

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

Вычислительной техники

Кафедра

Вычислительная техника

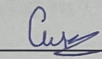
Направление подготовки 09.05.01 «Применение и эксплуатация
автоматизированных систем специального назначения»
Профиль «Эксплуатация вычислительных машин, комплексов,
систем и сетей специального назначения»

ДИПЛОМНЫЙ ПРОЕКТ

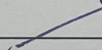
на тему

ИНФОРМАЦИОННАЯ СИСТЕМА
УЧРЕЖДЕНИЯ ЗДРАВООХРАНЕНИЯ

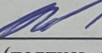
Студент

 27.05.2026 Синицын Р.Д.
(подпись, дата)

Руководитель

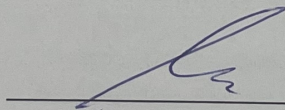
 27.05.2026 Митрохин М.А.
(подпись, дата)

Нормоконтролёр

 27.05.2026 А.С. Бычков
(подпись, дата)

Работа допущена к защите (протокол заседания кафедры от
22.05.2026 г. № 14)

Заведующий кафедрой

 М.А. Митрохин
(подпись)

Рецензент

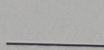
Мат. УИТ

 (подпись, дата)

Австеев А.В.

Работа защищена с отметкой удовлетворительно (протокол заседания ГЭК от 10.06.2026 № 7)

Секретарь ГЭК

 Гурин Е.И.
(подпись)

Пенза 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

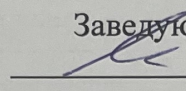
вычислительной техники

Кафедра

«Вычислительная техника»

«Утверждаю»

Заведующий кафедрой

 М.А. Митрохин

«11» 02 2026 г.

ЗАДАНИЕ

1. Студент Сеницын Роман Дмитриевич гр.21ВОЭ1 факультета вычислительной техники.

2. Тема работы: «Информационная система учреждения здравоохранения»

Тема утверждена приказом ПГУ № 122/о от "11" февраля 2026 г.

3. Руководитель дипломного проекта д.т.н., профессор кафедры ВТ Митрохин М.А.

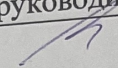
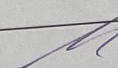
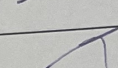
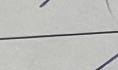
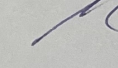
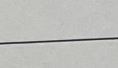
4. Задание на работу (назначение разработки, исходные данные и т.п.)

Требуется разработать веб-ориентированную информационную систему для автоматизации деятельности учреждения здравоохранения. Система должна обеспечивать ведение расписания врачей, управление записью пациентов на приём включая создание, перенос и отмену, хранение и просмотр медицинских записей и истории болезни, работу с медицинскими документами с возможностью загрузки и скачивания, личные кабинеты с разграничением прав для трёх ролей: пациент, врач и администратор, а также формирование отчётов по деятельности учреждения. Система должна быть реализована с использованием клиент-серверной архитектуры, обеспечивать безопасность персональных данных и соответствовать требованиям к медицинским информационным системам.

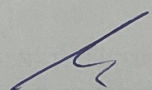
5. Перечень подлежащих разработке вопросов

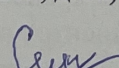
Требуется провести анализ предметной области и существующих аналогов, сформулировать требования к системе, обосновать выбор технологий. Необходимо спроектировать архитектуру системы и структуру базы данных, разработать серверную часть на FastAPI и клиентскую часть на React, реализовав модули аутентификации, личных кабинетов, расписания, записи на приём, медицинских записей, документов и отчётов.

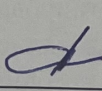
6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучение предметной области	15	12.02.2026-15.02.2026	
2.	Анализ требований	5	16.02.2026-28.02.2026	
3.	Выбор технологий, применяемых при проектировании и разработке	10	01.03.2026-20.03.2026	
4.	Проектирование базы данных и пользовательского интерфейса	20	20.03.2026-30.03.2026	
5.	Разработка серверной и клиентской части	60	31.03.2026-30.04.2026	
6.	Оформление пояснительной записки	15	01.05.2026-20.05.2026	

Дата выдачи задания "11" февраля 2026 г.

Руководитель дипломной работы  Митрохин М.А.
(подпись, дата)

Задание к исполнению принял студент  Сеницын Р.Д.
(подпись, дата)

Работу к защите допустить
Декан факультета  09.06.2020 Фионова Л.Р.
(подпись, дата)

Реферат

Бакалаврская работа содержит 105 листов, 23 рисунка, 2 таблицы, 10 источников и 3 приложения.

ИНФОРМАЦИОННАЯ СИСТЕМА, МЕДИЦИНСКИЕ ДАННЫЕ, АВТОМАТИЗАЦИЯ, FASTAPI, REACT, PIXEL POSTGRESQL

Цель работы — разработать веб-ориентированную информационную систему, обеспечивающую ведение расписания врачей, управление приёмами, хранение медицинских записей, работу с документами, а также предоставление пациентам удобного интерфейса для взаимодействия с учреждением.

В процессе выполнения работы проведён анализ предметной области, исследованы существующие программные решения, определены их преимущества и недостатки. На основе анализа сформулированы функциональные и нефункциональные требования к разрабатываемой системе. Выполнено проектирование архитектуры, разработана структура базы данных, реализованы основные модули системы: авторизация и аутентификация, личные кабинеты пользователей, управление расписанием, приёмами, медицинскими записями и документами.

Для реализации использованы современные технологии: FastAPI в качестве backend-фреймворка, PostgreSQL в качестве СУБД, React для разработки пользовательского интерфейса.

ПГУ 1.090501.16.001 ПЗ							
Изм.	Лист	№ докум.	Подпись	Дата			
Разраб.	Синицын Р.Д.			27.05.26	Лит.	Лист	Листов
Провер.	Митрохин М.А.			27.05.26		2	108
Ил. Контр.	Бычков А.С.			27.05.26	ФВТ, каф. ВТ		
Утверд.	Митрохин М.А.			27.05.26	гр. 21ВОЭ1		

Информационная система
учреждения здравоохранения
Пояснительная записка

Содержание

Перечень сокращений и обозначений	5
Введение.....	6
1 Анализ предметной области.....	8
1.1 Общая характеристика учреждения здравоохранения	8
1.1.1 Анализ существующих аналогов	10
1.1.2 Обоснование актуальности разработки программы	22
1.1.3 Постановка задачи.....	16
1.1.4 Выбор технологий и инструментов	19
1.1.5 Выбор серверной платформы.....	22
1.1.6 Выбор системы управления базами данных.....	26
1.1.7 Выбор технологий клиентской части.....	32
1.1.8 Формат обмена данными и взаимодействие компонентов	39
1.2 Проектирование и разработка	46
1.2.1 Проектирование базы данных.....	47
1.2.2 Проектирование пользовательского интерфейса.....	49
1.2.3 Разработка серверной части	57
1.2.4 Разработка клиентской части	58
1.2.5 Интеграция и взаимодействие компонентов	59
1.2.6 Обеспечение безопасности.....	60
1.2.7 Динамика работы приложения	61

1.2.8 Тестирование приложения	63
2 Безопасность жизнедеятельности.....	64
2.1 Характеристика рабочего места разработчика	64
2.2 Пожарная безопасность.....	65
2.3 Психофизиологические нагрузки и профилактика утомления	66
3 Экономика.....	67
Заключение.....	74
Список использованных источников.....	75
Приложение А – UML-диаграммы	76
Приложение Б – Листинг программы.....	81
Приложение В – Презентация.....	105

Перечень сокращений и обозначений

API (Application Programming Interface) – программный интерфейс для взаимодействия между клиентской и серверной частями системы

CRUD (Create, Read, Update, Delete) – базовые операции над данными в информационных системах

DB (DataBase) – база данных

DTO (Data Transfer Object) – объект передачи данных между слоями приложения

FastAPI – фреймворк для разработки серверной части на Python

HTTP (HyperText Transfer Protocol) – протокол передачи данных между клиентом и сервером

IP (Internet Protocol) – сетевой протокол, используемый для адресации устройств

JSON (JavaScript Object Notation) – текстовый формат обмена данными, который используется в качестве сохранения игровых настроек и прогресса

REST (Representational State Transfer) – архитектурный стиль построения веб-сервисов

React – библиотека JavaScript для создания пользовательских интерфейсов

SSL (Secure Sockets Layer) – технология шифрования данных при передаче

TLS (Transport Layer Security) – протокол защиты данных, используемый в HTTPS

UML (Unified Modeling Language) – унифицированный язык моделирования

UX (User Experience) – пользовательский опыт, удобство взаимодействия с системой

					ПГУ 1.090501.16.001 ПЗ	Лист
						5
Изм.	Лист	№ докум.	Подпись	Дата		

Введение

Современная система здравоохранения характеризуется высокой степенью сложности организационных процессов, огромным объемом информации, которую требуется обработать, а также высокими требованиями к удобству пациентов. Увеличивающийся объем пациентов, быстро растущий спектр оказываемых медицинских услуг, приводят к тому, что стандартные методы ведения документации и организации работы медицинского учреждения становятся не такими эффективными, как хотелось бы. В данных современных условиях необходимо внедрение информационных систем, которые позволяют разгрузить необходимые процессы внутри мед. учреждений и создать одно информационное пространство для всех участников медицинского обслуживания.

Информационные системы в здравоохранении представляют собой комплекс программных и технических средств, предназначенных для сбора, хранения, обработки и анализа медицинских данных. Их применение способствует повышению качества медицинской помощи, сокращению времени на выполнение рутинных операций, снижению вероятности ошибок, а также повышению прозрачности и управляемости деятельности учреждения. В условиях цифровой трансформации отрасли такие системы становятся неотъемлемой частью инфраструктуры любого современного медицинского учреждения.

Несмотря на наличие широкого спектра коммерческих и государственных решений, многие медицинские организации сталкиваются с проблемами адаптации существующих систем под собственные процессы, ограниченной функциональностью, недостаточной гибкостью или высокой стоимостью внедрения. Кроме того, значительная часть решений ориентирована на крупные учреждения и не учитывает потребности небольших клиник, частных медицинских центров и специализированных кабинетов.

					ПГУ 1.090501.16.001 ПЗ	Лист
						6
Изм.	Лист	№ докум.	Подпись	Дата		

Это создаёт необходимость разработки информационных систем, которые обладают модульной архитектурой, поддерживают масштабирование, обеспечивают удобство использования и соответствуют требованиям действующих нормативных документов.

Целью работы является создание веб-ориентированной информационной системы, обеспечивающей автоматизацию ключевых процессов медицинского учреждения и предоставляющей удобные инструменты для всех категорий пользователей.

Для достижения поставленной цели в работе решаются следующие задачи: анализ предметной области, исследование существующих аналогов, формирование требований к системе, выбор технологий разработки, проектирование архитектуры, реализация основных модулей, проведение тестирования и оценка эффективности разработанного решения.

					ПГУ 1.090501.16.001 ПЗ	Лист
						7
Изм.	Лист	№ докум.	Подпись	Дата		

1 Анализ предметной области

1.1 Общая характеристика учреждения здравоохранения

Учреждение здравоохранения представляет собой организацию, осуществляющую профилактическую, диагностическую, лечебную и реабилитационную деятельность. В зависимости от масштаба и профиля такие учреждения могут включать амбулаторно-поликлинические подразделения, стационарные отделения, диагностические центры, лаборатории, кабинеты узких специалистов и административные службы.

Функционирование медицинского учреждения включает множество взаимосвязанных процессов. Регистрация пациентов и ведение их медицинских карт требуют аккуратного хранения персональных данных, страховой информации и истории обращений. Организация работы врачей предполагает формирование расписаний, учёт загруженности, планирование приёмов и своевременное внесение изменений. Запись пациентов на приём должна обеспечивать удобный доступ к актуальным данным о доступных временных интервалах, а также возможность переноса или отмены визита. Проведение приёма связано с фиксацией жалоб, внесением медицинских записей, назначением обследований и формированием заключений.

Существенную роль играет работа с медицинскими документами, которые могут включать результаты анализов, заключения специалистов и иные файлы, требующие безопасного хранения и контроля доступа. Административная деятельность охватывает управление пользователями, формирование отчётности, анализ статистики посещений и контроль эффективности работы учреждения.

					ПГУ 1.090501.16.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		8

В предметной области выделяются три основные категории пользователей: пациенты, врачи и администраторы. Пациенты нуждаются в удобном доступе к личной информации, истории посещений, медицинским документам и возможностям записи на приём. Врачи работают с расписанием, проводят приёмы, формируют медицинские записи и анализируют данные пациентов. Администраторы обеспечивают организацию работы учреждения, управляют расписаниями, контролируют записи, формируют отчёты и поддерживают справочники. Каждая категория пользователей обладает собственными задачами и уровнем доступа, что требует реализации ролевой модели безопасности и строгого разграничения прав.

Информационные потоки в учреждении здравоохранения формируются вокруг данных о пациентах, расписаниях, приёмах, медицинских записях, документах и отчётности. Эти данные должны быть доступны в актуальном виде, корректно структурированы и защищены от несанкционированного доступа. В условиях бумажного документооборота или использования разрозненных программных решений возникают проблемы дублирования информации, затруднённого поиска, несогласованности данных между подразделениями, а также риски утраты документов. Кроме того, отсутствие единой информационной среды усложняет формирование отчётности и снижает эффективность взаимодействия между сотрудниками.

1.1.1 Анализ существующих аналогов

Современный рынок медицинских информационных систем представлен широким спектром решений, ориентированных на автоматизацию деятельности учреждений здравоохранения различного масштаба. Анализ существующих аналогов позволяет определить уровень развития отрасли, выявить сильные и слабые стороны популярных систем, а также сформировать требования к разрабатываемому программному продукту. Наиболее распространёнными являются комплексные медицинские информационные системы, интегрированные с государственными сервисами, а также коммерческие решения, ориентированные на частные клиники и медицинские центры.

Одним из наиболее известных решений является система «Медси», применяемая в крупных медицинских сетях. Она обеспечивает широкий функционал, включающий ведение электронной медицинской карты, управление расписанием, работу с лабораторными исследованиями и формирование отчётности. Однако подобные системы отличаются высокой стоимостью внедрения и обслуживания, а также значительной сложностью адаптации под индивидуальные процессы конкретного учреждения. Кроме того, их архитектура зачастую ориентирована на крупные организации, что делает их избыточными для небольших клиник.

Внешний вид приложения представлен на рисунке 1.

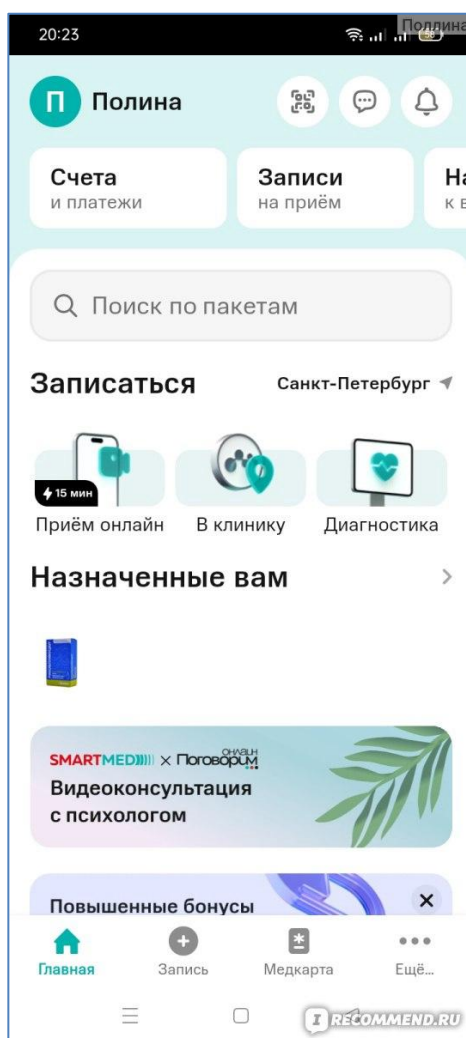


Рисунок 1 – Внешний вид «Медси»

Другим распространённым аналогом является система «1С:Медицина», представляющая собой модульное решение, интегрированное с платформой 1С. Она предоставляет возможности по ведению медицинской документации, учёту услуг, управлению расписанием и формированию отчётности. Преимуществом является гибкость настройки и широкая распространённость платформы, однако существенным недостатком остаётся сложность пользовательского интерфейса и необходимость привлечения специалистов для конфигурирования и сопровождения. Кроме того, система ориентирована преимущественно на бухгалтерский и управленческий учёт, что ограничивает её удобство для врачей и пациентов.

Внешний вид приложения представлен на рисунке 2.

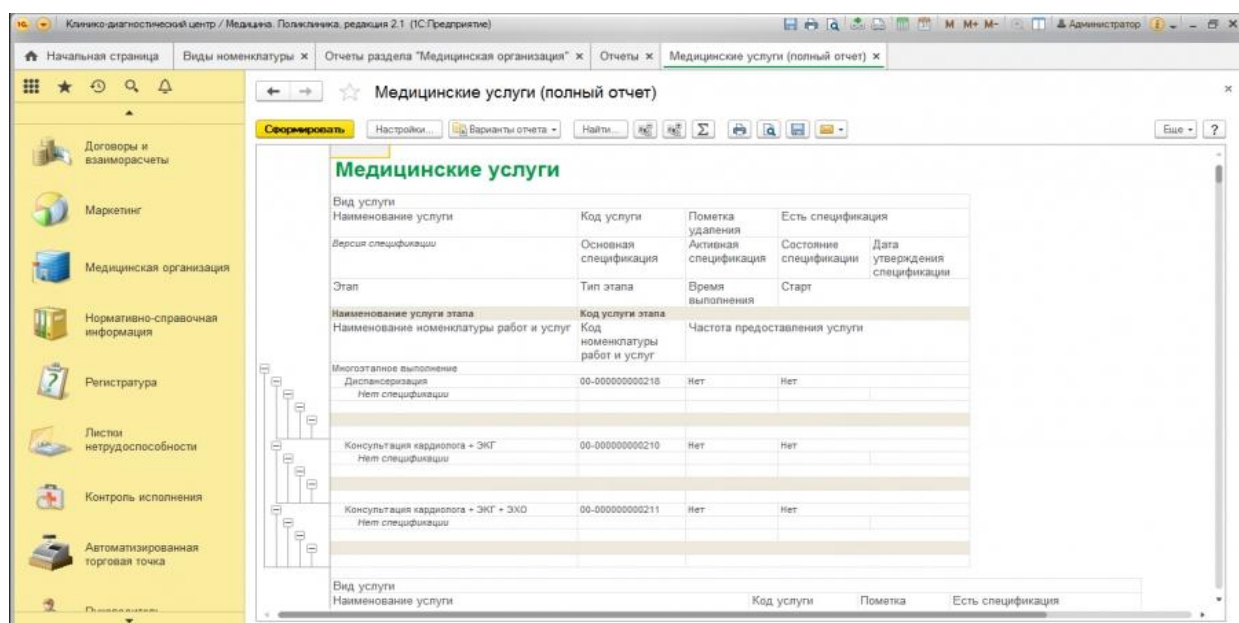


Рисунок 2 – Внешний вид «1С:Медицина»

Среди коммерческих решений для частных клиник можно выделить систему «Инфоклиника», ориентированную на автоматизацию записи на приём, ведение расписаний и управление медицинскими картами. Она отличается простотой использования и доступностью, однако её функциональность ограничена и не всегда позволяет реализовать специфические процессы учреждения. Кроме того, закрытая архитектура и отсутствие возможности расширения функционала затрудняют интеграцию с внешними сервисами и адаптацию под уникальные требования.

Внешний вид приложения представлен на рисунке 3

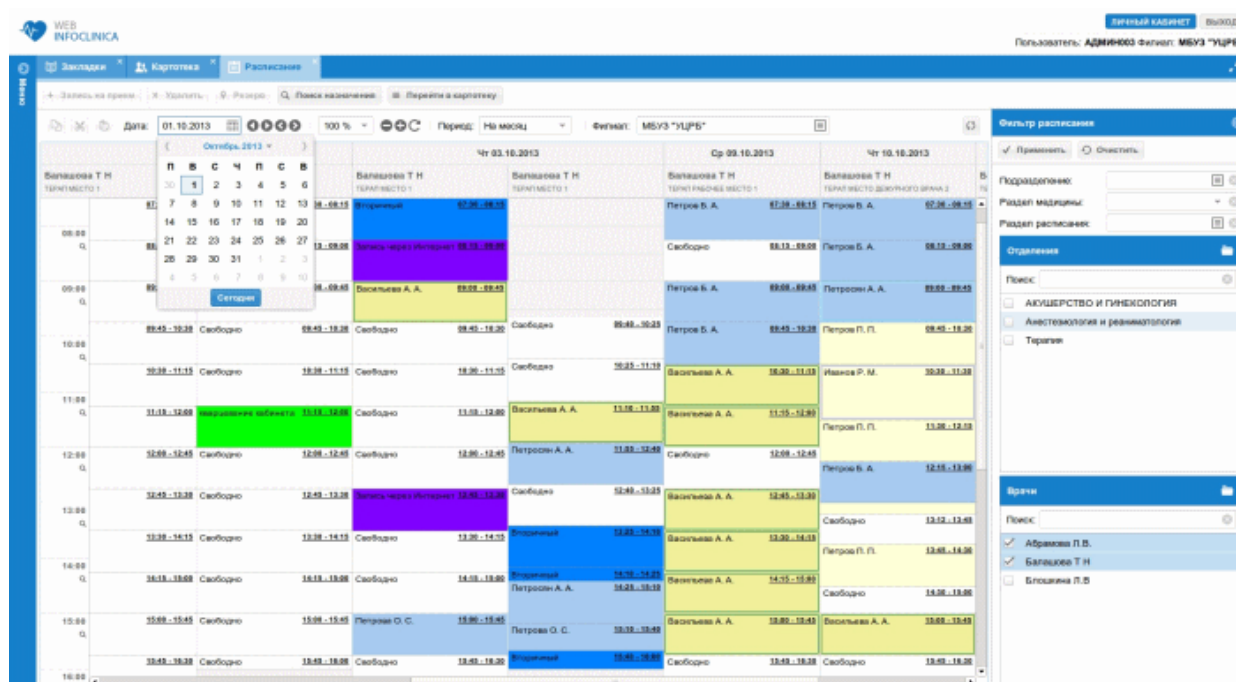


Рисунок 3 – Внешний вид «Инфоклиника»

В таблице 1 представлен анализ аналогов.

Таблица 1 – Анализ аналогов

Название системы	Основные возможности	Преимущества	Недостатки
Медси	Электронная медкарта, расписание, лабораторные модули, отчётность, интеграции.	Высокая функциональность, масштабируемость, развитая инфраструктура.	Высокая стоимость внедрения, сложность адаптации, избыточность для небольших учреждений.
1С:Медицина	Учёт услуг, расписание, медкарта, отчётность, интеграция с 1С.	Гибкость настройки, широкая распространённость платформы.	Сложный интерфейс, необходимость специалистов для сопровождения, ориентация на учёт.
Инфоклиника	Запись на приём, расписание, медкарта, управление пациентами.	Простота использования, доступность.	Ограниченная функциональность, закрытая архитектура, слабая расширяемость.

Название системы	Основные возможности	Преимущества	Недостатки
MedClinicls	Ведение приёмов, базовая медкарта, расписание	Простота, низкая стоимость	Отсутствие интеграций

1.1.2 Обоснование актуальности разработки программы

Актуальность разработки информационной системы для учреждения здравоохранения обусловлена совокупностью факторов, связанных с современным состоянием медицинской отрасли, ростом требований к качеству обслуживания пациентов и необходимостью повышения эффективности внутренних процессов. В условиях цифровой трансформации здравоохранения традиционные методы ведения документации и организации работы оказываются недостаточно эффективными. Бумажные носители, разрозненные таблицы и устаревшие локальные программы не обеспечивают необходимого уровня оперативности, точности и доступности данных. Это приводит к увеличению нагрузки на персонал, снижению скорости обработки информации, возникновению ошибок и затруднению контроля качества медицинских услуг.

Современные медицинские учреждения сталкиваются с необходимостью обработки значительных объёмов данных, включающих персональные сведения пациентов, результаты обследований, историю посещений, расписания врачей, медицинские записи и внутреннюю отчётность. При отсутствии единой информационной среды данные оказываются распределёнными между различными подразделениями, что затрудняет их согласование и повышает риск дублирования.

В результате нарушается целостность информации, усложняется взаимодействие между сотрудниками, а принятие управленческих решений становится менее эффективным. Дополнительным фактором, определяющим актуальность разработки, является повышение требований к защите персональных данных и соблюдению нормативных документов, регулирующих обработку медицинской информации. Использование устаревших или фрагментированных решений не позволяет обеспечить должный уровень безопасности, что создаёт риски утечки данных и нарушения законодательства. Современная информационная система должна обеспечивать надёжное хранение данных, разграничение прав доступа, контроль действий пользователей и соответствие требованиям действующих стандартов.

Существенную роль играет и изменение ожиданий пациентов, которые всё чаще ориентируются на цифровые сервисы и ожидают удобных инструментов для взаимодействия с медицинским учреждением. Возможность самостоятельной записи на приём, просмотра истории посещений, получения медицинских документов и оперативной коммуникации с врачом становится важным элементом качества обслуживания. Отсутствие таких возможностей снижает конкурентоспособность учреждения и ухудшает пользовательский опыт.

Анализ существующих решений показывает, что многие из них либо избыточны по функциональности и стоимости для небольших и средних учреждений, либо недостаточно гибки и не позволяют адаптировать систему под конкретные процессы. Это создаёт потребность в разработке программного продукта, который сочетал бы современную архитектуру, удобный интерфейс, возможность масштабирования и адаптации, а также поддержку ключевых процессов медицинского учреждения.

Таким образом, разработка информационной системы для учреждения здравоохранения является актуальной задачей, направленной на повышение эффективности работы персонала, улучшение качества обслуживания пациентов, обеспечение безопасности данных и создание единого цифрового пространства для всех участников медицинского процесса. Реализация такой системы позволит оптимизировать внутренние процессы, снизить нагрузку на сотрудников, повысить прозрачность деятельности и обеспечить соответствие современным требованиям отрасли.

1.1.3 Постановка задачи

Постановка задачи является ключевым этапом разработки программного обеспечения, поскольку она определяет цели, границы и ожидаемые результаты проекта. На основании анализа предметной области и выявленных проблем необходимо сформулировать конкретные направления, в рамках которых должна быть создана информационная система учреждения здравоохранения. Основная задача разработки заключается в создании программного продукта, обеспечивающего автоматизацию ключевых процессов медицинского учреждения, повышение эффективности работы персонала и улучшение качества взаимодействия с пациентами.

В рамках поставленной задачи требуется обеспечить возможность централизованного хранения и обработки медицинских данных, включая сведения о пациентах, расписания врачей, записи на приём, результаты обследований и медицинские документы. Система должна предоставлять удобные инструменты для работы врачей, позволяя им оперативно вносить медицинские записи, просматривать историю обращений и управлять собственным расписанием. Пациентам необходимо предоставить доступ к личному кабинету, где они смогут записываться на приём, просматривать историю посещений и получать медицинские документы. Административный персонал должен иметь возможность управлять пользователями, контролировать расписания, формировать отчётность и обеспечивать корректность данных.

Постановка задачи также включает определение требований к архитектуре и технологической основе системы. Программный продукт должен быть реализован в виде веб-ориентированного решения, обеспечивающего доступ с различных устройств и поддерживающего кроссплатформенность. Важным аспектом является обеспечение безопасности персональных данных, что требует реализации механизмов аутентификации, авторизации, разграничения прав доступа и защиты информации от несанкционированного вмешательства. Система должна обладать высокой производительностью, устойчивостью к нагрузкам и возможностью масштабирования по мере роста количества пользователей и объёма данных.

Таким образом, задача разработки заключается в создании современной, удобной и надёжной информационной системы, способной обеспечить автоматизацию основных процессов учреждения здравоохранения, повысить качество обслуживания пациентов и улучшить организацию работы персонала. Решение данной задачи предполагает последовательное выполнение этапов анализа, проектирования, реализации, тестирования и внедрения программного продукта, что определяет структуру и содержание последующих разделов работы.

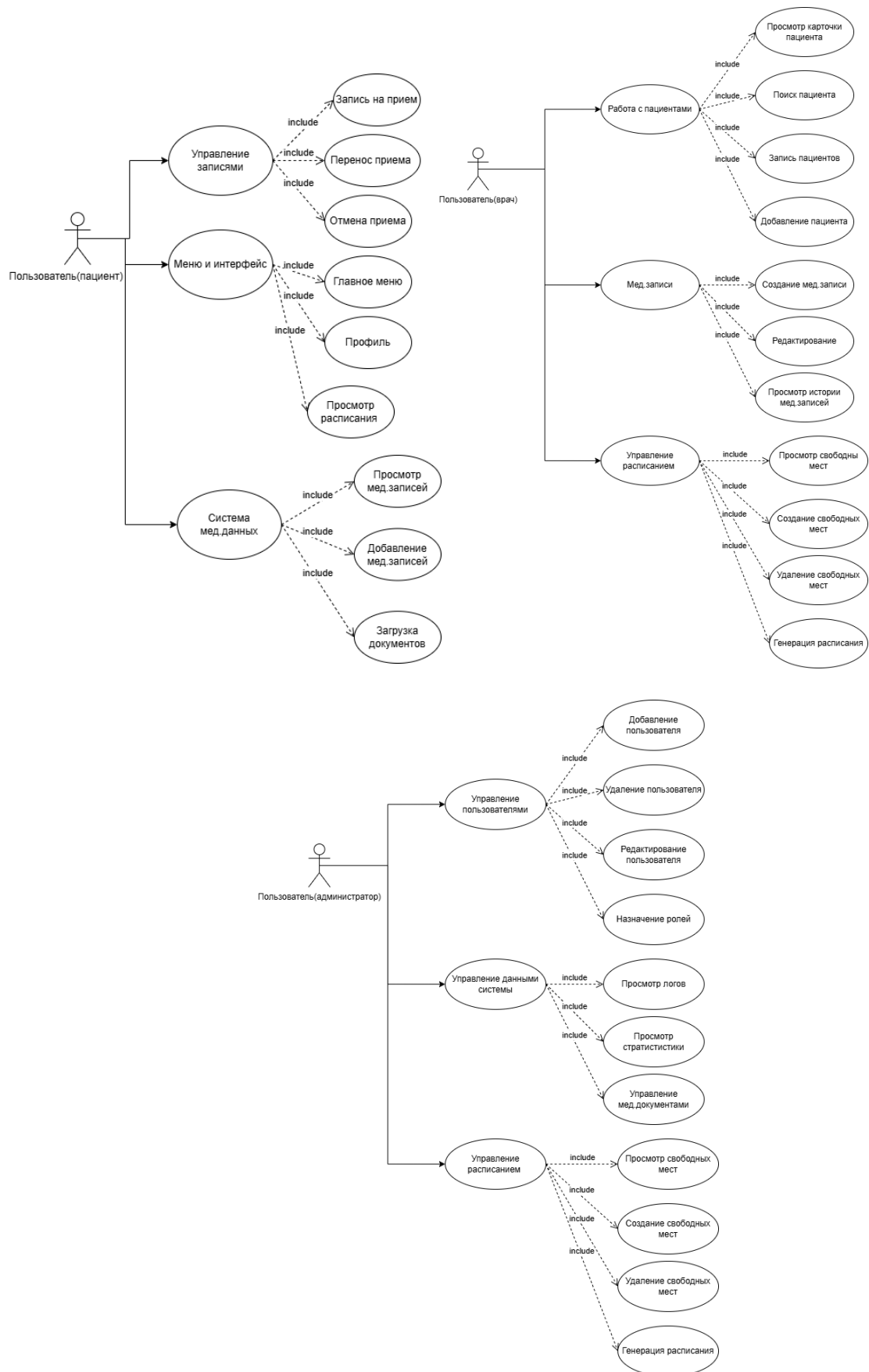


Рисунок 4 – UML-диаграмма вариантов использования

1.1.4 Выбор технологий и инструментов

Выбор технологий является ключевым этапом разработки информационной системы учреждения здравоохранения, поскольку именно технологическая основа определяет производительность, масштабируемость, удобство сопровождения и возможность дальнейшего развития программного продукта. При формировании технологического стека необходимо учитывать специфику предметной области, требования к безопасности медицинских данных, особенности взаимодействия различных категорий пользователей, а также необходимость обеспечения стабильной работы системы при увеличении нагрузки.

Одним из важнейших критериев выбора технологий является архитектурный подход. Для разработки информационной системы была выбрана клиент-серверная архитектура, предполагающая разделение логики на серверную и клиентскую части. Такой подход обеспечивает гибкость, возможность независимого масштабирования компонентов и удобство обновления функциональности без необходимости вмешательства в работу пользователей. Кроме того, он позволяет реализовать чёткое разграничение прав доступа и обеспечить защиту данных на уровне серверной логики.

Сравнительный анализ серверных платформ. Для реализации серверной части рассматривались три популярных фреймворка экосистемы Python: Django, Flask и FastAPI. Django представляет собой тяжёлый фреймворк, предоставляющий множество встроенных компонентов, таких как ORM, админ-панель и система аутентификации. Однако его производительность в асинхронном режиме ограничена, а избыточность функционала приводит к дополнительным накладным расходам при обработке запросов. Flask, напротив, является лёгким микрофреймворком, дающим разработчику свободу в выборе компонентов, однако он не поддерживает асинхронность из коробки и требует подключения сторонних расширений для валидации данных и генерации документации, что замедляет разработку.

FastAPI был выбран в качестве оптимального решения благодаря сочетанию высокой производительности, сопоставимой с фреймворками на Node.js и Go, полной поддержке асинхронных операций, что критически важно при работе с базой данных и файловым хранилищем, а также встроенной валидации данных на основе Pydantic и автоматической генерации документации OpenAPI. Эти особенности позволяют значительно ускорить разработку и упростить интеграцию с внешними сервисами, что особенно важно в условиях ограниченных сроков выполнения дипломной работы.

Сравнительный анализ систем управления базами данных. В качестве СУБД рассматривались MySQL, PostgreSQL и SQLite. SQLite был исключён из-за отсутствия поддержки многопользовательского режима и ограниченной производительности при высоких нагрузках, характерных для работы медицинского учреждения. MySQL является распространённым решением, однако его поддержка сложных запросов и соблюдение стандартов SQL уступают PostgreSQL. Выбор был сделан в пользу PostgreSQL, поскольку данная СУБД обеспечивает строгую целостность данных через механизм внешних ключей и транзакций, поддерживает сложные аналитические запросы, необходимые для формирования отчётности, и обладает расширяемой архитектурой, позволяющей подключать специализированные расширения, например для полнотекстового поиска по медицинским записям. Кроме того, PostgreSQL демонстрирует стабильную производительность при росте объёма данных до миллионов записей, что гарантирует долгосрочную эксплуатацию системы без необходимости миграции на другую платформу.

Сравнительный анализ технологий клиентской части. Для разработки пользовательского интерфейса рассматривались три подхода: чистый JavaScript с прямым манипулированием DOM, фреймворк Angular и библиотека React. Чистый JavaScript был отвергнут из-за сложности поддержки и переиспользования кода по мере роста функциональности системы, а также трудоёмкости реализации динамического обновления данных.

React был выбран благодаря компонентному подходу, позволяющему создавать независимые, легко тестируемые и переиспользуемые элементы интерфейса, такие как карточки приёмов, формы записи и таблицы расписания. Благодаря виртуальному DOM React обеспечивает высокую производительность обновлений интерфейса даже при интенсивной работе пользователя, а огромная экосистема библиотек и инструментов (React Router для маршрутизации, Axios для HTTP-запросов) ускоряет разработку. Кроме того, React позволяет создавать кроссплатформенные приложения, что даёт возможность в будущем развернуть мобильную версию системы на базе React Native без полной перезаписи кода.

Формат обмена данными и взаимодействие компонентов. Для обмена данными между клиентской и серверной частями используется формат JSON, ставший стандартом де-факто в веб-разработке. При его выборе также рассматривался XML, однако JSON обеспечивает меньший объём передаваемых данных, более высокую скорость сериализации и естественную совместимость с JavaScript, что критически важно для React-приложения. Взаимодействие организовано через REST API, что обеспечивает модульность и возможность независимого обновления клиентской и серверной частей. Дополнительно для защиты данных используется JWT-аутентификация, а передача информации осуществляется по протоколу HTTPS, что исключает перехват и подмену данных при передаче через открытые сети. Такой технологический стек обеспечивает баланс между производительностью, безопасностью, скоростью разработки и возможностью масштабирования в будущем.

1.1.5 Выбор серверной платформы

В качестве серверной платформы был выбран фреймворк FastAPI, основанный на языке программирования Python. Данный выбор обусловлен высокой производительностью, поддержкой асинхронных операций, удобством разработки REST-ориентированных API и наличием встроенных механизмов валидации данных. FastAPI обеспечивает автоматическую генерацию документации, что упрощает взаимодействие между разработчиками и способствует повышению качества кода. Кроме того, Python обладает широкой экосистемой библиотек, что позволяет эффективно решать задачи обработки данных и интеграции с внешними сервисами.

Для обоснования выбора FastAPI был проведён сравнительный анализ с альтернативными серверными платформами, наиболее распространёнными при разработке веб-приложений на языке Python: Django REST Framework и Flask. Каждая из этих платформ имеет свои сильные и слабые стороны, однако комплекс требований, предъявляемых к информационной системе учреждения здравоохранения, определил выбор в пользу FastAPI.

Django REST Framework представляет собой мощное и многофункциональное решение, построенное на основе полноценного веб-фреймворка Django. Он предоставляет разработчику обширный набор встроенных инструментов: сериализаторы для преобразования данных, удобную систему маршрутизации, встроенную админ-панель, механизмы аутентификации и авторизации, а также средства для пагинации и фильтрации ответов. Однако при всех своих достоинствах Django обладает рядом недостатков, ограничивающих его применение в данном проекте. Прежде всего, это сравнительно низкая производительность при обработке большого количества параллельных запросов, обусловленная синхронной моделью обработки. Несмотря на появление асинхронной поддержки в последних версиях, она не является полностью зрелой и требует осторожного использования.

Кроме того, Django характеризуется высокой степенью связанности компонентов, что затрудняет тонкую настройку поведения системы и приводит к избыточности кода — многие функции, предоставляемые фреймворком, не используются в проекте, однако их наличие создаёт дополнительную нагрузку на сервер. В контексте медицинской информационной системы, где критически важна скорость ответа при записи пациентов на приём в часы пиковой нагрузки, производительность Django может оказаться недостаточной.

Flask, в отличие от Django, является микрофреймворком, предоставляющим минимальный набор функций. Его главное преимущество — гибкость и свобода выбора компонентов. Разработчик самостоятельно решает, какие библиотеки использовать для валидации данных, работы с базой данных, аутентификации и других задач. Однако эта гибкость оборачивается и существенными издержками. Для создания полноценного REST API с нуля на Flask потребуется интегрировать несколько сторонних расширений: Marshmallow или Pydantic для валидации, Flask-SQLAlchemy для работы с базой данных, Flask-JWT-Extended для аутентификации, Flask-RESTX или аналоги для генерации документации. Такой подход усложняет архитектуру проекта, повышает риск конфликтов версий и несовместимости библиотек, а также замедляет разработку, так как разработчику приходится тратить время на настройку и согласование различных компонентов. Кроме того, Flask изначально не поддерживает асинхронные обработчики, что снижает эффективность при операциях ввода-вывода, таких как обращение к базе данных, отправка электронных писем или загрузка файлов. В медицинской системе, где многие операции требуют ожидания ответа от внешних сервисов, отсутствие асинхронности может привести к снижению пропускной способности сервера.

FastAPI был разработан специально для преодоления ограничений существующих фреймворков. Он построен на основе стандарта ASGI (Asynchronous Server Gateway Interface), который является эволюционным развитием WSGI и обеспечивает полноценную поддержку асинхронных операций.

Это означает, что при ожидании ответа от базы данных, файлового хранилища или внешнего API сервер не блокируется, а продолжает обрабатывать другие запросы. Такая модель особенно актуальна для медицинской информационной системы, где одновременно может выполняться множество операций: пациенты записываются на приём, врачи просматривают истории болезней, администраторы формируют отчёты, а система отправляет уведомления по электронной почте. Благодаря асинхронности FastAPI способен обрабатывать десятки тысяч запросов в секунду, демонстрируя производительность на уровне фреймворков для Node.js и Go, что многократно превосходит показатели Django и Flask.

Ещё одним ключевым преимуществом FastAPI является встроенная валидация данных на основе библиотеки Pydantic. Разработчик описывает структуры данных в виде Python-классов с аннотациями типов, наследуемых от `pydantic.BaseModel`. После этого фреймворк автоматически проверяет соответствие входящих JSON-запросов ожидаемым схемам, выполняет преобразование типов (например, строки в даты или целые числа) и возвращает понятные сообщения об ошибках с указанием конкретного поля и причины несоответствия. Для медицинской системы, где корректность данных имеет первостепенное значение, эта функциональность критически важна. Например, при записи на приём система должна гарантировать, что переданный идентификатор врача является целым числом, дата и время имеют правильный формат, а текст жалобы не превышает допустимую длину. Pydantic также поддерживает сложные правила валидации с использованием декоратора `@validator`, что позволяет реализовать пользовательские проверки, например, на то, что дата приёма не может быть в прошлом.

FastAPI также обеспечивает автоматическую генерацию интерактивной документации API в форматах OpenAPI и JSON Schema.

При запуске сервера документация становится доступной по эндпоинту /docs, где с помощью удобного веб-интерфейса Swagger UI можно наглядно изучить все доступные маршруты, ожидаемые форматы запросов, возможные коды ответов и примеры данных. Дополнительно документация в формате ReDoc доступна по эндпоинту /redoc. Такая функция существенно упрощает взаимодействие между разработчиком серверной части и разработчиком клиентского интерфейса, а также ускоряет процесс отладки и тестирования. При изменении API документация обновляется автоматически, что исключает риск расхождения между реальным поведением системы и её описанием.

Важным фактором, повлиявшим на выбор серверной платформы, является экосистема языка Python. Python обладает огромным количеством библиотек для решения самых разнообразных задач, что позволяет значительно ускорить разработку. Для работы с базой данных используется SQLAlchemy — мощный ORM-фреймворк, поддерживающий множество СУБД, включая PostgreSQL, и предоставляющий удобный интерфейс для выполнения запросов и управления миграциями через Alembic. Для обработки медицинских данных могут потребоваться библиотеки pandas для анализа статистики, matplotlib или plotly для визуализации отчётов, а в перспективе — scikit-learn для построения прогностических моделей, например, для предсказания загруженности врачей или выявления групп риска среди пациентов. Интеграция всех этих инструментов с FastAPI не представляет сложности, так как фреймворк спроектирован с учётом принципов модульности и не навязывает разработчику конкретных реализаций.

Отдельного внимания заслуживает удобство разработки на FastAPI. Благодаря использованию современных возможностей Python, таких как аннотации типов (type hints) и асинхронные функции (async/await), код получается чистым, читаемым и самодокументируемым. IDE и редакторы кода, поддерживающие анализ типов (например, PyCharm или Visual Studio Code с плагином Pylance), способны автоматически подсказывать атрибуты объектов, проверять корректность вызовов и выполнять рефакторинг, что снижает вероятность возникновения ошибок на этапе написания кода.

					ПГУ 1.090501.16.001 ПЗ	Лист
						25
Изм.	Лист	№ докум.	Подпись	Дата		

Кроме того, FastAPI имеет подробную и хорошо структурированную документацию с множеством примеров, что ускоряет процесс изучения фреймворка и решения типовых задач.

Таким образом, выбор FastAPI в качестве серверной платформы является оптимальным решением для разрабатываемой информационной системы учреждения здравоохранения. Данный фреймворк обеспечивает высокую производительность, необходимую для работы с большим количеством одновременных пользователей, надёжную асинхронную обработку запросов, строгую и гибкую валидацию данных, критически важную для медицинской информации, автоматическую генерацию документации, упрощающую разработку и тестирование, а также seamless-интеграцию с богатой экосистемой Python-библиотек. Совокупность перечисленных преимуществ делает FastAPI наиболее подходящим инструментом для реализации серверной части веб-приложения «Информационная система учреждения здравоохранения».

1.1.6 Выбор системы управления базами данных

Для хранения данных была выбрана реляционная система управления базами данных PostgreSQL, отличающаяся надёжностью, высокой производительностью и поддержкой сложных запросов. PostgreSQL обеспечивает строгую целостность данных, что особенно важно при работе с медицинской информацией. Возможность использования внешних ключей, транзакций и индексов позволяет эффективно организовать структуру данных и обеспечить стабильную работу системы при увеличении объёма информации. Дополнительным преимуществом является поддержка расширений, позволяющих адаптировать базу данных под специфические задачи.

Для обоснования выбора PostgreSQL был проведён сравнительный анализ с альтернативными системами управления базами данных, наиболее распространёнными при разработке веб-приложений: MySQL и SQLite. Каждая из этих СУБД имеет свои достоинства и недостатки, однако комплекс требований, предъявляемых к информационной системе учреждения здравоохранения, определил выбор в пользу PostgreSQL.

Сравнительный анализ с SQLite. SQLite представляет собой встраиваемую реляционную базу данных, хранящую всю информацию в одном файле и не требующую отдельного серверного процесса. Она широко используется в мобильных приложениях, десктопных программах и небольших веб-проектах благодаря простоте настройки и нулевой конфигурации. Однако SQLite обладает рядом ограничений, делающих её непригодной для использования в медицинской информационной системе. Прежде всего, SQLite не поддерживает многопользовательский режим на запись в полной мере: при одновременной попытке нескольких пользователей изменить данные возникает блокировка файла базы данных, что приводит к задержкам и ошибкам. В условиях медицинского учреждения, где одновременно работают десятки пациентов, врачей и администраторов, это неприемлемо. Кроме того, SQLite имеет ограниченную поддержку сложных типов данных и не обеспечивает строгого контроля целостности на уровне внешних ключей (хотя формально эта функция присутствует, она отключена по умолчанию и требует явного включения). Производительность SQLite при работе с большими объёмами данных (более нескольких гигабайт) значительно снижается, а отсутствие средств для репликации и резервного копирования делает систему уязвимой при сбоях. Таким образом, SQLite была исключена из рассмотрения как не соответствующая требованиям к масштабируемости, многопользовательской работе и надёжности.

Сравнительный анализ с MySQL. MySQL является одной из самых популярных реляционных СУБД в мире, особенно в связке с веб-приложениями. Она демонстрирует высокую производительность при операциях чтения и обладает простотой настройки. Однако при детальном сравнении с PostgreSQL выявляется ряд различий, критически важных для медицинской информационной системы.

Первое различие касается соблюдения стандартов SQL. PostgreSQL стремится к максимальной совместимости со стандартом ANSI SQL, что облегчает переносимость запросов между различными системами и упрощает обучение разработчиков. MySQL, напротив, исторически отступала от стандартов, реализуя многие функции по-своему, что может привести к неожиданному поведению при сложных запросах. Второе различие связано с поддержкой сложных типов данных. PostgreSQL из коробки поддерживает массивы, JSONB (бинарный JSON с возможностью индексации), составные типы, перечисления, интервалы времени, географические типы (через PostGIS) и другие. JSONB представляет особую ценность для медицинской системы, так как позволяет хранить полуструктурированные данные, например, результаты лабораторных анализов с переменным набором показателей, непосредственно в таблице, обеспечивая при этом возможность эффективного поиска и индексации. MySQL также поддерживает JSON, однако его реализация менее производительна и обладает ограниченными возможностями индексации.

Третье различие касается механизмов обеспечения целостности данных. PostgreSQL имеет более строгую и надёжную реализацию внешних ключей, проверочных ограничений и триггеров. В MySQL, особенно при использовании движка MyISAM (который до сих пор встречается в устаревших проектах), внешние ключи не поддерживаются вовсе. Даже при использовании движка InnoDB, который поддерживает внешние ключи, поведение при каскадных операциях может отличаться от ожидаемого согласно стандарту.

Для медицинской системы, где целостность данных критически важна — запись на приём должна ссылаться на существующего пациента и врача, а медицинская запись — на соответствующий приём, — использование СУБД с надёжной реализацией внешних ключей является обязательным требованием.

Четвёртое различие связано с производительностью при сложных аналитических запросах. PostgreSQL имеет более продвинутый оптимизатор запросов, способный эффективно обрабатывать многотабличные соединения, подзапросы и оконные функции. Это особенно важно для формирования отчётности, которая может требовать агрегации данных из таблиц приёмов, пациентов, врачей и медицинских записей одновременно. MySQL также поддерживает подобные операции, однако на сложных запросах с большим количеством соединений производительность PostgreSQL часто оказывается выше.

Преимущества PostgreSQL для медицинской информационной системы. Выбор PostgreSQL обусловлен не только его превосходством над альтернативами по ряду параметров, но и наличием специфических возможностей, востребованных в медицинской предметной области.

Строгая целостность данных реализуется через механизм внешних ключей, который гарантирует, что ссылочная целостность не может быть нарушена даже в случае программной ошибки или сбоя. Например, если в базе данных присутствует запись о приёме, ссылающаяся на определённого пациента, удалить этого пациента невозможно до тех пор, пока не будут удалены все связанные с ним приёмы, либо не будет настроено каскадное удаление. Это предотвращает возникновение «висячих» записей, которые могли бы привести к некорректной работе системы. Кроме того, PostgreSQL поддерживает проверочные ограничения (CHECK), позволяющие на уровне базы данных задать правила допустимости значений.

Например, можно задать, что дата начала приёма должна быть меньше даты окончания, а статус приёма может принимать только одно из допустимых значений (запланирован, подтверждён, завершён, отменён). Такие ограничения работают независимо от того, через какой интерфейс производится изменение данных, что обеспечивает дополнительный уровень защиты от ошибок.

Поддержка транзакций с соблюдением свойств ACID (атомарность, согласованность, изоляция, долговечность) гарантирует, что даже при сбое питания или программном сбое база данных не окажется в противоречивом состоянии. Например, при записи пациента на приём система может выполнить несколько операций: создать запись в таблице приёмов, обновить статус временного слота, отправить уведомление. Если любая из этих операций завершится ошибкой, все изменения, выполненные в рамках транзакции, будут отменены (откат), и база данных вернётся в исходное согласованное состояние. Такое поведение критически важно для медицинских данных, где недопустима ситуация, когда пациент считает себя записанным, а в расписании врача слот остался свободным, или наоборот.

Механизм индексов позволяет значительно ускорить выполнение поисковых запросов. В медицинской системе часто выполняются запросы на поиск пациентов по фамилии, дате рождения или полису, а также запросы на получение расписания врача на определённую дату. Создание соответствующих индексов (B-tree для точного сравнения и диапазонного поиска, GIN для полнотекстового поиска по жалобам и диагнозам) позволяет выполнять такие запросы за миллисекунды даже при наличии миллионов записей. PostgreSQL поддерживает несколько типов индексов, включая частичные индексы (индексируются только строки, удовлетворяющие условию), составные индексы (по нескольким полям одновременно) и индексы на выражениях (например, на функции приведения даты к формату без времени).

Поддержка расширений является ещё одним весомым аргументом в пользу PostgreSQL. Расширение PostGIS добавляет поддержку географических объектов и пространственных запросов, что может быть полезно для привязки пациентов к территориальным поликлиникам или расчёта времени в пути до учреждения. Расширение pg_trgm обеспечивает эффективный поиск по нечёткому совпадению строк, что полезно при поиске пациентов в случае опечаток в фамилии. Расширение hstore позволяет хранить пары «ключ-значение» в одной колонке, что может использоваться для хранения расширенных атрибутов пациентов или параметров исследований. Расширение pg_cron даёт возможность выполнять фоновые задачи внутри базы данных, например, ежедневное формирование сводных отчётов или автоматическую отмену устаревших записей. Возможность добавлять новые расширения по мере необходимости делает PostgreSQL гибкой платформой, способной адаптироваться к изменяющимся требованиям медицинского учреждения.

Масштабируемость и производительность. PostgreSQL способна эффективно работать с базами данных размером в терабайты и более, обслуживая тысячи одновременных подключений. В контексте медицинской информационной системы это означает, что даже при значительном росте количества пациентов, врачей и приёмов система не потребует немедленной замены СУБД. Поддержка репликации (как синхронной, так и асинхронной) позволяет создавать резервные копии базы данных в реальном времени и распределять нагрузку на чтение между несколькими серверами. Функции резервного копирования, включая Point-in-Time Recovery (PITR), дают возможность восстановить базу данных на любой момент времени, что критически важно для медицинских учреждений, обязанных хранить историю обращений пациентов в течение многих лет. Средства мониторинга и оптимизации производительности, такие как расширение pg_stat_statements, позволяют выявлять медленные запросы и оптимизировать их, что способствует поддержанию высокой скорости работы системы в долгосрочной перспективе.

Таким образом, выбор PostgreSQL в качестве системы управления базами данных для разрабатываемой информационной системы учреждения здравоохранения обусловлен комплексом факторов: строгой целостностью данных, соответствующей высоким требованиям к хранению медицинской информации; поддержкой сложных транзакций, гарантирующих согласованность данных при любых сбоях; мощным оптимизатором запросов и механизмом индексации, обеспечивающим высокую производительность; развитой системой расширений, позволяющей адаптировать СУБД под специфические задачи, а также надёжными средствами резервного копирования и масштабирования. Совокупность перечисленных преимуществ делает PostgreSQL наиболее подходящей СУБД для реализации веб-приложения «Информационная система учреждения здравоохранения».

1.1.7 Выбор технологий клиентской части

Клиентская часть системы реализована с использованием библиотеки React, которая предоставляет удобные инструменты для создания динамичных пользовательских интерфейсов. React позволяет организовать интерфейс в виде набора компонентов, что упрощает разработку, тестирование и повторное использование элементов. Благодаря виртуальному DOM обеспечивается высокая производительность интерфейса, а поддержка современных инструментов разработки позволяет создавать удобные и интуитивно понятные пользовательские сценарии. Использование React также обеспечивает кроссплатформенность, позволяя пользователям работать с системой на различных устройствах.

Для обоснования выбора React был проведён сравнительный анализ с альтернативными подходами к разработке пользовательских интерфейсов, наиболее распространёнными в современной веб-разработке: нативный JavaScript с прямым манипулированием DOM, фреймворк Angular и библиотека Vue.js. Каждый из этих подходов имеет свои сильные и слабые стороны, однако комплекс требований, предъявляемых к информационной системе учреждения здравоохранения, определил выбор в пользу React.

					ПГУ 1.090501.16.001 ПЗ	Лист
						32
Изм.	Лист	№ докум.	Подпись	Дата		

Сравнение с нативным JavaScript. Разработка интерфейса на чистом JavaScript без использования дополнительных библиотек и фреймворков возможна и в некоторых случаях оправдана для простых проектов с ограниченной функциональностью. Такой подход даёт полный контроль над кодом и избавляет от необходимости изучать сторонние инструменты. Однако при разработке медицинской информационной системы со сложной логикой, множеством интерактивных элементов и динамически обновляемыми данными нативный JavaScript сопряжён с рядом серьёзных проблем. Прежде всего, это сложность поддержки и масштабирования кода. По мере роста проекта количество манипуляций с DOM-элементами увеличивается, что приводит к появлению «спагетти-кода», в котором логика отображения, обработки событий и управления данными оказывается перемешанной. Разработчику приходится вручную отслеживать изменения состояния интерфейса, обновлять соответствующие элементы при получении новых данных с сервера и синхронизировать состояние между различными частями страницы. Такой подход не только трудоёмкий, но и чреват ошибками, когда какое-либо обновление интерфейса оказывается пропущенным или выполненным некорректно. Кроме того, производительность нативного JavaScript при частых обновлениях больших фрагментов интерфейса может быть низкой, поскольку каждое изменение DOM может вызывать перерисовку элементов браузером. В React эта проблема решается с помощью виртуального DOM, который будет рассмотрен далее. Таким образом, использование нативного JavaScript для разработки медицинской системы признано нецелесообразным из-за сложности поддержки, риска ошибок и низкой производительности при интенсивном взаимодействии пользователя с интерфейсом.

Сравнение с Angular. Angular представляет собой полноценный фреймворк, разрабатываемый и поддерживаемый компанией Google. Он предлагает комплексное решение для создания веб-приложений, включая двухстороннее связывание данных, внедрение зависимостей, маршрутизацию, работу с формами и HTTP-запросами, а также использование языка TypeScript в качестве основного. Angular имеет множество достоинств, однако в контексте данной работы его использование было признано менее предпочтительным по нескольким причинам. Во-первых, Angular характеризуется крутой кривой обучения. Разработчику необходимо освоить такие концепции, как модули, компоненты, сервисы, директивы, пайпы, RxJS для работы с асинхронными потоками и внедрение зависимостей. Для относительно небольшой команды (в данном случае — одного разработчика) это может привести к неоправданным временным затратам на изучение и отладку. Во-вторых, Angular является тяжёлым фреймворком: размер итогового бандла приложения даже с минимальной функциональностью значительно превышает размер аналогичного приложения на React. Это негативно сказывается на времени загрузки страницы, особенно при использовании мобильного интернета или устаревших устройств. В-третьих, Angular навязывает определённую архитектуру и стиль написания кода, что может быть избыточным для проекта, не требующего всех возможностей этого фреймворка. В то время как React предоставляет разработчику свободу выбора дополнительных библиотек для маршрутизации, управления состоянием и выполнения HTTP-запросов, Angular включает все эти компоненты «из коробки», что увеличивает сложность и снижает гибкость.

Сравнение с Vue.js. Vue.js является прогрессивным фреймворком, который часто сравнивают с React из-за схожего подхода к компонентной архитектуре и использования виртуального DOM. Vue.js отличается более низким порогом входа благодаря интуитивно понятному синтаксису и подробной документации.

Однако в пользу React был сделан выбор по нескольким причинам. Прежде всего, React имеет значительно большую экосистему и сообщество. Это означает, что для React существует больше сторонних библиотек, компонентов и инструментов, а также проще найти готовые решения типовых задач и ответы на возникающие вопросы. Для медицинской системы, где требуется, например, интеграция с календарём для выбора даты приёма или построение графиков для визуализации статистики, наличие проверенных и поддерживаемых библиотек (React Big Calendar, Recharts, Material-UI) существенно ускоряет разработку. Во-вторых, React обладает более высокой популярностью на рынке труда и среди разработчиков, что облегчает поддержку проекта в будущем и возможную передачу системы другому специалисту. В-третьих, React активно развивается компанией Meta (ранее Facebook) и крупным сообществом, что гарантирует долгосрочную поддержку и регулярные обновления с новыми возможностями и исправлениями ошибок. Несмотря на то, что Vue.js также является зрелым и надёжным решением, совокупность перечисленных факторов определила выбор в пользу React.

Компонентная архитектура React. Основной концепцией React является представление пользовательского интерфейса в виде композиции независимых, переиспользуемых компонентов. Компонент — это часть интерфейса, которая инкапсулирует в себе разметку (JSX), логику и стили. Такой подход обладает рядом существенных преимуществ.

Во-первых, компоненты способствуют повторному использованию кода. Например, карточка приёма, отображающая информацию о враче, дате, времени и статусе, может быть использована как в личном кабинете пациента для отображения его записей, так и в кабинете врача для отображения расписания.

Во-вторых, компоненты упрощают тестирование, поскольку каждый компонент можно тестировать изолированно от остальной части приложения.

В-третьих, компонентная архитектура улучшает читаемость и поддерживаемость кода. Разработчик, видя структуру компонентов, может быстро понять, какая часть интерфейса за какую функциональность отвечает. В-четвёртых, компоненты могут быть организованы в иерархию, что позволяет передавать данные от родительских компонентов к дочерним через свойства (props) и управлять состоянием на различных уровнях.

Виртуальный DOM. Одной из ключевых особенностей React, обеспечивающих высокую производительность, является использование виртуальной модели документа (Virtual DOM). При традиционной работе с DOM каждое изменение элемента интерфейса (например, изменение текста или добавление новой строки в таблицу) вызывает прямое обращение браузера к реальному DOM-дереву, что может быть затратной операцией, особенно при частых обновлениях. React работает иначе: при изменении состояния компонента обновляется не реальный DOM, а его лёгкая копия — виртуальный DOM, хранящаяся в памяти. Затем React выполняет сравнение (diffing) новой версии виртуального DOM с предыдущей, определяя минимальный набор изменений, которые необходимо применить к реальному DOM. После этого React применяет только эти изменения, а не перерисовывает весь интерфейс целиком. Такой подход позволяет добиться высокой производительности даже при интенсивном взаимодействии пользователя с системой, что особенно важно для медицинской информационной системы, где одновременно могут отображаться большие таблицы с расписаниями, списками пациентов и историями болезней. Например, при фильтрации списка приёмов по специальности врача React обновит только те строки таблицы, которые изменились, а не перерисует всю таблицу заново.

Кроссплатформенность и адаптивность. Использование React позволяет создавать интерфейсы, одинаково хорошо работающие на различных устройствах: персональных компьютерах, ноутбуках, планшетах и смартфонах. Это достигается за счёт использования каскадных таблиц стилей (CSS) с медиа-запросами, позволяющими адаптировать отображение компонентов под размер экрана, а также за счёт возможности интеграции с библиотеками для построения адаптивного дизайна, такими как Material-UI или React Bootstrap. В контексте медицинской системы это особенно важно, поскольку пациенты могут записываться на приём как с домашнего компьютера, так и с мобильного телефона, находясь в дороге. Врачи и администраторы, в свою очередь, преимущественно работают со стационарными компьютерами, но возможность использования планшета при обходе палат также может оказаться востребованной. Кроме того, React позволяет создавать нативные мобильные приложения с помощью React Native без необходимости полной перезаписи логики. Хотя в рамках данной работы мобильное приложение не разрабатывается, потенциальная возможность его создания в будущем делает React стратегически обоснованным выбором.

Экосистема и инструменты разработки. React обладает одной из самых богатых экосистем среди всех фронтенд-библиотек. Для маршрутизации между страницами используется React Router — стандартное решение, позволяющее организовать навигацию в одностраничном приложении (Single Page Application). Для выполнения HTTP-запросов к серверной части применяются библиотеки Axios или встроенный Fetch API.

Для управления глобальным состоянием приложения (например, данными об авторизованном пользователе) могут использоваться Context API (встроенный в React) или Redux (для более сложных сценариев). Для работы с формами и валидации ввода доступны библиотеки Formik и React Hook Form. Для визуализации данных и построения графиков — Recharts и Victory. Для создания календарей и выбора дат — React Big Calendar и React DatePicker.

Такое разнообразие инструментов позволяет разработчику выбирать оптимальные решения для каждой конкретной задачи, не тратя время на реализацию базовых механизмов с нуля.

Важным преимуществом React является наличие удобных инструментов для разработки и отладки. React Developer Tools — это расширение для браузеров Chrome и Firefox, которое позволяет инспектировать иерархию компонентов, просматривать их свойства и состояние, а также отслеживать производительность обновлений. В сочетании с возможностями современных браузеров по отладке JavaScript и анализу сетевых запросов это даёт разработчику мощный арсенал для выявления и исправления ошибок. Кроме того, инструменты сборки, такие как Vite или Create React App, позволяют быстро инициализировать проект с уже настроенной конфигурацией, включая поддержку JSX, горячую перезагрузку модулей (Hot Module Replacement), которая позволяет видеть изменения кода в реальном времени без перезагрузки страницы, и оптимизацию итогового бандла для production-окружения.

Обработка ошибок и пользовательский опыт. React предоставляет удобные механизмы для обработки ошибок на уровне компонентов. Error Boundaries (границы ошибок) позволяют перехватывать ошибки, возникающие в дочерних компонентах, и отображать запасной интерфейс (например, сообщение о том, что что-то пошло не так), не допуская полного краха всего приложения. Это особенно важно для медицинской системы, где сбой в одном модуле (например, при отображении сложного графика) не должен приводить к недоступности других функций, таких как запись на приём или просмотр истории болезней. Кроме того, сочетание React с библиотеками для отображения уведомлений (react-toastify) позволяет создавать информативные и ненавязчивые сообщения об ошибках и успешных операциях, улучшая пользовательский опыт.

Таким образом, выбор React в качестве технологии для реализации клиентской части информационной системы учреждения здравоохранения обусловлен рядом факторов: компонентная архитектура, упрощающая разработку, тестирование и повторное использование кода; виртуальный DOM, обеспечивающий высокую производительность при интенсивном взаимодействии с интерфейсом; кроссплатформенность и возможность создания адаптивного дизайна для работы на различных устройствах; богатая экосистема библиотек и инструментов, ускоряющих разработку; удобные средства отладки и обработки ошибок, повышающие надёжность приложения. Совокупность перечисленных преимуществ делает React наиболее подходящим решением для создания современного, удобного и производительного пользовательского интерфейса веб-приложения «Информационная система учреждения здравоохранения».

1.1.8 Формат обмена данными и взаимодействие компонентов

Для обмена данными между клиентом и сервером используется формат JSON (JavaScript Object Notation), который является стандартом де-факто для современных веб-приложений. Такой подход обеспечивает простоту интеграции, удобство обработки данных и совместимость с большинством существующих инструментов. Взаимодействие между компонентами системы осуществляется через REST API, что обеспечивает модульность и возможность расширения функциональности без изменения существующей архитектуры. Особое внимание уделено вопросам безопасности, включая токенную аутентификацию на основе JWT, передачу данных по защищённым каналам и валидацию входящих запросов.

Обоснование выбора формата JSON. При проектировании информационной системы рассматривались два основных формата для обмена данными между клиентской и серверной частями: JSON и XML. XML (Extensible Markup Language) долгое время являлся стандартом для обмена данными в корпоративных системах, особенно в сферах, где требовалась строгая валидация структуры с помощью XSD-схем. Однако JSON был выбран в качестве основного формата по ряду причин.

Во-первых, JSON обладает значительно меньшим объёмом передаваемых данных по сравнению с XML, поскольку не содержит закрывающих тегов и атрибутов. Для медицинской информационной системы, где ежедневно выполняется большое количество запросов (поиск пациентов, получение расписаний, создание приёмов), уменьшение объёма трафика положительно сказывается на скорости работы и снижает нагрузку на сеть. Во-вторых, JSON является родным форматом для JavaScript, что обеспечивает максимальную производительность при обработке данных на клиентской стороне, реализованной на React. В то время как XML требует использования специальных парсеров (например, DOMParser) и преобразования в объекты, JSON достаточно передать в функцию JSON.parse(), что выполняется на порядок быстрее. В-третьих, JSON имеет более простой и читаемый синтаксис, что упрощает отладку при разработке: разработчик может легко просмотреть содержимое ответа сервера в консоли браузера или инструментах разработчика. В-четвёртых, JSON поддерживается всеми современными серверными фреймворками, включая выбранный FastAPI, который обеспечивает автоматическую сериализацию и десериализацию объектов Pydantic в JSON и обратно. Несмотря на то, что XML предоставляет более развитые средства валидации (XSD) и пространства имён, для задач медицинской информационной системы эти преимущества оказались избыточными, а сложность работы с XML не оправдывает возможные выгоды. Таким образом, выбор JSON является оптимальным с точки зрения производительности, удобства разработки и совместимости.

Архитектурный стиль REST. Взаимодействие между клиентской и серверной частями организовано в соответствии с архитектурным стилем REST (Representational State Transfer). REST основывается на нескольких ключевых принципах, каждый из которых вносит вклад в качество разрабатываемой системы.

Первый принцип — клиент-серверная архитектура, которая предполагает чёткое разделение ответственности между компонентами. Клиентская часть отвечает за отображение интерфейса и взаимодействие с пользователем, серверная часть — за хранение данных, выполнение бизнес-логики и обработку запросов. Такое разделение позволяет разрабатывать и масштабировать компоненты независимо друг от друга.

Второй принцип — отсутствие сохранения состояния на сервере (stateless). Каждый запрос от клиента к серверу должен содержать всю необходимую информацию для его обработки, сервер не хранит контекст предыдущих запросов. В контексте медицинской системы это означает, что информация об авторизации передаётся с каждым запросом в виде JWT-токена в заголовке Authorization. Такой подход упрощает масштабирование серверной части: при увеличении нагрузки можно добавить дополнительные экземпляры сервера, и любой из них сможет обработать любой запрос, поскольку не требуется синхронизация сессионных данных между серверами.

Третий принцип — единообразие интерфейса. REST предполагает использование стандартных HTTP-методов для выполнения операций над ресурсами: GET используется для получения данных, POST — для создания новых ресурсов, PUT или PATCH — для обновления, DELETE — для удаления. Такой подход делает API предсказуемым и лёгким в освоении. Например, получение списка пациентов выполняется GET-запросом на /api/patients, создание новой записи на приём — POST-запросом на /api/appointments, обновление статуса приёма — PATCH-запросом на /api/appointments/{id}/status. Статусы HTTP-ответов также используются в соответствии со стандартом: 200 OK означает успешное выполнение, 201 Created — успешное создание ресурса, 400 Bad Request — ошибка в данных запроса, 401 Unauthorized — отсутствие аутентификации, 403 Forbidden — недостаточно прав, 404 Not Found — ресурс не найден.

Четвёртый принцип — идентификация ресурсов с помощью URI. Каждый ресурс (пациент, врач, приём, медицинская запись) имеет уникальный идентификатор, что позволяет клиенту обращаться к конкретному ресурсу. Например, получение информации о пациенте с идентификатором 42 выполняется GET-запросом на `/api/patients/42`. Такая адресация интуитивно понятна и упрощает навигацию по API.

Пятый принцип — представление ресурсов. Сервер передаёт клиенту не сам ресурс (который хранится в базе данных), а его представление в формате JSON. Клиент и сервер могут договориться о формате представления с помощью заголовка `Content-Type`. В разрабатываемой системе для всех запросов и ответов используется `application/json`. Это позволяет при необходимости изменить внутреннюю структуру хранения данных (например, добавить новое поле в таблицу) без изменения клиентского кода — достаточно модифицировать сериализатор на сервере, который преобразует внутреннее представление в JSON.

Документирование REST API. Для документирования разработанного REST API используется встроенный механизм FastAPI, основанный на стандартах OpenAPI и JSON Schema. При запуске сервера автоматически генерируется интерактивная документация, доступная по эндпоинту `/docs`. В этой документации перечислены все доступные маршруты, ожидаемые параметры запросов, форматы тел запросов, возможные коды ответов и примеры. Разработчик клиентской части может прямо из документации выполнять тестовые запросы к серверу, указывая параметры и наблюдая ответы. Это значительно ускоряет процесс интеграции и отладки, поскольку позволяет проверить работу каждого эндпоинта независимо от клиентского интерфейса. Документация всегда актуальна, поскольку генерируется автоматически на основе аннотаций типов и схем Pydantic, что исключает риск расхождения между документацией и реальным поведением API.

					ПГУ 1.090501.16.001 ПЗ	Лист
						42
Изм.	Лист	№ докум.	Подпись	Дата		

Безопасность обмена данными. Безопасности обмена данными уделено особое внимание, поскольку система работает с персональными данными пациентов и медицинской информацией, относящейся к категории чувствительных данных. Защита информации обеспечивается на нескольких уровнях.

На транспортном уровне используется протокол HTTPS (HTTP over SSL/TLS), который обеспечивает шифрование всех передаваемых данных между клиентом и сервером. Это исключает возможность перехвата информации злоумышленником при использовании небезопасных сетей, таких как общедоступный Wi-Fi в поликлинике или кафе. HTTPS также гарантирует аутентификацию сервера с помощью цифровых сертификатов, что предотвращает атаки типа «человек посередине» (Man-in-the-Middle), когда злоумышленник выдаёт себя за легитимный сервер. В production-окружении сервер должен быть настроен с использованием сертификата, подписанного доверенным центром сертификации (например, Let's Encrypt, который предоставляет бесплатные сертификаты).

На уровне аутентификации используется механизм токенов на основе стандарта JWT (JSON Web Token). При успешной аутентификации (входе в систему) сервер генерирует JWT-токен, который содержит три части: заголовок с информацией об алгоритме шифрования, полезную нагрузку (payload) с данными о пользователе (идентификатор, роль, время истечения токена) и цифровую подпись. Полезная нагрузка может включать такие поля, как user_id (идентификатор пользователя), role (роль — patient, doctor, admin), exp (время истечения токена) и iat (время выдачи). Токен подписывается секретным ключом, известным только серверу, что гарантирует его подлинность и неизменность — любая попытка модифицировать данные в токене приведёт к недействительности подписи.

На уровне авторизации реализовано разграничение прав доступа в соответствии с ролями пользователей. После успешной аутентификации сервер извлекает из токена информацию о пользователе и его роли. При обработке каждого запроса сервер проверяет, имеет ли пользователь необходимые права для выполнения запрашиваемой операции. Например, пациент может просматривать только свои собственные медицинские записи, но не записи других пациентов; врач может просматривать записи только своих пациентов; администратор имеет полный доступ ко всем данным. Такая проверка реализована в каждом обработчике маршрута с использованием зависимости `get_current_user`, которая извлекает пользователя из токена, после чего сервисный слой выполняет дополнительные проверки прав. Например, при запросе медицинской записи по идентификатору сервер сначала проверяет, что запрашивающий пользователь является либо самим пациентом, к которому относится запись, либо врачом этого пациента, либо администратором. В случае несоответствия прав сервер возвращает ошибку 403 Forbidden.

На уровне валидации данных реализованы механизмы, предотвращающие некорректные или потенциально опасные запросы. При получении запроса FastAPI с использованием схем Pydantic автоматически проверяет типы и форматы всех полей. Например, если клиент отправляет строку в поле, ожидающее целочисленный идентификатор врача, сервер вернёт ошибку 422 Unprocessable Entity с указанием причины. Кроме того, валидация на уровне Pydantic позволяет проверять бизнес-правила с помощью пользовательских валидаторов. Например, можно реализовать валидацию, проверяющую, что дата записи на приём не может быть в прошлом, а время окончания приёма наступает позже времени начала. Такая валидация выполняется до того, как данные попадут в сервисный слой или базу данных, что предотвращает сохранение некорректных данных. Дополнительно на уровне базы данных настроены проверочные ограничения (CHECK constraints), которые гарантируют целостность данных даже в случае программной ошибки, обходящей валидацию на уровне приложения.

На уровне защиты от распространённых атак реализован ряд мер. Для предотвращения межсайтовой подделки запросов (CSRF) используются JWT-токены, которые не полагаются на куки-сессии, что делает систему неуязвимой для данного типа атак. Для предотвращения межсайтового скриптинга (XSS) клиентская часть экранирует все данные, вставляемые в DOM, что является стандартным поведением React. Для предотвращения SQL-инъекций все запросы к базе данных выполняются через ORM SQLAlchemy, который автоматически экранирует параметры. Для ограничения частоты запросов (rate limiting) на сервере может быть настроено ограничение количества запросов от одного IP-адреса или пользователя за определённый промежуток времени, что предотвращает атаки типа «отказ в обслуживании» (DoS) и перебор паролей методом грубой силы.

Таким образом, выбранный подход к обмену данными и взаимодействию компонентов обеспечивает надёжность, производительность и гибкость разрабатываемой информационной системы. Использование формата JSON гарантирует высокую скорость обработки данных и совместимость с клиентской частью на React. REST API обеспечивает модульность, масштабируемость и единообразие интерфейса. Комплекс мер безопасности, включающий HTTPS-шифрование, JWT-аутентификацию, ролевую авторизацию, валидацию данных и защиту от распространённых атак, позволяет гарантировать конфиденциальность и целостность медицинских данных. Совокупность этих решений создаёт основу для построения масштабируемого и защищённого веб-приложения, способного адаптироваться к изменяющимся требованиям учреждения здравоохранения и обеспечивать высокий уровень качества обслуживания пользователей.

1.2 Проектирование и разработка

Проектирование информационной системы учреждения здравоохранения основано на необходимости обеспечить надёжность, масштабируемость и удобство использования программного продукта. На данном этапе формируется архитектурная модель, определяются ключевые компоненты системы, способы их взаимодействия и требования к структуре данных. Особое внимание уделяется вопросам безопасности, поскольку система работает с персональными и медицинскими данными, требующими строгой защиты. Проектирование направлено на создание гибкой архитектуры, позволяющей расширять функциональность без существенных изменений в существующих модулях. Такой подход обеспечивает долгосрочную жизнеспособность решения и возможность адаптации к изменяющимся требованиям учреждения.

Архитектура системы реализована по клиент-серверному принципу, что позволяет разделить функциональность на независимые уровни. Серверная часть отвечает за обработку запросов, выполнение бизнес-логики и взаимодействие с базой данных. Клиентская часть обеспечивает отображение информации и взаимодействие пользователя с системой. Между компонентами используется REST-интерфейс, обеспечивающий стандартизированный обмен данными. Такая архитектура позволяет масштабировать серверную и клиентскую части независимо друг от друга, а также упрощает интеграцию с внешними сервисами, включая государственные медицинские системы и лабораторные комплексы.

1.2.1 Проектирование базы данных

Проектирование базы данных направлено на обеспечение целостности, непротиворечивости и структурированности медицинской информации. В процессе разработки была сформирована ER-модель, включающая сущности пациентов, врачей, приёмов, медицинских записей, документов и расписаний. Между сущностями определены связи, отражающие реальные процессы учреждения здравоохранения. Использование реляционной модели позволяет обеспечить строгий контроль данных, а также реализовать механизмы валидации и ограничения целостности. Особое внимание уделено оптимизации запросов, поскольку система должна обеспечивать быстрый доступ к данным даже при значительном объёме информации.

Точка входа программы расположена в основном модуле серверной части, который запускается первым и отвечает за инициализацию всей системы. На этом этапе создаётся и настраивается веб-сервер FastAPI, подключаются маршруты REST-API, формируется подключение к базе данных через SQLAlchemy и при необходимости выполняется автоматическая генерация таблиц. После инициализации сервер подготавливает сервисный слой и репозитории, обеспечивая им доступ к базе данных и механизмам выполнения SQL-операций. Таким образом, взаимодействие всех компонентов приложения начинается именно с этого модуля, а база данных становится центральным хранилищем, к которому обращаются все части системы — от REST-запросов до внутренних сервисов, выполняющих бизнес-логику.

Пример структуры таблицы «appointments»:

```
appointments (  
    id SERIAL PRIMARY KEY,  
    patient_id INTEGER REFERENCES patients(id),  
    doctor_id INTEGER REFERENCES doctors(id),  
    datetime TIMESTAMP NOT NULL,  
    status VARCHAR(20) NOT NULL,  
    created_at TIMESTAMP DEFAULT NOW()  
)
```

Для демонстрации взаимодействия серверной части с базой данных ниже приведены примеры реальных SQL-запросов, которые формируются в процессе работы системы. После запуска основного модуля приложение инициализирует веб-сервер, подключает маршруты REST-API и устанавливает соединение с PostgreSQL. Каждый REST-запрос, поступающий от клиента, проходит через сервисный слой и приводит к выполнению конкретной SQL-операции.

Примеры SQL-запросов и взаимодействие с базой данных.

Получение пользователя по email (авторизация):

```
SELECT *  
FROM users  
WHERE email = :email  
LIMIT 1;
```

Создание нового приёма:

```
INSERT INTO appointments (patient_id, doctor_id, datetime, status)  
VALUES (:patient_id, :doctor_id, :datetime, :status);
```

Обновление статуса приёма:

```
UPDATE appointments  
SET status = :status  
WHERE id = :id;
```

Получение медицинских записей пациента:

```
SELECT *  
FROM medical_records  
WHERE patient_id = :patient_id  
ORDER BY created_at DESC;
```


1.2.2 Проектирование пользовательского интерфейса

Пользовательский интерфейс разработан с учётом принципов удобства, доступности и минимизации количества действий, необходимых для выполнения основных операций. Интерфейс разделён на функциональные модули, соответствующие ролям пользователей: пациент, врач и администратор. Для пациента предусмотрены инструменты записи на приём, просмотра истории посещений и работы с документами. Врач получает доступ к расписанию, медицинским картам и инструментам ведения приёмов. Администратор управляет пользователями, расписаниями и отчётностью. Интерфейс реализован в виде набора компонентов, что обеспечивает единообразие отображения и упрощает дальнейшее расширение функциональности.

Внешний вид пользовательского интерфейса представлен на рисунках 5,6,7. На вкладке «Профиль пациента» пользователь может изменять информацию о себе, а также сменить пароль, при необходимости.

MedClinicIS

Пациент — анЗ Выйти

Профиль

Данные успешно обновлены

ФИО: Ситицын Роман Дмитриевич

Дата рождения: 19.06.2003

Телефон: +79581101547

Email: Roma123@mail.ru

Адрес: г. Пенза, Суворова 15, кв. 104

Полис: 78954359756034

Паспорт: 5624 455345

Редактировать

Сменить пароль

Рисунок 5 – Внешний вид «Профиля пациента»

Изм.	Лист	№ докум.	Подпись	Дата

На вкладке «Мои документы» пациент может загрузить необходимые документы, а также посмотреть, какие документы загрузил для него его лечащий врач.

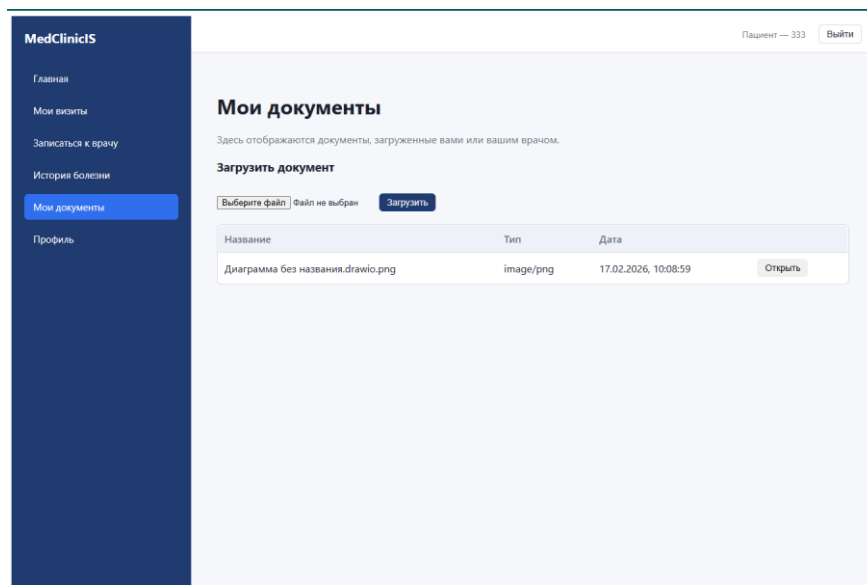


Рисунок 6 – Внешний вид «Мои документы»

В «История болезни» пациент может ознакомиться со своими записями

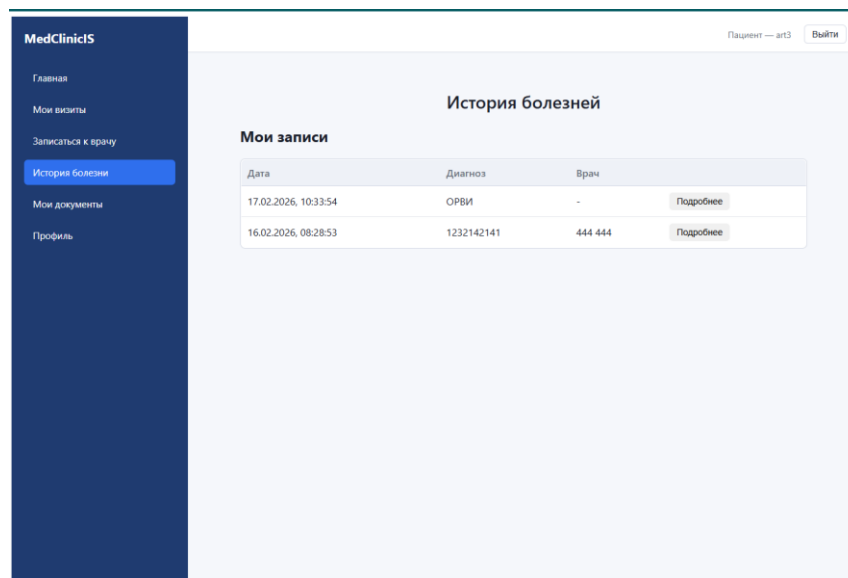


Рисунок 7 – Внешний вид «История болезни»

На рисунке 8 изображена вкладка с возможностью записаться к выбранному врачу на его свободное время.

The screenshot shows the 'MedClinicS' interface. On the left is a dark blue sidebar with links: Главная, Мои визиты, **Записаться к врачу** (highlighted), История болезни, Мои документы, and Профиль. The top right header shows 'Пациент — арт3' and a 'Выйти' button. The main content area is titled 'Записаться к врачу' and contains three sections: 'Врач:' with a dropdown menu showing 'Хмельков Роман', 'Время:' with a date and time range '26.01.2026, 14:00:00 — 26.01.2026, 14:30:00', and 'Жалоба:' with a text input field containing 'Боли в животе'. At the bottom of the form is a blue button labeled 'Записаться'.

Рисунок 8 – Внешний вид «Записаться к врачу»

Рисунок 9 представляет собой внешний вид страницы «Мои визиты», на которой пациент может увидеть все свои будущие, прошедшие, отмененные визиты, отсортировать их по статусу и специальности врача. На рисунке 10 изображена панель «Детали визита», открывающаяся по нажатию кнопки «Подробнее».

The screenshot shows the 'MedClinicS' interface for the 'Мои визиты' (My visits) page. The sidebar is identical to the previous screenshot. The main content area is titled 'Мои визиты' and features two filter dropdowns: 'Статус:' (set to 'Завершён') and 'Специальность:' (set to 'Все'). Below the filters is a card displaying visit details for 'Врач: Хмельков Роман Викторович', 'Специальность: Терапевт', 'Начало: 26.01.2026, 10:00:00', 'Конец: 26.01.2026, 10:30:00', and 'Статус: Завершён'. At the bottom of the card is a button labeled 'Подробнее'.

Рисунок 9 – Внешний вид «Записаться к врачу»

Детали визита

Врач:

Хмельков Роман Викторович

Специальность:

Терапевт

Кабинет:

23

Начало:

26.01.2026, 10:00:00

Конец:

26.01.2026, 10:30:00

Длительность:

30 минут

Статус:

Завершён

Создано:

16.02.2026, 08:23:54

Обновлено:

17.02.2026, 10:33:54

Заккрыть

Рисунок 10 – Внешний вид «Детали визита»

На главной странице выведена информация о ближайшем визите, полисе и номере телефона пациента. Главная страница изображена на рисунке 11.

MedClinicS

Главная

Мои визиты

Записаться к врачу

История болезни

Мои документы

Профиль

Пациент — art3

Выйти

Добро пожаловать, Роман

Ближайший визит

Дата: 17.02.2026, 15:00:00

Врач: Бритвин Андрей

Статус: Отменён

Будущих визитов

7

Полис

78954359756034

Телефон

+79581101547

Рисунок 11 – Внешний вид «Главная страница»

На главной странице пользователя «врач» выведены количество мест для записи, ближайшие приемы. Для врача главная страница изображена на рисунке 12.

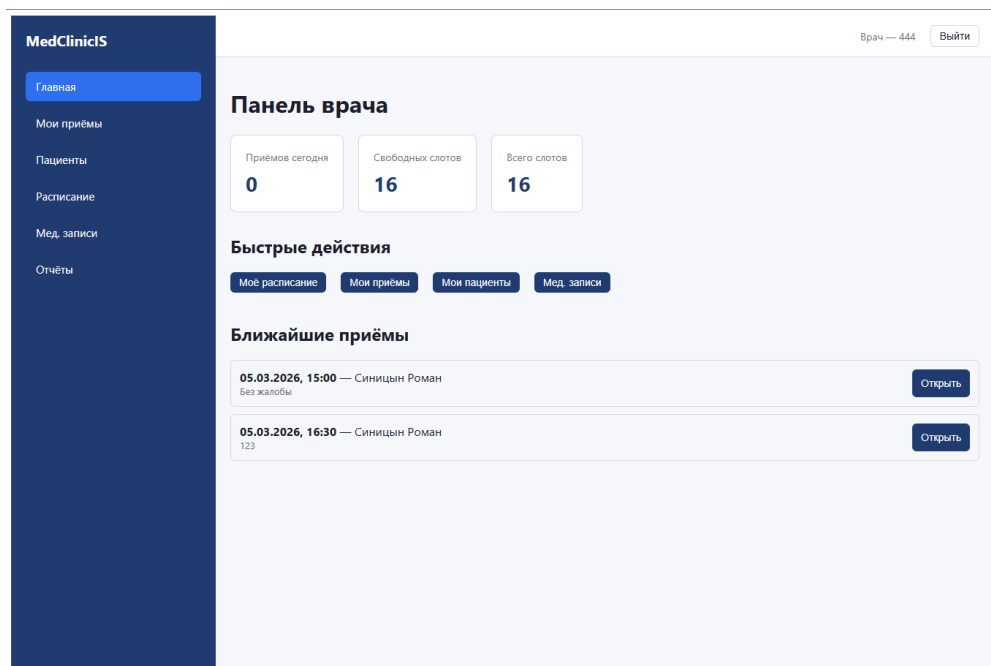


Рисунок 12 – Внешний вид «Главная страница для врача»

В моих приемах врач может посмотреть на все свои приемы, открыть мед. карточку пациента и создать медзапись. Представлено это на рисунках 13,14,15.

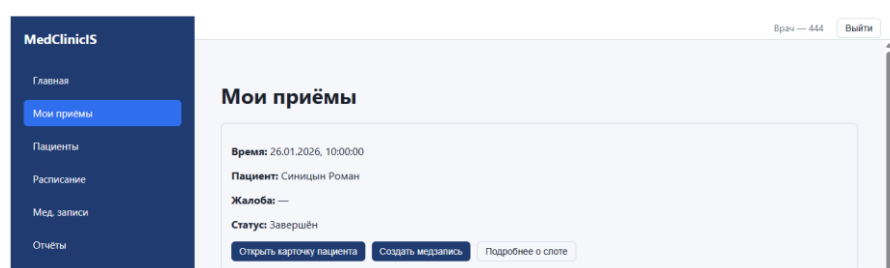


Рисунок 13 – Внешний вид «Мои приемы для врача»

Рисунок 14 – Внешний вид «Карточка пациента»

Рисунок 15 – Внешний вид «Создание медзаписи»

На рисунке 16 изображена вкладка «Расписание врачей», на которой врач удобно может задать свое расписание.

Рисунок 16 – Внешний вид «Расписание»

На странице «Пациенты» врач может посмотреть всех пациентов в системе, добавить новых и отредактировать. Страница изображена на рисунке 17.

MedClinicS

Главная
Мои приемы
Пациенты
Расписание
Мед. записи
Отчеты

Врач — 444 Выйти

Пациенты

Поиск по ФИО, телефону или дате рождения...

ФИО	Телефон	Дата рождения		
123 123 123	+7 (963) 110-12-47	2026-01-29	Редактировать	Удалить
Барсуков Геннадий Анатольевич	+7 (961) 101-42-37	2017-02-16	Редактировать	Удалить
Куликов Игорь Геннадьевич	+7 (963) 110-12-47	2026-01-29	Редактировать	Удалить
Синицын Роман Дмитриевич	+79581101547	2003-06-19	Редактировать	Удалить

История приёмов: Куликов Игорь

Дата	Диагноз	Врач
20.02.2026, 09:34:00	—	Бритвин Андрей
11.02.2026, 10:29:00	—	Бритвин Андрей

Редактирование пациента

Фамилия
Куликов

Имя
Игорь

Отчество
Геннадьевич

Дата рождения
29.01.2026

Пол
M

Телефон
+7 (963) 110-12-47

Email
igor123@mail.ru

Адрес
Ostrovskogo 10

Паспорт

Рисунок 17 – Внешний вид «Пациенты»

Страница «История болезней» нужна, чтобы посмотреть мед.записи любого пациента в системе. Она представлена на рисунке 18.

MedClinicS

Главная
Мои приемы
Пациенты
Мед. записи
Отчеты

Врач — 444 Выйти

История болезней

ФИО пациента
Например: Иванов Иван

Дата рождения
ДД.ММ.ГГГГ

Найти пациента

Добавить запись

Пациент не выбран — запись будет создана без привязки

Рисунок 18 – Внешний вид «Мед.записи»

В отчетах администратор и врач может посмотреть сводку учреждения за выбранный день. Посмотреть внешний вид можно на рисунке 19.

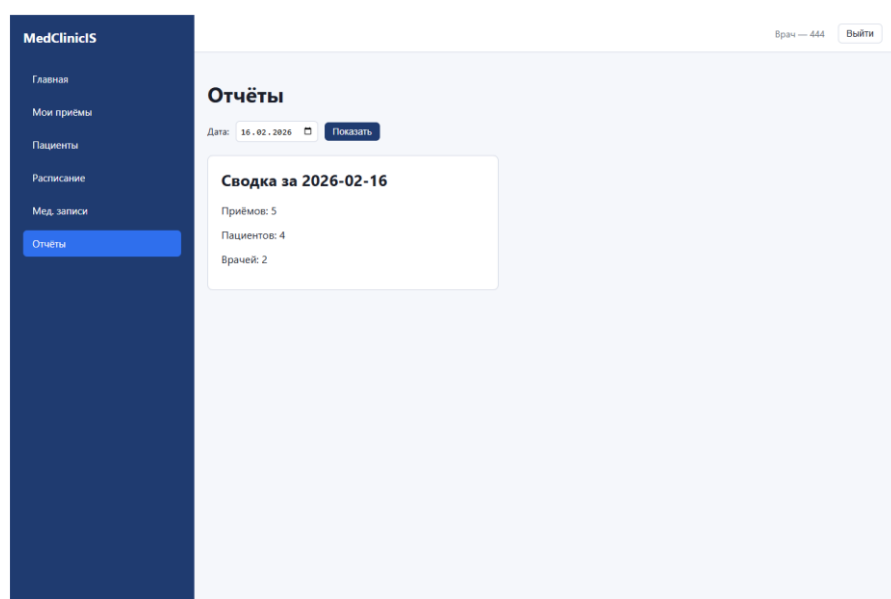


Рисунок 19 – Внешний вид «Отчеты»

Пользователь администратор может добавлять новых врачей и редактировать данные существующих. Также администратор может добавлять специальности для врачей, создавать расписание выбранному врачу. На рисунке 20 изображена страница «Врачи».

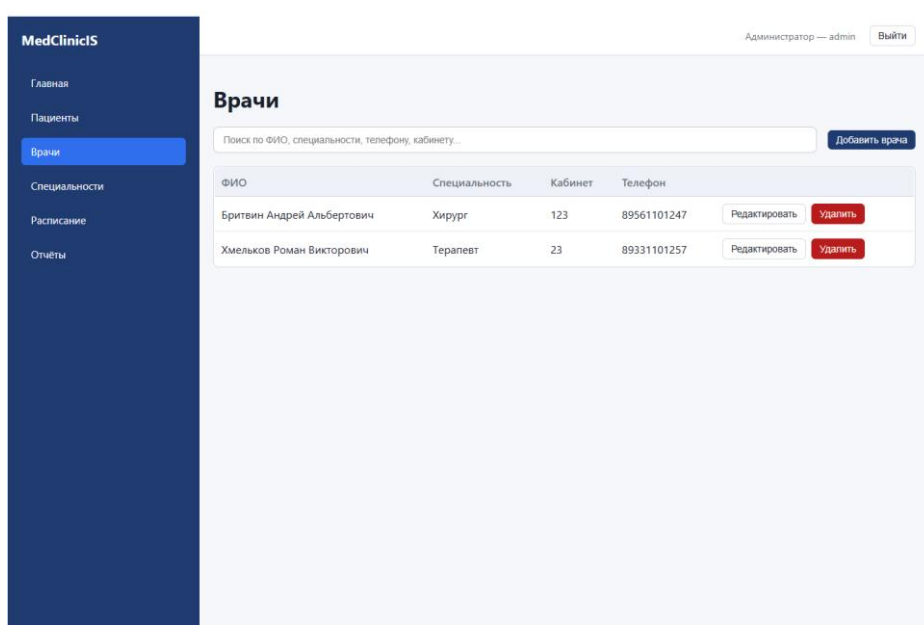


Рисунок 20 – Внешний вид «Врачи»

1.2.3 Разработка серверной части

Разработка серверной части выполнена на основе фреймворка FastAPI, что позволило реализовать высокопроизводительный и структурированный API. В процессе разработки были созданы модули для обработки запросов, управления пользователями, работы с расписаниями, приёмами и медицинскими записями. Для взаимодействия с базой данных используется ORM-слой, обеспечивающий удобную работу с сущностями и автоматическую генерацию SQL-запросов. В серверную часть встроены механизмы аутентификации и авторизации, основанные на токенах, что обеспечивает надёжную защиту данных и контроль доступа.

На рисунке 21 представлена диаграмма классов для серверного взаимодействия.

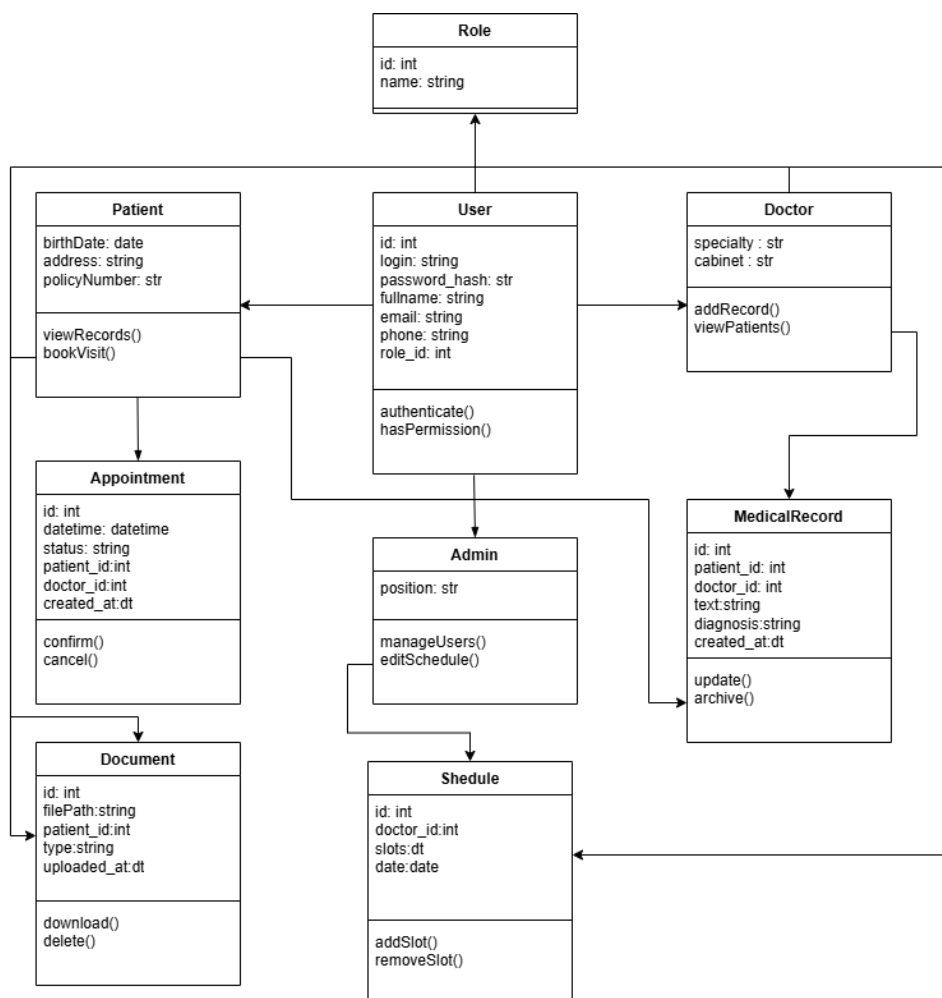


Рисунок 21 – UML-диаграмма классов

1.2.4 Разработка клиентской части

Клиентская часть реализована с использованием React, что позволило создать динамичный и отзывчивый интерфейс. Компонентная структура обеспечивает модульность и повторное использование элементов. В процессе разработки были созданы страницы для всех ролей пользователей, а также реализованы механизмы взаимодействия с сервером через REST-API. Особое внимание уделено обработке ошибок, отображению уведомлений и обеспечению корректной работы интерфейса на различных устройствах. Реализована система маршрутизации, позволяющая пользователям быстро переходить между разделами.

Пример запроса с клиента:

```
axios.post("/api/appointments/create", {
  doctor_id: selectedDoctor,
  datetime: selectedTime
}).then(response => {
  setMessage("Запись успешно создана");
});
```

Пример компонента:

```
function AppointmentCard({ appointment }) {
  return (
    <div className="card">
      <h3>{appointment.doctorName}</h3>
      <p>{appointment.datetime}</p>
      <p>Статус: {appointment.status}</p>
    </div>
  );
}
```

Изм.	Лист	№ докум.	Подпись	Дата

1.2.5 Интеграция и взаимодействие компонентов

Интеграция клиентской и серверной частей осуществляется через стандартизированные HTTP-запросы. Все данные передаются в формате JSON, что обеспечивает совместимость и удобство обработки. На этапе интеграции были проведены тесты корректности взаимодействия, проверена работа всех основных сценариев и обеспечена согласованность данных между компонентами. Важным аспектом является обработка ошибок и исключительных ситуаций, что позволяет обеспечить стабильность работы системы даже при некорректных действиях пользователей или временных сбоях.

Пример JSON-ответа:

```
{
  "appointment_id": 42,
  "status": "success"
}
```

На рисунке 22 изображена диаграмма взаимодействия.

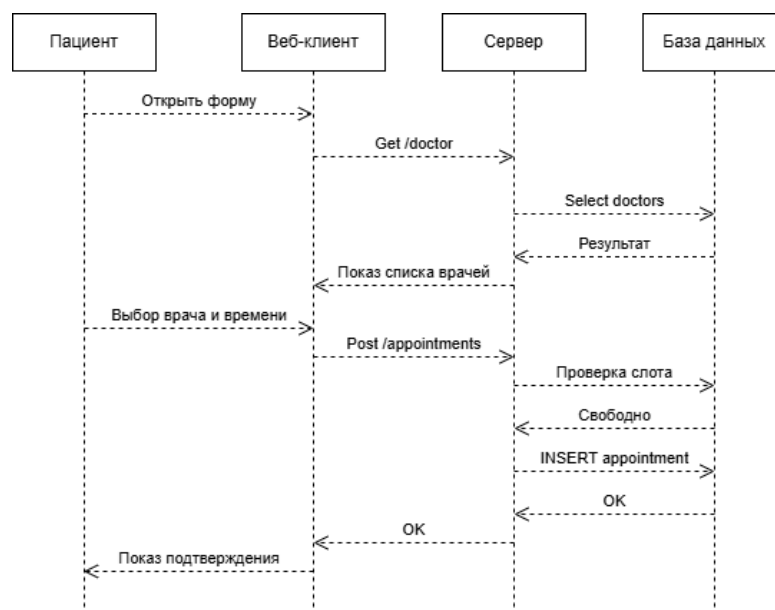


Рисунок 22 – UML-диаграмма взаимодействия

1.2.6 Обеспечение безопасности

Безопасность является критически важным аспектом разработки системы. В процессе проектирования и реализации были внедрены механизмы защиты данных, включая шифрование токенов, контроль доступа, валидацию входящих данных и защиту от распространённых атак. Доступ к медицинской информации строго разграничен в соответствии с ролью пользователя. Все операции, связанные с изменением данных, фиксируются, что обеспечивает возможность аудита и контроля действий пользователей. Передача данных осуществляется по защищённым каналам, что исключает возможность перехвата информации.

В системе используется JWT-аутентификация, шифрование паролей и разграничение прав доступа.

Пример генерации токена:

```
def create_access_token(data: dict):  
    return jwt.encode(data, SECRET_KEY, algorithm="HS256")
```

Пример middleware-проверки:

```
if user.role != "doctor":  
    raise HTTPException(status_code=403, detail="Недостаточно  
прав")
```

1.2.7 Динамика работы приложения

Динамика работы приложения отражает последовательность действий, выполняемых системой при взаимодействии пользователей с различными функциональными модулями. Работа приложения основана на клиент-серверной архитектуре: клиентская часть инициирует запросы, серверная часть обрабатывает их, взаимодействует с базой данных и возвращает результаты в виде структурированных ответов. Такой подход обеспечивает чёткое разделение логики, высокую производительность и возможность масштабирования.

При обращении пользователя к системе клиентская часть формирует HTTP-запрос, содержащий необходимые параметры. Серверная часть принимает запрос, выполняет валидацию данных, проверяет права доступа и инициирует выполнение соответствующего сценария. После обработки данных сервер формирует ответ в формате JSON, который отображается в пользовательском интерфейсе.

Процесс авторизации начинается с ввода логина и пароля. Клиентская часть отправляет запрос на сервер, где выполняется проверка корректности данных. В случае успешной авторизации сервер генерирует JWT-токен, содержащий информацию о пользователе и его роли. Токен передаётся клиенту и используется для всех последующих запросов.

При записи на приём пациент выбирает врача и доступный временной интервал. Клиентская часть отправляет запрос на сервер, где выполняется проверка доступности выбранного времени. Если интервал свободен, создаётся запись в базе данных, после чего клиент получает подтверждение.

Врач открывает карточку пациента, просматривает историю посещений и добавляет новую медицинскую запись. Клиентская часть отправляет данные на сервер, где они проходят валидацию и сохраняются в базе данных. После успешного сохранения обновлённая информация отображается в интерфейсе врача.

При загрузке документа клиентская часть формирует multipart-запрос, содержащий файл и метаданные. Сервер принимает файл, сохраняет его в файловом хранилище или базе данных, создаёт запись о документе и возвращает результат.

Администратор управляет пользователями, расписаниями и справочниками. Все операции выполняются через защищённые запросы, доступные только при наличии соответствующих прав. Сервер проверяет роль администратора и выполняет необходимые изменения в базе данных.

На рисунке 23 изображена диаграмма состояний для записи на прием.



Рисунок 23 – UML-диаграмма состояний

1.2.8 Тестирование приложения

Тестирование разработанной информационной системы направлено на проверку корректности работы всех функциональных модулей, выявление ошибок, оценку стабильности и соответствия системы требованиям, сформулированным на этапе проектирования. Основной целью является подтверждение того, что система обеспечивает корректную обработку медицинских данных, надёжно функционирует под нагрузкой и предоставляет пользователям удобный и предсказуемый интерфейс. Тестирование также позволяет убедиться в корректности взаимодействия клиентской и серверной частей, а также в соблюдении требований безопасности.

Для проверки работоспособности системы использовалась комбинированная методика, включающая модульное тестирование, интеграционное тестирование, функциональное тестирование и пользовательское тестирование. Модульное тестирование применялось для проверки отдельных функций серверной части, включая обработку запросов, валидацию данных и работу с базой данных. Интеграционное тестирование проводилось для проверки взаимодействия между модулями, включая работу REST-API и корректность передачи данных между клиентом и сервером.

Функциональное тестирование охватывало проверку пользовательских сценариев, таких как запись на приём, просмотр расписания, ведение медицинских записей и загрузка документов. Пользовательское тестирование проводилось с участием тестовых пользователей, имитирующих роли пациента, врача и администратора.

2 Безопасность жизнедеятельности

2.1 Характеристика рабочего места разработчика

Рабочее место программиста представляет собой персональный компьютер, оснащённый монитором, клавиатурой, манипулятором «мышь» и офисной мебелью. Продолжительная работа за компьютером может приводить к зрительному утомлению, статическим нагрузкам и снижению работоспособности. Поэтому организация рабочего места должна соответствовать требованиям СанПиН 1.2.3685-21 и ГОСТ Р 50923-96.

Рабочий стол должен обеспечивать размещение оборудования без излишнего напряжения мышц рук и спины. Высота стола и стула регулируется, что позволяет поддерживать правильную осанку. Монитор располагается на расстоянии 50–70 см от глаз, верхняя граница экрана — на уровне глаз или чуть ниже. Освещённость рабочего места должна составлять 300–500 лк, при этом следует избегать бликов на экране.

Микроклимат помещения оказывает существенное влияние на работоспособность сотрудников. Оптимальные параметры для офисных помещений включают температуру воздуха 20–24 °С, относительную влажность 40–60 % и скорость движения воздуха не более 0,1–0,2 м/с. Несоблюдение этих параметров может привести к снижению концентрации внимания, утомляемости и ухудшению самочувствия.

Освещение должно быть равномерным и сочетать естественный и искусственный свет. Использование светодиодных ламп с цветовой температурой 4000–5000 К обеспечивает комфортное восприятие информации с экрана. Рабочее место должно быть расположено таким образом, чтобы естественный свет падал сбоку, а не прямо на монитор.

2.2 Пожарная безопасность

Работа с компьютерным оборудованием требует соблюдения правил электробезопасности. Все устройства должны быть подключены через исправные розетки с заземлением. Использование сетевых фильтров снижает риск повреждения оборудования при скачках напряжения. Корпуса компьютеров и периферийных устройств должны быть исправны и не иметь повреждений изоляции.

Персонал должен быть проинструктирован о правилах поведения при возникновении аварийных ситуаций, таких как короткое замыкание или запах гари. В помещении должны находиться средства пожаротушения, соответствующие классу пожаров Е (электрооборудование под напряжением).

Помещение, в котором осуществляется разработка и эксплуатация системы, должно соответствовать требованиям пожарной безопасности. Необходимо обеспечить наличие исправных огнетушителей, свободных эвакуационных выходов и схем эвакуации. Электропроводка должна быть исправной, а использование удлинителей — минимизировано.

Запрещается хранить легковоспламеняющиеся материалы рядом с компьютерным оборудованием. Сотрудники должны проходить регулярный инструктаж по пожарной безопасности и уметь пользоваться первичными средствами тушения пожара.

2.3 Психофизиологические нагрузки и профилактика утомления

Работа программиста связана с высокой концентрацией внимания, длительным пребыванием в статичной позе и значительной нагрузкой на зрение. Для снижения утомляемости рекомендуется соблюдать режим труда и отдыха: каждые 60 минут делать перерыв 5–10 минут, выполнять упражнения для глаз и разминку для мышц шеи и спины. Использование эргономичной мебели, правильная организация рабочего пространства и соблюдение режима освещения позволяют снизить риск профессиональных заболеваний, таких как остеохондроз, синдром сухого глаза и туннельный синдром.

Характеристика психофизиологических нагрузок. Деятельность разработчика программного обеспечения относится к категории труда с высокими нервно-психическими нагрузками и значительным статическим напряжением. В процессе работы программист длительное время сохраняет вынужденную рабочую позу — сидя за компьютером с наклонённой вперёд головой и согнутыми в локтях и коленях конечностями. Такая поза приводит к хроническому напряжению мышц шеи, плечевого пояса, спины и поясницы, что со временем может вызвать развитие остеохондроза, сколиоза, протрузий и грыж межпозвонковых дисков. Особенно уязвимым является шейный отдел позвоночника, поскольку даже небольшой наклон головы вперёд многократно увеличивает нагрузку на шейные позвонки и межпозвонковые диски.

3 Экономика

Калькуляция себестоимости разработки выполнялась на основании фактических временных затрат по каждому этапу создания минимально жизнеспособного продукта (MVP). Проект реализовывался одним разработчиком, владеющим технологиями React, FastAPI и PostgreSQL. Исходные параметры для вычислений: должность специалиста — инженер-программист (full-stack), стоимость одного часа работы — 1 300 рублей (рассчитана исходя из среднерыночных показателей оплаты для специалиста данной квалификации в регионе с включением страховых отчислений и накладных издержек), длительность рабочей смены — 8 часов, общая продолжительность разработки — 4 календарных месяца.

Трудоёмкость выполнения работ по этапам приведена в таблице 2.

Таблица 2 — Трудоёмкость и стоимость разработки

Наименование этапа	Трудозатраты, чел.-час	Стоимость, руб.
Изучение предметной области, сбор и формализация требований	45	58 500
Проектирование архитектуры, логической модели БД и API	50	65 000

Наименование этапа	Трудозатраты, чел.-час	Стоимость, руб.
Разработка бэкенда (FastAPI, SQLAlchemy, PostgreSQL)	170	221 000
Разработка фронтенда (React, интерфейсы для разных ролей)	165	214 500
Объединение компонентов, развёртывание и отладка	45	58 500
Проведение тестирования и подготовка документации	45	58 500
Итого	520	676 000

В соответствии с главой 23 Налогового кодекса РФ (НДФЛ) и законодательством о страховых взносах, работодатель обязан удерживать из начисленной суммы налог на доходы физических лиц, а также начислить взносы во внебюджетные фонды поверх фонда оплаты труда.

Расчёт налоговой нагрузки (на 676 000 руб.).

Налог на доходы физических лиц (НДФЛ 13%)

$676\,000 \times 0,13 = 87\,880$ руб. Сумма к выдаче на руки ("нетто"):

$676\,000 - 87\,880 = 588\,120$ руб.

Страховые взносы (30% от gross) начисляются работодателем сверх суммы оплаты труда.

Пенсионный фонд (22%): $676\,000 \times 0,22 = 148\,720$ руб.

Фонд обязательного мед. страхования (5,1%): $676\,000 \times 0,051 = 34\,476$ руб.

Фонд социального страхования (2,9%): $676\,000 \times 0,029 = 19\,604$ руб.

Итого страховых взносов: $148\,720 + 34\,476 + 19\,604 = 202\,800$ руб.

Таким образом, реальная стоимость создания продукта для юридического лица складывается из суммы gross-зарплаты и начисленных страховых взносов.

Полная себестоимость = $676\,000$ (gross) + $202\,800$ (взносы) = $878\,800$ руб.

Данная величина отражает объём инвестиций, необходимых для коммерциализации или внедрения системы при условии, что разработка ведётся штатным специалистом.

Анализ предметной области и формирование требований заняли 45 часов. На этом этапе проводилось изучение существующих медицинских информационных систем, выявление их недостатков, определение ключевых функциональных требований к разрабатываемому приложению, а также согласование с потенциальными пользователями (врачами, администраторами, пациентами) основных сценариев работы. Тщательная проработка требований на начальном этапе позволила избежать существенных доработок в дальнейшем.

Проектирование архитектуры системы, структуры базы данных и API потребовало 50 часов. В рамках этого этапа была разработана ER-модель базы данных, определены связи между сущностями (пользователи, пациенты, врачи, приёмы, медицинские записи, документы), спроектирована клиент-серверная архитектура с REST API, а также разработаны схемы данных для валидации запросов и ответов. Особое внимание уделено обеспечению целостности и безопасности медицинских данных.

Разработка серверной части заняла 170 часов, что составляет примерно 33 % от общего объёма трудозатрат. На этом этапе реализованы: модуль аутентификации и авторизации с использованием JWT-токенов, обеспечивающий разграничение прав доступа для трёх ролей (пациент, врач, администратор); CRUD-операции для управления пользователями, расписаниями и приёмами; логика проверки доступности временных слотов при записи на приём; механизмы создания, переноса и отмены записей; модуль работы с медицинскими записями и историей болезни; подсистема загрузки и скачивания медицинских документов; формирование отчётов по деятельности учреждения. Интеграция с базой данных выполнена через SQLAlchemy, что обеспечило удобную работу с моделями и миграциями.

Разработка клиентской части заняла 165 часов (также около 32 % от общих трудозатрат). На этом этапе созданы: страницы авторизации и регистрации; три типа личных кабинетов с индивидуальным набором функций в зависимости от роли пользователя; интерфейс для просмотра и записи на приём с отображением свободных слотов; страница истории посещений и медицинских записей; модуль работы с документами (просмотр, загрузка, скачивание); административная панель для управления пользователями, расписаниями и отчётами. Все интерфейсы выполнены с использованием компонентного подхода React, что обеспечило их модульность, удобство поддержки и возможность повторного использования кода. Реализована адаптивная вёрстка, позволяющая комфортно работать с системой как с настольных компьютеров, так и с мобильных устройств.

					ПГУ 1.090501.16.001 ПЗ	Лист
						70
Изм.	Лист	№ докум.	Подпись	Дата		

Интеграция компонентов, развёртывание и отладка потребовали 45 часов. На этом этапе осуществлялась настройка взаимодействия клиентской и серверной частей через REST API, отладка обмена данными в формате JSON, настройка CORS-политики, конфигурирование серверной среды (Uvicorn) и подготовка скриптов для развёртывания. Также проводилась оптимизация запросов к базе данных с целью минимизации времени ответа сервера.

Тестирование и оформление документации заняли 45 часов. Выполнялось модульное тестирование ключевых серверных функций, интеграционное тестирование сценариев взаимодействия клиента и сервера, функциональное тестирование пользовательских сценариев (запись на приём, отмена записи, создание медицинской записи, загрузка документов), а также пользовательское тестирование с привлечением тестовых пользователей для оценки удобства интерфейса. Подготовлена пояснительная записка, включающая описание архитектуры, инструкцию по развёртыванию и руководство пользователя.

Этапы непосредственного программирования (разработка серверной и клиентской частей) суммарно составили 335 часов, или около 64 % от общего объёма работ. Оставшиеся 36 % пришлись на анализ, проектирование, интеграцию, тестирование и оформление документации, что является типичным распределением для проектов по разработке программного обеспечения.

Таким образом, расчётная стоимость создания MVP веб-приложения «Информационная система учреждения здравоохранения» составила 676 000 рублей.

Экономический и организационный эффект от внедрения разработанной системы складывается из следующих составляющих. Во-первых, автоматизация процессов записи пациентов и управления расписанием позволяет сократить временные затраты сотрудников регистратуры и администраторов примерно на 50–60 %, что высвобождает ресурсы для выполнения других задач. Во-вторых, исключаются ошибки, связанные с человеческим фактором, такие как двойная запись разных пациентов на одно время, конфликты в расписании врачей или потеря медицинских документов.

В-третьих, пациенты получают возможность самостоятельно записываться на приём через удобный веб-интерфейс без необходимости звонить в регистратуру или посещать учреждение лично, что повышает удовлетворённость качеством обслуживания. В-четвёртых, врачи и администраторы получают оперативный доступ к актуальной информации о расписании, пациентах и медицинских записях из любой точки, где есть доступ в интернет.

С учётом перечисленных факторов, инвестиции в разработку системы в размере 878 000 рублей могут окупиться за счёт экономии рабочего времени сотрудников, снижения административных издержек и повышения качества предоставляемых медицинских услуг в течение 6–12 месяцев после внедрения в зависимости от интенсивности использования системы и масштаба медицинского учреждения.

					ПГУ 1.090501.16.001 ПЗ	Лист
						72
Изм.	Лист	№ докум.	Подпись	Дата		

Заключение

В ходе выполнения выпускной квалификационной работы была разработана информационная система для автоматизации ключевых процессов учреждения здравоохранения. Проведённый анализ предметной области позволил выявить основные проблемы, связанные с использованием традиционных методов ведения медицинской документации, отсутствием централизованного хранения данных и недостаточной эффективностью взаимодействия между сотрудниками. Анализ существующих аналогов показал, что большинство доступных решений либо избыточны по функциональности и стоимости, либо недостаточно гибки и не позволяют адаптировать систему под конкретные потребности учреждения. Это подтвердило актуальность разработки собственной информационной системы.

В процессе работы были сформулированы требования к системе, определены цели и задачи разработки, а также выбран оптимальный технологический стек, включающий FastAPI, PostgreSQL и React. Такой выбор обеспечил высокую производительность, надёжность, удобство разработки и возможность дальнейшего масштабирования. На этапе проектирования была разработана архитектура системы, включающая клиентскую и серверную части, а также структурирована база данных, отражающая реальные процессы медицинского учреждения. Созданы интерфейсы для различных категорий пользователей: пациентов, врачей и администраторов.

В ходе разработки реализованы основные функциональные модули: авторизация и аутентификация, управление расписаниями, запись на приём, ведение медицинских записей, работа с документами и административные функции. Проведён анализ динамики работы приложения, описывающий последовательность взаимодействия компонентов системы при выполнении основных пользовательских сценариев.

Список использованных источников

1. Буч Г., Рамбо Дж., Якобсон А. UML. Унифицированный язык моделирования. Руководство пользователя. — М.: Вильямс, 2019. — 496 с.
2. Соммервилл И. Инженерия программного обеспечения. — 10-е изд. — М.: Вильямс, 2021. — 816 с.
3. Макконнелл С. Совершенный код. Мастер-класс. — СПб.: Питер, 2020. — 896 с.
4. Фаулер М. Архитектура корпоративных приложений. — М.: Вильямс, 2020. — 560 с.
5. Костылев В. А. Проектирование пользовательских интерфейсов. — М.: Юрайт, 2021. — 290 с.
6. React Official Documentation. [Электронный ресурс] — URL: <https://react.dev> (дата обращения: 15.02.2026).
7. PostgreSQL Global Development Group. PostgreSQL Documentation. [Электронный ресурс] — URL: <https://www.postgresql.org/docs> (дата обращения: 05.02.2026).
8. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы. Построение и анализ. — М.: Вильямс, 2020. — 1328 с.
9. Глушаков С. В. Разработка веб-приложений на Python. — М.: БХВ-Петербург, 2022. — 368 с.
10. Литвинов В. В. Информационные технологии в медицине. — М.: ГЭОТАР-Медиа, 2020. — 352 с.
11. DeepSeek-AI. DeepSeek-V3: [Электронный ресурс] — URL: <https://www.deepseek.com> (дата обращения: 10.05.2026). (В качестве промтов использовались запросы на поиск источников по заданной тематике и суммаризацию информации из найденных источников)

Приложение А

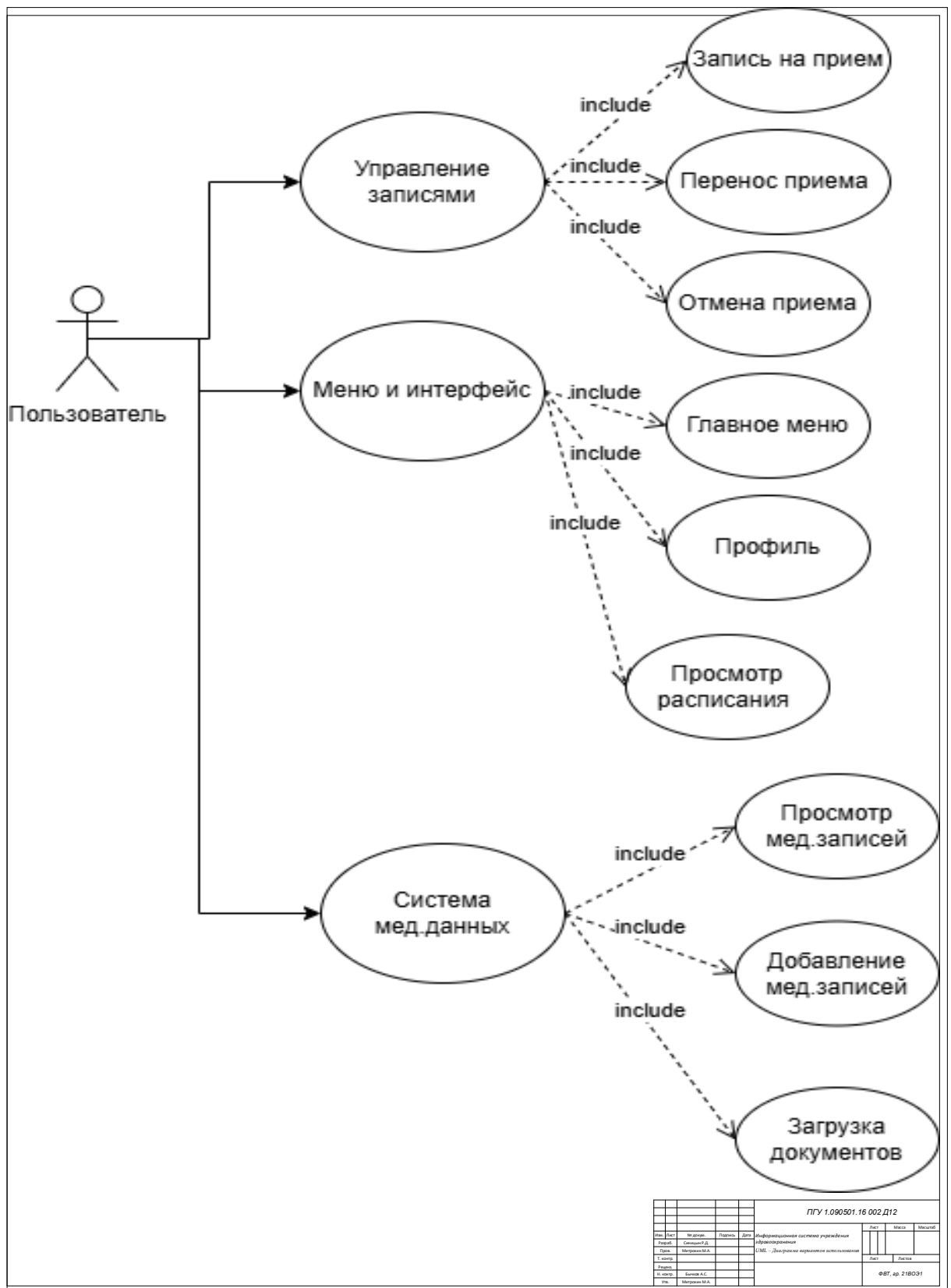
(обязательное)

Информационная система учреждения здравоохранения

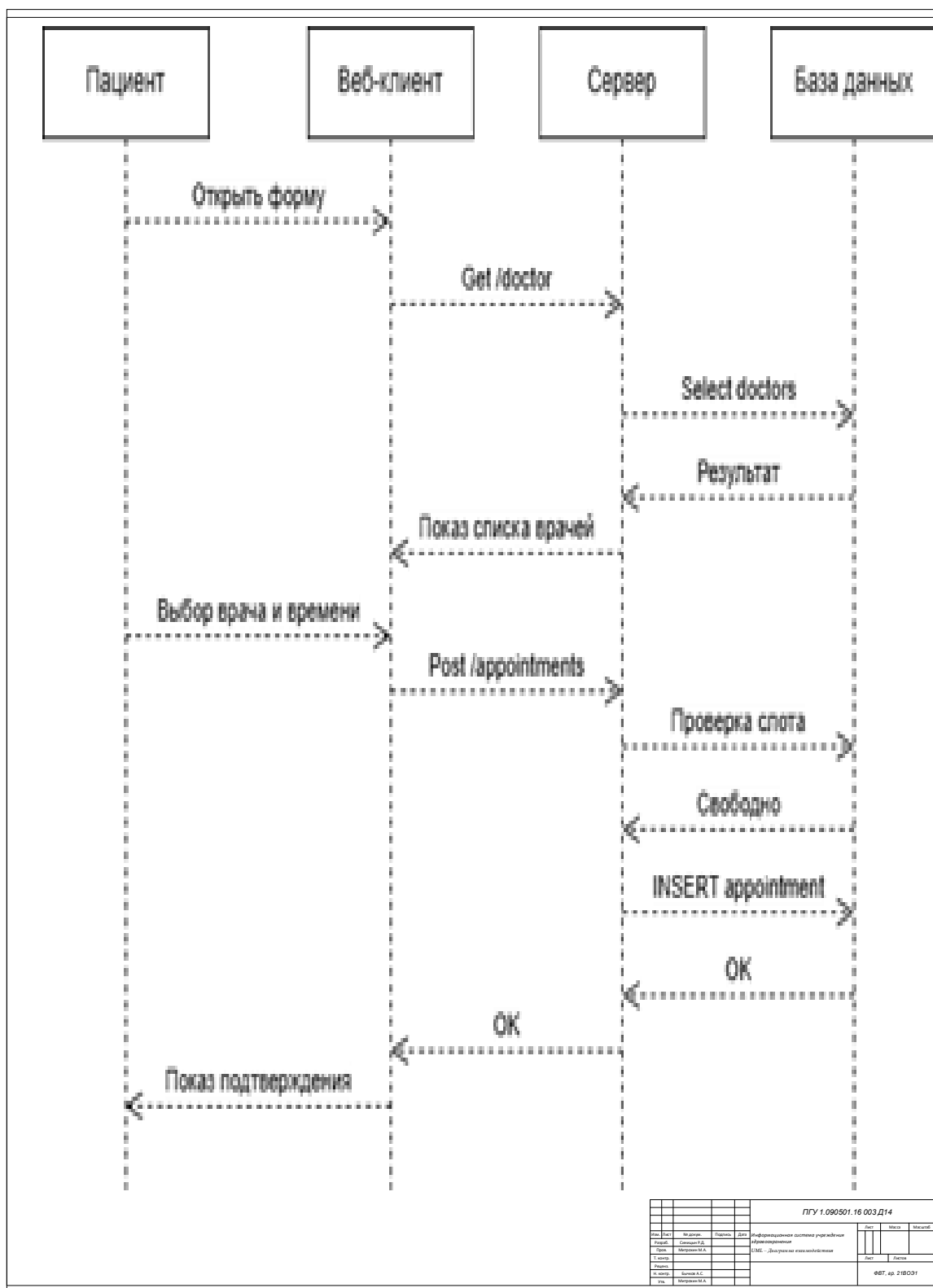
UML-диаграммы

					ПГУ 1.090501.16.001 ПЗ	Лист
						75
Изм.	Лист	№ докум.	Подпись	Дата		

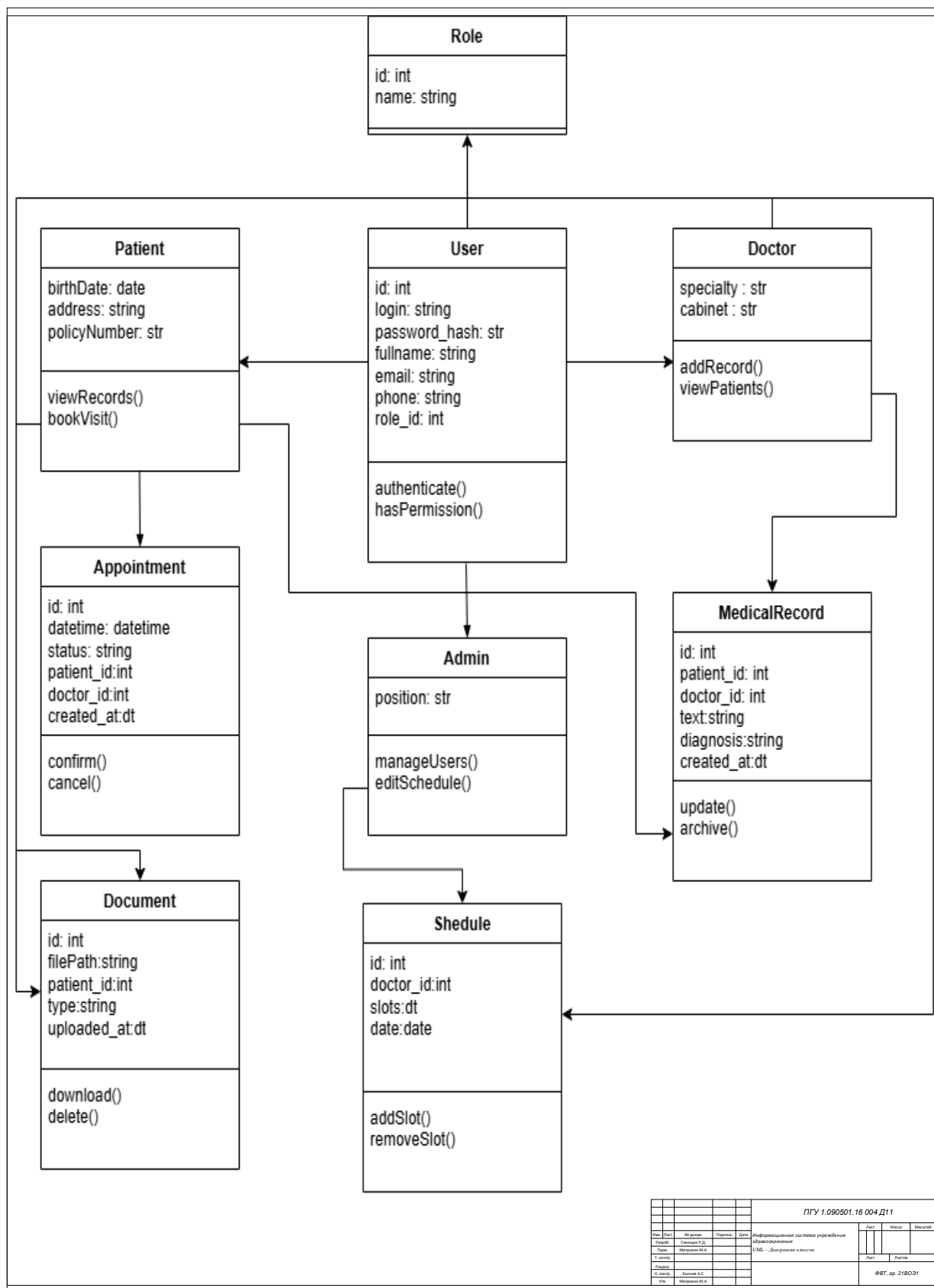
UML – Диаграмма вариантов использования



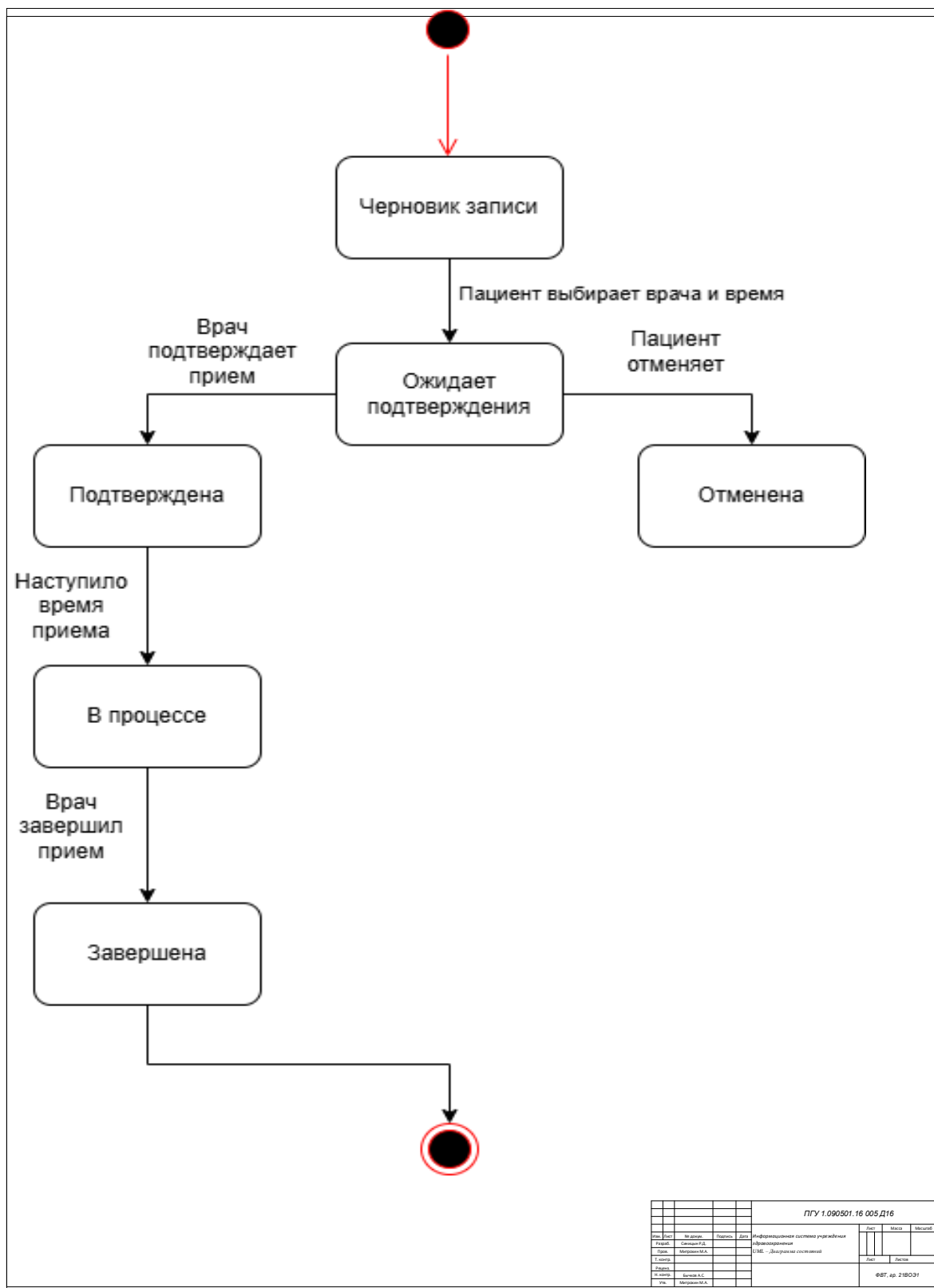
UML – диаграмма взаимодействия



UML – диаграмма классов



UML – диаграмма состояний



Приложение Б

(обязательное)

Информационная система учреждения здравоохранения

Листинг программы

					ПГУ 1.090501.16.001 ПЗ	Лист
						80
Изм.	Лист	№ докум.	Подпись	Дата		

Файл «AppointmentForm.js»

```
import React, { useState, useEffect } from "react";
import Button from "../components/common/Button";
import "./form.css";
import { fetchPatients } from "../api";

const AppointmentForm = ({ slots, onCreate }) => {
  const [patients, setPatients] = useState([]);
  const [form, setForm] = useState({
    slot_id: "",
    patient_id: "",
    complaint: ""
  });
  const [errors, setErrors] = useState({});

  useEffect(() => {
    fetchPatients()
      .then(setPatients)
      .catch(() => setPatients([]));
  }, []);

  const handleChange = (name, value) => {
    setForm((prev) => ({ ...prev, [name]: value }));
    setErrors((prev) => ({ ...prev, [name]: null }));
  };

  const validate = () => {
    const newErrors = {};
    if (!form.slot_id) newErrors.slot_id = "Выберите слот";
    if (!form.patient_id) newErrors.patient_id = "Выберите
пациента";
    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };
};
```

```

const handleSubmit = (e) => {
    e.preventDefault();
    if (!validate()) return;
    onCreate(form);
    setForm({ slot_id: "", patient_id: "", complaint: "" });
};

return (
    <form className="form" onSubmit={handleSubmit}>
        <div className="form-field">
            <label>Слот</label>
            <select
                className={errors.slot_id ? "input-error" : ""}
                value={form.slot_id}
                onChange={(e) => handleChange("slot_id",
e.target.value)}
            >
                <option value="">-- выберите --</option>
                {slots
                    .filter((s) => s.is_available)
                    .map((s) => (
                        <option key={s.id} value={s.id}>
                            {new Date(s.start_time).toLocaleString()} –{"
"}
                            {new Date(s.end_time).toLocaleString()}
                        </option>
                    ))}
                </select>
                {errors.slot_id && (
                    <div className="error-text">{errors.slot_id}</div>
                )}
            </div>

```

```

<div className="form-field">
  <label>Пациент</label>
  <select
    className={errors.patient_id ? "input-error" : ""}
    value={form.patient_id}
    onChange={(e) => handleChange("patient_id",
e.target.value)}
  >
    <option value="">-- выберите --</option>
    {patients.map((p) => (
      <option key={p.id} value={p.id}>
        {p.last_name} {p.first_name} {p.middle_name ||
""}

      </option>
    ))}
  </select>
  {errors.patient_id && (
    <div className="error-text">{errors.patient_id}</div>
  )}
</div>

<div className="form-field">
  <label>Жалоба</label>
  <textarea
    value={form.complaint}
    onChange={(e) => handleChange("complaint",
e.target.value)}
  />
</div>

<Button type="submit" variant="primary">
  Создать запись
</Button>
</form>
);

```

Файл «AppointmentPage.js»

```
import React, { useEffect, useState } from "react";
import {
  fetchDoctors,
  fetchSlotsForDoctor,
  createAppointment,
  getMyAppointments
} from "../api";

import { Table, Th, Td } from "../components/common/Table";
import Button from "../components/common/Button";
import Loader from "../components/common/Loader";

const AppointmentsPage = () => {
  const [doctors, setDoctors] = useState([]);
  const [slots, setSlots] = useState([]);
  const [myAppointments, setMyAppointments] = useState([]);
  const [selectedDoctor, setSelectedDoctor] = useState("");
  const [loadingSlots, setLoadingSlots] = useState(false);
  const [loadingMy, setLoadingMy] = useState(false);
  const [message, setMessage] = useState("");

  // Загружаем список врачей
  useEffect(() => {
    fetchDoctors().then(setDoctors);
  }, []);
```

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 1.090501.16.001 ПЗ

Лист

84

/ Загружаем записи пациента, если врач не выбран

```
useEffect(() => {  
  
  if (!selectedDoctor) {  
  
    loadMyAppointments();  
  
  }  
  
}, [selectedDoctor]);  
  
const loadMyAppointments = async () => {  
  
  setLoadingMy(true);  
  
  try {  
  
    const data = await getMyAppointments();  
  
    setMyAppointments(data);  
  
  } finally {  
  
    setLoadingMy(false);  
  
  }  
  
};  
  
const loadSlots = async (doctorId) => {  
  
  if (!doctorId) {  
  
    setSlots([]);  
  
    return;  
  
  }  
  
}
```

					ПГУ 1.090501.16.001 ПЗ	Лист
						85
Изм.	Лист	№ докум.	Подпись	Дата		

```

setLoadingSlots(true);

    try {
        const data = await fetchSlotsForDoctor(doctorId);
        setSlots(data);
    } finally {
        setLoadingSlots(false);
    }
};

const handleDoctorChange = (id) => {
    setSelectedDoctor(id);
    if (id) loadSlots(id);
};

// Запись пациента
const handleBook = async (slotId) => {
    try {
        await createAppointment({ slot_id: slotId });
        setMessage("Вы успешно записались на приём");
        loadSlots(selectedDoctor);
    } catch (err) {
        setMessage("Ошибка записи");
    }
};

return (
    <div>
        <h1 style={{ marginBottom: 20 }}>Расписание приёмов</h1>

        {message && (
            <div style={{ marginBottom: 20, color: "green",
fontSize: 16 }}>
                {message}
            </div>
        )}
    </div>
);

```

```

{ /* Выбор врача */}

<div style={{ marginBottom: 20 }}>
  <label style={{ fontSize: 14 }}>
    Выберите врача:
    <select
      value={selectedDoctor}
      onChange={(e) => handleDoctorChange(e.target.value)}
      style={{
        marginLeft: 10,
        padding: 6,
        borderRadius: 6,
        border: "1px solid #d4d9e5"
      }}
    >
      <option value="">Показать мои записи</option>
      {doctors.map((d) => (
        <option key={d.id} value={d.id}>
          {d.last_name} {d.first_name}
        </option>
      ))}
    </select>
  </label>
</div>

{ /* ----- */}

{ /* РЕЖИМ 1: врач НЕ выбран → показываем мои записи */}
{ /* ----- */}
{!selectedDoctor && (
  <>
    <h2 style={{ marginTop: 20 }}>Мои записи</h2>

    {loadingMy && <Loader />}

    {!loadingMy && myAppointments.length === 0 && (
      <p>У вас пока нет записей</p>
    )}
  </>
)}

```

Изм.	Лист	№ докум.	Подпись	Дата


```

{!loadingMy &&
    myAppointments.map((a) => (
        <div
            key={a.id}
            style={{
                padding: 12,
                border: "1px solid #d4d9e5",
                borderRadius: 6,
                marginBottom: 12
            }}
        >
            <div>
                <b>Врач:</b> {a.slot.doctor.last_name}
{a.slot.doctor.first_name}
            </div>

            <div>
                <b>Начало:</b> {new
Date(a.slot.start_time).toLocaleString()}
            </div>

            <div>
                <b>Конец:</b> {new
Date(a.slot.end_time).toLocaleString()}
            </div>

            <div>
                <b>Статус:</b> {a.status}
            </div>

```

```

{a.complaint && (
    <div>
        <b>Жалоба:</b> {a.complaint}
    </div>
    )}
</div>
)}}
</>
)}

{ /* ----- */
{ /* РЕЖИМ 2: врач выбран → показываем слоты врача */
{ /* ----- */
{selectedDoctor && !loadingSlots && (
    <>

<h2 style={{ marginTop: 20 }}>Слоты врача</h2>

    <Table>
        <thead>
            <tr>
                <Th>Начало</Th>
                <Th>Конец</Th>
                <Th>Статус</Th>
                <Th>Пациент</Th>
                <Th></Th>
            </tr>
        </thead>

<tbody>

        {slots.map((s) => (
            <tr key={s.id}>
                <Td>{new
Date(s.start_time).toLocaleString()}</Td>
                <Td>{new Date(s.end_time).toLocaleString()}</Td>
                <Td>{s.is_available ? "Свободен" : "Занят"}</Td>
                <Td>

```

```

        {s.appointment && s.appointment.patient
            ? `${s.appointment.patient.last_name}
${s.appointment.patient.first_name}`
            : "-" }
    </Td>

    <Td>
        {s.is_available && (
            <Button onClick={() => handleBook(s.id)}>
                Записаться
            </Button>
        )}

    </Td>
</tr>
    )}
</tbody>
</Table>
</>
    )}

    {selectedDoctor && loadingSlots && <Loader />}
</div>
);
};

export default AppointmentsPage;

```

Файл «BookVisitPage.js»

```
import React, { useEffect, useState } from "react";
import { fetchDoctors, fetchSlotsForDoctor, createAppointment }
from "../api";
import "../BookVisitPage.css";

const BookVisitPage = () => {
  const [doctors, setDoctors] = useState([]);
  const [slots, setSlots] = useState([]);
  const [selectedDoctorId, setSelectedDoctorId] = useState("");
  const [selectedSlotId, setSelectedSlotId] = useState("");
  const [complaint, setComplaint] = useState(""); // ← жалоба
  const [message, setMessage] = useState(null);

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

  const loadDoctors = async () => {
    const data = await fetchDoctors();
    setDoctors(data);
  };

  const loadSlots = async (doctorId) => {
    if (!doctorId) {
      setSlots([]);
      return;
    }

    const data = await fetchSlotsForDoctor(doctorId);

    // Показываем только свободные слоты
    const freeSlots = data.filter(
      (s) => s.is_available && !s.appointment
    );
  };
}
```

					ПГУ 1.090501.16.001 ПЗ	Лист
						91
Изм.	Лист	№ докум.	Подпись	Дата		

```

setSlots(freeSlots);
};

const handleSubmit = async (e) => {
  e.preventDefault();
  setMessage(null);

  if (!selectedSlotId) {
    setMessage("Выберите время приёма");
    return;
  }

  try {
    await createAppointment({
      slot_id: Number(selectedSlotId),
      complaint: complaint || null // ← передаём жалобу
    });

    setMessage("Вы успешно записались на приём");
    setSelectedSlotId("");
    setComplaint(""); // ← очищаем поле жалобы
  } if (selectedDoctorId) {
    loadSlots(selectedDoctorId);
  }

  } catch (err) {
    console.error(err);
    setMessage("Ошибка записи на приём");
  }
};

```

```

return (
  <div className="book-container">
    <h1 className="book-title">Записаться к врачу</h1>

    <form onSubmit={handleSubmit} className="book-form">

      {/* ВЫБОР ВРАЧА */}
      <label className="book-label">
        Врач:
        <select
          className="book-select"
          value={selectedDoctorId}
          onChange={(e) => {
            setSelectedDoctorId(e.target.value);
            loadSlots(e.target.value);
          }}
          <option value="">– выберите врача –</option>
          {doctors.map((d) => (
            <option key={d.id} value={d.id}>
              {d.last_name} {d.first_name}
            </option>
          ))}
        </select>
      </label>
    </form>
  </div>
)

```

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 1.090501.16.001 ПЗ

Лист

93

```

    { /* ВЫБОР ВРЕМЕНИ */ }

    <label className="book-label">
        Время:
        <select
            className="book-select"
            value={selectedSlotId}
            onChange={(e) => setSelectedSlotId(e.target.value)}
            disabled={!selectedDoctorId}
        >
            <option value="">- выберите время -</option>

            {slots.length === 0 && selectedDoctorId && (
                <option value="">Свободных слотов нет</option>
            )}

        {slots.map((s) => (
            <option key={s.id} value={s.id}>
                {new Date(s.start_time).toLocaleString()} -{" "}
                {new Date(s.end_time).toLocaleString()}
            </option>
        ))}
        </select>
    </label>
    { /* ЖАЛОБА */ }
    <label className="book-label">
        Жалоба:
        <textarea
            className="book-textarea"
            placeholder="Опишите вашу жалобу (необязательно)"
            value={complaint}
            onChange={(e) => setComplaint(e.target.value)}
            rows={3}
        />
    </label>

```

```

    <button type="submit" className="book-btn">
        Записаться
    </button>
</form>

    {message && <p className="book-message">{message}</p>}
</div>

    );
};

export default BookVisitPage;

```

Файл «ChangePasswordPage.js»

```

import React, { useState } from "react";

import { changePassword } from "../api";

import "../ChangePasswordPage.css";

const ChangePasswordPage = () => {

    const [oldPassword, setOldPassword] = useState("");

    const [newPassword, setNewPassword] = useState("");

    const [repeatPassword, setRepeatPassword] = useState("");

    const [message, setMessage] = useState(null);

    const handleSubmit = async () => {

        setMessage(null);
    }

```



```

if (newPassword !== repeatPassword) {

    setMessage("Новые пароли не совпадают");

    return;

}

try {

    await changePassword({

        old_password: oldPassword,

        new_password: newPassword,

    });

    setMessage("Пароль успешно изменён");

    setOldPassword("");

    setNewPassword("");

    setRepeatPassword("");

} catch (err) {

    console.error(err);

    setMessage(

        err.response?.data?.detail || "Ошибка при смене пароля"

    );

}

};

```

```

return (
  <div className="password-container">
    <h1 className="password-title">Смена пароля</h1>

    {message && <div className="password-
message">{message}</div>}

    <label className="password-label">
      Старый пароль:
      <input
        type="password"
        value={oldPassword}
        onChange={ (e) => setOldPassword(e.target.value)}
      />
    </label>
    <label className="password-label">
      Новый пароль:
      <input
        type="password"
        value={newPassword}
        onChange={ (e) =>
setNewPassword(e.target.value)}
      />
    </label>
    <label className="password-label">
      Повторите новый пароль:
      <input
        type="password"
        value={repeatPassword}
        onChange={ (e) =>
setRepeatPassword(e.target.value)}
      />
    </label>

```

```

    button className="password-btn" onClick={handleSubmit}>
      Изменить пароль
    </button>
  </div>
);
};

```

```
export default ChangePasswordPage;
```

Файл «DashBoard.js»

```

import React, { useEffect, useState } from "react";

import {
  fetchSummaryReport,
  getMyProfile,
  getMyAppointments,
  getMyAppointmentsAsDoctor,
  fetchSlotsForDoctor,
  getDoctorByUserId
} from "../api";

import Loader from "../components/common/Loader";
import ErrorBlock from "../components/common/ErrorBlock";
import Button from "../components/common/Button";
import { useNavigate } from "react-router-dom";

const DashboardPage = () => {
  const navigate = useNavigate();

```

```

    const [role, setRole] = useState(null);

    // patient data
    const [patientProfile, setPatientProfile] = useState(null);
    const [patientVisits, setPatientVisits] = useState([]);

    // admin summary
    const [summary, setSummary] = useState(null);

    // doctor data
    const [doctorId, setDoctorId] = useState(null);
    const [doctorAppointments, setDoctorAppointments] = useState([]);
    const [todayStats, setTodayStats] = useState({
      total: 0,
      free: 0,
      busy: 0
    });

    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    // Универсальный извлекатель пациента
    const extractPatient = (a) =>
      a.patient ||
      a.slot?.patient ||
      a.slot?.appointment?.patient ||
      null;

    const extractPatientId = (patient, a) =>
      patient?.id ||
      patient?.patient_id ||
      a.patient_id ||
      a.slot?.patient_id ||
      null;

```

```

        // 1. Определяем роль
useEffect(() => {
    const stored = localStorage.getItem("role");
    setRole(stored);
}, []);

// 2. Загружаем данные
useEffect(() => {
    if (!role) return;

// -----
// ПАЦИЕНТ
// -----
    if (role === "patient") {
        Promise.all([getMyProfile(), getMyAppointments()])
            .then(([profile, visits]) => {
                setPatientProfile(profile);
                setPatientVisits(visits || []);
                setLoading(false);
            })
            .catch(() => {
                setError("Не удалось загрузить данные пациента");
                setLoading(false);
            });
    }
}

```

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 1.090501.16.001 ПЗ

Лист

100

```

// -----
// АДМИН
// -----
if (role === "admin") {
    const today = new Date().toISOString().slice(0, 10);

    fetchSummaryReport(today)
        .then((data) => {
            setSummary(data);
            setLoading(false);
        })
        .catch(() => {
            setError("Не удалось загрузить данные отчёта");
            setLoading(false);
        });
}

// -----
// ВРАЧ
// -----
if (role === "doctor") {
    const user = JSON.parse(localStorage.getItem("user"));

    getDoctorById(user.id)
        .then((doctor) => {
            if (doctor) {
                setDoctorId(doctor.id);
            }
        });

    // Приёмы врача
    getMyAppointmentsAsDoctor().then((apps) => {
        setDoctorAppointments(apps || []);
    });
}

```

```

// Статистика слотов

    fetchSlotsForDoctor(doctor.id, true).then((slots) => {
        const today = new Date().toISOString().split("T")[0];
        const todaySlots = slots.filter((s) =>
            s.start_time.startsWith(today)
        );

        setTodayStats({
            total: todaySlots.length,
            free: todaySlots.filter((s) =>
s.is_available).length,
            busy: todaySlots.filter((s) =>
!s.is_available).length
        });
    });

    setLoading(false);
})
.catch(() => {
    setError("Не удалось загрузить данные врача");
    setLoading(false);
});
}

}, [role]);
if (loading) return <Loader />;
if (error) return <ErrorBlock message={error} />;

// -----
// PATIENT DASHBOARD
// -----

if (role === "patient") {
    const futureVisits = patientVisits.filter(
        (v) => new Date(v.slot.start_time) > new Date()
    );
}

```

```

const nextVisit =
  futureVisits.length > 0
    ? futureVisits.sort(
      (a, b) =>
        new Date(a.slot.start_time) - new
Date(b.slot.start_time)
    ) [0]
    : null;

const formatDate = (value) => new Date(value).toLocaleString();
const statusMap = {
  pending: "Ожидает",
  confirmed: "Подтверждён",
  cancelled: "Отменён",
  completed: "Завершён",
  in_progress: "В процессе",
};
return (
  <div>
    <h1 style={{ marginBottom: 20 }}>
      Добро пожаловать, {patientProfile.first_name}
    </h1>

    { /* Ближайший визит */ }
    <div className="next-visit-block" style={{ marginBottom: 20
}}>

      <div className="next-visit-title">Ближайший визит</div>

      {nextVisit ? (
        <div className="next-visit-info">
          <div>
            <strong>Дата:</strong>
{formatDate(nextVisit.slot.start_time)}
          </div>
          <div>

```



```

<strong>Врач:</strong>{" "}
  {nextVisit.slot.doctor
    ? `${nextVisit.slot.doctor.last_name} ${nextVisit.slot.doctor.first_name}`
    : "Не указан"}
</div>
<div>
  <strong>Статус:</strong> {statusMap[nextVisit.status] || nextVisit.status}

</div>
</div>
):(

  <div style={{ marginTop: 10 }}>У вас нет предстоящих визитов</div>
  )}
</div>

  {/* Быстрая статистика */}
  <div className="dashboard-grid" style={{ marginBottom: 20 }}>
    <div className="dashboard-card">
      <div className="dashboard-label">Будущих визитов</div>
      <div className="dashboard-value">{futureVisits.length}</div>
    </div>

    <div className="dashboard-card">
      <div className="dashboard-label">Полис</div>
      <div className="dashboard-value">
        {patientProfile.insurance_number || "—"}
      </div>
    </div>

    <div className="dashboard-card">
      <div className="dashboard-label">Телефон</div>
      <div className="dashboard-value">
        {patientProfile.phone || "—"}
      </div>
    </div>
  </div>
  </div>
  </div>
  );
}
/ -----
// DOCTOR DASHBOARD
// -----
if (role === "doctor") {
  const today = new Date().toISOString().split("T")[0];

  const todaysAppointments = doctorAppointments.filter(
    (a) =>
      extractPatient(a) &&
      a.slot.start_time.startsWith(today)
  );
}

```

Изм.	Лист	№ докум.	Подпись	Дата

Приложение В

(обязательное)

«Информационная система учреждения здравоохранения»

Презентация

					ПГУ 1.090501.16.001 ПЗ	Лист
						105
Изм.	Лист	№ докум.	Подпись	Дата		

Слайд 1:

Информационная система учреждения здравоохранения

Дипломная работа

Студент: Сеницын Р.Д.
Группа: 21ВОЭ1



Слайд 2:

Актуальность и проблема

Проблема:

- Рост объёма бумажной документации в поликлиниках
- ошибки при ручной записи
- очереди в регистратуру
- разрозненность данных между врачами и администрацией.

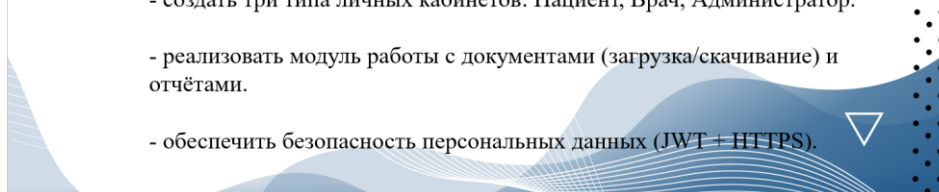
Цель работы: создать недорогую, веб-ориентированную систему для автоматизации записи и ведения электронной медицинской карты (ЭМК).



Слайд 3:

Задачи работы:

- провести анализ предметной области и существующих МИС (Медицинских Информационных Систем).
- спроектировать клиент-серверную архитектуру и базу данных (PostgreSQL).
- разработать серверную логику (FastAPI) для управления расписанием и приёмами.
- создать три типа личных кабинетов: Пациент, Врач, Администратор.
- реализовать модуль работы с документами (загрузка/скачивание) и отчётами.
- обеспечить безопасность персональных данных (JWT + HTTPS).



Изм.	Лист	№ докум.	Подпись	Дата

Слайд 4:



Анализ аналогов

- Медси:
 - +высокая функциональность
 - -высокая стоимость, сложность адаптации
- 1С:Медицина:
 - +гибкость учёта
 - -сложный интерфейс, ориентация на бухгалтерию
- Инфоклиника:
 - + простота использования, доступность
 - - ограниченная функциональность, закрытая архитектура, слабая расширяемость.





Слайд 5:



Архитектура и технологии

Архитектура: Клиент-серверная (Веб-приложение). Клиент не знает про базу данных.

Frontend (Клиент): Библиотека React (SPA, компоненты, JSX).

Backend (Сервер): Фреймворк FastAPI (Python, асинхронность, высокая производительность).

База данных: PostgreSQL (ACID, целостность данных).

Протокол: REST API, JSON, HTTPS.







Слайд 6:



