МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА №33

ОТЧЕТ ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

Старший преподаватель		Жиданов К.А
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЁТ О ЛАБОРАТОРОНОЙ РАБОТЕ №1:

VIBE CODING

по дисциплине: ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. №	3333		Сорокин А.Д.
		подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

ЦЕЛЬ РАБОТЫ

Разработка веб-сайт для управления задачами (Todo-лист) с аутентификацией пользователей и интеграцией Telegram-бота. Проект позволяет пользователям создавать, редактировать и удалять задачи как через сайт, так и через бота, обеспечивая синхронизацию данных между платформами.

ВВЕДЕНИЕ

Современные технологии стремительно меняют способы взаимодействия пользователей с цифровыми сервисами. Одним из ключевых трендов является создание, которые обеспечивают доступ к данным через различные интерфейсы — веб-сайт, мобильные приложения, чат-боты в мессенджерах. Такой подход значительно повышает удобство использования, позволяя выбирать наиболее подходящий способ работы в зависимости от контекста.

В данной лабораторной работе рассматривается разработка веб-сайта для управления задачами (Todo-лист) с интеграцией Telegram-бота.

Основные аспекты работы:

1. Разработка веб-сайта с системой аутентификации пользователей и возможностью управления задачами.
2. Создание Telegram-бота, дублирующего функционал сайта, что позволяет работать с задачами напрямую из мессенджера.
3. Обеспечение синхронизации данных между веб-интерфейсом и ботом через единую базу данных.
4. Использование современных технологий, включая Node.js, Express, EJS и библиотеку для работы с Telegram Bot API.

Результатом работы является готовый прототип, который можно масштабировать — добавлять новые функции, подключать дополнительные платформы или улучшать производительность.

СТРУКТУРА ПРОЕКТА

К каждому файлу прилагается в следующем пункте описание и код файла.

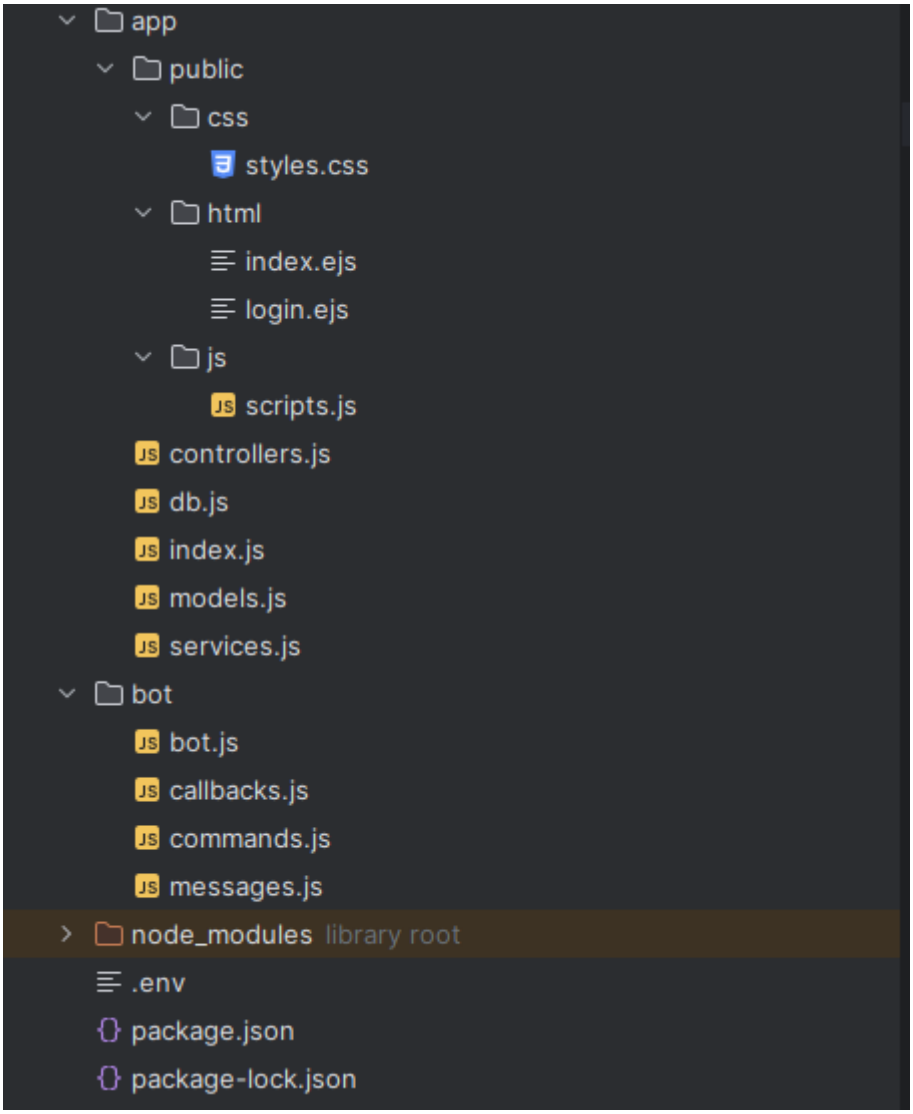

Рисунок 1 – структура проекта в WebStorm

КОД ПРОЕКТА

Код и описание файла `index.js`:

Этот файл является ядром серверной части приложения, реализующим всю основную логику работы веб-сервера. Он выполняет множество критически важных функций, обеспечивая взаимодействие между клиентской частью, базой данных и Telegram-ботом.

Код файла:

```
const express = require('express');
const path = require('path');
const jwt = require('jsonwebtoken');
const { User } = require('./models');
const app = express();
const cookieParser = require('cookie-parser');
const {todoController, userController} = require('./controllers');
const sequelize = require('./db');
require('dotenv').config();

const authMiddleware = (req, res, next) => {
    const token = req.cookies.accessToken;
    if (token) {
        try {
            req.user = jwt.verify(token, process.env.JWT_SECRET);
        } catch (err) {
            console.error('Invalid or expired token:', err.message);
            res.clearCookie('accessToken');
            res.clearCookie('refreshToken');
        }
    }
    next();
};

const blockCheckMiddleware = async (req, res, next) => {
    if (req.user) {
        try {
            const user = await User.findByPk(req.user.id);
            if (user && user.is_blocked) {
                res.clearCookie('accessToken');
                res.clearCookie('refreshToken');
                return res.status(403).render('login', { error: 'Ваш
аккаунт заблокирован' });
            }
        } catch (err) {
            console.error('Error checking block status:', err);
            return res.status(500).send('Internal Server Error');
        }
    }
    next();
```

```
};

const apiRouter = express.Router();
apiRouter.get('/todos', todoController.getTodosHandle);
apiRouter.post('/todos', todoController.addTodoHandle);
apiRouter.put('/todos/:id', todoController.updateTodoHandler);
apiRouter.delete('/todos/:id', todoController.deleteTodoHandler);
apiRouter.post('/login', userController.loginHandle);

const webRouter = express.Router();
webRouter.get('/', (req, res) => {
    if (req.user) {
        todoController.renderHome(req, res);
    } else {
        res.render('login', { error: null });
    }
});
webRouter.get('/register', userController.registerRedirect);
webRouter.get('/auto-login', userController.autoLoginHandle);

app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'public/html'));

app.use(express.static(path.join(__dirname, 'public')));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(cookieParser());
app.use(authMiddleware);
app.use(blockCheckMiddleware);

app.use('/api', apiRouter);
app.use('/', webRouter);

app.use((err, req, res, next) => {
    console.error(err);
    res.status(500).send('Internal Server Error');
});

const start = async () => {
    try {
        await sequelize.authenticate();
        console.log('Connection established');

        await sequelize.sync({ force: false });
        console.log('Database synced');

        app.listen(3000, () => console.log('Server started on port
3000'));
    } catch (err) {
        console.error(err);
        process.exit(1);
    }
}
```

```
start();
```

Код и описание файла `db.js`:

Файл db.js отвечает за подключение приложения к базе данных. Это ключевой модуль, который обеспечивает обмен данными между серверной частью и системой хранения информации. Его основная задача — безопасно установить соединение с базой данных и сделать это подключение доступным для всех компонентов приложения.

Код файла:

```
require('dotenv').config();
const { Sequelize } = require('sequelize');

const sequelize = new Sequelize(
    process.env.DB_NAME,
    process.env.DB_USER,
    process.env.DB_PASSWORD,
    {
        port: process.env.DB_PORT,
        host: process.env.DB_HOST,
        dialect: process.env.DB_DIALECT,
    }
);

module.exports = sequelize;
```

Код и описание файла `models.js`:

Файл models.js является центральным местом для определения структуры данных приложения. Он отвечает за создание моделей базы данных, которые представляют собой "отражение" таблиц в коде. Этот файл определяет, какие данные будут храниться в системе, как они связаны между собой и какие правила должны соблюдаться при работе с ними.

1. Определение структуры данных (таблиц): создает модели для двух основных сущностей: пользователей (User) и задач (Todo)

2. Описание модели пользователя (User): хранит информацию о пользователях системы (id, зашифрованный пароль, токены, флаг блокировки пользователя)
3. Описание модели задач (Todo): Хранит информацию о задачах пользователей (уникальный номер задачи, текст задачи, связь с пользователем через id)
4. Установка связей между моделями: определяет отношения между пользователями и задачами. Связь осуществляется через поле user_id в таблице задач
5. Экспорт моделей для использования в других частях приложения. Делает модели доступными для контроллеров и других модулей. Экспортирует как сами модели (User, Todo), так и экземпляр подключения (sequelize)

Код файла:

```
const { DataTypes } = require('sequelize');
const sequelize = require('./db');

const User = sequelize.define('users_data', {
    id: {
        type: DataTypes.STRING,
        primaryKey: true
    },
    username: {
        type: DataTypes.STRING,
        allowNull: true
    },
    first_name: {
        type: DataTypes.STRING,
        allowNull: true
    },
    last_name: {
        type: DataTypes.STRING,
        allowNull: true
    },
    hash_password: {
        type: DataTypes.STRING,
        allowNull: false
    },
    access_token: {
        type: DataTypes.STRING,
        allowNull: true
```

```
    },
    refresh_token: {
        type: DataTypes.STRING,
        allowNull: true
    },
    is_blocked: {
        type: DataTypes.BOOLEAN,
        defaultValue: false
    },
    one_time_token: {
        type: DataTypes.STRING,
        allowNull: true
    },
    one_time_token_expires: {
        type: DataTypes.DATE,
        allowNull: true
    }
});

const Todo = sequelize.define('todos_data', {
    id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
    },
    text: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    user_id: {
        type: DataTypes.STRING,
        allowNull: true,
        references: {
            model: 'users_data',
            key: 'id'
        }
    }
});

Todo.belongsTo(User, { foreignKey: 'user_id', targetKey: 'id' });
User.hasMany(Todo, { foreignKey: 'user_id', sourceKey: 'id' });

module.exports = { sequelize, Todo, User };
```

Код и описание файла `controllers.js`:

Файл controllers.js является связующим звеном между пользовательскими запросами и бизнес-логикой приложения. Он отвечает за обработку входящих HTTP-запросов, взаимодействие с сервисным слоем и формирование

соответствующих ответов клиенту. Контроллеры выступают посредниками между внешним интерфейсом (веб-страницами или API) и внутренней логикой обработки данных.

Структура и основные компоненты

Файл содержит два основных контроллера:

1. todoController - управляет операциями с задачами
2. userController - отвечает за аутентификацию и работу с пользователями

Каждый контроллер включает набор методов-обработчиков для конкретных маршрутов и действий.

Код файла:

```
const { userService, todoService } = require('./services');

const todoController = {
    renderHome: async (req, res) => {
        try {
            if (!req.user) {
                return res.render('login', { error: null });
            }
            const todos = await todoService.getTodos();
            const isAdmin = await todoService.isAdmin(req.user?.id);
            res.render('index', { todos, user: req.user, isAdmin });
        } catch (err) {
            console.error(err);
            res.status(500).send('Internal Server Error');
        }
    },

    getTodosHandle: async (req, res) => {
        try {
            const todos = await todoService.getTodos();
            res.status(200).json(todos);
        } catch (err) {
            console.error(err);
            res.status(500).json({ error: 'Internal Server Error' });
        }
    },

    addTodoHandle: async (req, res) => {
        try {
            if (!req.user) {
                return res.status(401).json({ error: 'Авторизация
требуется' });
```

```
            }
            const { text } = req.body;
            if (!text) {
                return res.status(400).json({ error: 'Текст обязателен'
});
            }
            await todoService.addTodo(text, req.user.id);
            res.status(201).json({ message: 'Задача добавлена' });
        } catch (err) {
            console.error(err);
            res.status(500).json({ error: 'Ошибка сервера' });
        }
    },

    updateTodoHandler: async (req, res) => {
        try {
            const { id } = req.params;
            const { text } = req.body;
            if (!text) {
                return res.status(400).json({ error: 'Текст обязателен'
});
            }
            const updated = await todoService.updateTodo(id, text,
req.user?.id);
            if (!updated) {
                return res.status(404).json({ error: 'Задача не найдена
или доступ запрещён' });
            }
            res.status(200).json({ message: 'Задача обновлена' });
        } catch (err) {
            console.error(err);
            res.status(500).json({ error: 'Ошибка сервера' });
        }
    },

    deleteTodoHandler: async (req, res) => {
        try {
            const { id } = req.params;
            const deleted = await todoService.deleteTodo(id,
req.user?.id);
            if (!deleted) {
                return res.status(404).json({ error: 'Задача не найдена
или доступ запрещён' });
            }
            res.status(200).json({ message: 'Задача удалена' });
        } catch (err) {
            console.error(err);
            res.status(500).json({ error: 'Ошибка сервера' });
        }
    }
};

const userController = {
    loginHandle: async (req, res) => {
```

```
        try {
            const { username, password } = req.body;
            const { accessToken, refreshToken } = await
userService.login(username, password);
            res.cookie('accessToken', accessToken, { httpOnly: true,
maxAge: 3600000 });
            res.cookie('refreshToken', refreshToken, { httpOnly: true,
maxAge: 7 * 24 * 3600000 });
            res.redirect('/');
        } catch (err) {
            console.error(err);
            res.render('login', { error: err.message || 'Ошибка сервера'
});
        }
    },

    autoLoginHandle: async (req, res) => {
        try {
            const { oneTimeToken } = req.query;
            if (!oneTimeToken) {
                return res.render('login', { error: 'Токен отсутствует'
});
            }
            const { accessToken, refreshToken } = await
userService.loginWithOneTimeToken(oneTimeToken);
            res.cookie('accessToken', accessToken, { httpOnly: true,
maxAge: 3600000 });
            res.cookie('refreshToken', refreshToken, { httpOnly: true,
maxAge: 7 * 24 * 3600000 });
            res.redirect('/');
        } catch (err) {
            console.error(err);
            res.render('login', { error: err.message || 'Ошибка сервера'
});
        }
    },

    registerRedirect: (req, res) => {
        res.redirect(`https://t.me/${process.env.BOT_NAME}`);
    }
};

module.exports = { todoController, userController };
```

Код и описание файла `services.js`:

Файл services.js представляет собой сервисный слой приложения, который содержит всю бизнес-логику работы с пользователями и задачами. Это промежуточное звено между контроллерами (которые принимают запросы) и

моделями (которые работают с базой данных). Сервисы отвечают за сложную обработку данных, выполнение бизнес-правил и взаимодействие с внешними системами.

1. Сервис пользователей (userService). Обрабатывает всю логику, связанную с аутентификацией и управлением пользователями:

- login - вход пользователя в систему:
- loginWithOneTimeToken - вход по одноразовому токену:

2. Сервис задач (todoService). Управляет всеми операциями с задачами

- getTodos - получение списка всех задач:
- addTodo - добавление новой задачи:
- updateTodo - обновление существующей задачи:
- deleteTodo - удаление задачи:
- isAdmin - проверка прав администратора:

Код файла:

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { Todo, User } = require('./models');
const { Op } = require('sequelize');

const userService = {
    login: async (username, password) => {
        try {
            const user = await User.findOne({ where: { username } });
            if (!user || !(await bcrypt.compare(password,
user.hash_password))) {
                throw new Error('Неверные данные');
            }
            const accessToken = jwt.sign({ id: user.id, username:
user.username }, process.env.JWT_SECRET, { expiresIn: '1h' });
            const refreshToken = jwt.sign({ id: user.id, username:
user.username }, process.env.JWT_SECRET, { expiresIn: '7d' });
            await user.update({ access_token: accessToken,
refresh_token: refreshToken });
            return { accessToken, refreshToken, user };
        } catch (err) {
            console.error('Error in login:', err);
```

```
            throw err;
        }
    },

    loginWithOneTimeToken: async (oneTimeToken) => {
        try {
            const user = await User.findOne({
                where: {
                    one_time_token: oneTimeToken,
                    one_time_token_expires: { [Op.gt]: new Date() }
                }
            });
            if (!user) {
                throw new Error('Недействительный или истёкший токен');
            }
            const accessToken = jwt.sign({ id: user.id, username:
user.username }, process.env.JWT_SECRET, { expiresIn: '1h' });
            const refreshToken = jwt.sign({ id: user.id, username:
user.username }, process.env.JWT_SECRET, { expiresIn: '7d' });

            await user.update({
                access_token: accessToken,
                refresh_token: refreshToken,
                one_time_token: null,
                one_time_token_expires: null
            });
            return { accessToken, refreshToken, user };
        } catch (err) {
            console.error('Error in login with one time token:', err);
            throw err;
        }
    }
};

const todoService = {
    getTodos: async () => {
        try {
            const todos = await Todo.findAll({
                include: [{
                    model: User,
                    attributes: ['username', 'is_blocked'],
                    required: false
                }]
            });
            console.log('Fetched todos:', todos.map(t => ({ id: t.id,
user_id: t.user_id, username: t.User ? t.User.username : null })));
            return todos;
        } catch (err) {
            console.error('Error fetching todos:', err);
            throw err;
        }
    },

    addTodo: async (text, userId) => {
```

```
        try {
            if (!userId) {
                console.error('No userId provided for addTodo');
                throw new Error('User ID is required');
            }
            const user = await User.findByPk(userId);
            if (!user) {
                console.error(`User with id ${userId} not found`);
                throw new Error('User not found');
            }
            const todo = await Todo.create({ text, user_id: userId });
            console.log('Created todo:', { id: todo.id, text, user_id:
userId });
            return todo;
        } catch (err) {
            console.error('Error adding todo:', err);
            throw err;
        }
    },

    updateTodo: async (id, text, userId) => {
        try {
            if (!userId) {
                console.error('No userId provided for updateTodo');
                throw new Error('User ID is required');
            }
            const user = await User.findByPk(userId);
            if (!user) {
                throw new Error('User not found');
            }
            const isAdmin = userId === process.env.ADMIN_CHAT_ID &&
await bcrypt.compare(process.env.ADMIN_PASSWORD, user.hash_password);
            const whereClause = isAdmin ? { id } : { id, user_id: userId
};
            const [updated] = await Todo.update({ text }, { where:
whereClause });
            if (updated) {
                const todo = await Todo.findByPk(id);
                console.log('Updated todo:', { id, text, user_id: userId
});
                return todo;
            }
            return null;
        } catch (err) {
            console.error('Error updating todo:', err);
            throw err;
        }
    },

    deleteTodo: async (id, userId) => {
        try {
            if (!userId) {
                console.error('No userId provided for deleteTodo');
                throw new Error('User ID is required');
```

```
            }
            const user = await User.findByPk(userId);
            if (!user) {
                throw new Error('User not found');
            }
            const isAdmin = userId === process.env.ADMIN_CHAT_ID &&
await bcrypt.compare(process.env.ADMIN_PASSWORD, user.hash_password);
            const whereClause = isAdmin ? { id } : { id, user_id: userId
};
            const deleted = await Todo.destroy({ where: whereClause });
            console.log('Deleted todo:', { id, user_id: userId, deleted
});
            return deleted > 0;
        } catch (err) {
            console.error('Error deleting todo:', err);
            throw err;
        }
    },

    isAdmin: async (userId) => {
        try {
            const user = await User.findByPk(userId);
            if (!user) return false;
            return userId === process.env.ADMIN_CHAT_ID && await
bcrypt.compare(process.env.ADMIN_PASSWORD, user.hash_password);
        } catch (err) {
            console.error('Error checking admin status:', err);
            return false;
        }
    }
};

module.exports = { userService, todoService };
```

Код и описание файла `index.ejs`:

Файл index.ejs представляет собой главную страницу веб-приложения для управления списком дел. Это шаблон, который динамически формирует HTML-код на сервере с использованием данных, переданных из контроллера, и затем отправляет готовую страницу в браузер пользователя.

Код файла:

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Список дел</title>
    <link rel="stylesheet" href="/css/styles.css">
```

```
</head>
<body>
<h2 class="center-text">Список дел</h2>
<% if (!user) { %>
    <p class="center-text">Пожалуйста, авторизуйтесь.</p>
    <p class="center-text"><a href="/login">Войти</a></p>
<% } else { %>
    <form id="addTodoForm" class="form-margin center-text">
        <input type="text" id="todoText" class="input-padding"
placeholder="Введите задачу" required>
        <button type="submit" class="button">Добавить дело</button>
    </form>
    <table id="todoList">
        <thead>
        <tr>
            <th>№</th>
            <th>Текст</th>
            <th>Имя пользователя</th>
        </tr>
        </thead>
        <tbody>
        <% todos.forEach(todo => { %>
            <% if (!todo.users_datum?.is_blocked) { %>
                <tr data-id="<%= todo.id %>">
                    <td><%= todo.id %></td>
                    <td>
                        <span class="todo-text"><%= todo.text %></span>
                        <% if (user.id === todo.user_id || isAdmin) { %>
                            <button class="delete-button" data-id="<%
todo.id %>">Удалить</button>
                            <button class="edit-button" data-id="<%=
todo.id %>" data-text="<%= todo.text %>">Изменить</button>
                        <% } %>
                    </td>
                    <td><%= todo.users_datum ? todo.users_datum.username
: 'Неизвестно' %></td>
                </tr>
            <% } %>
        <% }); %>
        </tbody>
    </table>
    <div id="editModal" class="modal">
        <div class="modal-content">
            <form id="editForm">
                <input type="text" id="editText" class="input-full-
width" required>
                <button type="submit" class="button">Сохранить</button>
                <button type="button" id="cancelEdit" class="button
cancel-button">Отмена</button>
            </form>
        </div>
    </div>
<% } %>
<script id="userData" type="application/json">
```

```
    <%- JSON.stringify({ id: user?.id, isAdmin: isAdmin, is_blocked:
user?.is_blocked }) %>
</script>
<script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<script src="/js/scripts.js"></script>
</body>
</html>
```

Код и описание файла `login.ejs`:

Файл login.ejs представляет собой шаблон страницы авторизации пользователя в веб-приложении. Это первая точка взаимодействия пользователя с системой, отвечающая за безопасный вход в личный кабинет и обработку ошибок аутентификации.

Код файла:

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Авторизация</title>
    <link rel="stylesheet" href="/css/styles.css">
</head>
<body>
<div class="login-container">
    <h2>Авторизация</h2>
    <% if (error) { %>
        <p class="error"><%= error %></p>
    <% } %>
    <form action="/api/login" method="POST">
        <div class="form-group">
            <label for="username">Имя пользователя</label>
            <input type="text" id="username" name="username" required>
        </div>
        <div class="form-group">
            <label for="password">Пароль</label>
            <input type="password" id="password" name="password"
required>
        </div>
        <button type="submit">Войти</button>
    </form>
    <div class="register-link">
        <a href="/register">Зарегистрироваться</a>
    </div>
</div>
<script
```

```
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
</body>
</html>
```

Код и описание файла `styles.css`:

Файл styles.css является таблицей стилей приложения, отвечающей за визуальное оформление всех элементов интерфейса. Он обеспечивает единый дизайн-систему, удобство взаимодействия и привлекательный внешний вид веб-приложения.

Код файла:

```
body {
    font-family: 'Roboto', sans-serif;
    background-color: #f4f7fa;
    color: #333;
    margin: 0;
    padding: 20px;
}

#todoList {
    border-collapse: collapse;
    width: 80%;
    margin: 20px auto;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
    background-color: #fff;
    border-radius: 8px;
    overflow: hidden;
}

#todoList th, #todoList td {
    padding: 12px;
    border-bottom: 1px solid #e0e0e0;
    text-align: left;
}

#todoList th {
    background-color: #2c3e50;
    color: #fff;
    font-weight: 500;
}

#todoList tr:hover {
    background-color: #f8f9fa;
}

.button {
    padding: 8px 16px;
```

```
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 14px;
    transition: background-color 0.3s, transform 0.1s;
}

.button:hover {
    transform: translateY(-1px);
}

.edit-button {
    background-color: #3498db;
    color: #fff;
    margin-left: 12px;
    float: right;
}

.edit-button:hover {
    background-color: #2980b9;
}

.delete-button {
    background-color: #e74c3c;
    color: #fff;
    margin-left: 12px;
    float: right;
}

.delete-button:hover {
    background-color: #c0392b;
}

.center-text {
    text-align: center;
}

.form-margin {
    margin-bottom: 25px;
}

.input-padding {
    padding: 8px;
    border: 1px solid #ccc;
    border-radius: 4px;
    font-size: 14px;
}

.modal {
    display: none;
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
```

```
    height: 100%;
    background: rgba(0, 0, 0, 0.6);
    z-index: 1000;
}

.modal-content {
    background: #fff;
    margin: 10% auto;
    padding: 30px;
    width: 60%;
    max-width: 500px;
    border-radius: 8px;
    box-shadow: 0 8px 16px rgba(0, 0, 0, 0.2);
}

.input-full-width {
    width: 100%;
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 4px;
    font-size: 16px;
}

.cancel-button {
    margin-top: 15px;
    background-color: #95a5a6;
    color: #fff;
}

.cancel-button:hover {
    background-color: #7f8c8d;
}

.login-container {
    width: 100%;
    max-width: 400px;
    margin: 60px auto;
    padding: 30px;
    border: none;
    border-radius: 10px;
    background: #fff;
    box-shadow: 0 6px 12px rgba(0, 0, 0, 0.15);
}

.login-container h2 {
    text-align: center;
    margin-bottom: 25px;
    color: #2c3e50;
    font-size: 24px;
    font-weight: 700;
}

.login-container .form-group {
    margin-bottom: 20px;
```

```
}

.login-container .form-group label {
    display: block;
    margin-bottom: 8px;
    font-weight: 500;
    color: #34495e;
}

.login-container .form-group input {
    box-sizing: border-box;
    width: 100%;
    padding: 10px;
    border: 1px solid #dfe6e9;
    border-radius: 6px;
    font-size: 16px;
    transition: border-color 0.3s;
}

.login-container .form-group input:focus {
    outline: none;
    border-color: #3498db;
}

.login-container button {
    width: 100%;
    padding: 12px;
    background-color: #3498db;
    color: #fff;
    border: none;
    border-radius: 6px;
    cursor: pointer;
    font-size: 16px;
    font-weight: 500;
    transition: background-color 0.3s;
}

.login-container button:hover {
    background-color: #2980b9;
}

.login-container .register-link {
    text-align: center;
    margin-top: 15px;
}

.login-container .register-link a {
    color: #3498db;
    text-decoration: none;
    font-weight: 500;
}

.login-container .register-link a:hover {
    text-decoration: underline;
```

```
}

.login-container .error {
    color: #e74c3c;
    text-align: center;
    margin-bottom: 15px;
    font-size: 14px;
}
```

Код и описание файла `bot.js`:

Файл bot.js является основным модулем Telegram-бота, который отвечает за его запуск, настройку и обработку входящих взаимодействий с пользователями. Это центральный файл, который связывает все компоненты бота (команды, сообщения, callback-запросы) в единую систему.

Код файла:

```
const TelegramBot = require('node-telegram-bot-api');
const { handleCommand } = require('./commands');
const { handleCallback } = require('./callbacks');
const { handleMessage } = require('./messages');
require('dotenv').config();

const token = process.env.BOT_TOKEN;
const bot = new TelegramBot(token, { polling: true });

let userStates = {};

bot.onText(/\/(.+)/, (msg) => handleCommand(msg, bot, userStates));
bot.on('callback_query', (callbackQuery) =>
handleCallback(callbackQuery, bot, userStates));
bot.on('message', (msg) => {
    if (!msg.text.startsWith('/')) {
        handleMessage(msg, bot, userStates);
    }
});

module.exports = bot;
```

Код и описание файла `callbacks.js`:

Файл callbacks.js отвечает за обработку нажатий на inline-кнопки Telegram-бота, управляя действиями пользователя в зависимости от текущего этапа взаимодействия и переданных callback-данных. Он обрабатывает переключение страниц списка задач или пользователей, обновление и удаление задач, а также

блокировку и разблокировку пользователей. Для выполнения этих действий бот использует информацию о текущем состоянии пользователя, проверяет права администратора (сравнивая ID и хэш-пароль), взаимодействует с базой данных моделей Todo и User, обновляет интерфейс и удаляет устаревшие сообщения. Все действия сопровождаются обратной связью пользователю и корректной обработкой callback-запроса.

Код файла:

```
const { Todo, User } = require('../app/models');
const bcrypt = require('bcrypt');
const { showTodoButtons, showUserButtons } = require('./commands');

const handleCallback = async (callbackQuery, bot, userStates) => {
    const chatId = callbackQuery.message.chat.id;
    const data = callbackQuery.data;
    const messageId = callbackQuery.message.message_id;
    const state = userStates[chatId];

    if (!state) return;

    const isAdmin = async (userId) => {
        const user = await User.findByPk(userId);
        if (!user) return false;
        return userId === process.env.ADMIN_CHAT_ID && await
bcrypt.compare(process.env.ADMIN_PASSWORD, user.hash_password);
    };

    if (data === 'prev' || data === 'next') {
        const page = data === 'prev' ? state.page - 1 : state.page + 1;
        state.page = page;
        if (state.step === 'block_user') {
            showUserButtons(chatId, state.users, page, bot, userStates,
messageId);
        } else {
            showTodoButtons(chatId, state.todos, page, state.userId,
bot, userStates, messageId);
        }
        bot.answerCallbackQuery(callbackQuery.id);
        return;
    }

    if (data.startsWith('todo_')) {
        const todoId = parseInt(data.split('_')[1]);
        const todo = state.todos.find(t => t.id === todoId);
        if (!todo) {
            bot.answerCallbackQuery(callbackQuery.id, { text: 'Задача не
найдена' });
            return;
```

```
        }

        if (state.step === 'update_todo') {
            state.selectedTodo = todo;
            bot.sendMessage(chatId, `Введите новый текст для дела
"${todo.text}":`);
            bot.answerCallbackQuery(callbackQuery.id);
        } else if (state.step === 'delete_todo') {
            const canDelete = (await isAdmin(state.userId)) ||
todo.user_id === state.userId;
            if (!canDelete) {
                bot.answerCallbackQuery(callbackQuery.id, { text: 'У вас
нет прав для удаления этой задачи' });
                return;
            }
            await Todo.destroy({ where: { id: todoId } });
            bot.sendMessage(chatId, `Дело "${todo.text}" удалено!`);
            delete userStates[chatId];
            bot.deleteMessage(chatId, messageId);
            bot.answerCallbackQuery(callbackQuery.id);
        }
    } else if (data.startsWith('user_')) {
        if (state.step !== 'block_user' || !(await
isAdmin(state.userId))) {
            bot.answerCallbackQuery(callbackQuery.id, { text: 'У вас нет
прав для этой операции' });
            return;
        }
        const userId = data.split('_')[1];
        const user = state.users.find(u => u.id === userId);
        if (!user) {
            bot.answerCallbackQuery(callbackQuery.id, { text:
'Пользователь не найден' });
            return;
        }
        const newStatus = !user.is_blocked;
        await User.update({ is_blocked: newStatus }, { where: { id:
userId } });
        bot.sendMessage(chatId, `Пользователь @${user.username} успешно
${newStatus ? 'заблокирован' : 'разблокирован'}!`);
        delete userStates[chatId];
        bot.deleteMessage(chatId, messageId);
        bot.answerCallbackQuery(callbackQuery.id);
    }
};

module.exports = { handleCallback };
```

Код и описание файла `commands.js`:

Файл commands.js — центральный диспетчер текстовых команд Telegram-бота: он объявляет набор доступных команд, реагирует на /start, /add, /update, /delete и скрытую админскую /block, проверяет права пользователя (в том числе сравнивая его ID и bcrypt-хэш пароля с данными из переменных окружения), фиксирует текущий «шаг» и вспомогательную информацию в userStates, а затем направляет пользователя по нужному сценарию.

Код файла:

```
const { Todo, User } = require('../app/models');
const bcrypt = require('bcrypt');
const { Op } = require('sequelize');

const handleCommand = async (msg, bot, userStates) => {
    const chatId = msg.chat.id;
    const command = msg.text;

    bot.setMyCommands([
        { command: '/start', description: 'Регистрация' },
        { command: '/add', description: 'Добавить дело' },
        { command: '/update', description: 'Изменить дело' },
        { command: '/delete', description: 'Удалить дело' }
    ]);

    const isAdmin = async (userId) => {
        const user = await User.findByPk(userId);
        if (!user) return false;
        return userId === process.env.ADMIN_CHAT_ID && await
bcrypt.compare(process.env.ADMIN_PASSWORD, user.hash_password);
    };

    if (command === '/start') {
        if (await User.findOne({ where: { id: chatId.toString() } })) {
            bot.sendMessage(chatId, 'Вы уже зарегистрированы!
Используйте команды для работы.');
            return;
        }
        userStates[chatId] = { step: 'password' };
        bot.sendMessage(chatId, 'Введите пароль:');
    } else if (command === '/add') {
        const user = await User.findOne({ where: { id: chatId.toString()
} });
        if (!user || user.is_blocked) {
            bot.sendMessage(chatId, 'Вы не авторизованы или
заблокированы! Зарегистрируйтесь с /start.');
            return;
        }
```

```
        userStates[chatId] = { step: 'add_todo', userId: user.id };
        bot.sendMessage(chatId, 'Введите текст дела:');
    } else if (command === '/update') {
        const user = await User.findOne({ where: { id: chatId.toString()
} });
        if (!user || user.is_blocked) {
            bot.sendMessage(chatId, 'Вы не авторизованы или
заблокированы!');
            return;
        }
        const todos = (await isAdmin(user.id))
            ? await Todo.findAll({ include: [{ model: User, attributes:
['username'] }] })
            : await Todo.findAll({ where: { user_id: user.id } });
        if (!todos.length) {
            bot.sendMessage(chatId, 'Нет дел для редактирования!');
            return;
        }
        userStates[chatId] = { step: 'update_todo', todos, page: 0,
userId: user.id };
        showTodoButtons(chatId, todos, 0, user.id, bot, userStates);
    } else if (command === '/delete') {
        const user = await User.findOne({ where: { id: chatId.toString()
} });
        if (!user || user.is_blocked) {
            bot.sendMessage(chatId, 'Вы не авторизованы или
заблокированы!');
            return;
        }
        const todos = (await isAdmin(user.id))
            ? await Todo.findAll({ include: [{ model: User, attributes:
['username'] }] })
            : await Todo.findAll({ where: { user_id: user.id } });
        if (!todos.length) {
            bot.sendMessage(chatId, 'Нет дел для удаления!');
            return;
        }
        userStates[chatId] = { step: 'delete_todo', todos, page: 0,
userId: user.id };
        showTodoButtons(chatId, todos, 0, user.id, bot, userStates);
    } else if (command === '/block') {
        const user = await User.findOne({ where: { id: chatId.toString()
} });
        if (!user || !(await isAdmin(user.id))) {
            return;
        }
        const users = await User.findAll({
            attributes: ['id', 'username', 'is_blocked'],
            where: { id: { [Op.ne]: process.env.ADMIN_CHAT_ID } }
        });
        if (!users.length) {
            bot.sendMessage(chatId, 'Нет пользователей для
управления!');
            return;
```

```
        }
        userStates[chatId] = { step: 'block_user', users, page: 0,
userId: user.id };
        showUserButtons(chatId, users, 0, bot, userStates);
    }
};

const showTodoButtons = (chatId, todos, page, userId, bot, userStates,
messageId = null) => {
    const perPage = 5;
    const start = page * perPage;
    const end = start + perPage;
    const paginatedTodos = todos.slice(start, end);

    const inline_keyboard = paginatedTodos.map(todo => [
        { text: `${todo.text} ${todo.User ? `(@${todo.User.username})` :
''}`, callback_data: `todo_${todo.id}` }
    ]);

    const navButtons = [];
    if (page > 0) navButtons.push({ text: '<', callback_data: 'prev' });
    if (end < todos.length) navButtons.push({ text: '>', callback_data:
'next' });
    if (navButtons.length) inline_keyboard.push(navButtons);

    const options = {
        reply_markup: {
            inline_keyboard
        }
    };

    const messageText =
        'Какое дело хотите ' +
        (userStates[chatId].step === 'update_todo' ? 'редактировать' :
'удалить') +
        '?';

    if (messageId) {
        bot.editMessageText(messageText, {
            chat_id: chatId,
            message_id: messageId,
            ...options
        });
    } else {
        bot.sendMessage(chatId, messageText, options);
    }
};

const showUserButtons = (chatId, users, page, bot, userStates, messageId
= null) => {
    const perPage = 5;
    const start = page * perPage;
    const end = start + perPage;
    const paginatedUsers = users.slice(start, end);
```

```
    const inline_keyboard = paginatedUsers.map(user => [
        { text: `@${user.username} (${user.is_blocked ? 'Заблокирован' :
'Не заблокирован'})`, callback_data: `user_${user.id}` }
    ]);

    const navButtons = [];
    if (page > 0) navButtons.push({ text: '<', callback_data: 'prev' });
    if (end < users.length) navButtons.push({ text: '>', callback_data:
'next' });
    if (navButtons.length) inline_keyboard.push(navButtons);

    const options = {
        reply_markup: {
            inline_keyboard
        }
    };

    const messageText = 'Выберите пользователя для
блокировки/разблокировки:';

    if (messageId) {
        bot.editMessageText(messageText, {
            chat_id: chatId,
            message_id: messageId,
            ...options
        });
    } else {
        bot.sendMessage(chatId, messageText, options);
    }
};

module.exports = { handleCommand, showTodoButtons, showUserButtons };
```

Код и описание файла `messages.js`:

Файл messages.js реализует обработчик обычных текстовых сообщений, он использует объект userStates, чтобы определить текущий этап взаимодействия и выполнить соответствующую логику. Таким образом, модуль отвечает за интерактивную текстовую часть диалога — регистрацию, добавление и изменение задач — в зависимости от ранее заданного сценария через команды или кнопки.

Код файла:

```
const { Todo, User } = require('../app/models');
const bcrypt = require('bcrypt');
const { v4 } = require('uuid');
```

```
const handleMessage = async (msg, bot, userStates) => {
    const chatId = msg.chat.id;
    const text = msg.text;
    const state = userStates[chatId];

    if (!state) return;

    const isAdmin = async (userId) => {
        const user = await User.findByPk(userId);
        if (!user) return false;
        return userId === process.env.ADMIN_CHAT_ID && await
bcrypt.compare(process.env.ADMIN_PASSWORD, user.hash_password);
    };

    if (state.step === 'password') {
        state.password = text;
        bot.sendMessage(chatId, 'Подтвердите пароль:');
        state.step = 'confirm_password';
    } else if (state.step === 'confirm_password') {
        if (state.password === text) {
            try {
                const hash = await bcrypt.hash(text, 10);
                const oneTimeToken = v4();
                const username = msg.from.username || `user_${chatId}`;
                const user = await User.create({
                    id: chatId.toString(),
                    username: msg.from.username || null,
                    first_name: msg.from.first_name || null,
                    last_name: msg.from.last_name || null,
                    hash_password: hash,
                    one_time_token: oneTimeToken,
                    one_time_token_expires: new Date(Date.now() + 15 *
60 * 1000)
                });
                console.log('User created:', { id: user.id, username });
                bot.sendMessage(chatId, 'Вы успешно зарегистрированы!',
{
                    reply_markup: JSON.stringify({
                        inline_keyboard: [[
                            { text: 'Сайт', url:
`${process.env.APP_URL}/auto-login?oneTimeToken=${oneTimeToken}` }
                        ]]
                    })
                });
                delete userStates[chatId];
            } catch (err) {
                console.error('Error creating user:', err);
                bot.sendMessage(chatId, 'Ошибка при регистрации.
Попробуйте снова с /start.');
                delete userStates[chatId];
            }
        } else {
            bot.sendMessage(chatId, 'Пароли не совпадают! Попробуйте еще
```

```
раз с /start.');
            delete userStates[chatId];
        }
    } else if (state.step === 'add_todo') {
        try {
            const user = await User.findByPk(state.userId);
            if (!user) {
                bot.sendMessage(chatId, 'Пользователь не найден.
Пожалуйста, зарегистрируйтесь с /start.');
                delete userStates[chatId];
                return;
            }
            await Todo.create({ text, user_id: state.userId });
            bot.sendMessage(chatId, 'Дело успешно добавлено!');
            delete userStates[chatId];
        } catch (err) {
            console.error('Error adding todo:', err);
            bot.sendMessage(chatId, 'Ошибка при добавлении дела.');
            delete userStates[chatId];
        }
    } else if (state.step === 'update_todo' && state.selectedTodo) {
        try {
            const canUpdate = (await isAdmin(state.userId)) ||
state.selectedTodo.user_id === state.userId;
            if (!canUpdate) {
                bot.sendMessage(chatId, 'У вас нет прав для
редактирования этой задачи.');
                delete userStates[chatId];
                return;
            }
            await Todo.update({ text }, { where: { id:
state.selectedTodo.id } });
            bot.sendMessage(chatId, `Дело "${state.selectedTodo.text}"
успешно изменено на "${text}"!`);
            delete userStates[chatId];
        } catch (err) {
            console.error('Error updating todo:', err);
            bot.sendMessage(chatId, 'Ошибка при обновлении дела.');
            delete userStates[chatId];
        }
    }
};

module.exports = { handleMessage };
```

Код и описание файла `.env`:

Файл .env содержит конфигурационные переменные окружения, которые используются приложением для безопасного хранения чувствительных данных и параметров работы. В данном случае переменные DB_NAME, DB_USER,

DB_PASSWORD, DB_PORT, DB_HOST и DB_DIALECT задают параметры подключения к базе данных PostgreSQL: имя базы данных, имя пользователя, пароль, порт, хост и используемый диалект. APP_URL содержит адрес веб-приложения, который используется, например, для генерации ссылок авторизации. BOT_NAME и BOT_TOKEN определяют имя и токен Telegram-бота, необходимый для его работы через Telegram API. ADMIN_CHAT_ID и ADMIN_PASSWORD задают идентификатор администратора в Telegram и его пароль (в незахэшированном виде, используется далее для сверки при входе), позволяя разграничить доступ к административным функциям.

Код файла:

```
DB_NAME=postgres
DB_USER=postgres
DB_PASSWORD=1234
DB_PORT=5433
DB_HOST=localhost
DB_DIALECT=postgres

APP_URL=https://leha-vibe.cloudpub.ru

BOT_NAME=leha_vibe_bot
BOT_TOKEN=8070421124:AAGPqcGli88pDulb982b67i3JLgLti_-8w4

ADMIN_CHAT_ID=858744790
ADMIN_PASSWORD=1234

JWT_SECRET=leha-vibe
```

Код и описание файла `package.json`:

Файл package.json является основным конфигурационным файлом Node.js-проекта, содержащим информацию о самом проекте, его зависимостях и сценариях запуска. В данном случае он описывает проект с именем lw_1 версии 1.0.0, где основным входным файлом считается app/index.js. Таким образом, package.json описывает структуру, зависимости и сценарии запуска проекта, позволяя управлять его установкой, разработкой и развертыванием.

Код файла:

```
{
  "name": "lw_1",
  "version": "1.0.0",
  "main": "app/index.js",
  "scripts": {
    "start_app": "nodemon app/index.js",
    "start_bot": "nodemon bot/bot.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "bcrypt": "^6.0.0",
    "cookie-parser": "^1.4.7",
    "dotenv": "^16.5.0",
    "ejs": "^3.1.10",
    "express": "^5.1.0",
    "jsonwebtoken": "^9.0.2",
    "node-telegram-bot-api": "^0.66.0",
    "pg": "^8.14.1",
    "sequelize": "^6.37.7",
    "uuid": "^11.1.0"
  },
  "devDependencies": {
    "nodemon": "^3.1.9"
  }
}
```

ПРИМЕР РАБОТЫ ПРОЕКТА

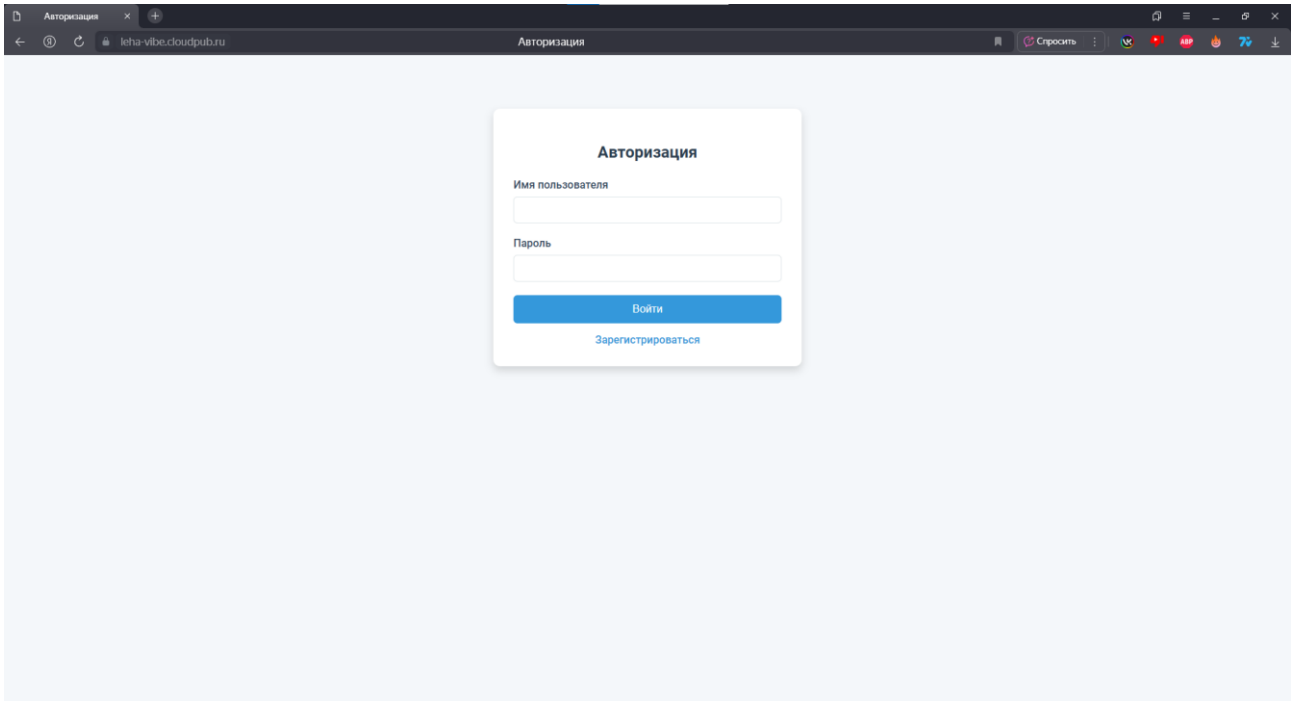

Рисунок 2 – авторизация через сайт

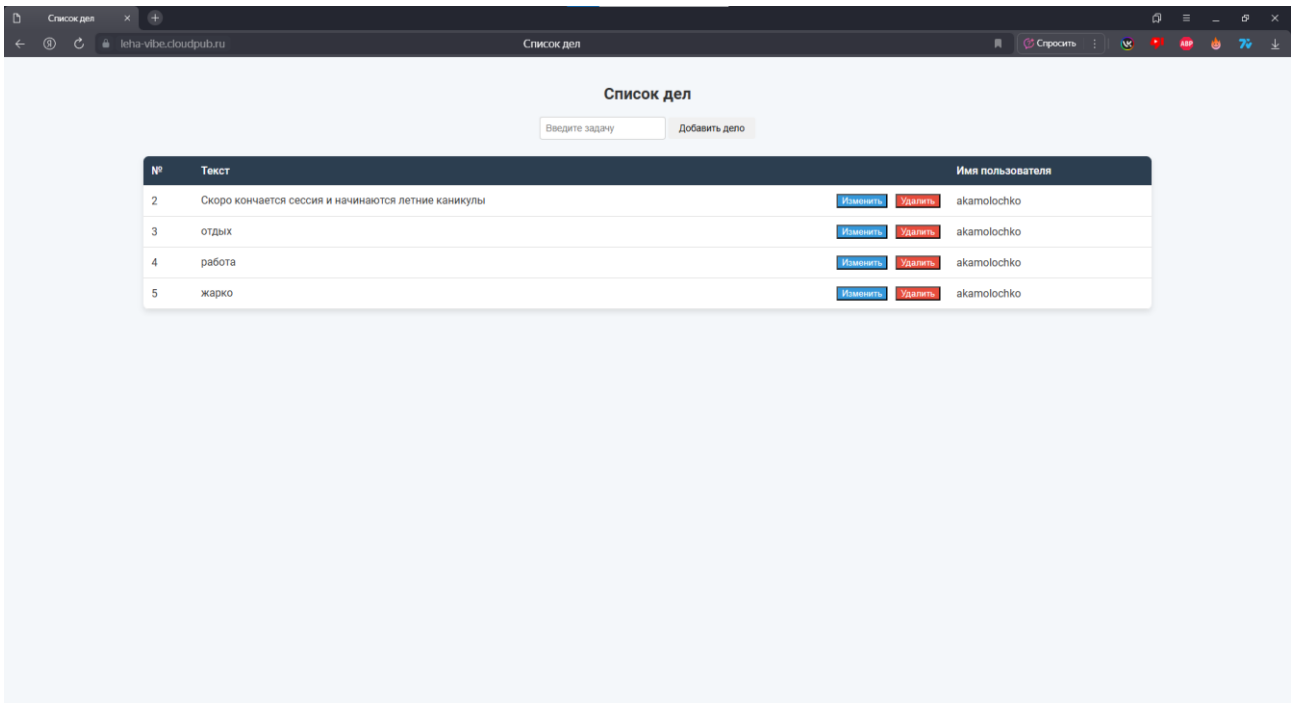

Рисунок 3 – добавление задач

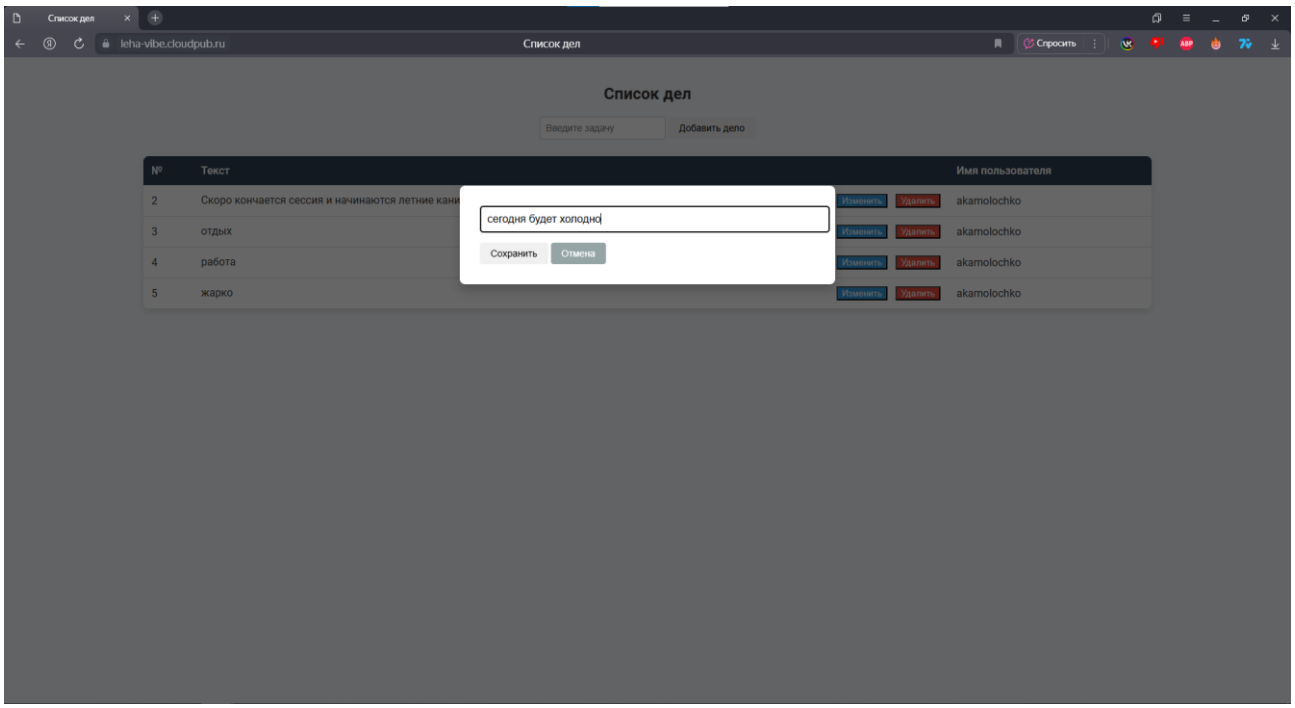

Рисунок 4 – изменение “жарко” на “сегодня будет холодно”

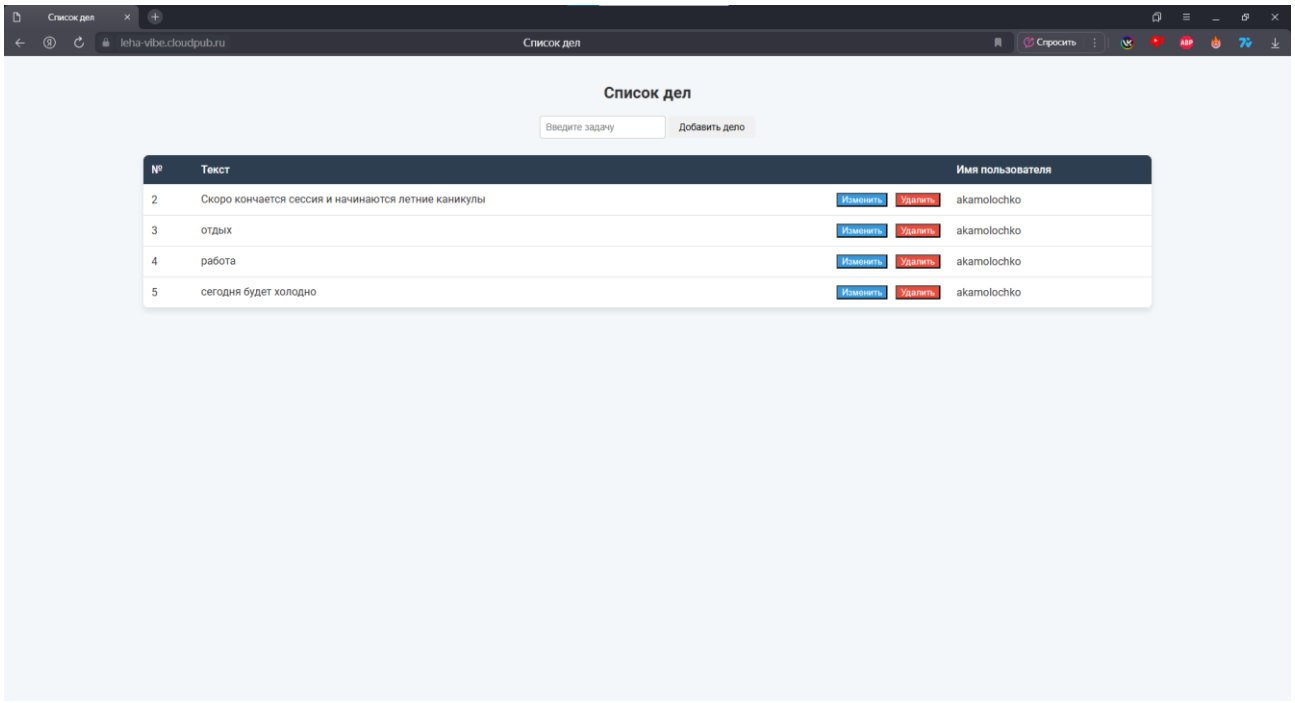

Рисунок 5 – результат изменения

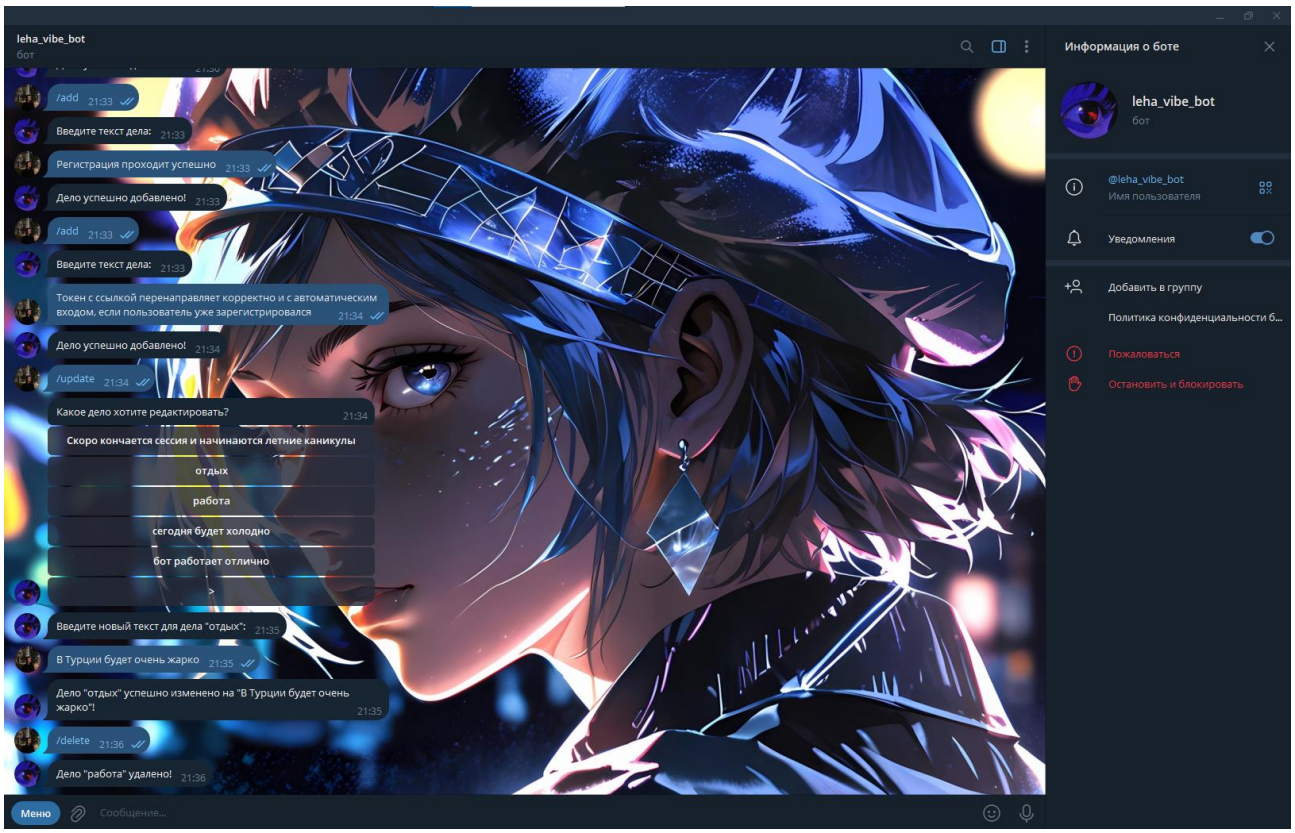

Рисунок 7 – произведение всех операций через телеграмм-бота

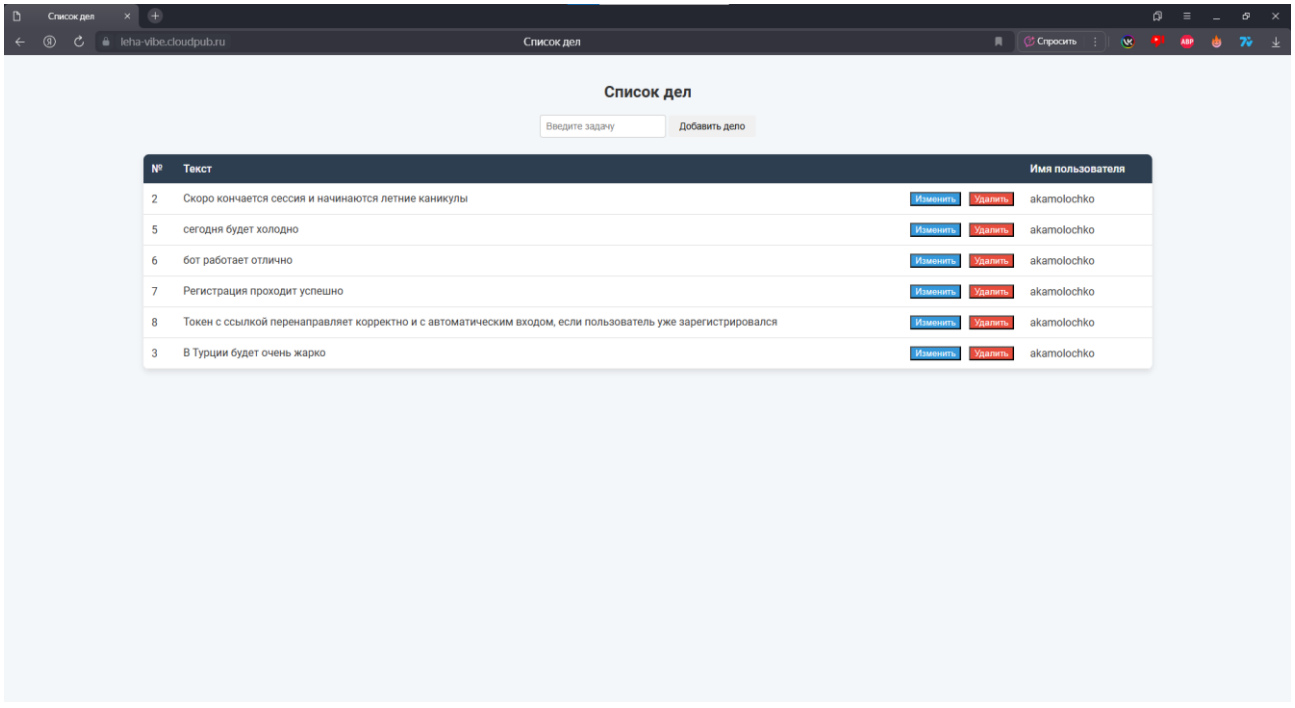

Рисунок 8 – результаты изменений при помощи бота на самом сайте

По рисунку 8 можно увидеть, что все операции, проделанные через телеграмм-бота на рисунке 8 успешно перенеслись и на сам сайт. То есть база данных корректно обновляется и при помощи сайта и при помощи бота.

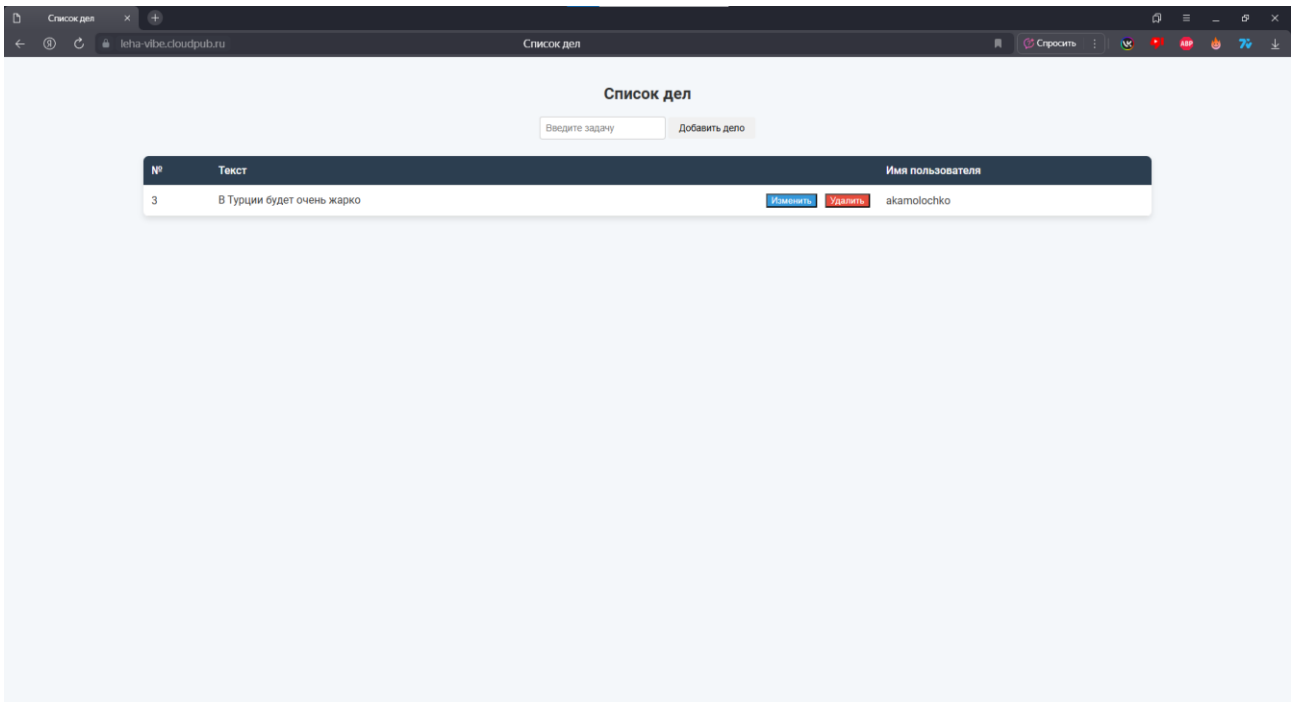

Рисунок 9 – удалили все задачи кроме одной с помощью кнопки удалить на самом сайте

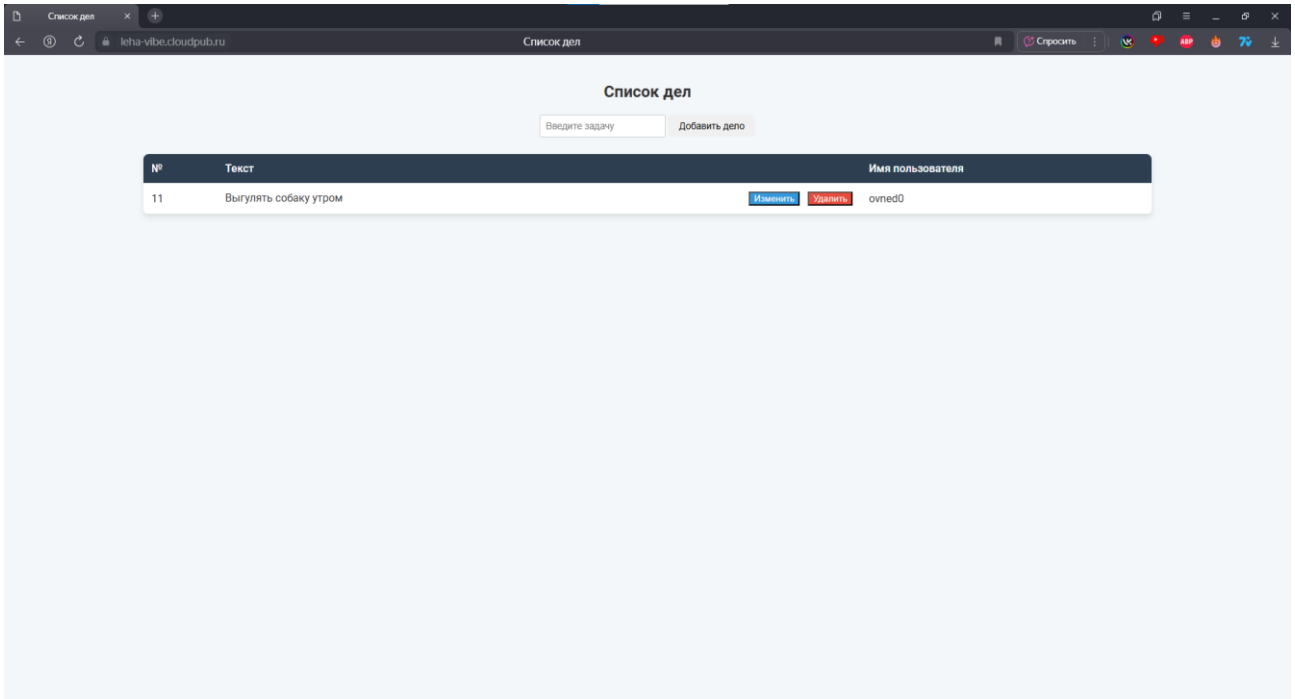

Рисунок 10 – другой пользователь добавил задачу и она у нас корректно отображается

По рисунку 10 можно увидеть, что мы можем удалять и редактировать дела другого пользователя, так как обладаем правами администратора.

Рисунок 11 – скрытая функция администратора

По рисунку 11 можно увидеть, что мы можем блокировать или разблокировать новых пользователей, если они имеются.

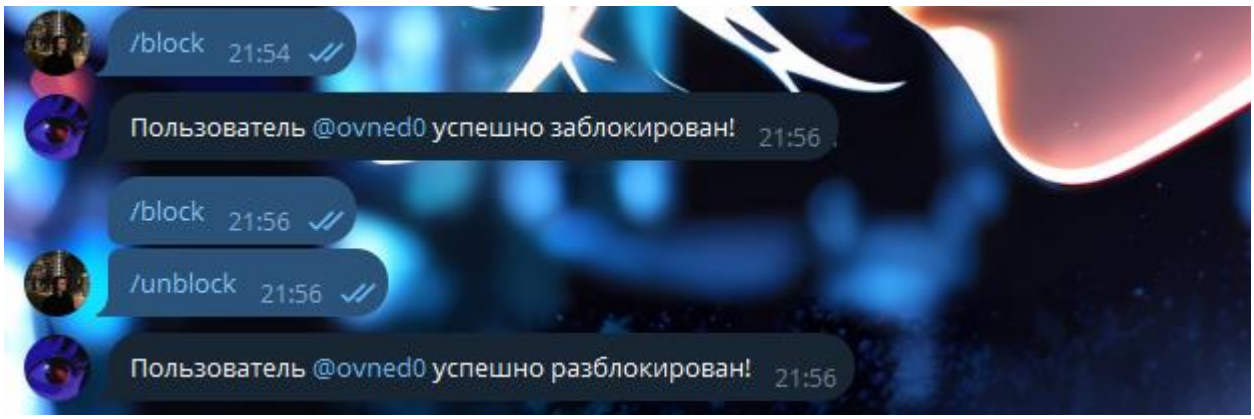

Рисунок 12- блокировка и разблокировка пользователя

По рисунку 11 и 12 можно увидеть, что мы можем блокировать или разблокировать новых пользователей, если они имеются.

ПРИМЕРЫ ЗАПРОСОВ ДЛЯ ИИ

1) Сделай пожалуйста структуру Node.js проекта для Todo-приложения с авторизацией и Telegram-ботом. При этом включи следующие пункты: сервер на Express, EJS шаблоны, Sequelize ORM, папки для контроллеров, сервисов, моделей.

2) Напиши файл index.js с cookie-parser, роутами для API и веб-интерфейса, обработкой ошибок, интеграцией с Sequelize.

3) Сделай пожалуйста models.js с Sequelize: User и Todo. При этом добавь связь между ними для корректной работы.

4) Напиши controllers.js с: аутентификацией (login/register), с задачами по удалению, изменению, добавленнию, а также проверкой по правам доступа в зависимости от администратора и обычного пользователя.

5) Сделай services.js с бизнес-логикой: хеширование паролей, генерация токенов, запросы к базе данных через Sequelize.

6) Сделай bot.js для Telegram-бота: обработкой команд (/start, /add), callback-кнопок, состояний пользователя, интеграцией с API сервера."

7) Напиши commands.js для бота с: регистрацией через /start, добавлением/удалением задач, админ-командой для блокировки пользователей.

8) Сделай EJS-шаблоны (index.ejs, login.ejs): форма входа, таблица задач, модальное окно редактирования, кнопки с проверкой прав (можно удалять то, что введут другие пользователи, но они при этом не смогут удалить то, что ввёл админ).

9) Напиши styles.css для интерфейса: стили таблицы задач, кнопок (при этом синие для edit, красные для delete), модального окна, формы входа в единой стилистики.

10) Создай файл для синхронизации данных между веб-интерфейсом и Telegram-ботом.

ВЫВОД

В ходе работы был разработан полноценный сервис для управления задачами с веб-интерфейсом и Telegram-ботом, интегрированными через общую базу данных. Основной код был сгенерирован с помощью ИИ по конкретным запросам, что значительно ускорило процесс разработки. ИИ использовался для создания типовых компонентов: настройки Express-сервера, моделей Sequelize, CRUD-логики, Telegram-бота и EJS-шаблонов. Вручную дорабатывались только различные ключевые моменты.

Такой подход доказал свою эффективность: время разработки сократилось в 3-4 раза, а ИИ помог быстро освоить незнакомые технологии (например, Telegram Bot API). Однако выявились и ограничения: код иногда требовал ручной правки для согласованности, а сложная логика (например, админ-панель) нуждалась в доработке.

По итогу можно сказать, что ИИ идеален для прототипирования и рутинных задач, но критически важные части (бизнес-логика, безопасность) требуют контроля разработчика. Проект успешно работает, а полученный опыт показывает, что грамотное сочетание ИИ и программиста даёт максимальный результат при минимальных затратах. Для дальнейшего развития можно добавить верификацию email, WebSocket или Docker, используя тот же гибридный подход.