Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

**СЕТЕВОЙ ТРЕКЕР ЗАДАЧ И ПРИВЫЧЕК**

БГУИР КР 6-05-0612-01 008 ПЗ

Студент Т.А. Пашкович

Руководитель С.В. Болтак

Минск 2025

**СОДЕРЖАНИЕ**

Введение...................................................................................................................3

1 Постановка задачи………....................................................................................5

1.1 Описание предметной области.......................................................................5

1.2 Сравнительный анализ существующих решений.........................................5

1.3 Информационная база задачи.........................................................................8

1.4 Функциональное назначение........................................................................10

2 Проектирование задачи……..............................................................................12

2.1 Алгоритм решения задачи…………………….............................................12

2.2 Логическое моделирование…………...........................................................13

2.3 Выбор и обоснование инструментов разработки……………....................14

3 Программная реализация………………….......................................................16

3.1 Физическая структура…...............................................................................16

3.2 Описание разработанных модулей...............................................................18

4 Тестирование…………………...........................................................................21

5 Применение программы…………….................................................................20

5.1 Руководство пользователя….........................................................................20

Список используемых источников.......................................................................31

Приложение ….......................................................................................................32

**ВВЕДЕНИЕ**

Сетевой-трекер задач и привычек — это современное веб-приложение, предназначенное для эффективного управления личными и рабочими задачами, а также формирования полезных привычек. Основная цель трекера — помочь пользователям систематизировать ежедневные дела, отслеживать прогресс и повышать продуктивность за счет автоматизации рутинных процессов.

В условиях высокой загруженности и многозадачности важно иметь удобный инструмент, который позволяет фиксировать планы, контролировать их выполнение и анализировать результаты. Данный трекер решает эту проблему, предоставляя интуитивно понятный интерфейс для создания задач, установки сроков, категоризации активности и мониторинга достижений. Такие системы особенно востребованы среди студентов, фрилансеров, предпринимателей и всех, кто стремится к осознанному тайм-менеджменту.

Современные трекеры задач и привычек сочетают в себе функции планировщика, ежедневника и аналитической системы. Они позволяют:

* Формировать списки дел с приоритезацией.
* Устанавливать напоминания и дедлайны.
* Отслеживать регулярные привычки (например, спорт, чтение, изучение языков).
* Визуализировать прогресс с помощью графиков и статистики.
* Синхронизировать данные между устройствами.

Внедрение подобного приложения способствует дисциплине, снижает уровень стресса и помогает пользователям достигать поставленных целей. Автоматизация учета задач минимизирует риск забыть важные дела, а анализ статистики позволяет корректировать планы для большей эффективности.

Пояснительная записка к курсовому проекту на тему «Онлайн-трекер задач и привычек» включает следующие разделы:

1. Постановка задачи – определение целей разработки, анализ потребностей пользователей, описание функциональных требований к системе.
2. Проектирование системы – выбор архитектуры приложения, проектирование базы данных, описание основных модулей (управление задачами, трекинг привычек, аналитика).
3. Программная реализация – описание ключевых алгоритмов (добавление задач, напоминания, учет прогресса), используемых технологий и фреймворков.
4. Тестирование – проверка корректности работы интерфейса, тестирование функций создания задач и формирования отчетов, оценка производительности.
5. Применение системы – сценарии использования трекера в реальных условиях, оценка его эффективности для повышения личной продуктивности.
6. Заключение – анализ результатов разработки, возможности дальнейшего расширения функционала (интеграция с календарями, голосовые помощники, геймификация).
7. Список использованных источников – перечень материалов, использованных при проектировании и разработке приложения.

Курсовая работа на тему «Cетевой-трекер задач и привычек» способствует эффективному управлению личной продуктивностью, помогая пользователям систематизировать ежедневные дела, формировать полезные привычки и достигать поставленных целей. Приложение упрощает процесс планирования, обеспечивает наглядную визуализацию прогресса и способствует развитию самодисциплины.

Для разработки программы используются интегрированные среды разработки (IDE) IntelliJ IDEA и WebStorm, которые предоставляют мощный инструментарий для работы с языками программирования Java, JavaScript и TypeScript. Данные среды предлагают удобную систему управления проектами, интеллектуальное автодополнение кода, встроенные средства отладки и тестирования, а также широкие возможности для создания современного пользовательского интерфейса и работы с базами данных.

**1 ПОСТАНОВКА ЗАДАЧИ**

**1.1 Описание предметной области**

Курсовой проект посвящён разработке программного обеспечения «Онлайн-трекер задач и привычек». Основной целью проекта является создание удобного и эффективного инструмента для планирования, отслеживания и анализа личной продуктивности.

В условиях высокой загруженности, многозадачности и необходимости формирования полезных привычек автоматизация процесса самоорганизации становится особенно актуальной. Онлайн-трекер призван помочь пользователям систематизировать ежедневные задачи, контролировать выполнение целей, а также вырабатывать устойчивые положительные привычки. Программа предоставляет интуитивно понятный интерфейс, позволяющий легко добавлять задачи, устанавливать сроки, отслеживать прогресс и анализировать статистику.

Разработка ориентирована на широкий круг пользователей, включая студентов, фрилансеров, офисных сотрудников и всех, кто стремится к повышению личной эффективности. Приложение поддерживает гибкие настройки, напоминания и визуализацию данных, что способствует мотивации и дисциплине. Внедрение такого трекера в повседневную жизнь позволяет снизить уровень стресса, улучшить тайм-менеджмент и добиваться поставленных целей более осознанно.

## **1.2 Сравнительный анализ существующих решений**

Для разработки программного средства "Онлайн-трекер задач и привычек" важно учитывать существующие решения на рынке, чтобы создать продукт, соответствующий современным стандартам и предпочтениям пользователей. Аналоги – это программы, которые имеют сходные функции и цели. Несмотря на общую задачу, такие решения могут значительно отличаться по удобству использования, функциональности и технологическим возможностям.

В сфере управления задачами и формирования привычек существует несколько популярных решений, которые могут служить ориентиром для разработки. Рассмотрим два наиболее известных из них, проведем анализ их сильных и слабых сторон.

Todoist

Программа Todoist является одним из самых распространённых решений для управления задачами. Она предоставляет комплексные инструменты для планирования дел, организации проектов и контроля выполнения.

Преимущества**:**

* Поддержка широкого спектра функций для управления задачами, включая создание проектов, установку меток и приоритетов.
* Гибкие возможности интеграции с другими сервисами, включая Google Calendar, Slack и Dropbox.
* Кроссплатформенность, позволяющая работать с приложением на ПК, смартфонах и через веб-браузер.
* Простой и интуитивно понятный интерфейс.

Недостатки**:**

* Отсутствие специализированных инструментов для отслеживания привычек.
* Многие полезные функции (например, напоминания и расширенная аналитика) доступны только в платной версии.
* Ограниченные возможности визуализации прогресса.

На рисунке 1.1 изображен интерфейс Todoist.

Изображение выглядит как текст, программное обеспечение, веб-страница, Значок на компьютере

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1.1 – Интерфейс Todoist

Habitica

Программа Habitica представляет собой уникальное решение, сочетающее трекинг привычек с элементами геймификации.

Преимущества:

* Мотивационная система, основанная на игровых механиках (уровни, награды, внутриигровая валюта).
* Возможность совместной работы в группах для достижения общих целей.
* Визуализация прогресса в виде графиков и статистики.
* Увлекательный подход к формированию привычек.

Недостатки:

* Нестандартный интерфейс может вызывать сложности у новых пользователей.
* Ориентация на игровые элементы в ущерб серьёзным инструментам анализа продуктивности.
* Ограниченные возможности для профессионального тайм-менеджмента.

На рисунке 1.2 изображен интерфейс Habitica.

Изображение выглядит как текст, программное обеспечение, веб-страница, Веб-сайт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1.2 – Интерфейс Habitica

Анализ существующих решений показал, что каждая из рассмотренных программ имеет свои преимущества и недостатки. Для создания программного средства "Онлайн-трекер задач и привычек" необходимо учитывать успешные аспекты данных систем, такие как:

* Простота и удобство интерфейса (на примере Todoist)
* Эффективные механизмы мотивации (на примере Habitica)
* Возможности интеграции с другими сервисами
* Кроссплатформенная доступность

В то же время следует избегать характерных недостатков существующих решений:

* Излишней сложности интерфейса
* Ограничения базового функционала платной подпиской
* Чрезмерного акцента на игровые элементы в ущерб практической полезности
* Недостаточных возможностей аналитики

Разрабатываемый трекер должен объединить сильные стороны рассмотренных аналогов, предлагая:

* Интуитивно понятный интерфейс для быстрого освоения
* Сбалансированную систему мотивации
* Комплексные инструменты анализа продуктивности
* Полноценный бесплатный функционал

Такой подход позволит создать современный и конкурентоспособный продукт для управления личной эффективностью.

## **1.3 Информационная база задачи**

В рамках данной курсовой работы разрабатывается программное средство "Онлайн-трекер задач и привычек", предназначенное для управления личной продуктивностью. Основной задачей разработки является создание удобного интерфейса для планирования задач, отслеживания привычек и анализа прогресса.

Основные информационные объекты:

1. Пользователи
   * Основная сущность системы, содержащая данные о зарегистрированных пользователях:
   * Email (уникальный идентификатор)
   * Пароль (хэшированный)
   * Имя пользователя
   * Имя и фамилия
   * Ссылка на Telegram
   * Статус аккаунта
   * Роль (обычный пользователь/администратор)
   * Дата регистрации
2. Привычки
   * Основной объект для трекинга повторяющихся действий:
   * Название привычки
   * Описание
   * Время на выполнение
   * Крайний срок выполнения
   * ID пользователя
   * Дата последнего выполнения
   * Дата создания
3. Логи привычек
   * История выполнения привычек:
   * Запланированная дата выполнения
   * Статус выполнения (выполнено/не выполнено)
   * Ссылка на привычку
4. Проекты
   * Группы задач, объединенные общей целью:
   * Название проекта (уникальное)
   * Дата создания
   * Дата последнего обновления
   * ID пользователя
   * Список состояний задач
5. Состояния задач
   * Этапы выполнения задач в рамках проекта:
   * Название состояния
   * Тип отображения (список, доска и др.)
   * Левое и правое состояние (для организации workflow)
   * Дата создания
   * Ссылка на проект
   * Список задач
6. Задачи
   * Основная единица планирования:
   * Название задачи
   * Описание
   * Дедлайн
   * Категория (работа, учеба, личное и др.)
   * Приоритет (высокий, средний, низкий)
   * Задачи с более высоким и низким приоритетом
   * Состояние задачи
   * Дата создания
   * Дата обновления

Дополнительные аспекты:

* Система поддерживает различные категории задач (работа, учеба, здоровье и др.)
* Реализована система приоритетов для эффективного тайм-менеджмента
* Поддерживается гибкий workflow через состояния задач
* Ведется полная история выполнения привычек

Информационная база задачи охватывает все ключевые аспекты управления личной продуктивностью и обеспечивает удобный способ организации задач и привычек для пользователей.

## **1.4 Функциональное назначение**

Программное средство «Онлайн-трекер задач и привычек» предназначено для решения следующих задач:

1. Добавление задач: Пользователь может создавать новые задачи, указывая название, описание, срок выполнения, категорию и приоритет через простой интерфейс.
2. Управление задачами:
   * Редактирование существующих задач (изменение названия, описания, сроков)
   * Удаление выполненных или ненужных задач
   * Изменение статуса задач (в работе, выполнено, отложено)
3. Создание привычек: Возможность добавлять регулярные действия с указанием названия, описания и частоты выполнения.
4. Отслеживание привычек:
   * Отметка о выполнении ежедневных привычек
   * Просмотр статистики выполнения за неделю/месяц
   * Редактирование параметров привычек
5. Организация задач:
   * Сортировка задач по дате, приоритету или категории
   * Группировка задач по проектам
   * Фильтрация задач по статусу или тегам
6. Просмотр статистики:
   * Анализ выполнения задач за период
   * Отображение прогресса по привычкам
   * Формирование отчетов о продуктивности

Эти функции реализованы через удобный веб-интерфейс с использованием современных технологий (Angular, Spring Boot), что обеспечивает простоту и интуитивность использования приложения как на компьютерах, так и на мобильных устройствах. Интерфейс разработан с учетом потребностей различных пользователей - от студентов до профессионалов.

.

# **2 ПРОЕКТИРОВАНИЕ ЗАДАЧИ**

## **2.1 Алгоритм решения задачи**

Процесс создания программного средства "Онлайн-трекер задач и привычек" состоит из нескольких этапов:

1. Анализтребований  
   На этом этапе определяются основные функции системы:

* Управление задачами (создание, редактирование, удаление, изменение статуса)
* Формирование и отслеживание привычек
* Организация задач по проектам и категориям
* Просмотр статистики продуктивности
* Разрабатывается структура данных: определяются сущности (пользователи, задачи, привычки, проекты) и их взаимосвязи в базе данных

1. Проектирование системы  
   На этапе проектирования создается архитектура приложения:

* Система будет состоять из backend-сервера, базы данных и frontend-интерфейса
* Выбираются технологии: Spring Boot для backend, PostgreSQL для БД, React для frontend
* Разрабатывается API для взаимодействия компонентов системы
* Проектируется пользовательский интерфейс: экраны управления задачами, привычками и просмотра статистики

1. Реализация  
   Выполняется непосредственная разработка системы:

* Создается база данных для хранения информации о пользователях, задачах и привычках
* Реализуется бизнес-логика работы с задачами и привычками
* Разрабатываются REST API endpoints для взаимодействия с клиентскими приложениями
* Создается веб-интерфейс для работы с системой

1. Тестирование  
   Проводится комплексная проверка системы:

* Тестируется корректность работы всех функций (создание задач, отметка привычек)
* Проверяется производительность системы при нагрузке
* Оценивается удобство пользовательского интерфейса
* Выявляются и исправляются обнаруженные ошибки

1. Внедрение

* Настройка рабочего окружения для production
* Развертывание системы на сервере
* Подготовка документации для пользователей
* Обучение работе с системой

## **2.2 Логическое моделирование**

Для создания системы онлайн-трекера задач и привычек база данных организована в виде следующих основных сущностей: пользователи, привычки, проекты, задачи и история выполнения. Такой подход позволяет организовать данные так, чтобы они удовлетворяли всем требованиям приложения и обеспечивали удобство работы с информацией.

Коллекция "Пользователи" содержит данные для авторизации и работы с системой: уникальный идентификатор, email, хэш пароля, имя и фамилия, контактные данные (включая ссылку на Telegram), а также роль (пользователь или администратор). Это обеспечивает безопасный доступ к системе и персонализацию работы.

Коллекция "Привычки" хранит информацию о регулярных действиях пользователя: название привычки, подробное описание, время на выполнение, периодичность следования, текущий статус выполнения и связь с конкретным пользователем. Такая структура позволяет эффективно отслеживать регулярную активность.

Для организации задач используется коллекция "Проекты", которая включает название проекта, дату создания, список связанных задач и принадлежность пользователю. Это помогает группировать задачи по тематическим категориям.

Основная рабочая единица системы - сущность "Задачи" содержит: название задачи, детальное описание, срок выполнения, показатель приоритета, категорию (работа, учеба или личное), статус выполнения и связь с конкретным проектом.

Дополнительная сущность "История выполнения" фиксирует прогресс пользователя: дату выполнения, тип действия (привычка или задача), статус выполнения и затраченное время. Это позволяет анализировать продуктивность.

Для обеспечения высокой производительности реализована индексация по ключевым полям: email пользователей, названиям задач и привычек, датам выполнения и статусам задач. Такая структура обеспечивает быстрый доступ к актуальным данным, удобную фильтрацию по различным параметрам, эффективный анализ продуктивности и возможность масштабирования системы.

**2.3 Выбор и обоснование инструментов разработки**

Для разработки программного обеспечения "Онлайн-трекер задач и привычек" был выбран следующий набор инструментов и технологий:

1. IntelliJ IDEA Ultimate в качестве основной среды разработки. Этот выбор обусловлен:
   * Полноценной поддержкой Java и Spring Framework
   * Умным автодополнением кода и рефакторингом
   * Встроенными инструментами для работы с базами данных
   * Отличной интеграцией с Docker и системами контроля версий
2. Spring Boot как основной фреймворк для backend-разработки:
   * Быстрая настройка и конфигурирование приложения
   * Автоматическое управление зависимостями
   * Встроенная поддержка безопасности (Spring Security)
   * Удобная работа с REST API
3. PostgreSQL в качестве системы управления базами данных:
   * Надежность и стабильность работы
   * Поддержка сложных запросов и транзакций
   * Хорошая интеграция с Java-приложениями
   * Возможности масштабирования
4. Docker для контейнеризации приложения:
   * Простота развертывания и масштабирования
   * Изоляция компонентов системы
   * Удобство управления зависимостями
   * Кроссплатформенность
5. Angular для frontend-разработки:
   * Компонентный подход к построению интерфейса
   * Высокая производительность
   * Богатая экосистема библиотек
   * Простота интеграции с backend
6. Postman для тестирования API:
   * Удобное создание и выполнение запросов
   * Возможность сохранения коллекций тестов
   * Автоматизация тестирования
   * Генерация документации

Выбранный набор инструментов обеспечивает:

* Высокую производительность разработки
* Надежность работы приложения
* Удобство тестирования и отладки
* Простоту развертывания и масштабирования
* Современный и удобный пользовательский интерфейс
* Хорошую поддержку командной работы

Все компоненты хорошо интегрируются между собой и представляют собой современный, надежный и производительный стек технологий для разработки веб-приложений.

# **3 РЕАЛИЗАЦИЯ СИСТЕМЫ**

**3.1 Физическая структура**

Физическая структура приложения "Онлайн-трекер задач и привычек" основана на взаимодействии трех ключевых компонентов: базы данных PostgreSQL, пользовательского интерфейса на React и модулей бизнес-логики на Spring Boot. Такая архитектура обеспечивает четкое разделение ответственности между слоями приложения, что упрощает его разработку, сопровождение и масштабирование.

База данных PostgreSQL   
 Для хранения данных используется реляционная СУБД PostgreSQL, которая обеспечивает надежное хранение и целостность данных. Информация организована в таблицах, каждая из которых отвечает за определенный аспект системы:

1. Таблица "users" содержит данные пользователей:
   * Уникальный идентификатор
   * Email и хэш пароля
   * Личные данные (имя, фамилия)
   * Контактную информацию
   * Роль в системе
2. Таблица "habits" хранит информацию о привычках:
   * Название и описание
   * Периодичность выполнения
   * Время на выполнение
   * Связь с пользователем
3. Таблица "projects" содержит данные о проектах:
   * Название проекта
   * Дата создания
   * Связь с пользователем
4. Таблица "tasks" включает информацию о задачах:
   * Название и описание
   * Срок выполнения
   * Приоритет и статус
   * Связь с проектом
5. Таблица "habit\_logs" фиксирует историю выполнения привычек:
   * Дата выполнения
   * Статус (выполнено/не выполнено)
   * Связь с привычкой

Использование PostgreSQL обеспечивает надежное хранение данных и поддержку сложных запросов. Индексация ключевых полей (email пользователей, названия задач и привычек) гарантирует высокую производительность даже при увеличении объема данных.

Пользовательский интерфейс

Интерфейс разработан с использованием Angular, что обеспечивает:

* Современный и отзывчивый дизайн
* Быстрое взаимодействие с пользователем
* Адаптивность под разные устройства
* Удобное управление задачами и привычками через интуитивно понятные элементы

Интерфейс взаимодействует с backend через REST API, обеспечивая безопасный и эффективный обмен данными.

Модули бизнес-логики

Серверная часть на Spring Boot обрабатывает основные бизнес-процессы:

* Аутентификацию и авторизацию пользователей
* Управление жизненным циклом задач и привычек
* Валидацию входящих данных
* Генерацию отчетов и статистики
* Обеспечение безопасности данных

Модули бизнес-логики реализуют:

* Создание, чтение, обновление и удаление данных (CRUD)
* Проверку прав доступа к данным
* Бизнес-правила (например, контроль сроков выполнения задач)
* Интеграцию с внешними сервисами (например, Telegram для уведомлений)

Такая структура делает приложение надежным, производительным и готовым к дальнейшему расширению функциональности. Использование Docker для контейнеризации компонентов упрощает развертывание и масштабирование системы.

**3.2 Описание разработанных модулей**

Программное средство "Сетевой трекер задач и привычек" построено на основе микросервисной архитектуры, что обеспечивает масштабируемость, гибкость и независимость компонентов. Система разделена на три ключевых сервиса: auth-gateway, task-service и habit-service, каждый из которых отвечает за отдельный функциональный домен. Взаимодействие между сервисами осуществляется через REST API с централизованным управлением запросами через API Gateway.

1. Микросервис auth-gateway

Выполняет функции аутентификации, авторизации и маршрутизации запросов. Включает следующие модули:

1. Модуль аутентификации:
   * Регистрация пользователей с верификацией email
   * JWT-аутентификация с генерацией токенов доступа и обновления
   * Ролевая модель (user/admin) с управлением правами через claims
   * Интеграция с OAuth2-провайдерами (Google, GitHub)
   * Механизм инвалидации токенов при выходе из системы
2. Модуль администратора:
   * Просмотр и редактирование пользовательских профилей (только для роли admin)
   * Управление блокировкой аккаунтов
   * Сброс паролей с использованием хеширования bcrypt
   * Аудит действий пользователей
3. API Gateway:
   * Единая точка входа для всех внешних запросов
   * Маршрутизация к task-service и habit-service на основе URL-путей
   * Реализация rate limiting для защиты от DDoS-атак
   * Валидация и трансформация входящих/исходящих данных

2. Микросервис task-service

Отвечает за функциональность управления проектами, задачами и workflow. Состоит из следующих модулей:

1. Модуль задач:
   * Создание задач с приоритетом, сроками и тегами
   * Динамическая фильтрация (по статусу, исполнителю, дате)
   * Drag-and-drop для изменения статуса (to-do, in progress, done)
   * Уведомления о дедлайнах через WebSocket
2. Модуль проектов:
   * Иерархическая структура проектов с настраиваемыми ролями участников
   * Кастомизация столбцов (цвета, иконки, лимиты задач)
   * Визуализация прогресса через Gantt-диаграммы
   * Экспорт данных в CSV/PDF
3. Модуль аналитики:
   * Статистика по выполненным задачам
   * Отчеты о загрузке команды
   * Интеграция с внешними BI-инструментами

3. Микросервис habit-service

Реализует логику формирования и отслеживания привычек. Основные компоненты:

1. Модуль привычек:
   * Создание целей с настройкой периодичности (ежедневно, еженедельно)
   * Трекер выполнения с календарём и графиком прогресса
   * Система напоминаний через Telegram-бота
   * Расчет "цепочки привычек" и анализ пропущенных дней
2. Модуль мотивации:
   * Награды за достижение целей (бейджи, уровни)
   * Публичные рейтинги пользователей
   * Персонализированные рекомендации на основе ML-моделей

Особенности взаимодействия сервисов

1. Схема аутентификации:
   * Все запросы к task-service и habit-service требуют JWT-токена, валидируемого auth-gateway
   * Для межсервисного взаимодействия используются внутренние API-ключи
2. Обмен данными:
   * Сервисы изолированы: task-service и habit-service имеют отдельные БД (PostgreSQL)
3. Масштабируемость:
   * Каждый микросервис может развертываться в Docker-контейнерах с балансировкой нагрузки
   * Все БД для микросервисов развертываются в в Docker-контейнерах, для удобного использования и быстрой разработки приложения
4. Безопасность:
   * Изоляция ошибок: сбои в habit-service не влияют на работу task-service
   * Для получения данных с сайта нужно быть обязательно авторизованным, также пользователь получит только те проекты и привычки, к которым он имеет доступ

Архитектура системы обеспечивает минимальное время отклика за счёт декомпозиции ответственностей и оптимизированных протоколов взаимодействия.

**4 ТЕСТИРОВАНИЕ**

Для обеспечения высокого качества и надежности работы программного продукта "Сетевой трекер задач и привычек" было проведено комплексное тестирование всех ключевых компонентов системы. Основной целью тестирования стала верификация корректности работы каждого функционального модуля и предотвращение потенциальных ошибок в процессе эксплуатации приложения.

В процессе разработки были реализованы модульные тесты для проверки базовых функций системы. Каждый тест был направлен на валидацию работы отдельных компонентов, включая:

* Создание и управление задачами
* Формирование и отслеживание привычек
* Систему напоминаний и уведомлений
* Статистику и аналитику продуктивности
* Авторизацию и аутентификацию пользователей

Для каждой значимой функции были разработаны тестовые сценарии, которые на основе различных входных данных проверяют соответствие фактических результатов ожидаемым. Особое внимание уделялось:

1. Граничным случаям при вводе данных
2. Обработке ошибочных сценариев
3. Взаимодействию между модулями
4. Производительности критических операций

Для реализации модульных тестов использовался фреймворк JUnit 5 в сочетании с Mockito для тестирования Java-приложения. Этот инструментарий предоставляет:

* Аннотации для определения тестовых методов (@Test, @BeforeEach и др.)
* Механизмы параметризованного тестирования
* Возможности мокирования зависимостей
* Удобные ассерты для проверки результатов
* Гибкую систему организации тестовых сценариев

Пример тестового метода представлен на рисунке 4.1.

Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 4.1 – пример тестового метода

В данном примере реализованы тесты для проверки корректности работы функций получения и фильтрации проектов в системе. Эти тесты позволяют убедиться, что основные операции с проектами выполняются корректно и соответствуют бизнес-логике приложения.

Метод getProject\_shouldReturnProject\_whenProjectExists()

Тест проверяет корректность получения информации о конкретном проекте по его идентификатору. В тесте:

1. Мокируется возврат тестового объекта ProjectEntity
2. Проверяется, что контроллер возвращает ожидаемый ProjectDto
3. Верифицируется вызов необходимых сервисных методов

Если проект существует в системе и принадлежит авторизованному пользователю, метод возвращает полные данные о проекте в формате DTO.

Метод fetchProject\_shouldReturnAllProjects\_whenNoPrefix()

Тест проверяет работу фильтрации проектов:

1. Создается тестовый список проектов пользователя
2. Проверяется возврат полного списка при отсутствии фильтра
3. Контролируется корректность преобразования Entity в DTO
4. Верифицируется вызов репозитория

Если параметр фильтрации не указан, система возвращает все проекты, принадлежащие пользователю.

Результаты выполнения тестов показаны на Рисунке 4.2.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 4.2 – Отображение выполненных тестов

На рисунке 4.2 видно, что тесты для следующих функций завершились успешно:

1. getProject\_shouldReturnProject\_whenProjectExists()  
Проверяет загрузку информации о конкретном проекте по его ID из базы данных. Тест подтверждает, что система корректно возвращает данные существующего проекта.

2. fetchProject\_shouldReturnAllProjects\_whenNoPrefix()  
Проверяет загрузку всех проектов пользователя при отсутствии фильтров. Тестирует работу системы в режиме полного списка без ограничений.

3. fetchProject\_shouldReturnFilteredProjects\_whenPrefixProvided()  
Проверяет фильтрацию проектов по названию. Тестирует работу поискового функционала и корректность применения фильтров.

4. createOrUpdateProject\_shouldCreateNewProject\_whenIdNotProvided()  
Проверяет добавление новых проектов в систему. Тестирует корректность создания проектов и сохранения их в базе данных.

5. createOrUpdateProject\_shouldThrowException\_whenCreateAndNameEmpty()  
Проверяет обработку ошибок при создании проекта с пустым названием. Тестирует валидацию входных данных.

6. createOrUpdateProject\_shouldUpdateProject\_whenIdProvided()  
Проверяет обновление данных существующих проектов. Тестирует функционал редактирования проектов.

7. createOrUpdateProject\_shouldThrowException\_whenNameAlreadyExists()  
Проверяет обработку конфликтов имен проектов. Тестирует систему на предотвращение дублирования названий.

8. deleteProject\_shouldReturnAck\_whenProjectExists()  
Проверяет удаление проектов из системы. Тестирует корректность работы механизма удаления и возвращаемый статус операции.

Пример тестового класса представлен на рисунке 4.3.

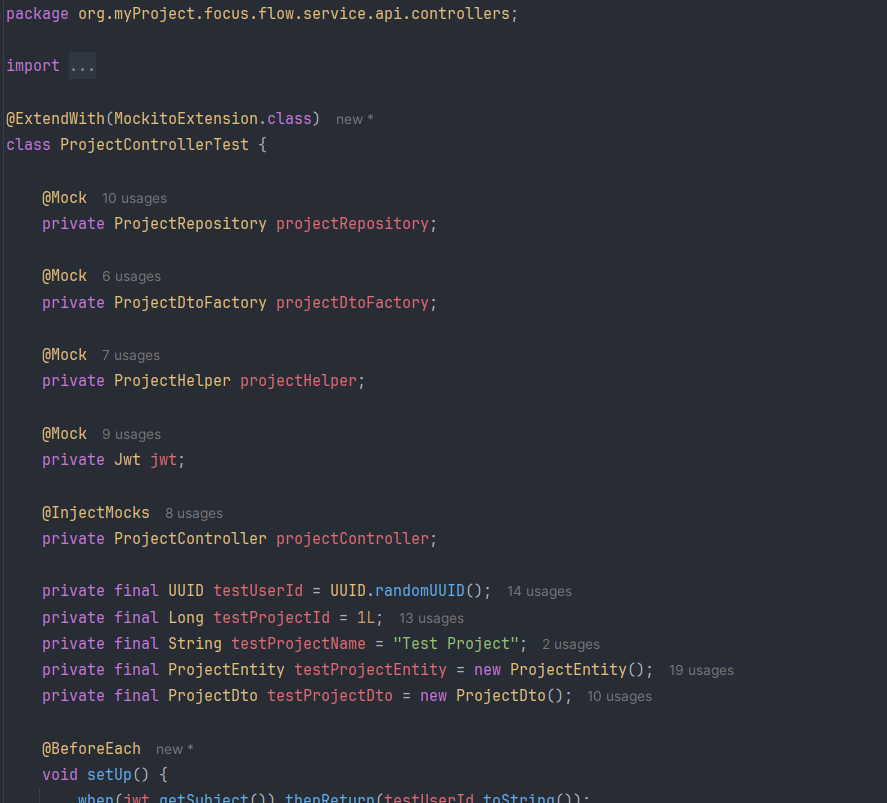


Рисунок 4.3 – Пример тестового класса

Тестовый класс включает методы, которые проверяют различные аспекты работы приложения. Процесс тестирования помогает гарантировать стабильность и правильность функционирования основных функций приложения «Сетевой трекер задач и привычек».

**5 ПРИМЕНЕНИЕ ПРОГРАММЫ**

**5.1 Руководство пользователя**

Программное средство «Онлайн-трекер задач и привычек» представляет собой веб-приложение, предназначенное для управления личной продуктивностью. Система позволяет пользователям регистрироваться, авторизовываться, создавать и отслеживать задачи, формировать полезные привычки, а также анализировать свою эффективность.

При запуске приложения отображается начальное окно, которое также является окном авторизации (рисунок 5.1), где пользователю предлагается выбор между авторизацией и регистрацией.

Авторизация предназначена для входа в уже существующую учетную запись. Пользователь вводит логин и пароль, после чего получает доступ ко всем функциям приложения.

При успешной авторизации открывается основное окно приложения, где отображаются доступные задачи и привычки.

В случае ошибок (например, неверного логина или пароля) выводится соответствующее уведомление.

Регистрация позволяет создать новую учетную запись, заполнив необходимые данные (например, имя пользователя, адрес электронной почты и пароль)

Изображение выглядит как текст, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 5.1 – Начальное окно

Перейдя к окну регистрации (Рисунок 5.2), пользователь может создать новую учётную запись. Для этого необходимо заполнить следующие поля:

* Email
* Пароль
* Имя пользователя
* Имя
* Фамилия
* Ссылка на telegram

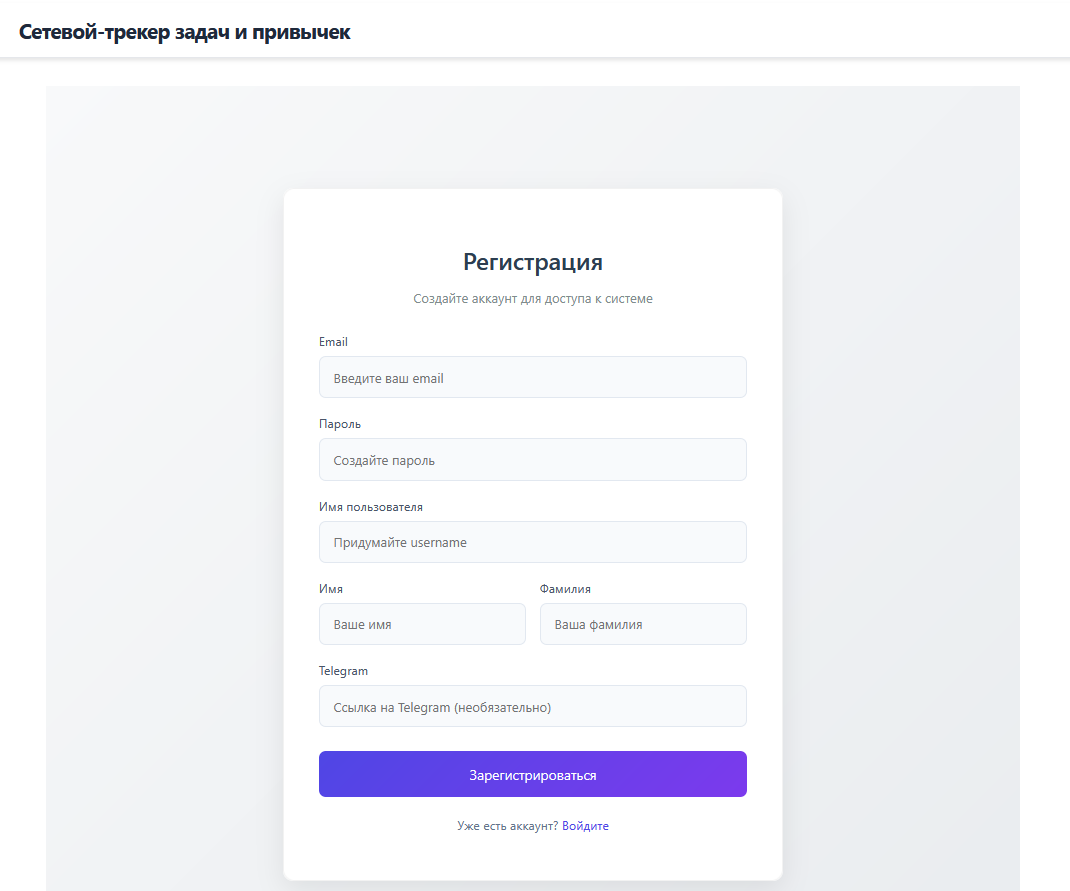


Рисунок 5.2 - Окно регистрации

После успешной аутентификации пользователь попадает в главное окно приложения (Рисунок 5.3), где доступны следующие действия:

* Просмотр проектов: отображение всех текущих проектов в виде карточек. Каждая карточка показывает название проекта, дату его создания и последнего обновления
* Создание и переключение проектов: возможность быстро создать новый проект(Рисунок 5.4) через простую форму или переключиться между уже существующими проектами.
* Фильтрация и поиск:
  + быстрый и расширенный поиск по названию проектов;
* Управление задачами:
  + создание задач с указанием названия, описания, сроков, приоритета и назначения в определённую колонку;
  + редактирование задач, добавление подзадач, комментариев и вложений;
  + удаление задач с подтверждением действия.

Главное окно разделено на две зоны:

* Верхняя область — выбор проектов или привычек, также есть возможность выйти из аккаунта;
* Центральная область — основное рабочее пространство с карточками проектов;

На верхней панели расположены кнопки для перехода между основными окнами приложения.

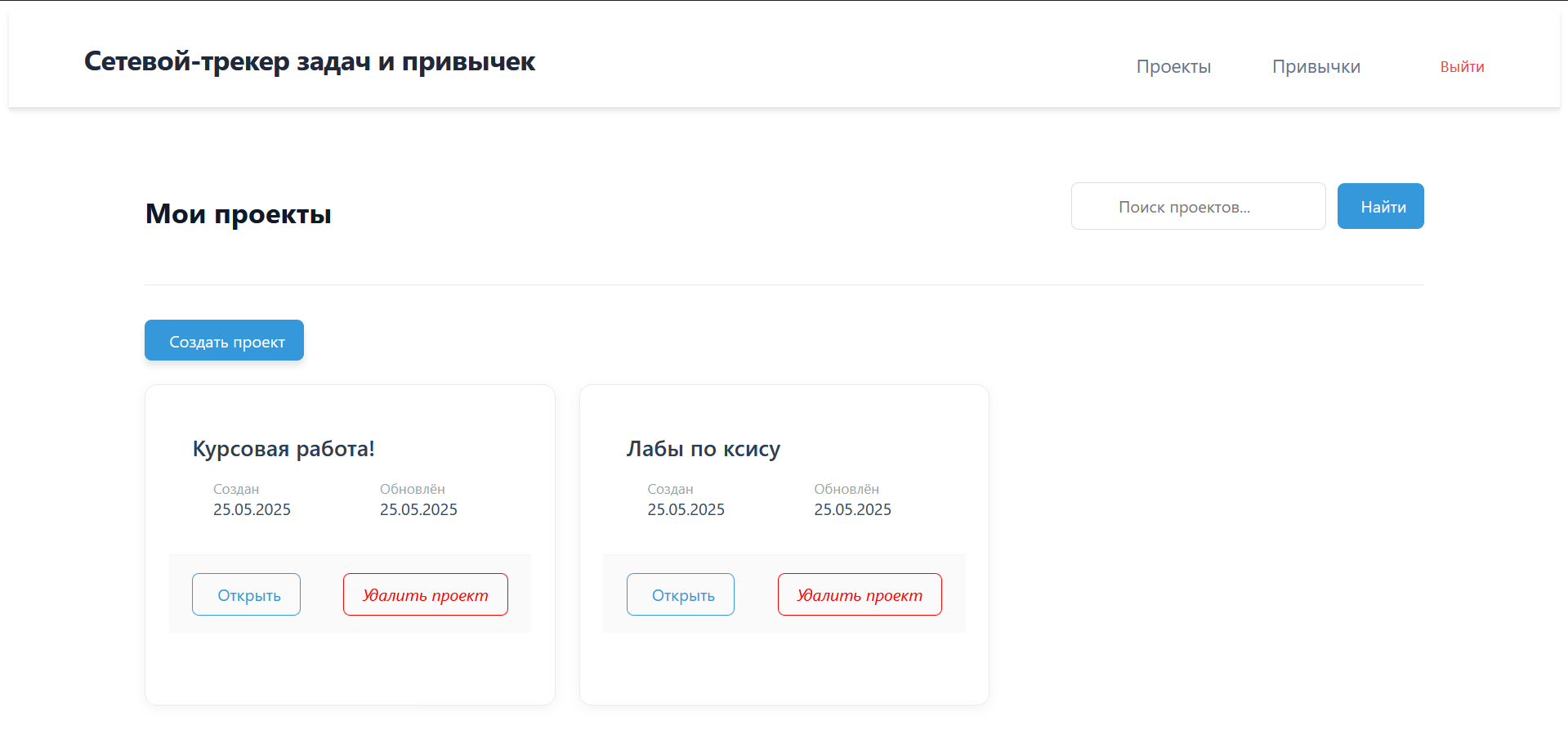
****

Рисунок 5.3 – Главное окно проекта с карточками проектов

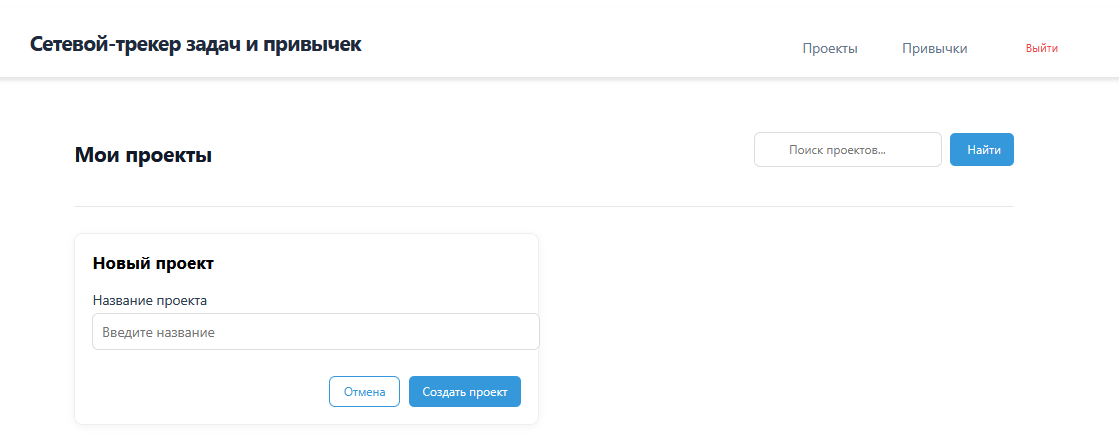
****

Рисунок 5.4 – Окно для создания проектов

При нажатии на карточку проекта открывается детальная информация о проекте с уже существующими столбцами и задачами (Рисунок 5.5). Данное окно позволяет создавать столбцы для проекта, дальше создания столбца можно добавить задачу к этому столбцу. Также можно изменить любую информацию для каждого столбца (Рисунок 5.6), задачи (Рисунок 5.7.) и проекта (Рисунок 5.8).

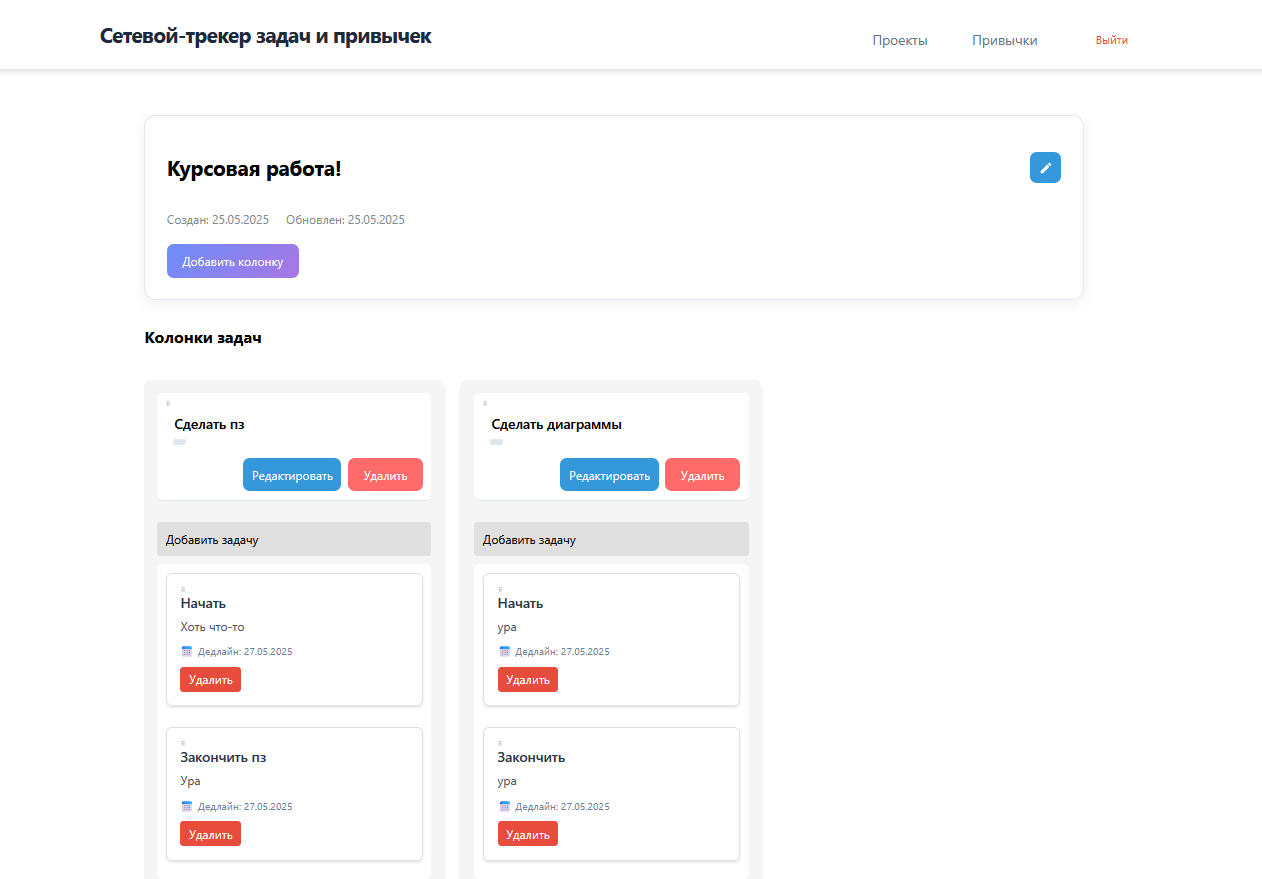


Рисунок 5.5 – Окно с полной информацией о проекте

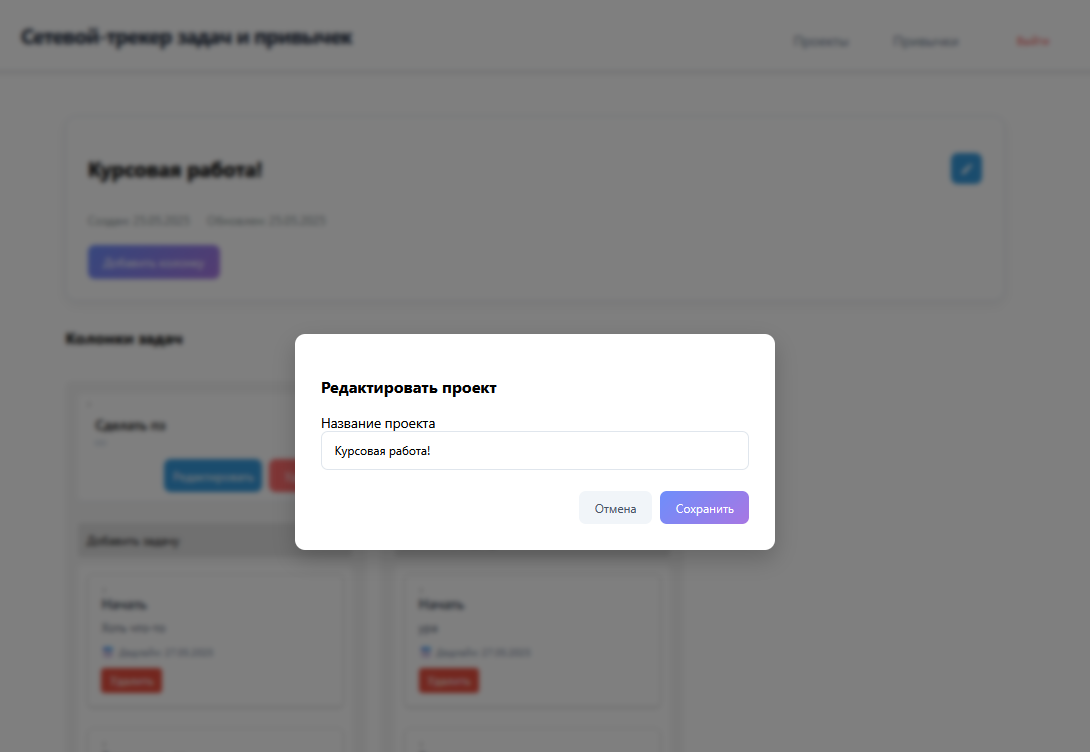


Рисунок 5.6 – Окно редактирования проекта

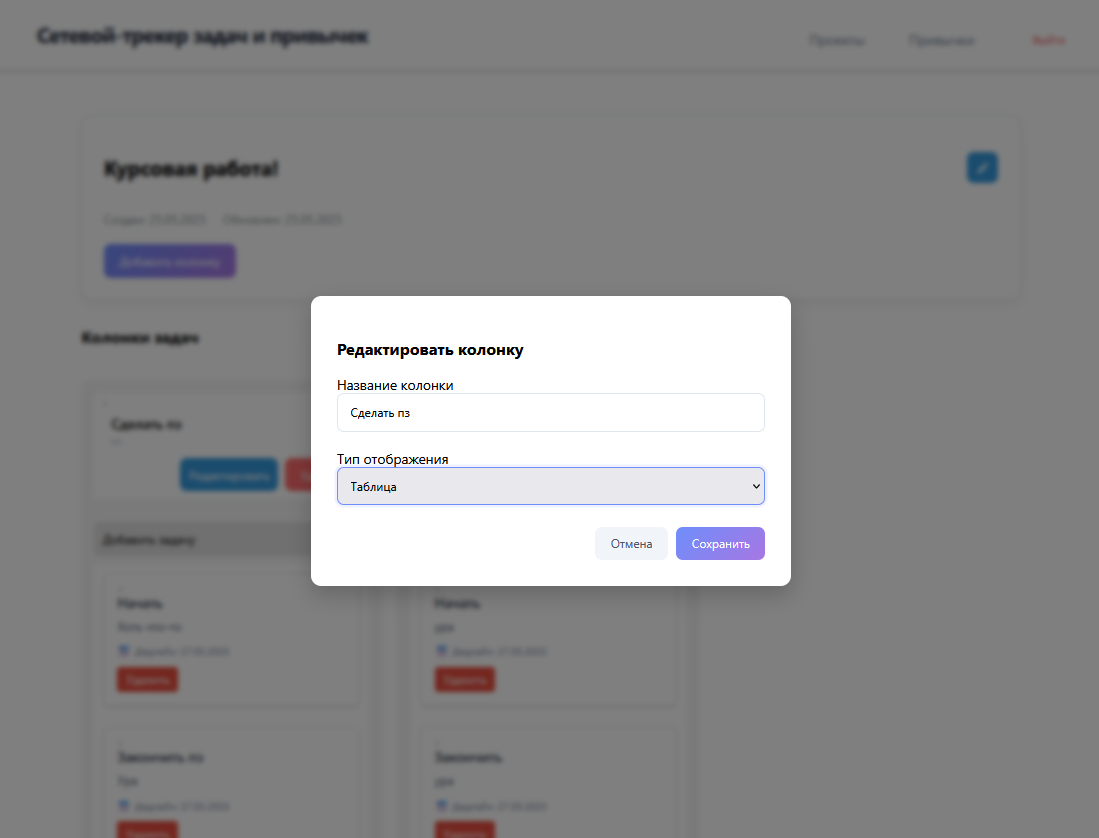


Рисунок 5.7 – Окно редактирования столбца

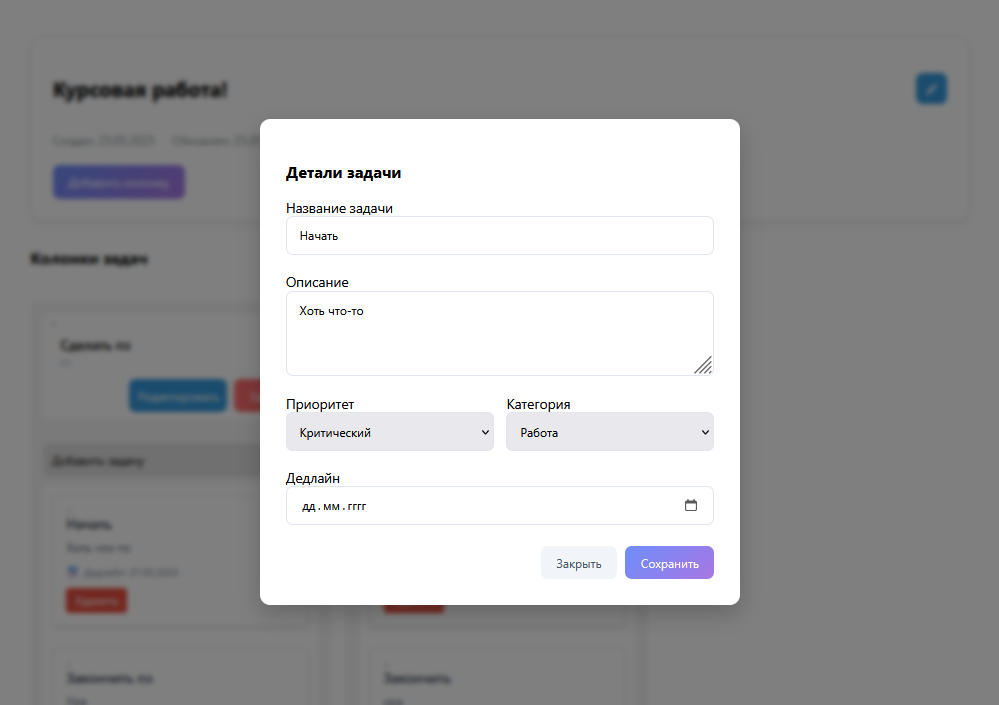


Рисунок 5.8 – Окно редактирования задачи

Если какое-либо поле было заполнено некорректно или оставлено пустым, система предупредит пользователя и предложит внести исправления перед сохранением.

После нажатия на кнопку «Добавить колонку» пользователь переходит в модальное окно для добавления столбца (Рисунок 5.9). Здесь нужно ввести основные поля для столбца:

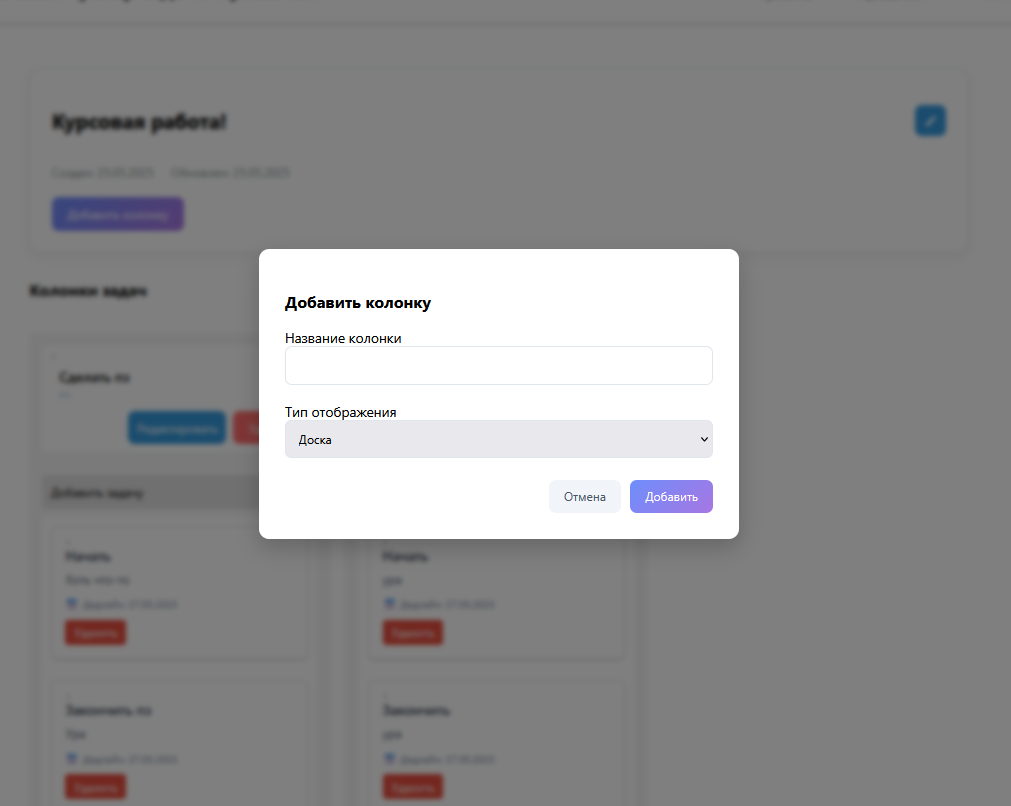


Рисунок 5.9 – Окно создания столбца

После нажатия кнопки «Добавить задачу» пользователь переходит к форме добавления задачи (Рисунок 5.10). Здесь доступны следующие действия:

Заполнение данных для задачи: Пользователь должен заполнить поля:

* Название задачи;
* Описание;
* Приоритет;
* Категория;
* Дедлайн;

Сохранение задачи: после заполнения всех необходимых полей нажмите кнопку «Добавить» для сохранения новой задачи в столбце.

Если введены некорректные данные или поле оставлено пустым, система уведомит пользователя об ошибке и попросит исправить информацию.

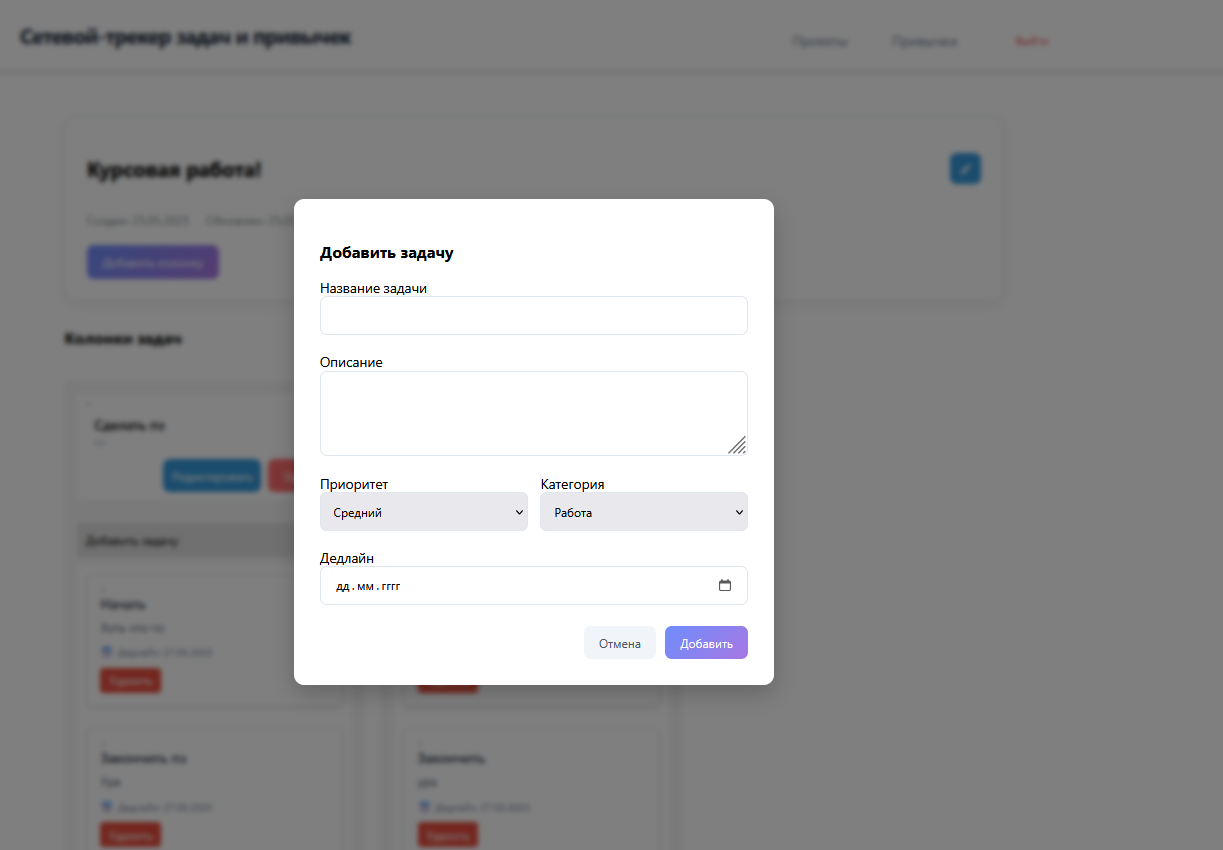
****

Рисунок 5.10 – Окно добавления задачи

После нажатия на кнопку «Привычки» пользователь попадает в главное окно приложения (Рисунок 5.11), где доступны следующие действия:

* Просмотр списка привычек: все активные привычки отображаются с указанием названия, периодичности выполнения и текущего прогресса;
* Создание новой привычки: при нажатии на кнопку "Добавить привычку" открывается форма для ввода основных параметров:
  + Название привычки (обязательное поле)
  + Описание (дополнительные детали)
  + Время выполнения
  + Дата окончания
* Отметка выполнения: для каждой привычки доступна кнопка "Выполнено" для фиксации текущего прогресса для конкретного дня;
* Редактирование привычки: при выборе привычки открывается форма с возможностью изменения всех параметров;
* Удаление привычки: доступно через контекстное меню с обязательным подтверждением;

Все изменения сохраняются автоматически. Для удобства пользователя реализована визуализация прогресса в виде календаря выполнения и графиков, показывающих динамику формирования привычки.

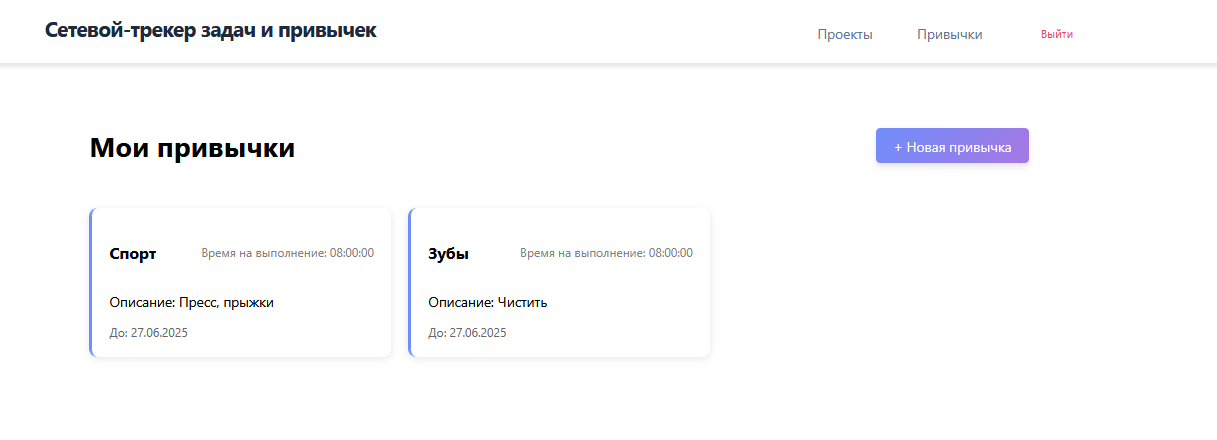


Рисунок 5.10 - Окно со всеми привычками пользователя

После нажатия кнопки "Создать привычку" в основном меню приложения открывается форма добавления новой привычки (Рисунок 5.11). В данном окне пользователь может выполнить следующие действия:

Основные параметры:

* + Ввести название привычки (обязательное поле)
  + Добавить описание (необязательно)
  + Указать время на выполнение
  + Выбрать дату окончания привычки

После заполнения всех необходимых полей пользователь нажимает кнопку "Сохранить", что приводит к:

* Созданию новой привычки в системе
* Появлению визуального подтверждения успешного создания

Если обязательные поля не заполнены или содержат ошибки, система отобразит подсказку с требованием исправить данные перед сохранением.

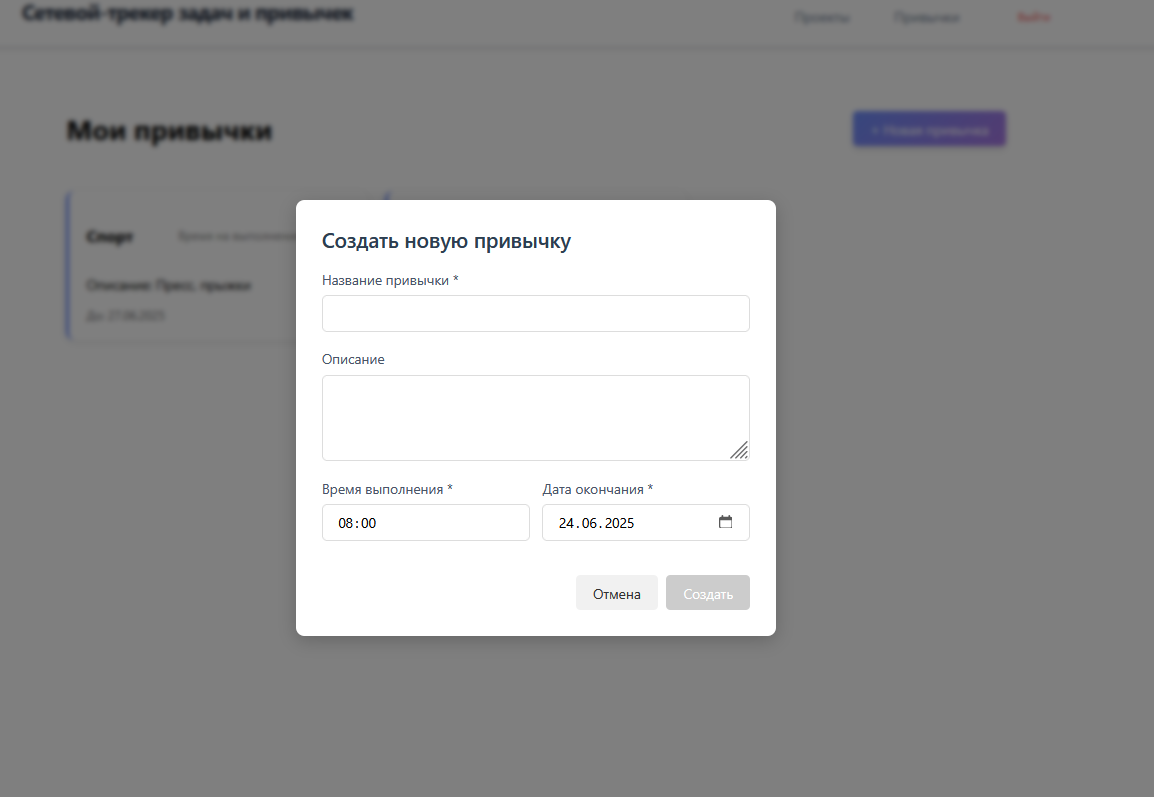
****

Рисунок 5.11 - Окно добавления привычки

После нажатия на конкретную привычку в общем списке открывается детальная страница с полной информацией (Рисунок 5.12). Данный экран состоит из двух основных разделов:

1. Информационная панель:
   * Отображается название привычки и ее описание
   * Указано время выполнения
   * Отображается дата окончания привычки
   * Присутствует кнопка быстрой для любого дня
   * Доступны кнопки редактирования и удаления привычки
2. Интерактивный календарь выполнения:
   * Визуализируется месяц в виде таблицы с днями
   * Отмеченные дни (когда привычка была выполнена) выделены зеленым цветом
   * Кнопки навигации позволяют переключаться между месяцами

При нажатии на кнопку "Отметить сегодня" система автоматически фиксирует выполнение привычки на текущую дату и обновляет все показатели в реальном времени.

Все изменения сохраняются автоматически. Пользователь может в любой момент вернуться к списку привычек с помощью кнопки "Назад" в верхнем левом углу экрана.

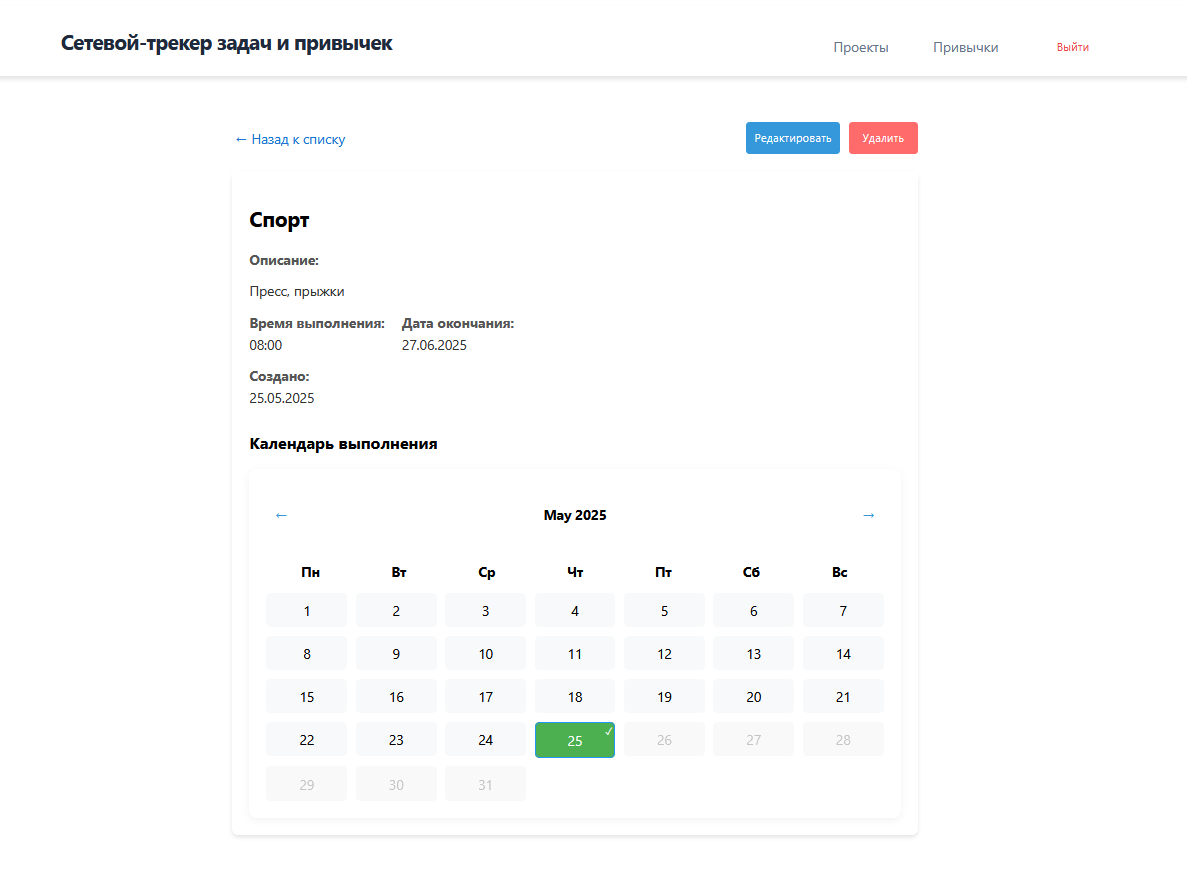
****

Рисунок 5.12 – Окно с детально информацией привычки

После нажатия на кнопку "Редактировать" в детальном просмотре привычки открывается форма изменения параметров (Рисунок 5.13). В данном окне пользователь может:

Изменить основные параметры:

* + Отредактировать название привычки (текстовое поле)
  + Обновить описание (многострочное текстовое поле)
  + Выбрать новое время для выполнения
  + Указать новую дату окончания

Все изменения синхронизируются между устройствами пользователя в реальном времени. После сохранения система показывает краткое уведомление об успешном обновлении данных и возвращает пользователя на детальную страницу привычки с актуальной информацией.

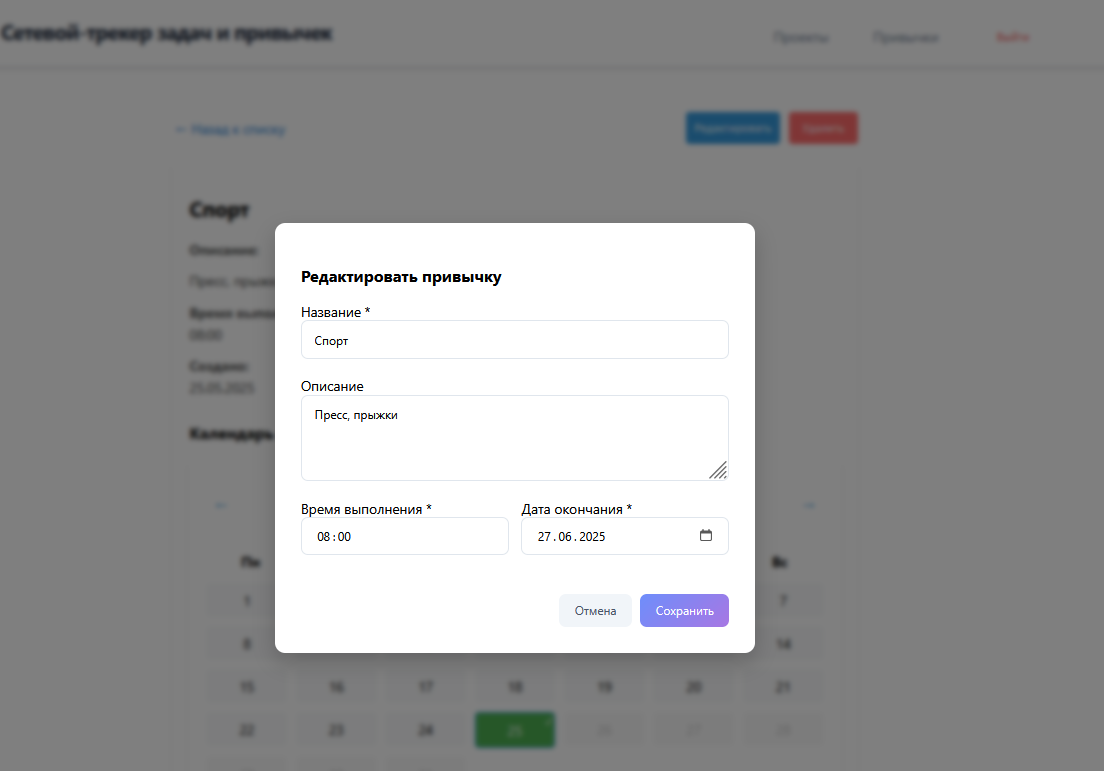
****

Рисунок 5.13 – Окно редактирования привычки

**Список использованных источников**

1. Стандарт предприятия [Электронный ресурс]. – Режим доступа: СТП\_2024.pdf.
2. Фаулер М. Рефакторинг: улучшение существующего кода. — СПб.: Питер, 2023. — 448 с.
3. Microsoft Documentation. Официальная документация по Spring Boot и React [Электронный ресурс]. — URL: <https://docs.microsoft.com/ru-ru/> (дата обращения: 15.05.2024).
4. Spring Framework. Официальная документация [Электронный ресурс]. — URL: <https://spring.io/docs> (дата обращения: 15.05.2024).
5. React Documentation. Руководство по React.js [Электронный ресурс]. — URL: <https://reactjs.org/docs/getting-started.html> (дата обращения: 15.05.2024).
6. Харматюк В.В. Микросервисная архитектура на Java. — М.: ДМК Пресс, 2022. — 356 с.
7. Richards M. Fundamentals of Software Architecture. — O'Reilly, 2020. — 410 p.
8. Docker Documentation. Официальное руководство [Электронный ресурс]. — URL: <https://docs.docker.com/> (дата обращения: 15.05.2024).
9. PostgreSQL Documentation. Руководство администратора [Электронный ресурс]. — URL: <https://www.postgresql.org/docs/> (дата обращения: 15.05.2024).
10. GitHub. Репозиторий примеров микросервисных приложений [Электронный ресурс]. — URL: <https://github.com/topics/microservices> (дата обращения: 15.05.2024).
11. Stack Overflow. Вопросы и решения по Spring Boot и React [Электронный ресурс]. — URL: <https://stackoverflow.com/> (дата обращения: 15.05.2024).
12. Habr. Статьи по разработке трекеров привычек [Электронный ресурс]. — URL: [https://habr.com/ru/search/?q=трекер%20привычек](https://habr.com/ru/search/?q=%D1%82%D1%80%D0%B5%D0%BA%D0%B5%D1%80%2520%D0%BF%D1%80%D0%B8%D0%B2%D1%8B%D1%87%D0%B5%D0%BA) (дата обращения: 15.05.2024).
13. Medium. Best Practices for Habit Tracking Apps [Электронный ресурс]. — URL: <https://medium.com/tag/habit-tracking> (дата обращения: 15.05.2024).
14. Белов А.А. Современные веб-технологии. — М.: БИНОМ, 2021. — 288 с.

**ПРИЛОЖЕНИЕ А**

(обязательное)

Листинг кода

import {Injectable} from '@angular/core';  
import {HttpClient} from '@angular/common/http';  
import {catchError, Observable, of} from 'rxjs';  
import {Project} from '../../models/project.model';  
import {Layouts} from '../../models/enums/layouts.enum';  
import {TaskState} from '../../models/task-state.model';  
import {Task} from '../../models/task.model';  
import {Priority} from '../../models/enums/priority.enum';  
import {Category} from '../../models/enums/category.enum';  
  
@Injectable({  
 providedIn: 'root'  
})  
export class ProjectService {  
 private gatewayUrl = 'http://localhost:3246';  
  
 constructor(private http: HttpClient) {}  
  
 // Project methods  
 getProjects(): Observable<Project[]> {  
 return this.http.get<Project[]>(`${this.gatewayUrl}/api/projects`);  
 }  
  
 getProjectById(id: number): Observable<Project> {  
 return this.http.get<Project>(`${this.gatewayUrl}/api/projects/${id}`);  
 }  
  
 searchProjects(prefix: string): Observable<Project[]> {  
 return this.http.get<Project[]>(`${this.gatewayUrl}/api/projects`, {  
 params: { prefix\_name: prefix }  
 }).pipe(  
 catchError(error => {  
 console.error('Search error:', error);  
 return of([]);  
 })  
 );  
 }  
  
 createOrUpdateProject(id: number | null, name: string): Observable<Project> {  
 const params: any = { project\_name: name };  
 if (id) {  
 params.project\_id = id.toString();  
 }  
 return this.http.put<Project>(`${this.gatewayUrl}/api/projects`, null, { params });  
 }  
  
 deleteProject(id: number): Observable<any> {  
 return this.http.delete(`${this.gatewayUrl}/api/projects/${id}`);  
 }  
  
 getTaskStates(projectId: number): Observable<TaskState[]> {  
 return this.http.get<TaskState[]>(  
 `${this.gatewayUrl}/api/projects/${projectId}/tasks-states`  
 ).pipe(  
 catchError(error => {  
 console.error('Error loading task states:', error);  
 return of([]);  
 })  
 );  
 }  
  
 createTaskState(projectId: number, name: string, layout: Layouts): Observable<TaskState> {  
 return this.http.post<TaskState>(  
 `${this.gatewayUrl}/api/projects/${projectId}/tasks-states`,  
 null,  
 {  
 params : {  
 task\_state\_name: name,  
 type\_of\_layout: layout  
 }  
 }  
 ).pipe(  
 catchError(error => {  
 console.error('Error creating task state:', error);  
 throw error;  
 })  
 );  
 }  
  
 updateTaskState(taskStateId: number, name: string, layout: Layouts | string): Observable<TaskState> {  
 // Convert layout to string if it's an enum value  
 const layoutString = typeof layout === 'string' ? layout : Layouts[layout];  
  
 return this.http.patch<TaskState>(  
 `${this.gatewayUrl}/api/tasks-states/${taskStateId}`,  
 null,  
 {  
 params: {  
 task\_state\_name: name,  
 type\_of\_layout: layoutString  
 }  
 }  
 ).pipe(  
 catchError(error => {  
 console.error('Error updating task state:', error);  
 throw error;  
 })  
 );  
 }  
  
 changeTaskStatePosition(taskStateId: number, rightTaskStateId?: number): Observable<TaskState> {  
 const params: any = {};  
 if (rightTaskStateId) {  
 params.right\_task\_state\_id = rightTaskStateId;  
 }  
 return this.http.patch<TaskState>(  
 `${this.gatewayUrl}/api/tasks-states/${taskStateId}/position/change`,  
 null,  
 { params }  
 );  
 }  
  
 deleteTaskState(taskStateId: number): Observable<any> {  
 return this.http.delete(`${this.gatewayUrl}/api/tasks-states/${taskStateId}`);  
 }  
  
 getTasks(taskStateId: number): Observable<Task[]> {  
 return this.http.get<Task[]>(`${this.gatewayUrl}/api/task-state/${taskStateId}/tasks`);  
 }  
  
 createTask(  
 taskStateId: number,  
 title: string,  
 description: string,  
 deadline: string | Date,  
 category: Category,  
 priority: Priority  
 ): Observable<Task> {  
 // Форматируем дату в ISO 8601 без Z и без миллисекунд (LocalDateTime формат)  
 const formatDeadline = (d: Date): string => {  
 const pad = (n: number): string => n.toString().padStart(2, '0');  
  
 return `${d.getFullYear()}-${pad(d.getMonth() + 1)}-${pad(d.getDate())}` +  
 `T${pad(d.getHours())}:${pad(d.getMinutes())}:${pad(d.getSeconds())}`;  
 };  
  
 const deadlineStr = deadline instanceof Date ? formatDeadline(deadline) : deadline;  
  
 return this.http.post<Task>(  
 `${this.gatewayUrl}/api/task-state/${taskStateId}/tasks`,  
 null,  
 {  
 params: {  
 title,  
 description,  
 deadline: deadlineStr,  
 category: category.toString(),  
 priority: priority.toString()  
 }  
 }  
 );  
 }  
  
 updateTask(  
 taskId: number,  
 title: string,  
 description: string,  
 deadline: Date,  
 category: Category,  
 priority: Priority  
 ): Observable<Task> {  
 const formatDeadline = (d: Date): string => {  
 const pad = (n: number) => n.toString().padStart(2, '0');  
 return `${d.getFullYear()}-${pad(d.getMonth() + 1)}-${pad(d.getDate())}T${pad(d.getHours())}:${pad(d.getMinutes())}:${pad(d.getSeconds())}`;  
 };  
  
 const formattedDeadline = formatDeadline(deadline);  
  
 return this.http.patch<Task>(  
 `${this.gatewayUrl}/api/tasks/${taskId}`,  
 null,  
 {  
 params: {  
 title: title.trim(),  
 description: description || '',  
 deadline: formattedDeadline,  
 category: category.toString(),  
 priority: priority.toString()  
 }  
 }  
 ).pipe(  
 catchError(error => {  
 console.error('Error updating task:', error);  
 throw error;  
 })  
 );  
 }  
  
 changeTaskPosition(taskId: number, lowerTaskId?: number): Observable<Task> {  
 const params: any = {};  
 if (lowerTaskId) {  
 params.lower\_task\_id = lowerTaskId;  
 }  
 return this.http.patch<Task>(  
 `${this.gatewayUrl}/api/tasks/${taskId}/position/change`,  
 null,  
 { params }  
 );  
 }  
  
 deleteTask(taskId: number): Observable<any> {  
 return this.http.delete(`${this.gatewayUrl}/api/tasks/${taskId}`);  
 }  
}

package org.myProject.focus\_flow\_gateway\_api.api.controllers.helpers.util;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import org.myProject.focus\_flow\_gateway\_api.api.exceptions.CustomAppException;  
import org.springframework.http.HttpStatus;  
  
import java.io.\*;  
import java.net.HttpURLConnection;  
import java.net.URL;  
import java.nio.charset.StandardCharsets;  
import java.util.\*;  
import java.util.stream.Collectors;  
  
public class HttpHelper {  
  
 public static HttpResponse sendHttpRequest(String urlString, String method, String requestBody,  
 String contentType, String authToken) throws Exception {  
 URL url = new URL(urlString);  
 HttpURLConnection connection = (HttpURLConnection) url.openConnection();  
  
 try {  
 connection.setRequestMethod(method);  
 connection.setRequestProperty("Content-Type", contentType);  
  
 if (authToken != null) {  
 connection.setRequestProperty("Authorization", authToken.startsWith("Bearer ")  
 ? authToken  
 : "Bearer " + authToken);  
 }  
  
 if (requestBody != null && (method.equals("POST") || method.equals("PUT"))) {  
 connection.setDoOutput(true);  
 try (OutputStream os = connection.getOutputStream()) {  
 byte[] input = requestBody.getBytes(StandardCharsets.UTF\_8);  
 os.write(input, 0, input.length);  
 }  
 }  
  
 int responseCode = connection.getResponseCode();  
 String responseBody;  
  
 try (InputStream inputStream = responseCode >= 400  
 ? connection.getErrorStream()  
 : connection.getInputStream()) {  
  
 if (inputStream == null) {  
 responseBody = "";  
 } else {  
 responseBody = new BufferedReader(  
 new InputStreamReader(inputStream, StandardCharsets.UTF\_8))  
 .lines()  
 .collect(Collectors.joining("\n"));  
 }  
 }  
  
 return new HttpResponse(  
 responseBody,  
 connection.getHeaderFields(),  
 responseCode  
 );  
  
 } finally {  
 connection.disconnect();  
 }  
 }  
  
 private static String readErrorResponse(HttpURLConnection connection) throws IOException {  
 try (BufferedReader br = new BufferedReader(  
 new InputStreamReader(connection.getErrorStream(), StandardCharsets.UTF\_8))) {  
 StringBuilder response = new StringBuilder();  
 String responseLine;  
 while ((responseLine = br.readLine()) != null) {  
 response.append(responseLine.trim());  
 }  
 return response.toString();  
 }  
 }  
}

package org.myProject.focus\_flow\_gateway\_api.api.controllers.helpers.util;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import java.util.List;  
import java.util.Map;  
  
@Data  
@AllArgsConstructor  
public class HttpResponse {  
 String body;  
 Map<String, List<String>> headers;  
 int statusCode;  
  
 public String getBody() {  
 return body;  
 }  
  
 public Map<String, List<String>> getHeaders() {  
 return headers;  
 }  
  
 public int getStatusCode() {  
 return statusCode;  
 }  
}

package org.myProject.focus\_flow\_gateway\_api.api.controllers.helpers;  
  
import com.fasterxml.jackson.core.type.TypeReference;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.nimbusds.jwt.SignedJWT;  
import lombok.AccessLevel;  
import lombok.RequiredArgsConstructor;  
import lombok.experimental.FieldDefaults;  
import lombok.extern.slf4j.Slf4j;  
import org.keycloak.representations.idm.\*;  
import org.myProject.focus\_flow\_gateway\_api.api.controllers.helpers.util.HttpHelper;  
import org.myProject.focus\_flow\_gateway\_api.api.controllers.helpers.util.HttpResponse;  
import org.myProject.focus\_flow\_gateway\_api.api.dto.UserRequestDto;  
import org.myProject.focus\_flow\_gateway\_api.api.dto.enums.Role;  
import org.myProject.focus\_flow\_gateway\_api.api.exceptions.CustomAppException;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.http.HttpStatus;  
import org.springframework.stereotype.Component;  
  
import javax.annotation.PostConstruct;  
import java.io.\*;  
import java.net.HttpURLConnection;  
import java.net.URL;  
import java.nio.charset.StandardCharsets;  
import java.util.\*;  
import java.util.stream.Collectors;  
  
@Slf4j  
@Component  
@FieldDefaults(level = AccessLevel.PRIVATE)  
@RequiredArgsConstructor  
public class UserAuthHelper {  
  
 @Value("${KEYCLOAK\_URL}")  
 String keycloakUrl;  
  
 @Value("${KEYCLOAK\_REALM}")  
 String realm;  
  
 @Value("${KEYCLOAK\_CLIENT\_ID}")  
 String clientId;  
  
 @Value("${KEYCLOAK\_CLIENT\_SECRET}")  
 String clientSecret;  
  
 private final ObjectMapper objectMapper;  
 private String adminToken;  
  
 @PostConstruct  
 public void init() {  
 try {  
 this.adminToken = getAdminToken();  
 log.info("Admin token successfully obtained");  
 } catch (Exception e) {  
 log.error("Critical error during admin token initialization", e);  
 throw new RuntimeException("Application startup failed due to authentication issues");  
 }  
 }  
  
 private String getAdminToken() {  
 try {  
 // Исправленный URL для текущего realm  
 String tokenUrl = keycloakUrl + "/realms/" + realm + "/protocol/openid-connect/token";  
  
 // Используем client\_id из конфигурации приложения  
 String requestBody = "grant\_type=client\_credentials" +  
 "&client\_id=" + clientId +  
 "&client\_secret=" + clientSecret;  
  
 HttpResponse response = HttpHelper.sendHttpRequest(tokenUrl, "POST", requestBody,  
 "application/x-www-form-urlencoded", null);  
  
 Map<String, Object> tokenData = objectMapper.readValue(response.getBody(), new TypeReference<>() {});  
 return (String) tokenData.get("access\_token");  
  
 } catch (Exception e) {  
 log.error("Failed to get admin token", e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR, "Failed to initialize admin access");  
 }  
 }  
  
 public String registerUser(UserRequestDto requestDto) {  
 log.info("Registering user: {}", requestDto.getUsername());  
 checkIfUserExists(requestDto);  
  
 String usersUrl = keycloakUrl + "/admin/realms/" + realm + "/users";  
 UserRepresentation user = createUserRepresentation(requestDto);  
  
 try {  
 HttpResponse response = HttpHelper.sendHttpRequest(usersUrl, "POST", objectMapper.writeValueAsString(user),  
 "application/json", adminToken);  
  
 List<String> locationList = response.getHeaders().get("Location");  
 if (locationList == null || locationList.isEmpty()) {  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR, "User created but no location header returned");  
 }  
  
 String location = locationList.get(0);  
 String userId = location.substring(location.lastIndexOf('/') + 1);  
  
 Thread.sleep(500); // небольшая задержка перед назначением роли  
 assignRolesToUser(userId);  
 return userId;  
  
 } catch (Exception e) {  
 log.error("Failed to register user", e);  
 throw new CustomAppException(HttpStatus.BAD\_REQUEST, "Failed to register user: " + e.getMessage());  
 }  
 }  
  
 private UserRepresentation createUserRepresentation(UserRequestDto requestDto) {  
 UserRepresentation user = new UserRepresentation();  
 user.setUsername(requestDto.getUsername());  
 user.setEmail(requestDto.getEmail());  
 user.setFirstName(requestDto.getFirstName());  
 user.setLastName(requestDto.getLastName());  
 user.setEnabled(true);  
  
 CredentialRepresentation credential = new CredentialRepresentation();  
 credential.setType(CredentialRepresentation.PASSWORD);  
 credential.setValue(requestDto.getPassword());  
 credential.setTemporary(false);  
 user.setCredentials(List.of(credential));  
  
 Map<String, List<String>> attributes = new HashMap<>();  
 attributes.put("telegramLink", List.of(requestDto.getTelegramLink()));  
  
 if (requestDto.getRegistrationDate() != null) {  
 attributes.put("registrationDate", List.of(requestDto.getRegistrationDate().toString()));  
 }  
  
 if (requestDto.getStatus() != null) {  
 attributes.put("status", List.of(String.valueOf(requestDto.getStatus())));  
 }  
  
 user.setAttributes(attributes);  
 return user;  
 }  
  
 private void checkIfUserExists(UserRequestDto requestDto) {  
 String searchUrl = keycloakUrl + "/admin/realms/" + realm + "/users?username=" +  
 requestDto.getUsername() + "&exact=true";  
  
 try {  
 List<UserRepresentation> users = sendHttpRequest(searchUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<UserRepresentation>>() {});  
  
 if (!users.isEmpty()) {  
 throw new CustomAppException(HttpStatus.BAD\_REQUEST,  
 "User with this username already exists");  
 }  
  
 // Проверка по email  
 searchUrl = keycloakUrl + "/admin/realms/" + realm + "/users?email=" + requestDto.getEmail();  
 users = sendHttpRequest(searchUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<UserRepresentation>>() {});  
  
 if (!users.isEmpty()) {  
 throw new CustomAppException(HttpStatus.BAD\_REQUEST,  
 "User with this email already exists");  
 }  
 } catch (CustomAppException e) {  
 throw e;  
 } catch (Exception e) {  
 log.error("Error checking user existence", e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Error checking user existence: " + e.getMessage());  
 }  
 }  
  
 private void assignRolesToUser(String userId) {  
 try {  
 // First check if the role exists  
 String rolesUrl = keycloakUrl + "/admin/realms/" + realm + "/roles";  
 List<RoleRepresentation> availableRoles = sendHttpRequest(rolesUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<RoleRepresentation>>() {});  
  
 Optional<RoleRepresentation> userRole = availableRoles.stream()  
 .filter(r -> r.getName().equals(Role.USER.name()))  
 .findFirst();  
  
 if (userRole.isEmpty()) {  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Role " + Role.USER.name() + " not found in Keycloak");  
 }  
  
 // Assign the role to user  
 String userRolesUrl = keycloakUrl + "/admin/realms/" + realm + "/users/" + userId + "/role-mappings/realm";  
  
 HttpResponse response = HttpHelper.sendHttpRequest(  
 userRolesUrl,  
 "POST",  
 objectMapper.writeValueAsString(List.of(userRole.get())),  
 "application/json",  
 adminToken  
 );  
  
 if (response.getStatusCode() >= 400) {  
 throw new CustomAppException(HttpStatus.valueOf(response.getStatusCode()),  
 "Failed to assign role: " + response.getBody());  
 }  
  
 log.info("Role {} successfully assigned to user {}", Role.USER, userId);  
 } catch (Exception e) {  
 log.error("Error assigning role {} to user {}: {}", Role.USER.name(), userId, e.getMessage(), e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Error assigning role: " + e.getMessage());  
 }  
 }  
  
 public Map<String, Object> authenticate(String username, String password) {  
 String tokenUrl = keycloakUrl + "/realms/" + realm + "/protocol/openid-connect/token";  
 String requestBody = "client\_id=" + clientId +  
 "&client\_secret=" + clientSecret +  
 "&grant\_type=password" +  
 "&username=" + username +  
 "&password=" + password;  
  
 try {  
 Map<String, Object> response = sendHttpRequest(tokenUrl, "POST", requestBody,  
 "application/x-www-form-urlencoded", null);  
 return extractTokenData(response);  
 } catch (Exception e) {  
 log.error("Authentication failed", e);  
 throw new CustomAppException(HttpStatus.UNAUTHORIZED, "Authentication failed: " + e.getMessage());  
 }  
 }  
  
 public Map<String, Object> refreshToken(String refreshToken) {  
 String tokenUrl = keycloakUrl + "/realms/" + realm + "/protocol/openid-connect/token";  
 String requestBody = "client\_id=" + clientId +  
 "&client\_secret=" + clientSecret +  
 "&grant\_type=refresh\_token" +  
 "&refresh\_token=" + refreshToken;  
  
 try {  
 Map<String, Object> response = sendHttpRequest(tokenUrl, "POST", requestBody,  
 "application/x-www-form-urlencoded", null);  
 return extractTokenData(response);  
 } catch (Exception e) {  
 log.error("Token refresh failed", e);  
 throw new CustomAppException(HttpStatus.UNAUTHORIZED, "Token refresh failed: " + e.getMessage());  
 }  
 }  
  
 private Map<String, Object> extractTokenData(Map<String, Object> response) {  
 Map<String, Object> result = new HashMap<>();  
 result.put("access\_token", response.get("access\_token"));  
 result.put("expires\_in", response.get("expires\_in"));  
 result.put("refresh\_expires\_in", response.get("refresh\_expires\_in"));  
 result.put("refresh\_token", response.get("refresh\_token"));  
 return result;  
 }  
  
 public void updateUser(String userId, String email, String password, String username,  
 String firstName, String lastName, String telegramLink,  
 String status, String token, boolean isAdminUpdate) {  
  
 // Проверяем права доступа  
 String currentUserId = getUserIdFromToken(token);  
  
 if (!isAdminUpdate && !currentUserId.equals(userId)) {  
 throw new CustomAppException(HttpStatus.FORBIDDEN, "You can only update your own profile");  
 }  
  
 if (isAdminUpdate && !isAdmin(token)) {  
 throw new CustomAppException(HttpStatus.FORBIDDEN, "Admin privileges required");  
 }  
  
 try {  
 // Получаем текущие данные пользователя  
 String userUrl = keycloakUrl + "/admin/realms/" + realm + "/users/" + userId;  
 UserRepresentation user = sendHttpRequest(userUrl, "GET", null,  
 "application/json", adminToken, UserRepresentation.class);  
  
 if (user == null) {  
 throw new CustomAppException(HttpStatus.NOT\_FOUND, "User not found");  
 }  
  
 // Обновляем поля  
 if (email != null) user.setEmail(email);  
 if (username != null) user.setUsername(username);  
 if (firstName != null) user.setFirstName(firstName);  
 if (lastName != null) user.setLastName(lastName);  
  
 // Обновляем атрибуты  
 Map<String, List<String>> attributes = Optional.ofNullable(user.getAttributes())  
 .orElse(new HashMap<>());  
  
 if (telegramLink != null) {  
 attributes.put("telegramLink", Collections.singletonList(telegramLink));  
 }  
  
 // Только админ может обновлять эти поля  
 if (isAdminUpdate && status != null) {  
 attributes.put("status", Collections.singletonList(status));  
 }  
  
 user.setAttributes(attributes);  
  
 // Обновление пароля  
 if (password != null) {  
 CredentialRepresentation credential = new CredentialRepresentation();  
 credential.setType(CredentialRepresentation.PASSWORD);  
 credential.setValue(password);  
 credential.setTemporary(false);  
 user.setCredentials(Collections.singletonList(credential));  
 }  
  
 // Отправляем обновленные данные  
 sendHttpRequest(userUrl, "PUT", objectMapper.writeValueAsString(user),  
 "application/json", adminToken);  
  
 log.info("User {} updated by {}", userId, isAdminUpdate ? "admin" : "user");  
 } catch (Exception e) {  
 log.error("Error updating user", e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Error updating user: " + e.getMessage());  
 }  
 }  
  
 public String getCurrentUserId(String token) {  
 return getUserIdFromToken(token);  
 }  
  
 public void logout(String authHeader) {  
 String accessToken = extractAccessToken(authHeader);  
 String logoutUrl = keycloakUrl + "/realms/" + realm + "/protocol/openid-connect/logout";  
  
 String requestBody = "client\_id=" + clientId + "&client\_secret=" + clientSecret;  
  
 try {  
 sendHttpRequest(logoutUrl, "POST", requestBody,  
 "application/x-www-form-urlencoded", accessToken);  
 } catch (Exception e) {  
 log.error("Logout failed", e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Logout failed: " + e.getMessage());  
 }  
 }  
  
 private String extractAccessToken(String authHeader) {  
 if (authHeader == null || !authHeader.startsWith("Bearer ")) {  
 throw new CustomAppException(HttpStatus.BAD\_REQUEST, "Invalid authorization header");  
 }  
 return authHeader.substring(7);  
 }  
  
 public UserRepresentation getUserById(String userId, String token) {  
 validateUserAccess(userId, token);  
  
 try {  
 String userUrl = keycloakUrl + "/admin/realms/" + realm + "/users/" + userId;  
 return sendHttpRequest(userUrl, "GET", null,  
 "application/json", adminToken, UserRepresentation.class);  
 } catch (Exception e) {  
 log.error("Error getting user by id", e);  
 throw new CustomAppException(HttpStatus.NOT\_FOUND, "User not found");  
 }  
 }  
  
 public List<UserRepresentation> getAllUsers(String token) {  
 validateAdminAccess(token);  
  
 try {  
 String usersUrl = keycloakUrl + "/admin/realms/" + realm + "/users";  
 return sendHttpRequest(usersUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<UserRepresentation>>() {});  
 } catch (Exception e) {  
 log.error("Error getting all users", e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Error getting users: " + e.getMessage());  
 }  
 }  
  
 private void validateAdminAccess(String token) {  
 if (!isAdmin(token)) {  
 throw new CustomAppException(HttpStatus.FORBIDDEN, "Admin access required");  
 }  
 }  
  
 private void validateUserAccess(String requestedUserId, String token) {  
 if (!isAdmin(token) && !getUserIdFromToken(token).equals(requestedUserId)) {  
 throw new CustomAppException(HttpStatus.FORBIDDEN, "Access denied");  
 }  
 }  
  
 private boolean isAdmin(String token) {  
 String userId = getUserIdFromToken(token);  
  
 try {  
 // Проверяем realm-level роли  
 String realmRolesUrl = keycloakUrl + "/admin/realms/" + realm + "/users/" + userId + "/role-mappings/realm";  
 List<RoleRepresentation> realmRoles = sendHttpRequest(realmRolesUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<RoleRepresentation>>() {});  
  
 boolean hasRealmAdmin = realmRoles.stream()  
 .anyMatch(r -> r.getName().equals(Role.ADMIN.name()));  
  
 // Проверяем client-level роли  
 String clientsUrl = keycloakUrl + "/admin/realms/" + realm + "/clients?clientId=" + clientId;  
 List<ClientRepresentation> clients = sendHttpRequest(clientsUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<ClientRepresentation>>() {});  
  
 if (clients.isEmpty()) {  
 log.error("Client {} not found in Keycloak", clientId);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR, "Client not found");  
 }  
  
 String clientUuid = clients.get(0).getId();  
 String clientRolesUrl = keycloakUrl + "/admin/realms/" + realm + "/users/" + userId +  
 "/role-mappings/clients/" + clientUuid;  
  
 List<RoleRepresentation> clientRoles = sendHttpRequest(clientRolesUrl, "GET", null,  
 "application/json", adminToken, new TypeReference<List<RoleRepresentation>>() {});  
  
 boolean hasClientAdmin = clientRoles.stream()  
 .anyMatch(r -> r.getName().equals(Role.ADMIN.name()));  
  
 return hasRealmAdmin || hasClientAdmin;  
 } catch (Exception e) {  
 log.error("Error checking admin privileges", e);  
 throw new CustomAppException(HttpStatus.INTERNAL\_SERVER\_ERROR,  
 "Error checking admin privileges: " + e.getMessage());  
 }  
 }  
  
 public String getUserIdFromToken(String token) {  
 if (token == null || !token.startsWith("Bearer ")) {  
 throw new CustomAppException(HttpStatus.UNAUTHORIZED, "Invalid authorization header format");  
 }  
  
 String jwtToken = token.substring(7).trim();  
  
 try {  
 SignedJWT signedJWT = SignedJWT.parse(jwtToken);  
 return signedJWT.getJWTClaimsSet().getSubject(); // "sub"  
 } catch (Exception e) {  
 log.error("Failed to parse JWT", e);  
 throw new CustomAppException(HttpStatus.UNAUTHORIZED, "Invalid JWT token");  
 }  
 }  
  
 // Общий метод для отправки HTTP запросов  
 private <T> T sendHttpRequest(String urlString, String method, String requestBody,  
 String contentType, String authToken, TypeReference<T> typeRef) throws Exception {  
 HttpResponse response = HttpHelper.sendHttpRequest(urlString, method, requestBody, contentType, authToken);  
 return objectMapper.readValue(response.getBody(), typeRef);  
 }  
  
 private <T> T sendHttpRequest(String urlString, String method, String requestBody,  
 String contentType, String authToken, Class<T> responseType) throws Exception {  
 HttpResponse response = HttpHelper.sendHttpRequest(urlString, method, requestBody, contentType, authToken);  
 return objectMapper.readValue(response.getBody(), responseType);  
 }  
  
 private Map<String, Object> sendHttpRequest(String urlString, String method, String requestBody,  
 String contentType, String authToken) throws Exception {  
 HttpResponse response = HttpHelper.sendHttpRequest(urlString, method, requestBody, contentType, authToken);  
 return objectMapper.readValue(response.getBody(), new TypeReference<>() {});  
 }  
  
 private String sendHttpRequestRaw(String urlString, String method, String requestBody,  
 String contentType, String authToken) throws Exception {  
 return HttpHelper.sendHttpRequest(urlString, method, requestBody, contentType, authToken).getBody();  
 }  
  
  
 private String readErrorResponse(HttpURLConnection connection) throws IOException {  
 try (BufferedReader br = new BufferedReader(  
 new InputStreamReader(connection.getErrorStream(), StandardCharsets.UTF\_8))) {  
 StringBuilder response = new StringBuilder();  
 String responseLine;  
 while ((responseLine = br.readLine()) != null) {  
 response.append(responseLine.trim());  
 }  
 return response.toString();  
 }  
 }  
  
 private String getResponseHeader(String headerName) {  
 // Этот метод нужно реализовать, если требуется доступ к заголовкам ответа  
 // В текущей реализации он не используется, но оставлен для будущих расширений  
 return null;  
 }  
}