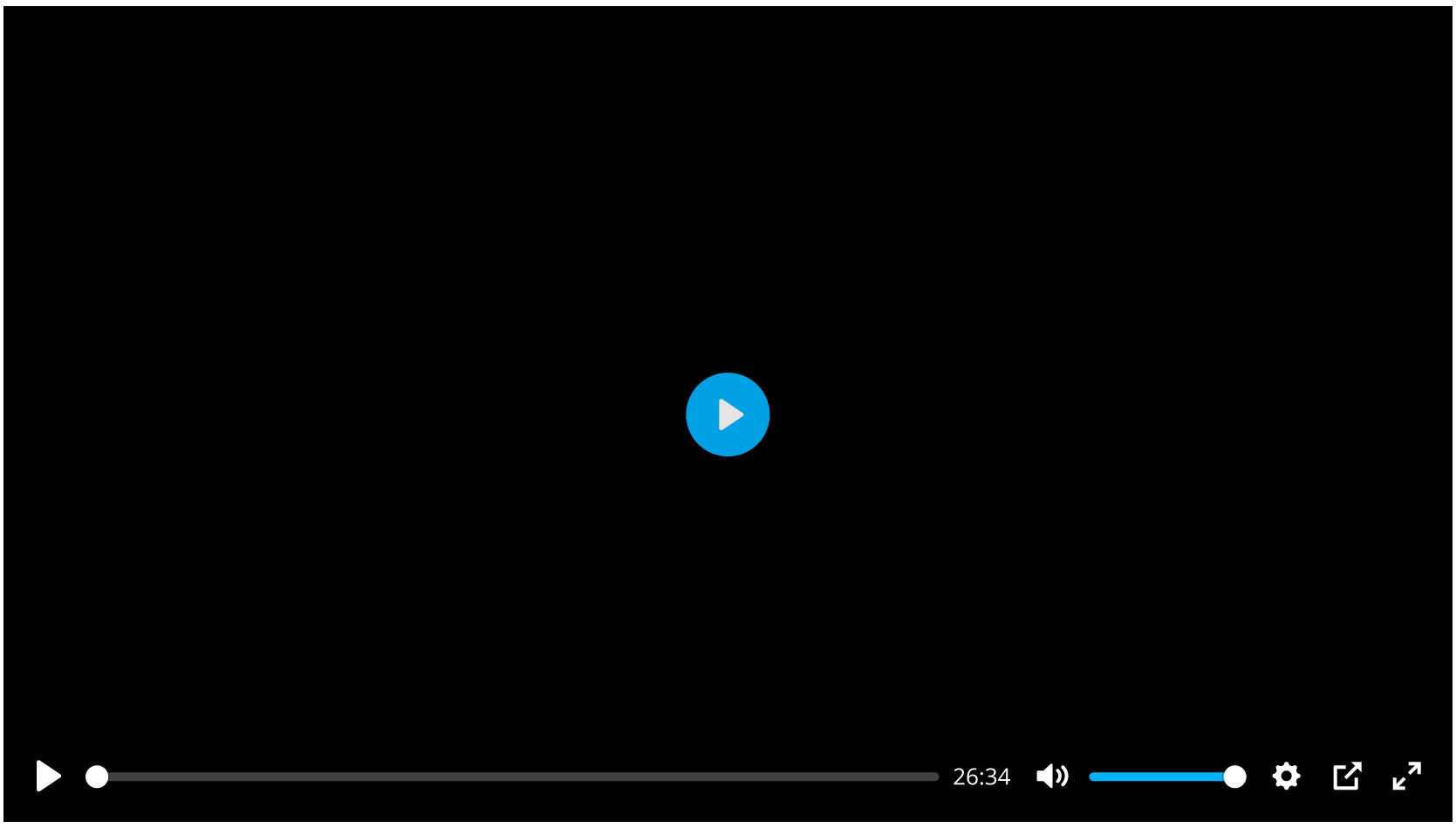


Текстовая расшифровка видео:

ORM: УДОБСТВА И НЕДОСТАТКИ

План:

- Стандарты работы с БД;
- Object-Relational Mapping;
- SQLAlchemy;
- SQLAlchemy: компоненты;
- Pro и Contra;
- Альтернативы.

Стандарты работы с БД

Когда мы начинаем делать проект, мы берем язык программирования и смотрим набор средств для работы с БД. В большинстве современных языков есть встроенные/реализованные в виде стороннего модуля подходы для работы с базами.

Ранее разработчики PHP столкнулись с такой проблемой, как SQL-инъекция.

SQL-инъекция – это уловка пользователя, когда он специально подставляет символы для выполнения произвольного кода на уровне БД.

Собирание SQL-запросов из строк конкатенацией – верный способ выстрелить в ногу (например, получив SQL-инъекцию).

В разных языках существуют нативные для языка стандарты для работы с реляционными базами, например:

- database/sql в Go;
- PEP 249 - DB API 2.0 в Python;

- JDBC в Java;
- ...
- PROFIT

Object-Relational Mapping

Можно напрямую кидать SQL-запросы, но придется писать на двух языках:

- На языке, на котором вы пишете back-end / pipeline;
- На SQL.

Придумали Object-Relational Mapping (ORM), а именно способ маппинга структуры в БД на объекты в языке программирования (классы, инстансы, структуры и т.д.). Вместо того, чтобы писать запросы напрямую в базу, мы взаимодействуем с объектами, вызываем у них методы и синхронизируем состояния объектов в коде с объектами в базе (получается дополнительный уровень абстракций).

Однако придется положиться на автогенерируемый код, а в случае проблем – писать SQL вручную.

SQLAlchemy

Поскольку в основном в курсе мы используем примеры на Python, расскажем про стандартный ORM в Python – **SQLAlchemy**. Он позволяет работать с разными объектами в разных реляционных хранилищах.

Рассмотрим пример:

```
class Employee(Base):
    __tablename__ = "employees"

    employee_id = Column(Integer, primary_key=True)
    name = Column(String(20))
    title = Column(String(100))
    salary = Column(Integer)
    dept = Column(String(20))
e = Employee(
    name="Vasya Poupkine",
    title="Programmer",
    salary=180000,
    dept="IT"
)
...
e.salary += 20000
```

Это пример кода, того, как может выглядеть объект. Это то же самое описание таблички в реляционной базе, просто в виде кода на Python.

Мы можем создавать объекты, указывать у них поля, вызывать в определенном месте добавления в сессию. Соответствующие данные будут записаны в табличке в базе.

SQLAlchemy: компоненты

У нас есть несколько полезных абстракций, которые дает SQLAlchemy:

- **Engine** – движок, используемый для подключения к БД и взаимодействующий с драйвером.

Рассмотрим, как это выглядит:

```
engine = create_engine("postgresql+psycopg2://nikolay:dataengineer@localhost:5432/de_base", echo=True)
Base = declarative_base()
```

Мы прописываем Connection string. Мы указываем базу, драйвер, реквизиты, хост, порт и базу внутри базы. Также указываем параметр «echo=true» для того, чтобы логировались все запросы, летящие в базу.

- **Base (+ Metadata)** – класс-основа для создания моделей.

Рассмотрим, как это выглядит:

```
13 class Employee(Base):
14     __tablename__ = "employees"
15
16     employee_id = Column(Integer, primary_key=True)
17     name = Column(String(20))
18     title = Column(String(100))
19     salary = Column(Integer)
20     # start_date = Column(DateTime, server_default=func.now())
21     dept = Column(String(20))
22
```

Если мы хотим все эти объекты создать в базе, используем дополнительный промежуточный объект, который называется «**Metadata**». Это дочерний объект класса «Base». Когда импортируются классы с описаниями таблиц, в Metadata автоматически появляется информация в реестре о том, что мы создали.

Если мы вызовем create all, у нас автоматически в базе создастся структура из таблиц, которую мы описали в классах.

- **Session** – объект для управления подключениями к БД.

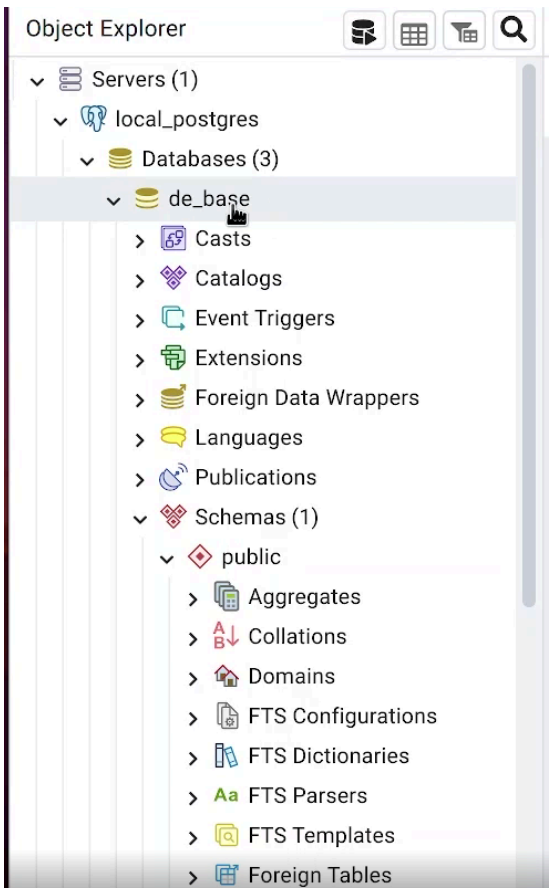
Рассмотрим, как это выглядит:

```
34     with Session(engine) as session:
35         session.add(e)
36         session.commit()
37         print(e.salary)
38         e.salary += 20000      Cannot assign member "salary" for type "Employee"    Expression of type "ColumnOp
39         session.commit()
```

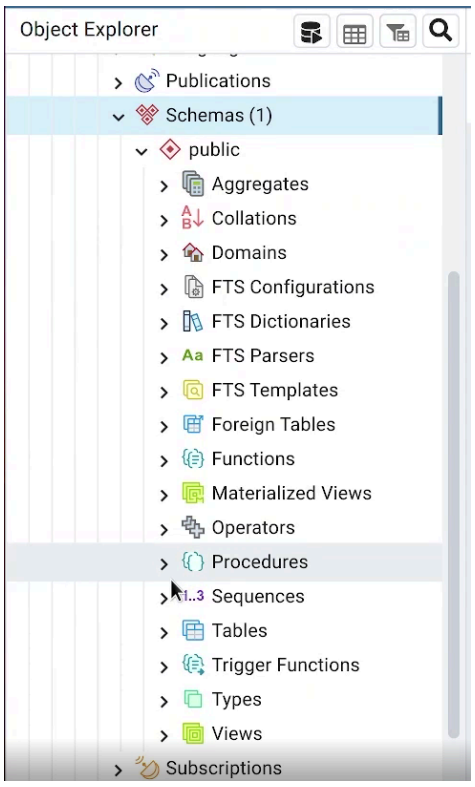
Внутри есть Connection pool. Также при помощи «With» создадим контекст в коде.

Запустим и проверим. Заходим в pgAdmin.

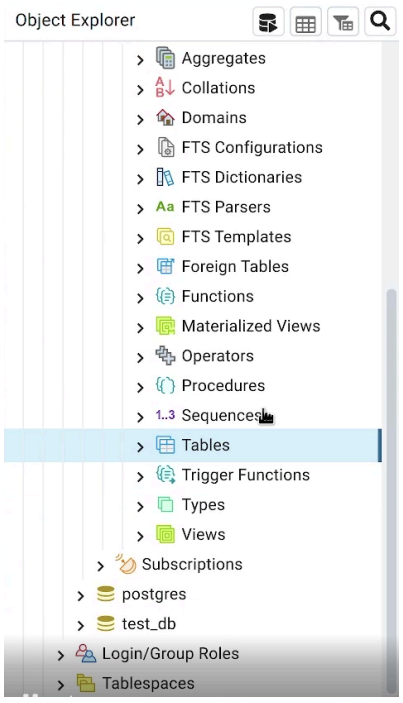
Видим базу:



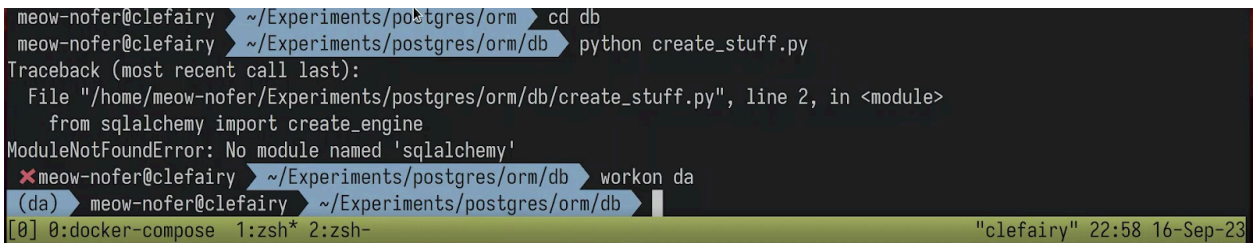
Видим схемы:



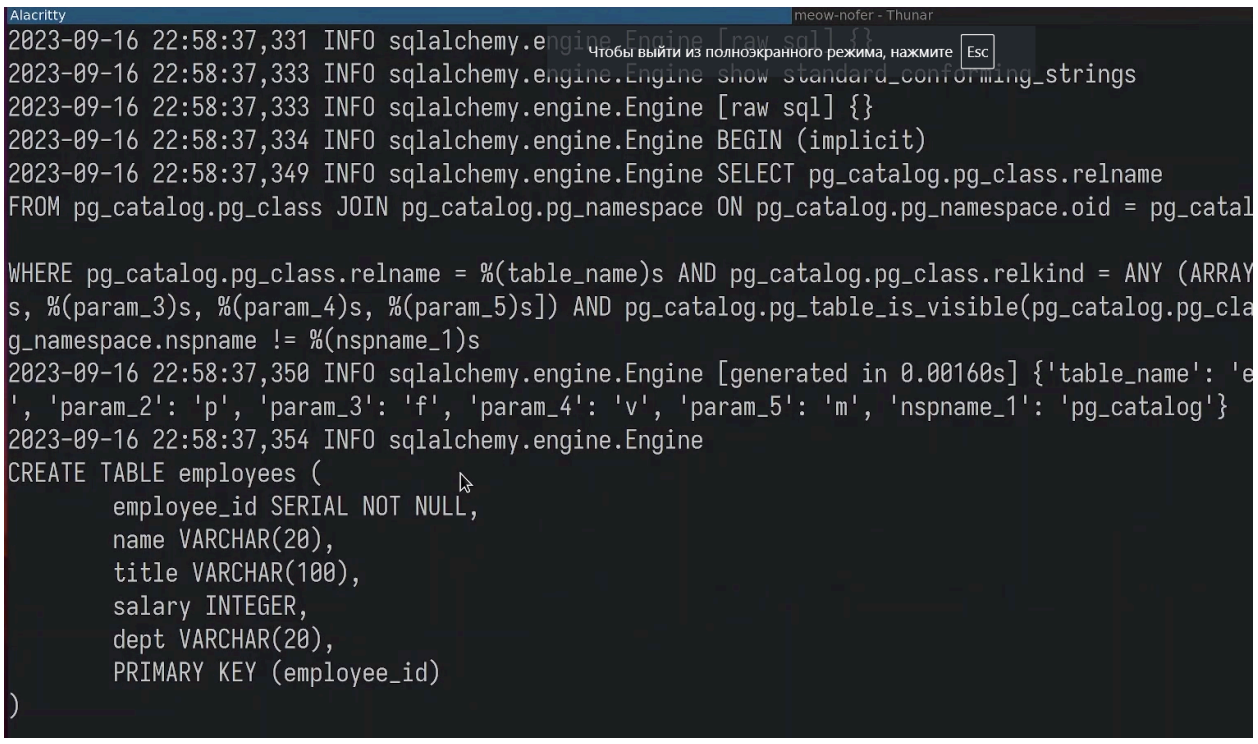
Таблиц пока нет:



Запустим код:



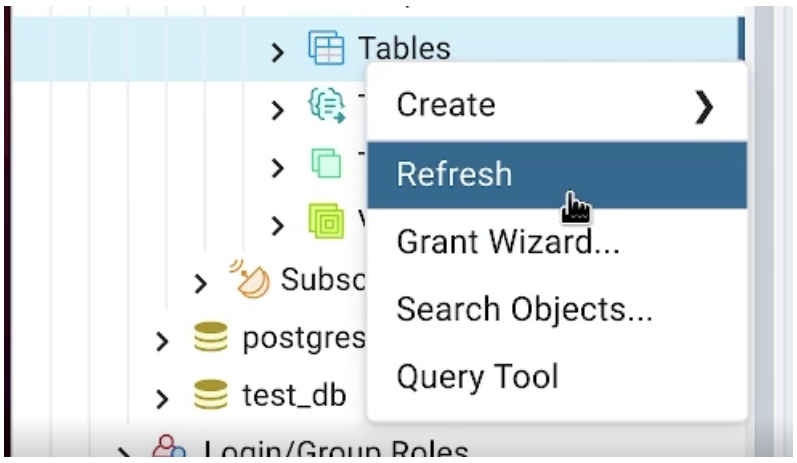
У нас выполнится целая пачка SQL-запросов. Поскольку мы указали «echo=true» все запросы увидим в виде log’ов в стандартном выводе. Также должен выполниться «Create table»:



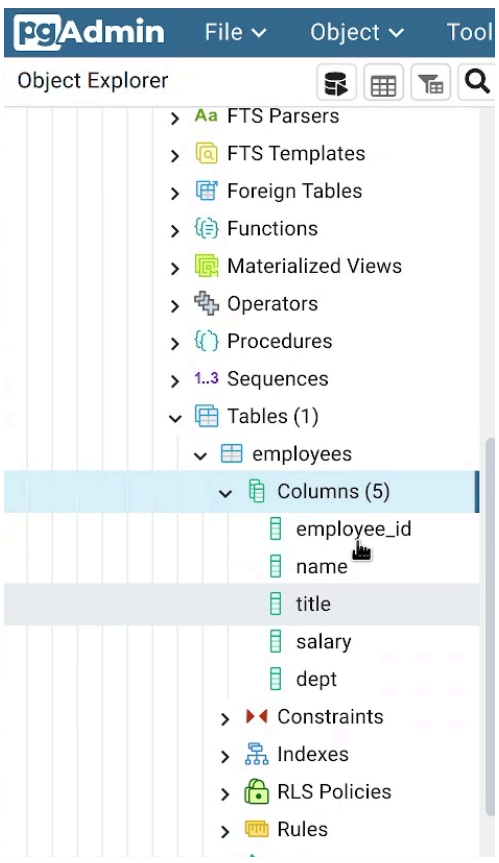
Мы создали экземпляр сотрудника, у нас вызывался insert into с соответствующими полями, значениями:

```
2023-09-16 22:58:37,354 INFO sqlalchemy.engine.Engine [no key 0.00025s] {}
2023-09-16 22:58:37,367 INFO sqlalchemy.engine.Engine COMMIT
2023-09-16 22:58:37,373 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2023-09-16 22:58:37,378 INFO sqlalchemy.engine.Engine INSERT INTO employees (name, title, salary, dept) VALUES (%(name)
s, %(title)s, %(salary)s, %(dept)s) RETURNING employees.employee_id
```

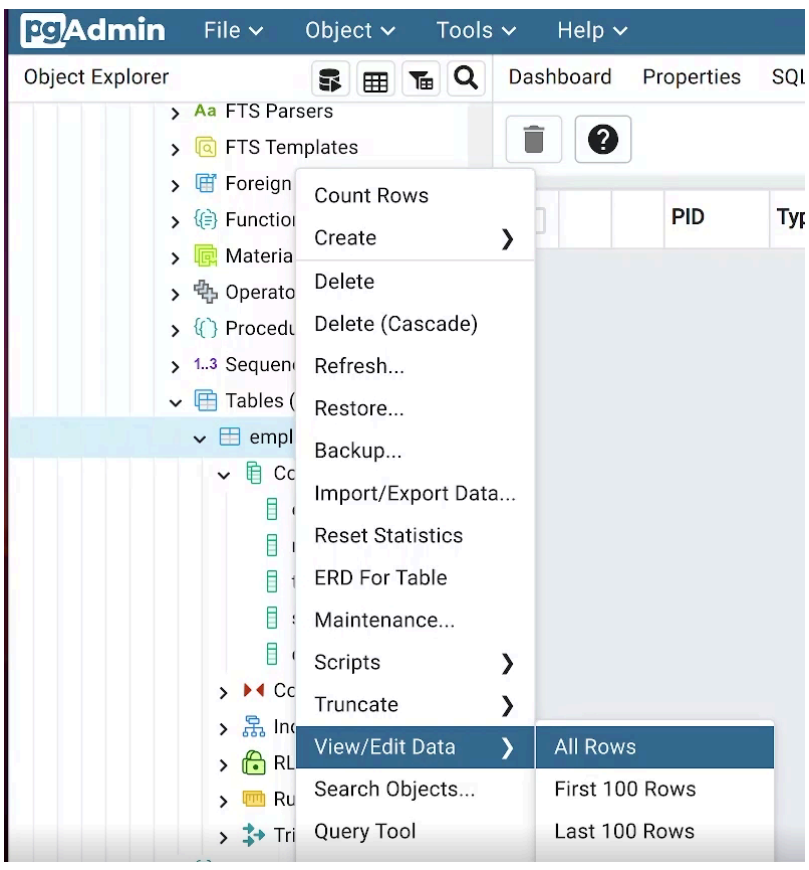
Посмотрим, что создалось в базе. Вызовем «Refresh»:



Видим, что создалась табличка «Employees» с колонками:



Посмотрим, что в этой табличке лежит:



Видим следующее:

Все работает.

Когда мы работаем со структурой в базе, особенно, если это back-end, на каком-то этапе нужно делать миграции. В таком случае можно использовать проект «Alembic».

Alembic – это менеджер миграций для моделей SQLAlchemy, который позволяет отслеживать изменения в коде и автоматически генерировать код, приводящий базу в новое состояние.

Посмотрим, как это выглядит.

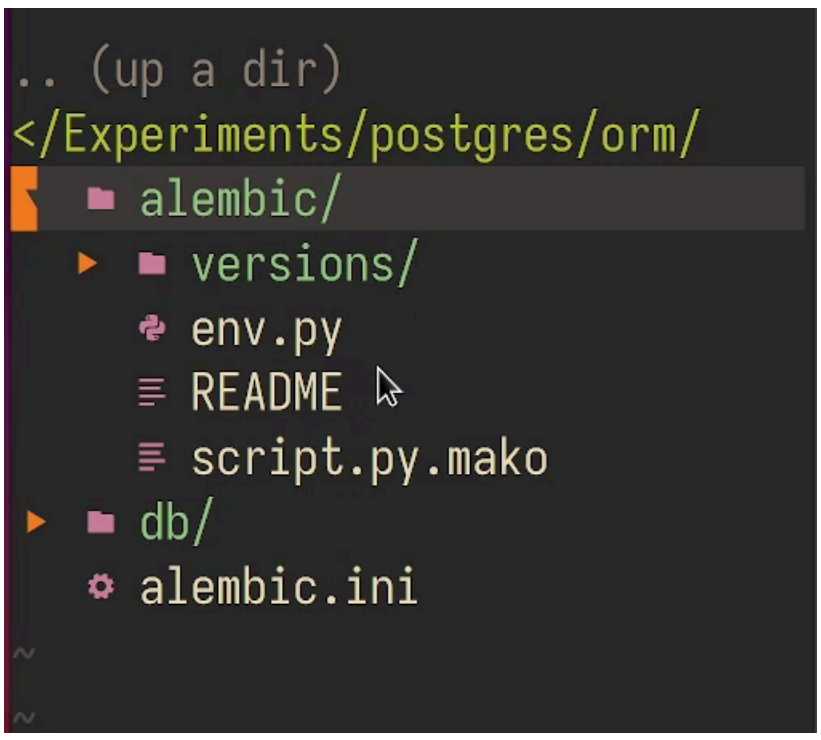
Для начала вызовем установку:

Чтобы создать базовые файлы, вызовем следующую команду:

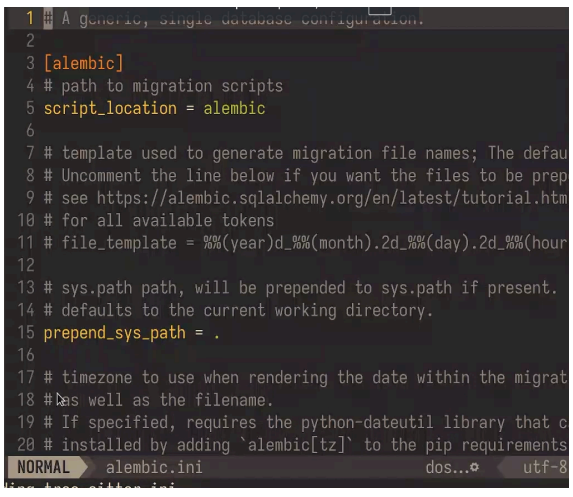
```
~$ pip install alembic
```

```
~$ alembic init alembic
```

Мы вызываем Alembic и предлагаем ему хранить информацию о миграциях в каталоге с названием «Alembic». Создастся каталог «Alembic»:



В этом каталоге есть папка «Versions», в которой находятся файлы с миграциями. Создастся файл «Alembic.init» с разными базовыми настройками:



Также создастся «env.py», в котором будут все настройки Python:

```python
from logging.config import fileConfig

from sqlalchemy import engine_from_config
from sqlalchemy import pool

from alembic import context

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
if config.config_file_name is not None:
    fileConfig(config.config_file_name)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
```

Чтобы подключить миграции к моделям нужно:

- Добавить **sqlalchemy.url** в *alembic.ini*

Это тот самый Connection string. Достанем его:

```
("postgresql+psycopg2://nikolay:dataengineer@localhost:5432/de_base", echo=True)
e()
engine = create_engine("postgresql+psycopg2://nikolay:dataengineer@localhost:5432
Base = declarative_base()
```

Пропишем его:

```
sqlalchemy.url = postgresql+psycopg2://nikolay:dataengineer@localhost:5432/de_base
```

Таким образом Alembic найдет, куда подключаться.

- Указать **target_metadata** в *alembic/env.py* (поиск файлов с моделями)

Пропишем следующее:

```
target_metadata = db.create_sdb.create_stuff.Base.metadata
```

```
from db.create_stuff import Base
```

```
target_metadata = Base.metadata
```

Если все сделано правильно, Alembic увидит модели, которые мы прописали и автоматически появится возможность сгенерировать необходимые миграции. Нужно, чтобы Alembic увидел локальную папку с db.

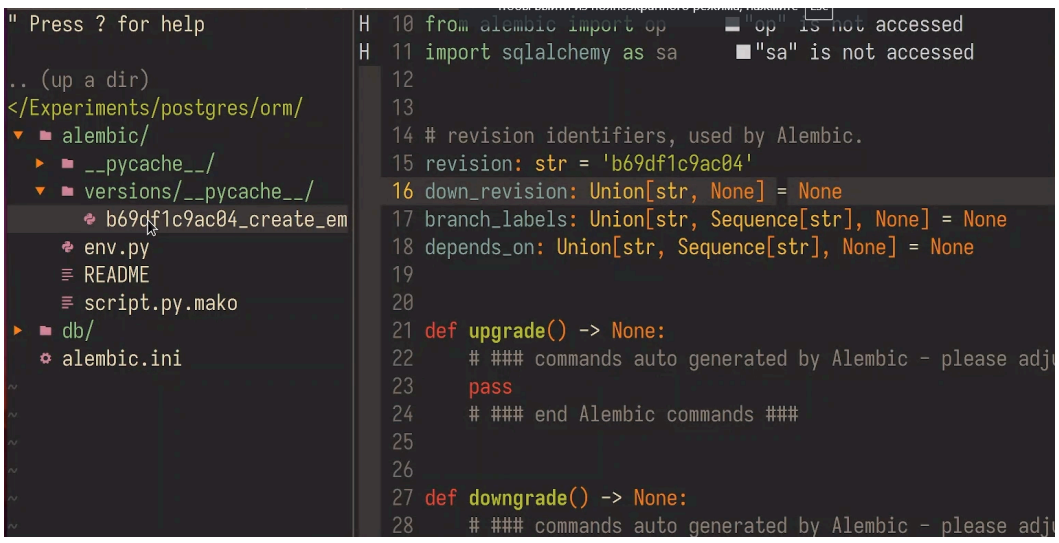
- В случае первого запуска – проверить, что табличка не существует;
- `~$ PYTHONPATH=. alembic revision -m "create employees table" –autogenerate;`

Посмотрим, что получилось:

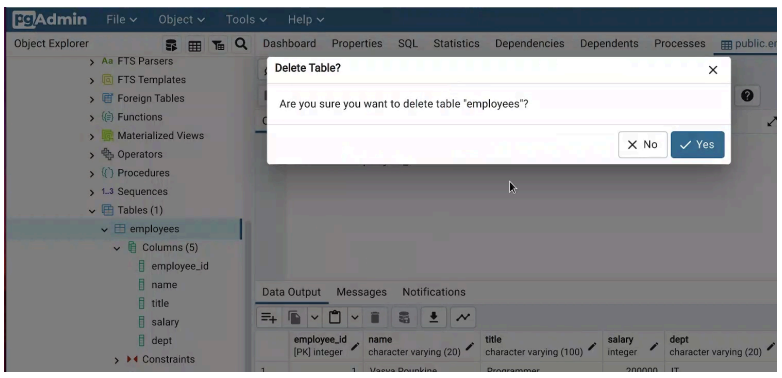
```
" Press ? for help
.. (up a dir)
</Experiments/postgres/orm/
alembic/
  __pycache__/
  versions/
    __pycache__/
    b69df1c9ac04_create_em
  env.py
  README
  script.py.mako
db/
alembic.ini
```

```python
down_revision: Union[str, None] = None
branch_labels: Union[str, Sequence[str], None] = None
depends_on: Union[str, Sequence[str], None] = None


def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjus
    pass
    # ### end Alembic commands ###


def downgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    pass
    # ### end Alembic commands ###
```

У нас уже создалась вся структура. Сделаем вот так:



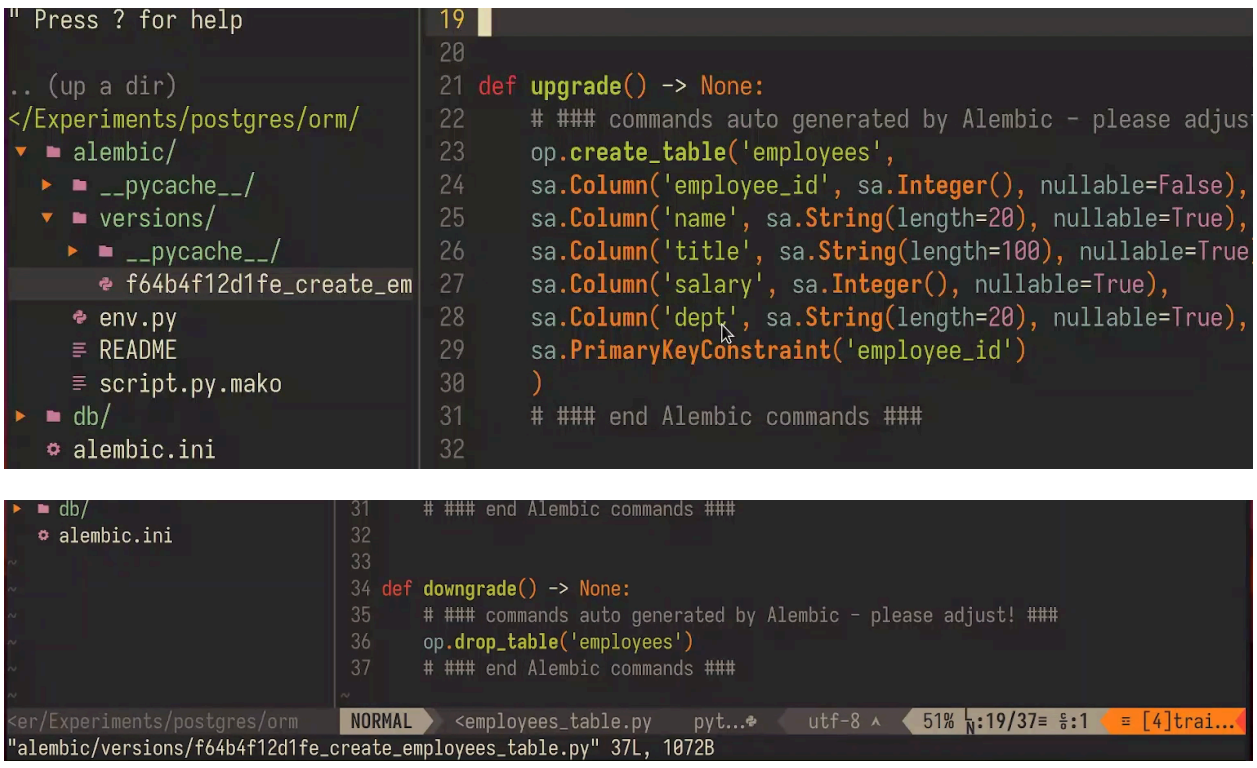
Снесем в базе созданную табличку:



Мы создавали ее вручную, а правило хорошего тона обязывает создавать ее через миграцию.

Вызовем еще раз Generate:

Посмотрев в свежесозданный файл с миграцией, мы увидим, что класс, описывающий табличку, был обнаружен:



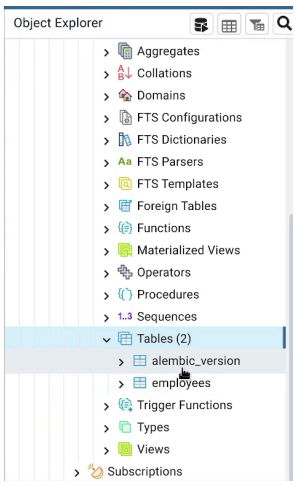
Alembic автоматически сгенерировал, как шаг для миграции вперед (апгрейд), так и назад (даунгрейд).

Для того, чтобы еще раз создать табличку, нужно вызвать команду:

- `~$ alembic upgrade head`

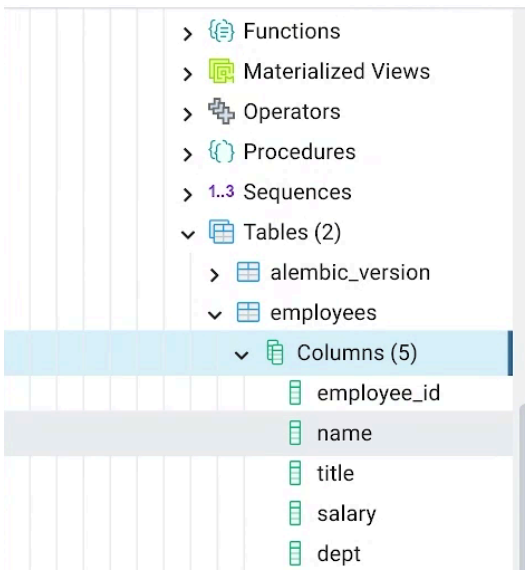
Head – это вызов всех известных миграций на текущий момент.

В базе должна создаться табличка:



Также появилась мета-табличка Alembic, которая отслеживает версии в базе.

В итоге создалась та же самая табличка, и мы можем работать дальше:



Подправим что-то в коде, например, добавим здесь новое поле в класс:

```
16      employee_id = Column(Integer, primary_key=True)
17      name = Column(String(20))
18      title = Column(String(100))
19      salary = Column(Integer)
20      start_date = Column(DateTime, server_default=func.now())
21      dept = Column(String(20))
22
```

Например, мы хотим хранить информацию о том, когда сотрудник впервые вышел на работу.

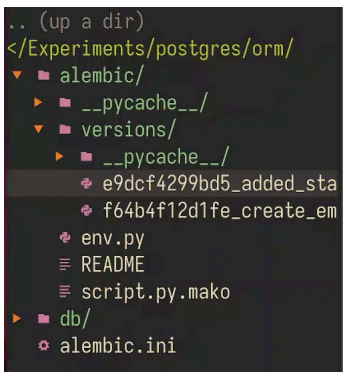
Еще раз вызовем Alembic и зададим название, сказав «создай еще одну миграцию»:

```
(da) meow-nofer@clefairy ~/Experiments/postgres/orm vim
(da) meow-nofer@clefairy ~/Experiments/postgres/orm PYTHONPATH=. alembic revision -m "added start time" --autoge
nerate
```

У нас создалась новая колонка в таблице:

```
INFO  [alembic.autogenerate.compare] Detected added column 'employees.start_date'
```

Посмотрим, как это выглядит в контексте миграции:



Видим первую и вторую миграции.

Вторая миграция:

```python
# revision identifiers, used by Alembic.
revision: str = 'e9dcf4299bd5'
down_revision: Union[str, None] = 'f64b4f12d1fe'
branch_labels: Union[str, Sequence[str], None] = None
depends_on: Union[str, Sequence[str], None] = None


def upgrade() -> None:
```

```python
def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('employees', sa.Column('start_date', sa.DateTime(), server_defaul
    # ### end Alembic commands ###


def downgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_column('employees', 'start_date')
    # ### end Alembic commands ###
```

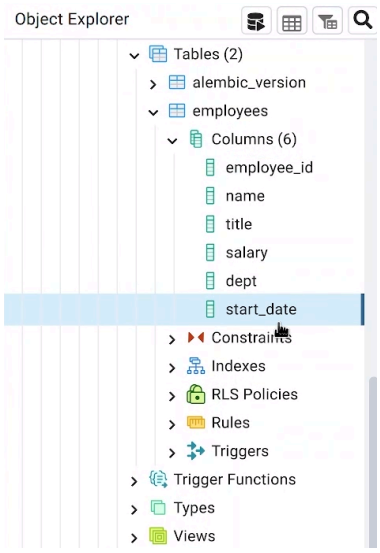
Здесь будет вызываться Alter table, который создаст в табличке новую колонку.

Вызываем следующее:

```
(da) meow-nofer@clefairy ~/Experiments/postgres/orm vim
(da) meow-nofer@clefairy ~/Experiments/postgres/orm alembic upgrade head
```

Состояние базы прокрутится до последней миграции.

Посмотрим, что поменялось, сделав refresh:



Добавилось поле «Start_date».

Выполним запрос:

```sql
SELECT * FROM public.employees
ORDER BY employee_id ASC
```

Data Output Messages Notifications

employee_id [PK] integer	name character varying (20)	title character varying (100)	salary integer	dept character varying (20)	start_date timestamp without time zone

Данных никаких нет, но все поля есть. Через ORM можно продолжать работать с этими объектами.

Pro и Contra

У данного подхода есть положительные и отрицательные стороны.

Плюсы:

- Можно не писать SQL-код руками, это дает быстрее TTM;
- Нормально написанный код на Python (Java/Go/etc.) может быть гораздо читабельнее, чем SQL;
- Императивные трюки делать намного легче (хоть это и опасно!).

Минусы:

- Без знания SQL и навыков отладки SQL-запросов полностью полагаться на ORM может быть не очень хорошей идеей;
- Генерируемый код не всегда будет соответствовать вашим представлениям об идеале и правильности;
- Плохо умеет запускаться в составе асинхронного кода.

Альтернативы

SQL Alchemy. Он поделен на:

- SQL Alchemy Core (это маленькая тонкая обертка в Python, позволяющая писать код, который напрямую превращается в SQL);
- SQL Alchemy ORM (позволяет писать классы, абстрагировать таблички и т.д).

Dbt.

Это то, что позволит генерировать SQL из YAML'ов.

Power BI.

Больше для работы с аналитической стороны. Некоторые BI-инструменты достаточно визуальны и без SQL.

Как вам урок? 😠 😐 🔥

Изучил, далее ›