

Министерство науки и высшего образования Российской Федерации

ФГАОУ ВО «Уральский федеральный университет имени первого  
Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий-РТФ

## ОТЧЕТ

По лабораторной работе №2

«Работа с SQLAlchemy и alembic»

По дисциплине «Разработка приложений»

Выполнил: Гуламов А.С.

Группа: РИМ-150950

Проверил преподаватель:  
Кузьмин Д.

Екатеринбург

2025

**Цель работы:** Освоить принципы работы с библиотеками SQLAlchemy и Alembic для создания и управления реляционными базами данных на Python, изучить механизмы миграции базы данных.

**Задачи:**

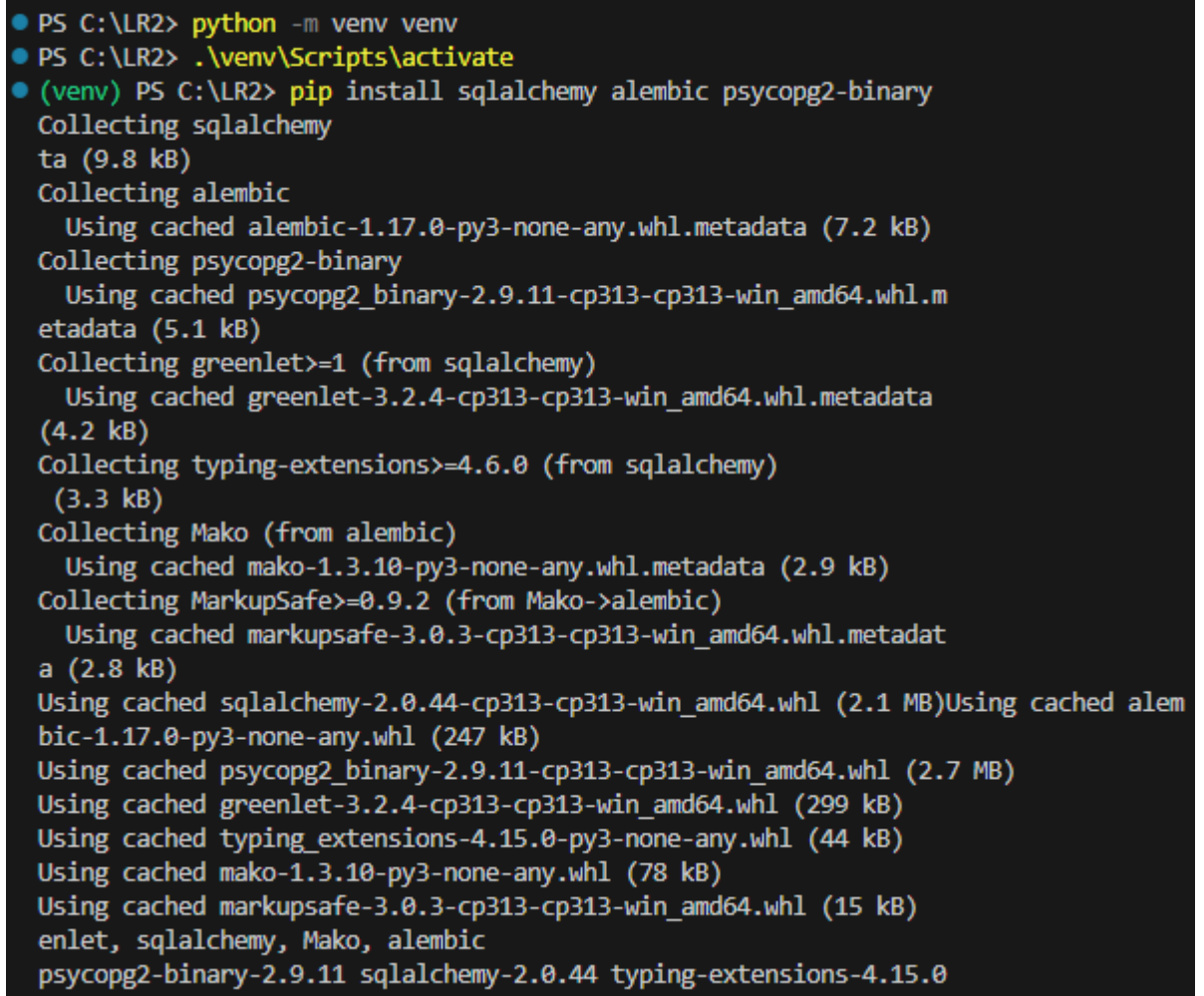
1. Настроить рабочее окружение.
2. Разработать ORM-модели данных с использованием SQLAlchemy.
3. Настроить и использовать систему миграций Alembic.
4. Реализовать работу с данными через ORM
5. Проверить корректность работы системы

## Ход работы:

### 1. Подготовка среды разработки

В VS Code создаем папку проекта, создаем и активируем виртуальное окружение, загружаем требуемые библиотеки с помощью команд (Рисунок 1):

- `mkdir LR2;`
- `cd LR2;`
- `python -m venv venv;`
- `.\venv\Scripts\activate.`
- `pip install sqlalchemy alembic psycopg2-binary`



```
PS C:\LR2> python -m venv venv
PS C:\LR2> .\venv\Scripts\activate
(venv) PS C:\LR2> pip install sqlalchemy alembic psycopg2-binary
Collecting sqlalchemy
  Using cached sqlalchemy-2.0.44-cp313-cp313-win_amd64.whl (2.1 MB)
Collecting alembic
  Using cached alembic-1.17.0-py3-none-any.whl (247 kB)
Collecting psycopg2-binary
  Using cached psycopg2-binary-2.9.11-cp313-cp313-win_amd64.whl (2.7 MB)
Collecting greenlet>=1 (from sqlalchemy)
  Using cached greenlet-3.2.4-cp313-cp313-win_amd64.whl (299 kB)
Collecting typing-extensions>=4.6.0 (from sqlalchemy)
  Using cached typing_extensions-4.15.0-py3-none-any.whl (44 kB)
Collecting Mako (from alembic)
  Using cached mako-1.3.10-py3-none-any.whl (78 kB)
Collecting MarkupSafe>=0.9.2 (from Mako->alembic)
  Using cached markupsafe-3.0.3-cp313-cp313-win_amd64.whl (15 kB)
Installing collected packages: typing-extensions, sqlalchemy, Mako, alembic, psycopg2-binary
Successfully installed alembic-1.17.0 Mako-1.3.10 psycopg2-binary-2.9.11 sqlalchemy-2.0.44 typing-extensions-4.15.0
```

Рисунок 1 – Подготовка

Далее запускаем PostgreSQL в Docker и убеждаемся в работе (Рисунок 2).

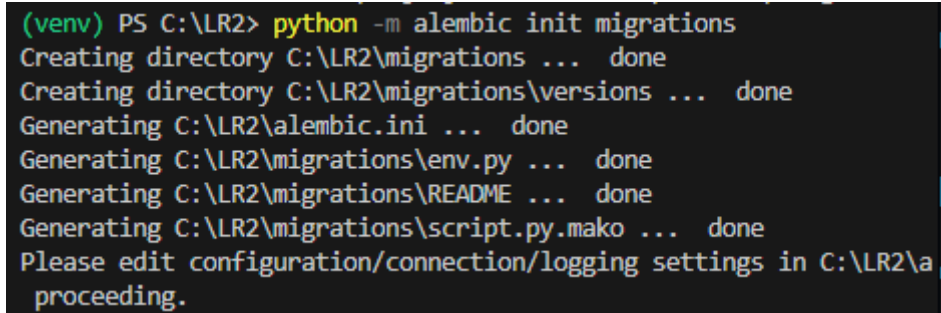
```
(venv) PS C:\LR2> docker run --name lr2-postgres -e POSTGRES_DB=lr2_db -e POSTGRES
_USER=postgres -e POSTGRES_PASSWORD=secret -p 5432:5432 -d postgres:15
c90b0d8b5da345b296c60803be2d5844836510fbe3552af895ddf0c211398e0e
(venv) PS C:\LR2> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
c90b0d8b5da3	postgres:15	"docker-entrypoint.s..."	5 seconds ago	Up 5 seconds
0.0.0.0:5432->5432/tcp, [::]:5432->5432/tcp		lr2-postgres		

Рисунок 2 – PostgreSQL в Docker

## 2. Создание и настройка файлов

Создали файл моделей `models.py` (файл прикреплен в репозитории). Далее инициализировали Alembic с помощью команды «`python -m alembic init migrations`», появились папка «`migrations`» и файл «`alembic.ini`» (Рисунок 3)



```
(venv) PS C:\LR2> python -m alembic init migrations
Creating directory C:\LR2\migrations ... done
Creating directory C:\LR2\migrations\versions ... done
Generating C:\LR2\alembic.ini ... done
Generating C:\LR2\migrations\env.py ... done
Generating C:\LR2\migrations\README ... done
Generating C:\LR2\migrations\script.py.mako ... done
Please edit configuration/connection/logging settings in C:\LR2\alembic.ini
proceeding.
```

Рисунок 3 – Запуск команды «`python -m alembic init migrations`»

В файле «`alembic.ini`» нашли строку:

- «`sqlalchemy.url = driver://user:pass@localhost/dbname`»

и заменили ее на

- «`sqlalchemy.url =postgresql://postgres:secret@localhost:5432/lr2_db`»

В файле «`migrations/env.py`» нашли строку:

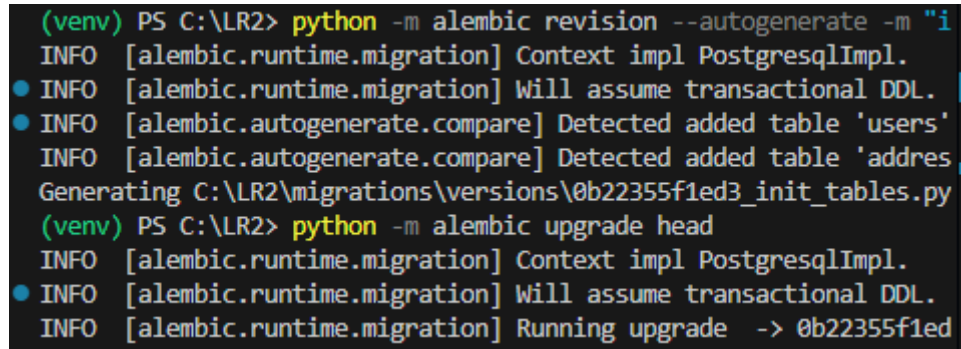
- «`target_metadata = None`»

и заменили ее на:

- «`from models import Base`»
- «`target_metadata = Base.metadata`»

Далее создали и применили миграцию с помощью команд (Рисунок 4):

- «python -m alembic revision --autogenerate -m "init tables"»
- «python -m alembic upgrade head»



```
(venv) PS C:\LR2> python -m alembic revision --autogenerate -m "i
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.
• INFO [alembic.runtime.migration] Will assume transactional DDL.
• INFO [alembic.autogenerate.compare] Detected added table 'users'
INFO [alembic.autogenerate.compare] Detected added table 'addres
Generating C:\LR2\migrations\versions\0b22355f1ed3_init_tables.py
(venv) PS C:\LR2> python -m alembic upgrade head
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.
• INFO [alembic.runtime.migration] Will assume transactional DDL.
INFO [alembic.runtime.migration] Running upgrade -> 0b22355f1ed
```

Рисунок 4 – Работа с миграциями

Проверили корректность работы (Рисунок 5).

```

(venv) PS C:\LR2> docker exec -it lr2-postgres psql -U postgres -
psql (15.14 (Debian 15.14-1.pgdg13+1))
Type "help" for help.

lr2_db=# \dt
               List of relations
-----+-----+-----+-----
 public | addresses      | table | postgres
 public | alembic_version | table | postgres
 public | users           | table | postgres
(3 rows)

lr2_db=# \d users
                    Table "public.users"
   Column   |          Type          | Collation | Nullable
-----+-----+-----+-----
 id          | uuid                   |           | not null
 username    | character varying      |           | not null
 email       | character varying      |           | not null
 created_at  | timestamp without time zone |         |
 updated_at  | timestamp without time zone |         |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_email_key" UNIQUE CONSTRAINT, btree (email)
    "users_username_key" UNIQUE CONSTRAINT, btree (username)
Referenced by:
    TABLE "addresses" CONSTRAINT "addresses_user_id_fkey" FOREIGN
REFERENCES users(id)

lr2_db=# SELECT * FROM alembic_version;
 version_num
-----
 0b22355f1ed3
(1 row)

lr2_db=# \q

```

Рисунок 5 – Подключение к БД внутри контейнера

### 3. Наполнение БД данными

Создадим скрипт для наполнения БД «seed\_data.py» (файл прикреплен в репозитории) и запустим его командой «python seed\_data.py», в результате чего БД наполнилась 5 пользователями и их адресами. Проверим корректность работы (Рисунок 6).

```
lr2_db=# SELECT street, city, country FROM addresses;
 street      | city      | country
-----+-----+-----
 123 Main St | New York  | USA
 456 Oak Ave | Los Angeles | USA
 789 Pine Rd | Chicago   | USA
 101 Maple Ln | Madrid    | Spain
 202 Bamboo St | Beijing   | China
(5 rows)
```

```
lr2_db=# SELECT username, email FROM users;
 username | email
-----+-----
 li_wei   | li.wei@example.com
 maria_garcia | maria.garcia@example.com
 john_doe | john.doe@example.com
 jane_smith | jane.smith@example.com
 alex_wilson | alex.wilson@example.com
(5 rows)
```

Рисунок 6 – Подключение к БД внутри контейнера для просмотра наполнения БД



#### 4. Запрос связанных данных

Создали файл «query\_data.py» (файл прикреплен в репозитории) и запустим его командой «python query\_data.py» (Рисунок 7).

```
(venv) PS C:\LR2> python query_data.py
● Пользователь: li_wei (li.wei@example.com)
  Адрес: 202 Bamboo St, Beijing, China
-----
Пользователь: maria_garcia (maria.garcia@example.com)
  Адрес: 101 Maple Ln, Madrid, Spain
-----
Пользователь: john_doe (john.doe@example.com)
  Адрес: 123 Main St, New York, USA
-----
Пользователь: jane_smith (jane.smith@example.com)
  Адрес: 456 Oak Ave, Los Angeles, USA
-----
Пользователь: alex_wilson (alex.wilson@example.com)
  Адрес: 789 Pine Rd, Chicago, USA
-----
```

Рисунок 7 – Вывод данных о каждом пользователе

## 5. Модификация БД

В файле «models.py» добавили поле «description» в класс User (Рисунок 8). А также добавили новые таблицы «Product» и «Order» и чуть не забыли про связи! (Рисунок 9).

```
class User(Base):
    __tablename__ = 'users'

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    username = Column(String, nullable=False, unique=True)
    email = Column(String, nullable=False, unique=True)
    description = Column(String, nullable=True) #-----
    created_at = Column(DateTime, default=datetime.now)
    updated_at = Column(DateTime, onupdate=datetime.now)

    addresses = relationship("Address", back_populates="user")
```

Рисунок 8 – Добавление поля «description»

```
#-----
class Product(Base):
    __tablename__ = 'products'

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    name = Column(String, nullable=False)
    price = Column(Numeric(10, 2), nullable=False) # например, 199.99
    description = Column(String)
    created_at = Column(DateTime, default=datetime.now)

class Order(Base):
    __tablename__ = 'orders'

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey('users.id'), nullable=False)
    address_id = Column(UUID(as_uuid=True), ForeignKey('addresses.id'), nullable=False)
    product_id = Column(UUID(as_uuid=True), ForeignKey('products.id'), nullable=False)
    quantity = Column(Integer, default=1)
    status = Column(String, default="pending") # например: pending, shipped, delivered
    created_at = Column(DateTime, default=datetime.now)

    user = relationship("User")
    address = relationship("Address")
    product = relationship("Product")
```

Рисунок 9 – Добавление таблиц «Product» и «Order»

Далее, чтобы сгенерировать миграцию, прописали команду «python -m alembic revision --autogenerate -m "add description to user, add products and orders"» и применили миграцию командой «python -m alembic upgrade head»

Наполним БД новыми данными, создав файл «seed\_more\_data.py» и запустив его командой «python seed\_more\_data.py». Проверим изменения (Рисунок 10).

```
lr2_db=# SELECT id, status FROM orders;
```

id	status
4b2861b7-1998-49fc-a4b2-95f0b2cece05	pending
23a39963-9e8a-49cf-a30f-1850cc75b33e	pending
8b075e99-b8a3-498b-9999-a47ac40c7bc5	pending
d4604ab8-f426-4c4b-83aa-b16c7f3f631d	pending
17c24107-6873-4d77-af45-473e9ae4eb52	pending

(5 rows)

```
lr2_db=# SELECT name, description FROM products;
```

name	description
Ноутбук	Мощный игровой ноутбук
Книга 'Python для начинающих'	Популярное учебное пособие
Беспроводные наушники	Шумоподавление
Футболка	Хлопок, размер L
Кофемашина	С капучинатором

(5 rows)

Рисунок 10 – Новые данные

## Вопросы:

1. Какие есть подходы маппинга в SQLAlchemy? Когда следует использовать каждый подход?

- Декларативный - самый простой: требует написания класса, после этого всё сразу понятно.
- Классический - сначала нужно описать таблицу, потом класс, потом связать их.
- Автомаппинг - когда имеется база, а SQLAlchemy сам придумывает классы

В данной работе мы использовали первый, потому что он самый лёгкий.

2. Как Alembic отслеживает текущую версию базы данных?

Alembic создаёт в базе специальную служебную таблицу под названием «alembic\_version». В ней хранится одна строчка - аналог «ярлыка», который хранит ID последней применённой миграции. При запуске «alembic upgrade head», Alembic смотрит, какая версия сейчас в таблице, какие миграции есть в папке versions и применяет только те, которых ещё нет.

3. Какие типы связей между таблицами вы реализовали в данной работе?

Мы сделали связь «один-ко-многим»: один пользователь может иметь несколько адресов, но каждый адрес принадлежит только одному пользователю. В таблице «addresses» есть поле «user\_id», которое ссылается на «id» в таблице «users». В коде мы использовали ForeignKey и relationship, чтобы SQLAlchemy знал, как эти таблицы связаны. Связи «многие ко многим» (например, один пользователь - много продуктов, и один продукт - у многих

пользователей) у нас не было, но можно было бы добавить, если бы за один заказ можно было бы купить несколько товаров.

#### 4. Что такое миграция базы данных и почему она важна?

Миграция - файл с инструкциями, как изменить структуру базы данных: добавить таблицу, колонку, изменить тип и т.д. Она важна, потому что без неё пришлось бы вручную редактировать базу на каждом компьютере, а с миграциями же все изменения сохраняются в коде. В случае ошибки можно откатиться назад.

#### 5. Как обрабатываются отношения многие-ко-многим в SQLAlchemy

Связь «многие ко многим» (например, «пользователь может покупать много товаров, и товар может быть у многих пользователей») нельзя сделать напрямую в реляционной базе. Нужна дополнительная таблица-посредник.

## **Вывод:**

В ходе выполнения лабораторной работы были освоены основы работы с библиотеками SQLAlchemy и Alembic для управления реляционной базой данных в Python.

Были созданы ORM-модели пользователей, адресов, продуктов и заказов, настроены связи между ними и запущены PostgreSQL в Docker без установки СУБД на компьютер.

С помощью Alembic я были сгенерированы и применены миграции, что позволило безопасно изменять структуру базы данных без ручного вмешательства.

При наполнении БД тестовыми данными и выполнении запросов к связанным таблицам убедились, что всё работает корректно.

Работа показала, насколько удобно и надёжно можно управлять базой данных через код: легко воспроизводятся и не зависят от конкретного окружения. Полученные навыки могут быть использованы в будущих проектах для построения гибких и поддерживаемых приложений.

**Ссылка на репозиторий:**

**<https://github.com/bepis-art/application-development-2025-urfu.git>**