

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

Факультет
вычислительной техники

Кафедра
МО и ПЭВМ

Направление подготовки 09.03.02 «Информационные системы и технологии»

Направленность Информационные системы и технологии

БАКАЛАВРСКАЯ РАБОТА

на тему

Разработка персональной системы напоминаний
пользователя на основе Telegram-бота

Студент _____ Светов Николай Вячеславович
(подпись, дата) (ФИО полностью)

Руководитель _____ Кольчугина Е.А.
(подпись, дата) (фамилия, инициалы)

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

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

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

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

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

Пенза, 2025

Федеральное государственное бюджетное образовательное учреждение
высшего образования

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МО и ПЭВМ

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

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

1. Студент Светов Николай Вячеславович гр. 21ВИ1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Разработка персональной системы напоминаний пользователя на основе Telegram-бота

Тема утверждена приказом ПГУ № _____ от " ____ " _____ 2024 г.

3. Руководитель работы профессор кафедры «МО и ПЭВМ» Кольчугина Е.А.

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

1) *Разработать приложение создания напоминаний, с возможностью их просмотра, редактирования, удаления.*

2) *Среда и средства разработки:*

Языковые средства разработки: Python, UML.

Инструментальные средства: Visual Studio 2022, Visual Paradigm, Draw.io

*Технологии разработки: ООП, шаблоны проектирования, Telegram API
сервисно-ориентированный подход.*

3) *Среда функционирования: мессенджер Telegram:*

4) *Основные функции приложения:*

а) создание напоминания;

б) просмотр напоминаний;

в) редактирование напоминания;

г) удаление напоминания;

5) *Особенности: адаптация под все устройства, способные поддерживать
корректную работу приложения Telegram, локализация*

6) *Требования к СУБД и модели данных: гибкость, масштабируемость, быстрота
чтения/записи*

7) *Провести тестирование с использованием мессенджера Telegram*

5. Перечень подлежащих разработке вопросов:

1) Анализ предметной области и постановка задачи;
2) Проектирование;
3) Реализация;
4) Контроль качества программного обеспечения;
5) План разработки бюджета;
6) Оформление пояснительной записки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и постановка задачи	10%	13.04.2025 – 27.04.2025	
2	Проектирование	25%	27.04.2025 – 10.05.2025	
3	Реализация	35%	10.05.2025 – 30.06.2025	
4	Контроль качества программного обеспечения	10%	31.05.2025 – 02.06.2025	
5	План разработки бюджета	5%	02.06.2025 – 03.06.2025	
6	Оформление пояснительной записки	15%	03.06.2025 – 14.06.2025	

Дата выдачи задания «22» _____ декабря _____ 2024 г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан факультета _____
(подпись, дата)

Реферат

Пояснительная записка содержит 103 листа, 39 рисунков, 11 таблиц, 20 использованных источников, 3 приложения.

ЧАТ-БОТ, СИСТЕМА НАПОМИНАНИЙ, PYTHON, UML, БАЗА ДАННЫХ, MICROSOFT VISUAL STUDIO, SQLITE3

Объектом разработки является система персональных напоминаний пользователя.

Цель выпускной квалификационной работы – разработка персональной системы напоминаний пользователя, которая позволит развить пунктуальность и повысить производительность труда.

Технология разработки – чат-бот для работы в мессенджере Telegram на языке программирования Python 3.9. Логика бота включает обработку команд, многошаговые диалоги. Для хранения данных используется встраиваемая база данных SQLite3. Среда разработки – Microsoft Visual Studio 2022.

В результате выпускной квалификационной работы были разработаны клиентская и серверная часть персональной системы напоминаний пользователя на основе Telegram-бота. Все требования к разработке приложения соблюдены.

					ПГУ 09.03.02– 8BP191.27 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Разработка персональной системы напоминаний пользователя на основе Telegram бота Пояснительная записка	Лит.	Лист	Листов
Разраб.		Светов Н.В.						
Пров.		Кольчугина Е.А.					4	103
Н. Контр.		Такташкин Д.В.				Группа 21ВИ1		
Утв.		Козлов А.Ю.						

Содержание

Введение.....	7
1 Анализ альтернативных реализаций систем напоминаний	9
1.1 Анализ предметной области и постановка задачи.....	9
1.2 Обзор и анализ программного обеспечения аналогичного назначения	10
1.2.1 Анализ программы TeuxDeux	10
1.2.2 Анализ органайзера напоминаний Google Tasks	11
1.2.3 Анализ программы TiMemory для напоминания.....	12
1.2.4 Сравнительный анализ аналогичных систем	13
1.3 Выбор методов и средств решения поставленной задачи.	14
2 Проектирование персональной системы напоминаний пользователя	19
2.1 Архитектура приложения.....	19
2.2 Проектирование структуры данных.....	20
2.2.1 Концептуальная модель базы данных.....	20
2.2.2 Логическая модель базы данных	22
2.3 Проектирование пользовательского интерфейса.....	23
2.4 Проектирование программных средств	30
2.4.1 Проектирование классов серверной части приложения	30
2.4.3 Проектирование алгоритма удаления напоминания	34
3 Реализация персональной системы напоминаний пользователя	36
3.1 Реализация серверной части приложения	36
3.2 Реализация базы данных.....	39
4 Контроль качества программного обеспечения.....	43
4.1 Выбор режима тестирования	43
4.2 Тестирование серверной части приложения	44
4.3 Тестирование клиентской части приложения	46

5 Контроль качества.....	48
5.1 Метрики Холстеда.....	48
5.2 Метрики SLOC	51
6 Планирование разработки и оценка бюджета.....	53
Заключение	56
Список использованных источников	58
Приложение А (рекомендуемое) Глоссарий	61
Приложение Б (обязательное) Листинг программы.....	63
Приложение В (обязательное) Результаты тестирования.....	100

Введение

В современном мире с высоким темпом жизни и большим объёмом задач всё чаще возникает необходимость в инструментах, помогающих организовать время и не упустить важные события. Люди привыкли писать напоминания просто в блокнот на телефоне или же в свой ежедневник, что является не совсем правильным, ведь для профессионального подхода к своему времени необходимо использовать специализированное пространство, а именно приложение или сервис.

Существующих решений крайне мало, они имеют ограниченный функционал или сложны в использовании. Особенно остро стоит проблема с напоминаниями, которые требуют гибкости в настройке времени, повторений и приоритетов.

Все вышесказанное показывает необходимость создания удобного и функционального чат-бота для управления напоминаниями в Telegram, как в одном из самых популярных мессенджеров, предоставляющим большие возможности для создания ботов, способных быстро и качественно помочь в распределении своего времени, одновременно объединив в себе простоту использования и мощный функционал для эффективного управления напоминаниями [1].

Целью выпускной квалификационной работы является разработка персональной системы напоминаний пользователя на основе Telegram-бота. Разработка включает в себя анализ требований, проектирование архитектуры, реализацию программного обеспечения и его тестирование.

Для выполнения выпускной квалификационной работы были поставлены следующие цели:

- изучить аналогичные программные продукты и предоставить сравнительный анализ;
- выполнить проектирование архитектуры приложения и структуры данных разрабатываемого программного обеспечения;

- выполнить разработку программного обеспечения;
- провести тестирование разработанного приложения;
- оформить пояснительную записку в соответствии с требованиями.

1 Анализ альтернативных реализаций систем напоминаний

1.1 Анализ предметной области и постановка задачи

Классические методы напоминаний (бумажные списки, календари или встроенные приложения) имеют ряд недостатков. Во-первых, они не предоставляют гибких настроек повторения и приоритизации задач. Во-вторых, ручное управление напоминаниями требует больших временных затрат. В-третьих, сами напоминания не всегда могут быть доступны в нужный момент.

Telegram-бот для управления напоминаниями это цифровой помощник, который доступен каждому и в каждой точке мира. Он позволит решить все вышеперечисленные проблемы. Пользователям будет предоставлена возможность создавать, редактировать, удалять и получать напоминания о важных событиях. Чат-бот сможет собирать всю основную информацию о напоминаниях: название, описание, дату и время напоминания, параметры повторения и приоритет. Бот служит инструментом для удобной организации личного времени, автоматизации напоминаний.

В зависимости от потребностей пользователя, бот позволяет настраивать различные типы напоминаний:

- разовые или повторяющиеся (ежедневно, еженедельно, ежемесячно, ежегодно);
- с разными уровнями приоритета (высокий, средний, низкий);
- с возможностью гибкого редактирования параметров.

Основными пользователями бота являются люди, которым важно не забыть об особенных событиях и необходимых задачах. Категория пользователей системы охватывает многих — от школьников до успешных бизнесменов.

Бот решает ключевые проблемы традиционных методов:

- обеспечивает удобное создание и редактирование напоминаний через интуитивно понятный интерфейс [2];
- автоматизирует процесс уведомлений в заданное время;
- предоставляет надёжное хранение данных с возможностью просмотра истории прихода напоминаний.

Таким образом, требуется разработать кроссплатформенный Telegram-бот, который включает:

- серверную часть для обработки команд и хранения данных;
- клиентский интерфейс в мессенджере [3];
- базу данных для надёжного хранения информации о напоминаниях;
- систему отправки напоминаний.

1.2 Обзор и анализ программного обеспечения аналогичного назначения

1.2.1 Анализ программы TeuxDeux

Первым на рассмотрении среди аналогов был выбран TeuxDeux. Данное решение является онлайн сервисом, то есть поддерживается на персональных устройствах, способных запустить любой из доступных браузеров. На рисунке 1 приведено окно интерфейса данного программного обеспечения.

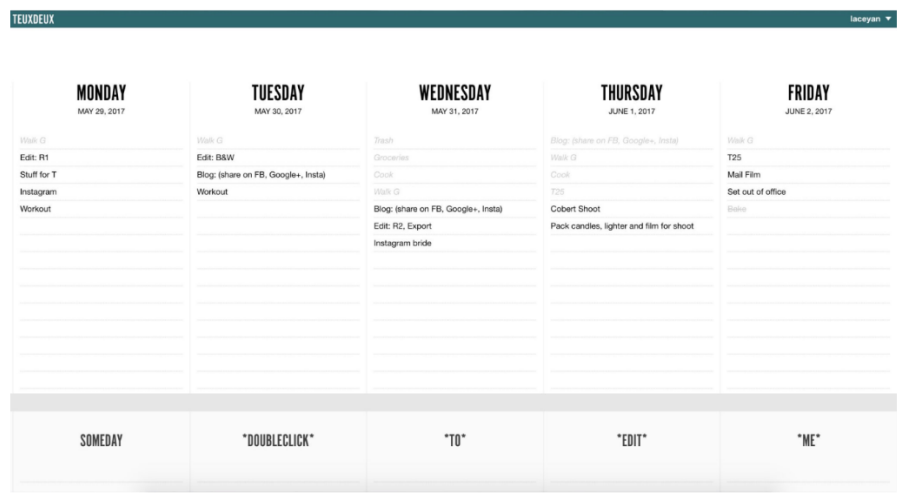


Рисунок 1 – Окно программы TeuxDeux

Одним из главных достоинств данной программы является юзер-френдли интерфейс с простым и красивым дизайном, регистрация занимает несколько секунд, нужно просто ввести электронный адрес с паролем.

Недостатком является платный доступ к большей части возможностей сервиса, а также отсутствие гибкой настройки даты напоминаний.

1.2.2 Анализ органайзера напоминаний Google Tasks

Google Tasks – удобный планировщик задач и органайзер напоминаний. Реализован в виде доски с картами для записи рабочих и личных дел. Google Tasks имеет лаконичный интерфейс, который будет по нраву ценителям минимализма. Отслеживать выполнение напоминаний можно через почту или календарь в телефоне, при условии синхронизации аккаунтов. Интерфейс представлен на рисунке 2.

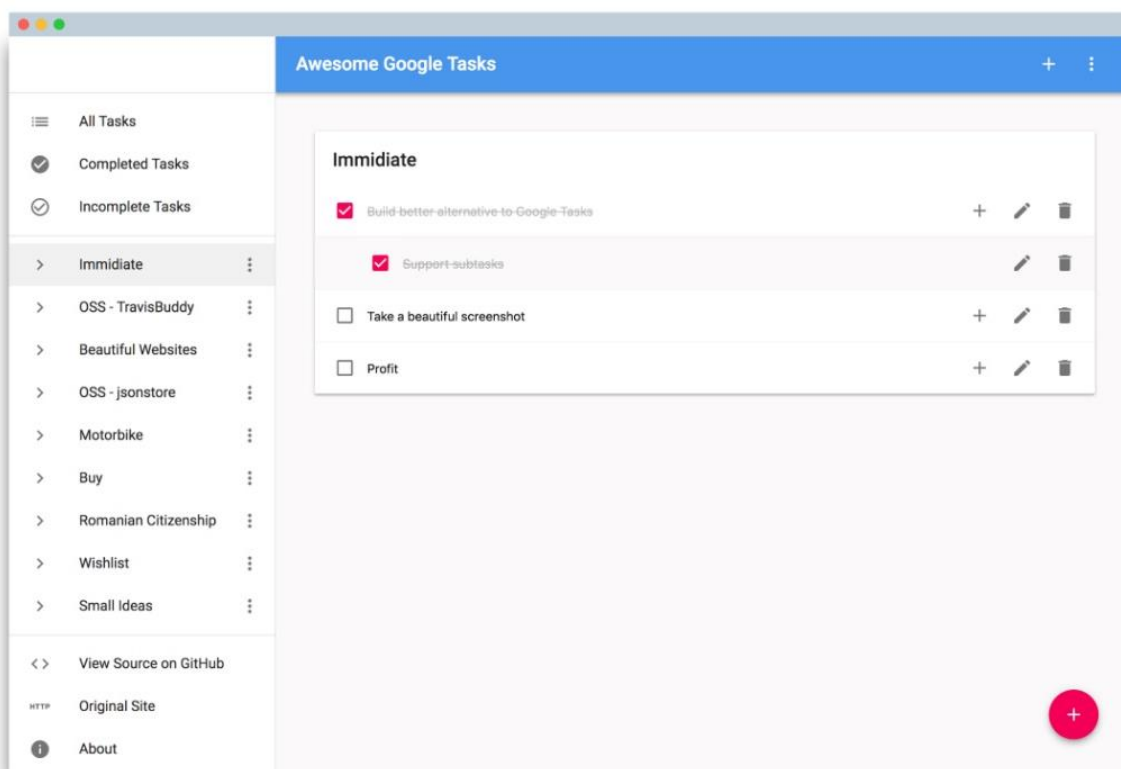


Рисунок 2 – Интерфейс сервиса Google Tasks

Основным преимуществом Google Tasks является онлайн формат, что позволяет иметь доступ к нему в любом месте с доступом к интернету.

Недостатком является обязательное подключение к сети, без которой доступ к списку задач и напоминаний получить невозможно. Также к проблемам данного аналога можно отнести ограниченный функционал. Сам сайт сервиса не является в полной мере оптимизированным, что может быть вызвать определённые проблемы на устройствах со слабой производительностью.

1.2.3 Анализ программы TiMemory для напоминания о незавершённых делах

TiMemory – программа для персональных устройств под управлением операционной системы Windows для напоминания о незавершенных делах. Работает как будильник, отправляя уведомления пользователю до тех пор, пока не получит реакцию на них или не отключит. Если напоминание придёт, когда устройство будет выключено, сообщение придёт сразу после включения. На рисунке 3 приведён интерфейс программы TiMemory.

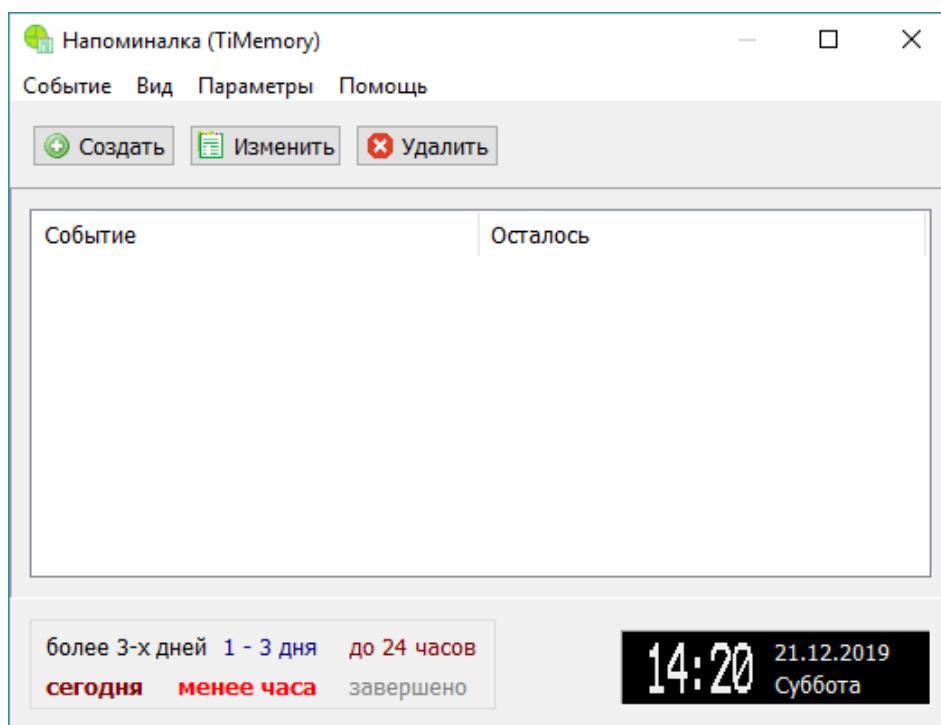


Рисунок 3 – Интерфейс главной странички TiMemory

Основным достоинством данного сервиса является автоматическая синхронизация для одновременной работы сразу на нескольких персональных компьютерах, к примеру на ноутбуке и компьютере. Огромным плюсом является наличие звуковых уведомлений о приходе напоминаний, также есть возможность хранения и открытия ссылок прямо из приложения.

К недостаткам стоит отнести платный доступ к основному функционалу программы после демоверсии в две недели. В приложении отсутствует английская локализация, что также может негативно сказаться на пользовательском опыте [4]. Программа доступна исключительно на устройствах с операционной системой Windows.

1.2.4 Сравнительный анализ аналогичных систем

Проведено качественное сравнение рассмотренных ранее аналогичных программных продуктов.

TeuxDeux – это минималистичный и стильный планировщик задач с упором на визуальную простоту. Его главное преимущество – чистый, интуитивный интерфейс без лишних функций – идеально для тех, кто хочет сфокусироваться на задачах. Однако сервис платный, а бесплатной версии с ограничениями нет. Кроме того, он доступен только на iOS и в веб-версии, что исключает пользователей Android. Отсутствие интеграций с другими сервисами и русского языка также является весомым недостатком системы.

Google Tasks – полностью бесплатный и простой инструмент, глубоко интегрированный в экосистему Google. Он синхронизируется с Gmail и Google Календарём, позволяя создавать задачи прямо из писем. Подходит для тех, кто хочет быстро записывать дела без сложных настроек. Однако функционал здесь базовый – нет тегов, фильтров или гибких повторяющихся задач. Зато он работает на всех платформах, включая Android и iOS, и не требует подписки.

TiMemory – это платное приложение с уникальным подходом: оно фокусируется не только на задачах, но и на учёте времени. Задачи отображаются в виде временных блоков, это помогает оценить сколько

времени уходит на выполнение. Однако, у него есть серьёзные ограничения: работает только на операционной системе Windows, нет мобильной версии, а интерфейс может показаться сложным для пользователей, только начавших пользоваться сервисом.

Качественное сравнение аналогов представлено в таблице 1.

Таблица 1 – Сравнение аналогичных систем

Системы-аналоги	Интерфейс	Кроссплатформенность	Русский язык	Платное
TeuxDeux	Развитый	Да	Отсутствует	Да
Google Tasks	Развитый	Да	Присутствует	Нет
TiMemory	Базовый	Нет	Присутствует	Да

Оптимальным выбором для работы с напоминаниями будет Google Tasks – он бесплатный, кроссплатформенный и достаточно удобный для повседневного использования.

1.3 Выбор методов и средств решения поставленной задачи.

Изучив аналогичные программные решения, были сформированные следующие функциональные требования к разрабатываемому приложению:

- обеспечение создания и хранения напоминаний;
- просмотр всех напоминаний;
- редактирование напоминаний;
- возможность просмотра инструкции по использованию бота;
- возможность выбора языка (русский/английский);
- просмотр истории напоминаний.

Исходя из описанных выше функциональных требований была разработана UML диаграмма вариантов использования, которая представлена на рисунке 4.

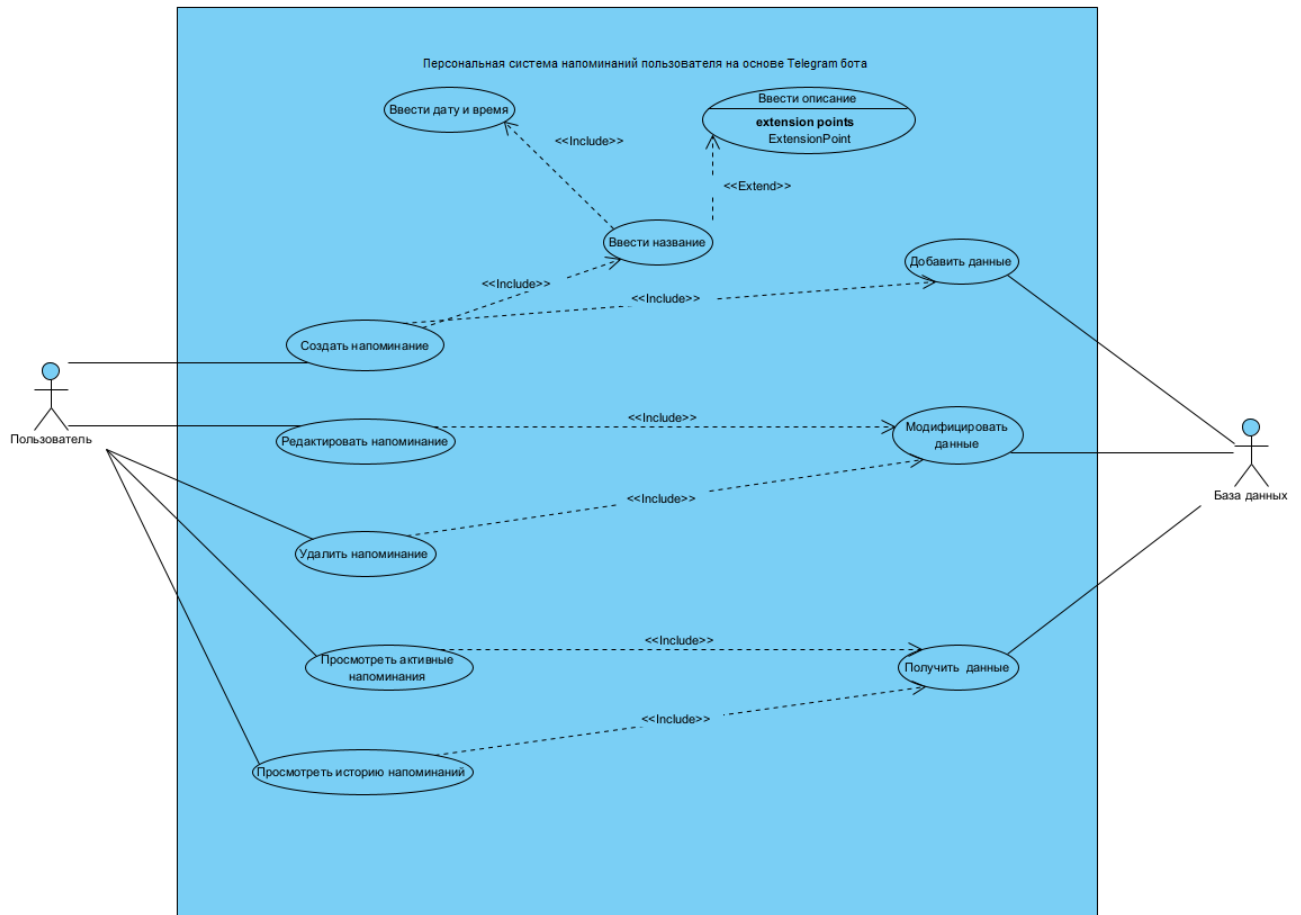


Рисунок 4 – Диаграмма вариантов использования системы напоминаний пользователя на основе Telegram бота

Для более подробной документации и визуализации, а также для более чёткого понимания функциональных требований разрабатываемого приложения, были описаны сценарии основных вариантов использования: создание напоминания, редактирование напоминания, удаление напоминания, просмотр активных напоминаний и истории их прихода. Все они изображены на данной диаграмме [5].

На рисунке 5 представлен сценарий для варианта использования «Создать напоминание».

Прецедент: Создать напоминание
ID: 1
Краткое описание: Пользователь создаёт новое напоминание
Главные актеры: Пользователь
Второстепенные актеры: Нет
Предусловие: 1. Пользователь открыл приложение.
Основной поток: 1. Если пользователь создаёт напоминание и заполняет все поля 1.1 Создаётся напоминание 2. Иначе, если пользователь выбирает вернуться обратно в меню 2.1 Приложение прерывает создание нового напоминания
Постусловия: Напоминание сохраняется в базе данных
Альтернативные потоки: Нет

Рисунок 5 – Сценарий варианта использования «Создать напоминание»

На рисунке 6 представлен сценарий варианта использования «Редактировать данные напоминания».

Прецедент: Редактировать данные напоминания
ID: 11
Краткое описание: Пользователь редактирует напоминание
Главные актеры: Пользователь
Второстепенные актеры: Нет
Предусловие: 1. Пользователь нажал на кнопку редактирование 2. Пользователь выбрал нужное напоминание
Основной поток: 1. Пользователь внес изменения напоминания
Постусловия: Внесенные изменения сохраняются в базе данных
Альтернативные потоки: Нет

Рисунок 6 – Сценарий варианта использования «Редактировать данные напоминания»

Следующим на рассмотрении будет сценарий удаления напоминания. На рисунке 7 представлен сценарий варианта использования «Удалить напоминание».

Прецедент: Удалить напоминание
ID: 5
Краткое описание: Пользователь удаляет напоминание
Главные актеры: Пользователь
Второстепенные актеры:
Предусловие: 1. Пользователь нажал на кнопку удаления 2. Пользователь выбрал нужное напоминание для удаления
Основной поток: 1. Если пользователь нажимает на напоминание и подтверждает его удаление 1.1 Удаляется напоминание 2. Иначе, если пользователь выбирает нужное напоминание, но уходит назад в меню 2.1 Приложение прерывает удаление напоминания
Постусловия: Данные о напоминании удаляются
Альтернативные потоки: Нет

Рисунок 7 – Сценарий варианта использования «Удаление напоминания»

В ходе анализа функциональных требований к разрабатываемому приложению была создана диаграмма вариантов использования, а также выделены основные её элементы и их спецификации, которые отражают ход взаимодействия пользователя с приложением.

Нефункциональные требования описывают такие характеристики системы, которые не связаны с её функционалом. В ходе разработки персональной системы напоминаний пользователя на основе Telegram-бота были определены следующие нефункциональные требования:

- удобство и простота использования – интерфейс системы должен быть интуитивно понятен и прост в использовании;
- доступность – система должна быть доступна с минимальным временем простоя;
- производительность – система должна обеспечивать быстрое время отклика;
- масштабируемость – система должна поддерживать увеличение количества данных без значительной потери производительности;
- расширяемость – архитектура системы должна быть гибкой, чтобы позволять добавление новых функций.

Исходя из выявленных требований приложение будет представлять чат-бота в Telegram. Мессенджер является платформой для разработки чат-ботов с открытым исходным кодом и API, разработанной компанией Telegram FZ-LLC (Free Zone Limited Liability Company). Таким образом и серверная и клиентская части будут написаны на языке Python с использованием API мессенджера и библиотек, необходимых для работы с Telegram.

В качестве базы данных была выбрана SQLite3 [6]. Данная встраиваемая реляционная база данных обладает рядом преимуществ, таких как: широкая совместимость, простота развёртывания, надёжность и стабильность. В целом это проверенное решение с хорошей отказоустойчивостью и экономией ресурсов.

В качестве среды разработки всего проекта будет использоваться Microsoft Visual Studio 2022. Клиентское приложение должно успешно функционировать в обновлённой версии Telegram абсолютно в любой системе: Windows, IOS, Android, Linux [7].

2 Проектирование персональной системы напоминаний пользователя на основе Telegram-бота

2.1 Архитектура приложения

Разработка чат-бота подразумевает использование трёхуровневой конструкции архитектуры [8]. Схема архитектуры представлена на рисунке 8.

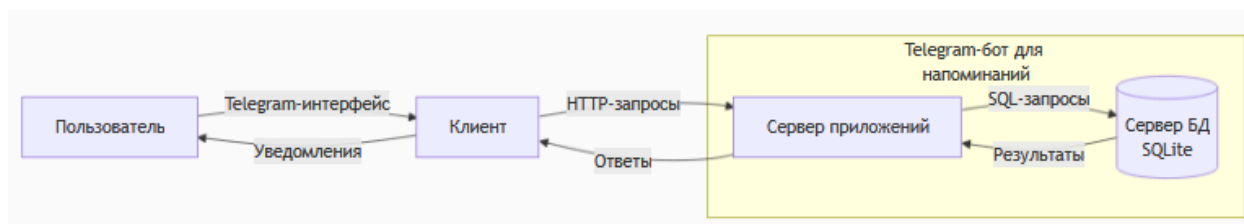


Рисунок 8 – Клиент-серверная архитектура

В данной трёхуровневой архитектуры есть три ключевых слоя:

- слой представления (клиентский уровень);
- слой бизнес-логики (уровень сервера);
- слой данных (уровень хранения).

Слой представления выполняет роль пользовательского интерфейса и включает в себя Telegram Bot API Gateway, с помощью которого возможна коммуникация пользователя и системы через мессенджер, динамические интерфейсные элементы, такие как адаптивные клавиатуры (ReplyKeyboard, InlineKeyboard), контекстное меню и шаблоны сообщений, систему локализации с выбором русского и английского языков, а также валидатор ввода, помогающий проверять вводимые пользователем временные параметры на корректность [9]. Особенность уровня состоит в реализации контекстно-зависимого интерфейса, который изменяется в зависимости от состояния диалога с пользователем.

Уровень сервера же это своеобразное ядро системы, включающее в себя несколько модулей. Первый модуль – это менеджер напоминаний, который отвечает за выполнение CRUD-операций с напоминаниями, валидацию временных параметров, обработку повторяющихся событий [10]. Следующий

модуль – сервис уведомлений, который используется для отложенного выполнения задач, он поддерживает различные настройки повторения, такие как ежедневные или еженедельные события, и обеспечивает автоматическую коррекцию временных зон с использованием интеграции pytz. В системе присутствует поддержка интерфейса на двух языках – русском и английском, есть динамическое переключение языков.

Слой данных обеспечивает надёжное хранение информации и включает в себя модель данных и технологический стек.

Модель данных содержит сущность Напоминание, которая имеет несколько полей: базовые атрибуты, такие как название и описание, временные параметры, настройки повторения и приоритет напоминания. Также в модели данных предусмотрена история срабатываний и пользовательские настройки. Технологический стек включает использование SQLite в качестве системы управления данными. Для повышения эффективности используются оптимизированные запросы. Операции выполняются транзакционно для обеспечения целостности данных, для обеспечения безопасности используются подготовленные выражения.

Особенностью реализации является асинхронная обработка запросов с использованием `async/await`, управление состоянием диалога и обработка исключительных ситуаций [11].

2.2 Проектирование структуры данных

2.2.1 Концептуальная модель базы данных

На рисунке показана концептуальная модель базы данных в виде диаграммы «сущность-связь», отражающая предметную область разрабатываемой информационной системы. Эта модель включает ключевые сущности, то есть таблицы и связи между ними, без учета ограничений конкретных СУБД. Как правило, в подобной модели указываются только названия сущностей, без какой-либо детализации атрибутов. Создаётся

концептуальная модель для проектирования базы данных и перевод её, к примеру, в реляционную базу данных.

В нашем случае модель данных состоит из трёх основных сущностей:

- reminders – хранение напоминаний;
- history – история срабатывания напоминаний;
- user_settings – настройки.

Концептуальная модель базы данных представлена на рисунке 9.



Рисунок 9 – Концептуальная модель базы данных

В таблице 2 представлены сущности, сформированные на основании определённых данных бизнес-процессов.

Таблица 2 – Описание сущностей информационной системы

Наименование сущности	Идентификатор сущности	Назначение сущности
Напоминания	reminders	Основная таблица для хранения напоминаний. Содержит информацию о названии, описании, времени срабатывания, настройках повторения, приоритете и статусе выполнения напоминания.
История	history	Таблица истории срабатывания напоминаний. Фиксирует когда и какие напоминания срабатывали для каждого из пользователей.

Продолжение таблицы 2

Наименование сущности	Идентификатор сущности	Назначение сущности
Настройки пользователя	user_settings	Таблица пользовательских настроек. В текущей реализации хранит только выбранный язык интерфейса для каждого пользователя.

Концептуальная модель была спроектирована в среде Visual Paradigm в нотации ERD (Entity Relationship Diagram) [12].

2.2.2 Логическая модель базы данных

На основе концептуальной модели данных была разработана логическая модель, которая расширяет и детализирует её структуру. Логическая модель включает сущности, атрибуты, ключи и связи, отражающие бизнес-правила и информацию предметной области. В данном случае она описывает базу данных персональной системы напоминаний пользователя на основе Telegram-бота. Диаграмма логической модели представлена на рисунке 10.

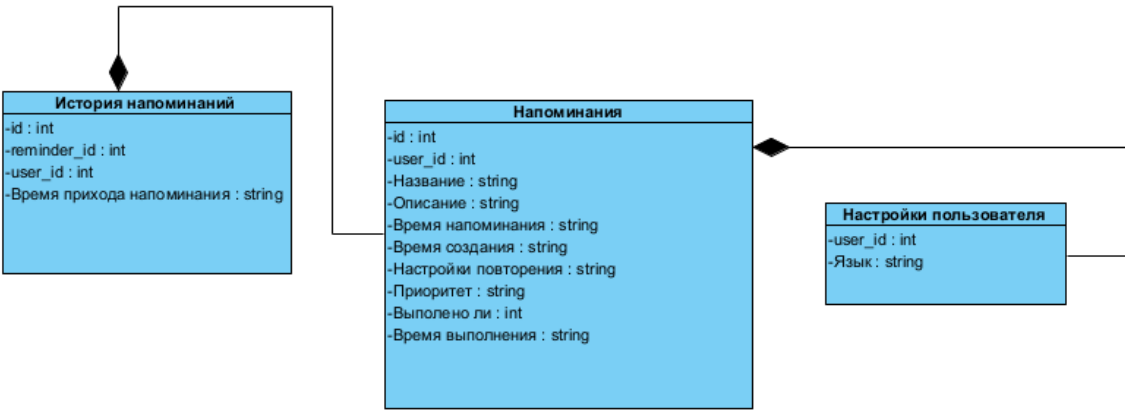


Рисунок 10 – Логическая модель базы данных

Логическая модель построена в среде Visual Paradigm в нотации UML Class Diagram.

2.3 Проектирование пользовательского интерфейса

На основе анализа функциональных требований, а также диаграммы вариантов использования был спроектирован интерактивный интерфейс Telegram-бота, обеспечивающий удобное управление напоминаниями. Интерфейс реализован с использованием библиотеки python-telegram-bot и включает в себя несколько ключевых моментов. На рисунке 11 представлено главное меню бота.

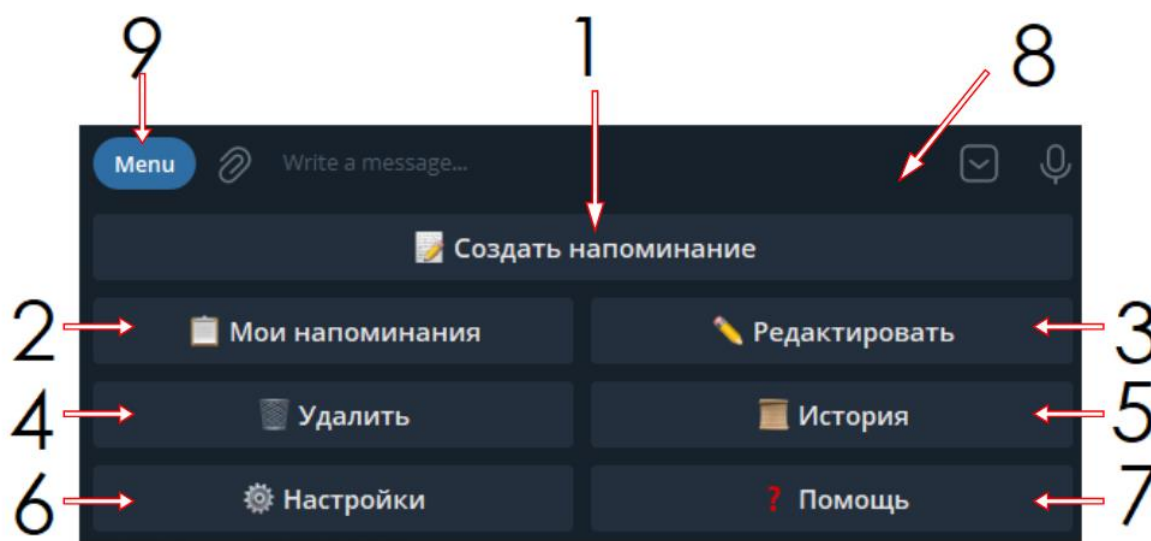


Рисунок 11 – Главное меню системы персональных напоминаний на основе Telegram-бота

Основные элементы окна главного меню:

- 1 – кнопка, инициирующая начало создания напоминания;
- 2 – кнопка, позволяющая посмотреть личные напоминания пользователя;
- 3 – кнопка, позволяющая начать редактирование информации напоминаний;
- 4 – кнопка, способная начать процедуру удаления напоминания;
- 5 – кнопка, открывающая историю сработавших напоминаний;
- 6 – кнопка настроек, в меню которых в дальнейшем можно изменить параметры локализации;

7 – кнопка, открывающая Telegraph-статью с подробной инструкцией по использованию бота для тех, кто впервые пользуется сервисом;

8 – пространство для дальнейшего ввода каких-либо данных напоминания;

9 – кнопка меню, добавленная средствами Telegram Father, то есть сервисом, управляющим всей экосистемой чат-ботов в мессенджере Telegram.

При нажатии по кнопке «Создать напоминание» бот пришлёт сообщение с предложением ввести название напоминания, снизу сообщения находится кнопка с одноимённым названием «Назад в меню». Реакция бота на действие пользователя показана на рисунке 12.

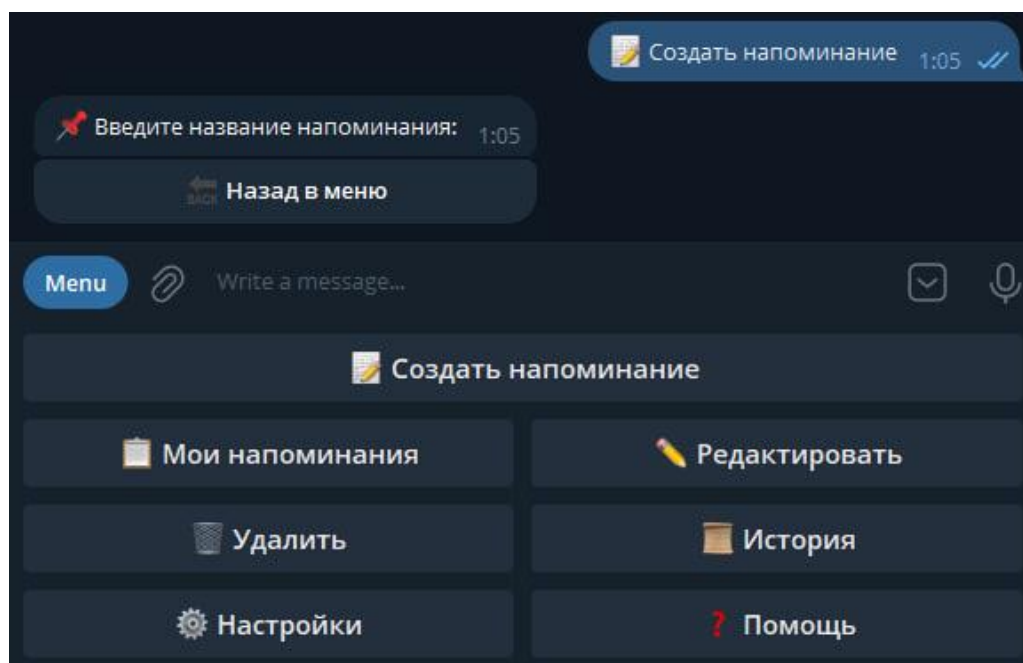


Рисунок 12 – Первый этап создания напоминания

Во время создания напоминания доступны все остальные функции программы, которыми можно воспользоваться, нажав на одну из кнопок. Кнопка «Назад в меню» отменяет создание напоминания и вводит программу в состояние игнорирования сообщений, введённых пользователем. В нашем случае продолжаем дальше. Ввод названия, а затем и описания напоминания приведён на рисунке 13.

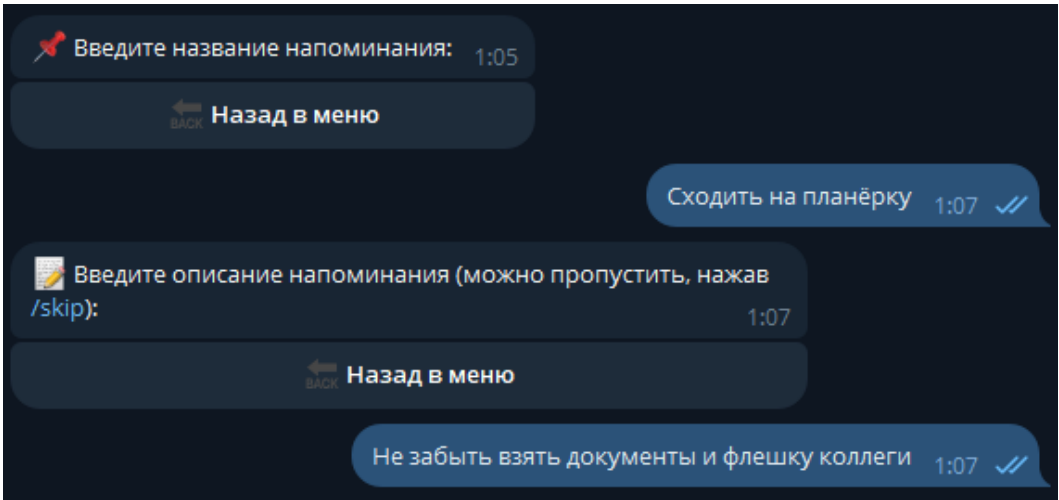


Рисунок 13 – Второй этап создания напоминания – ввод названия и описания

Далее бот просит ввести время в нужном формате. Часовой пояс берётся с сервера, где работает бот – по умолчанию. Для реализации используется библиотека `pytz`. Она позволяет корректно конвертировать время между разными поясами и учитывать исторические изменения (например, переход на летнее время). После корректного ввода времени, бот просит установить тип повторения напоминания. Установка времени, а также типа повторения напоминания представлено на рисунке 14.

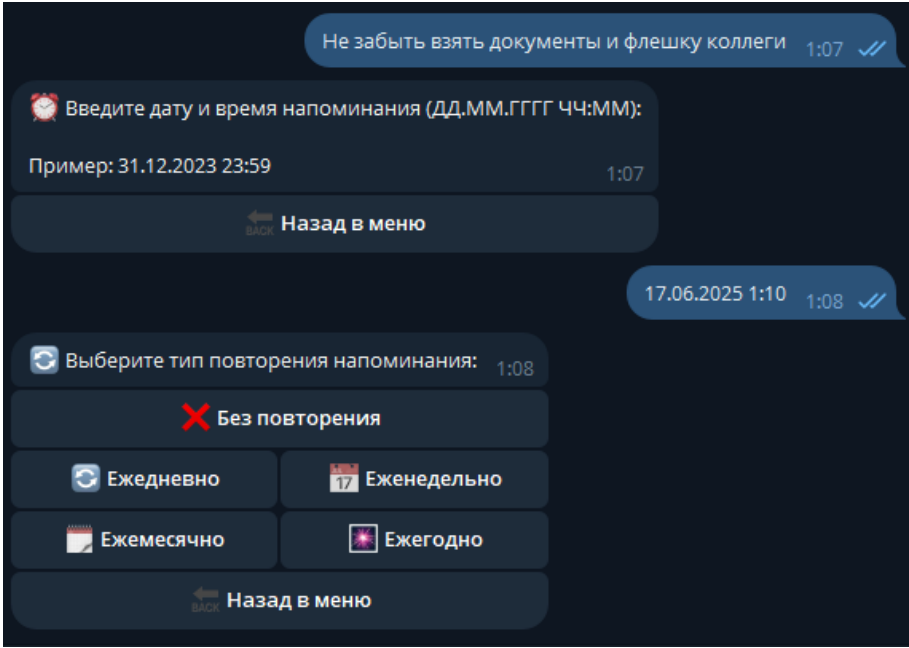


Рисунок 14 – Предпоследний этап создания напоминания – установка времени прихода и повтора напоминания

Затем пользователю предлагается выбрать приоритет напоминания, реализовано данное решение, изображённое на рисунке 15, для наилучшего визуального взаимодействия.

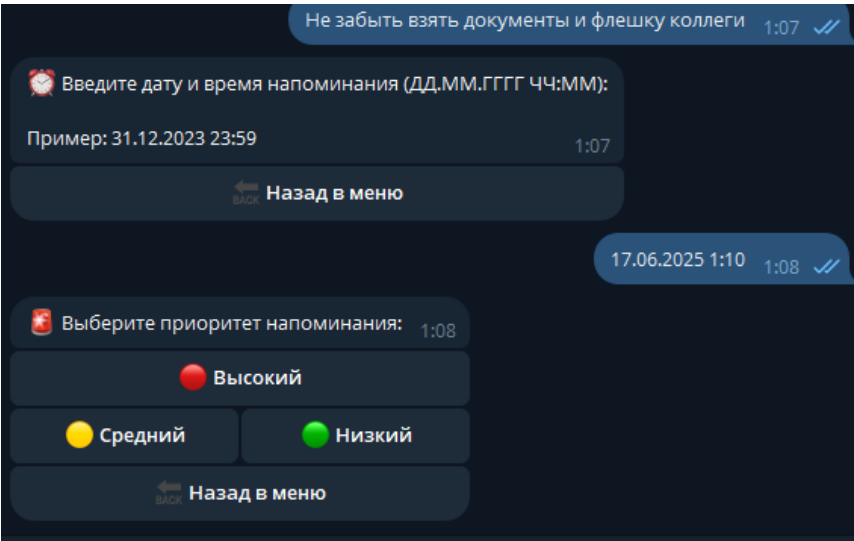


Рисунок 15 – Последний этап создания напоминания – выбор приоритета

В конце всех действий выводится сообщение об успешном добавлении напоминания, представлено это на рисунке 16.

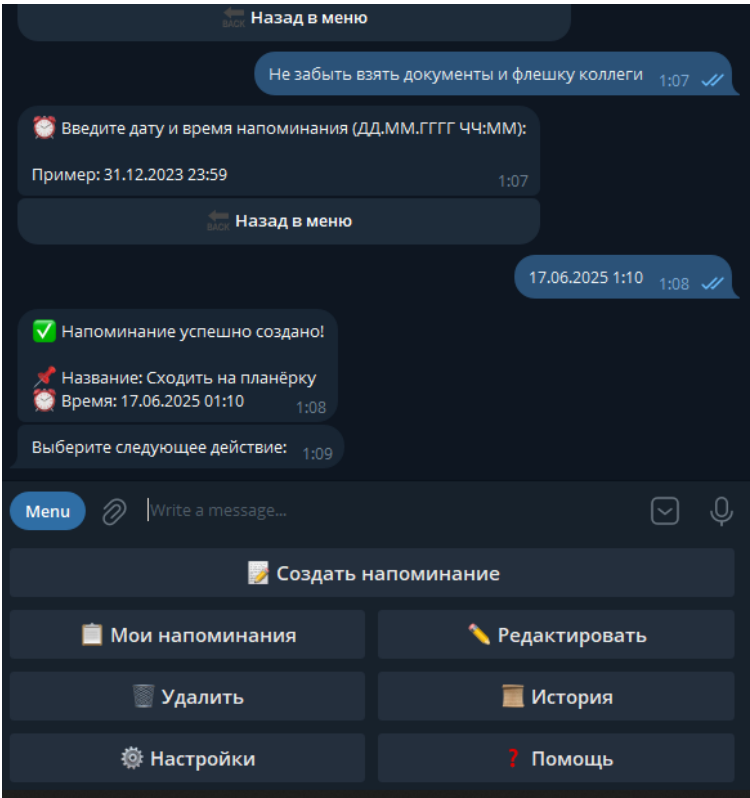


Рисунок 16 – Успешное создание напоминания

На данном моменте пользователю всё так же остаётся доступен весь функционал программы, добавляется только лишь ожидание желанного напоминания в необходимый момент, который и представлен на рисунке 17.

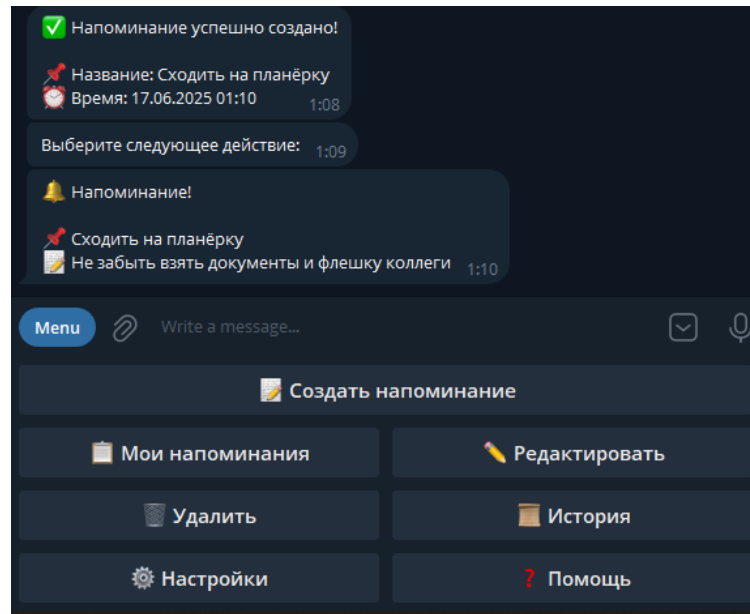


Рисунок 17 – Успешный приход напоминания

Следующей рассматриваемой функцией будет редактирование напоминания. При нажатии на кнопку «Редактировать» бот присылает интерактивный список с кнопками, нажав одну из которых, можно перейти на следующий этап редактирования. Первый этап редактирования изображён на рисунке 18.

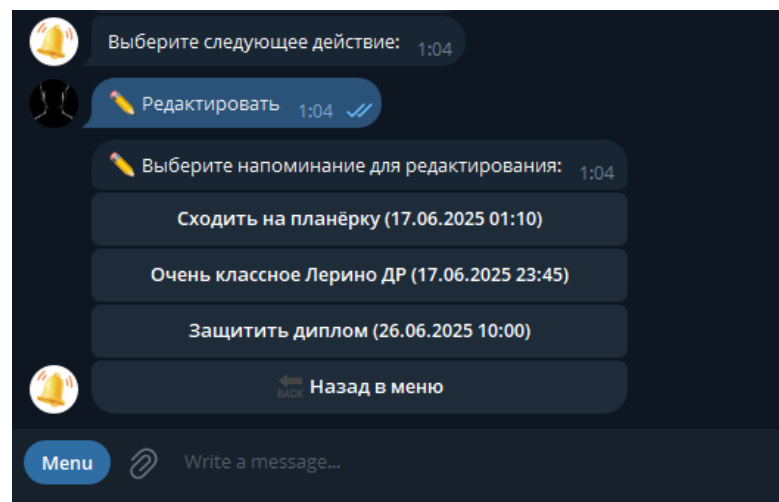


Рисунок 18 – Первый этап редактирования – нажатие на кнопку в меню

После того, как пользователь выбирает и нажимает на желаемое напоминание, бот динамически меняет содержание сообщения. Второй этап редактирования показан на рисунке 19.

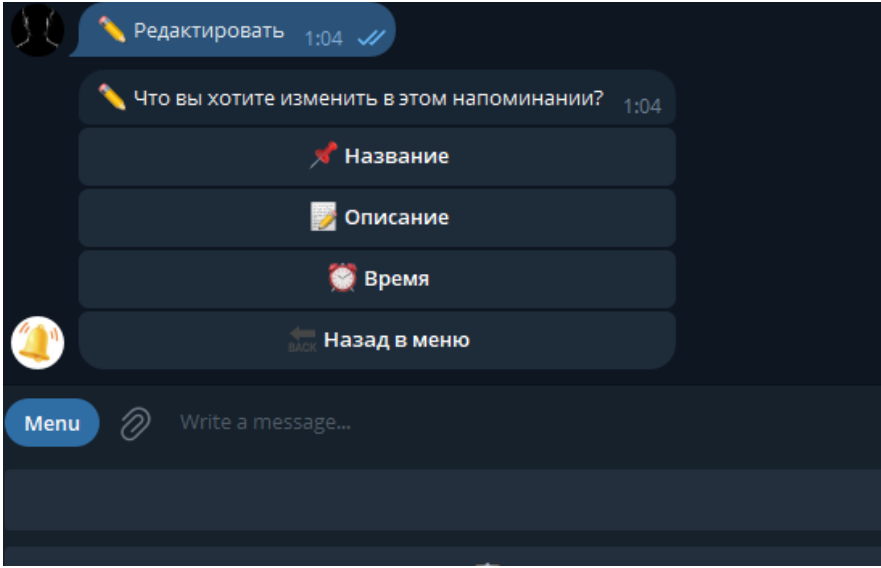


Рисунок 19 – Второй этап редактирования – показ меню с полями, которые можно изменить

После выбора, к примеру описание напоминания, бот запрашивает новое значение для этого поля, в дальнейшем проверяя корректность ввода, особенно для даты и сохраняет изменения в базу данных. Показано это на рисунке 20.

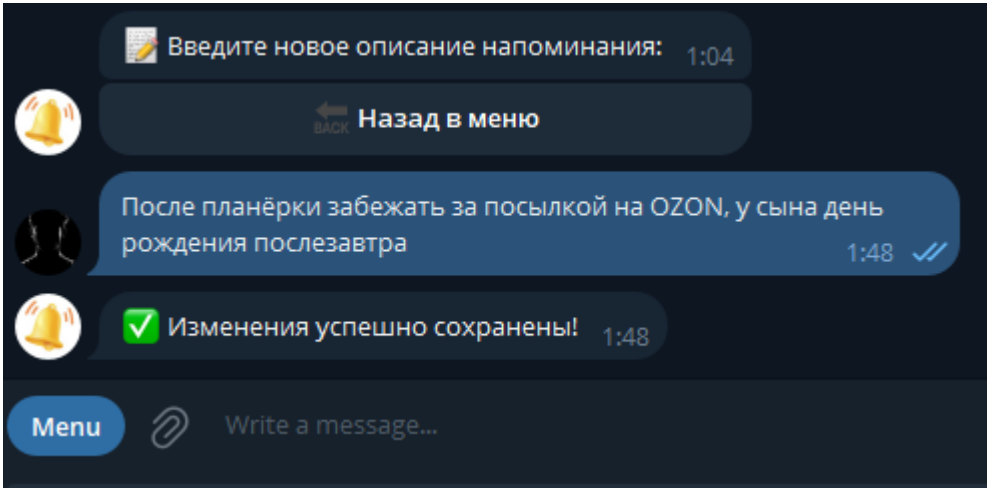


Рисунок 20 – Последний этап редактирования – ввод нового значения и сохранение его в базу данных

Следующей функцией на рассмотрении будет функция удаления, которая, непосредственно, удаляет выбранное пользователем напоминание без возможности восстановления. Пользователь, как и раньше, нажимает на одну из кнопок в меню, а именно «Удалить», бот же моментально присылает список из существующих напоминаний. Представлено это на рисунке 21.

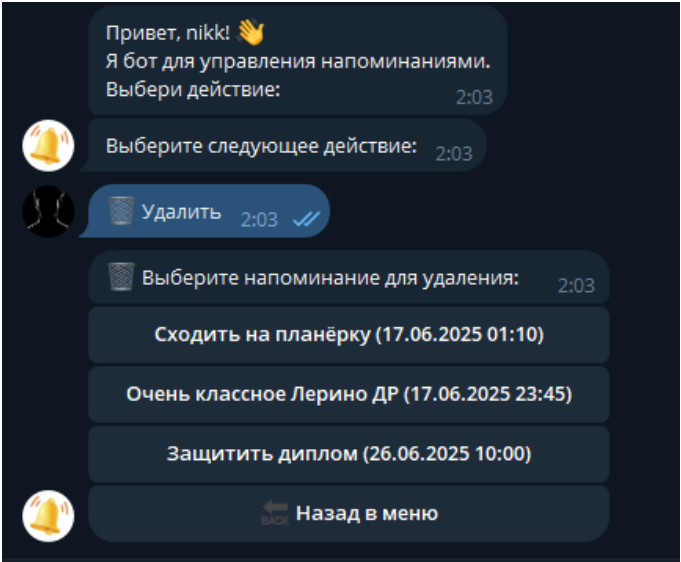


Рисунок 21 – Начало удаления напоминания

Затем, выбрав нужное напоминание, необходимо подтвердить его удаление или наоборот. Процесс подтверждения удаления изображён на рисунке 22.

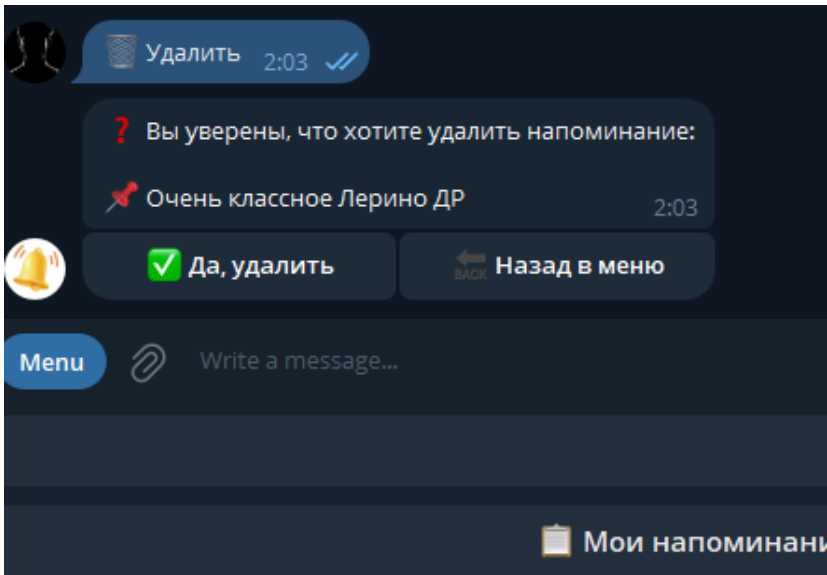


Рисунок 22 – Финальный этап удаления – подтверждение удаления

Последняя основная функция чат-бота – просмотр напоминаний. При нажатии на кнопку бот присылает пользователю список из напоминаний с информацией о них вплоть до времени создания самого напоминания. Показано это на рисунке 23.

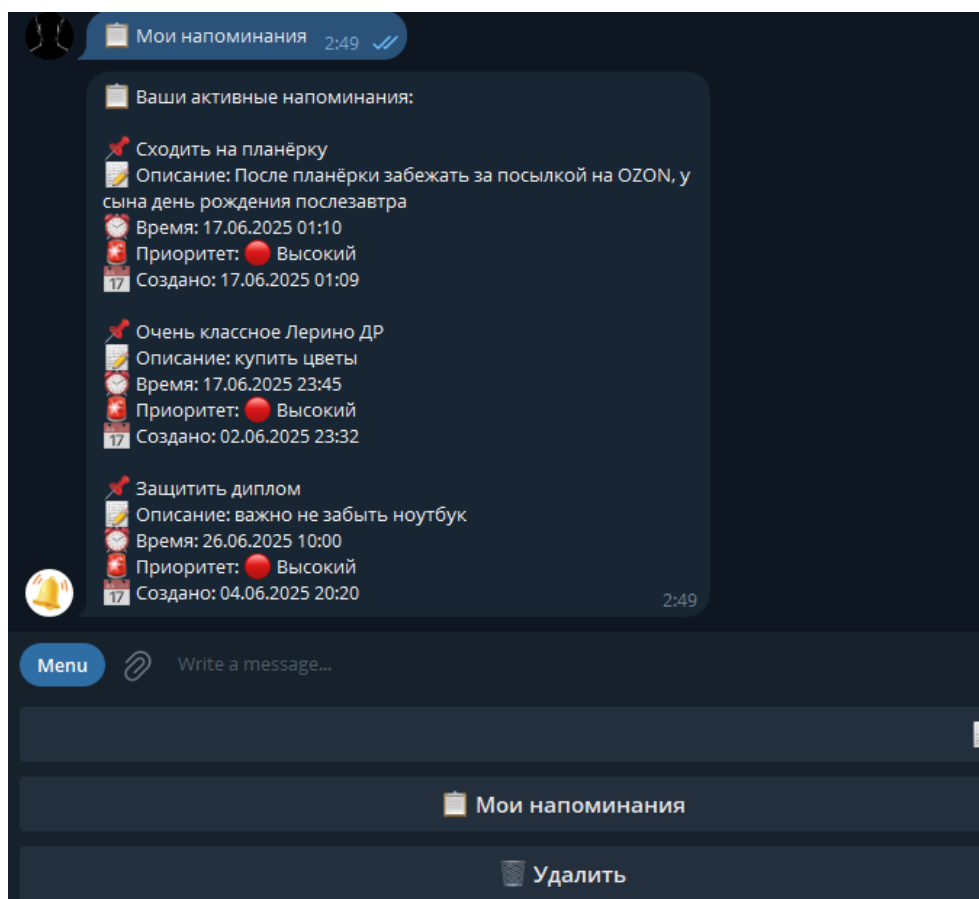


Рисунок 23 – Показ напоминаний

Таким образом, пользовательский интерфейс удовлетворяет функциональным требованиям приложения.

2.4 Проектирование программных средств

2.4.1 Проектирование классов серверной части приложения

На основе полученной ранее логической модели базы данных, была разработана диаграмма классов для серверной части системы напоминаний пользователя. Данная диаграмма отражает структуру, атрибуты, методы и взаимосвязи между классами программы, помогает проектировать систему

перед её реализацией, документировать архитектуру и визуально анализировать разрабатываемое программное обеспечение [13].

На рисунке 24 представлена диаграмма классов серверной части приложения.

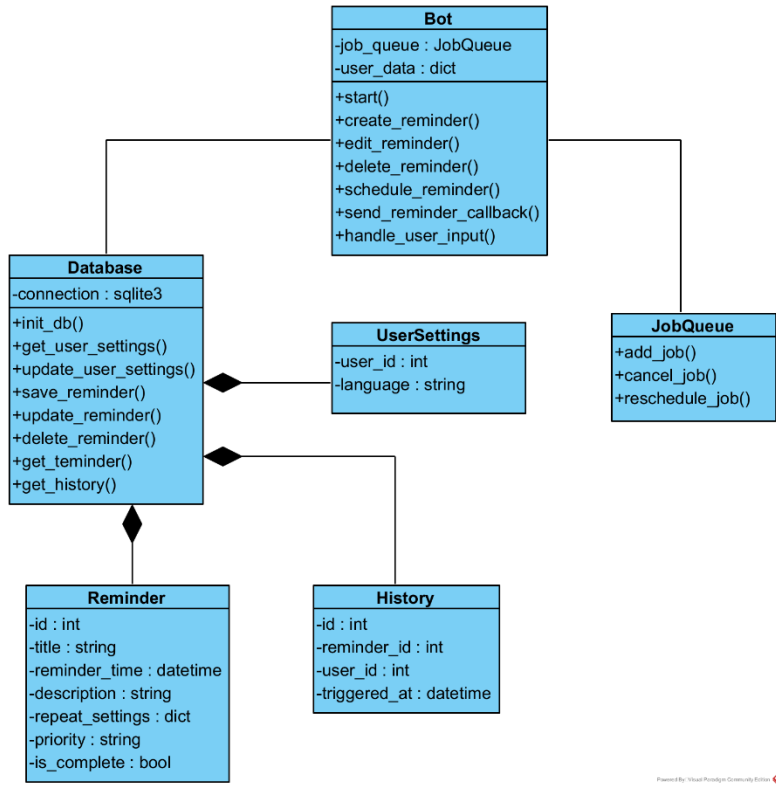


Рисунок 24 – Диаграмма классов серверной части приложения

Краткое описание и функциональное назначение каждого из классов на данной диаграмме приведено в таблице 3.

Таблица 3 – Описание классов

Класс	Описание
Database	Отвечает за взаимодействие с базой данных SQLite, включая инициализацию, управление настройками пользователей, напоминаниями и историей.
Reminder	Представляет напоминание с атрибутами: ID, заголовок, время, описание, настройки повтора, приоритет и статус выполнения.

Продолжение таблицы 3

Класс	Описание
Bot	Основной класс бота, обрабатывающий команды пользователя, управляющий напоминаниями и их планированием через JobQueue[14].
UserSettings	Хранит настройки пользователя, такие как ID и язык интерфейса.
History	Фиксирует историю срабатывания напоминаний, включая ID напоминания, пользователя и время события.
JobQueue	Управляет очередью задач, позволяя добавлять, отменять и перепланировать напоминания.

Класс Bot использует Database для сохранения и загрузки данных, включая Reminder, UserSettings и History. JobQueue применяется Bot для планирования задач, связанных с Reminder, таких как отправка уведомлений. Reminder и UserSettings хранят информацию, которая записывается в Database, а History регистрирует события, связанные с выполнением напоминаний. Вместе эти классы образуют систему управления напоминаниями, где Bot выступает центральным координатором.

Для визуализации взаимодействия между объектами в системе была создана диаграмма коммуникаций, которая отражает структурные связи между частями программы и порядок обмена сообщениями. Данная диаграмма представлена на рисунке 25.

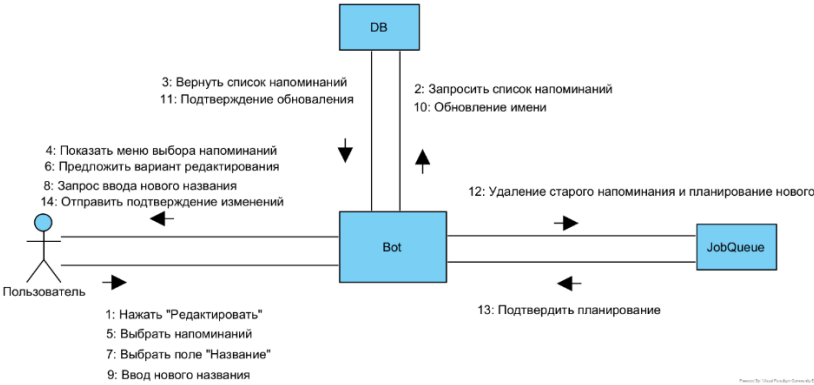


Рисунок 25 – Диаграмма коммуникации

На диаграмме отражён порядок обмена сообщениями между элементами приложения, когда пользователь непосредственно редактирует напоминание.

2.4.2 Проектирование алгоритма срабатывания напоминания

Алгоритм отправки напоминания был выполнен в виде диаграммы последовательности, которая описывает динамику взаимодействия отдельных компонентов системы на едином временном отрезке. Диаграмма последовательности алгоритма отправки напоминания представлена на рисунке 26.

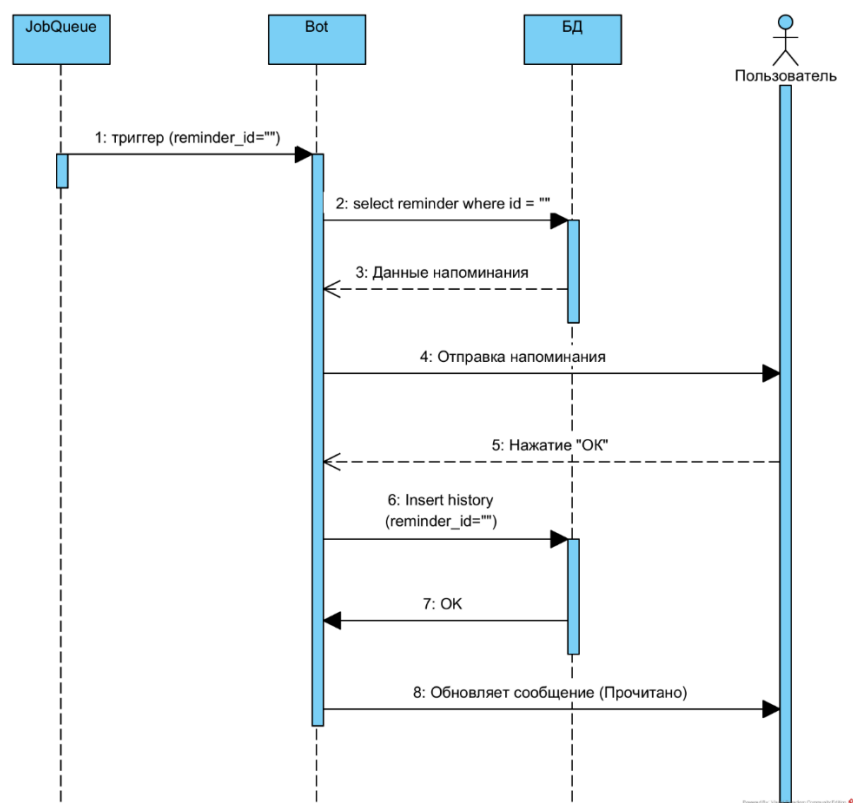


Рисунок 26 – Диаграмма последовательности процесса отправки напоминания

Алгоритм отправки напоминания представлен в виде диаграммы последовательности, отражающей взаимодействие компонентов системы. Процесс начинается, когда JobQueue отправляет триггер с reminder_id. Бот, получив этот сигнал, запрашивает у базы данных соответствующее

напоминание, используя переданный идентификатор. После получения данных бот формирует и отправляет напоминание пользователю.

Пользователь, ознакомившись с сообщением, нажимает кнопку "ОК", что служит подтверждением прочтения. В ответ на это действие бот регистрирует событие в истории, сохраняя reminder_id, и уведомляя пользователя об успешной обработке. Затем исходное сообщение обновляется, помечаясь как «Прочитано». После этого цикл обработки напоминания завершается.

2.4.3 Проектирование алгоритма удаления напоминания

Процесс удаления напоминания из базы данных приведён на рисунке 27.

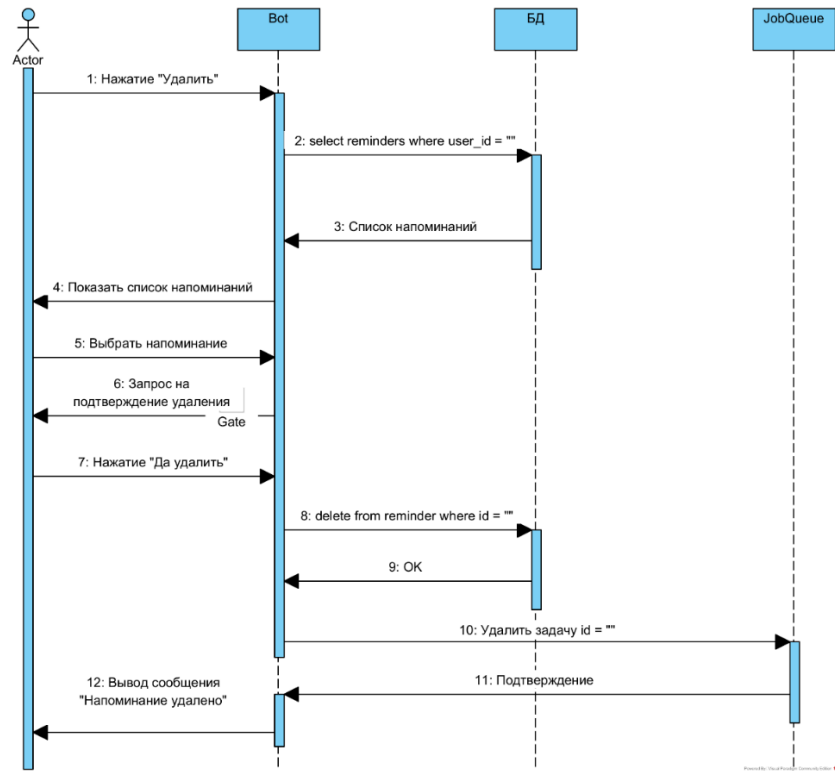


Рисунок 27 – Диаграмма последовательности процесса удаления напоминания

Когда пользователь нажимает кнопку "Удалить" в интерфейсе, система активирует обработчик команды delete_reminder. Этот обработчик первым делом запрашивает из базы данных актуальный список напоминаний,

принадлежащих текущему пользователю. Для этого выполняется SQL-запрос, который фильтрует записи по идентификатору пользователя [15]. Полученные данные форматируются в удобный для восприятия вид с поддержкой пагинации, что особенно важно при большом количестве активных напоминаний [16].

После отображения списка система переходит в режим ожидания выбора пользователя. Когда конкретное напоминание выбрано, бот инициирует этап подтверждения действия, демонстрируя подробную информацию о выбранном напоминании и предлагая варианты «Да, удалить» или «Нет, отменить». Этот шаг критически важен для предотвращения случайного удаления важных данных.

Если пользователь подтверждает свое намерение, система выполняет непосредственно операцию удаления. На уровне базы данных это реализовано в виде атомарной транзакции, включающей не только удаление записи из таблицы reminders, но и очистку связанных заданий из очереди выполнения (JobQueue). Такая комплексная обработка гарантирует, что удаленное напоминание не будет случайно активировано в будущем.

Завершающим этапом процесса становится отправка пользователю подтверждающего сообщения "Напоминание удалено", которое формируется с учётом выбранного языка интерфейса [17].

3 Реализация персональной системы напоминаний пользователя на основе Telegram-бота

3.1 Реализация серверной части приложения

Telegram-бот реализовывался в среде разработки Visual Studio с использованием языка Python версии 3.10 и библиотеки python-telegram-bot версии 22.1 для взаимодействия с Telegram API 9.0. Для хранения данных применялась база данных SQLite, доступ к которой осуществлялся через стандартный модуль sqlite3, входящий в состав Python Standard Library [18].

В серверной части существуют контроллеры, получающие HTTP-запросы от клиента и выполняющие сохранение, изменение или получение данных из базы данных. API запросы и их спецификации представлены в таблице 4.

Таблица 4 – Спецификация API запросов

№	Назначение сервиса	HTTP метод	URI сервиса	Класс контроллера
1	Запуск бота	GET	/api/bot/start	CommandHandler('start', start)
2	Получение справки	GET	/api/bot/help	CommandHandler('help', help_command)
3	Пропуск описания напоминания	GET	/api/reminders/skip-description	CommandHandler('skip', skip_description)
4	Просмотр списка напоминаний	GET	/api/reminders	MessageHandler(filters.Regex)
6	Создание нового напоминания	POST	/api/reminders	ConversationHandler (create_reminder)
7	Редактирование существующего напоминания	GET	/api/reminders/{reminder_id}	ConversationHandler (edit_reminder)
8	Удаление существующего напоминания	DELETE	/api/reminders/{reminder_id}	ConversationHandler (delete_reminder)
9	Получение настроек	GET	/api/user/settings>	ConversationHandler (settings_menu)

Класс контроллера – это компонент в архитектуре приложения, который отвечает за обработку входящих запросов и возврат ответов. В контексте Telegram-бота класс контроллера – это как таковое название обработчиков,

которые управляют логикой команд и сообщений. В коде эту роль выполняют обработчики `CommandHandler`, `MessageHandler`, `CallbackQueryHandler`, `ConversationHandler`.

Для отображения физического расположения компонентов серверной части приложения была создана диаграмма развёртывания, представленная на рисунке 28.

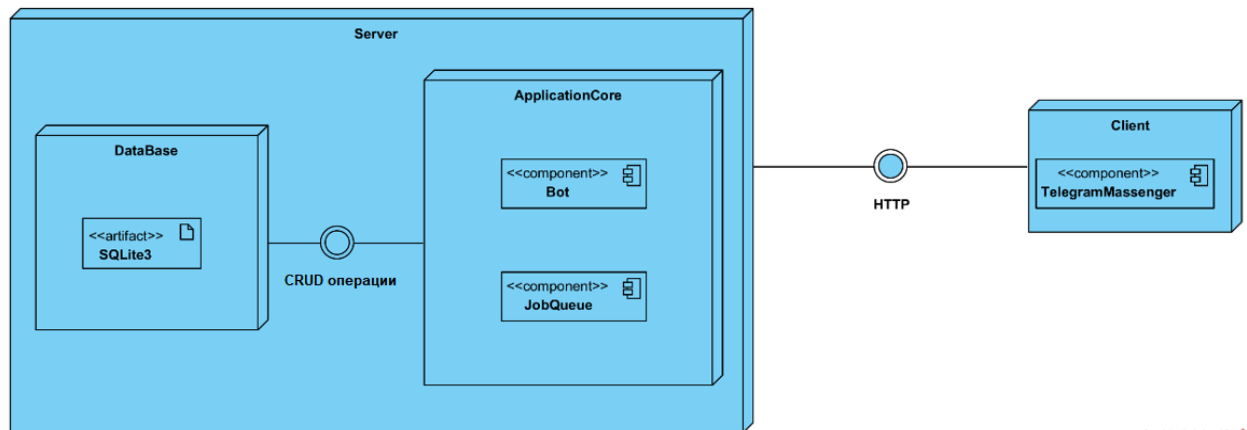


Рисунок 28 – Диаграмма развёртывания

Система представляет собой Telegram-бота для управления напоминаниями. В основе лежит серверная часть, использующая SQLite3 в качестве базы данных. Взаимодействие с БД осуществляется через SQLite API, обеспечивающий надежное выполнение запросов [19].

Ядро приложения (`ApplicationCore`) включает два основных компонента: модуль `Bot`, отвечающий за бизнес-логику и обработку команд пользователей, и `JobQueue` - систему планирования и выполнения напоминаний с поддержкой различных интервалов повторения. Для взаимодействия с данными пользователей используется HTTP-протокол [20].

Система поддерживает многоязычный интерфейс (русский/английский), гибкие настройки повторения, пагинацию списка напоминаний и ведение истории выполненных задач. Архитектура обеспечивает масштабируемость и надёжность за счет транзакционной работы с БД и удобство использования через Telegram-интерфейс.

Ключевыми особенностями являются: автоматическое восстановление напоминаний при перезапуске сервиса, обработку временных зон пользователей, подтверждение важных действий. Система предназначена для работы на любых устройствах, поддерживающих Telegram, что доказывает её доступность и кроссплатформенность.

Для наилучшего отображения структуры разработанного приложения и взаимосвязей между его отдельными элементами была построена диаграмма компонентов, отображённая на рисунке 29.

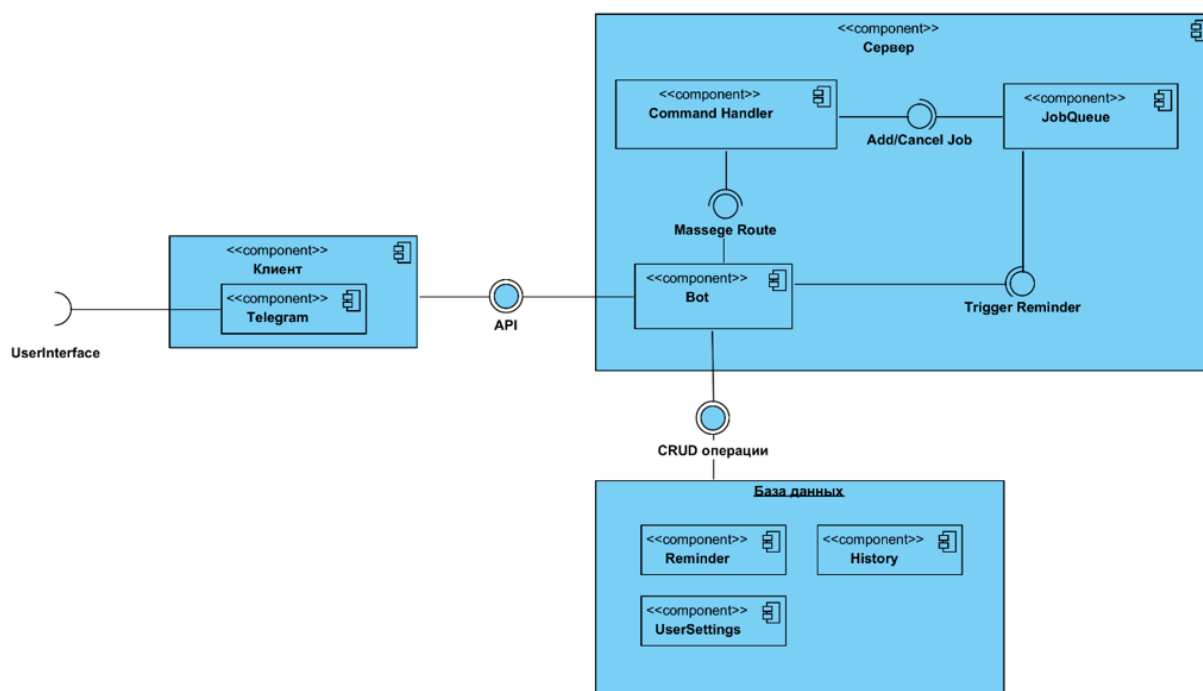


Рисунок 29 – Диаграмма компонентов

Компоненты взаимодействуют через определенные интерфейсы. Например, Telegram передает сообщения в Command Handler, который обрабатывает команды и делегирует задачи в JobQueue или Reminder. JobQueue управляет задачами и использует Trigger Reminder для активации напоминаний, которые затем доставляются пользователю. Все данные сохраняются в базу данных.

3.2 Реализация базы данных

В проекте реализовано хранение данных с использованием реляционной системы управления базами данных SQLite. Данная СУБД была выбрана по двум причинам, а именно легковесности и автономности работы – SQLite не требует отдельного серверного процесса и хранит всю базу в едином файле, что идеально подходит для нашего случая. Несмотря на свою компактность, SQLite поддерживает большинство стандартных SQL-функций, включая сложные запросы и транзакции, что позволяет реализовать полноценную бизнес-логику. Важным преимуществом является встроенная ACID-совместимость, гарантирующая целостность данных при всяческих сбоях, а также кроссплатформенность, которая позволяет развёртывать решение на любых операционных системах без дополнительных модификаций.

При реализации особое внимание уделялось эффективному хранению напоминаний с их настройками, поддержке повторяющихся событий, ведению истории срабатываний для анализа и персонализации через пользовательские настройки. Центральной сущностью системы стала таблица `reminders`, содержащая все ключевые данные о напоминаниях: уникальный идентификатор, ссылку на пользователя, название, описание, время срабатывания в ISO-формате, который предполагает собой хранения в формате (2025 - 06 - 10T23:59:00), настройки повторения в JSON-формате, приоритет, а также флаг и время выполнения. Для хранения истории срабатываний используется таблица `history` с записями о каждой активации напоминания, связанными через внешние ключи с основной таблицей. Персонализация реализована через таблицу `user_settings`, хранящую предпочтения пользователей, такие как язык интерфейса.

Оптимизация производительности достигнута за счет продуманной индексации ключевых полей, использования пакетных операций в транзакциях и кэширования часто используемых данных. Все критические операции выполняются с учётом безопасности данных. Особое внимание

уделено работе с временными зонами – все даты хранятся в UTC с автоматической конвертацией в локальное время пользователя при отображении.

На основе прошлых моделей, то есть концептуальной и логической, была разработана физическая модель базы данных, которая представлена на рисунке 30.

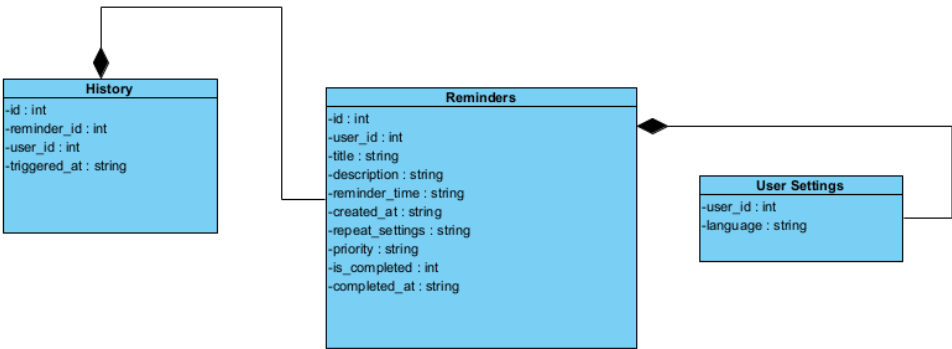


Рисунок 30 – Физическая модель данных

В таблице 5 представлена спецификация сущности «История напоминаний».

Таблица 5 – Описание сущности «История напоминаний»

Наименование	Описание	Тип данных	Возможность значения NULL	ОДЗ	Значение по умолчанию	Роль
id	Уникальный идентификатор записи истории	INTEGER	Нет	> 0		PK
Наименование	Описание	Тип данных	Возможность значения NULL	ОДЗ	Значение по умолчанию	Роль
reminder_id	ID связанного напоминания	INTEGER	Нет	существует в reminders.id		FK
user_id	ID пользователя телеграм Telegram	INTEGER	Нет	> 0		FK
triggered_at	Дата и время срабатывания)	TEXT	Нет	дата ≥ времени напоминания		

В таблице 6 представлена спецификация сущности «Напоминания».

Таблица 6 – Описание сущности «Напоминания»

Наименование	Описание	Тип данных	Возможность значения NULL	ОДЗ	Значение по умолчанию	Роль
id	Уникальный идентификатор напоминания	INTEGER	Нет	> 0		PK
user_id	ID пользователя Telegram, создавшего напоминание	INTEGER	Нет	> 0		
title	Заголовок/название напоминания	TEXT	Нет	не пустая строка ≤ 255 символов		FK
description	Описание напоминания (может быть пустым)	TEXT	Да	≤ 1000		
reminder_time	Время срабатывания напоминания (в формате ISO)	TEXT	Нет	дата в будущем в формате ДД.ММ.ГГГГ ЧЧ:ММ		
created_at	Дата и время срабатывания напоминания (в формате ISO)	TEXT	Нет	дата создания $\leq$ текущей даты		
repeat_settings	Настройки повторения в формате JSON	TEXT	Да	одно из: null, daily, weekly, monthly, yearly		
priority	Приоритет напоминания	TEXT	Да	Высокий, Средний или Низкий		
is_completed	Флаг завершения (0 – активно, 1 – завершено)	INTEGER	Нет	0 (активно) или 1 (завершено)	0	
completed_at	Дата и время завершения напоминания (в формате ISO)	TEXT	Да	дата $\geq$ created_at		

В таблице 7 представлена спецификация сущности «Настройки пользователя».

Таблица 7 – Описание сущности «Настройки пользователя»

Наименование	Описание	Тип данных	Возможность значения NULL	ОДЗ	Значение по умолчанию	Роль
user_id	ID пользователя Telegram	INTEGER	Нет	> 0		PK
language	Язык интерфейса	TEXT	Нет	ru или en	ru	

Созданная база данных полностью соответствует разработанной физической модели.

4 Контроль качества программного обеспечения

4.1 Выбор режима тестирования

Тестирование программного обеспечения – стандартный, и одновременно один из важнейших этапов разработки, особенно когда речь идёт о Telegram-ботах, взаимодействующих с API. Были рассмотрены ключевые виды тестирования, их особенности и применение в контексте ботов.

Функциональное тестирование фокусируется на проверке соответствия программы бизнес-требованиям. Оно оценивает итоговый результат действий, не затрагивая промежуточные состояния системы. В контексте телеграм-бота это означает проверку корректности создания, редактирования и удаления напоминаний, а также локализацией интерфейса.

Модульное тестирование проводится на уровне исходного кода. Основная цель – проверка отдельных методов, функций или модулей приложения. Такие тесты зачастую легко автоматизируются и выполняются быстро, что делает их весьма удобными для использования в системах непрерывной интеграции. Этот подход особенно полезен для тестирования таких элементов, как парсинг дат, работа с базой данных SQLite, обработка повторяющихся событий и система многоязыковой поддержки. Модульные тесты не требуют сложной инфраструктуры, они легко автоматизируются и быстро выполняются в рамках CI/CD.

Интеграционное тестирование оценивает корректность взаимодействия между различными компонентами системы, например, работу с базой данных или согласованность микросервисов. Этот вид тестирования сложнее в реализации, так как требует настройки взаимодействия между компонентами.

После анализа всех подходов было принято решение сосредоточиться на функциональном тестировании, так как основное требование к боту – корректное выполнение заявленных функций. Ниже приведено ещё несколько

аргументов в пользу функционального тестирования разрабатываемого решения:

- бот ориентирован на UX в мессенджере;
- требуется проверка полного user flow, то есть последовательности шагов, которые предпринимает пользователь для достижения цели в цифровом продукте;
- важна корректность работы с Telegram-специфичными элементами (Reply-клавиатуры, инлайн-кнопки, команды).

Тестирование охватит создание и управление напоминаниями, обработку пользовательского ввода (например, валидацию дат и времени), а также работу локализации (русский и английский интерфейсы). Будут проверены основные сценарии работы с командами бота:

- команда /start с инициализацией ReplyKeyboardMarkup;
- полный цикл создания напоминания через многоуровневый ConversationHandler;
- работа с инлайн-кнопками (InlineKeyboardMarkup), к примеру для выбора языка, приоритета и периодичности.

4.2 Тестирование серверной части приложения

Для основных вариантов использования было выполнено функциональное тестирование. Результаты отображены в таблице 8.

Таблица 8 – Тесты серверной части приложения

Тест	Назначение теста	Описание	Актуальный результат
1	Тест предназначен для тестирования серверной части варианта использования «Создание напоминание»	Ввод данных от пользователя в систему:	Создание успешно совершено (рисунок 31)
2	Тест предназначен для тестирования серверной части варианта использования «Редактирование напоминания»	Данные, подвергнутые изменению	Редактирование успешно (рисунок 32)

Продолжение таблицы 8

Тест	Назначение теста	Описание	Актуальный результат
3	Тест предназначен для тестирования серверной части варианта использования «Удаление напоминания»	Удаление напоминания	Удаление успешно завершено (рисунок 33)
4	Тест предназначен для тестирования серверной части «Приход напоминания»	Приход напоминания в нужное время	Напоминание успешно отправлено (рисунок 34)

Скриншоты логов серверной части системы приведены ниже.

```
2025-06-22 06:31:08,389 - __main__ - INFO - User 387985755 set priority: high
2025-06-22 06:31:08,396 - __main__ - INFO - Reminder created: id=32, user_id=387985755, title='тест 123', time=2025-06-22T06:33:00+03:00, priority=Высокий, repeat=null
2025-06-22 06:31:08,398 - apscheduler.scheduler - INFO - Added job "32" to job store "default"
2025-06-22 06:31:08,477 - httpx - INFO - HTTP Request: POST https://api.telegram.org/bot7965324005:AAGMd1Bt2bLJ66vgjtywp0E91f5gN46EaAs/editMessageText "HTTP/1.1 200 OK"
```

Рисунок 31 – Log запроса, отображающий успешное создание напоминания

```
2025-06-22 06:43:46,466 - httpx - INFO - HTTP Request: POST https://api.telegram.org/bot7965324005:AAGMd1Bt2bLJ66vgjtywp0E91f5gN46EaAs/getUpdates "HTTP/1.1 200 OK"
2025-06-22 06:43:46,484 - __main__ - INFO - User 387985755 edited reminder 32: field=title, old_value='тест 123', new_value='тест для теста'
```

Рисунок 32 – Log запроса, отображающий успешное редактирование напоминания

```
2025-06-22 06:46:21,300 - httpx - INFO - HTTP Request: POST https://api.telegram.org/bot7965324005:AAGMd1Bt2bLJ66vgjtywp0E91f5gN46EaAs/answerCallbackQuery "HTTP/1.1 200 OK"
2025-06-22 06:46:21,318 - __main__ - INFO - User 387985755 deleted reminder: id=32, title='тест для теста', time=2025-06-22T06:33:00+03:00
```

Рисунок 33 – Log запроса, отображающий успешное удаление напоминания

```
2025-06-22 06:33:00,216 - httpx - INFO - HTTP Request: POST https://api.telegram.org/bot7965324005:AAGMd1Bt2bLJ66vgjtywp0E91f5gN46EaAs/sendMessage "HTTP/1.1 200 OK"
2025-06-22 06:33:00,233 - __main__ - INFO - Reminder sent: user_id=387985755, reminder_id=32
```

Рисунок 34 – Log запроса, отображающий успешное создание напоминания

В процессе тестирования серверной части приложения не было выявлено ошибок при обработке API-запросов к серверной части приложения. Все методы успешно взаимодействуют с базой данных и возвращают ожидаемые результаты.

4.3 Тестирование клиентской части приложения

В процессе тестирования клиентской части разработанного приложения был выполнен запуск на различных устройствах для проверки корректного функционирования системы. Запуск чат-бота и проверка корректности работы пользовательского интерфейса были произведены на устройствах с различной версией мессенджера Telegram, а также с разнообразными операционными системами: Windows, IOS, Android. План тестирования пользовательского интерфейса приведён в таблице 9.

Таблица 9 – План тестирования

№	Входные данные	Ожидаемый результат	Результат
1	Запуск чат-бота на устройстве с операционной системой IOS 17.0.3 и версией Telegram 11.12.1	Все элементы интерфейса чат-бота отображены корректно	Тест пройден (рисунок В.1)
2	Запуск чат-бота на устройстве с операционной системой IOS 18.0.1 и версией Telegram 11.10.2	Все элементы интерфейса чат-бота отображены корректно	Тест пройден (рисунок В.2)
3	Запуск чат-бота на устройстве с операционной системой Windows 10 и версией Telegram 5.15.4 x64	Все элементы интерфейса чат-бота отображены корректно	Тест пройден (рисунок В.3)
4	Запуск чат-бота на смартфоне с альбомной ориентацией экрана	Все элементы интерфейса чат-бота отображены корректно	Тест пройден (рисунок В.4)

Тестирование клиентской части чат-бота показало корректную работу пользовательского интерфейса на устройствах с разной операционной системой и различной версией мессенджера Telegram.

5 Контроль качества

5.1 Метрики Холстеда

Метрики Холстеда – это набор метрик, предназначенных для оценки сложности программного кода на основе анализа операторов и операндов программы. Основные метрики Холстеда включают:

- число уникальных операторов;
- число уникальных операндов;
- общее количество операторов;
- общее количество операндов.

Для проведения оценки метрик Холстеда был выбран метод `get_date` класса `CommandHandler`, код которого представлен в листинге 1.

Листинг 1 – Код метода `get_date`

```
async def get_date(update: Update, context: CallbackContext) ->
int:
    user = update.effective_user
    settings = get_user_settings(user.id)

    if not update.message or not update.message.text:
        await update.message.reply_text(
            get_text("invalid_date_format",
settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
        return DATE

    if update.callback_query and update.callback_query.data ==
"back":
        await back_callback(update, context)
        return ConversationHandler.END

    try:
        date_text = update.message.text
        if not ('.' in date_text and ' ' in date_text and ':'
in date_text or '.' in date_text):
            raise ValueError("Неверный формат даты")

        date_part, time_part = date_text.split()
        time_part = time_part.replace('.', ':')

        # Получаем часовой пояс пользователя автоматически
```

```

        user_timezone = pytz.timezone('Etc/GMT' + str(-
int(datetime.now().astimezone().utcoffset().total_seconds() /
3600)))
        reminder_time = datetime.strptime(f"{date_part}
{time_part}", "%d.%m.%Y %H:%M")
        reminder_time = user_timezone.localize(reminder_time)

        if reminder_time <= datetime.now(user_timezone):
            await update.message.reply_text(
                get_text("date_in_past", settings["language"]),

reply_markup=get_back_keyboard(settings["language"])
            )
            return DATE

        context.user_data['reminder_time'] = reminder_time
        await update.message.reply_text(
            get_text("choose_repeat", settings["language"]),

reply_markup=get_repeat_keyboard(settings["language"])
        )
        return REPEAT

    except ValueError as e:
        logger.error(f"Date parsing error: {e}")
        await update.message.reply_text(
            get_text("invalid_date_format",
settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
        return DATE

```

Операторы метода `get_date` приведены в таблице 10.

Таблица 10 – Операторы метода `get_date`

Операторы	Число вхождений
<code>async</code>	1
<code>def</code>	1
<code>if</code>	3
<code>not</code>	2
<code>await</code>	6
<code>try</code>	1
<code>=</code>	10
<code>raise</code>	1
<code>.</code>	30
<code>(</code>	15
<code>)</code>	15
<code>:</code>	1
<code>,</code>	10
<code>+</code>	1
<code><</code>	1

Продолжение таблицы 10

Операторы	Число вхождений
<=	1
return	3
->	1
==	1
in	2
split	1
replace	1
str	1
int	1
n2 = 21	N2 = 48

Операнды метода get_date приведены в таблице 11.

Таблица 11 – Операнды метода get_date

Операнды	Число вхождений
get_date	1
update	8
context	6
CallbackContext	1
user	2
settings	5
get_user_settings	1
id	1
message	3
text	2
invalid_date_format	1
reply_markup	2
get_back_keyboard	2
language	5
callback_query	2
data	2
back	2
back_callback	2
ConversationHandler	2
END	2
date_text	3
ValueError	2
n2 = 22	N2 = 50

Для расчёта длины программы применяется формула:

$$N=N1+N2=48+50=98.$$

Расчет словаря программы остается прежним:

$$n=n1+n2=23+32=55.$$

Для расчета объема программы применяется формула:

$$V=N \times \log_2(n)=98 \times \log_2(55) \approx 98 \times 5.78=566.44.$$

Потенциальный объем программы рассчитывается по формуле:

$$V^*=(2+n_2) \times \log_2(2+n_2)=34 \times \log_2(34) \approx 34 \times 5.09=173.06.$$

Теоретический уровень программы вычисляется по формуле:

$$L=V^* / V=173.06 / 566.44 \approx 0.31.$$

Для расчета сложности программы применяется формула:

$$HD=n_1 / 2 \times N_2 / n_2=23 / 2 \times 50 / 32=11.5 \times 1.56=17.94.$$

Таким образом, с новыми значениями N_1 и N_2 , теоретический уровень программы составляет примерно 0.31, а объем программы указывает на средний уровень сложности.

5.2 Метрики SLOC

Метрика SLOC (Source Lines of Code) измеряет объем программного кода, учитывая количество строк в исходном файле. Она применяется для оценки размера программы, анализа ее сложности, а также для прогнозирования трудозатрат на разработку и сопровождение кода. В рамках анализа функции `get_date` были получены следующие данные:

- физические строки: 40;
- логические строки: 28;
- строки комментариев: 4;
- пустые строки: 8.

Физические строки включают весь текст файла, тогда как логические строки показывают количество строк, содержащих операторы и операнды. Исходя из подсчитанных строк, значение метрики SLOC составляет 28, что является приемлемым значением для данного типа функций.

Анализ кода с использованием метрик Холстеда и SLOC предоставил информацию о сложности и объеме разработанного приложения. Метрики Холстеда показали средний уровень сложности программного кода, а анализ SLOC также выявил умеренное число логических строк. Структура кода

вполне прозрачна и читаема, что упрощает дальнейшее развитие и поддержку без чрезмерных затрат ресурсов.

6 Планирование разработки и оценка бюджета

Для планирования разработки проекта и оценки его бюджета была задействована программа Microsoft Project. Это мощный инструмент, предоставляющий большой функционал для эффективного управления проектами, включая: детальное планирование задач, оптимальное распределение ресурсов, контроль соблюдения сроков, а также мониторинг загрузки сотрудников на всех этапах реализации проекта. Работа по разработке проекта была начата 9 апреля 2025 года.

В процессе планирования было выделено 24 задачи, список которых представлен на рисунке 35.

Имя задачи	Название задачи	Длительность	Начало	Окончание	Предшественники	Названия ресурсов
	Начало работы над проектом	0 дней	Ср 09.04.25	Ср 09.04.25		
1	Анализ требований	7 дней	Ср 09.04.25	Чт 17.04.25		Python-разработчик
	Анализ предметной области	2 дней	Ср 09.04.25	Чт 10.04.25		Аналитик
	Анализ аналогичных программных решений	2 дней	Пт 11.04.25	Пн 14.04.25	3	Аналитик
	Постановка задачи на разработку	2 дней	Вт 15.04.25	Ср 16.04.25	4	Аналитик
	Анализ требований к разрабатываемой программе	2 дней	Чт 17.04.25	Пт 18.04.25	5	Python-разработчик Аналитик
	Анализ требований и постановка задачи завершён	0 дней	Пн 21.04.25	Пн 21.04.25	6	
2	Проектирование	13 дней	Сб 19.04.25	Вт 06.05.25		
	Проектирование архитектуры	2 дней	Пн 21.04.25	Вт 22.04.25	7	Python-разработчик
	Проектирование базы данных	2 дней	Ср 23.04.25	Чт 24.04.25	9	
	Проектирование пользовательского интерфейса	7 дней	Пт 25.04.25	Пн 05.05.25	10	
	Проектированных программных средств	2 дней	Вт 06.05.25	Ср 07.05.25	11	Python-разработчик
	Проектирование завершено	0 дней	Чт 08.05.25	Чт 08.05.25	12	
3	Реализация приложения	31 дней	Чт 08.05.25	Чт 19.06.25		
	Реализация серверной и клиентской частей	19 дней	Чт 08.05.25	Вт 03.06.25	13	Python-разработчик
	Реализация базы данных	10 дней	Ср 04.06.25	Вт 17.06.25	15	Python-разработчик
	Реализация приложения завершена	0 дней	Ср 18.06.25	Ср 18.06.25	16	
4	Тестирование	8 дней	Вт 10.06.25	Чт 19.06.25	17	
	Выбор метода тестирования	1 день	Вт 10.06.25	Вт 10.06.25		
	Тестирование серверной части приложения	4 дней	Вт 10.06.25	Пт 13.06.25		
	Тестирование клиентской части приложения	2 дней	Пт 13.06.25	Пн 16.06.25		
	Анализ метрик кода программы	3 дней	Пн 16.06.25	Ср 18.06.25		Аналитик
	Исправление ошибок	0 дней	Ср 18.06.25	Ср 18.06.25		Python-разработчик
	Конец проекта	0 дней	Ср 18.06.25	Ср 18.06.25	23	

Рисунок 35 – Список задач

Для работы над проектом был определён список ресурсов, состоящий из двух сотрудников. На рисунке 36 представлен список используемых в проекте ресурсов.

Название ресурса	Тип	Единицы измерения материала	Краткое название	Группа	Макс. единиц	Стандартная ставка	Ставка сверхурочн	Затраты на исполн.	Начисление	Базовый календарь	Код
Python-разработчик	Трудовой		P		100%	500,00 Р/ч	800,00 Р/ч	0,00 Р	Пропорционал	Стандартный	
Аналитик	Трудовой		A		100%	450,00 Р/ч	700,00 Р/ч	0,00 Р	Пропорционал	Стандартный	

Рисунок 36 – Список ресурсов

Полученная в результате планирования проекта диаграмма Ганта изображена на рисунке 37.

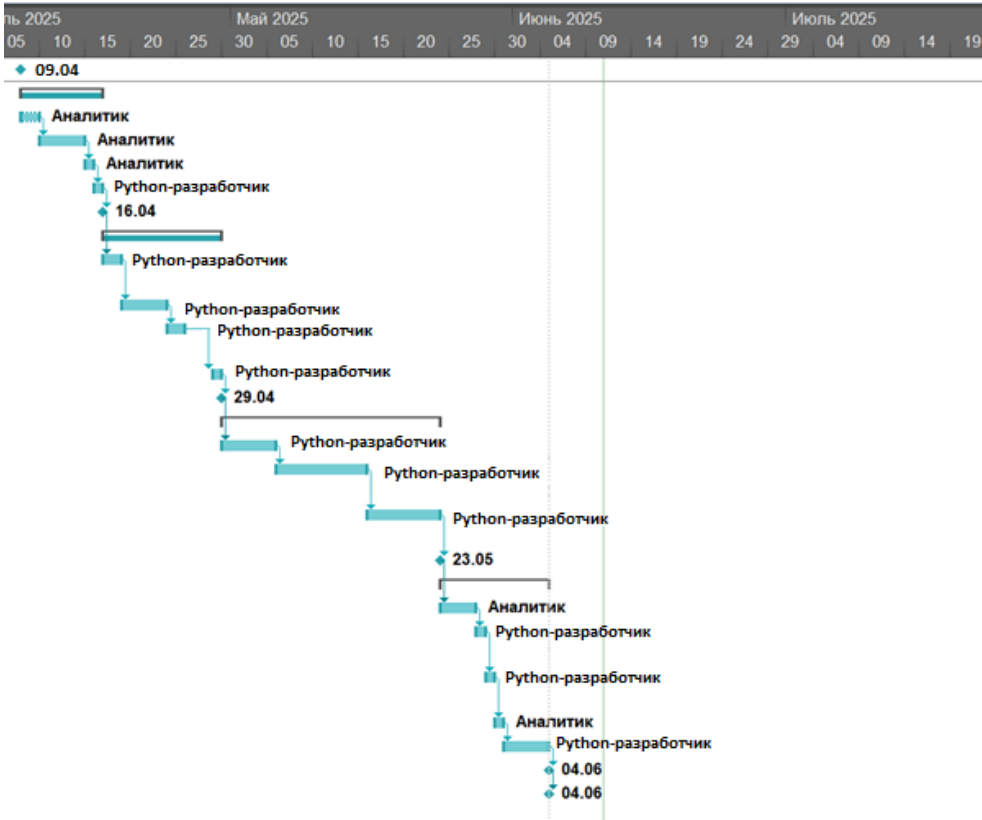


Рисунок 37 – Диаграмма Ганта

Также были подсчитаны трудозатраты человеко-часов, отражающие количество времени, затраченное каждым сотрудником на выполнение проекта. На рисунке 38 представлены трудозатраты каждого из сотрудников.

СОСТОЯНИЕ РЕСУРСОВ

Оставшиеся трудозатраты для всех трудовых ресурсов.

Название	Начало	Окончание	Оставшиеся трудозатраты
Python-разработчик	Ср 09.04.25	Ср 18.06.25	296 ч
Аналитик	Пт 11.04.25	Ср 18.06.25	72 ч

Рисунок 38 – Трудозатраты каждого из сотрудников

Затраты на разработку проекта представлены на рисунке 39.

СВЕДЕНИЯ О ЗАТРАТАХ

Сведения о затратах для всех задач верхнего уровня.

Название	Фиксирован ные затраты	Фактические затраты	Оставшиеся затраты	Затраты	Базовые затраты	Отклонение по стоимости
Начало работы над проектом	0,00р.	0,00р.	0,00р.	0,00р.	0,00р.	0,00р.
Анализ требований	0,00р.	0,00р.	34 000,00р.	34 000,00р.	0,00р.	34 000,00р.
Проектирование приложения	0,00р.	0,00р.	17 120,00р.	17 120,00р.	0,00р.	17 120,00р.
Реализация приложения	0,00р.	0,00р.	36 960,00р.	36 960,00р.	0,00р.	36 960,00р.
Тестирование приложения	0,00р.	0,00р.	26 640,00р.	26 640,00р.	0,00р.	26 640,00р.

Рисунок 39 – Трудозатраты на разработку проекта

Из предоставленных выше данных было выявлено, что затраты на разработку проекта составят 124720, трудозатраты же в размере 368 человеко-часов. Срок выполнения проекта составит 64 дня.

Заключение

В ходе выполнения выпускной квалификационной работы была создана и протестирована система персональных напоминаний, реализованная через Telegram-бота.

На этапе анализа требований был проведён анализ существующих решений, что помогло определиться с оптимальным набором функций: создание напоминаний с гибкими настройками времени, поддержку повторяющихся событий, систему приоритетов и двуязычный интерфейс.

Для реализации был выбран язык Python с использованием библиотеки `python-telegram-bot`. Серверная часть реализована с применением асинхронного подхода, что обеспечивает стабильную работу при высоких нагрузках. Хранение данных организовано в SQLite, включая три основные таблицы: для активных напоминаний, истории срабатываний и пользовательских настроек.

Особое внимание уделено обработке временных параметров. Бот автоматически определяет часовой пояс пользователя и корректно интерпретирует вводимые даты. Система использует встроенный планировщик задач `JobQueue` для своевременной доставки уведомлений с автоматическим восстановлением напоминаний при перезапуске системы.

Интерфейс реализован через комбинацию текстовых команд и интерактивных кнопок, с поддержкой русского и английского языков. `ConversationHandler` обеспечивает пошаговый процесс создания напоминаний.

Проведённое тестирование подтвердило корректность выполнения всех функций: создание, редактирование и удаление напоминаний работают без сбоев. Проверка под нагрузкой показала стабильную работу.

Разработанный бот представляет собой законченное решение для управления персональными напоминаниями, сочетающее удобный интерфейс с относительно надёжной технической реализацией. Система готова к

дальнейшему расширению функционала и масштабированию пользовательской базы.

Таким образом, цели выпускной квалификационной работы были достигнуты, а полученная система персональных напоминаний пользователя на основе Telegram-бота соответствует заявленным требованиям.

Список использованных источников

1. Джонсон Б. Управление полярностями / Барри Джонсон. – М.: Marathon Group, 2018. – 256 с.
2. Skillfactory Media, Интерфейс – что это простыми словами. Кто с ним работает и какие бывают типы. 2024. [Электронный ресурс]. – URL: <https://blog.skillfactory.ru/glossary/interface/> (дата обращения 11.05.2025)
3. Хувер Р., Эяль Н. На крючке. Как создавать продукты, формирующие привычки / Райан Хувер, Нир Эяль. – М.: Манн, Иванов и Фербер, 2014. – 272 с.
4. Центр программ развития руководителей НИУ ВШЭ User Experience Пользовательский опыт: Определение и основные характеристики: 2025. [Электронный ресурс]. – URL: <https://hsbi.hse.ru/articles/user-experience-opredelenie-i-osnovnye-kharakteristiki/>
5. Галиаскаров, Э.Г. Анализ и проектирование систем с использованием UML: учебник для вузов / Э. Г. Галиаскаров, А. С. Воробьев. – Москва: Издательство Юрайт, 2025. – 125 с.
6. С. В. Рындина Цифровые технологии управления получением, хранением, передачей и обработкой больших данных: SQLite: учебно-методическое пособие – Пенза, Издательство ПГУ, 2023. – 83 с.
7. Бондаренко А. В., Лисицын А. В., Лисицын Н. А. Кроссплатформенный информационный обмен на основе технологии распределенных баз данных / А. В. Бондаренко, А. В. Лисицын, Н. А. Лисицын. – М.: КнигИздат, 2023. – 146 с.
8. Демиденко А. С. Создание чат-ботов для начинающих: Telegram и Python / Артем Демиденко.: Изд.автора, 2025. – 90с.
9. Авторизация через Telegram Gateway: Quick-start Guide [Электронный ресурс].URL: <https://core.telegram.org/gateway/verification-tutorial> (дата обращения 17.05.2025).

10. Операции CRUD HTTP API [Электронный ресурс]. URL: https://ru.hexlet.io/courses/http-api/lessons/crud/theory_unit (дата обращения 19.05.2025).
11. Руководство по обработке исключений в Python [Электронный ресурс]. URL: <https://habr.com/ru/companies/wunderfund/articles/736526/> (дата обращения 20.05.2025).
12. Осипов Д. Л. Технологии проектирования баз данных. – М.: ДМК Пресс, 2019. – 498 с.
13. Мартин Р. Чистый код: создание, анализ и рефакторинг. – СПб.: Питер, 2024. – 464 с.
14. JobQueue Тutorial по использованию класса [Электронный ресурс]. URL: <https://docs.python-telegram-bot.org/en/v21.7/telegram.ext.jobqueue.html> (дата обращения 18.06.2025).
15. Фильтрация данных: подробное руководство по методам, преимуществам и передовым практикам [Электронный ресурс]. URL: <https://www.astera.com/ru/type/blog/data-filtering/> (дата обращения 19.06.2025).
16. Оптимизация работы с REST API: когда и как использовать пагинацию [Электронный ресурс]. URL: <https://selectel.ru/blog/pagination/> (дата обращения 20.06.2025).
17. Бирман И. Пользовательский интерфейс. – М.: Бюро Горбунова, 2023. – 274 с.
18. Хеллман Д. Стандартная библиотека Python 3: справочник с примерами. – М.: Диалектика-Вильямс, 2019. – 1375 с.
19. Гоф Дж. Проектирование архитектуры API. Как правильно проектировать, развивать и эксплуатировать API. – М.: АЛИСТ, 2024. – 288 с.
20. Особенности работы с HTTP в Python: методы, установка библиотеки, аутентификация [Электронный ресурс]. URL: <https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.nic.ru/help/osobennosti-raboty-s-http-v-python-metody-ustanovka-biblioteki->

autentifikaciya_11654.html&ved=2ahUKEwiZq_3WtIGOAxX1IRAIHSjgJGsQFn_oECBMQAQ&usg=AOvVaw1LnUi81S9iKAzsHMakf1jI (дата обращения 20.06.2025).

Приложение А
(рекомендуемое)
ГЛОССАРИЙ

API Gateway (Application Programming Interface) – цифровой помощник, который связывает пользователей с некоторыми микросервисами приложения и отдаёт результат. Он выполняет роль посредника между клиентами и микросервисами, управляя запросами, проверяя доступы и собирая статистику.

UML, или Unified Modeling Language – язык графического описания для объектного моделирования. Его используют, чтобы создавать диаграммы и схемы для визуализации процессов или явлений.

SQL (Structured Query Language) – язык программирования (язык структурированных запросов), который используют для создания, обработки и хранения данных в реляционных БД.

Telegram Bot API – это набор HTTP-методов, с помощью которых можно отправлять запросы к серверу Telegram, получать ответы в формате JSON и управлять ботами.

User Flow – это последовательность шагов, которые пользователь выполняет для достижения определенной цели на веб-сайте или в приложении.

PK – Primary Key (ключ первичный), в реляционной модели данных один из потенциальных ключей отношения, выбранный в качестве основного ключа (или же ключа по умолчанию).

FK – Foreign Key (внешний ключ), понятие теории реляционных баз данных, относящееся к ограничениям целостности базы данных.

Приложение Б
(обязательное)
ЛИСТИНГ ПРИЛОЖЕНИЯ

```

import logging
from datetime import datetime, timedelta, timezone
from telegram import Update, ReplyKeyboardMarkup,
InlineKeyboardMarkup, InlineKeyboardButton
from telegram.ext import (
    Application,
    CommandHandler,
    MessageHandler,
    CallbackContext,
    ConversationHandler,
    CallbackQueryHandler,
    filters
)
import sqlite3
import json
from enum import Enum
import pytz

# Настройка логирования
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s',
    level=logging.INFO
)
logger = logging.getLogger(__name__)

# Состояния для ConversationHandler
TITLE, DESCRIPTION, DATE, REPEAT, PRIORITY = range(5)
EDIT_CHOICE, EDIT_TITLE, EDIT_DESC, EDIT_DATE, EDIT_REPEAT,
EDIT_PRIORITY = range(6, 12)
DELETE_REMINDER = range(1)
SETTINGS, SET_LANGUAGE = range(2)
LIST_REMINDERS, EDIT_SELECT_REMINDER, DELETE_SELECT_REMINDER =
range(12, 15)

# Константы
TUTORIAL_URL = "https://telegra.ph/Spravka-bota-Napominaniya-06-
03"

# Приоритеты
class Priority(Enum):
    HIGH = {"ru": "🔴 Высокий", "en": "🔴 High"}
    MEDIUM = {"ru": "🟡 Средний", "en": "🟡 Medium"}
    LOW = {"ru": "🟢 Низкий", "en": "🟢 Low"}

# Языки
class Language(Enum):
    RU = "RU Русский"
    EN = "GB English"

# Инициализация базы данных

```

```

def init_db():
    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS reminders (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                user_id INTEGER,
                title TEXT,
                description TEXT,
                reminder_time TEXT,
                created_at TEXT,
                repeat_settings TEXT,
                priority TEXT,
                is_completed INTEGER DEFAULT 0,
                completed_at TEXT
            )
        ''')

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS history (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                reminder_id INTEGER,
                user_id INTEGER,
                triggered_at TEXT,
                FOREIGN KEY(reminder_id) REFERENCES
reminders(id)
            )
        ''')

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS user_settings (
                user_id INTEGER PRIMARY KEY,
                language TEXT DEFAULT 'ru'
            )
        ''')

        conn.commit()
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
    finally:
        if conn:
            conn.close()

init_db()

# Утилиты для работы с пользовательскими настройками
def get_user_settings(user_id: int) -> dict:
    try:

```

```

        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()
        cursor.execute("SELECT language FROM user_settings WHERE
user_id = ?", (user_id,))
        settings = cursor.fetchone()

        if not settings:
            default_settings = {"language": "ru"}
            cursor.execute(
VALUES (?, ?)",
                (user_id, default_settings["language"])
            )
            conn.commit()
            return default_settings
        return {"language": settings[0]}
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
        return {"language": "ru"}
    finally:
        if conn:
            conn.close()

def update_user_settings(user_id: int, language: str = None):
    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        if language:
            cursor.execute(
                "UPDATE user_settings SET language = ? WHERE
user_id = ?",
                (language, user_id))

            conn.commit()
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
    finally:
        if conn:
            conn.close()

# Утилиты для локализации
TEXTS = {
    "main_menu": {
        "ru": [
            ['📅 Создать напоминание'],
            ['📅 Мои напоминания', '✎ Редактировать'],
            ['🗑 Удалить', '📖 История'],
            ['⚙ Настройки', '❓ Помощь']
        ]
    }
}

```

```

    ],
    "en": [
        ['📅 Create reminder'],
        ['📅 My reminders', '✎ Edit'],
        ['🗑 Delete', '📖 History'],
        ['⚙ Settings', '❓ Help']
    ]
},
"welcome": {
    "ru": lambda name: f"Привет, {name}! 🙌\nЯ бот для управления напоминаниями.\nВыбери действие:",
    "en": lambda name: f"Hello, {name}! 🙌\nI'm your reminder management bot.\nChoose action:"
},
"back_button": {
    "ru": "⬅️ Назад в меню",
    "en": "⬅️ Back to menu"
},
"create_reminder_title": {
    "ru": "📌 Введите название напоминания:",
    "en": "📌 Enter reminder title:"
},
"create_reminder_description": {
    "ru": "📝 Введите описание напоминания (можно пропустить, нажав /skip):",
    "en": "📝 Enter reminder description (you can skip with /skip):"
},
"create_reminder_date": {
    "ru": "🕒 Введите дату и время напоминания (ДД.ММ.ГГГГ ЧЧ:ММ):\n\nПример: 31.12.2023 23:59",
    "en": "🕒 Enter reminder date and time (DD.MM.YYYY HH:MM):\n\nExample: 31.12.2023 23:59"
},
"invalid_date_format": {
    "ru": "❌ Неверный формат даты! Пожалуйста, используйте формат ДД.ММ.ГГГГ ЧЧ:ММ\n\nПопробуйте еще раз:",
    "en": "❌ Invalid date format! Please use DD.MM.YYYY HH:MM format\n\nTry again:"
},
"date_in_past": {
    "ru": "❌ Дата должна быть в будущем! Пожалуйста, введите корректную дату:",
    "en": "❌ Date must be in the future! Please enter correct date:"
},
"choose_repeat": {
    "ru": "🔄 Выберите тип повторения напоминания:",

```

```

    "en": "🔄 Choose reminder repeat type:"
  },
  "choose_priority": {
    "ru": "📌 Выберите приоритет напоминания:",
    "en": "📌 Choose reminder priority:"
  },
  "reminder_created": {
    "ru": lambda title, time: f"✅ Напоминание успешно  
создано!\n\n📌 Название: {title}\n🕒 Время: {time}",
    "en": lambda title, time: f"✅ Reminder created  
successfully!\n\n📌 Title: {title}\n🕒 Time: {time}"
  },
  "next_action": {
    "ru": "Выберите следующее действие:",
    "en": "Choose next action:"
  },
  "no_active_reminders": {
    "ru": "📌 У вас нет активных напоминаний.",
    "en": "📌 You have no active reminders."
  },
  "active_reminders_header": {
    "ru": "📌 Ваши активные напоминания:\n\n",
    "en": "📌 Your active reminders:\n\n"
  },
  "reminder_format": {
    "ru": lambda title, desc, time, priority, created: f"📌  
{title}\n📄 Описание: {desc or 'Нет описания'}\n🕒 Время:  
{time}\n📌 Приоритет: {priority}\n📅 Создано: {created}\n",
    "en": lambda title, desc, time, priority, created: f"📌  
{title}\n📄 Description: {desc or 'No description'}\n🕒 Time:  
{time}\n📌 Priority: {priority}\n📅 Created: {created}\n"
  },
  "select_reminder_to_edit": {
    "ru": "✎ Выберите напоминание для редактирования:",
    "en": "✎ Select reminder to edit:"
  },
  "select_reminder_to_delete": {
    "ru": "🗑 Выберите напоминание для удаления:",
    "en": "🗑 Select reminder to delete:"
  },
  "edit_choice": {
    "ru": "✎ Что вы хотите изменить в этом напоминании?",
    "en": "✎ What would you like to change in this  
reminder?"
  },
  "edit_field_title": {
    "ru": "📌 Введите новое название напоминания:",

```

```

    "en": "📌 Enter new reminder title:"
  },
  "edit_field_desc": {
    "ru": "📝 Введите новое описание напоминания:",
    "en": "📝 Enter new reminder description:"
  },
  "edit_field_date": {
    "ru": "🕒 Введите новое время напоминания (ДД.ММ.ГГГГ ЧЧ:ММ):",
    "en": "🕒 Enter new reminder date (DD.MM.YYYY HH:MM):"
  },
  "changes_saved": {
    "ru": "✅ Изменения успешно сохранены!",
    "en": "✅ Changes saved successfully!"
  },
  "delete_reminder_prompt": {
    "ru": "🗑 Введите ID напоминания для удаления:",
    "en": "🗑 Enter reminder ID to delete:"
  },
  "confirm_delete": {
    "ru": lambda title: f"? Вы уверены, что хотите удалить напоминание:\n\n📌 {title}",
    "en": lambda title: f"? Are you sure you want to delete reminder:\n\n📌 {title}"
  },
  "delete_confirmed": {
    "ru": "✅ Напоминание успешно удалено!",
    "en": "✅ Reminder deleted successfully!"
  },
  "delete_cancelled": {
    "ru": "❌ Удаление отменено.",
    "en": "❌ Deletion cancelled."
  },
  "no_history": {
    "ru": "ℹ История напоминаний пуста.",
    "en": "ℹ Reminder history is empty."
  },
  "history_header": {
    "ru": "📖 История ваших напоминаний:\n",
    "en": "📖 Your reminders history:\n"
  },
  "history_item": {
    "ru": lambda idx, title, time: f"{idx}. {title}\n🕒 {time}\n",
    "en": lambda idx, title, time: f"{idx}. {title}\n🕒 {time}\n"
  },

```

```

    "stats_header": {
        "ru": "📊 Ваша статистика:\n\nВсего напоминаний:
{total}\nВыполнено: {completed}",
        "en": "📊 Your stats:\n\nTotal reminders:
{total}\nCompleted: {completed}"
    },
    "settings_menu": {
        "ru": "⚙️ Настройки:\nТекущий язык: RU Русский",
        "en": "⚙️ Settings:\nCurrent language: GB English"
    },
    "current_settings": {
        "ru": "⚙️ Ваши текущие настройки:",
        "en": "⚙️ Your current settings:"
    },
    "choose_language": {
        "ru": "🌐 Выберите язык интерфейса:",
        "en": "🌐 Choose interface language:"
    },
    "language_changed": {
        "ru": "✅ Язык изменен на RU Русский",
        "en": "✅ Language changed to GB English"
    },
    "help_message": {
        "ru": f"📖 Руководство по использованию
бота:\n\n{TUTORIAL_URL}",
        "en": f"📖 Bot usage guide:\n\n{TUTORIAL_URL}"
    },
    "skip_description": {
        "ru": "ℹ️ Описание пропущено",
        "en": "ℹ️ Description skipped"
    },
    "reminder_received": {
        "ru": lambda title, desc: f"🔔 Напоминание!\n\n📌
{title}\n📝 {desc or 'Нет описания'}",
        "en": lambda title, desc: f"🔔 Reminder!\n\n📌
{title}\n📝 {desc or 'No description'}"
    },
    "reminder_read": {
        "ru": "\n\n✅ Прочитано",
        "en": "\n\n✅ Read"
    }
}

def get_text(key: str, language: str = "ru") -> str:
    return TEXTS.get(key, {}).get(language, key)

def get_repeat_text(repeat_type: str, language: str = "ru") ->
str:

```

```

types = {
    'none': {"ru": "✕ Без повторения", "en": "✕ No
repeat"},
    'daily': {"ru": "🔄 Ежедневно", "en": "🔄 Daily"},
    'weekly': {"ru": "📅 Еженедельно", "en": "📅 Weekly"},
    'monthly': {"ru": "📅 Ежемесячно", "en": "📅 Monthly"},
    'yearly': {"ru": "🌸 Ежегодно", "en": "🌸 Yearly"}
}
return types.get(repeat_type, {}).get(language, "")

# Клавиатуры
def get_main_keyboard(language: str = "ru") ->
ReplyKeyboardMarkup:
    return ReplyKeyboardMarkup(
        get_text("main_menu", language),
        resize_keyboard=True,
        one_time_keyboard=True
    )

def get_back_keyboard(language: str = "ru") ->
InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back")]
    ])

def get_repeat_keyboard(language: str = "ru") ->
InlineKeyboardMarkup:
    buttons = [
        [
            InlineKeyboardButton(get_repeat_text("none",
language), callback_data="repeat_none"),
        ],
        [
            InlineKeyboardButton(get_repeat_text("daily",
language), callback_data="repeat_daily"),
            InlineKeyboardButton(get_repeat_text("weekly",
language), callback_data="repeat_weekly")
        ],
        [
            InlineKeyboardButton(get_repeat_text("monthly",
language), callback_data="repeat_monthly"),
            InlineKeyboardButton(get_repeat_text("yearly",
language), callback_data="repeat_yearly")
        ],
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back")]
    ]
    return InlineKeyboardMarkup(buttons)

```

```

def get_priority_keyboard(language: str = "ru") ->
InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [
            InlineKeyboardButton(Priority.HIGH.value[language],
callback_data="priority_high"),
        ],
        [
            InlineKeyboardButton(Priority.MEDIUM.value[language],
callback_data="priority_medium"),
            InlineKeyboardButton(Priority.LOW.value[language],
callback_data="priority_low")
        ],
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back")]
    ])

def get_settings_keyboard(language: str = "ru") ->
InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [InlineKeyboardButton("🌐 Язык / Language",
callback_data="set_language")],
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back_to_menu")]
    ])

def get_language_keyboard(language: str = "ru") ->
InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [InlineKeyboardButton("RU Русский",
callback_data="lang_ru")],
        [InlineKeyboardButton("GB English",
callback_data="lang_en")],
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back_to_settings")]
    ])

def get_edit_choice_keyboard(language: str = "ru") ->
InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [InlineKeyboardButton("📌 Название" if language == "ru"
else "📌 Title", callback_data="edit_title")],
        [InlineKeyboardButton("📄 Описание" if language == "ru"
else "📄 Description", callback_data="edit_desc")],
        [InlineKeyboardButton("🕒 Время" if language == "ru"
else "🕒 Time", callback_data="edit_date")],
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back")]
    ])

```

```

def get_confirm_delete_keyboard(reminder_id: int, language: str
= "ru") -> InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [
            InlineKeyboardButton("✅ Да, удалить" if language ==
"ru" else "✅ Yes, delete",
callback_data=f"confirm_delete_{reminder_id}"),
            InlineKeyboardButton("❌ Нет, не надо" if language
== "ru" else "❌ No, cancel",
callback_data=f"cancel_delete_{reminder_id}")
        ],
        [InlineKeyboardButton(get_text("back_button", language),
callback_data="back")]
    ])

def get_reminder_received_keyboard(reminder_id: int, language:
str = "ru") -> InlineKeyboardMarkup:
    return InlineKeyboardMarkup([
        [InlineKeyboardButton("✅ OK",
callback_data=f"mark_read_{reminder_id}")]
    ])

# Функция для получения списка напоминаний с пагинацией
async def get_reminders_paginated(user_id: int, page: int = 1,
per_page: int = 5):
    conn = None
    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        # Получаем общее количество
        cursor.execute('SELECT COUNT(*) FROM reminders WHERE
user_id = ? AND is_completed = 0', (user_id,))
        total = cursor.fetchone()[0]

        # Получаем напоминания для страницы
        offset = (page - 1) * per_page
        cursor.execute('''
            SELECT id, title, description, reminder_time,
priority, created_at
            FROM reminders
            WHERE user_id = ? AND is_completed = 0
            ORDER BY
                CASE priority
                    WHEN '🔴 Высокий' THEN 1
                    WHEN '🔴 High' THEN 1
                    WHEN '🟡 Средний' THEN 2

```

```

        WHEN '🟡 Medium' THEN 2
        WHEN '🟢 Низкий' THEN 3
        WHEN '🟢 Low' THEN 3
        ELSE 4
    END,
    reminder_time ASC
LIMIT ? OFFSET ?
'', (user_id, per_page, offset))

reminders = cursor.fetchall()
return {
    'reminders': reminders,
    'page': page,
    'total_pages': (total + per_page - 1) // per_page,
    'total': total
}
except sqlite3.Error as e:
    logger.error(f"Database error: {e}")
    return None
finally:
    if conn:
        conn.close()

# Клавиатура для выбора напоминания с пагинацией
def get_reminders_keyboard(reminders_data, action_type: str,
language: str = "ru"):
    page = reminders_data['page']
    total_pages = reminders_data['total_pages']
    reminders = reminders_data['reminders']

    buttons = []
    for reminder in reminders:
        reminder_time =
datetime.fromisoformat(reminder[3]).strftime('%d.%m.%Y %H:%M')
        text = f"{reminder[1]} ({reminder_time})"
        buttons.append([InlineKeyboardButton(text,
callback_data=f"{action_type}_reminder_{reminder[0]}")])

    # Кнопки пагинации
    pagination_buttons = []
    if total_pages > 1:
        if page > 1:
            pagination_buttons.append(InlineKeyboardButton("⬅️",
callback_data=f"{action_type}_page_{page-1}"))

            pagination_buttons.append(InlineKeyboardButton(
                f"Страница {page}/{total_pages}" if language == "ru"
            else f"Page {page}/{total_pages}",
                callback_data="noop"
            ))

```

```

        if page < total_pages:
            pagination_buttons.append(InlineKeyboardButton("➡",
callback_data=f"{action_type}_page_{page+1}"))

    if pagination_buttons:
        buttons.append(pagination_buttons)

    buttons.append([InlineKeyboardButton(get_text("back_button",
language), callback_data="back")])

    return InlineKeyboardMarkup(buttons)

# Обработчики команд
async def start(update: Update, context: CallbackContext) ->
None:
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} ({user.first_name}) started the
bot")

    await update.message.reply_text(
        get_text("welcome",
settings["language"])(user.first_name),
        reply_markup=get_main_keyboard(settings["language"])
    )

async def help_command(update: Update, context: CallbackContext)
-> None:
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} requested help")

    await update.message.reply_text(
        get_text("help_message", settings["language"]),
        reply_markup=get_main_keyboard(settings["language"])
    )

async def back_callback(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = query.from_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} pressed back button")

    await query.edit_message_text(
        get_text("next_action", settings["language"]),
        reply_markup=None
    )

```

```

    await context.bot.send_message(
        user.id,
        get_text("next_action", settings["language"]),
        reply_markup=get_main_keyboard(settings["language"])
    )
    return ConversationHandler.END

async def back_to_menu(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = query.from_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} returned to main menu")

    await query.edit_message_text(
        get_text("next_action", settings["language"]),
        reply_markup=None
    )
    await context.bot.send_message(
        user.id,
        get_text("next_action", settings["language"]),
        reply_markup=get_main_keyboard(settings["language"])
    )
    return ConversationHandler.END

async def back_to_settings(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = query.from_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} returned to settings")

    await show_settings(update, context)
    return SETTINGS

async def show_settings(update: Update, context:
CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)

    language_name = "RU Русский" if settings["language"] == "ru"
    else "GB English"

    if update.callback_query:
        await update.callback_query.edit_message_text(
            get_text("settings_menu",
settings["language"]).replace("Текущий язык:", f"Текущий язык:
{language_name}"),

```

```

reply_markup=get_settings_keyboard(settings["language"])
    )
    else:
        await update.message.reply_text(
            get_text("settings_menu",
settings["language"]).replace("Текущий язык:", f"Текущий язык:
{language_name}"),

reply_markup=get_settings_keyboard(settings["language"])
    )
    return SETTINGS

# Создание напоминания
async def create_reminder(update: Update, context:
CallbackContext) -> int:
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} started creating a reminder")

    await update.message.reply_text(
        get_text("create_reminder_title", settings["language"]),
        reply_markup=get_back_keyboard(settings["language"])
    )
    return TITLE

async def get_title(update: Update, context: CallbackContext) ->
int:
    user = update.effective_user
    settings = get_user_settings(user.id)

    if update.callback_query and update.callback_query.data ==
"back":
        await back_callback(update, context)
        return ConversationHandler.END

    title = update.message.text
    context.user_data['title'] = title

    logger.info(f"User {user.id} set reminder title: {title}")

    await update.message.reply_text(
        get_text("create_reminder_description",
settings["language"]),
        reply_markup=get_back_keyboard(settings["language"])
    )
    return DESCRIPTION

async def skip_description(update: Update, context:
CallbackContext) -> int:
    user = update.effective_user

```

```

settings = get_user_settings(user.id)

context.user_data['description'] = ""

logger.info(f"User {user.id} skipped reminder description")

await update.message.reply_text(
    get_text("skip_description", settings["language"]),
    reply_markup=get_back_keyboard(settings["language"])
)
await update.message.reply_text(
    get_text("create_reminder_date", settings["language"]),
    reply_markup=get_back_keyboard(settings["language"])
)
return DATE

async def get_description(update: Update, context:
CallbackContext) -> int:
    user = update.effective_user
    settings = get_user_settings(user.id)

    if update.callback_query and update.callback_query.data ==
"back":
        await back_callback(update, context)
        return ConversationHandler.END

    description = update.message.text
    context.user_data['description'] = description

    logger.info(f"User {user.id} set reminder description:
{description}")

    await update.message.reply_text(
        get_text("create_reminder_date", settings["language"]),
        reply_markup=get_back_keyboard(settings["language"])
    )
    return DATE

async def get_date(update: Update, context: CallbackContext) ->
int:
    user = update.effective_user
    settings = get_user_settings(user.id)

    if not update.message or not update.message.text:
        await update.message.reply_text(
            get_text("invalid_date_format",
settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
    return DATE

```

```

        if update.callback_query and update.callback_query.data ==
"back":
            await back_callback(update, context)
            return ConversationHandler.END

    try:
        date_text = update.message.text
        if not ('.' in date_text and ' ' in date_text and (':'
in date_text or '.' in date_text)):
            raise ValueError("Неверный формат даты")

        date_part, time_part = date_text.split()
        time_part = time_part.replace('.', ':')

        # Получаем часовой пояс пользователя автоматически
        user_timezone = pytz.timezone('Etc/GMT' + str(-
int(datetime.now().astimezone().utcoffset().total_seconds() /
3600)))
        reminder_time = datetime.strptime(f"{date_part}
{time_part}", "%d.%m.%Y %H:%M")
        reminder_time = user_timezone.localize(reminder_time)

        if reminder_time <= datetime.now(user_timezone):
            await update.message.reply_text(
                get_text("date_in_past", settings["language"]),
reply_markup=get_back_keyboard(settings["language"])
            )
            return DATE

        context.user_data['reminder_time'] = reminder_time

        logger.info(f"User {user.id} set reminder time:
{reminder_time}")

        await update.message.reply_text(
            get_text("choose_repeat", settings["language"]),
reply_markup=get_repeat_keyboard(settings["language"])
        )
        return REPEAT

    except ValueError as e:
        logger.error(f"Date parsing error: {e}")
        await update.message.reply_text(
            get_text("invalid_date_format",
settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
        return DATE

```

```

async def get_repeat(update: Update, context: CallbackContext) -
> int:
    query = update.callback_query
    await query.answer()
    user = update.effective_user
    settings = get_user_settings(user.id)

    if query.data == "back":
        await back_callback(update, context)
        return ConversationHandler.END

    if query.data.startswith("repeat_"):
        repeat_type = query.data.split('_')[1]
        context.user_data['repeat_type'] = repeat_type

        if repeat_type == 'none':
            context.user_data['repeat_settings'] = None
        elif repeat_type in ['daily', 'weekly', 'monthly',
'yearly']:
            context.user_data['repeat_settings'] = {'interval':
repeat_type}

        logger.info(f"User {user.id} set repeat type:
{repeat_type}")

        await query.edit_message_text(
            get_text("choose_priority", settings["language"]),

reply_markup=get_priority_keyboard(settings["language"])
        )
        return PRIORITY

async def get_priority(update: Update, context: CallbackContext)
-> int:
    query = update.callback_query
    await query.answer()
    user = update.effective_user
    settings = get_user_settings(user.id)

    if query.data == "back":
        await back_callback(update, context)
        return ConversationHandler.END

    priority = query.data.split('_')[1]
    context.user_data['priority'] =
Priority[priority.upper()].value[settings["language"]]

    logger.info(f"User {user.id} set priority: {priority}")

    await save_reminder(update, context, user.id)

```

```

        reminder_time =
context.user_data['reminder_time'].strftime('%d.%m.%Y %H:%M')

        await query.edit_message_text(
            get_text("reminder_created", settings["language"])(
                context.user_data['title'],
                reminder_time
            ),
            reply_markup=None
        )

        await context.bot.send_message(
            user.id,
            get_text("next_action", settings["language"]),
            reply_markup=get_main_keyboard(settings["language"])
        )

        return ConversationHandler.END

async def save_reminder(update: Update, context:
CallbackContext, user_id: int):
    conn = None
    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        title = context.user_data['title']
        description = context.user_data.get('description', '')
        reminder_time =
context.user_data['reminder_time'].isoformat()
        priority = context.user_data['priority']
        repeat_settings =
json.dumps(context.user_data.get('repeat_settings', None))

        cursor.execute(
            '''INSERT INTO reminders
            (user_id, title, description, reminder_time,
created_at, repeat_settings, priority)
            VALUES (?, ?, ?, ?, ?, ?, ?)''',
            (
                user_id,
                title,
                description,
                reminder_time,
                datetime.now().isoformat(),
                repeat_settings,
                priority
            )
        )

        reminder_id = cursor.lastrowid

```

```

        conn.commit()

        logger.info(f"Reminder created: id={reminder_id},
user_id={user_id}, "
                    f"title='{title}', time={reminder_time},
priority={priority}, "
                    f"repeat={repeat_settings}")

        await schedule_reminder(context, reminder_id, user_id)
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
    finally:
        if conn:
            conn.close()

async def schedule_reminder(context: CallbackContext,
reminder_id: int, user_id: int):
    conn = None
    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM reminders WHERE id = ?",
(reminder_id,))
        reminder = cursor.fetchone()

        if not reminder:
            return

        reminder_time = datetime.fromisoformat(reminder[4])
        now = datetime.now(reminder_time.tzinfo)
        delay = (reminder_time - now).total_seconds()

        if delay <= 0:
            logger.warning(f"Reminder time {reminder_time} is in
the past")
            return

        job_context = {
            'user_id': user_id,
            'reminder_id': reminder_id,
            'title': reminder[2],
            'description': reminder[3]
        }

        repeat_settings = json.loads(reminder[6]) if reminder[6]
else None

        if repeat_settings and 'interval' in repeat_settings:
            interval_map = {
                'daily': timedelta(days=1),
                'weekly': timedelta(weeks=1),

```

```

        'monthly': timedelta(days=30),
        'yearly': timedelta(days=365)
    }

    if repeat_settings['interval'] in interval_map:
        context.job_queue.run_repeating(
            send_reminder_callback,

interval=interval_map[repeat_settings['interval']],
            first=delay,
            data=job_context,
            name=str(reminder_id))
    else:
        context.job_queue.run_once(
            send_reminder_callback,
            when=delay,
            data=job_context,
            name=str(reminder_id))
except sqlite3.Error as e:
    logger.error(f"Database error: {e}")
except json.JSONDecodeError:
    logger.error(f"Failed to parse repeat settings for
reminder {reminder_id}")
finally:
    if conn:
        conn.close()

async def send_reminder_callback(context: CallbackContext):
    job = context.job
    user_id = job.data['user_id']
    reminder_id = job.data['reminder_id']
    settings = get_user_settings(user_id)
    conn = None

    try:
        message = await context.bot.send_message(
            user_id,
            get_text("reminder_received", settings["language"])(
                job.data['title'],
                job.data['description']
            ),

reply_markup=get_reminder_received_keyboard(reminder_id,
settings["language"])
        )

        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO history (reminder_id, user_id,
triggered_at) VALUES (?, ?, ?)",

```

```

        (reminder_id, user_id, datetime.now().isoformat())
    )
    conn.commit()

    logger.info(f"Reminder sent: user_id={user_id},
reminder_id={reminder_id}")
    except Exception as e:
        logger.error(f"Failed to send reminder: {e}")
    finally:
        if conn:
            conn.close()

async def mark_reminder_as_read(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = query.from_user
    settings = get_user_settings(user.id)

    reminder_id = int(query.data.split('_')[2])

    logger.info(f"User {user.id} marked reminder {reminder_id}
as read")

    try:
        await query.edit_message_text(
            query.message.text + get_text("reminder_read",
settings["language"]),
            reply_markup=None
        )
    except Exception as e:
        logger.error(f"Failed to mark reminder as read: {e}")

async def show_history(update: Update, context:
CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)
    conn = None

    logger.info(f"User {user.id} requested history")

    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        cursor.execute('''
            SELECT r.title, h.triggered_at
            FROM history h
            JOIN reminders r ON h.reminder_id = r.id
            WHERE h.user_id = ?
            ORDER BY h.triggered_at DESC
        ''')

```

```

        LIMIT 10
        '', (user.id,))

    history = cursor.fetchall()

    if not history:
        await
    update.message.reply_text(get_text("no_history",
settings["language"]))
        return

    text = get_text("history_header", settings["language"])
    for idx, item in enumerate(history, 1):
        triggered_time =
datetime.fromisoformat(item[1]).strftime('%d.%m.%Y %H:%M')
        text += get_text("history_item",
settings["language"])(
            idx, item[0], triggered_time
        )

    await update.message.reply_text(text)
except sqlite3.Error as e:
    logger.error(f"Database error: {e}")
    await update.message.reply_text("Ошибка базы данных")
finally:
    if conn:
        conn.close()

async def show_stats(update: Update, context: CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)
    conn = None

    logger.info(f"User {user.id} requested stats")

    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        cursor.execute("SELECT COUNT(*) FROM reminders WHERE
user_id = ?", (user.id,))
        total = cursor.fetchone()[0]

        cursor.execute("SELECT COUNT(*) FROM history WHERE
user_id = ?", (user.id,))
        completed = cursor.fetchone()[0]

        text = get_text("stats_header",
settings["language"]).format(
            total=total, completed=completed
        )

```

```

        await update.message.reply_text(text)
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
        await update.message.reply_text("Ошибка базы данных")
    finally:
        if conn:
            conn.close()

async def list_reminders(update: Update, context:
CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)
    conn = None

    logger.info(f"User {user.id} requested reminders list")

    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        cursor.execute('''
            SELECT title, description, reminder_time, priority,
created_at
            FROM reminders
            WHERE user_id = ? AND is_completed = 0
            ORDER BY
                CASE priority
                    WHEN '🔴 Высокий' THEN 1
                    WHEN '🔴 High' THEN 1
                    WHEN '🟡 Средний' THEN 2
                    WHEN '🟡 Medium' THEN 2
                    WHEN '🟢 Низкий' THEN 3
                    WHEN '🟢 Low' THEN 3
                    ELSE 4
                END,
                reminder_time ASC
        ''', (user.id,))

        reminders = cursor.fetchall()

        if not reminders:
            await
update.message.reply_text(get_text("no_active_reminders",
settings["language"]))
            return

        text = get_text("active_reminders_header",
settings["language"])
        for reminder in reminders:

```

```

        reminder_time =
datetime.fromisoformat(reminder[2]).strftime('%d.%m.%Y %H:%M')
        created_at =
datetime.fromisoformat(reminder[4]).strftime('%d.%m.%Y %H:%M')
        text += get_text("reminder_format",
settings["language"])(
            reminder[0], reminder[1], reminder_time,
reminder[3], created_at
        )
        text += "\n" # Добавляем пустую строку между
напоминаниями

        await update.message.reply_text(text)
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
        await update.message.reply_text("Ошибка базы данных")
    finally:
        if conn:
            conn.close()

# Редактирование напоминания
async def edit_reminder(update: Update, context:
CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} started editing a reminder")

    reminders_data = await get_reminders_paginated(user.id)
    if not reminders_data or not reminders_data['reminders']:
        await update.message.reply_text(
            get_text("no_active_reminders",
settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
    return ConversationHandler.END

    await update.message.reply_text(
        get_text("select_reminder_to_edit",
settings["language"]),
        reply_markup=get_reminders_keyboard(reminders_data,
"edit", settings["language"])
    )
    return EDIT_SELECT_REMINDER

# Удаление напоминания
async def delete_reminder(update: Update, context:
CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} started deleting a reminder")

```

```

reminders_data = await get_reminders_paginated(user.id)
if not reminders_data or not reminders_data['reminders']:
    await update.message.reply_text(
        get_text("no_active_reminders",
settings["language"]),
        reply_markup=get_back_keyboard(settings["language"])
    )
    return ConversationHandler.END

    await update.message.reply_text(
        get_text("select_reminder_to_delete",
settings["language"]),
        reply_markup=get_reminders_keyboard(reminders_data,
"delete", settings["language"])
    )
    return DELETE_SELECT_REMINDER

# Обработчик пагинации и выбора напоминания
async def handle_reminder_selection(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = query.from_user
    settings = get_user_settings(user.id)

    data = query.data
    if data == "back":
        await back_callback(update, context)
        return ConversationHandler.END

    if data == "noop":
        return

    # Обработка пагинации
    if data.startswith(("edit_page_", "delete_page_")):
        action, _, page = data.split('_')
        page = int(page)

        reminders_data = await get_reminders_paginated(user.id,
page)
        if not reminders_data:
            await query.edit_message_text(
                "Ошибка загрузки напоминаний",

reply_markup=get_back_keyboard(settings["language"])
            )
            return

        text = (
            get_text("select_reminder_to_edit",
settings["language"]) if action == "edit"

```

```

        else get_text("select_reminder_to_delete",
settings["language"])
    )

    await query.edit_message_text(
        text,
        reply_markup=get_reminders_keyboard(reminders_data,
action, settings["language"])
    )
    return

# Обработка выбора напоминания
action, _, reminder_id = data.split('_')
reminder_id = int(reminder_id)

if action == "edit":
    # Сохраняем ID для редактирования
    context.user_data['edit_id'] = reminder_id

    logger.info(f"User {user.id} selected reminder
{reminder_id} for editing")

    await query.edit_message_text(
        get_text("edit_choice", settings["language"]),
reply_markup=get_edit_choice_keyboard(settings["language"])
    )
    return EDIT_CHOICE
elif action == "delete":
    # Подтверждение удаления
    conn = None
    try:
        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()
        cursor.execute("SELECT title FROM reminders WHERE id
= ? AND user_id = ?", (reminder_id, user.id))
        reminder = cursor.fetchone()

        if reminder:
            logger.info(f"User {user.id} selected reminder
{reminder_id} for deletion")

            await query.edit_message_text(
                get_text("confirm_delete",
settings["language"])(reminder[0]),

reply_markup=get_confirm_delete_keyboard(reminder_id,
settings["language"])
            )
        except sqlite3.Error as e:
            logger.error(f"Database error: {e}")

```

```

        finally:
            if conn:
                conn.close()
        return ConversationHandler.END

async def edit_reminder_field(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = update.effective_user
    settings = get_user_settings(user.id)

    if query.data == "back":
        await back_callback(update, context)
        return ConversationHandler.END

    field = query.data.split('_')[1]
    context.user_data['edit_field'] = field

    logger.info(f"User {user.id} chose to edit {field} of
reminder")

    if field == "title":
        await query.edit_message_text(
            get_text("edit_field_title", settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
        return EDIT_TITLE
    elif field == "desc":
        await query.edit_message_text(
            get_text("edit_field_desc", settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
        return EDIT_DESC
    elif field == "date":
        await query.edit_message_text(
            get_text("edit_field_date", settings["language"]),
            reply_markup=get_back_keyboard(settings["language"])
        )
        return EDIT_DATE

async def save_edited_field(update: Update, context:
CallbackContext):
    field = context.user_data['edit_field']
    reminder_id = context.user_data['edit_id']
    user = update.effective_user
    settings = get_user_settings(user.id)
    conn = None

    try:
        if field in ["title", "desc"]:
            new_value = update.message.text

```

```

        column = "title" if field == "title" else
"description"
        elif field == "date":
            try:
                date_text = update.message.text
                if not ('.' in date_text and ' ' in date_text
and (':' in date_text or '.' in date_text)):
                    raise ValueError("Неверный формат даты")

                date_part, time_part = date_text.split()
                time_part = time_part.replace('.', ':')

                user_timezone = pytz.timezone('Etc/GMT' + str(-
int(datetime.now().astimezone().utcoffset().total_seconds() /
3600)))

                new_value = datetime.strptime(f"{date_part}
{time_part}", "%d.%m.%Y %H:%M")
                new_value =
user_timezone.localize(new_value).isoformat()
                column = "reminder_time"
            except ValueError:
                await update.message.reply_text(
                    get_text("invalid_date_format",
settings["language"]),

reply_markup=get_back_keyboard(settings["language"])
                )
                return EDIT_DATE

        conn = sqlite3.connect('reminders.db',
check_same_thread=False)
        cursor = conn.cursor()

        # Получаем старые данные для логов
        cursor.execute(f"SELECT {column} FROM reminders WHERE id
= ?", (reminder_id,))
        old_value = cursor.fetchone()[0]

        # Обновляем данные
        cursor.execute(f"UPDATE reminders SET {column} = ? WHERE
id = ?", (new_value, reminder_id))
        conn.commit()

        logger.info(f"User {user.id} edited reminder
{reminder_id}: "
                    f"field={column}, old_value='{old_value}',
new_value='{new_value}'")

        # Обновляем задание в очереди
        current_jobs =
context.job_queue.get_jobs_by_name(str(reminder_id))
        for job in current_jobs:

```

```

        job.schedule_removal()

    await schedule_reminder(context, reminder_id, user.id)

    await update.message.reply_text(
        get_text("changes_saved", settings["language"]),
        reply_markup=get_main_keyboard(settings["language"])
    )
    return ConversationHandler.END
except sqlite3.Error as e:
    logger.error(f"Database error: {e}")
    await update.message.reply_text(
        "Ошибка базы данных",
        reply_markup=get_back_keyboard(settings["language"])
    )
    return EDIT_TITLE if field == "title" else EDIT_DESC if
field == "desc" else EDIT_DATE
finally:
    if conn:
        conn.close()

async def execute_delete(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = update.effective_user
    settings = get_user_settings(user.id)
    conn = None

    if query.data == "back":
        await query.edit_message_text(
            "Удаление отменено",
            reply_markup=None
        )
        await context.bot.send_message(
            user.id,
            get_text("next_action", settings["language"]),
            reply_markup=get_main_keyboard(settings["language"])
        )
        return

    if query.data.startswith("cancel_delete_"):
        await query.edit_message_text(
            get_text("delete_cancelled", settings["language"]),
            reply_markup=None
        )
        await context.bot.send_message(
            user.id,
            get_text("next_action", settings["language"]),
            reply_markup=get_main_keyboard(settings["language"])
        )
        return

```

```

reminder_id = int(query.data.split('_')[2])

try:
    conn = sqlite3.connect('reminders.db',
check_same_thread=False)
    cursor = conn.cursor()

    # Получаем данные напоминания для логов перед удалением
    cursor.execute("SELECT title, reminder_time FROM
reminders WHERE id = ?", (reminder_id,))
    reminder_data = cursor.fetchone()

    if reminder_data:
        title, reminder_time = reminder_data

        # Удаляем напоминание
        cursor.execute("DELETE FROM reminders WHERE id = ?",
(reminder_id,))
        conn.commit()

        logger.info(f"User {user.id} deleted reminder:
id={reminder_id}, "
                    f"title='{title}', time={reminder_time}")

        # Удаляем связанные задания в очереди
        current_jobs =
context.job_queue.get_jobs_by_name(str(reminder_id))
        for job in current_jobs:
            job.schedule_removal()

        await query.edit_message_text(
            get_text("delete_confirmed",
settings["language"])
        )
        await context.bot.send_message(
            user.id,
            get_text("next_action", settings["language"]),
reply_markup=get_main_keyboard(settings["language"])
        )
    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
        await query.edit_message_text(
            "Ошибка при удалении",
            reply_markup=None
        )
    finally:
        if conn:
            conn.close()

# Настройки

```

```

async def settings_menu(update: Update, context:
CallbackContext):
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} opened settings")

    await show_settings(update, context)
    return SETTINGS

async def set_language(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = update.effective_user
    settings = get_user_settings(user.id)

    logger.info(f"User {user.id} opened language settings")

    await query.edit_message_text(
        get_text("choose_language", settings["language"]),
        reply_markup=get_language_keyboard(settings["language"])
    )
    return SET_LANGUAGE

async def save_language(update: Update, context:
CallbackContext):
    query = update.callback_query
    await query.answer()
    user = update.effective_user
    settings = get_user_settings(user.id)

    if query.data == "back_to_settings":
        await show_settings(update, context)
        return SETTINGS

    language = query.data.split('_')[1]
    update_user_settings(user.id, language=language)

    logger.info(f"User {user.id} changed language to
{language}")

    await query.edit_message_text(
        get_text("language_changed", language),
        reply_markup=None
    )

    await context.bot.send_message(
        user.id,
        get_text("next_action", language),
        reply_markup=get_main_keyboard(language)
    )

```



```

        'yearly': timedelta(days=365)
    }

    if repeat_settings['interval'] in interval_map:
        application.job_queue.run_repeating(
            send_reminder_callback,

interval=interval_map[repeat_settings['interval']],
            first=delay,
            data=job_context,
            name=str(reminder_id))
    else:
        application.job_queue.run_once(
            send_reminder_callback,
            when=delay,
            data=job_context,
            name=str(reminder_id))

except sqlite3.Error as e:
    logger.error(f"Database error: {e}")
except json.JSONDecodeError:
    logger.error("Failed to parse repeat settings")
finally:
    if conn:
        conn.close()

def main() -> None:
    # Замените YOUR_TOKEN_HERE на реальный токен вашего бота
    application =
Application.builder().token("7965324005:AAGMd1Bt2bLJ66vgjtywp0E9
1f5gN46EaAs").build()

    # Восстановление напоминаний при запуске
    application.add_handler(MessageHandler(filters.COMMAND,
restore_reminders_on_startup), group=-1)

    # Обработчики команд
    application.add_handler(CommandHandler("start", start))
    application.add_handler(CommandHandler("help",
help_command))
    application.add_handler(CommandHandler("skip",
skip_description))
    application.add_handler(MessageHandler(filters.Regex('^📅 Мои
напоминания$|^📅 My reminders$'), list_reminders))
    application.add_handler(MessageHandler(filters.Regex('^📖
История$|^📖 History$'), show_history))
    application.add_handler(MessageHandler(filters.Regex('^?
Помощь$|^? Help$'), help_command))

    # Обработчик создания напоминания
    conv_handler = ConversationHandler(

```

```

        entry_points=[MessageHandler(filters.Regex('^📅 Создать
напоминание$|^📅 Create reminder$'), create_reminder)],
        states={
            TITLE: [
                MessageHandler(filters.TEXT & ~filters.COMMAND,
get_title),
                CallbackQueryHandler(back_callback,
pattern="^back$")
            ],
            DESCRIPTION: [
                MessageHandler(filters.TEXT & ~filters.COMMAND,
get_description),
                CommandHandler("skip", skip_description),
                CallbackQueryHandler(back_callback,
pattern="^back$")
            ],
            DATE: [
                MessageHandler(filters.TEXT & ~filters.COMMAND,
get_date),
                CallbackQueryHandler(back_callback,
pattern="^back$")
            ],
            REPEAT: [
                CallbackQueryHandler(get_repeat),
                CallbackQueryHandler(back_callback,
pattern="^back$")
            ],
            PRIORITY: [
                CallbackQueryHandler(get_priority),
                CallbackQueryHandler(back_callback,
pattern="^back$")
            ],
        },
        fallbacks=[],
    )
    application.add_handler(conv_handler)

    # Обработчик редактирования напоминания
    edit_handler = ConversationHandler(
        entry_points=[MessageHandler(filters.Regex('^✎
Редактировать$|^✎ Edit$'), edit_reminder)],
        states={
            EDIT_SELECT_REMINDER:
[CallbackQueryHandler(handle_reminder_selection)],
            EDIT_CHOICE:
[CallbackQueryHandler(edit_reminder_field)],
            EDIT_TITLE: [
                MessageHandler(filters.TEXT & ~filters.COMMAND,
save_edited_field),
                CallbackQueryHandler(back_callback,
pattern="^back$")

```

```

        ],
        EDIT_DESC: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
save_edited_field),
            CallbackQueryHandler(back_callback,
pattern="^back$")
        ],
        EDIT_DATE: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
save_edited_field),
            CallbackQueryHandler(back_callback,
pattern="^back$")
        ],
    },
    fallbacks=[],
)
application.add_handler(edit_handler)

# Обработчик удаления напоминания
delete_handler = ConversationHandler(
    entry_points=[MessageHandler(filters.Regex('^🗑
Удалить$|^🗑 Delete$'), delete_reminder)],
    states={
        DELETE_SELECT_REMINDER:
[CallbackQueryHandler(handle_reminder_selection)],
    },
    fallbacks=[],
)
application.add_handler(delete_handler)

# Обработчик настроек
settings_handler = ConversationHandler(
    entry_points=[MessageHandler(filters.Regex('^⚙
Настройки$|^⚙ Settings$'), settings_menu)],
    states={
        SETTINGS: [
            CallbackQueryHandler(set_language,
pattern="^set_language$"),
            CallbackQueryHandler(back_to_menu,
pattern="^back_to_menu$")
        ],
        SET_LANGUAGE: [
            CallbackQueryHandler(save_language),
            CallbackQueryHandler(back_to_settings,
pattern="^back_to_settings$")
        ],
    },
    fallbacks=[],
)
application.add_handler(settings_handler)

```

```

    # Обработчики кнопок
    application.add_handler(CallbackQueryHandler(execute_delete,
pattern="^confirm_delete_"))
    application.add_handler(CallbackQueryHandler(execute_delete,
pattern="^cancel_delete_"))

application.add_handler(CallbackQueryHandler(mark_reminder_as_re
ad, pattern="^mark_read_"))

    # Восстановление напоминаний при запуске
    application.job_queue.run_once(lambda ctx:
restore_reminders_on_startup(application), when=0)

    application.run_polling()

if __name__ == '__main__':
    main()

```

Приложение В
(обязательное)
РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

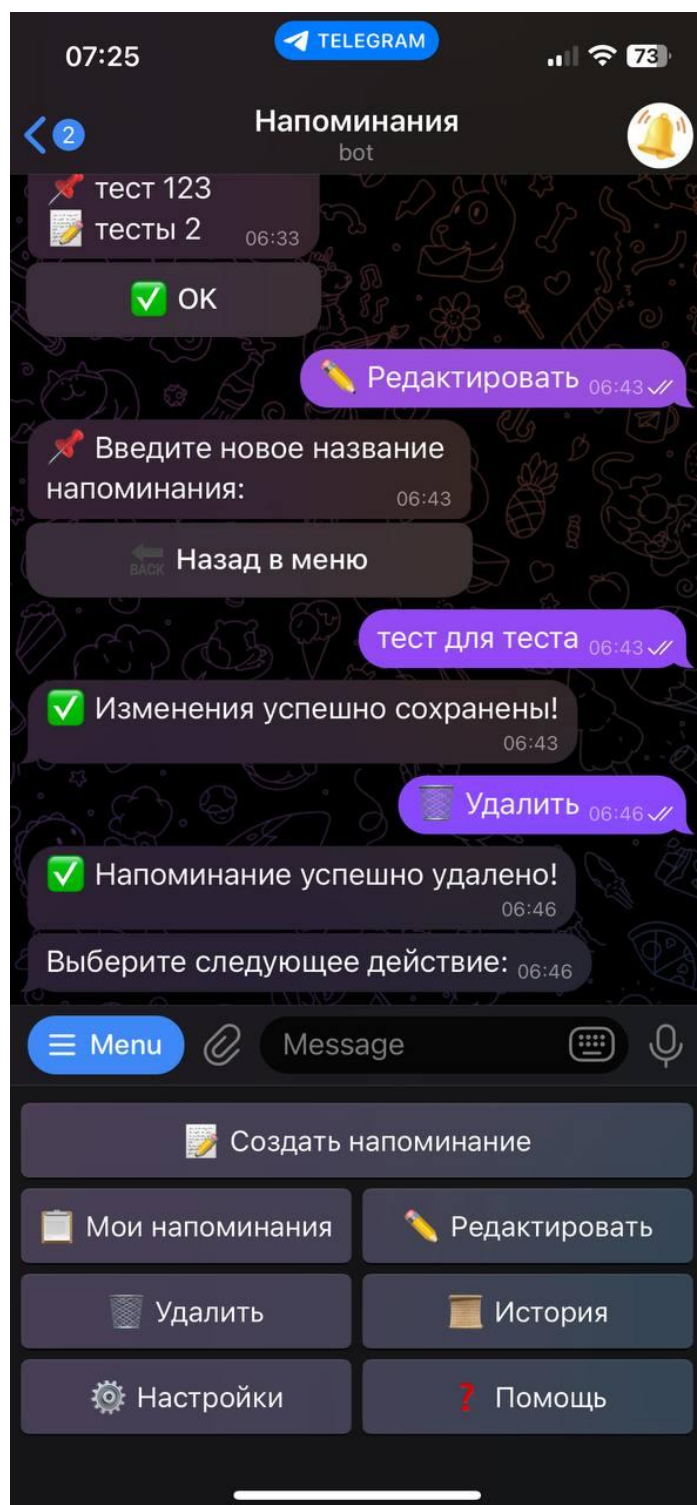


Рисунок В.1 – Результат тест кейса 1

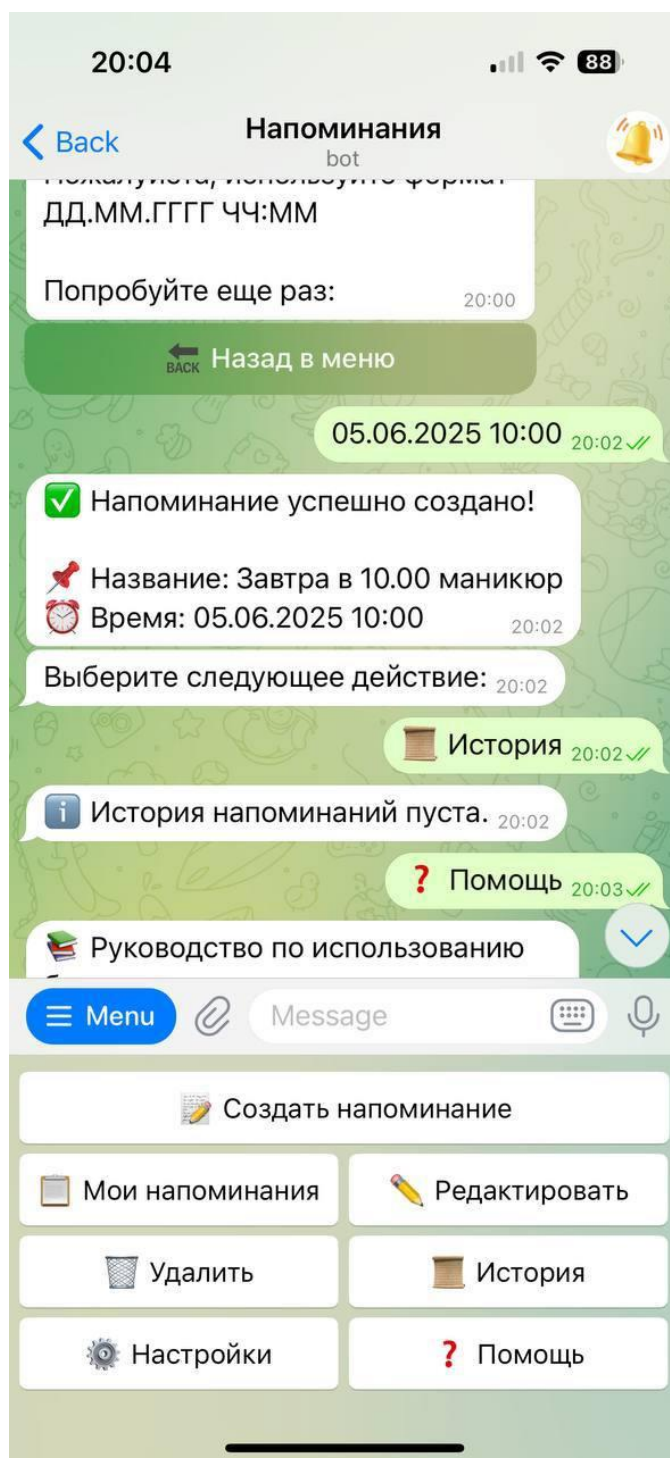


Рисунок В.2 – Результат тест кейса 2

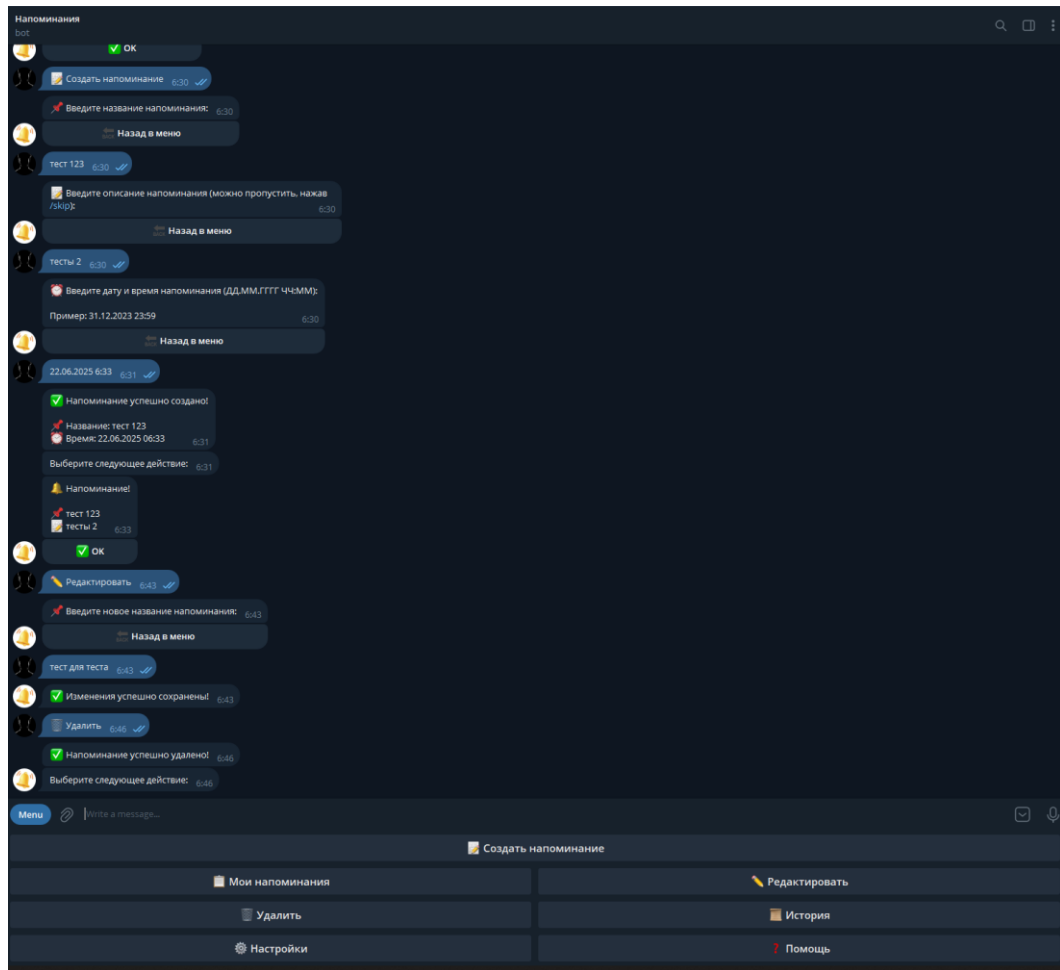


Рисунок В.3 – Результат тест кейса 3

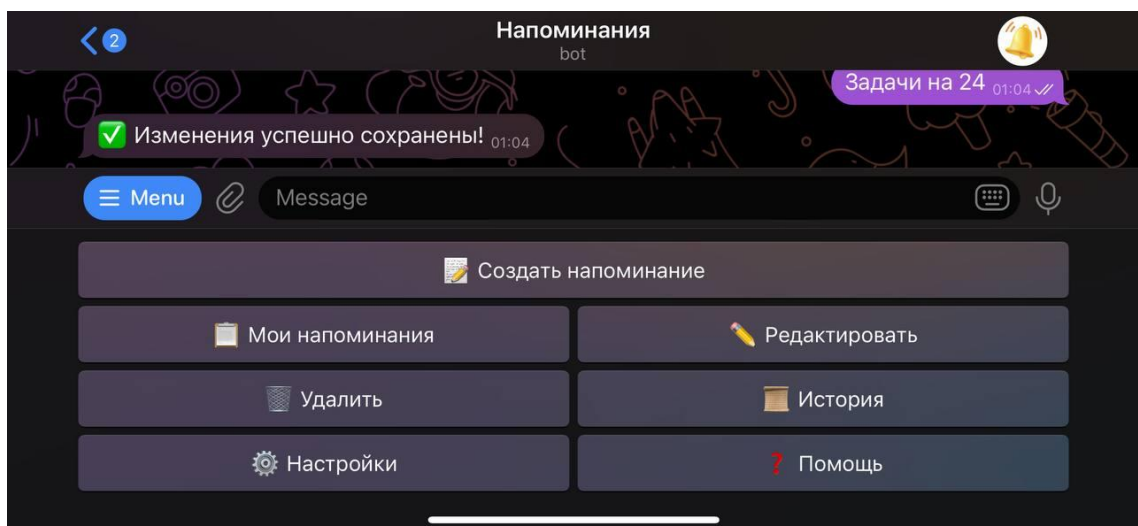


Рисунок В.4 – Результат тест кейса 4