**Архитектурная схема проекта**

В основе проекта лежит проверенный шаблон приложения на FastAPI, который успешно использовался в предыдущих проектах, что позволяет создавать масштабируемый микросервисы, был вдохновлен Django структурой в каком то смысле.

**Структура проекта:**

* **.github/** — содержит настройки GitHub Actions, реализующие всю логику CI/CD для автоматизации сборки, тестирования и деплоя проекта.
* **tests/** — папка с тестами приложения.

**conftest.py** — конфигурационный файл для тестовой среды, где:

1. применяются миграции к тестовой базе данных,
2. добавляются тестовые данные с помощью Faker,
3. настраивается тестовая сессия, которая переопределяет зависимость FastAPI **get\_session** для работы с тестовой базой, чтобы изолировать тесты от боевой БД.

Остальные файлы содержат юнит-тесты и интеграционные тесты:

**dao-test.py** — юнит-тесты методов доступа к данным,

**router-test.py** — интеграционные тесты, проверяющие корректность ответов API и взаимодействие с базой данных.

* **Корень проекта** — содержит дополнительные файлы конфигурации:

настройки pre-commit с линтерами,

1. Dockerfile для сборки образа,
2. скрипт применения миграций при старте приложения,
3. Makefile с готовыми командами для сборки, проверки и запуска проекта.

* **app/** — основная папка с кодом приложения.

**dao/** — базовый класс для доступа к данным в базе. Реализует общие методы работы с базой, которые переопределяются в дочерних классах для каждой конкретной таблицы.

**exceptions/** — кастомные обработчики исключений, возвращаемых сервером.

**logs/** — директория с логами приложения.  
Логи настроены так, что успешные операции записываются в один файл (**log**), а ошибки — в отдельный файл (**error**).

**migrations/** — папка с миграциями базы данных.

**config/** — настройки приложения:

1. получение значений из переменных окружения,
2. настройка логирования loguru.

**data\_generate/** — генерация тестовых данных с помощью Faker, чтобы база не оставалась пустой.

**database/** — файл с базовой настройкой для асинхронной работы с базой данных через SQLAlchemy.

1. Создаются асинхронные движки (**engine** и **test\_engine**) для основной и тестовой БД с URL из настроек.
2. Определяются асинхронные сессии (**async\_session** и **async\_test\_session**).
3. Заданы общие типовые аннотации для колонок (первичный ключ, даты создания/обновления, уникальные строки).
4. Объявлен базовый класс **Base** для моделей, который:

автоматически задаёт имя таблицы во множественном числе (нижний регистр),

содержит поля **created\_at** и **updated\_at** с автозаполнением датой и временем,

предоставляет метод **to\_dict()** для конвертации модели в словарь с колонками и значениями.

**dependencies/** — содержит функцию **get\_session**, которая возвращает асинхронную сессию базы данных (**AsyncSession**) с использованием внедрения зависимостей FastAPI. Это позволяет разделить работу с основной и тестовой базой данных, предотвращая взаимное влияние.

**main.py** — точка входа в приложение.

1. В **lifespan** реализован асинхронный менеджер жизненного цикла приложения, который:

\*запускает миграции базы через Alembic,

\*удаляет и создаёт таблицы заново для инициализации,

\*заполняет базу тестовыми докторами, пациентами и приёмами.

\*инициализируется FastAPI с метаданными, тегами и контактной информацией.

\*подключается маршрутизатор с эндпоинтами для управления приёмами (**router\_appointment**).

\*регистрируются обработчики исключений (HTTP ошибки, ошибки целостности, ошибки валидации).

\*добавлен эндпоинт **/health** для проверки статуса сервиса (возвращает **{"status": "ok"}**).

\*запуск сервера происходит через **uvicorn** с автообновлением при изменениях.

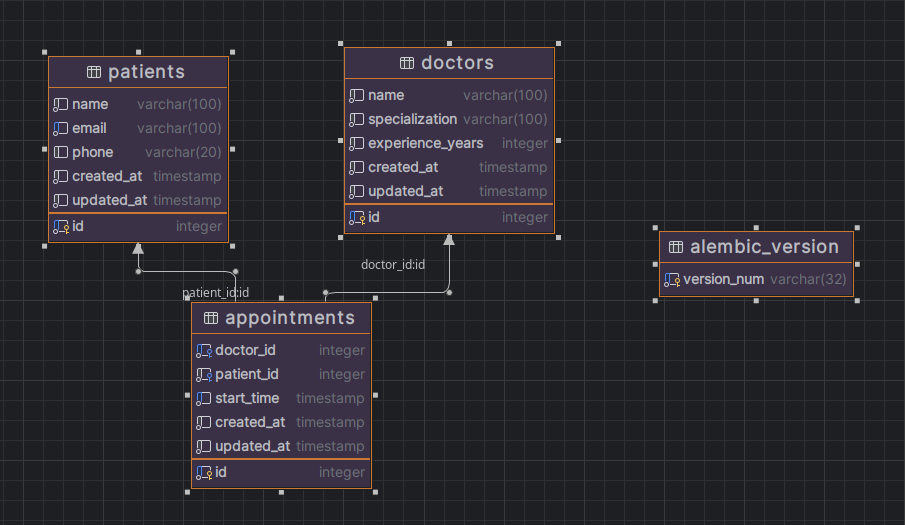
**appointments/** — модуль, реализующий логику эндпоинтов заявок на приём.

В **models** описаны таблицы базы данных.

В **router** реализованы маршруты API.

В **schemas** и **rb** описаны Pydantic-модели для валидации входящих запросов и формирования ответов API.

**ER-диаграмма**



**Activity-диаграмма главного сценария «пациент записывается на приём».**

**Начало**  
⬇️

**1. Пациент открывает приложение / сайт**  
⬇️

**2. Пациент выбирает врача**  
⬇️

**3. Проверка: существует ли выбранный врач в системе**  
⬇️

Если врач не найден —  
 ↳ показать ошибку и вернуться к выбору врача

Если врач найден —  
 ↳ перейти к следующему шагу  
⬇️

**4. Пациент выбирает время приёма**  
⬇️

**5. Пациент вводит свои данные (ФИО, контакты и т.д.)**  
⬇️

**6. Проверка: существует ли пациент в системе**  
⬇️

Если пациент не найден —  
 ↳ пока выдаст ошибку ,но в будущем логично зарегистрировать пользователя

Если пациент найден —  
 ↳ перейти к следующему шагу  
⬇️

**7. Проверка уникальности записи (врач + пациент + время)**

Проверяется, что для этого врача и этого пациента нет уже существующей записи на выбранное время

Проверяется, что пациент и врач не совпадают (в том смысле что такого сочетания id нет в БД не зависимо от времени приема) (уникальность id доктора и пациента)  
⬇️

Проверка занятости врача с учетом интервала ± 1 час, что б не вышло что запись будет через 1 после начала приема другого пациента.

(в Faker я даже делал что б записи были в период с 8:00 до 18:00)

Если проверка не пройдена —  
 ↳ показать ошибку и вернуться к корректировке данных

Если проверка прошла успешно —  
 ↳ перейти к следующему шагу  
⬇️

**9. Система сохраняет запись в базе данных**  
⬇️

**10. Пациент получает подтверждение записи**  
⬇️

**Конец**

**Документ бизнес-процесса**

Три вида ролей:

Админ — полный доступ

Доктор — получения своих записей, получение своих пациентов. В контексте записей не может ничего поменять.

Пациент — получает список докторов, список своих записей ,создает запись, удаляет запись, изменяет запись в смысле время приема.

**Раздел «Проектирование → реализация»**

Проектирование предшествовало реализации, о чём свидетельствует последовательность шагов и принятые архитектурные решения:

1. **Выбор шаблона проекта**  
   Использован проверенный шаблон FastAPI-проекта, применяемый мной в предыдущих задачах. Он обеспечивает быструю стартовую конфигурацию с уже организованной структурой директорий, системой зависимостей и dev-утилитами.
2. **Настройка инструментов качества кода**  
   С самого начала настроены линтеры (flake8, mypy, black) и интегрированы через pre-commit хуки, чтобы поддерживать единый стиль и избежать потенциальных ошибок.  
   Дополнительно настроен **GitHub Actions**, запускающий линтеры и типизацию при каждом пуше/мерже в репозиторий.
3. **Проектирование модели данных**  
   В соответствии с заданием были спроектированы основные сущности: Doctor, Patient, Appointment.  
   Принятые проектные решения:
   * Связи через ForeignKey.
   * on\_delete=CASCADE — чтобы при удалении доктора/пациента соответствующие записи удалялись.
   * В модели Appointment добавлены дополнительные поля для валидации времени и логики записи.
4. **Подготовка БД и генерация тестовых данных**  
   Ещё до реализации логики был настроен Faker для генерации тестовых данных — это позволило сразу работать с наполненной БД и проектировать интерфейсы запросов под реальные данные.
5. **Проектирование и реализация API**
   * Сначала реализован GET /appointments/ — просмотр существующих записей.
   * Затем добавлен POST /appointments/ с кастомной логикой добавления: проверка уникальности записи и недопустимости временных пересечений.
   * Базовый DAO-класс содержит общие CRUD-методы, а AppointmentDAO переопределяет метод создания под заданные условия.
6. **Валидация входных/выходных данных**
   * Разработаны Pydantic-схемы для входных и выходных данных.
   * Продумана валидация (validator) времени в AppointmentCreate.
   * Добавлены кастомные сообщения в случае некорректных doctor\_id или patient\_id.
7. **Обработка ошибок**  
   Использована уже готовая и проверенная структура HTTP-исключений, общая для всех моих проектов. Она полностью удовлетворяет требованиям текущей задачи.
8. **Тестирование**
   * Настроены фикстуры для изолированной работы с тестовой БД (lifespan с автозаполнением данных).
   * Реализованы юнит-тесты для GET и POST методов, включая граничные случаи и негативные сценарии.
   * Применены как общие фикстуры, так и специфичные для данной задачи.
9. **Docker и развёртывание**
   * С самого начала использовалась PostgreSQL в контейнере.
   * В финале дописан docker-compose.yml с ресурсными ограничениями и готовностью к использованию в Docker Swarm.
   * Также подготовлен Makefile с командами для локальной сборки, тестирования и запуска всех проверок.
10. **Финальное тестирование**  
    Все этапы интеграции и проверки завершаются через make check + GitHub Actions, которые подтверждают, что проект проходит полную проверку линтерами, типизацией и тестами — без ошибок как локально так и на сревере.