Министерство науки и высшего образования Российской Федерации
ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»
Институт радиоэлектроники и информационных технологий – РТФ

Основы работы с Docker и PostgreSQL

Лабораторное задание №3

Группа: РИМ-150950
Студент: Владимиров Н.А.

Преподаватель: Кузьмин Д. И.

Екатеринбург
2025

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

Задачи:

- Написать слой взаимодействия с базой данных для создания, обновления, получения и удаления пользователей.
- Создать сервисный слой, который занимается бизнес логикой.
- Создать контроллер для управления пользователем.
- Собирать приложение и добавить зависимости в litestar.

Ход работы:

1. Репозиторий

```python
class UserRepository:

    async def get_by_id(self, session: AsyncSession, user_id: UUID) -> User | None:
        stmt = select(User).where(User.id == user_id)
        result = await session.execute(stmt)
        return result.scalar_one_or_none()
```

```python
async def get_by_filter(
    self, session: AsyncSession, count: int, page: int, **kwargs
) -> List[User]:
    stmt = select(User)
```

```python
async def create(self, session: AsyncSession, user_data: UserCreate) -> User:
    user = User(
        username=user_data.username,
        email=user_data.email,
        description=user_data.description
    )
    session.add(user)
    await session.flush()
    await session.refresh(user)
    return user
```

```python
async def update(
    self, session: AsyncSession, user_id: UUID, user_data: UserUpdate
) -> User:
    user = await self.get_by_id(session, user_id)
    if not user:
        raise ValueError(f"User with ID {user_id} not found")

    update_data = user_data.model_dump(exclude_unset=True)
    if update_data:
        for key, value in update_data.items():
            setattr(user, key, value)
        await session.flush()
        await session.refresh(user)

    return user
```

```python
async def delete(self, session: AsyncSession, user_id: UUID) -> None:
    user = await self.get_by_id(session, user_id)
    if not user:
        raise ValueError(f"User with ID {user_id} not found")
    await session.delete(user)
    await session.flush()
```

2. Сервисный слой

```python
class UserService:
    def __init__(self, user_repository: UserRepository, db_session: AsyncSession):
        self.user_repository = user_repository
        self.db_session = db_session

    async def get_by_id(self, user_id: UUID) -> User | None:
        return await self.user_repository.get_by_id(self.db_session, user_id)

    async def get_by_filter(
        self, count: int, page: int, **kwargs
    ) -> List[User]:
        return await self.user_repository.get_by_filter(
            self.db_session, count, page, **kwargs
        )
```

```python
async def create(self, user_data: UserCreate) -> User:
    existing_user = await self.user_repository.get_by_filter(
        self.db_session, count=1, page=1, email=user_data.email
    )
    if existing_user:
        raise ValueError(f"User with email {user_data.email} already exists")

    existing_username = await self.user_repository.get_by_filter(
        self.db_session, count=1, page=1, username=user_data.username
    )
    if existing_username:
        raise ValueError(f"User with username {user_data.username} already exists")

    user = await self.user_repository.create(self.db_session, user_data)
    await self.db_session.commit()
    return user

async def update(self, user_id: UUID, user_data: UserUpdate) -> User:
    if user_data.email:
        existing_user = await self.user_repository.get_by_filter(
            self.db_session, count=1, page=1, email=user_data.email
        )
        if existing_user and existing_user[0].id != user_id:
            raise ValueError(f"User with email {user_data.email} already exists")

    if user_data.username:
        existing_username = await self.user_repository.get_by_filter(
            self.db_session, count=1, page=1, username=user_data.username
        )
        if existing_username and existing_username[0].id != user_id:
            raise ValueError(f"User with username {user_data.username} already exists")

    user = await self.user_repository.update(self.db_session, user_id, user_data)
    await self.db_session.commit()
    return user

async def delete(self, user_id: UUID) -> None:
    await self.user_repository.delete(self.db_session, user_id)
    await self.db_session.commit()
```

3. Контроллер

```python
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 = Parameter(description="ID пользователя"),
    ) -> UserResponse:
        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(default=10, ge=1, le=100, description="Количество записей на странице"),
        page: int = Parameter(default=1, ge=1, description="Номер страницы"),
        username: str | None = Parameter(default=None, description="Фильтр по username"),
        email: str | None = Parameter(default=None, description="Фильтр по email"),
    ) -> dict:
        filters = {}
        if username:
            filters["username"] = username
        if email:
            filters["email"] = email

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

        return {
            "users": [UserResponse.model_validate(user) for user in users],
            "total": total_count,
            "page": page,
            "count": count
        }
```

```python
@post()
async def create_user(
    self,
    user_service: UserService,
    user_data: UserCreate,
) -> UserResponse:
    try:
        user = await user_service.create(user_data)
        return UserResponse.model_validate(user)
    except ValueError as e:
        raise ClientException(status_code=400, detail=str(e))

@put("/{user_id:uuid}")
async def update_user(
    self,
    user_service: UserService,
    user_id: UUID,
    user_data: UserUpdate,
) -> UserResponse:
    try:
        user = await user_service.update(user_id, user_data)
        return UserResponse.model_validate(user)
    except ValueError as e:
        if "not found" in str(e).low (function) detail: Any
            raise NotFoundException(detail=str(e))
        raise ClientException(status_code=400, detail=str(e))

@delete("/{user_id:uuid}")
async def delete_user(
    self,
    user_service: UserService,
    user_id: UUID,
) -> None:
    try:
        await user_service.delete(user_id)
    except ValueError as e:
        raise NotFoundException(detail=str(e))
```

4. Главное приложение

```python
DATABASE_URL = os.getenv(
    "DATABASE_URL",
    "postgresql+asyncpg://user:password@localhost:5432/dbname"
)

engine = create_async_engine(DATABASE_URL, echo=True)
async_session_factory = async_sessionmaker[AsyncSession](
    engine, class_=AsyncSession, expire_on_commit=False
)


async def provide_db_session() -> AsyncSession:
    session = async_session_factory()
    try:
        yield session
    finally:
        await session.close()


async def provide_user_repository(db_session: AsyncSession) -> UserRepository:
    return UserRepository()


async def provide_user_service(
    user_repository: UserRepository,
    db_session: AsyncSession
) -> UserService:
    return UserService(user_repository, db_session)


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)
```

Вывод:

В ходе выполнения данной лабораторной работы я освоил принципы Dependency Injection и интеграцию SQLAlchemy ORM в веб-приложении на базе фреймворка Litestar написав слой взаимодействия с базой данных для создания, обновления, получения и удаления пользователей, создал сервисный слой, который занимается бизнес логикой, создал контроллер для управления пользователем и собирал приложение, и добавил зависимости в litestar.

Ответы на вопросы:

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

Это паттерн, при котором зависимости не создаются внутри класса, а передаются снаружи через конструктор или параметры.

Проблема, которую решает: Без DI классы создают зависимости внутри себя, что делает код жестко связанным, сложным для тестирования и изменения.

Преимущества: Легко тестировать, легко менять реализации, слабая связанность между компонентами, единая точка управления зависимостями.

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?**

Repository - низкоуровневая работа с БД: SQL-запросы, CRUD операции, фильтрация, пагинация.

Service - бизнес-логика: валидация данных, проверка правил, координация работы репозиториев, управление транзакциями (commit, rollback), интеграция с внешними сервисами.

Controller - обработка HTTP-запросов: валидация параметров, преобразование данных (DTO), обработка HTTP-статусов и ошибок.

Каждый слой решает одну задачу, код легко тестировать и переиспользовать, можно менять один слой без влияния на другие, легче масштабировать.

При объединении Repository и Controller бизнес-логика смешается с HTTP-обработкой, сложно тестировать, невозможно переиспользовать логику в других местах, при изменении БД придется менять контроллер.

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

При обработке HTTP-запроса Litestar создает зависимости через провайдеры. provide_db_session() создает новую сессию БД, затем создаются репозиторий и сервис, которым передается эта сессия. Сессия активна во время всей обработки запроса. После завершения обработки выполняется блок finally, который закрывает сессию. Каждый HTTP-запрос получает свою изолированную сессию БД.

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

async - объявление асинхронной функции. await - ожидание выполнения асинхронной операции.

Используется для неблокирующих I/O операций. Когда функция ожидает ответа от БД, поток освобождается для обработки других запросов.

Вместо блокировки потока на время ожидания БД, поток может обрабатывать множество запросов одновременно, что повышает производительность.

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()?**

При создании session внутри репозитория непонятно, кто управляет жизненным циклом сессии, когда её закрывать, как управлять транзакциями, как обеспечить изоляцию между операциями, как подменить сессию для тестов.

Передача session как аргумента позволяет управлять жизненным циклом извне через DI, контролировать транзакции на уровне сервиса, обеспечивать изоляцию, легко тестировать.

commit() и rollback() вызываются в Service слое, потому что сервис знает контекст бизнес-логики и может решать, когда завершать транзакцию.

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

Пагинация ограничивает количество возвращаемых записей. С пагинацией передается только нужная порция данных , память используется эффективно, пользователь видит управляемое количество записей на странице.

count - количество записей на странице, page - номер страницы.

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.**

Хеширование пароля - перед сохранением пароль должен быть захеширован с использованием bcrypt или аналогичного алгоритма.

Отправка приветственного письма - после создания пользователя отправляется email с приветствием и инструкциями.

Валидация бизнес-правил - проверка, что email принадлежит разрешенному домену, username не содержит запрещенных слов, соблюдение требований к сложности пароля.

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

GET /users/{user_id}: 200 OK при успехе, 404 Not Found если пользователь не найден.

GET /users: 200 OK при успехе, 400 Bad Request при неверных параметрах запроса.

POST /users: 201 Created при успешном создании, 400 Bad Request при ошибке валидации или нарушении бизнес-прави.

PUT /users/{user_id}: 200 OK при успешном обновлении, 404 Not Found если пользователь не найден, 400 Bad Request при ошибке валидации или нарушении бизнес-правил.

DELETE /users/{user_id}: 204 No Content при успешном удалении, 404 Not Found если пользователь не найден.

Обоснование:

200 OK - стандартный статус для успешных GET и PUT запросов.

201 Created - ресурс успешно создан.

204 No Content - успешное удаление без тела ответа.

404 Not Found - ресурс не существует.

400 Bad Request - ошибка клиента в данных запроса или параметрах.

500 Internal Server Error - неожиданная ошибка сервера (Litestar делает это автоматически).

HTTP-статусы помогают клиенту понять результат операции и правильно обработать ответ. Следование стандартам REST делает API понятным и предсказуемым.