Тема 9.6: Python и БД. ORM

План занятия:

1. Концепт и инициализация ORM
2. Подключение и session
3. Связанные таблицы
4. Выборка и наполнение
5. Мотивация использовать ORM
6. SQLAlchemy vs Django ORM

ORM (Object-Relational Mapping) — это технология программирования, которая обеспечивает взаимодействие между объектно-ориентированным программированием (ООП) и реляционными базами данных (РБД). ORM позволяет разработчикам работать с объектами в коде, а система автоматически выполняет преобразование данных между объектами и записями в базе данных.

Основные принципы ORM:

1.Объекты вместо таблиц: ORM позволяет использовать объекты, представляющие сущности в приложении, вместо явной работы с таблицами в базе данных. Например, объект "Пользователь" в коде может соответствовать записи в таблице "users" в базе данных.

2.Преобразование данных: ORM автоматически обеспечивает преобразование данных между объектами и записями в базе данных. Это включает в себя сохранение объектов в базу данных (INSERT), загрузку объектов из базы данных (SELECT), а также обновление (UPDATE) и удаление (DELETE) данных.

3.Отсутствие прямой работы с SQL: Разработчикам не нужно явным образом писать SQL-запросы для взаимодействия с базой данных. ORM обеспечивает более высокий уровень абстракции, что упрощает разработку и поддержку кода.

4.Упрощение кода и повышение читаемости: ORM делает код более чистым и читаемым, уменьшает количество необходимого кода для доступа к данным, а также упрощает изменение структуры базы данных без необходимости изменения всего кода приложения.

ООП

Объектно-ориентированное программирование (ООП) - это парадигма программирования, в основе которой лежит концепция объектов, представляющих сущности в программе, и взаимодействия между этими объектами.

В ООП основными строительными блоками являются классы и объекты.

Класс

Класс - это шаблон или чертеж, описывающий структуру и поведение объектов. Он содержит описание атрибутов и методов, которые будут унаследованы объектами, созданными на основе этого класса.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")
```

Экземпляр

Экземпляр - это конкретный объект, созданный на основе определенного класса. Он представляет собой конкретный экземпляр объекта, обладающий своими уникальными значениями атрибутов.

```
my_dog = Dog(name="Buddy", age=3)
```

Атрибут

Атрибут - это переменная, хранящая данные, связанные с объектом. Атрибуты определяют характеристики объекта.

```
print(my_dog.name)  # Выводит значение атрибута "name" объекта my_dog
```

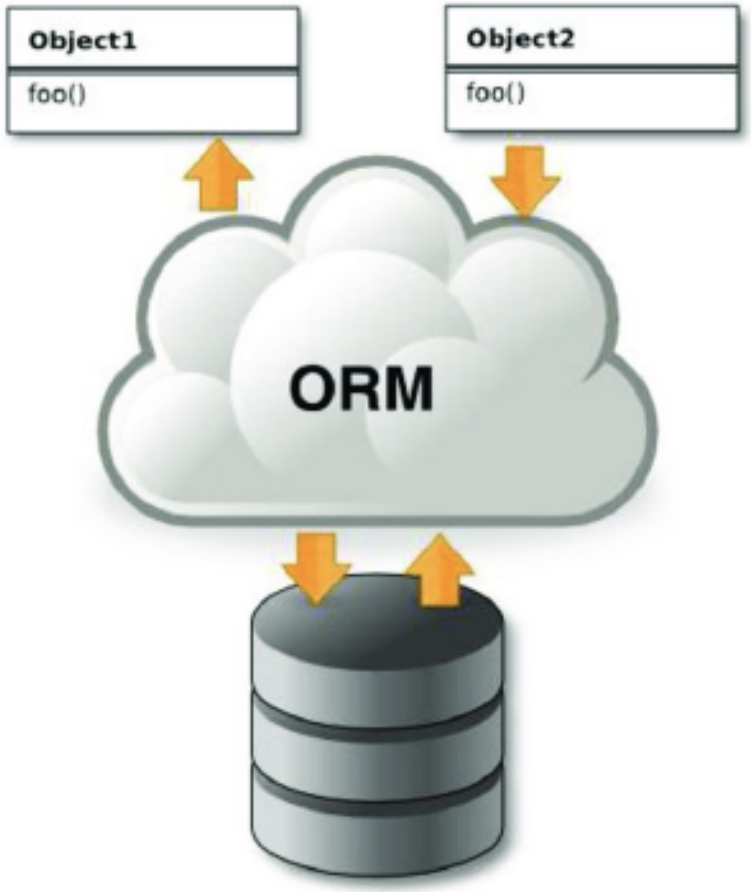
Метод

Метод - это функция, определенная внутри класса, предназначенная для выполнения операций с объектом. Методы представляют собой действия, которые объект может выполнить.

```
my_dog.bark()  # Вызывает метод "bark" объекта my_dog
```

Object-Relational Mapping

ORM (объектно-реляционное отображение) — это дополнительный способ взаимодействия с БД из кода, который работает с таблицами и запросами к БД, как с классами, объектами и методами в ООП.



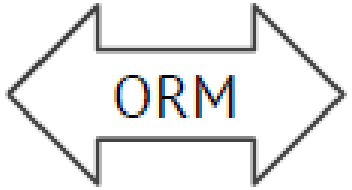
ORM — не панацея, и разработчики могут умышленно отказываться от ORM в пользу сырых SQL запросов, если им так удобнее.

ORM

Чтобы начать работу с базой через ORM, необходимо описать таблицы в виде классов, наследуясь от общего класса Base для всей схемы.

Таблица artist

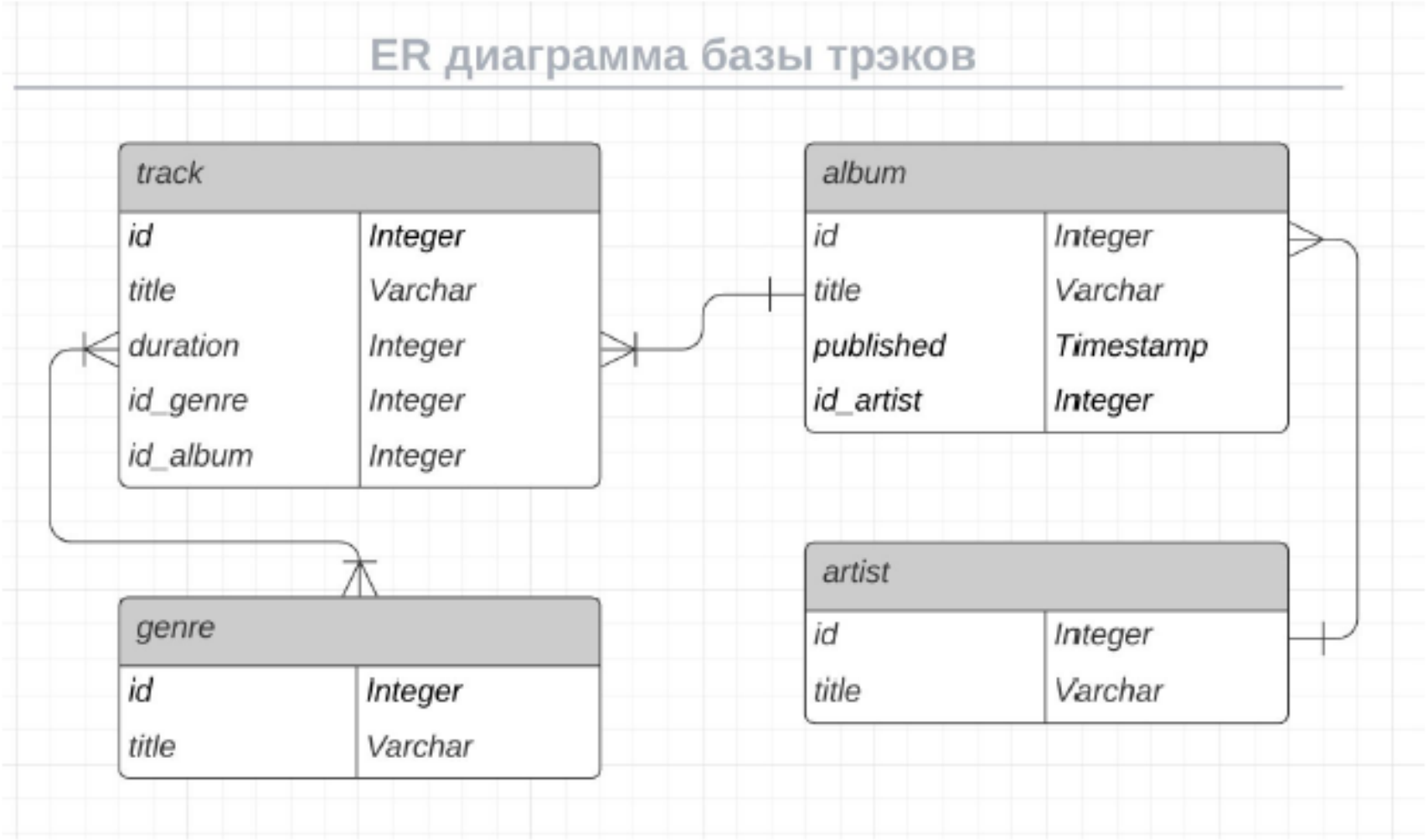
id	name
1	Queen
2	AC/DC
3	The Beatles




```python
import sqlalchemy as sq

Base = declarative_base()

class Artist(Base):
    __tablename__ = 'artist'

    id = sq.Column(sq.Integer, primary_key=True)
    name = sq.Column(sq.String)

queen = Artist(name='Queen')
ac_dc = Artist(name='AC/DC')
beatles = Artist(name='The Beatles')
```

Обёртка

ORM работает поверх стандартных SQL-запросов и закономерности реляционных связей остаются.



Вариант схемы хранения музыкальных треков для дальнейших запросов.

Инициализация

```
import sqlalchemy as sq

Base = declarative_base()

class Album(Base):
    __tablename__ = 'album'

    id = sq.Column(sq.Integer, primary_key=True)
    title = sq.Column(sq.String)
    tracks = relationship('Track', back_populates='album')
    published = sq.Column(sq.Date)

    id artist = sq.Column(sq.Integer, sq.ForeignKey('artist.id'))
    artist = relationship(Artist, back_populates='albums')

class Track(Base):
    __tablename__ = 'track'

    id = sq.Column(sq.Integer, primary_key=True)
    title = sq.Column(sq.String)
    duration = sq.Column(sq.Integer)
    genres = relationship(Genre, secondary=track to genre,
                                 back_populates='tracks')

    id_album = sq.Column(sq.Integer, sq.ForeignKey('album.id'))
    album = relationship(Album, back_populates='tracks')
```

Запросы

В запросах к БД через ORM необходимо использовать созданные классы и параметры, которые укажут на нужные таблицы и их свойства.

```
>>> query = session.query(Track).join(Track.album).join(Album.artist).filter(
    Artist.name == 'The Beatles'
)
>>> query.all()
[<__main__.Track object at 0x1087efdc0>, <__main__.Track object at 0x108843070>]
```

для фильтра в примере использовался класс *Artist* и его пареметр *name*.
Метод *.all* совершил запрос в БД и выдал на консоль все подходящие трэки хранимые в БД.

Распечатав *query*, мы можем увидеть настоящий запрос в БД, выполняемый ORM.
Это полезно для анализа.

```
>>> print(query)
SELECT track.id AS track_id, track.title AS track_title, track.duration AS track_duration,
track.id_album AS track_id_album
FROM track JOIN album ON album.id = track.id_album JOIN artist ON artist.id = album.id_artist
WHERE artist.name = %(name_1)s
```

Подключение и session

Для запросов через ORM так же, как и в обычном подключении, необходимо создать экземпляр движка (драйвера) для подключения и сессию для запроса.

Подключение **без** ORM:

```
DSN = 'postgresql://postgres:admin@localhost:5432/postgres'

connect = psycopg2.connect(DSN)
with connect.cursor() as cursor:
    cursor.execute("SELECT * FROM tracks")
```

psycopg2 — модуль Python (драйвер) для подключения к PostgreSQL

SQLalchemy так же способен выполнять «сырые» запросы SQL:

```
engine = sqlalchemy.create_engine(DSN)
with engine.connect() as connection:
    result = connection.execute("SELECT * FROM tracks")
```

*Это как носить велосипед **на** руках. Не используется по назначению, но **иногда** необходимо.*

Функциональное использование **ORM** в SQLalchemy:

```
engine = sqlalchemy.create_engine(DSN)
Session = sessionmaker(bind=engine)
result = Session().query(Tracks).all()
```

Связанные таблицы

One To Many

Связь **один-к-многим** создается через параметр **ForeignKey** в связываемой таблице и **relationship** в экземпляре родителя.

```
import sqlalchemy as sq

class Album(Base):
    __tablename__ = 'album'
    id = sq.Column(sq.Integer, primary_key=True)
    tracks = relationship('Track', backref='album')

class Track(Base):
    __tablename__ = 'track'
    id = sq.Column(sq.Integer, primary_key=True)
    id_album = sq.Column(sq.Integer, sq.ForeignKey('album.id'))
```

Для доступа к объекту родителя из экземпляра ребенка используется **необязательный** параметр *backref*.

https://docs.sqlalchemy.org/en/13/orm/basic_relationships.html

Many To One, One To One

Для обратной связи **многие-к-одному**, вторичный ключ помещается в родительскую таблицу там же, где был relationship.

```
import sqlalchemy as sq

class Album(Base):
    __tablename__ = 'album'
    id = sq.Column(sq.Integer, primary_key=True)
    tracks = relationship('Track')
    id_track = sq.Column(sq.Integer, sq.ForeignKey('track'.id'))

class Track(Base):
    __tablename__ = 'track'
    id = sq.Column(sq.Integer, primary_key=True)
```

Инициализация **один-к-одному** аналогична один-к-многим с добавлением ключа *uselist* в функции *relationship*, для сопоставления одного элемента, а не списка:

```
class Album(Base):
    __tablename__ = 'album'
    id = sq.Column(sq.Integer, primary_key=True)
    tracks = relationship('Track', uselist=False)
```

Пример не по схеме, если бы у одного артиста допускался только один альбом за всю карьеру.

Many To Many

Связь **многие-к-многим** достигается через создание промежуточной таблицы, которая, условно, имеет связи ManyToOne и OneToMany, между целевыми таблицами.

```
import sqlalchemy as sq

class Genre(Base):
     tablename    = 'genre'
   id = sq.Column(sq.Integer, primary key=True)
   tracks = relationship('Track', secondary='track_to_genre')

track_to_genre = sq.Table(
   'track_to_genre', Base.metadata,
   sq.Column('genre_id', sq.Integer, sq.ForeignKey('genre.id')),
   sq.Column('track_id', sq.Integer, sq.ForeignKey('track.id')),
)

class Track(Base):
   __tablename__ = 'track'
   id = sq.Column(sq.Integer, primary_key=True)
   genres = relationship(Genre, secondary=track_to_genre)
```

Промежуточная таблица ассоциации указывается в параметр relationship.secondary.

Объект **Table** — практический синоним класса Base в другом синтаксисе, в данной схеме он необходим только для настройки связи многие-к-многим.

Выборка и наполнение БД

Query

Запрос можно создать через класс Query иди из метода query в объекте сессии:

```
session = Session()

# query = Query([Track], session=session)
query = session.query(Track)
```

Экземпляр запроса имеет множество сервисных методов:

```
query.filter(Track.duration > 120).all()

query.filter(Track.duration > 120).first()
```

Пример вложенного запроса:

```
subq = session.query(Genre).filter(Genre.title == 'blues').subquery()
q = session.query(Track).join(subq, Track.id == subq.c.genre_id)
```

Create, Update, Delete

Новые экземпляры классов могут быть добавлены в сессию для дозаписи в БД.

```
beatles = Artist(name='The Beatles')
album_1 = Album(title='Please Please Me', artist=beatles,
                published=date.fromisoformat('1963-03-22'))
album_2 = Album(title='With the Beatles', artist=beatles,
                published=date.fromisoformat('1987-02-26'))


session.add(beatles)
session.add_all([album_1, album_2])
session.commit()
```

Create, Update, Delete

Вспомогательные функции поиска помогают определить выборки для обновления:

```
session.query(Album).filter(Album.title == 'With the Beatles').update(
     {"published": date.fromisoformat('1963-11-22')}
)
session.commit()
```

или удаления:

```
session.query(Artist).filter(Artist.name == 'Black Beatles').delete()
```

https://docs.sqlalchemy.org/en/13/orm/query.html

Мотивация использовать ORM

- ✚ Необязательно знать SQL и специальные функции СУБД.
- ✚ Возможность десериализации (распаковки) результата из БД в удобном формате для API или обработки. Не нужно придумывать сложные парсеры массивов, а работаем с готовыми структурами данных.
- ✚ Один и тот же код может работать в разных БД, если на это рассчитана ORM
  - SQLAlchemy работает с диалектами:

    PostgreSQL, MySQL, SQLite, Oracle, Microsoft SQL Server, и др.
- ✚ Разработчиками ORM продуманы примитивные вопросы безопасности, экранирования и оптимизации запросов
- ➖ Изначально не очевидны итоговые запросы.
- ➖ Ограничения и читаемость при построении сложных, многоуровневых запросов.

SQLAlchemy vs Django ORM

Django ORM использует паттерн active record, а SQLAlchemy – data mapper.

active record – каждая строка в базе оборачивается в отдельный python-объект.

+ Проще для понимания и популярен.
+ Хорошо интегрирован в экосистему Django, работает из коробки.
- Нельзя использовать отдельно от Django, поддерживается сообществом Django.
- Почти невозможно писать комплексные (сложные) запросы.

data mapper – позволяет управлять отображением из БД.

+ Больше возможностей, писать запросы любой сложности.
+ Независим от фреймворка, популярен и поддерживается всем OpenSource.
- Требует большего опыта для настройки конфигураций
- Относительная сложность форматирования.