Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский государственный социальный университет»

Факультет информационных технологий

Направление подготовки – 09.03.04 Программная инженерия

Квалификация: Бакалавр

Выпускная квалификационная работа

Тема: Разработка информационной системы взаимодействия медицинских клиник с клиентами

Обучающийся _____ Юмаев Артур Русланович
Подпись

Дата _____

Руководитель _____ _____
Подпись (ученая степень, ученое звание, фамилия, инициалы)

ВКР допущена к защите « » _____ 2021 г.

Декан факультета информационных технологий, канд. пед. наук, доцент _____ Крапивка С. В.
Подпись

Москва, 2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 6
Анализ существующих решений систем автоматизации работы с клиентами медицинскх клиник 9
1.2 Анализ и выбор инструментальных средств для реализации проекта 18
Выводы по главе 1 26
2. ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ МОДЕЛЕЙ И РАЗРАБОТКА ИНФОРМАЦИОННОЙ СИСТЕМЫ ВЗАИМОДЕЙСТВИЯ МЕДИЦИНСКИХ КЛИНИК С КЛИЕНТАМИ 28
2.1 Выбор методов и средств проектирования 28
2.2 Логическое проектирование системы 31
2.3 Физическое проектирование системы 33
2.4 Проектирование макета пользовательского интерфейса 35
2.5 Описание конечного вида интерфейса 42
2.6 Алгоритмы работы 51
Выводы по главе 2 55
3. РАЗРАБОТКА КОМПОНЕНТОВ ИНФОРМАЦИОННОЙ СИСТЕМЫ 56
3.1 Описание среды разработки и программы 56
3.2 Программная реализация информационных моделей 58
3.2.1 Авторизация 62
3.2.2 Профиль 63
3.2.3 Записи 64
3.2.4 Календарь 65
3.2.5 Процедуры, пациенты, кабинеты и отделения 68
3.2.6 Модуль взаимодействия с базой данных 69
3.2.7 Модуль отправки Email оповещений 70
Выводы по главе 3 72
4. РАСХОДЫ НА РАЗРАБОТКУ ИНФОРМАЦИОННОЙ СИСТЕМЫ 73

4.1 Оценка стоимости разработки, внедрения и поддержки 73
Выводы по главе 4..... 76
ЗАКЛЮЧЕНИЕ 77
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 79
ПРИЛОЖЕНИЕ А. Фрагменты исходного кода программы 82

ВВЕДЕНИЕ

Тема выпускной квалификационной работы «Разработка информационной системы взаимодействия медицинских клиник с клиентами».

Внедрение разработанной информационной системы позволит сократить затраты времени на взаимодействие с клиентами медицинских клиник, обработку и хранение информации о клиентах, получение быстрого доступа к информации о клиентах, анализ исторических данных пациента, основываясь на его записях к врачам клиники и медицинской карте, а также ускорит процесс записи новых пациентов.

Объект исследования: процесс разработки информационной системы взаимодействия медицинских клиник с клиентами.

Предмет исследования: содержательные и технологические особенности разработки информационной системы взаимодействия медицинских клиник с клиентами.

Цель работы: разработать информационную систему взаимодействия медицинских клиник с клиентами.

Для реализации поставленной цели сформулированы следующие задачи:

1) провести анализ предметной области и инструментальных средств для реализации проекта;
2) разработать информационную модель пользовательского интерфейса системы взаимодействия медицинских клиник с клиентами;
3) осуществить программную реализацию информационной модели и алгоритмов учета пациентов и записей к врачам;
4) привести затраты на реализацию, внедрение и поддержку информационной системы.

Выпускная квалификационная работа содержит 4 главы, сто десять страниц, пять таблиц, сорок один рисунок, 14 листингов и одно приложение.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

В современной цифровой экономике институтам здравоохранения все чаще требуется внедрять информационные и телекоммуникационные технологии для повышения эффективности, особенно в условиях всемирной пандемии 2020 года, когда для предотвращения распространения вируса необходимо максимально сократить взаимодействие с пациентом. Исследование «Индекс здоровья будущего», проведенное компанией Philips в 2018 году показало, что 82% населения России были бы готовы дистанционно консультироваться с врачом в определенных жизненных ситуациях, 60% россиян испытывали потребность в медицинском уходе на дому сами или для тех, о ком они заботятся и 80% респондентов полагают, что интегрированные цифровые технологии важны для повышения качества этих услуг [15].

Некрасова Т. А., Наролина Т. С., Смотрова Т. И. и Пургаева И. А. проанализировали перспективы развития российского рынка цифровой медицины [8]. В своей статье они рассмотрели особенности и барьеры развития цифровой медицины в России и мире. По данным Global Market Insights, к 2025 году объем мирового рынка цифровой медицины (Digital Health) составит $504,4 млрд [25]. Эксперты прогнозируют увеличение рынка почти в 6 раз к 2025 году относительно 2018 года. Наиболее перспективными нишами в г. Москве в отрасли цифрового здравоохранения являются телемедицинские системы, системы поддержки принятия врачебных решений, носимые устройства мониторинга состояния здоровья и системы управления медицинскими учреждениями, на что была направлена разработка, описанная в ВКР. Россия в целом отстает от использования информационных технологий в медицине, что является возможностью для компаний к разработке передовых продуктов с использованием искусственного интеллекта и других методов интеллектуальной обработки данных.

В 2018 году компания Philips провела исследование «Индекс здоровья будущего 2018», которое поможет в понимании вектора развития разрабатываемой информационной системы. Так, например, Российские институты здравоохранения вдвое отстают по уровню цифровизации, сбора и обработки информации. Сейчас Россия находится ниже среднего показателя по 16 странам по сбору медицинских данных (14,53 средний показатель по России и 31,03 средний по 16 странам) и испытывает сложности в отношении ПО для администрирования электронных записей на прием к врачу. Эксперты отмечают, что недостаток инвестиций цифровые медицинские технологии сказывается на качестве предоставления медицинских услуг. Тем не менее это говорит о возможностях для роста на российском рынке [16].

Аналогичный цикл исследований был проведен в ноябре и декабре 2019 года, в котором приняло участие молодое поколение врачей до 40 лет [17]. Специалисты считают, что развитие и внедрение цифровых медицинских технологий окажут положительное влияние на помощь пациентам, рационализируют рабочие процессы и минимизируют уровень стресса. Исследование показало следующее.

2. 83% опрошенных считают, что цифровые технологии – важный инструмент для достижения лучших результатов лечения пациентов;
3. 66% опрошенных ожидают снижение уровня стресса от внедрения цифровых медицинских технологий;
4. 87% отметили потенциал снижения нагрузки;
5. 59% называют внутреннюю бюрократию главным барьером, влияющим на их способность управлять изменениями в медицинских учреждениях.

Во время приема пациента врачу приходится очень много писать от руки. При отсутствии автоматизации процессов в медицинской клинике фиксация

результатов осмотра и выписка направлений происходит также вручную. Это занимает большое количество времени и сил врача. Цифровые сервисы помогут быстро и безошибочно отражать результаты своей работы. Когда автоматизированные рабочие места соединены в единую сеть, обмен информацией между сотрудниками намного упрощается. Запись клиента на повторный прием можно или к другому врачу можно произвести сразу во время приема, не обращаясь при этом в регистратуру. Также врач со своего рабочего места может назначить исследования или лечебные процедуры, и результаты этих исследований и процедур не будут утеряны, ведь немаловажным преимуществом автоматизации является резервирование, которое обеспечивает надежное хранение всей медицинской информации.

Автоматизация помогает врачу планировать многие рутинные мероприятия, например диспансерное наблюдение хронических больных, льготное лекарственное обеспечение, иммунизация и иммунопрофилактика, профессиональные и прочие осмотры. Автоматизация упрощает согласование протоколов совместного осмотра или решений врачебной комиссии: вся информация вносится в электронном виде, нет необходимости использовать бумажные карты.

Исследования показали, что больше половины (53%) медицинских работников информированы о цифровых технологиях в здравоохранении, но среди населения только 22% имеет о них какое-либо представление. Тем не менее 95% специалистов в медицинских учреждения и 85% населения признают важность внедрения цифровых решений и многие согласны с тем, что эти технологии уже активно применяются.

Одним из наиболее значимых факторов для внедрения в медицинских клиниках информационных систем является наличие квалифицированных работников. В мире среднем на 10,000 работников приходится 87 специалистов по

информационным технологиям, из которых только половина является работниками высшего уровня квалификации. В России эта цифра намного выше – 230 человек на 10,000 имеет квалификацию в информационных технологиях.

Также существуют другие препятствия для внедрения цифровой медицины в России:

1. Сложности с привлечением финансирования (22%);
2. Несовершенство законодательства, отсутствие стандартизации (13%);
3. Сложности взаимодействия с органами власти (11%);
4. Незрелость и консерватизм рынка (11%);
5. Сложности с выводом на рынок и продвижение продукта (9%).

Таким образом, в этой части были рассмотрены преимущества и недостатки внедрения цифровой медицины в России, которые показали, что преимущества преобладают над недостатками и в данный момент на российском рынке существует потенциал для внедрения цифровых услуг в здравоохранении.

Анализ существующих решений систем автоматизации работы с клиентами медицинскх клиник

На сегодняшний день существует большое количество информационных систем взаимодействия медицинских клиник с клиентами, каждая из которых может быть удобна в различных ситуациях. Крупные компании могут позволить себе более сложные и дорогие системы, либо иметь отдел разработки или заказать у сторонних разработчиков полностью готовую систему с долгосрочной поддержкой. Зачастую в медицинских клиниках, как государственных, так и частных используются такие системы учёта как «1С-Предприятие», которые настраиваются под заказчика, но довольно сложны в освоении работниками

медицинских клиник. Рассмотрим существующие решения по автоматизации работы с пациентами, представленные на рынке.

Клиника онлайн

Клиника Онлайн – информационная система для управления медицинской клиникой. Программа предоставляет владельцу клиники контроль над ключевыми и сложными процессами такими, как например, динамика выручки, возвращаемость и отток клиентов, контроль и мотивация сотрудников. На основе собранных данных клиника может принимать наиболее эффективные решения и увеличивать доходы [26].

Компания дает возможность интегрироваться с ЕГИСЗ. Возможности Клиники онлайн:

- передача медицинских карт и дневников приема в систему ЕГИСЗ;
- медицинская карта и шаблоны дневников приема по стандартам постановления правительства;
- автоматическое заполнение 80% полей, требуемых ЕГИСЗ.

Медицинская система Archimed+

Archimed+ дает возможность работать в ней почти всем сотрудникам клиники: медрегистраторы, врачи, бухгалтер, лаборанты и другие. Для каждого из

них в системе реализован богатый функционал. Давайте посмотрим, что может делать каждый сотрудник.

Программа поможет быстро найти карту пациента и любую информацию в ней – когда в последний раз был, что проходил, каких врачей посещал, какие имеются документы – договор, информированное согласие пациента и другие [20].

Archimed+ предоставляет следующие возможности:

- ведение базы пациентов;
- ввод и хранение персональных данных;
- печать амбулаторной карты 025/у;
- печать стоматологической карты 043/у;
- выгрузка базы данных в Excel.

Medesk – облачная медицинская информационная система

Medesk – это платформа для управления частными клиниками, позволяющая интегрировать медцентры с партнерами, а также с устройствами и сервисами мониторинга состояния здоровья пациента. Благодаря этому врачи клиники или лаборатории получают возможность лучше и эффективнее взаимодействовать друг с другом и зарабатывать на предоставлении качественных услуг пациентам и корпоративным клиентам [21].

Рисунок 1. Пример интерфейса платформы Medesk

Платформа Medesk позволяет:

- управлять рабочим процессом удаленно;
- распределять задачи для персонала;
- записывать телефонные звонки для контроля качества;
- создавать индивидуальные отчеты.

MedicalCRM

MedicalCRM – облачная система для медицинских клиник, которая является интернет платформой и не требует инсталляции на ПК. Работа ведется через браузер на любом устройстве, подключенном к Интернет. MedicalCRM представляет из себя модульную систему, каждая часть из которой подключается отдельно и предоставляет ряд функций для автоматизации работы каждой конкретной части медицинской клиники. Рассмотрим модули и их функции [24].

Расписание:

- наглядное изображение загруженности специалистов;
- внесение изменений о приемах;
- быстрая работа с большими каталогами услуг;
- возможность настройки интерфейса.

Сотрудники:

- любое количество сотрудников;
- гибкие схемы расчета зарплаты врачей;
- разделение прав доступа;
- задания и напоминания на определенное время.

Документооборот:

- электронная медицинская карта;
- настраиваемые шаблоны протоколов;
- печать любых документов.

Medidea

Medidea – программа для клиник, медицинских центров и косметологических клиник с возможностью работы в системе ОМС, учет и списание медикаментов, электронная история болезни, финансовая и экономическая аналитика клиники, онлайн запись, АТС, расчет заработной платы [22].

Информационная система Medidea также является модульной и предоставляет следующие модули:

- модуль работы в омс;
- модуль складского учета;
- модуль сетки расписания;
- модуль рассылки;
- модуль истории болезни;
- модуль online записи;
- модуль кассы.

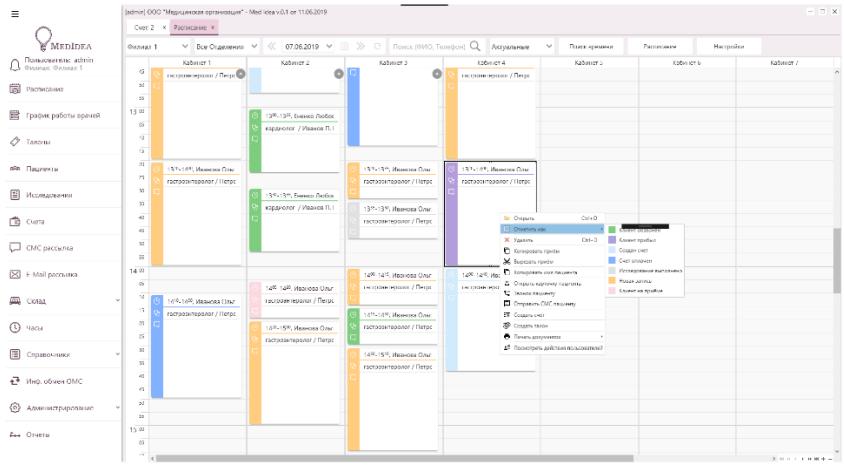

Рисунок 2. Пример интерфейса МИС Medidea

Renovatio

Медицинская информационная система Renovatio позволяет клиникам любого размера и профиля оцифровывать и сделать максимально прозрачной и эффективной работу каждого сотрудника. Renovatio предоставляет широкий набор

различных функциональных модулей, назначение которых – полноценная и эффективная автоматизация медицинских учреждений [23].

Renovatio позволяет интегрироваться в различные типы учреждений:

- частные врачи;
- медцентры;
- стоматологии;
- ветеринарии;
- сетевые клиники;
- франшизы.

Renovatio состоит из нескольких модулей, которые можно отдельно интегрировать в систему в зависимости от типа предприятия:

- врачебный модуль;
- бухгалтерия;
- профилактические осмотры;
- складской учет;
- телефония;
- зарплата;
- аналитика;
- интеграция с лабораториями.

Каждый из модулей предоставляет роли в системе и в зависимости от роли предоставляет набор функций по управлению системой:

Администратору:

- гибкая настройка графика работы;
- система записи;
- быстрое оформление пациента;
- интеграция с кассой и эквайрингом;
- листы ожидания и резерва.

Врачу:

- шаблоны и протоколы приемов;
- быстрое копирование с предыдущего приема;
- транзитные данные;
- планы лечения и контроль наначений.

Пациенту:

- система бонусов и персональные скидки;
- система оповещений;
- автоматическая рассылка результатов анализов;
- личный кабинет пациента (онлайн запись, оплата, история лечения).

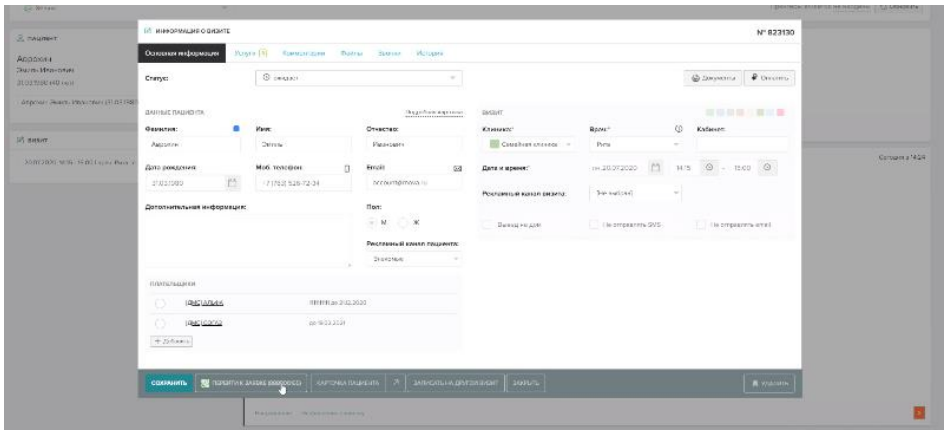

Рисунок 3. Пример интерфейса информационной системы Renovatio

CleverBox:CRM

CleverBox:CRM – информационная система для автоматизации управления салоном красоты, косметологическим центром, медицинской клиникой и СПА [11]. Система представляет из себя облачную платформу для управлений медицинским центром и предоставляет следующие возможности:

- удобное расписание;
- создание визита в несколько кликов;
- полный контроль переноса и отметы визитов;
- настройка статусов и текущее состояние визита;
- настройка учета и работа с данными пациентов;
- история визитов;
- финансовый анализ деятельности;
- эффективность работы персонала;
- оценка рентабельности;
- учет расходов по статьям, учет внереализационных доходов.

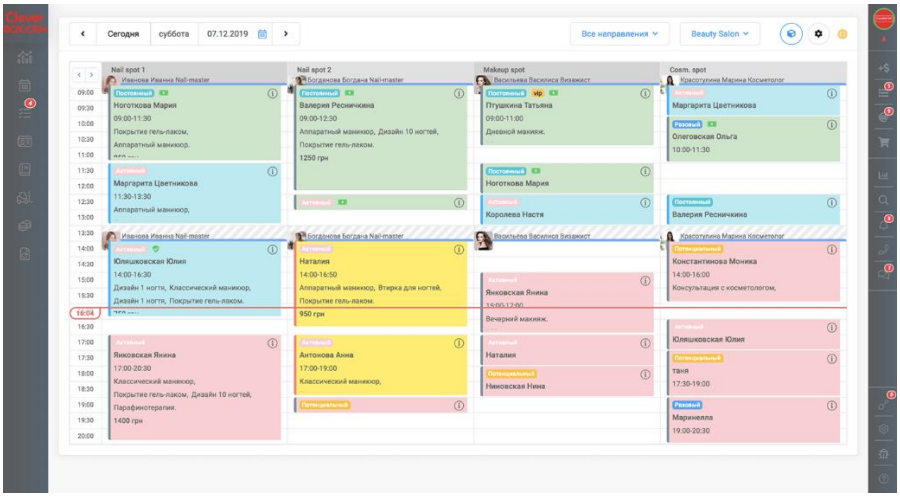

Рисунок 4. Пример интерфейса информационной системы CleverBox:CRM

1.2 Анализ и выбор инструментальных средств для реализации проекта

Как показало исследование в анализе предметной области, на сегодняшний день растет спрос на автоматизацию работы с клиентами в частных и государственных медицинских клиниках. Также мы наблюдаем рост медицинской отрасли в целом ввиду развития технологий. В связи с этим в данный момент актуальна разработка системы автоматизации медицинской клиники – эту задачу решает «ИС Медицинский Сервис», разработка которой описана в данной ВКР.

«ИС Медицинский Сервис» является веб-приложением, доступным через интернет-браузеры и не зависит от операционной системы, на которой запускается. Основным критерием является наличие современных версий браузеров таких, как Google Chrome, Mozilla Firefox и Apple Safari.

В разработке веб-приложений можно использовать различные виды баз данных. Одними из самых популярных баз данных являются реляционные базы данных (SQL базы данных) и NoSQL базы данных. Реляционные базы данных – одни из самых первых и до сих пор используются повсеместно при разработке информационных систем. Данные и связи между этими данными организованы в виде набора таблиц. Каждый столбец в такой таблице имеет свой тип и уникальный идентификатор (зачастую имя столбца). Каждая строка в таблице представляет определенную информацию или элемент таблицы (запись), которая содержит значения столбцов в соответствии с указанным типом. Поле или столбец в таблице может называться внешним ключом и содержать ссылку на столбцы в другой таблице. Это свойство позволяет разделять базу данных на сущности (entities), выделять их свойства (столбцы) и значения этих свойств (записи). Распределенность таких баз данных делает их гибкими, так как изменения в одной таблице не влекут за собой существенные изменения в других таблицах. Для

доступа к такой структуре данных используется язык запросов SQL. В настоящий момент реляционные базы с поддержкой SQL являются надежным выбором при разработке приложений. Пример реляционных баз данных: MySQL, MariaDB, PostgreSQL, SQLite, Oracle [3].

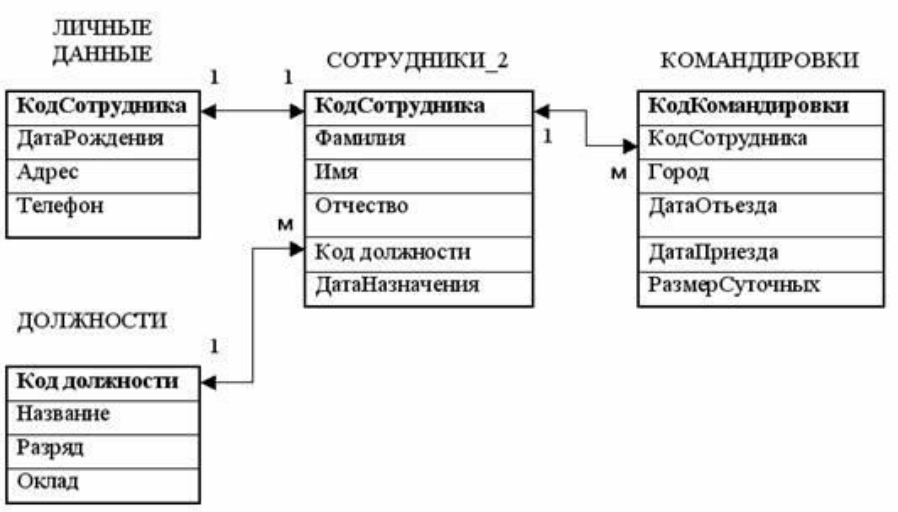

Рисунок 5. Пример реляционной базы данных

Вторым типов баз данных являются NoSQL базы данных. Она предполагает структуру, отличную от реляционного типа. Ее можно понимать, как “не только SQL”. В NoSQL типе базы данных также можно создать реляционную структуру. Существуют базы данных “ключ-значение”, которые являются коллекциями записей, где по ключу в базе данных находится объект данных, который может представлять уникальную структуру. Такие хранилища предоставляют быстрый и мало затратный доступ, могут хранить данные конфигураций, хранить различные типы данных [9]. Примеры таких баз данных: Redis, memcashed, etcd.

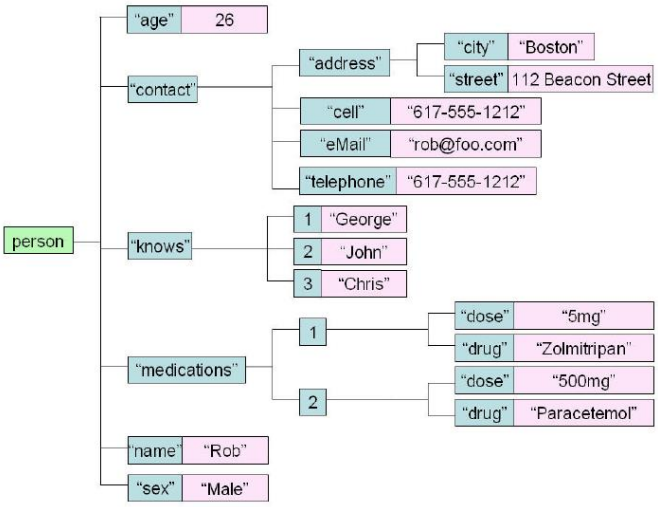

Рисунок 6. Пример хранения данных в нереляционной базе данных

Также существуют ряд дополнительных типов баз данных, но в результате анализа была выбрана база данных с реляционной структурой и поддержкой SQL языка – PostgreSQL. В разработке использовались несколько инструментов для подключения и выполнения запросов к базе данных. Первый – Datagript – продукт от компании JetBrains, который обеспечивает работу с разными базами данных: PostgreSQL, MySQL, Oracle, SQL Server и многими другими. Datagrip может получать информацию о разных важных объектах базы данных и отображать их в виде дерева. Он может строить UML диаграмму, что является полезной функцией для отладки при создании базы данных. В нем есть интерфейс для создания столбцов, индексов, ограничений, авто дополнение запросов и многое другое. Данные можно редактировать в табличном редакторе, столбцы можно группировать и сортировать через интерфейс. Он помогает определять ошибки в SQL запросах, генерировать код, и также компания JetBrains предоставляет студенческую лицензию на пользование продуктом бесплатно [31].

PosgreSQL – свободная и популярная реляционная база данных. Она поддерживает неограниченный размер базы данных, надежные механизмы

транзакций, C-совместимые модули, наследованием и легкую расширяемость. PostgreSQL позволяет создавать распределенные системы ввиду того, что доступ к ней осуществляется через специальный процесс базы данных. Данную базу данных можно устанавливать на отдельном сервере и посылать ей запросы через сеть. Таким образом, к базе данных могут подключаться клиенты с разных серверов и разных платформ. К примеру, PostgreSQL можно установить на UNIX сервер, а клиентское приложение будет работать на Windows или вовсе из браузера клиента. PostgreSQL является надежной – она соответствует принципам атомарности, изолированности, непротиворечивости и сохранности данных. Она производительна, так как использует планировщик запросов, систему управления буферами памяти и кэширование. Она расширяется, поддерживается и реализует широкий спектор типов данных. Далее рассмотрены подробнее ключевые особенности данной базы данных.

Ключевые особенности PosgreSQL [16]:

1. Надежность
    a. PostgreSQL соответствует принципам ACID – атомарность (транзакция является единой логической единицей – все ее изменения сохраняются целиком или полностью откатываются), непротиворечивость (транзакция переводит базу данных из одного целостного состояния в другое), изолированность (изменения, внесенные в базу данных при двух конкурируемых транзакциях изолированны друг от друга) и сохранность данных (PosgreSQL гарантирует, что изменения в транзакции гарантированно сохранятся на диск);
    b. возможность восстановления базы данных;
    c. открытость исходного кода.
2. Производительность
    a. поддержка индексов (обобщенное поисковое дерево);

b. планировщик запросов – предоставляет возможность пользователю откладывать запросы и учитывать различные факторы;
c. использует кэширование и алгоритмы для эффективного расходования ресурсов.

3. Расшияремость
    a. пользователь может создавать новые функции, типы, языки, индексы и операторы;
    b. дает возможности моделировать область знаний, не являясь специалистом в базах данных.
4. Богатый набор типов данных
    a. text с неограниченной длиной;
    b. Numeric – поддерживает произвольную точность и является полезным в финансовой и научной отрасли;
    c. объекты размером больше 2 Гб;
    d. геометрические типы ;
    e. временные типы.
5. Безопасность
    a. обеспечивает 4 уровня безопасности;
    b. SSL, SSH шифрование трафика между клиентом и сервером;
    c. детализированная система разграничения прав на основе ролей к каждому объекту базы данных.

Существует 2 самых распространенных типов построения API – REST и RPC. REST (Representational State Transfer) – один из способов построения API. REST позволяет создавать сервисы, которых хорошо кэшируются и не содержат состояния клиента, то есть между клиентом и сервером не существует общего состояния. Реализациями REST архитектуры являются протоколы RESTful,

JSON:API, GraphQL. Функции в RESTful сервисах обычно состоят из URL и HTTP метода. Дополнительные параметры обычно отправляются в url запросе в виде url queries или в теле HTTP запроса body. В качестве формата данных обычно используется JSON (JavaScript Object Notation) [10] [12]. В качестве примера рассмотрим запросы к RESTful сервису:

- GET /user/;
- POST /user/;
- DELETE /user/3;
- PUT /user/2.

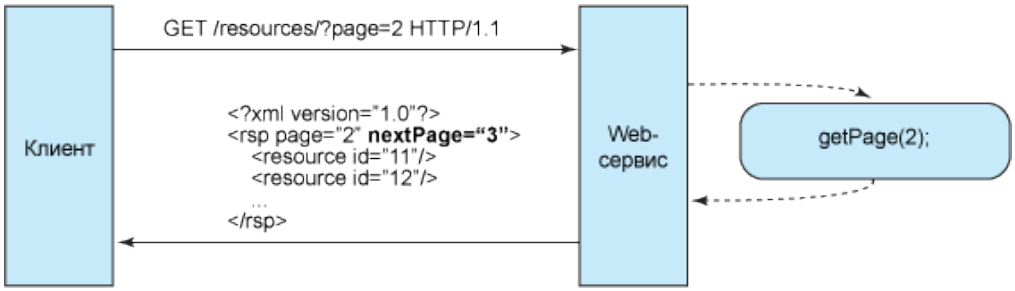

Рисунок 7. Модель взаимодействия REST без сохранения состояния

Второй тип API – RPC. RPC (Remote Procedure Call) – заключается в вызове удаленной процедуры с набором аргументов. Реализациями RPC архитектуры являются протоколы SOAP, JSON-RPC, gRPC.

В качестве взаимодействия между клиентом и сервером была выбрана модель REST API и был построен RESTful сервер.

Для разработки веб-приложения был выбран основной язык JavaScript. Современные возможности JavaScript позволяют писать как клиентскую часть, так и серверную. В качестве фреймворков для клиентской части существуют Angular JS, React JS, Vue JS, Nuxt JS. React JS является самым популярным из них, имеет

самое большое сообщество разработчиков и постоянную поддержку со стороны разработчиков в компании Facebook. В качестве разработки клиентской части был выбран React JS версии 17 [34]. Единственной платформой для разработки сервера на JavaScript является Node JS [35]. Самым популярным фреймворком для разработки HTTP веб-сервера является Express JS, который был выбран в качестве разработки [32].

В качестве типизации JavaScript был выбран язык TypeScript. TypeScript это язык с открытым исходным кодом, построенный на основе JavaScript. Он позволяет добавлять статическую типизацию и определения типов в программу. Типы предоставляют способ описания формы объекта, документируя программу и валидируя ее на корректность работы. Каждая валидная JavaScript программа является валидной TypeScript программой. TypeScript код компилируется в JavaScript при помощи компилятора или Babel – инструмент транспиляции, который позволяет переносить JavaScript код на более старые версии ECMA Script для запуска в старых браузерах [42].

Для сборки клиентской части будет использоваться утилита Webpack. Webpack это сборщик модулей клиентской части приложения. Он занимается анализом модулей, создает граф зависимостей и собирает модуль в правильном порядке для построения “бандла” – production версии клиентского приложения, который потом с помощью веб-сервера доставляется клиенту [42]. В качестве тергетируемого файла выступает файл index.html – входная точка в веб-страницу. Вебпак решает много проблем при сборке проекта в один или несколько файлов. Первая из них – JavaScript модули, которые подключаются в теге <script> в файл index.html. Программист может забыть подключить некоторые модули или подключить их в неверном порядке, что может привести к неявным ошибкам в работе веб-приложения. Также вебпак помогает преобразовать файлы стилей из

более современных форматов LESS/SASS в более поддерживаемые старыми браузерами CSS, либо JavaScript последних версий в ECMAScript 4 или 5.

NPM (Node Package Manager) – это менеджер пакетов для платформы Node.js. Он позволяет программисту управлять зависимостями в проекте, создавать свои пакеты, которые потом могут использовать другие разработчики, публиковать свои проекты с открытым исходным кодом. NPM является утилитой для командной строки. NPM устанавливается вместе с Node.js и управляет зависимостями в файле package.json. Внутри этого файла можно указать следующую конфигурацию проекта:

- name (название проекта);
- version (первоначальная версия проекта);
- description (описание);
- main (точка входа);
- scripts (текстовые команды для старта проекта, тестирования, для production и development версий);
- author (автор);
- keywords (ключевые слова);
- license (лицензия);
- dependencies (зависимости);
- devDependencies (зависимости, нужные для разработки проекта).

Ant Design – дизайн система для веб-приложений, разработанная китайской компанией Alibaba Group. Это очень удобная и расширяемая библиотека компонентов для React.js. Ant Design написан на TypeScript, для стилизации использовался Less и также существуют версии для Vue.js и Angular.js. В Ant Design

существует множество компонентов для таблиц (с пагинацией), кнопок, форм с валидацией, разметки, всплывающих окон, селектов, календарей и прочего [28].

Express.js – фреймворк для создания http сервера для Node.js. Express.js предоставляет множество служебных методов HTTP и промежуточных обработчиков (middleware). Данный фреймворк показывает лучшую производительность в realtime приложениях, так как задействуют push-технологию через веб-сокеты. В данной модели соединение клиент-сервер может инициировать сервер. Взаимодействие идет через стандартный порт 80. Node.js подходит для создания быстрых масштабируемых сетевых приложений, поскольку позволяет одновременно обрабатывать огромное количество соединений с большой пропускной способностью, что равноценно высокой масштабируемости. Node.js, на основе которого работает фреймворк Express.js не создает новый поток для каждого клиентского соединения, как это делают другие веб-фреймворки. Вместо этого он позволяет обрабатывать соединения в одном потоке, что позволяет обрабатывать десятки тысяч конкурентных соединений. Потенциально он может масштабироваться до миллиона конкурентных соединений [36].

Данные зависимости оказывают большую помощь при разработке и дальнейшей поддержке продукта.

Выводы по главе 1

В данной главе был проведен анализ предметной области. Данный анализ показал, что реализация данного веб-приложения с выбранными инструментами разработки является актуальной и представляет ценность для пользователей данной системы. Исходя из проведенного анализа, были выбраны следующие инструменты разработки:

- язык JavaScript и его расширение TypeScript для клиента и сервера;
- СУБД PostgreSQL;
- фреймворк для серверного JavaScript Node.js;
- NPM пакетный менеджер для Node.js;
- Express.js http фреймворк для Node.js;
- Ant Design библиотека компонентов;
- Webpack – утилита для сборки клиентской части веб-приложения.

Данный стек инструментов разработки является довольно популярным, что гарантирует обширную обратную связь от сообщества при возникновении проблем. Это является важным критерием при выборе инструментов. Также стек является простым, легковесным и имеет низкий порог вхождения в технологии, что сокращает время на разработку [1].

Минимальная функциональность информационной системы для автоматизации медицинской клиники должна включать в себя:

- инструменты для создания, редактирования, поиска, сортировки и удаления профилей пользователей;
- создание, редактирование, удаление и поиск записей на прием;
- создание, редактирование, удаление и поиск процедур докторов;
- создание, редактирование, удаление и поиск локаций;
- создание, редактирование, удаление и поиск комнат;
- создание, редактирование, удаление и поиск департаментов клиники.

Главное преимущество автоматизации медицинской клиники – ускоренное оформление новых пациентов, легкий поиск по записям на прием, пациентам, процедурам и другим сущностям системы, устранение вероятностей возникновения

ошибок при обработке данных пациента, уменьшение затрат на хранение информации о клиентах клиники. Все это будет являться причиной уменьшения затрат предприятия в целом.

Еще одним преимуществом является сокращение затраченного времени и сил персонала на поиск и обработку информации. Занятость и силы персонала являются ключевым фактором, так как они сказываются на оказании медицинских услуг и могут нарушать концентрацию врачей.

В настоящее время существует большое количество решений информационных систем автоматизации медицинских клиник, каждое из которых имеет свои достоинство и недостатки. Каждая конкретная медицинская клиника выбирает решение исходя из особенностей своей организации. В критерии отбора включены цена, наличие мобильного приложения, обширность функционала, поддержка системы электронных платежей и др. Каждый из этих параметров является опциональным для разных клиник и не всегда важен в наличии.

Современные стандарты веб-приложений диктуют наличие интуитивно понятного интерфейса. Существующие крупные приложения являются простыми и понятными, соответственно разрабатываемый продукт должен соответствовать этим критериями, для того, чтобы не создавать пользователю дискомфорт.

2. ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ МОДЕЛЕЙ И РАЗРАБОТКА ИНФОРМАЦИОННОЙ СИСТЕМЫ ВЗАИМОДЕЙСТВИЯ МЕДИЦИНСКИХ КЛИНИК С КЛИЕНТАМИ

2.1 Выбор методов и средств проектирования

Создание современных информационных систем – трудоемкий процесс, для эффективной разработки программных комплексов необходимо знать о методах

проектирования, реализации и тестирования ПО. Проектирование ПО включает в себя:

- проектирование структуры ПО – разделение на основные части (компоненты) и их взаимосвязи между собой;
- декомпозиция и построение структурной иерархии в соответствии с блочно-иерархическим подходом;
- проектирование отдельных компонентов.

Проектирование программного обеспечения – это процесс создания проекта программного обеспечения, оценки и выбора средств для его реализации. Проектирование ПО включает следующие процессы:

- выбор методов и средств реализации;
- выбор моделей представления внутренних хранимых данных;
- разработку алгоритма работы;
- документирование;
- написание тестов и тестирование;
- выбор представления входных данных и выходных данных.

Для проектирования программного обеспечения необходимо выбрать методы проектирования. В данной работе выбор остановился на макете пользовательского интерфейса, UML-диаграмме для проектирования взаимодействие классов и компонентов, ER-диаграмме для проектирования базы данных и блок схемах для проектирования алгоритма работы программы.

Блок схема – схематическое представление системы, процесса или алгоритма действий. Применение блок-схем можно найти в различных областях знаний, чтобы описывать, изучать, планирования или объяснять сложные процессы. Для

построения блок-схем применяются прямоугольники, ромбы, овалы и др, а также стрелки для указания последовательностей шагов. Для нотации блок схем применяют следующие названия: процесс, начало/конец, документ, решение, соединитель, межстраничный соединитель, ввод/вывод, комментарий [29].

ER-диаграмма (диаграмма сущность-связь) – это разновидность блок-схемы, которая показывает взаимодействие сущностей между собой. Под сущностью понимает объект людей природы – человек, таблица в базе данных, концепция. Сущность имеет набор свойств, которые ее описывают. ER-диаграммы чаще всего применяются в проектировании и разработке баз данных, так как ER модель отражает только реляционную структуру, то есть структуру, основанную на отношениях объектов между собой. Существует 3 типа сущностей в ER диаграмме:

- сильная сущность (не зависит от других сущностей, подчиняет себе слабые сущности) и имеет первичный, который однозначно идентифицирует ее;
- слабая сущность (зависит другой сущности, имеет внешний ключ);
- ассоциативная сущность (соединяет сущности между собой.

Существует 2 вида связей, которые выражают отношения между сущностями:

- связь (отношение между сущностями);
- слабая связь (сущность между зависимоей сущностью и ее “хозяином”) [30].

UML-диаграмма (Unified Modeling Language) – это унифицированный язык моделирования. Моделирование отвечает за создание модели, описывающей объект. UML подходит для широкого класса проектируемых программных систем, различных областей приложений, типов организаций, уровней компетентности, размеров проектов [7].

2.2 Логическое проектирование системы

Логическое проектирование состоит из проектных операций, которые непосредственно не зависят от имеющихся технических и программных средств, которые составляют среду функционирования будущего продукта.

SPA (Single Page Application или одностраничное приложение) – это архитектуру веб-приложений, при которых загрузка веб-страницы происходит один раз и далее необходимая информация обновляется по запросу на клиенте путем взаимодействия с сервером в формате JSON, то есть в отличие от MPA (Multi Page Application), HTML страницы не загружаются по мере обращения к ним. SPA является достаточно новым и современным подходом. Он помогает снижать нагрузку на сервер и уменьшает время взаимодействия с клиентом. SPA позволяет не обновлять страницы при переходе по ним, так как все страницы загружаются один раз при запросе к главной странице [4]. Как уже говорилось, клиент общается с сервером по протоколу HTTP форматами данных JSON. Это позволяет упростить разработку и избавиться от модели шаблонов, когда веб-страницы динамически генерируются на стороне сервера и возвращаются в готовом виде клиенту. Расширение SPA архитектуры является новый подход – Server Side Rendering (SSR) [5]. После загрузки начальной веб-страницы одностраничного веб-приложения, главная страница начинает загрузку данных через API JavaScript скриптами, содержащимися на веб-странице. SSR подход позволяет это сделать на стороне сервера, чтобы пользователь не ждал окончания загрузки на стороне клиента. Это одни из тех причин, почему SPA приложения работают быстрее классических MPA приложений. На рисунке ниже представлена модель взаимодействия клиента с веб-серверов.

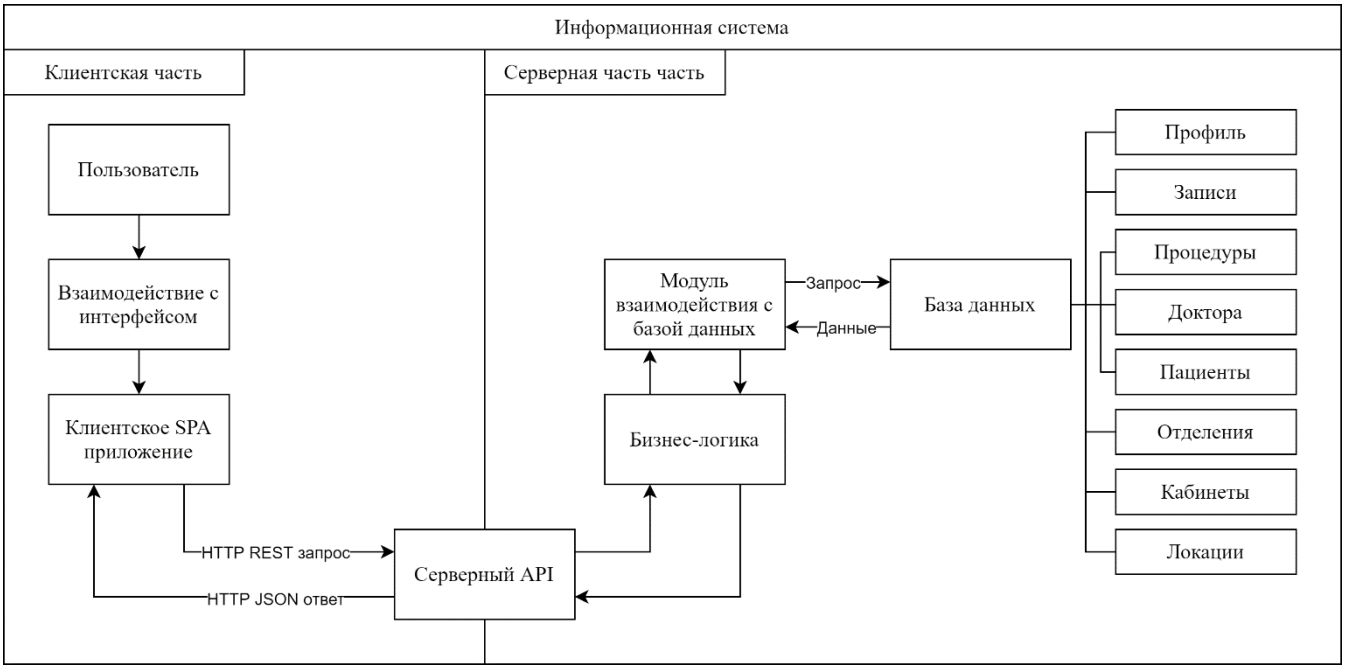

Рисунок 8. Архитектура взаимодействия клиентской и серверной части

На следующем рисунке показана UML диаграмма модуля взаимодействия с базой данных. Модуль состоит из нескольких пакетов:

- Server (в нем хранится серверное http приложение, которое обрабатывает запросы, управляет бизнес-логикой приложения и отдает ответы);
- Storage (в этом пакете хранятся модули взаимодействия с базой данных – модуль создания клиентского экземпляра базы данных, чтения конфигурационного файла базы данных и сам конфигурационный файл);
- PostgreSQLQuery (в нем содержатся классы, которые реализуют SQL запросы к конкретной базе данных).

Модуль storageInterface реализован с применением паттерна проектирования Singleton. Этот паттерн часто используется для создания подключений к базе данных, так как предполагает, что создается только один экземпляр данного класса через статический метод класса, в котором реализуется проверка: если существует экземпляр данного класса – вернуть этот экземпляр, иначе вернуть новый.

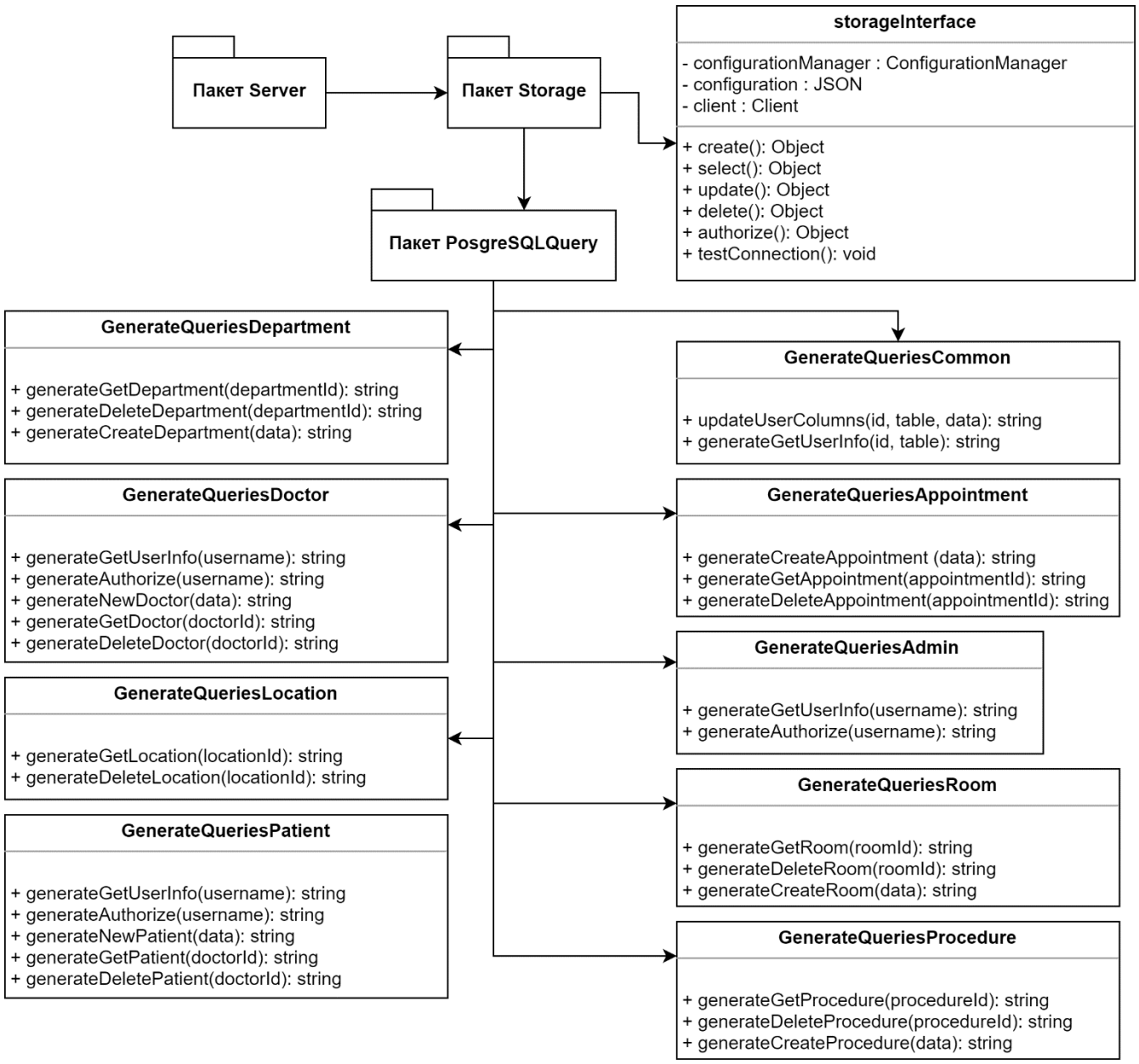

Рисунок 9. Структура классов и модулей в пакете Server

2.3 Физическое проектирование системы

В отличие от логического проектирования, физическое проектирование заключается в привязке к конкретным техническим и программным средствам функционирования. Под базой данных понимается форма представления и

организации набора данных. База данных систематизирует некоторую область знаний и обрабатывает их с помощью вычислительной системы [6].

Как уже говорилось, для хранения информации была выбрана СУБД PostgreSQL. Для нее были спроектированы модели хранения данных и выстроена структура базы данных. База данных состоит из 8 таблиц: Doctor, Patient, Admin, Appointment, AppointmentProcedure, Department, Location, Room. Сущности Admin, Doctor и Patient имеют общие свойства. Данные свойства программно были вынесены в таблицу User, а таблицы Admin, Doctor, Patient унаследовали эти свойства. Это является удобным способом избавиться от дубликации данных при создании таблицы. На следующем рисунке показана схема базы данных.

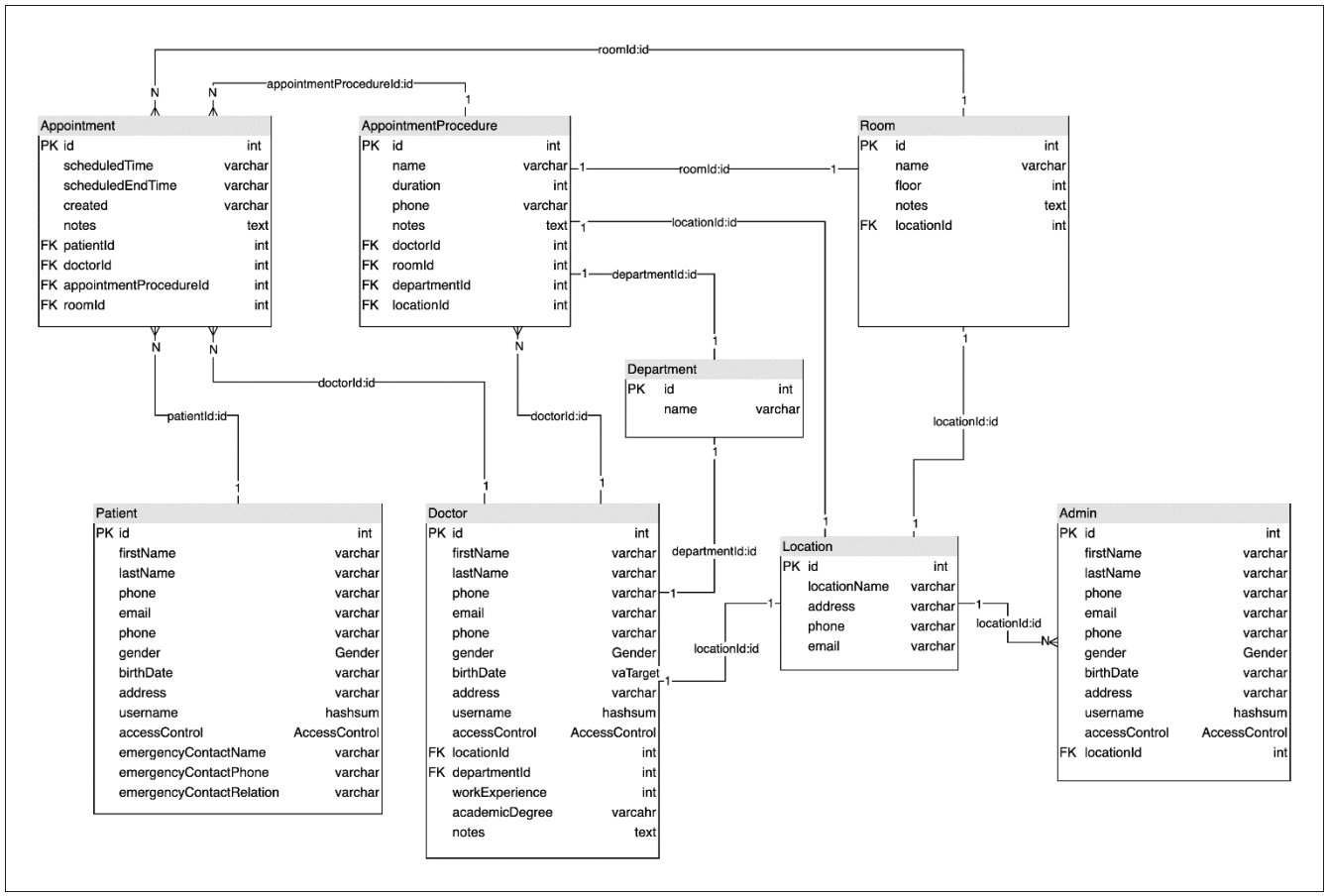

Рисунок 10. Схема базы данных

2.4 Проектирование макета пользовательского интерфейса

Для качественного проектирования пользовательского интерфейса выделим основные действия разных ролей пользователей.

Таблица 1. Возможности влиять на состояние системы в зависимости от роли

	Админ	Доктор	Пациент
Личный профиль	Изменять Смотреть	Изменять Смотреть	Изменять Смотреть
Записи на прием	Создавать Изменять Смотреть	Смотреть (только свои)	Смотреть (только свои)
Процедуры	Создавать Изменять Смотреть	Смотреть (только свои)	Смотреть (только свои)
Пациенты	Создавать Изменять Смотреть	Смотреть (только свои)	-
Доктора	Создавать Изменять Смотреть	-	-
Локации	Создавать Изменять Смотреть	Смотреть	Смотреть
Кабинеты	Создавать Изменять Смотреть	Смотреть	Смотреть
Отделения	Создавать Изменять Смотреть	Смотреть	Смотреть

Далее приведены спроектированные макеты пользовательского интерфейса, по которым создавался дизайн и сервис.

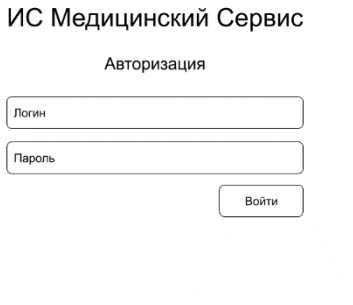

Рисунок 11. Страница входа на сервис ИС Медицинский Сервис

Рисунок 12. Страница просмотра и изменения профиля

На макетах далее рассмотрен каркас интерфейса календаря в разных режимах. Данный календарь доступен для просмотра всем видам пользователей, но Доктор и Пациент не могут вносить правки в созданные записи на прием, а могут

лишь просматривать в удобном режиме приемы с возможность нажать на активный прием и просмотреть подробности о нем. Также Пациент и Администратор имеет возможность производить фильтрацию записей на прием над календарем. Пациент имеет возможность фильтровать только по докторам, которые назначили ему прием – администратор, напротив, имеет возможность просматривать записи на прием у всех докторов. У пользователя с ролью Доктор эта возможность отсутствует, так как он не имеет прав для просмотра записей других докторов по вопросам безопасности. По умолчанию для него выбраны поля для селекта “Департамент” и “Доктор”, он может лишь выбирать процедуры. Для просмотра доступно несколько режимов – режим “Месяц”, “Неделя” и “День”, а также отдельный режим “Распис.” (Расписание) для удобного просмотра записей в виде списка и возможностью распечатки страницы на принтере.

В верхнем левом углу календаря есть возможность удобной навигации по календарю “Сегодня”, “Назад” и “Вперед”.

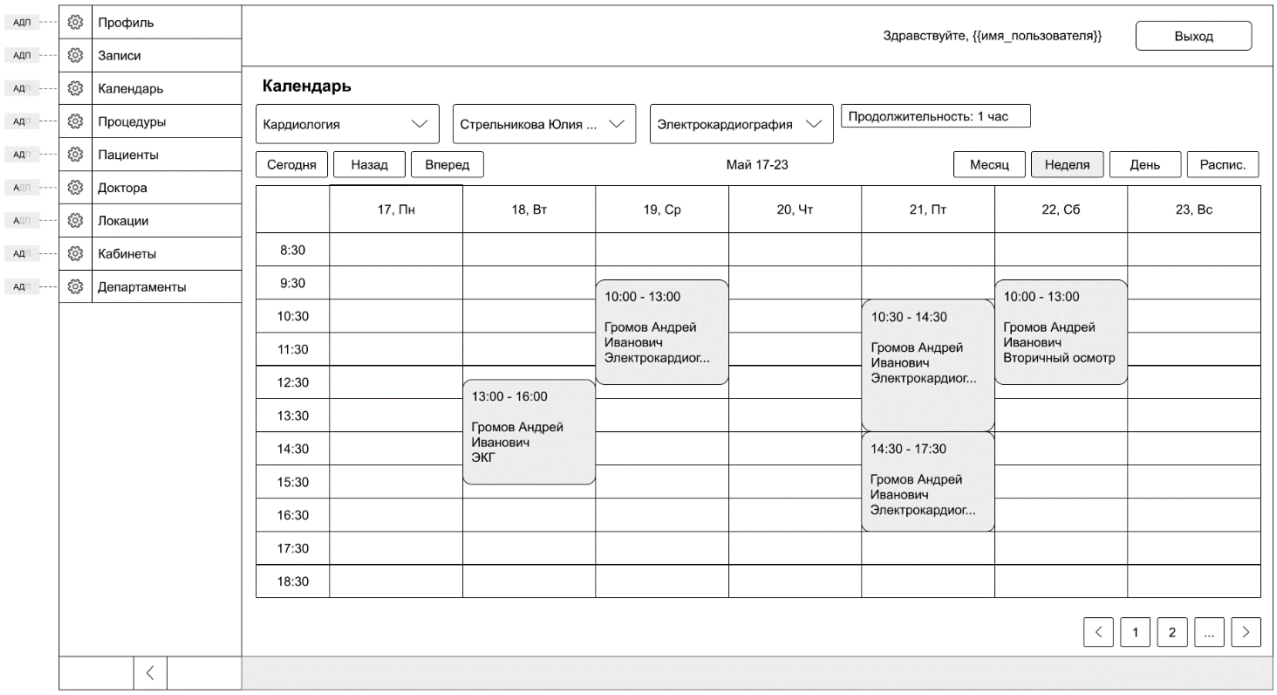

Рисунок 13. Страница записей на прием (неделя)

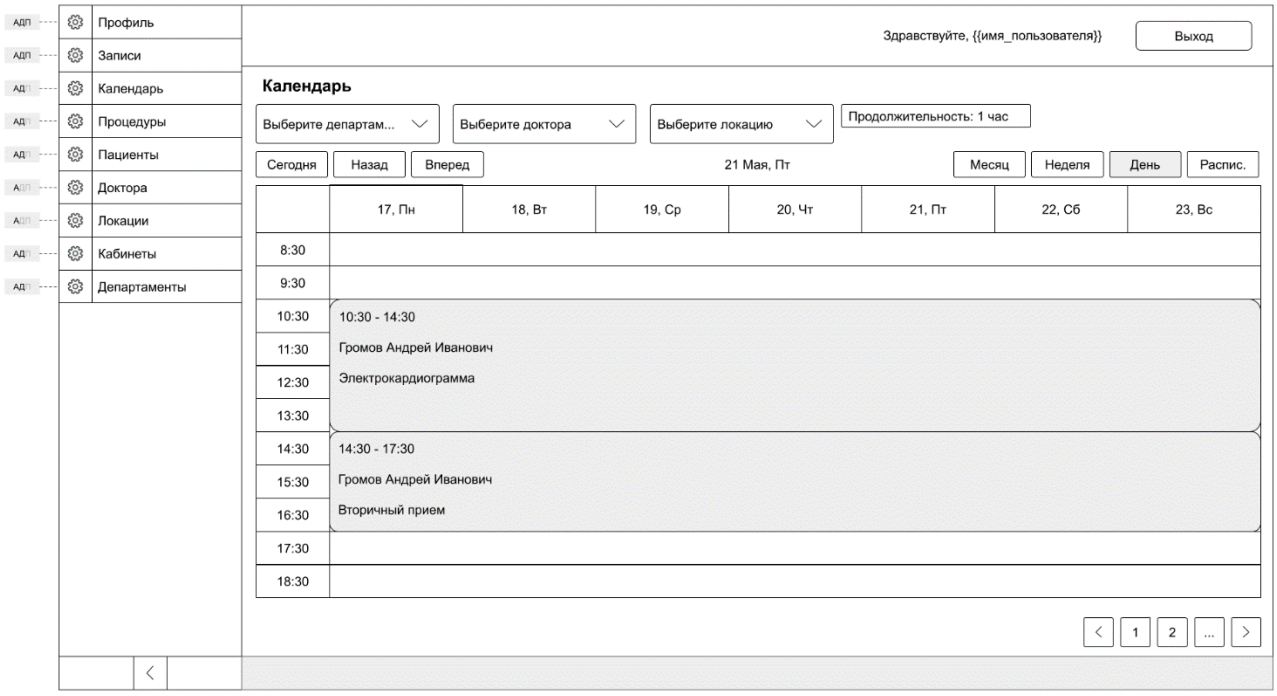

Рисунок 14. Страница создания записей на прием (день)

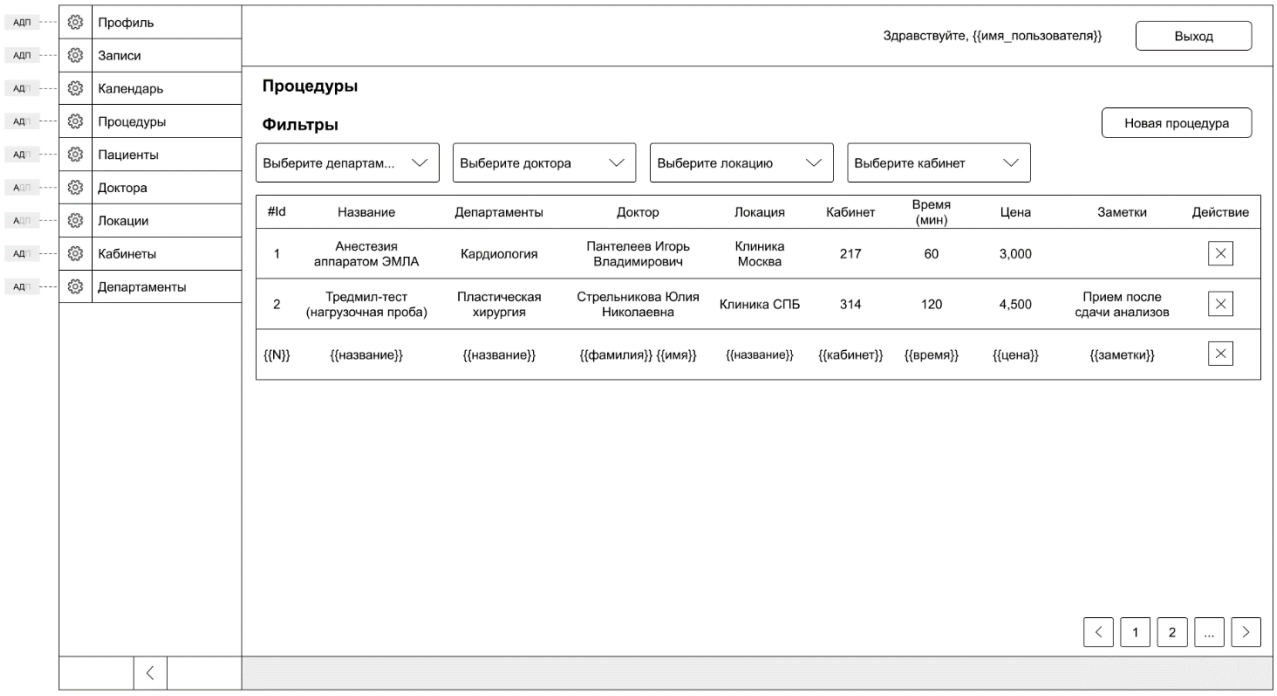

Рисунок 15. Страница просмотра и создания процедур

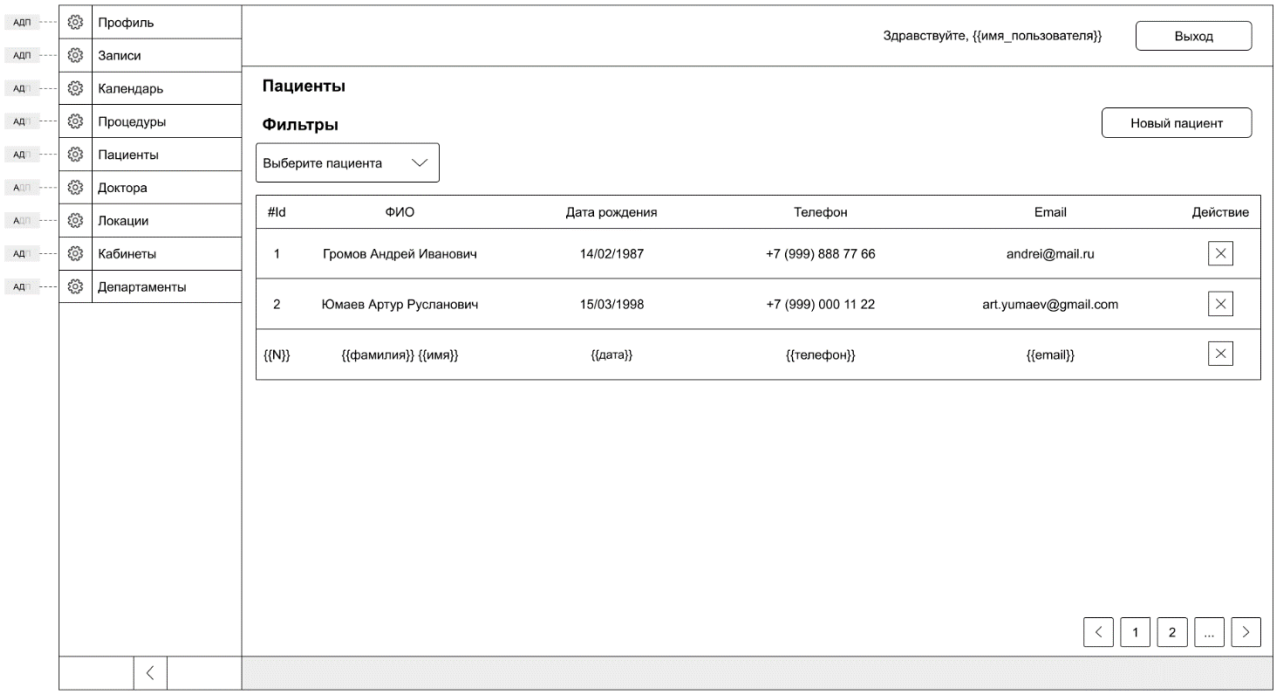

Рисунок 16. Страница создания и просмотра пациентов

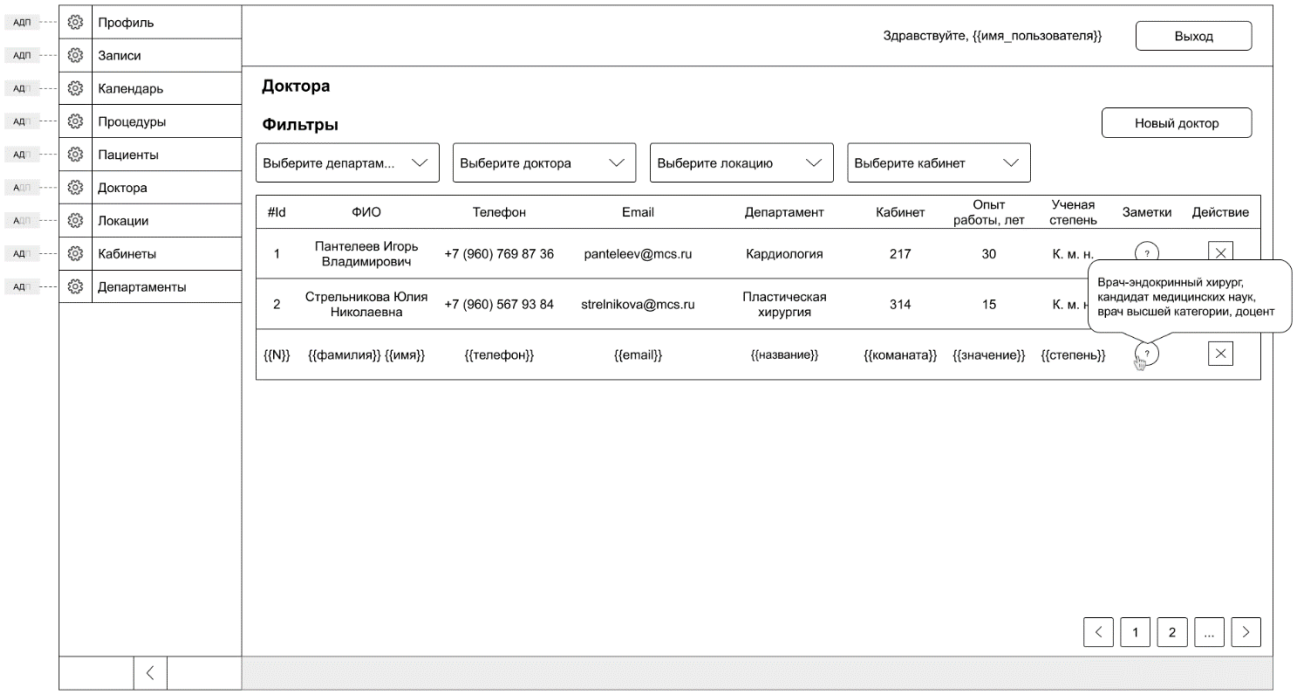

Рисунок 17. Страница создания и просмотра докторов

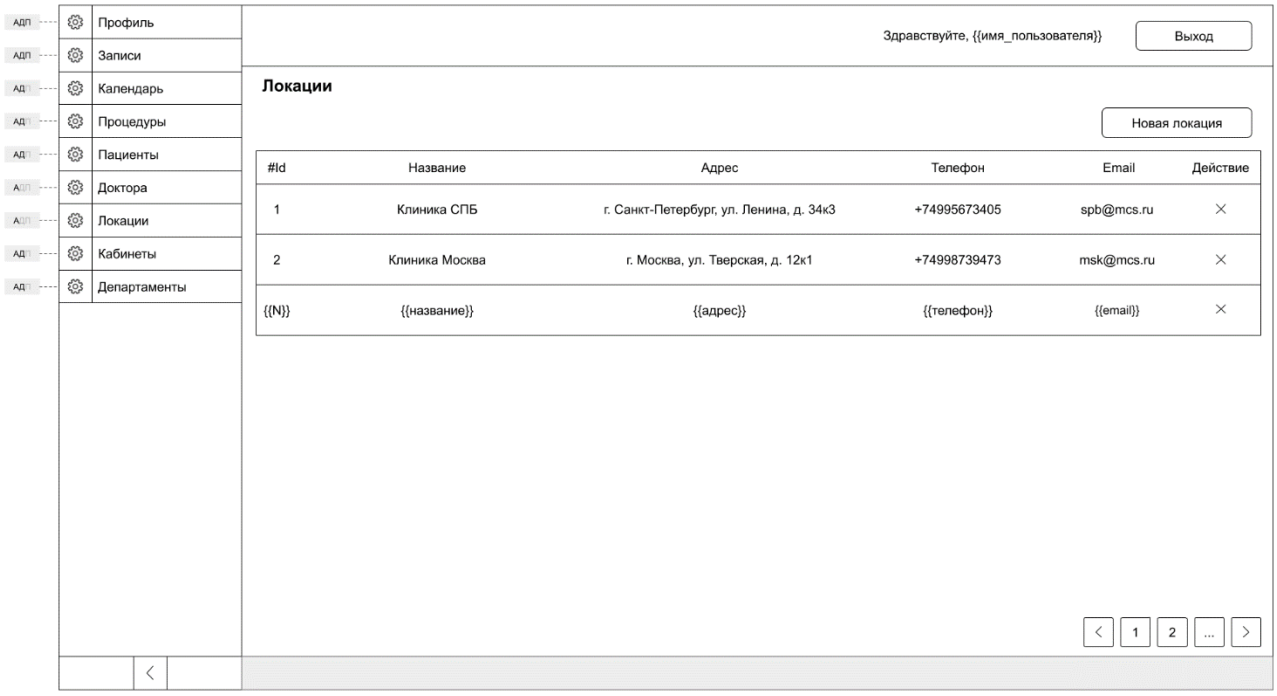

Рисунок 18. Страница создания и просмотра локаций

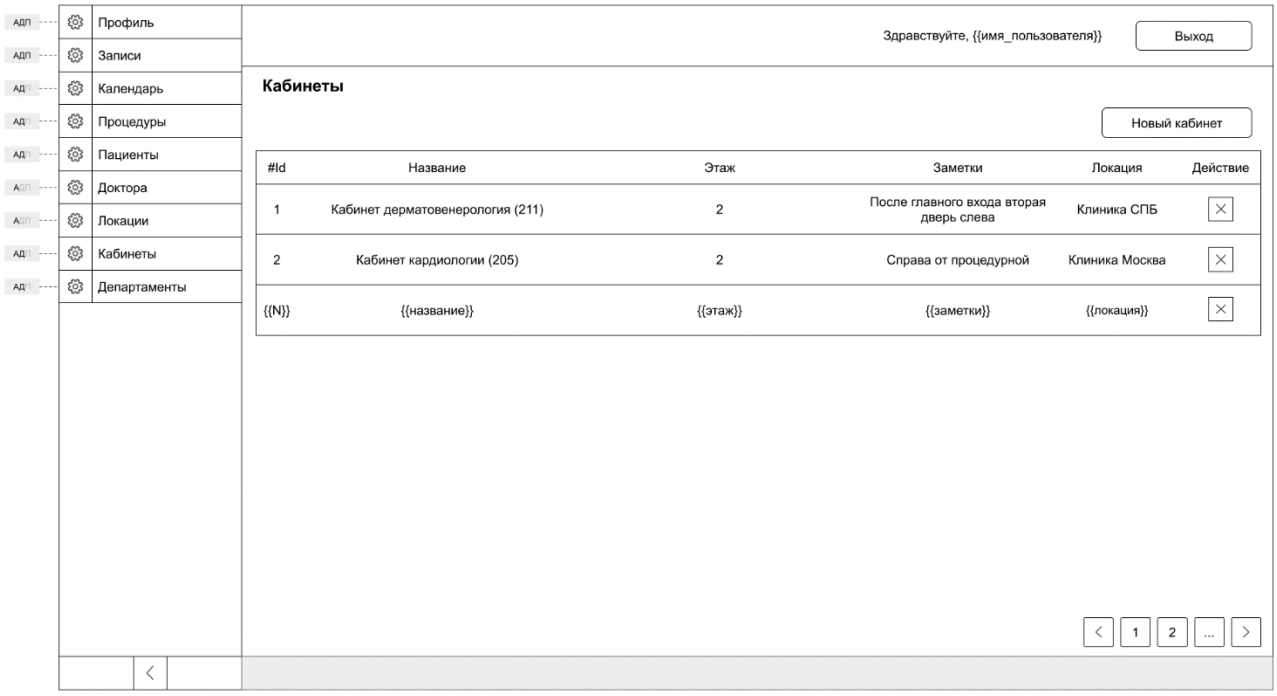

Рисунок 19. Страница создания и просмотра кабинетов

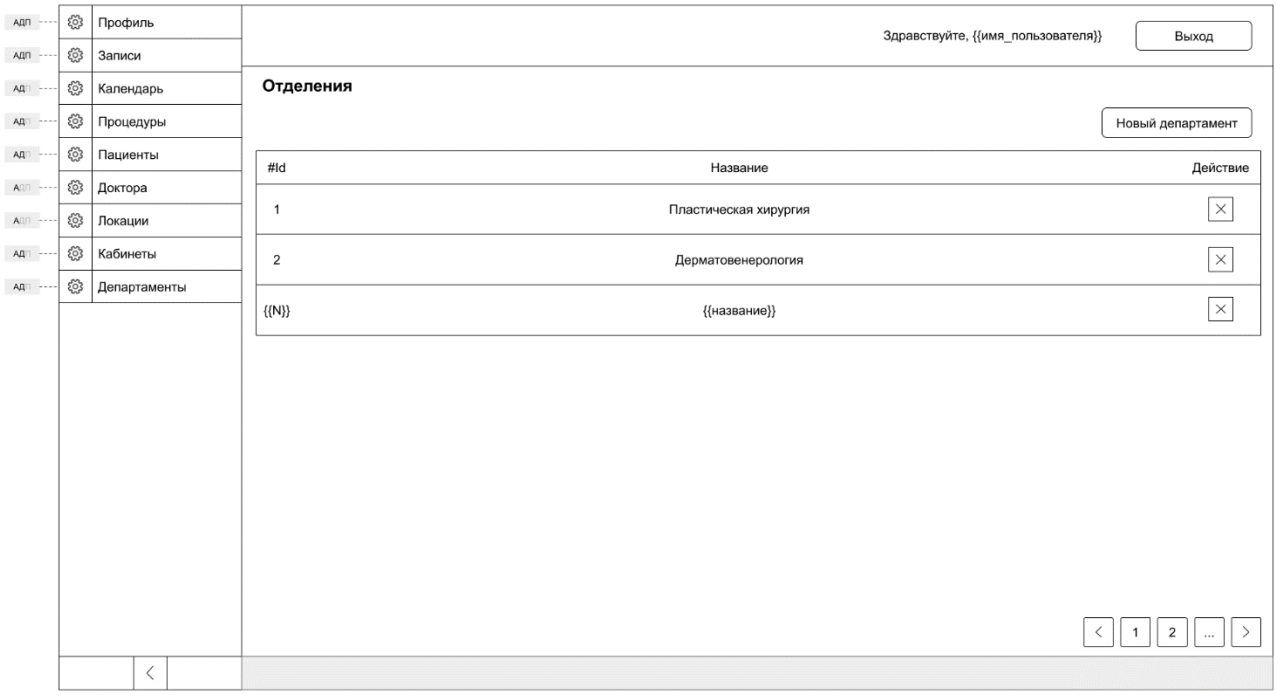

Рисунок 20. Страница создания и просмотра отделений

На макетах пользовательского интерфейса слева для каждой вкладки указаны роли, которые могут просматривать данную вкладку А – Администратор, Д – Доктор, П – Пациент. К примеру для вкладки “Календарь” указано “АД” – это означает, что вкладку могут просматривать только Администратор и Доктор. Для вкладки “Доктора” указано только “А”, это означает, что только Администратор может видеть список всех докторов, так как ни Пациент, ни другой Доктор не может видеть личную информацию о Докторах в системе. То же самое для вкладки “Пациент”, только Администратор может видеть список всех пациентов, Доктор видеть список только своих Пациентов, а Пациент не может видеть список других Пациентов. Подробные ограничения на просмотр сущностей в системе указан выше в таблице 1.

2.5 Описание конечного вида интерфейса

Как было показано в макете пользовательского интерфейса, интерфейс веб-приложения состоит из нескольких частей: Профиль, Записи, Календарь, Процедуры, Пациенты, Доктора, Локации, Кабинеты, Отделения. Это минимальный набор сущностей системы, которые могут покрыть медицинское учреждение или предприятие, предоставляющее медицинские услуги или СПА. Информационная система может дорабатываться под конкретного заказчика с небольшими изменения, доработками и расширением функционала, других сущностей и интерфейса. На рисунках далее приведен конечный интерфейс информационной системы.

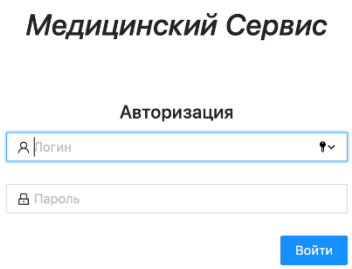

Рисунок 21. Страница входа в информационную систему

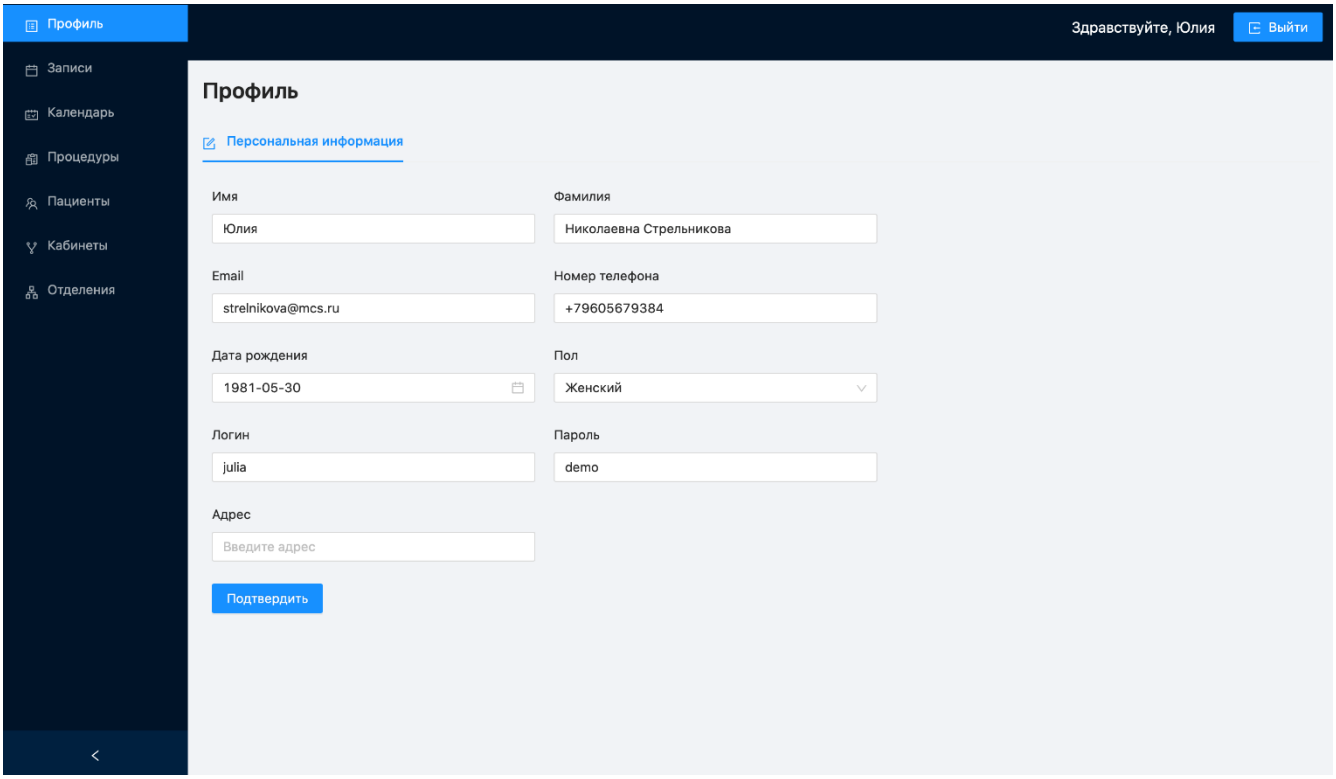

Рисунок 22. Страница просмотра и редактирования информации о пользователе (на примере личного кабинета врача)

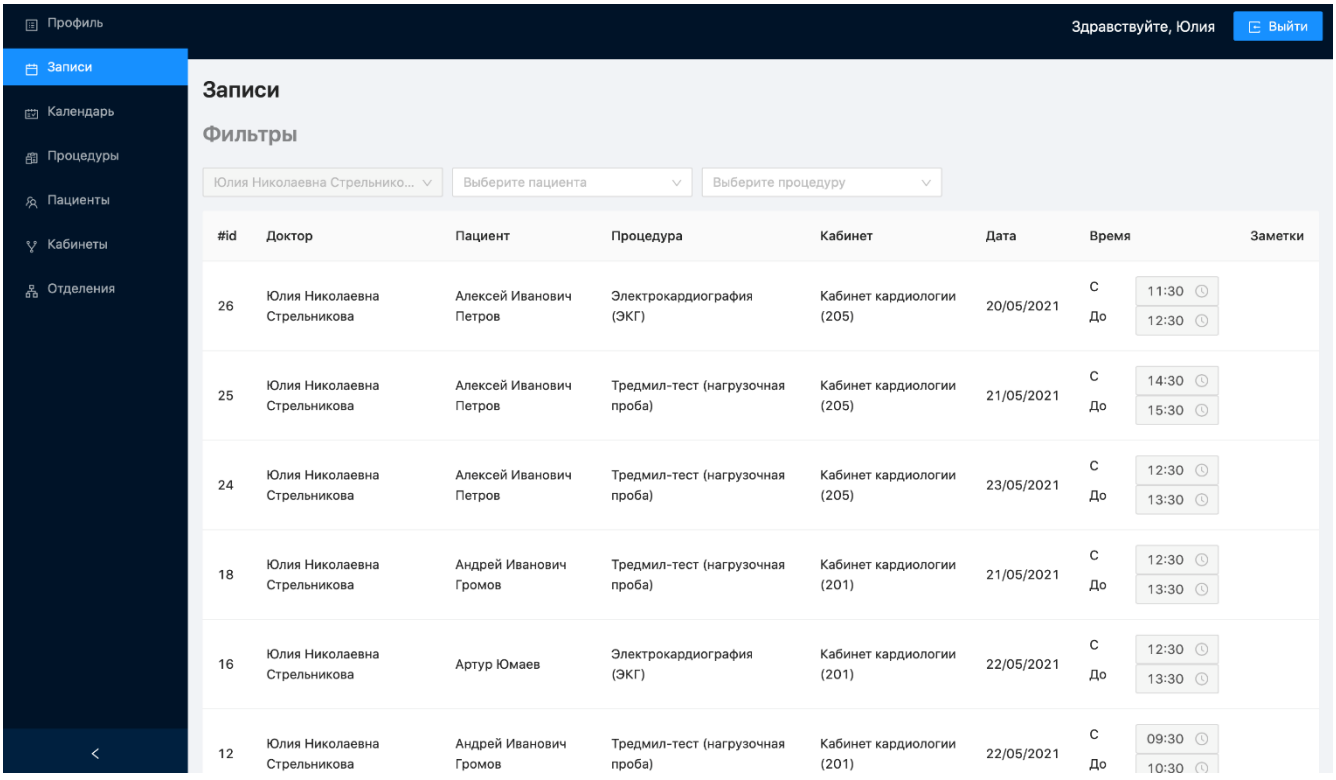

Рисунок 23. Страница просмотра записей на прием (врач)

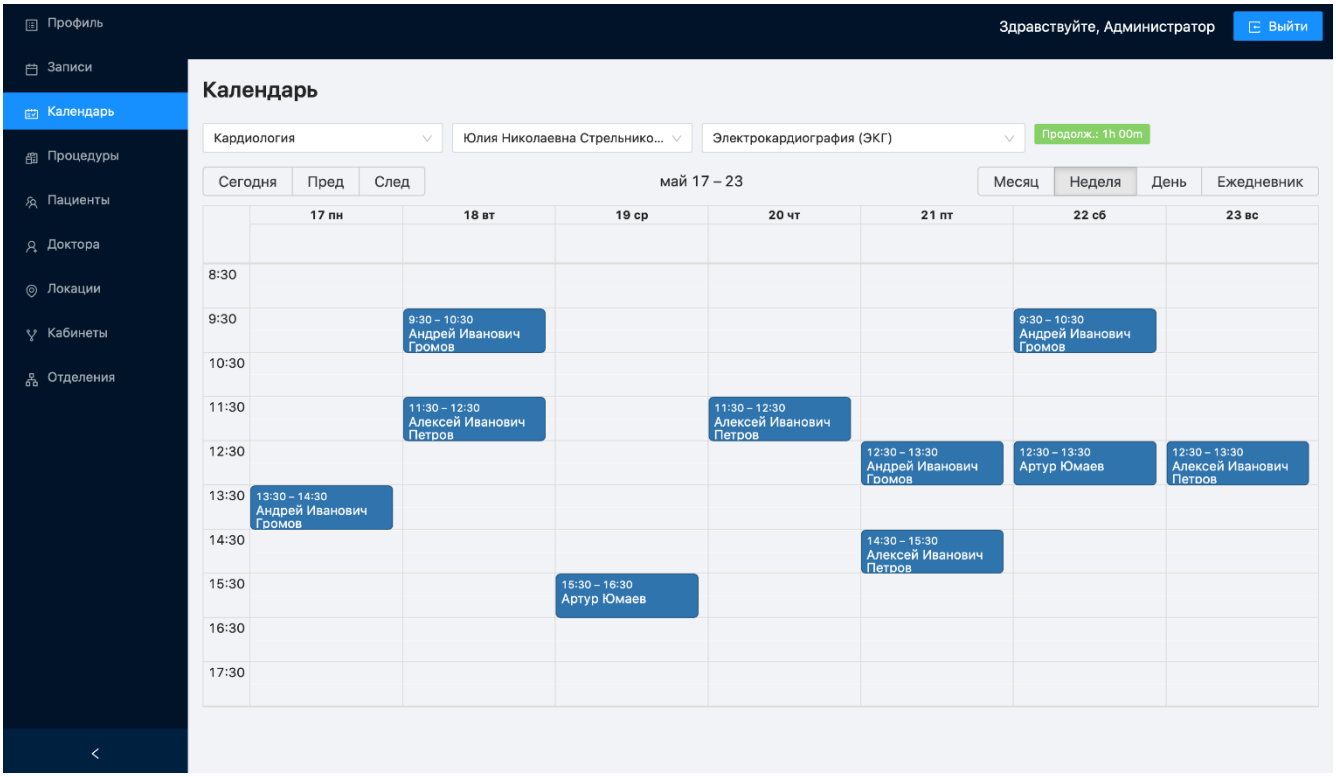

Рисунок 24. Страница календаря – просмотр, создание и удаление записей (администратор)

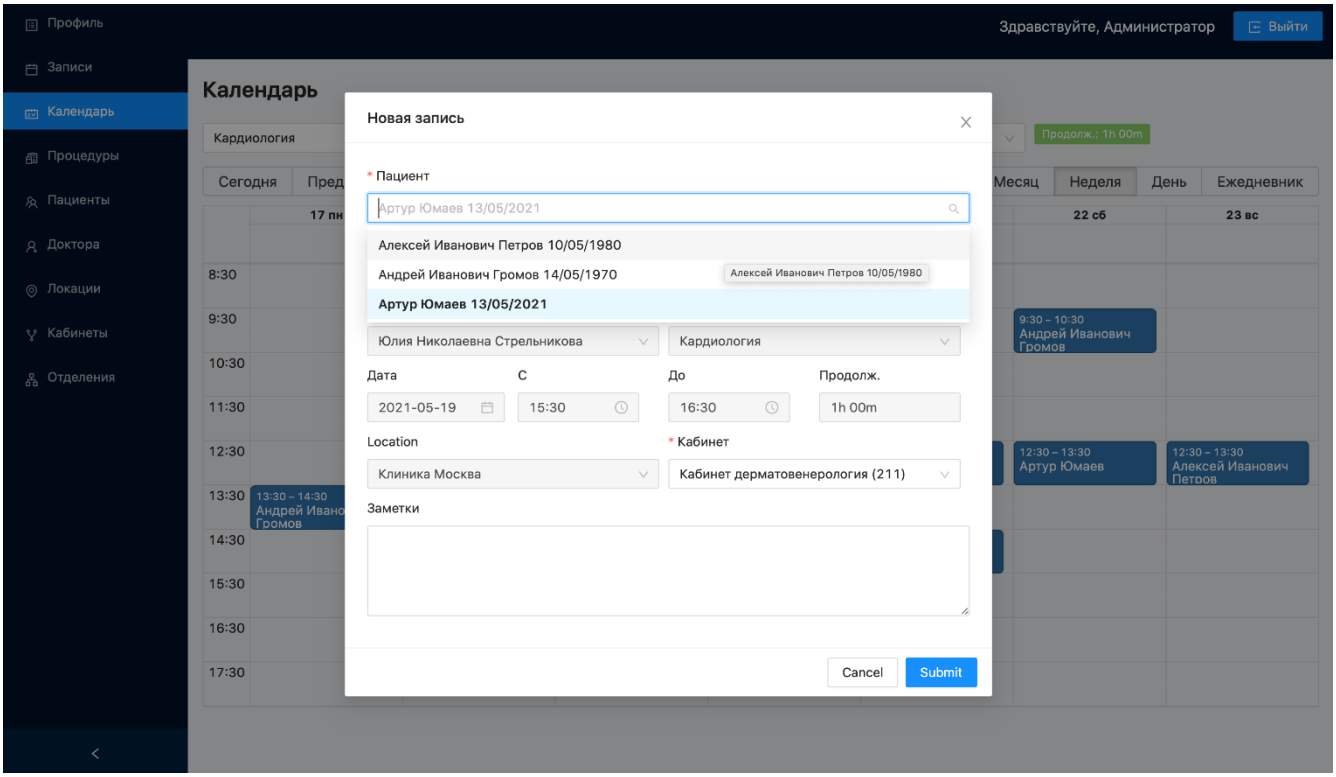

Рисунок 25. Страница записей – создание новой записи (администратор)

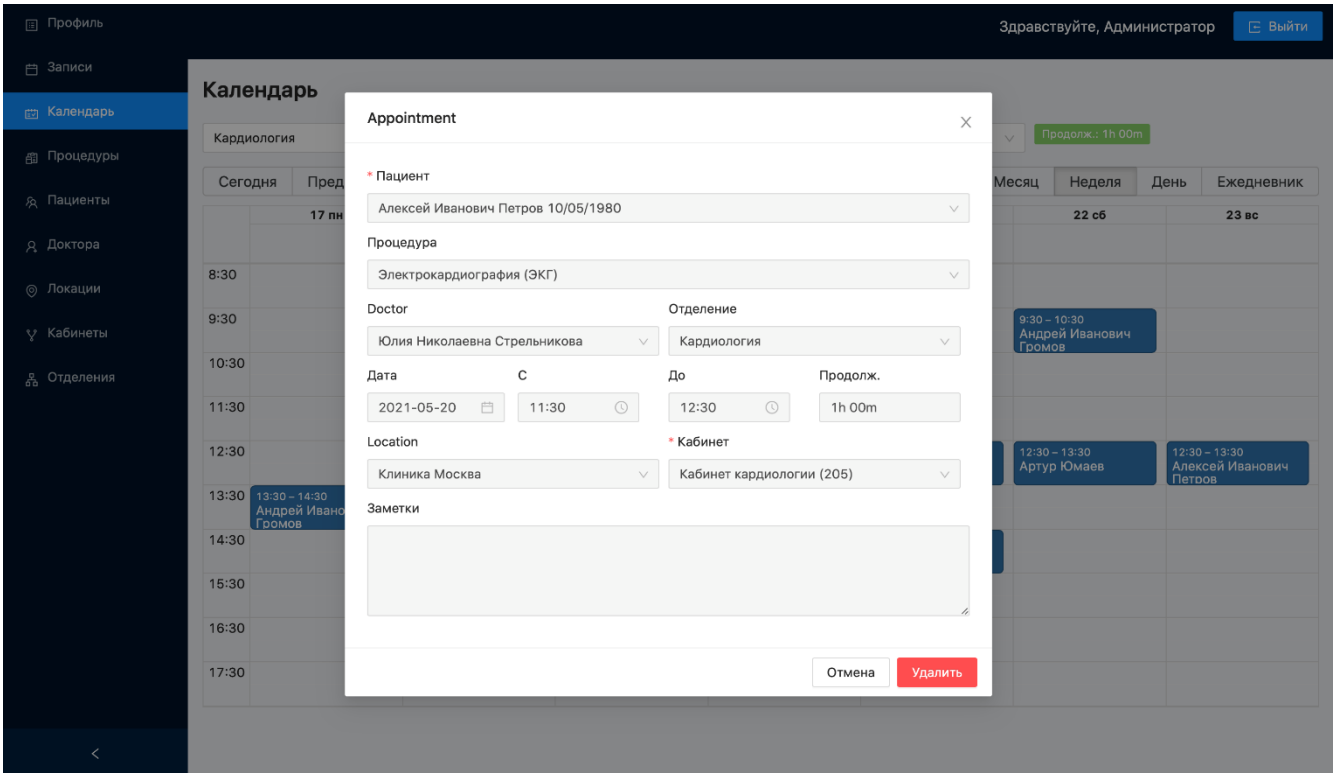

Рисунок 26. Страница записей – удаление записи (администратор)

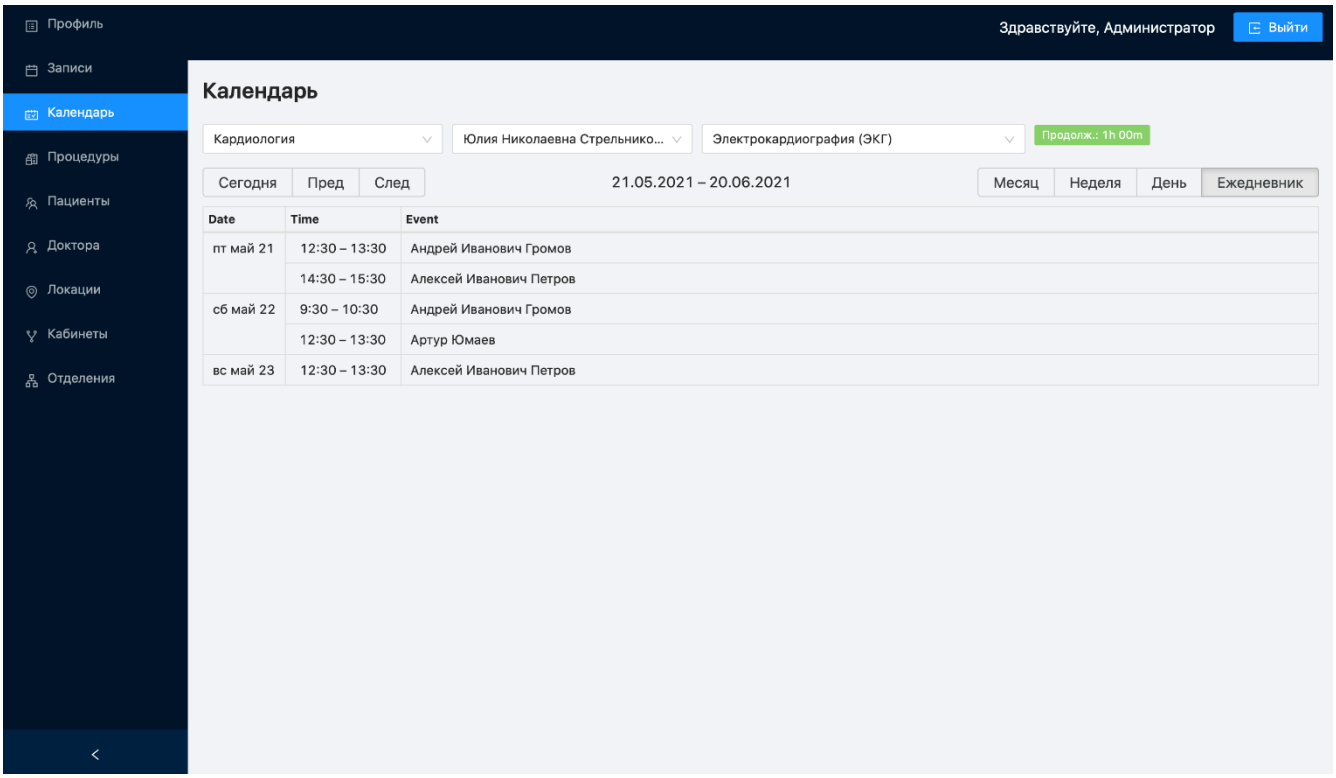

Рисунок 27. Страница удобного просмотра расписания

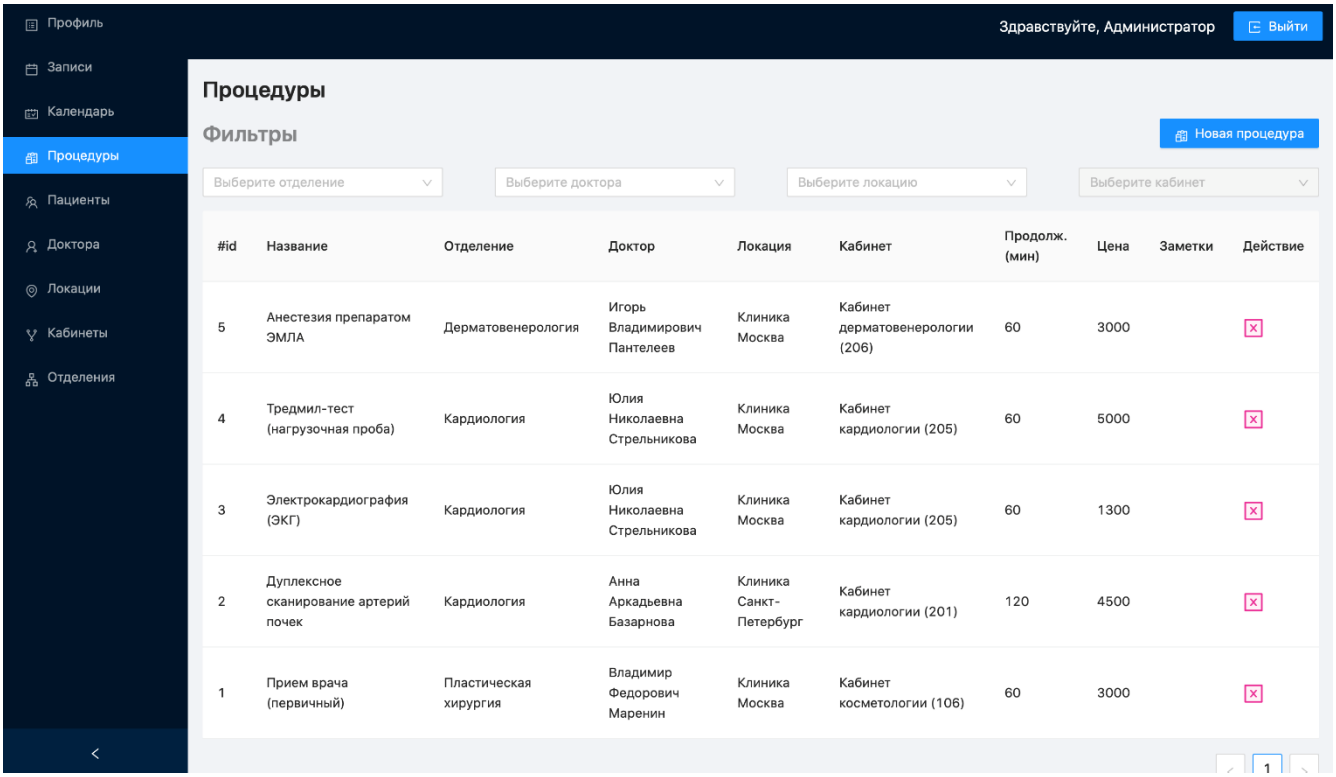

Рисунок 28. Страница процедур (администратор)

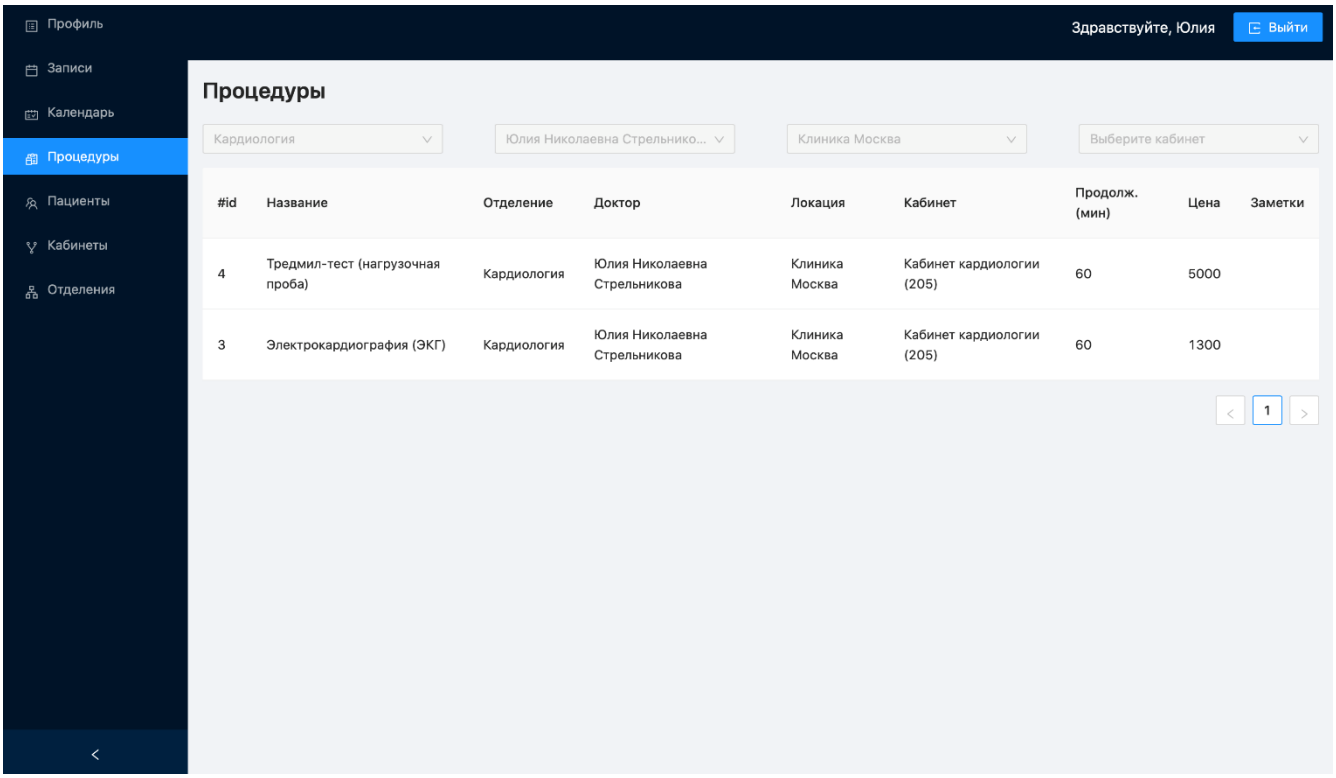

Рисунок 29. Страница процедур (врач)

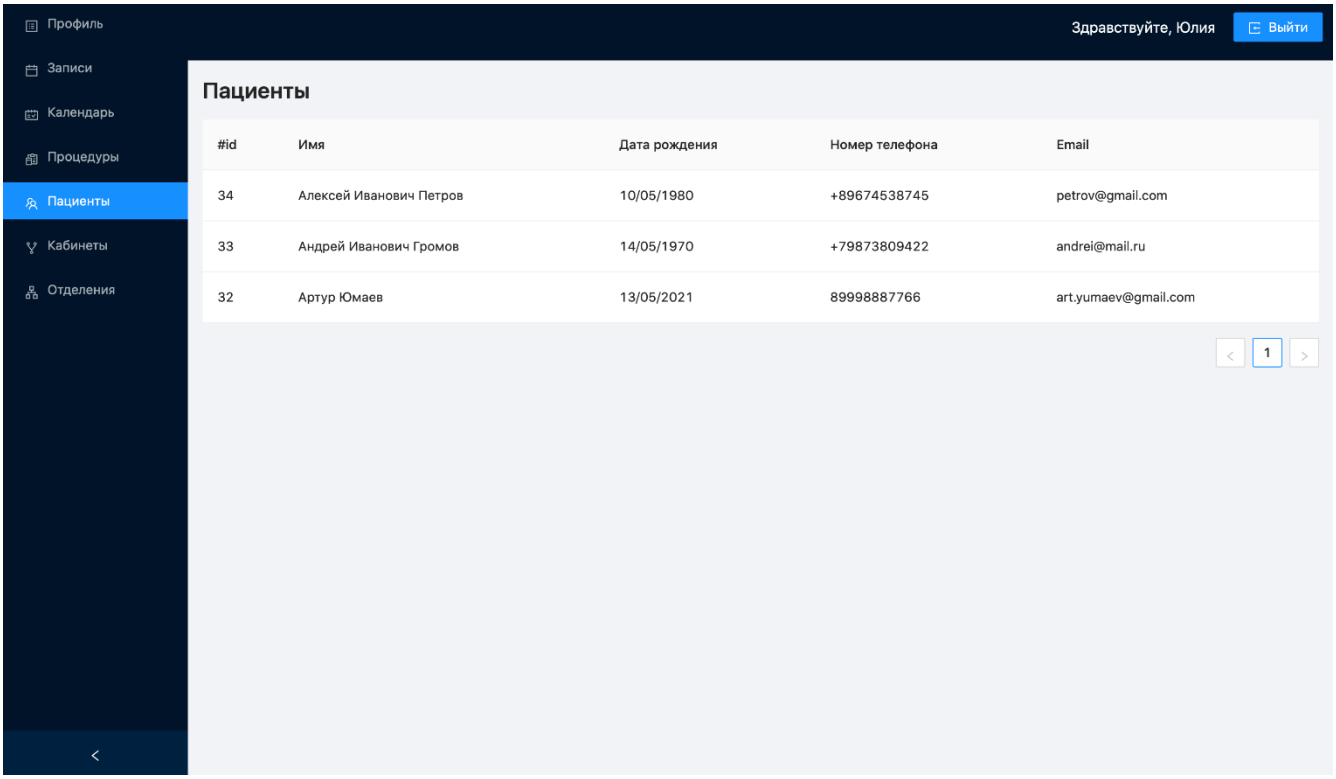

Рисунок 30. Страница пациентов (врач)

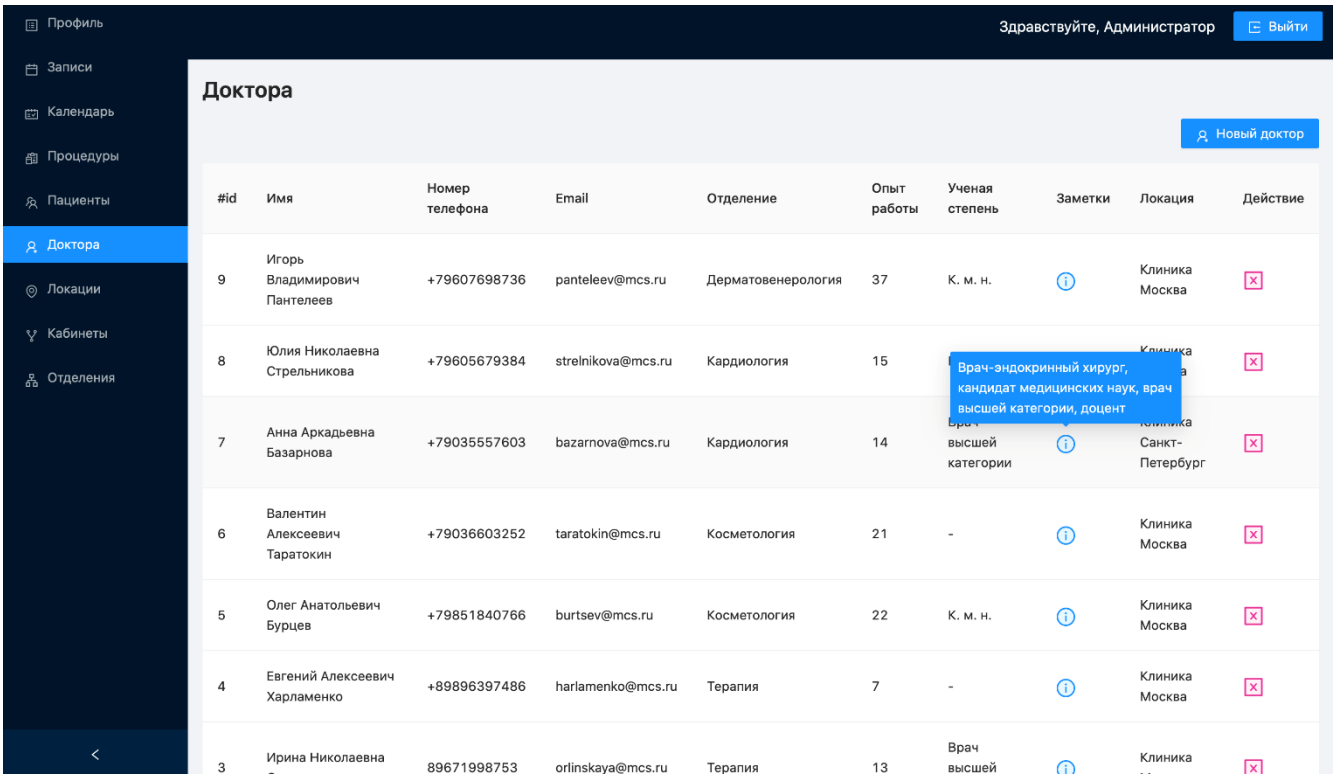

Рисунок 31. Страница просмотра врачей (администратор)

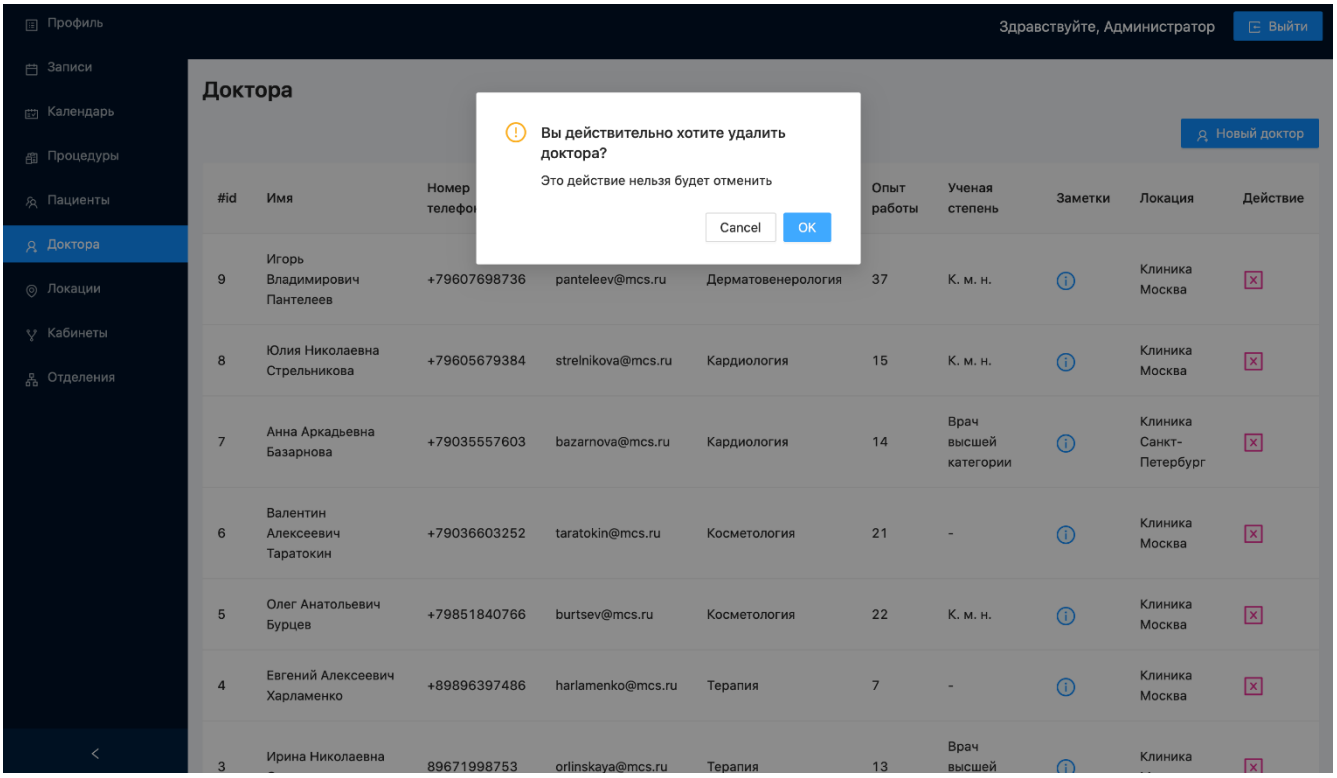

Рисунок 32. Страница врачей (возможность удаления врача)

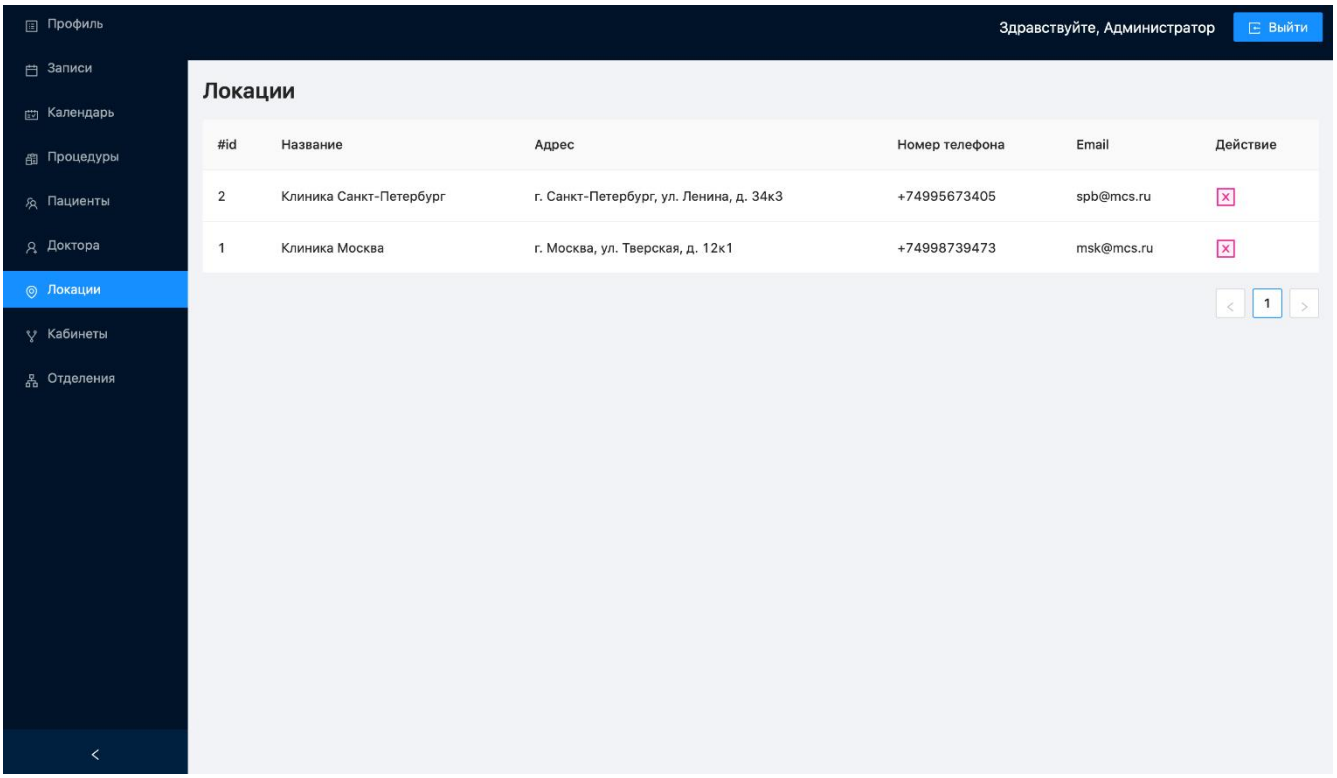

Рисунок 33. Страница локаций (администратор)

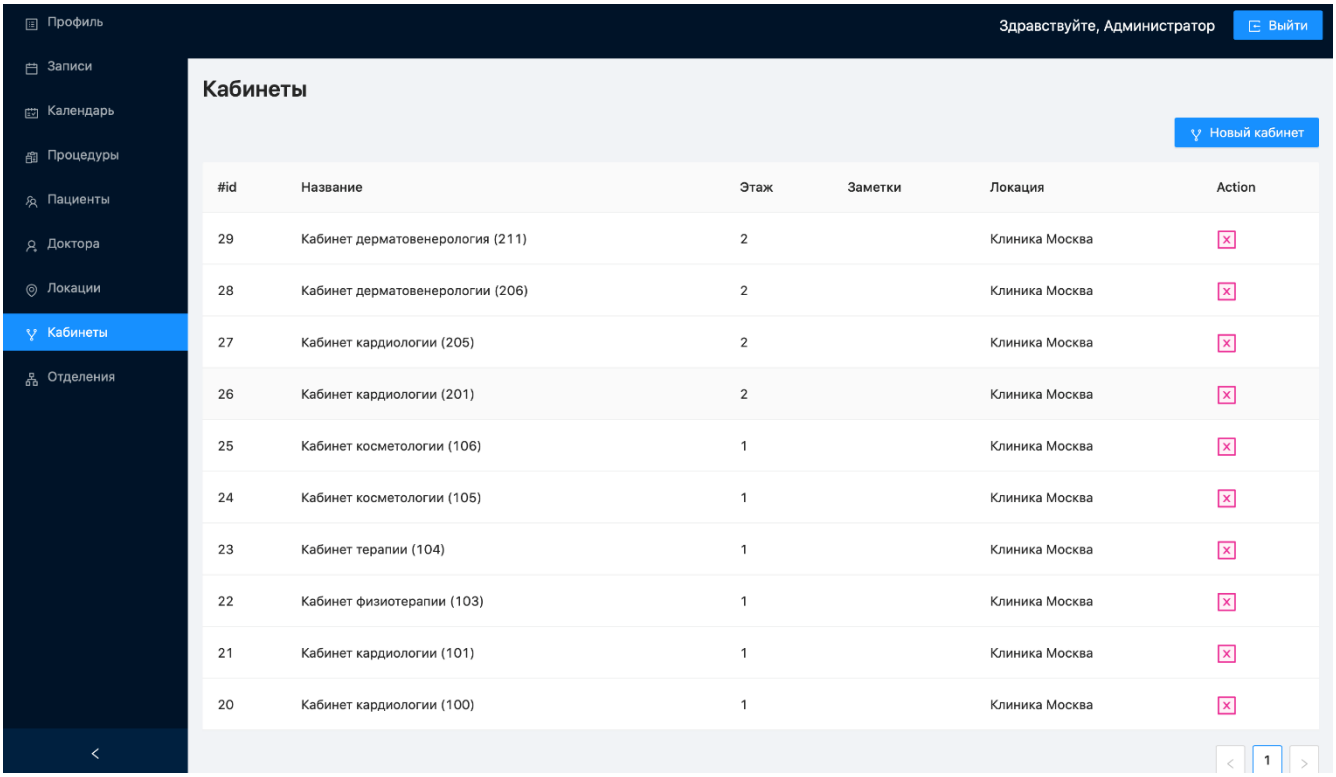

Рисунок 34. Страница кабинетов (администратор)

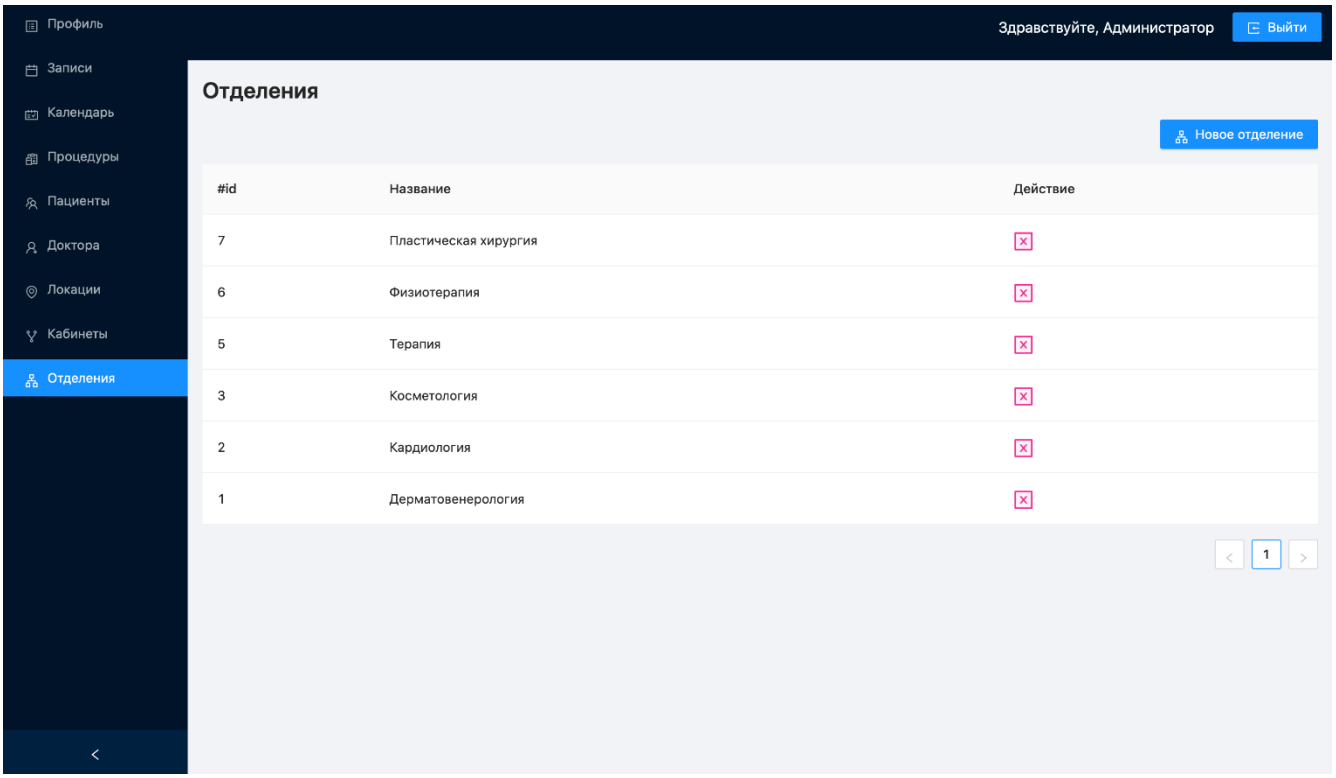

Рисунок 35. Страница отделений (администратор)

На рисунке далее изображено письмо, которое получает новый пользователь системы в автоматическом режиме после регистрации администратором в системе. В нем содержатся изначальные логин и пароль для доступа к личному кабинету. Далее пользователь может изменить их в разделе “Профиль”. Также в письме указана ссылка на сайт, по которому пользователь может перейти в личный кабинет. Данное письмо отправляется автоматически с помощью сервиса рассылок EmailJS.com.

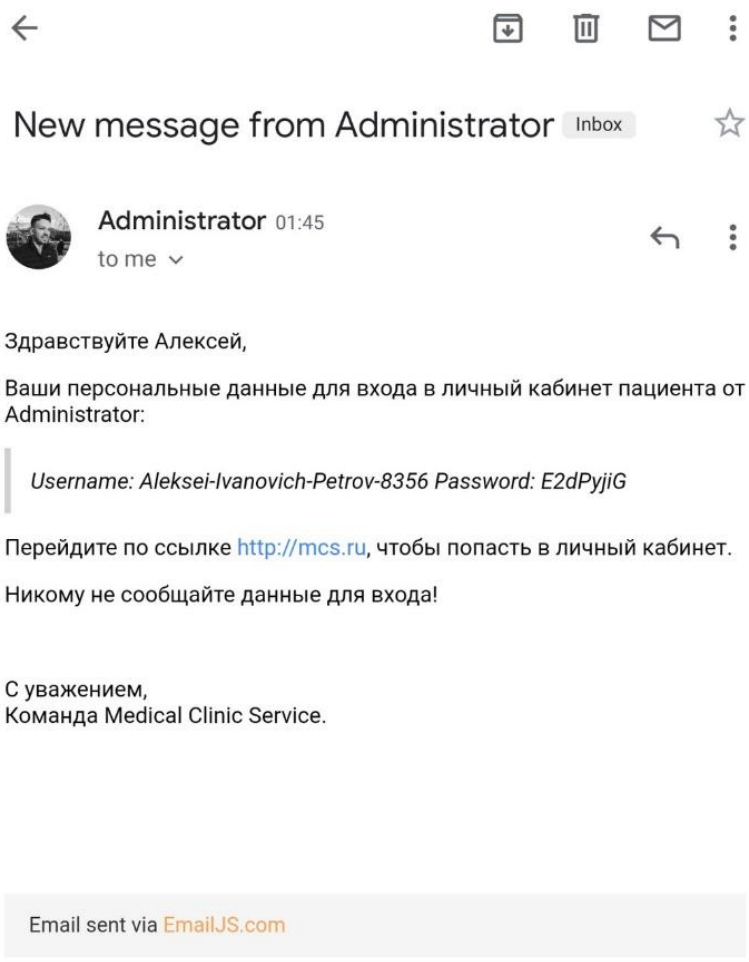

Рисунок 36. Письмо на электронную почту с личными данными пациенту, врачу или клиенту для входа в личный кабинет

2.6 Алгоритмы работы

В данном разделе будет продемонстрирован алгоритм работы интерфейса для разных ролей пользователей в системе – Админ, Доктор, Пациент. Иногда действия для этих пользователей могут пересекаться, некоторые пользователи имеют права, которых нет у других. На следующем рисунке показан алгоритм взаимодействия с информационной системой на страницах “Профиль” и “Записи”. На странице профиля пользователь системы может просмотреть и изменить свои личные данные. На странице записей пользователь может просмотреть информацию о записях и отфильтровать их в удобном формате.

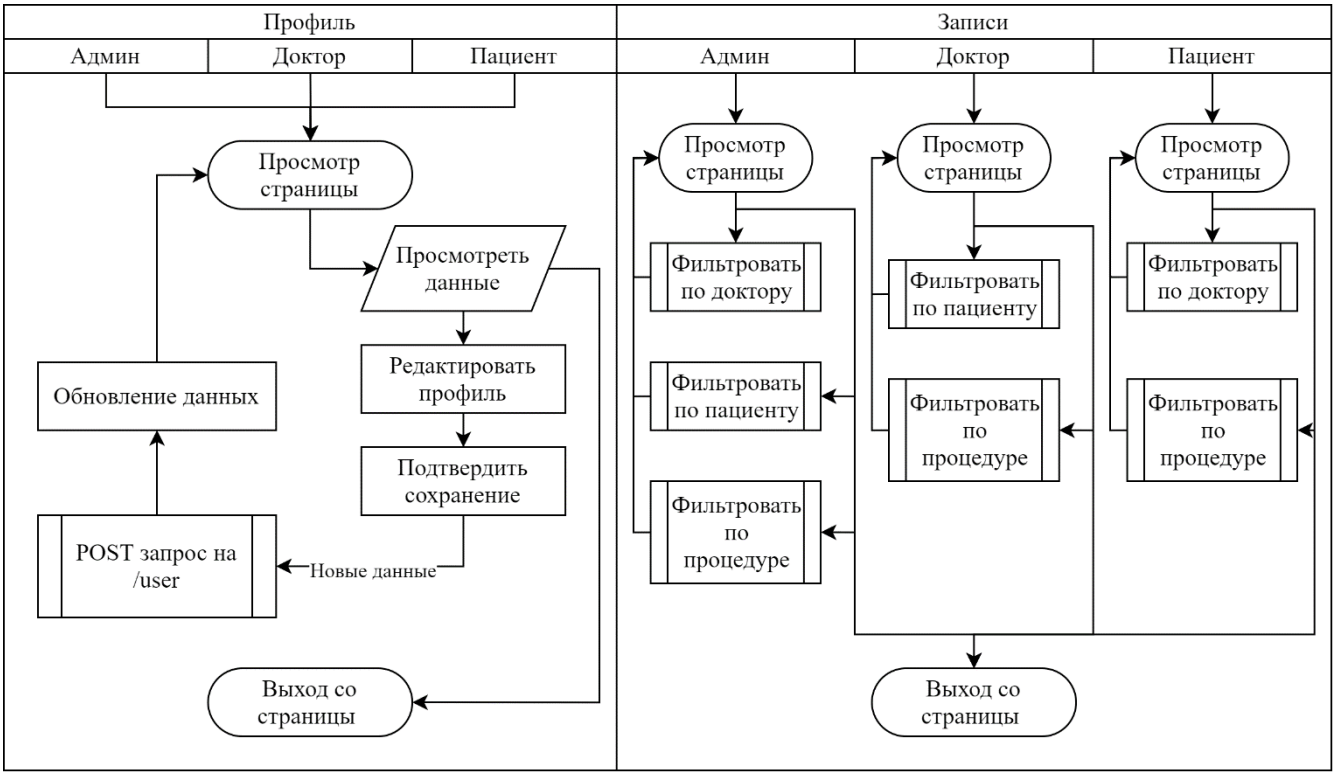

Рисунок 37. Алгоритм взаимодействия с информационной системой на страницах Профиль и Записи

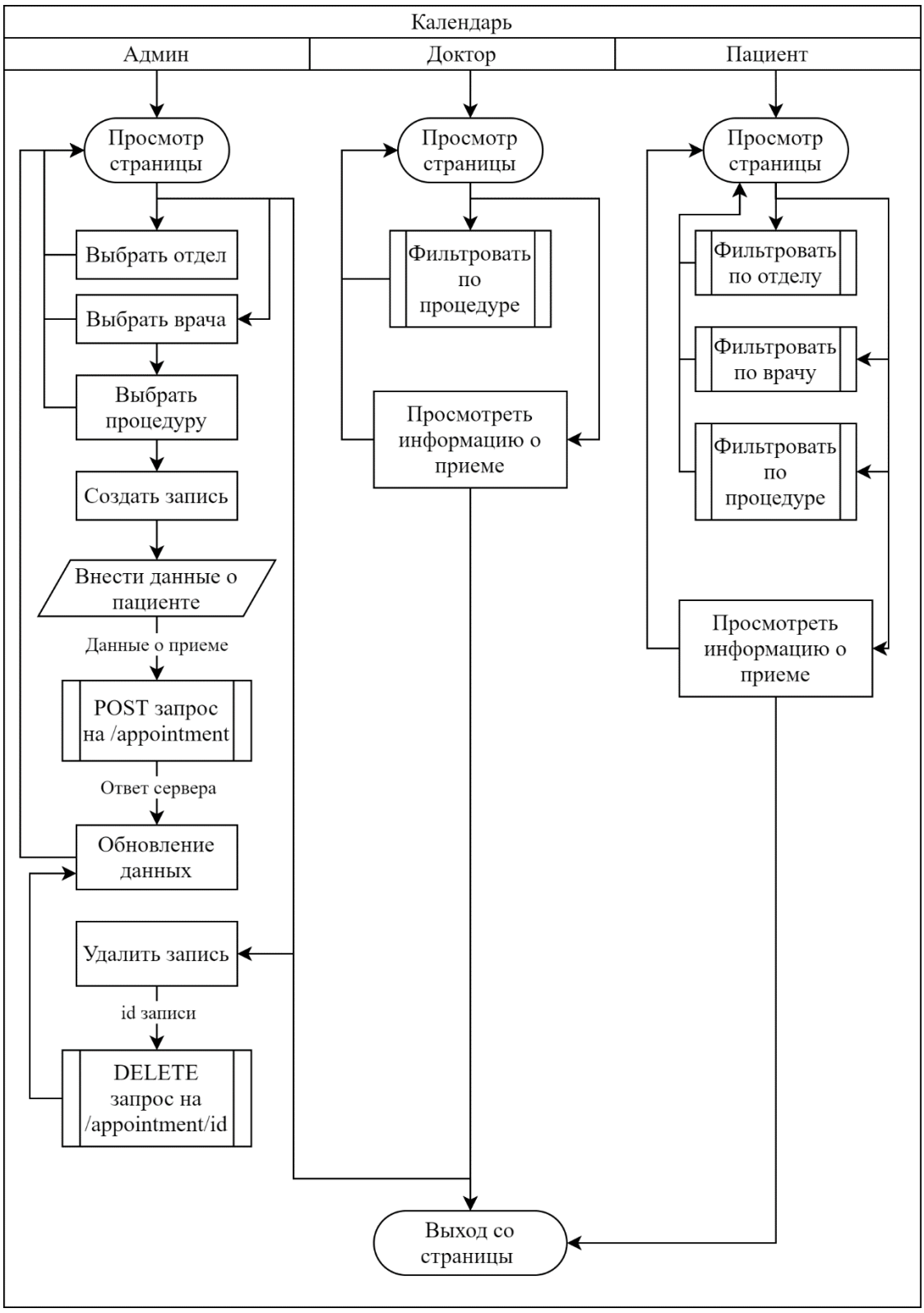

Рисунок 38. Алгоритм взаимодействия с информационной системой на странице Календарь

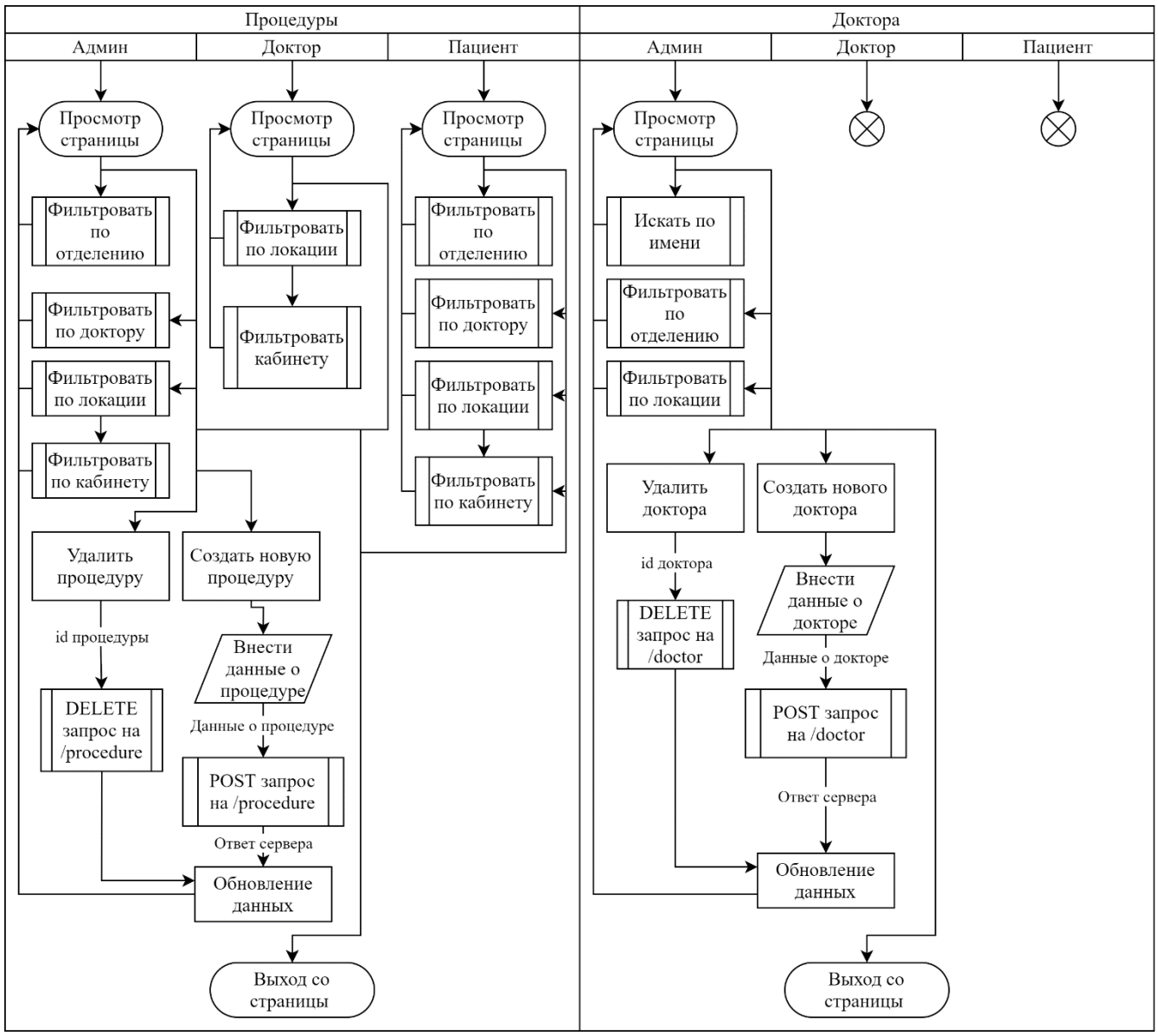

Рисунок 39. Алгоритм взаимодействия с системой на страницах Процедуры и Доктора

На рисунке выше показан алгоритм взаимодействия с информационной системой на страницах Процедуры и Доктора. Как видно из правой части, пользователи с ролью Доктор и Пациент не имеют доступа к странице Доктор, так как она содержит конфиденциальные данные о докторах и не может быть доступна для пациентов. К странице Доктора имеет доступ только пользователь с ролью Админ, где он может создавать и удалять докторов.

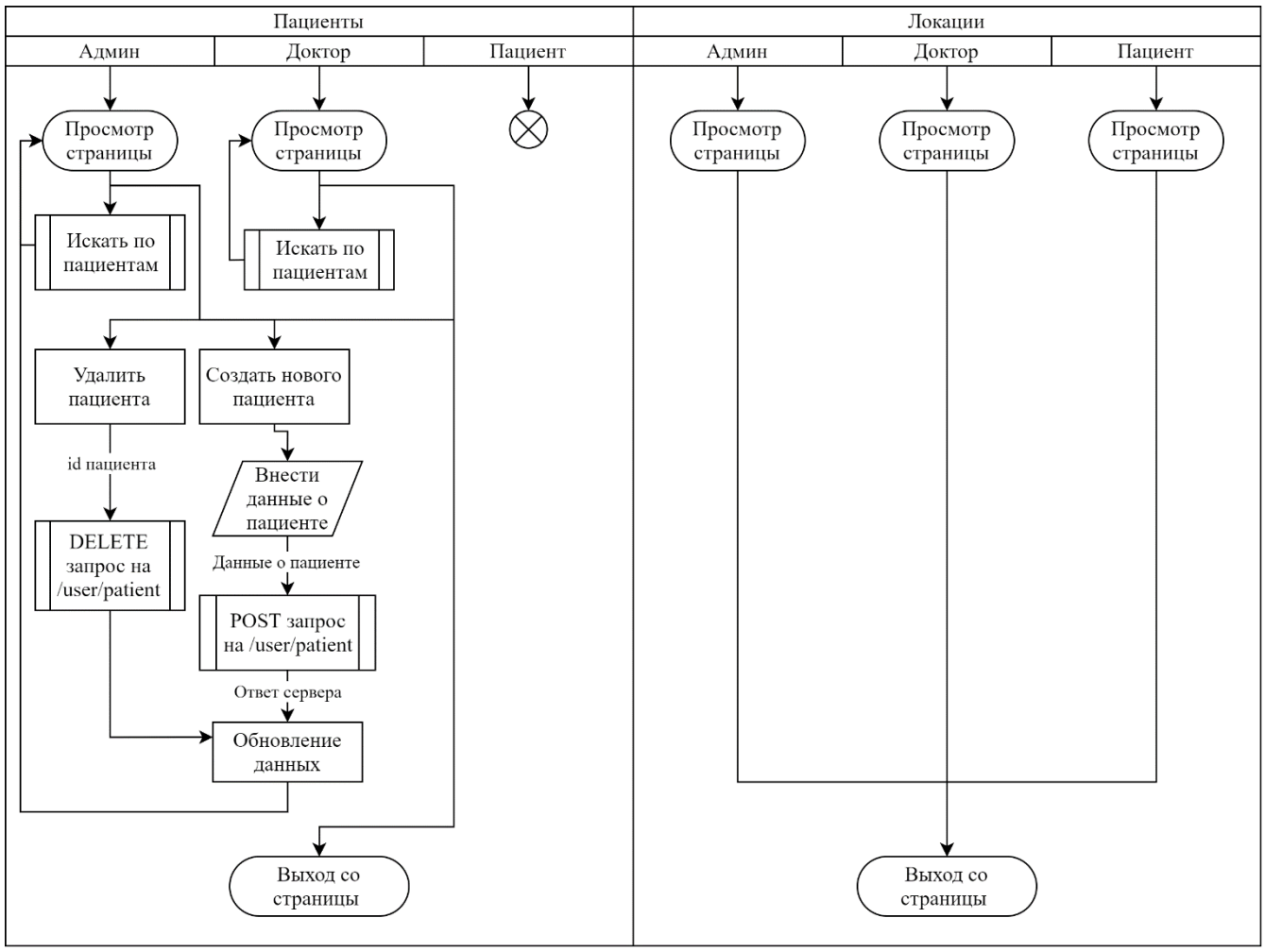

Рисунок 40. Алгоритм взаимодействия с информационной системой на страницах Пациенты и Локации

На рисунке выше показан алгоритм взаимодействия с информационной системой на страницах Пациенты и Локации. Как можно видеть, пользователи со всеми типами ролей могут просматривать страницу Локации. Страницу Пациенты могут просматривать только пользователи с ролями Админ и Доктор. Доктора могут видеть только своих пациентов и искать их по имени, чтобы получить более подробную информацию. Пользователи с ролью Админ могут искать, создавать и удалять пациентов. Перед тем, как создать запись на прием, Админ создает пациента и для него выделяется запись в базе данных.

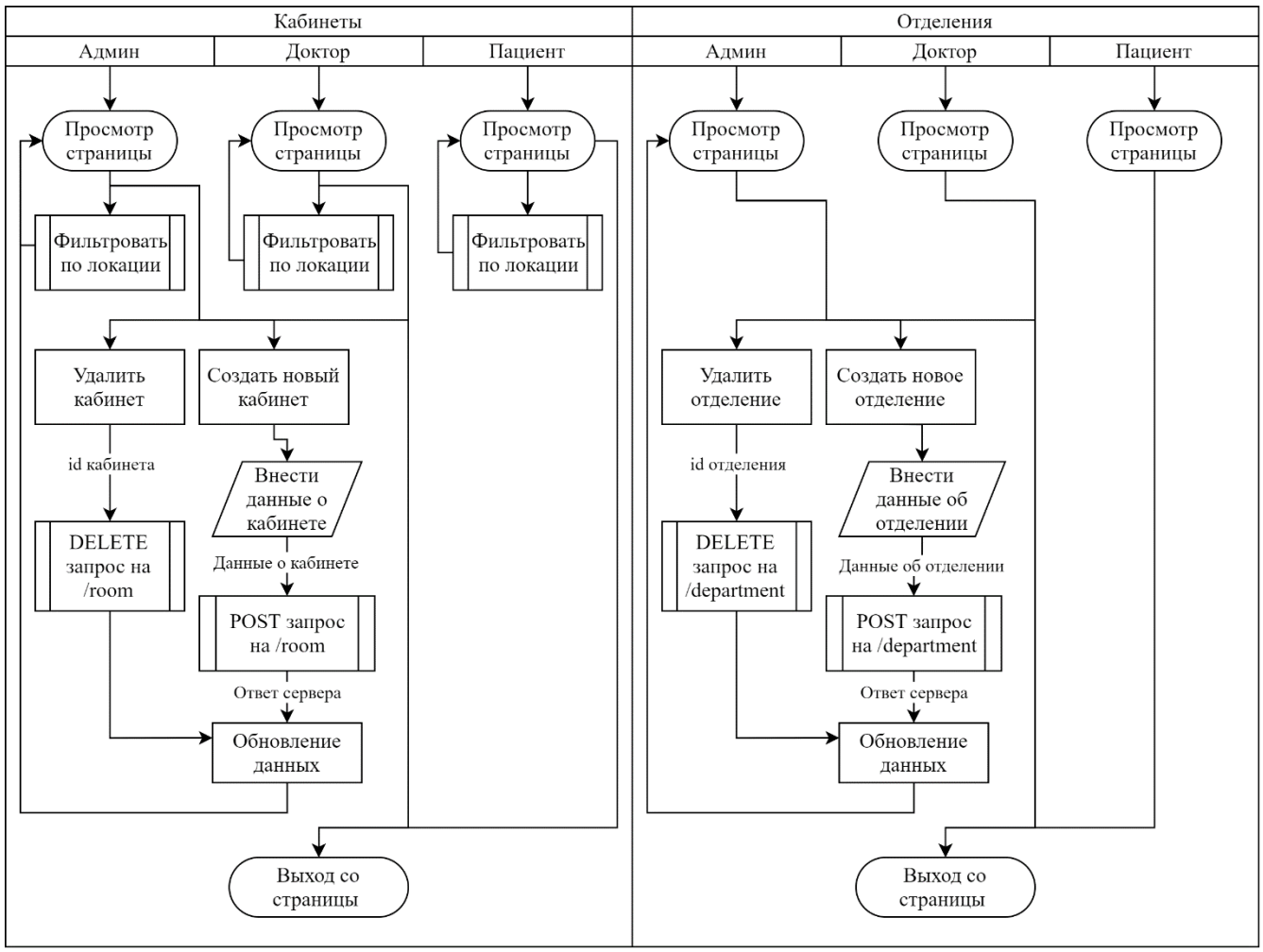

Рисунок 41. Алгоритм взаимодействия с информационной системой на страницах Кабинеты и Отделения

Выводы по главе 2

В данной главе была проделана работа по проектированию основных компонентов информационной системы – макета пользовательского интерфейса и архитектуры системы в целом. Предполагаемый функционал удовлетворяет всем современным требованиям разработки такого рода информационной системы. Предполагается реализация страницы Профиля, Записей, Календаря, Процедур, Пациентов, Докторов, Кабинетов, Отделений и Локаций. Каждая из данных страниц

имеет свои права доступа и функционал, доступный пользователям с разными ролями. Страница Профиль была создана и спроектирована для просмотра и изменения информации о профиле пользователя в системе. Страница Календарь создана для удобного планирования записей на прием к докторам в системе, а также просмотра и удаления записей на прием. Страница Записи создана для просмотра записей на прием пациентами, врачами и администраторами. Пациент может узнать актуальный список приемов и всю информацию о них. Страницы Доктора, Пациенты, Локации, Департаменты и Процедуры являются информационными и не доступны некоторым ролям в системе. На них в соответствии с алгоритмом взаимодействия с информационной системой пользователь может просматривать, создавать и удалять информацию о некоторых сущностях. Данная информационная система спроектирована по модульной архитектуре, где каждая сущность (Пациент, Локация) является отдельным модулем и может изменяться, добавляться новая или удаляться в зависимости от требований заказчика. Данная система является расширяемой, что позволяет в короткие сроки улучшить и доработать систему.

3. РАЗРАБОТКА КОМПОНЕНТОВ ИНФОРМАЦИОННОЙ СИСТЕМЫ

3.1 Описание среды разработки и программы

Среда разработки, в которой разрабатывалась информационная система называется Visual Studio Code [43]. Данная программа разработана в компании Microsoft и является легковесной версией Visual Studio. Она является расширяемой и почти не содержит встроенных модулей, а позволяет скачивать их дополнительно под нужды разработчика или конкретного проекта. Именно поэтому она является одной из самых популярных в наше время [41]. VS Code позволяет компилировать

и собирать решение, работать с Git и системой контейнеризации Docker, а также многое другое.

Проект информационной системы состоит из следующих файлов:

- .ts (файлы, которые содержат код на языке TypeScript, обычно содержат классы и интерфейсы в отличие от .tsx);
- .tsx (файлы React приложений, которые содержат разметку JSX на языке TypeScript – TypeScript XML) ;
- .js (файлы JavaScript, которые используются на серверной стороне);
- app.config.json (файл, в котором содержится вся конфигурация проекта, например порт для веб-приложения, протокол, страндартные заголовки HTTP и другая конфигурация для сборки проекта);
- babel.config.json (файл конфигурации утилиты babel, которая занимается трансляцией новых версий ECMA Script кода в более старые версии, например ES5 или ES4, для поддержки кода более старыми браузерами);
- package.json (файл утилиты управления пакетами NPM, в котором содержатся все зависимости проекта, скрипты запуска и тестирования проекта, текущая версия проекта, информация об авторе, лицензия и зависимости, нужные для разработки, но не использующиеся в production версии проекта);
- tsconfig.json (файл конфигурации TypeScript – он содержит опции компилятора TypeScript, версию ECMA Script, в которую компилируется код и другие параметры);
- webpack.config.js (файл конфигурации сборщика проекта Webpack, в нем содержатся правила для сборки проекта, название файлы, в который проводится сборка, плагины для анализа статических файлов и другие полезные возможности Webpack для разработки);

- .scss (файлы стилей SASS, который является утилитой препроцессинга для стилей и имеет более обширные возможности, чем обычная таблица каскадных стилей) [39].

3.2 Программная реализация информационных моделей

Для инициализации изначальной конфигурации клиентской части проекта используется файлы конфигурации app.config.json. Он позволяет задать хост и порт, на котором будет работать проекта. Содержимое файла указано в листинге 1. В случае production версии проекта в нем указывает номер хоста и номер порта, на котором будет запущена клиентская часть, а также origin – параметр, который отвечает за хост и порт, на которым работает сервер. Это нужно для SOAP политики браузеров, которая блокирует соединенения со сторонними источниками.

Листинг 1. Файл конфигурации клиентской части проекта

```
{
  “server” : {
    “protocol”: “http”,
    “host”: “localhost”,
    “port”: “3000”,
    “credentials”: true,
    “origin”: “http://localhost:8080”,
    “headers”: {
      “Content-Type”: “application/json;charset=utf-8”
    }
  }
}
```

Первой страницей, которая отображается незарегистрированному пользователю является страница входа в систему. На ней пользователь вводит свои учетные данные и отправляет запрос на сервер. Ему выдается ключ сессии, с которым он может обращаться для получения данных на других страницах. На листинге 2 продемонстрирован родительский код React компонента.

Листинг 2. Код приложения на React для входа в приложение

```
export const App: React.FC<{}> = () => {
  const [session, setSession] = useState(getSession());

  return (
    <GlobalContext.Provider
      value={{
        sessionUpdated: () => setSession(getSession()),
      }}
    >
      <BrowserRouter>
        <div className="full-height">
          <Switch>
            <Route exact path='/'>
              {session
                ? <Redirect to='/dashboard' />
                : <Authorization />}
            </Route>
            <Route component={Dashboard} path='/dashboard' />
          </Switch>
        </div>
      </BrowserRouter>
    </GlobalContext.Provider>
  );
};
```

Основной входной файл для веб-приложения на сервере продемонстрирован в листинге 3. В нем указаны роуты для приложения, которые обрабатывают запрос в зависимости от url в соответствии с REST подходом.

Листинг 3. Код обработчика запроса на сервере

```
const port = 3000;
const corsConfig = { credentials: true, origin: 'http://localhost:8080' };

/* Routers */
const auth = require('./Auth/auth');
const logout = require('./Auth/logout');
const user = require('./User/user');
const location = require('./Location/location');
const room = require('./Room/room');
const procedure = require('./Procedure/procedure');
const department = require('./Department/department');
const appointment = require('./Appointment/appointment');

app.use(cors(corsConfig));
app.use(cookieParser());
app.use('/auth', auth);
app.use('/logout', logout);
app.use('/user', user);
app.use('/location', location);
app.use('/room', room);
app.use('/procedure', procedure);
app.use('/department', department);
app.use('/appointment', appointment);

app.listen(port, () => {
  console.log(`Listening at http://localhost:${port}`)
});
```

Как мы видим на листинге 3, в файл загружаются модули для обработки запросов на каждую из сущностей, также указывается конфигурация CORS и утилита для парсинга cookie.

Листинг 4. Функция fetchApi для запроса к API со стороны клиента

```
enum HTTPMethod {
  GET = 'GET',
  POST = 'POST',
  PUT = 'PUT',
  DELETE = 'DELETE',
}
const fetchApi = (entity:string,method:HTTPMethod,postData?:object) => {
  const protocol = 'http';
  const host = 'localhost';
  const port = 3000;
  const url = `${protocol}://${host}:${port}/${entity}`;

  let requestOptions: RequestInit = {
    credentials: 'include',
    method: method,
    headers: {
      'Content-Type': 'application/json;charset=utf-8',
      'Access-Control-Allow-Origin': '*',
    }
  };

  if ((method==HTTPMethod.POST || method==HTTPMethod.PUT) && postData) {
    requestOptions.body = JSON.stringify(postData);
  }

  return fetch(url, requestOptions);
};
```

На листинге 4 показан код запроса к серверному API со стороны клиента, который может обрабатывать сразу несколько типов запросов и разработан как обертка над функцией fetch в JavaScript [33].

3.2.1 Авторизация

Для получения данных пользователя и отображения их на сервисе используется метод POST на адрес /auth. В качестве body передается JSON объект с полями username и password. За авторизацию пользователей отвечает асинхронный модуль authorizationManager.js, который обращается в хранилище и ищет данные пользователя, если находит, то возвращает объект с данными. На листинге 5 показан обработчик запроса. Он возвращается пользовательские данные, которые затем помещаются в хранилище Redux и хранятся на протяжении сессии клиента. Также пользователь может выйти из кабинета, при этом у него удаляется ключ сессии. На листинге 6 показан обработчик удаления пользовательской сессии.

Листинг 5. Обработчик POST запроса /auth

```
router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  authorizationManager(req)
    .then(authResult => {
      authResult.accessControl &&
      res.cookie(
          'session',
          `${authResult.accessControl}-abc`,
          { maxAge: 1000 * 60 * 15 * 4 }
      );
      res.json(authResult);
    });
});
```

Листинг 6. Удаление пользовательской сессии

```
router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  res.cookie('session', '');
  res.json({ payload: "Successfully logged out", status: 'ok' });
});
```

3.2.2 Профиль

При переходе на страницу профиля пользователь может редактировать собственные данные такие, как ФИО, email, номер телефона, дату рождения, пол, логин, пароль, адрес и другие. При входе на страницу запроса за получением данных не происходит, так как они приходят при авторизации и устанавливаются в хранилище Redux. Запрос на /user происходит при сохранении данных, который возвращает обновленные данные, которые затем обновляются в хранилище Redux. На листинге 7 показан метод сохранения обновленных данных пользователя.

Листинг 7. Обновление данных в профиле пользователя

```
const handleOnFinish = () => {
  const values = { // Новые данные пользователя
    …form.getFieldsValue(),
    id: props.userProfile.id,
    accessControl: props.userProfile.accessControl, // Роль в системе
    birthDate: form.getFieldValue('birthDate') == null
      ? ''
      : moment(form.getFieldValue('birthDate')).format()
  };
```

```
  fetchApi('user', HTTPMethod.PUT, values) // Асинхронный запрос
    .then((response) => response.json()) // Парсинг JSON ответа
    .then(json => props.dispatch(updateUser(json))); // Обновление
    notification.success({ // Уведомление об изменении данных
      message: 'Data has been updated successfully', duration: 3
    });
}
```

Метод на листинге 7 срабатывает после того, как пользователь нажмет кнопку “Подтвердить” под формой редактирования данных.

3.2.3 Записи

На вкладке записей, как говорилось выше, отображаются записи на прием пациентов к докторам. В зависимости от роли, отображаются разные данные. На листинге 8 показана функция useEffect, которая делает данный запрос. В терминах библиотеки React функция useEffect называет хук [13]. Хуки в React позволяют более гибко использовать состояние компонента и манипулировать Virtual DOM[14]. Хук useEffect помогает в использовании побочных эффектов в компоненте [18]. Побочные эффекты могут быть обращениями к API, обновлением состояния компонента и любым другим действием. Первым аргументом в useEffect является функция-коллбэк [19], которая вызывает побочные эффекты, вторым аргументом является массив зависимостей. Если какое-либо из значений в массиве зависимостей изменилось, React вызывает useEffect. Если передан пустой массив зависимостей, значит useEffect будет вызван только 1 раз при первоначальном рендере компонента и при обновлении состояния компонента он вызываться не будет.

Листинг 8. Запрос данных для отображения страницы Записи

```
useEffect(() => {
  setIsLoading(true);
  Promise.all([
    fetchApi('appointment/-1', HTTPMethod.GET),
    fetchApi('user/doctor/-1', HTTPMethod.GET),
    fetchApi('room/-1', HTTPMethod.GET),
    fetchApi('procedure/-1', HTTPMethod.GET),
    fetchApi('user/patient/-1', HTTPMethod.GET),
  ])
  .then(values => Promise.all(values.map(res => res.json())))
  .then((data: Array<any>) => {
    let temp: any = {};
    data.map((obj: any) => { temp = {…temp, …obj}; })
    setState(temp);
  })
  .then(() => setIsLoading(false));
}, []);
```

В 3 строчке на листиге 8 можно видеть вызов метода all класса Promise. Данный класс был разработан для работы с асинхронными операциями, внутри он представляет собой событийную модель [27]. Promise.all в данном случае означает, что метод будет “ждать” ответа на все запросы и только после этого перейдет к обработке коллбэка в методах then. Данный подход называется асинхронным выполнением кода [2].

3.2.4 Календарь

На странице календаря пользователи в зависимости от роли могут просматривать и создавать новые записи на прием. Создавать может только администратор. Доктор и Пациент могут лишь просматривать созданные приемы.

Листинг 9. Интерфейс записи в календаре, метод создания и удаления записи

```
export interface CalendarEvent { // Интерфейс записи на прием
  id: number;
  start: Date;
  end: Date;
  created: string;
  doctorId: number;
  patientId: number;
  procedureId: number;
  roomId: number;
  title: string;
  notes: string;
}

const handleModalSubmit = (values: any) => { // Создание записи
  const appointmentData = {…values, created: '' }
  // Запрос к API на создание записи
  fetchApi('appointment', HTTPMethod.POST, appointmentData);
  if (selectedDoctortId) {
    fetchAppointments(selectedDoctortId) // Получение списка записей
      .then(res => res.json())
      .then(data => setDoctorAppointments(data.appointments))
      .then(() => setOpenModal(false)); // Закрытие модального окна
  }
};

const handleDeleteAppointment = () => { // Обработчик удаления записи
  fetchApi(`appointment/${editAppointmentData.id}`, HTTPMethod.DELETE)
    .then(res => res.json())
    .then(() => {
      setAppointmentDescrModalOpen(false);
      setEditAppointmentData({} as CalendarEvent);
      setLoadAppointments(true); // Загрузить новый список записей
    })
};
```

На листинге 9 показан интерфейс записи в календаре, метод создания записи и метод удаления записи. Методы работы с записью обращаются к API и работают с обработчиком Appointment на сервере (листинг 10).

Листинг 10. Роуты для обработки запросов на /appointment

```
router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders); // Установка заголовков
  const newAppointment = req.body; // Получение POST body
  const dbres = await storageInterface.create(
    appointmentQueryGenerator.generateCreateAppointment(newAppointment)
  );
  res.json({ status: 'OK', message: 'Appointment are being created' });
});

router.get('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const appointmentId = Number(req.params['id']);
  const sqlQuery = appointmentQueryGenerator.generateGetAppointment(
    appointmentId
  );
  const result = await storageInterface.select(sqlQuery);
  res.json({ appointments: result.rows });
});

router.delete('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const appointmentId = Number(req.params['id']);
  const sqlQuery = appointmentQueryGenerator.generateDeleteAppointment(
    appointmentId
  );
  const result = await storageInterface.delete(sqlQuery);
  res.json({ appointments: result.rows });
});
```

На листинге 10 можно видеть обработчики запросов на разные методы HTTP, а именно обработчики POST, GET и DELETE запросов на /appointment. Методы для GET и DELETE принимают id как query parameter из url [37]. Query parameters позволяют отправлять данные в качестве аргументов запроса на сервер, которые затем сервер парсит и использует в бизнес-логики. В данном случае по id сервер обращается в базу данных и получает либо удаляет нужную запись. В качестве получения текстов запросов в базу данных используется класс AppointmentQueryGenerator, который предоставляет методы для генерации разных запросов к базе данных.

3.2.5 Процедуры, пациенты, кабинеты и отделения

Интерфейс взаимодействия со страницами процедур, пациентов, кабинетов и отделений схож, поэтому данные страницы были вынесены в общую главу. Данные сущности предполагают создание и удаление. Полное редактирование перечисленных сущностей будет перенесено на следующую итерацию разработки. Для каждой из сущностей существует отдельная страница, на которой можно просматривать и удалять экземпляры сущности. Для удаления посылается DELETE запрос на /entity/id, где entitry = [procedure, user/patient, room, department] в зависимости от сущности. При создании соответственно посылается POST запрос на /entity. В будущей разработке при редактировании будет посылаться PUT или PATCH запрос на /entity/id и возвращаться измененный экземпляр сущности. Под экземпляром сущности понимается запись в базе данных. Например, если сущность Пациент (/user/patient), то экземпляр сущности это конкретный пациент Иванов Иван Иванович и все его персональные данные, включая созданные приемы. На сервере для каждой из сущностей существует обработчик GET, POST и DELETE запросов, которые генерируют SQL скрипты, выполняют их через класс StorageInterface (листинг 11) и возвращают ответ с данными клиенту.

3.2.6 Модуль взаимодействия с базой данных

За взаимодействие с базой данных отвечает класс StorageInterface, который реализует шаблон CRUD интерфейса (create, read, update, delete) для каждого из запросов POST, GET, PUT (PATCH), DELETE соответственно. Модуль читает конфигурационный файл в конструкторе и инициализирует соединение с базой данных. Далее экземпляр данного класса экспортируется и используется.

Листинг 11. Модуль взаимодействия с базой данных storageInteface.js

```
class StorageInterface {
  constructor() {
    const configurationManager = new ConfigurationManager();
    this.configuration = configurationManager.importConfiguration();
    this.client = new Client(configuration);
    this.client.connect();
  }
  create = async query => await this.client.query(query);
  select = async query => await this.client.query(query);
  update = async (query) => await this.client.query(query);
  delete = async (query) => await this.client.query(query);
}
```

Листинг 12. Конфигурационный файл для подключения к базе данных PostgreSQL

```
{
    “user”: “postgres”,
    “host”: “localhost”,
    “database”: “testdb”,
    «password»: «admin»,
    «port»: 5432
}
```

3.2.7 Модуль отправки Email оповещений

В сервисе присутствует модуль отправки Email оповещений на почту новому пользователю для того, чтобы он мог получить данные для входа в личный кабинет (листинг 13, 14).

Листинг 13. Модуль отправки Email оповещений

```
const fetch = require("node-fetch");
const readConfigFile = require("read-config-file");
const path = require('path');

module.exports = class EmailService {
  constructor() {
    this.config = {};

    readConfigFile.loadConfig({
      packageKey: '1',
      configFilename: 'application.local',
      projectDir: path.resolve(__dirname)
    }).then((jsonCfg) => { this.config = jsonCfg.result; });
  }

  sendAccessData(name, username, password) {
    const templateParams = {
      'to_name': name,
      'message': `Username: ${username} Password: ${password}`,
      'link_name': 'http://mcs.ru',
      'from_name': 'Administrator',
    };
```

```
    return fetch(this.config.apiPath,
      {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Access-Control-Allow-Origin': '*',
          'Accept': '*/*',
        },
        body: JSON.stringify({
          'template_id': this.config.template_id,
          'service_id': this.config.service_id,
          'user_id': this.config.user_id,
          'template_params': templateParams,
          'accessToken': this.config.accessToken,
        })
      }
    );
  }
}
```

Листинг 14. Пример конфигурации для модуля отправки Email оповещений

```
{
  "template_id": "template_hvqoj7f",
  "service_id": "service_4hlqv5w",
  "user_id": "user_piwy7uOic9Asc3vPLBwmW",
  "accessToken": "bd55ce1dc9dab06e58168d4fdede959c",
  "apiPath": "https://api.emailjs.com/api/v1.0/email/send"
}
```

Как видно на листинге 13, в конструкторе модуль читает конфигурационный файл, пример которого указан на листинге 14. В конфигурационный файл вносятся токены с сервиса Emailjs, который помогает делать рассылки на email [40]. Библиотека read-config-file помогает в удобном формате читать и парсить

конфигурационные файлы для проектов [38]. В конструкторе класса EmailService читает конфигурационный файл, далее в методе sendAccessData происходит отправка данных для входа в личный кабинет. Логин берется из ФИО пользователя, к которому добавляется случайный набор значений для добавления уникальности и избежания ситуации, когда клиенты с одинаковыми ФИО получат одинаковые логины в систему. Пароль генерируется также из случайных значений. Далее пользователь на почтовый ящик получает данные для входа и может начинать пользоваться сервисом.

Выводы по главе 3

В данной главе была приведена программная реализация разработанных алгоритмов и моделей ИС. Были рассмотрены схемы и способы взаимодействия клиента с сервером на страницах Профиль, Записи, Календарь, Доктора, Пациенты, Локации, Кабинеты и Департаменты. Были приведены фрагменты исходного кода основных модулей программы из слоев клиентской части, бизнес логики и модуля доступа к базе данных. Также в данной главе было приведено описание API серверной части и основных запросов, которые обрабатывает веб-приложение на сервере, типы файлов для серверной и клиентской части и конфигурационные файлы сторонних модулей проекта. Разработанная система обрабатывает ошибки при некорректном вводе данных, что делает ее работу стабильной. В следующей итерации разработки планируется добавить более комплексную модель авторизации, добавить редактирование сущностей и расширять функционал под каждого заказчика в соответствии с требованиями.

4. РАСХОДЫ НА РАЗРАБОТКУ ИНФОРМАЦИОННОЙ СИСТЕМЫ

4.1 Оценка стоимости разработки, внедрения и поддержки

На разработку информационной системы было затрачено 60 дней. Расчет стоимости работы происходит по затратам, которые были понесены в процессе создания и разработки. Затраты считаются на основе сложения статей калькуляции себестоимости: электроэнергия и интернет, амортизация на технику, оплата труда программиста, тестировщика и дизайнера. Расчет стоимости продукта с учетом стандартного восьмичасового рабочего дня работы трех технических специалистов с арендой серверов для развертывания ПО приведен в таблице 2. Данный расчет не включает в себя поддержку продукта. Стоимость поддержки приведена в таблице 3.

Таблица 2. Стоимости разработки информационной системы

Наименование	Единица измерения	Количество	Цена, руб.	Сумма, руб.
Зарплата дизайнера	Руб.	2	70000	140000
Зарплата программист (frontend)	Руб.	2	80000	160000
Зарплата программиста (backend + deploy)	Руб.	2	90000	180000
Аренда хостинга бэкэнд	шт.	2	1890	3780
Аренда хостинга фронтенд	шт.	2	1890	3780
Аренда хостига база данных	шт.	2	1890	3780
Сервис отправки email	шт.	2	2884*	5768
Итого				**497108**

* приведена примерная цена в рублях на момент составления стоимости по курсу центробанка. Стоимость в долларах – 40$.

Далее на таблице 3 приведена примерная стоимость поддержки продукта в зависимости от требований заказчика и дополнительного функционала платформы. В стоимость поддержки входит стоимость оплаты непосредственно хостинга и дополнительных сервисов, которые понадобятся заказчику и стоимость оплаты труда специалистов по поддержке продукта.

Таблица 3. Стоимость поддержки информационной системы в месяц

Наименование	Единица измерения	Количество, в месяц	Цена, руб.	Сумма, руб.
Поддержка хостинга бэкэнд	шт.	1	10000	10000
Поддержка хостинга фронтенд	шт.	1	10000	10000
Поддержка хостига база данных	шт.	1	10000	10000
Сервис отправки email	шт.	1	8000	8000
Итого				**38000**

Зарплаты технических специалистов указаны в соответствии со средним уровнем заработных плат и являются примерными. Конечная стоимость разработки может отличатся.

При отсутствии компьютерной техники у заказчика также в таблице 4 приведены расчеты по стоимости компьютерного оборудования, а в таблице 5 указана амортизация. Данные являются усредненными и зависят от требований заказчика по качеству компьютерной техники.

Таблица 4. Стоимость покупки компьютерной техники на 1 человека

Наименование	Единица измерения	Количество	Цена, руб.	Сумма, руб.
Системный блок	шт.	1	40000	40000
Монитор	шт.	1	15000	15000
Компьютерная мышь	шт.	1	400	400
Клавиатура	шт.	1	500	500
Итого				**55900**

В таблице 5 приведены месячные расходы на оборудование и амортизацию с учетом оплаты интернета и электроэнергии. В среднем компьютер с монитором потребляют в час 220 Ватт энергии. На 9 часовой рабочий день получается около 2 кВт, расход в режиме ожидания составит 4 Ватт в час, на 15 часов получаем 1,06 кВт. При 22 рабочих днях в месяце получаем 22*(2+1,06) ≈ 68 кВт потребленной энергии. Данные также являются примерными, конечная сумма может отличатся.

Таблица 5. Стоимость обслуживания компьютерной техники на 1 человека в месяц

Наименование	Единица измерения	Количество	Цена, руб.	Сумма, руб.
Электроэнергия	кВт/ч	68	5,5	374
Системный блок амортизация	Руб.	1	1250	1250
Монитор амортизация	Руб.	1	470	470
Компьютерная мышь амортизация	Руб.	1	35	35
Клавиатура амортизация	Руб.	1	44	44
Итого				**2173**

Формула для расчета амортизации с учетом средней годовой инфляции на территории РФ в 4%.

$$A = \frac{C}{T * \left(1 + \frac{I}{100}\right)^{T} * 12},$$

где A – ежемесячная сумма износа;

C – цена приобритения товара;

T – период службы;

I – годовая инфлияция в %;

12 – количество месяцев в году.

Выводы по главе 4

Внедрение информационной системы сегодня является необходимым требованием для ведения электронной регистратуры. Цифровые системы позволят докторам и администраторам меньше времени тратить на обработку информации, а пациентам в понятном режиме отслеживать свои приемы и узнавать актуальную информацию о записях. Как было сказано в исследовательской части, существуют неявные издержки, такие как стресс докторов, потерю от которых сложно оценить. Положительным эффектом от внедрения информационной системы станет ускорение записи пациентов, удобство пациентов в отслеживании приемов, постепенное снижение нагрузки на врачей и их уровня стресса на рабочем месте, а также возможность создания отчетов, которые могут помочь в бизнес-планировании. Все эти факторы помогут снизить явные и неявные издержки и увеличить прибыль предприятия.

ЗАКЛЮЧЕНИЕ

В настоящей выпускной квалификационной работе разработана информационная система взаимодействия медицинских клиник с клиентами, позволяющая снизить расходы на обработку информации о пациентах, приемах и докторах путем автоматизации бизнес-процессов предприятия.

В первой главе выпускной квалификационной работы был проведен анализ предметной области и инструментов для реализации информационной системы. В рамках анализа были выявлены основные проблемы предметной области, с которыми сталкиваются врачи медицинских клиник сегодня. Из них являются основными: высокая нагрузка и нехватка цифровых процессов на предприятии, дороговизна существующих комплексных решений, которые зачастую являются избыточными и новизна рынка цифровой медицины в России и мире. Рассмотренные существующие решения оказываются достаточно дорогими для внедрения и поддержки. Данная разработка, напротив, предоставляет минимальный функционал для автоматизации предприятия и при условии требований заказчика может быть доработана в короткие сроки. Также в данной главе был выполнен анализ и выбор инструментальных средств для реализации проекта. В качестве клиентской части приложения была выбрана библиотека React для JavaScript и библиотека компонентов Ant Design. Для серверной части была выбрана платформа Node.js и фреймворк Express.js. В качестве базы данных выбрана PostgreSQL.

Во второй главе были спроектированы и реализованы информационные модели, выбраны методы и средства проектирования. Также было проведено логическое и физическое проектирование системы. Физическое проектирование является технической реализацией логически спроектированной системы. В качестве физического проектирования была приведена модель реализации базы данных. Также в данной главе были приведены макеты пользовательского

интерфейса с разделением доступа к страницам по ролям и представлен конечный вид интерфейса информационной системы. Описан алгоритм работы на каждой из страниц.

В третьей главе были описаны программные реализации компонентов, сервисов и отдельных модулей информационной системы таких, как модуль доступа к базе данных, сервис отправки email оповещений и модули запросов к API.

В четверой главе были представлены расчеты по стоимости проекта по реализации информационной системы. Была рассчитана стоимость разработки, внедрения и поддержки программного продукта с учетом амортизации и среднегодовой инфляции на территории РФ.

Целью выпускной квалификационной работы являлась разработка информационной системы взаимодействия медицинских клиник с клиентами. Итогом достижения поставленной цели является реализация полноценной программной платформы, которая автоматизирует деятельность медицинской клиники.

В ходе работы были реализованы следующие задачи:

1. проведен анализ предметной области и инструментальных средств для реализации проекта;
2. разработана информационная модель пользовательского интерфейса системы взаимодействия медицинских клиник с клиентами;
3. осуществлена программная реализация информационной модели и алгоритмов учета пациентов и записей к врачам;
4. приведены затраты на реализацию, внедрение и поддержку информационной системы.

Таким образом, все задачи выпускной квалификационной работы выполнены, цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ Р 51904-2002 -Программное обеспечение встроенных систем
2. Васильев П. А. Обновление языка программирования JavaScript (es5) - ECMAScript 2015 (ES6) // Современные инновации. 2016. №12 (14).
3. Васильева К. Н., Хусаинова Г. Я. Реляционные базы данных // Colloquium-journal. 2020. №2 (54).
4. Газизуллин Н. И., Плещинская И. Е. Разработка клиентской части веб-приложения с использованием технологий spa // StudNet. 2020. №8.
5. Гонтарев А. А. Обзор и сравнительная характеристика технологий для разработки web приложения // Прикладная математика: современные проблемы математики, информатики и моделирования. 2020. №II.
6. Конюхов В.Г. База данных. Понятие, значение и роль в современном мире // Системные технологии. 2017. №24
7. Литвинов В. В., Задорожний А. А., Богдан И. В. Язык блочного имитационного моделирования на базе модифицированных диаграмм деятельности UML // ММС. 2017. №4.
8. Некрасова Т. А., Наролина Т. С., Смотрова Т. И., Пургаева И. А. 2020. «ПЕРСПЕКТИВЫ РАЗВИТИЯ РОССИЙСКОГО РЫНКА ЦИФРОВОЙ МЕДИЦИНЫ». Современная экономика: проблемы и решения 5 (июнь), 149-58.
9. Савоськин И.В., Фирсов А.О. Исследование способов применения NoSQL и реляционных баз данных // E-Scio. 2019. №6 (33).
10. Чеглаков А.Л. Композиция web-сервисов на основе архитектуры rest // Инновационная наука. 2016. №12-2.
11. Автоматизация управления салоном красоты, косм. центром, клиникой и СПА [Электронный ресурс]. URL: https://cleverbox-crm.com (дата обращения: 20.02.2021)
12. Архитектура REST [Электронный ресурс]. URL: https://habr.com/ru/post/38730/ (дата обращения: 11.02.2021)
13. Введение в хуки [Электронный ресурс]. URL: https://ru.reactjs.org/docs/hooks-intro.html (дата обращения: 20.05.2021)
14. Виртуальный DOM и детали его реализации в React [Электронный ресурс]. URL: https://ru.reactjs.org/docs/faq-internals.html (дата обращения: 20.05.2021)
15. «Индекс здоровья будущего 2018» Часть III: 82% россиян готовы консультироваться с врачом дистанционно. [Электронный ресурс]. 2018. Дата обновления: 25.02.2021.URL: https://www.philips.ru/a-w/about-philips/future-

health-index/reports/2018/moving-data-to-the-heart-of-health-systems.html (дата обращения: 25.02.2021).
16. «Индекс здоровья будущего 2018» Часть I: Роль цифровых технологий для создания интегрированной системы оказания медицинской помощи. [Электронный ресурс]. 2018. Дата обновления: 25.02.2021. URL: https://www.philips.ru/a-w/about-philips/future-health-index/reports/2018/building-systems-for-better-outcomes.html (дата обращения: 25.02.2021).
17. «Индекс здоровья будущего 2020». Новое поколение медицинских специалистов на пути к трансформации здравоохранения. [Электронный ресурс]. 2020. Дата обновления: 25.02.2021.URL: https://www.philips.ru/a-w/about-philips/future-health-index/reports/2020/the-age-of-opportunity.html (дата обращения: 25.02.2021).
18. Использование хука эффекта [Электронный ресурс]. URL: https://ru.reactjs.org/docs/hooks-effect.html (дата обращения: 20.05.2021)
19. Колбэк-функция [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Glossary/Callback_function (дата обращения: 20.05.2021)
20. Медицинская система Archimed+ [Электронный ресурс]: URL: https://archimed.pro (дата обращения: 24.02.2021)
21. Медицинская информационная система Medesk [Электронный русурс]. URL: www.medesk.net (дата обращения: 24.02.2021)
22. Медицинская информационная система [Электронный ресурс]. URL: https://med-idea.ru (дата обращения: 20.02.2021)
23. Медицинская информационная система Renovatio [Электронный ресурс]. URL: https://rnova.ru (дата обращения: 20.02.2021)
24. 9Облачная crm-система для медицинских центров [Электронный ресурс]. URL: https://medicalcrm.ru (дата обращения: 25.02.2021)
25. Официальный сайт Агенства Инноваций города Москвы. Стартап-кафе Digital Tech. URL: https://innoagency.ru (дата обращения: 25.02.2021)
26. Официальный сайт Клиника Онлайн [Электронный ресурс]. URL: https://klinikon.ru (дата обращения: 13.02.2021)
27. Промисы [Электронный ресурс]. URL: https://learn.javascript.ru/promise-basics (дата обращения: 14.06.2021)
28. Что мы знаем об Ant [Электронный ресурс]. URL: https://habr.com/ru/company/simbirsoft/blog/416925 (дата обращения: 12.02.2021)

29. Что такое блок-схема и как ее создать? [Электронный ресурс]. URL: https://www.lucidchart.com/pages/ru (дата обращения: 18.01.2021)
30. Что такое ER-диаграмма? [Электронный ресурс]. URL: https://www.lucidchart.com/pages/ru/erd-диаграмма (дата обращения: 18.01.2021)
31. DataGrip [Электронный ресурс]. URL: https://www.jetbrains.com/ru-ru/datagrip/features/ (дата обращения: 20.02.2021)
32. Fast, unopinionated, minimalist web framework for Node.js [Электронный ресурс]. URL: https://expressjs.com (дата обращения: 12.02.2021)
33. Fetch API - Web APIs [Электронный ресурс]. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (дата обращения: 20.05.2021)
34. JavaScript-библиотека для создания пользовательских интерфейсов [Электронный ресурс]. URL: https://ru.reactjs.org (дата обращения: 12.02.2021)
35. JavaScript-окружение построенное на движке Chrome V8 [Электронный ресурс]. URL: https://nodejs.org/ru (дата обращения: 12.02.2021)
36. Node.js w/1M concurrent connections [Электронный ресурс]. URL: https://blog.caustik.com/2012/08/19/node-js-w1m-concurrent-connections (дата обращения: 12.01.2021)
37. Query Parameters [Электронный ресурс]. URL: https://branch.io/glossary/query-parameters/ (дата обращения: 14.06.2021)
38. read-config-file - read configuration file in various formats [Электронный ресурс]. URL: https://www.npmjs.com/package/read-config-file (дата обращения: 14.06.2021)
39. Sass: Syntactically Awesome Style Sheets [Электронный ресурс]: URL: https://sass-lang.com (дата обращения: 20.05.2021)
40. Send Email Directly From JavaScript [Электронный ресурс]. URL: https://www.emailjs.com (дата обращения: 14.06.2021)
41. Top IDE Index [Электронный ресурс]. URL: https://pypl.github.io/IDE.html (дата обращения: 19.05.2021)
42. Typed JavaScript at Any Scale Fast, unopinionated, minimalist web framework for Node.js [Электронный ресурс]. URL: https://www.typescriptlang.org (дата обращения: 12.02.2021)
43. Visual Studio Code [Электронный ресурс]. URL: https://code.visualstudio.com (дата обращения: 18.01.2021)
44. Webpack: руководство для начинающих [Электронный ресурс]. URL: https://habr.com/ru/post/514838 (дата обращения: 12.02.2021)

ПРИЛОЖЕНИЕ А. Фрагменты исходного кода программы

/Server/Appointment/appointment.js

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');

const GenerateQueriesAppointment =
require('../Storage/PostgreSQLQuery/GenerateQueriesAppointment');
const storageInterface = require('../connection');

const jsonParser = bodyParser.json();
const appointmentQueryGenerator = new GenerateQueriesAppointment();

// Create
router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const newAppointment = req.body;

  const dbres = await
storageInterface.create(appointmentQueryGenerator.generateCreateAppointme
nt(newAppointment));
  res.json({ status: 'OK', message: 'Appointment are being created' });
});

// Read
router.get('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const appointmentId = Number(req.params['id']);
  const sqlQuery =
appointmentQueryGenerator.generateGetAppointment(appointmentId);
  console.log(sqlQuery);
  const result = await storageInterface.select(sqlQuery);
  res.json({ appointments: result.rows });
});

router.get('/doctor/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const doctorId = Number(req.params['id']);
  const sqlQuery =
appointmentQueryGenerator.generateGetDoctorAppointment(doctorId);
  console.log(sqlQuery);
  const result = await storageInterface.select(sqlQuery);
  res.json({ appointments: result.rows });
});

// Delete
router.delete('/:id', jsonParser, async (req, res) => {
```

```
  res.set(responseHeaders);

  const appointmentId = Number(req.params['id']);
  const sqlQuery =
appointmentQueryGenerator.generateDeleteAppointment(appointmentId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ appointments: result.rows });
});

module.exports = router;
```

/Server/Auth/auth.js

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');
const authorizationManager = require('./authorizationManager');

const jsonParser = bodyParser.json();
// var urlencodedParser = bodyParser.urlencoded({ extended: false });

router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  authorizationManager(req)
    .then(authResult => {
      console.log('authResult', authResult);
      authResult.accessControl && res.cookie('session',
`${authResult.accessControl}-abc`, { maxAge: 1000 * 60 * 15 * 4 });
      res.json(authResult);
    });
});

module.exports = router;
```

/Server/Auth/authorizationManager.js

```
const StorageInterface = require("../Storage/storageInterface");
const storageInterface = require('../connection');

module.exports = authorizationManager = async (request) => {
  const username = request.body.username;
  const password = request.body.password;

  const result = await storageInterface.authorize(username);
  console.log(result);
  return Object(result);
};
```

/Server/Auth/logout.js

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');

const jsonParser = bodyParser.json();
// var urlencodedParser = bodyParser.urlencoded({ extended: false });

router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  console.log('logging out user', res.cookie);
  res.cookie('session', '');
  res.json({ payload: "Successfully logged out", status: 'ok' });
});

module.exports = router;
```

/Server/Database/Scripts/CreateTables.sql

```
create type "AppointmentStatus" as enum ('Created', 'In Process', 'Done',
'Canceled');
create type "AppointmentPaymentStatus" as enum('Paid', 'NotPaid');
create type "Gender" as enum ('Male', 'Female', 'Other');
create type "AccessControl" as enum ('Admin', 'Doctor', 'Patient');

create table if not exists "User" (
    "firstName" varchar(50) not null ,
    "lastName" varchar(50) not null ,
    email varchar(50) not null ,
    phone varchar(50) not null ,
    gender "Gender" not null ,
    "birthDate" varchar not null ,
    address varchar(300) not null ,
    username varchar(50) not null ,
    hashsum varchar(50) not null ,
    "accessControl" "AccessControl" not null
);

create table if not exists "Location" (
    id serial unique,
    "locationName" varchar(50) not null ,
    address varchar(300) not null ,
    phone varchar(50) not null ,
    email varchar not null ,

    primary key (id)
);

create table if not exists "Doctor" (
    id serial unique ,
```

```
    "locationId" int not null ,
    "departmentId" int not null ,
    "workExperience" int not null ,
    "academicDegree" varchar not null ,
    notes text not null ,

    primary key (id),

    constraint fk_locationId
        foreign key ("locationId")
        references "Location"(id),

    constraint fk_departmentId
        foreign key ("departmentId")
        references "Department"(id)
) inherits ("User");

create table if not exists "Patient" (
    id serial unique,
    "emergencyContactName" varchar(50),
    "emergencyContactPhone" varchar(50),
    "emergencyContactRelation" varchar(50),

    primary key (id)
) inherits ("User");

create table if not exists "Appointment" (
    id serial unique,
    "scheduledTime" varchar not null,
    "scheduledEndTime" varchar not null,
    created varchar not null,
    notes text not null,

    "patientId" int not null,
    "doctorId" int not null,
    "appointmentProcedureId" int not null,
    "roomId" int not null,

    primary key (id),

    constraint "fk_patientId"
        foreign key ("patientId")
        references "Patient"(id),

    constraint "fk_doctorId"
        foreign key ("doctorId")
        references "Doctor"(id),

    constraint "fk_roomId"
        foreign key ("roomId")
        references "Room"(id),

    constraint "fk_appointmentProcedureId"
```

```
        foreign key ("appointmentProcedureId")
        references "AppointmentProcedure"(id)
);

create table if not exists "Room" (
    id serial unique,
    name varchar not null,
    floor int not null,
    notes text not null,
    "locationId" int not null,

    primary key (id),

    constraint "fk_locationId"
        foreign key ("locationId")
        references "Location"(id)
);

create table if not exists "AppointmentProcedure" (
    id serial unique,
    name varchar(400) not null,
    duration int not null,
    price int not null,
    notes text not null,

    "doctorId" int not null,
    "roomId" int not null,
    "departmentId" int not null,
    "locationId" int not null,

    primary key (id),

     constraint "fk_doctorId"
        foreign key ("doctorId")
        references "Doctor"(id),

    constraint "fk_roomId"
        foreign key ("roomId")
        references "Room"(id),

    constraint "fk_departmentId"
        foreign key ("departmentId")
        references "Department"(id),

    constraint "fk_locationId"
        foreign key ("locationId")
        references "Location"(id)
);

create table if not exists "Department" (
    id serial unique ,
    name varchar not null
);
```

```
create table if not exists  "Admin" (
    id serial unique,
    "locationId" int not null ,

    primary key (id),

    constraint fk_locationId
        foreign key ("locationId")
        references "Location"(id)
) inherits ("User");
```

/Server/Department/department.js

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');

const GenerateQueriesDepartment =
require('../Storage/PostgreSQLQuery/GenerateQueriesDepartment');
const storageInterface = require('../connection');

const jsonParser = bodyParser.json();
const departmentQueryGenerator = new GenerateQueriesDepartment();

// Create
router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const newDepartment = req.body;

  const dbres = await
storageInterface.create(departmentQueryGenerator.generateCreateDepartment
(newDepartment));
  res.json({ status: 'OK', message: 'Department are being created' });
});

// Read
router.get('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const departmentId = Number(req.params['id']);
  const sqlQuery =
departmentQueryGenerator.generateGetDepartment(departmentId);
  const result = await storageInterface.select(sqlQuery);
  res.json({ departments: result.rows });
});

// Delete
router.delete('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const departmentId = Number(req.params['id']);
```

```
  const sqlQuery =
departmentQueryGenerator.generateDeleteDepartment(departmentId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ departments: result.rows });
});

module.exports = router;
```

/Server/EmailService/EmailService.js

```
const fetch = require("node-fetch");
const readConfigFile = require("read-config-file");
const path = require('path');

module.exports = class EmailService {
  constructor() {
    this.config = {};

    readConfigFile.loadConfig({
      packageKey: '1',
      configFilename: 'application.local',
      projectDir: path.resolve(__dirname)
    }).then((jsonCfg) => {
      this.config = jsonCfg.result;
    });
  }

  sendAccessData(name, username, password) {
    const templateParams = {
      'to_name': name,
      'message': `Username: ${username} Password: ${password}`,
      'link_name': 'http://mcs.ru',
      'from_name': 'Administrator',
    };

    return fetch(this.config.apiPath,
      {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Access-Control-Allow-Origin': '*',
          'Accept': '*/*',
        },
        body: JSON.stringify({
          'template_id': this.config.template_id,
          'service_id': this.config.service_id,
          'user_id': this.config.user_id,
          'template_params': templateParams,
          'accessToken': this.config.accessToken,
        })
      }
    );
  }
```

```
}
```

/Server/Location/location.js /

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');

const GenerateQueriesLocation =
require('../Storage/PostgreSQLQuery/GenerateQueriesLocation');
const storageInterface = require('../connection');

const jsonParser = bodyParser.json();
const locationQueryGenerator = new GenerateQueriesLocation();

router.get('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const locationId = Number(req.params['id']);
  const sqlQuery =
locationQueryGenerator.generateGetLocation(locationId);
  console.log(sqlQuery);
  const result = await storageInterface.select(sqlQuery);
  res.json({ locations: result.rows });
});

router.delete('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const locationId = Number(req.params['id']);
  const sqlQuery =
locationQueryGenerator.generateDeleteLocation(locationId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ locations: result.rows });
});

module.exports = router;
```

/Server/Procedure/procedure.js/

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');

const GenerateQueriesProcedure =
require('../Storage/PostgreSQLQuery/GenerateQueriesProcedure');
const storageInterface = require('../connection');

const jsonParser = bodyParser.json();
const procedureQueryGenerator = new GenerateQueriesProcedure();
```

```
// Create
router.post('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const newProcedure = req.body;

  const dbres = await
storageInterface.create(procedureQueryGenerator.generateCreateProcedure(n
ewProcedure));
  res.json({ status: 'OK', message: 'Procedure are being created' });
});

// Read
router.get('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const procedureId = Number(req.params['id']);
  const sqlQuery =
procedureQueryGenerator.generateGetProcedure(procedureId);
  console.log(sqlQuery);
  const result = await storageInterface.select(sqlQuery);
  res.json({ procedures: result.rows });
});

// Delete
router.delete('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const procedureId = Number(req.params['id']);
  const sqlQuery =
procedureQueryGenerator.generateDeleteProcedure(procedureId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ procedures: result.rows });
});

module.exports = router;
```

/Server/Room/room.js

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
const responseHeaders = require('../responseConfig');

const GenerateQueriesRoom =
require('../Storage/PostgreSQLQuery/GenerateQueriesRoom');
const storageInterface = require('../connection');

const jsonParser = bodyParser.json();
const roomQueryGenerator = new GenerateQueriesRoom();

// Create
router.post('/', jsonParser, async (req, res) => {
```

```
  res.set(responseHeaders);
  const newRoom = req.body;

  const dbres = await
storageInterface.create(roomQueryGenerator.generateCreateRoom(newRoom));
  res.json({ status: 'OK', message: 'Room are being created' });
});

// Read
router.get('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const roomId = Number(req.params['id']);
  const sqlQuery = roomQueryGenerator.generateGetRoom(roomId);
  console.log(sqlQuery);
  const result = await storageInterface.select(sqlQuery);
  res.json({ rooms: result.rows });
});

// Delete
router.delete('/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const roomId = Number(req.params['id']);
  const sqlQuery = roomQueryGenerator.generateDeleteRoom(roomId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ rooms: result.rows });
});

module.exports = router;
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesAdmin.js

```
module.exports = class GenerateQueriesAdmin {
  constructor() {}

  generateGetUserInfo(username) {
    let sqlQuery = `
      SELECT * FROM "Admin"
      WHERE username = '${username}';
    `;
    console.log(sqlQuery);
    return sqlQuery;
  }

  generateAuthorize(username) {
    let sqlQuery = `
      SELECT hashsum FROM "Admin"
      WHERE username = '${username}';
    `;

    return sqlQuery;
  }
```

```
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesAppointment.js

```
module.exports = class GenerateQueriesAppointment {
  constructor() { }

  generateCreateAppointment(data) {
    let keys = [];
    let values = [];

    for (let key in data) {
      keys.push(`"${key}"`);
      values.push(`'${data[key]}'`);
    }

    let sqlQuery = `
      INSERT INTO "Appointment"(${keys.join(', ')})
        values (${values.join(', ')});
    `;
    return sqlQuery;
  }

  generateGetAppointment(appointmentId) {
    let sqlQuery = `
      SELECT * FROM "Appointment"
      ${(appointmentId !== -1)
        ? `WHERE id = ${appointmentId}`
        : ''
      }
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateGetDoctorAppointment(doctorId) {
    let sqlQuery = `
      SELECT * FROM "Appointment"
      WHERE "doctorId" = ${doctorId}
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateDeleteAppointment(appointmentId) {
    let sqlQuery = `
      DELETE FROM "Appointment"
      WHERE id = ${appointmentId};
    `;
    return sqlQuery;
  }
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesCommon.js

```
module.exports = class GenerateQueriesCommon {
  constructor() { }

  updateUserColumns(id, table, data) {
    const updatedColumns = [];
    for (let key in data) {
      (key !== 'id' && key !== 'accessControl') &&
        updatedColumns.push(`"${key}" = '${data[key]}'`)
    }

    let sqlQuery = `
      UPDATE "${table}"
      SET ${updatedColumns.join(', ')}
      WHERE id = ${id};
    `;

    return sqlQuery;
  }

  generateGetUserInfo(id, table) {
    let sqlQuery = `
      SELECT * FROM "${table}"
      WHERE id = '${id}';
    `;
    console.log(sqlQuery);
    return sqlQuery;
  }
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesDepartment.js

```
module.exports = class GenerateQueriesDepartment {
  constructor() { }

  generateGetDepartment(departmentId) {
    let sqlQuery = `
      SELECT * FROM "Department"
      ${(departmentId !== -1)
        ? `WHERE id = ${departmentId}`
        : ''
      }
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateDeleteDepartment(departmentId) {
    let sqlQuery = `
      DELETE FROM "Department"
      WHERE id = ${departmentId};
    `;
```

```
    return sqlQuery;
  }

  generateCreateDepartment(data) {
    let keys = [];
    let values = [];

    for (let key in data) {
      keys.push(`"${key}"`);
      values.push(`'${data[key]}'`);
    }

    let sqlQuery = `
      INSERT INTO "Department"(${keys.join(', ')})
        values (${values.join(', ')});
    `;
    return sqlQuery;
  }
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesDoctor.js

```
module.exports = class GenerateQueriesDoctor {
    constructor() {}

    generateGetUserInfo(username) {
      let sqlQuery = `
        SELECT * FROM "Doctor"
        WHERE username = '${username}';
      `;
      return sqlQuery;
    }

    generateAuthorize(username) {
      let sqlQuery = `
        SELECT hashsum FROM "Doctor"
        WHERE username = '${username}';
      `;
      return sqlQuery;
    }

    generateNewDoctor(data) {
      let keys = [];
      let values = [];

      for (let key in data) {
        keys.push(`"${key}"`);
        values.push(`'${data[key]}'`);
      }

      let sqlQuery = `
        INSERT INTO "Doctor"(${keys.join(', ')})
          values (${values.join(', ')});
```

```
      `;
      return sqlQuery;
    }

    generateGetDoctor(doctorId) {
      let sqlQuery = `
        SELECT * FROM "Doctor"
        ${(doctorId !== -1)
          ? `WHERE id = ${doctorId}`
          : ''
        }
        ORDER BY id DESC;
      `;
      return sqlQuery;
    }

    generateDeleteDoctor(doctorId) {
      let sqlQuery = `
        DELETE FROM "Doctor"
        WHERE id = ${doctorId};
      `;
      return sqlQuery;
    }

    generateDeleteDoctor(doctorId) {
      let sqlQuery = `
        DELETE FROM "Appointment"
        WHERE "doctorId" = ${doctorId};
        DELETE FROM "Doctor"
        WHERE id = ${doctorId};
      `;
      return sqlQuery;
    }
  }
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesLocation.js

```
module.exports = class GenerateQueriesLocation {
  constructor() { }

  generateGetLocation(locationId) {
    let sqlQuery = `
      SELECT * FROM "Location"
      ${(locationId !== -1)
        ? `WHERE id = ${locationId}`
        : ''
      }
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateDeleteLocation(locationId) {
```

```
    let sqlQuery = `
      DELETE FROM "Location"
      WHERE id = ${locationId};
    `;
    return sqlQuery;
  }
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesPatient.js

```
module.exports = class GenerateQueriesPatient {
  constructor() { }

  generateGetUserInfo(username) {
    let sqlQuery = `
      SELECT * FROM "Patient"
      WHERE username = '${username}';
    `;
    return sqlQuery;
  }

  generateAuthorize(username) {
    let sqlQuery = `
      SELECT hashsum FROM "Patient"
      WHERE username = '${username}';
    `;
    return sqlQuery;
  }

  generateNewPatient(data) {
    let keys = [];
    let values = [];
    console.log('new patient', data);

    for (let key in data) {
      keys.push(`"${key}"`);
      values.push(`'${data[key]}'`);
    }

    let sqlQuery = `
      INSERT INTO "Patient"(${keys.join(', ')})
        values (${values.join(', ')});
    `;
    return sqlQuery;
  }

  generateGetPatient(patientId) {
    let sqlQuery = `
      SELECT * FROM "Patient"
      ${(patientId !== -1)
        ? `WHERE id = ${patientId}`
        : ''
      }
```

```
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateDeletePatient(patientId) {
    let sqlQuery = `
      DELETE FROM "Appointment"
      WHERE "patientId" = ${patientId};
      DELETE FROM "Patient"
      WHERE id = ${patientId};
    `;
    return sqlQuery;
  }
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesProcedure.js

```
module.exports = class GenerateQueriesProcedure {
  constructor() { }

  generateGetProcedure(procedureId) {
    let sqlQuery = `
      SELECT * FROM "AppointmentProcedure"
      ${(procedureId !== -1)
        ? `WHERE id = ${procedureId}`
        : ''
      }
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateDeleteProcedure(procedureId) {
    let sqlQuery = `
      DELETE FROM "Appointment"
      WHERE "procedureId" = ${procedureId};

      DELETE FROM "AppointmentProcedure"
      WHERE id = ${procedureId};
    `;
    return sqlQuery;
  }

  generateCreateProcedure(data) {
    let keys = [];
    let values = [];

    for (let key in data) {
      keys.push(`"${key}"`);
      values.push(`'${data[key]}'`);
    }
```

```
    let sqlQuery = `
      INSERT INTO "AppointmentProcedure"(${keys.join(', ')})
        values (${values.join(', ')});
    `;
    return sqlQuery;
  }
}
```

/Server/Storage/PostgreSQLQuery/GenerateQueriesRoom.js

```
module.exports = class GenerateQueriesRoom {
  constructor() { }

  generateGetRoom(roomId) {
    let sqlQuery = `
      SELECT * FROM "Room"
      ${(roomId !== -1)
        ? `WHERE id = ${roomId}`
        : ''
      }
      ORDER BY id DESC;
    `;
    return sqlQuery;
  }

  generateDeleteRoom(roomId) {
    let sqlQuery = `
      DELETE FROM "Appointment"
      WHERE "roomId" = ${roomId};
      DELETE FROM "Room"
      WHERE id = ${roomId};
    `;
    return sqlQuery;
  }

  generateCreateRoom(data) {
    let keys = [];
    let values = [];

    for (let key in data) {
      keys.push(`"${key}"`);
      values.push(`'${data[key]}'`);
    }

    let sqlQuery = `
      INSERT INTO "Room"(${keys.join(', ')})
        values (${values.join(', ')});
    `;
    return sqlQuery;
  }
}
```

/Server/Storage/storageInterface.js

```
const { request } = require('express');
const { _ } = require('lodash');
const { Client } = require('pg');
const pg = require('pg');
const ConfigurationManager = require('./configurationManager');
const GenerateQueriesAdmin =
require('./PostgreSQLQuery/GenerateQueriesAdmin');
const GenerateQueriesPatient =
require('./PostgreSQLQuery/GenerateQueriesPatient');
const GenerateQueriesDoctor =
require('./PostgreSQLQuery/GenerateQueriesDoctor');
const GenerateQueriesAppointment =
require('./PostgreSQLQuery/GenerateQueriesAppointment');
const pool = new pg.Pool();

const generateQueriesAdmin = new GenerateQueriesAdmin();
const generateQueriesPatient = new GenerateQueriesPatient();
const generateQueriesDoctor = new GenerateQueriesDoctor();
const generateQueriesAppointment = new GenerateQueriesAppointment();

module.exports = class StorageInterface {
  /* Implementation of CRUD interface */
  constructor() {
    const configurationManager = new ConfigurationManager();
    this.configuration = configurationManager.importConfiguration();
    console.log('Imported database config:', this.configuration);

    this.client = new Client({
      user: 'postgres',
      host: 'localhost',
      database: 'testdb',
      password: 'rara22',
      port: 5432,
    })

    this.client.connect();
  }

  create = async query => {
    const result = await this.client.query(query);
    return result;
  }

  select = async query => {
    const result = await this.client.query(query);
    return result;
  }

  update = async (query) => {
    const result = await this.client.query(query);
    return result;
```

```
  }

  delete = async (query) => {
    const result = await this.client.query(query);
    return result;
  }

  authorize = async (username, password) => {
    const adminData = await
this.client.query(generateQueriesAdmin.generateGetUserInfo(username));
    const patientData = await
this.client.query(generateQueriesPatient.generateGetUserInfo(username));
    const doctorData = await
this.client.query(generateQueriesDoctor.generateGetUserInfo(username));

    return Promise
      .all([adminData, patientData, doctorData])
      .then(results => {
        const data = results.filter(res => res.rowCount);
        return data.length ? data[0].rows[0] : {};
      });
  }

  testConnection() {
    this.client
      .query('SELECT NOW()')
      .then(result => console.log(result))
      .catch(e => console.error(e.stack))
      .then(() => this.client.end());
  }
}
```

/Server/User/user.js

```
// Libraries
const express = require('express');
const bodyParser = require('body-parser');
var cyrillicToTranslitJs = require("cyrillic-to-translit-js");
const router = express.Router();
const responseHeaders = require('../responseConfig');

// Storage
const GenerateQueriesCommon =
require('../Storage/PostgreSQLQuery/GenerateQueriesCommon');
const GenerateQueriesPatient =
require('../Storage/PostgreSQLQuery/GenerateQueriesPatient');
const GenerateQueriesDoctor =
require('../Storage/PostgreSQLQuery/GenerateQueriesDoctor');
const storageInterface = require('../connection');

// Utils and services
const EmailService = require('../EmailService/EmailService');
const utils = require('../utils');
```

```
const { generateNRandomNumbers, generatePassword } = utils;

const jsonParser = bodyParser.json();

const commonQueryGenerator = new GenerateQueriesCommon();
const patientQueryGenerator = new GenerateQueriesPatient();
const doctorQueryGenerator = new GenerateQueriesDoctor();
const cyrillicToTranslit = new cyrillicToTranslitJs();
const emailService = new EmailService();

router.put('/', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const data = req.body;
  const query = commonQueryGenerator.updateUserColumns(data.id,
data.accessControl, data);
  const result = await storageInterface.update(query);
  const updatedUser = await
storageInterface.select(commonQueryGenerator.generateGetUserInfo(data.id,
data.accessControl));
  res.json(updatedUser.rows[0]);
});

/* PATIENTS */

router.post('/patient', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const data = req.body;

  const username = cyrillicToTranslit.transform(
    `${data.firstName} ${data.lastName} `, '-',
  ) + generateNRandomNumbers(2).join('');
  const password = generatePassword();

  const newPatient = {
    ...data,
    username,
    hashsum: password,
    accessControl: 'Patient',
  }

  const dbres = await
storageInterface.create(patientQueryGenerator.generateNewPatient(newPatie
nt));
  emailService.sendAccessData(newPatient.firstName, username, password)
    .then(
      (response) => res.json({ result: 'OK', status: response.status,
text: response.text }),
      (err) => res.json({ result: 'ERROR', error: err })
    );
});

router.get('/patient/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);
```

```
  const patientId = Number(req.params['id']);
  const sqlQuery = patientQueryGenerator.generateGetPatient(patientId);
  console.log(sqlQuery);
  const result = await storageInterface.select(sqlQuery);
  res.json({ patients: result.rows });
});

router.delete('/patient/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const patientId = Number(req.params['id']);
  const sqlQuery =
patientQueryGenerator.generateDeletePatient(patientId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ patients: result.rows });
});

/* DOCTORS */

router.post('/doctor', jsonParser, async (req, res) => {
  res.set(responseHeaders);
  const data = req.body;

  const username = cyrillicToTranslit.transform(
    `${data.firstName} ${data.lastName} `, '-',
  ) + generateNRandomNumbers(2).join('');
  const password = generatePassword();

  const newDoctor = {
    ...data,
    username,
    hashsum: password,
    accessControl: 'Doctor',
  };

  const dbres = await
storageInterface.create(doctorQueryGenerator.generateNewDoctor(newDoctor)
);
  emailService.sendAccessData(newDoctor.firstName, username, password)
    .then(
      (response) => res.json({ result: 'OK', status: response.status,
text: response.text }),
      (err) => res.json({ result: 'ERROR', error: err })
    );
});

router.get('/doctor/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const doctorId = Number(req.params['id']);
  const sqlQuery = doctorQueryGenerator.generateGetDoctor(doctorId);
  console.log(sqlQuery);
```

```
  const result = await storageInterface.select(sqlQuery);
  res.json({ doctors: result.rows });
});

router.delete('/doctor/:id', jsonParser, async (req, res) => {
  res.set(responseHeaders);

  const doctorId = Number(req.params['id']);
  const sqlQuery = doctorQueryGenerator.generateDeleteDoctor(doctorId);
  const result = await storageInterface.delete(sqlQuery);
  res.json({ patients: result.rows });
});

module.exports = router;
```

/Server/app.js

```
const express = require('express');
const cookieParser = require('cookie-parser');
const cors = require('cors');
const app = express();

const port = 3000;
const corsConfig = { credentials: true, origin: 'http://localhost:8080'
};

/* Routers */
const auth = require('./Auth/auth');
const logout = require('./Auth/logout');
const user = require('./User/user');
const location = require('./Location/location');
const room = require('./Room/room');
const procedure = require('./Procedure/procedure');
const department = require('./Department/department');
const appointment = require('./Appointment/appointment');

app.use(cors(corsConfig));
app.use(cookieParser());
app.use('/auth', auth);
app.use('/logout', logout);
app.use('/user', user);
app.use('/location', location);
app.use('/room', room);
app.use('/procedure', procedure);
app.use('/department', department);
app.use('/appointment', appointment);

app.listen(port, () => {
  console.log(`Listening at http://localhost:${port}`)
});
```

/Server/connection.js

```
const StorageInterface = require("./Storage/storageInterface");

const storageInterface = new StorageInterface();

module.exports = storageInterface;
```

/Server/package.json

```
{
  "name": "medical-clinic-service-server",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "dev": "nodemon app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Artur Yumaev",
  "license": "ISC",
  "dependencies": {
    "@types/express": "^4.17.11",
    "@types/lodash": "^4.14.168",
    "body-parser": "^1.19.0",
    "cookie-parser": "^1.4.5",
    "cors": "^2.8.5",
    "cyrillic-to-translit-js": "^3.1.0",
    "emailjs-com": "^2.6.4",
    "express": "^4.17.1",
    "lodash": "^4.17.21",
    "node-fetch": "^2.6.1",
    "pg": "^8.6.0",
    "read-config-file": "^6.1.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

/Server/responseConfig.js

```
module.exports = {
  'Content-Type': 'application/json;charset=utf-8',
}
```

/Server/utils.js

```
const generateNRandomNumbers = N => {
  var arr = [];
  while (arr.length < N) {
    var r = Math.floor(Math.random() * 100) + 1;
```

```
    (arr.indexOf(r) === -1) && arr.push(r);
  }

  return arr;
};

const generatePassword = () => {
  var length = 8,
      charset =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",
      retVal = "";
  for (var i = 0, n = charset.length; i < length; ++i) {
      retVal += charset.charAt(Math.floor(Math.random() * n));
  }
  return retVal;
}

module.exports = {
  generateNRandomNumbers,
  generatePassword,
}
```

/Client/babel.config.json

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "useBuiltIns": "usage",
        "corejs": 3
      }
    ],
    "@babel/preset-react",
    "@babel/preset-typescript"
  ],
  "plugins": [
    "@babel/proposal-class-properties",
    "@babel/proposal-object-rest-spread"
  ]
}
```

/Client/package.json

```
{
  "name": "medical-clinic-service",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "dev": "webpack serve --open --hot",
    "build": "webpack --mode production --progress"
```

```
  },
  "devDependencies": {
    "@babel/core": "^7.12.16",
    "@babel/plugin-proposal-class-properties": "^7.12.13",
    "@babel/plugin-proposal-object-rest-spread": "^7.12.13",
    "@babel/preset-env": "^7.12.16",
    "@babel/preset-react": "^7.12.13",
    "@babel/preset-typescript": "^7.12.16",
    "@types/lodash": "^4.14.169",
    "@types/react": "^17.0.2",
    "@types/react-big-calendar": "^0.31.0",
    "@types/react-dom": "^17.0.1",
    "@types/react-router-dom": "^5.1.7",
    "@types/styled-components": "^5.1.9",
    "babel-loader": "^8.2.2",
    "core-js": "^3.8.3",
    "css-loader": "^5.0.2",
    "raw-loader": "^4.0.2",
    "regenerator-runtime": "^0.13.7",
    "sass-loader": "^11.0.1",
    "style-loader": "^2.0.0",
    "typescript": "^4.1.5",
    "webpack": "^5.22.0",
    "webpack-cli": "^4.5.0",
    "webpack-dev-server": "^3.11.2"
  },
  "dependencies": {
    "antd": "^4.12.3",
    "bootstrap": "^4.6.0",
    "dayjs": "^1.10.4",
    "globalize": "^1.6.0",
    "less-loader": "^8.0.0",
    "lodash": "^4.17.21",
    "moment": "^2.29.1",
    "node-sass": "^5.0.0",
    "react": "^17.0.1",
    "react-big-calendar": "^0.33.2",
    "react-dom": "^17.0.1",
    "react-loader-spinner": "^4.0.0",
    "react-redux": "^7.2.4",
    "react-router-dom": "^5.2.0",
    "redux": "^4.1.0",
    "redux-persist": "^6.0.0",
    "redux-thunk": "^2.3.0",
    "sass": "^1.32.7",
    "styled-components": "^5.2.3",
    "typesafe-actions": "^5.1.0",
    "uninstall": "0.0.0"
  }
}
```

/Client/webpack.config.js

```
const path = require('path')

module.exports = {
  entry: './src/index.tsx',
  mode: 'development',
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.(js|ts)x?$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
        },
      },
      {
        test: /\.(s[ac]ss|css)$/i,
        use: ["style-loader", "css-loader", "sass-loader"],
      }
    ],
  },
  resolve: {
    alias: {
      Interfaces: path.resolve(__dirname, 'src/interfaces'),
      Ccomponents: path.resolve(__dirname, 'src/components'),
    },
    extensions: ['.tsx', '.ts', '.jsx', '.js'],
  },
  devServer: {
    contentBase: path.resolve(__dirname, './dist'),
    historyApiFallback: true,
  }
}
```

/Client/src/api/Api.ts

```
enum HTTPMethod {
  GET = 'GET',
  POST = 'POST',
  PUT = 'PUT',
  DELETE = 'DELETE',
}

const fetchApi = (entity: string, method: HTTPMethod, postData?: object):
Promise<Response> => {
  const protocol = 'http';
  const host = 'localhost';
  const port = 3000;
```

```
  const url = `${protocol}://${host}:${port}/${entity}`;

  let requestOptions: RequestInit = {
    credentials: 'include',
    method: method,
    headers: {
      'Content-Type': 'application/json;charset=utf-8',
      'Access-Control-Allow-Origin': '*',
    }
  };

  if ((method == HTTPMethod.POST || method == HTTPMethod.PUT) &&
postData) {
    requestOptions.body = JSON.stringify(postData);
  }

  return fetch(url, requestOptions);
};

export {
  fetchApi as fetchApi,
  HTTPMethod as HTTPMethod,
};
```

/Client/src/interfaces/Appointment/Appointment.ts

```
import WithId from '../WithId';

interface Appointment extends WithId {
  scheduledTime:string;
  scheduledEndTime: string;
  created: string;
  notes: string;

  patientId: number;
  doctorId: number;
  appointmentProcedureId: number;
  roomId: number;
}

export default Appointment;
```

/Client/src/interfaces/Appointment/AppointmentProcedure.ts

```
import WithId from "../WithId";

export default interface AppointmentProcedure extends WithId {
  name: string;
  duration: number;
  price: number;
  notes: string;
```

```
  doctorId: number;
  locationId: number;
  roomId: number;
  departmentId: number;
}
```

/Client/src/interfaces/Appointment/AppointmentStatus.ts

```
enum AppointmentStatus {
    Confirmed = 'Confirmed',
    InProcess = 'In Process',
    Done = 'Done',
}

export default AppointmentStatus
```

/Client/src/interfaces/Admin.ts

```
import User from "./User";

export default interface Admin extends User {
  locationId: number;
  location: Location;
}
```

/Client/src/interfaces/Department.ts

```
import WithId from "./WithId";

export interface Department extends WithId {
  name: string;
}
```

/Client/src/interfaces/Doctor.ts

```
import User from "./User";

export default interface Doctor extends User {
  locationId: number;
  departmentId: number;
  workExperience: number;
  academicDegree: string;
  notes: string;
}
```

/Client/src/interfaces/Location.ts

```
import WithId from "./WithId";

export default interface Location extends WithId {
  locationName: string;
```

```
  address: string;
  phone: string;
  email: string;
}
```

/Client/src/interfaces/Patient.ts

```
import User from "./User";

export default interface Patient extends User {
  emergencyContactName: string;
  emergencyContactPhone: string;
  emergencyContactRelation: string;
}
```

/Client/src/interfaces/WithId.ts

```
interface WithId {
  id: number;
}
export default WithId
```

/Client/src/interfaces/Utils/AccessControl.ts

```
enum AccessControl {
  Admin = 'Admin',
  Doctor = 'Doctor',
  Patient = 'Patient',
}
export default AccessControl
```

/Client/src/interfaces/Utils/Currency.ts

```
enum Currency {
    RUB = "РУБ",
    USD = "USD",
}
export default Currency
```

/Client/src/interfaces/Utils/Gender.ts

```
enum Gender {
  Male = 'Male',
  Female = 'Female',
  Other = 'Other',
}
export default Gender;
```