МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт цифровых технологий
(наименование института полностью)

09.03.03 Прикладная информатика
(код и наименование направления подготовки, специальности)

Разработка программного обеспечения
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Разработка приложения для автоматизации управления эксплуатацией жилого фонда»

Обучающийся И.С. Михеев
(Инициалы Фамилия) (личная подпись)

Руководитель Н.Н. Казаченок
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Тема бакалаврской работы – «Разработка приложения для автоматизации управления эксплуатацией жилого фонда».

Актуальность работы состоит в том, что приложение для автоматизации управления жилым фондом позволит повысить эффективность работы управляющих компаний и улучшить качество услуг для жильцов.

Объектом исследования является процесс управления эксплуатацией жилого фонда.

Предмет исследования – программное обеспечение для автоматизации управления жилым фондом.

Целью данной работы является разработка приложения для автоматизации управления эксплуатацией жилого фонда, которое позволит оптимизировать все процессы и улучшить качество обслуживания.

Выпускная квалификационная работа состоит из введения, трех глав, заключения, списка используемой литературы и приложений.

В первой главе приводится описание предметной области, анализ существующего программного обеспечения и формулируются требования к разрабатываемому приложению.

Во второй главе рассматриваются этапы проектирования программного обеспечения.

В третьей главе приводятся результаты реализации и тестирования приложения.

В заключении описаны результаты выпускной квалификационной работы.

Практическая значимость работы заключается в разработке приложения, обеспечивающего автоматизацию процесса управления эксплуатацией жилого фонда.

Бакалаврская работа состоит из 76 страниц текста, 23 рисунков, 1 таблицы, 25 источников и 9 приложений.

Оглавление

Введение

Управление эксплуатацией жилого фонда – это комплекс организационно-технических мероприятий по поддержанию жилых зданий в надлежащем состоянии, включая техническое обслуживание, текущий и капитальный ремонт, управление коммунальными услугами и создание комфортных условий проживания для жильцов.

Организации в сфере управления эксплуатацией жилого фонда решают задачи по обработке больших объемов информации и сложных регламентированных процессов. Поэтому разработка программного обеспечения, которое позволит оптимизировать процессы, централизовать данные, автоматизировать различные задачи, обеспечить соблюдение регламентов и качественное взаимодействие между субъектами является важнейшей задачей.

Актуальность данной работы заключается в том, что приложение для автоматизации управления эксплуатацией жилого фонда позволит обеспечить эффективную работу организации: автоматизировать рутинные задачи, сократить затраты и время на выполнение заявок, а также улучшить контроль и оперативное реагирование.

Объектом исследования является процесс управления эксплуатацией жилого фонда.

Предмет исследования – программное обеспечение для автоматизации управления жилым фондом.

Целью данной выпускной квалификационной работы является разработка приложения для автоматизации управления эксплуатацией жилого фонда, которое позволит оптимизировать все процессы и улучшить качество обслуживания.

Для достижения цели были поставлены следующие задачи: изучить и проанализировать существующее в этой области программное обеспечение;

- разработать требования к приложению;

– спроектировать архитектуру программного обеспечения;
– создать приложение и произвести его тестирование.

Методы исследования – методы и технологии проектирования информационных систем, технологии программирования.

Практическая значимость работы заключается в разработке приложения, обеспечивающего автоматизацию процесса управления эксплуатацией жилого фонда.

В первой главе приводится анализ и описание предметной области, аналогичного программного обеспечения и формулируются требования к разрабатываемому приложению.

Во второй главе рассматриваются этапы проектирования программного обеспечения.

В третьей главе приводятся результаты реализации и тестирования приложения.

В заключении описаны результаты выпускной квалификационной работы, сделаны выводы.

Результаты бакалаврской работы представляют собой научно-практический интерес и могут быть рекомендованы к использованию в организациях по управлению жилым фондам. Также возможно дальнейшее усовершенствование приложения.

Глава 1 Постановка задачи на разработку приложения для автоматизации управления эксплуатацией жилого фонда

1.1 Особенности программного обеспечения для управления эксплуатацией жилого фонда

Эффективное управление эксплуатацией жилого фонда подразумевает использование специализированных программ в работе управляющих компаний (УК) и товариществ собственников жилья (ТСЖ). Для автоматизации рабочих процессов и эффективного взаимодействия с жильцами такое программное обеспечение должно осуществлять ряд определенных функций [2].

Специализированному приложению по управлению эксплуатацией жилого фонда необходимы следующие опции:

- начисление платежей за коммунальные услуги и их учет: приложение должно автоматически рассчитывать и начислять квартплату, вести учет поступающих от жильцов платежей, а также отслеживать задолженности и переплаты – это позволит исключить ошибки в расчетах и ускорит процесс обработки информации о платежах;
- онлайн-оплата начислений и долгов: напоминание о начислениях коммунальных платежей и возможность оплатить их онлайн – важный и удобный аспект, позволяющий жильцам экономить время и своевременно производить оплату;
- диспетчерское обслуживание онлайн: прием от жильцов заявок на ремонт и обслуживание, обработка заявок, оповещение о ходе выполнения работ, а также контроль исполнения заявок со стороны управления – необходимая часть приложения;

- мобильная версия приложения для жителей: возможность получать уведомления о начислениях и событиях, оплачивать коммунальные платежи, передавать показания приборов учета и оставлять заявки через мобильное приложение – простота и удобство взаимодействия жильцов и УК или ТСЖ;
- формирование отчетов и анализ деятельности: инструменты для создания отчетности и анализа бюджета организации, ее финансовой деятельности и предоставление информации контролирующим органам обеспечивают эффективное управление;
- взаимодействие со сторонними сервисами: для автоматизации процесса обмена данными и эффективной работы необходима интеграция с внешними системами: платежными системами, банками, информационными системами (ГИС ЖКХ) и другими;
- проведение опросов жильцов: онлайн-опрос жильцов, проведение собраний и сбор голосов – функция, упрощающая решение вопросов управления жилым фондом.

Современные специализированные приложения, включающие в себя вышеуказанные функции, позволяют автоматизировать выполнение рутинных задач, повысить эффективность обслуживания жилого фонда со стороны УК и ТСЖ и удовлетворить запросы со стороны жильцов [16].

1.2 Обзор и анализ аналогов программного обеспечения

Выбор программного обеспечения для УК и ТСЖ важен для повышения эффективности управления жилым фондом, так как современные программы автоматизируют рутинные процессы, облегчая такие задачи, как расчет квартплаты, прием заявок от жильцов, учет обращений и контроль выполнения работ.

Одними из наиболее известных и востребованных являются: «1С:ЖКХ», «Диспетчер 24», «Инфокрафт ЖКХ 365», «Жилфонд Сервис».

«1С:ЖКХ – это специализированное программное обеспечение для автоматизации работы УК и ТСЖ, разработанное на базе платформы «1С:Предприятие», оно помогает упростить и оптимизировать процессы, связанные с расчетом квартплаты, управлением объектами недвижимости, ведением бухгалтерского и налогового учета, а также взаимодействием с ресурсоснабжающими организациями и жильцами. Благодаря интеграции с ГИС ЖКХ, программа обеспечивает оперативный обмен данными и позволяет избежать дублирования информации, что делает её незаменимым инструментом для малых и средних организаций в данной сфере» [18].

Плюсы:

- автоматический расчет коммунальных платежей, учет задолженностей и переплат;
- интеграция с внешними информационными системами (ГИС ЖКХ);
- простой и понятный всем пользователям веб-интерфейс для работы и дистанционного доступа;
- гибкие настройки под нужды различных ЖКХ-организаций;
- регулярные обновления и соблюдение действующего законодательства.

Минусы:

- высокая сложность внедрения и эксплуатации без квалифицированных специалистов;
- необходимость платного обновления программного обеспечения;
- возможные проблемы с интеграцией и производительностью при работе с большими объемами данных.

Окно интерфейса «1С:ЖКХ» представлено на рисунке 1.

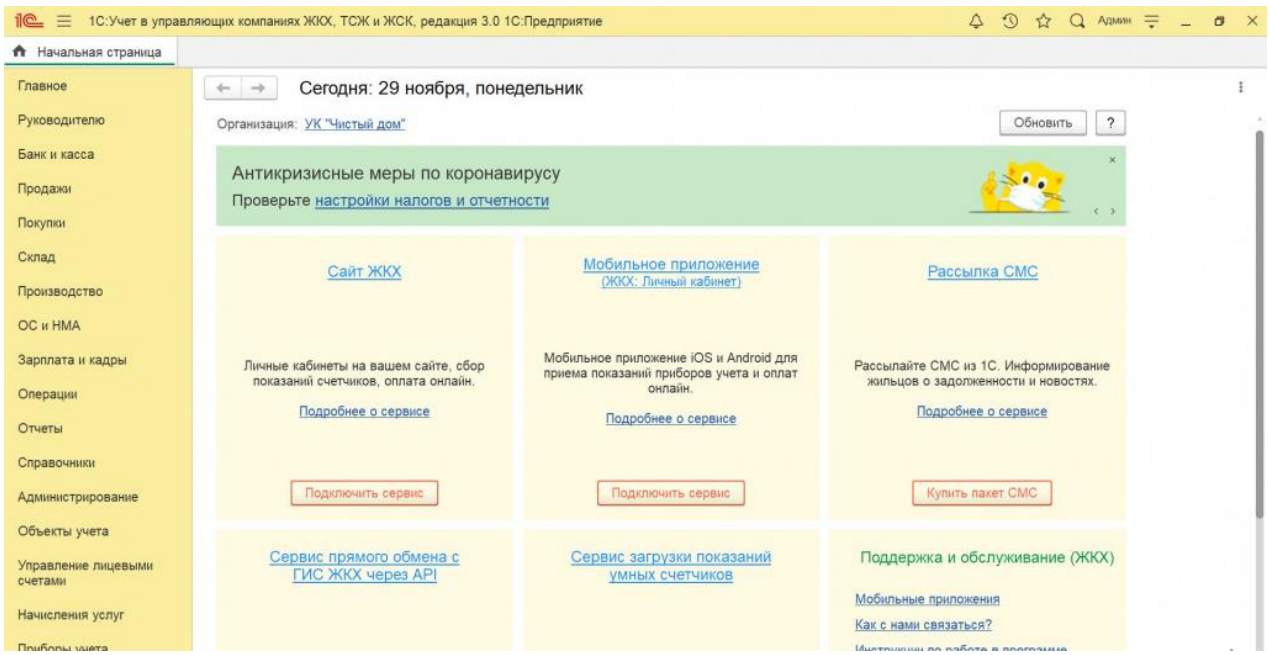

Рисунок 1 – Интерфейс программы «1С:ЖКХ»

«Диспетчер 24 – это цифровая платформа, специально разработанная для автоматизации работы УК и ТСЖ. Этот сервис позволяет эффективно управлять заявками от жильцов, отслеживать выполнение задач и улучшать коммуникацию между управляющей организацией и жильцами. Благодаря интеграции с современными технологиями, такими как облачные решения, платформа обеспечивает бесперебойную работу и высокую отказоустойчивость, что особенно важно для крупных компаний с большим количеством пользователей» [18].

«Платформа также помогает автоматизировать рутинные процессы, такие как обработка заявок, управление задолженностями, и взаимодействие с подрядчиками. Это не только ускоряет работу, но и снижает риски, связанные с человеческим фактором» [18].

Плюсы:

– автоматизированный процесс обработки заявок, мониторинг их выполнения;

– удобство пользования для жильцов обеспечивается интеграцией с мобильным приложением;
– применение облачных технологий для обеспечения высокой устойчивости к сбоям;
– простота и интуитивность пользовательского интерфейса для ТСЖ и УК.

Минусы:

– ограничения по объему текста, что делает платформу менее наглядной и функциональной, чем аналоги;
– высокая стоимость.

Интерфейс программы представлен на рисунке 2.

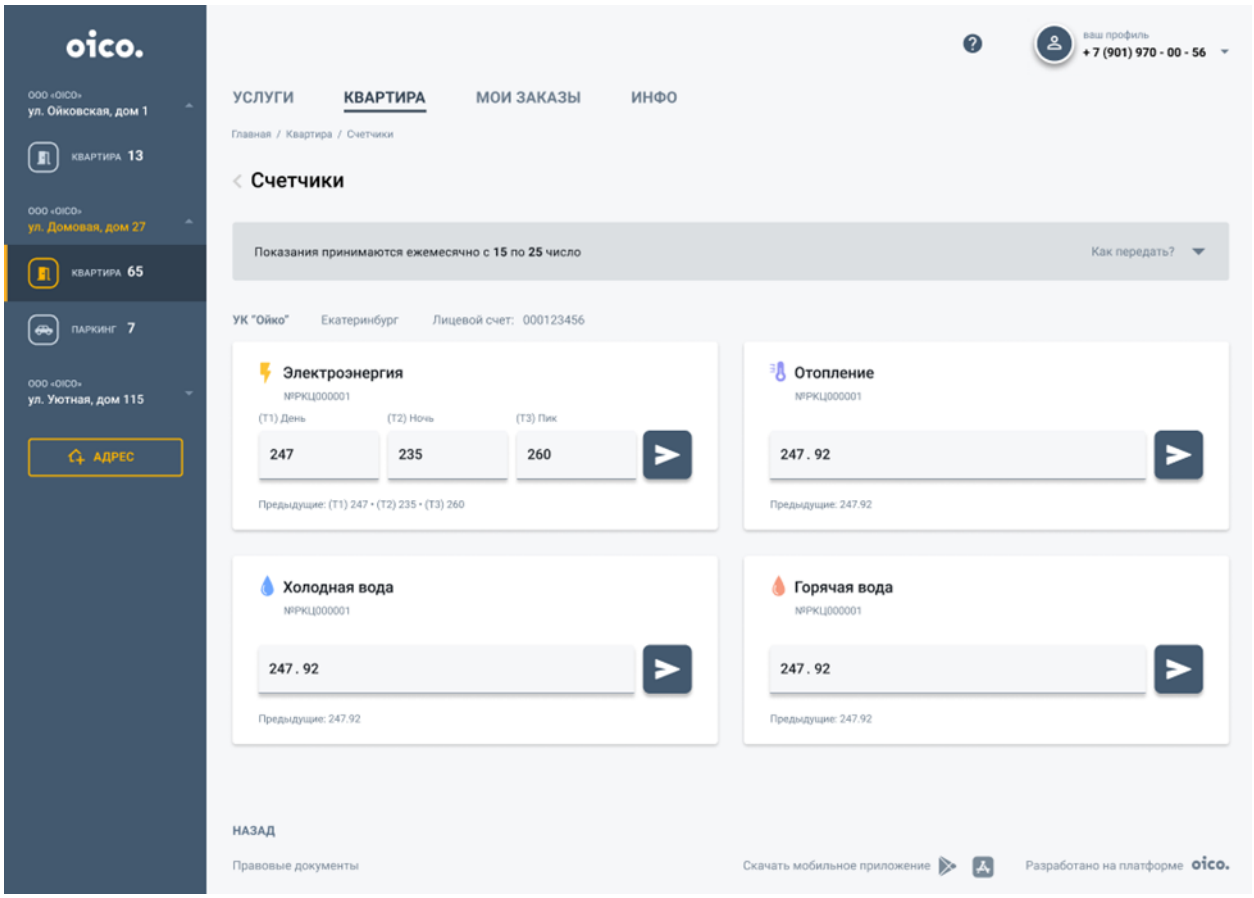

Рисунок 2 – Интерфейс программы «Диспетчер 24»

«Инфокрафт ЖКХ 365 – это комплекс программных решений, предназначенных для автоматизации управления и бухгалтерского учета в управляющих компаниях и товариществах собственников жилья. Разработанное на платформе «1С:Предприятие», это ПО предоставляет инструменты для расчета квартплаты, ведения бухгалтерского учета, учета жилого фонда и обслуживания собственников жилья. Благодаря интеграции с облачными сервисами, программы «Инфокрафт ЖКХ 365» позволяют работать из любой точки мира, что особенно удобно для современных управляющих организаций» [18].

«Это решение отличается высокой степенью автоматизации рутинных процессов и помогает существенно сократить затраты времени на ведение документации. Важным преимуществом является возможность настройки под конкретные нужды организации. Помимо расчетов и учета, «Инфокрафт ЖКХ 365» позволяет вести претензионную работу и интегрироваться с государственными информационными системами, такими как ГИС ЖКХ» [18].

Плюсы:

- использование платформы «1С:Предприятие» гарантирует надежную и стабильную работу;
- мобильность пользователей и удобный доступ обеспечиваются возможностью работы в облаке;
- автоматизированный расчет стоимости коммунальных услуг и квартплаты;
- возможность настройки приложения под специфические требования каждого ТСЖ или УК;
- интеграция с ГИС ЖКХ.

Минусы:

- платное использование мобильного приложения для жильцов;

– возможные ошибки и нестабильность работы, особенно при обновлениях.

Окно интерфейса приложения представлено на рисунке 3.

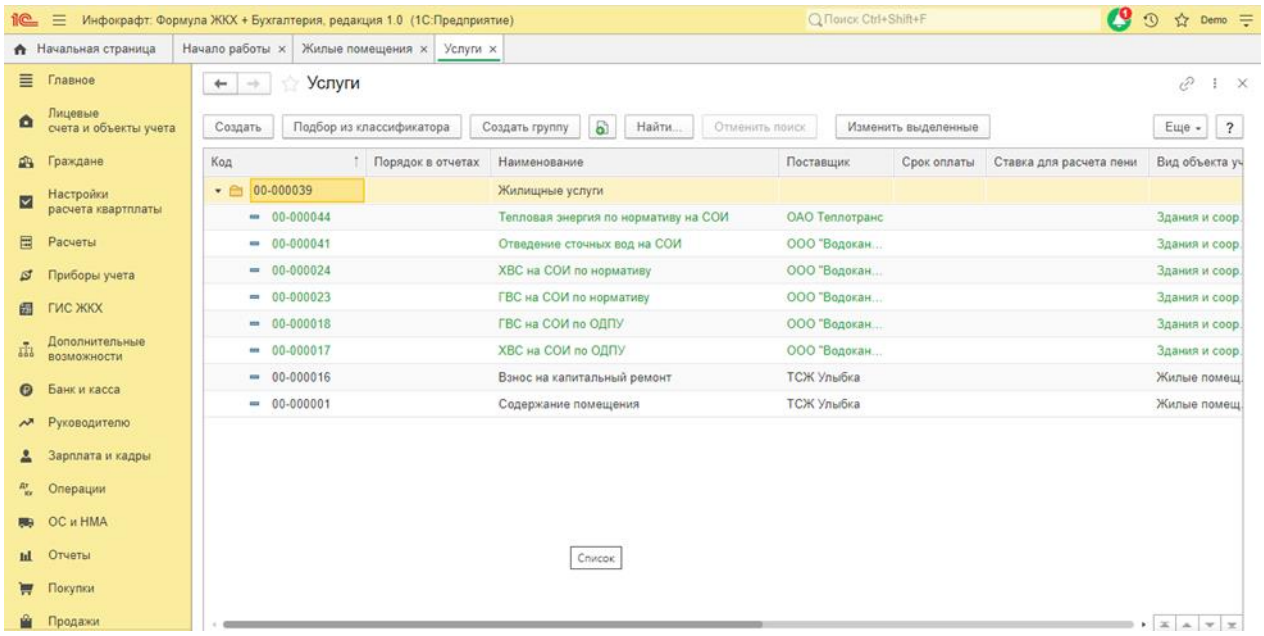

Рисунок 3 – Интерфейс приложения «Инфокрафт ЖКХ 365»

«Жилфонд Сервис – это специализированная программа, предназначенная для автоматизации деятельности управляющих компаний и товариществ собственников жилья. Она создана для упрощения управления жилищным фондом, обеспечения прозрачности расчетов и улучшения взаимодействия с жителями. Программа предоставляет широкий функционал для учета коммунальных платежей, управления заявками, контроля за состоянием объектов и работы с должниками. «Жилфонд Сервис» помогает существенно экономить время, автоматизируя многие процессы, что позволяет управляющим компаниям сосредоточиться на более стратегических задачах» [18].

Плюсы:

– интерфейс интуитивно понятный, не требует много времени на обучение персонала;

– интеграция с государственными информационными системами;
– автоматизированное начисление коммунальных платежей и их учет;
– удобное управление с любого устройства обеспечивается благодаря поддержке мобильных приложений;
– повышенная защита данных с регулярными обновлениями мер безопасности.

Минусы:

– неудобный интерфейс;
– платное использование мобильного приложения.

На рисунке 4 представлен интерфейс программы.

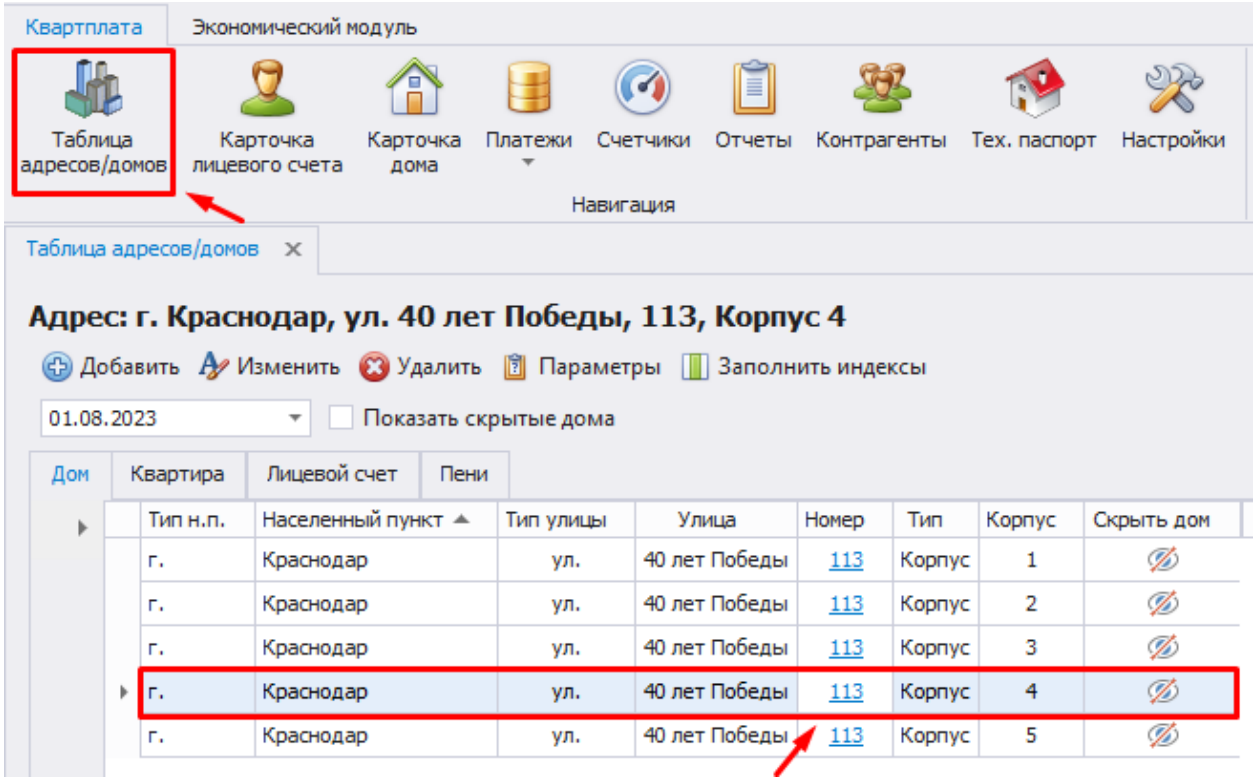

Рисунок 4 – Интерфейс программы «Жилфонд Сервис»

Проанализировав представленные программы, можно сделать заключение, что для качественной работы и эффективной автоматизации

управления эксплуатацией жилого фонда необходима интеграция программного обеспечения с государственными системами, ресурсоснабжающими организациями и мобильными устройствами, что обеспечит повышение оперативности работы и удобства для пользователей.

Выявленные преимущества и недостатки существующих программ позволяют сформулировать требования к разработке программного обеспечения.

1.3 Разработка требований к приложению

Разрабатываемое приложение должно соответствовать определенным требованиям, которые представляют собой совокупность запросов относительно атрибутов, свойств или качеств программной системы, подлежащей реализации [14].

Исходя из анализа аналогов программного обеспечения, приложение для автоматизации управления эксплуатацией жилого фонда должно отвечать следующим требованиям: функциональным, нефункциональным, и бизнес-требованиям. Архитектура приложения также должна соответствовать определенным требованиям [10].

Функциональные требования:

- регистрация сотрудников с указанием должностей, их авторизация для доступа к различным сервисам;
- регистрация и управление заявками от жильцов с отслеживанием их статуса;
- автоматический расчет и формирование квитанций на оплату жилищно-коммунальных услуг;
- формирование отчетов о платежах, долгах и состоянии жилого фонда;
- ведение базы данных жильцов и объектов недвижимости;

- управление техническим обслуживанием и ремонтами, включая связь с подрядчиками;
- формирование отчетов (по заявкам, обслуживанию, материальной базе, финансовых);
- интеграция с модулем уведомлений;
- интеграция с внешними государственными системами.

Нефункциональные требования:

- производительность: выдерживать до 300–500 заявок в сутки без значительного снижения отклика;
- надежность: резервное копирование базы данных, базовые механизмы отказоустойчивости (горячая или холодная запасная копия сервиса, автоматическое переключение);
- безопасность: защита конфиденциальных данных пользователей и финансовой информации, разграничение ролей и прав доступа, шифрование всего трафика (HTTPS/TLS), надёжные протоколы аутентификации и авторизации (JWT, OAuth 2.0, возможность интеграции с корпоративными LDAP/AD);
- gRPC интерфейсы и тестирование API: использование gRPC reflection для просмотра доступных методов и сервисов без необходимости ручного изучения документации, возможность автоматической генерации документации и клиентских SDK на основе gRPC (через protobuf и инструменты типа Buf, grpc-gateway);
- SDK и инструменты: готовые клиентские библиотеки (Go, Python, Java и др.), CLI утилиты и кодогенерация по спецификации для быстрого старта;
- Event driven интеграция: поддержка webhooks, или очередей сообщений (Kafka/RabbitMQ) для мгновенной доставки событий внешним системам;

- централизованный реестр Protobuf схем: все контракты описаны в *.proto файлах, объединены в едином репозитории с управлением версиями и механизмами валидации [23];
- удобство использования: интуитивно понятный и простой интерфейс для разных категорий пользователей;
- масштабируемость: возможность расширения функционала и объема обрабатываемых данных.

Бизнес-требования:

- снижение операционных расходов на управление жилым фондом;
- повышение прозрачности и оперативности взаимодействия с жильцами;
- соответствие законодательным нормам в сфере жилищно-коммунального хозяйства.

Требования к архитектуре:

- язык разработки: Go (Golang) [20];
- микросервисная структура: сервисы (управление заявками, управление пользователями, взаимодействие с уведомлениями);
- взаимодействие сервисов через REST и внутренние взаимодействие сервисов по средствам gRPC (внутреннее общение сервисов);
- хранилище данных: реляционная БД (PostgreSQL) для хранения заявок, пользователей, журналов действий;
- масштабируемость: возможность добавлять новые сервисы (аналитика, финансы).

Приложение, разработанное по представленным требованиям, будет обеспечивать эффективную работу организации, автоматизацию и оптимизацию рутинных задач, повышать скорость обработки заявок и улучшать коммуникацию с жильцами [16].

Выводы по главе 1

Для разработки приложения для автоматизации управления эксплуатацией жилого фонда были сформулированы требования, необходимые для его проектирования. Функционал приложения должен охватывать все процессы управления, включая полный цикл обработки заявок и управление техническим обслуживанием. Важным требованием является интеграция с внешними государственными системами. Интерфейс приложения должен быть понятен всем пользователям (жильцам, диспетчерам, исполнителям).

При проектировании приложения нужно учитывать современные архитектурные подходы, такие как микросервисная архитектура и использование реляционных БД. Важными требованиями являются безопасность данных и защита конфиденциальной информации.

Создание приложения по автоматизации управления эксплуатацией жилого фонда позволит повысить качество обслуживания жильцов, оптимизировать процессы управления, снизить расходы на управление жилым фондом.

Глава 2 Проектирование программного обеспечения приложения для автоматизации управления эксплуатацией жилого фонда

2.1 Выбор методологии проектирования программного обеспечения

Архитектурный подход и методология проектирования определяют основу создаваемой системы и во многом задают её способность к дальнейшему развитию и интеграции с внешними сервисами. Поэтому выбор архитектуры и процесса разработки был выполнен с учётом требований к гибкости, технологической независимости и эволюционности решения [6].

Среди архитектурных подходов рассматривались следующие варианты: трёхслойная архитектура, шаблон MVC, микросервисная архитектура и гексагональная архитектура (Ports & Adapters) [12].

Трёхслойная архитектура (пользовательский интерфейс, бизнес-логика, доступ к данным) проста для реализации и понятна, но обладает жёсткой линейной структурой и затрудняет интеграцию с внешними системами.

MVC (Model–View–Controller) удобен для построения пользовательских интерфейсов веб-приложений и обеспечивает разделение представления, данных и управляющей логики, но остаётся тесно привязан к выбранным технологиям и фреймворкам.

Микросервисная архитектура обеспечивает масштабируемость и независимость компонентов, но требует развитой инфраструктуры (контейнеризация, оркестрация, мониторинг), что повышает сложность администрирования.

Гексагональная архитектура (Ports & Adapters) устраняет избыточную зависимость бизнес-логики от интерфейсов и технологий. Центральное ядро содержит доменные модели и правила, а взаимодействие с внешним миром происходит через абстрактные порты; конкретные технологии инкапсулируются в адаптерах. Такой подход сохраняет неизменность ядра при замене UI, базы данных, каналов уведомлений и внешних интеграций [11].

При проектировании приложения для автоматизации управления эксплуатацией жилого фонда выбрана гексагональная архитектура, реализованная в виде трёх микросервисов. Каждый микросервис построен по принципу Ports & Adapters: бизнес-логика вынесена в ядро сервиса; работа с базой данных, уведомлениями, отчётностью и внешними городскими сервисами выполняется через выходные порты и соответствующие адаптеры; взаимодействие пользователей и внешних приложений осуществляется через входные порты (REST/gRPC API) [12].

Ключевые причины выбора:

- независимость бизнес-логики от конкретных технологий хранения и коммуникаций;
- возможность подключать новые интеграции (городские службы, платёжные шлюзы) без изменения ядра;
- повышаемая тестопригодность за счёт подмены адаптеров заглушками;
- модульность и эволюционность: система развивается за счёт добавления портов/адаптеров и новых микросервисов.

Архитектура приложения не является жёстким ограничением. При изменении требований и технологий вносятся точечные изменения в адаптеры и порты, в то время как доменная логика ядра остаётся стабильной.

С точки зрения процесса разработки были проанализированы каскадная модель (Waterfall), Agile-подход и фреймворк Scrum [3].

Каскадная модель предполагает последовательное прохождение стадий и подходит при фиксированных и неизменных требованиях, гибкость вносимых изменений низкая.

Agile ориентирован на инкрементальную разработку с регулярной обратной связью и возможностью корректировки требований в процессе работы.

Scrum как наиболее распространённый фреймворк Agile организует короткие итерации (спринты) длительностью 1–4 недели и завершается представлением работоспособного инкремента системы [3].

С учётом ограниченных сроков и необходимости быстро адаптироваться к возможным изменениям выбран инкрементальный подход на базе Agile/Scrum. Практически это отобразилось в коротких спринтах, внутри которых выполнялись проектирование, реализация и тестирование небольших функциональных блоков (приём и регистрация заявок, уведомления, интеграции). Такой формат позволил оперативно получать обратную связь, устранять имеющиеся замечания и развивать систему без накопления технического долга.

В проекте принята гексагональная архитектура, реализованная тремя микросервисами, и использован инкрементальный подход на базе Agile/Scrum с короткими спринтами. Такое сочетание обеспечивает технологическую независимость ядра, удобную интеграцию через порты и адаптеры, а также эволюционное развитие системы: при появлении новых требований и сервисов достаточно добавить или заменить соответствующие адаптеры, не затрагивая доменную логику.

2.2 Логическое моделирование программного обеспечения

Проектирование программного обеспечения для управления эксплуатацией жилого фонда требует определения ключевых процессов и участников, задействованных в функционировании системы [17].

К основным принципам, которые должны быть реализованы в системе, относятся:

- прозрачность и оперативность обработки заявок;
- удобство взаимодействия с управляющей организацией для жильцов при подаче и отслеживании обращений;

– снижение трудоёмкости работы сотрудников управляющей компании за счёт автоматизации регистрации и контроля исполнения заявок;
– возможность масштабирования и интеграции системы с дополнительными модулями (аналитикой, отчётностью, сервисом уведомлений).

Для формализации функциональных требований применяется диаграмма вариантов использования (Use Case Diagram), которая позволяет выделить основные сценарии взаимодействия пользователей с системой, установить роли участников и продемонстрировать последовательность процессов.

В системе определяются следующие субъекты:

– пользователь – формирует заявку на проведение работ по обслуживанию жилого фонда и может отслеживает её статус;
– диспетчер – принимает и регистрирует заявки, распределяет задачи между исполнителями и осуществляет контроль исполнения;
– технический специалист – получает назначенные задачи от диспетчера и фиксирует результаты их выполнения.

Основой для дальнейшей детализации архитектуры программного обеспечения и разработки модулей, обеспечивающих эффективное управление эксплуатацией жилого фонда, является диаграмма вариантов использования.

Основные процессы взаимодействия между субъектами системы: подача заявок жильцами, их регистрация, распределение задач между исполнителями, контроль выполнения заявок, фиксация результатов выполнения работ и возможность отслеживания статуса заявок жильцами – отражены на диаграмме вариантов использования, которая представлена на рисунке 5.

Рисунок 5 – Диаграмма вариантов использования

Для уточнения логики работы приложения необходимо описать варианты использования, которые отражают взаимодействие пользователей с системой. Каждый вариант содержит краткое описание процесса, предусловия выполнения, основной поток событий, возможные альтернативные сценарии и постусловия. В таблице 1 приведён вариант использования процесса обработки заявки, охватывающий полный цикл обращения.

Таблица 1 – Вариант использования процесса обработки заявки

Элемент	Содержание
Краткое описание	Процесс обработки заявки от жителя: от момента подачи обращения до закрытия заявки после выполнения работ и уведомления пользователя.
Субъекты	Пользователь, диспетчер, система, технический специалист.
Предусловия	Система доступна для работы, пользователи авторизованы.
Основной поток	Пользователь подаёт заявку. Диспетчер принимает и регистрирует её, система присваивает идентификатор и сохраняет данные. Далее диспетчер распределяет задачу исполнителю, система обновляет статус и при необходимости ставит заявку в очередь. Технический специалист выполняет работы и фиксирует результат. После этого система закрывает заявку и уведомляет жителя, который проверяет уведомление, отслеживает статус и может оценить качество работ.
Альтернативные потоки	При выявлении неполных или некорректных данных заявка переводится в статус «ожидает уточнения». Если нет свободных исполнителей, заявка помещается в очередь, а пользователь получает уведомление о задержке.
Постусловия	Заявка переведена в завершённый статус («выполнена» или «закрыта»), результат работ зафиксирован, и заявитель уведомлён о завершении.

Процесс обработки заявки начинается с подачи жителем обращения. Диспетчер выполняет его приём и регистрацию, система автоматически фиксирует заявку, а также обновляет её статус в процессе проведения работ. После распределения задачи на соответствующего исполнителя заявка может быть помещена в очередь или сразу передана специалисту. По завершении работ система фиксирует результат и уведомляет жителя о выполнении, который имеет возможность ознакомиться с информацией и оценить качество обслуживания.

Процесс обработки заявки разработан с учетом современных методов, по методологиям разработки ПО и учитывает требования, предъявляемые к разрабатываемому приложению.

На рисунке 6 представлена диаграмма процесса обработки заявки.

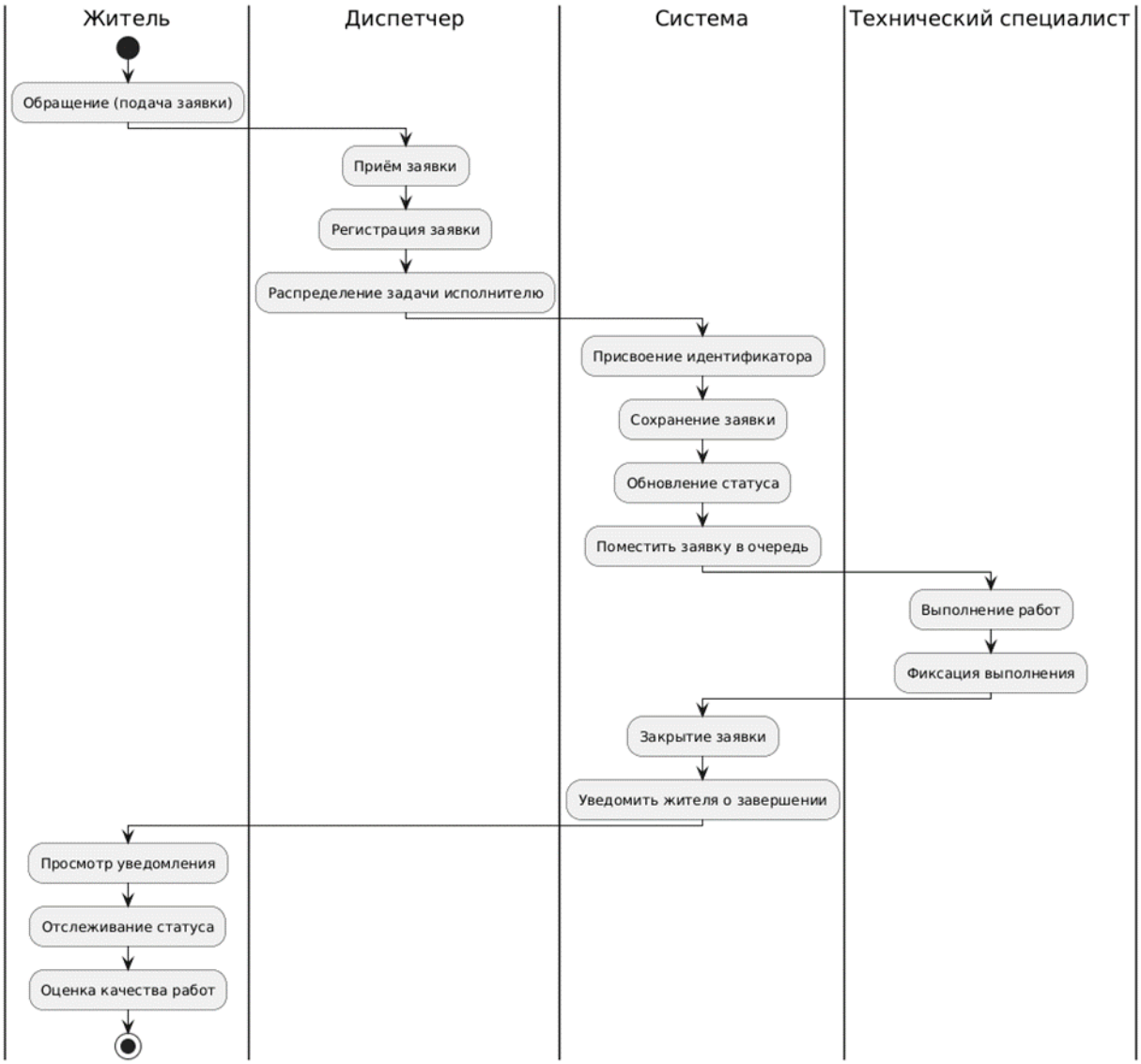

Рисунок 6 – Диаграмма активности процесса обработки заявки

Логическое проектирование программного обеспечения для автоматизации управления эксплуатацией жилого фонда направлено на выбор архитектурного подхода, структурирование компонентов и определение взаимодействия между ними [9]. Для проектируемой системы управления была выбрана гексагональная архитектура (Hexagonal Architecture, Ports & Adapters).

В отличие от многослойных моделей, гексагональный подход обеспечивает независимость бизнес-логики от конкретных технологий.

Центральное ядро реализует все доменные правила и сценарии работы, а взаимодействие с пользователями приложения и внешними системами (городские службы, банки, государственные информационные системы) осуществляется через порты и реализующие их адаптеры.

Основные компоненты архитектуры разрабатываемого программного обеспечения:

- Core (ядро системы) – включает бизнес-логику и доменные объекты («Заявка», «Пользователь», «Диспетчер», «Технический специалист»), здесь находятся сервисы управления заявками и пользователями;
- входные порты (Input Ports) – интерфейсы, через которые система принимает команды от внешних пользователей и приложений (для системы это REST/gRPC API, веб-портал для жителей и интерфейс рабочего места диспетчера);
- выходные порты (Output Ports) – интерфейсы, через которые ядро обращается к инфраструктуре: Repository Port – работа с хранилищем данных (PostgreSQL), Notification Port – взаимодействие с сервисами уведомлений (email), External Systems Port – интеграция с внешними системами (городские службы, платёжные шлюзы, информационные системы);
- адаптеры (Adapters) – конкретные реализации портов: PostgreSQL Adapter, Notification Adapter, External Systems Adapter.

Такое разделение обеспечивает модульность и гибкость разрабатываемой системы. Так, при замене оповещения по Email на sms-уведомления требуется изменить только адаптер, ядро остаётся неизменным.

На рисунке 7 представлена общая схема гексагональной архитектуры системы управления эксплуатацией жилого фонда.

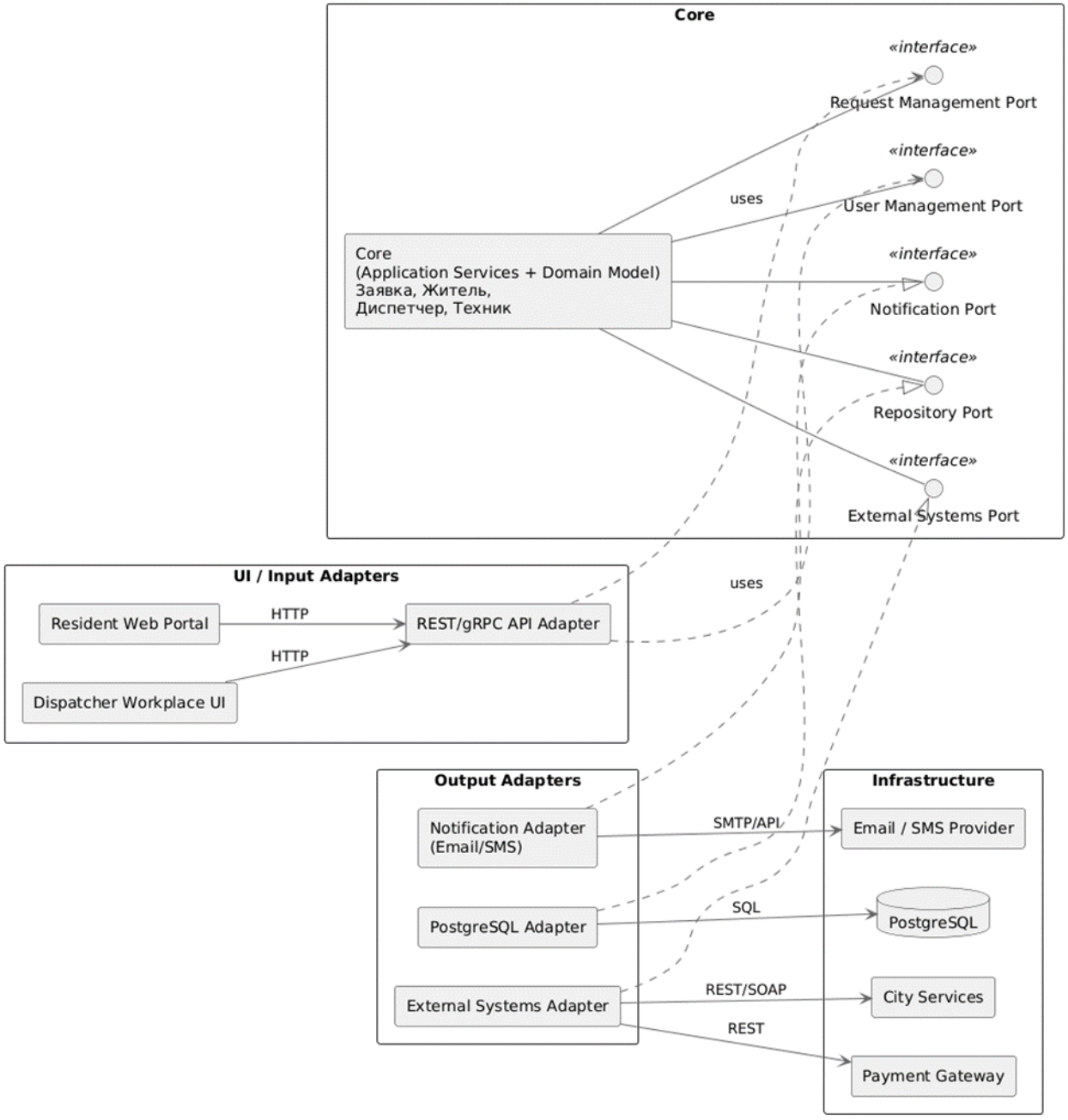

Рисунок 7 – Схема гексагональной архитектуры системы управления эксплуатацией жилого фонда

Для демонстрации работы архитектуры разрабатываемого приложения для автоматизации управления эксплуатацией жилого фонда представлен рисунок 8, на котором показан пример последовательной обработки заявки жителя.

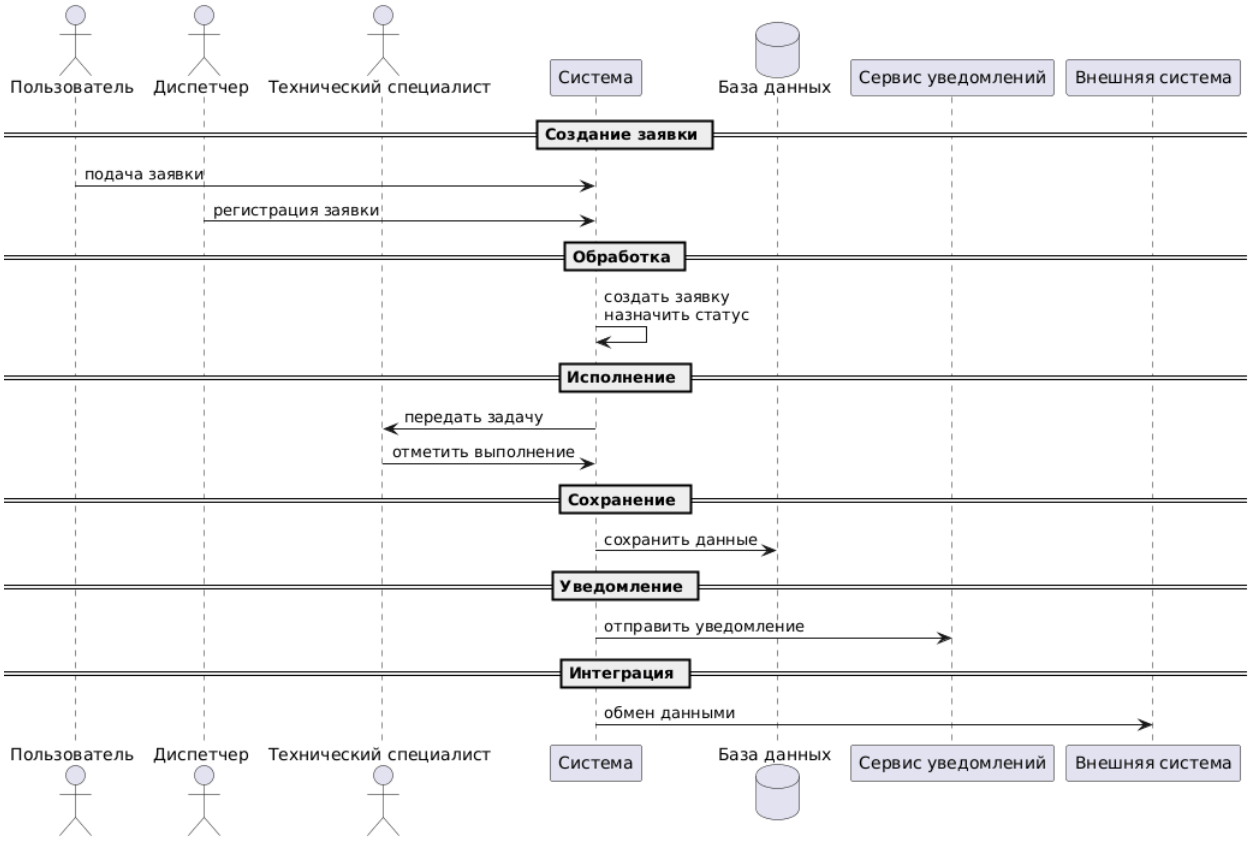

Рисунок 8 – Последовательность обработки заявки

Процесс взаимодействия включает следующие шаги:

- пользователь создаёт заявку через веб-портал, либо диспетчер регистрирует её через своё рабочее место;
- запрос поступает в систему через input Port (REST API);
- Core обрабатывает заявку: создаёт объект «Заявка», присваивает статус, определяет исполнителя;
- для сохранения данных Core обращается к Repository Port, который реализован через PostgreSQL Adapter;
- при изменении статуса система вызывает Notification Port, а адаптер уведомляет жителя по выбранному каналу связи (SMS или e-mail);
- в случае необходимости интеграции Core использует External Systems Port, который реализуется через соответствующий адаптер.

Таким образом, гексагональная архитектура обеспечивает изоляцию бизнес-логики от конкретных технологий, гибкость интеграции с внешними сервисами и простоту дальнейшего развития системы.

2.3 Моделирование данных системы

Цель моделирования данных – обеспечить надёжное хранение и быстрый доступ к сведениям об аварийных заявках, пользователях, уведомлениях и истории изменений, реализованной на Go и взаимодействующей через REST/gRPC поверх PostgreSQL [7].

Логическая модель ориентирована на транзакционную нагрузку ядра (Emergency, User, Notification-сервисы) и поддерживает асинхронную интеграцию с модулем уведомлений, а также последующее подключение аналитики и отчётности. Такой подход согласован с принятой микросервисной архитектурой и гексагональными принципами отдельных сервисов.

Выбор реляционной БД PostgreSQL обоснован тем, что она обеспечивает строгую ссылочную целостность, транзакционность (ACID) и богатые средства индексации, что критично для SLA по времени регистрации/назначения заявок и прослеживаемости операций. Данные нормализованы до 3НФ: каждый атрибут зависит от ключа своей таблицы, избыточность сведена к минимуму, неизбежные дубли носят осознанный характер для ускорения типовых выборок и внешних интеграций. Концепция «одно центральное хранилище правды» для заявок и истории изменений соответствует целевой TO-BE-модели, где все процессы работают поверх единой БД [4].

Состав сущностей и их назначение:

а) перечисления (доменные словари) – задают допустимые значения и упрощают проверку инвариантов бизнес-процессов в коде и в БД.:

1) emergency_status: NEW → IN_PROGRESS → RESOLVED / CANCELLED (и служебный DELETED) – непрерывный

жизненный цикл заявки, удобная валидация на уровне БД и прикладной логики;

2) notification_status: PENDING → SENT / FAILED → READ – поддерживает конвейер доставки и метрики качества оповещений;
3) user_role: RESIDENT, EMPLOYEE, MANAGER, ADMIN – опора для разграничения прав (RBAC) и фильтрации данных в сервисах;

б) таблица users – хранит учётные записи и контактные данные (e-mail, телефон, ФИО), а также роль:
   1) уникальность email исключает дубли, индексы по email и role ускоряют аутентификацию и выборки по ролям;
   2) поля адреса и квартиры описывают место проживания пользователя, но не подменяют адрес инцидента, который фиксируется в addresses как «снимок» на момент события;

в) таблица emergencies – описывает заявку: категория, приоритет, статус, автор (created_by) и текущий исполнитель (assigned_to);

г) таблица addresses – 1:1 с emergencies (PK=FK emergency_id), отдельная сущность по трём причинам:
   1) снимок контекста инцидента (улица, дом, подъезд, этаж, квартира) независимо от последующих изменений профиля пользователя;
   2) расширяемость (геокодирование, координаты, привязка к участку/домоуправлению);
   3) безопасность: позволяет регулировать доступ к PII-полям в портах/адаптерах;

д) таблица emergency_history – журналирует каждое значимое действие над заявкой (тип операции, «было/стало», автор изменения и пояснение); индексы по emergency_id и changed_at обеспечивают мгновенную отрисовку «ленты событий» в карточке; хранение

old_value/new_value как text допускает сериализованные фрагменты (например, JSON), сохраняя гибкость без усложнения схемы (обеспечивает управленческую прозрачность и контроль качества);

е) таблица notifications – очередь исходящих оповещений (тип (канал/шаблон), получатель (user_id, email), тело/метаданные, статусы доставки, счётчик попыток и тайм-метки (last_attempt_at, sent_at, read_at)); индексы по status и created_at позволяют Notification-сервису подбирать партии PENDING и вести ретраи FAILED; по user_id – строить «историю уведомлений» в профиле; по email – ускорять внешние интеграции (SMTP-адаптер/провайдеры).

Спроектированная логическая модель данных представлена на рисунке 9.

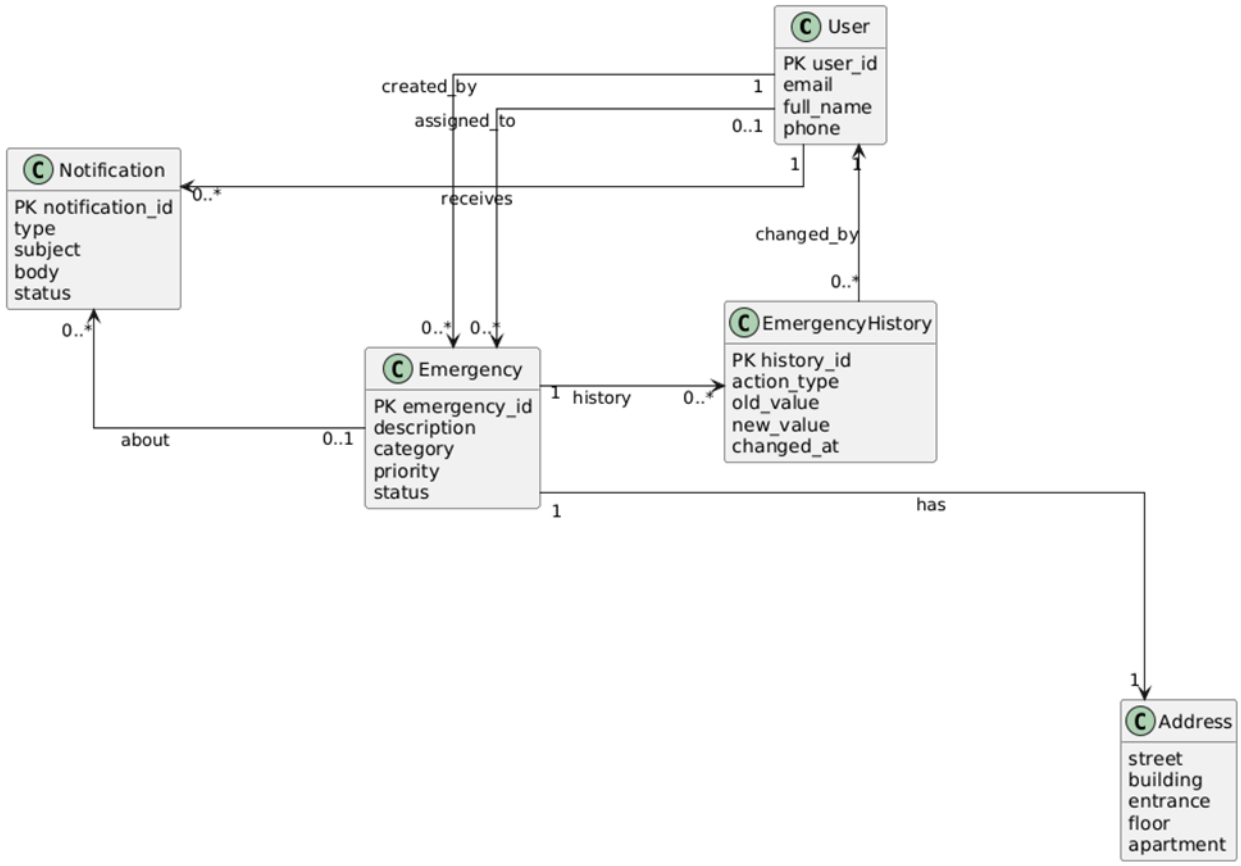

Рисунок 9 – Спроектированная логическая модель данных

Спроектированная физическая модель данных представлена на рисунке 10.

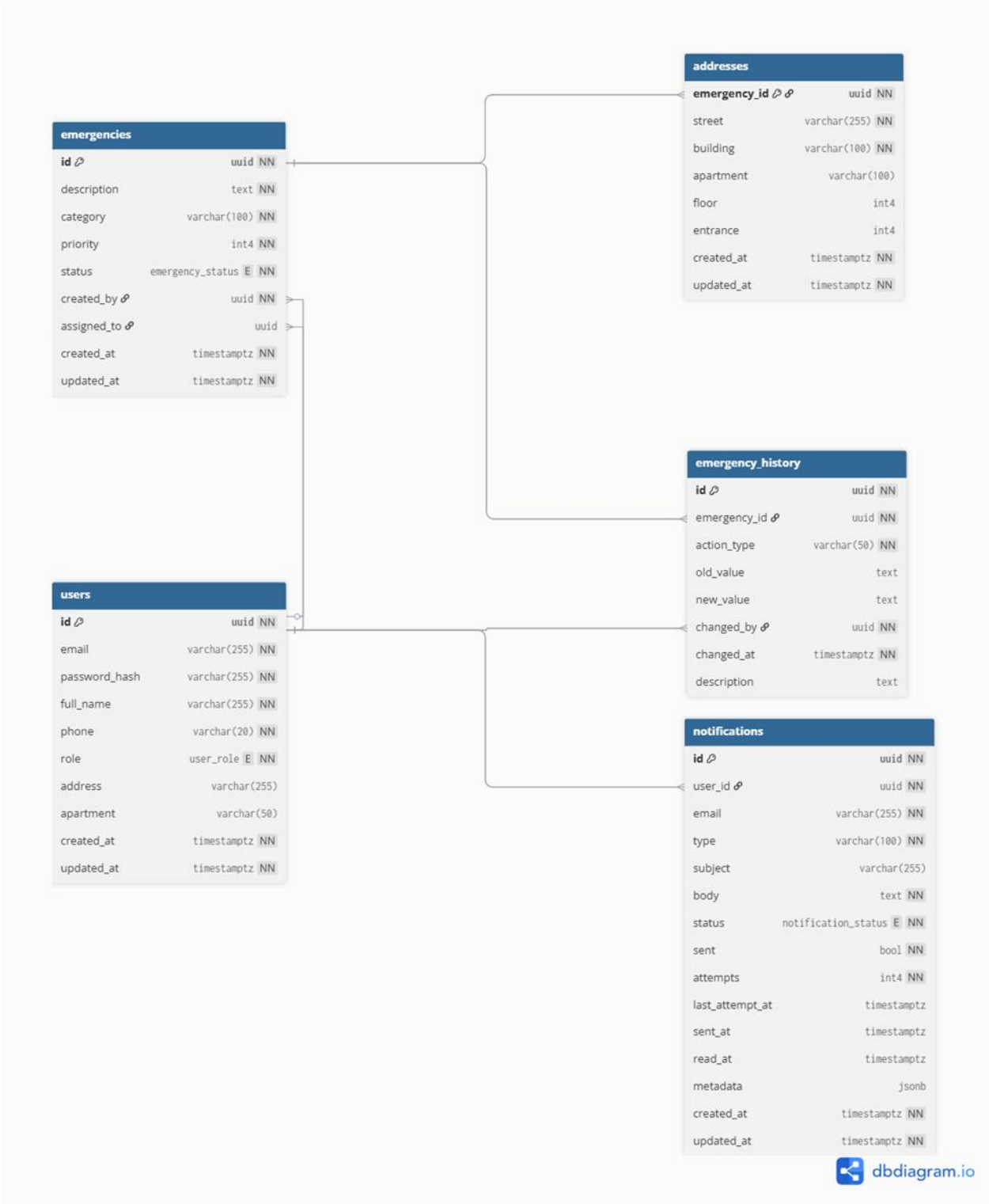

Рисунок 10 – Спроектированная физическая модель данных

Связи и их назначение:

а) Users (1) → Emergencies (N):

1) created_by (обязательная FK): кто создал заявку (пользоватль/оператор);
2) assigned_to (необязательная FK): кто назначен исполнителем (сотрудник/бригада).

Почему так: чётко разделяет автора и ответственного; позволяет историям эскалаций и переназначений не терять первоисточник.

б) Emergencies (1) → Addresses (1):

1) PK=FK addresses.emergency_id, ON DELETE CASCADE.

Почему так: адрес – атрибут инцидента, а не пользователя; каскад защищает от «висящих» адресов при физическом удалении тестовых/черновых заявок.

в) Emergencies (1) → EmergencyHistory (N):

1) FK emergency_id, ON DELETE CASCADE.

Почему так: история принадлежит конкретной заявке; при редком физическом удалении – журнал уходит за ней, а в рабочем процессе используется статус DELETED (мягкое удаление).

г) Users (1) → Notifications (N):

1) FK user_id.

Почему так: уведомления адресуются конкретным субъектам (жителям и сотрудникам); позволяет строить личные и управленческие ленты оповещений.

Кардинальности один-ко-многим и один-к-одному выбраны для прозрачности, соответствия предметной области и минимизации аномалий вставки/обновления. Такое построение облегчает транзакционную модель Core-слоёв и асинхронную обработку событий воркерами уведомлений.

Правила целостности и жизненные циклы:

– статусы заявок: NEW (создана) → IN_PROGRESS (в работе) → RESOLVED (устранена) или CANCELLED (отменена); DELETED

предназначен как «мягкое удаление» – запись остаётся для аналитики и аудита; физическое удаление применяется лишь к ошибочно созданным/тестовым данным (тогда каскады защищают целостность);

- история изменений: каждое изменение (статуса, исполнителя, приоритета) фиксируется в emergency_history с указанием автора (changed_by) и времени; это обеспечивает доказуемость операций и ретроспективный анализ KPI;
- статусы уведомлений: конвейер PENDING → SENT/FAILED; при открытии письма/сообщения – READ; пара status + attempts + временные метки позволяют реализовать ретраи и дедé-дубликацию на уровне воркеров.

Безопасность и доступ: RBAC на основе user_role ограничивает видимость и операции (жители видят только свои заявки/уведомления; сотрудники – назначенные им; менеджеры/администраторы – сквозной просмотр и действия).

Расширяемость: выделение адреса в отдельную сущность и полная журнализация изменений упрощают подключение аналитики (BI) и внешних интеграций (городские системы, платёжные шлюзы, геосервисы) без миграции существующих таблиц; Event-driven-интеграции Notification-сервиса и API-шлюза остаются неизменными при наращивании домена (например, добавление «бригады», «материалы работ», «SLA таймеры»), что согласуется с общей архитектурной стратегией проекта [1].

2.4 Проектирование пользовательского интерфейса

Пользовательский интерфейс системы проектировался с учётом специфики работы управляющей компании и потребностей всех категорий пользователей. Интерфейс разрабатывался от ролей и ключевых сценариев. Для жителя – быстрая авторизация, минимальная форма создания обращения

(категория, описание, приоритет, адрес), наглядный список заявок со статусами и историей. Для оператора – единый реестр с поиском и фильтрами и карточка обращения в формате master–detail со сменой статуса, назначением исполнителя и просмотром журнала действий. Для администратора – управление пользователями и ролями.

Для реализации выбрана архитектура одностраничного приложения с серверным рендерингом. Такой подход обеспечивает плавную навигацию без перезагрузки страниц, быструю первоначальную загрузку и SEO-оптимизацию публичных разделов.

Информационная архитектура включает публичную главную страницу (краткая информация и вход), личный кабинет жителя (дашборд, создание заявки) и административную зону (реестр, карточка, пользователи). Навигация ролевая: недоступные разделы скрыты в меню и заблокированы при прямых переходах; предусмотрены «хлебные крошки» и стабильные точки возврата.

Адаптивность заложена на уровне макетов: на широких экранах списки представлены таблицами, на узких – карточками; формы перестраиваются в один столбец; второстепенные панели открываются по запросу (модальные окна). Пустые и ошибочные состояния содержат пояснения и быстрые действия (сброс фильтров, «создать заявку», повтор запроса), чтобы не оставлять пользователя без следующего шага.

В интерфейсе учтены требования доступности: используется семантическая разметка страниц (header, nav, main, footer) и базовые ARIA-атрибуты; у форм – корректные связи меток и полей (label/for, aria-labelledby, aria-describedby). Поддерживается работа со скринридерами и клавиатурой (логичный порядок табуляции, видимый фокус, закрытие модальных окон по Esc и возврат фокуса). Уведомления объявляются через aria-live, индикаторы загрузки помечаются aria-busy. Контраст ориентирован на WCAG 2.1 AA; таблицы снабжены caption и корректными заголовками (scope); локаль задаётся через lang, даты и числа форматируются по региональным правилам.

Выводы по главе 2

В процессе проектирования приложения для автоматизации управления эксплуатацией жилого был разработан пошаговый план, который обеспечит эффективное функционирование программного обеспечения.

При проектировании приложения для автоматизации управления эксплуатацией жилого фонда выбрана гексагональная архитектура, реализованная тремя микросервисами, и использован инкрементальный подход на базе Agile/Scrum с короткими спринтами, что обеспечит технологическую независимость ядра, удобную интеграцию через порты и адаптеры, а также эволюционное развитие системы. В отличие от многослойных моделей, гексагональный подход обеспечивает независимость бизнес-логики от конкретных технологий.

Также была спроектирована логическая модель данных, ориентированная на транзакционную нагрузку ядра и поддерживающая асинхронную интеграцию с модулем уведомлений и подключение аналитики и отчётности. Такой подход согласован с принятой микросервисной архитектурой и гексагональными принципами отдельных сервисов.

Такой подход соответствует заявленным к приложению требованиям.

Глава 3 Реализация и тестирование программного обеспечения

3.1 Средства разработки

Разработка программного обеспечения для автоматизированной системы управления эксплуатацией жилого фонда базируется на современных технологиях, обеспечивающих масштабируемость, отказоустойчивость и простоту интеграции.

Серверная часть.

В качестве языка программирования выбран Go (Golang), отличающийся высокой производительностью, простотой конкурентного программирования через goroutines и быстрым циклом компиляции и деплоя. Микросервисная архитектура с применением принципов гексагональной модели обеспечивает модульность: каждый сервис развивается и масштабируется независимо [13].

Взаимодействие сервисов.

Внутренние вызовы реализованы через gRPC с использованием Protocol Buffers [23]. Это даёт строгую типизацию, автоматическую генерацию клиентских библиотек и высокую скорость передачи данных. Для внешних интеграций применяется REST API через HTTP/HTTPS, что облегчает подключение мобильных и веб-клиентов, а также сторонних систем.

Хранение данных.

Используется PostgreSQL как основная реляционная СУБД. Она обеспечивает поддержку сложных транзакций, индексов и аналитических запросов, а также стабильную работу при увеличении нагрузки [22].

Клиентская часть.

Фронтенд реализован на базе Next.js и React с TypeScript, что обеспечивает надёжную типизацию и поддержку современного серверного рендеринга. Tailwind CSS и Radix UI применяются для быстрой стилизации и создания доступных компонентов интерфейса [8].

Инфраструктура.

Для локальной разработки применяются Docker и Docker Compose, что гарантирует единообразие среды. В промышленной эксплуатации планируется Kubernetes для автоматического масштабирования и управления контейнерами [19].

Нефункциональные цели.

На текущем этапе проект рассчитан на обработку до 300-500 заявок в сутки. Архитектура предусматривает горизонтальное масштабирование сервисов. Целевые показатели качества: задержка p95 для критичных операций ≤ 500 мс, доступность системы не ниже 99,5%.

3.2 Реализация модели данных и миграции базы данных

Хранилище реализовано в PostgreSQL раздельными миграциями для трёх доменов: пользователи, заявки и уведомления. Каждая миграция оформлена как SQL-скрипт, создающий необходимые типы, таблицы, индексы и триггеры [4].

В части пользователей миграция объявляет перечислимый тип user_role с ролями RESIDENT, EMPLOYEE, MANAGER, ADMIN и создаёт таблицу users с полями email (уникальный), password_hash, ФИО, телефоном, ролью и временными метками. Для ускорения выборок заведены индексы на email и role. Обновление updated_at автоматизировано триггером, вызывающим функцию update_updated_at_column() перед любым UPDATE по таблице users (сама функция задаёт NEW.updated_at = NOW()).

В домене заявок миграция вводит тип emergency_status и таблицу emergencies с описанием, категорией, приоритетом, статусом, авторами/исполнителями и метками времени; отдельная таблица addresses связывается с заявкой 1:1 и хранит детали адреса (улица, дом, квартира, этаж, подъезд). Для повышения производительности создаются индексы по статусу, категории, автору, исполнителю и дате создания. Поле updated_at

поддерживается тем же триггером на уровне таблиц emergencies и addresses. Для аудита создаётся таблица emergency_history с индексами по emergency_id, changed_at и action_type. Заполнение истории обеспечивают два триггера: log_emergency_created (после INSERT) и log_emergency_changes (после UPDATE), которые пишут события CREATED, STATUS_CHANGED, ASSIGNED, UNASSIGNED с сохранением старых/новых значений и инициатора. Для «мягко удалённых» записей предусмотрена серверная процедура cleanup_deleted_emergencies(), удаляющая заявки со статусом DELETED, старше 90 дней по updated_at. В конце миграции статусам добавлено значение DELETED через ALTER TYPE ... ADD VALUE, что закрепляет семантику soft-delete без физического удаления основной строки.

Подсистема уведомлений создаётся отдельной миграцией: тип notification_status (PENDING, SENT, FAILED, READ) и таблица notifications с полями получателя (user_id, email), типом уведомления, темой, телом, флагом успешной отправки, счётчиком попыток, отметками времени отправки/прочтения и полем metadata (JSONB) для параметров канала/шаблона. Индексы на user_id, status, created_at, email позволяют быстро выбирать сообщения для доставки, ретраев и аналитики; updated_at обновляется тем же триггером.

Прикладные детали, учтённые в реализации миграций:

- скрипты написаны идемпотентно там, где возможно (CREATE TABLE IF NOT EXISTS, CREATE INDEX IF NOT EXISTS), что упрощает начальную и повторную инициализацию окружений разработчиков и стендов;
- триггерная функция update_updated_at_column() определена в каждом доменном скрипте; если схемы сервисов разделены на уровне БД, имена не конфликтуют; при единой схеме рекомендуется квалифицировать функцию по схеме или переименовать, чтобы исключить переопределение;

– для генерации UUID по умолчанию в истории заявок используется gen_random_uuid(); для корректной работы на пустой базе необходимо на этапе bootstrap включить расширение pgcrypto (CREATE EXTENSION IF NOT EXISTS pgcrypto;) – это требование зафиксировано в первой миграции окружения;
– добавление значения DELETED в emergency_status выполнено отдельной командой ALTER TYPE ... ADD VALUE;
– порядок применения миграций выстроен так, чтобы обеспечить успешные внешние ключи и функциональность аудита: сначала пользователи, затем заявки и адреса, затем история заявок и сервисные функции, после этого – уведомления на уровне CI/CD миграции каждого сервиса выполняются до выката его контейнера; проверка успешности встраивается в пайплайн.

В результате реализованная модель данных отражает три изолированных домена и обеспечивает готовые для эксплуатации механизмы: индексацию под основные запросы интерфейса, автоматическое обновление временных меток, аудит жизненного цикла заявок, отложенную очистку soft-deleted записей и универсальное хранение параметров канала доставки уведомлений.

Всё это собрано в миграциях, которые можно безопасно и воспроизводимо применять на всех средах – от локальной разработки до промышленной эксплуатации.

3.3 Разработка пользовательского интерфейса

Пользовательский интерфейс системы разработан с учётом потребностей всех категорий пользователей: жителей, сотрудников диспетчерской службы и администраторов. Основная цель интерфейса – предоставить удобный и интуитивно понятный инструмент для работы с заявками и управления процессами обслуживания.

Пользовательский интерфейс системы автоматизации управления эксплуатацией жилого фонда представлен на рисунке 11.

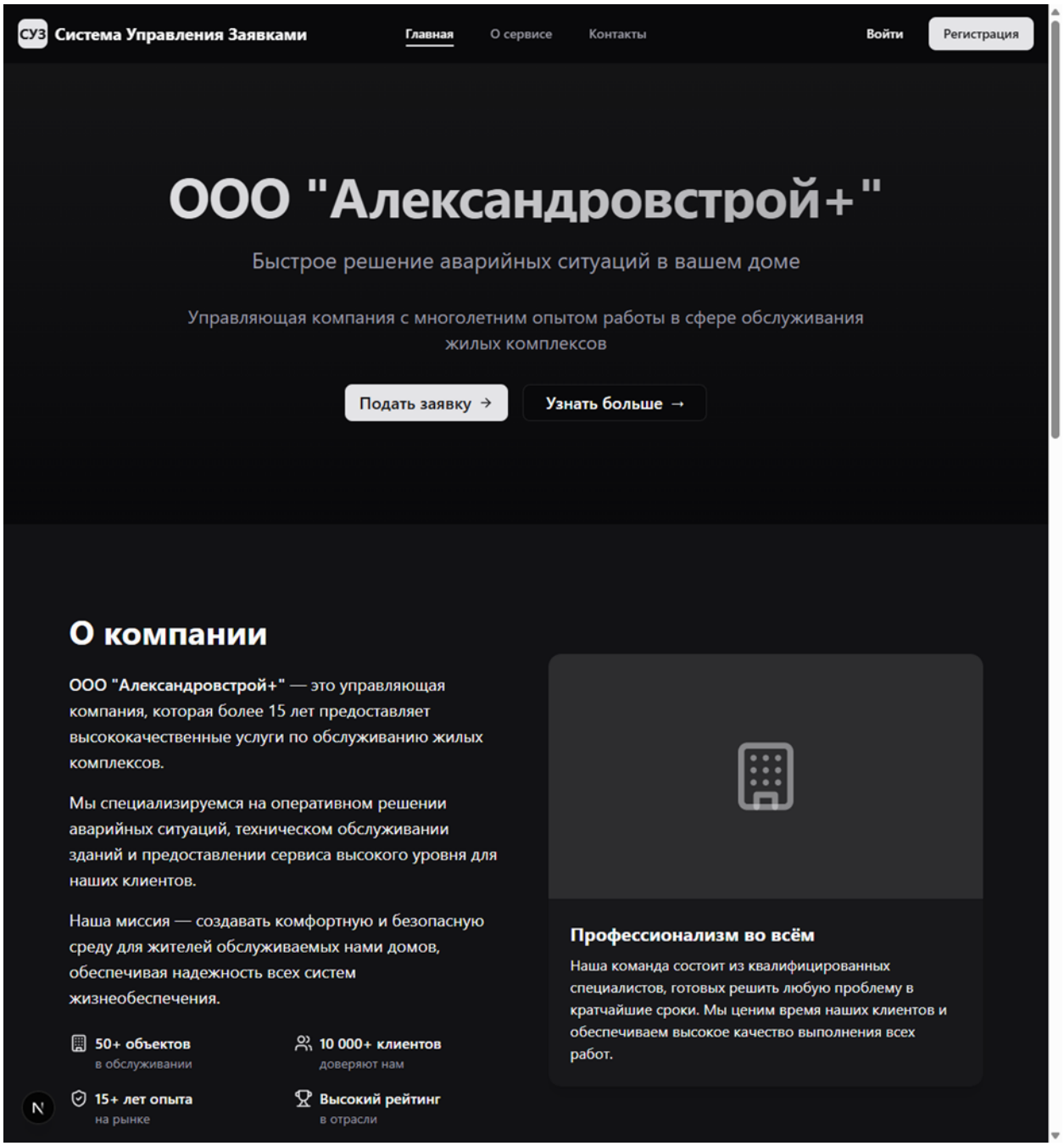

Рисунок 11 – Главная страница приложения

Архитектура интерфейса построена на современном технологическом стеке. В качестве основы используется React с TypeScript, что позволяет

контролировать типы данных на этапе разработки и снижает количество ошибок при сопровождении. Для организации маршрутизации и серверного рендеринга применяется Next.js с Turbopack, обеспечивающий высокую скорость загрузки страниц и плавное взаимодействие пользователя с приложением. Стилизация выполнена с использованием Tailwind CSS, а библиотека Radix UI обеспечивает доступные и стандартизированные элементы управления. Компоненты реализованы по модульному принципу: каждый отвечает за конкретный блок функциональности, что упрощает повторное использование и тестирование [8].

Структура интерфейса разделена по ролям пользователей. Главная страница служит точкой входа и содержит общую информацию о компании, а также формы авторизации и регистрации.

Для жителей предусмотрен личный кабинет в виде дашборда, где отображается список поданных заявок и их статусов, а также доступна форма создания новых обращений. Страница личного кабинета для жителей представлена на рисунке 12.

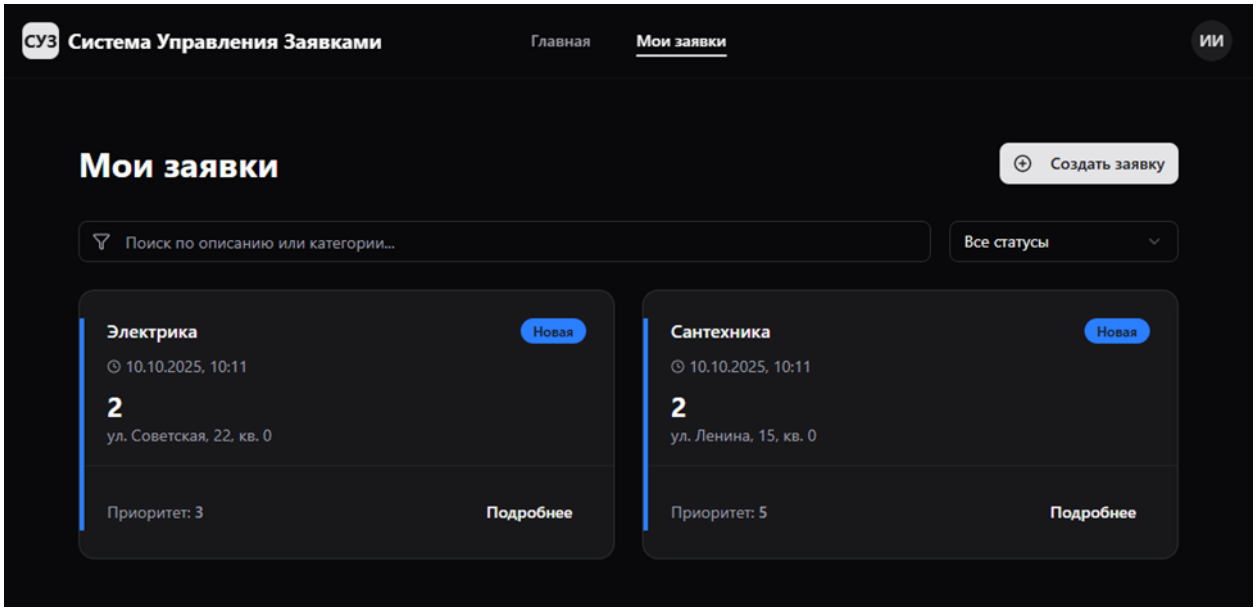

Рисунок 12 – Страница личного кабинета жителя

Также жители могут отследить статус своей заявки на странице ее детализации, которая отображает информацию о времени работы над заявкой, этапы ее обработки и конечный результат. Пример страницы можно увидеть на рисунке 13.

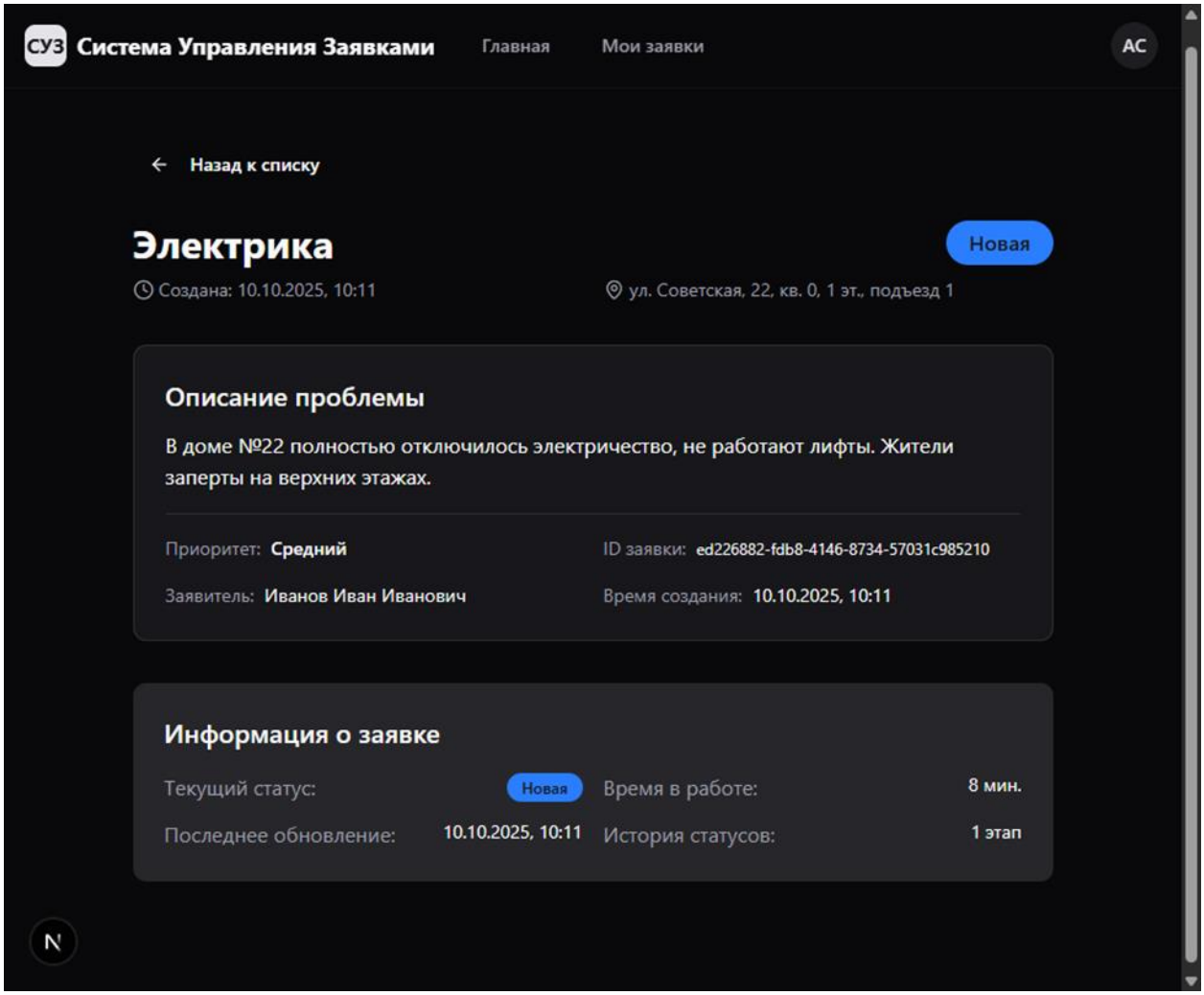

Рисунок 13 – Страница детализации заявки

Форма страницы для создания новой заявки содержит только необходимые поля – категорию, описание, приоритет и адрес – что упрощает процесс взаимодействия с системой. Интерфейс страницы для создания заявки представлен на рисунке 14.

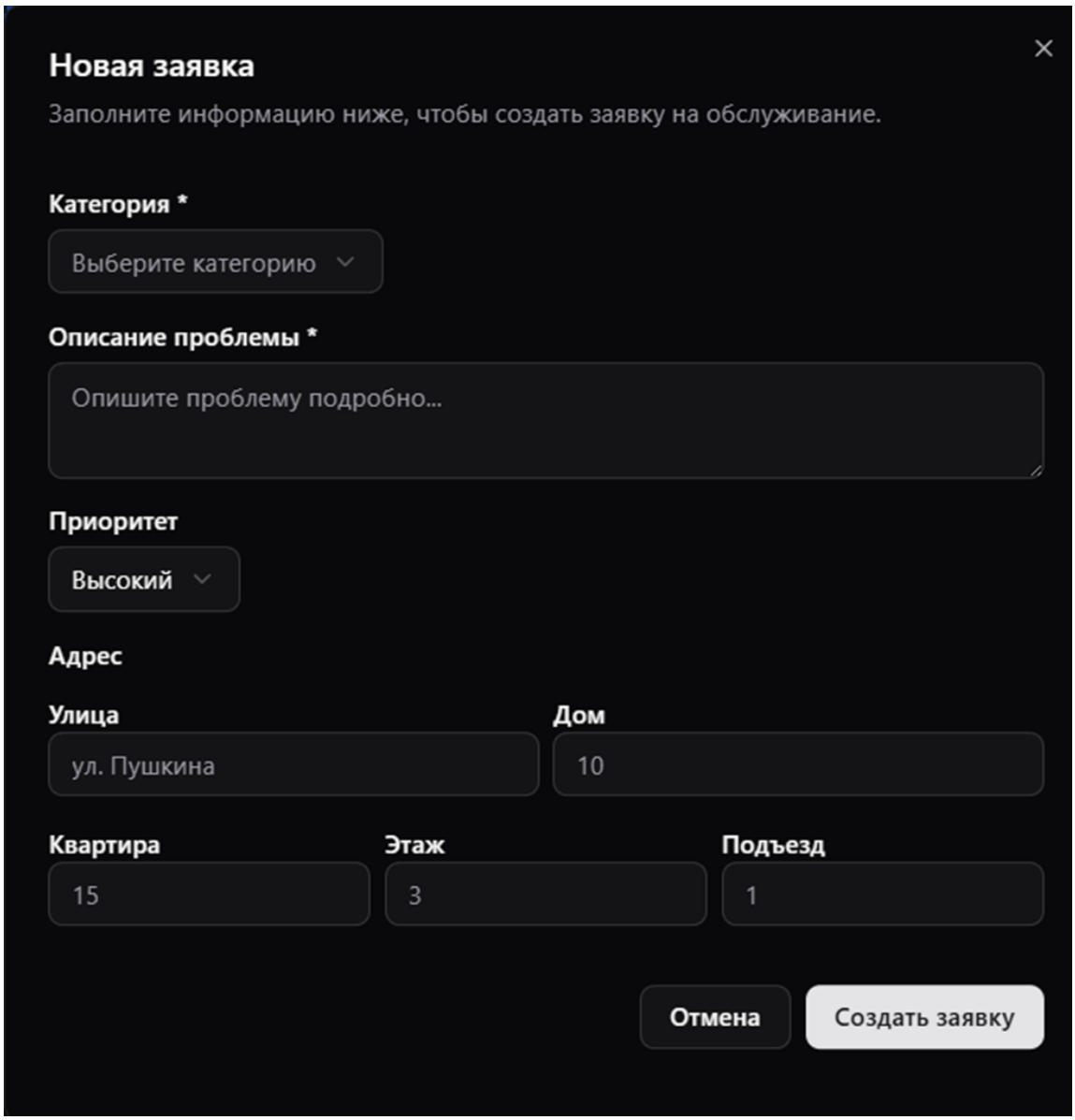

Рисунок 14 – Окно создания заявки

Сотрудники управляющей компании работают в административной панели. Она включает полный список заявок с возможностью фильтрации и поиска, а также предоставляет доступ к детальной карточке обращения, где можно изменять статус, назначать исполнителей и просматривать историю изменений. Для администраторов предусмотрен дополнительный функционал управления пользователями и их ролями. Пример административной панели приложения можно увидеть на рисунке 15.

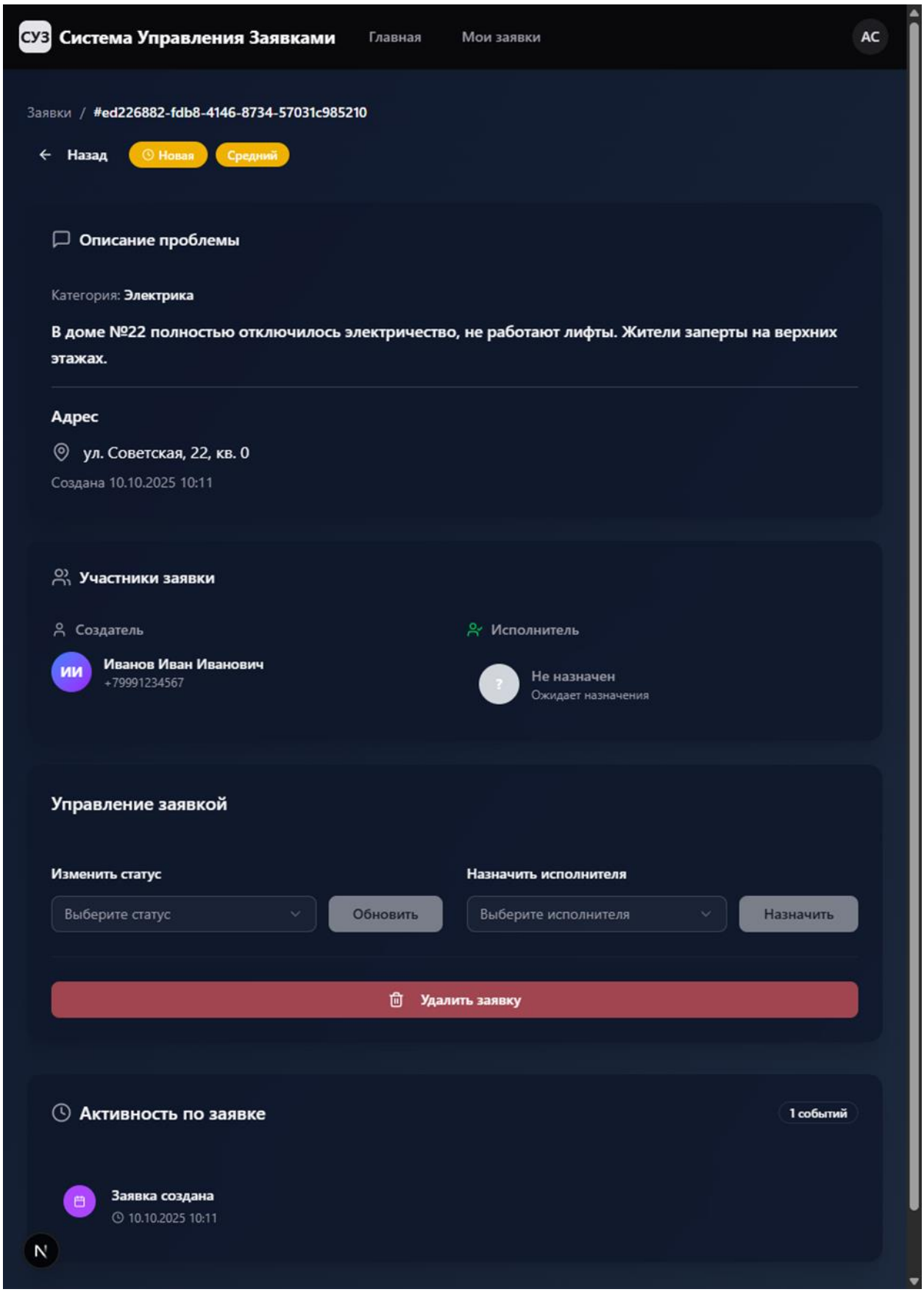

Рисунок 15 – Административная панель

Интерфейс реализован в адаптивном дизайне и корректно работает на различных устройствах. За счёт использования CSS Grid и Flexbox страницы автоматически перестраиваются под размер экрана: таблицы переходят в карточный формат, формы выстраиваются в вертикальные блоки. Это обеспечивает удобство работы как на настольных компьютерах, так и на мобильных устройствах. Базовые требования доступности учтены с помощью aria-атрибутов и встроенной поддержки Radix UI, что позволяет корректно использовать систему со скринридерами.

Для обеспечения интерактивности внедрена система уведомлений, реализованная через библиотеку Sonner. Пользователь получает моментальную обратную связь о результатах действий: успешном создании заявки, изменении её статуса или возникновении ошибки. Пример уведомления показан на рисунке 16.

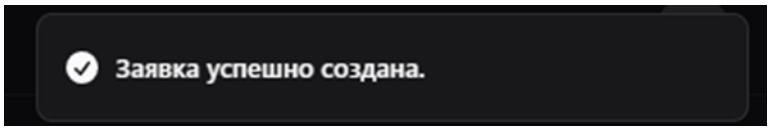

Рисунок 16 – Система уведомлений

В процессе загрузки данных отображаются скелетоны и индикаторы, что делает работу с системой предсказуемой и удобной. Ошибки сопровождаются понятными сообщениями. Пример ошибки можно увидеть на рисунке 17.

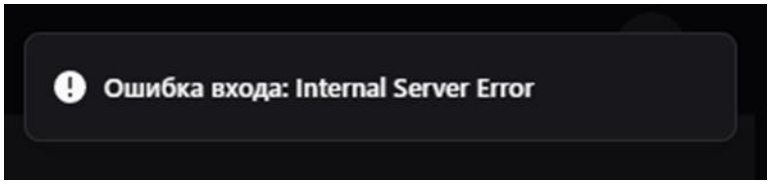

Рисунок 17 – Ошибка на клиенте

Управление состоянием разделено на два уровня. Локальное состояние, включая авторизацию и права доступа, хранится в Zustand. Серверное состояние обрабатывается с помощью TanStack Query, который обеспечивает кэширование, автоматическое обновление данных и оптимизацию количества запросов к серверу, что позволяет поддерживать актуальность интерфейса.

Модульная архитектура приложения и строгая типизация кода позволяют расширять интерфейс. Возможность добавления новых категорий заявок, расширение системы уведомлений, а также интеграция каких-либо внешних сервисов не требуют вносить серьезные изменения в существующий код.

Такая структура повышает устойчивость программного обеспечения к изменениям требований управляющей компании, а также снижает стоимость его развития.

В результате разработки приложения создан простой и удобный интерфейс, который интуитивно понятен жителям и в то же время помогает обеспечивать эффективность работы сотрудников и административный контроль процессов, осуществляемый руководством организации. Приложение надежно и эффективно функционирует в реальных условиях использования благодаря выбранным техническим решениям, которые обеспечивают высокую производительность, масштабируемость и доступность продукта.

3.4 Маршрутизация веб-страниц приложения

Веб-клиент использует файловую маршрутизацию Next.js App Router. Структура каталогов в `src/app` напрямую задаёт адресное пространство: каждый сегмент папки соответствует сегменту URL, а компонент `page.tsx` определяет содержимое страницы. Такой подход упрощает чтение кода, поддерживает автогенерацию маршрутов и обеспечивает единообразие навигации между разделами системы [25].

Маршруты сгруппированы по ролям и сценариям использования. Корневой сегмент `/` служит точкой входа и предоставляет доступ к авторизации и краткой информации о системе. Группа `(auth)` инкапсулирует экраны входа и регистрации: в рамках сегментов `/login` и `/register` выполняется валидация форм и первичная обработка ответов от API-шлюза. Пользовательские сценарии собраны в группе `(user)`: личный кабинет доступен по адресу `/dashboard`, где отображается список заявок текущего пользователя и реализована форма создания новой заявки. Детальный просмотр обращения реализован динамическим сегментом `/dashboard/[id]`: идентификатор извлекается из URL, после чего серверный компонент инициирует загрузку данных и историю изменений, не блокируя остальной интерфейс.

Производственные операции сотрудников вынесены в административные сегменты. Раздел `/requests` предоставляет реестр всех заявок с расширенными фильтрами по статусу их исполнения, приоритету, дате и исполнителю; детальная карточка открывается по адресу `/requests/[id]` и позволяет изменять статус, назначать исполнителя и просматривать аудит определенной заявки.

Управление пользователями организовано в пространстве `/users` с динамическими маршрутами для просмотра профилей: структура адресов унифицирована, что упрощает формирование «глубоких ссылок» из внешних источников и навигацию внутри приложения.

Дополнительные административные функции сгруппированы в `/admin`, включая обзорные панели и операции управления ролями.

Механизм динамических сегментов App Router используется повсеместно и сочетается с параметрами запроса: выборка списков и фильтров синхронизируется с URL, благодаря чему состояние представления воспроизводимо при обновлении страницы и передаче ссылки. Для этого серверные компоненты `page.tsx` и клиентские контейнеры читают `searchParams`, формируют запросы к API и обновляют кэш TanStack Query.

Переходы между страницами выполняются без полной перезагрузки, а встроенное предзаполнение ссылок (prefetch) уменьшает задержки при частых навигационных действиях операторов.

Единые макеты и навигационные элементы собираются в `layout.tsx` на уровне соответствующих сегментов. Это позволяет переиспользовать общие компоненты (шапка, боковое меню) и одновременно изолировать стили и контекст для пользовательских и административных зон.

Для повышения отзывчивости применяются стандартные файлы `loading.tsx` и `error.tsx`: первый показывает индикаторы загрузки при смене маршрута и серверных фетчах, второй – перехватывает ошибки и выводит понятные сообщения без «обрушения» всего приложения.

В местах, где допустимы отсутствующие ресурсы (например, удалённая заявка), используется `not-found.tsx` с корректным возвратом статуса 404 и ссылками на безопасные точки возврата. Пример отображения оповещения представлен на рисунке 18.

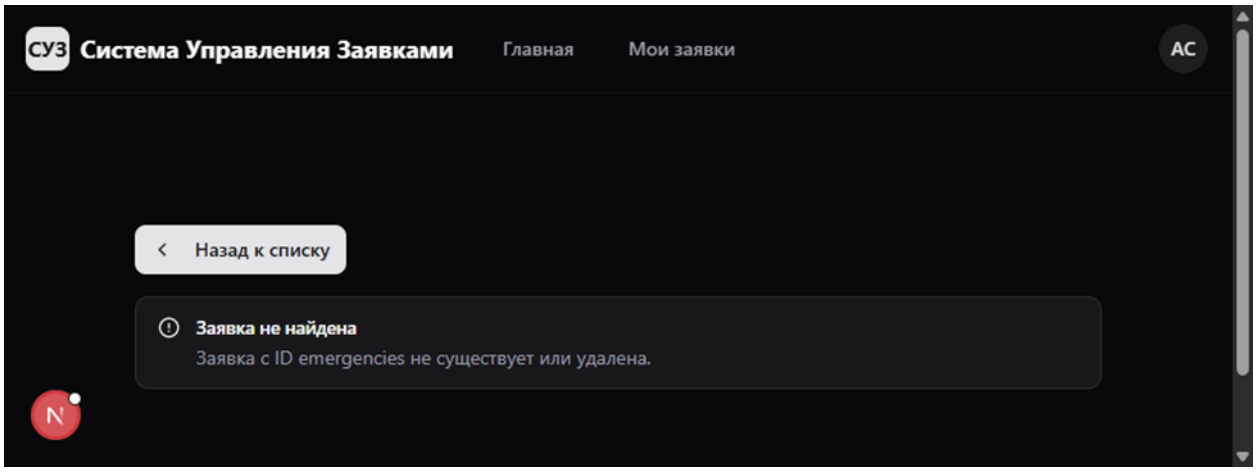

Рисунок 18 – Отображение сообщения «Заявка не найдена»

Контроль доступа реализован многослойно. На серверной стороне используется `middleware.ts`, который читает токен из cookie, извлекает роль и перенаправляет неавторизованных посетителей на страницу входа, а также

запрещает доступ к административным сегментам пользователям без достаточных прав. На клиентской стороне состояние авторизации синхронизируется через Zustand; компоненты интерфейса скрывают недоступные действия и маршруты, чтобы исключить «слепые» переходы, и в случае несоответствия роли пользователь получает информативное уведомление. Благодаря такому разделению проверок достигается баланс между безопасностью (ранний отбор на уровне маршрутизации) и удобством (мягкая деградация на клиенте).

Обмен данными с бэкендом выстроен единообразно: маршруты App Router инициируют запросы к REST-эндпоинтам API-шлюза, который транслирует их в gRPC-вызовы соответствующих микросервисов. Детальные страницы используют серверные компоненты для первичного получения данных, что сокращает время первого ответа сервера (TTFB) и индексируемость, а клиентские контейнеры берут на себя обновления и интерактивность.

Такой паттерн позволяет совмещать преимущества серверного рендеринга и одностраничной навигации, обеспечивая быстрые переходы и предсказуемое поведение интерфейса.

В итоге маршрутизация организует понятную карту приложения, поддерживает глубокие ссылки, синхронизирует фильтры со строкой адреса и обеспечивает строгий контроль доступа на уровне URL. Использование групп, динамических сегментов, макетов и промежуточного ПО Next.js делает навигацию устойчивой к ошибкам, а работу операторов – быстрой и повторяемой в ежедневных сценариях.

3.5 Интеграции и gRPC-контракты

Внутреннее взаимодействие компонентов системы организовано по схеме «микросервисы + gRPC». Контракты описаны в файлах Protocol Buffers и служат единственным источником истины для межсервисных интерфейсов.

На их основе генерируется серверный код и клиентские обёртки для сервисов, реализованных на Go, что исключает расхождения в сигнатурах методов и форматах сообщений. Веб-клиент на Next.js к gRPC не обращается: он работает с REST-интерфейсом через API-шлюз, который принимает HTTP-запросы, проверяет подлинность и полномочия пользователя и транслирует их во внутренние вызовы gRPC [15].

Состав домена разделён на четыре автономных компонента. API-шлюз является единой точкой входа, отвечает за проверку JWT-токенов, маршрутизацию запросов и приведение кодов ошибок к понятным клиенту статусам HTTP. Сервис заявок управляет жизненным циклом обращений: создаёт записи, предоставляет выборки с фильтрами, изменяет статусы и назначает исполнителей. Сервис пользователей ведёт учётные записи, профили и роли, а также выполняет аутентификацию. Сервис уведомлений формирует и отправляет сообщения, фиксирует состояние доставки и предоставляет историю отправок.

Каждый микросервис использует собственную схему в PostgreSQL; обмен данными между ними происходит только через публичные gRPC-методы, прямого доступа к чужим таблицам нет, что поддерживает изоляцию и низкую связанность [15].

Структуры сообщений в контрактах типизированы явно: идентификаторы передаются в формате UUID, временные значения – типом google.protobuf.Timestamp, статусы и роли оформлены перечислениями. Такой формат обеспечивает однозначную интерпретацию данных и стабильность при эволюции API.

Расширение сообщений выполняется без нарушения совместимости: новые поля добавляются как необязательные, существующие номера полей не переиспользуются, а изменения в перечислениях выполняются путём добавления новых значений.

Версии контрактов фиксируются в структуре каталогов и в комментариях к методам, что позволяет параллельно поддерживать несколько поколений интерфейсов при поэтапных обновлениях.

Маршрутизация во внешнем контуре построена через API-шлюз: например, обращение по адресу создания заявки преобразуется в вызов соответствующего метода сервиса заявок, а запрос профиля пользователя – в метод сервиса пользователей. Ответы и ошибки нормализуются на границе: коды gRPC приводятся к стандартным статусам HTTP с читаемыми сообщениями. Это упрощает работу клиентского приложения и сторонних интеграций, сохраняя при этом высокую производительность и строгую типизацию во внутреннем контуре.

Бизнес-процессы реализованы как согласованные последовательности межсервисных вызовов. При создании заявки система проверяет существование инициатора через сервис пользователей, сохраняет обращение и инициирует отправку уведомления заинтересованным сторонам через сервис уведомлений. При смене статуса или назначении исполнителя формируются соответствующие сообщения. В ответах сервисов возвращаются как основные данные, так и производные атрибуты (время создания, текущий статус, идентификаторы участников), что позволяет клиентской части корректно обновлять интерфейс без дополнительных запросов.

Надёжность взаимодействий обеспечивается ограничением времени ожидания на каждом вызове, повторными попытками для безопасных операций чтения, а также использованием уникальных идентификаторов запросов для предотвращения повторного создания сущностей при сетевых сбоях. Массовые выборки поддерживают постраничную навигацию на основе маркера, что снижает нагрузку на базу данных и делает результаты стабильными при параллельных изменениях. Авторизация работает в два уровня: шлюз проверяет токен и роль, а доменные методы дополнительно сверяют права на конкретное действие (например, изменение статуса доступно исполнителю, менеджеру или администратору).

Для наблюдаемости каждый запрос помечается корреляционным идентификатором, который прокидывается через метаданные gRPC; по нему в журналах и метриках прослеживается полный путь обработки. Для отладки и интеграционных испытаний включена рефлексия gRPC, что позволяет анализировать доступные методы и типы сообщений инструментом grpcurl без локального хранения файлов схем [15].

В результате интеграционный контур опирается на чётко определённые контракты и прозрачный шлюз внешних вызовов. Такое решение поддерживает высокую производительность и масштабируемость, упрощает эволюцию интерфейсов и делает поведение системы предсказуемым как для внутренних сервисов, так и для клиентских приложений.

3.6 Архитектура программного обеспечения

Архитектура системы построена как набор взаимодействующих микросервисов, каждый из которых отвечает за свой предметный участок: управление заявками, управление пользователями и уведомления.

Внешнее взаимодействие клиента с сервером организовано через единый шлюз, который принимает HTTP-запросы и преобразует их во внутренние вызовы по gRPC [12].

Такой слой отделяет клиентскую часть от деталей реализации сервисов и обеспечивает единые правила аутентификации, авторизации, журналирования и обработки ошибок.

Клиентский слой реализован в виде веб-приложения на React с использованием TypeScript и рамок Next.js. Это позволяет совмещать серверную и клиентскую отрисовку страниц, ускорять первый показ интерфейса и стабильно взаимодействовать с API.

Управление локальным состоянием и правами доступа выполняется на стороне клиента, а работа с данными сервера сопровождается кэшированием и автоматическим обновлением, что уменьшает число сетевых обращений и

делает интерфейс отзывчивым. Стили оформлены единообразно, интерфейс адаптирован под мобильные и настольные устройства [24].

Шлюз является точкой входа для всех клиентских запросов. Он проверяет токен пользователя, определяет его роль и права, сопоставляет адрес внешнего запроса с конкретным методом внутреннего сервиса, а затем возвращает ответ в унифицированном формате. Коды ошибок и диагностические сообщения приводятся к понятным для клиента статусам, при этом сохраняются технические подробности в журналах для последующего анализа. Через тот же слой проходят единые правила валидации входных данных и ограничения на объёмы запросов.

Серверная часть разделена на независимые процессы. Сервис заявок управляет жизненным циклом обращений: создаёт записи, меняет статусы, назначает и снимает исполнителей, ведёт историю изменений. Сервис пользователей отвечает за учётные записи, роли и проверку подлинности. Сервис уведомлений формирует сообщения, отправляет их получателям и фиксирует состояние доставки.

Каждый сервис использует собственную схему в единой базе PostgreSQL, что сохраняет изоляцию данных и упрощает обслуживание. Схемы развиваются через систему миграций: изменения структуры применяются автоматически при выкатывании новой версии, предусмотрены сценарии отката и правила совместимости, исключающие остановку работы в ходе обновления.

Внутреннее взаимодействие сервисов осуществляется по gRPC с использованием строго описанных контрактов. Сообщения содержат однозначно типизированные поля, идентификаторы передаются в формате UUID, временные значения – в стандартном представлении с часовыми поясами. При расширении функциональности новые поля вводятся так, чтобы не нарушать совместимость с уже развернутыми версиями. Это позволяет обновлять отдельные части системы поэтапно [21].

Слой данных построен на PostgreSQL, где активные таблицы снабжены индексами под типовые выборки. История изменений по заявкам сохраняется отдельно, что делает возможным разбор инцидентов и ретроспективный анализ без влияния на производительные операции. Для обслуживания «мягко удалённых» записей используются фоновые процедуры очистки. Подключения к базе выполняются через пул, параметры которого подбираются для равномерной нагрузки.

Контейнеризация всех компонентов обеспечивает одинаковую среду на машинах разработчиков, на тестовых стендах и в эксплуатационной среде. Сервисы и их зависимости запускаются в отдельных контейнерах, для локальной разработки применяется компоновка через Go-compose, в производственной среде предусмотрено развертывание в оркестраторе с автоматическим перезапуском, проверками готовности и здоровья процессов, а также пошаговым обновлением без простоя. Каждый контейнер публикует служебные точки проверки, что позволяет поддерживать корректный порядок запуска и быстро определять неисправные экземпляры [5].

Безопасность реализована на нескольких уровнях. Подлинность пользователей подтверждается по токенам, срок действия и правила обновления заданы на уровне шлюза. Доступ к операциям проверяется как в шлюзе, так и в самих доменных методах: даже при наличии действительного токена сервис сверяет право на конкретное действие с учётом роли и принадлежности сущности. Пароли хранятся только в виде стойких хешей, секретные ключи и параметры подключения передаются сервисам через переменные окружения и не сохраняются в кодовой базе. Межсервисный трафик изолирован на уровне внутренней сети контейнеров; при необходимости включается шифрование соединений.

Масштабируемость достигается за счёт независимого увеличения числа экземпляров тех сервисов, где наблюдается рост нагрузки. Из-за малых накладных расходов gRPC выдерживает высокую частоту внутренних вызовов, а кэширование данных на стороне клиента снижает объём повторных

запросов. В местах с повышенными требованиями к времени ответа используются серверные компоненты для первичной загрузки данных, что уменьшает задержки при первом открытии страницы.

Наблюдаемость покрывает ключевые аспекты работы. Каждый запрос помечается идентификатором, который прослеживается через все слои и сервисы. Журналы содержат отметки времени, сведения о пользователе, методе и коде ответа, что упрощает поиск проблемных участков. Собираются метрики времени ответа, частоты ошибок и загрузки ресурсов; на их основе можно настраивать оповещения и автоматические действия. Для ручных проверок включена служебная рефлексия интерфейсов gRPC, что ускоряет интеграционные испытания.

В итоге архитектура обеспечивает требуемые свойства системы: модульность и возможность независимого развития частей, устойчивость к сбоям, контролируемые обновления, предсказуемую производительность и безопасность обработки данных. Такое построение позволяет безболезненно наращивать функциональность – добавлять новые типы заявок, расширять каналы уведомлений, подключать внешние учётные системы – не затрагивая уже работающие элементы.

3.7 Тестирование программного обеспечения

Тестирование проводилось для подтверждения корректности бизнес-процессов и оценки устойчивости системы при ограниченных вычислительных ресурсах.

В качестве методологической основы использовалась «пирамида тестирования» с акцентом на интеграционные и нагрузочные проверки, модульные тесты в данной версии не применялись.

«Пирамида тестирования» представлена на рисунке 19.

Рисунок 19 – Пирамида тестирования

Испытания проводились в контейнерной среде с намеренно жёсткими лимитами. Для части микросервисов был установлен предел примерно 256 МГц доли одного ядра процессора и не более 256 МБ оперативной памяти на контейнер.

В таком режиме работали три доменных сервиса – пользователей, заявок и уведомлений; API-шлюз и PostgreSQL функционировали в стандартной конфигурации.

Это позволило оценить поведение системы при дефиците ресурсов и зафиксировать нижнюю границу производительности.

Интеграционные проверки выполнялись утилитой grpcurl с включённой рефлексией интерфейсов.

Последовательно воспроизводились сквозные сценарии. Воспроизведение сценария «Создание нового пользователя» представлено в Приложении А, сценария «Аутентификация и получение токена» – в

Приложении Б, сценария «Регистрация заявки» – в Приложении В, сценария «Чтение заявки по идентификатору» – в Приложении Г, сценария «Выборка обращений со статусом NEW» – в Приложении Д, сценария «Перевод в статус IN_PROGRESS» – в Приложении Е, сценария «Назначение исполнителя» – в Приложении Ж, сценария «Обновление профиля» – в Приложении И, сценария «Удаление пользователя» – в Приложении К.

Отдельно проверялась работоспособность рефлексии на примере сервисов заявок и пользователей. Данные проверки показаны на рисунке 20.

```
root@v317326:~/diplom# grpcurl -plaintext localhost:50052 list
grpc.health.v1.Health
grpc.reflection.v1.ServerReflection
grpc.reflection.v1alpha.ServerReflection
user.v1.UserService
root@v317326:~/diplom# grpcurl -plaintext localhost:50052 list user.v1.UserService
user.v1.UserService.Authenticate
user.v1.UserService.CreateUser
user.v1.UserService.DeleteUser
user.v1.UserService.GetUser
user.v1.UserService.ListUsers
user.v1.UserService.UpdateUser
root@v317326:~/diplom# grpcurl -plaintext localhost:50051 list emergency.v1.EmergencyService
emergency.v1.EmergencyService.AssignEmergency
emergency.v1.EmergencyService.CreateEmergency
emergency.v1.EmergencyService.GetEmergency
emergency.v1.EmergencyService.ListEmergencies
emergency.v1.EmergencyService.UpdateEmergencyStatus
root@v317326:~/diplom# grpcurl -plaintext localhost:50052 describe user.v1.CreateUserRequest
user.v1.CreateUserRequest is a message:
message CreateUserRequest {
  string email = 1;
  string password = 2;
  string full_name = 3;
  string phone = 4;
  .user.v1.Role role = 5;
  string address = 6;
  optional string apartment = 7;
}
root@v317326:~/diplom#  grpcurl -plaintext localhost:50051 describe emergency.v1.Address
emergency.v1.Address is a message:
message Address {
  string street = 1;
  string building = 2;
  optional string apartment = 3;
  optional int32 floor = 4;
  optional int32 entrance = 5;
}
root@v317326:~/diplom#
```

Рисунок 20 – Проверка работоспособности рефлексии

Все сценарии завершались успешно: данные валидировались, история изменений формировалась корректно, уведомления инициировались согласно логике.

Нагрузочное тестирование проводились утилитой ghz. Для всех методов тестирования использовались одинаковые условия нагрузки: 50 параллельных соединений и 1000 запросов.

В CreateUser зафиксирована пропускная способность около 45 запросов в секунду при среднем времени отклика 1,09 с; медиана составила 1,08 с, девяносто пятый перцентиль – 1,49 с, девяносто девятый – 1,63 с. Данные проверки отражены на рисунке 21.

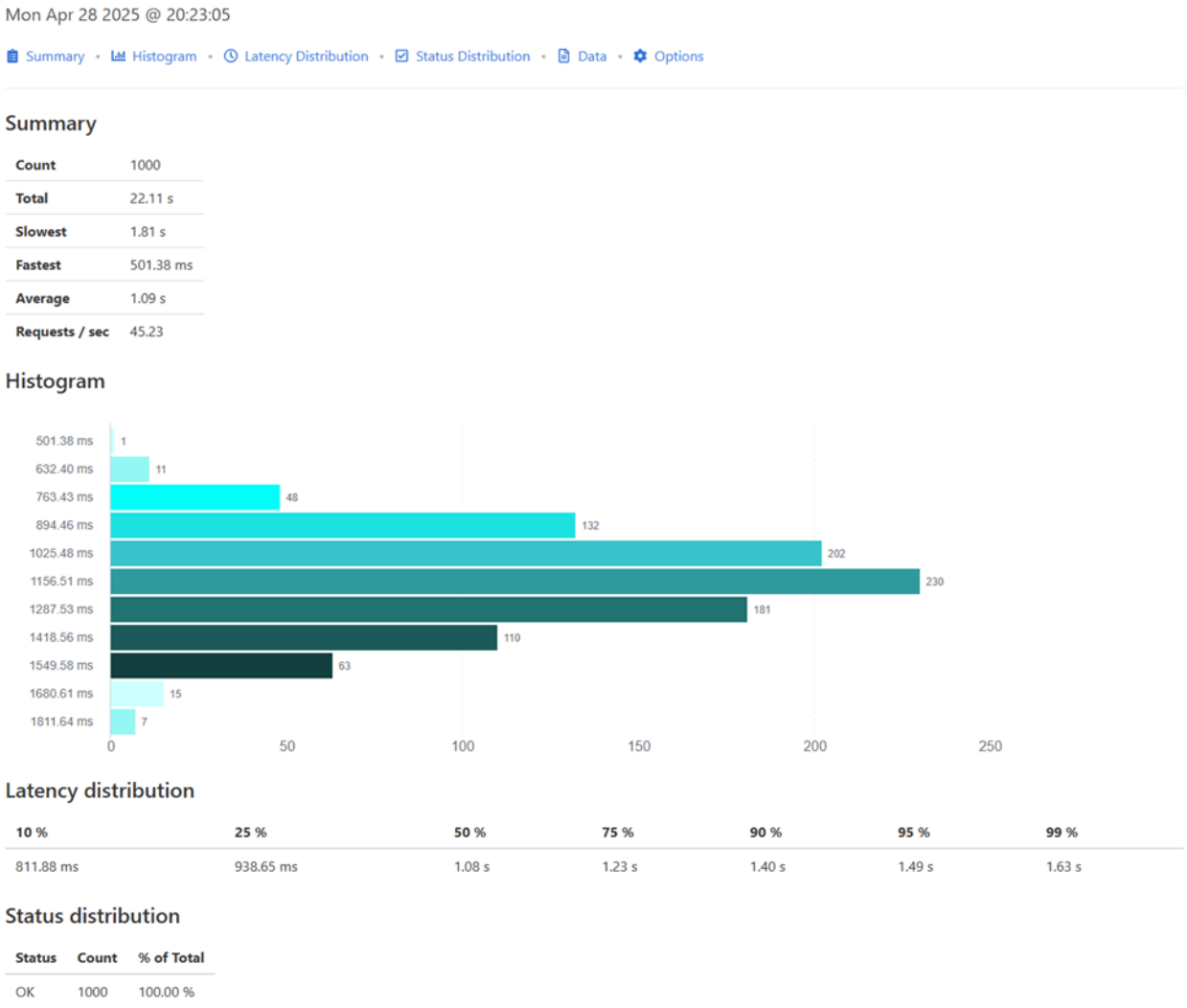

Рисунок 21 – График нагрузочного тестирования CreateUser

Метод CreateEmergency сервиса заявок показал среднюю задержку 252 мс при пропускной способности порядка 194 запросов в секунду; девяносто пятый перцентиль – 409 мс, ошибок не наблюдалось.

Данные можно увидеть на рисунке 22.

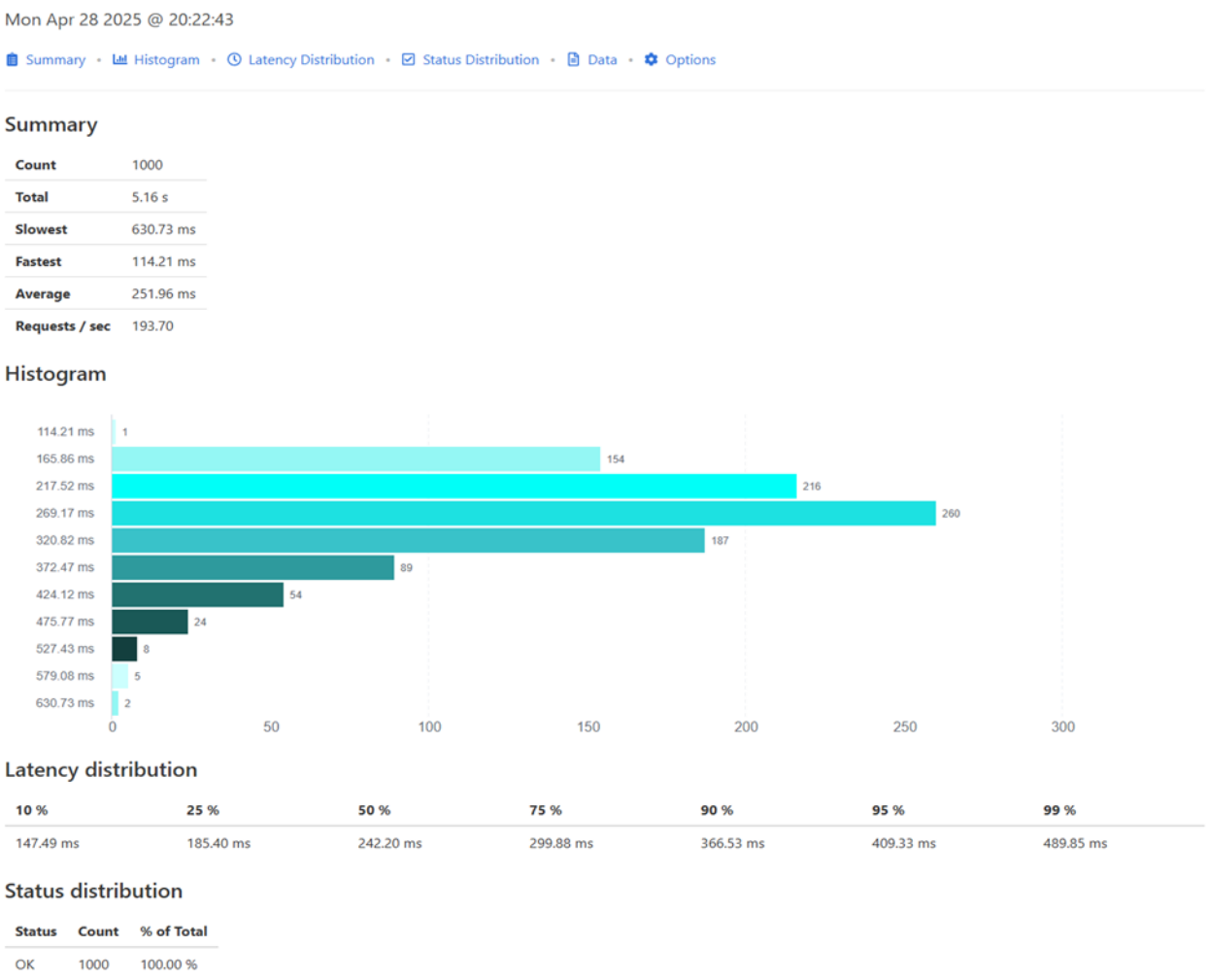

Рисунок 22 – График нагрузочного тестирования CreateEmergency

Метод SendEmail сервиса уведомлений обеспечил около 264 запросов в секунду при средней задержке 185 мс; девяносто пятый перцентиль – 317 мс, все ответы успешны.

Данные тестирования представлены на рисунке 23.

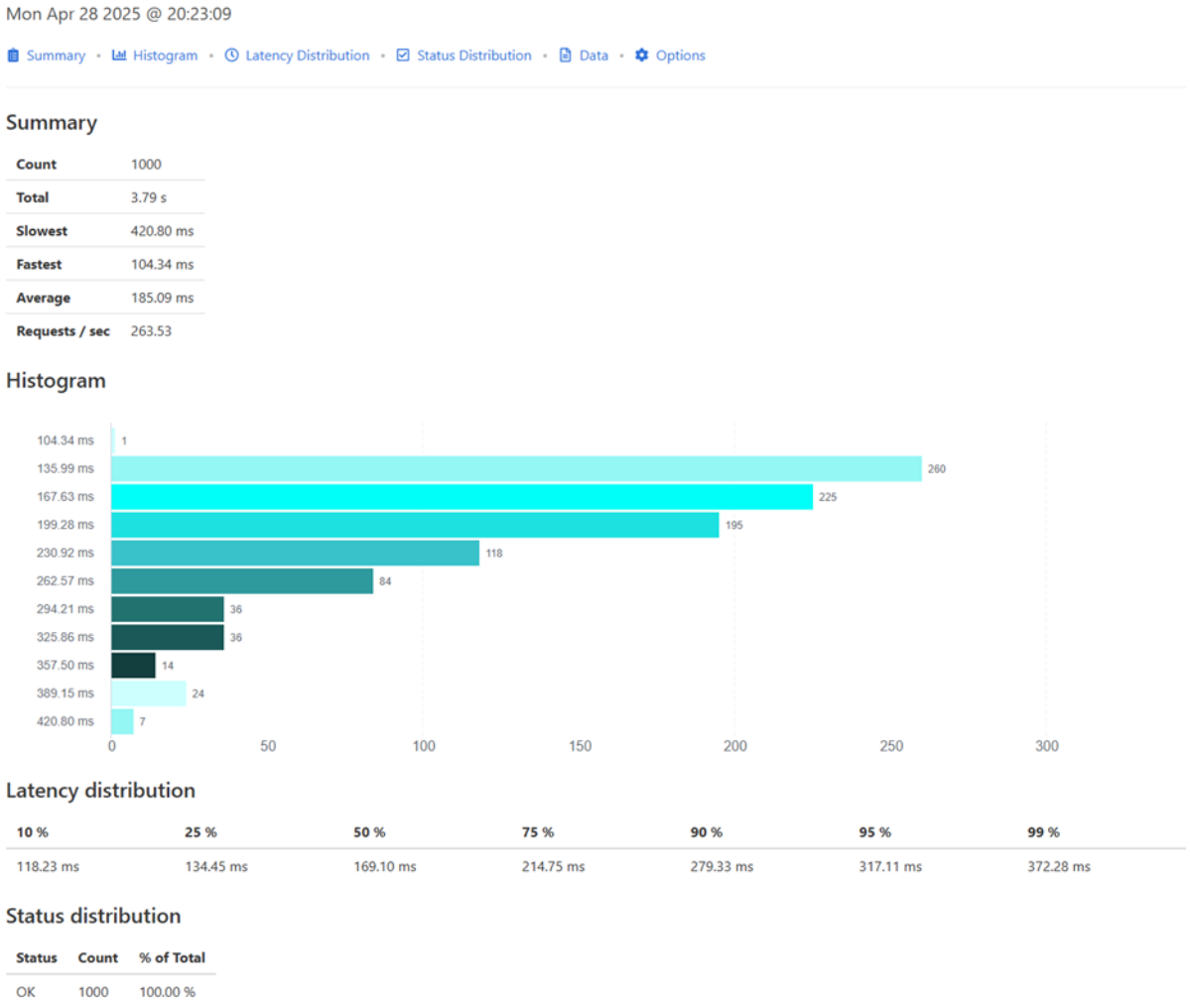

Рисунок 23 – График нагрузочного тестирования SendEmail

Итог: интеграционные проверки подтвердили корректную сквозную работу сервисов, а нагрузочные – устойчивую обработку параллельных запросов при жёстких ограничениях по CPU и памяти.

Рисунки 21-23 показывают ход тестов и полученные результаты.

Выводы по главе 3

В данной главе описан процесс реализации приложения для автоматизации управления эксплуатацией жилого фонда. Приложение было разработано с учетом масштабируемости, производительности и удобства интеграции.

В качестве языка программирования выбран Go (Golang), отличающийся высокой производительностью, простотой конкурентного программирования и быстрым циклом компиляции и деплоя.

Как основная реляционная СУБД используется PostgreSQL, что обеспечивает поддержку сложных транзакций, индексов и аналитических запросов, а также стабильную работу программного обеспечения при увеличении нагрузки.

Интерфейс приложения разработан с учетом простоты и удобства для всех категорий пользователей: жителей, сотрудников диспетчерской службы и администраторов. Он реализован в адаптивном дизайне и корректно работает на различных устройствах.

Также проведены тесты программного обеспечения, которые показали устойчивость системы, корректную работоспособность приложения и не выявили сбоев.

Заключение

В ходе выполнения работы по разработке приложения для автоматизации управления эксплуатацией жилого фонда был проведен ряд исследований предметной области и осуществлен анализ существующих аналогичных приложений.

Также были определены требования к программному обеспечению, которые определяют его функциональные особенности и обеспечивают качественную работу.

На стадии проектирования была выбрана гексагональная архитектура приложения, реализованная тремя микросервисами, использован инкрементальный подход на базе Agile/Scrum с короткими спринтами и спроектирована логическая и физическая модель данных.

Приложение реализовано и прошло интеграционное и нагрузочное тестирование с использованием утилиты ghz.

Результаты тестирования подтвердили эффективность и устойчивость разработанной системы, ее способность выдерживать ожидаемые нагрузки и оперативно обрабатывать поступающие заявки.

Таким образом, все задачи выпускной квалификационной работы выполнены, разработанное приложение полностью соответствует поставленной цели.

Список используемой литературы и источников

1. Алпатов А. Н. Архитектура, проектирование и разработка программных средств: учебное пособие / А. Н. Алпатов, И. Е. Рогов. – Москва : РТУ МИРЭА, 2023. – 120 с.
2. Бабушкин В. М. Разработка защищенных программных средств информатизации производственных процессов предприятия: учебное пособие / В. М. Бабушкин. – Казань : КНИТУ–КАИ, 2020. – 256 с.
3. Баланов А. Н. Внедрение методологий в IT: Agile, Scrum и другие: учебное пособие для вузов / А. Н. Баланов. – Санкт-Петербург. : Лань, 2024. – 188 с.
4. Бутчер С. Введение в PostgreSQL / С. Бутчер. – Москва : ДМК Пресс, 2021. – 528 с.
5. Вон В., Аккерман П. Микросервисы и контейнеры Docker / В. Вон, П. Аккерман. – Москва : ДМК Пресс, 2019. – 240 с.
6. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. – Москва : Стандартинформ, 2011.
7. Донован А., Керниган Б. Язык программирования Go / А. Донован, Б. Керниган. – Москва : Вильямс, 2016. – 432 с.
8. Карлос Б., Пейтон Д. React и Next.js для разработки современных веб-приложений / Б. Карлос, Д. Пейтон. – Москва : ДМК Пресс, 2022. – 368 с.
9. Клеппман М. Высоконагруженные приложения: программирование, масштабирование, поддержка / М. Клеппман. – Санкт-Петербург : Питер, 2018. – 640 с.
10. Котлинский С. В. Разработка моделей предметной области автоматизации: учебник для вузов / С. В. Котлинский. – Санкт-Петербург : Лань, 2021. – 412 с.

11. Остроух А. В. Проектирование информационных систем: монография / А. В. Остроух, Н. Е. Суркова. – 2-е изд., стер. – Санкт-Петербург : Лань, 2021. – 164 с.

12. Ричардсон К. Микросервисы: паттерны разработки и рефакторинга / К. Ричардсон. – Санкт-Петербург : Питер, 2019. – 544 с.

13. Роббинс Д. Отладка, тестирование и оптимизация программ на Go / Д. Роббинс. – Москва : ДМК Пресс, 2021. – 256 с.

14. Тихонова Н. А. Проектирование информационной системы: учебное пособие / Н. А. Тихонова [Электронный ресурс]. – Омск : ОмГУПС, 2021. URL: https://e.lanbook.com/book/190259 (дата обращения: 21.05.2025).

15. Хеллер М. gRPC: запуск и эксплуатация облачных приложений / М. Хеллер. – Санкт-Петербург : БХВ-Петербург, 2020. – 328 с.

16. Цифровизация производства: учебно-методическое пособие / И. Н. Хаймович, Е. Г. Демьяненко, С. Г. Симагина, Е. А. Мешкова. – Самара : Самарский университет, 2023. – 168 с.

17. Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем / Э. Эванс. – Москва : Вильямс, 2018. – 448 с.

18. ТОП 15 лучших ЖКХ программ для УК и ТСЖ для управления многоквартирными домами: [Электронный ресурс]. URL: https://vc.ru/links/1468566-top-15-luchshih-zhkh-programm-dlya-uk-i-tszh-dlya-upravlenie-mnogokvartirnymi-domami (дата обращения: 21.05.2025).

19. Docker Inc. Docker Documentation: [Электронный ресурс]. URL: https://docs.docker.com/ (дата обращения: 21.05.2025).

20. Go Documentation: [Электронный ресурс]. URL: https://golang.org/doc/ (дата обращения: 21.05.2025).

21. gRPC Documentation: [Электронный ресурс]. URL: https://grpc.io/docs/ (дата обращения: 21.05.2025).

22. PostgreSQL Documentation: [Электронный ресурс]. URL: https://www.postgresql.org/docs/ (дата обращения: 21.05.2025).

23. Protocol Buffers Documentation: [Электронный ресурс]. URL: https://protobuf.dev/ (дата обращения: 21.05.2025).

24. TypeScript Documentation: [Электронный ресурс]. URL: https://www.typescriptlang.org/docs/ (дата обращения: 21.05.2025).

25. Vercel Inc. Next.js Documentation: [Электронный ресурс]. URL: https://nextjs.org/docs (дата обращения: 21.05.2025).

Приложение А

Воспроизведение сценария «Создание нового пользователя»

Запрос:

```
grpcurl -plaintext -d '{
  "email": "test@alexandrovstroy.ru",
  "password": "SecurePass123!",
  "fullName": "Иванов Иван Иванович",
  "phone": "+7 (999) 123-45-67",
  "role": "ROLE_RESIDENT",
  "address": "ул. Ленина, д. 42",
  "apartment": "15"
}' localhost:50052 user.v1.UserService/CreateUser
```

Ответ:

```
{
  "id": "73bc30f0-cc4f-430b-b4e4-e351328214f7",
  "email": "test@alexandrovstroy.ru",
  "fullName": "Иванов Иван Иванович",
  "phone": "+7 (999) 123-45-67",
  "role": "ROLE_ADMIN",
  "address": "ул. Ленина, д. 42",
  "createdAt": "2025-10-01T13:55:23.726172910Z",
  "updatedAt": "2025-10-01T13:55:23.726172910Z",
  "apartment": "15"
}
```

Приложение Б

Воспроизведение сценария «Аутентификация и получение токена»

Запрос:

```
grpcurl -plaintext -d '{
  "email": "test@alexandrovstroy.ru",
  "password": "SecurePass123!"
}' localhost:50052 user.v1.UserService/Authenticate
```

Ответ:


```
{
  "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RAYWxleGFu
ZHJvdnN0cm95LnJ1IiwiZXhwIjoxNzU5NDEzMzI0LCJpYXQiOjE3NTkzMjY5
MjQsInJvbGUiOiJBRE1JTiIsInN1YiI6IjczYmMzMGYwLWNjNGYtNDMwYi1i
NGU0LWUzNTEzMjgyMTRmNyJ9.Lb6B88OYHYpv5WKhDSDq2_4IxuzR71nJ
L4X2tUBKh8A",
  "user": {
    "id": "73bc30f0-cc4f-430b-b4e4-e351328214f7",
    "email": "test@alexandrovstroy.ru",
    "fullName": "Иванов Иван Иванович",
    "phone": "+7 (999) 123-45-67",
    "role": "ROLE_ADMIN",
    "address": "ул. Ленина, д. 42",
    "createdAt": "2025-10-01T13:55:23.726173Z",
    "updatedAt": "2025-10-01T13:55:23.726173Z",
    "apartment": "15"
  }
```

Приложение В

Воспроизведение сценария «Регистрация заявки»

Запрос:

```
grpcurl -plaintext -d '{
  "address": {
    "street": "ул. Ленина",
    "building": "42",
    "apartment": "15",
    "floor": 3,
    "entrance": 1
  },
  "description": "Прорыв трубы в ванной комнате, требуется срочный ремонт",
  "category": "Водоснабжение",
  "priority": 2,
  "createdBy": "73bc30f0-cc4f-430b-b4e4-e351328214f7"
}' localhost:50051 emergency.v1.EmergencyService/CreateEmergency
```

Ответ:

```
{
  "emergency": {
    "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
    "address": {
      "street": "ул. Ленина",
      "building": "42",
      "apartment": "15",
      "floor": 3,
      "entrance": 1
    },
    "description": "Прорыв трубы в ванной комнате, требуется срочный
ремонт",
```

```
    "category": "Водоснабжение",
    "priority": 2,
    "status": "NEW",
    "createdAt": "2025-10-01T13:55:24.130377051Z",
    "updatedAt": "2025-10-01T13:55:24.130377081Z",
    "createdBy": "73bc30f0-cc4f-430b-b4e4-e351328214f7"
  }
}
```

Приложение Г

Воспроизведение сценария «Чтение заявки по идентификатору»

Запрос:

```
grpcurl -plaintext -d '{
  "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0"
}' localhost:50051 emergency.v1.EmergencyService/GetEmergency
```

Ответ:


```
{
  "emergency": {
    "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
    "address": {
      "street": "ул. Ленина",
      "building": "42",
      "apartment": "15",
      "floor": 3,
      "entrance": 1
    },
    "description": "Прорыв трубы в ванной комнате, требуется срочный ремонт",
    "category": "Водоснабжение",
    "priority": 2,
    "status": "NEW",
    "createdAt": "2025-10-01T13:55:24.130377Z",
    "updatedAt": "2025-10-01T13:55:24.130377Z",
    "createdBy": "73bc30f0-cc4f-430b-b4e4-e351328214f7"
  }
}
```

Приложение Д

Воспроизведение сценария «Выборка обращений со статусом NEW»

Запрос:

```
grpcurl -plaintext -d '{
  "status": "NEW"
}' localhost:50051 emergency.v1.EmergencyService/ListEmergencies
```

Ответ:


```
{
  "emergencies": [
    {
      "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
      "address": {
        "street": "ул. Ленина",
        "building": "42",
        "apartment": "15",
        "floor": 3,
        "entrance": 1
      },
      "description": "Прорыв трубы в ванной комнате, требуется срочный ремонт",
      "category": "Водоснабжение",
      "priority": 2,
      "status": "NEW",
      "createdAt": "2025-10-01T13:55:24.130377Z",
      "updatedAt": "2025-10-01T13:55:24.130377Z",
      "createdBy": "73bc30f0-cc4f-430b-b4e4-e351328214f7"
    }
  ]
}
```

Приложение Е

Воспроизведение сценария «Перевод в статус IN_PROGRESS»

Запрос:

```
grpcurl -plaintext -d '{
  "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
  "status": "IN_PROGRESS"
}' localhost:50051 emergency.v1.EmergencyService/UpdateEmergencyStatus
```

Ответ:


```
{
  "emergency": {
    "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
    "address": {
      "street": "ул. Ленина",
      "building": "42",
      "apartment": "15",
      "floor": 3,
      "entrance": 1
    },
    "description": "Прорыв трубы в ванной комнате, требуется срочный ремонт",
    "category": "Водоснабжение",
    "priority": 2,
    "status": "IN_PROGRESS",
    "createdAt": "2025-10-01T13:55:24.130377Z",
    "updatedAt": "2025-10-01T13:55:24.603520206Z",
    "createdBy": "73bc30f0-cc4f-430b-b4e4-e351328214f7"
  }
}
```

Приложение Ж

Воспроизведение сценария «Назначение исполнителя»

Запрос:

```
grpcurl -plaintext -d '{
  "emergency_id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
  "assignee_id": "d182bd1e-ca75-4139-bbc3-bbccc08e1629"
}' localhost:50051 emergency.v1.EmergencyService/AssignEmergency
```

Ответ:


```
{
  "emergency": {
    "id": "a1ae0aed-8fb7-45b4-82da-deab761c1bd0",
    "address": {
      "street": "ул. Ленина",
      "building": "42",
      "apartment": "15",
      "floor": 3,
      "entrance": 1
    },
    "description": "Прорыв трубы в ванной комнате, требуется срочный ремонт",
    "category": "Водоснабжение",
    "priority": 2,
    "status": "IN_PROGRESS",
    "createdAt": "2025-10-01T13:55:24.130377Z",
    "updatedAt": "2025-10-01T13:55:24.976764606Z",
    "createdBy": "73bc30f0-cc4f-430b-b4e4-e351328214f7",
    "assignedTo": "d182bd1e-ca75-4139-bbc3-bbccc08e1629"
  }
}
```

Приложение И

Воспроизведение сценария «Обновление профиля»

Запрос:

```
grpcurl -plaintext -d '{
  "id": "73bc30f0-cc4f-430b-b4e4-e351328214f7",
  "phone": "+7 (999) 111-22-33",
  "address": "ул. Ленина, д. 42А",
  "apartment": "16"
}' localhost:50052 user.v1.UserService/UpdateUser
```

Ответ:

```
{
  "id": "73bc30f0-cc4f-430b-b4e4-e351328214f7",
  "email": "test@alexandrovstroy.ru",
  "fullName": "Иванов Иван Иванович",
  "phone": "+7 (999) 111-22-33",
  "role": "ROLE_ADMIN",
  "address": "ул. Ленина, д. 42А",
  "createdAt": "2025-10-01T13:55:23.726173Z",
  "updatedAt": "2025-10-01T13:55:25.194869119Z",
  "apartment": "16"
}
```

Приложение К

Воспроизведение сценария «Удаление пользователя»

Запрос:

```
grpcurl -plaintext -d '{
  "id": "73bc30f0-cc4f-430b-b4e4-e351328214f7"
}' localhost:50052 user.v1.UserService/DeleteUser
```

Ответ:

```
{
  "success": true
}
```