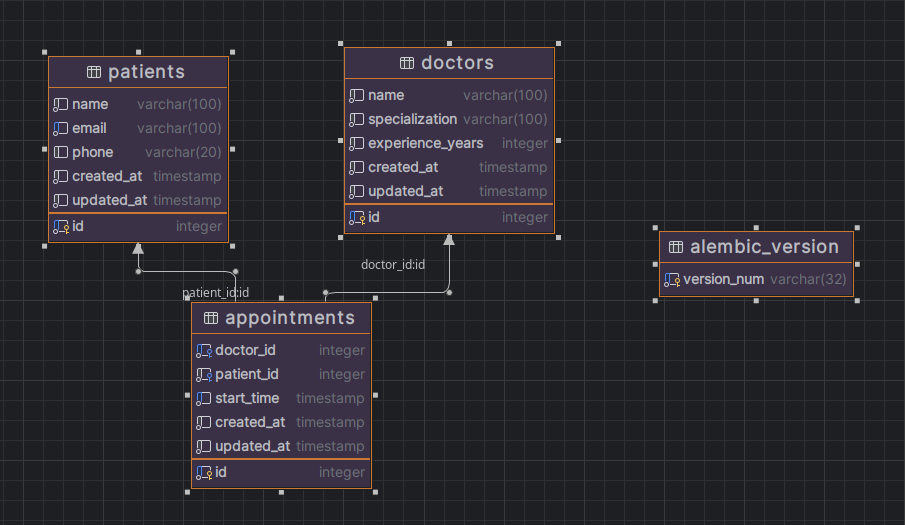
### **Архитектурная схема проекта**

**В основе проекта лежит проверенный шаблон приложения на FastAPI, который успешно использовался в предыдущих проектах.**

**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:**
    - **применяются миграции к тестовой базе данных,**
    - **добавляются тестовые данные с помощью Faker,**
    - **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,**
  + **Dockerfile для сборки образа,**
  + **скрипт применения миграций при старте приложения,**
  + **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:**
    - **получение значений из переменных окружения,**
    - **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.**
    - **Создаются асинхронные движки (engine и test\_engine) для основной и тестовой БД с URL из настроек.**
    - **Определяются асинхронные сессии (async\_session и async\_test\_session).**
    - **Заданы общие типовые аннотации для колонок (первичный ключ, даты создания/обновления, уникальные строки).**
    - **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.**
  + **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 (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 преод 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.

**Отдельный раздел «Проектирование → реализация»**: докажите, что сначала спроектировали, а потом писали код (короткий порядок шагов и принятые решения).

- подготовка шаблона в работе (пользуюсь не первый раз)

- настройка линтеров через precomit

- настройка GitHub Actions на проверку Линтерами

- Создание моделей Доктор, Пациент, Запись (Добавил полей что б было не скучно). Связь через Foreign Key была определан в задании , добавил лишь что б при удалениии Доктора или Пациента запись удалялаась

- Создание routera с двумя маршрутами.

- Так как с самого начала в Faker добавил данные в БД, то сначло разработал Get метод.

- Продумал валидатор Pydentic

- Следом Post метод , там переопределил метод добавленния записи в таблицу Appointment под мои условия(уникальность id, не пререкрывать запись по вермени,.), как я говорил у меня в DAO определен базовый класс взаимодействия с БД и он переопределяется для каждой таблицы отдельно, но по умолчанию там уже есть готовын CRUD методы для работы с таблицей, потому метод добавления я в таблице Appointmetn переопределил, под свои задачи

- Добавил модель ответа Pydentic на post запросов

- Не переопределял структуры ответа на HTTP Exceptions, они у меня реализованы и меня устаривают, если нужно, то можно поправить структуру ответа.

- При проверка понял что не хватает ответа в случае некорректного ввода id доктора или пациента, добавил

- Настройка и реализация тестирования, продумака доплнительных фикстур, просто базовые у меня уже есть , там преопределить натсройки для работы с тестовой БД , что б lifespan вносил тестовые записи и готовил базу для тестов, нужно было просто создать еще пару фикстур. Потом писал тесты для разных методов, что нужно было проверить.

- Так как с самом начале я уже поднял postgres sql для работы теперь завершаю работу с докер, дописал Docker compose манифест, пригодный для развернтывания и сразу с ограничением ресурсов docker swarm.У себя на мониторинге просто так пользуюсь.

- разработка make file

Потом все тестирую в сборе через makefile, проверяб что git actions отрабатывают без ошибок