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

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

ОТЧЁТ О ЛАБОРАТОРОНОЙ РАБОТЕ

по дисциплине: ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

РАБОТУ ВЫПОЛНИЛ

Студент гр. № 3333		В.А.Виговский
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ

Создать полнофункциональное веб-приложение для управления списком задач с возможностью взаимодействия через Telegram-бота. Реализовать авторизацию, хранение данных в базе и обработку задач с клиентской и серверной сторон.

Задачи, решаемые в ходе работы:

1. Создание пользовательского интерфейса для авторизации и работы со списком задач.
2. Реализация серверной части на Node.js с обработкой запросов от клиента.
3. Работа с базой данных SQL для хранения информации о пользователях и задачах.
4. Реализация Telegram-бота, способного выполнять команды по добавлению, удалению, редактированию и отображению задач.
5. Обеспечение взаимодействия между компонентами системы и отладка приложения.

Пример работы программы:

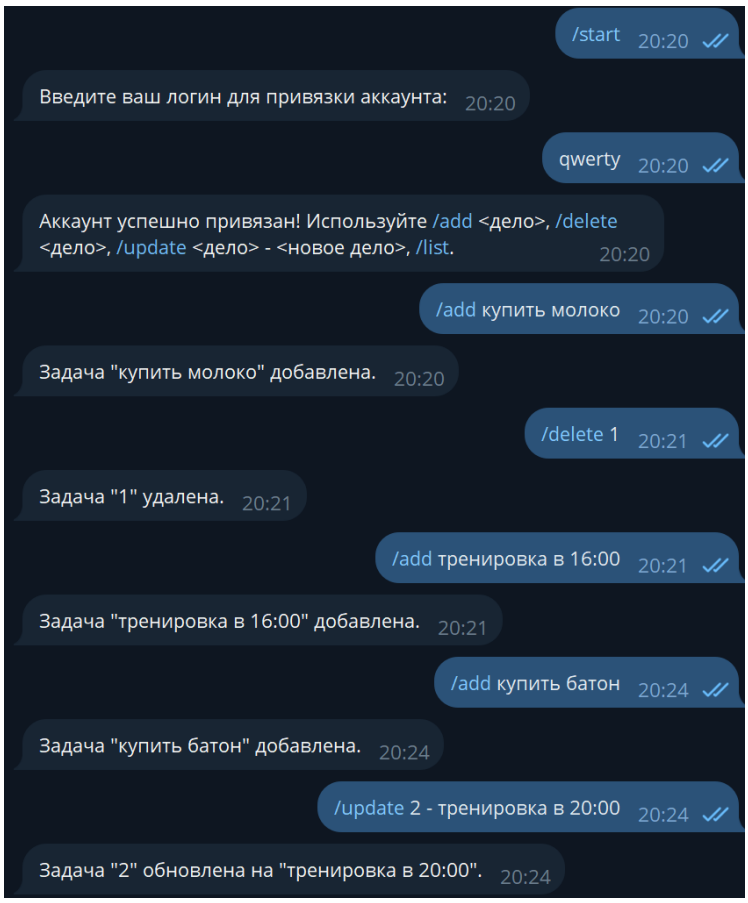

Рисунок 1 – интеграция телеграмма

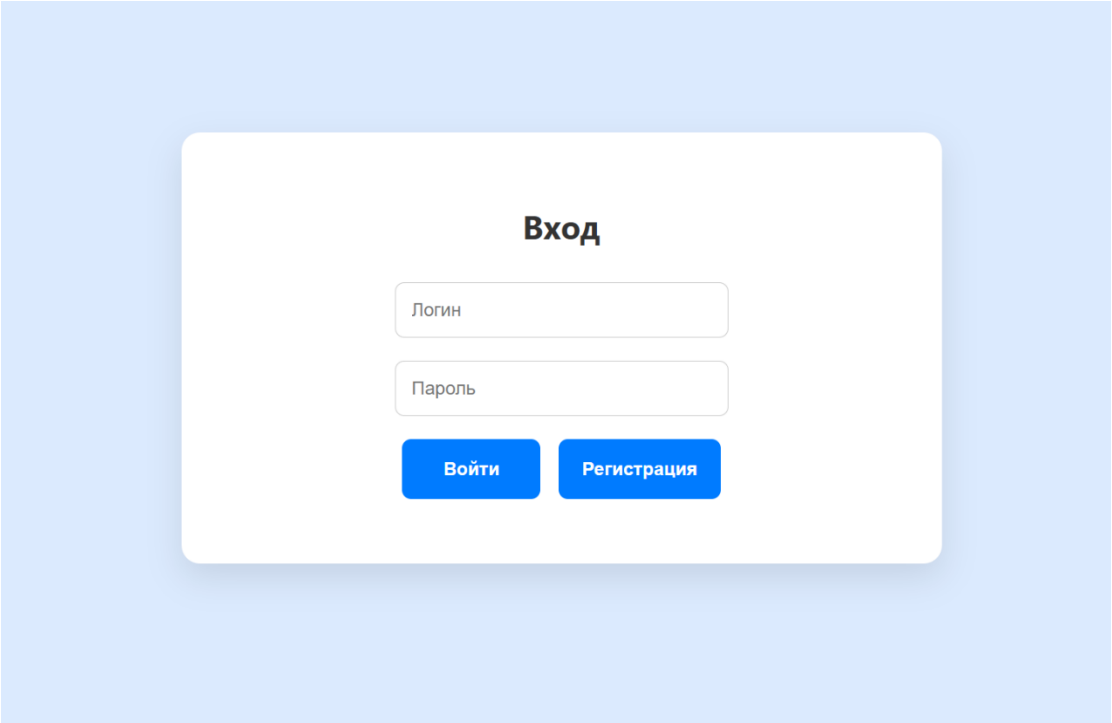

Рисунок 2 – авторизация/регистрация

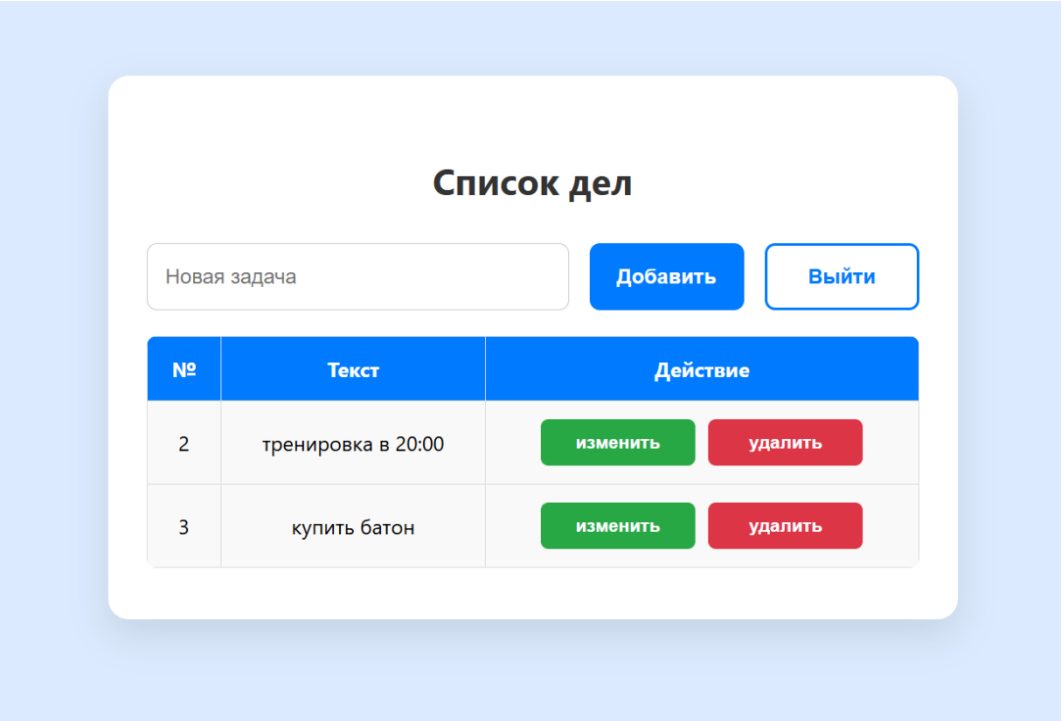

Рисунок 3 – интерфейс

Проблемы, выявленные по ходу выполнения работы:

1. Конфликт версий Node.js и модулей

Суть: при установке зависимостей через npm install возникали ошибки несовместимости модулей.

Причина: проект написан под другую версию Node.js, или пакеты устарели.

Решение:

Обновление/понижение версии Node.js с помощью nvm (Node Version Manager).

Удаление node_modules и package-lock.json, повторная установка зависимостей.

rm -rf node_modules package-lock.json

npm install

2. Сессии не сохраняются между запросами

Суть: после входа пользователь не остаётся авторизованным.

Причина: не настроено хранилище сессий или забыто подключение express-session.

Решение:

Убедиться в использовании app.use(session({...})) с секретом и настройками.

Проверить, поддерживает ли сервер куки (например, в Telegram нет).

3. База данных SQLite не создаётся автоматически

Суть: при первом запуске проект не создаёт файл .db или таблицы.

Причина: нет прав на запись в папке или не вызывается initDatabase().

Решение:

Убедиться, что initDatabase() вызывается при запуске.

Проверить пути к файлу базы и разрешения на запись.

4. Telegram-бот не отвечает

Суть: бот не реагирует на команды или вылетает.

Причина: неверный токен, ошибка в коде или блокировка ngrok.

Решение:

Проверить токен в BotFather.

Запустить ngrok http 3000 и указать правильный webhook.

Использовать bot.onText правильно и обрабатывать msg.text.

5. Ошибка при выполнении SQL-скрипта

Суть: mysql -u root -p < db.sql не работает в PowerShell.

Причина: PowerShell не поддерживает такой синтаксис.

Решение:

Выполнять скрипт через CMD.

Или использовать: Get-Content db.sql | mysql -u root -p

6. Кнопки "Изменить" и "Удалить" не работают

Суть: события не срабатывают на динамически созданных кнопках.

Причина: обработчики событий навешиваются до загрузки задач.

Решение:

Создать функцию attachEventListeners() и вызывать её после loadTasks().

Или использовать делегирование событий (event delegation).

7. Разные пользователи видят общие задачи

Причина: при выборке задач не учитывается user_id.

Решение:

В SQL-запросах всегда использовать фильтрацию по user_id:

SELECT * FROM tasks WHERE user_id = ?

8. Неточные промпты для ИИ

Причина: неточные формулировки задач для ИИ (например, "добавить Telegram-бота") приводили к неполным решениям, которые не учитывали всех требований (например, необходимость авторизации через /start).

Решение: уточнение запросов с чётким описанием ожидаемого поведения, например, указание, что бот должен работать только для привязанных пользователей.

Работа процессов:

1. Механизм авторизации

1.1 Регистрация

1. Студент вводит желаемый логин и пароль.
2. Пароль шифруется методом bcrypt.hash() (с солью по умолчанию = 10).
3. Полученные данные заносятся в таблицу **users**.

SQL-запрос

sql

```
INSERT INTO users (username, password)
VALUES (?, ?);
```

1.2 Вход в систему

1. Пользователь указывает логин и пароль во форме авторизации.
2. Сервер по логину ищет запись в БД и проверяет пароль через bcrypt.compare().
3. При успешной проверке создаётся сессия: объект req.session.user содержит id и username.

Код

js

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;
  const user = await getUserFromDB(username);

  if (user && await bcrypt.compare(password, user.password)) {
    req.session.user = { id: user.id, username: user.username };
    res.json({ success: true });
  } else {
    res.status(401).json({ error: 'Неверный логин или пароль' });
  }
});
```

1.3 Выход

Сброс сессионных данных и перенаправление на страницу входа:

```
js
app.get('/logout', (req, res) => {
  req.session.destroy();
  res.redirect('/login');
});
```

2. Интеграция с Telegram:

2.1 Привязка Telegram-аккаунта

1. Пользователь отправляет боту команду /start.
2. Бот запрашивает логин из веб-приложения.
3. При совпадении логина Telegram-ID фиксируется в users.telegram_id.

Код:

```
bot.onText(/\/start/, async (msg) => {
  const chatId = msg.chat.id;
  bot.sendMessage(chatId, 'Введите ваш логин:');
  bot.once('message', async (msg) => {
    const username = msg.text.trim();
    await linkTelegramId(username, chatId.toString());
  });
});
```

2.2 Управление задачами из чата

Добавление

/add Купить молоко

→ выполняется

sql

INSERT INTO tasks (text, user_id) VALUES (?, ?);

Удаление

/delete 3

→ запись с id = 3 удаляется у текущего пользователя.

Просмотр списка

/list

→ бот выводит все задачи, выбранные запросом

sql

SELECT id, text FROM tasks WHERE user_id = ?;

Каждая команда проверяет, чтобы Telegram-ID был привязан именно к текущему пользователю; в противном случае бот отвечает сообщением о необходимости авторизации.

Вывод:

Разработанное приложение успешно соответствует поставленным требованиям. Реализована стабильная авторизация пользователей и полноценная интеграция с Telegram-ботом, что создаёт базу для последующей доработки и расширения функционала. В ходе выполнения проекта возникали сложности, связанные с тем, что ИИ не всегда точно интерпретировал сформулированные запросы. Эти трудности удалось преодолеть путём уточнения формулировок и последовательного диалога с ИИ, что в конечном итоге позволило получить корректные решения и завершить реализацию.

Приложение 1 – Код index.js

```
const express = require('express');
const mysql = require('mysql2/promise');
const session = require('express-session');
const bcrypt = require('bcrypt');
const path = require('path');
const fs = require('fs').promises;
const TelegramBot = require('node-telegram-bot-api');
const app = express();
const port = 3000;
const telegram_token = '8136358504:AAFYF6NFQtOZ4HvzAfOXcveXivkR_a44Zv4';
const bot = new TelegramBot(telegram_token, { polling: true });
const dbConfig = {
  host: 'Localhost',
  user: 'root',
```

```
  password: 'qwe123!@#',
  database: 'todolist',
};

app.use(express.json());
app.use(session({
  secret: 'your-secret-key',
  resave: false,
  saveUninitialized: false,
  cookie: { secure: false }
}));

async function initDatabase() {
  try {
    const connection = await mysql.createConnection({
      host: dbConfig.host,
      user: dbConfig.user,
      password: dbConfig.password
    });
    console.log('Connected to MySQL server');

    await connection.query('CREATE DATABASE IF NOT EXISTS todolist');
    await connection.query('USE todolist');
    console.log('Database todolist selected');

    await connection.query(`
      CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(50) NOT NULL UNIQUE,
        password VARCHAR(255) NOT NULL,
        telegram_id VARCHAR(50) UNIQUE
      )
    `);
    console.log('Table users created');

    await connection.query('DROP TABLE IF EXISTS items');
    await connection.query(`
      CREATE TABLE items (
        id INT AUTO_INCREMENT PRIMARY KEY,
        text VARCHAR(255) NOT NULL,
        user_id INT,
        FOREIGN KEY (user_id) REFERENCES users(id)
      )
    `);
    console.log('Table items created');

    await connection.end();
    console.log('Database and tables initialized successfully');
  } catch (error) {
    console.error('Error initializing database:', error);
    throw error;
  }
```

```
}

async function retrieveListItems(userId) {
  try {
    const connection = await mysql.createConnection(dbConfig);
    const query = 'SELECT id, text FROM items WHERE user_id = ?';
    const [rows] = await connection.execute(query, [userId]);
    await connection.close();
    return rows;
  } catch (error) {
    console.error('Ошибка при получении элементов:', error);
    throw error;
  }
}

async function addListItem(text, userId) {
  try {
    const connection = await mysql.createConnection(dbConfig);
    const query = 'INSERT INTO items (text, user_id) VALUES (?, ?)';
    const [result] = await connection.execute(query, [text, userId]);
    await connection.close();
    return { id: result.insertId, text };
  } catch (error) {
    console.error('Ошибка при добавлении элемента:', error);
    throw error;
  }
}

async function deleteListItem(id, userId) {
  try {
    const connection = await mysql.createConnection(dbConfig);
    const query = 'DELETE FROM items WHERE id = ? AND user_id = ?';
    const [result] = await connection.execute(query, [id, userId]);
    await connection.close();
    return result.affectedRows > 0;
  } catch (error) {
    console.error('Ошибка при удалении элемента:', error);
    throw error;
  }
}

async function updateListItem(id, newText, userId) {
  try {
    const connection = await mysql.createConnection(dbConfig);
    const query = 'UPDATE items SET text = ? WHERE id = ? AND user_id = ?';
    const [result] = await connection.execute(query, [newText, id, userId]);
    await connection.close();
    return result.affectedRows > 0;
  } catch (error) {
    console.error('Ошибка при обновлении элемента:', error);
    throw error;
  }
```

```
}

async function getUserByTelegramId(telegramId) {
  try {
    const connection = await mysql.createConnection(dbConfig);
    const [rows] = await connection.execute('SELECT * FROM users WHERE telegram_id = ?',
[telegramId]);
    await connection.close();
    return rows[0];
  } catch (error) {
    console.error('Ошибка при получении пользователя:', error);
    throw error;
  }
}

async function linkTelegramId(username, telegramId) {
  try {
    const connection = await mysql.createConnection(dbConfig);
    const [userRows] = await connection.execute('SELECT * FROM users WHERE username = ?',
[username]);

    if (userRows.length === 0) {
      await connection.close();
      return { success: false, message: `Пользователь с логином "${username}" не найден.
Зарегистрируйтесь на сайте.` };
    }

    const [telegramRows] = await connection.execute('SELECT * FROM users WHERE telegram_id
= ?', [telegramId]);

    if (telegramRows.length > 0) {
      await connection.close();
      return { success: false, message: `Этот Telegram ID уже привязан к пользователю
"${telegramRows[0].username}".` };
    }

    await connection.execute('UPDATE users SET telegram_id = ? WHERE username = ?',
[telegramId, username]);
    await connection.close();

    console.log(`Telegram ID ${telegramId} успешно привязан к пользователю ${username}`);
    return {
      success: true,
      message: 'Аккаунт успешно привязан! Используйте /add <дело>, /delete <дело>, /update
<дело> - <новое дело>, /list.'
    };
  } catch (error) {
    console.error('Ошибка при привязке Telegram ID:', error);
    return { success: false, message: 'Ошибка сервера при привязке Telegram ID. Попробуйте
позже.' };
  }
}
```

```
async function getHtmlRows(userId) {
  const todoItems = await retrieveListItems(userId);
  return todoItems.map(item => `
    <tr>
      <td>${item.id}</td>
      <td class="text-cell" data-id="${item.id}">${item.text}</td>
      <td>
        <button class="edit-btn" data-id="${item.id}">изменить</button>
        <button class="delete-btn" data-id="${item.id}">удалить</button>
      </td>
    </tr>
  `).join('');
}

function isAuthenticated(req, res, next) {
  if (req.session.user) {
    next();
  } else {
    res.redirect('/login');
  }
}

app.get('/login', async (req, res) => {
  const html = await fs.readFile(path.join(__dirname, 'index.html'), 'utf8');
  res.send(html.replace('{{rows}}', ''));
});

app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  if (!username || !password || username.trim().length === 0 || password.trim().length === 0) {
    return res.status(400).json({ error: 'Логин и пароль не могут быть пустыми' });
  }

  try {
    const connection = await mysql.createConnection(dbConfig);
    const [rows] = await connection.execute('SELECT * FROM users WHERE username = ?',
[username]);
    await connection.close();

    if (rows.length === 0) {
      return res.status(401).json({ error: 'Пользователь не найден' });
    }

    const user = rows[0];
    const match = await bcrypt.compare(password, user.password);

    if (match) {
      req.session.user = { id: user.id, username: user.username };
      return res.json({ message: 'Успешный вход' });
    } else {
```

```
      return res.status(401).json({ error: 'Неверный пароль' });
    }
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Ошибка сервера' });
  }
});

app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  if (!username || !password || username.trim().length === 0 || password.trim().length === 0) {
    return res.status(400).json({ error: 'Логин и пароль не могут быть пустыми' });
  }

  try {
    const connection = await mysql.createConnection(dbConfig);
    const hashedPassword = await bcrypt.hash(password, 10);
    await connection.execute('INSERT INTO users (username, password) VALUES (?, ?)',
[username, hashedPassword]);
    await connection.close();
    return res.json({ message: 'Пользователь зарегистрирован' });
  } catch (error) {
    console.error(error);
    return res.status(400).json({ error: 'Пользователь уже существует или ошибка сервера' });
  }
});

app.get('/logout', (req, res) => {
  req.session.destroy();
  res.redirect('/login');
});

app.get('/', isAuthenticated, async (req, res) => {
  try {
    const html = await fs.readFile(path.join(__dirname, 'index.html'), 'utf8');
    const processedHtml = html.replace('{{rows}}', await getHtmlRows(req.session.user.id));
    res.send(processedHtml);
  } catch (err) {
    console.error(err);
    res.status(500).send('Ошибка загрузки страницы');
  }
});

app.post('/add', isAuthenticated, async (req, res) => {
  const { text } = req.body;

  if (!text || typeof text !== 'string' || text.trim() === '') {
    res.status(400).json({ error: 'Некорректный или отсутствующий текст' });
    return;
  }
```

```
  try {
    const newItem = await addListItem(text.trim(), req.session.user.id);
    res.json(newItem);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Не удалось добавить элемент' });
  }
});

app.delete('/delete', isAuthenticated, async (req, res) => {
  const id = req.query.id;

  if (!id || isNaN(id)) {
    res.status(400).json({ error: 'Некорректный или отсутствующий ID' });
    return;
  }

  try {
    const success = await deleteListItem(id, req.session.user.id);
    if (success) {
      res.json({ message: 'Элемент удален' });
    } else {
      res.status(404).json({ error: 'Элемент не найден' });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Не удалось удалить элемент' });
  }
});

app.put('/update', isAuthenticated, async (req, res) => {
  const id = req.query.id;
  const { text } = req.body;

  if (!id || isNaN(id) || !text || typeof text !== 'string' || text.trim() === '') {
    res.status(400).json({ error: 'Некорректный или отсутствующий ID или текст' });
    return;
  }

  try {
    const success = await updateListItem(id, text.trim(), req.session.user.id);
    if (success) {
      res.json({ message: 'Элемент обновлен', text });
    } else {
      res.status(404).json({ error: 'Элемент не найден' });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Не удалось обновить элемент' });
  }
});
```

```
//бот тг
bot.onText(/\/start/, async (msg) => {
  const chatId = msg.chat.id;
  bot.sendMessage(chatId, 'Введите ваш логин для привязки аккаунта:');

  bot.once('message', async (msg) => {
    const username = msg.text.trim();
    const result = await linkTelegramId(username, chatId.toString());
    bot.sendMessage(chatId, result.message);
  });
});

bot.onText(/\/add (.+)/, async (msg, match) => {
  const chatId = msg.chat.id;
  const text = match[1].trim();
  const user = await getUserByTelegramId(chatId.toString());

  if (!user) {
    bot.sendMessage(chatId, 'Ваш Telegram ID не привязан к аккаунту. Используйте /start для
привязки.');
    return;
  }

  try {
    await addListItem(text, user.id);
    bot.sendMessage(chatId, `Задача "${text}" добавлена.`);
  } catch (error) {
    bot.sendMessage(chatId, 'Ошибка при добавлении задачи.');
  }
});

bot.onText(/\/delete (.+)/, async (msg, match) => {
  const chatId = msg.chat.id;
  const text = match[1].trim();
  const user = await getUserByTelegramId(chatId.toString());

  if (!user) {
    bot.sendMessage(chatId, 'Ваш Telegram ID не привязан к аккаунту. Используйте /start для
привязки.');
    return;
  }

  try {
    const success = await deleteListItem(text, user.id);
    if (success) {
      bot.sendMessage(chatId, `Задача "${text}" удалена.`);
    } else {
      bot.sendMessage(chatId, `Задача "${text}" не найдена.`);
    }
  } catch (error) {
    bot.sendMessage(chatId, 'Ошибка при удалении задачи.');
  }
```

```
});

bot.onText(/\/update (.+) - (.+)/, async (msg, match) => {
  const chatId = msg.chat.id;
  const oldText = match[1].trim();
  const newText = match[2].trim();
  const user = await getUserByTelegramId(chatId.toString());

  if (!user) {
    bot.sendMessage(chatId, 'Ваш Telegram ID не привязан к аккаунту. Используйте /start для
привязки.');
    return;
  }

  try {
    const success = await updateListItem(oldText, newText, user.id);
    if (success) {
      bot.sendMessage(chatId, `Задача "${oldText}" обновлена на "${newText}".`);
    } else {
      bot.sendMessage(chatId, `Задача "${oldText}" не найдена.`);
    }
  } catch (error) {
    bot.sendMessage(chatId, 'Ошибка при обновлении задачи.');
  }
});

bot.onText(/\/list/, async (msg) => {
  const chatId = msg.chat.id;
  const user = await getUserByTelegramId(chatId.toString());

  if (!user) {
    bot.sendMessage(chatId, 'Ваш Telegram ID не привязан к аккаунту. Используйте /start для
привязки.');
    return;
  }

  try {
    const items = await retrieveListItems(user.id);
    if (items.length === 0) {
      bot.sendMessage(chatId, 'Список задач пуст.');
      return;
    }

    const message = items.map(item => `${item.id}. ${item.text}`).join('\n');
    bot.sendMessage(chatId, `Ваши задачи:\n${message}`);
  } catch (error) {
    bot.sendMessage(chatId, 'Ошибка при получении списка задач.');
  }
});

initDatabase().then(() => {
  app.listen(port, () => console.log(`Сервер запущен на порту ${port}`));
```

```
}).catch(err => {
  console.error('Failed to initialize database and start server:', err);
  process.exit(1);
});
```

Приложение 2 – Код программы index.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Список дел</title>
  <style>
    body {
      margin: 0;
      padding: 0;
      font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
      background: #dbeafe;
      display: flex;
      align-items: center;
      justify-content: center;
      min-height: 100vh;
    }

    .card {
      background: white;
      padding: 40px 30px;
      border-radius: 16px;
      box-shadow: 0 12px 30px rgba(0,0,0,0.1);
      width: 100%;
      max-width: 600px;
    }

    h2 {
      text-align: center;
      margin-bottom: 30px;
      font-size: 28px;
      color: #333;
    }

    #authContainer form {
      display: flex;
      flex-direction: column;
      align-items: center;
    }

    #authContainer input {
      max-width: 260px;
      width: 100%;
      padding: 14px;
```

```
  margin-bottom: 20px;
  border: 1px solid #ccc;
  border-radius: 8px;
  font-size: 16px;
}

.button-group {
  display: flex;
  justify-content: center;
  gap: 16px;
}

button {
  padding: 0 20px;
  min-width: 120px;
  height: 52px;
  display: flex;
  align-items: center;
  justify-content: center;
  background: #007bff;
  color: white;
  border: none;
  border-radius: 8px;
  font-weight: bold;
  font-size: 16px;
  cursor: pointer;
  transition: 0.3s ease;
}

button:hover {
  background: #0056b3;
}

.logout-btn {
  background: white;
  border: 2px solid #007bff;
  color: #007bff;
  font-weight: bold;
  transition: .3s ease;
}

.logout-btn:hover {
  background: #007bff;
  color: white;
}

.todo-container {
  margin-top: 40px;
}

.task-input-group {
  display: flex;
```

```
  justify-content: center;
  align-items: stretch;
  gap: 16px;
  margin-bottom: 20px;
}

.task-input-group input {
  flex: 1 1 auto;
  padding: 14px;
  border: 1px solid #ccc;
  border-radius: 8px;
  font-size: 16px;
}

table {
  width: 100%;
  border-collapse: collapse;
  background: #f9f9f9;
  border-radius: 8px;
  overflow: hidden;
}

th, td {
  padding: 14px;
  text-align: center;
  border: 1px solid #ddd;
}

th {
  background: #007bff;
  color: white;
}

.edit-btn, .delete-btn {
  display: inline-block;
  width: 80px;
  height: 36px;
  line-height: 36px;
  padding: 0;
  font-size: 14px;
  font-weight: bold;
  border-radius: 6px;
  color: white;
  cursor: pointer;
}

.edit-btn {
  background-color: #28a745;
  margin-right: 6px;
}

.delete-btn {
```

```
    background-color: #dc3545;
   }

   .error {
    color: red;
    text-align: center;
    font-weight: bold;
    margin-top: 10px;
   }
  </style>
</head>
<body>
  <div class="card" id="authContainer">
   <h2>Вход</h2>
   <form onsubmit="event.preventDefault();">
    <input type="text" id="username" placeholder="Логин">
    <input type="password" id="password" placeholder="Пароль">
   </form>
   <div class="button-group">
    <button onclick="login()">Войти</button>
    <button onclick="register()">Регистрация</button>
   </div>
   <p id="authError" class="error"></p>
  </div>

  <div class="card todo-container" id="todoContainer" style="display: none;">
   <h2>Список дел</h2>
   <div class="task-input-group">
    <input type="text" id="taskInput" placeholder="Новая задача">
    <button onclick="addTask()">Добавить</button>
    <button class="logout-btn" onclick="logout()">Выйти</button>
   </div>
   <p id="taskError" class="error"></p>
   <table id="taskTable">
    <tr>
     <th>№</th>
     <th>Текст</th>
     <th>Действие</th>
    </tr>
    {{rows}}
   </table>
  </div>

  <script>
   async function showError(id, message) {
    const el = document.getElementById(id);
    el.textContent = message;
    el.style.display = 'block';
    setTimeout(() => { el.style.display = 'none'; }, 3000);
   }

   async function login() {
```

```
  const username = document.getElementById('username').value.trim();
  const password = document.getElementById('password').value.trim();
  if (!username || !password) return showError('authError', 'Заполните все поля');
  try {
    const res = await fetch('/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ username, password })
    });
    const data = await res.json();
    if (res.ok) {
      document.getElementById('authContainer').style.display = 'none';
      document.getElementById('todoContainer').style.display = 'block';
      loadTasks();
    } else {
      showError('authError', data.error || 'Ошибка входа');
    }
  } catch {
    showError('authError', 'Сервер недоступен');
  }
}

async function register() {
  const username = document.getElementById('username').value.trim();
  const password = document.getElementById('password').value.trim();
  if (!username || !password) return showError('authError', 'Заполните все поля');
  try {
    const res = await fetch('/register', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ username, password })
    });
    const data = await res.json();
    showError('authError', data.message || data.error);
  } catch {
    showError('authError', 'Сервер недоступен');
  }
}

async function logout() {
  await fetch('/logout');
  document.getElementById('todoContainer').style.display = 'none';
  document.getElementById('authContainer').style.display = 'block';
}

async function loadTasks() {
  const res = await fetch('/');
  const html = await res.text();
  const parser = new DOMParser();
  const doc = parser.parseFromString(html, 'text/html');
  const newTable = doc.querySelector('#taskTable');
  document.querySelector('#taskTable').innerHTML = newTable.innerHTML;
```

```
  attachEventListeners();
}

async function addTask() {
  const text = document.getElementById('taskInput').value.trim();
  if (!text) return showError('taskError', 'Введите задачу');
  try {
    const res = await fetch('/add', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ text })
    });
    const data = await res.json();
    if (res.ok) loadTasks();
    else showError('taskError', data.error || 'Ошибка');
  } catch {
    showError('taskError', 'Ошибка сервера');
  }
}

async function deleteTask(id) {
  try {
    const response = await fetch(`/delete?id=${id}`, {
      method: 'DELETE'
    });
    const data = await response.json();
    if (response.ok) {
      document.querySelector(`tr td[data-id="${id}"]`).parentElement.remove();
    } else {
      showError('taskError', data.error || 'Ошибка удаления');
    }
  } catch {
    showError('taskError', 'Ошибка сервера');
  }
}

async function editTask(id, textCell) {
  const newText = prompt('Введите новый текст задачи:', textCell.textContent);
  if (newText === null || newText.trim() === '') return;
  try {
    const response = await fetch(`/update?id=${id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ text: newText.trim() })
    });
    const data = await response.json();
    if (response.ok) {
      textCell.textContent = data.text;
    } else {
      showError('taskError', data.error || 'Ошибка редактирования');
    }
  } catch {
```

```
        showError('taskError', 'Ошибка сервера');
      }
    }

    function attachEventListeners() {
      document.querySelectorAll('.delete-btn').forEach(button => {
        button.onclick = () => deleteTask(button.dataset.id);
      });

      document.querySelectorAll('.edit-btn').forEach(button => {
        button.onclick = () => {
          const textCell = document.querySelector(`td[data-id="${button.dataset.id}"]`);
          editTask(button.dataset.id, textCell);
        };
      });
    }

    document.addEventListener('DOMContentLoaded', attachEventListeners);
  </script>
</body>
</html>
```

Приложение 3 – Код программы – init.sql

```
CREATE DATABASE IF NOT EXISTS todolist;

USE todolist;

CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    telegram_id VARCHAR(50) UNIQUE
);

CREATE TABLE IF NOT EXISTS items (
    id INT AUTO_INCREMENT PRIMARY KEY,
    text VARCHAR(255) NOT NULL,
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```