Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПЕРВОГО ПРЕЗИДЕНТА РОССИИ Б.Н. ЕЛЬЦИНА
(УрФУ имени первого Президента России Б.Н. Ельцина)
Институт радиоэлектроники и информационных технологий — РТФ
Школа профессионального и академического образования

ВНЕДРЕНИЕ DEPENDENCY INJECTION И SQLALCHEMY В LITESTAR

Отчет по лабораторной работе №3

по дисциплине «Разработка приложений»

	Дата	Подпись	
Преподаватель:	_____	_____	Кузьмин Д.
Студенты:	_____	_____	Пантелеев Е.А.

Группа: РИМ-150950

Екатеринбург

2025

ЦЕЛЬ РАБОТЫ

Освоить принципы Dependency Injection и интеграцию SQLAlchemy ORM в веб-приложении на базе фреймворка Litestar написав CRUD.

ХОД РАБОТЫ

1. Репозиторий

В начале работы в папку с лабораторной работой №3 был перенесен файл models.py с моделями SQLAlchemy и для дальнейшего обращения к БД. Затем был создан файл schemas.py для валидации полей.

schemas.py > Config

```python
from pydantic import BaseModel
from typing import Optional, List
import uuid
from datetime import datetime


GigaCode: explain | explain step by step | doc | test
class UserBase(BaseModel):
    """Базовые поля, общие для создания и обновления"""
    username: str
    email: str
    description: Optional[str] = None

GigaCode: explain | explain step by step | doc | test
class UserCreate(UserBase):
    """Для создания пользователя - наследует все от UserBase"""
    pass

GigaCode: explain | explain step by step | doc | test
class UserUpdate(BaseModel):
    """Для обновления пользователя - все поля опциональны"""
    username: Optional[str] = None
    email: Optional[str] = None
    description: Optional[str] = None

GigaCode: explain | explain step by step | doc | test
class UserResponse(UserBase):
    """Для ответа API - добавляем системные поля"""
    id: uuid.UUID
    created_at: datetime
    updated_at: datetime

GigaCode: explain | explain step by step | doc | test
class UsersListResponse(BaseModel):
    """Для вывода всех пользователей с общим количеством"""
    users: List[UserResponse]
    total_count: int

GigaCode: explain | explain step by step | doc | test
class Config:
    from_attributes = True
```

После этого был создан файл репозитория для непосредственного обращения к БД.

```python
from sqlalchemy import select, func
from sqlalchemy.ext.asyncio import AsyncSession
from typing import List, Optional
import uuid
from models import User
from schemas import UserCreate, UserUpdate


GigaCode: explain | explain step by step | doc | test
class UserRepository:
    def __init__(self, session: AsyncSession):
        self.session = session
    async def get_by_id(self, user_id: uuid.UUID) -> Optional[User]:
        """Получить пользователя по ID"""
        result = await self.session.execute(
            select(User).where(User.id == user_id)
        )
        return result.scalar_one_or_none()

    async def get_by_filter(self, count: int = 10, page: int = 1, **kwargs) -> List[User]:
        """Получить пользователей с фильтрацией и пагинацией"""
        query = select(User)

        # Применяем фильтры
        for key, value in kwargs.items():
            if hasattr(User, key):
                query = query.where(getattr(User, key) == value)

        # Пагинация
        query = query.offset((page - 1) * count).limit(count)

        result = await self.session.execute(query)
        return list(result.scalars().all())

    async def create(self, user_data: UserCreate) -> User:
        """Создать нового пользователя"""
        user = User(
            username=user_data.username,
            email=user_data.email,
            description=user_data.description
        )
        self.session.add(user)
        await self.session.commit()
        await self.session.refresh(user)
        return user
```

```python
    async def update(self, user_id: uuid.UUID, user_data: UserUpdate) -> Optional[User]:
        """Обновить пользователя"""
        user = await self.get_by_id(user_id)
        if not user:
            return None

        # Обновляем только переданные поля
        update_data = user_data.dict(exclude_unset=True)
        for field, value in update_data.items():
            setattr(user, field, value)

        await self.session.commit()
        await self.session.refresh(user)
        return user

    async def delete(self, user_id: uuid.UUID) -> None:
        """Удалить пользователя"""
        user = await self.get_by_id(user_id)
        if user:
            await self.session.delete(user)
            await self.session.commit()

    async def get_total_count(self, **kwargs) -> int:
        """Получить общее количество пользователей с фильтрацией"""
        query = select(func.count(User.id))

        for key, value in kwargs.items():
            if hasattr(User, key):
                query = query.where(getattr(User, key) == value)

        result = await self.session.execute(query)
        count = result.scalar()

        return count
```

2. Сервисный слой

Для описания бизнес-логики, хотя бы и минимальной в данной работе необходим сервисный слой, он был реализован в файле user_service.py.

services > user_service.py > ...

```python
from typing import List, Optional
import uuid

from repositories.user_repository import UserRepository
from models import User
from schemas import UserCreate, UserUpdate

GigaCode: explain | explain step by step | doc | test
class UserService:
    def __init__(self, user_repository: UserRepository):
        self.user_repository = user_repository

    async def get_by_id(self, user_id: uuid.UUID) -> Optional[User]:
        """Получить пользователя по ID"""
        return await self.user_repository.get_by_id(user_id)

    async def get_by_filter(self, count: int, page: int, **kwargs) -> list[User]:
        """Получить пользователей с фильтрацией и пагинацией"""
        return await self.user_repository.get_by_filter(count=count, page=page, **kwargs)

    async def create(self, user_data: UserCreate) -> User:
        """Создать нового пользователя"""
        return await self.user_repository.create(user_data)

    async def update(self, user_id: uuid.UUID, user_data: UserUpdate) -> User:
        """Обновить пользователя"""
        return await self.user_repository.update(user_id, user_data)

    async def delete(self, user_id: int) -> None:
        """Удалить пользователя"""
        return await self.user_repository.delete(user_id)
    async def get_total_count(self, **kwargs) -> int:
        """Получить общее количество пользователей"""
        return await self.user_repository.get_total_count(**kwargs)
```

3. Контроллер

Для взаимодействия с внешним миром и управления пользователями в БД был создан контроллер.

```python
from litestar import Controller, get, post, put, delete
from litestar.params import Parameter, Body
from litestar.di import Provide
from litestar.exceptions import NotFoundException
from typing import List, Optional
import uuid

from services.user_service import UserService
from models import User
from schemas import UserCreate, UserUpdate, UserResponse, UsersListResponse

GigaCode: explain | explain step by step | doc | test
class UserController(Controller):
    path = "/users"
#    dependencies = {"user_service": Provide("user_service")}

    @get("/{user_id:uuid}")
    async def get_user_by_id(
        self,
        user_service: UserService,
        user_id: uuid.UUID = Parameter()
        ) -> UserResponse:
        """Получить пользователя по ID"""
        user = await user_service.get_by_id(user_id)
        if not user:
            raise NotFoundException(detail=f"User with ID {user_id} not found")
        return UserResponse.model_validate(user)

    @get()
    async def get_all_users(
        self,
        user_service: UserService,
        count: int = Parameter(gt=0, le=100, default=10),
        page: int = Parameter(ge=1, default=1),
        username: Optional[str] = Parameter(default=None),
    ) -> UsersListResponse:
        """Получить всех пользователей"""
        # Собираем фильтры
        filters = {}
        if username:
            filters["username"] = username

        users = await user_service.get_by_filter(count=count, page=page, **filters)
        total_count = await user_service.get_total_count(**filters)

        user_responses = [UserResponse.model_validate(user, from_attributes=True) for user in users]
```

```python
        user_responses = [UserResponse.model_validate(user, from_attributes=True) for user in users]
        response = UsersListResponse(
            users=user_responses,
            total_count=total_count
        )

        return response

    @post()
    async def create_user(
        self,
        user_service: UserService,
        data: UserCreate = Body(),
    ) -> UserResponse:
        """Создать нового пользователя"""
        user = await user_service.create(data)
        response = UserResponse.model_validate(user, from_attributes=True)

        return response

    @delete("/{user_id:uuid}")
    async def delete_user(
        self,
        user_service: UserService,
        user_id: uuid.UUID = Parameter(),
    ) -> None:
        """Удалить пользователя"""
        await user_service.delete(user_id)

    @put("/{user_id:uuid}")
    async def update_user(
        self,
        user_service: UserService,
        user_id: uuid.UUID = Parameter(),
        data: UserUpdate = Body(),
    ) -> UserResponse:
        """Обновить пользователя"""
        user = await user_service.update(user_id, data)
        if not user:
            raise NotFoundException(detail=f"User with ID {user_id} not found")
        return UserResponse.model_validate(user)

```

4. Главное приложение

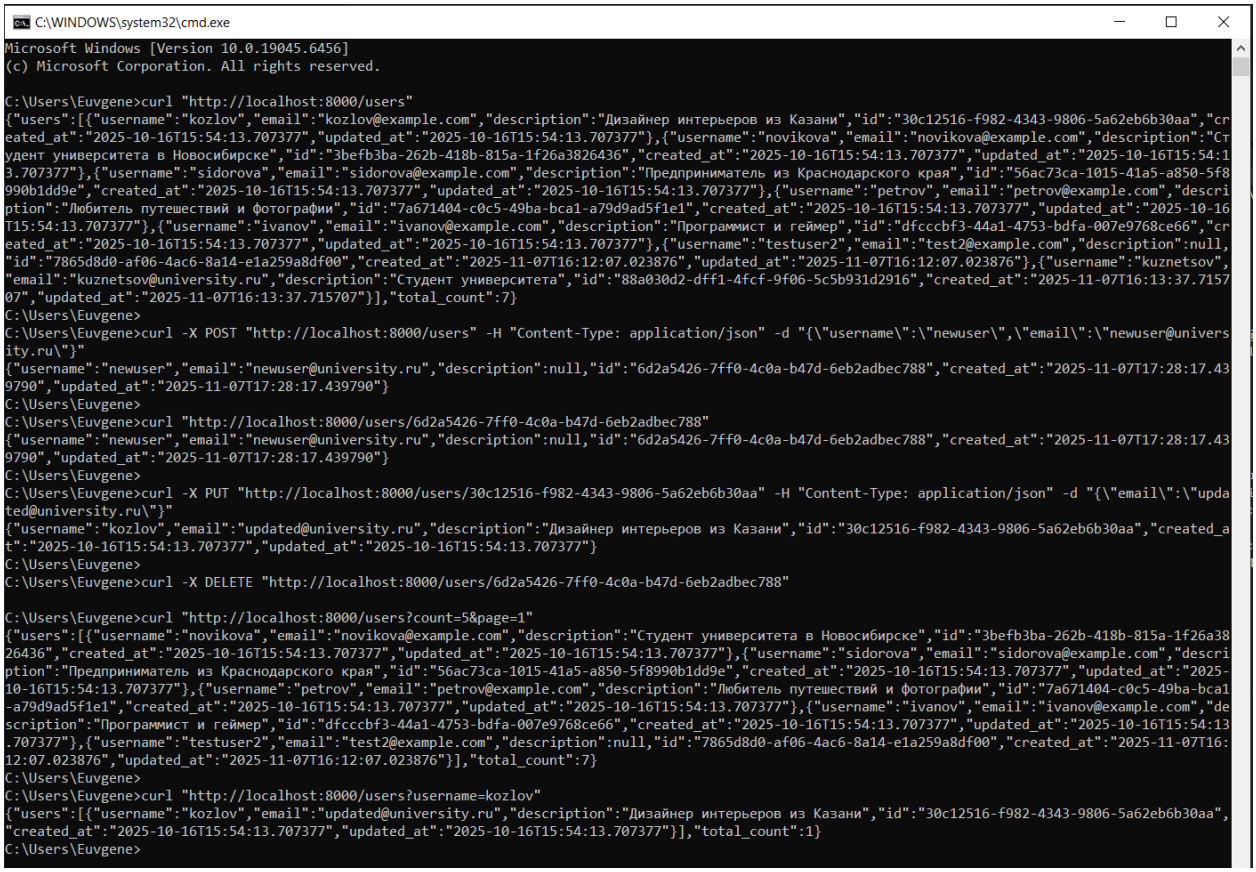
Для сбора всех файлов вместе и корректной работы было создано главное приложение в файле main.py. Весь функционал представлен сразу для задания со звездочкой, с выводом количества всех юзеров.

main.py > ...

```python
from repositories.user_repository import UserRepository
from services.user_service import UserService
from controllers.user_controller import UserController
from schemas import UserCreate
from dotenv import load_dotenv
import os
from litestar import Litestar
from litestar.di import Provide
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession, async_sessionmaker

load_dotenv()

# Настройка базы данных
DB_URL = os.getenv("DB_URL")

# Создаем асинхронный движок
engine = create_async_engine(DB_URL, echo=True)

# Создаем фабрику для асинхронных сессий
async_session_factory = async_sessionmaker(
    engine,
    class_=AsyncSession,
    expire_on_commit=False
)

GigaCode: explain | explain step by step | doc | test
async def provide_db_session() -> AsyncSession:
    """Провайдер сессии базы данных"""
    async with async_session_factory() as session:
        try:
            yield session
        finally:
            await session.close()
GigaCode: explain | explain step by step | doc | test
async def provide_user_repository(db_session: AsyncSession) -> UserRepository:
    """Провайдер репозитория пользователей"""
    return UserRepository(db_session)

GigaCode: explain | explain step by step | doc | test
async def provide_user_service(user_repository: UserRepository) -> UserService:
    """Провайдер сервиса пользователей"""
    return UserService(user_repository)

app = Litestar(
    route_handlers=[UserController],
    dependencies={
        "db_session": Provide(provide_db_session),
        "user_repository": Provide(provide_user_repository),
        "user_service": Provide(provide_user_service),
    },
)

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

5. Демонстрация работы

Для оценки работы проверим сделанные нами функции с помощью терминала и curl. Можно все делать и через веб-интерфейс Swagger, но через консоль показалось проще.



Команды для работы приложения будут представлены в README файле в репозитории.

6. Ответы на вопросы

1. **Объясните принцип Dependency Injection (DI) своими словами.** Какую проблему он решает в контексте разработки приложений и какие преимущества дает?

Основной принцип заложен в названии Внедрение зависимостей, то есть зависимости должны внедряться, а не создаваться внутри класса. Это дает больше гибкости: можно менять реализации и сервисы, простоту в тестировании, можно тестить отдельно блоки, ну и проще поддерживать.

2. **Каковы основные обязанности каждого из трех слоев приложения (Repository, Service, Controller)?** Почему такое разделение считается хорошей практикой? Что произойдет, если объединить логику репозитория и контроллера?

Контроллер по сути работает с внешним миром, принимает HTTP-запросы, преобразует JSON в python-объекты.
Сервисный слой отвечает за всю бизнес-логику, может координировать работу нескольких репозиториев, интегрируется с внешними сервисами.
Репозиторий работает напрямую с БД и строит SQL-запросы.
Это разделение является хорошей практикой, потому что дает разделение ответственности, тестируемость, гибкость в случае замены продуктов и безопасность.
Если объединить репозиторий и контроллер, то мы сразу получаем огромный файл, с которым очень сложно работать, нужен огромный класс, который делает все, нельзя так просто заменить какой-либо сервис и также это небезопасно, так как уровень внешнего мира и БД становятся одним.

3. **Объясните жизненный цикл зависимости в Litestar.** Как именно создается и когда уничтожается экземпляр сессии базы данных при обработке одного HTTP-запроса?

Жизненный цикл Litestar состоит из следующих этапов:

1. приходит http-запрос
2. создается сессия к БД
3. репозиторий получает сессию
4. сервис получает репозиторий
5. выполняется бизнес-логика
6. формируется ответ
7. закрывается сессия

4. **Что такое `async/await` и зачем они используются в данном приложении?** Как асинхронность влияет на производительность при работе с базой данных?

Async/await нужны для асинхронного выполнения кода, это позволяет ускорить его выполнения за счет того, что во время ожидания запроса из БД, другой код может работать параллельно.

5. **Почему в сигнатурах методов** UserRepository **первым аргументом передается** session**?** Почему бы не создать его внутри репозитория? Кто и когда должен вызывать session.commit() или session.rollback()?

Session передается для инъекции зависимости и изоляции запросов. В репозитории мы бы создавали миллион различных подключений к БД, что тяжело отлаживать и тестировать.

Сервсный слой должен вызывать commit и rollback, если у нас сложная бизнес-логика для нормального оформления транзакций (ACID).D нашем коде commit в репозитории, потому нам необходимо было реализовать простой CRUD и нет необходимости пока что в сложной бизнес-логике.

6. **Для чего в методе get_by_filter используется пагинация (count и page)?**

Для увеличения производительности, так как не нужно держать соединение с БД и выводить миллион записей и держать их в памяти, например. Также для удобства юзера, проще смотреть на 10 записей, чем на 1000.

7. **В текущей реализации `UserService` практически не содержит бизнес-логики и является "прокси" для `UserRepository`.** Приведите пример конкретной бизнес-логики (например, проверка уникальности email, хеширование пароля, отправка приветственного письма), которую можно было бы добавить в сервисный слой.
   Написал в основном коде пример

```python
async def create(self, user_data: UserCreate) -> User:
    """Создать нового пользователя"""
    # Проверка уникальности email
    existing_user = await self.user_repository.get_by_email(user_data.email)
    if existing_user:
        raise ValueError(f"Email {user_data.email} уже занят")

    # Валидация возраста
    if user_data.age < 18:
        raise ValueError("Минимальный возраст - 18 лет")

    # Хеширование пароля
    if hasattr(user_data, 'password'):
        user_data.password = self._hash_password(user_data.password)

    return await self.user_repository.create(user_data)
    #return await self.user_repository.create(user_data)
```

8. **Какие HTTP-статусы должны возвращать каждый из эндпоинтов (`get`, `post`, `put`, `delete`) в различных сценариях (успех, ошибка, не найден)?** Обоснуйте свой выбор.

Эндпоинт get
Код 200 – если эндопинт что-то возвращает, но не создает что-то новое
Код 400 – неверные параметры
Код 404 – например, UUID не найден
Код 500 – проблема на стороне сервера
Эндпоинт post
Код 201 – если эндпоинт что-то создает
Код 400 Bad Request – невалидные данные
Код 409 Conflict – конфликт данных (email уже существует)
Код 422 Unprocessable Entity – ошибки валидации Pydantic
Код 500 – ошибка сервера
Эндпоинт put
Код 200 – пользователь обновлен
Код 400 – неверные параметры
Код 404 – пользователь не найден
Код 409 Conflict – конфликт данных (email уже существует)
Код 500 – проблема на стороне сервера
Эндпоинт delete
Код 204 – пользователь удален
Код 404 – пользователь не найден
Код 500 – проблема на стороне сервера

ЗАКЛЮЧЕНИЕ

В ходе работы были освоены принципы Dependency Injection и сделана интеграция SQLAlchemy ORM в веб-приложение на базе фреймворка Litestar с написанием CRUD.