

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.02 «Информационные системы и технологии»

Направленность Информационные системы и технологии

БАКАЛАВРСКАЯ РАБОТА

на тему

Автоматизированная информационная система
страховой компании

Студент _____ Стрижков Илья Олегович
(подпись, дата) (ФИО полностью)

Руководитель _____ Попова Н.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2025

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет

Вычислительной техники

Кафедра

МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____ А.Ю. Козлов
«20» декабря 2024 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

1. Студент Стрижков Илья Олегович гр. 21ВИ1 факультета ВТ
<Фамилия, имя, отчество полностью>
2. Тема работы Автоматизированная информационная система страховой компании

Тема утверждена приказом ПГУ № _____ от " _____ " _____ 20 _____

3. Руководитель работы Доцент кафедры «МО и ПЭВМ» Попова Н.А.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) *Назначение: автоматизированная информационная система страховой компании*

2) *Среда выполнения: операционная система Linux*

Языковые средства разработки: Python3, TypeScript, PostgreSQL

Инструментальные средства: NeoVim, Electron js, PgAdmin 4

Технологии разработки: микросервисная архитектура, ООП.

3) *Основные функции приложения:*

а) регистрация и авторизация пользователей;

б) создание заявок страхования;

в) добавление клиентов в базу;

г) просмотр аналитики страховой компании;

д) Редактирование ролей пользователей..

5. Перечень подлежащих разработке вопросов

1) *Анализ предметной области и требований к системе;*

2) *Проектирование программных средств;*

3) *Реализация программных средств;*

4) *Тестирование программных средств;*

5) *Оформление пояснительной записки.*

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	08.04.2025 – 14.04.2025	
2	Проектирование программных средств	20%	15.04.2025 – 25.04.2025	
3	Реализация программных средств	30%	26.04.2025 – 12.05.2025	
4	Тестирование программных средств	20%	13.05.2025 – 22.05.2025	
5	Оформление пояснительной записки	20%	23.05.2025 – 03.06.2025	

Дата выдачи задания «20» декабря 2024г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 77 листов, 27 рисунков, 17 таблиц, 20 использованных источников и 2 приложения.

WEB-ПРИЛОЖЕНИЕ, МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, REST, POSTGRESQL, TYPESCRIPT, БАЗА ДАННЫХ, REACTJS.

Объект исследования – автоматизированная информационная система страховой компании

Цель выпускной квалификационной работы – разработать web-приложение для работы страховой компании.

Технологии разработки – разработка клиентской части приложения проводилась на языке программирования TypeScript с использованием библиотеки ReactJS в среде разработки NeoVim. Разработка серверной части приложения производилась на языке программирования Python3 в среде разработки NeoVim. Разработка базы данных проводилась в СУБД PostgreSQL.

Среда исполнения – Linux, web-приложение.

Результаты работы – в ходе выполнения выпускной квалификационной работы были созданы клиентская и серверная часть автоматизированной информационной системы страховой компании. Все требования к разработке приложения соблюдены.

					ПГУ 09.03.02 – 08BP211.30 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Автоматизированная информационная система страховой компании. Пояснительная записка	Лит.	Лист	Листов
Разраб.		Стрижков И.О.					4	77
Пров.		Попова Н.А.				Группа 21ВИ1		
Реценз.								
Н. Контр.		Такташкин Д.В						
Утв.		Козлов А.Ю.						

Содержание

Введение.....	7
1 Анализ предметной области и постановки задачи на разработку автоматизированных средств работы страховой компании	8
1.1 Общая характеристика страховой деятельности	8
1.1.1 Типовые процессы, подлежащие автоматизации	9
1.1.2 Проблемы при ручном управлении страховыми операциями	11
1.2 Сравнительный анализ приложений аналогов	11
1.2.1 Цели и задачи сравнительного анализа	11
1.2.2 Выбор приложений аналогов для анализа	11
1.2.3 Обзор приложений аналогов	12
1.2.4 Результаты сравнительного анализа	15
1.3 Постановка задачи на разработку автоматизированной информационной системы страховой компании	17
1.3.1 Что именно реализуется	17
1.3.2 Основные функции системы	17
1.3.3 Ограничения системы	18
1.4 Планирование процесса разработки программного обеспечения	19
2 Проектирование автоматизированных средств информационной системы страховой компании	21
2.1 Общие принципы проектирования	21
2.2 Разработка архитектуры программного обеспечения	24
2.3 Прототипирование пользовательского интерфейса	27
2.4 Обоснование выбора средств разработки	28
2.4.1 Обоснование выбора средств автоматизации управления проектом	28

2.4.2 Обоснование выбора инструментальных средств разработки программного обеспечения	28
2.4.3 Обоснование выбора языковых средств разработки	30
3.Реализация приложения для работы страховой компании.....	33
3.1 Разработка клиентской части приложения.....	33
3.1.1 Разработка пользовательского интерфейса клиентского приложения.....	33
3.2 Разработка программного обеспечения клиентского приложения.....	36
3.3 Разработка серверной части приложения	39
3.4 Разработка базы данных страховой компании	46
4. Контроль качества программного обеспечения автоматизированной информационной системы страховой компании	50
4.1. Планирование процесса контроля качества программного обеспечения.....	50
4.1.1 Мероприятия по контролю качества программного обеспечения ...	50
4.2 Обоснование выбора модели тестирования	51
4.3 Обоснование выбора метрик программного кода	52
4.4 Тестирование программных средств.....	52
4.4.1 Тестирование с помощью юнит-тестов	52
4.4.2 Автоматическое сквозное тестирование	54
4.5 Оценка качества программного кода на основе метрик.....	55
4.5.1 Метрика SLOC.....	55
Заключение	56
Список использованных источников	57
Приложение А (обязательное) Листинг программы	59
Приложение Б (рекомендуемое) Скриншоты работы приложения.....	74

Введение

Страховая организация – юридическое лицо, созданное в соответствии с законодательством Российской Федерации для осуществления деятельности по страхованию и (или) перестрахованию и получившее лицензию на осуществление соответствующего вида страховой деятельности в установленном настоящим законом порядке.

Возникает необходимость в средстве автоматизации страхования клиентов, которое бы позволяло оптимизировать процесс регистрации клиентов и ведения базы договоров.

Целью разработки является создание программного обеспечения для автоматизации работы страховой компании. При этом должна будет осуществляться кроссплатформенность исходного приложения.

Для выполнения поставленной цели нужно решить следующие задачи:

- проанализировать предметную область;
- осуществить сравнительный анализ приложений-аналогов;
- выполнить постановку задачи на разработку средства для автоматизации работы страховой компании;
- смоделировать функциональные возможности приложения с помощью языка UML;
- выбрать средства для разработки, развертывания, тестирования и интеграции приложения;
- произвести проектирование архитектуры приложения, пользовательского интерфейса;
- разработать приложения для автоматизации работы страховой компании;
- произвести непрерывное развертывание и интеграцию приложения;
- выполнить тестирование программного обеспечения.

В результате выполнения работы должно быть реализовано средство для автоматизации работы страховой компании.

1 Анализ предметной области и постановки задачи на разработку автоматизированных средств работы страховой компании

1.1 Общая характеристика страховой деятельности

Оформление страхования - регулярная практика по всему миру. Само страхование представляет собой совокупность различных процессов, преимущественно направленных на защиту каких-либо имущественных интересов физических и юридических лиц при наступлении определенных неблагоприятных событий, называемых страховыми случаями. Данная совокупность разделяется по функциональной направленности процессов, поскольку основные из них являются экономическими, правовыми, информационными, а также социальными. Основой страхования выступает заключение договора между страхователем и страховщиком, который подразумевает выплату страховой премии страхователем и выплату страховой компанией определенной суммы средств, называемой страховым возмещением, при наступлении страхового случая.

На данный момент рынок предоставляет широкий выбор видов страхования, охватывающих различные сферы жизни и деятельности человека. Комплексная классификация позволяет выделить ключевые признаки при заключении страхового договора и разделить спектр оказываемых услуг по обязательности заключения договора, по объекту страховой защиты и по отраслям страховой практики. В условиях постоянного роста объема клиентов страховых агентств, а как следствие, и роста объема проводимых операций, компании остро нуждаются в автоматизированных средствах управления.

Оказание страховой деятельности на территории Российской Федерации сопровождается федеральным законом №401 5-1 «Об организации страхового дела», а также гражданским кодексом РФ в виде статей о договоре страхования, а также нормативными актами Центрального Банка РФ (ЦБ РФ), в том числе по обязательному страхованию. Вышеперечисленные документы выступают законодательной базой, без которой оказание как таковых страховых услуг было бы невозможным.

Внутри страховых компаний можно выделить несколько ключевых ролей, каждая из которых взаимодействует с информацией и участвует в её обработке:

- клиенты (страхователи): физические и юридические лица, оформляющие договоры страхования;
- страховые агенты: сотрудники, отвечающие за взаимодействие с клиентами, консультирование, заключение договоров;
- андеррайтеры: специалисты, анализирующие страховые риски и принимающие решения о возможности страхования;
- специалисты по урегулированию убытков: рассматривают заявления на выплату по страховым случаям, проверяют документы, принимают решения о компенсациях;
- бухгалтерия/финансовый отдел: обрабатывает платежи, премии, начисления и выплаты;
- руководство и аналитики: используют отчётность для принятия управленческих решений;
- администраторы ИТ-систем: поддерживают работу автоматизированной системы, контролируют доступ, обеспечивают информационную безопасность.

Таким образом, предметная область страхования охватывает широкий спектр процессов, что требует строгого учета, регламентации и точности. Автоматизация данных процессов не только повышает эффективность работы компании, но и обеспечивает соответствие законодательным требованиям, ускоряет обслуживание клиентов и минимизирует риски человеческих ошибок. Внедрение специализированной информационной системы становится неотъемлемым условием для фиксирования современной страховой компании в конкурентной среде.

1.1.1 Типовые процессы, подлежащие автоматизации

Современная страховая компания ежедневно сталкивается с повторяющимися задачами, которые требуют высокой точности и скорости выполнения. Основные процессы, которые следует автоматизировать в угоду эффективности работы компаний:

- регистрация и учет клиентов;
- ведение базы данных клиентов;
- история обращений и страховых случаев;
- учет паспортных и юридических данных, контактной информации;
- оформление договоров страхования;
- генерация договоров по шаблонам;
- расчёт страховых премий с учетом коэффициентов (возраст, стаж, регион, КБМ, франшиза и др.);
- возможность автоматического продления или завершения полиса;
- учёт платежей;
- контроль поступления платежей от клиентов;
- формирование графика платежей по рассрочке;
- уведомление об оплате или просрочке;
- обработка страховых случаев;
- приём заявлений;
- назначение экспертизы;
- хранение документации (фото, акты, заключения);
- принятие решения по выплатам и расчёт суммы компенсации;
- отчетность и аналитика;
- формирование регламентированных отчётов (для ЦБ РФ и внутреннего аудита);
- анализ количества страховых случаев, убыточности, прибыльности клиентов;
- визуализация данных по направлениям деятельности;
- документооборот;
- генерация PDF-документов;
- электронная подпись;
- архивирование договоров, актов, заявлений;
- интеграции;
- взаимодействие с внешними государственными информационными ресурсами;
- обмен данными с банками и платёжными шлюзами;

Для страховой компании выделено много процессов для автоматизации. Рассмотрим некоторые из них

1.1.2 Проблемы при ручном управлении страховыми операциями

При отсутствии автоматизированной системы страховые компании регулярно сталкиваются с рядом различных проблем. Значительная часть из которых влечет за собой финансовые потери, например, ошибки при расчетах страховых премий и выплат или сложности с контролем за пролонгацией договоров и неуплаченными взносами. Задержки в обработке заявлений снижают удовлетворенность клиентов, как и проблемы с информацией, её дублирование или отсутствие единых справочников. Потеря эффективности работы компаний возникает из-за низкой оперативности при формировании отчетности, что увеличивает временные затраты всех последующих процессов компании, а также из-за трудностей ведения историй клиентов и страховых случаев. Также существует невозможность интеграции с внешними сервисами, такими как, например, российский союз автостраховщиков (РСА), из-за разрозненности данных.

1.2 Сравнительный анализ приложений аналогов

1.2.1 Цели и задачи сравнительного анализа

Перед разработкой автоматизированной информационной системы страховой компании важно исследовать существующие решения, присутствующие на рынке, с целью выявления достоинств и недостатков.

Это позволит учесть сильные стороны при проектировании собственной системы и избежать распространённых ошибок.

1.2.2 Выбор приложений аналогов для анализа

В анализ включены три популярных и разноплановых продукта: отечественное универсальное решение на базе 1С, мощная зарубежная платформа «INSIS» и специализированный веб – сервис «Е-полис», предназначенный для работы с обязательным страхованием, например, с полисом обязательного страхования автогражданской ответственности (ОСАГО).

В таблице 1 приведен список приложений для работы со страхованием

Таблица 1 – Список приложений для страхования

Наименование приложения	Основные функции	Условия лицензирования	Класс приложений	Операционная система
1С Страхование	Финансовый учет, ведение клиентской базы, расчет страховых премий, интеграция с сайтами	Платный	Desktop-приложение	Любая
INSIS	Финансовый учет, ведение клиентской базы, расчет страховых премий, интеграция с сайтами	Платный	Desktop-приложение	Windows 8, 10, 11
Е Полис	Финансовый учет, ведение клиентской базы, расчет страховых премий, интеграция с сайтами	Бесплатный	Web-приложение	Windows 7,8,10

Таким образом были выбраны наиболее используемые в настоящее время приложения для страховых компаний.

1.2.3 Обзор приложений аналогов

«1С: Страхование» – программный продукт на платформе «1С: Предприятие», предназначенный для автоматизации процессов в страховых компаниях. В системе реализованы механизмы учета договоров страхования, регистрации клиентов, расчёта страховых премий и обработки страховых случаев. Имеется поддержка различных видов страхования – автострахование, имущество, здоровье и др. Система позволяет формировать регламентированную отчетность и вести учет в соответствии с российским законодательством. Также

предусмотрена интеграция с бухгалтерией и другими модулями 1С. Интерфейс настраиваемый, однако требует предварительной адаптации под нужды конкретной компании. Веб-доступ реализуется через платформу «1С: Fresh» или с использованием внешних решений. Приложение активно используется средними и крупными страховыми организациями.

Начисление зарплаты 0000-000001 от 31.10.2020 23:59:59

Провести и закрыть Записать Провести Дт КТ [Иконка] [Иконка] Отчеты Выплатить Еще

Зарплата за: Октябрь 2020 [Календарь]

Номер: 0000-000001 от: 31.10.2020 [Календарь] [Сообщения]

[Доля ЕНВД: 0% ?](#)

Заполнить Добавить Подбор Начислить Удержать [Вверх] [Вниз] Еще

N	Сотрудник	Дни	Часы		Взносы
1	Воронин Георгий Михайлович	22,00	176,00		
2	Жуланова Галина Алексан...	22,00	176,00		
				150 000,00	12,00 19 500,00 45 300,00

Комментарий: [Поле для комментария]

Рисунок 1 – Интерфейс программы «1С: Страхование»

«INSIS» от компании Fadata – мощная интегрированная платформа, предназначенная для автоматизации всех бизнес-процессов страховой компании. Программа охватывает полный жизненный цикл страхового продукта – от создания и расчёта премий до урегулирования страховых случаев и анализа эффективности.

В системе реализована поддержка международных стандартов, управление договорами, клиентами, платежами, убытками и отчетностью. «INSIS» построена на модульной архитектуре и может быть адаптирована под требования конкретной компании.

Имеется поддержка мультивалютности, многоязычности и мультиканальности. «INSIS» работает как веб-приложение с возможностью интеграции с внешними сервисами. Из-за очень высокой сложности и большой

стоимости внедрение чаще всего осуществляется в крупных международных компаниях.

The screenshot shows the INSIS Claims interface. At the top, there's a header with 'INSIS Claims' and navigation tabs for 'CLAIMS', 'CLAIMS', 'CLAIMS', and 'CLAIMS'. Below the header, there's a summary section with fields for 'Case', 'Policy', 'Claim', and 'Status'. The main part of the interface is a table with columns: 'Claim number', 'Policy number', 'Designated name', 'Amount type', 'Amount', 'Currency', 'Due date', 'Registration date', 'Paid on', 'Payment amount', and 'Status'. The table contains several rows of data, including 'Unpaid Premium' and 'Paid' entries.

Claim number	Policy number	Designated name	Amount type	Amount	Currency	Due date	Registration date	Paid on	Payment amount	Status
22000000000000000000	000000000000000000	000000000000000000	Unpaid Premium	99.00	EUR	05.04.2015	02.12.2022	05.04.2015	220000000000000000	PAID
22000000000000000000	000000000000000000	000000000000000000	Paid	99.00	EUR	05.04.2015	02.12.2022	05.04.2015	220000000000000000	PAID

Рисунок 2 – Интерфейс программы INSIS

«Е-полис» – государственный веб-сервис, разработанный РСА для оформления электронных полисов ОСАГО. Система позволяет пользователю рассчитать стоимость полиса, выбрать страховую компанию и оформить электронный договор онлайн. Для расчета используется база данных коэффициента бонус-малус (БД КБМ), данных о транспортном средстве (ТС) и водителях. Сервис интегрирован с системами государственной инспекции безопасности дорожного движения (ГИБДД) и единой автоматизированной информационной системой технического осмотра (ЕАИСТО), что обеспечивает актуальность проверяемой информации. Интерфейс прост и предназначен преимущественно для конечных пользователей, а не для внутреннего документооборота страховых компаний. Мобильного приложения нет, доступ осуществляется через браузер. Программа выполняет строго определённую функцию и не подходит для комплексной автоматизации работы страховой компании.

The screenshot shows the 'Е-полис' (E-policy) web interface. At the top, there's a header with 'ЭЛЕКТРОННЫЙ ПОЛИС' and navigation tabs for 'ОСАГО', 'ВОЗ', 'ИМУЩЕСТВО', 'НС', 'КАСКО', and 'ЗК'. Below the header, there's a section for 'Выкупать полис ОСАГО' (Buy OSAGO policy). The main part of the interface is a form with fields for 'Физическое лицо' (Physical person), 'Юридическое лицо' (Legal person), and 'Индивидуальный предприниматель' (Individual entrepreneur). The form includes fields for 'Фамилия' (Surname), 'Имя' (Name), 'Отчество' (Patronymic), 'Дата рождения' (Date of birth), 'Паспорт гражданина РФ' (Russian Federation passport), 'Срок' (Term), 'Номер' (Number), 'Дата выдачи' (Date of issue), 'Кем выдан' (Issued by), 'Адрес постоянной регистрации' (Permanent residence address), and 'Почтовый (фактический) адрес' (Postal (actual) address).

Рисунок 3 – Интерфейс программы «Е-полис»

В целом представленные программные продукты удовлетворяют основным требованиям, предъявляемым к информационным системам страховых компаний: ведение клиентской базы, оформление страховых договоров, расчёт премий, обработка страховых случаев и формирование отчётности. Однако между системами имеются различия по степени функциональной полноты, удобству интерфейса, поддержке мобильных платформ и гибкости настройки. Некоторые решения ориентированы исключительно на крупные компании, другие - на автоматизацию отдельных этапов (например, только оформление ОСАГО).

1.2.4 Результаты сравнительного анализа

Выделим основные критерии для оценки программных решений:

- критерии;
- модульность и возможность расширения функционала;
- поддержка различных видов страхования;
- наличие CRM и системы учета клиентов;
- средства организации финансового учета;
- ведение учета страховых случаев и урегулирования убытков;
- возможность расчета страховых премий;
- интеграция с внешними системами;
- наличие веб-доступа для пользователей и агентов;
- стоимость владения и лицензирования;
- сложность внедрения и сопровождения;
- гибкость настройки под бизнес-процессы компании.

Лучшим приложением с точки зрения функционального охвата и гибкости настройки является «INSIS» от компании Fadata, поскольку программный продукт обладает широкой поддержкой всех видов страхования, имеет встроенные модули управления клиентами, премиями, убытками и финансами. Однако данное решение обладает высокой сложностью внедрения и требует значительных финансовых затрат, что делает продукт доступным преимущественно для крупных страховых компаний. Оптимальным выбором для средних и малых компаний можно считать «1С: Страхование», так как приложение обеспечивает необходимую функциональность и адаптацию под

российское законодательство, сохранив при этом разумный уровень внедрения. Рассмотрим общие недостатки, выявленные среди всех рассмотренных решений.

В таблице 2 представлены сравнение программного обеспечения для страховых компаний.

Таблица 2 – Сравнение программного обеспечения страховых компаний

Критерий	1С:Страхование	INSIS (Fadata)	Е-полис (РСА)
Тип системы	Корпоративная, гибридная	Корпоративная, веб	Веб-приложение (государственное)
Модульность	Да (расширяется конфигурациями)	Да (модульный подход)	Нет
Поддержка разных видов страхования	Да	Да	Только ОСАГО
CRM/учет клиентов	Да	Да	Нет
Финансовый учет	Да (через 1С)	Да (встроенный)	Нет
Учет страховых случаев	Да	Да	Ограниченно
Расчет премий	Да	Да	Да (ОСАГО)
Интеграция с внешними системами	Да	Да	Да
Поддержка веб-доступа	Ограниченно	Полностью	Да
Стоимость	Средняя/высокая	Очень высокая	Бесплатно
Сложность внедрения	Средняя	Высокая	Не требуется
Гибкость настройки	Высокая	Очень высокая	Отсутствует

Проанализировав информацию, представленную в таблице 2, ни одна из представленных систем не является универсальной для всех категорий страховых компаний. Например, веб - сервис «Е-полис» ориентирован исключительно на оформление полисов ОСАГО и не имеет встроенных модулей управления клиентами, финансами или полноценной аналитики. Программные продукты компаний 1С и Fadata не предоставляют полноценного веб-доступа для конечных клиентов без дополнительной доработки. Кроме того, не во всех решениях предусмотрена возможность гибкого управления параметрами страховых продуктов в упрощенной форме – в некоторых случаях требуется привлечение внешних интеграторов или разработчиков.

1.3 Постановка задачи на разработку автоматизированной информационной системы страховой компании

1.3.1 Что именно реализуется

В рамках дипломного проекта предполагается разработка автоматизированной информационной системы страховой компании с упором на подсистему учета договоров страхования и обработки страховых случаев. Основное назначение системы звучит следующим образом: «Автоматизация работы страховых агентов и специалистов по урегулированию убытков в части оформления, сопровождения и анализа страховых договоров, а также регистрация обращений клиентов по наступившим страховым случаям.»

Система должна обеспечивать:

- ведение единой базы клиентов;
- формирование и хранение страховых договоров;
- расчёт страховых премий;
- прием заявлений о страховых случаях;
- контроль за выполнением выплат.

Разрабатываемое программное обеспечение призвано повысить эффективность взаимодействия между подразделениями страховой компании, минимизировать человеческий фактор, обеспечить точность расчетов и прозрачность всех процессов.

1.3.2 Основные функции системы

Предполагается реализация ключевого функционала в виде модулей. Модуль, отвечающий за регистрацию клиентов, включает в себя создание и редактирование карточек клиентов, а также ведение клиентской истории договоров и страховых случаев. Также требуется разработка непосредственного модуля заключения договоров страхования, в который входит выбор типа страхования, с автоматическим заполнением шаблона и расчетом премии. Следующий модуль – модуль расчета страховой премии предполагается с учетом тарифных коэффициентов, особенностей клиента и объекта страхования. Модуль обработки страховых случаев включает в себя регистрацию обращений,

предусматривает загрузку документов, изменение статуса и принятие решения. Для грамотного отслеживания работы компании требуется модуль отчётности и аналитики с формированием отчетов по страховым случаям, выданным полисам, просроченным платежам и другим. Работа модуля аналитики невозможна без модуля истории операций с журналом действий по клиентам и договорам. Также требуется модуль уведомлений, суть которого в напоминаниях об необходимости продления договора или предоставления недостающих документов.

1.3.3 Ограничения системы

В процессе разработки также важно учесть перечень из следующих ограничений. Разрабатываемая система будет реализована в виде веб-интерфейса, работа над которым будет производиться в пределах одного офиса или локальной сети, с возможностью последующего масштабирования проекта до приложения с увеличением ресурсов, затраченных на производство и поддержание программного продукта.

Минимальные требования для корректной работы веб-интерфейса подразумевают использование программного продукта на платформах Windows 10, Windows 11 или Linux, с установкой сервера и клиента на одном устройстве, а также должна быть предусмотрена возможность запуска и использования системы в автономной режиме, без подключения к интернету.

Поскольку разрабатываемое программное обеспечение будет реализовано в виде веб - интерфейса нужно учитывать отсутствие поддержки мобильной версии на первом этапе реализации и отсутствие полноценной интеграции с внешними государственными сервисами в базовой версии, а также поддержку ограниченного количества видов страхования и поддержку одного языка интерфейса – русского.

Выделим основных пользователей системы:

- страховые агенты – оформляют договоры, рассчитывают страховые премии и взаимодействуют с клиентами;
- специалисты отдела урегулирования убытков – принимают заявления, проводят анализ и принимают решения о выплатах;

- операторы или офис-менеджеры – ведут клиентскую базу, контролируют оплату и сроки действия договоров;
- руководство страховой компании – получают аналитическую информацию и контролируют отчетность;
- системный администратор – обеспечивает техническое сопровождение, управление пользователями и настройку прав доступа.

Система предусматривает разграничение прав пользователей по ролям и уровням. Каждая роль имеет собственный уровень доступа и функциональность в рамках приложения.

1.4 Планирование процесса разработки программного обеспечения

Одним из важнейших пунктов разработки программного обеспечения является планирование процесса разработки. Особенность процесса планирования в проектном виде заключается в постоянной корректировке плана от начала до стадии завершения.

Для планирования разработки использовалась диаграмма Ганта. Эта диаграмма является одной из самых популярных для построения планирования деятельности, в данном случае для разработки приложения.

Для построения диаграммы Ганта была выбрана программа Microsoft Project 2016, ибо представленные средства позволяют отдельно создавать ресурсы и задачи, которые требуются на проект.

На рисунке 4 представлены ресурсы проекта.

	Название ресурса	Тип	Единица измерения	Краткое название	Группа	Макс. единиц	Стандартная ставка	Ставка сверхурочн	Затраты на исполн.	Начисление	Базовый календарь
	Аналитик	Трудовой	Ан			100%	110,00 Р/ч	200,00 Р/ч	0,00 Р	Пропорционал	Стандартный
	Разработчик ПО	Трудовой	П			100%	160,00 Р/ч	300,00 Р/ч	0,00 Р	Пропорционал	Стандартный
	DevOps-инженер	Трудовой	О			100%	100,00 Р/ч	190,00 Р/ч	0,00 Р	Пропорционал	Стандартный
	Тестирующий	Трудовой	Т			100%	100,00 Р/ч	190,00 Р/ч	0,00 Р	Пропорционал	Стандартный
	Архитектор ПО	Трудовой	А			100%	120,00 Р/ч	230,00 Р/ч	0,00 Р	Пропорционал	Стандартный
	Интернет	Затраты	И							Пропорционал	

Рисунок 4 – Ресурсы проекта.

Исходя из имеющейся информации о ресурсах проекта можно создать графическое представление календарного плана проекта.

На рисунке 5 представлена диаграмма Ганта.

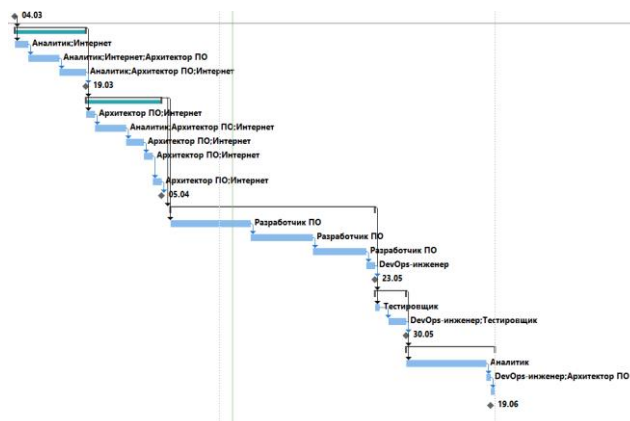


Рисунок 5 – Диаграмма Ганта

Microsoft Project имеет в своем распоряжении достаточный набор готовых отчетов и панель мониторинга, позволяющий контролировать общее состояние проекта и осуществлять проверку и соответствие графику работ.

Планирование разработки системы позволило рационально распределить ресурсы, определить последовательность выполнения задач и предусмотреть временные резервы. Использование диаграммы Ганта и современных инструментов управления проектами обеспечивает прозрачность, гибкость и контроль над реализацией всех этапов разработки программного продукта.

2 Проектирование автоматизированных средств информационной системы страховой компании

2.1 Общие принципы проектирования

При проектировании автоматизированной информационной системы страховой компании необходимо определить функциональные возможности программного обеспечения.

Исходя из поставленной задачи и специфики предметной области. Учесть и представить все функциональные возможности информационной системы позволяет диаграмма прецедентов.

Диаграмма вариантов использования или диаграмма прецедентов – диаграмма, описывающая отношения между пользователями и системой, состоит из следующих компонентов:

1. актер (Actor) – пользователь или то лицо, которое выполняет какие-то действия в системе;
2. варианты использования (Use-cases) – действия, которые выполняет актер;
3. ассоциации – это связи пользователем и вариантами использования;
4. отношения – связи между вариантами использования.

Бывают следующие связи: включение (Include), расширение (Extend), обобщение (Generalization).

Границы системы – задают пределы системы для use кейсов. Если какой-от из них выходит за границы системы, то считывает, что они являются выходящими за пределы данной системы.

На рисунке 6 показана диаграмма вариантов использования функциональных возможностей приложения для автоматизации страховой компании.

Были выделены в качестве основных функций:

- создание справочника;
- создание документ;
- загрузка документа;

– Формирование документа.

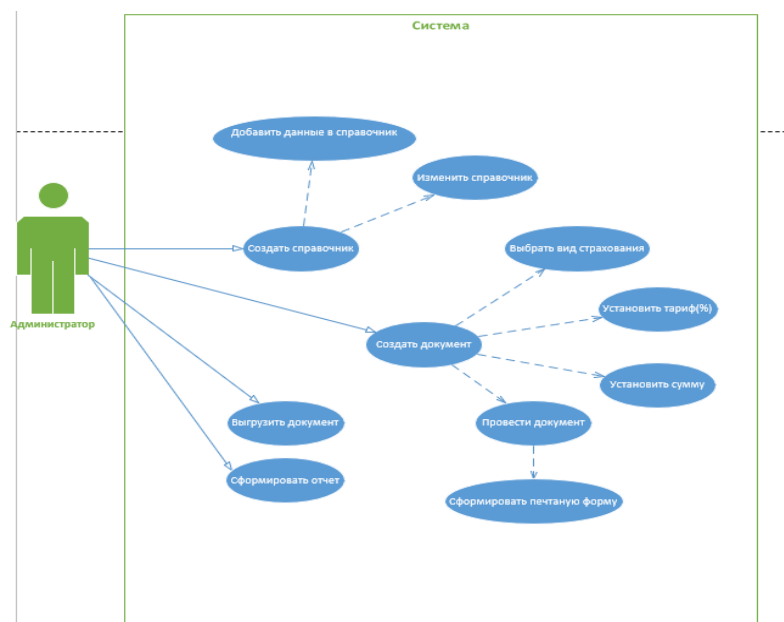


Рисунок 6 – Диаграмма вариантов использования

Таблица 3 – Описание вариантов использования

Цель	Описание
Создать справочник	Пользователь (администратор) инициирует создание нового справочника, в который в дальнейшем будут добавлены данные.
Добавить данные в справочник	В созданный справочник администратор может внести новую запись: название, значение, описание.
Изменить справочник	Возможность редактирования уже существующего справочника: изменение названий, удаление или корректировка записей.
Создать документ	Администратор инициирует создание документа, содержащего условия страхования.
Выбрать вид страхования	При создании документа пользователь выбирает тип страхования (например, авто, здоровье, имущество).
Установить тариф (%)	Устанавливается процент страховой премии, соответствующий выбранному типу страхования.
Установить сумму	Указывается сумма страхования по договору или полису.
Провести документ	Документ переводится в состояние «проведён», после чего недоступен для редактирования.
Сформировать печатную форму	Автоматическое создание печатного документа по шаблону с данными из договора.
Выгрузить документ	Система позволяет сохранить документ в файл (например, PDF или DOCX).
Сформировать отчет	Генерация аналитического или регламентированного отчета по данным в системе.

Таблица 4 – Сценарий и описание варианта использования «Создать справочник»

Наименование	Создать справочник
Краткое описание	Позволяет администратору создать новый справочник в системе
Основные действующие лица	Администратор
Второстепенные действующие лица	Нет
Предусловие	Пользователь авторизован как администратор и находится в разделе управления справочниками
Основной сценарий	<ol style="list-style-type: none"> 1. Администратор переходит в раздел «Справочники» в интерфейсе системы. 2. Нажимает кнопку «Создать справочник». 3. Вводит название нового справочника и описание (при необходимости). 4. Сохраняет изменения. 5. Справочник появляется в общем списке с возможностью дальнейшего редактирования
Постусловие	Новый справочник добавлен в систему и доступен для заполнения и использования.

Администратор переходит в раздел «Справочники» в интерфейсе системы.

При создании справочника необходимо ввести название и описание. При сохранении изменения справочник появляется в общем списке с возможностью дальнейшего редактирования. Справочник добавляется в систему и доступен для заполнения и использования.

В таблице 5 показан сценарий, а также полное описание варианта использования «Выбрать вид страхования».

Таблица 5 – Сценарий и полное описание варианта использования «Выбрать вид страхования»

Название пункта	Описание
Наименование	Выбрать вид страхования
Краткое описание	Позволяет выбрать тип страхового продукта при создании документа
Основные действующие лица	Администратор
Второстепенные действующие лица	Нет
Предусловие	Пользователь открыл форму создания страхового документа
Основной сценарий	<ol style="list-style-type: none"> 1. Администратор переходит в форму создания документа. 2. В поле «Вид страхования» выбирает нужную опцию из выпадающего списка (например: ОСАГО, КАСКО, имущество, здоровье). 3. После выбора система автоматически подгружает связанные параметры (доступные тарифы, поля, коэффициенты и т.п.). 4. Администратор продолжает заполнение документа.
Постусловие	Вид страхования выбран, и соответствующие параметры подгружены в форму документа.

Таким образом были смоделированы функциональные возможности приложения.

2.2 Разработка архитектуры программного обеспечения

Проектирование архитектуры программного обеспечения является ключевым этапом создания автоматизированной информационной системы страховой компании.

Архитектура определяет логическую организацию системы, взаимодействие её компонентов, способы обмена данными и технические решения, направленные на достижение надежности, масштабируемости и гибкость.

Система предназначена для автоматизации ряда процессов, в том числе: оформление страховых договоров, расчета страховых премий, ведения

справочников, учета обращений по страховым случаям, а также генерации отчётности.

Исходя из требований к системе, была выбрана клиент-серверная архитектура, соответствующая современным стандартам построения информационных систем.

В качестве базовой модели принята трёхуровневая архитектура, включающая уровень представления, бизнес-логики и хранения данных.

Первый уровень, представления (frontend), реализован с использованием библиотеки React.js и отвечает за отображение интерфейса, обработку действий пользователя и отправку запросов к серверу.

Второй уровень, бизнес-логики (backend), реализован с помощью Python и фреймворка FastAPI, которые обеспечивают работу API, обработку бизнес-правил, валидацию данных и взаимодействие с базой данных.

Последний уровень хранения данных (database) – реализован на базе СУБД PostgreSQL. Отвечает за надёжное хранение всех сущностей: пользователей, клиентов, договоров, страховых событий и справочников.

Выбранная архитектура позволяет распределить ответственность между слоями, упростить тестирование, сопровождение и масштабирование системы.

При запуске клиентского приложения сервер запускается в виде дочернего процесса клиента. Последний, в свою очередь, взаимодействует с ним посредством REST API интерфейса.

Клиентское приложение во время работы производит сохранение файлов, а сервер сохраняет историю страхования, которая в последствии доступна пользователю в клиентском приложении.

Взаимодействие элементов архитектуры для работы автоматизированной информационной системы страховой компании можно представить с помощью диаграммы последовательности.

Диаграмма последовательности позволяет описывать последовательность работы в приложении, где определенные действия приводят к какому-либо результату.

На рисунке 7 представлена диаграмма последовательности взаимодействия компонентов приложения для автоматизации процессов внутри страховой компании.

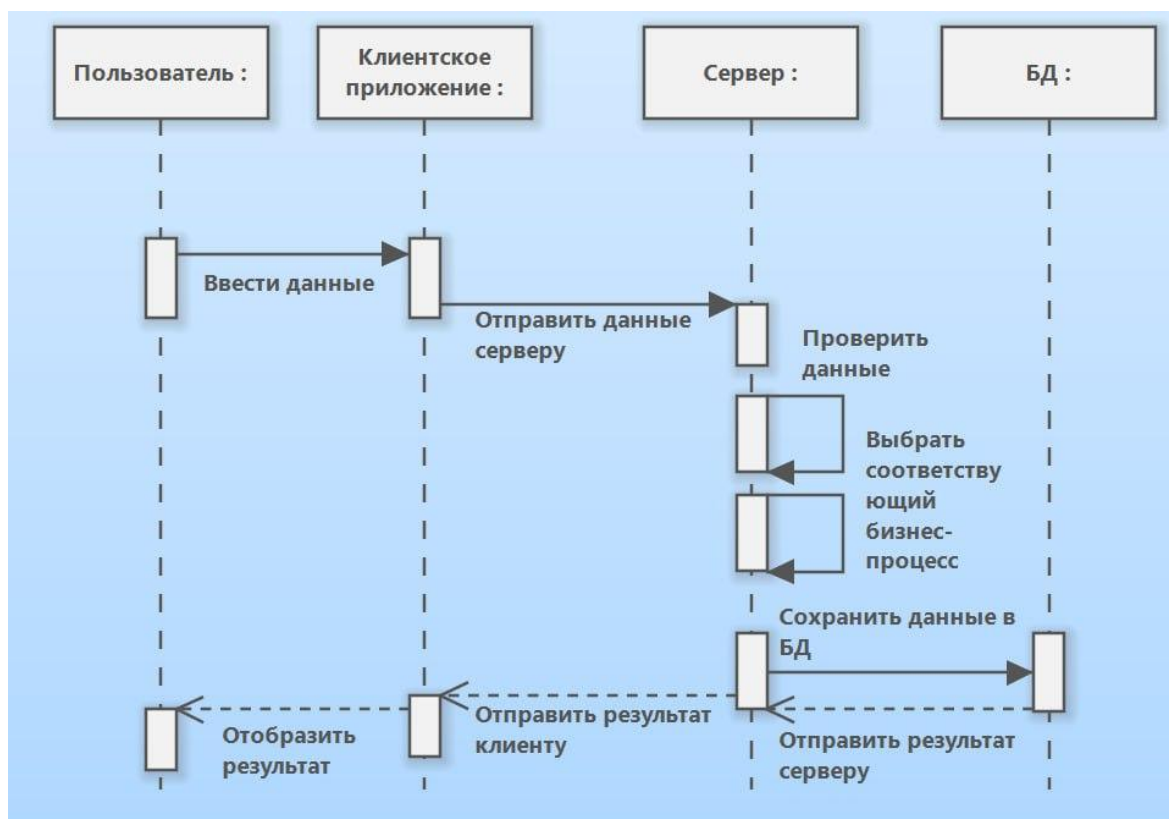


Рисунок 7 – Диаграмма последовательности взаимодействия компонентов приложения

Пользователь инициирует процесс создания страхового договора, вводя необходимые данные через клиентское приложение. Интерфейс, реализованный на базе React.js, сериализует введенную информацию в формате JSON и отправляет её на сервер через HTTP-запрос к API.

Сервер, реализованный на FastAPI, принимает запрос, выполняет валидацию данных и выбирает соответствующий бизнес-процесс: создание нового договора, расчет страховой премии или обновление данных клиента. Далее происходит взаимодействие с базой данных PostgreSQL, где сохраняются или обновляются сведения о клиенте, договоре и параметрах страхования.

После выполнения операций сервер формирует ответ, содержащий рассчитанные значения, уникальный номер договора и служебную информацию. Клиентское приложение получает ответ, десериализует данные и отображает результат пользователю в виде формы, таблицы или печатного шаблона.

Таким образом, взаимодействие компонентов системы построено по классической клиент-серверной архитектуре, с четким разделением ответственности между интерфейсом, логикой и хранилищем данных.

2.3 Прототипирование пользовательского интерфейса

Прототипирование пользовательского интерфейса (UI) является важнейшим этапом разработки, позволяющим на ранней стадии сформировать общее представление о внешнем виде, структуре и логике взаимодействия пользователей с системой. Это особенно актуально для автоматизированных информационных систем, где точность и удобство работы с данными критически важны.

Для системы страховой компании основное внимание уделено созданию интуитивно понятного и функционального интерфейса, позволяющего минимизировать количество действий для выполнения ключевых операций, таких как, оформление договора, расчета страховой премии, регистрации страхового случая и формирования отчетов.

При проектировании интерфейса применялись принципы модульности, адаптивности, доступности и интерактивности. Принцип модульности и повторяемости заключается в том, что компоненты используются по несколько раз в разных частях системы, могут содержать такие элементы как таблицы, формы, карточки. Принцип адаптивности затрагивает проектирование интерфейса и приносит адаптивность изображения на различных устройствах, так как в дальнейшем рассматривается мобильная версия сервиса.

Принцип доступности про использование четкой типографики, соблюдение иерархии и цветовых акцентов для быстрого восприятия информации пользователями.

Принцип интерактивности затрагивает обработку действий пользователя без перезагрузки страницы, благодаря использованию React.js и SPA-подходу.

2.4 Обоснование выбора средств разработки

2.4.1 Обоснование выбора средств автоматизации управления проектом

Github Actions – предоставляет собой web-приложение для управления разработкой, тестированием, развертыванием и сопровождением продукта.

На рисунке 8 показана главная страница с текущими проектами пользователя в Github Actions.

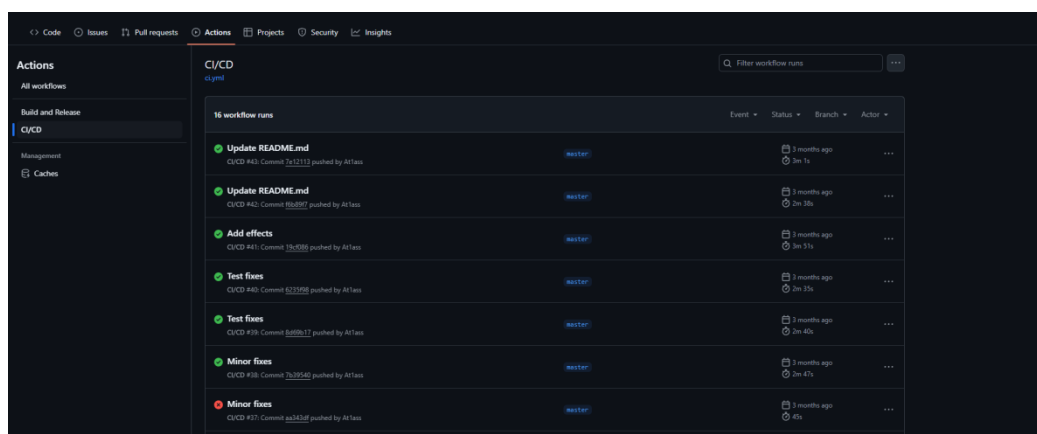


Рисунок 8 – Главная страница с проектами в Github Actions

Github Actions имеет следующие свойства:

- гибкость настройки;
- минимум зависимостей;
- репозиторий должен быть размещен;
- требует настройки, можно установить множество плагинов.

Таким образом, были выбраны средства для непрерывной интеграции и поставки с целью автоматизации и упрощения реализации средства для автоматизации работы страховой компании.

2.4.2 Обоснование выбора инструментальных средств разработки программного обеспечения

Для быстрого и качественного написания кода, необходимо выбрать наиболее подходящее инструментальное средство разработки программного обеспечения. В качестве таких продуктов были выбраны NeoVim с lsp сервером ruff для python и tsserver для React.

NeoVim – это современный текстовый редактор, являющийся улучшенной версией классического Vim. Он предоставляет гибкость и настраиваемость, что делает его удобным инструментом для профессиональных разработчиков. В отличие от традиционных IDE, NeoVim позволяет полностью адаптировать интерфейс и рабочие процессы под потребности конкретного проекта, сохраняя при этом высокую производительность и минимальное потребление ресурсов.

В связке с Language Server Protocol (LSP) с сервером `rust` NeoVim превращается в полноценную среду разработки. Благодаря LSP-серверу `ruff` обеспечивается дополнительная поддержка типов и более глубокий анализ кода, что особенно актуально при создании систем средней и высокой сложности, таких как автоматизированные информационные системы.

На рисунках 9 и 10 представлен интерфейс NeoVim.



Рисунок 9 – Интерфейс NeoVim

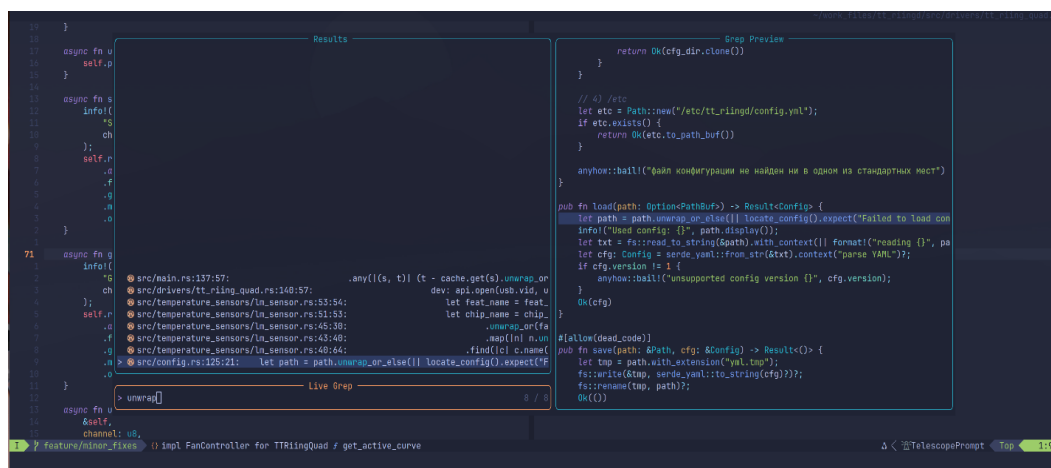


Рисунок 10 – Интерфейс NeoVim

Аналогами NeoVim являются: Visual Studio Code, Subline Text, Atom, Notepad ++.

NeoVim был выбран в качестве основного средства разработки, так как предоставляет наиболее полный функционал в плане написания кода и интеграции с GitHub, а также имеет удобный пользовательский интерфейс.

2.4.3 Обоснование выбора языковых средств разработки

Разработанное средство для автоматизации информационной системы страховой компании состоит из клиентской и серверной части.

Серверная часть была написана с использованием языка программирования Python, а также фреймворка FastAPI.

Python – язык программирования, появившийся в начале 90-х годов прошлого века, общего назначения, имеющий динамическую типизацию и автоматическое управление памятью.

Язык ориентирован на повышение читабельности кода, благодаря минимализму в синтаксисе. Популярен в анализе данных, машинном обучении, DevOps и веб-разработке, а также в разработке игр.

Для написания сервера использовался также фреймворк FastAPI, имеющий широкий функционал. Данный фреймворк позволяет быстрое написание HTTP API-серверов со встроенной валидацией, сериализацией и асинхронностью.

Клиентская часть была написана с использованием языка программирования TypeScript с использованием фреймворка Electron JS и библиотеки React JS.

TypeScript является языком программирования, представляющим собой строго типизированное надмножество JavaScript, добавляющее статическую типизацию и дополнительные возможности объектно-ориентированного программирования.

TypeScript компилируется в чистый JavaScript, полностью совместим с существующими браузерами и JavaScript-фреймворками. Стандарты TypeScript определяются компанией Microsoft и регулярно обновляются вместе с развитием JavaScript.

Electron JS – представляет собой фреймворк для разработки приложений под ПК, с использованием HTML, CSS, JavaScript.

Такие приложения являются кроссплатформенными и могут работать на Windows, Mac, Linux. Разработчиком Electron JS является GitHub Inc.

На рисунке 11 представлен официальный сайт Electron JS.

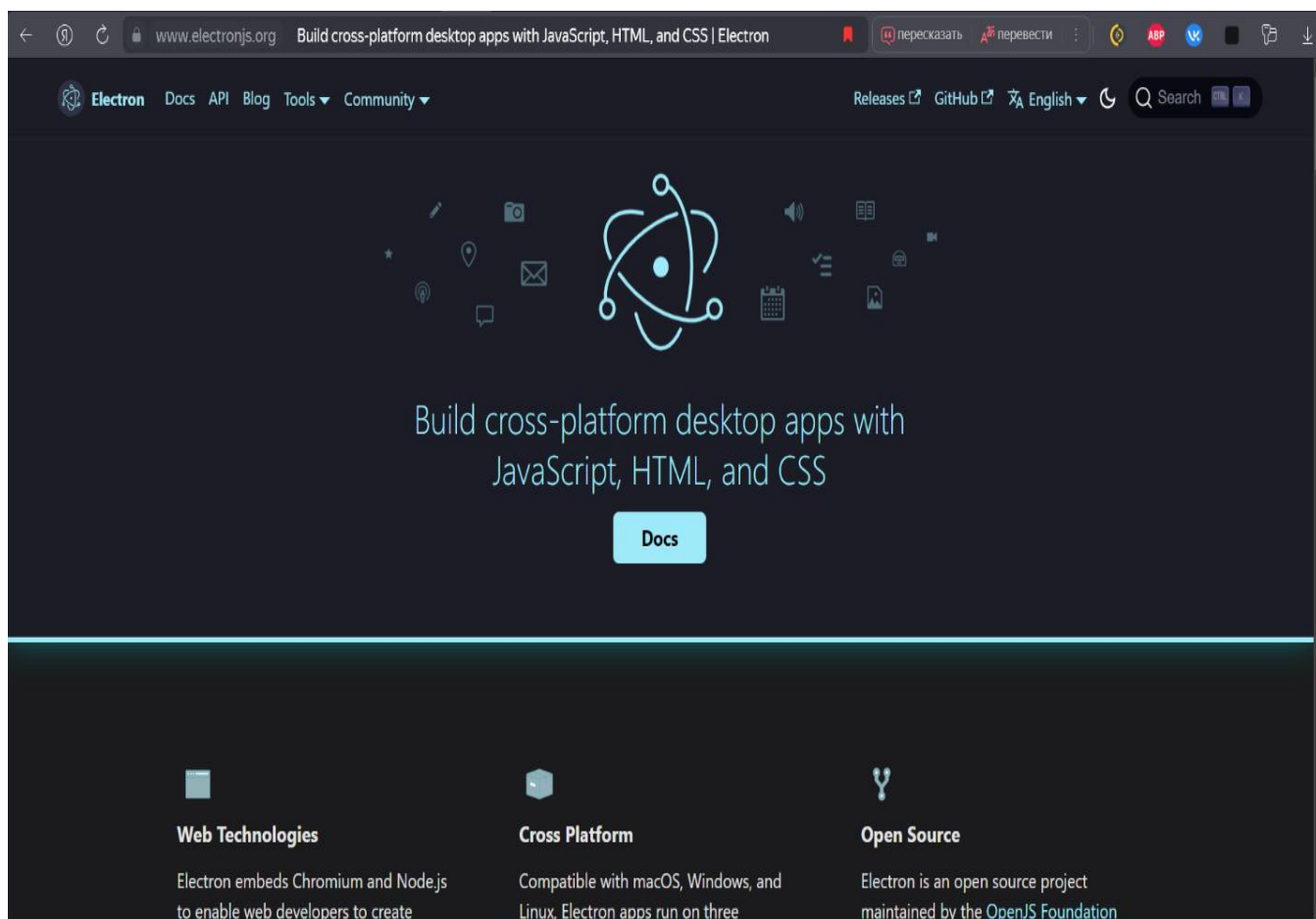


Рисунок 11 – Официальный сайт Electron JS

В основе работы фреймворка лежит использование браузерного движка Chromium и Node JS, которые объединены в единую среду, позволяющую писать настольные приложения с использованием веб-технологий.

Для быстрого и качественного написания пользовательского интерфейса использовалась библиотека React JS, которая была интегрирована в Electron JS приложение.

React JS – библиотека с открытым исходным кодом, используемая для разработки веб-интерфейсов.

Широко используется при реализации web-приложений на ряду с другими популярными библиотеками, такими как Angular JS, Vue JS.

На рисунке 12 представлен официальный сайт React JS.

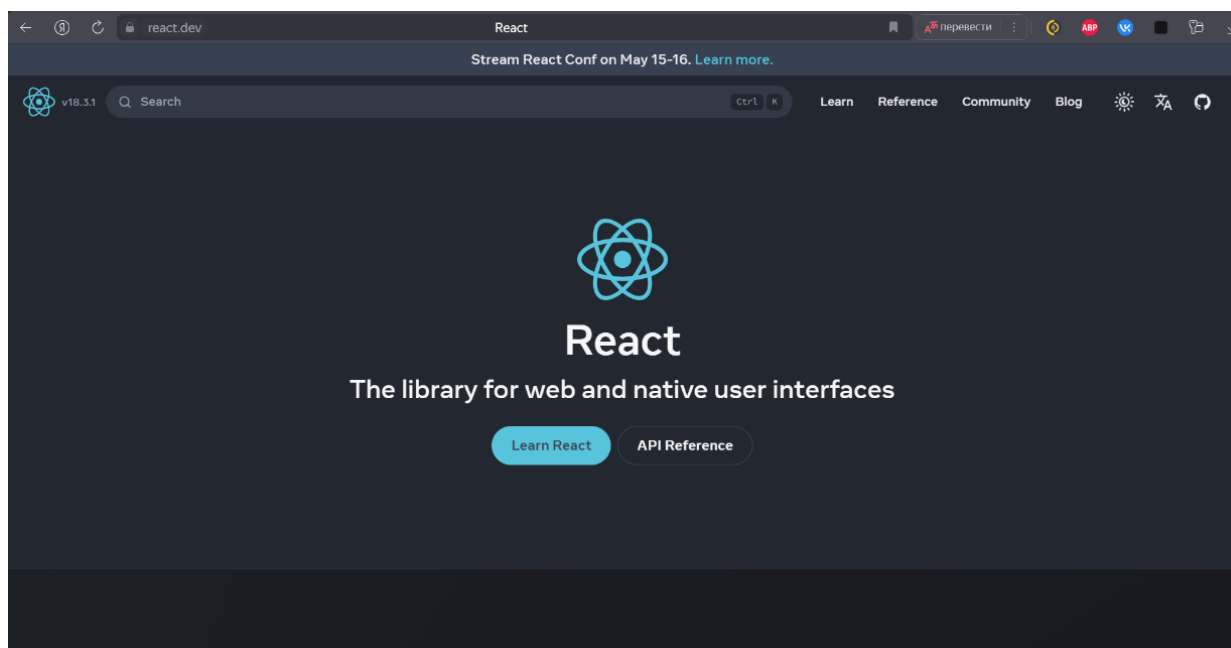


Рисунок 12 – Официальный сайт React JS

Основой React является создание компонент, которые отображаются в браузере в виде блоков HTML кода. При написании компонент можно использовать два стиля написания: функциональный и объектно-ориентированный. Последний используется обычно, если разрабатываемое приложение создается с помощью языка JavaScript. Также в приложении на React можно добавлять интерактивность, различные стили поведения объектов на странице. React JS имеет большое сообщество разработчиков.

Средство для автоматизации работы страховой компании на исходные заготовки должно быть кроссплатформенным, включая возможность работы на Android, iOS, в виде настольного приложения под Windows, Linux, Mac OS, а также в виде web-приложения. Использование Electron JS позволит обеспечить кроссплатформенность. Библиотека React JS имеет расширение библиотеки React Native, позволяющее писать мобильные приложения под Android и iOS. В свою очередь, она будет использоваться в web-приложении.

3.Реализация приложения для работы страховой компании

3.1 Разработка клиентской части приложения

3.1.1 Разработка пользовательского интерфейса клиентского приложения

На рисунке 13 пользовательский интерфейс, который был разработан исходя из прототипа в разделе 2.3.

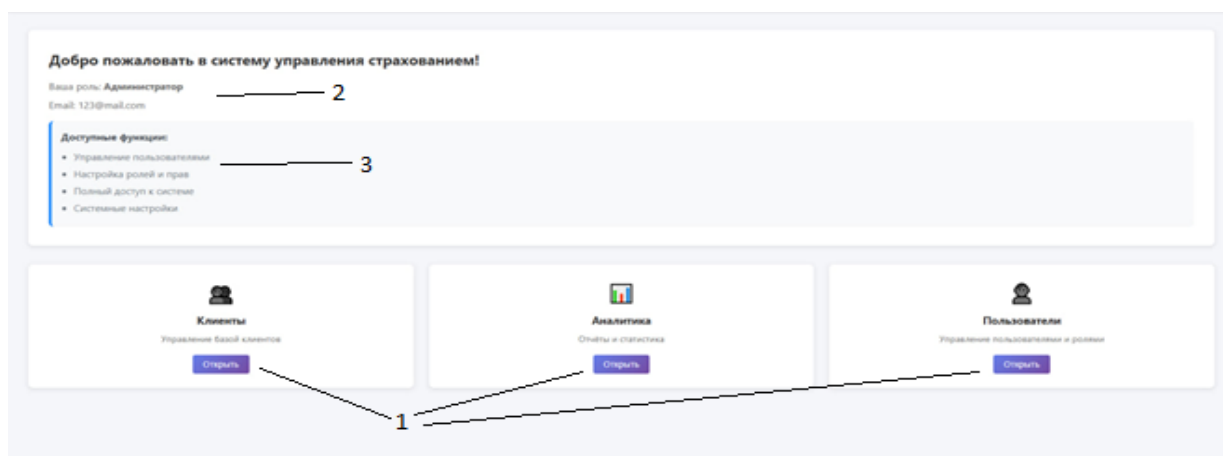


Рисунок 13 – Главное окно администратора

На рисунке расположены элементы:

1. кнопки;
2. отображение роли;
3. доступный функционал роли;

На рисунке 14 показана Страница управления клиентами.

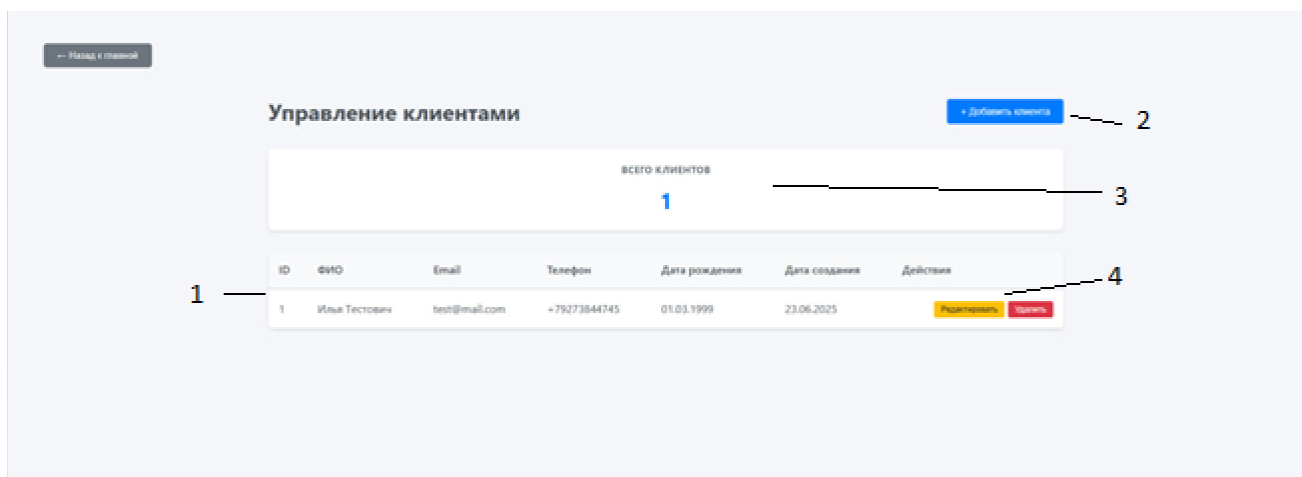


Рисунок 14 – Страница управления клиентами страховой компании

На рисунке показаны элементы:

1. Поле со списком клиентов;
2. Кнопка добавления нового клиента;
3. Общее количество клиентов;
4. Кнопки редактирования и удаления клиента;

На рисунке 15 показана главная страница аналитики и отчетов страховой компании.

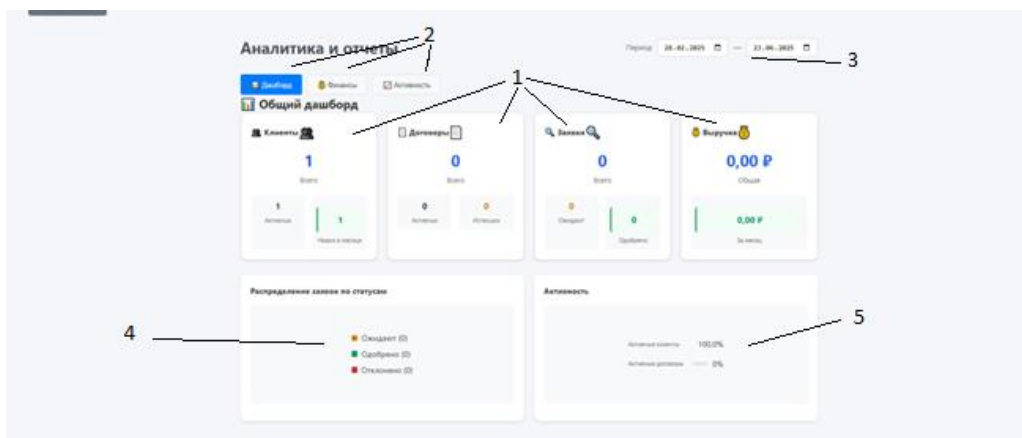


Рисунок 15 – Страница с аналитикой

На рисунке показаны элементы:

1. Счетчики общего количества: клиентов, договоров, заявок, выручки;
2. Кнопки;
3. Выбор периода для аналитики;
4. Поле распределения заявок;
5. Поле активности клиентов.

На рисунке 16 показано окно управления пользователями.

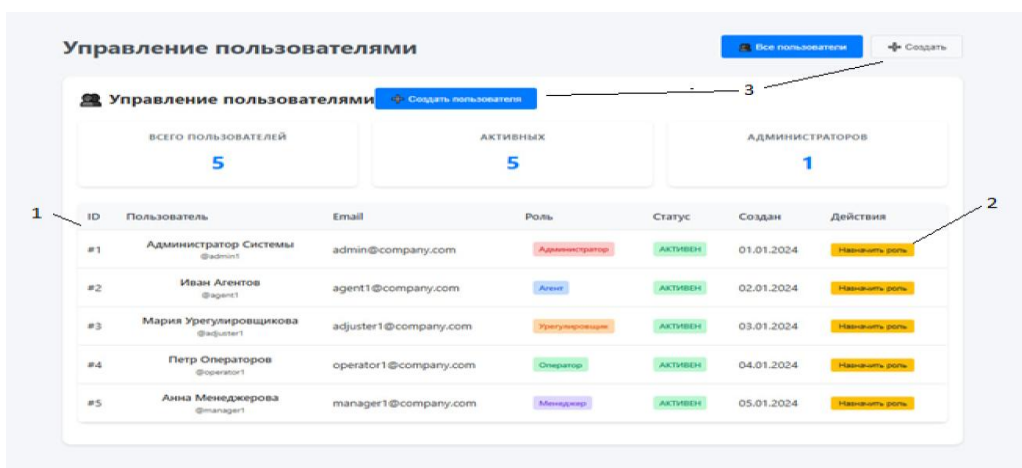


Рисунок 16 – Окно управления пользователями

На рисунке показаны следующие элементы:

1. Список всех пользователей системы;
2. кнопка назначения роли пользователю;
3. кнопки создания нового пользователя;

На рисунке 17 показана страница управления страховыми случаями.

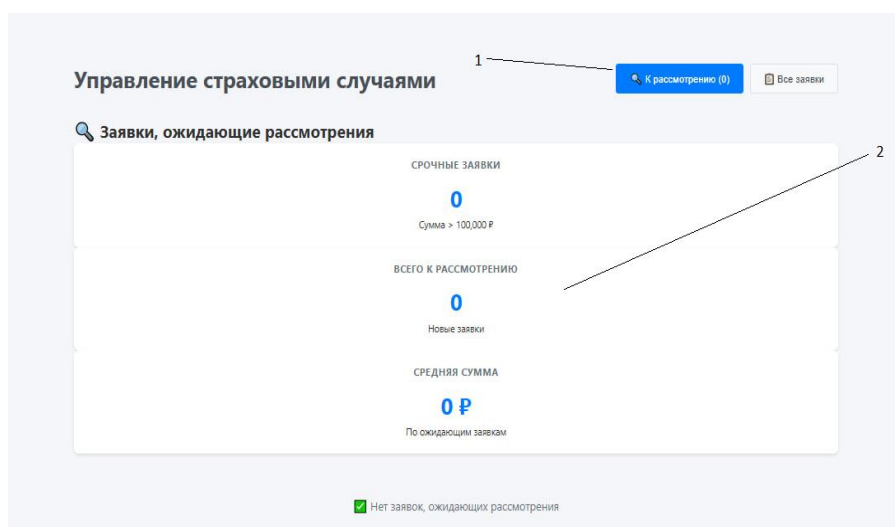


Рисунок 17 – управление страховыми случаями

1. Кнопки просмотра всех заявок и представленных к рассмотрению
2. Список заявок, представленных к рассмотрению

На рисунке 18 показана страница рабочего места оператора

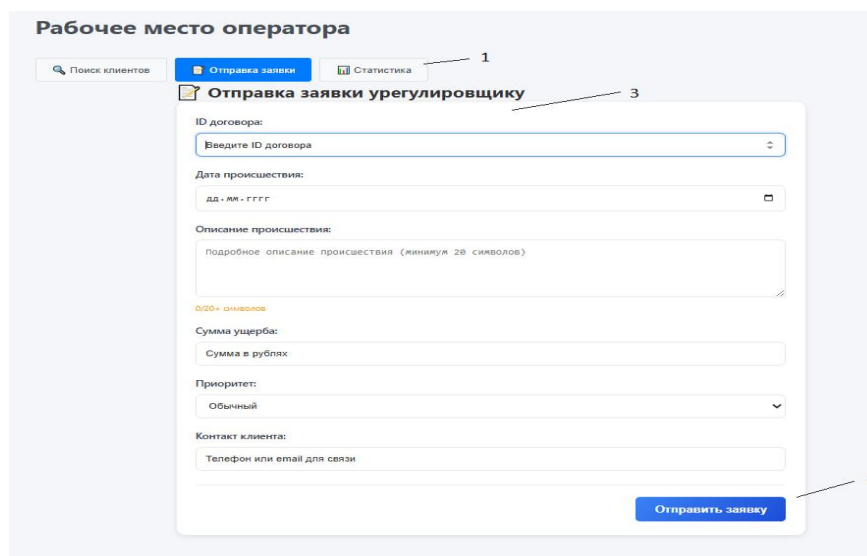


Рисунок 18 – Рабочее место оператора

- 1 – Кнопки выбора статистики, отправки заявок, поиска клиентов
- 2 – Кнопка отправки заявки урегулировщику
- 3 – форма заявки

Таким образом был разработан пользовательский интерфейс для приложения работы страховой компании.

3.2 Разработка программного обеспечения клиентского приложения

Программное обеспечение клиентского приложения разрабатывалось на языке программирования JavaScript с использованием Electron JS и React JS.

Компоненты React JS могут писаться в функциональном стиле или в стиле объектно-ориентированного программирования. Второй способ подходит, если разработка ведется на языке TypeScript.

В работе клиентское приложение писалось на языке JavaScript в функциональном стиле.

На рисунке 19 показана диаграмма компонентов клиентского приложения.

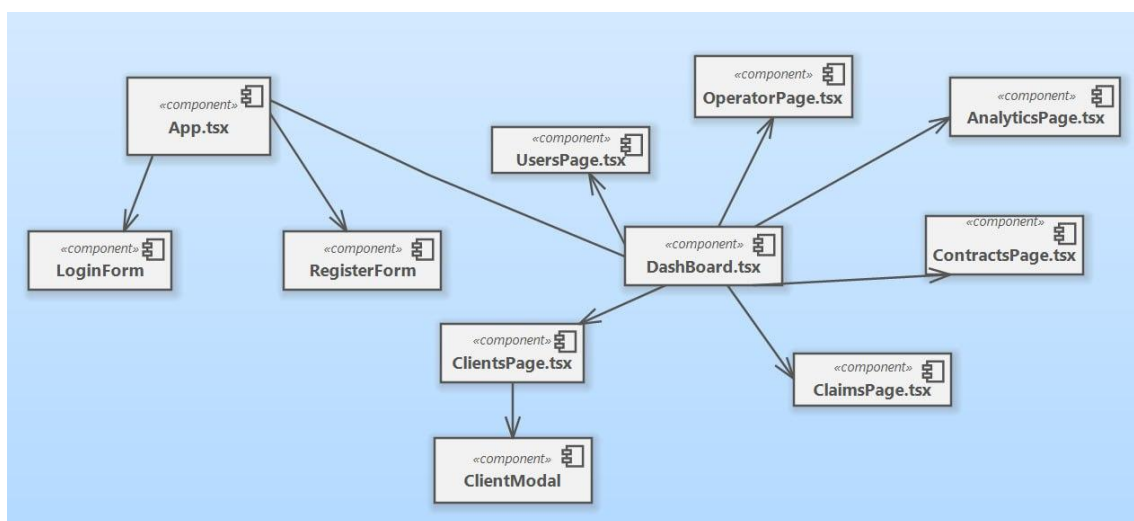


Рисунок 19 – Диаграмма компонентов клиентского приложения

Приведем описание каждого компонента:

- компоненты LoginForm/RegisterForm – формы для входа и регистрации;
- компонент DashBoard.tsx – основная панель управления после входа, которая отображает различные страницы;
- компоненты ClientsPage/ContractsPage/ClaimsPage/AnalyticsPage/UserPage/OperatorPage являются основными разделами(страницами) приложения;

- компонент `App.tsx` – точка входа, которая управляет аутентификацией;
- компонент `ClientModal` – элемент, который используется для добавления или редактирования клиентов.

Код главного компонента `App.tsx` представлен в листинге 1.

Листинг 1 – Код главного компонента `App.js`

```
import React, { useState, useEffect } from 'react';
import './App.css';
import { AuthService } from '@services/authService';
import { LoginForm } from '@components/LoginForm';
import { RegisterForm } from '@components/RegisterForm';
import { Dashboard } from '@components/Dashboard';

interface User {
  id: string;
  username: string;
  email: string;
  full_name: string;
  role: string;
}

const App: React.FC = () => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [user, setUser] = useState<User | null>(null);
  const [authMode, setAuthMode] = useState<'login' | 'register'>('login');
  const [loading, setLoading] = useState(true);

  const authService = new AuthService();

  useEffect(() => {
    checkAuth();
  }, []);

  const checkAuth = async () => {
    try {
      const isValid = await authService.verifyToken();
      if (isValid) {
        const userData = authService.getCurrentUser();
        setUser(userData);
        setIsAuthenticated(true);
      }
    } catch (error) {
      console.error('Auth check failed:', error);
    } finally {
      setLoading(false);
    }
  };

  const handleLogin = async (credentials: { username: string; password: string }) => {
```

```

try {
  const response = await authService.login(credentials.username,
  credentials.password);
  setUser(response.user);
  setIsAuthenticated(true);
} catch (error) {
  throw error;
}
};

```

```

const handleRegister = async (userData: any) => {
  try {
    await authService.register(userData);

```

После успешной регистрации переключаемся на логин:

```

    setAuthMode('login');
  } catch (error) {
    throw error;
  }
};

```

```

const handleLogout = async () => {
  try {
    await authService.logout();
    setIsAuthenticated(false);
    setUser(null);
  } catch (error) {
    console.error('Logout failed:', error);
  }
};

```

```

if (loading) {
  return (
    <div className="loading-container">
      <div className="spinner"></div>
      <p>Загрузка...</p>
    </div>
  );
}

```

```

if (isAuthenticated && user) {
  return <Dashboard user={user} onLogout={handleLogout} />;
}

```

```

return (
  <div className="auth-container">
    <div className="auth-card">
      <div className="logo">IMS</div>
      <h1>Insurance Management System</h1>
      <p className="subtitle">
        {authMode === 'login'
          ? 'Войдите в систему для продолжения'
          : 'Создайте новый аккаунт'
        }
      </p>

```

```

<div className="auth-toggle">
  <button
    className={`toggle-btn ${authMode === 'login' ? 'active' : ''}`}
    onClick={() => setAuthMode('login')}
  >
    Вход
  </button>
  <button
    className={`toggle-btn ${authMode === 'register' ? 'active' : ''}`}
    onClick={() => setAuthMode('register')}
  >
    Регистрация
  </button>
</div>

{authMode === 'login' ? (
  <LoginForm onLogin={handleLogin} />
) : (
  <RegisterForm onRegister={handleRegister} />
)}
</div>
</div>
);
};

```

```
export default App;
```

Таким образом была разработана клиентская часть автоматизированной информационной системы страховой компании.

3.3 Разработка серверной части приложения

Серверная часть была реализована на Python с использованием фреймворка Fast API. В таблице 6 представлено описание REST API интерфейса

Таблица 6 – API клиентского модуля

Название функции	Строка адреса	Метод REST API	Описание
get_clients	/	GET	Получение списка клиентов
create_clients	/	POST	Создание нового клиента
get_client	/ {client_id}	GET	Получение информации о клиенте по ID
update_client	/ {client_id}	PUT	Обновление данных клиента
delete_client	/ {client_id}	DELETE	Удаление клиента

В таблице 6 представлена работа API клиентского модуля

Таблица 7 – API модуля договора

Название функции	Строка адреса	Метод REST API	Описание
get_contracts	/	GET	Получение списка договора
calculate_premium	/calculate	POST	Расчет страховой премии
create_contract	/	POST	Создание нового договора страхования
get_contract	/ {contract_id}	GET	Получение информации о договоре по ID
update_contract	/ {contract_id}	PUT	Обновление данных договора
activate_contract	/ {contract_id} / activate	POST	Активация договора

В таблице 7 представлена работа API модуля договора

Таблица 8 – API модуля заявок

Название функции	Строка адреса	Метод REST API	Описание
get_claims	/	GET	Получение списка страховых случаев
get_pendint_claims	/pending	GET	Получение заявок, ожидающих рассмотрения
create_claim	/	POST	Создание новой заявки
get_claim	/ {claim_id}	GET	Получение детальной информации о заявке
make_claim_decision	/ {claim_id} / decision	PUT	Принятие решения по страховому случаю
update_claim	/ {claim_id}	PUT	Обновление информации по заявке
submit_claim_to_adjuster	/submit	POST	Отправка заявки на рассмотрение

В таблице 8 представлена работа API модуля заявок

Таблица 9 – API модуля продукта

Название функции	Строка адреса	Метод REST API	Описание
get_products	/	GET	Получение всех активных страховых продуктов
create_product	/	POST	Создание нового страхового продукта
update_product	/ {product_id}	PUT	Обновление данных страхового продукта
delete_product	/ {product_id}	DELETE	Деактивация страхового продукта

В таблице 9 представлена работа API модуля продукта

Таблица 10 – API модуля пользователей

Название функции	Строка адреса	Метод REST API	Описание
get_users	/	GET	Получение списка пользователей
create_user	/	POST	Создание нового пользователя
get_user	/ {user_id}	GET	Получение данных пользователя по ID
update_user	/ {user_id}	PUT	Обновление данных пользователя
get_roles_management	/admin/roles	GET	Получение данных для управления ролями
assign_user_role	admin/roles/assign	POST	Назначение роли пользователю
delete_user	/ {user_id}	DELETE	Удаление пользователя
get_audit_logs	/admin/audit	GET	Получение логов аудита

В таблице 10 представлена работа API модуля пользователей

Таблица 11 – API модуля аналитики

Название функции	Строка адреса	Метод REST API	Описание
get_dashboard_data	/dashboard	GET	Получение данных для аналитической панели
get_finance_report	/reports/finance	GET	Генерация финансового отчета
get_activity_report	/reports/activity	GET	Генерация отчета по активности

В таблице 11 представлена работа API модуля аналитики

Все методы являются асинхронными, то есть не требует обратного ответа от клиентского приложения.

В листинге 2 представлен код метода products REST API интерфейса серверного приложения.

Листинг 2 – Код REST API интерфейса приложения

```

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from typing import List
from app.utils.auth import get_current_user, require_roles
from app.db.database import get_db
from app.db.models import Client
from app.schemas.client import ClientCreate, ClientUpdate, Client as ClientSchema, ClientList
from app.functions.client_service import ClientService
router = APIRouter()
@router.get("/", response_model=ClientList)
async def get_clients(
    skip: int = 0,
    limit: int = 100,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("agent", "operator", "admin"))

```

```

):
client_service = ClientService(db)
clients, total = client_service.get_clients(skip=skip, limit=limit)
return ClientList(
clients=clients,
total=total,
skip=skip,
limit=limit
)
@router.post("/", response_model=ClientSchema)
async def create_client(
client_data: ClientCreate,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("agent", "operator",
"admin")))
):

```

Далее описывается создание нового пользователя:

```

client_service = ClientService(db)

# Check if client with this email already exists
existing_client =
client_service.get_client_by_email(client_data.email)
if existing_client:
raise HTTPException(
status_code=status.HTTP_400_BAD_REQUEST,
detail="Client with this email already exists"
)

client = client_service.create_client(client_data,
created_by=current_user.get("user_id"))
return client

@router.get("/{client_id}", response_model=ClientSchema)
async def get_client(
client_id: int,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("agent", "operator",
"admin")))
):
"""Get client by ID"""
client_service = ClientService(db)
client = client_service.get_client(client_id)

if not client:
raise HTTPException(
status_code=status.HTTP_404_NOT_FOUND,
detail="Client not found"
)
return client

@router.put("/{client_id}", response_model=ClientSchema)
async def update_client(
client_id: int,
client_data: ClientUpdate,
db: Session = Depends(get_db),

```

```
current_user: dict = Depends(require_roles("agent", "operator",
"admin"))
):
```

Далее описывается обновление информации о клиенте:

```
client_service = ClientService(db)
# Check if client exists
existing_client = client_service.get_client(client_id)
if not existing_client:
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Client not found"
    )
if client_data.email and client_data.email != existing_client.email:
    email_client = client_service.get_client_by_email(client_data.email)
    if email_client and email_client.id != client_id:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Client with this email already exists"
        )

client = client_service.update_client(client_id, client_data)
return client
@router.delete("/{client_id}")
async def delete_client(
    client_id: int,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("admin"))
):
    """Delete client (admin only)"""
    client_service = ClientService(db)
```

В этом фрагменте описывается проверка наличия клиента:

```
existing_client = client_service.get_client(client_id)
if not existing_client:
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Client not found"
    )
if client_service.has_active_contracts(client_id):
    raise HTTPException(
        status_code=status.HTTP_400_BAD_REQUEST,
        detail="Cannot delete client with active contracts"
    )
client_service.delete_client(client_id)
return {"message": "Client deleted successfully"}
```

Для передачи данных между клиентом и сервером необходимо сформировать json строку с полями.

Сервер получает от клиента эти данные и преобразует их для передачи в функции работы компании. Для этого были реализованы модели, содержащие необходимые типы данных (листинг 3).

Листинг 3 – Код моделей, содержащие необходимые типы данных

```

from pydantic import BaseModel, EmailStr
from typing import Optional
from datetime import date, datetime
class ClientBase(BaseModel):
    first_name: str
    last_name: str
    email: EmailStr
    phone: Optional[str] = None
    address: Optional[str] = None
    date_of_birth: Optional[date] = None
    identification_number: Optional[str] = None
class ClientCreate(ClientBase):
    pass
class ClientUpdate(BaseModel):
    first_name: Optional[str] = None
    last_name: Optional[str] = None
    email: Optional[EmailStr] = None
    phone: Optional[str] = None
    address: Optional[str] = None
    date_of_birth: Optional[date] = None
    identification_number: Optional[str] = None
class Client(ClientBase):
    id: int
    created_at: datetime
    updated_at: Optional[datetime] = None
    created_by: Optional[int] = None
class Config:
    from_attributes = True
class ClientList(BaseModel):
    clients: list[Client]
    total: int
    skip: int
    limit: int

```

`ClientCreate` будет содержать данные, необходимые для сощдания нового клиента, такие как имя, почту, номер телефона.

`ClientUpdate` будет содержать данные для обновления информации о существующем клиенте. `Client` будет содержать в себе всю информацию о клиенте, включая его уникальный номер(`id`), контактные данные.

`Clientlist` будет содержать в себе список всех существующих клиентов.

Таким образом была реализована серверная часть приложения работы страховой компании.

3.4 Разработка базы данных страховой компании

Для хранения данных о документах, пользователях и клиентах была развернута база данных информационной системы страховой компании.

Концептуальная модель базы данных представлена на рисунке 20.

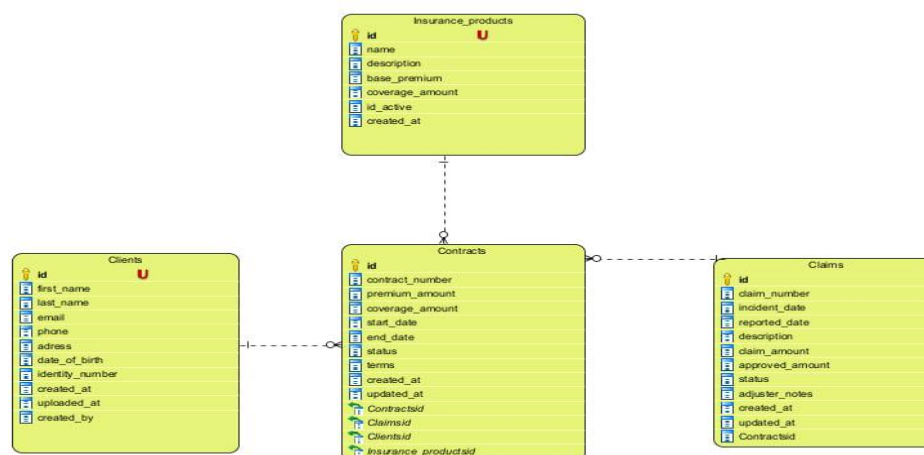


Рисунок 20 – Концептуальная модель базы данных

Концептуальная модель базы данных страховой компании обеспечивает общее понимание и помогает четко определить сущности и связи.

Логическая модель базы данных представлена на рисунке 21.

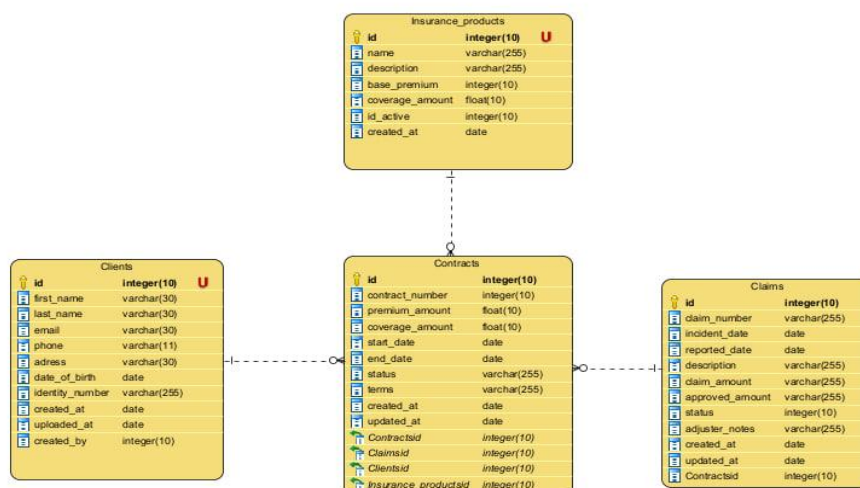


Рисунок 21 – Логическая модель базы данных

Логическая модель базы данных страховой компании способствует определению хранимых типов данных и служит основой для разработки физической модели

Физическая модель развернутой базы данных представлена на рисунке 22.

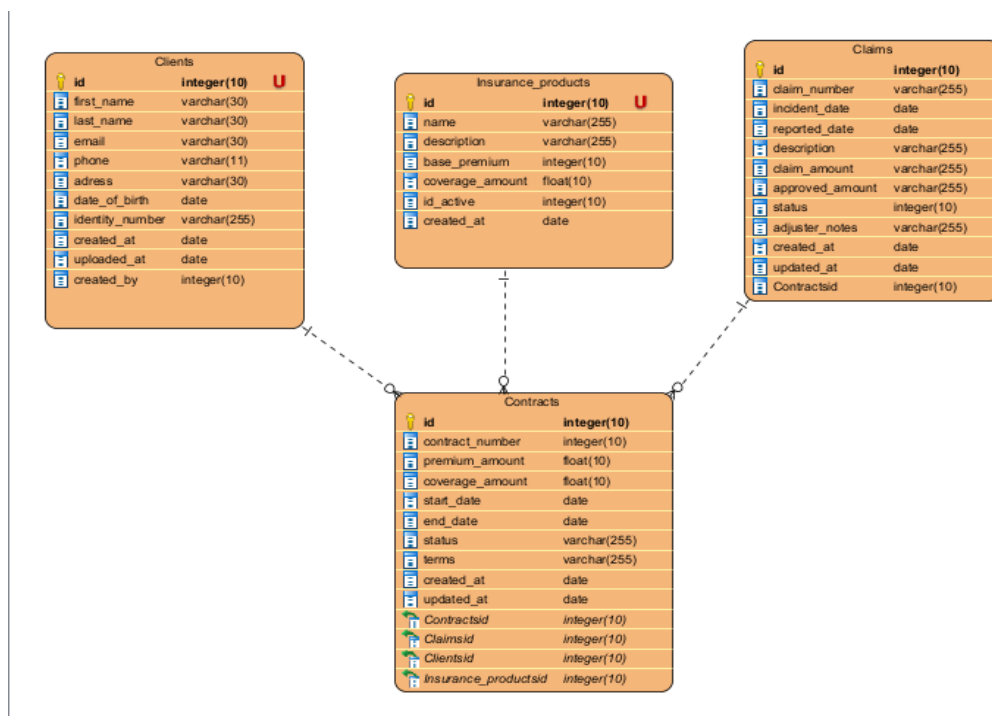


Рисунок 22 – Физическая модель базы данных

Для создания базы данных, используемой в работе веб-приложения страховой компании, была выбрана PostgreSQL – реляционная СУБД, которая обеспечивает надежность, согласованность связей между сущностями, может работать под любой операционной системой. Гарантирует целостность данных и хорошую производительность при одновременной работе многих пользователей.

Ниже представлены таблицы описания сущностей:

Таблица 12 – Описание сущностей клиентов

Атрибут	Тип данных	Ограничения	Описание
Id	Integer	PK	Уникальный идентификатор клиента
First_name	Varchar(255)	Not null	Имя клиента
Last_name	Varchar(255)	Not null	Фамилия клиента
Phone	Varchar(11)	Unique	Номер телефона
Email	Varchar(255)	Not null	Электронная почта
Date_of_birth	Date		Дата рождения
Gender	Varchar(25)		Пол клиента
Created_at	Date	Not null	Дата создания записи
Updated_at	Date		Дата последнего обновления

Таблица 13 – Таблица сущностей страховых продуктов

Атрибут	Тип данных	Ограничения	Описание
Id	Integer(10)	PK	Уникальный идентификатор продукта
Name	Varchar(255)	Not null	Название страхового продукта
Description	Varchar(255)		Описание продукта
Base_premium	Integer(10)	Not null	Базовая страховая премия
Coverage_amount	Float(10)	Not null	Страховая сумма
Is_active	Integer(10)		Статус активности
Created_at	Date	Not null	Дата создания продукта

Таблица 14 – сущности договора страхования

Атрибут	Тип данных	Ограничения	Описание
Id	Integer(10)	PK	Уникальный идентификатор договора
Contract_number	Varchar(255)	Not null	Номер договора
Premium_amount	Float(10)	Not null	Размер страховой премии
Coverage_amount	Float(10)	Not null	Страховая сумма
Start_date	Date	Not null	Дата начала действия
End_date	Date	Not null	Дата окончания действия
Status	Varchar(255)	Not null	Статус договора
Terms	Varchar(255)		Условия договора
Created_at	Date	Not null	Дата создания
Updated_at	Date		Дата обновления
Client_id	Integer(10)	FK	Идентификатор клиента
Owner_id	Integer(10)	FK	Идентификатор агента
Insurance_product_id	Integer(10)	FK	Идентификатор страхового продукта

Таблица 15 – описание сущностей страховых случаев

Атрибут	Тип данных	Ограничения	Описание
Id	Integer(10)	PK	Уникальный идентификатор заявки
Contract_number	Varchar(255)	Not null	Номер договора
Incident_date	Date	Not null	Дата страхового случая
Reported_date	Date	Not null	Дата подачи заявки
Description	Varchar(255)	Not null	Описание страхового случая
Claim_amount	Varchar(255)	Not null	Заявленная сумма
Approved_amount	Varchar(255)		Одобренная сумма
Status	Integer(10)	Not null	Статус заявки
Adjuster_notes	Varchar(255)		Примечания урегулировщика
Created_at	Date	Not null	Дата создания
Updated_at	Date		Дата обновления
Contract_id	Integer(10)	FK	Идентификатор договора

4. Контроль качества программного обеспечения автоматизированной информационной системы страховой компании

4.1. Планирование процесса контроля качества программного обеспечения

4.1.1 Мероприятия по контролю качества программного обеспечения

В ходе разработки автоматизированной информационной системы страховой компании были выполнены мероприятия по контролю качества разработанного программного обеспечения. Было проведено тестирование и оценка трудозатрат кода

Тестирование приложения – значимая часть реализации программного обеспечения. В процессе тестирования выявляются ошибки и проверяется корректность.

Существует несколько моделей тестирования: функциональное тестирование, нефункциональное тестирование.

Функциональное тестирование состоит из проверки функций приложения, которые уже готовы, но могут показывать некорректную работу в разных вариантах поведения приложения.

К нефункциональному тестированию можно отнести:

- юнит-тестирование;
- тестирование с помощью интеграционных тестов;
- нагрузочное тестирование;
- тестирование пользовательского интерфейса;
- конфигурационное тестирование;
- тестирование с помощью автотестов.

В качестве моделей тестирования для программного обеспечения автоматизированной информационной системы страховой компании были выбраны юнит-тесты и тестирование с помощью интеграционных тестов.

Оценка качества кода – этап контроля качества программного обеспечения, которое позволяет произвести анализ программного кода с целью

выявления его недостатков с возможностью последующего рефакторинга. В

качестве метрик программного кода была выбрана метрика: SLOC.

4.2 Обоснование выбора модели тестирования

Для тестирования автоматизированной информационной системы страховой компании было выбрано юнит-тестирование и сквозное тестирование.

Юнит-тестирование – это вид тестирования, который проверяет работоспособность отдельных модулей, процессов или частей кода приложения. Оно позволяет быстро исправить ошибки при обновлении программного обеспечения.

Преимущества использования модели юнит-тестирования:

- простота;
- параллельная разработка;
- информативность;
- имеется возможность повторного использования.

Недостатки использования системы юнит-тестирования:

- модульное тестирование не гарантирует, что будут найдены все ошибки;
- применяется к изолированным фрагментам кода;
- не способно выявить системные ошибки продукта в целом.

Сквозное тестирование – вид тестирования, который проверяет всё целиком.

Оно проверяет все ПО на предмет зависимостей, целостности данных и возможности подключения к другим системам, интерфейсам и базам данных для имитации полной производственной среды, пакетную обработку данных из различных систем восходящего или нисходящего потока в дополнение к системе программы.

Основная цель сквозного тестирования – определить системные зависимости, чтобы убедиться в целостности данных и базами данных для обеспечения полной производительности.

Преимущества сквозного тестирования:

- комплексное тестирование;
- упрощённый процесс тестирования;
- реалистичные сценарии;

- тестирование, ориентированное на клиента;
- снижение будущих рисков;
- повышение уверенности в качестве продукта;
- раннее обнаружение ошибок.

Недостатки сквозного тестирования:

- трудоемкость;
- дороговизна;
- ограниченный охват;
- проблемы, которые трудно изолировать;
- обслуживание.

Таким образом, выбранные модели тестирования обеспечили проверку работы приложения страховой компании.

4.3 Обоснование выбора метрик программного кода

Применение метрик позволяет оценить разные аспекты работы программного обеспечения, такие как качество, сложность, надежность, трудозатратность и объемы проекта

SLOC – подсчет количества строк кода, является одной из самых простых метрик. Данная метрика была разработана для оценки труда по проекту.

В качестве основной метрики была выбрана метрика SLOC, так как она позволит оценить объем кода.

4.4 Тестирование программных средств

4.4.1 Тестирование с помощью юнит-тестов

Для тестирования серверной части приложения были реализованы юнит-тесты. В таблице 16 представлен список юнит-тестов.

Данные тесты были написаны на Python с помощью библиоте.

Таблица 16 – Юнит тесты серверной части приложения

Название юнит-теста	Описание
test_create_client_valid_data	Тестируем создание клиента с корректными входными данными
test_get_client_by_id_exists	Тестируем получение клиента по id
test_get_client_by_id_not_exists	Тестируем получение несуществующего клиента по id
test_make_decision_approve	Тестируем одобрение заявки урегулировщиком
test_make_decision_reject	Тестируем отклонение заявки урегулировщиком
test_create_user_success	Тестируем успешное создание пользователя
test_get_users_list	Тестируем получение списка пользователей системы

Данные тесты были нацелены на проверку бизнес-логики страхового приложения

Можно сделать вывод, что серверная часть приложения покрыта тестами на 50 процентов, что является средним результатом.

Общий результат выполнения юнит-тестов представлен на рисунке 23.

```

$ python test_unit_demo.py |
| ЮНИТ-ТЕСТЫ ДЛЯ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ УПРАВЛЕНИЯ СТРАХОВАНИЕМ |
|
|   [✓] Создание клиента с корректными данными - ПРОЙДЕН |
|   [✓] Получение существующего клиента - ПРОЙДЕН |
|   [✓] Расчет страховой премии - ПРОЙДЕН |
|   [✓] Активация договора - ПРОЙДЕН |
|   🏆 ВСЕ ЮНИТ-ТЕСТЫ ПРОЙДЕНЫ УСПЕШНО! |
|   Процент успешности: 100.0% |

```

Рисунок 23 – Результат выполнения юнит-тестов

В целом по компонентам показано достаточно низкое покрытие (около 20%), но основные компоненты, для которых написаны юнит-тесты имеют хорошее покрытие.

Остальные компоненты были протестированы помощью автоматического сквозного тестирования.

4.4.2 Автоматическое сквозное тестирование

Для тестирования клиентской части приложения и проверки её взаимодействия с серверной частью были написаны автотесты на языке python.

Таблица 17 – Спецификации автотестов

Название	Описание
Setup_test_users	Проверяет создание тестовых пользователей для всех ролей в системе. Проверяет доступность сервисов и возможность авторизации
Test_agent_role	Проверяет функционал роли агент: управление клиентами, управление договорами
Test_adjuter_role	Проверяет функционал роли урегуливщик: просмотр заявок на урегулирование, получение заявок, принятие решения по одобрению заявки
Test_operator_role	Проверяет функционал роли оператор: просмотр клиентов, создание заявок, отправка заявок, поиск и фильтрация данных.
Test_manager_role	Проверяет функционал роли менеджер: доступ к аналитике, генерация отчетов, отчеты активности.
Test_admin_role	Проверяет функционал роли администратор: управление пользователями, управление ролями, назначение ролей, полный доступ к аналитике.

В процессе выполнения автотестов был запущен сервер, чтобы проверить связь, а также создавались тестовые пользователи.

Общий результат выполнения автотестов представлен на рисунке 24.

ID	Пользователь	Email	Роль	Статус	Создан	Действия
#1	Администратор Системы @admin1	admin@company.com	Администратор	АКТИВЕН	01.01.2024	Назначить роль
#2	Иван Агентов @agent1	agent1@company.com	Агент	АКТИВЕН	02.01.2024	Назначить роль
#3	Мария Урегуливщикова @adjuster1	adjuster1@company.com	Урегуливщик	АКТИВЕН	03.01.2024	Назначить роль
#4	Петр Операторов @operator1	operator1@company.com	Оператор	АКТИВЕН	04.01.2024	Назначить роль
#5	Анна Менеджера @manager1	manager1@company.com	Менеджер	АКТИВЕН	05.01.2024	Назначить роль

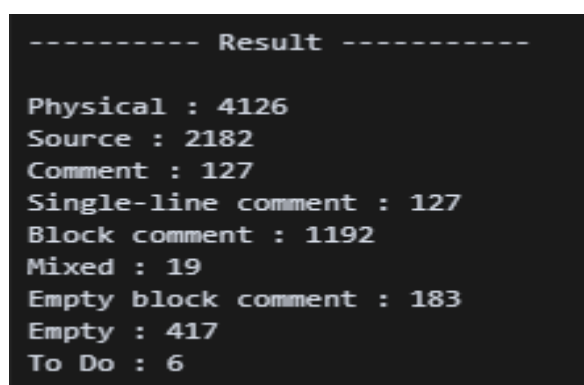
Рисунок 24 – Результат выполнения автотестов

4.5 Оценка качества программного кода на основе метрик

4.5.1 Метрика SLOC

Количество строк кода (SLOC) – метрика программного обеспечения, которая позволяет определить объем кода с помощью подсчета строк в тексте кода. Возьмем код серверной части приложения автоматизированной информационной системы страховой компании. Код приведен в приложение А.

Для расчета метрики используем приложение cloc, которое позволяет автоматически подсчитать метрику SLOC



```
----- Result -----  
  
Physical : 4126  
Source : 2182  
Comment : 127  
Single-line comment : 127  
Block comment : 1192  
Mixed : 19  
Empty block comment : 183  
Empty : 417  
To Do : 6
```

Рисунок 25 – Результат измерений серверной части приложения

Из приведенных данных можно выделить:

- физические строки кода – 4126;
- логические строки кода – 2182;
- комментарии – 127;
- пустые строки – 417.

Таким образом из результатов измерений можно отметить, что серверная часть приложения достаточно объемна, однако большое количество пустых строк и минимальные комментарии обеспечивают хорошую читаемость кода.

Заключение

В ходе выполнения выпускной квалификационной работы спроектирована автоматизированная информационная система страховой компании.

На этапе анализа требований была изучена предметная область, проведен сравнительный анализ приложений аналогов, выявлены требования к приложению и выбрана среда разработки.

В ходе выполнения анализа средой разработки для реализации приложения страховой компании был выбран neovim. Разработка была выполнена на языках Python3, TypeScript.

Проведена оценка качества работы приложения с помощью метрики SLOC, был выявлен высокий объем кода, что типично для приложений такого рода.

Таким образом, поставленные цели выпускной квалификационной работы достигнуты. Разработано web-приложение страховой компании.

Список использованных источников

1. Страховая организация [Электронный ресурс] URL: https://www.consultant.ru/document/cons_doc_LAW_1307/4b882a523ba34cb8f1870b896f5994e74b449b5c/ (дата обращения: 08.06.2025).
2. UML основные типы диаграмм [Электронный ресурс]. URL: <https://habr.com/ru/articles/738428/> (дата обращения: 08.06.2025).
3. Документация neovim [Электронный ресурс]. URL: <https://neovim.io/doc/> (дата обращения: 08.06.2025).
4. Документация Github URL: <https://docs.github.com/en/actions/about-github-actions/about-continuous-integration-with-github-actions/> (дата обращения: 09.06.2025).
5. Документация Python URL: <https://docs.python.org/3/> (дата обращения: 09.06.2025).
6. TypeScript [Электронный ресурс]. URL: <https://www.typescriptlang.org/docs/> (дата обращения: 08.06.2025).
7. Диаграммы компонентов [Электронный ресурс]. URL: <https://habr.com/ru/articles/756552/> (дата обращения: 09.06.2025).
8. ER диаграммы [Электронный ресурс]. URL: <https://www.lucidchart.com/pages/ru/erd-diagram> (дата обращения: 15.06.2025).
9. PostgreSQL doc [Электронный ресурс]. URL: <https://www.postgresql.org/docs/> (дата обращения: 15.06.2025).
10. Electron linux [Электронный ресурс]. URL: <https://www.electronjs.org/docs/latest/development/build-instructions-linux> (дата обращения: 15.06.2025).
11. React.js [Электронный ресурс]. URL: <https://react.dev/learn> (дата обращения: 15.06.2025).
12. Клиент серверная архитектура [Электронный ресурс]. URL: <https://servergate.ru/articles/klient-servernaya-arkhitektura/> (дата обращения: 15.06.2025).

13. Тестирование и 7 основных этапов его проведения // NEIROS [Электронный ресурс]. URL: <https://neiros.ru/blog/code/testirovanie-i-7-osnovnykh-etapov-ego-provedeniya/> (дата обращения: 10.06.2025).
14. Типы тестирования программного обеспечения: различия и примеры // LOADVIEW [Электронный ресурс]. URL: <https://www.loadview-testing.com/ru/blog/типы-тестирования-программного-обес/> (дата обращения: 10.06.2025).
15. Unit-тестирование // SKILLFACTORY MEDIA [Электронный ресурс]. URL: <https://blog.skillfactory.ru/glossary/unit-testirovanie/> (дата обращения: 10.06.2025).
16. Программный код и его метрики // Хабр [Электронный ресурс]. URL: <https://habr.com/ru/companies/intel/articles/106082/> (дата обращения: 10.06.2025).
17. Метрики сложности программ // Studwood [Электронный ресурс]. URL: https://studwood.net/2361018/informatika/metriki_slozhnosti_programm (дата обращения 15.06.2025).
18. Source lines of code // Wikipedia [Электронный ресурс]. URL: https://en.m.wikipedia.org/wiki/Source_lines_of_code (дата обращения: 14.05.2025).
19. Rest API документация [Электронный ресурс] URL: <https://docs.github.com/ru/rest?apiVersion=2022-11-28>(12.06.2025)
20. Законы РФ [Электронный ресурс]. URL <https://www.consultant.ru/popular/> (дата обращения. 13.06.2025)

**Приложение А
(обязательное)
Листинг программы**

Листинг 1А – роутер Analytics.py

```

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from typing import List
from datetime import datetime, date, timedelta
from app.utils.auth import get_current_user, require_roles
from app.db.database import get_db
from app.schemas.reports import FinanceReportData, ActivityReportData
from app.functions.analytics_service import AnalyticsService
router = APIRouter()
@router.get("/dashboard")
async def get_dashboard_data(
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("manager", "admin"))
):
    """Get dashboard analytics data"""
    from app.db.models import Client, Contract, Claim, ContractStatus, ClaimStatus
    from sqlalchemy import func
    total_clients = db.query(Client).count()
    active_clients = total_clients # Все клиенты считаются активными
    # Статистика за текущий месяц
    today = date.today()
    month_start = today.replace(day=1)
    new_clients_this_month = db.query(Client).filter(
        Client.created_at >= month_start
    ).count()
    # Статистика договоров
    total_contracts = db.query(Contract).count()
    active_contracts = db.query(Contract).filter(
        Contract.status == ContractStatus.ACTIVE
    ).count()
    expired_contracts = db.query(Contract).filter(
        Contract.status == ContractStatus.EXPIRED
    ).count()
    # Статистика заявок
    total_claims = db.query(Claim).count()
    pending_claims = db.query(Claim).filter(
        Claim.status.in_(['submitted', 'under_review'])
    ).count()
    approved_claims = db.query(Claim).filter(
        Claim.status == 'approved'
    ).count()
    rejected_claims = db.query(Claim).filter(
        Claim.status == 'rejected'
    ).count()
    # Статистика выручки
    total_revenue = db.query(func.sum(Contract.premium_amount)).scalar()
    or 0
    monthly_revenue = db.query(func.sum(Contract.premium_amount)).filter(
        Contract.created_at >= month_start
    ).scalar() or 0
    return {
        "clients": {
        "total": total_clients,

```

```

"active": active_clients,
"new_this_month": new_clients_this_month
},
"contracts": {
"total": total_contracts,
"active": active_contracts,
"expired": expired_contracts
},
"claims": {
"total": total_claims,
"pending": pending_claims,
"approved": approved_claims,
"rejected": rejected_claims
},
"revenue": {
"total": float(total_revenue),
"monthly": float(monthly_revenue)
}
}

@router.get("/reports/finance", response_model=FinanceReportData)
async def get_finance_report(
start_date: date = None,
end_date: date = None,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("manager", "admin"))
):
analytics_service = AnalyticsService(db)
# Устанавливаем дефолтные даты если не указаны
if not end_date:
end_date = date.today()
if not start_date:
start_date = date.today() - timedelta(days=90) # последние 3 месяца
from app.db.models import Contract, Claim, InsuranceProduct
from sqlalchemy import func, and_, extract
# Получаем договоры за период
contracts = db.query(Contract).filter(
and_(
Contract.created_at >= start_date,
Contract.created_at <= end_date
)
).all()
# Получаем выплаченные заявки за период
paid_claims = db.query(Claim).filter(
and_(
Claim.updated_at >= start_date,
Claim.updated_at <= end_date,
Claim.status == 'approved'
)
).all()
total_premiums = sum(c.premium_amount for c in contracts)
total_claims = sum(c.approved_amount or 0 for c in paid_claims)
profit = total_premiums - total_claims
# Помесячная разбивка
monthly_data = {}
for contract in contracts:

```

```

month_key = contract.created_at.strftime('%Y-%m')
if month_key not in monthly_data:
monthly_data[month_key] = {'premiums': 0, 'claims': 0, 'profit': 0}
monthly_data[month_key]['premiums'] += contract.premium_amount
for claim in paid_claims:
month_key = claim.updated_at.strftime('%Y-%m')
if month_key in monthly_data:
monthly_data[month_key]['claims'] += claim.approved_amount or 0
# Вычисляем прибыль по месяцам
for month in monthly_data:
monthly_data[month]['profit'] = monthly_data[month]['premiums'] -
monthly_data[month]['claims']
by_month = [
{
'month': month,
'premiums': data['premiums'],
'claims': data['claims'],
'profit': data['profit']
}
for month, data in sorted(monthly_data.items())
]
# По продуктам
product_data = {}
for contract in contracts:
product_id = contract.product_id
if product_id not in product_data:
product = db.query(InsuranceProduct).filter(InsuranceProduct.id ==
product_id).first()
product_data[product_id] = {
'product_name': product.name if product else f'Product {product_id}',
'premiums': 0,
'claims': 0,
'count': 0
}
product_data[product_id]['premiums'] += contract.premium_amount
product_data[product_id]['count'] += 1
# Добавляем данные по заявкам к продуктам
for claim in paid_claims:
if hasattr(claim, 'contract') and claim.contract:
product_id = claim.contract.product_id
if product_id in product_data:
product_data[product_id]['claims'] += claim.approved_amount or 0
by_product = list(product_data.values())
return FinanceReportData(
total_premiums=total_premiums,
total_claims=total_claims,
profit=profit,
period={"start": start_date.isoformat(), "end":
end_date.isoformat()},
by_month=by_month,
by_product=by_product
)

@router.get("/reports/activity", response_model=ActivityReportData)
async def get_activity_report(
start_date: date = None,

```

```

end_date: date = None,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("manager", "admin"))
):
analytics_service = AnalyticsService(db)
# Устанавливаем дефолтные даты если не указаны
if not end_date:
end_date = date.today()
if not start_date:
start_date = date.today() - timedelta(days=90)
# Имитируем данные о пользователях
total_users = 25 # Пример
active_users = 18 # Пример
# Статистика по ролям
by_role = [
{"role": "agent", "count": 8, "active": 6},
{"role": "adjuster", "count": 4, "active": 3},
{"role": "operator", "count": 6, "active": 5},
{"role": "manager", "count": 3, "active": 2},
{"role": "admin", "count": 4, "active": 2}
]
# Топ агентов по договорам
from app.db.models import Contract
from sqlalchemy import func, and_
agent_stats = db.query(
Contract.agent_id,
func.count(Contract.id).label('contracts_count'),
func.sum(Contract.premium_amount).label('total_premium')
).filter(
and_(
Contract.created_at >= start_date,
Contract.created_at <= end_date,
Contract.agent_id.isnot(None)
)
).group_by(Contract.agent_id).order_by(
func.sum(Contract.premium_amount).desc()
).limit(10).all()
top_agents = [
{
'agent_name': f'Агент {stat.agent_id}', # В реальности нужно
получать из auth-service
'contracts_count': stat.contracts_count,
'total_premium': float(stat.total_premium)
}
]
for stat in agent_stats
]
return ActivityReportData(
total_users=total_users,
active_users=active_users,
by_role=by_role,
top_agents=top_agents
)

@router.get("/reports/contracts")
async def get_contracts_report(
start_date: date = None,

```

```

end_date: date = None,
current_user: dict = Depends(require_roles("manager", "admin"))
):
# TODO: Implement contracts report logic
return {
    "report_type": "contracts",
    "period": {"start": start_date, "end": end_date},
    "data": []
}

@router.get("/reports/claims")
async def get_claims_report(
    start_date: date = None,
    end_date: date = None,
    current_user: dict = Depends(require_roles("manager", "admin"))
):
    """Generate claims report"""
    # TODO: Implement claims report logic
    return {
        "report_type": "claims",
        "period": {"start": start_date, "end": end_date},
        "data": []
    }

@router.get("/reports/revenue")
async def get_revenue_report(
    start_date: date = None,
    end_date: date = None,
    current_user: dict = Depends(require_roles("manager", "admin"))
):
    # TODO: Implement revenue report logic
    return {
        "report_type": "revenue",
        "period": {"start": start_date, "end": end_date},
        "total_revenue": 0,
        "by_month": [],
        "by_product": []
    }

@router.get("/statistics/overview")
async def get_overview_statistics(
    current_user: dict = Depends(require_roles("manager", "admin"))
):
    # TODO: Implement overview statistics logic
    return {
        "clients": {"total": 0, "active": 0, "new_this_month": 0},
        "contracts": {"total": 0, "active": 0, "expired": 0},
        "claims": {"total": 0, "pending": 0, "approved": 0, "rejected": 0},
        "agents_performance": []
    }

```

Листинг 2А - claims.py

```

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from typing import List
from pydantic import BaseModel

```

```

from app.utils.auth import get_current_user, require_roles
from app.db.database import get_db
from app.schemas.claim import (
    ClaimCreate, ClaimUpdate, Claim as ClaimSchema,
    ClaimList, ClaimDecisionRequest, PendingClaimsList,
    ClaimWithDetails
)
from app.functions.claim_service import ClaimService

router = APIRouter()

class ClaimCreate(BaseModel):
    contract_id: int
    incident_date: str
    description: str
    claim_amount: float
    documents: List[str] = []

class ClaimSubmitRequest(BaseModel):
    contract_id: int
    incident_date: str
    description: str
    claim_amount: float
    documents: List[str] = []
    priority: str = "normal" # "low", "normal", "high", "urgent"
    customer_contact: str = None
    witnesses: List[str] = []
class ClaimSubmitResponse(BaseModel):
    claim_id: int
    claim_number: str
    status: str
    estimated_processing_time: str
    adjuster_assigned: bool
    validation_checklist: dict

class ClaimDecision(BaseModel):
    decision: str # "approved", "rejected", "requires_investigation"
    approved_amount: float = None
    rejection_reason: str = None
    notes: str = None

@router.get("/", response_model=ClaimList)
async def get_claims(
    skip: int = 0,
    limit: int = 100,
    status_filter: str = None,
    contract_id: int = None,
    db: Session = Depends(get_db),
    current_user: dict = Depends(get_current_user)
):
    claim_service = ClaimService(db)
    claims, total = claim_service.get_claims(
        skip=skip,
        limit=limit,
        status_filter=status_filter,
        contract_id=contract_id

```

```

)
return ClaimList(
    claims=claims,
    total=total,
    skip=skip,
    limit=limit
)

@router.get("/pending", response_model=PendingClaimsList)
async def get_pending_claims(
    skip: int = 0,
    limit: int = 100,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("adjuster"))
):
    claim_service = ClaimService(db)
    pending_claims, total = claim_service.get_pending_claims(skip=skip,
    limit=limit)
    return PendingClaimsList(
        pending_claims=pending_claims,
        total=total,
        skip=skip,
        limit=limit
    )

@router.post("/", response_model=ClaimSchema)
async def create_claim(
    claim_data: ClaimCreate,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("agent", "operator",
    "admin"))
):
    """Create new insurance claim"""
    claim_service = ClaimService(db)
    claim = claim_service.create_claim(claim_data,
    created_by=current_user.get("user_id"))
    return claim

@router.get("/{claim_id}", response_model=ClaimWithDetails)
async def get_claim(
    claim_id: int,
    db: Session = Depends(get_db),
    current_user: dict = Depends(get_current_user)
):
    """Get claim by ID"""
    claim_service = ClaimService(db)
    claim = claim_service.get_claim_with_details(claim_id)
    if not claim:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Claim not found"
        )
    return claim

@router.put("/{claim_id}/decision", response_model=ClaimSchema)
async def make_claim_decision(

```

```

claim_id: int,
decision_data: ClaimDecisionRequest,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("adjuster"))
):
    """Make decision on insurance claim (adjuster only)"""
    claim_service = ClaimService(db)
    # Check if claim exists
    existing_claim = claim_service.get_claim(claim_id)
    if not existing_claim:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Claim not found"
        )
    claim = claim_service.make_decision(
        claim_id,
        decision_data,
        adjuster_id=current_user.get("user_id")
    )
    return claim

@router.put("/{claim_id}", response_model=ClaimSchema)
async def update_claim(
    claim_id: int,
    claim_data: ClaimUpdate,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("adjuster", "manager",
        "admin"))
):
    claim_service = ClaimService(db)
    existing_claim = claim_service.get_claim(claim_id)
    if not existing_claim:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Claim not found"
        )
    claim = claim_service.update_claim(claim_id, claim_data)
    return claim

@router.post("/{claim_id}/process")
async def process_claim(
    claim_id: int,
    current_user: dict = Depends(get_current_user)
):
    if current_user.get("role") != "adjuster":
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Only adjusters can process claims"
        )
    return {"message": "Claim processed successfully"}

@router.post("/{claim_id}/approve")
async def approve_claim(
    claim_id: int,
    approval_amount: float,
    current_user: dict = Depends(get_current_user)

```

```

):
    allowed_roles = ["manager", "admin"]
    if current_user.get("role") not in allowed_roles:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Only managers can approve claims"
        )
    return {"message": "Claim approved successfully", "amount":
        approval_amount}

@router.post("/{claim_id}/reject")
async def reject_claim(
    claim_id: int,
    rejection_reason: str,
    current_user: dict = Depends(get_current_user)
):
    """Reject insurance claim"""
    allowed_roles = ["adjuster", "manager", "admin"]
    if current_user.get("role") not in allowed_roles:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Insufficient permissions"
        )
    return {"message": "Claim rejected", "reason": rejection_reason}

@router.post("/submit", response_model=ClaimSubmitResponse)
async def submit_claim_to_adjuster(
    claim_data: ClaimSubmitRequest,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("operator"))
):
    claim_service = ClaimService(db)
    from app.db.models import Contract
    contract = db.query(Contract).filter(Contract.id ==
        claim_data.contract_id).first()
    if not contract:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Contract not found"
        )
    if contract.status != "active":
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Contract is not active"
        )
    # Валидационный чек-лист
    validation_checklist = {
        "contract_valid": True,
        "contract_active": contract.status == "active",
        "incident_date_valid": True, # TODO: проверить что дата не в
        будущем
        "description_complete": len(claim_data.description) >= 20,
        "amount_reasonable": claim_data.claim_amount <=
        contract.coverage_amount,
        "documents_attached": len(claim_data.documents) > 0,
        "contact_provided": claim_data.customer_contact is not None
    }

```

```

}
# Все проверки должны быть пройдены
all_valid = all(validation_checklist.values())
if not all_valid:
    raise HTTPException(
        status_code=status.HTTP_400_BAD_REQUEST,
        detail={
            "message": "Validation failed",
            "checklist": validation_checklist
        }
    )
# Создаем заявку
create_data = ClaimCreate(
    contract_id=claim_data.contract_id,
    incident_date=claim_data.incident_date,
    description=claim_data.description,
    claim_amount=claim_data.claim_amount,
    documents=claim_data.documents
)
claim = claim_service.create_claim(create_data,
    created_by=current_user.get("user_id"))
# Назначаем урегулировщика (простая логика)
adjuster_assigned = True
if claim_data.priority == "urgent":
    processing_time = "1-2 дня"
elif claim_data.priority == "high":
    processing_time = "3-5 дней"
else:
    processing_time = "5-10 дней"
return ClaimSubmitResponse(
    claim_id=claim.id,
    claim_number=claim.claim_number,
    status=claim.status,
    estimated_processing_time=processing_time,
    adjuster_assigned=adjuster_assigned,
    validation_checklist=validation_checklist
)

```

Листинг 3А - clients.py

```

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from typing import List
from app.utils.auth import get_current_user, require_roles
from app.db.database import get_db
from app.db.models import Client
from app.schemas.client import ClientCreate, ClientUpdate, Client as
ClientSchema, ClientList
from app.functions.client_service import ClientService

router = APIRouter()

@router.get("/", response_model=ClientList)
async def get_clients(
    skip: int = 0,

```

```

limit: int = 100,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("agent", "operator",
"admin"))
):
client_service = ClientService(db)
clients, total = client_service.get_clients(skip=skip, limit=limit)
return ClientList(
clients=clients,
total=total,
skip=skip,
limit=limit
)

@router.post("/", response_model=ClientSchema)
async def create_client(
client_data: ClientCreate,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("agent", "operator",
"admin"))
):
client_service = ClientService(db)
existing_client =
client_service.get_client_by_email(client_data.email)
if existing_client:
raise HTTPException(
status_code=status.HTTP_400_BAD_REQUEST,
detail="Client with this email already exists"
)
client = client_service.create_client(client_data,
created_by=current_user.get("user_id"))
return client

@router.get("/{client_id}", response_model=ClientSchema)
async def get_client(
client_id: int,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("agent", "operator",
"admin"))
):
"""Get client by ID"""
client_service = ClientService(db)
client = client_service.get_client(client_id)
if not client:
raise HTTPException(
status_code=status.HTTP_404_NOT_FOUND,
detail="Client not found"
)
return client

@router.put("/{client_id}", response_model=ClientSchema)
async def update_client(
client_id: int,
client_data: ClientUpdate,
db: Session = Depends(get_db),
current_user: dict = Depends(require_roles("agent", "operator",

```

```

"admin"))
):
    client_service = ClientService(db)
    # Check if client exists
    existing_client = client_service.get_client(client_id)
    if not existing_client:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Client not found"
        )
    # Check email uniqueness if email is being updated
    if client_data.email and client_data.email != existing_client.email:
        email_client = client_service.get_client_by_email(client_data.email)
        if email_client and email_client.id != client_id:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Client with this email already exists"
            )
    client = client_service.update_client(client_id, client_data)
    return client

@router.delete("/{client_id}")
async def delete_client(
    client_id: int,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("admin"))
):
    client_service = ClientService(db)
    existing_client = client_service.get_client(client_id)
    if not existing_client:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Client not found"
        )
    # Check if client has active contracts
    if client_service.has_active_contracts(client_id):
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Cannot delete client with active contracts"
        )
    client_service.delete_client(client_id)
    return {"message": "Client deleted successfully"}

```

Листинг 4А - products.py

```

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from typing import List
from app.utils.auth import get_current_user, require_roles
from app.db.database import get_db
from app.db.models import InsuranceProduct
from pydantic import BaseModel

router = APIRouter()

```

```

class ProductCreate(BaseModel):
    name: str
    description: str
    base_premium: float
    coverage_amount: float

class Product(BaseModel):
    id: int
    name: str
    description: str
    base_premium: float
    coverage_amount: float
    is_active: bool

class Config:
    from_attributes = True

@router.get("/", response_model=List[Product])
async def get_products(
    db: Session = Depends(get_db),
    current_user: dict = Depends(get_current_user)
):
    products =
    db.query(InsuranceProduct).filter(InsuranceProduct.is_active ==
    True).all()
    return products

@router.post("/", response_model=Product)
async def create_product(
    product_data: ProductCreate,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("manager", "admin"))
):
    product = InsuranceProduct(
        **product_data.dict(),
        is_active=True
    )
    db.add(product)
    db.commit()
    db.refresh(product)
    return product

@router.put("/{product_id}", response_model=Product)
async def update_product(
    product_id: int,
    product_data: ProductCreate,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("manager", "admin"))
):
    """Update insurance product (manager/admin only)"""
    product = db.query(InsuranceProduct).filter(InsuranceProduct.id ==
    product_id).first()
    if not product:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Product not found"

```

```

)
for field, value in product_data.dict().items():
    setattr(product, field, value)
db.commit()
db.refresh(product)
return product

@router.delete("/{product_id}")
async def delete_product(
    product_id: int,
    db: Session = Depends(get_db),
    current_user: dict = Depends(require_roles("admin"))
):
    product = db.query(InsuranceProduct).filter(InsuranceProduct.id ==
product_id).first()
    if not product:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Product not found"
        )
    product.is_active = False
    db.commit()
    return {"message": "Product deleted successfully"}

```

Приложение Б
(рекомендуемое)
Скриншоты работы приложения

Рисунок 1Б – форма регистрации пользователя

Создайте новый аккаунт

[Вход](#) [Регистрация](#)

Имя пользователя *

Email *

Полное имя *

Роль

Пароль *

Подтвердите пароль *

[Зарегистрироваться](#)

Рисунок 2Б – форма входа пользователя

Insurance Management System

Войдите в систему для продолжения

[Вход](#) [Регистрация](#)

Имя пользователя или email

Пароль

[Войти](#)

Рисунок 3Б – форма статистики работы

Рабочее место оператора

● В работе

Поиск клиентов Отправка заявки **Статистика**

Статистика работы

СЕГОДНЯ

Обработано обращений:	12
Создано заявок:	8
Консультаций:	15

Рисунок 4Б – форма поиска клиентов страховой компании

Поиск клиентов Отправка заявки Статистика

Поиск клиентов и договоров

Поиск по ФИО, email, телефону...

Введите запрос для поиска клиентов

Рисунок 5Б – форма калькулятора страховых премий

Калькулятор страховых премий

Продукт страхования	Сумма покрытия (₽)
Автострахование ОСАГО	100000
Возраст клиента	Срок (месяцев)
30	12

Факторы риска

☐ Высокий риск региона ☐ Система безопасности

Количество предыдущих страховых случаев

0

Рассчитать премию