которых жанр – «Sci-Fi», будут иметь одного и того же режиссера, и если не задать это выражение, база данных вернет все строки, что приведет к дублированию результатов.

Группировка также применяется в цепочке отношений. Следующий запрос получает список режиссеров, которые работают более чем в двух жанрах, вместе с количеством жанров:

```
genre_count = func.count(Genre.id.distinct()).label(None)
query = (select(Director,
genre_count).join(Director.movies).join(Movie.genres).group_by(Director).having(genre_count >= 3))
session.execute(query ).all()

> [(Director(2, "Francis Ford Coppola"), 3), (Director(3, "Quentin Tarantino"), 3), ..., (Director(31,
"Stanley Kubrick"), 3)]
```

Агрегация genre_count в этом случае должна быть расширена выражением distinct(), потому что при подсчете строк будут включаться дубликаты записей, полученные в результате объединения.

Удаление из отношений «многие ко многим»

Отношения «многие ко многим», настроенные с помощью опции secondary, имеют то преимущество, что SQLAlchemy выполняет всю работу по обслуживанию таблицы ассоциации, и эта особенность распространяется и на удаление. Когда удаляется сущность, SQLAlchemy находит все сущности на другой стороне отношения, которые связаны с ней, и удаляет эти связи.

В следующем примере удаляется жанр, в результате чего он автоматически исчезает из фильмов, в которых был указан:

```
genre = session.get(Genre, 11)
genre

> Genre(11, "Mystery")

movie = session.get(Movie, 37)
movie

> Movie(37, "Memento")

movie.genres

> [Genre(9, "Thriller"), Genre(11, "Mystery")]

session.delete(genre)
session.commit()
movie.genres

> [Genre(9, "Thriller")]
```

Также можно отсоединить сущности, связанные отношениями «многие ко многим», не удаляя при этом ни одной из сущностей. Отсоединение может

быть выполнено с любой стороны отношения с использованием семантики списка:

```
genre = session.get(Genre, 1)
genre

> Genre(1, "Drama")

movie = session.get(Movie, 8)
movie

> Movie(8, "Forrest Gump")

movie.genres

> [Genre(7, "Romance"), Genre(1, "Drama")]

genre in movie.genres

> True

movie.genres.remove(genre)
session.commit()
movie.genres

> [Genre(7, "Romance")]

genre in movie.genres

> False
```

В приведенном выше примере жанр удаляется из фильма. Тот же результат можно получить, удалив фильм из жанра:

```
genre.movies.remove(movie)
```

Отношение «один ко многим» между режиссерами и их фильмами было сконфигурировано с ненулевым внешним ключом. Поэтому все фильмы должны иметь связанного с ними режиссера, фактически делая невозможным существование фильма без режиссера. В отношениях «многие ко многим» нет автоматического способа обеспечить условие, чтобы каждая сущность имела хотя бы одну связь с сущностью на другой стороне. В приведенном ниже

примере удаляется единственный жанр, связанный с фильмом, в результате чего фильм остается без связанных с ним жанров:

```
genre = session.get(Genre, 1)
genre

> Genre(1, "Drama")

movie = session.get(Movie, 1)
movie

> Movie(1, "The Shawshank Redemption")

movie.genres

> [Genre(1, "Drama")]

genre.movies.remove(movie)
session.commit()
movie.genres

> []
```

Для отношений «многие ко многим», для которых желательно, чтобы всегда существовала хотя бы одна связанная сущность, в приложении необходимо вручную выполнять проверки и предотвращать удаление последней связи.

ЗАКЛЮЧЕНИЕ

Библиотека SQLAlchemy представляет собой мощный и гибкий инструмент для работы с базами данных на языке программирования Python. Функциональные возможности библиотеки охватывают широкий спектр задач, начиная от выполнения SQL-запросов и заканчивая сложным объектно-реляционным отображением (ORM). SQLAlchemy позволяет разработчикам эффективно управлять базами данных, обеспечивая при этом высокую производительность и масштабируемость приложений.

Одним из ключевых преимуществ SQLAlchemy является ее модульная архитектура, которая позволяет использовать как низкоуровневый язык запросов, так и высокоуровневый ORM для работы с базами данных на уровне объектов. Эти особенности делают библиотеку универсальным инструментом, подходящим для различных сценариев использования: от разработки небольших проектов до построения крупных корпоративных систем.

Библиотека SQLAlchemy поддерживает множество различных СУБД, что обеспечивает ее широкую применимость в различных средах и проектах. Встроенные механизмы миграции схемы базы данных, такие как Alembic, позволяют легко управлять изменениями структуры данных, что особенно важно в условиях динамично развивающихся приложений.

СПИСОК ИСТОЧНИКОВ

1. https://www.sqlalchemy.org

ПРИЛОЖЕНИЯ

Модели версии № 1:

```
from sqlalchemy import String, Date
from sqlalchemy.orm import Mapped, mapped_column
from db import Base
import datetime


class Movie(Base):
    __tablename__ = "movies"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director: Mapped[str] = mapped_column(String(64))
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    genre: Mapped[str] = mapped_column(String(32))
    rating: Mapped[float] = mapped_column()

    def __repr__(self):
        return f'Movie({self.id}, "{self.title}")'
```

Скрипт импорта № 1:

```
import datetime
import csv
from db import Base, Session, engine
from models import Movie


def main():
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)

    with Session() as session:
        with session.begin():
            with open("movies.csv") as f:
                reader = csv.DictReader(f)
                for row in reader:
                    try:
                        row["release_date"] = datetime.datetime.strptime(
                            row["release_date"], "%Y-%m-%d"
                        ).date()
                        row["duration"] = int(row["duration"])
                        row["rating"] = float(row["rating"])
                        movie = Movie(**row)
                        session.add(movie)
                    except ValueError as e:
                        print(f"Error processing row: {row}")
                        print(f"Error message: {str(e)}")
                        continue


if __name__ == "__main__":
    main()
```

Модели версии № 2:

```
from sqlalchemy import String, Date, ForeignKey
from sqlalchemy.orm import Mapped, mapped_column, relationship
from db import Base
import datetime


class Director(Base):
    __tablename__ = "directors"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), unique=True)

    movies: Mapped[list["Movie"]] = relationship("Movie", back_populates="director")

    def __repr__(self):
        return f'Director({self.id}, "{self.name}")'


class Movie(Base):
    __tablename__ = "movies"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director_id: Mapped[int] = mapped_column(ForeignKey("directors.id"), index=True)
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    genre: Mapped[str] = mapped_column(String(32))
    rating: Mapped[float] = mapped_column()

    director: Mapped[Director] = relationship("Director", back_populates="movies")

    def __repr__(self):
        return f'Movie({self.id}, "{self.title}")'
```

Скрипт импорта № 2:

```
import datetime
import csv
from db import Base, Session, engine
from models import Movie, Director


def main():
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)

    with Session() as session:
        with session.begin():
            with open("movies.csv") as f:
                reader = csv.DictReader(f)
                all_director_names = {}
                for row in reader:
                    try:
                        row["release_date"] = datetime.datetime.strptime(
                            row["release_date"], "%Y-%m-%d"
                        ).date()
                        row["duration"] = int(row["duration"])
                        row["rating"] = float(row["rating"])
                        director_name = row.pop("director")
                        movie = Movie(**row)
                        if director_name not in all_director_names:
                            director = Director(name=director_name)
                            session.add(director)
                            all_director_names[director_name] = director
                        all_director_names[director_name].movies.append(movie)
                    except ValueError as e:
                        print(f"Error processing row: {row}")
                        print(f"Error message: {str(e)}")
                        continue


if __name__ == "__main__":
    main()
```

Модели версии № 3:

```
from sqlalchemy import String, Date, ForeignKey, Table, Column, Integer, Float
from sqlalchemy.orm import Mapped, mapped_column, relationship
from db import Base
import datetime

MovieGenre = Table(
    "movies_genres",
    Base.metadata,
    Column("movie_id", ForeignKey("movies.id"), primary_key=True, nullable=False),
    Column("genre_id", ForeignKey("genres.id"), primary_key=True, nullable=False),
)


class Director(Base):
    __tablename__ = "directors"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), unique=True)

    movies: Mapped[list["Movie"]] = relationship("Movie", back_populates="director")

    def __repr__(self):
        return f'Director({self.id}, "{self.name}")'


class Movie(Base):
    __tablename__ = "movies"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), unique=True)
    director_id: Mapped[int] = mapped_column(ForeignKey("directors.id"), index=True)
    release_date: Mapped[datetime.date] = mapped_column(Date)
    duration: Mapped[int] = mapped_column()  # in minutes
    rating: Mapped[float] = mapped_column()
```

```
    director: Mapped[Director] = relationship("Director", back_populates="movies")

    genres: Mapped[list["Genre"]] = relationship(
        "Genre", secondary=MovieGenre, back_populates="movies"
    )

    def __repr__(self):
        return f'Movie({self.id}, "{self.title}")'


class Genre(Base):
    __tablename__ = "genres"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(32), unique=True)

    movies: Mapped[list["Movie"]] = relationship(
        "Movie", secondary=MovieGenre, back_populates="genres"
    )

    def __repr__(self):
        return f'Genre({self.id}, "{self.name}")'
```

Скрипт импорта № 3:

```
import datetime
import csv
from db import Base, Session, engine
from models import Movie, Director, Genre, MovieGenre


def main():
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)

    with Session() as session:
        with session.begin():
            with open("movies.csv") as f:
                reader = csv.DictReader(f)
                all_director_names = {}
                all_genre_names = {}
                for row in reader:
                    try:
                        row["release_date"] = datetime.datetime.strptime(
                            row["release_date"], "%Y-%m-%d"
                        ).date()
                        row["duration"] = int(row["duration"])
                        row["rating"] = float(row["rating"])

                        director_name = row.pop("director")
                        genre_names = row.pop("genre").split()

                        movie = Movie(**row)

                        if director_name not in all_director_names:
                            director = Director(name=director_name)
                            session.add(director)
                            all_director_names[director_name] = director
                        all_director_names[director_name].movies.append(movie)
```

```
                for genre_name in genre_names:
                    if genre_name not in all_genre_names:
                        genre = Genre(name=genre_name)
                        session.add(genre)
                        all_genre_names[genre_name] = genre
                    all_genre_names[genre_name].movies.append(movie)

            except ValueError as e:
                print(f"Error processing row: {row}")
                print(f"Error message: {str(e)}")
                continue


if __name__ == "__main__":
    main()
```

ОГЛАВЛЕНИЕ

ЕЛИСЕЕВ Алексей Игоревич
ПОЛЯКОВ Дмитрий Вадимович

РАБОТА С БАЗАМИ ДАННЫХ НА ЯЗЫКЕ Python С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ SQLAlchemy 2.0

Учебное пособие

Редактирование Е. С. Мордасовой
Графический и мультимедийный дизайнер Н. И. Кужильная
Обложка, упаковка, тиражирование Е. С. Мордасовой

ISBN 978-5-8265-2820-4

Подписано к использованию 15.10.2024.
Тираж 50 шт. Заказ № 110

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел./факс (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru