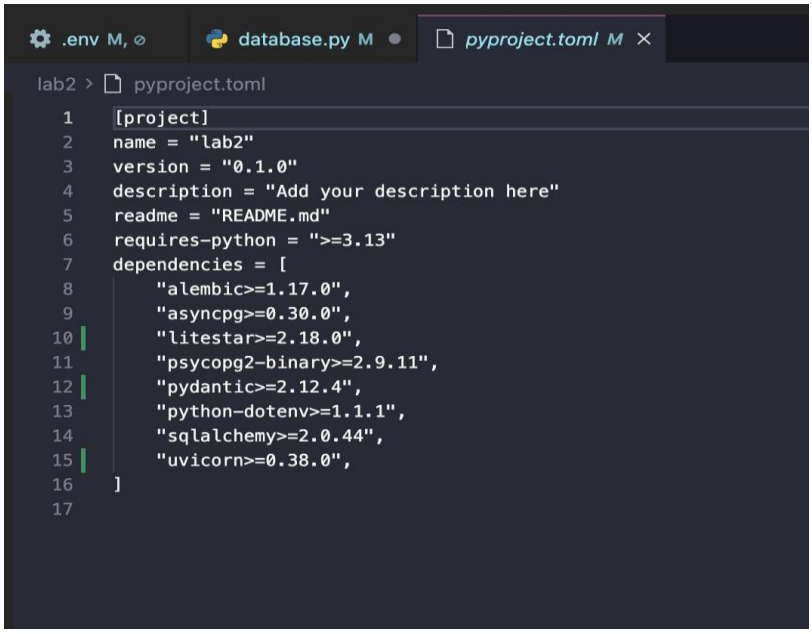
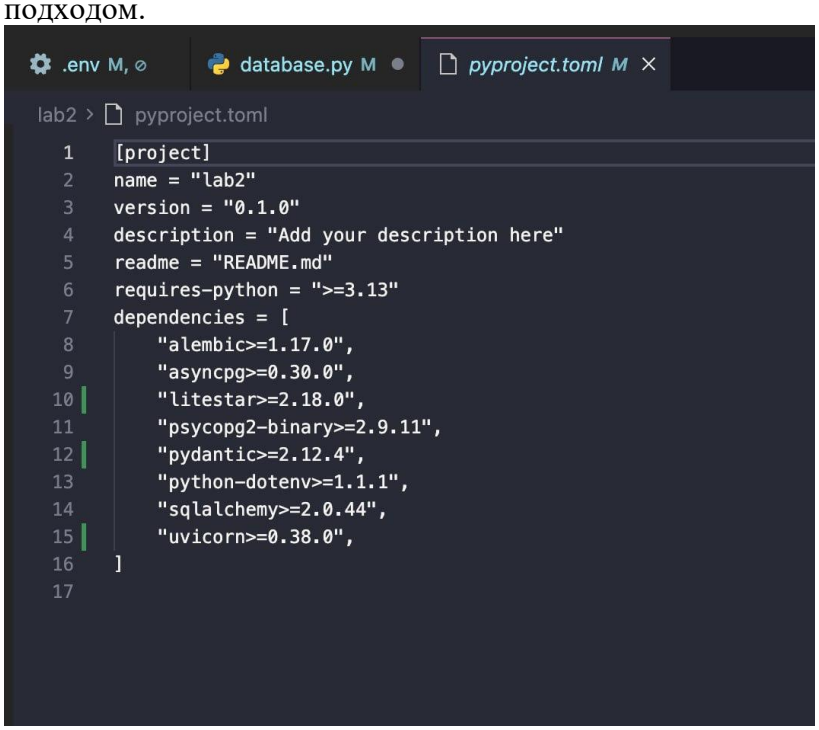
Лабораторная работа 3
«Внедрение Dependency Injection и SQLAlchemy в Litestar»
Жунев Андрей РИМ-15090

Подготовка проекта
Для последнего коммита по выполнению лабораторной 2 был установлен тег git tag lab_2.
Далее были установлены необходимые зависимости для выполнения лабораторной 3:

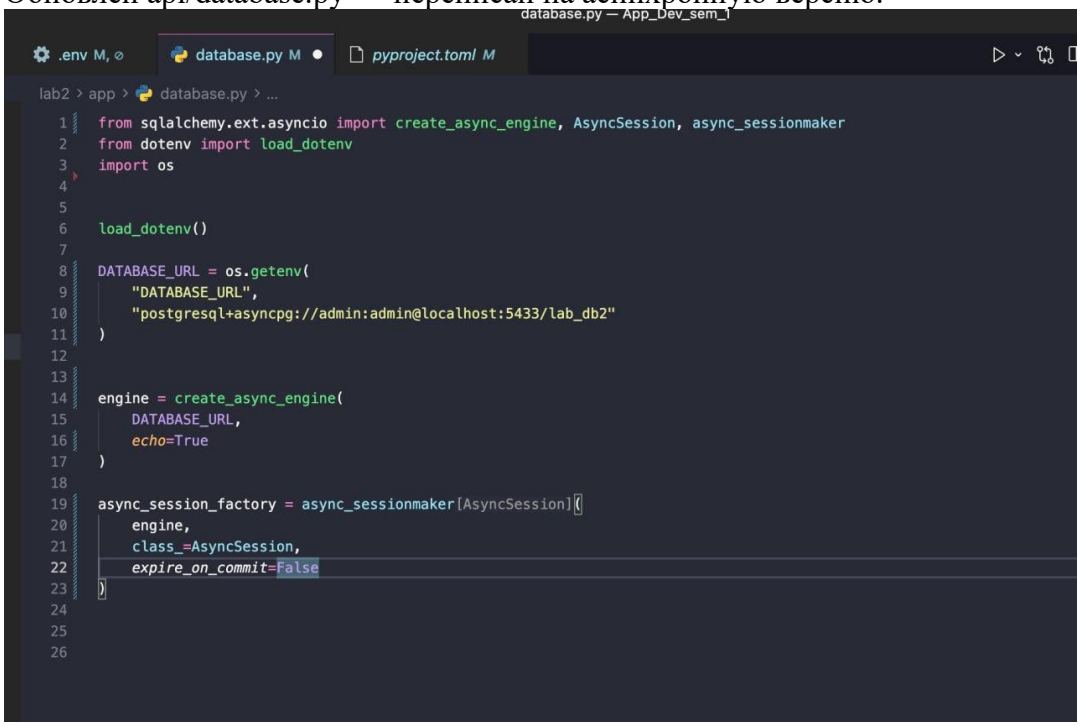
- **litestar**
- **pydantic**
- **uvicorn**



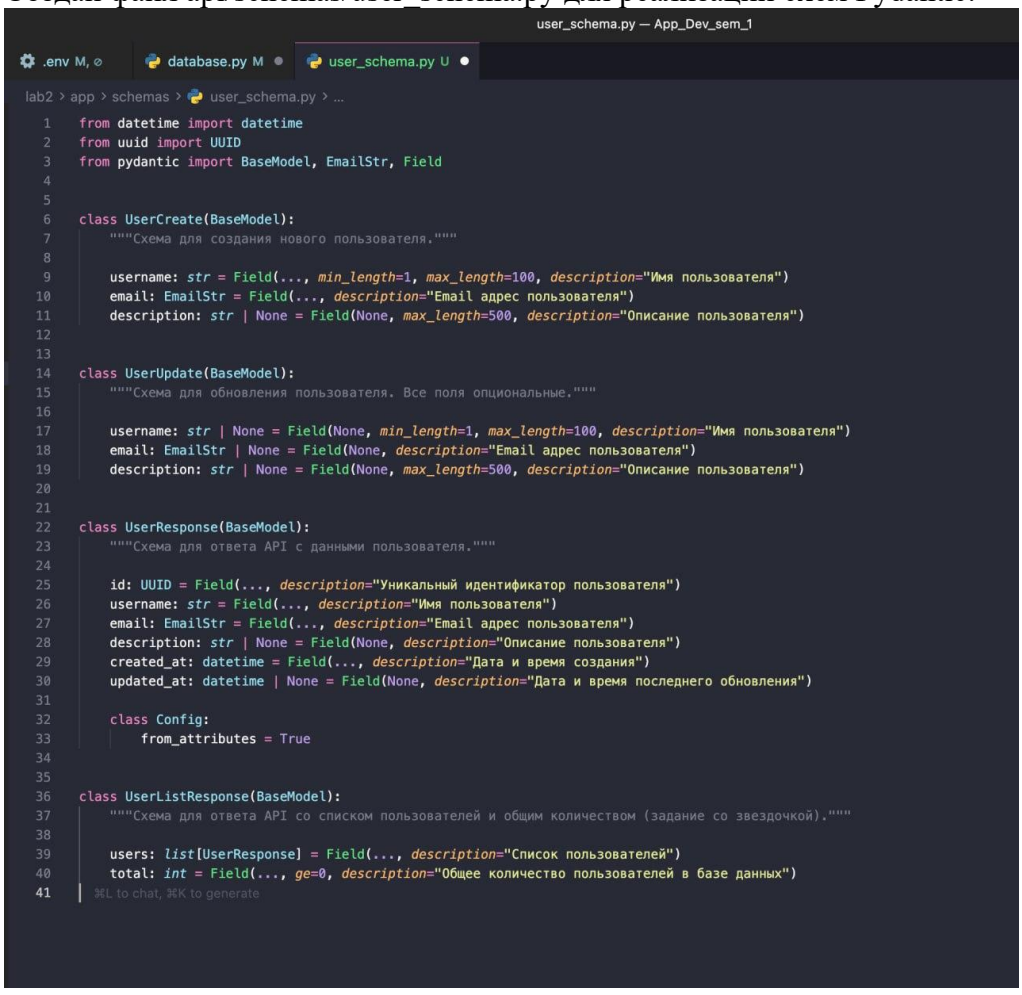
1. Настройка окружения
Был отредактирован .env файл — изменен DATABASE_URL для доступа к БД асинхронным подходом.



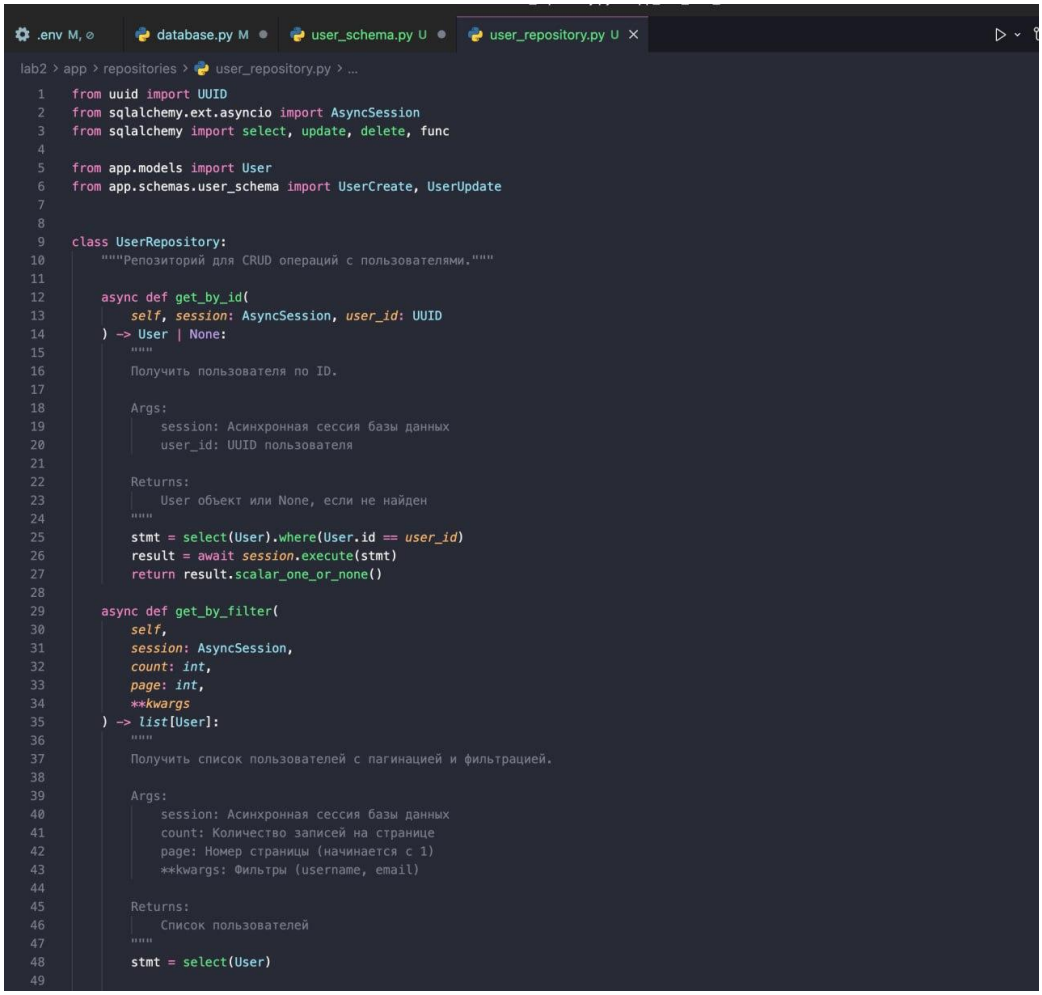
2. Обновление базы данных
Обновлен api/database.py — переписан на асинхронную версию.



3. Создание схем Pydantic
Создан файл api/schemas/user_schema.py для реализации схем Pydantic.



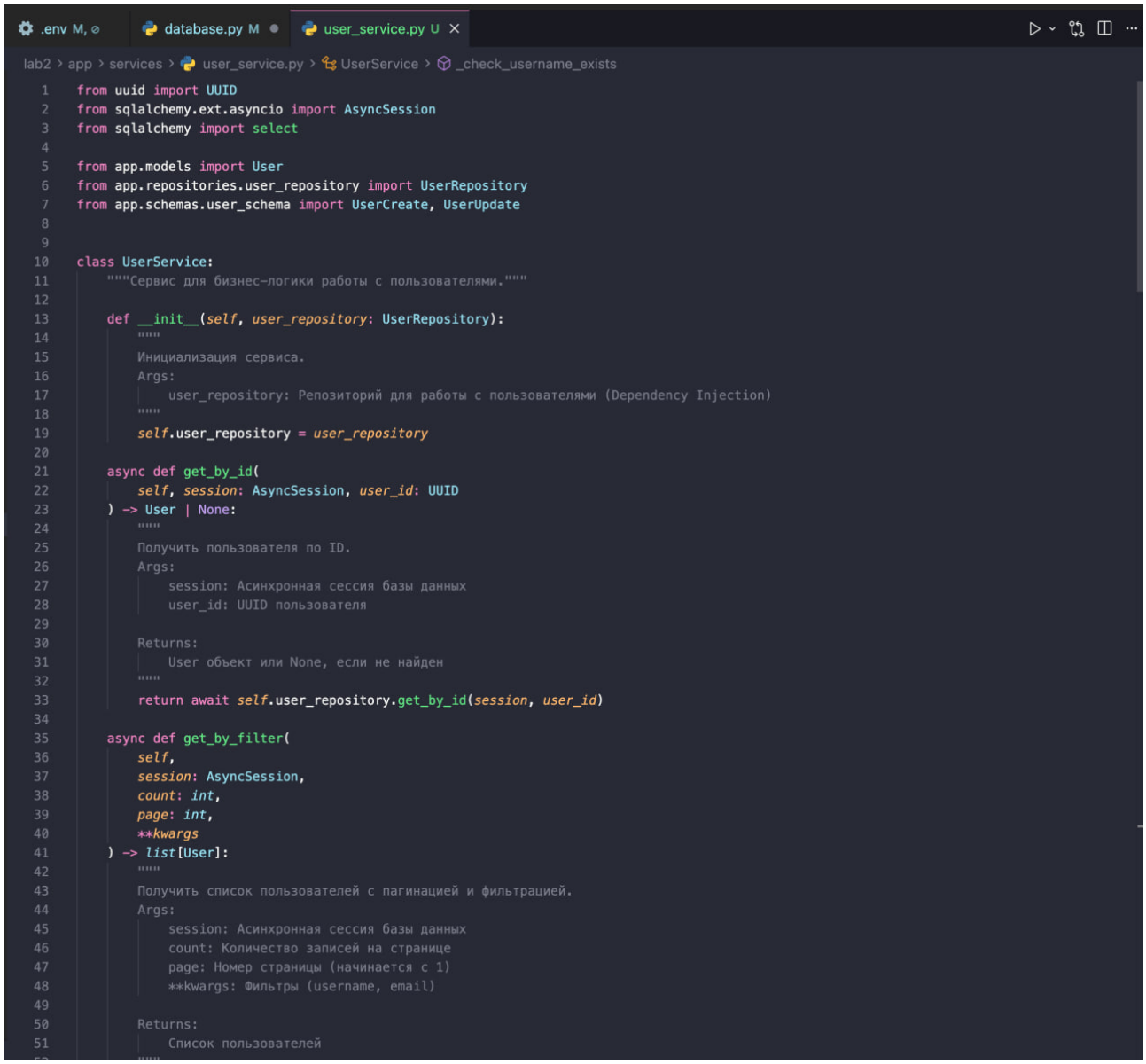
4. Реализация репозитория
Реализован репозиторий для взаимодействия с базой данных.



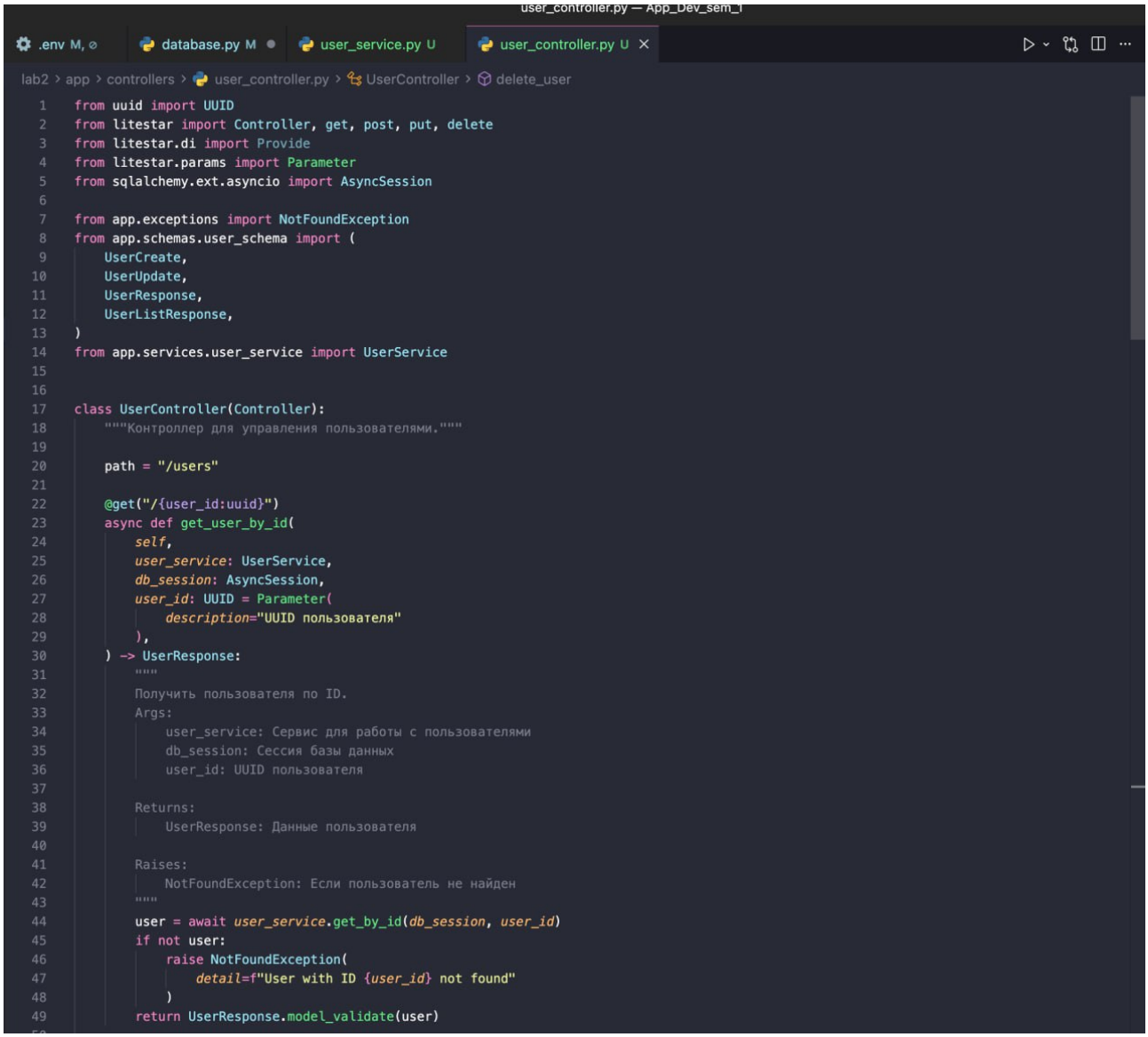

```python
from uuid import UUID
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy import select, update, delete, func

from app.models import User
from app.schemas.user_schema import UserCreate, UserUpdate


class UserRepository:
    """Репозиторий для CRUD операций с пользователями."""

    async def get_by_id(
        self, session: AsyncSession, user_id: UUID
    ) -> User | None:
        """
        Получить пользователя по ID.

        Args:
            session: Асинхронная сессия базы данных
            user_id: UUID пользователя

        Returns:
            User объект или None, если не найден
        """
        stmt = select(User).where(User.id == user_id)
        result = await session.execute(stmt)
        return result.scalar_one_or_none()

    async def get_by_filter(
        self,
        session: AsyncSession,
        count: int,
        page: int,
        **kwargs
    ) -> list[User]:
        """
        Получить список пользователей с пагинацией и фильтрацией.

        Args:
            session: Асинхронная сессия базы данных
            count: Количество записей на странице
            page: Номер страницы (начинается с 1)
            **kwargs: Фильтры (username, email)

        Returns:
            Список пользователей
        """
        stmt = select(User)
```

```python
    async def count(
        self, session: AsyncSession, **kwargs
    ) -> int:
        """
        Получить общее количество пользователей с учетом фильтров.
        Используется для задания со звездочкой.

        Args:
            session: Асинхронная сессия базы данных
            **kwargs: Фильтры (username, email)

        Returns:
            Количество пользователей
        """
        stmt = select(func.count(User.id))

        if "username" in kwargs and kwargs["username"]:
            stmt = stmt.where(User.username.ilike(f"%{kwargs['username']}%"))
        if "email" in kwargs and kwargs["email"]:
            stmt = stmt.where(User.email.ilike(f"%{kwargs['email']}%"))

        result = await session.execute(stmt)
        return result.scalar_one() or 0
```

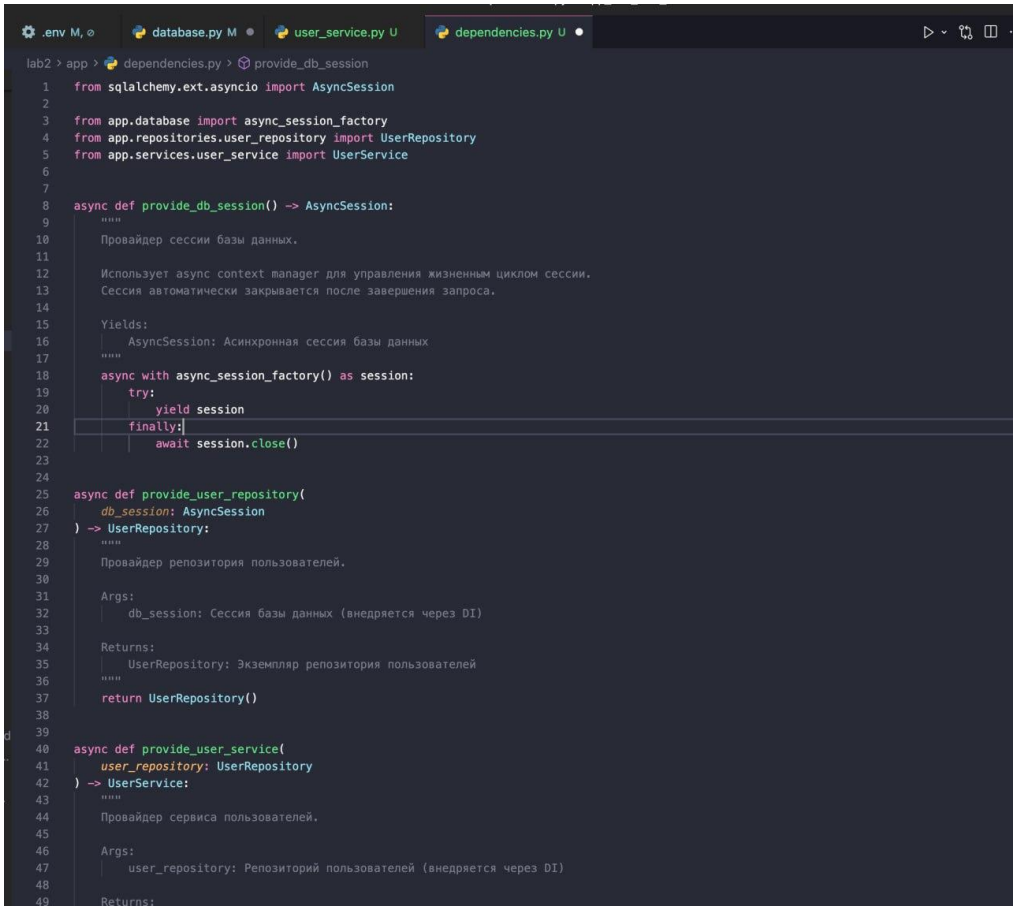
5. Сервисный слой
Реализован сервисный слой для управления бизнес-логикой.

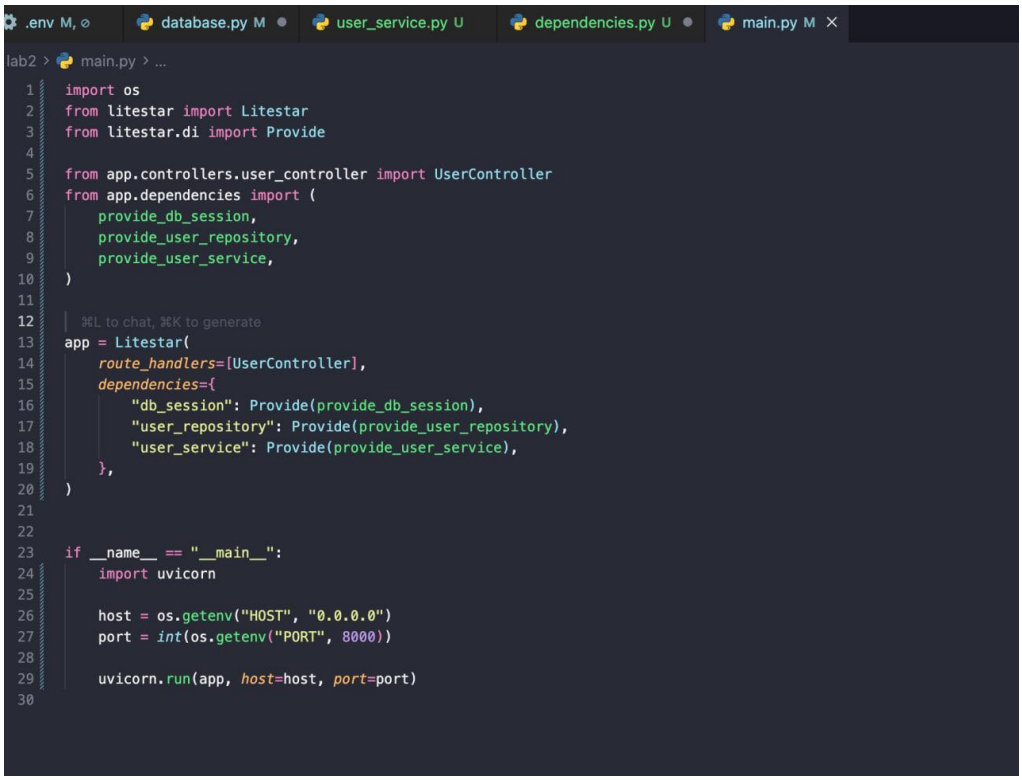


6. Контроллер
Реализован контроллер для обработки HTTP-запросов.



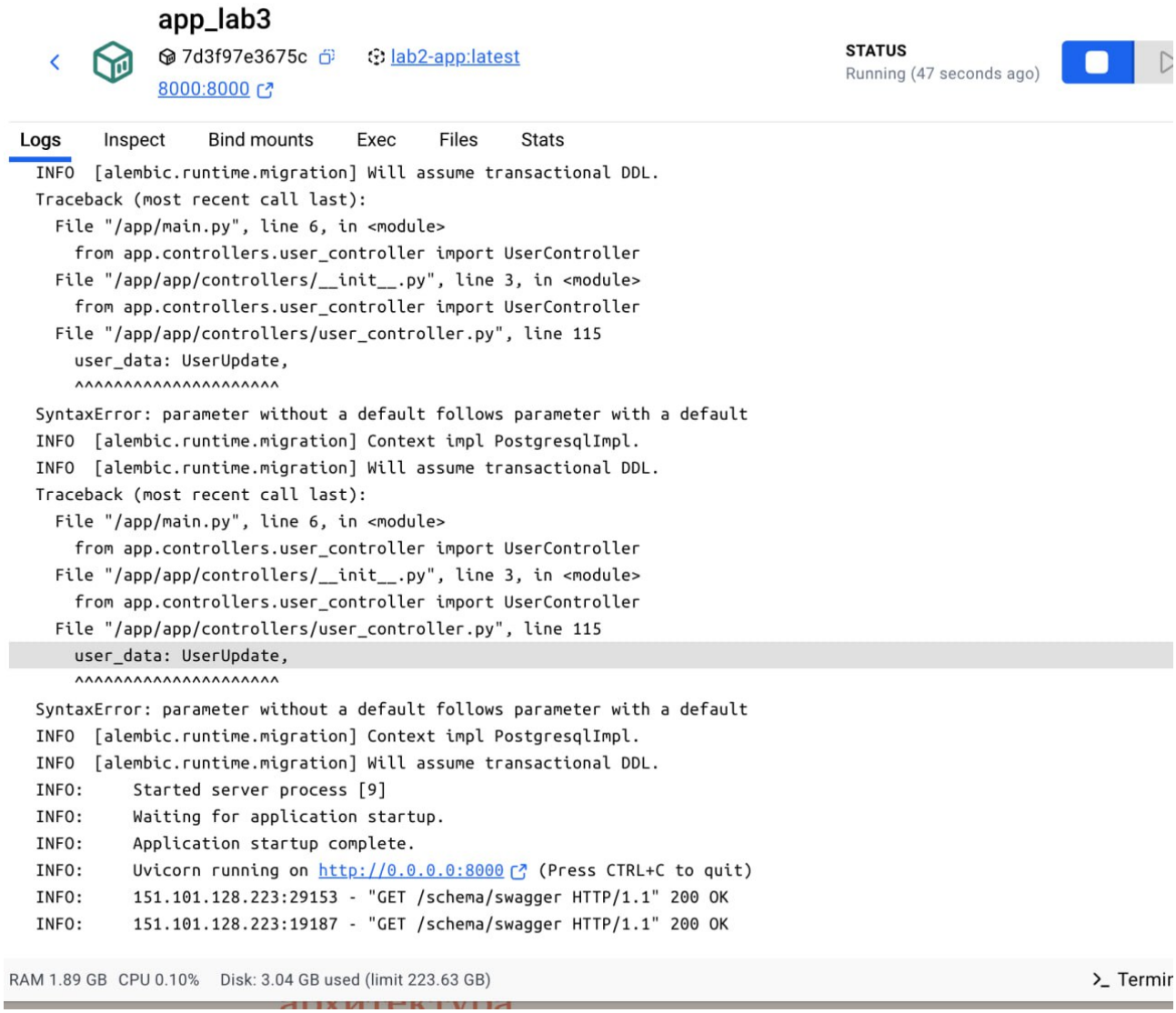
7. Настройка DI и главной точки входа
Настроены главная точка входа в приложение и Dependency Injection.

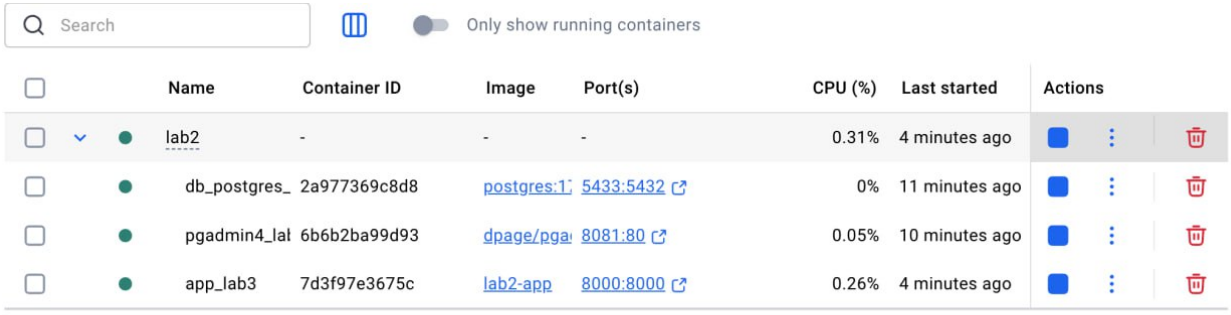




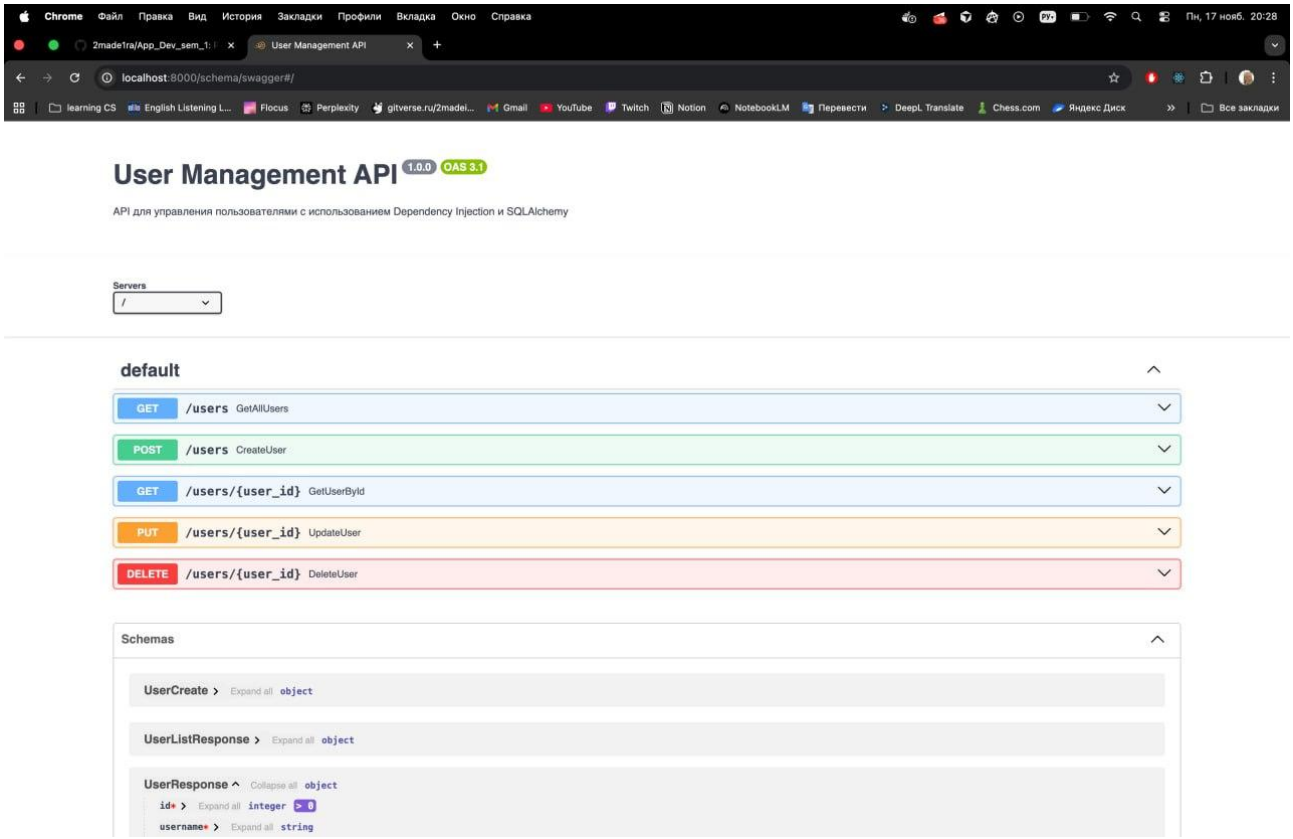
8. Документация
Документация по запуску оформлена в README.md в папке lab2.

9. Docker Compose
Для удобства запуска приложение было добавлено в docker-compose. Все контейнеры успешно запускаются.





Тестирование API
API можно проверить через Swagger-документацию по адресу: localhost:8000/schema/swagger



Ответы на вопросы:

1. Объясните принцип Dependency Injection (DI) своими словами

Это принцип проектирования при котором объекты нужные классу для работы создаются не внутри него, а передаются извне - принцип разделения ответственности. Его преимущества в том, что этот паттерн облегчает тестирование, можно легко обновить реализацию (изменить объекты, при этом не меняя реализацию самого класса), легче поддерживать.

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

Разделение приложения на три слоя - Controller, Service и Repository - делает код чище и понятнее. Контроллер отвечает только за приём HTTP-запросов, валидацию данных и формирование ответа. Он не должен знать, как устроена база или какие бизнес-правила действуют в системе. Вся логика принятия решений, проверки условий и организация работы разных частей приложения находится в сервисе. Репозиторий же занимается чисто доступом к данным: знает таблицы, построение SQL-запросов и возвращает модели. Такое разделение помогает поддерживать код, легко менять отдельные части и удобно тестировать каждый слой в изоляции.

Если же смешать контроллер и репозиторий, то контроллер начнёт делать слишком много: принимать запросы, выполнять SQL, проверять бизнес-правила. Такой код быстро становится запутанным, его сложно тестировать и расширять. Любое изменение БД или логики потянет за собой переписывание контроллера, а повторное использование логики станет практически невозможным. Поэтому слои нужны не как какая то формальность, а как реальный инструмент, который делает проект гибким, и долговечным

3. Объясните жизненный цикл зависимости в Litestar. Как создается и когда уничтожается экземпляр сессии базы при http запросе.

В Litestar жизненный цикл зависимости устроен так, что каждый http запрос получает свой собственный набор зависимостей. Когда приходит запрос, фреймворк создаёт экземпляры всех зависимостей, которые отмечены как scope=request, в том числе и сессию базы данных. То есть при обработке каждого запроса Litestar вызывает фабрику зависимости, создаёт новую SQLAlchemy сессию и передаёт её в контроллеры и сервисы, которым она нужна.

Когда запрос заканчивается - успешно или с ошибкой - Litestar автоматически вызывает "финализатор" зависимости. В случае с сессией это означает закрытие соединения с БД, откат незавершённых транзакций и освобождение ресурсов. Благодаря этому каждая сессия живёт строго в рамках одного запроса, а приложение избегает утечек соединений и конфликтов между разными обработчиками.

4. Что такое async/await и зачем они используются? Как асихронность влияет на производительность?

Async/await это синтаксис (в данном случае python) для написания асинхронного кода (async ключевое слово, указывающее, что функция асинхронная; await ключевое слово указывающие, что нужно ожидать завершения другой асинхронной операции). Данный способ позволяет писать асинхронный код так как будто он синхронный, не блокируя выполнения программы.

Асинхронность позволяет обрабатывать огромное множество запросов одновременно, не создает для каждого запроса отдельный поток, можно обрабатывать огромное количество одновременных подключений.

Асинхронность сильно влияет на производительность за счёт того, что программа не простаивает. В обычном (синхронном) коде, если вы ждёте ответа от БД 100 мс, поток в эти 100 мс просто ничего не делает. В асинхронной модели эти 100 мс могут быть использованы для обработки других запросов или задач. В итоге одно приложение сможет обслуживать гораздо больше клиентов на том же количестве ресурсов.

5. Почему в сигнатурах методов UserRepository первым аргументом передается session? Почему не создавать его внутри? Кто и когда вызывает session.commit()?

Сессия не передается в сигнатуру, потому что она не должна управлять транзакциями. Его задача — только работать с данными: добавить объект, получить записи, выполнить запрос. А решение о том, когда именно изменения должны быть сохранены, остается за сервисом. Если бы репозиторий создавал свою собственную сессию, то каждая операция жила бы в отдельной транзакции — невозможно было бы объединить несколько действий в одно целое. Например, создать пользователя и его профиль атомарно уже бы не получилось: одно бы успело сохраниться, другое — нет.

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

Пагинация нужна для того, чтобы не отдавать клиенту весь набор данных сразу, а возвращать его небольшими порциями — страницами. Если в базе тысячи или миллионы записей, запрос без ограничений будет работать долго, нагружать сервер, занимать много памяти и трафика.

Параметры count и page позволяют клиенту запросить только нужный фрагмент: например, по 10 записей за раз, переходя между страницами. В итоге интерфейс работает быстрее, а сервер не перегружается.

7. Пример бизнес логики для сервисного слоя (например, уникальность email, хэширование пароля, письмо)?

Бизнес-логика — это правила и ограничения, которые определяют работу приложения с точки зрения бизнеса, а не технических деталей. Именно в сервисном слое сосредоточены проверки, преобразования данных и последовательность действий.

1. Проверка уникальности (email, username)

Сервис решает, можно ли создать пользователя:

```
if await self._check_email_exists(session, user_data.email):
    raise ValueError("Email already exists»)
```

правило бизнеса — один email должен принадлежать только одному пользователю.

2. Хеширование пароля

```
user_data.password = pwd_context.hash(user_data.password)
```

Пароль никогда не хранится открытым текстом. Это не задача репозитория — он лишь пишет данные в БД.

3. Отправка приветственного письма

```
await self._send_welcome_email(user.email, user.username)
```

Бизнес-требование — отправить письмо после регистрации. Репозиторий не должен знать о внешних сервисах.

4. Управление транзакцией

```
await session.commit()
```

Сервис решает, когда изменения можно зафиксировать. Репозиторий делает только `add()` и `flush()`.

8. Какие http статусы должны возвращать эндпоинты get, post, put, delete?

Метод	Основной статус	Почему
GET	`200 OK`	Ресурс найден и возвращён
POST	`201 Created`	Ресурс создан
PUT	`200 OK`	Обновление ресурса
DELETE	`204 No Content`	Ресурс удалён, тело не нужно

Стандартность — клиенты понимают логику API

Правильная обработка ошибок — разные статусы = разные реакции

REST-совместимость — API становится предсказуемым

Упрощённая отладка — легко видеть, что именно произошло