Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА АНАЛИЗА ДАННЫХ И ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Направление: 09.03.03 – Прикладная информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

СИСТЕМА ЗАПИСИ НА ПРИЁМ В МЕДИЦИНСКОЕ УЧРЕЖДЕНИЕ

Работа завершена:
Студент 4 курса
группы 09-951
«____»_____ 2023г. _____/Колесников Д.А.

Работа допущена к защите:
Научный руководитель
старший преподаватель
«____»_____ 2023г. _____/Еникеев И.А.

Заведующий кафедрой анализа данных
и технологий программирования
«____»_____ 2023г. _____/Бандеров В.В.

Казань — 2023

Содержание

Глоссарий

API – Описание способов взаимодействия одной компьютерной программы с другими [3].

Backend (Бэкенд, серверная часть) – Часть веб-приложения или сайта, которая запускается на сервере. Отвечает за обработку запросов от фронтенда, выполнение бизнес-логики, взаимодействие с базами данных

CRUD – акроним, обозначающий четыре базовые функции, используемые при работе с базами данных: создание (create), чтение (read), модификация (update), удаление (delete).

Frontend (Фронтенд, клиентская часть) – Часть веб-приложения или сайта, которую пользователь видит и с которой взаимодействует в своем браузере. Отвечает за представление данных пользователю и обработку пользовательских взаимодействий.

HTTP – протокол, используемый для передачи гипертекстовых документов в Интернете.

JSON – текстовый формат обмена данными, основанный на JavaScript. Представляет собой набор пар ”ключ: значение”.

URL – строка, используемая для идентификации ресурса в Интернете.

Алерт – пользовательское уведомление, информирующее о каком-либо событии.

Аннотация – метаданные, позволяющие декларативно влиять на код.

База данных – структурированный набор данных для простого взаимодействия пользователя с ними.

Бизнес-логика – часть программы, определяющая правила обработки данных. Описывает процессы организации в виде кода.

Класс – шаблон для создания объектов. Определяет набор полей и методов, доступных для объекта.

Конечная точка – url, на который приложения отправляют запросы для взаимодействия с веб-сервисами.

Массив – структура данных, представляющая набор идентифицируемых элементов.

Метод – Блок кода, выполняющий конкретную задачу. Используется для организации кода.

Много к одному – тип отношения в базе данных, когда несколько записей в одной таблице связаны с одной и той же записью в другой таблице.

Нормализация – Процесс организации данных в базе данных, нацеленный на повышение её гибкости и защищённости.

Объект – экземпляр класса, то есть созданная и действующая по правилам класса структура.

Один к одному – тип отношения в базе данных, когда одна запись в таблице связана не более чем с одной записью в другой таблице.

Окружение – набор условий и параметров, в которых выполняется программа. Может включать аппаратное и программное обеспечение, операционные системы, переменные окружения и другие компоненты.

Поле – элемент данных, который является частью структуры, как класс или запись в базе данных.

Разрешение экрана – количество пикселей, отображаемых на устройстве вывода изображения.

РКИБ – ГАУЗ «Республиканская клиническая инфекционная больница имени Агафонова».

Ресурс – в контексте API – единица данных или функциональности, которую можно получить через API, в иных случаях – любой элемент, который может использоваться программой (файлы, память и так далее).

СУБД – комплекс программ, предназначенный для создания, управления и обработки баз данных.

Сущность (Entity) – объект, представляющий запись в базе данных.

Технологический стек – конкретный набор технологий, используемых для разработки и запуска приложения.

Триггер – в контексте базы данных – специальная процедура, автоматически срабатывающая при заданных событиях.

Фреймворк – абстрактная платформа, предоставляющая общую структуру приложения для упрощения разработки.

Введение

Медицинской отрасли не хватает информатизации. Это устоявшаяся сфера, однако она не соответствует современности: никто не хочет стоять в очередях, вручную заполнять документы, потерять выданный рецепт. В 2018 году Правительство РФ создало национальный проект «Здравоохранение», направленный на улучшение медицины. Одна из задач проекта – перенос Минздравом в электронный формат части услуг: выдача рецептов, запись к врачу, подача заявления на полис, хранение медицинских документов [1].

На деле перевели услуги в электронный формат не везде. В России 30 000 медицинских учреждений и нужно много времени для внедрения системы в каждое из них. При этом нужно учитывать особенности каждого учреждения – то, что подойдёт больнице в Москве, может не подойти поликлинике из маленького города.

Одно из учреждений так и не перешедших в электронный формат – Республиканская клиническая инфекционная больница имени Агафонова (далее РКИБ). Сейчас в неё активно внедряются цифровые технологии, например – оплата услуг. Но некоторые элементы остаются прежними, в частности, запись на приём. С ней то нам и предстоит разобраться.

Цель: Информатизировать запись на приём в РКИБ

Задачи:

1. Изучить предметную область
2. Проанализировать существующие аналоги
3. Составить техническое задание
4. Спроектировать части будущей системы
5. Реализовать спроектированные части системы

Объект исследования: Информатизация в системе здравоохранения

Предмет исследования: Запись на приём в медицинское учреждение

Структура: В первой части работы анализируется предметная область и определяется необходимый функционал. Во второй главе будущая система проектируется, в третьей – реализуется. Заключительная, четвертая часть – введение готовой системы в эксплуатацию.

1. Предметная область

1.1. Основные сведения

РКИБ специализируется на помощи с инфекционными патологиями. Сюда приходят лечиться люди с хроническими заболеваниями и те, кто хочет исследоваться на наличие болезней. Другие поликлиники направляют пациентов в РКИБ, потому что здесь предоставляется большое количество услуг: рентген, диагностика и лечение разных болезней, проведение лабораторных исследований, содержание больных в стационаре.

В этой больнице нет живых очередей. Каждый пациент записывается на какое-то время и принимается только по записи. Нет ситуаций, когда приходят ”Только спросить”или когда требуется неотложная помощь. Этим РКИБ отличается от большинства других поликлиник.

Врачи сами задают время своего приёма. Они составляют расписание на 2 недели вперёд, но оно может и измениться, например, если врач заболел. Расписание передаётся в регистратуру, где пациентов и записывают на приём.

1.2. Существующие решения

1.2.1. Текущее решение проблемы

На момент начала работы в РКИБ нет никакой информатизированной системы записи на приём. На каждого врача и, иногда, услугу заведена отдельная тетрадь. Пример такой тетради приведён на Рисунке 1.

Пациент, приходя в больницу, иду в регистратуру. Там его вручную записывают в тетрадь на конкретное время к конкретному врачу, вписывают его данные в тетрадь. Далее в указанное время человек приходит к врачу. На первый взгляд схема проста и в ней сложно допустить ошибки, но если посмотреть внимательнее, то у неё много недостатков:

- **Отсутствие синхронизации.** Если пациент отменил запись, то изменение нужно сделать везде, где было упоминание о приёме.
- **Невозможность копирования.** Каждый врач должен несколько раз в день сверяться с записью к нему, потому что актуальная копия только в регистратуре. При составлении отчётов также требуется переносить сведения о приёме. То есть вручную переписывать сотни строк.

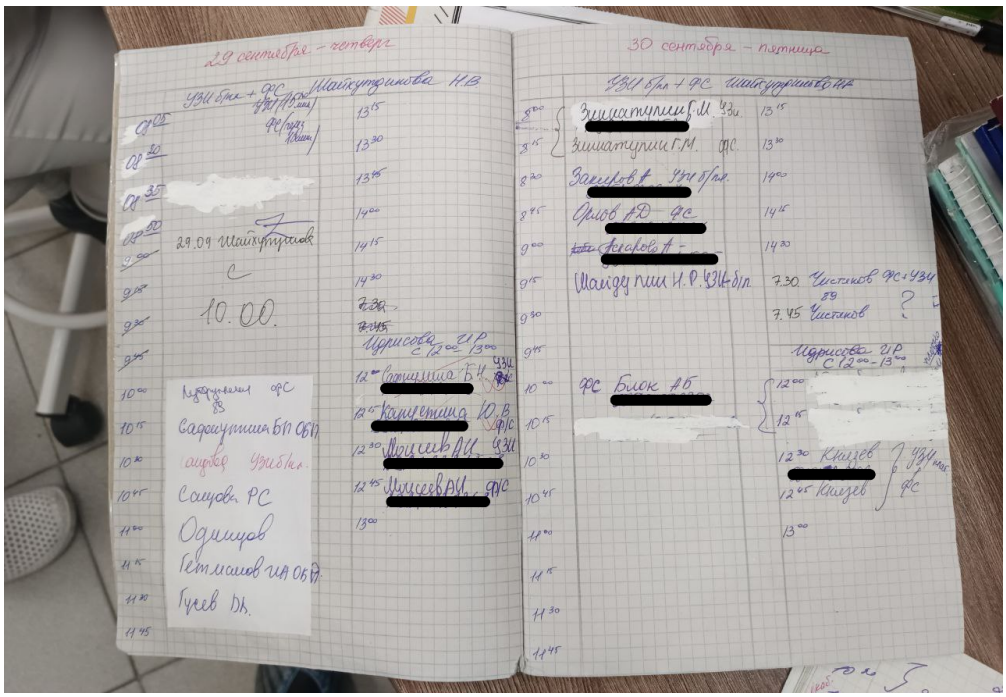

Рисунок 1. Пример заполненной тетради на УЗИ+ФС. Чёрным закрашены номера телефонов

– **Невозможность обработки.** Нужно постараться, чтобы найти данные о пациенте, который приходил месяц назад. Если тетрадь поменялась, то потребуется ещё и находить архивные записи. Восстановить таким образом графики врачей задача тоже не из простых.
– **Дублирование информации.** В РКИБ, по словам сотрудников, много регулярных пациентов. Если кто-то приходит каждый месяц, то в тетрадях он будет записан столько раз, сколько приходил. Редко когда у человека меняется, например, отчество или дата рождения. Эти данные записываются по нескольку раз, хотя это не имеет никакого смысла.
– **Невозможность качественного планирования расписания.** У врачей часто меняется график работы, а в тетради строки фиксированы и ограничены. При любом изменении в графике нужно вручную найти конфликтующие пункты и изменить их.
– **Хранение неактуальной информации.** Человек, поменявший место работы, навсегда в какой-то из тетрадей останется на прошлом.

Несмотря на недостатки, РКИБ использует этот способ. Всё из-за того, что он прост для понимания, доступен и к нему привыкли. Разработанная в результате система должна быть приближена к этому, чтобы под неё не нужно было долго переучиваться.

1.2.2. 1С:Медицина

«1С:Медицина» – решение от компании 1С. Продуманное, сделанное профессионалами, проверенное временем. Пример интерфейса на Рисунке 2. Это мог бы быть хороший вариант, но у него тоже есть недостатки:

– **Отсутствие интеграции с существующими системами.** Несмотря на то, что в РКИБ слабо информатизирована, что-то у неё уже есть. Переход на 1С означает отказ от практически всего разработано ранее, либо дополнительные затраты на интеграцию.
– **Избыточность.** Система нацелена на универсальность, поэтому в ней много функций, не нужных этой больнице, их можно было бы упростить. Сложность системы приводит к сложности её внедрения и обслуживания.
– **Цена и поддержка.** Система стоит не малых денег, а поддержка будет требовать дополнительных вложений. При этом, если верить отзывам из открытых источников, поддержка и документация не всегда могут оперативно помочь.

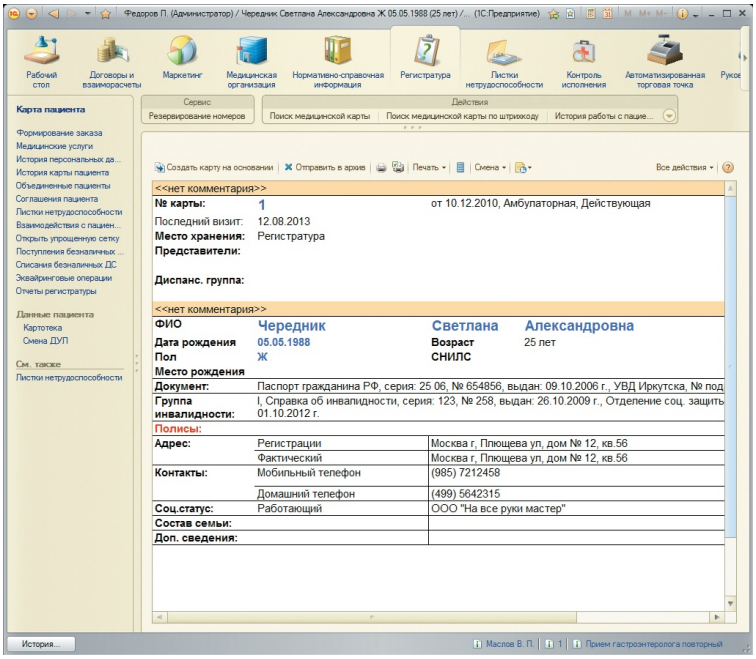

Рисунок 2. Интерфейс 1С:Медицина

1.2.3. Другая предлагавшаяся система

Разрабатываемая система – не первая, которую хотели внедрить в РКИБ. По словам сотрудников, им уже предлагали готовое решение. Проблема в

том, что система не была приспособлена к использованию в этой больнице. Она подходила обычной поликлинике, но РКИБ слишком сильно от них отличается. Здесь нет привычных участков. Обслуживаются не только жители ближайших районов, но и, во том числе, других регионов. Это помешало внедрить её в РКИБ.

Приспособленность к работе в условиях этой конкретной больницы – важное условие для запуска системы.

1.3. Техническое задание

В системе выделяются 4 роли:

– Пациент. Тот, кто имеет возможность попасть на приём.
– Доктор. Принимает пациентов согласно своему расписанию.
– Регистратор. Записывает и подтверждает запись к врачу.
– Администратор. Имеет доступ ко всем данным для их редактирования в случае возникновения проблем. Управляет редкоизменяемыми данные.

Каждая из них отличается требуемым функционалом. При этом каждый пользователь может иметь от 1 до 4 ролей, ведь ничто не мешает врачу записаться на приём к другому врачу. Рассмотрим функциональные требования к системе для каждой роли и нефункциональные требования, необходимые системе в целом.

1.3.1. Функциональные требования

Как уже было сказано в пункте 1.3, в системе есть 4 роли. Диаграмма вариантов использования для этих ролей представлена на Рисунке 3. Далее рассмотрим требования для каждой роли подробнее.

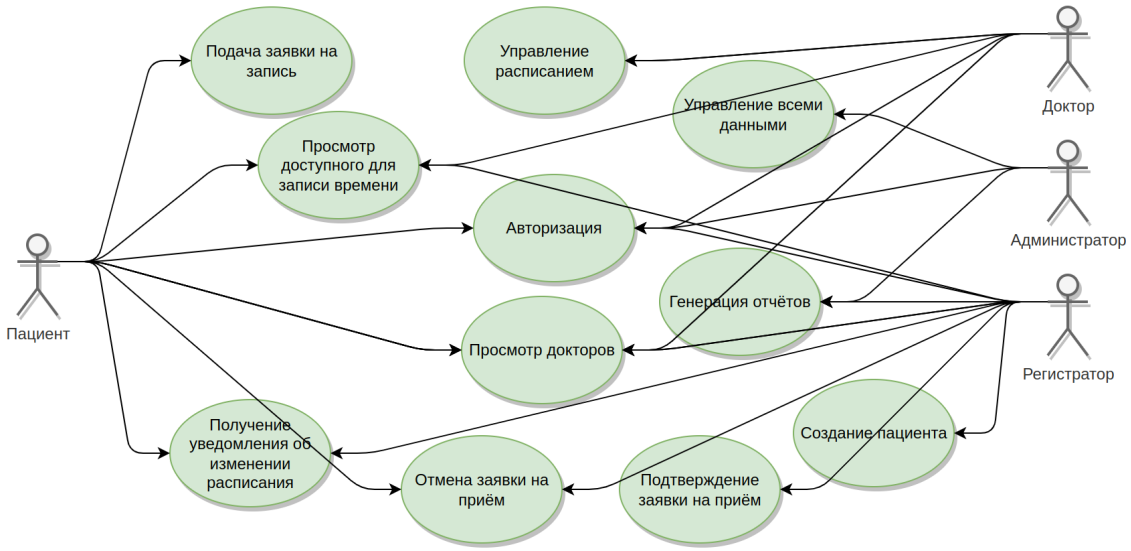

Рисунок 3. Диаграмма вариантов использования

1.3.1.1. Пациент

Функциональные требования для пациента:

– **Авторизация.** На первой странице приложения должна быть авторизация для пациентов. По логину и паролю пациент получает доступ к остальным функциям.
– **Просмотр докторов.** Возможность узнать какие врачи работают в учреждении и какие у них специализации. Нужно для того, чтобы даже если врач болеет, то была информация о нём.
– **Просмотр времени для записи.** Созданный врачом график записи должен отображаться для пользователя там же, где и ”Просмотр докторов но при выборе конкретного доктора.
– **Заявка на запись.** Выбрав время приёма, пользователь может нажать на кнопку ”Записаться”. Заявка отправляется регистратору и время для записи закрывается. Одновременно активных заявок на приём у пациента может быть не более двух.
– **Отмена заявки на запись.** Если по какой-то причине пациент не может

попасть на приём, то на странице своих визитов он может нажать на кнопку ”Отменить запись”. Сразу после этого время для приёма должно освободиться, а регистратору должно прийти об этом уведомление.

– **Просмотр своих визитов.** Пациент может просмотреть информацию о своих прошлых визитах и о текущих записях с указанием информации об обращении.
– **Уведомления об изменении расписания.** Если получилось так, что пациент не может быть принят в указанное время по причине изменения расписания, то ему должно прийти уведомление об этом.

1.3.1.2. Доктор

Функциональные требования для доктора:

– **Авторизация.** На первой странице приложения около панели входа должна быть кнопка для перехода на страницу входа как сотрудника. Отличается он возможностью указать роль в учреждении. После входа под конкретной ролью пользователя переносит на страницу этой роли.
– **Управление своим расписанием.** Возможность создавать новые элементы своего расписания, отмечать недоступными те, в которые врач не сможет совершить приём.
– **Просмотр списка записанных к нему.** Пациенты, которые записались на приём к врачу доступны для быстрого просмотра этому врачу. Если никто не записан на какое-то время, то об этом тоже должно быть написано. Если изменения в расписании касаются текущего рабочего дня, то об изменении в списке приёма дополнительно должно приходить уведомление.
– **Просмотр всех пациентов.** Должна быть возможность доктору просматривать всех пациентов.
– **Заполнение данных о приёме.** Основная информация о приёме не затрагивает текущую систему, однако, должна быть возможность указания диагноза и комментария по приёму.

1.3.1.3. Регистратор

Функциональные требования для регистратора:

– **Авторизация.** То же самое, что авторизация врача.
– **Создание/подтверждение записи пациента.** Регистратор может самостоятельно записать пациента на приём если он, например, пришёл в учреждение. Актуально для пожилых. Для этого регистратор указывает пациента и время приёма. Если же речь о подтверждении, то всё уже указано и требуется только его проверка.
– **Отмена записи на приём.** По каким-либо причинам пациент может не прийти на приём. Если он сообщил об этом заранее, то есть до приёма есть время, то у регистратора есть возможность отменить запись.
– **Уведомления об изменениях в расписании.** Если врач поменял своё расписание, а на эти элементы расписания были записаны пациенты, то регистратору приходит уведомление, что с это нужно обработать дополнительно.
– **Генерация отчётов.** Возможность генерации отчётов за указанный промежуток времени. Отчёты могут быть разные, например, количество приёма по врачам, количество фактического времени приёма для врачей.

1.3.1.4. Администратор

Функциональные требования для администратора:

– **Авторизация.** То же самое, что авторизация врача.
– **Управление всеми доступными данными системы.** Должна быть страница для каждого элемента, который вообще можно изменять. Только администратор может менять список услуг, добавлять сотрудников, взаимодействовать со статусами.
– **Генерация отчётов.** То же самое, что генерация отчётов для регистратора.

1.3.2. Нефункциональные требования

В этом разделе рассмотрим критерии, используемые для оценки работы системы – характеристики производительности и некоторые атрибуты качества.

1.3.2.1. Совместимость

Система должна корректно работать Chromium, Яндекс.Браузер, Edge версий, актуальных на начало 2023 года. Должна гарантироваться работа на экранах с разрешениями от 1024x768 до 1920x1440 для соотношения сторон 4:3, и с разрешениями 1280x720 до 1920x1080 для соотношения сторон 16:9. Указанные разрешения предполагают развёрнутость окно браузера на полный экран.

1.3.2.2. Безопасность

Использование системы должно происходить только через HTTPS. Пароли должны храниться в таком виде, чтобы из него было невозможно узнать изначальный пароль. Следовательно, не должно быть возможности восстановить пароль – только сбросить и создать новый.

1.3.2.3. Производительность

Система должна корректно работать при одновременном подключении как минимум 10 пользователей без значительного снижения производительности – не более 10 секунд на запрос при условии стабильного подключения интернета 100Мбит/с.

2. Проектирование

Разрабатываемая система – клиент-серверное приложение. Простейшая схема клиент-серверного приложения представлена на Рисунке 4.

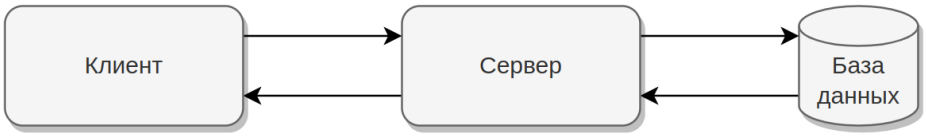

Рисунок 4. Схема клиент-серверного приложения

В этой главе займёмся проектированием указанных частей приложения, а в главе 3 реализуем спроектированные части.

2.1. База данных

Для реализации системы потребуется база данных. Её цель – упростить работу с данными для серверной логики. Появляется выбор – использовать реляционную или нереляционную базу данных.

Нереляционная база данных, с одной стороны, может повысить гибкость приложение, а с другой – усложняет работу с данными. В разрабатываемой системе выигрыш от использования нереляционной базы данных минимален, потому что даже в тетрадях всё записывалось в виде простых таблиц. Реляционная база данные в этом случае подойдёт больше – с ней гораздо проще работать и они лучше подойдут для, например, поиска, потому что список полей фиксирован.

Выделим список компонентов базы данных исходя из данных в пункте 1.2.1.

2.1.1. Основные компоненты

Для создания схемы данных нужно проанализировать задачу: Система должна позволять связывать заявки людей на приём и время приёма. Получается, что основных компонентов должно быть 2: **обращение** и **расписание**. К этому же можно прийти из пункта 1.2.1: Для каждого обращения в тетради используется отдельная строка. Для расписания врачей тоже использует-

ся одна строка на одно время приёма. Простейшая возможная база данных – простой перенос строк в поля таблиц, получим:

- **Обращение:** Содержит данные пациента: ФИО, занятость, адрес для идентификации пациента. Дополнительно содержит направление и диагноз.
- **Расписание:** Включает в себя ФИО врача, услугу, которую он оказывает, время и, опционально, данные об обратившемся человеке. Причём приём может быть сразу на 2 и более ячейки расписания. Связь «Много к одному».

2.1.2. Вспомогательные компоненты

Рассмотрим данные, которые можно было бы вынести в отдельные таблицы, для удобства работы с ними. Дополнительные компоненты можем разделить на 2 категории:

- Отражающие существующую логику. Они напрямую взяты из тетрадей и понятны обычному человеку.
- Упрощающие работу с данными. Нужны только для использования внутри системы.

2.1.2.1. Отражающие существующую логику

- **Пациент:** Хранит информацию о пациенте в общем. Нужна для хранения общей информации о человеке, которая не обязательно должна быть у, например, врача. Так совсем не обязательно врачу указывать свой личный телефон, если он не является одновременно пациентом.
- **Сотрудник:** Нужна только для определения кто из людей какую роль занимает в учреждении.
- **Учреждение:** Если пациент пришёл по направлению, то это направление должен был кто-то выдать.
- **Услуга:** Медицинские учреждения предоставляют конкретный список услуг. Таблица фиксирует эти услуги.

2.1.2.2. Упрощающие работу с данными

– **Человек:** Сущность, объединяющая пациента и сотрудника. В обеих таблицах есть ФИО. К тому же аккаунт должен быть и у тех, и у тех. Более того, один и тот же человек может быть и пациентом, и сотрудником. Объединение нормализует данные.
– **Статус обращения:** Вспомогательная таблица. Нужна для определения того, что делать с обращением. Например, чтобы отменённое обращение не показывалось в расписании врача.
– **Статус элемента расписания:** Тоже вспомогательная таблица, но для других целей. Например, если врач заболел, то записанные к нему люди будут отдельно обработаны регистратором.

2.1.3. Логическая схема данных

Исходя из пунктов 2.1.1 и 2.1.2 получаем схему данных, представленную на Рисунке 5.

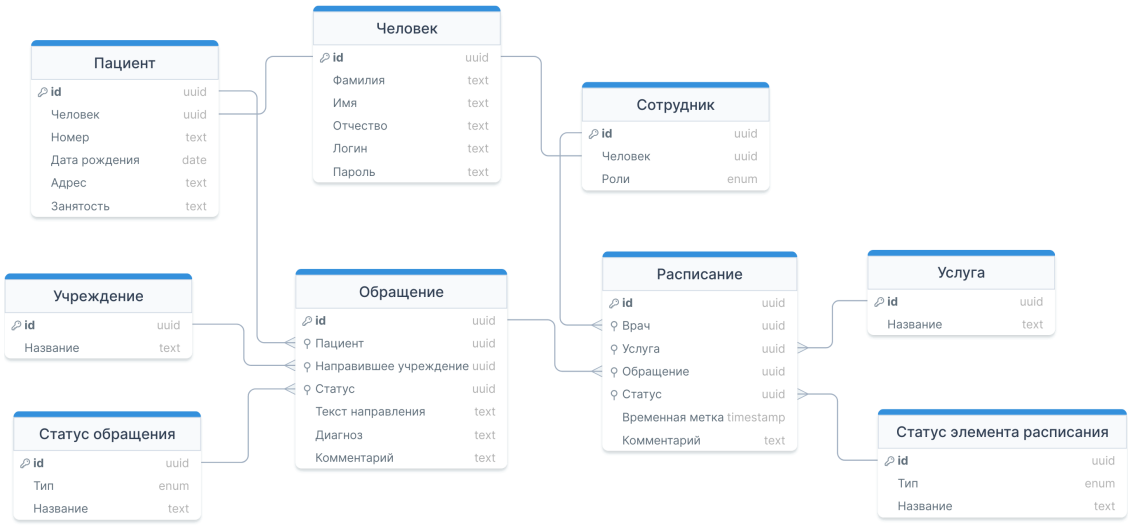

Рисунок 5. Логическая схема данных

2.2. Серверная часть

Клиентский код не должен напрямую обращаться к базе данных. Ему нужно промежуточное звено – сервер. Передавать данные клиент и сервер должны заранее определённым способом – API. Данные между клиентом и сервером передаются в формате, понятном обоим элементам. После преобразования данных в сущности, сервер работает напрямую с базой данных. Архитектура серверной части представлена на Рисунке 6

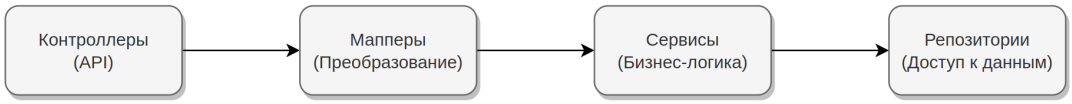

Рисунок 6. Общая архитектура серверной части

2.2.1. Выбор вида API

В зависимости от потребностей системы можно использовать разные виды API. Сравнение популярных видов API представлено в Таблице 1.

Название	Плюсы	Минусы
REST	– Простота – Универсальность – Поддержка CRUD – Не сохраняет состояние (легко масштабируется)	– Отсутствие строго стандарта – Низкая производительность при большом количестве запросов
SOAP	– Безопасность – Поддержка транзакций – Строгий стандарт	– Большая избыточность данных – Сложность использования и настройки
GraphQL	– Нет избыточных данных – Возможность сложных запросов – Сильная типизация	– Сложность разработки – Сложность масштабирования – Перегрузка из-за сложности запросов

gRPC	– Быстрое выполнение – Сильная типизация	– Сложность в настройке и использовании
JSON-RPC и XML-RPC	– Простота использования и разработки – Лёгкость интеграции	– Нет поддержки HTTP-методов – Затруднение масштабирования

Таблица 1. Популярные виды API

Для разработки был выбран REST. Его минусы в рамках проекта незначительны: в системе не будет большого количества запросов, а отсутствие строгого стандарта всего лишь делает его принципы рекомендациями. При этом REST удовлетворяет всем потребностям приложения – поддержка HTTP-методов, отсутствие сохранения состояния.

2.2.2. Авторизация

Название	Плюсы	Минусы
Basic Auth	– Хорошо работает с HTTPS – Не требует сессий – Прост в реализации	– Нет средств управления сессиями – Передача данных в незашифрованном виде
Digest Auth	– Безопасность из-за хеширования для передачи данных	– Сложность реализации – Нет управления сессиями
Token-based Auth (JWT, OAuth)	– Есть управление сессиями – Передача метаданных	– Сложная реализация и управление токенами – Проблемы при больших объёмах данных
Session-based Auth	– Управление сессиями – Сервер контролирует данные сессии	– Хранение и управление данными сессии – Плохо масштабируется

Таблица 2. Популярные виды API

Для данного проекта была выбрана Basic Auth, потому что он прост в

реализации. Его проблема безопасности решается использованием HTTPS и он хорошо подходит для проектов небольшого масштаба.

Заголовок авторизации – специальная строка, передающаяся вместе с запросом. Выглядеть строка должна так: «Authorization: Basic *токен*», где *токен* – Это зашифрованная с помощью Base64 строка вида «*логин*:*пароль*». При запуске приложения создаётся аккаунт по умолчанию, поэтому для всех методов, кроме установки пароля, потребуется авторизация.

2.2.3. Методы API

Каждый запрос на сервер в разрабатываемой системе характеризуется несколькими параметрами:

– URL
– Метод HTTP
– Заголовок авторизации
– Параметры в теле запроса, либо в URL

Зная эти данные мы можем дать нужный ответ. Теперь нужно определить – кто к каким конечным точкам должен иметь доступ. Для этого нужно составлять таблицы с указанием приведённых выше параметров. Поскольку полное расписывание API займёт очень много места, здесь будет приведён сокращённый вариант – без тел запроса. Подробный вариант можно найти в приложении.

Хорошей практикой считается разделение версий API и выделение его в URL. Поэтому перед каждой конечной точкой в реальных запросах к API дополнительно используется /api/v1, если не сказано иное. Спроектированные методы API представлены в Таблице 3

URL	HTTP метод	Краткое описание
/people/{id}	GET, PUT, DELETE	Получение, изменение, удаление общих данных о человеке
/people/me	PUT, GET	Получение, изменение данных о себе
/people	POST, GET	Получение списка всех пользователей
/patients/{id}	GET, PUT, DELETE	Получение, изменение, удаление пациента
/patients	POST, GET	Получение списка всех пациентов

/employees/{id}	GET, PUT, DELETE	Получение, изменение, удаление сотрудника
/employees	POST, GET	Получение списка всех сотрудников
/institutions/{id}	GET, PUT, DELETE	Получение, изменение, удаление учреждения
/institutions	POST, GET	Получение списка всех учреждений
/appointments/{id}	GET, PUT, DELETE	Получение, изменение, удаление обращения
/appointments	POST, GET	Получение списка всех обращений
/schedules/{id}	GET, PUT, DELETE	Получение, изменение, удаление элемента графика
/schedules	POST, GET	Получение списка всех элементов графика

Таблица 3. Методы API

Отдельно стоит выделить таблицы, к которым должен иметь доступ только администратор. Это такие таблицы, которые меняются очень редко и их изменение сильно влияет на всю систему. Например, изменение в таблице «Услуги». Перечень запросов, доступных только Администратору представлен в Таблице 4.

URL	HTTP метод	Краткое описание
/procedures/{id}	GET, PUT, DELETE	Получение, изменение, удаление услуги
/procedures	POST, GET	Добавление услуги, получение всех услуг
/appointmentStatuses/{id}	GET, PUT, DELETE	Получение, изменение, удаление статуса обращения
/appointmentStatuses	POST, GET	Добавление статуса обращения, получение всех статусов обращений
/scheduleStatuses/{id}	GET, PUT, DELETE	Получение, изменение, удаление статуса элемента графика

/scheduleStatuses	POST, GET	Добавление статуса элемента графика, получение всех статусов элементов графика

Таблица 4. Методы API только для Администратора

2.2.4. Обработка ошибок

Обработку ошибок можно разделить на 2 больших пункта:

– На стороне сервера. Рассматривается в этом пункте
– На стороне клиента. Будет рассматриваться в пункте 2.3.3

2.2.4.1. Обработка ошибок базы данных

Отдельно выделяются ошибки, связанные с базой данных. Это такие ошибки, которые может быть сложно, либо затратно обнаружить и обработать. Это связано с тем, что база данных – отдельная часть системы, с которой взаимодействует сервер. Есть несколько основных моментов, которые стоит затронуть:

– **Транзакционность.** База данных хранит данные – это её основная задача. Контроль корректности вводимых данных на её стороне возможен либо частичный (ограничиваясь, например, уникальностью полей), либо очень затратен в создании и поддержке (триггеры). Если данные записались в базу данных, то может быть проблемой удалить их. Особенно это актуально при одновременном создании множества объектов, когда может выполниться только часть запросов. В таких случаях помогает транзакционность – возможность отменить все изменения в рамках связанных запросов. В рамках разрабатываемой системы необходимо учитывать это на стороне сервера.
– **Определение конкретного места ошибки.** Стандартные ошибки, выдаваемые базой данных, не всегда информативны для пользователя, а иногда и для администратора. Так, например, нарушение связи один к одному, либо конфликт из-за уникальности – разные ошибки и их нужно по-разному идентифицировать.

2.2.4.2. Коды ошибок

Сопоставление каждой возможной ошибке конкретного кода может значительно ускорить процесс её обработки. Невозможно заранее знать все возможные причины ошибок – тогда бы их просто не было. Но вполне осуществимо выявление части программы, в которой произошла ошибка. Например, есть код, отвечающий за чтение Person из базы данных. Если мы пометим эту часть кода как вызывающую ошибку с кодом drp-01, то при её возникновении заранее будет известно её расположение. Конечно, это не отменяет вспомогательных сообщений и логирования – лишь помогает им. Список определённых кодов ошибок достаточно большой, их список приведён в Приложении {TODO}.

2.2.4.3. Единый формат ошибок

Стандартные форматы ошибок могут отличаться от фреймворка к фреймворку, что уменьшает гибкость приложения. Дополнительно, не каждый формат ошибок подходит для каждого приложения – в некоторых много избыточной информации, в некоторых её недостаточно. Поэтому для приложения был разработан единый формат ошибок, использующий JSON. Представлен на Листинге 1.

Листинг 1. Формат ошибок

```
{
    httpCode: "",
    errors: [
        {
            code: "",
            message: "",
            details: ""
        },
        ...
    ]
}
```

Разберём по частям:

– **httpCode.** У протокола http есть перечень поддерживаемых кодов состояния. Например, коды 2XX – коды успешного выполнения запроса,

4XX – ошибки клиента, 5XX – ошибки сервера. Это общепринятые заданные стандартом коды. Здесь потребуются только коды, отвечающие за ошибки. Явное указание кода в ответе упрощает работу с ошибкой при отладке.

- **errors.** Массив ошибок. Перечень того, что сервер в процессе обработки запроса посчитал некорректным. Стоит отметить, что перечисляются не все ошибки, а лишь те, до которых сервер обработал данные. Несколько элементов может быть указано только если ошибки одного типа, например, все касаются валидации полей.
  - **code.** Внутренний код ошибки системы, определённый в пункте 2.2.4.2.
  - **message.** Написанное доступным простому пользователю языком сообщение об ошибке, выводимое на экран.
  - **details.** Более развёрнутое сообщение об ошибке, содержащее максимум данных для отладки. Опциональное поле.

Использование этого стандарта гарантирует, что даже при переходе на другой фреймворк одной из частей приложения, изменения коснутся только её. Иначе в зависимости от фреймворка нужно было бы либо приводить вид одного к другому, либо переделывать и клиентскую, и серверную части.

2.3. Клиентская часть

Клиентская часть завершает список необходимых компонентов для клиент-серверного приложения. Это та часть, которую пользователь видит на экране. Сюда, по большей части, относится графический интерфейс и работа с ним. Сюда же входит работа с API сервера: формирование запросов, обработка ответов.

2.3.1. Архитектура клиентского приложения

Для разработки понадобятся различные компоненты, такие как диалоговые окна, кнопки, элементы для поиска элементов. Рассмотрим, какие компоненты нужны для работы системы:

Поля ввода. На первый взгляд, это один компонент. Но для удобной работы нужно детализировать его для разных нужд:

– Обычные поля, не требующие особого подхода. Например, комментарий к обращению может быть написан в свободной форме.
– Пароль. Требует сокрытия вводимых данных.
– Поля со строго определённой структурой. Необходимы для, например, наглядного ввода номера телефона

Кроме того, большинство полей в базе данных приложения требуют гарантии заполненности. Так услуга не имеет смысла без её названия. Нужна возможность визуальных подсказок для полей.

Алерты. Компоненты, информирующие пользователя о событиях. Когда на странице не могут загрузиться данные, либо произошла какая-то ещё ошибка, то нужно об этом сказать пользователю. Для этого хорошо подходит отдельный компонент.

Выпадающие списки. Заменяют поля ввода, когда список его значений заранее известен и достаточно мал.

Диалоговые окна. Это элементы, появляющиеся поверх основной страницы и служащие для совершения какого-то фиксированного действия. Например, отправка запроса на создание элемента или выбор элемента из множества на основе фильтров. В разделе 1.3 упоминалось, что в системе будет Администратор. Он должен иметь возможность редактировать все записи в базе данных. Для этого ему потребуется диалоговое окно на каждую из сущ-

ностей. Они же будут использоваться и для остальных ролей в случаях, когда доступ к сущностям есть.

2.3.2. Проектирование интерфейса пользователя

В пункте 1.3 выделено 4 роли: Пациент, доктор, регистратор, администратор. У каждого из них, в соответствии с пунктами 1.3.1.1 – 1.3.1.4 в системе свои возможности. Для этого необходимо разграничение интерфейса. Роли с указанием доступных им страниц представлены в Таблице 5

Роль	Доступные страницы
Пациент	– Просмотр расписания врача – Подача заявки на приём
Доктор	– Запись к себе – Создание своего расписания – Просмотр расписаний других врачей
Регистратор	– Расписания врачей – Пациенты
Администратор	– Страницы с доступом для редактирования для каждой сущности

Таблица 5. Роли системы и доступные им страницы

2.3.3. Обработка ошибок

Важная часть разработки клиентского приложения – правильная обработка ошибок. Она помогает предотвратить прерывание работы приложения, выявить и устранить причину ошибок. Продуманная система обработки ошибок важна для стабильности приложения.

Существуют разные виды ошибок, которые должно обрабатывать клиентская сторона, например:

– Обработка ошибок на уровне приложения
– Ошибки, генерируемые сервером. Например, запрос недоступного пользователю ресурса
– Некорректный ввод пользователя. Частично относится к прошлому пункту, но выделение позволяет сделать интерфейс удобнее

– Недоступность сервера

Подробнее рассмотрим важные элементы обработки ошибок

2.3.3.1. Валидация данных

Для предотвращения ошибок, связанных с неверными данными, в приложении необходимы механизмы проверки ввода – валидации. Это включает валидацию свойств компонентов, использование библиотек для валидации форм и других пользовательских данных посредством специальных заранее определённых правил.

2.3.3.2. Логирование ошибок

Для упрощения отладки и мониторинга работы приложения необходимо использовать систему логирования ошибок. Она включает вывод и сохранение информации об ошибках, отправку данных об ошибках на сервер. Важно одновременно и уведомлять пользователя об ошибках, и сохранять данные о них, поскольку это разные уровни взаимодействия. Первые события рассчитаны больше на пояснение того, что случилось, в понятной простому человеку форме, а вторые – на то, что их будет разбирать специалист, для чего нужно сохранять больше информации.

2.3.3.3. Обработка ошибок в пользовательском интерфейсе

Информативные сообщения об ошибках и уведомлениях в пользовательском интерфейсе улучшают пользовательский опыт. Основная реализация – всплывающие подсказки и уведомления. Стоит затронуть и необходимость визуального выделения ошибок, касающихся валидаций данных из пункта 2.3.3.1.

3. Реализация

После анализа предметной области и проектирования остаётся последний большой шаг – реализация. Потребуется написать код всех частей системы, сделать так, чтобы они работали вместе и без ошибок. Важным аспектом является тестирование, которое должно проводиться на протяжении всей разработки системы и даже до неё – подход «Разработка через тестирование»

3.1. Выбор технологий

Выбор инструментов реализации проводится параллельно с проектированием. Это начальный этап разработки, который может определять структуру системы. В пункте 2 указано, что частей системы 3: база данный, серверная и клиентские части. Рассмотрим реализацию каждой из них ниже. Отдельно стоит упомянуть технологию изоляции приложений, которую будет использовать каждая из частей – Docker.

3.1.1. Развёртывание

Docker – специальное средство, позволяющее создавать изолированное окружение для приложения. Это обеспечивает одинаковую работу на любом устройстве, где приложение будет запущено. Основной файл конфигурации – Dockerfile. Он содержит необходимые настройки для работы приложения. Пример такого файла конфигурации для серверной части приведён на Листинге 2.

Листинг 2. Пример Dockerfile для бекенда

```
1 FROM openjdk:17
2 ARG JAR_FILE=build/libs/*.jar
3 RUN mkdir /opt/rkib-back
4 COPY ${JAR_FILE} /opt/rkib-back/rkib-back.jar
5 ENTRYPOINT ["java","-jar","/opt/rkib-back/rkib-back.jar"]
```

С помощью этого файла конфигурации уже можно запускать приложение в изолированной среде. Есть ещё один инструмент, помогающий в работе с контейнерами – docker-compose. Он помогает в организации нескольких контейнеров, если они должны быть запущены вместе. Также позволяет

писать удобочитаемый файл конфигурации. Пример такого файла конфигурации – docker-compose.yml представлен на Листинге 3. Теперь для полного разворачивания приложения достаточно одной команды в директории с этим файлом – ”docker-compose up -d”.

Листинг 3. Пример файла конфигурации docker-compose.yml

```
version: "3.9"
services:
    rkib-back:
        image: zalimannard/rkib-back:latest
        container_name: rkib-back
        ports:
        - 8116:8116
        depends_on:
        - rkib-database
        environment:
        - DB_URL=*database url*
        - DB_USER=*database user username*
        - DB_PASSWORD=*database user password*
        - DDL_BEHAVIOUR=*what happens to the database*
        - ADMIN_USERNAME=*back's admin username*
        - ADMIN_PASSWORD=*back's admin password*
```

3.1.2. База данных

Изначальное условие для системы было – использование СУБД Oracle. Если разработанная система будет внедряться, то чтобы не было конфликтов в базе данных. Потому что разные СУБД имеют разные ограничения на имена, например, на длину и зарезервированные слова. Забегая вперёд – вся работа с базой данных будет через спецификацию JPA, то есть смена СУБД почти не потребует затрат.

3.1.3. Серверная часть

При выборе технологического стека для серверной части важно учитывать такие требования, как производительность, стабильность, безопасность, масштабируемость. Сравнение основных стеков представлено в Таблице 6.

Состав стека	Особенности
Java + Spring	– Высокая производительность и скорость выполнения – Хорошая интеграция с Oracle Database – Богатый набор библиотек для различных задач – Строгая статическая типизация для повышения надежности кода – Многопоточная модель, подходящая для CPU-интенсивных операций и масштабируемых систем
Python + Django	– Скорость разработки – Много функций ”из коробки” – Динамическая типизация – Меньшая производительность по сравнению с Java – Интеграция с Oracle может быть менее гибкой
JavaScript + Node.js	– JavaScript на сервере и клиенте, упрощает разработку – Однопоточная модель с асинхронной I/O, эффективной для обработки большого количества коротких запросов – Низкая производительность для CPU-интенсивных операций по сравнению с Java – Динамическая типизация и возможные проблемы безопасности

Таблица 6. Сравнение технологических стеков для серверной части

Для представления API в удобном виде будет использоваться **Swagger**. Он обеспечивает автоматическую генерацию документации по API с возможностью запуска команд из него же. Удобный инструмент для тестирования в

процессе разработки.

Для автотестов будет использоваться **RestAssured** для запросов на сервер, **JUnit5** для организации тестов, **AssertJ** для повышения читаемости тестов и **Allure** для визуализации результатов тестирования. Они помогают быстро находить проблемы в коде и увеличивают эффективность рефакторинга.

Lombok – библиотека для Java, позволяющая сосредотачиваться на написании бизнес-логики, а не на написании шаблонного кода. Так, практически пропадает необходимость написания геттеров, сеттеров, конструкторов и прочих компонентов. Код становится более чистым и легко поддерживаемым.

3.1.4. Клиентская часть

У клиентский части приложения большое количество нюансов, которые нужно учесть. Они связаны с тем, что этот код будет выполняться на разных компьютерах и нужно обеспечить корректную работу на них.

Основной язык разработки – JavaScript. TypeScript – разработанный компанией Microsoft язык, расширяющий возможности TypeScript: обладает статической типизацией, поддержкой полноценных классов. К тому же он совместим с JavaScript. Он повышает читаемость кода, его повторное использование и скорость поиска ошибок.

JavaScript отвечает за внутреннюю логику, за внешний вид отвечает HTML и CSS. Можно писать код используя только их, но такой подход хорош только для очень маленьких приложений. Так, работу на разных экранах придётся обеспечивать вручную. Как и следить за цветовой палитрой, стилями и анимацией. Гораздо удобнее использовать фреймворки, так как они представляют готовый набор стандартных элементов и логики. Популярных фреймворка 3:

- **React.** Гибкий, с большим количеством библиотек, но для полноценной разработки именно потребуется использование сторонних библиотек, что может быть затруднением.
- **Angular.** Большой фреймворк со всеми нужными компонентами из коробки, чёткая структура, но сложный для изучения и малопроизводительный.
- **Vue.js.** Гибкий, самодостаточный фреймворк с возможностью интегра-

ции со сторонними библиотеками. Прост в освоении, хорошая документация, но меньше материала по сравнению с другими.

С Vue.js я работал и это, пожалуй, лучший выбор для данной системы. Приложению не нужны супермасштабы и встроенных инструментов хватит для разработки приложения.

3.2. База данных

Как уже было сказано в пункте 3.1.2, база данных будет создана через JPA. Это спецификация, обеспечивающая переносимость данных между реляционными базами данных для приложений на Java. В файле конфигурации указано, как поступать с базой данных при каждом запуске приложения. Это указано в файле конфигурации, описанном в пункте 3.1.1. За это отвечает параметр DDL_BEHAVIOUR. Если требуемая база данных отличается от существующей, то в существующую вносятся необходимые изменения.

Таблицы и поля в базе данных создаются с помощью JPA, то есть объекту в базе данных соответствует объект в Java. Это позволяет работать с хранящимися данными так, будто они уже являются объектами в Java [4]. Это уже часть пункта 3.3, но его необходимо разобрать отдельно. Это некая прослойка между СУБД и сервером. Для примера возьмём сущность Patient, описанную как Пациент в логической схеме данных на Рисунке 5. Код представлен в Листинге 4. По порядку разберём написанное:

Строки 1-2 – аннотации JPA, отвечающие за создание таблицы в базе данных. @Entity говорит, что класс является объектом базы данных, @Table указывает конкретное название таблицы в базе данных.

Строки 3-7 – аннотации библиотеки Lombok. Создают геттеры и сеттеры (методы для доступа к полям), реализует шаблон строитель (позволяет гибко создавать объекты) и генерирует 2 конструктора – с требуемыми параметрами и всеми. Это нужно для инъекции зависимости, используемой в Spring и для работы шаблона строитель. Дополнительно опущены методы .equals() и hashcode, который сравнивает 2 экземпляра одного класса по заданным параметрам. В этом случае – сравнение по полю id, так как в базе данных гарантируется уникальность первичного ключа.

Все остальные таблицы делаются по аналогии, схема данных представлена на Рисунке 5.

Листинг 4. Пример реализации таблицы Patient через JPA

```
1  @Entity
2  @Table(name = "patients")
3  @Getter
4  @Setter
5  @Builder(toBuilder = true)
6  @RequiredArgsConstructor
7  @AllArgsConstructor
8  public class Patient {
9      @Id
10     @GeneratedValue(strategy = GenerationType.UUID)
11     @Column(name = "id")
12     private String id;
13     @OneToOne
14     @JoinColumn(name = "person_id", referencedColumnName = "
           id", nullable = false)
15     private Person person;
16
17     @Column(name = "birthdate")
18     LocalDate birthdate;
19     @Column(name = "phoneNumber", nullable = false)
20     private String phoneNumber;
21     @Column(name = "address")
22     private String address;
23     @Column(name = "occupation")
24     private String occupation;
25 }
```

В строках 9-12 объявляется первичный ключ и его тип генерации – UUID. Есть разные способы генерации первичного ключа – по последовательности, автоувеличением значения, какой-либо строкой, либо можно написать вообще свою функцию генерации. Одно из изначальных ограничений системы – использование строкового первичного ключа. UUID подходит под эту роль. Это способ генерации уникальной последовательности, из-за чего сложно угадать его соседние элементы, что более безопасно, чем другие способы. Распространённая практика делать поле id в каждой таблице.

В сторках 13-15 указано, что таблицы Person и Patient имеют связь «Один к одному», то есть одному пациенту соответствует один аккаунт. Содержится ссылка и то, что пациент гарантировано имеет аккаунт.

Строки 17-18 содержат информацию о дате рождения. Дата всегда сложный элемент, потому что, например, часовой пояс не должен влиять на дату рождения. У даты рождения не должно быть времени вообще. LocalDate позволяет гибко работать с датой.

Далее в строках 17-24 содержатся обычные текстовые поля, характерные для таблицы пациент. Указывается имя и то, что это обычное поле в базе данных.

Компонент Patient полностью покрывает все нюансы базы данных в создаваемой системе, кроме перечислений, которые будут рассмотрены в пункте 3.3. Все остальные таблицы делаются по аналогии с этой. При запуске приложения все таблицы автоматически создадутся и свяжутся. Всё это происходит через обычные запросы к базе данных, но автоматически, через обёртку JPA.

3.3. Серверная часть

Следующий этап – разработка серверной части. Нужно сделать приложение на Java+Spring используя REST API.

3.3.1. Структура проекта

Сначала рассмотри структуру файлов, она представлена на Листинге 5. Два важных файла вынесены на самый верх структуры – главный файл приложения RkibAppointmentBackendApplication, содержащий только метод main с запуском Spring.

Пакет config содержит файлы конфигурации безопасности, он будет рассмотрен в пункте 3.3.3. Validator содержит только файл с указанием шаблона для номера. Его использование будет продемонстрировано в пункте 3.3.2.

Листинг 5. Структура серверного приложения

```
1  RkibAppointmentBackendApplication.java
2  schema/
3      appointment/
4          status/
5      institution/
6      person/
7          employees/
```

```
 8            patient/
 9        procedures/
10        schedule/
11            status/
12    config/
13    exception/
14    validator
```

Пакет exception содержит реализацию ошибок, спроектированных в пункте 2.2.4.3. Приложение может выдавать ошибки с кодами 400 – неверный запрос, 401 – неавторизованный запрос, 403 – нет доступа, 404 – не найдено, 409 – конфликт. Причём ошибки 401 и 403 может генерировать сам Spring (конкретно – Spring Security). Нужно перехватывать эти ошибки, это делается с помощью класса с аннотацией @ControllerAdvice, перехватывающим конкретные ошибки Spring Security, например, валидацию. Код части этого класса приведён на Листинге 6.

Листинг 6. Класс-прехватчик ошибок Spring

```
 1  @ControllerAdvice
 2  public class ValidationAdvice {
 3      @ResponseBody
 4      @ExceptionHandler(ConstraintViolationException.class)
 5      @ResponseStatus(HttpStatus.BAD_REQUEST)
 6      public ExceptionResponse onConstraintValidationException(
 7      ConstraintViolationException e
 8      ) {
 9          List<ExceptionMessage> errors = e.
                getConstraintViolations().stream().map(
10          violation -> ExceptionMessage.builder()
11          .message(violation.getMessage())
12          .details(violation.getPropertyPath().toString()).
                build()
13          ).toList();
14          return ExceptionResponse.builder()
15          .httpCode(HttpCodes.BAD_REQUEST.getCode())
16          .errors(errors)
17          .build();
18      }
19      ...
20  }
```

Указывается конкретный тип перехватываемой ошибки, далее формируется ошибка в заданном для системы формате и вместо стандартной ошибки Spring на запрос возвращается она.

Основное приложение находится в schema – описание сущностей и работы с ними. Разделение – по логическим единицам, таким как Учреждение, Человек. При этом Пациент и Сотрудник вложены в пакет Человек, а статус вложен в то, статусом чего он является. В каждом из пакетов содержатся файле, которые соответствуют архитектуре слоёв, которая будет рассмотрена далее в пункте 3.3.2.

3.3.2. Архитектура слоёв

Архитектура слоёв – подход к проектированию, где функциональность разделяется на отдельные слои. Каждый слой изолирует свою логику. Архитектура сервера представлена на Рисунке 7.

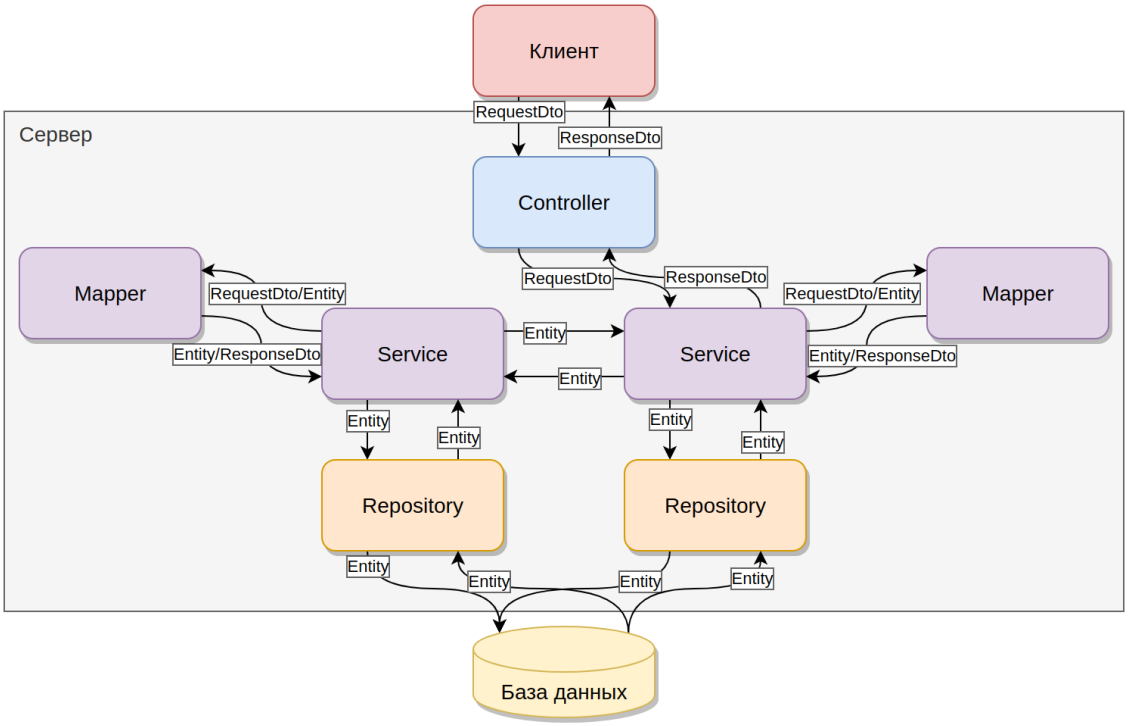

Рисунок 7. Архитектура слоёв сервера

Здесь изображено разделение логики на слои представления (Controller), бизнес-логики(Service и Mapper), доступа к данным (Repository). Отображено, какими объектами они обмениваются. Далее рассмотрим каждый из компонентов подробнее. Один из компонентов – Entity уже был рассмотрен в пункте 3.1.2

3.3.2.1. DTO

Клиент и сервер обмениваются данными через HTTP-запросы. В теле запросов передаётся информация в формате JSON, которая должна преобразовываться во внутренний объект сервера – DTO (объект передачи данных) в Entity.

Возникает проблема, когда запрос и ответ должны иметь разную структуру. Так, неразумно возвращать пароль пользователю. Да, можно его убирать перед отправкой, но надёжнее вообще не давать возможности отправить нежелательные данные. Также если в теле ответа нужно возвращать дополнительные данные, то им вовсе не обязательно быть в теле запроса. Поэтому DTO дополнительно разделяется на DTO для запросов и DTO для ответов. Сравнение этих двух типов DTO для представлено в Таблице 7.

Тело запроса	Тело ответа
– username – password * – lastName – firstName – patronymic	– id * – username – lastName – firstName – patronymic – patient * – employee *

Таблица 7. Сравнение объектов передачи данных для запроса и ответа

Символом «*» отмечены отличающиеся элементы. Здесь это значит, что password не будет возвращён в ответе. При этом будет добавится id – первичный ключ в базе данных, служащий идентификатором. С его помощью можно делать запросы конкретно с этим элементом. Далее появляются patient и employee. С обеими из таблиц Пациент и Сотрудник у таблицы Человек связь один к одному. Если человек является сотрудником, то для него будет дополнительно вложен объект такой, будто запросили информацию не просто о человеке, а о нём как сотруднике. Это уменьшает количество запросов и упрощает работу с данными.

Оба типа DTO используются только для ”общения”клиента и сервера и при первой же возможности заменяются на внутренние объекты – это увели-

чивает гибкость системы.

Важным аспектом DTO является валидация. Это проверка того, что данные удовлетворяют ограничениям. Ранее мы затронули самописную аннотацию @Phone. Это пример того, как декларативно можно поставить ограничение на значение поля. Использование аннотаций валидации выглядит как показано на листинге 7

Листинг 7. Пример валидации

```
1  @NotNull(message = "Phone number not specified")
2  @Phone(message = "Invalid phone number")
3  String phoneNumber;
```

При невыполнении какого-то правила возникает ошибка с определённым сообщением. Перехват таких ошибок был реализован в Листинге 6. Стоит заметить, что для работы валидации необходимо дополнительно указывать аннотацию @Validated на классе и @Valid на объекте, к которому применяется валидация. Так же она может проводиться и для других параметров, не обязательно в теле запроса.

3.3.2.2. Controller

Служит для связи клиента и сервера. В нём определяется url для получения ресурса и через необходимые параметры и тело в виде DTO он однозначно может передать запрос дальше по системе. Его единственная задача – определять API. На этом его ответственность заканчивается и через соответствие метода в контроллере и дальше работает слой сервисов (бизнес-логики). Пример кода такого класса с двумя из методов представлен на Листинге 8.

Листинг 8. Пример класса слоя представления

```
1  @RestController
2  @RequestMapping("${application.baseApi}${application.apiV1}${
       application.endpoint.employees}")
3  @Tag(name = "Employee")
4  @RequiredArgsConstructor
5  public class EmployeeController {
6      private final EmployeeService employeeService;
7      @GetMapping("{id}")
```

```
 8       @Operation(summary = "Get an employee")
 9       public EmployeeResponseDto get(@PathVariable String id) {
10           return employeeService.read(id);
11       }
12       @GetMapping("${application.endpoint.me}")
13       @Operation(summary = "Employee gets himself")
14       public EmployeeResponseDto getMe() {
15           return employeeService.readMe();
16       ...
17   }
```

Аннотации @Tag и @Operation относятся к автодокументированию кода, это будет рассмотрено в пункте 3.3.4. Остальные аннотации, кроме уже рассмотренного конструктора относятся к Spring. Они определяют путь и методы запроса. Единственная строка в методе – вызов соответствующего метода слоя бизнес-логики.

3.3.2.3. Service

Слой бизнес-логики. На этом слое запросы уже обрабатываются и формируется ответ. Состоит, как правило, из двух компонентов – интерфейса и его реализации, так выполняются принципы SOLID, инкапсуляции, улучшается тестируемость.

Как показано на Рисунке 7, элементы сервисного слоя могут передавать данные между друг другом. Для этого в классах помимо методов, возвращающих DTO должны быть методы, возвращающие сущности. Пример такого класса приведён на Листинге 9

Листинг 9. Пример класса слоя бизнес-логики

```
 1   @Service
 2   @RequiredArgsConstructor
 3   public class PatientServiceImpl implements PatientService {
 4
 5       private final PatientMapper mapper;
 6       private final PatientRepository repository;
 7       @Override
 8       public PatientResponseDto read(String id) {
 9           Patient patient = readEntity(id);
10           return mapper.toDto(patient);
```

```
11      }
12      @Override
13      public Patient readEntity(String id) {
14          return repository.findById(id)
15          .orElseThrow(() -> new NotFoundException("pas-02",  "
                    Patient  id=" + id, null));
16      }
17  }
```

3.3.2.4. Mapper

Вспомогательный компонент для работы с ним сервисов. Нужен исключительно для преобразования объекта передачи данных запроса в сущность и сущности в объект передачи данных ответа. Полезен и метод перевода массив одного в массив другого. Пример такого класса на Листинге 10. Можно использовать различные библиотеки, как Modelmapper или MapStruct, но у них есть большие проблемы – они плохо работают с Lombok, а отказ от него значительно ухудшит код.

Листинг 10. Пример класса-маппера

```
1   @Component
2   public class InstitutionMapper {
3       public Institution toEntity(InstitutionRequestDto
            institutionRequestDto) {
4           return Institution.builder()
5           .name(institutionRequestDto.getName())
6           .build();
7       }
8       public InstitutionResponseDto toDto(Institution
            institution) {
9           return InstitutionResponseDto.builder()
10          .id(institution.getId())
11          .name(institution.getName())
12          .build();
13      }
14  }
```

3.3.2.5. Repository (DAO)

Компонент, который практически не нужно писать самостоятельно. Как сказано в пункте 3.2, практически всё, что касается базы данных передаётся под контроль JPA. Самостоятельно требуется сделать только несколько вещей:

- Написание интерфейса для генерируемого класса работы с базой данных
- Свои простые запросы. По названию методов библиотека сама определит какой запрос нужен для проведения операции. Достаточно лишь передать свои параметры.
- Свои сложные запросы. Есть возможность с помощью аннотации @Query написать любой SQL-запрос, не отходя от логики созданных сущностей. Полезно, например, для фильтрации.

Пример обычного интерфейса для Репозитория – на Листинге 11. Указывается сущность, для которой нужно создать объект доступа к данным и тип его первичного ключа. После этого с помощью инъекции зависимостей можно будет использовать его в слое бизнес-логики.

Листинг 11. Пример интерфейса для репозитория

```
1  public interface AppointmentRepository
2          extends JpaRepository<Appointment, String> {
3  }
```

3.3.3. Безопасность

3.3.3.1. Хранение паролей

При добавлении пользователя логин и пароль передаются в незашифрованном виде. Хранить логин можно и без шифрования, а вот пароль – нет. Стандартно в Spring уже есть шифровальщик паролей – BCryptPasswordEncoder, который шифрует текст с помощью BCrypt алгоритма. Пароль хешируется и возвращается в виде строки. Эта строка будет хранить одновременно и информацию о том, что это BCrypt, и количество итераций при хешировании, и соль и, непосредственно, сам хешированный пароль.

3.3.3.2. Ограничение доступа

С помощью Spring Security можно определять, какие url каким пользователям будут доступны.

Для разрабатываемой системы хорошо подходит система ролей и доступа по ним. Роль (или другие характеристики) определяется путём нахождения пользователя в базе данных через его авторизационные данные. Если у пользователя есть необходимая роль, то доступ ему разрешается. Определяется пользователь исходя из передаваемого вместе с каждый запросом заголовка «Authorization». Далее, когда пользователь уже известен, происходит предоставление доступа, либо отказ в соответствии с конфигурацией, часть которой указана на Листинге 12.

Листинг 12. Пример кода для разграничения доступа

```
1  http.userDetailsService(userDetailsService).csrf().disable()
2      .authorizeHttpRequests(auth -> auth
3      .requestMatchers(HttpMethod.GET, appointmentPath + "/{
         appointmentId}").authenticated()
4      .requestMatchers(HttpMethod.GET, appointmentPath).
         authenticated()
5      .requestMatchers(HttpMethod.POST, appointmentPath).
         hasAnyAuthority(
6          AccountRole.ADMIN.toString(), AccountRole.DOCTOR.
             toString())
7      .requestMatchers(HttpMethod.PUT, appointmentPath + "/{
         appointmentId}").hasAnyAuthority(
8          AccountRole.ADMIN.toString(), AccountRole.DOCTOR.
             toString())
9      .requestMatchers(HttpMethod.DELETE, appointmentPath + "/{
         appointmentId}").hasAuthority(
10         AccountRole.ADMIN.toString())
```

3.3.4. Документация

Документация, которая может пригодиться – документация API. В случае если потребуется взаимодействовать с сервером не только написанному в рамках данного проекта, а кому-то, кому неизвестен исходный код, то она сделает разработку возможной.

Для документирования есть очень удобный инструмент – Swagger. Исходя из того, что написано в слое контроллеров и того, из чего состоят DTO он может сделать полноценную интерактивную документацию для API проекта. Выглядит это как на Рисунке 8.

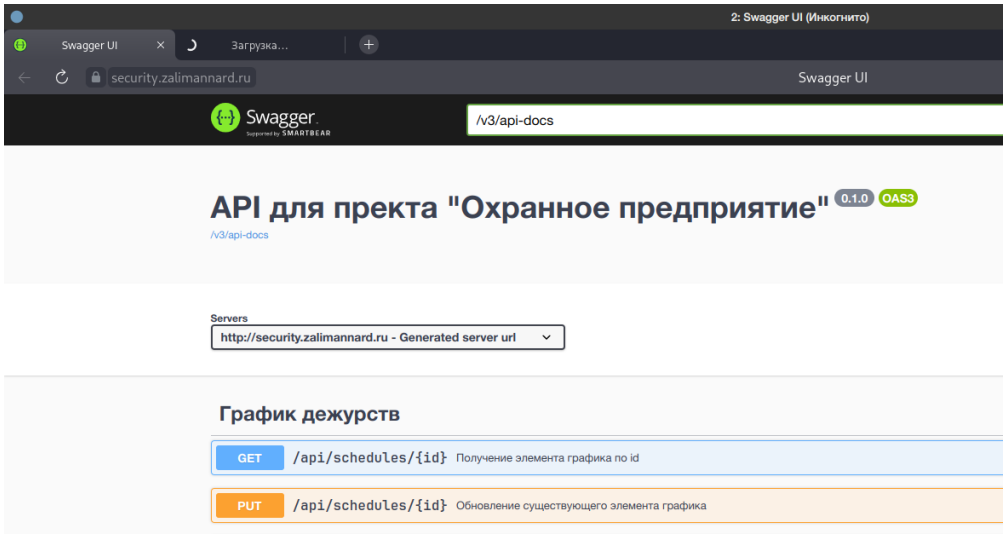

Рисунок 8. Пример документации Swagger

У него есть уже упомянутые аннотации, как @Tag, которые позволяют составлять документацию ещё и в удобном для чтения формате. Присутствует возможность отправлять запросы прямо из этой панели.

Главное преимущество автодокументирования – то, что такая документация всегда актуальна. Она подстраивается под изменения кода и не может быть такого, что она устарела.

3.3.5. Тестирование

Основной вид тестирования, которым нужно покрывать разрабатываемую систему – интеграционное тестирование. Это вид тестирования, который позволяет удостовериться, что на конкретный запрос при конкретном состоянии системы возвращается заранее известный ответ. Таким образом, если все интеграционные тесты выполняются и они правильно написаны, то можно быть уверенным, что сервер работает правильно.

Разработка тестов – это отдельная большая часть работы над системой. Тестировать нужно не только уже разработанную систему на наличие ошибок. Тесты нужно создавать и проводить на протяжении всей разработки, причём, что важно – до написания кода. Этот подход называется TDD (Раз-

работка через тестирование) и является хорошей практикой в программировании, потому что тесты пишутся ”на свежую голову”по этапу проектирования, а не тогда, когда вся программа уже написана и тесты подстраиваются под неё. Эффективность тестирования при TDD выше. [2]

Для данной системы тесты были разработаны в соответствии с TDD в отдельном Java-проекте. Можно сказать, что каждый из методов – это эмуляция запроса клиента. Поэтому для них требуется наличие только DTO для запроса и ответа и путей к конечным точкам приложения. Так можно будет покрыть автоматическими тестами API всё приложение.

Пример автоматического теста приведён на Листинге 13. Здесь пользователь ADMIN, который создаётся по умолчанию при запуске создаёт пользователей с разными ролями. Все они должны создаться, тест позитивный. После завершения теста созданные данные удаляются – это важный элемент в написании автотестов, ведь это сохраняет в чистоте тестовую среду и делает тесты более предсказуемыми и отслеживаемыми.

Листинг 13. Пример автотеста для разработанной системы

```
1  @ParameterizedTest
2  @Severity(SeverityLevel.CRITICAL)
3  @DisplayName("Successful addition with all data from ADMIN")
4  @CsvSource(value = {"ADMIN", "REGISTRAR",
5      "DOCTOR", "PATIENT"})
6  void testCreatePerson_AllCorrectDataByAdmin_Created(String
       role) {
7      PersonRequest personToCreate = PersonFactory.
           createPersonRequest();
8      PersonResponse actual = personSteps.post(personToCreate,
           adminAuth, specifications.responseSpecificationV1(201)
           , PersonResponse.class);
9      assertThat(actual).isNotNull();
10     assertThat(actual.getId()).isNotNull();
11     PersonResponse expected = PersonFactory.
           createPersonResponse(actual.getId(), personToCreate);
12     assertThat(actual).isEqualTo(expected);
13     PersonResponse existedPerson = personSteps.get(actual.
           getId(), adminAuth, specifications.
           responseSpecificationV1(200), PersonResponse.class);
14     assertThat(existedPerson).isEqualTo(expected);
15     personSteps.delete(actual.getId(), adminAuth);
```

```
16    }
```

Часто бывает необходимо не просто знать, что тест есть, а иметь возможность быстро понять что именно привело к непройденному тесту. В таких случаях очень помогает специально средство для визуализации тестирования. В Gradle есть такой по умолчанию, но он не наглядный. Лучшая замена ему – Allure. Он представляет результаты тестов в виде удобного отчёта. Пример отчёта Allure для разрабатываемой системы приведён на Рисунке 9.

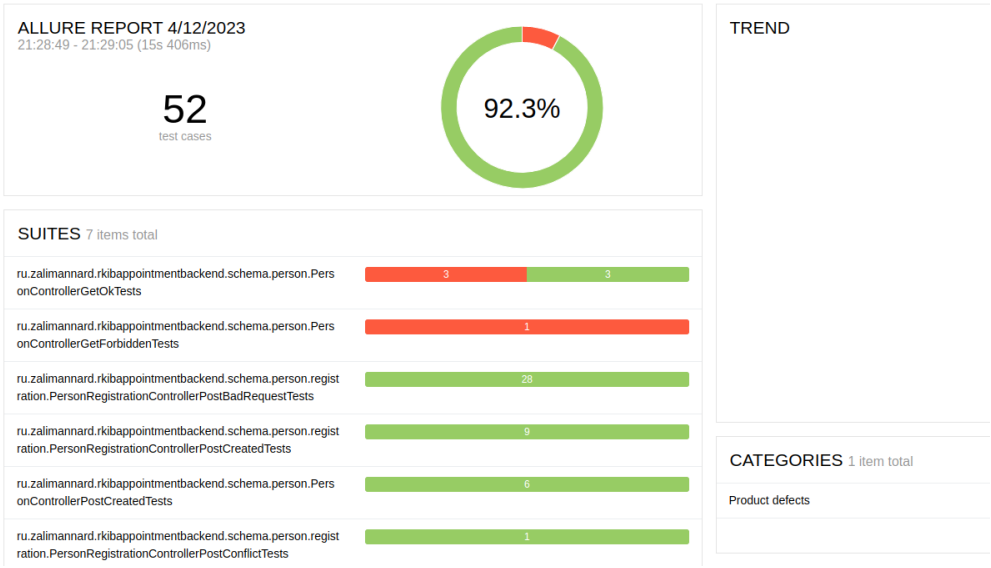

Рисунок 9. Пример визуализации результатов тестирования

Тесты бывают позитивные, которые проверяют нормальный ход работы системы, и негативные, которые проверяют правильность ошибки, появляющийся при том или ином действии. Большую часть всегда занимают негативные, потому что возможных сценариев ошибки значительно больше, чем сценариев нормальной работы. Это тоже можно увидеть на Рисунке 9.

3.4. Клиентская часть

3.4.1. Структура проекта

3.4.2. Архитектура слоёв

3.4.3. Пользовательский интерфейс

3.4.4. Тестирование

Листинг 14. Структура клиентского приложения

```
views/
    admin/
    doctor/
    registrar/
components/
    alert/
    button/
    dialog/
    select/
    table/
    textfield/
plugins/
router/
types/
backspaceHandlers.ts
main.ts
masks.ts
rules.ts
utils.ts
```

3.5. Сборка и развертывание

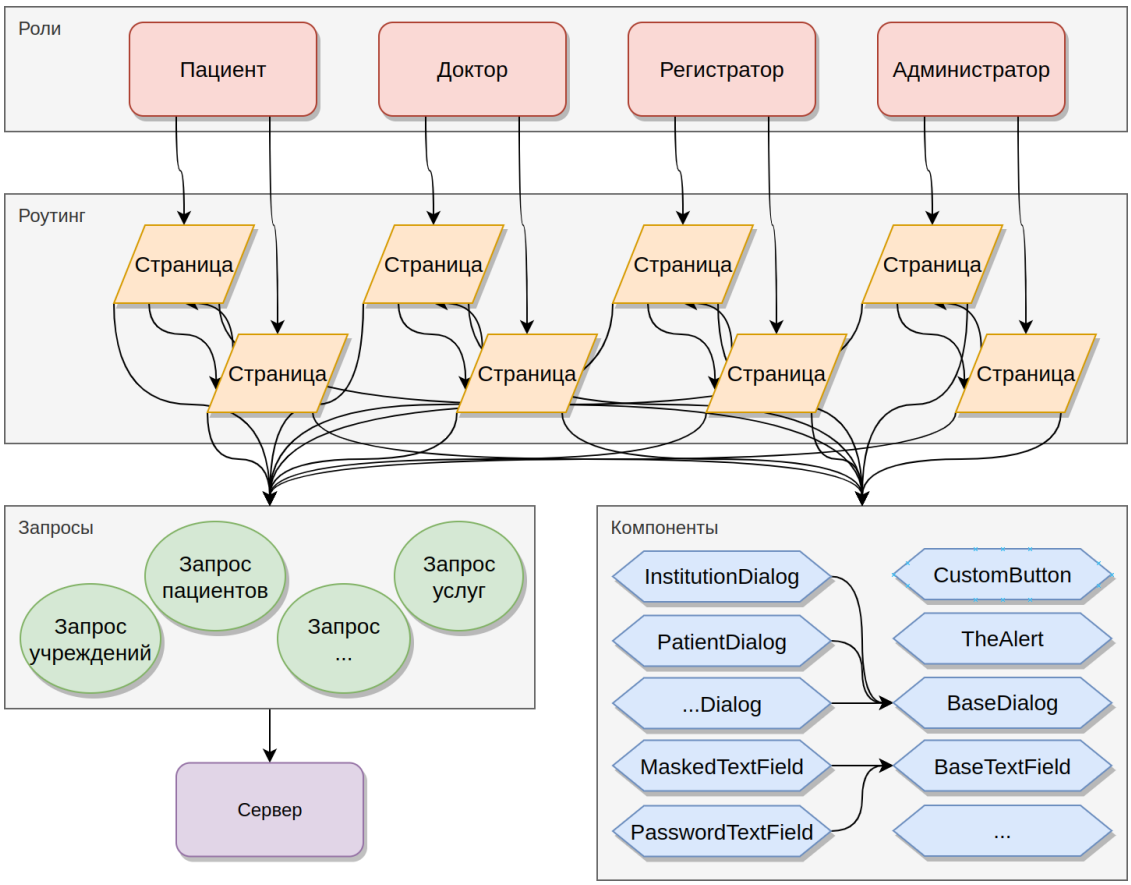

Рисунок 10. Схема работы клиента

Заключение

Список использованных источников

[1] Федеральный проект «Создание единого цифрового контура в здравоохранении на основе единой государственной информационной системы в сфере здравоохранения (ЕГИСЗ)» – 2019 – 9 августа [Электронный ресурс] – URL: https://minzdrav.gov.ru/poleznye-resursy/natsproektzdravoohranenie/tsifra/ (Дата обращения: 13.12.2022)

[2] Мартин Р. Чистый код: создание, анализ и рефакторинг / Р. Мартин; пер. с англ. – СПб.: Питер, 2022. – 464 с.

[3] Википедия. API [Электронный ресурс] – URL: https://ru.wikipedia.org/wiki/API (Дата обращения: 20.01.2023)

[4] Eugen. «Build your API with Spring» [Электронный ресурс] – URL: https://www.baeldung.com/rest-api-spring-guide (Дата обращения: 01.12.2022)