# Тема-6. Backend (продолжение)

**Что тут есть?**
**Об ORM; Об SQLAlchemy; драйвер базы данных; подходы к использованию ORM; как подключить и использовать SQLAlchemy; работа с БД из обработчиков; миграции и Alembic.**

# Об ORM

**Серверу часто необходимо работать с базой данных.**

**Чтобы программистам не приходилось делать это в „сыром“ виде существуют так называемые ORM (Object-Relational Mapping) – технология, которая позволяет работать с базой данных, используя объекты, написанные на языке программирования сервера.**

# SQLAlchemy

**Существует много реализаций ORM. Одна из наиболее полных, удобных и бесплатных реализаций – SQLAlchemy.**

**Установить SQLAlchemy просто: pip install sqlalchemy**

**При этом SQLAlchemy (как и многие другие ORM) позволяет абстрагироваться от синтаксиса конкретной базы данных.**

# Драйвер базы данных

**SQLAlchemy „не знает“, какой синтаксис каждая из СУБД имеет. Чтобы связать SQLAlchemy с СУБД используется так называемый драйвер базы данных.**

**Для каждой СУБД драйвер свой. Например, для PostgreSQL используется psycopg2. Обычно непосредственно с ним взаимодействовать не требуется.**

**Устанавливается он также через pip:**
**pip install psycopg2**

# Подходы к использованию ORM

**Есть разные подходы к работе с БД через ORM:**

**- декларативный – позволяет задекларировать, какие есть таблицы в базе данных и связи между ними, то есть достаточно только описать БД;**

**- классический – позволяет по необходимости создавать соединения с таблицами БД, в момент, когда это нужно.**

# Чем декларативный подход лучше

**Чаще используется декларативный подход, поскольку он позволяет:**

**- вынести описания таблиц и их связей отдельно от остального кода, что повышает его читабельность;**

**- вся информация, связанная с таблицей, хранится вместе;**

**- не требуется самостоятельно писать код для отображения классов на таблицы.**

# Как подключить SQLAlchemy

**Чтобы начать работать с SQLAlchemy необходимо создать так называемый движок – объект, который будет управлять соединением. Делается это так:**

```
from sqlalchemy import create_engine
create_engine("postgresql://easy-learn:qwerty@0.0.0.0:5432/easy-learn")
```

# Как подключить SQLAlchemy

**При этом, если используется декларативный подход, то нужно создать специальный базовый класс, благодаря которому SQLAlchemy сможет автоматически подхватывать все классы, отображающие таблицы:**

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

# SQLAlchemy

## Определение класса может выглядеть примерно так:

```
# импорт класса колонки типов данных
from sqlalchemy import Column, Integer, String
from app.common.db import Base  # импорт базового класса

class Lang(Base):  # создание класса с любым именем, наследованного от Base
    __tablename__ = 'lang'  # указание, для какой таблицы в БД этот класс

    id = Column(Integer, primary_key=True)  # создание числового первичного ключа
    name = Column(String(200), nullable=False)  # строковое поле
    abbr = Column(String(10), nullable=False)  # короткое строковое поле

    def __init__(self, *args, **kwargs):  # необязательная часть, чтобы не ругались IDE
        super(Lang, self).__init__(*args, **kwargs)
```

# SQLAlchemy

**Можно также указать внешний ключ на другую таблицу:**

```
from sqlalchemy import Column, Integer, ForeignKey
from sqlalchemy.orm import relationship, backref

word_id = Column(Integer, ForeignKey("word.id", ondelete="CASCADE"))
word = relationship(
    "Word",
    backref=backref("translations", order_by="Translation.name", cascade="all, delete"),
)
```

# SQLAlchemy

## Или сделать связь „многие ко многим“:

```
from sqlalchemy.schema import Table
...

book_has_tag_table = Table('book_has_tag', Base.metadata,
    Column('book_id', Integer, ForeignKey('book.id')),
    Column('tag_id', Integer, ForeignKey('tag.id'))
)

class Tag(Base):
    # ...
    books = relationship("Book", secondary=book_has_tag_table, backref="tags")
```

# SQLAlchemy

**SQLAlchemy позволяет отобразить фактически всё, что можно написать непосредственно на SQL (конечно, это зависит также от драйвера).**

**Полную документацию по декларированию классов в SQLAlchemy можно посмотреть по ссылке в конце лекции (сайт SQLAlchemy).**

# Работа с БД из обработчиков

**Часто удобно автоматически создавать соединение с БД при обработке запроса.**

**Для этого удобно использовать Depends (который рассматривался в прошлой лекции) и так называемую мидлваре (middleware) – программу „по середине“ – программу, которая выполняется как бы между бекендом и фронтендом.**

**Как это делается, будет показано в демонстрации.**

# Работа с БД из обработчиков

**Пока будем считать, что в обработчике уже можно использовать соединение с БД и объект соединения находится в переменной session.**

```
@router.get("/")
def handler_name(session: Session = Depends(get_db)):
```

# Работа с БД из обработчиков

**Получить данные из БД можно так (несколько вариантов):**

```
from app.modules.dictionary.models import Word
...
session.query(Word).all()  # получить все записи
session.query(Word).first()  # получить первую попавшуюся запись

'''получить единственную запись (если она не единственная или записей нет, будет брошено исключение)'''
session.query(Word).one()
```

# Работа с БД из обработчиков

**Данные можно фильтровать (как WHERE):**

```
from sqlalchemy import or_

session.query(Word).filter(Word.id > 10).all()
session.query(Word).filter(Word.id > 10, Word.name != "test").all()  # and
session.query(Word).filter(or_(Word.id <= 10, Word.name == "test")).all()

session.query(Word).filter_by(id > 10).all()
```

**Можно также использовать and_ явно, например, внутри or_.**

# Работа с БД из обработчиков

**Можно соединять таблицы (как JOIN):**

```
session.query(Word).join(Translation)
    .filter(Word.id == Translation.word_id).first()
```

# Работа с БД из обработчиков

**Можно отложить загрузку тяжелого поля, пока оно не будет запрошено явно:**

```
session.query(Book).options(defer("data")).filter(Book.id == current_id).all()
```

# Работа с БД из обработчиков

**Можно делать сложные запросы с подзапросами:**

```
books = session.query(Book).filter(
    Book.tag == current_tag,
    Book.author_id.in_([author.id for author in session.query(Author)
        .filter(is_author_relevant(Author.id)).all()
    ]),
    Book.edition > oldest_relevant_edition
).all()
```

# Работа с БД из обработчиков

**Можно создавать записи в БД через классы:**

```
new_word = Word(name=word, lang_id=lang.id)
session.add(new_word)
session.commit()  # применение изменений
```

**Удаление записи:**

```
session.delete(new_word)
```

# Работа с БД из обработчиков

**Изменение записи в БД через классы:**

```
word = session.query(Word)
    .filter_by(id==current_id).one()
word.name = correct_name
session.commit()
```

# Сложности изменения схемы БД

**С ORM работать удобно, но что если нужно что-то изменить в схеме базы данных?**

**Раньше приходилось менять схему два раза: сначала в СУБД, потом на бекенде (переписывать измененные классы).**

**Однако, это, во-первых, неудобно – делать дважды одну работу. Во-вторых, нужно помнить, когда что было изменено в БД, особенно если ветка не одна.**

# Alembic

**Чтобы решить это проблемы был создан Alembic – инструмент для автоматического создания миграций схемы БД.**

**Миграция – это новое изменение в схеме БД.**

**Alembic можно установить через pip: pip install alembic**
**После этого в папке с бекендом нужно создать файлы конфигурации Alembic: alembic init alembic**

# Alembic

**После инициализации Alembic нужно настроить его:**

**- добавить в alembic/env.py строку:**

```
config.set_main_option('sqlalchemy.url', os.getenv('DATABASE_URL'))
```

**- указать в alembic/env.py, где находится базовый класс:**

```
from app.common.db import Base
target_metadata = Base.metadata
```

# Alembic

**После настройки Alembic будет сам находить изменения с классах и применять их к схеме БД. Для этого нужно только фиксировать момент, когда пора изменить схему БД. Это делается с помощью команд:**

- **alembic revision --autogenerate -m "Add IconSet table"**
- **alembic upgrade head**

**При выполнении первой команды создается файл миграции. При выполнении второй изменения применяются в БД.**

# Alembic

**Файл миграций содержит команды, которые нужно выполнить, чтобы применить или откатить миграции. Если изменение сложное, может понадобиться поправить его вручную, но обычно это не требуется.**

**Важно помнить, что применение миграции может удалить данные (например, если удален столбец или колонка).**

**Откатить миграцию можно командой:**
**alembic downgrade -1**

# Alembic

**Пример сгенерированной миграции (часть кода опущена):**

```
def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('translation', sa.Column('word_id', sa.Integer(), nullable=False))
    op.create_foreign_key(None, 'translation', 'word', ['word_id'], ['id'],
ondelete='CASCADE')
    # ### end Alembic commands ###


def downgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_constraint(None, 'translation', type_='foreignkey')
    op.drop_column('translation', 'word_id')
    # ### end Alembic commands ###
```

# Полезные ссылки

- **https://sqlalchemy.org/ – сайт SQLAlchemy;**
- **https://alembic.sqlalchemy.org/ – сайт Alembic.**

# Следующая тема: клиент-серверное взаимодействие