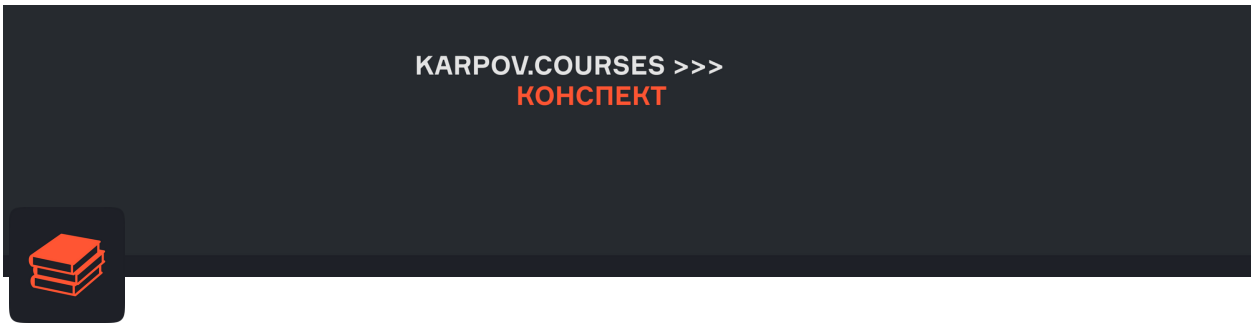


> Конспект > 10 урок > Базы данных в Python: ORM

Оглавление 10 урока

>SQLAlchemy

ORM (Object-Relational Mapping, объектно-реляционное отображение) — технология программирования, суть которой заключается в создании «виртуальной объектной базы данных».

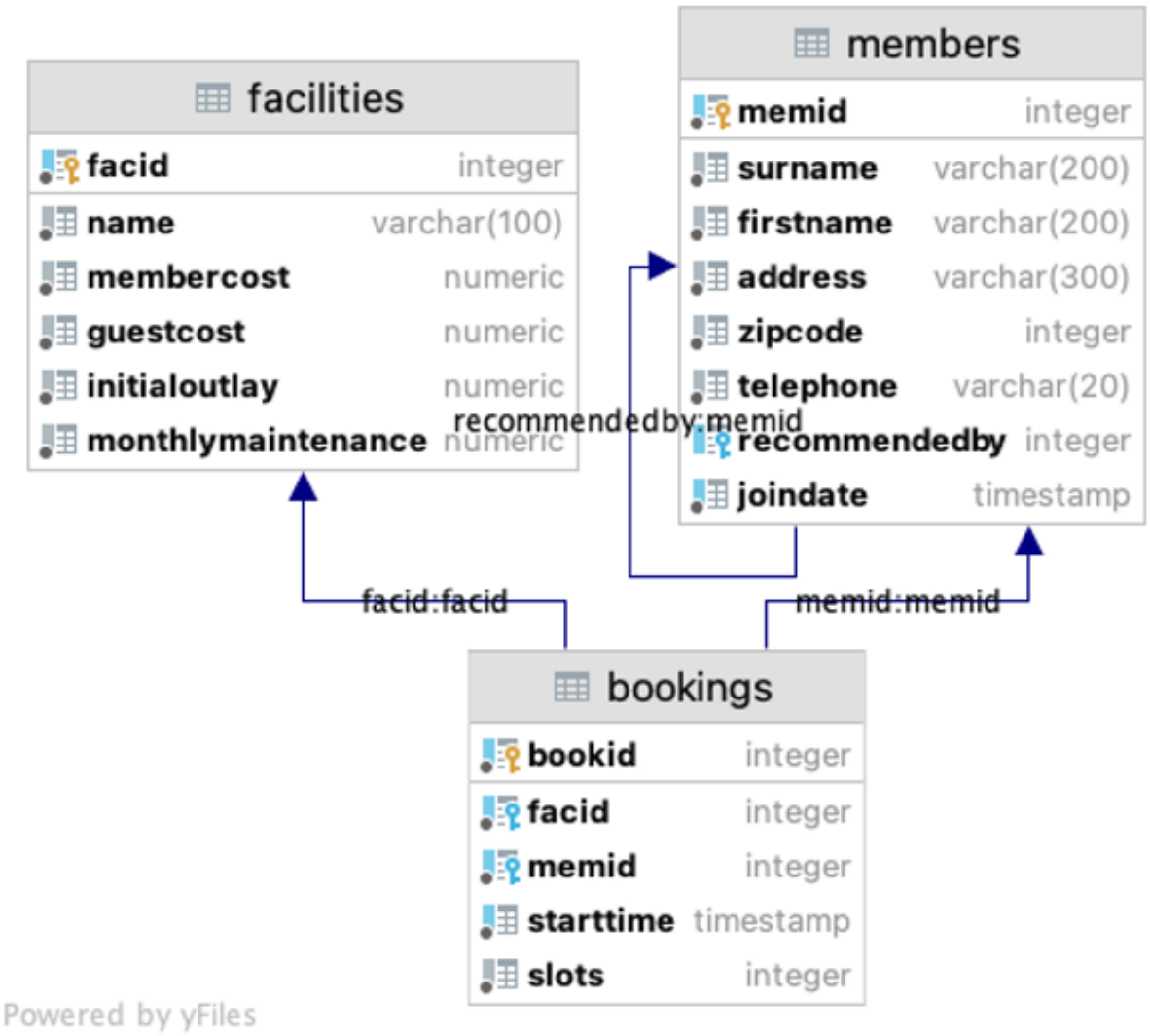
Иными словами, можно обращаться к объектам классов для управления данными в таблицах БД. Также можно создавать, изменять, удалять, фильтровать и, самое главное, наследовать объекты классов, сопоставленные с таблицами БД, что существенно сокращает наполнение кодовой базы.

SQLAlchemy одна из самых популярных ORM на сегодня.

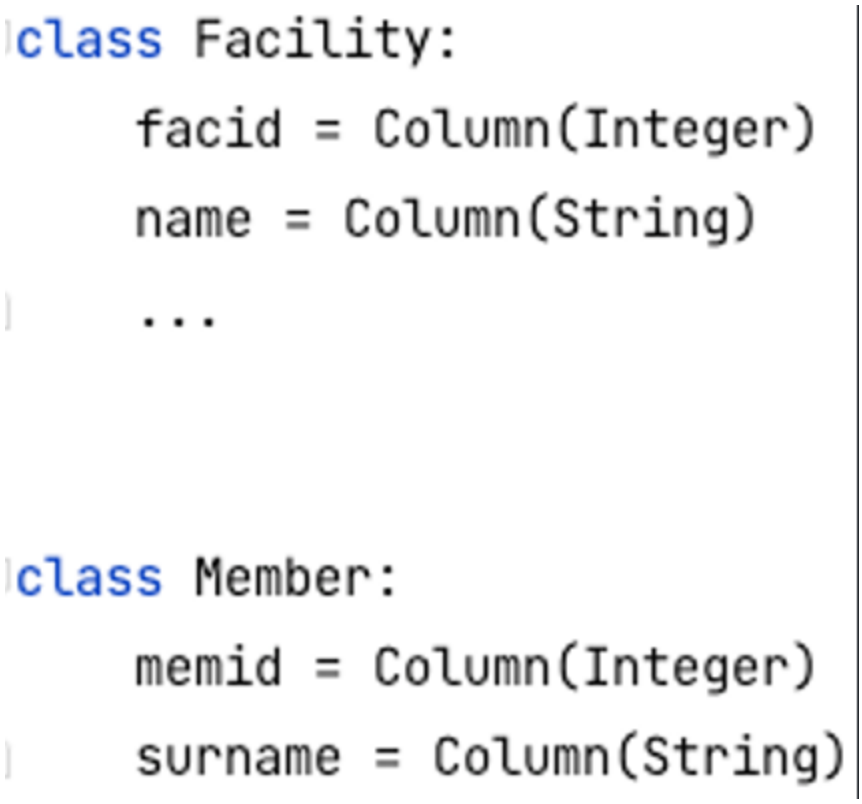
SQLAlchemy предоставляет хороший способ взаимодействия с базами данных с помощью **Python** . Таким образом, вместо того, чтобы иметь дело с различиями

между диалектами традиционного **SQL** (MySQL, PostgreSQL или Oracle), вы можете использовать **SQLAlchemy** для более эффективной работы с данными.

Есть 3 таблицы: facilities, members(члены клуба) и bookings(таблица с записями, которая соединяет facilities и members). Так это выглядело в SQL диаграмме:



В python это будет выглядеть примерно так:



Создадим по классу на каждую таблицу, т.е будет класс Facilities, класс Members и класс Bookings. И далее будем работать с этими объектами как с питоновскими классами.

Создаем простое приложение с интеграцией с **SQLAlchemy:**

```
from sqlalchemy import Column, Integer, String, Boolean, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# NOTE be careful when sharing
SQLALCHEMY_DATABASE_URL = "postgresql://username:password@localhost/postgres"
```

Импортируем нужные библиотеки и подключимся к базе данных

`SQLALCHEMY_DATABASE_URL` - строка для подключения к базе данных:

- `postgresql` — это имя базы данных, диалект (может быть mysql, postgresql, mssql, oracle и т.д).
- `username` и `password` — данные для получения доступа к базе данных.

- `localhost` — расположение сервера базы данных.
- `database` — название базы данных.

```
# создаём engine
engine = create_engine(SQLALCHEMY_DATABASE_URL)
# настройка класса Session с требуемыми настройками
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Далее создаем `engine` - прослойку **SQLAlchemy** которая уничтожает все различия между бд.

SessionLocal – фабрика для создания экземпляров **Session** с заданными параметрами. Вместо того, чтобы каждый раз указывать список аргументов у сессии, его достаточно один раз указать у фабрики, а дальше уже создавать сессии без указания аргументов.

Сессии налаживают обмен данными с бд и предоставляют «holding zone» (зону проведения) для всех объектов, которые вы загрузили или связываете с базой в течение рабочего цикла. Анологом сессий в **SQLAlchemy** является система контроля версий Git.

`bind=engine` в аргументах `sessionmaker()` означает, что сессии которые мы создаем должны быть привязаны к движку(который ранее создали)

Каждый класс, представляющий таблицу в БД, должен наследоваться от базового класса который создается при помощи функции `declarative_base()`. Это тот скелет, по которому мы и создаём модели. Класс, на основе которого создаются другие классы

Опишем таблицу:

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String)
    surname = Column(String)
    age = Column(Integer)
    sex = Column(Boolean)
```

Класс `User` наследуется от базового класса `Base`.

Таблицы в SQLAlchemy представлены в виде экземпляров класса(в данном случае класса `User`). Его конструктор принимает название таблицы, метаданные и одну или несколько колонок. Разберём подробнее:

Перед созданием таблицы импортируем несколько типов из **SQLAlchemy**, которые используются для создания колонки.

Далее, создается схема таблицы. Колонки создаются с помощью экземпляра `Column`. Конструктор этого класса принимает название колонки, тип данных, еще можно передать дополнительные аргументы для обозначения ограничений и конструкций SQL(`primary_key`, `nullable`, `default`). Также, **SQLAlchemy** не до конца понимает название таблицы, которую вы хоите создать, поэтому его необходимо заранее прописать с помощью свойства `__tablename__`.

В примере выше была определена колонка `sex` которая описывает пол. Ее тип — `Boolean`. Это общий тип. Для базы данных PostgreSQL тип будет `boolean`. А для MySQL — `SMALLINT`, т.к там нет Boolean. В Python же этот тип данных представлен типом `bool` (`True` или `False`).

Основные типы в SQLAlchemy и аналоги в Python и SQL:

SQLAlchemy	SQL	Python
Text	**TEXT**	**str**
Float	**FLOAT\REAL**	**float**
Integer	**INTEGER**	**int**
Date	**DATE**	**datetime.date**
Boolean	**BOOLEAN\SMALLINT**	**bool**
BigInteger	**BIGINT**	**int**

Посмотреть эти типы можно в документации или вызвав sqlalchemy.types

Чтобы **SQLAlchemy** создала таблицу нужно явно об этом попросить:

```
if __name__ == "__main__":
    Base.metadata.create_all(engine)
```

Официальная документация **SQLAlchemy** (обязательно посмотрите)

Подробнее о сессиях

>Создание таблиц

Для начала стоит активировать виртуальное окружение.

Создаётся окружение командой:

```
python -m venv имя_окружения
```

Далее, активация окружение командой (**windows**):

```
имя_окружения\Scripts\activate
```

Далее, активация окружение командой (**Linux** и **MacOS**):

```
source имя_окружения/bin/activate
```

Будьте внимательны, все последующие команды будет выполнять не системный **python**, а **python** в виртуальном окружении.

Ещё, у Python есть такая особенность, что если вы хотите, чтобы все файлы импортировались без проблем, то в папке нужно создать файл `__init__.py` тогда в глазах python эта папка будет выглядеть как модуль. Сохраним написанный ранее скрипт `simple_model` в папке `examples`. Теперь мы можем запустить этот скрипт в командной строке.

Отлично, мы разметили базу данных, теперь можем с ней работать(манипулировать, создавать, забирать ее объекты).

Теперь, в **новом файле**, который называется `crud.py` мы можем импортировать написанный нами ранее `User` и объект создающий сессии - `SessionLocal`.

```
from examples.simple_model import SessionLocal, User
```

Давайте создадим пользователя с именем `aleksei`, фамилией `random` и возрастом `18`

```
if __name__ == "__main__":
    user = User(name="aleksei", surname="random", age=18)
```

Создали пользователя, теперь надо его добавить. Для этого нам и понадобится сессия(объект, создающий сессии мы уже импортировали).

```
session = SessionLocal()
session.add(user)
session.commit()
```

Просто добавление объектов(`add()`) не влияет на запись в базу, а лишь готовит объекты к сохранению в следующем коммите. Для сохранения данных используется метод `commit()`.

И теперь, если мы вызовем таблицу, то увидим добавленные строки. Сейчас таблица выглядит вот так:

id	name	surname	age
1	aleksei	random	18

Отношения

Создадим таблицу `Facility`:

```
class Facility(Base):
    __tablename__ = "facilities"
    __table_args__ = {"schema": "cd"}
    id = Column(Integer, primary_key=True, name="facid")
    name = Column(String)
    member_cost = Column(Float, name="membercost")
    guest_cost = Column(Float, name="guestcost")
    initial_outlay = Column(Float, name="initialoutlay")
    monthly_maintenance = Column(Float, name="monthlymaintencance")
```

и таблицу `Booking` :

```
class Booking(Base):
    __tablename__ = "bookings"
    __table_args__ = {"schema": "cd"}
    id = Column(Integer, primary_key=True, name="bookid")
    facility_id = Column(
        Integer, ForeignKey("cd.facilities.facid"), primary_key=True, name="facid")
    facility = relationship("Facility")
    start_time = Column(TIMESTAMP, name="starttime")
    slots = Column(Integer)
```

Строчка `facility_id = Column(Integer, ForeignKey("cd.facilities.facid"))` устанавливает отношение один-ко-многим между моделями Booking и Facility.

Функция `relationship()` добавляет атрибуты в модели для доступа к связанным данным.

Строчка `facility = relationship("Facility")` добавляет атрибут `facility` классу `Booking`.

>PYTHONPATH

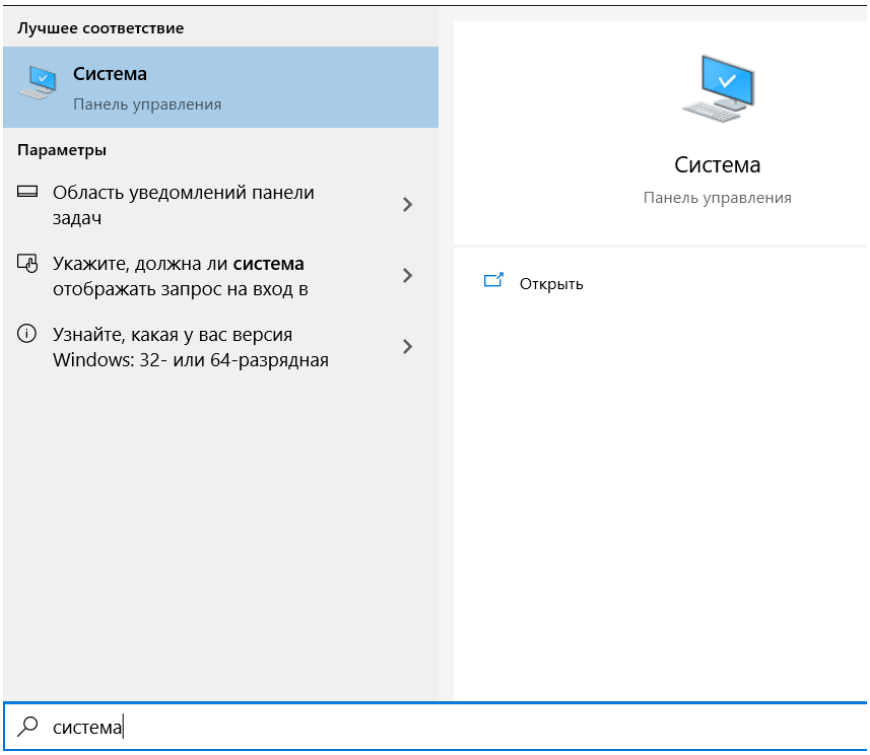
Но, если вы запустите `crud.py` в командной строке, то столкнётесь с ошибкой и сообщением, что модуль не найден. Связанно это с тем, что **Python** по-умолчанию ищет все библиотеки, которые прописаны в переменной **PYTHONPATH**. И если ваша папка не находится в **PYTHONPATH**, то Python ничего не сможет из нее импортировать. Папку, в которой мы работаем надо явно добавить в **PYTHONPATH**.

На MacOS и Linux для этого нужно ввести

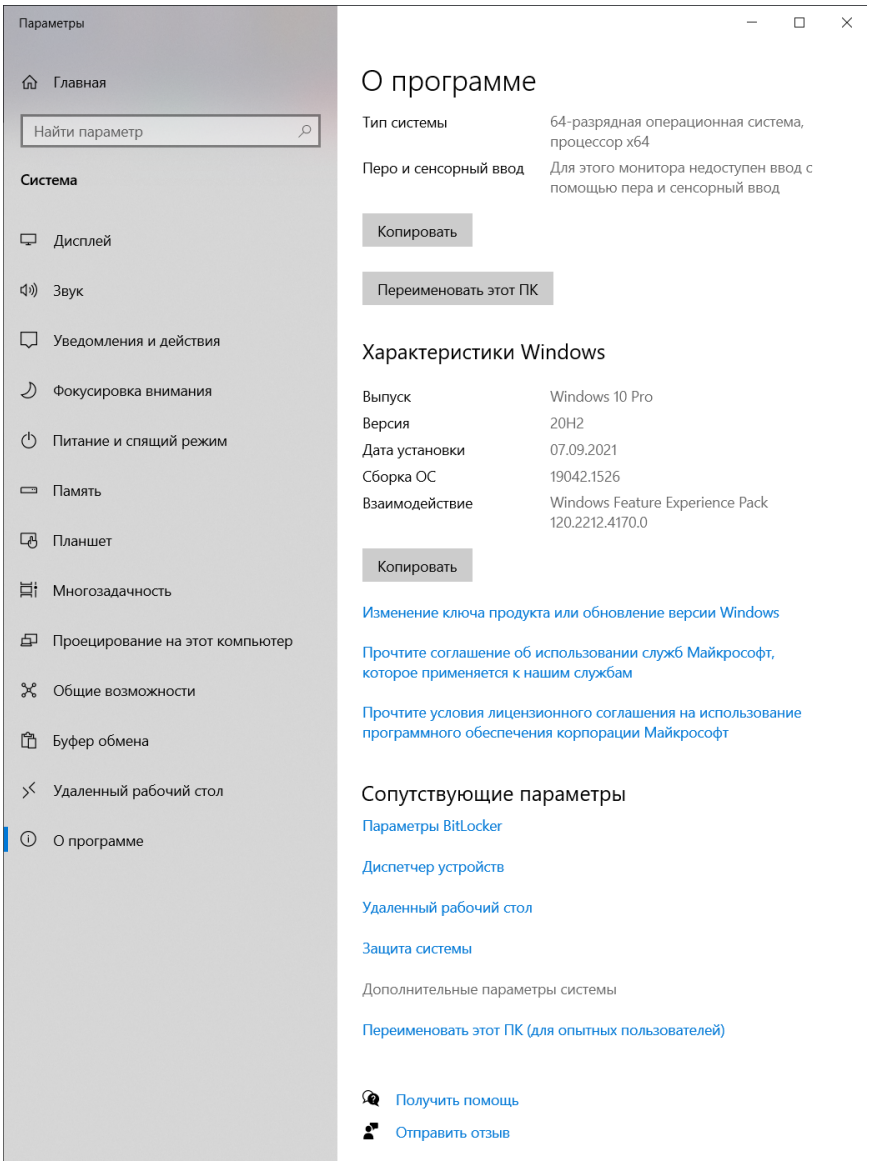
```
export PYTHONPATH=$PYTHONPATH:/some/path/
```

На Windows немного сложнее:

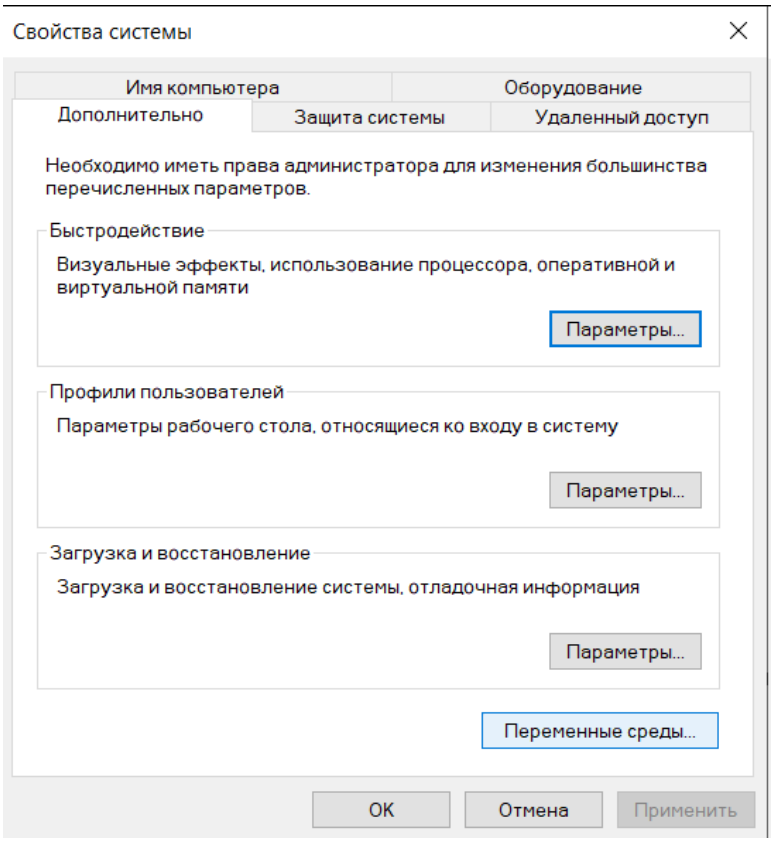
- В строке "Поиск" выполните поиск: Система



- Нажмите на ссылку **Дополнительные параметры системы**.



- Нажмите **Переменные среды**.



- В разделе **Переменные среды** выберите переменную среды `PYTHONPATH`. Нажмите **изменить**
- В окне **Изменение системной переменной** (или **Новая системная переменная**) укажите значение переменной среды `PYTHONPATH` (ваша папка). Нажмите **ОК**. Закройте остальные открытые окна, нажимая **ОК**.

Также, помимо добавления папки в PYTHONPATH, надо еще добавить в папку файл `__init__.py`. Это маркер пакета(если в папке нет `__init__.py` то это не пакет для python). Т.е python по факту наличия `__init__.py` в папке понимает, что это не просто папка, а модуль.

Грубо говоря, `__init__.py` нужен для того что бы можно было загружать именно пакет и модель работы пакетов функционировала.

>Работа с таблицей

Мы создали объект и добавили его в базу данных. Но это не всё. С таким же успехом мы можем забирать объекты из базы данных с помощью `session.query()`. С указание названия таблички в качестве аргумента. Метод `query()` объекта `session` возвращает объект типа `sqlalchemy.orm.query.Query`. Далее можно указать какие-то требования с помощью `.filter()`. Например:

```
print(
    session.query(User)
    .filter(User.name == "aleksei")
    .filter(User.age == 18)
    .limit(2)
    .all()
      )
```

Данный код выведет таблицу Юзеров с именем "aleksei", и возрастом 18. Пользуясь ORM можно делать не только фильтры. Можно делать всё то, что вы привыкли делать к SQL (`GROUP BY`, `HAVING`, `ORDER BY`).

Как мы уже выяснили, `session.query()` возвращает объект типа `sqlalchemy.orm.query.Query`. Вот распространенные методы этого класса:

- `all()` - Возвращает результат запроса в виде списка
- `filter()` - Этот метод позволяет отфильтровать результаты. В качестве аргументов принимает колонку, оператор и значение. SQL-эквивалент - оператор WHERE. Можно комбинировать условия с помощью союзов `and_()`, `or_()` и `not_()`.

  ```
  # найти юзеров с именем aleksei, наличием возраста и фамилией НЕ Random
  print(
      session.query(User)
      .filter(
            and_(User.name == 'aleksei', Order.age != None,
            not_(User.last_name == 'Random'))
                ).all()
  ```

- `limit()` - Принимает количество записей, которые нужно вернуть. SQL-эквивалент - оператор LIMIT
- `cast()` - конвертирует данных от одного типа к другому

```
session.query(
    cast(age, Integer),
    cast("2010-12-01", DateTime)
).all()
```

- `order_by()` - Сортирует результат. Принимает названия колонок, по которым необходимо сортировать результат. По умолчанию сортирует по возрастанию, чтобы сортировать по убыванию используйте `order_by(desc())`. SQL-эквивалент - оператор ORDER BY.
- `group_by()` - Группирует результат. Принимает одну или несколько колонок и группирует записи в соответствии со значениями в колонке.
- `join()` - Используется для создания SQL INNER JOIN. Он принимает название таблицы, с которой нужно выполнить SQL JOIN.

```
session.query(Facility)
.join(bookings)
.all()
```

Представим, что у нас есть таблица с авиаперелётов со столбцами:

- `id`
- `company` - Идентификатор компании-перевозчика
- `plane` - Модель самолёта
- `town_from` - Город вылета
- `time_out` - Время вылета

Например, нам нужно вывести 5 рейсов, совершенных из Москвы на самолете Tu-134, отсортировав по убыванию время вылета. SQL запрос будет выглядеть вот так:

```
SELECT *
FROM Trip
WHERE town_from = "Moscow" and plane = "Tu-134"
ORDER BY time_out DESC
LIMIT 5
```

А запрос в SQLAlchemy так:

```
session.query(Trip)
  .filter(and_(Trip.town_from = "Moscow", plane = "Tu-134")
  .order_by(Trip.time_out.desc())
  .limit(5)
  .all()
```

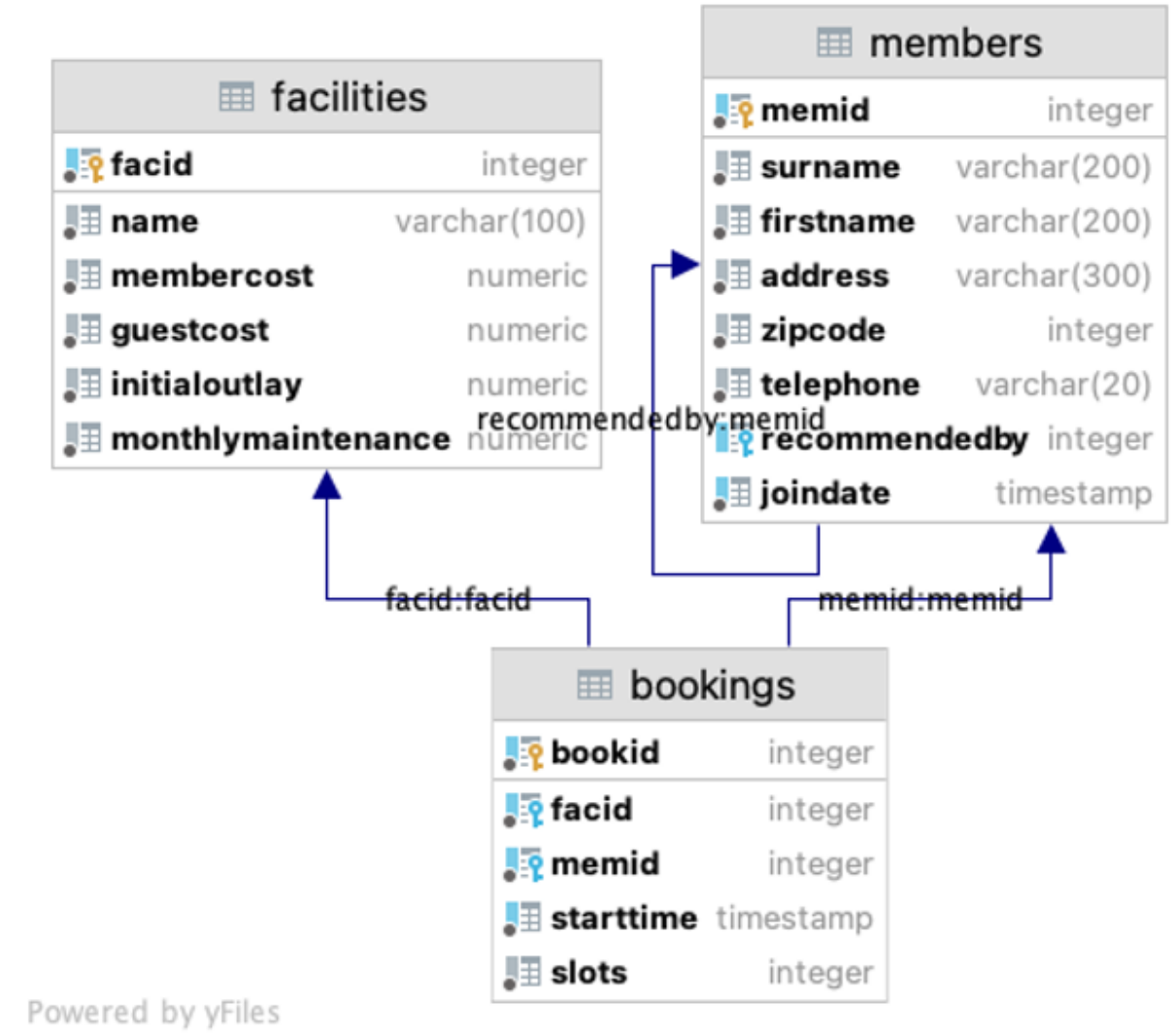
Больше примеров

>FastApi и SQLAlchemy

Давайте теперь поработаем с FastApi. У нас есть папка `db_app`, в которой лежит:

- `database.py` - в этом файле скрипт по инициализации SQLAlchemy и созданию сессий (DATABASE_URL, engine, sessionmaker и пр.).
- `app.py` - тут лежит FastApi. 3 endpoint для получения всех данных из бд
- `models.py` - тут мы создали все таблицы(схема ниже).
- `schemas.py` - схемы Pydantic
- `__init__.py` - Файл обозначающий, что `db_app` это модуль

Таблица:



Посмотрим на `app.py` чуть подробнее:

Импортируем нужные библиотеки:

```
from typing import List

from fastapi import Depends, FastAPI
from sqlalchemy.orm import Session

from .database import SessionLocal
from .models import Booking, Facility, Member
from .schemas import BookingGet, UserGet
app = FastAPI()
```

Реализуем отдельную функцию `get_db()` которая будет создавать подключение к базе данных с помощью `SessionLocal()`. Сначала в блоке будет вызван `SessionLocal()`, он вернёт результат, который сохранится в переменную db, затем, будут выполнены действия описанные в `SessionLocal()`. По сути, эта функция - обёртка, которая безопасно создаёт соединение и закрывает его.

```
def get_db():
    with SessionLocal() as db:
        return db
```

Добавим несколько функций и навесим на них декораторы:

```
@app.get("/user/all", response_model=List[UserGet])
def get_all_users(limit: int = 10, db: Session = Depends(get_db)):
    return db.query(Member).limit(limit).all()


@app.get("/facility/all")
def get_all_facilities(limit: int = 10, db: Session = Depends(get_db)):
    return db.query(Facility).limit(limit).all()


@app.get("/booking/all", response_model=List[BookingGet])
def get_all_bookings(limit: int = 10, db: Session = Depends(get_db)):
    return db.query(Booking).limit(limit).all()
```

В `get_all_users()` есть параметр db, у которого тип `Session` и значение по умолчанию `Depends(get_db)`. Что это? Нам нужно подключиться к базе данных и получить результаты во время получения запроса. В **FastAPI** есть удобный механизм как это всё соединить и состыковать. Называется **Dependency Injection**, подстановка зависимостей.

В каждом эндпоинте мы описываем зависимость, которая ему нужна, а затем мы описываем функцию, которая реализует эту зависимость. И соответствующим образом **FastAPI** подставляет нужные зависимости тем, кто в них нуждается. Сделано это, чтобы можно было легко подменить поставщика одной зависимости на другого поставщика и всё приложение работало бы схожим образом.

Через синтаксис `Depends(get_db())` **FastAPI** понимает, что db это не параметр, а зависимость которую удовлетворяет функция `get_db()`. И соответственно, мы

сможем работать с db как работали раньше со всем и результатами `SessionLocal()` (т.е сможем вызывать `query()`, `.limit()` и т.д)

FastAPI работает независимо от **SQLAlchemy** и его проверки основаны на библиотеке **Pydantic**, которая тоже не имеет к **SQLAlchemy** никакого отношения. Поэтому необходимо объявлять все схемы для валидации через **Pydantic** и не пользоваться готовыми классами от **SQLAlchemy**.

Давайте опишем все схемы, которые должны возвращаться в `schemas.py`:

```
import datetime
from typing import Optional

from pydantic import BaseModel, Field   # импортировали нужные библиотеки

class UserRegister(BaseModel):   # В SQLAlchemy был очень похожий
    name: str
    surname: str

class UserGet(BaseModel):
    first_name: str = ""
    surname: str = ""
    recommended_by: Optional["UserGet"] = None

    class Config:
        orm_mode = True

class BookingGet(BaseModel):
    member_id: int
    member: UserGet
    facility_id: int
    start_time: datetime.datetime
    slots: int

    class Config:
        orm_mode = True
```

В каждой схеме описали что должно возвращаться. `UserGet`, говорит, что должна возвращаться строка `first_name` и surname, и у них может быть пустое значение. `recommended_by` ссылается сам на себя (обратите внимание на ситаксис, `UserGet` в кавычках) и опционально имеет тип UserGet(т.е может иметь тип UserGet, а может иметь None). Т.е мы явно указали, что мы ожидаем, что в `UserGet` будет: `first_name`, `surname` и может быть(наверное) `recommended_by`. точно также иы описали и другие классы. И теперь, если мы всё это запустим, то получим автоматическую валидацию результатов:

Как можете заметить, у нас получилась схема в схеме(как матрёшка). Если бы у Darren Smith был recommended_by, то эта схема бы продолжилась.

>Версионирование схем данных

Представьте, что у вас был табличка в которой вы написали несколько колонок, явно указав их типы, затем спустя некоторое время таблица перестала удовлетворять вашим требованиям(например, вы хотите добавить еще одну колонки и изменить тип уже существующей). Если вы просто поменяете SQL-выражение и поменяете таблицу на сервере, то эти изменения не будут исторически сохранены. Отслеживать историю изменений таблицы можно с помощью инструментов по миграции таблиц. Переезд от старой версии таблицы к новой.

Alembic — это инструмент для миграции базы данных, используемый в **SQLAlchemy**. Миграция базы данных похожа на систему контроля версий для баз данных.