

Лабораторная работа 3

«Внедрение Dependency Injection и SQLAlchemy в Litestar»

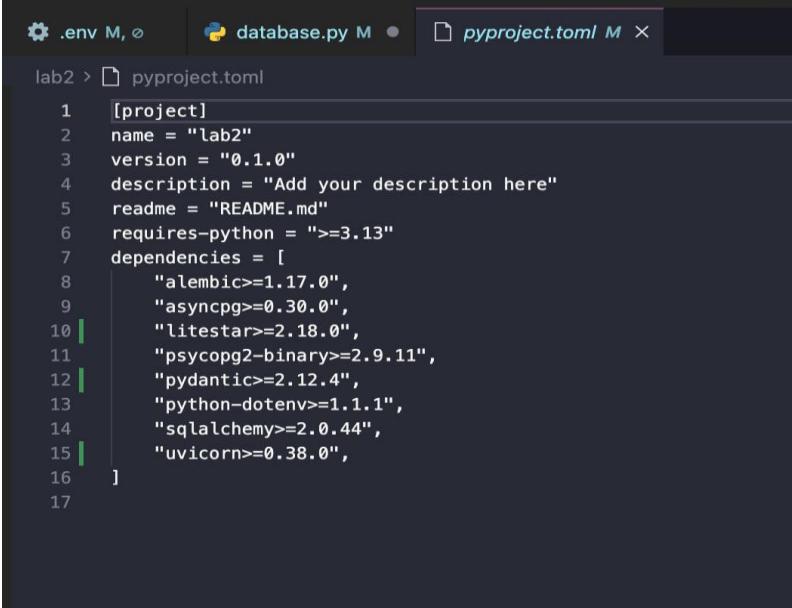
Жунев Андрей РИМ-15090

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:

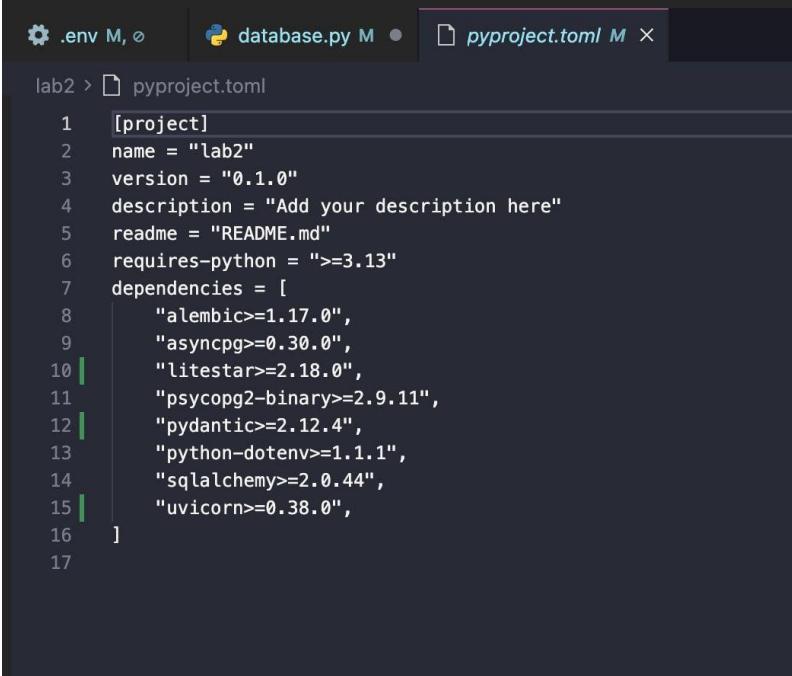
- **litestar**
- **pydantic**
- **uvicorn**



```
lab2 > pyproject.toml
1 [project]
2   name = "lab2"
3   version = "0.1.0"
4   description = "Add your description here"
5   readme = "README.md"
6   requires-python = ">=3.13"
7   dependencies = [
8     "alembic>=1.17.0",
9     "asyncpg>=0.30.0",
10    "litestar>=2.18.0",
11    "psycopg2-binary>=2.9.11",
12    "pydantic>=2.12.4",
13    "python-dotenv>=1.1.1",
14    "sqlalchemy>=2.0.44",
15    "uvicorn>=0.38.0",
16  ]
17
```

1. Настройка окружения

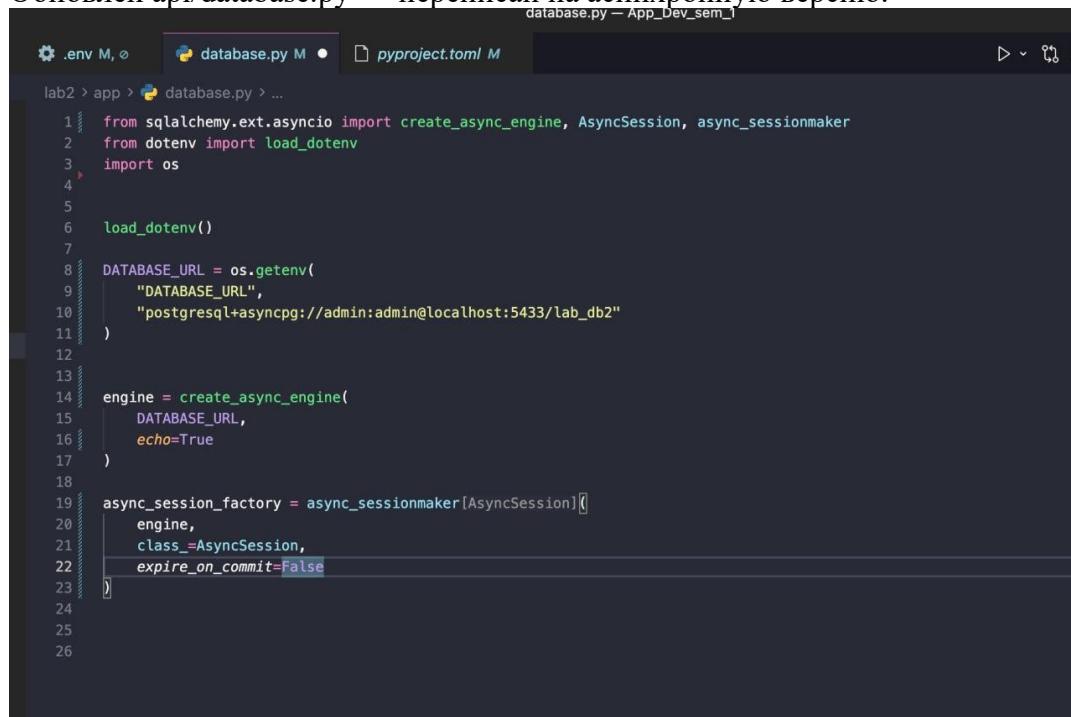
Был отредактирован .env файл — изменен DATABASE_URL для доступа к БД асинхронным подходом.



```
lab2 > pyproject.toml
1 [project]
2   name = "lab2"
3   version = "0.1.0"
4   description = "Add your description here"
5   readme = "README.md"
6   requires-python = ">=3.13"
7   dependencies = [
8     "alembic>=1.17.0",
9     "asyncpg>=0.30.0",
10    "litestar>=2.18.0",
11    "psycopg2-binary>=2.9.11",
12    "pydantic>=2.12.4",
13    "python-dotenv>=1.1.1",
14    "sqlalchemy>=2.0.44",
15    "uvicorn>=0.38.0",
16  ]
17
```

2. Обновление базы данных

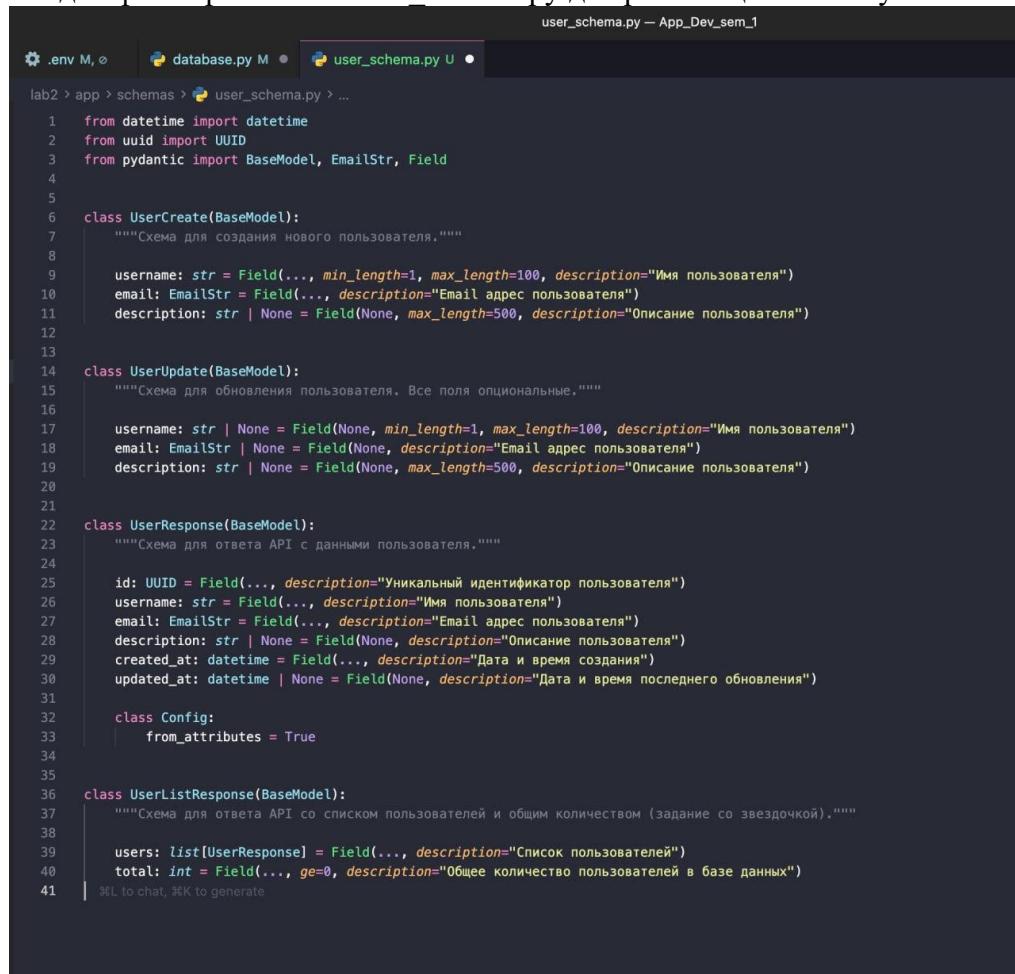
Обновлен `api/database.py` — переписан на асинхронную версию.



```
1> from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession, async_sessionmaker
2  from dotenv import load_dotenv
3  import os
4
5  load_dotenv()
6
7  DATABASE_URL = os.getenv(
8      "DATABASE_URL",
9      "postgresql+asyncpg://admin:admin@localhost:5433/lab_db2"
10 )
11
12
13  engine = create_async_engine(
14      DATABASE_URL,
15      echo=True
16  )
17
18  async_session_factory = async_sessionmaker[AsyncSession](
19      engine,
20      class_=AsyncSession,
21      expire_on_commit=False
22  )
23
24
25
26
```

3. Создание схем Pydantic

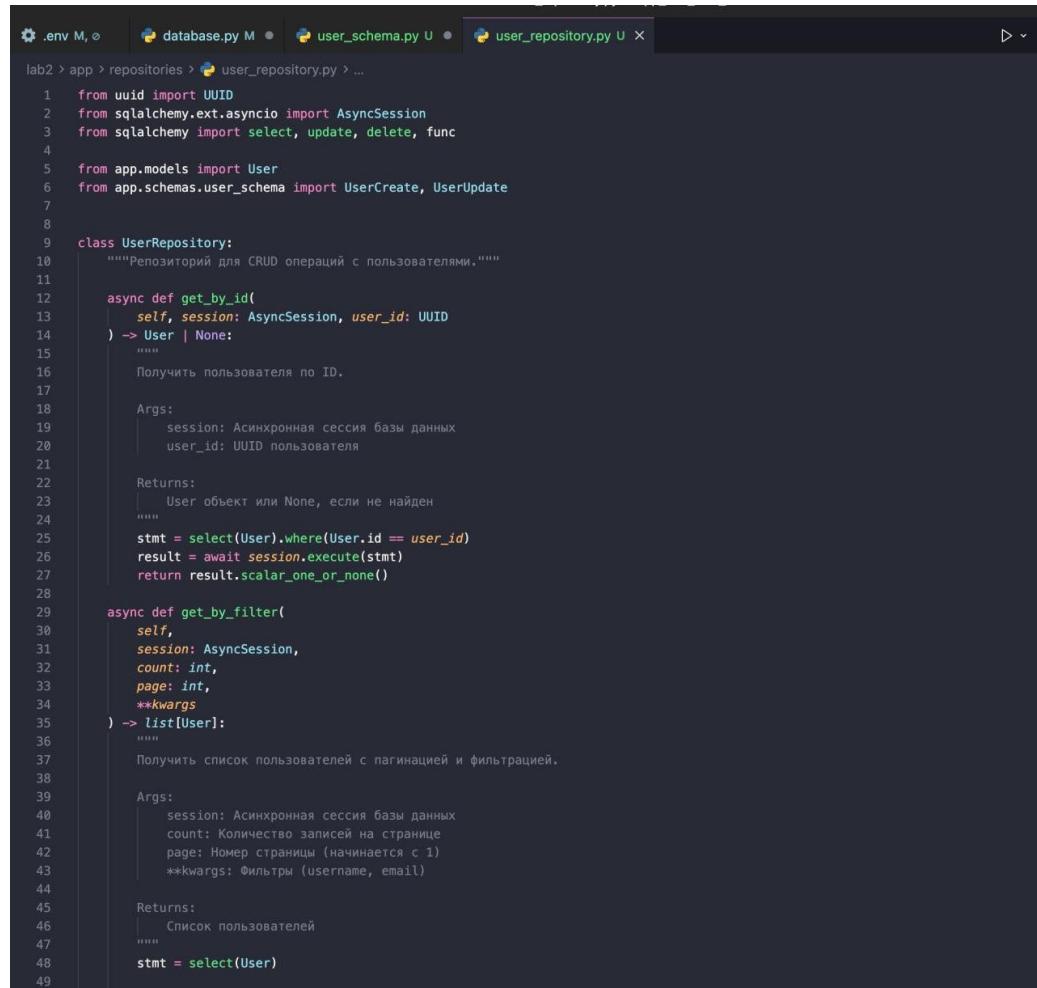
Создан файл `api/schemas/user_schema.py` для реализации схем Pydantic.



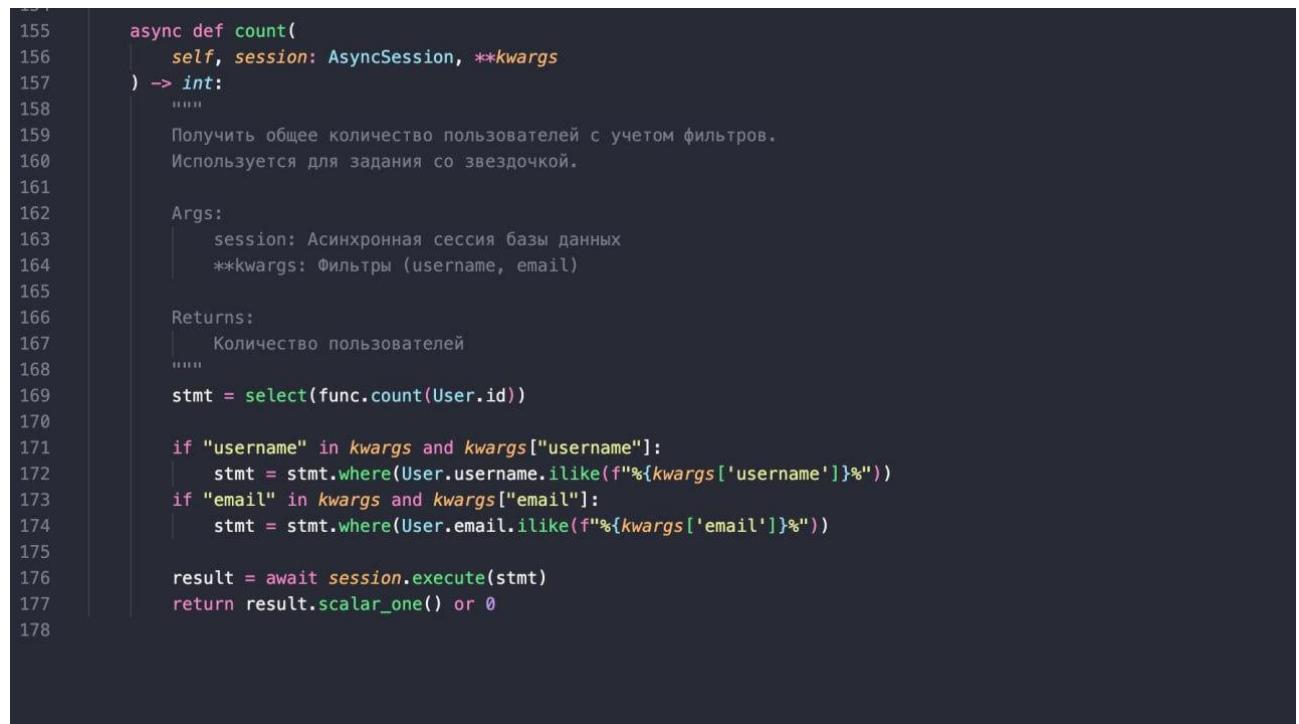
```
1  from datetime import datetime
2  from uuid import UUID
3  from pydantic import BaseModel, EmailStr, Field
4
5
6  class UserCreate(BaseModel):
7      """Схема для создания нового пользователя."""
8
9      username: str = Field(..., min_length=1, max_length=100, description="Имя пользователя")
10     email: EmailStr = Field(..., description="Email адрес пользователя")
11     description: str | None = Field(None, max_length=500, description="Описание пользователя")
12
13
14  class UserUpdate(BaseModel):
15      """Схема для обновления пользователя. Все поля опциональные."""
16
17      username: str | None = Field(None, min_length=1, max_length=100, description="Имя пользователя")
18      email: EmailStr | None = Field(None, description="Email адрес пользователя")
19      description: str | None = Field(None, max_length=500, description="Описание пользователя")
20
21
22  class UserResponse(BaseModel):
23      """Схема для ответа API с данными пользователя."""
24
25      id: UUID = Field(..., description="Уникальный идентификатор пользователя")
26      username: str = Field(..., description="Имя пользователя")
27      email: EmailStr = Field(..., description="Email адрес пользователя")
28      description: str | None = Field(None, description="Описание пользователя")
29      created_at: datetime = Field(..., description="Дата и время создания")
30      updated_at: datetime | None = Field(None, description="Дата и время последнего обновления")
31
32
33  class Config:
34      from_attributes = True
35
36
37  class UserListResponse(BaseModel):
38      """Схема для ответа API со списком пользователей и общим количеством (задание со звездочкой)."""
39
40      users: list[UserResponse] = Field(..., description="Список пользователей")
41      total: int = Field(..., ge=0, description="Общее количество пользователей в базе данных")
```

4. Реализация репозитория

Реализован репозиторий для взаимодействия с базой данных.



```
lab2 > app > repositories > 🐍 user_repository.py > ...
1  from uuid import UUID
2  from sqlalchemy.ext.asyncio import AsyncSession
3  from sqlalchemy import select, update, delete, func
4
5  from app.models import User
6  from app.schemas.user_schema import UserCreate, UserUpdate
7
8
9  class UserRepository:
10     """Репозиторий для CRUD операций с пользователями."""
11
12     async def get_by_id(
13         self,
14         session: AsyncSession,
15         user_id: UUID
16     ) -> User | None:
17         """
18             Получить пользователя по ID.
19
20             Args:
21                 session: Асинхронная сессия базы данных
22                 user_id: UUID пользователя
23
24             Returns:
25                 User объект или None, если не найден
26         """
27         stmt = select(User).where(User.id == user_id)
28         result = await session.execute(stmt)
29         return result.scalar_one_or_none()
30
31     async def get_by_filter(
32         self,
33         session: AsyncSession,
34         count: int,
35         page: int,
36         **kwargs
37     ) -> list[User]:
38         """
39             Получить список пользователей с логикой пагинации и фильтрацией.
40
41             Args:
42                 session: Асинхронная сессия базы данных
43                 count: Количество записей на странице
44                 page: Номер страницы (начинается с 1)
45                 **kwargs: Фильтры (username, email)
46
47             Returns:
48                 Список пользователей
49         """
50         stmt = select(User)
```



```
155     async def count(
156         self,
157         session: AsyncSession,
158         **kwargs
159     ) -> int:
160         """
161             Получить общее количество пользователей с учетом фильтров.
162             Используется для задания со звездочкой.
163
164             Args:
165                 session: Асинхронная сессия базы данных
166                 **kwargs: Фильтры (username, email)
167
168             Returns:
169                 Количество пользователей
170         """
171         stmt = select(func.count(User.id))
172
173         if "username" in kwargs and kwargs["username"]:
174             stmt = stmt.where(User.username.ilike(f"%{kwargs['username']}%"))
175
176         if "email" in kwargs and kwargs["email"]:
177             stmt = stmt.where(User.email.ilike(f"%{kwargs['email']}%"))
178
179         result = await session.execute(stmt)
180         return result.scalar_one() or 0
```

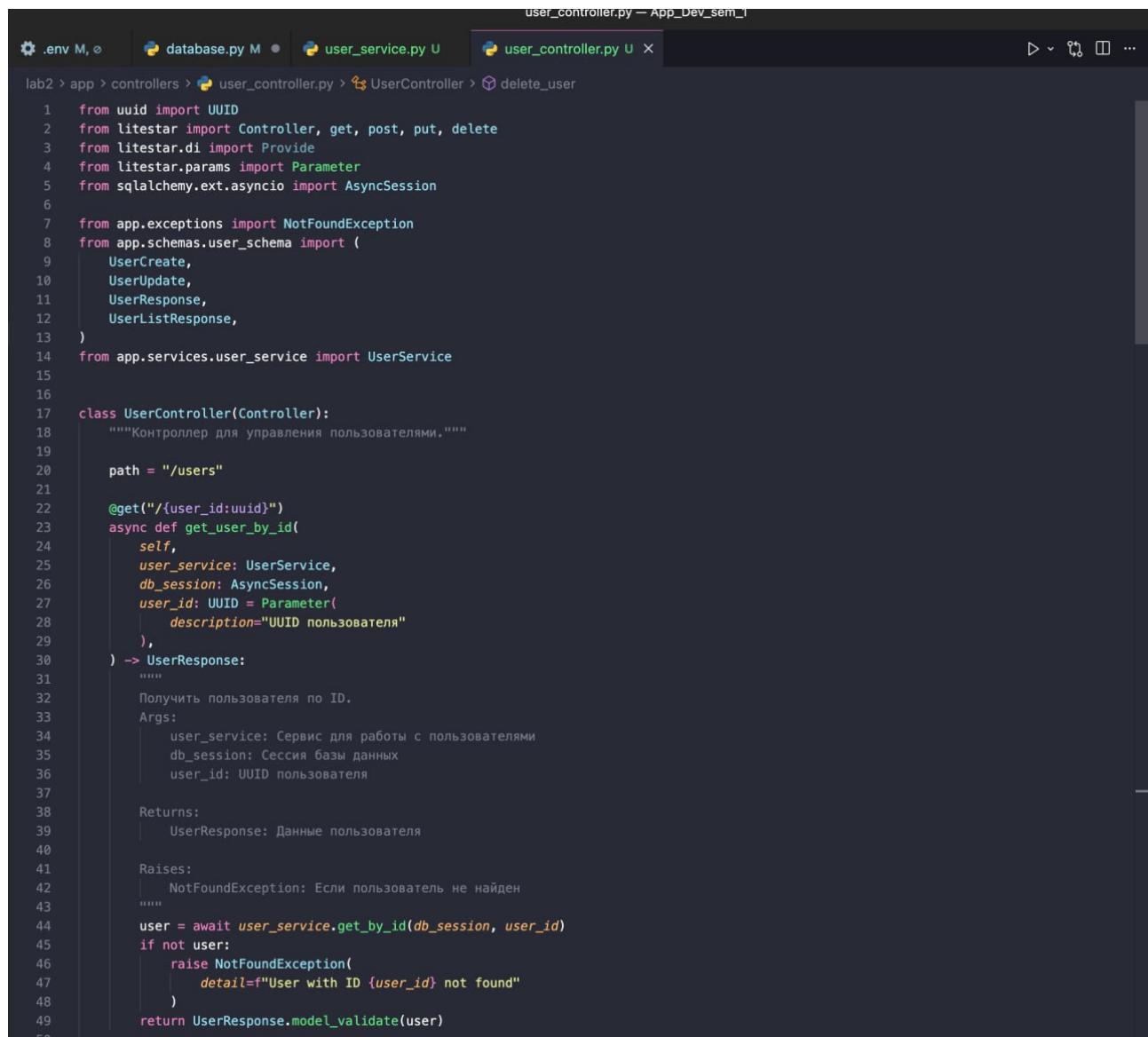
5. Сервисный слой

Реализован сервисный слой для управления бизнес-логикой.

```
lab2 > app > services > user_service.py > UserService > _check_username_exists
1  from uuid import UUID
2  from sqlalchemy.ext.asyncio import AsyncSession
3  from sqlalchemy import select
4
5  from app.models import User
6  from app.repositories.user_repository import UserRepository
7  from app.schemas.user_schema import UserCreate, UserUpdate
8
9
10 class UserService:
11     """Сервис для бизнес-логики работы с пользователями."""
12
13     def __init__(self, user_repository: UserRepository):
14         """
15             Инициализация сервиса.
16             Args:
17                 user_repository: Репозиторий для работы с пользователями (Dependency Injection)
18             """
19
20         self.user_repository = user_repository
21
22     async def get_by_id(
23         self, session: AsyncSession, user_id: UUID
24     ) -> User | None:
25         """
26             Получить пользователя по ID.
27             Args:
28                 session: Асинхронная сессия базы данных
29                 user_id: UUID пользователя
30
31             Returns:
32                 User объект или None, если не найден
33             """
34
35         return await self.user_repository.get_by_id(session, user_id)
36
37     async def get_by_filter(
38         self,
39         session: AsyncSession,
40         count: int,
41         page: int,
42         **kwargs
43     ) -> list[User]:
44         """
45             Получить список пользователей с пагинацией и фильтрацией.
46             Args:
47                 session: Асинхронная сессия базы данных
48                 count: Количество записей на странице
49                 page: Номер страницы (начинается с 1)
50                 **kwargs: Фильтры (username, email)
51
52             Returns:
53                 Список пользователей
```

6. Контроллер

Реализован контроллер для обработки HTTP-запросов.

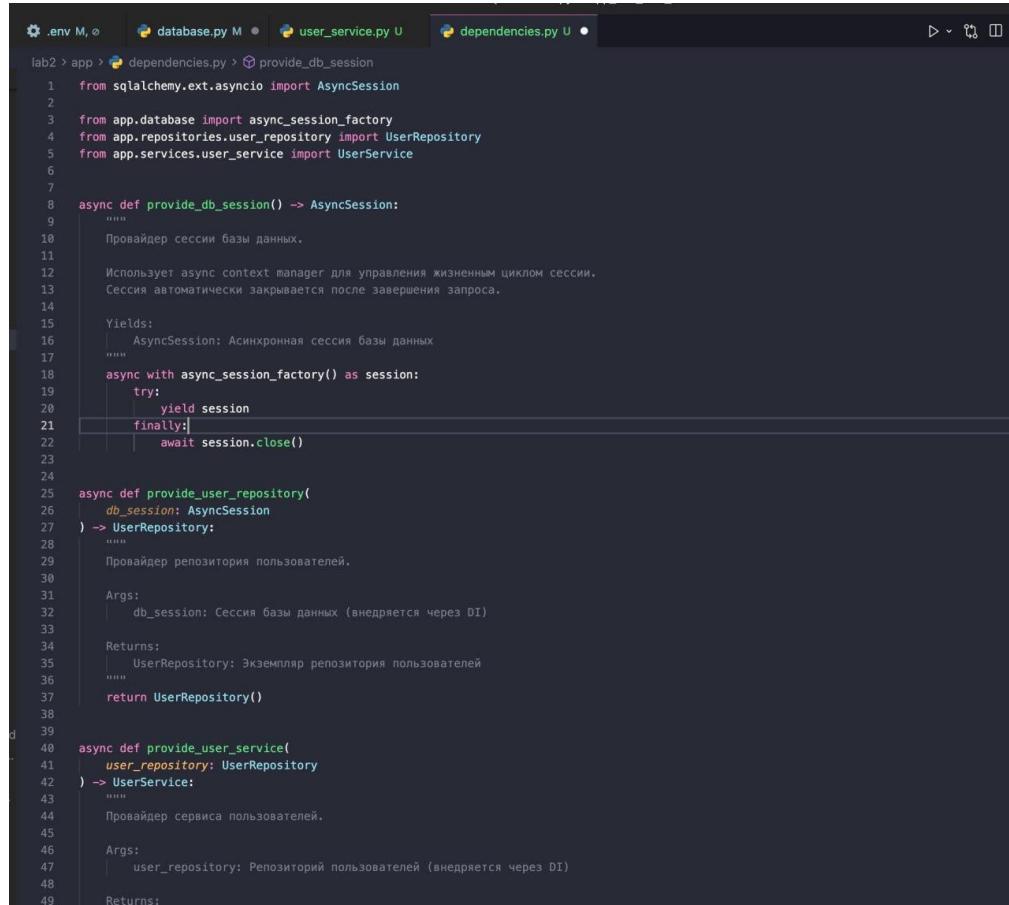


The screenshot shows a code editor window with a dark theme. The tab bar at the top has several tabs: '.env M', 'database.py M', 'user_service.py U', and 'user_controller.py U' (which is the active tab). The code itself is a Python file named 'user_controller.py'. It imports various modules from the 'app' package and defines a class 'UserController' that inherits from 'Controller'. The class has a docstring explaining it's a controller for managing users. It includes a method '@get("/{user_id:uuid}")' named 'get_user_by_id' which takes a user ID as a parameter and returns a UserResponse object. The method uses dependency injection for 'user_service' and 'db_session'. It also handles the case where no user is found by raising a 'NotFoundException'. The code is annotated with comments and docstrings throughout.

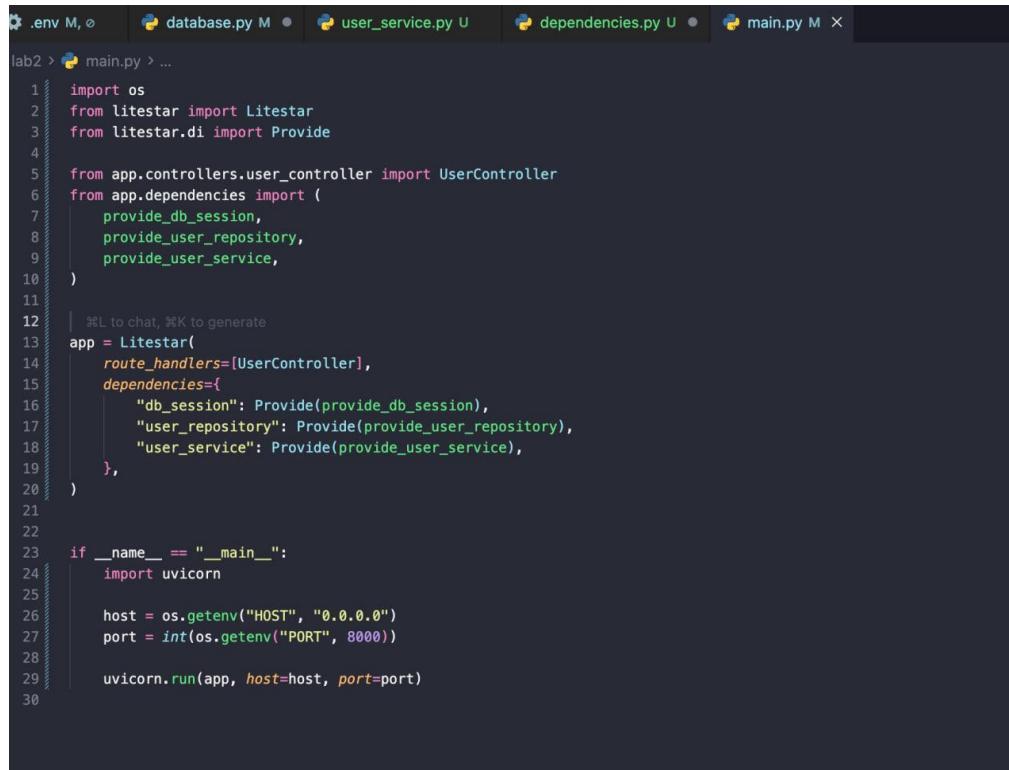
```
1  from uuid import UUID
2  from litestar import Controller, get, post, put, delete
3  from litestar.di import Provide
4  from litestar.params import Parameter
5  from sqlalchemy.ext.asyncio import AsyncSession
6
7  from app.exceptions import NotFoundException
8  from app.schemas.user_schema import (
9      UserCreate,
10     UserUpdate,
11     UserResponse,
12     UserListResponse,
13 )
14 from app.services.user_service import UserService
15
16
17 class UserController(Controller):
18     """Контроллер для управления пользователями."""
19
20     path = "/users"
21
22     @get("/{user_id:uuid}")
23     async def get_user_by_id(
24         self,
25         user_service: UserService,
26         db_session: AsyncSession,
27         user_id: UUID = Parameter(
28             description="UUID пользователя"
29         ),
30     ) -> UserResponse:
31         """
32             Получить пользователя по ID.
33             Args:
34                 user_service: Сервис для работы с пользователями
35                 db_session: Сессия базы данных
36                 user_id: UUID пользователя
37
38             Returns:
39                 UserResponse: Данные пользователя
40
41             Raises:
42                 NotFoundException: Если пользователь не найден
43
44             user = await user_service.get_by_id(db_session, user_id)
45             if not user:
46                 raise NotFoundException(
47                     detail=f"User with ID {user_id} not found"
48                 )
49             return UserResponse.model_validate(user)
```

7. Настройка DI и главной точки входа

Настроены главная точка входа в приложение и Dependency Injection.



```
lab2 > app > dependencies.py > provide_db_session
1  from sqlalchemy.ext.asyncio import AsyncSession
2
3  from app.database import async_session_factory
4  from app.repositories.user_repository import UserRepository
5  from app.services.user_service import UserService
6
7
8  async def provide_db_session() -> AsyncSession:
9      """
10         Провайдер сессии базы данных.
11
12         Использует async context manager для управления жизненным циклом сессии.
13         Сессия автоматически закрывается после завершения запроса.
14
15         Yields:
16             |     AsyncSession: Асинхронная сессия базы данных
17             |
18         async with async_session_factory() as session:
19             try:
20                 yield session
21             finally:
22                 await session.close()
23
24
25  async def provide_user_repository(
26      db_session: AsyncSession
27  ) -> UserRepository:
28      """
29      Провайдер репозитория пользователей.
30
31      Args:
32          |     db_session: Сессия базы данных (внедряется через DI)
33
34      Returns:
35          |     UserRepository: Экземпляр репозитория пользователей
36          |
37
38      return UserRepository()
39
40
41  async def provide_user_service(
42      user_repository: UserRepository
43  ) -> UserService:
44      """
45      Провайдер сервиса пользователей.
46
47      Args:
48          |     user_repository: Репозиторий пользователей (внедряется через DI)
49
50      Returns:
```



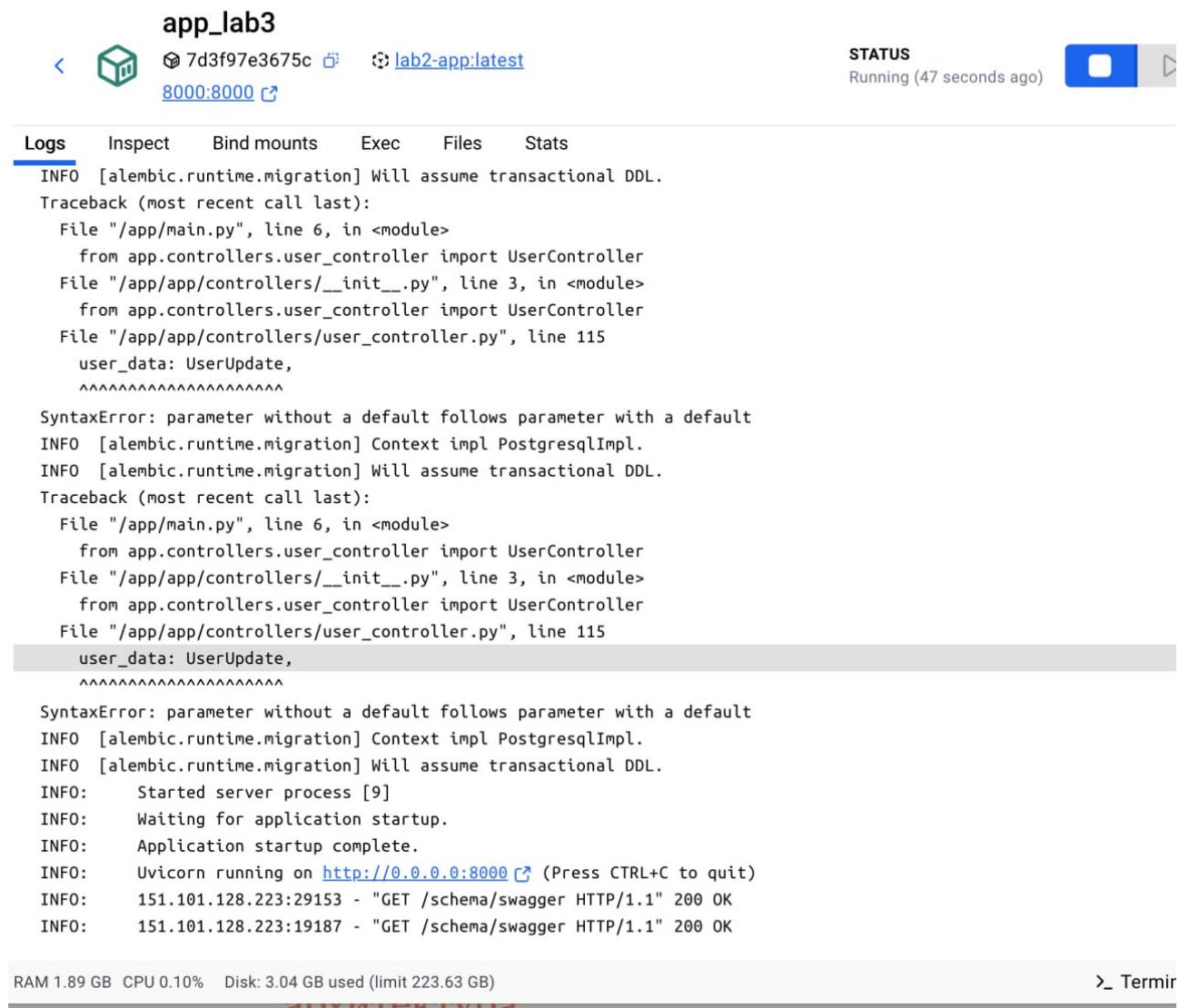
```
lab2 > main.py > ...
1  import os
2  from litestar import Litestar
3  from litestar.di import Provide
4
5  from app.controllers.user_controller import UserController
6  from app.dependencies import (
7      provide_db_session,
8      provide_user_repository,
9      provide_user_service,
10 )
11
12  app = Litestar(
13      route_handlers=[UserController],
14      dependencies={
15          "db_session": Provide(provide_db_session),
16          "user_repository": Provide(provide_user_repository),
17          "user_service": Provide(provide_user_service),
18      },
19  )
20
21
22
23  if __name__ == "__main__":
24      import uvicorn
25
26      host = os.getenv("HOST", "0.0.0.0")
27      port = int(os.getenv("PORT", 8000))
28
29      uvicorn.run(app, host=host, port=port)
30
```

8. Документация

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

9. Docker Compose

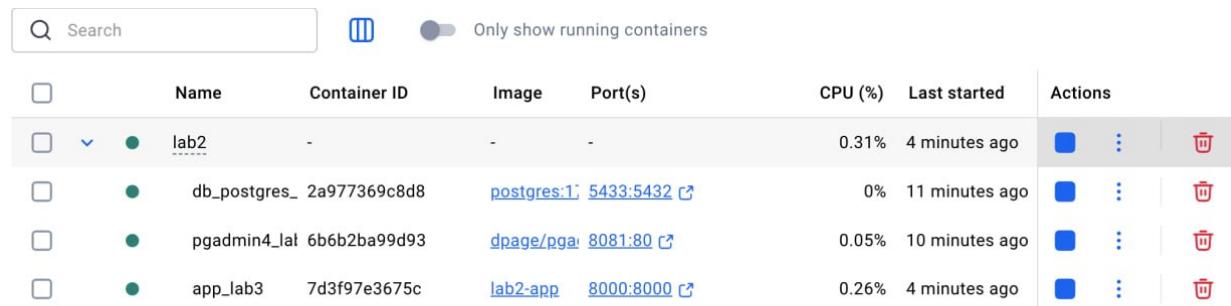
Для удобства запуска приложение было добавлено в docker-compose. Все контейнеры успешно запускаются.



The screenshot shows the Docker Compose logs for the 'app_lab3' service. The logs output several error messages related to Alembic migrations and syntax errors in the application code. The application starts a server and handles Swagger API requests. The logs are timestamped and show the progress of the application's startup and initial requests.

```
INFO [alembic.runtime.migration] Will assume transactional DDL.  
Traceback (most recent call last):  
  File "/app/main.py", line 6, in <module>  
    from app.controllers.user_controller import UserController  
File "/app/app/controllers/__init__.py", line 3, in <module>  
    from app.controllers.user_controller import UserController  
File "/app/app/controllers/user_controller.py", line 115  
    user_data: UserUpdate,  
    ^^^^^^^^^^^^^^^^^^  
  
SyntaxError: parameter without a default follows parameter with a default  
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.  
INFO [alembic.runtime.migration] Will assume transactional DDL.  
Traceback (most recent call last):  
  File "/app/main.py", line 6, in <module>  
    from app.controllers.user_controller import UserController  
File "/app/app/controllers/__init__.py", line 3, in <module>  
    from app.controllers.user_controller import UserController  
File "/app/app/controllers/user_controller.py", line 115  
    user_data: UserUpdate,  
    ^^^^^^^^^^^^^^  
  
SyntaxError: parameter without a default follows parameter with a default  
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.  
INFO [alembic.runtime.migration] Will assume transactional DDL.  
INFO: Started server process [9]  
INFO: Waiting for application startup.  
INFO: Application startup complete.  
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)  
INFO: 151.101.128.223:29153 - "GET /schema/swagger HTTP/1.1" 200 OK  
INFO: 151.101.128.223:19187 - "GET /schema/swagger HTTP/1.1" 200 OK
```

RAM 1.89 GB CPU 0.10% Disk: 3.04 GB used (limit 223.63 GB) [>_ Termir](#)



The screenshot shows the list of running Docker containers. There are four containers listed: 'lab2', 'db_postgres', 'pgadmin4_lal', and 'app_lab3'. Each container has its name, container ID, image, port mapping, CPU usage, and last start time. Actions for each container are shown in the 'Actions' column.

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	lab2	-	-	-	0.31%	4 minutes ago	[] [:] [x]
<input type="checkbox"/>	db_postgres	2a977369c8d8	postgres:1	5433:5432	0%	11 minutes ago	[] [:] [x]
<input type="checkbox"/>	pgadmin4_lal	6b6b2ba99d93	dpage/pgadmin4	8081:80	0.05%	10 minutes ago	[] [:] [x]
<input type="checkbox"/>	app_lab3	7d3f97e3675c	lab2-app	8000:8000	0.26%	4 minutes ago	[] [:] [x]

Тестирование API

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

The screenshot shows a Chrome browser window with the title "User Management API 0.0.0 OAS 3.1". The address bar indicates the URL is "localhost:8000/schema/swagger#/". The main content area is titled "default" and lists several API endpoints:

- GET /users** GetAllUsers
- POST /users** CreateUser
- GET /users/{user_id}** GetUserByld
- PUT /users/{user_id}** UpdateUser
- DELETE /users/{user_id}** DeleteUser

Below the endpoints, there is a section titled "Schemas" which shows the definitions for the UserCreate, UserListResponse, and UserResponse objects.

```
schema:object
  properties:
    id:integer
    username:string
```

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 потребует 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 асинхронность 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 простояивает. 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. Её 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