Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий – РТФ

Лабораторная работа №3
Внедрение Dependency Injection и SQLAlchemy в Litestar

Студент группы РИМ – 150950: _____ Вальнева А.Д.

Екатеринбург 2025

Цель работы

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

Задачи:

1. Изучить паттерн Dependency Injection в контексте Litestar
2. Настроить асинхронное подключение к PostgreSQL через SQLAlchemy
3. Реализовать слой репозитория для работы с базой данных
4. Создать сервисный слой для бизнес-логики
5. Разработать REST API контроллер с полным набором CRUD операций
6. Реализовать пагинацию и подсчет общего количества записей

Структура проекта

```
lab3/
├── src/
│   ├── models/
│   │   ├── base.py          # Базовый класс для моделей
│   │   └── user.py          # Модель User
│   ├── schemas/
│   │   └── user.py          # Pydantic схемы
│   ├── repositories/
│   │   └── user_repository.py
│   ├── services/
│   │   └── user_service.py
│   ├── controllers/
│   │   └── user_controller.py
│   └── main.py              # Точка входа
├── create_tables.py         # Скрипт создания таблиц
├── pyproject.toml
└── uv.lock
```

Ход выполнения

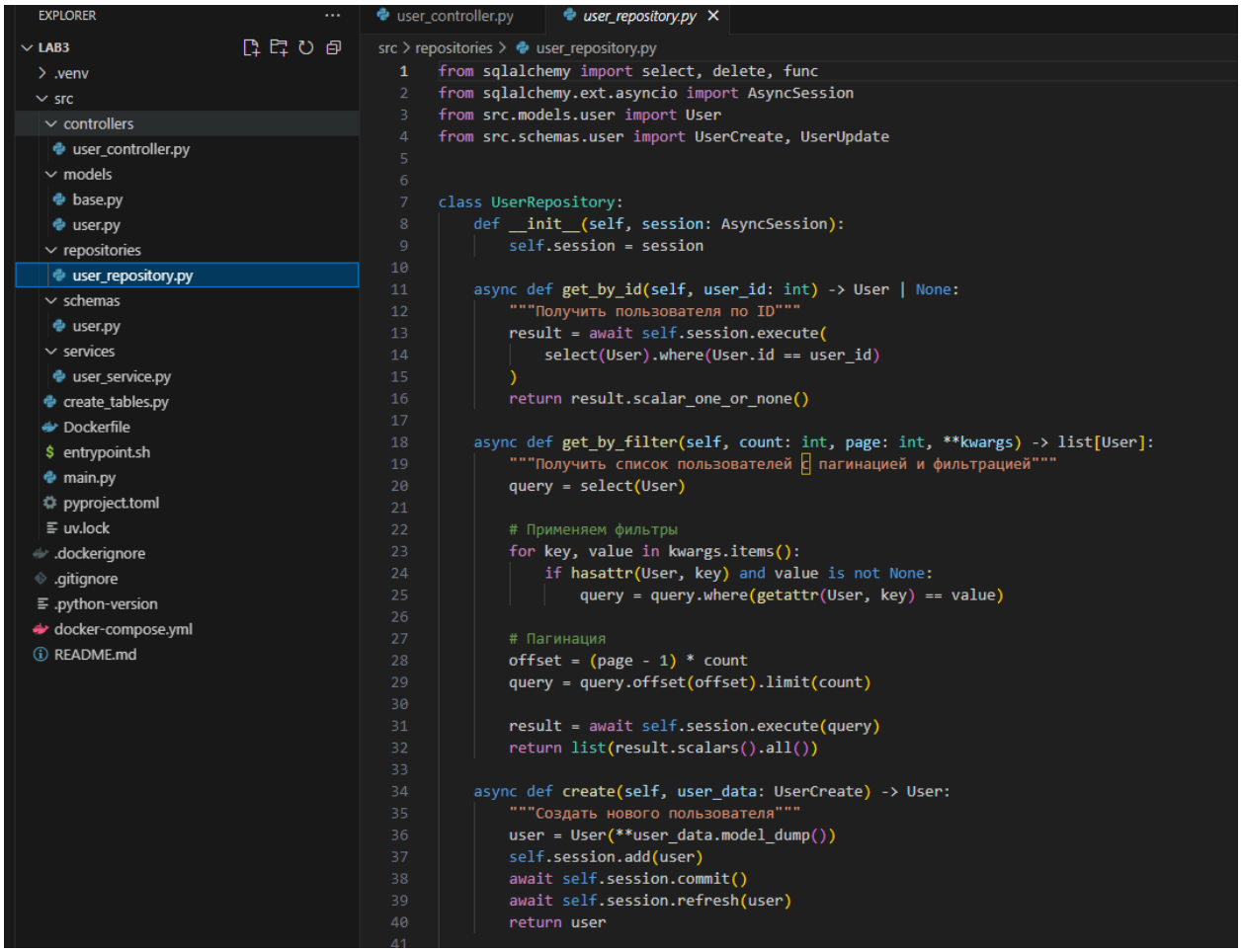
Часть 1: Создание репозитория

UserRepository – отвечает за взаимодействие с базой данных для создания, обновления, получения и удаления пользователей.

Ключевая идея: в репозитории сосредоточена вся логика работы с БД (запросы SQLAlchemy, транзакции, commit/refresh), чтобы остальные слои приложения (Service, Controller) “оставались чистыми”, т.е. работали с абстракцией и не заботились о деталях SQL.

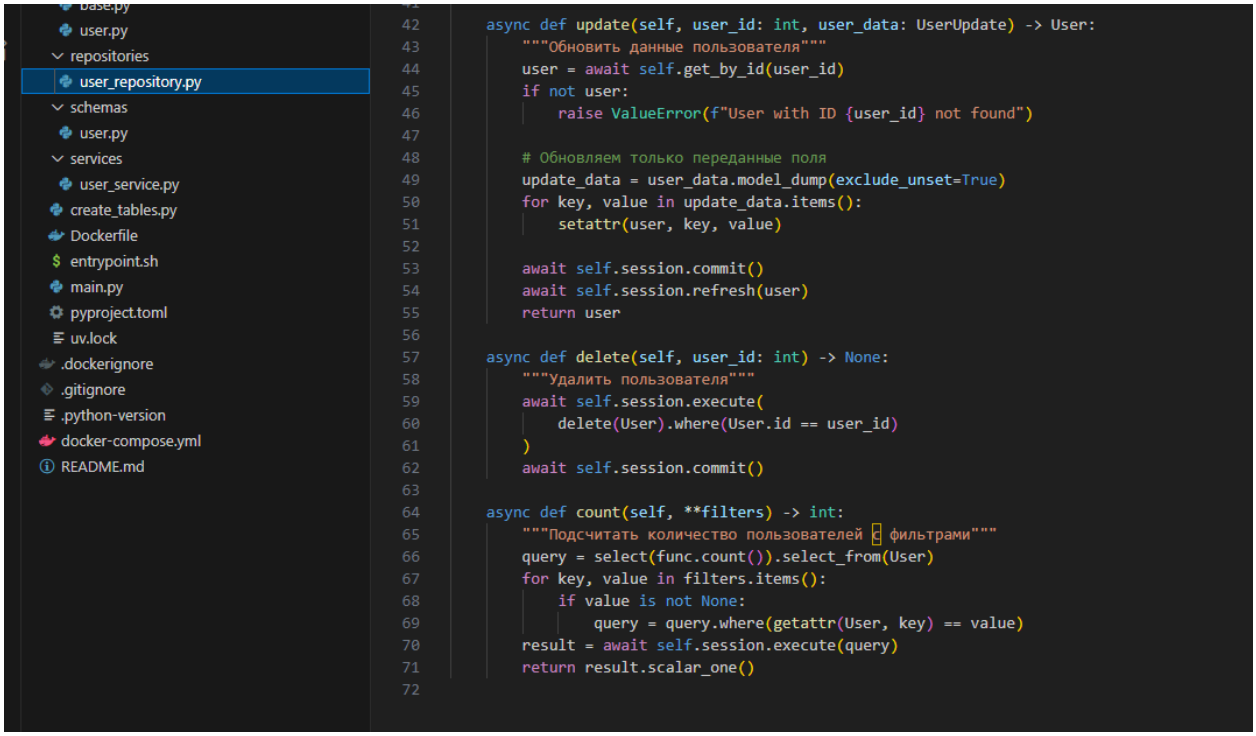
Описание работы:

- инкапсулирует всю работу с базой данных по сущности User
- все CRUD-операции выполнены асинхронно: get_by_id, get_by_filter, create, update, delete, count и используют SQLAlchemy выражения, например: result = await self. session. execute (select (User). where (User.id == user_id)).



Метод get_by_filter(self, count: int, page: int, **kwargs):

- строит базовый select(User)
- реализована динамическая фильтрация по переданным **kwargs
- **динамически добавляет where - условия по переданным kwargs**: для каждой пары (key, value) выполняется проверка hasattr(User, key) и добавляется where(getattr(User, key) == value).
- **выполняет пагинацию с вычислением смещения**: offset = (page - 1) * count и offset(offset). limit(count).
- возвращает return list(result.scalars().all()) — список сущностей User.



Метод count (self, **filters) -> int:

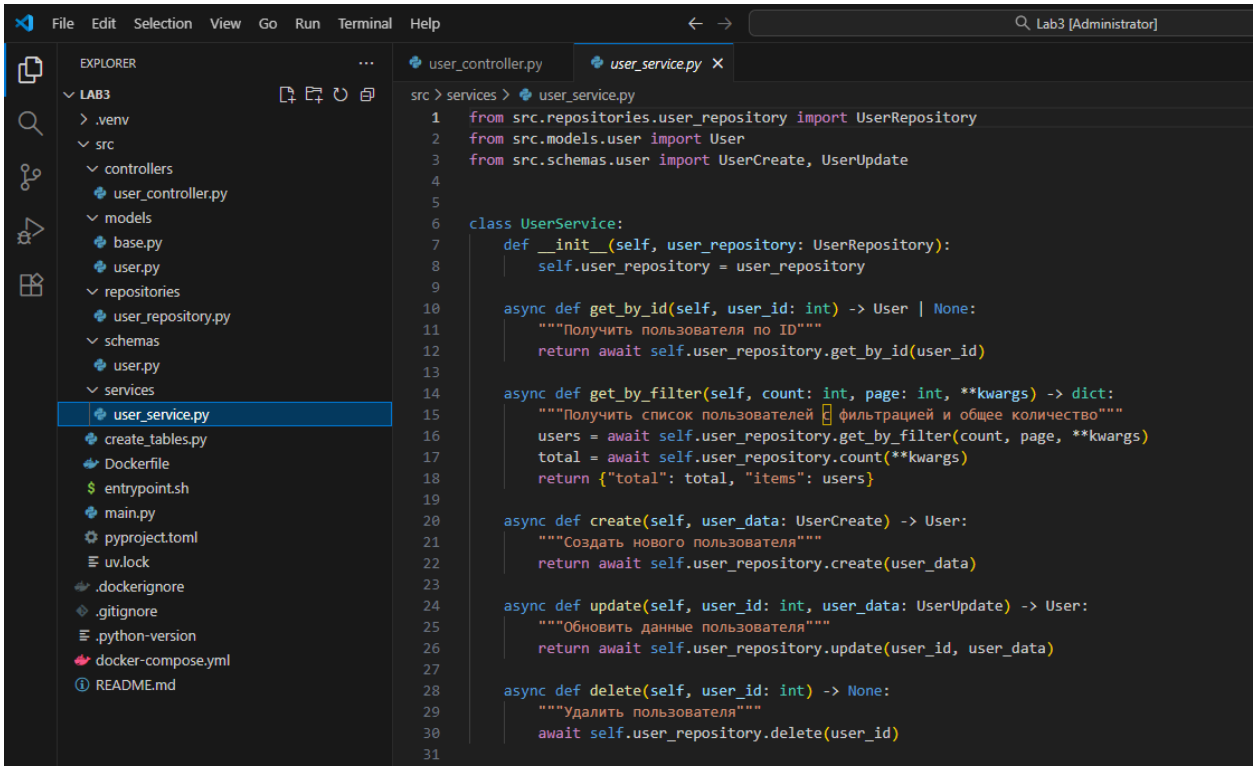
- Метод count позволяет посчитать общее количество пользователей в базе, с учётом фильтров (если они есть).
- Выполняет select(func.count()).select_from(User) и добавляет те же фильтры, что применяются в get_by_filter.

- Возвращает result.scalar_one() — количество записей, соответствующих фильтру.
- Когда пользователь делает запрос на список (get_by_filter), кроме самих записей возвращает также общее количество пользователей

Часть 2: Сервисный слой

Файл: src/services/user_service.py

UserService - слой, предназначенный для бизнес-логики приложения



Описание работы:

- разделяет бизнес-логику и работу с БД;
- делегирует реализацию UserRepository, для этого получает UserRepository через DI и хранит его в конструкторе;
- все методы асинхронные, вызывают соответствующие методы репозитория через await;
- метод get_by_filter собирает список пользователей и одновременно; запрашивает общее количество - возвращает словарь dict {"total": total, "items": users}

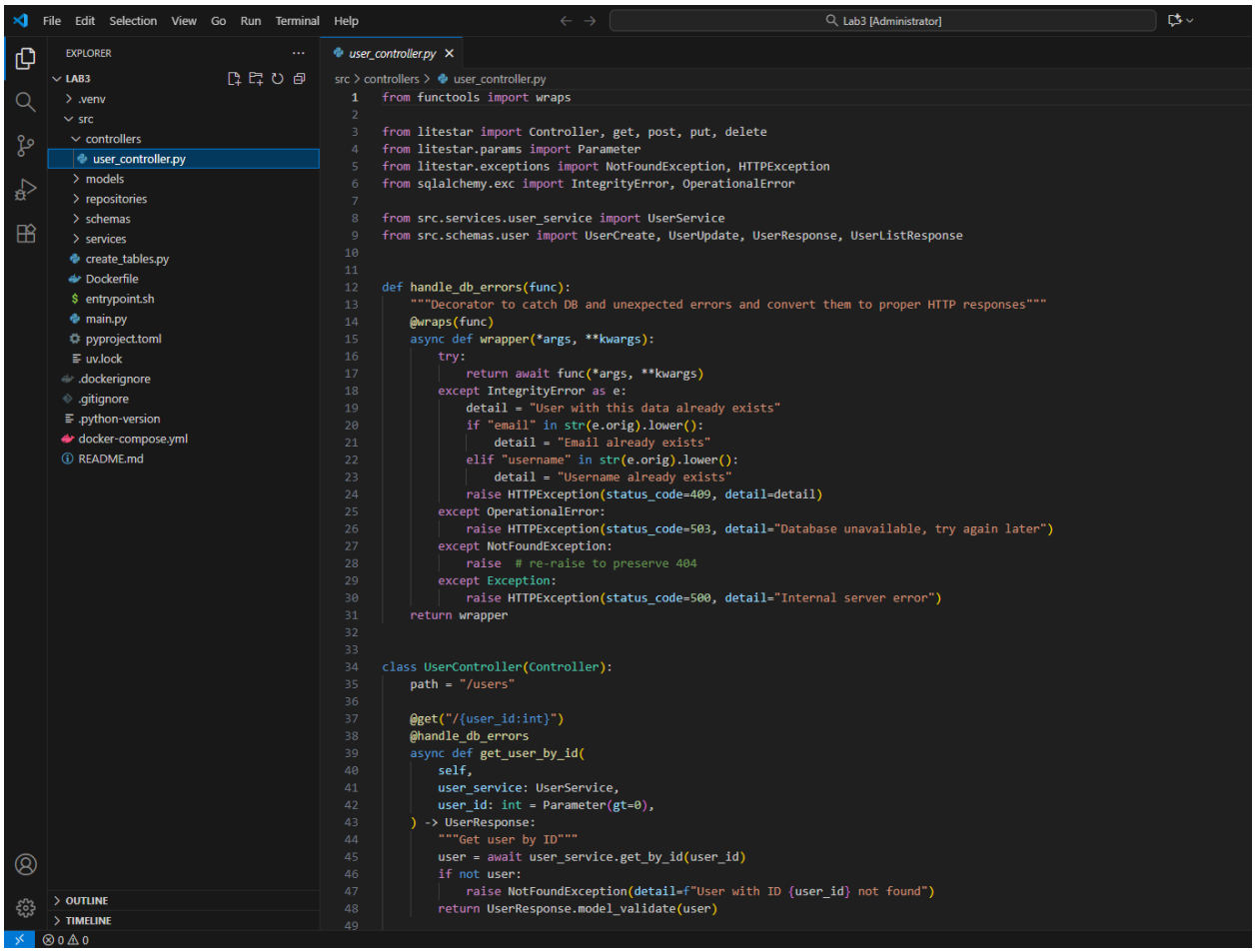
- методы create и update принимают Pydantic-схемы (UserCreate, UserUpdate) и создают/обновляют пользователя через репозиторий.
- метод delete удаляет пользователя по id, снова вызывая репозиторий и не возвращая значения.

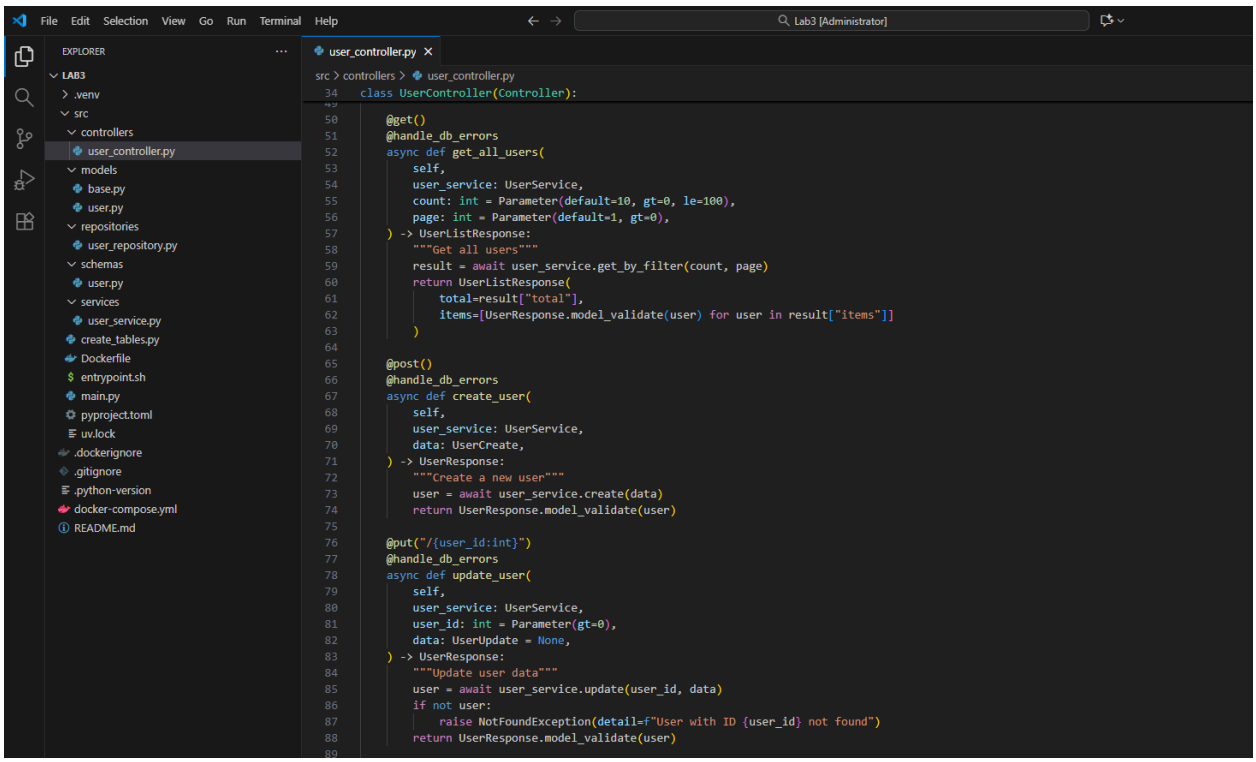
Часть 3: Слой контроллер

Файл: src/controllers/user_controller.py

UserController – представляет «внешнюю» часть API: описывает маршруты, гарантирует корректность входных данных, централизует обработку ошибок и возвращает ответы клиенту. Бизнес-логика и работа с базой вынесены в сервис/репозиторий, поэтому контроллер остаётся компактным.

Описание работы:





1) UserController задаёт path /users и содержит методы обработчики HTTP-запросов:

- GET /users/{user_id} --> get_user_by_id
- GET /users --> get_all_users (пагинация через query params count и page)
- POST /users --> create_user
- PUT /users/{user_id} --> update_user
- DELETE /users/{user_id} --> delete_user

Эти декораторы (@get, @post, @put, @delete) связывают функции контроллера с конечными точками API.

2) Dependency Injection (DI)

Каждый хэндлер принимает user_service: UserService — Litestar автоматически внедряет сервис. Контроллер не знает о базе напрямую: он делегирует логику сервису/репозиторию, то есть соблюдается разделение слоёв.

3) Пагинация с ограничением до 100 записей (count: int = Parameter(default=10, gt=0, le=100), page: int = Parameter(default=1, gt=0)), т.е. по умолчанию count=10, page=1, но le=100 запрещает запрашивать больше 100 записей за раз.

В репозитории это также реализовано через offset = (page - 1) * count и limit(count).

4) Реализована централизованная обработка ошибок - handle_db_errors:

Декоратор handle_db_errors обрабатывает ошибки БД и преобразует их в HTTP-ошибки:

- IntegrityError --> HTTP 409 (дубликаты, с разбором «email/username» для сообщения);
- OperationalError --> HTTP 503 (БД недоступна);
- NotFoundException пробрасывается дальше, чтобы вернуть 404;
- остальные ошибки --> HTTP 500;

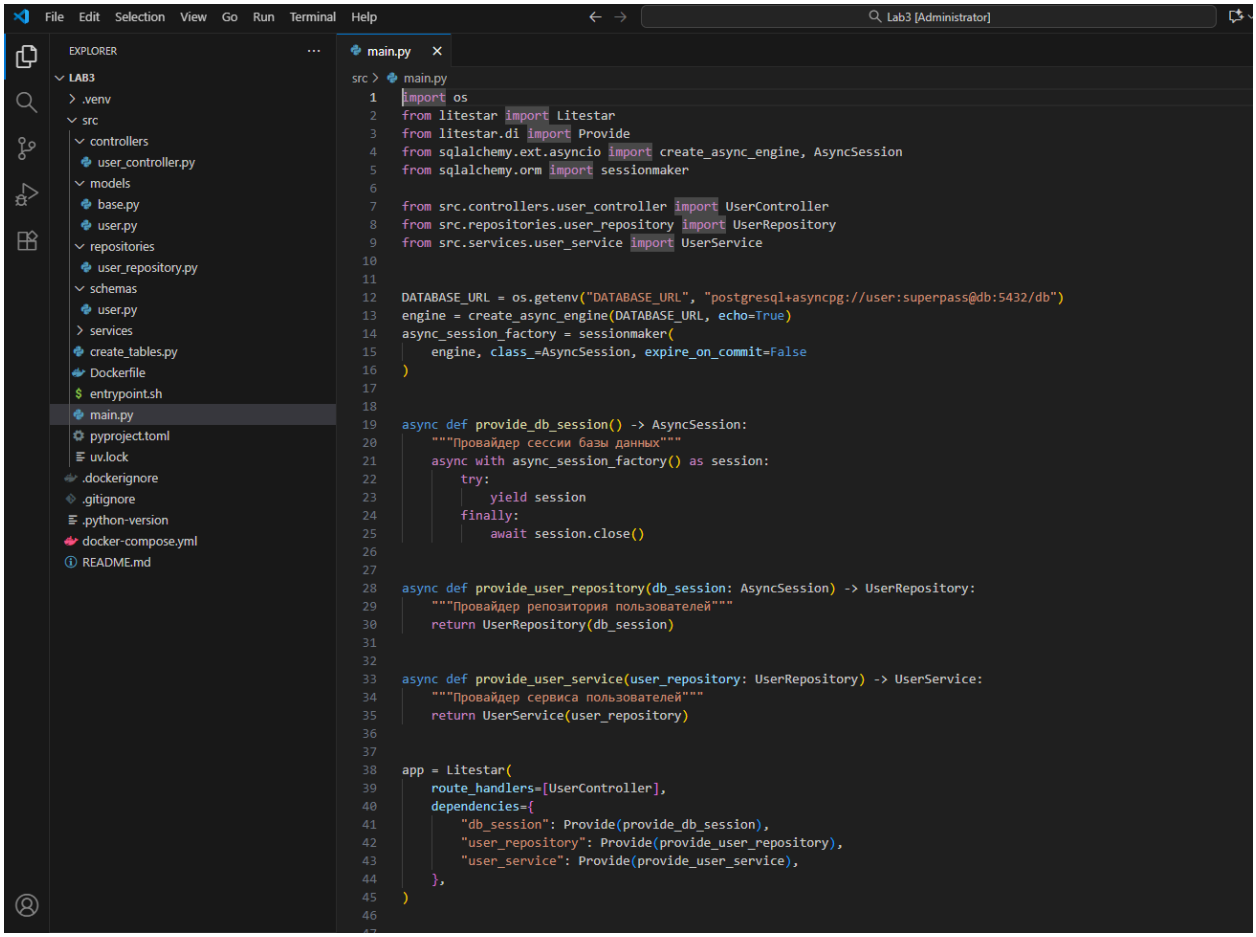
Это избавляет от повторяющегося кода обработки ошибок в каждом методе и скрывает внутренние детали БД от клиента.

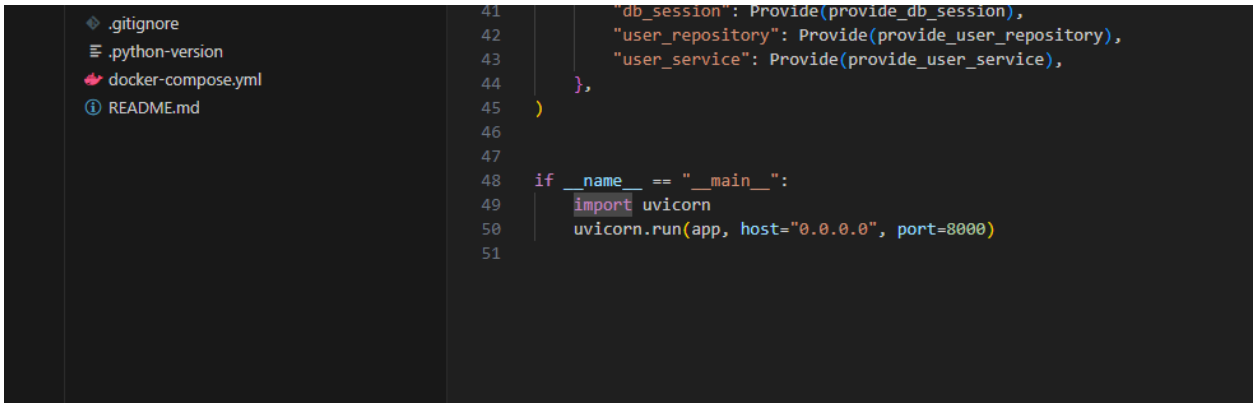
5) Возврат UserListResponse с общим количеством пользователей.

Функция get_all_users вызывает сервис, который возвращает total=result["total"], items=[UserResponse.model_validate(user) for user in result["items"]]. Контроллер оборачивает это в UserListResponse)

Часть 4: Главное приложение с Dependency Injection

Файл: src/main.py





Описание Dependency Injection:

Цепочка зависимостей:

- db_session → user_repository → user_service → UserController

Преимущества:

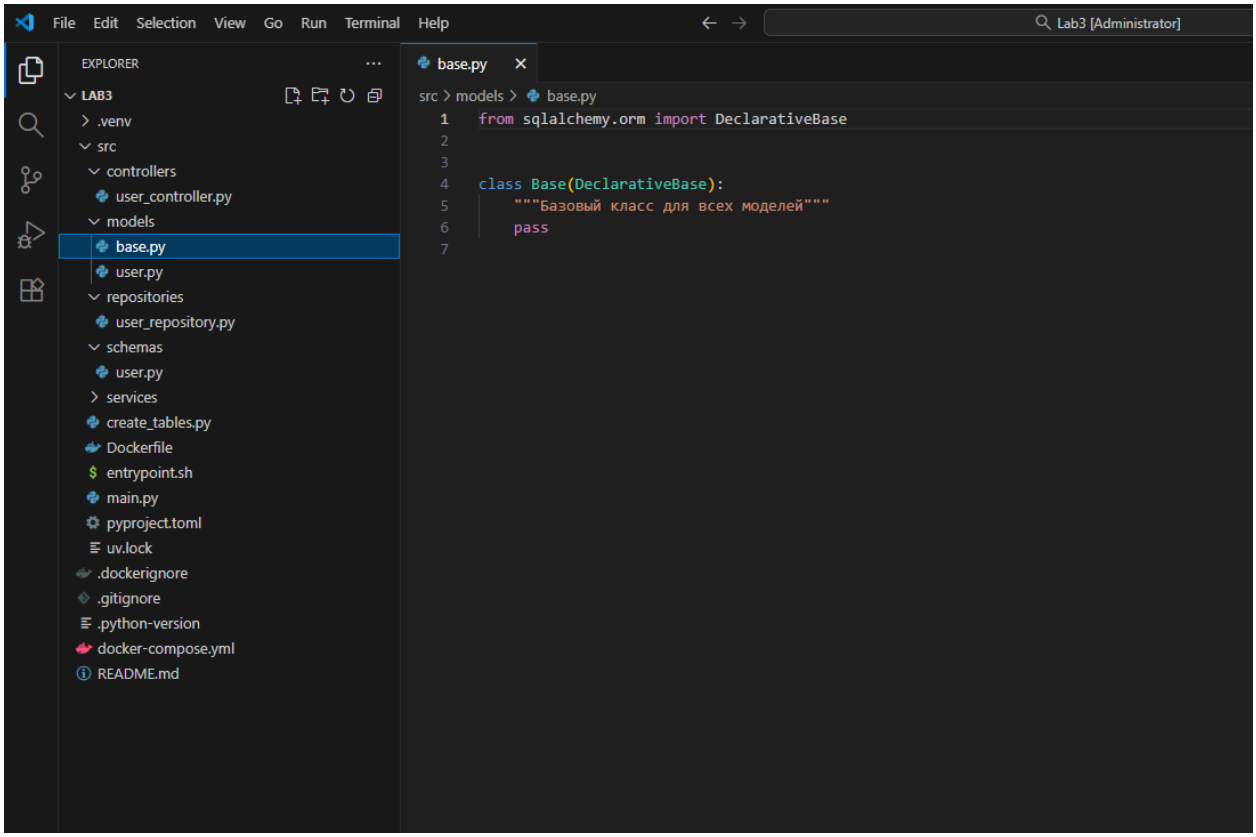
- Автоматическое управление жизненным циклом зависимостей
- Правильное закрытие сессий БД через yield
- Легкость тестирования (можно подменить зависимости)
- Явная декларация зависимостей

Описание работы:

- Litestar видит, что UserController требует UserService
- Запрашивает user_service из контейнера зависимостей
- Для создания UserService нужен UserRepository
- Для UserRepository нужен db_session
- Создает сессию БД через provide_db_session
- Выстраивает всю цепочку автоматически

Часть 5: Models

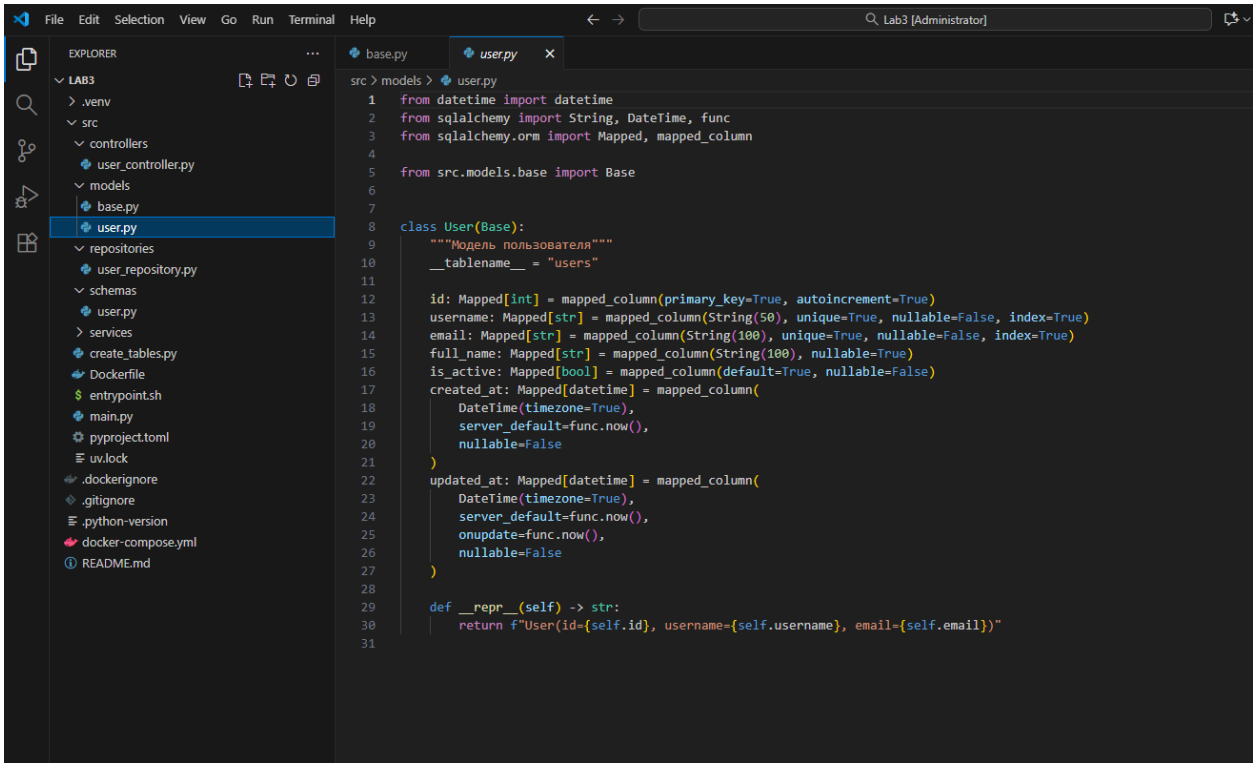
Файл: src/models/base.py



- `Базовый класс для всех моделей`

- Base - базовый класс для всех моделей, от которого они наследуются

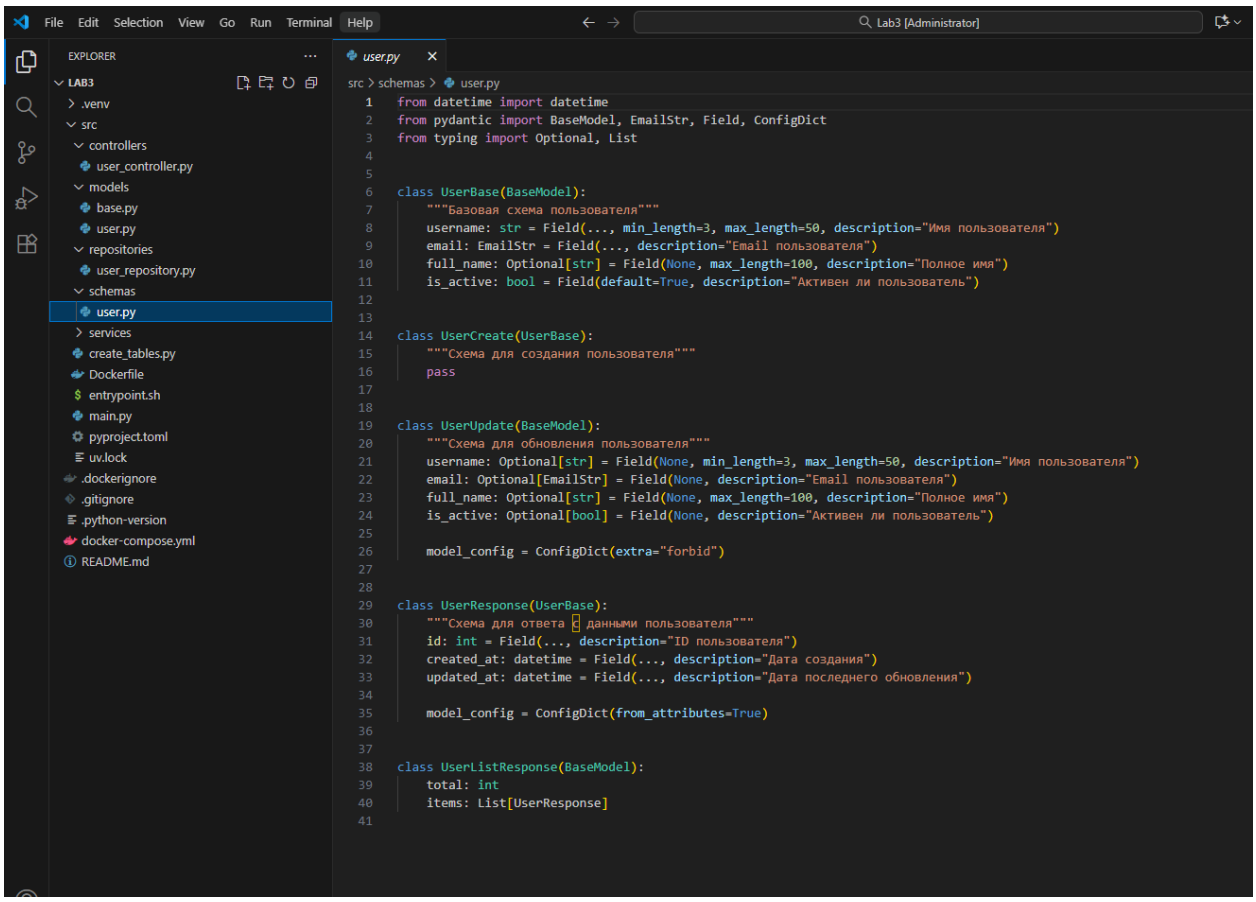
Файл: src/models/user.py



- User - модель пользователя.
- Поля username и email уникальны и индексированы для быстрого поиска
- Автоматическое проставление временных меток создания и обновления
- server_default=func.now() - значение по умолчанию устанавливается на уровне БД

Часть 6: Pydantic схемы

Файл: src/schemas/user.py



Для ответа сервиса используются следующие схемы:

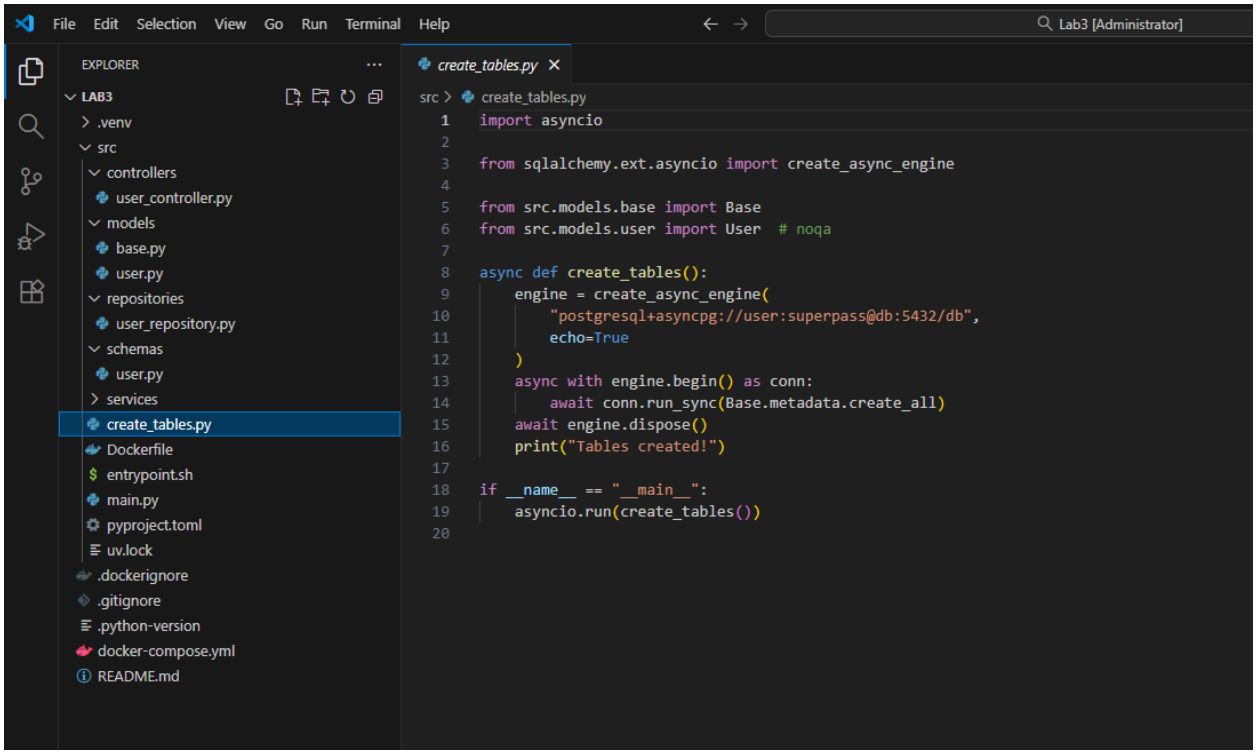
- UserCreate - валидация входных данных при создании (все поля обязательны)
- UserUpdate - валидация при обновлении (все поля опциональны, model_config = ConfigDict(extra="forbid") запрещает лишние поля)
- UserResponse - формат ответа API (from_attributes=True для конвертации из ORM объектов)
- UserListResponse - ответ со списком пользователей и общим количеством (задание со звездочкой)

UserListResponse объединяет список пользователей и total — это позволяет вернуть и данные текущей страницы, и общее количество записей в одном ответе:

- return UserListResponse(total=total, items=[UserResponse.model_validate(u) for u in users])

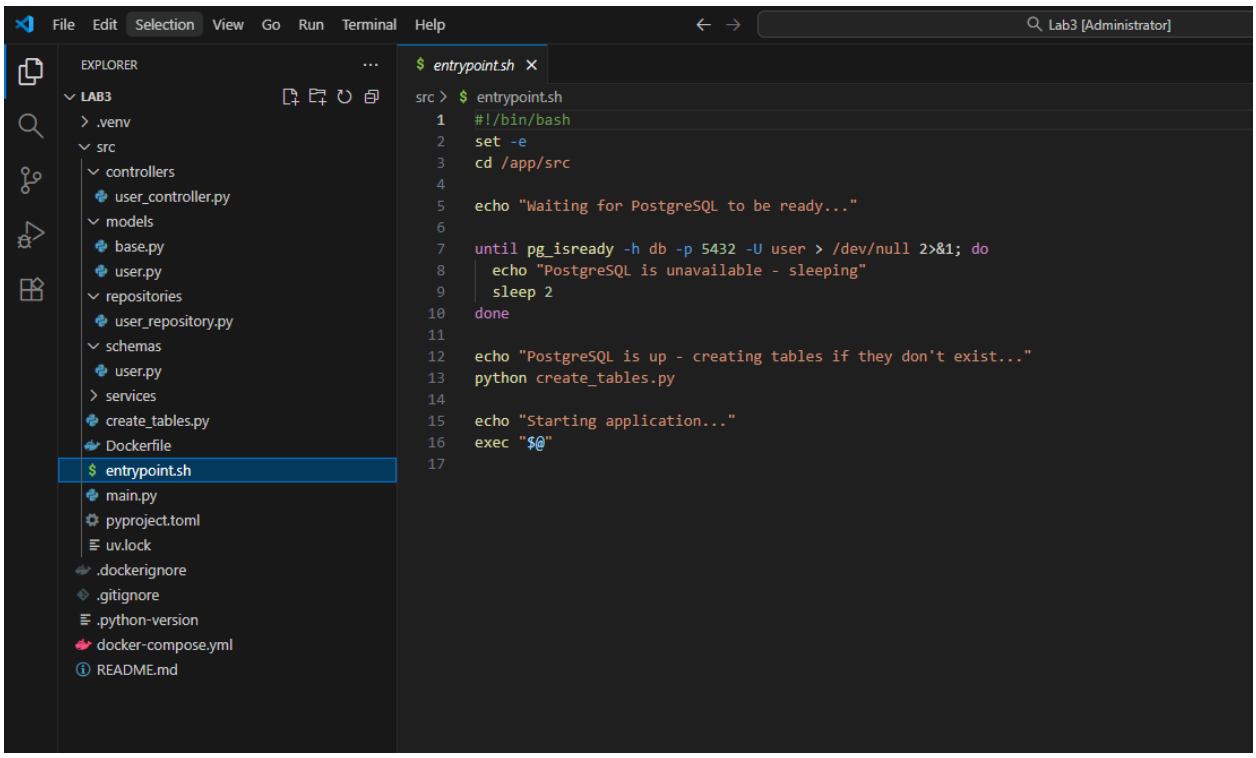
Часть 7: Скрипт создания таблиц

Файл: create_tables.py



- create_tables.py создаёт таблицы в базе данных при запуске;
- подключается к PostgreSQL с помощью create_async_engine, выполняет Base.metadata. create_all () для всех моделей и после успешного создания выводит сообщение "Tables created!".

Файл: entrypoint.sh



- python create_tables.py – создает таблицы во время запуска контейнера

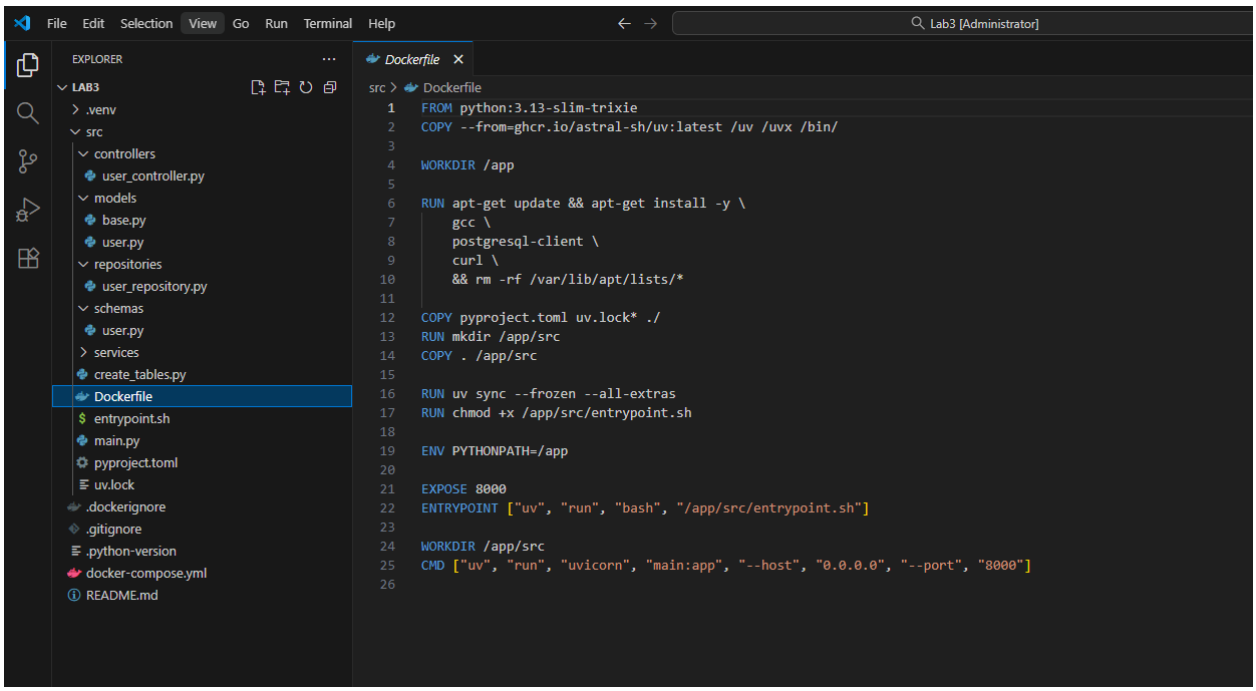
Часть 8. Сборка сервиса и запуск проекта.

Прим.

- ➢ База данных и PGAdmin переиспользованы из предыдущей лабораторной работы;
- ➢ Приложение представляет собой билд текущего сервиса, содержащий реализованный REST/API-слой, сервисы и слой доступа к данным (SQLAlchemy + DI).

Dockerfile

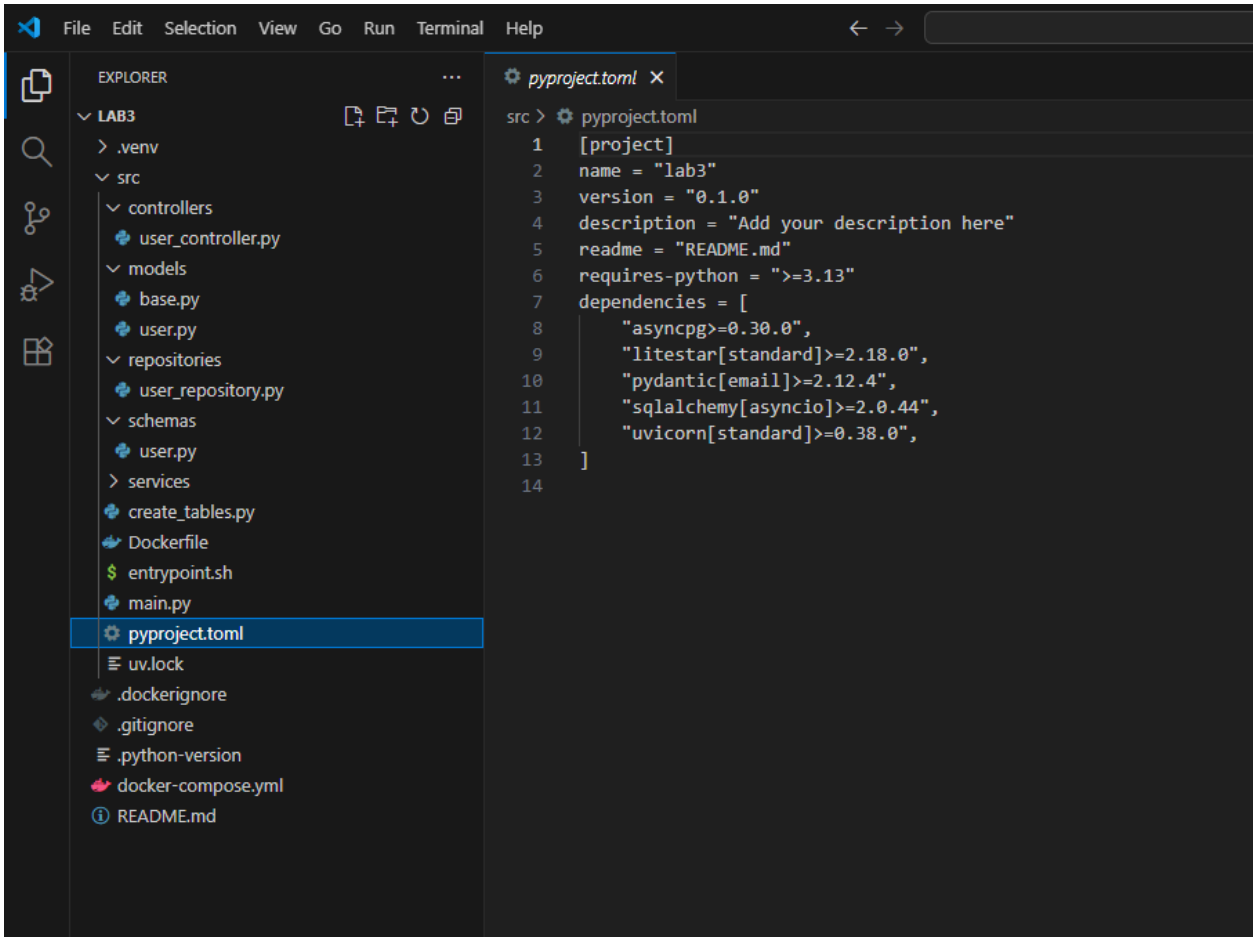
(используется для запуска проекта)



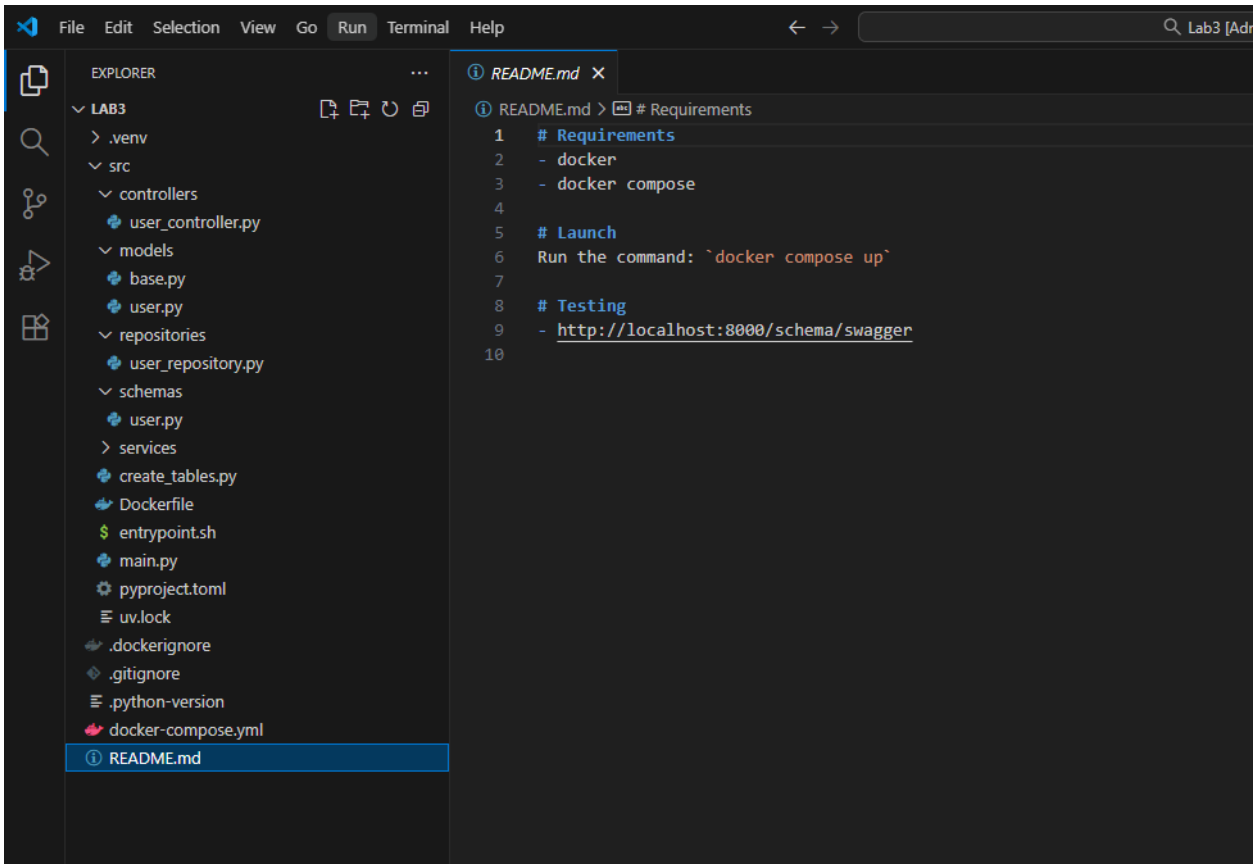
- Сервис app собирается через Dockerfile и DockerCompose

Конфигурация проекта

Файл: pyproject.toml



Инструкция для запуска проекта



- Для проверки работы сервиса использовалась Swagger-схема по адресу: http://localhost:8000/schema/swagger

*. Задание со звездочкой

(необходимо при запросе списка пользователей также возвращать количество пользователей в базе данных)

Реализовано через связку репозитория, сервиса и контроллера:

1. В UserRepository добавлен метод count(), который делает SQL-запрос: select(func.count()).select_from(User) и возвращает общее количество пользователей в базе (с учётом фильтров, если они есть).
2. В UserService метод get_by_filter() вызывает get_by_filter() из репозитория для получения списка пользователей и count() - чтобы узнать общее количество записей. Результат возвращается в виде словаря: {"total": total, "items": users}
3. В контроллере (UserController) метод get_all_users() принимает этот

результат и формирует ответ в виде схемы UserListResponse, где: total - общее количество пользователей, items - список пользователей текущей страницы.

Ответы на вопросы

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

Dependency Injection (DI) — паттерн проектирования, при котором зависимости объекта не создаются внутри него, а передаются извне.

- Без DI: если класс сам создаёт всё, что ему нужно, он становится громоздким и жёстко привязанным.
- С DI: мы выносим все что можно за пределы класса, а он просто использует.

Проблема, которую решает DI:
Без DI классы сами создают свои зависимости, что приводит к их высокой связанности друг с другом и затрудняет тестирование, модификацию и переиспользование кода.

Преимущества DI:

- Слабая связанность: компоненты не зависят от конкретных реализаций
- Тестируемость: легко подменить зависимости на моки/заглушки
- Переиспользование: компоненты могут быть использованы в разных контекстах
- Управление жизненным циклом: фреймворк контролирует создание и уничтожение объектов

2.1. Каковы основные обязанности каждого из трех слоев приложения (Repository, Service, Controller)?

Repository отделяет логику доступа к данным от бизнес-логики

- Отвечает за работу с базой данных;
- Выполняет CRUD-операции и сложные запросы;
- Не знает ничего про HTTP, API или валидацию данных;
- Наример: методы get_by_id(), create(), count() в UserRepository;

Service - слой бизнес-логики

- Получает данные из репозитория, выполняет дополнительные проверки, объединяет несколько операций.
- Подготавливает данные для контроллера в нужном формате.
- Пример: метод get_by_filter() в UserService, который возвращает и список пользователей, и общее количество записей.

Controller - «внешний» слой API.

- Обрабатывает HTTP-запросы и формирует ответы.
- Валидирует входные параметры (через Parameter и Pydantic-схемы), обрабатывает ошибки, сериализует данные в JSON.
- Не содержит бизнес-логики или прямой работы с БД.

2.2. Почему такое разделение считается хорошей практикой?

- Чистота кода: каждый слой отвечает за свою задачу, функции не перегружены.

- Тестируемость: можно тестировать сервис без HTTP, репозиторий без API, контроллер без БД (через mock).
- Поддерживаемость: изменения в одном слое не ломают другие. Например, можно поменять базу данных, не трогая контроллер.
- Переиспользуемость: сервисы и репозитории можно использовать в разных API или CLI без переписывания логики.

2.3. Что произойдет, если объединить логику репозитория и контроллера?

- Контроллер станет слишком «тяжёлым», он будет делать и работу с БД, и бизнес-логику, и обработку HTTP — нарушается принцип единственной ответственности.
- Тестирование усложнится: нельзя будет протестировать логику без поднятия сервера и базы данных.
- Код станет менее читаемым и поддерживаемым, изменения в структуре БД могут сломать API.
- Потенциально увеличивается риск ошибок: смешение уровней доступа к данным и обработки запросов усложняет выявление багов.

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

В Litestar зависимости реализованы через DI. Каждая зависимость может иметь определённый scope (область видимости):

- request – создаётся заново для каждого HTTP-запроса.
- app – создаётся один раз при старте приложения и живёт до его завершения.
- session – для WebSocket-сессий или пользовательских сессий (не всегда

используется для HTTP-запросов).

Для работы с базой данных обычно используется scope request, чтобы для каждого запроса создавался новый экземпляр Session.

Что происходит при обработке одного HTTP-запроса (на примере приложения) :

- запрос приходит в приложение.
- Litestar проверяет, какие зависимости нужны для обработчика.
- создаётся экземпляр зависимости get_db():
  - функция запускается до начала обработчика.
  - async with async_session_factory() as session: создаёт новый объект сессии SQLAlchemy.
- обработчик получает этот экземпляр сессии через аргумент функции.
- обработчик выполняет запросы к базе данных.
- экземпляр сессии уничтожается

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

Async/await - синтаксический сахар для работы с асинхронным кодом в Python.

- **async def** — объявление корутины (асинхронной функции)
- **await** — ожидание результата асинхронной операции

Подход позволяет одному процессу/потоку обслуживать много одновременных HTTP-запросов без блокировки на I/O (БД, сеть, файлы). Когда запрос к БД ждёт ответа, Python не блокирует поток, а переключается на обработку других запросов. Один процесс может обслуживать тысячи одновременных запросов, не создавая десятки потоков.

Асинхронность не делает один запрос к базе данных быстрее, но сильно влияет на масштабируемость и эффективность сервера, дает следующие преимущества:

- **Масштабируемость**: один процесс может обрабатывать тысячи одновременных запросов
- **Эффективность**: во время ожидания I/O (БД, сеть) процессор обрабатывает другие задачи
- **Современный стандарт**: Python 3.13 полностью поддерживает async/await

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 передаётся в конструктор репозитория первым аргументом, обеспечивая таким образом:

- Разделение ответственности: репозиторий отвечает только за SQL-запросы и работу с моделями. Он не создаёт и не управляет жизненным циклом сессии**,** чтобы код оставался чистым и тестируемым.
- Dependency Injection (DI): сессия создаётся Litestar на один HTTP-запрос и передаётся во все репозитории/сервисы, которые её используют.
- Гибкость и переиспользуемость: один и тот же репозиторий можно использовать с разными сессиями (например, для тестов, нескольких баз или транзакций).

Создавать сессию внутри репозитория нельзя, т.к. если бы репозиторий создавал сессию сам, каждый вызов метода создавал бы новую сессию, кроме того нельзя было бы объединить несколько операций в одну транзакцию, а контроллер/сервис потеряли бы контроль над временем жизни сессии и над commit/rollback.

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

В проекте пагинация (count и page) в методе get_by_filter нужна для нескольких целей:

1) Ограничение объёма данных за один запрос
- Если в базе много пользователей, возвращать их всех сразу неэффективно — это может сильно нагрузить БД и сеть.
- count задаёт, сколько записей вернуть на одной странице.

2) Навигация по списку пользователей (page позволяет «листать» результаты)

3) Сервис возвращает не только пользователей текущей страницы, но и общее количество записей (total), что позволяет фронтенду строить количество страниц и навигацию.

Таким образом пагинация делает запросы к базе эффективными, управляемыми и безопасными, а API — удобным для клиента.

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.

Пример бизнес-логики для UserService:

- Проверка уникальности email перед созданием пользователя.

- Хеширование пароля перед сохранением в базу.
- Отправка приветственного письма новому пользователю после регистрации.
- Ограничение создания пользователей по роли или квоте.

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

1. GET /users/{user_id}

    200 OK — пользователь найден, возвращается UserResponse.

    404 Not Found — пользователь с таким ID отсутствует (NotFoundException).

    500 Internal Server Error —ошибка на сервере.

2. GET /users (список с пагинацией)

    200 OK — успешно возвращён список пользователей вместе с общим количеством (UserListResponse).

    400 Bad Request — если переданы некорректные параметры

    500 Internal Server Error

3. POST /users (создание пользователя)

    201 Created — пользователь успешно создан, возвращается UserResponse.

    409 Conflict — дубликат email или username

    400 Bad Request — некорректные данные в теле запроса (валидация Pydantic).

    500 Internal Server Error

4. PUT /users/{user_id} (обновление пользователя)

200 OK — пользователь успешно обновлён, возвращается UserResponse.

404 Not Found — пользователь с таким ID отсутствует.

400 Bad Request — некорректные данные для обновления.

409 Conflict — попытка обновить на уже существующий email/username.

500 Internal Server Error — ошибка сервера/БД.

5. DELETE /users/{user_id}

204 No Content — пользователь успешно удалён (ответ без тела).

404 Not Found — пользователь с таким ID отсутствует.

500 Internal Server Error — ошибка на сервере.

Обоснование:

- 200 OK / 201 Created / 204 No Content — стандартные успешные статусы для соответствующих операций (чтение, создание, удаление).
- 404 Not Found — корректно сигнализирует, что ресурс отсутствует.
- 400 Bad Request — ошибки клиента (валидация параметров или тела запроса).
- 409 Conflict — бизнес-правило уникальности (email/username).
- 500 Internal Server Error — любые неожиданные сбои, которые ловит декоратор

Вывод:

В ходе выполнения лабораторной работы была успешно реализована трехслойная архитектура веб-приложения с использованием современных практик разработки на Python.

1) Repository Layer - инкапсулирует работу с БД

2) Service Layer - содержит бизнес-логику
3) Controller Layer - обрабатывает HTTP-запросы

Изучен и применен паттерн Dependency Injection в контексте Litestar, что обеспечило слабую связанность компонентов и упростило тестирование.

Настроена асинхронная работа с PostgreSQL через SQLAlchemy 2.0 и asyncpg, что позволяет эффективно обрабатывать множество одновременных запросов.

Реализована полноценная CRUD-функциональность с поддержкой:

- Создания, чтения, обновления и удаления пользователей
- Пагинации с гибкой настройкой размера страницы
- Подсчета общего количества записей (задание со звездочкой)
- Валидации данных через Pydantic
- Обработки ошибок с корректными HTTP-статусами

Полученные навыки работы с Dependency Injection, ORM и асинхронным программированием являются фундаментальными для разработки современных масштабируемых веб-приложений на Python.