**Министерство науки и высшего образования Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**(Университет ИТМО)**

Факультет **Прикладной информатики**

Направление подготовки **09.03.03 Прикладная информатика**

Образовательная программа **Мобильные и сетевые технологии**

**К У Р С О В О Й   П Р О Е К Т**

Тема: «Сервис для управления медицинскими данными на основе блокчейн»

Обучающийся: Трикула Артем Константинович K3141

Санкт-Петербург 2024

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_gjdgxs)

[1 Описание проекта 5](#_30j0zll)

[2 Работа над проектом 6](#_1fob9te)

[2.1 Задачи команды 6](#_3znysh7)

[2.2 Задачи, поставленные передо мной 7](#_2et92p0)

[2.3 Выполнение задачи 8](#_tyjcwt)

[3 Коммуникация в команде 20](#_1t3h5sf)

[4 Анализ собственной работы 21](#_4d34og8)

[ЗАКЛЮЧЕНИЕ 21](#_2s8eyo1)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 23](#_17dp8vu)

[ПРИЛОЖЕНИЕ 24](#_3rdcrjn)

# **ВВЕДЕНИЕ**

В настоящее время в сфере хранения медицинских данных существует ряд проблем.

Проблема разрозненного хранения данных. В большинстве медицинских учреждений данные пациентов хранятся в локальных базах данных. Каждый пациент имеет отдельные карточки в разных учреждениях, что создаёт серьёзные сложности при попытке собрать полную медицинскую историю. Эта разрозненность препятствует быстрой и точной диагностике, а также снижает качество медицинского обслуживания.

Проблема отсутствия прозрачности доступа к данным.В текущей системе пациенты не имеют контроля над тем, кто и когда просматривает их медицинские данные. Таким образом получение доступа происходят без ведома пациента, что ставит под угрозу его право на приватность и защиту данных. К тому же пациенты не имеют уверенности в подлинности внесенных записей, так как нет прозрачного механизма отслеживания изменений.

Проблема централизованного хранения с риском утечек данных.В большинстве медицинских учреждений данные хранятся на централизованных серверах, которые часто подвергаются риску взломов и утечек. Учитывая, что многие учреждения сталкиваются с ограничениями по бюджету, уделять внимания мерам информационной безопасности оказывается недостаточным. Это создает условия для возможных утечек конфиденциальной информации, о которых пациент зачастую даже не будет знать.

Разработка безопасной и прозрачной системы хранения медицинских данных на основе блокчейна решает указанные проблемы, улучшая качество медицинского обслуживания.

Результаты проекта предназначены для медицинских учреждений, врачей и пациентов. Благодаря данной системе пациенты смогут получить контроль над своими личными данными, а врачи – доступ к актуальной медицинской информации, что поможет повысить качество обслуживания.

Целью данного проекта является разработка серверной части прототипа системы хранения медицинских данных на основе блокчейна, направленной на улучшение качества обслуживания пациентов, а также обеспечение безопасности их личных данных.

Проект решает ряд ключевых задач:

- обеспечение безопасного и децентрализованного хранения данных: разработка системы с использованием блокчейн-технологий позволит обеспечить более безопасное хранение личных данных пользователей,

- прозрачность операций, связанных с изменением данных: система будет отслеживать все произведенные изменения и отображать, кто и когда их внес,

- контроль доступа к данным: для получения доступа к данным пациента врачу будет необходимо запросить его. После получения запроса, пользователь сможет разрешить или запретить врачу доступ к своим данным,

- актуальность и облегчение поиска медицинских данных: благодаря системе, которая позволит хранить в себе все необходимые для врача медицинские данные о пациенте, повысится качество обслуживания посетителей медицинских учреждений.

Все перечисленные задачи направлены на создание безопасной системы, которая позволит пользователям самостоятельно распоряжаться доступом к своим личным данным, а врачи смогут быстро получать доступ ко всей необходимой для них информации.

# **1 Описание проекта**

Система управления медицинскими данными представляет собой сервис для специальных учреждений. Разработанный веб-сайт направлен на улучшение качества обслуживания пациентов, безопасности хранения медицинских данных, а также облегчение поиска необходимой для врачей информации.

Проект был разработан с использованием современных технологий, обеспечивающих кроссплатформенность и высокую производительность. Более подробное описание используемых технологий:

- сервис Figma использовался для разработки дизайна веб-сайта. Это отличный инструмент для командной работы, а также довольно простой в использовании,

- React.js использовался на клиентской части сервиса для обеспечения кроссплатформенности, а также для облегчения работы в команде. Благодаря архитектуре, он позволяет разрабатывать свою часть системы, не думая о том, что можно повредить чужую,

- для создания серверной части сервиса использовался Python. Python является довольно удобным и простым инструментом для прототипирования,

- для работы с базой данных использовался PostgreSQL. Это СУБД с открытым исходным кодом и поддержкой множества типов данных, которая отлично подходит для прототипирования.

Таким образом, комбинация React.js, Figma, Python, PostgreSQL обеспечила разработку эффективного и кроссплатформенного прототипа системы управления медицинскими данными с высокой производительностью и удобным пользовательским интерфейсом.

# **2 Работа над проектом**

## **2.1 Задачи команды**

В начале работы над проектом командой был составлен план разработки:

1. создание и редактирование технического задания,
2. распределение обязанностей и установка сроков выполнения задач,
3. выполнение индивидуальных задач,
4. подготовка к защите проекта,
5. защита проекта — командная защита с презентацией,
6. написание индивидуального отчета.

Далее для бэкенд команды были выделены общие задачи, которые далее были разделены между всеми участниками:

- спроектировать архитектуру базы данных: создать ER-диаграммы, чтобы отобразить структуру данных, основные сущности (таблицы) и связи между ними,

- разработать схемы API-запросов: с помощью Swagger создать прототип запросов. Прототип запросов на Swagger поможет быстрее наладить интеграцию фронтенда с бэкендом. Для доступа к документации API по адресу {serverAddress}/docs настроить URL и конфигурации Swagger в проекте Django,

- развернуть приложение на сервере с использованием Docker: настроить Docker-контейнеры для изоляции приложения и его зависимостей, создать Dockerfile и docker-compose файлы для автоматизации сборки и развертывания, что упростит последующее управление средами разработки и продакшена, подключить приложение к базе данных,

- поддерживать интеграцию бэкенда: после окончания разработки серверной части оказывать помощь в интеграции фронтенд-разработчикам смежной команды, что позволит избежать реализацию непредусмотренных возможностей приложения и с ускорит разработку,

- провести тестирование эндпоинтов: роведение ручного тестирования для проверки работоспособности всех эндпоинтов и бизнес-логики. Тестирование включает проверку корректности данных, безопасности и стабильности работы системы на некорректных данных,

- развернуть приложение на сервере: настроить виртуальную машину или сервер и развернуть приложение. Настроить веб-сервер (например, Nginx), подключить его к Django через Gunicorn или другой WSGI-сервер. На этом этапе проверить работу миграций, подключить базу данных, кэш и фоновые задачи, если они требуются,

- написать отчет и подготовить презентацию: подготовить отчет о проделанной в ходе проекта работы; подготовить презентацию с демонстрацией работы разработанной системы, описанием пройденных этапов и полученных навыков.

## **2.2 Задачи, поставленные передо мной**

Глобально передо мной была поставлена задача создания бэкенд приложения и поддержка его на каждом этапе. Мне необходимо было:

1. инициализировать пустой python проект на с файловой структурой, необходимой для используемого нами фреймворка Django,
2. реализовать эндпоинты для врачей и администраторов со специальными «заглушками» – шаблонными ответами сервера, что позволило другим командам использовать приложение, не дожидаясь его полной разработки,
3. добавить к эндпоинтам логику: обращение к базе данных и отправку ответов сервера к клиенту,
4. написать интерактивную документацию для API, что позволит членам других команд быстрее внедрять взаимодействие с приложением, а также позволит делать запросы через специальный интерфейс,
5. реализовать запуск приложения на удаленном сервере с помощью Docker.

## **2.3 Выполнение задачи**

Первая поставленная задача – реализация пустого проекта. Как уже упоминалось, для создания приложения был выбран фреймворк Python – Django. Так как перед нами стоит задача создания web API, было принято решение использовать набор инструментов Django REST framework (сокращено DRF), который служит именно для таких целей. Стоит отметить, что ранее у меня был очень небольшой опыт работы с Django, поэтому для начала я погрузился в изучение документации [1, 2]. После этого был создан проект с файловой структурой, представленной ниже (см. рисунок 1).

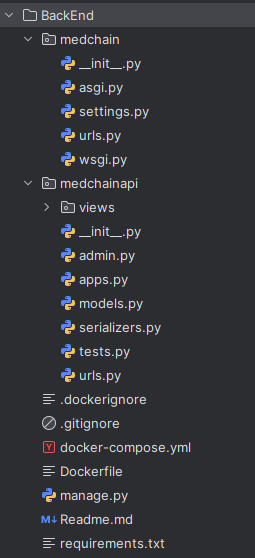


Рисунок 1 – Структура проекта

medchain:

- \_\_init\_\_.py – файл инициализации пакета,

- asgi.py – точка входа для ASGI-сервера, используется для асинхронного обслуживания запросов,

- settings.py – файл конфигурации проекта, содержит настройки базы данных, приложений, статических файлов и других параметров,

- urls.py – маршрутизация основного проекта, определяет пути к различным приложениям,

- wsgi.py – точка входа для WSGI-сервера, необходим для запуска приложения на сервере.

medchainapi:

- views – реализация обработчиков запросов (контроллеров),

- admin.py – настройка административной панели Django,

- apps.py – регистрация приложения в проекте,

- models.py – описание моделей данных (структур таблиц базы данных),

- serializers.py – сериализация и десериализация данных, связующая звено между моделями и API,

- tests.py – модуль для написания тестов,

- urls.py – маршрутизация приложения, связывает запросы с обработчиками.

Корневые файлы проекта:

- .dockerignore – список файлов и директорий, игнорируемых Docker при создании образа,

- .gitignore– файлы и директории, исключенные из контроля версий,

- docker-compose.yml – описание сервисов, необходимых для запуска проекта в контейнерах Docker,

- Dockerfile – инструкции для создания Docker-образа приложения,

- manage.py – утилита для выполнения команд Django (например, запуск сервера разработки, миграции),

- requirements.txt – список зависимостей проекта,

- Readme.md – файл документации для краткого описания проекта.

На следующем этапе разработки мне необходимо было добавить описанные ранее шаблонные ответы сервера – «заглушки». Для этого были созданы специальные модели и сериализаторы, которые играют ключевую роль в реализации поставленной передо мной задачи.

Как уже было упомянуто, файл models.py в Django используется для описания моделей данных. Модель – это класс Python, который соответствует таблице в базе данных. Каждое поле в классе представляет колонку таблицы, а экземпляры класса – строки. Модели позволяют работать с данными высокого уровня, абстрагируясь от SQL-запросов, что упрощает взаимодействие с базой данных. Примеры реализованных моделей представлены ниже (см. рисунки 2, 3).

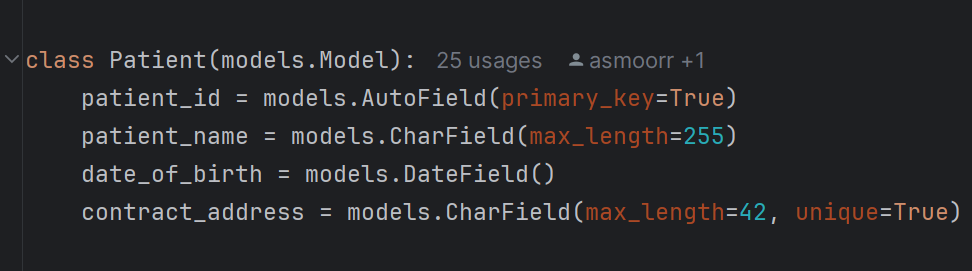


Рисунок 2 – Модель django, описывающая таблицу «пациент»

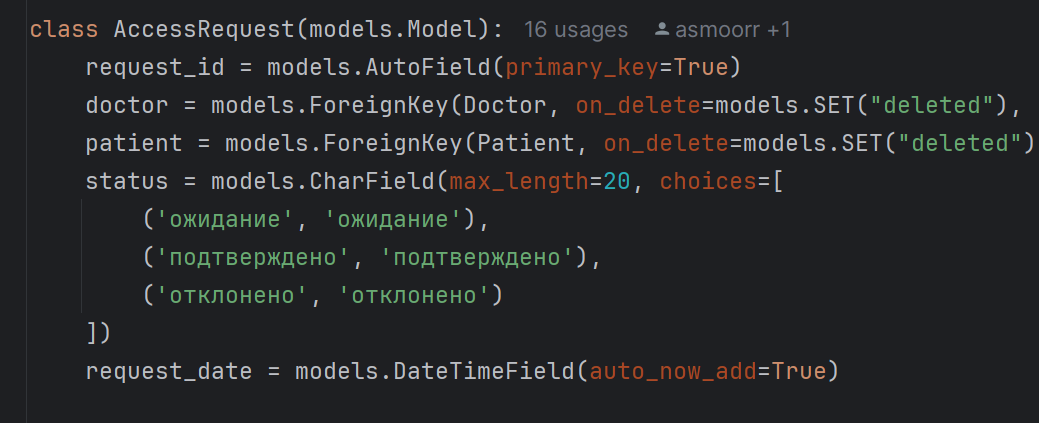


Рисунок 3 – Модель django, описывающая таблицу «запрос на доступ»

В свою очередь, сериализаторы описанные в файле serializers.py, в Django REST Framework отвечают за преобразование данных между форматами. Таким образом сериализатор выполняет следующие задачи:

- сериализация данных – преобразование объектов моделей (например, экземпляров класса Patient) в формат, который может быть передан через API, чаще всего, как и у нас в проекте, в JSON,

- десериализация данных – преобразование данных, полученных через API в объекты моделей,

- валидация данных – проверка корректности входных данных, включая проверки типа, длины, уникальности и пользовательские правила,

- управление представлением данных – выборочное отображение полей модели в зависимости от потребностей клиента.

Ниже представлены некоторые из реализованных сериализаторов (см. рисунки 4, 5).



Рисунок 4 – Сериализатор Django DRF, основанный на модели Doctor

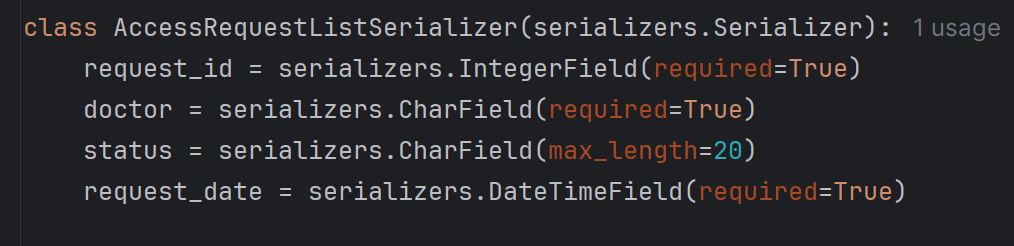


Рисунок 5 – Сериализатор Django DRF, не основанный на модели

Данный подход позволил упростить обмен данными между сервером и клиентами, обеспечив структурированность, надёжность и удобство использования API. С помощью моделей и сериализаторов была заложена основа для построения полноценной системы взаимодействия между серверной частью и внешними клиентами.

После создания моделей и сериализаторов можно было переходить к основному содержанию задачи – созданию заглушек. Для этого в директории views создается файл Python, в котором описывается специальный класс (контроллер), принимающий запросы и отправляющий подготовленную заглушку. Впоследствии в этом же классе будет описана логика работы с базой данных. Пример контроллера для поиска пациентов представлен ниже (см. рисунок 6).

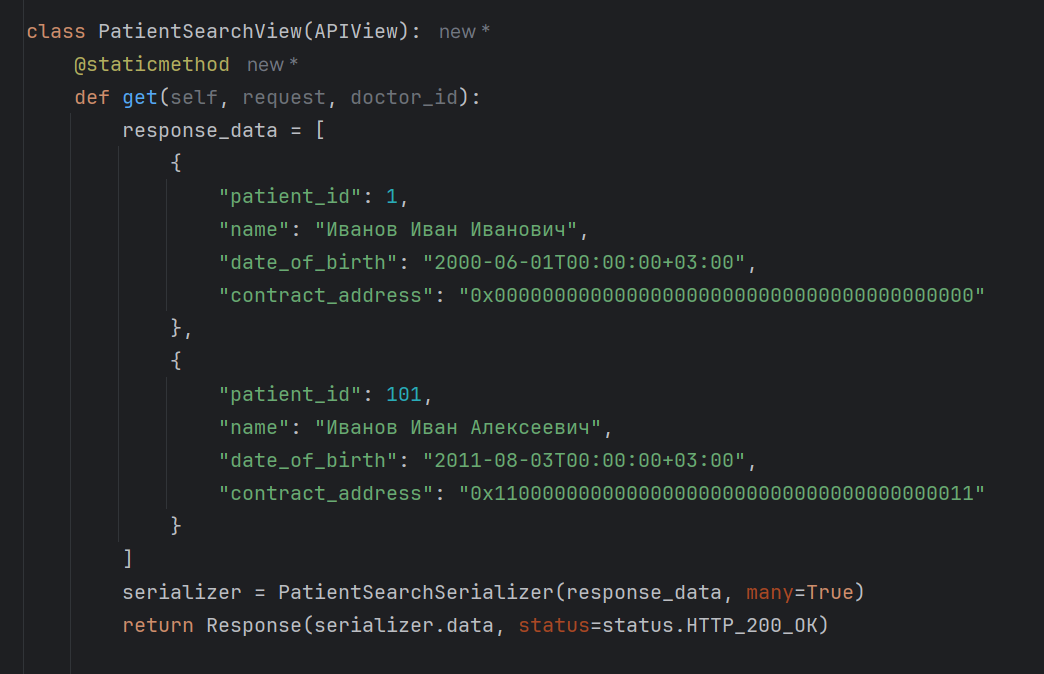


Рисунок 6 – Контроллер поиска пациентов с заглушкой

Следующим этап разработки являлось добавление функционала к уже частично написанным эндпоинтам. Необходимо добавить базу данных в Django проект и прописать логику взаимодействия с ней. Добавление происходит в файле settings.py: мы указаываем саму базу данных (PostgreSQL), ее название, пользователя, пароль и данные для запуска базы. Данные указываются отдельно для локального запуска и отдельно для запуска через Docker, через переменные среды (см. рисунок 7).

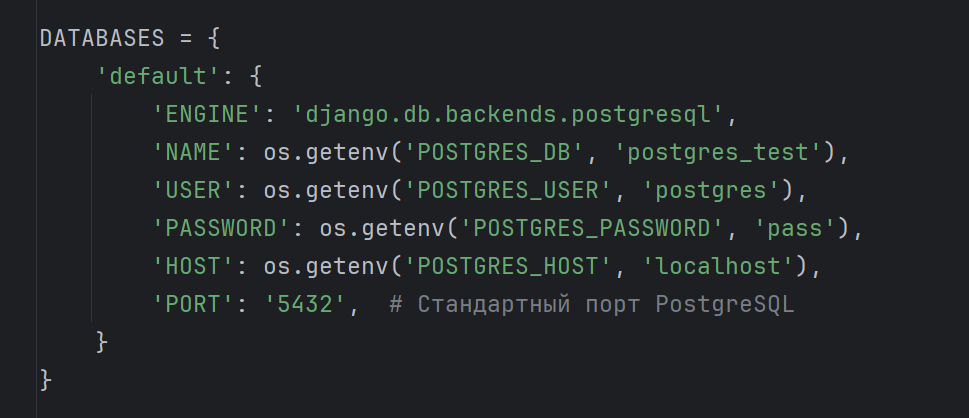


Рисунок 7 – Подключение базы данных

#### На данном этапе можно приступать к описанию логики программы. Необходимо получать данные с клиента, сериализовать их, делать запросы к базе данных и возвращать клиенту ответ.

Первым этапом обработки запроса является создание экземпляра сериализатора PatientFilterSerializer, которому передаются данные из тела запроса. Сериализатор выполняет валидацию и преобразование входящих данных, таких как имя пациента, дата рождения и идентификатор пациента. Код реализующий данную логику представлен ниже (см. рисунок 8).

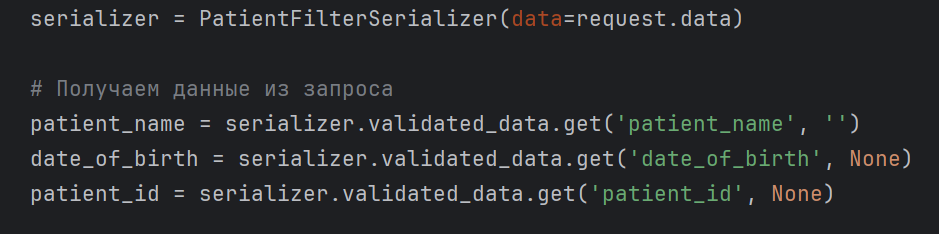


Рисунок 8 – Получение данных запроса

После сериализации создаётся базовый запрос ко всем объектам модели Patient, что позволяет получить полный набор данных для дальнейшей фильтрации по указанным в запросе параметрам (см. рисунок 9).

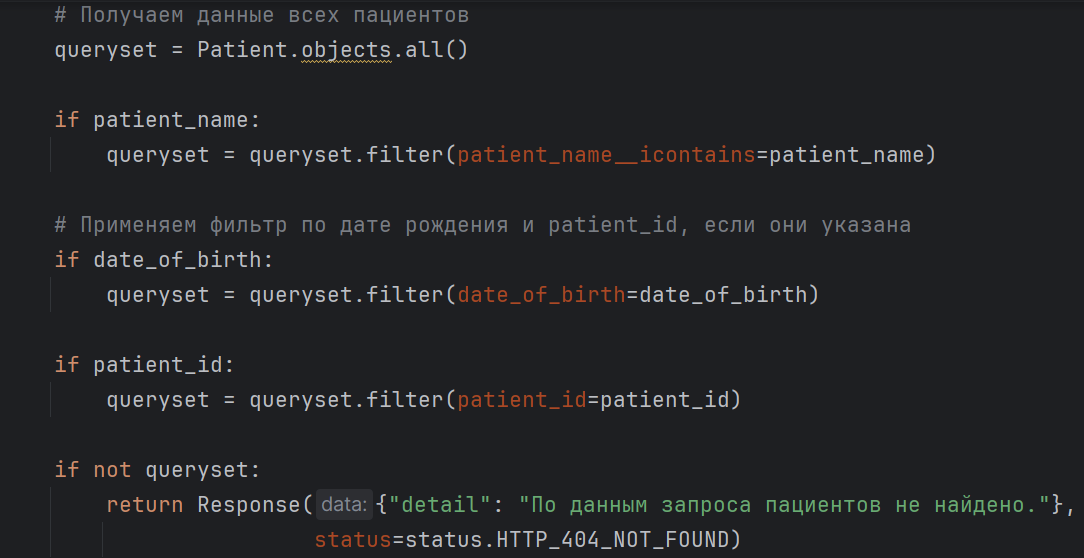


Рисунок 9 – Фильтрация по параметрам указанным в запросе

Если после применения фильтров подходящих записей не найдено, сервер возвращает сообщение об ошибке с кодом 404 (Not Found). В случае успешного поиска данных результаты сериализуются с помощью PatientSearchSerializer и отправляются клиенту в формате JSON с кодом успешного ответа 200 (OK). Код реализации представлен ниже (см. рисунок 10).

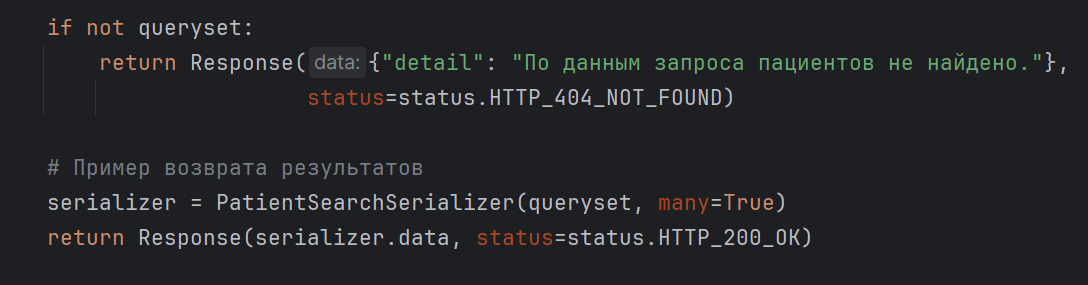


Рисунок 10 – Возвращение ответа клиенту

Далее следовал этап написания документации. Необходимо было реализовать документацию с использованием спецификации OpenAPI. Был выбран подход описания документации в коде, чтобы частично реализовать ее автоматическую генерацию. Для той части документации, которую необходимо было писать вручную была использована библиотека drf-spectrum, используемая для генерации схем с явным акцентом на расширяемость, настраиваемость и генерацию клиентов. Пример описания документации в коде программы и итоговый результат приведены ниже (см. рисунки 11, 12). Примеры для написания были взяты с официальной документации drf-spectrum [3].

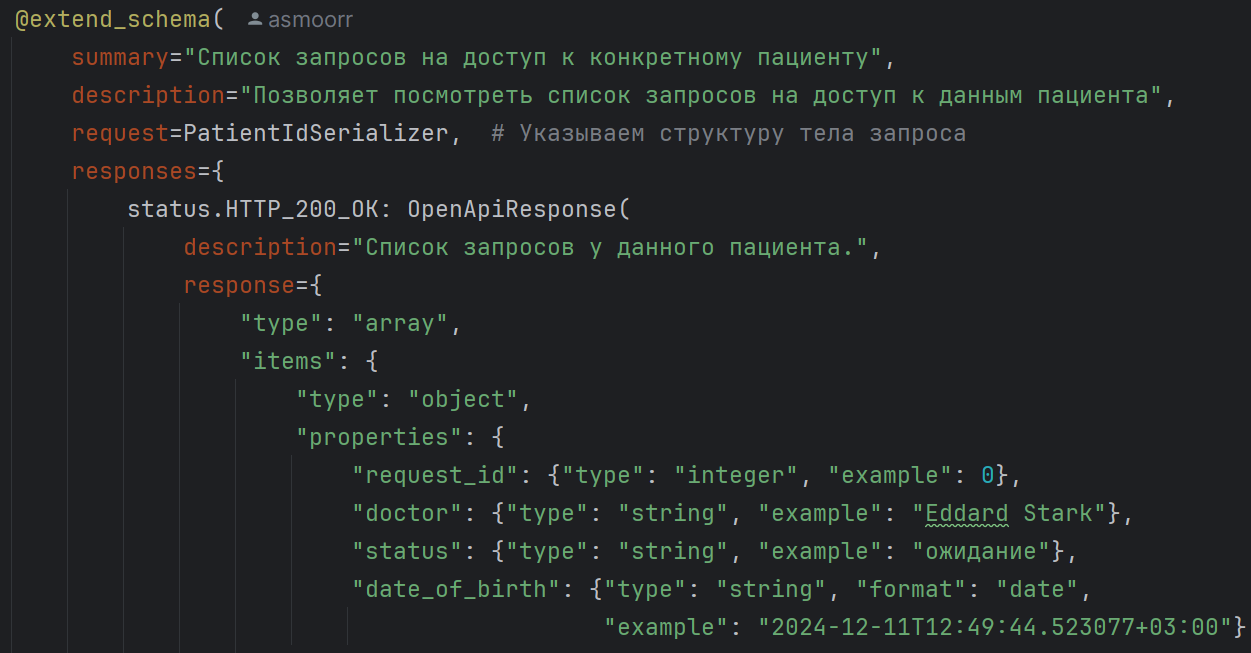


Рисунок 11 – Описание документации в коде приложения

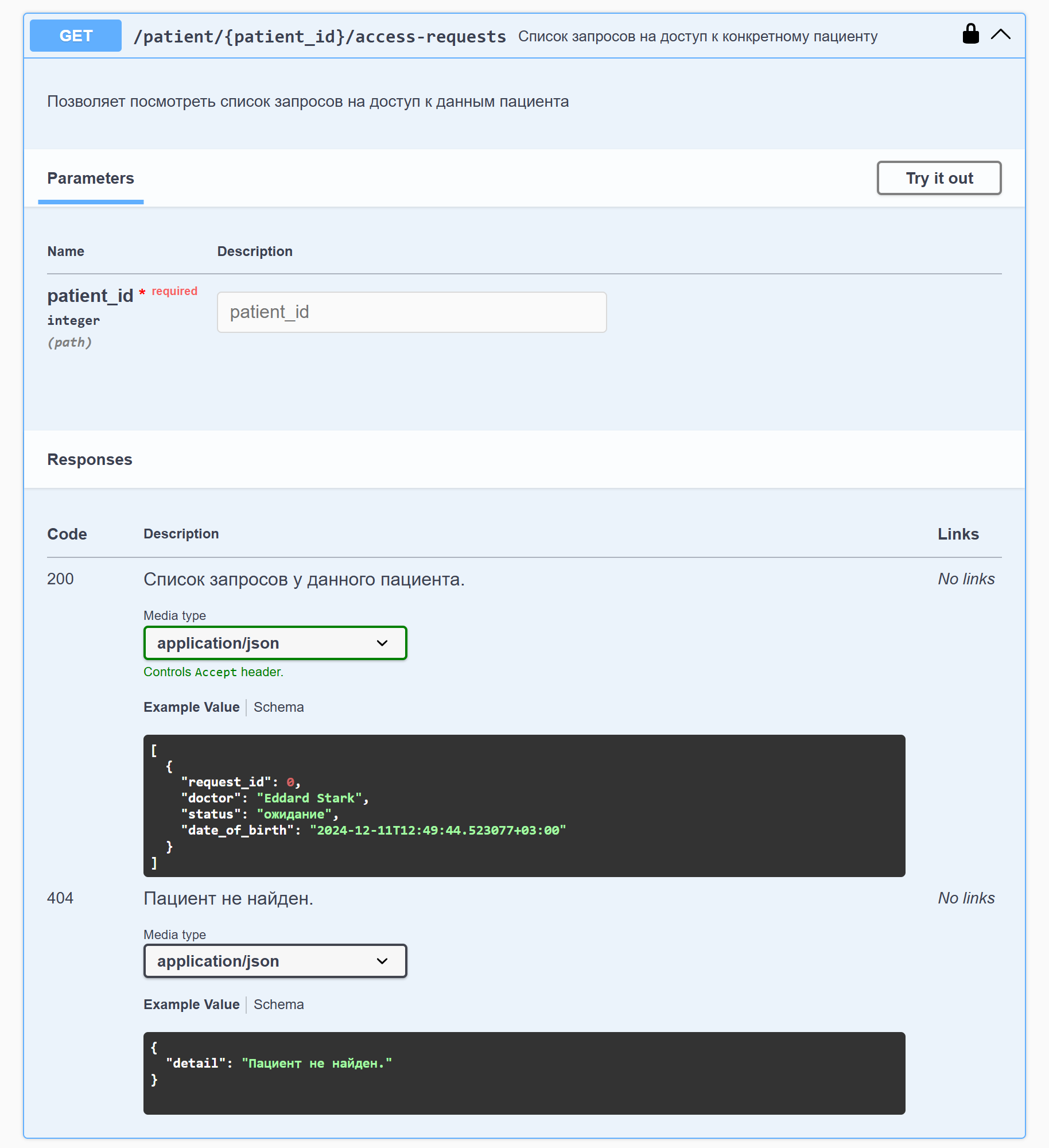


Рисунок 12 – Документация, открываемая в браузере

Таким образом, мной были написаны 7 эндпоинтов: 5 запросов для врачей (см. рисунок 13) и 2 эндпоинта, доступ к которым имеют только администраторы: по запросу на данные адреса происходит создание новых пациентов и докторов в базе данных (см. рисунок 14)

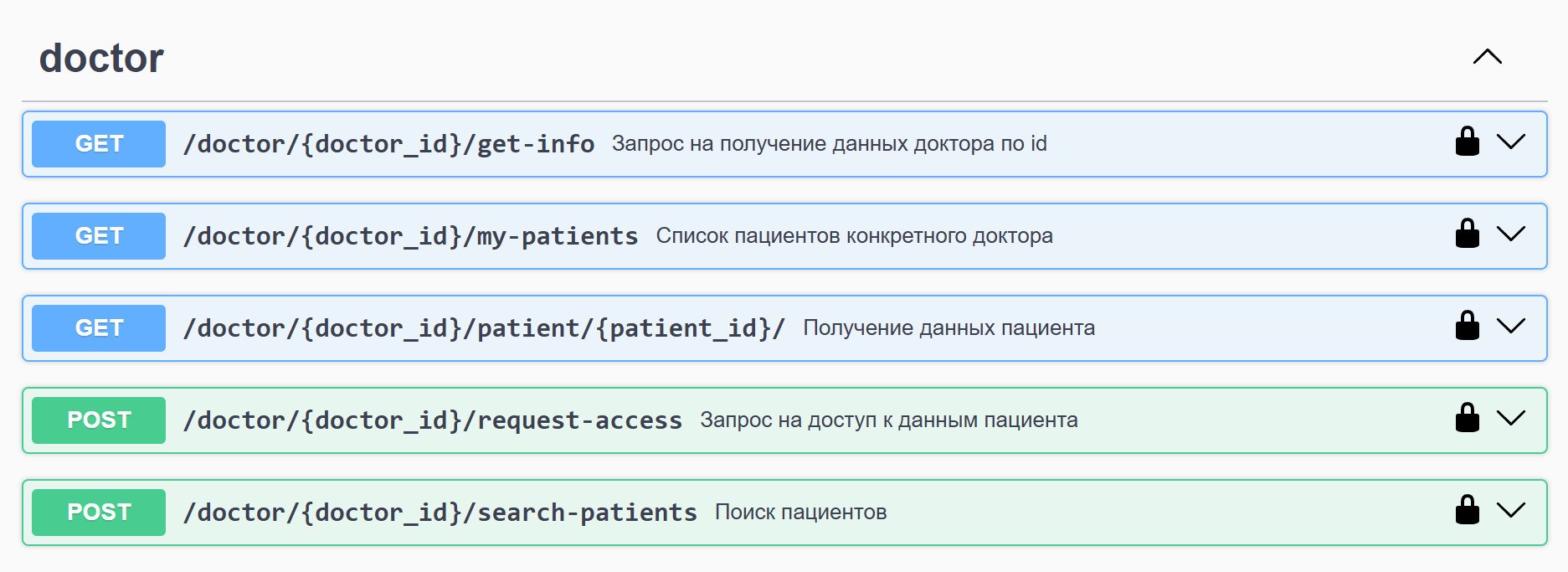


Рисунок 13 – Документация эндпоинтов докторов

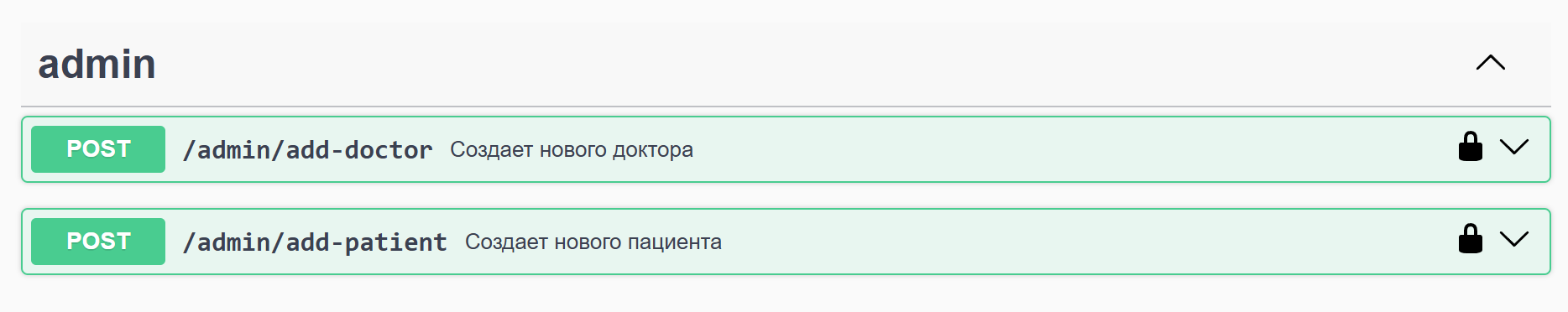


Рисунок 14 – Документация эндпоинтов администраторов

На следующем этапе необходимо реализовать запуск приложения через Docker контейнеры, для того чтобы их можно было удобно запускать на удаленном сервере с помощью одной команды. Для этого с опорой на официальные уроки [4] был написан Dockerfile, в котором описывается процесс создания контейнера с приложением. Данный процесс происходит в 2 этапа. На первом выполняется установка зависимостей (если зависимости были обновлены). Этот этап позволяет значительно ускорить сборку приложения. На втором этапе в контейнер копируются файлы проекта и устанавливается команда для запуска приложения: выполнение миграций и запуск встроенного сервера Django. Код с комментариями представлен ниже (см. рисунок 15).

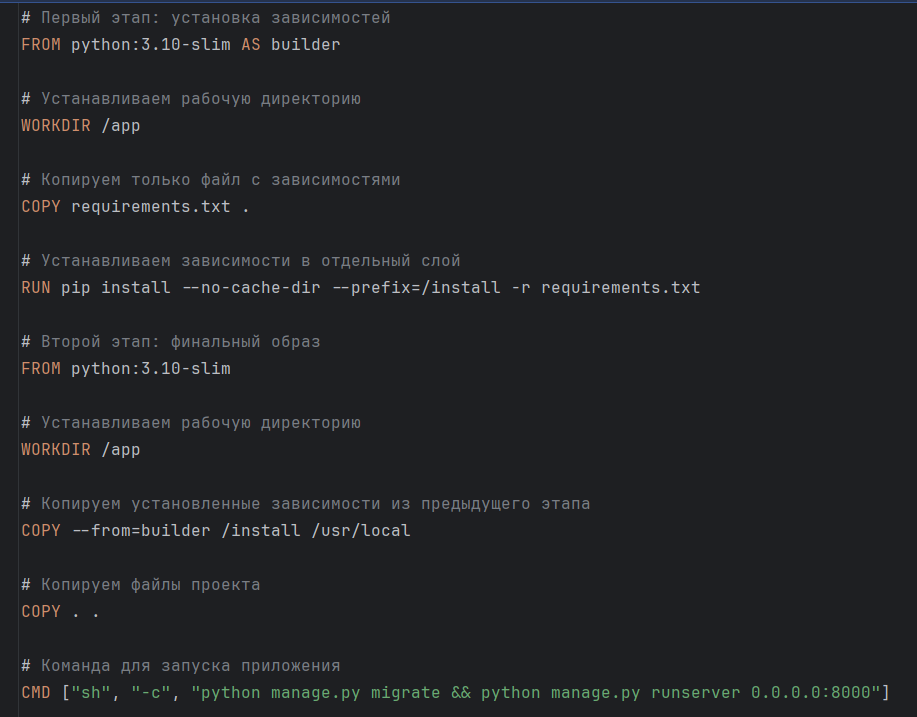
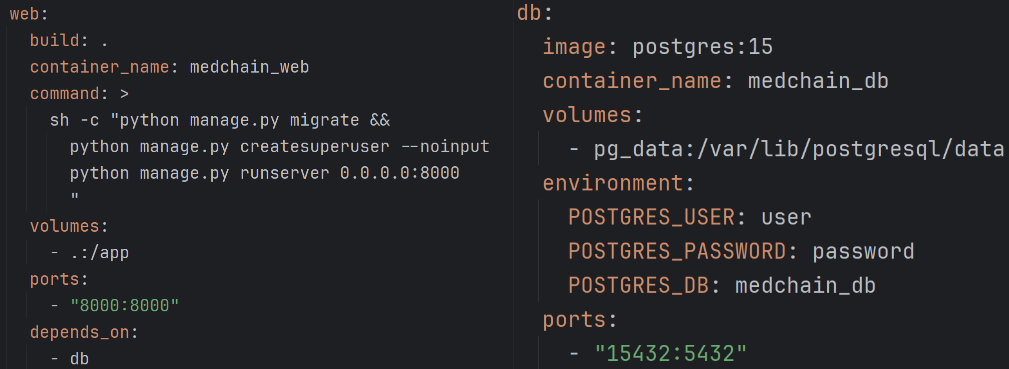


Рисунок 15 – Код Dockerfile

В файле docker-compose.yml определяется 2 сервиса. Первый (web) содержит написанное нами приложение. Он запускает сервер Django, предварительно выполняя миграции и создавая суперпользователя. Второй (db) – сервис на базе PostgreSQL, запускает базу данных, с которой взаимодействует приложение.

Оба сервиса настраиваются через переменные окружения, что обеспечивает гибкость и безопасность, а также позволяет изменять параметры (например, данные для подключения к базе или параметры приложения) без необходимости редактировать сами файлы конфигурации. Ниже представлен основной код сервисов (см. рисунки 16).

Рисунок 16 – Код сервисов web и db в файле docker-compose.yml

# **3 Коммуникация в команде**

Основное взаимодействие в команде происходило в телеграм-чате, на платформе itmo-learning-projects.ru, а также на еженедельных консультациях, на которых руководитель подробно объясняла поставленные перед нами задачи и оказывала помощь в поиске информации для их решения. В телеграм-чате всегда можно было задать вопрос или уточнить какую-либо информацию.

Разработка бэкенда приложения происходила совместно с Куделенской Эмилией. Первые несколько задач мы решали автономно, но уже на 2 неделю разработки стали активно взаимодействовать и помогать друг другу с возникающими вопросами. Так как у нас была небольшая команда, каждый смог вложить очень существенный вклад в итоговое приложение.

Кроме нашей команды, занимающейся разработкой бэкенд части приложения, была команда блокчейна, с который мы не очень много взаимодействовали, в связи с тем, что реализовали функционал, который не взаимодействует между собой напрямую (взаимодействие происходит через фронтэнд приложения). Но так как данный проект разрабатывался совместно с командой Алмазовой Лианы, которая отвечала за клиентскую часть приложения, происходило взаимодействие между нами и командой фронтэнд разработки. Это связано с тем, что они напрямую использовали наше приложение и иногда необходимо было изменять логику работы программы для более корректной отправки данных.

Отдельно хочется отметить работу нашего руководителя – Лавровой Анастасии. Она всегда была готова нам помочь со всеми возникающими трудностями. Кроме того, как только началось формирование команд, она сразу же сделала телеграм-чат, где было выложено техническое задание. Таким образом на начальном этапе разработки создавалось понимание будущих задач и примерных сроков.

# **4 Анализ собственной работы**

За время выполнения курсового проекта я на базовом уровне изучил множество технологий, с которыми до этого не был знаком: Django, Django Rest Framework, Swagger, Docker, PostgreSQL и некоторые другие. Кроме того, был получен очень ценный опыт совместной разработки внутри собственной команды и взаимодействия с другими командами, которые напрямую не связаны с разрабатываемым приложением, но которым необходимо впоследствии с ним взаимодействовать. Также стоит отдельно упомянуть работу с Git и Github. Раньше мне были знакомы только основные команды и возможности этих сервисов, а теперь есть небольшой опыт совместного использования с другими разработчики.

Работа происходила планомерно, так как руководитель устраивала консультации, регулярно ставила новые задачи и просила в срок завершать уже поставленные. Иногда стабильной работе мешал небольшой опыт работы с используемой на проекте технологией, так как приходилось изучать много новой информации и сразу же внедрять найденное решение. Отсутствие опыта также вызывало основные трудности. Иногда на изучение какой-то не очень сложной части фреймворка уходило много времени, которое можно было бы использовать для реализации другого функционала или тестирования приложения.

Несмотря на возникающие трудности, выполнение проекта принесло неоценимый опыт. Я получил опыт работы с большим количеством новой документации, а также опыт получения задач от руководителя и последующего расставления приоритетов для успешного выполнения.

# **ЗАКЛЮЧЕНИЕ**

Цель проекта была достигнута: команда создала серверную часть сервиса для управления медицинскими данными. Все поставленные задачи выполнены, кроме одной части реализации запросов для медицинских учреждений. Данную задачу пришлось отложить в связи с нехваткой времени. Нами был разработан полноценный веб-сайт, предоставляющий все функции, заявленные в техническом задании.

Мой вклад в проект заключался в выполнении поставленных задач, а именно в разработке и поддержке бэкенд приложения. Я инициировал новый Python-проект с необходимой структурой, реализовал ключевые эндпоинты с тестовыми ответами. Позже я добавил логику работы эндпоинтов, включая взаимодействие с базой данных и отправку ответов клиенту. По мере работы над проектом приходилось изучать множество новых инструментов разработки, которые были необходимы для выполнения задач.

Таким образом, у нашей команды получилось реализовать серверное приложение, а также поддержать интеграцию с клиентской частью. Все поставленные цели и задачи проекта были выполнены.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1 Документация Django. Все, что нужно знать о Django. – [https://django.fun/docs/django/5.0](https://django.fun/docs/django/5.0/)

2 Документация DRF. Django Rest Framework documentation – [https://www.django-rest-framework.org](https://www.django-rest-framework.org/)

3 Информации о библиотеке для написания документации в коде. drf-spectacular documentation – [https://drf-spectacular.readthedocs.io/en/latest](https://drf-spectacular.readthedocs.io/en/latest/)

4 Уроки по использованию Docker. Docker guides – [https://docs.docker.com/guides](https://docs.docker.com/guides/)

# **ПРИЛОЖЕНИЕ**

Техническое задание

Сервис для управления медицинскими данными на основе блокчейн

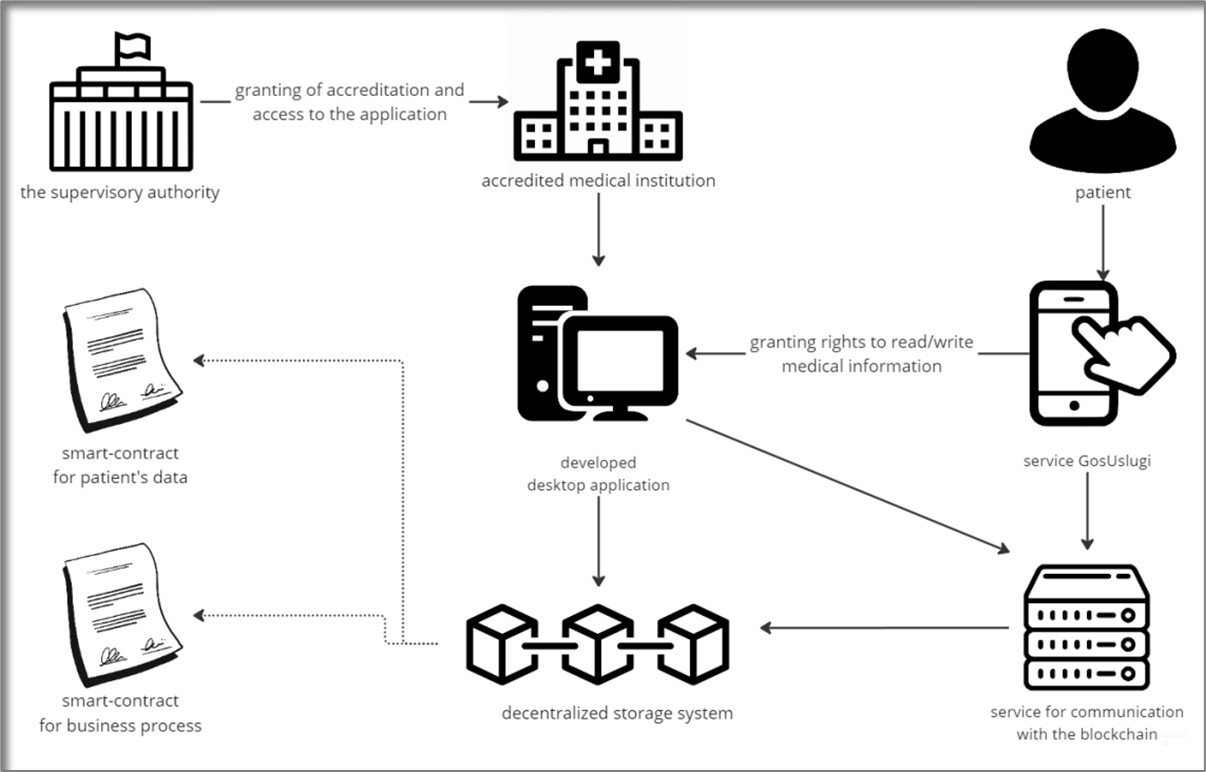
Клиентская часть: Алмазова Л.

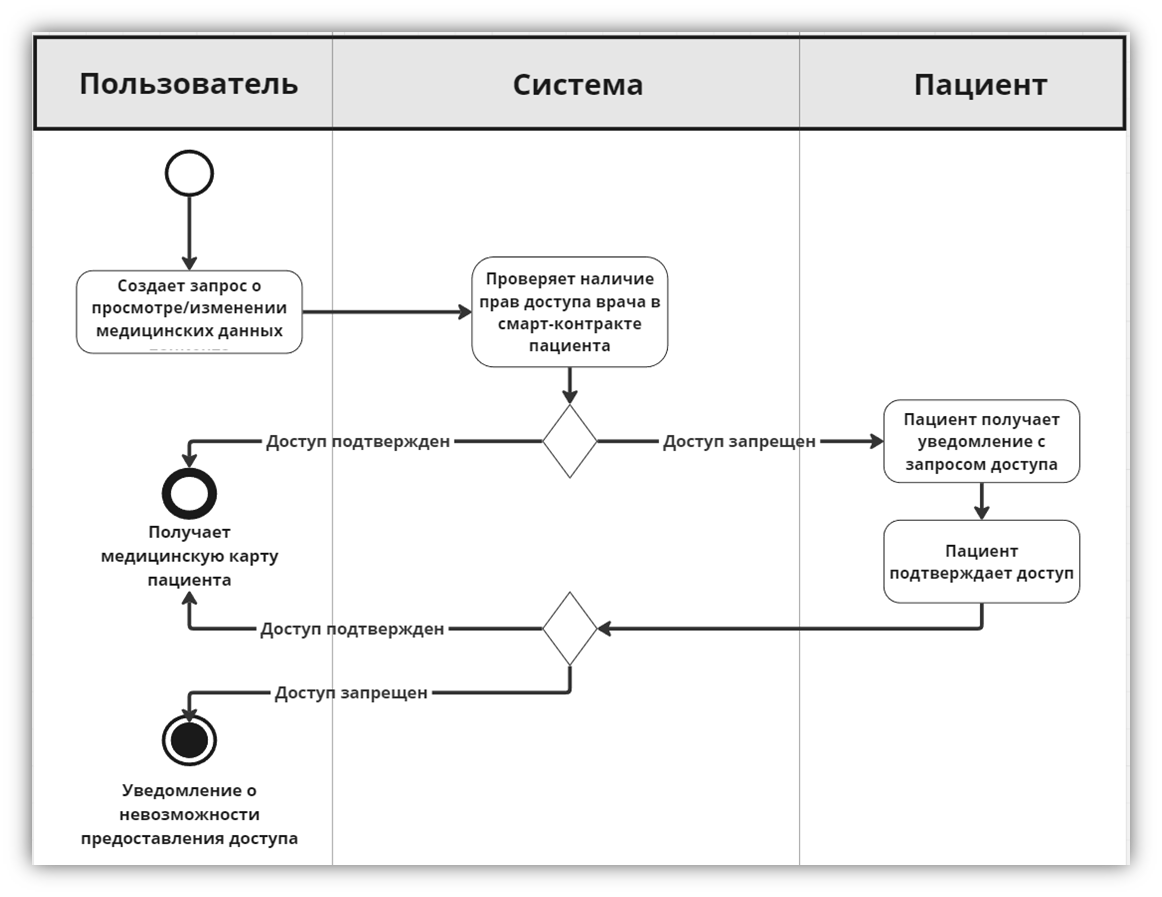
Серверная часть: Лаврова А.К.

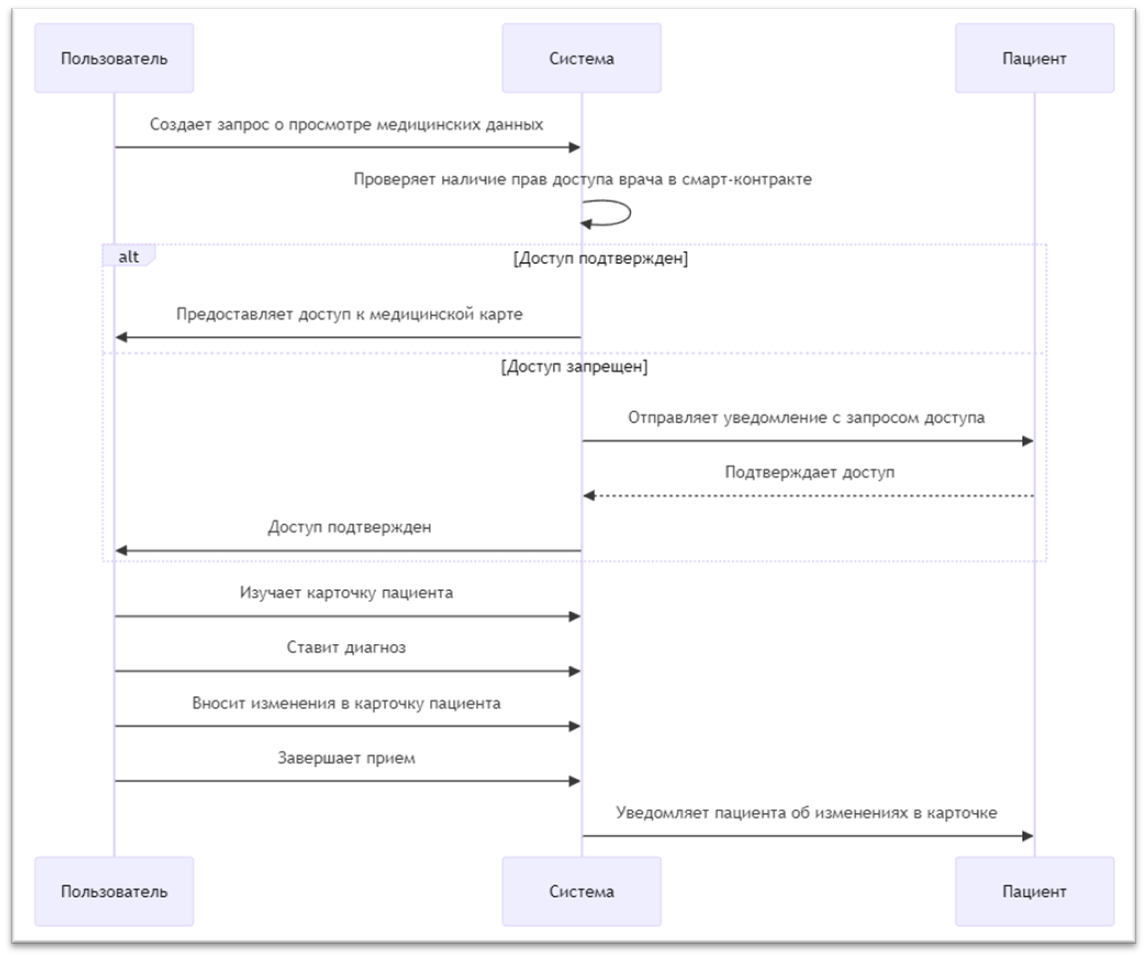
# 1. Общее описание проекта

В настоящее время в сфере хранения медицинских данных существует ряд проблем:

1. **Разрозненное хранение данных**В большинстве медицинских учреждений данные пациентов хранятся в локальных базах данных. Каждый пациент имеет отдельные карточки в разных учреждениях, что создаёт серьёзные сложности при попытке собрать полную медицинскую историю. Эта разрозненность препятствует быстрой и точной диагностике, а также снижает качество медицинского обслуживания.
2. **Отсутствие прозрачности доступа к данным**В текущей системе пациенты не имеют контроля над тем, кто и когда просматривает их медицинские данные. Часто такие доступы происходят без ведома пациента, что ставит под угрозу его право на приватность и защиту данных. К тому же пациенты не имеют уверенности в подлинности внесённых записей, так как нет прозрачного механизма отслеживания изменений.
3. **Централизованное хранение с риском утечек данных**В большинстве медучреждений данные хранятся на централизованных серверах, которые часто подвергаются риску взломов и утечек. Учитывая, что многие учреждения сталкиваются с ограничениями по бюджету, уделять внимания мерам информационной безопасности оказывается недостаточным. Это создаёт условия для возможных утечек конфиденциальной информации, о которых пациент зачастую даже не будет знать.

Общая идея решения выглядит следующим образом:





# Требования

## План MVP

Реализовать следующие сценарии:

1. Пациент заходит на свою страницу (временная замена госуслугам), видит запрос на доступ со стороны врача А. Пользователь подтверждает/отклоняет этот запрос.
2. Пациент заходит на свою страницу и видит перечень врачей, которым выдан доступ к его странице
3. Врач ищет пациента в базе и запрашивает доступ на изменение его данных.
4. Врач ознакамливается с карточкой пациента.
5. Врач вносит запись в карточку пациента.

## Требования к стеку

* Дизайн: Figma
* Front-end: React.JS
* Back-end: Python (Django)
* Блокчейн: Solidity (Hardhat, Ethers.js)

## Полезные ссылки

* Репозиторий GitHub: <https://github.com/EgoInc/MedChainMVP>

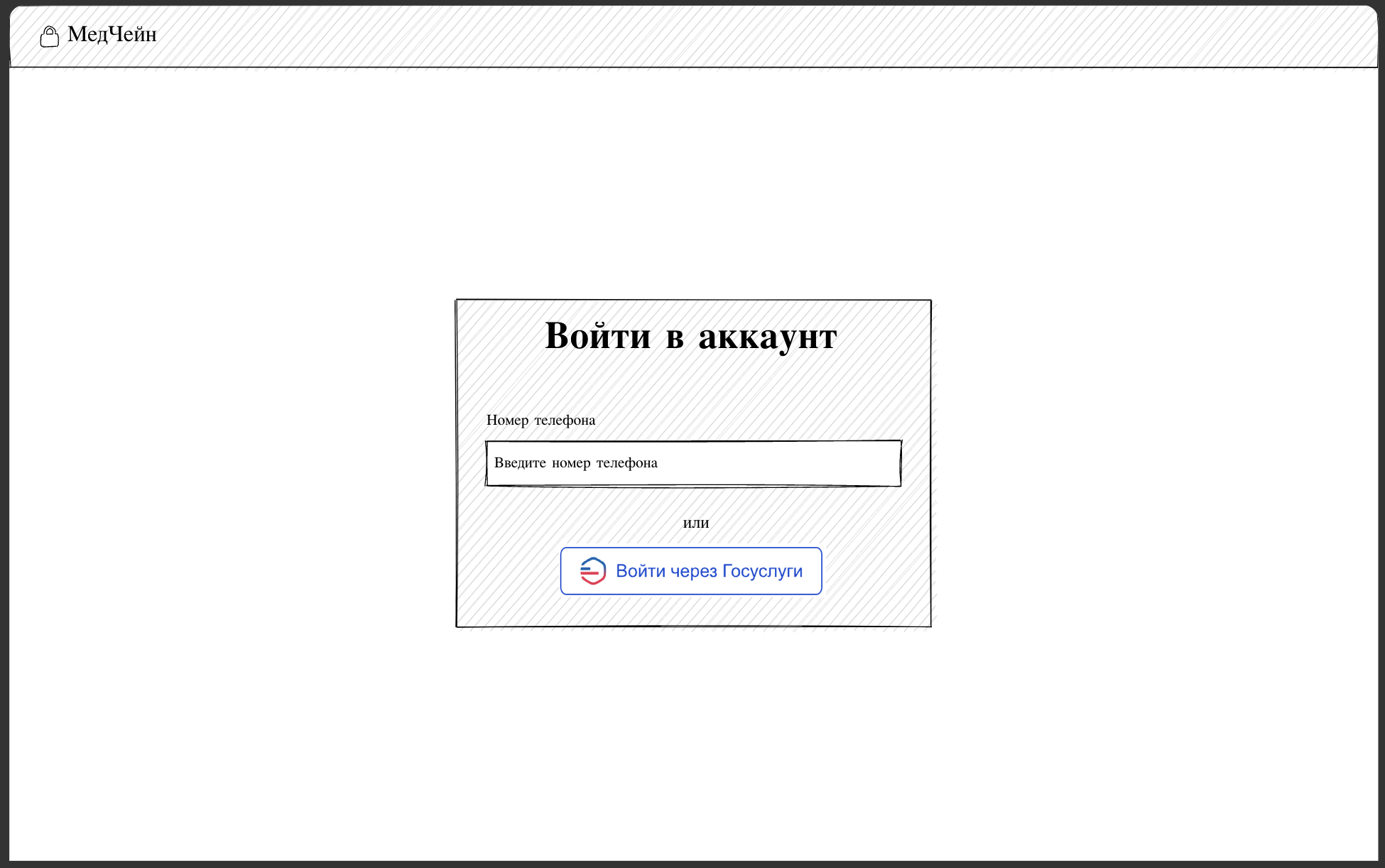
# 

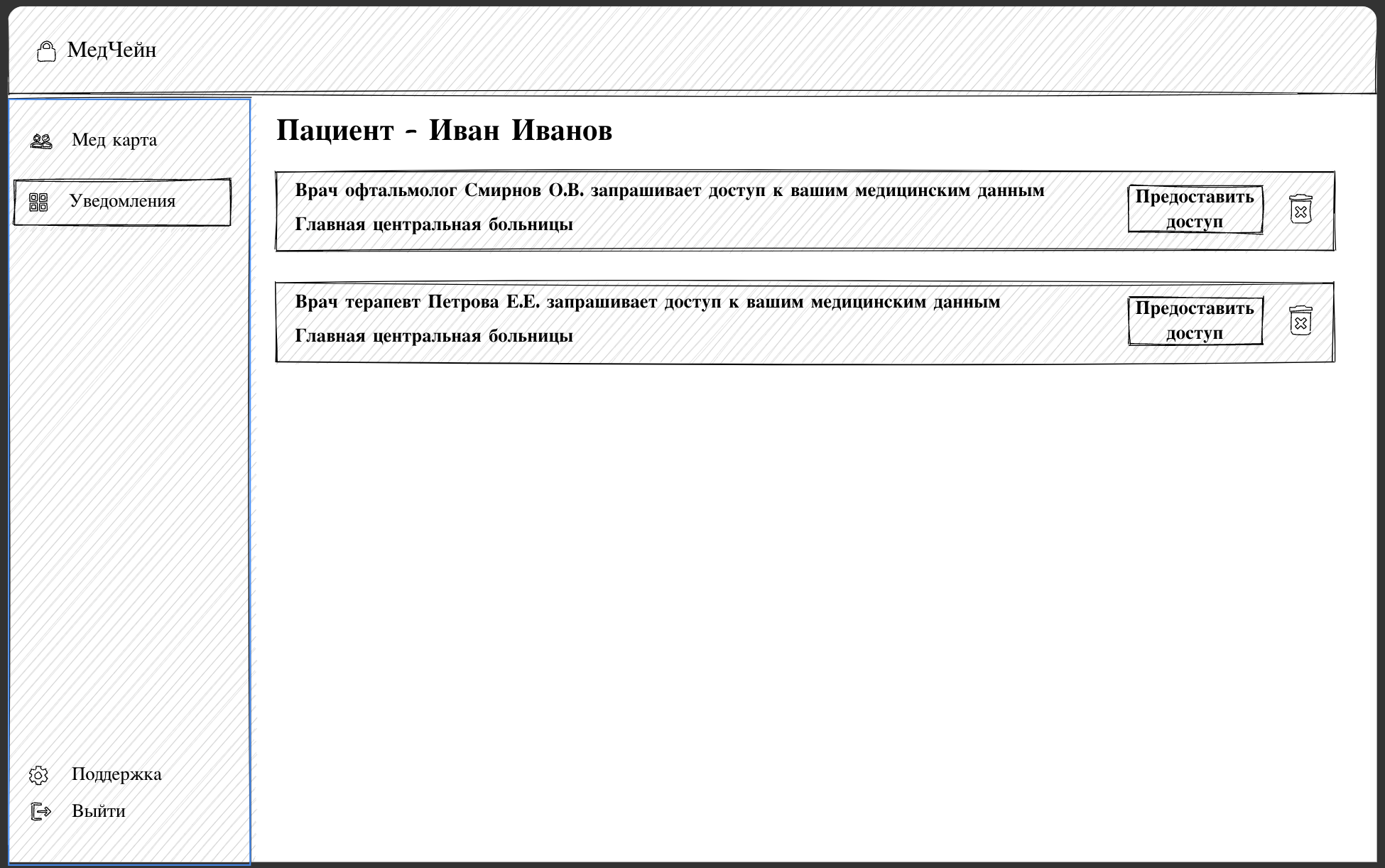
# Задачи дизайн

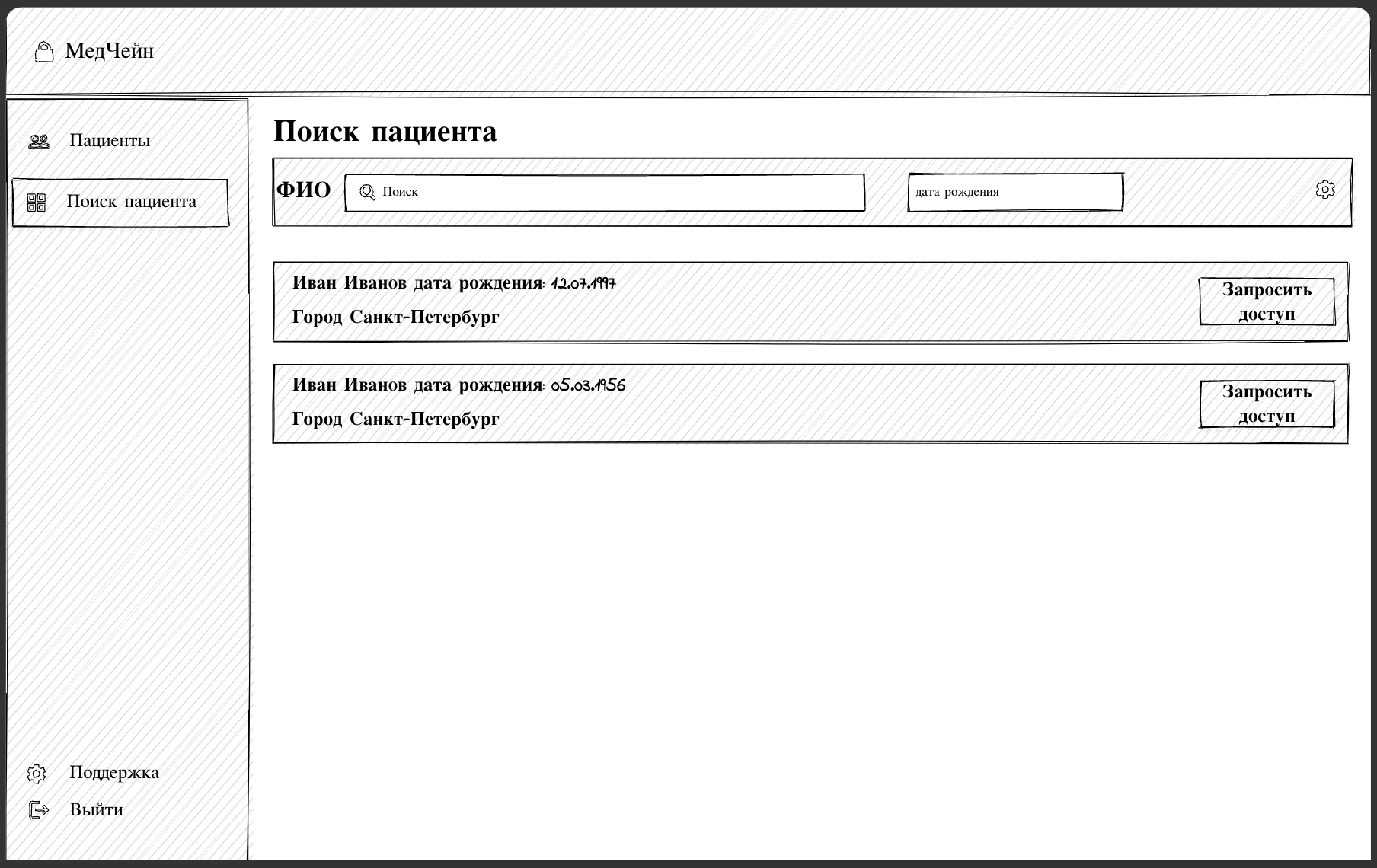
* Проработать цветовую палитру и стилистику
* Создать макеты страниц в figma
* Создать макеты адаптива под мобильные устройства

Основные сценарии, которые хотим реализовать, описаны в плане MVP. Откуда следует, что нам нужны следующие страницы:

* страница входа, регистрации
* страница пациента (от лица пациента), где отображается список врачей, которые уже получили доступ к странице и новые запросы
* страница пациента (от лица врача) на которой представлена история мед записей и есть возможность добавить новую
* список пациентов (от лица врача) с возможностью отправить запрос доступа к их страницам







# Задачи Front-end

* По макетам сверстать страницы
* Учесть адаптив под мобильные устройства
* Настроить взаимодействие с беком

Страницы, которые хотим получить, описаны чуть выше в задачах дизайна.

В качестве основного фреймворка используем React, можно ознакомиться с документацией [https://ru.legacy.reactjs.org](https://ru.legacy.reactjs.org/)

Перед началом работы стоит завести аккаунт в git, работать будем в репозитории <https://github.com/EgoInc/MedChainMVP> и скачать среду разработки

# Задачи Back-end

## 1. Спроектировать архитектуру БД

Создать ER-диаграммы, чтобы отобразить структуру данных, основные сущности (таблицы) и связи между ними.

### **Данные для хранения на бэкенде:**

#### Информация о пациенте (базовые данные)

* + **Цель**: Хранение основных данных для идентификации пациента и связи с его смарт-контрактом в блокчейне.
  + **Поля**:
    - patient\_id: уникальный идентификатор пациента в системе.
    - name: полное имя пациента.
    - date\_of\_birth: дата рождения пациента.
    - contract\_address: адрес смарт-контракта пациента в блокчейне.

#### Информация о враче

* + **Цель**: Хранение данных для идентификации врача и подтверждения его прав доступа, включая его публичный ключ для верификации.
  + **Поля**:
    - doctor\_id: уникальный идентификатор врача в системе.
    - name: полное имя врача.
    - organization\_id: идентификатор медицинского учреждения, к которому привязан врач.
    - public\_key: публичный ключ врача, который используется для проверки его подписи и аутентификации при доступе к данным пациента.

#### Информация о медицинских учреждениях

* + **Цель**: Хранение базовой информации о медучреждениях, к которым привязаны врачи.
  + **Поля**:
    - organization\_id: уникальный идентификатор медучреждения.
    - name: название учреждения.
    - address: адрес учреждения.
    - contact\_info: контактная информация (телефон, электронная почта).

#### Запросы на доступ к данным пациента

* + **Цель**: Отслеживание и хранение запросов от врачей на доступ к данным пациента. Само подтверждение запроса будет происходить через блокчейн, но бэкенд будет помогать управлять статусами запросов.
  + **Поля**:
    - request\_id: уникальный идентификатор запроса.
    - doctor\_id: ID врача, запрашивающего доступ.
    - patient\_id: ID пациента, к которому запрашивается доступ.
    - status: статус запроса (ожидание, подтверждено, отклонено).
    - request\_date: дата и время создания запроса.

#### Журнал действий врачей

* + **Цель**: Логирование действий врачей для внутреннего аудита и безопасности, например, для отслеживания успешных запросов и записей.
  + **Поля**:
    - log\_id: уникальный идентификатор записи лога.
    - doctor\_id: ID врача, выполняющего действие.
    - patient\_id: ID пациента, к которому относится действие.
    - action\_type: тип действия (запрос доступа, изучение медкарты, изменение медкарты).
    - action\_date: дата и время действия.

## 2. Разработать схемы API-запросов

С помощью Swagger создать прототип запросов. Почитать можно например: <https://medium.com/django-unleashed/a-beginner-guide-to-implement-swagger-documentation-with-django-0de05fbfae3f>   
Прототип запросов на Swagger поможет быстрее наладить интеграцию фронтенда с бэкендом. Для доступа к документации API по адресу {serverAddress}/docs настройте URL и конфигурации Swagger в проекте Django.

### **Запросы пациентов**

#### Запрос на получение списка запросов доступа

**Описание**: Позволяет пациенту просмотреть все текущие запросы на доступ к его данным.

**Схема эндпоинта**: /patient/{patient\_id}/access-requests

**Тип запроса**: GET

**Входные данные**:  
{

"patient\_id": "ID пациента, для которого запрашиваются запросы на доступ. Тип: integer"

}

**Выходные данные**:  
  
[

{

"request\_id": "ID запроса на доступ. Тип: integer",

"doctor": "ФИО врача, который запрашивает доступ. Тип: string",

"status": "Текущий статус запроса (ожидание, подтверждено, отклонено). Тип: string",

"request\_date": "Дата и время создания запроса. Тип: string (ISO 8601)"

}

]

Заглушка:  
[

{

"request\_id": 0",

"doctor": "Иванов Иван Иванович",

"status": "подтверждено",

"request\_date": "2024-06-15T13:00:27+03:00"

},

{

"request\_id": "02",

"doctor": "Петров Петр Петрович",

"status": "отклонено",

"request\_date": "2024-11-04T10:15:27+03:00"

},

{

"request\_id": "03",

"doctor": "Сергеев Сергей Сергеевич",  
"status": "ожидание",  
"request\_date": "2024-11-05T14:48:27+03:00"

}

]

#### Запрос на подтверждение или отклонение доступа

**Описание**: Позволяет пациенту подтвердить или отклонить запрос на доступ к его данным.

**Схема эндпоинта**: /patient/{patient\_id}/access-request/{request\_id}/respond

**Тип запроса**: POST

**Входные данные**:  
{

"patient\_id": "ID пациента, который обрабатывает запрос. Тип: integer",

"request\_id": "ID запроса, который нужно подтвердить или отклонить. Тип: integer",

"approve": "Ответ пациента на запрос (подтвердить - true, отклонить - false). Тип: boolean"

}

**Выходные данные**:  
{

"message": "Результат операции (Запрос подтвержден или Запрос отклонен). Тип: string"

}

Заглушка:

{

"message": "Запрос подтвержден"

}

#### Запрос на получение списка врачей с доступом

**Описание**: Возвращает перечень врачей, которым пациент предоставил доступ к своим данным.

**Схема эндпоинта**: /patient/{patient\_id}/authorized-doctors

**Тип запроса**: GET

**Входные данные**:  
{

"patient\_id": "ID пациента, для которого запрашивается список врачей с доступом. Тип: integer"

}

**Выходные данные**:  
[

{

"doctor\_id": "ID врача, имеющего доступ. Тип: integer",

"doctor\_name": "Полное имя врача. Тип: string",

"organization\_id": "ID организации, к которой принадлежит врач. Тип: integer",

"organization\_name": "Название организации, к которой принадлежит врач. Тип: string",

"access\_date": "Дата и время предоставления доступа. Тип: string (ISO 8601)"

}

]

Заглушка:

[

{

"doctor\_id": 1,

"doctor\_name": "Иванов Иван Иванович",

"organization\_id": 1,

"organization\_name": "Поликлиника №1",

"access\_date": "2024-06-15T13:00:27+03:00"

},

{

"doctor\_id": 2,

"doctor\_name": "Петров Петр Петрович",

"organization\_id": 25,

"organization\_name": "Санкт-Петербургская Клиническая Больница Российской Академии Наук",

"access\_date": "2024-10-20T13:00:27+03:00"

}

]

#### Запрос на добавление пациента

**Описание**: Позволяет создать запись о новом пациенте в системе и сохранить основные данные для связи с его смарт-контрактом в блокчейне.

**Схема эндпоинта**: /admin/add-patient

**Тип запроса**: POST

**Входные данные**:  
  
{

"name": "Полное имя пациента. Тип: string",

"date\_of\_birth": "Дата рождения пациента. Тип: string (ISO 8601)",

"contract\_address": "Адрес смарт-контракта пациента в блокчейне. Тип: string"

}

**Выходные данные**:  
{

"patient\_id": "ID созданного пациента. Тип: integer",

}

Заглушка:  
{

"patient\_id": 100,

}

### **Запросы для врачей**

#### Запрос на получение данных пациента

**Описание**: Позволяет врачу получить базовые данные о пациенте, чтобы подтвердить его личность перед запросом доступа.

**Схема эндпоинта**: /doctor/{doctor\_id}/patient/{patient\_id}

**Тип запроса**: GET

**Входные данные**:  
{

"doctor\_id": "ID врача, запрашивающего данные. Тип: integer",

"patient\_id": "ID пациента, чьи данные запрашиваются. Тип: integer"

}

**Выходные данные**:  
{

"patient\_id": "ID пациента. Тип: integer",

"name": "Полное имя пациента. Тип: string",

"date\_of\_birth": "Дата рождения пациента. Тип: string (ISO 8601)",

"contract\_address": "Адрес смарт-контракта пациента в блокчейне. 42 символа, начинается с 0x. Тип: string"

}

Заглушка:

{

"patient\_id": 1,

"name": "Иванов Иван Иванович",

"date\_of\_birth": "2000-06-01T00:00:00+03:00",

"contract\_address": "0x0000000000000000000000000000000000000000"

}

#### Запрос на доступ к данным пациента

**Описание**: Позволяет врачу отправить запрос на доступ к данным пациента.

**Схема эндпоинта**: /doctor/{doctor\_id}/request-access

**Тип запроса**: POST

**Входные данные**:  
{

"doctor\_id": "ID врача, запрашивающего доступ. Тип: integer",

"patient\_id": "ID пациента, к которому запрашивается доступ. Тип: integer"

}

**Выходные данные**:  
{

"request\_id": "ID созданного запроса на доступ. Тип: integer"

}

Заглушка:

{

"request\_id": 127

}

#### Поиск пациентов

**Описание**: Позволяет врачу найти пациентов по имени, фамилии или полному ФИО, а также с помощью дополнительных параметров, таких как дата рождения.

**Схема эндпоинта**: /doctor/{doctor\_id}/search-patients

**Тип запроса**: GET

**Входные данные**:  
{

"doctor\_id": "ID врача, выполняющего поиск. Тип: integer",

"name": "Имя пациента. Тип: string",

"date\_of\_birth": "Дата рождения пациента (необязательно, д

ля уточнения результатов). Тип: string (ISO 8601)"}

**Выходные данные**:  
[

{

"patient\_id": "ID найденного пациента. Тип: integer",

"name": "Полное имя пациента. Тип: string",

"date\_of\_birth": "Дата рождения пациента. Тип: string (ISO 8601)",

"contract\_address": "Адрес смарт-контракта пациента в блокчейне. 42 символа, начинается с 0x. Тип: string"

}

]

Заглушка:

[

{

"patient\_id": 1,

"name": "Иванов Иван Иванович",

"date\_of\_birth": "2000-06-01T00:00:00+03:00",

"contract\_address": "0x0000000000000000000000000000000000000000"

},

{

"patient\_id": 101,

"name": "Иванов Иван Алексеевич",

"date\_of\_birth": "2011-08-03T00:00:00+03:00",

"contract\_address": "0x1100000000000000000000000000000000000011"

}

]

#### Запрос "Мои пациенты"

**Описание**: Возвращает список пациентов, доступ к данным которых был подтвержден для данного врача.

**Схема эндпоинта**: /doctor/{doctor\_id}/my-patients

**Тип запроса**: GET

**Входные данные**:  
{

"doctor\_id": "ID врача, для которого запрашивается список пациентов с доступом. Тип: integer"

}

**Выходные данные**:  
[

{

"patient\_id": "ID пациента, к которому у врача есть доступ. Тип: integer",

"name": "Полное имя пациента. Тип: string",

"date\_of\_birth": "Дата рождения пациента. Тип: string (ISO 8601)",

"contract\_address": "Адрес смарт-контракта пациента в блокчейне. 42 символа, начинается с 0x. Тип: string",

"access\_granted\_date": "Дата и время, когда доступ был подтвержден. Тип: string (ISO 8601)"

}

]

Заглушка:

[

{

"patient\_id": 101,

"name": "Иванов Иван Иванович",

"date\_of\_birth": "2000-06-01T00:00:00+03:00",

"contract\_address": "Адрес смарт-контракта пациента в блокчейне. Тип: string",

"access\_granted\_date": "2024-12-24T12:00:00+03:00"

},

{

"patient\_id": 101,

"name": "Петров Петр Петрович",

"date\_of\_birth": "2000-06-01T00:00:00+03:00",

"contract\_address": 0x0000000000000000000000000000000000000000",

"access\_granted\_date": "2024-08-22T15:00:00+03:00"

},

]

### **Запрос для больниц**

#### Запрос на добавление больницы

**Описание**: Позволяет создать запись о новом медицинском учреждении (больнице) в системе.

**Схема эндпоинта**: /admin/add-hospital

**Тип запроса**: POST

**Входные данные**:  
{

"name": "Название медицинского учреждения. Тип: string",

"address": "Адрес учреждения. Тип: string",

"contact\_info": "Контактная информация (телефон, электронная почта). Тип: string"

}

**Выходные данные**:  
{

"organization\_id": "ID созданного учреждения. Тип: integer"

}

Заглушка:

{

"organization\_id": 100

}

#### Запрос на добавление нового врача

**Описание**: Позволяет медучреждению зарегистрировать нового врача в системе.

**Схема эндпоинта**: /organization/{organization\_id}/add-doctor

**Тип запроса**: POST

**Входные данные**:  
  
{

"organization\_id": "ID медицинского учреждения, регистрирующего врача. Тип: integer",

"doctor": "Полное имя врача. Тип: string",

"public\_key": "Публичный ключ врача для аутентификации. Тип: string"

}

**Выходные данные**:  
{

"doctor\_id": "ID созданного врача. Тип: integer"

}

Заглушка:

{

"doctor\_id": 1

}

#### Запрос на получение информации о врачах учреждения

**Описание**: Позволяет получить информацию о всех врачах, связанных с конкретным медицинским учреждением.

**Схема эндпоинта**: /organization/{organization\_id}/doctors

**Тип запроса**: GET

**Входные данные**:  
{

"organization\_id": "ID медицинского учреждения. Тип: integer"

}

**Выходные данные**:  
[

{

"doctor\_id": "ID врача. Тип: integer",

"doctor": "Полное имя врача. Тип: string",

"public\_key": "Публичный ключ врача. 42 символа, начинается с 0x. Тип: string"

}  
]

Заглушка:

[

{

"doctor\_id": 1,

"doctor": "Иванов Иван Иванович",

"public\_key": "0x0000000000000000000000000000000000000000"

},

{

"doctor\_id": 2,

"doctor": "Сергеев Сергей Сергеевич,

"public\_key": "0x0000000000000000000000000000000000000000"

}  
]

## 3. Настроить контейнеризацию и сборку приложения

### **Основные файлы и папки**

#### **1. manage.py**

* **Описание**: Основной файл для управления Django-проектом.
* **Назначение**:
  + Запуск сервера разработки (python manage.py runserver).
  + Применение миграций (python manage.py migrate).
  + Создание приложений (python manage.py startapp).
  + Выполнение команд и скриптов Django.

#### **2. medchain (основная папка проекта)**

* **Описание**: Папка с настройками проекта.
* **Содержит**:
  + **\_\_init\_\_.py**: Делает папку модулем Python. Этот файл часто пуст.
  + **asgi.py**: Настройки для ASGI (асинхронный серверный шлюзовый интерфейс). Используется для запуска асинхронных приложений.
  + **settings.py**: Основные настройки проекта (база данных, приложения, параметры конфигурации).
  + **urls.py**: Основные маршруты (роуты) проекта. Содержит ссылки на файлы маршрутов приложений.
  + **wsgi.py**: Настройки для WSGI (веб-серверный шлюзовый интерфейс). Используется для запуска приложения на продакшн-серверах.

#### **3. medchainapi (папка приложения)**

* **Описание**: Это приложение внутри вашего проекта Django. Django проект может включать несколько приложений.
* **Содержит**:
  + **\_\_init\_\_.py**: Делает папку модулем Python.
  + **admin.py**: Настройки для административной панели Django. Здесь вы можете регистрировать модели для их отображения в панели администратора.
  + **apps.py**: Настройки приложения. Определяет конфигурацию приложения.
  + **models.py**: Определение моделей базы данных. Каждая модель соответствует таблице в базе данных.
  + **migrations/**: Папка с файлами миграций, которые Django создает для управления изменениями структуры базы данных.
  + **serializers.py**: Файл для создания сериализаторов (в вашем случае используется для работы с API).
  + **tests.py**: Файл для написания тестов.
  + **views.py**: Основная бизнес-логика приложения. Определяет функции и классы, которые обрабатывают запросы.

### **Вспомогательные файлы**

#### **4. .gitignore**

* **Описание**: Файл для указания файлов и папок, которые не должны отслеживаться Git.
* **Назначение**: Исключает из репозитория такие файлы, как виртуальные окружения, миграции, логи, статические файлы и скомпилированные файлы Python.

#### **5. .dockerignore**

* **Описание**: Аналог .gitignore, но для Docker.
* **Назначение**: Указывает файлы и папки, которые не нужно копировать в контейнер Docker.

#### **6. docker-compose.yml**

* **Описание**: Конфигурационный файл Docker Compose.
* **Назначение**:
  + Описывает и координирует запуск нескольких сервисов (Django, PostgreSQL).
  + Автоматизирует запуск контейнеров.

#### **7. Dockerfile**

* **Описание**: Скрипт для сборки Docker-образа.
* **Назначение**: Определяет, как упаковать ваш Django-проект в Docker-образ.

#### **8. requirements.txt**

* **Описание**: Файл с зависимостями проекта.
* **Назначение**: Указывает пакеты Python и их версии, которые должны быть установлены для работы проекта.

#### **9. db.sqlite3**

* **Описание**: Файл SQLite-базы данных.
* **Назначение**: Содержит данные вашего проекта, такие как записи моделей, данные пользователей, логи и прочее.

#### **10. migrations/**

* **Описание**: Папка с миграциями (внутри каждого приложения).
* **Назначение**:
  + Миграции — это скрипты для обновления структуры базы данных (например, создание или изменение таблиц).
  + Автоматически создаются Django при изменении моделей.

## 4. Развертка на сервере

Настройте виртуальную машину или сервер и разверните приложение. Настройте веб-сервер (например, Nginx), подключите его к Django через Gunicorn или другой WSGI-сервер. На этом этапе проверьте работу миграций, подключите базу данных, кэш и фоновые задачи, если они требуются.

## Задачи блокчейн

## 0. Познакомиться с блокчейном

Необходимо разобраться с основными понятиями блокчейн-сферы:

* Общее:
  + Блокчейн
  + Узлы блокчейна
  + EVM-блокчейн
  + Транзакции
  + Газ
  + Смарт-контракты
  + Публичный и приватный ключ
* Для разработчиков блокчейнов:
  + Алгоритм консенсуса (понять разницу PoW, PoS)
  + Блоки в блокчейне
* Для разработчиком смарт-контрактов:
  + ABI смарт-контракта
  + Bytecode смарт-контракта
  + Криптокошелек

В этом могут помочь видео:

* **Что такое блокчейн**[What is a Blockchain? (Animated + Examples)](https://www.youtube.com/watch?v=kHybf1aC-jE)

*Там очень быстро расскажут про базовые термины, введут в курс дела*

* ***Что такое смарт-контракты***[*What are Smart Contracts in Crypto? (4 Examples + Animated)*](https://www.youtube.com/watch?v=pyaIppMhuic&list=PLHx4UicbtUoYPDWk2aUwZoVKMkdRKtKWe&index=11&pp=iAQB)

*Именно это мы и будем писать, поэтому рекомендую посмотреть, возможно что-то еще почитать и разобраться*

* ***Что такое газ в Ethereum***[*What is Ethereum Gas? (Examples + Easy Explanation)*](https://www.youtube.com/watch?v=3ehaSqwUZ0s&list=PLHx4UicbtUoYPDWk2aUwZoVKMkdRKtKWe&index=6&pp=iAQ)

*Это важный компонент в экосистеме EVM-блокчейном, поэтому тоже хорошо бы понять что это*

* **Публичные и приватные ключи. RSA**

[Asymmetric Encryption - Simply explained](https://youtu.be/AQDCe585Lnc?si=G_9IiLKXmPBc_qYK)

*Поможет разобраться в чем отличие приватных и публичных ключей и зачем вообще они нужны*

## 1. Запустить частный EVM-блокчейн

Развернуть частный EVM-блокчейн, адаптированный под нужды системы, т.е.:

* Пользователям не нужны реальные деньги чтоб осуществлять транзакции
* Может хранить много данных в рамках одного смарт-контракта
* Должен работать на слабых компьютерах и не требовать мощного железа, которого нет у больниц

## 2. Написать смарт-контракты

Написать два стандарта смарт-контрактов:

### **Смарт-контракт пациентов**

* Структуры данных:
  + MedicalRecord – структура для хранения данных о конкретной записи в истории болезни (дата, врач, диагноз, жалобы);
  + Patient – структура для хранения данных о пациенте и его медицинской истории.
* Функции:
  + addDoctor и removeDoctor - функции для управления списком авторизованных врачей, только владелец контракта (пациент) может добавлять или удалять врачей;
  + addMedicalRecord – функция для добавления новой записи в медицинскую историю пациента, доступна только авторизованным врачам;
  + getMedicalHistory – функция для получения списка записей в медицинской истории пациента, доступна только авторизованным врачам;
  + getPatientData – функция получения информации о пациенте (ФИО, дата рождения).

### **Смарт-контракт администратор:**

* Структуры данных:
  + patientContracts – ассоциативный массив (mapping), который хранит адрес смарт-контракта пациента для каждого пациента, ключом является адрес пациента, значением — адрес смарт-контракта;
  + allPatients – массив, содержащий адреса всех пациентов, этот массив используется для получения списка всех пациентов в системе.
* Функции:
  + createPatientContract – функция создает новый смарт-контракт пациента и его адрес сохраняется в patientContracts, адрес пациента также добавляется в allPatients;
  + getPatientContract – функция возвращает адрес смарт-контракта пациента по его адресу и позволяет другим пользователям системы (например, врачам) находить контракт пациента для доступа к его медицинской информации;
  + getAllPatients – функция возвращает массив адресов всех пациентов, может быть полезна для администраторов системы или для анализа данных.

## 3. Развернуть смарт-контракты в блокчейне

Развернуть смарт-контракты:

**А.** В локальной сети, т.ч. частном развернутом на шаге 1 блокчейне

**Б.** В публичном тестнете

## 4. Написать примеры взаимодействия с блокчейном для фронтенда

С помощью библиотеки ethers.js написать скрипты для взаимодействия с написанными смарт-контрактами, которые затем будут интегрированы в код фронтендеров