Geekbrains

**Архитектура и разработка приложения**

**для организации заметок на основе Node.js и React**

Программа:

**Разработчик**

Специализация:

**Frontend разработка (React)**

**Петкевич**

**Виктор**

**Алексеевич**

Нижний Новгород

2024 год

**Содержание**

[Введение 4](#_Toc169530876)

[Составление требований к функционалу приложения 6](#_Toc169530877)

[Создание заметок 6](#_Toc169530878)

[Закрепление заметок 8](#_Toc169530879)

[ToDo-листы 8](#_Toc169530880)

[Поиск заметок 9](#_Toc169530881)

[Настройки приложения 9](#_Toc169530882)

[Выбор технологий для разработки приложения 10](#_Toc169530883)

[Разработка backend части приложения 13](#_Toc169530884)

[Просмотр заметок 16](#_Toc169530885)

[Добавление заметки 16](#_Toc169530886)

[Изменение заметки 17](#_Toc169530887)

[Закрепление заметки 18](#_Toc169530888)

[Удаление заметки 18](#_Toc169530889)

[Получение настроек 18](#_Toc169530890)

[Изменение настроек 19](#_Toc169530891)

[Разработка frontend части приложения на React 20](#_Toc169530892)

[calls.js 23](#_Toc169530893)

[AppContext.js 25](#_Toc169530894)

[NotesList 26](#_Toc169530895)

[Note 27](#_Toc169530896)

[NoteMenu 28](#_Toc169530897)

[NoteDialog 29](#_Toc169530898)

[Controls 29](#_Toc169530899)

[Settings 30](#_Toc169530900)

[Search 30](#_Toc169530901)

[NewNote 31](#_Toc169530902)

[StatusMessage 31](#_Toc169530903)

[Интеграция backend и frontend частей 32](#_Toc169530904)

[Перспективы развития приложения 34](#_Toc169530905)

[Заключение 35](#_Toc169530906)

[Список использованных источников 37](#_Toc169530907)

[Приложение 1 38](#_Toc169530908)

[Приложение 2 41](#_Toc169530909)

[App.js 41](#_Toc169530910)

[calls.js 42](#_Toc169530911)

[AppContext.js 44](#_Toc169530912)

[NotesList 46](#_Toc169530913)

[Note 47](#_Toc169530914)

[renderers.js 48](#_Toc169530915)

[ToDoEl 49](#_Toc169530916)

[NoteMenu 50](#_Toc169530917)

[NoteDialog 52](#_Toc169530918)

[Controls 54](#_Toc169530919)

[Settings 55](#_Toc169530920)

[Search 58](#_Toc169530921)

[NewNote 59](#_Toc169530922)

[StatusMessage 59](#_Toc169530923)

[Приложение 3 61](#_Toc169530924)

# Введение

Ведение заметок (или конспектирование; англ. note-taking) — практика записи информации из разных источников и платформ. Делая заметки, автор записывает суть информации, освобождая свой разум от необходимости все вспоминать. Заметки обычно берутся из временного источника, такого как устная дискуссия на собрании или лекция (заметки собрания обычно называются протоколами), и в этом случае заметки могут быть единственной записью события. С появлением письменности и ростом грамотности заметки традиционно и, почти всегда, делались от руки (часто в блокнотах), но появление программного обеспечения для ведения заметок сделало ведение цифровых заметок возможным и широко распространенным. Ведение заметок является основополагающим навыком в управлении личными знаниями.

Ведение заметок - это важная составляющая часть истории человечества и научного развития. Древние греки разработали гипомнему — личные записи на важные темы. В эпоху Возрождения и начале Нового времени студенты учились делать заметки в школах, академиях и университетах, часто создавая красивые книги, которые затем служили справочниками после окончания учёбы. В доцифровые времена люди использовали множество видов записных книжек, в том числе книги, черновики и заметки на полях. Философ Джон Локк разработал и опубликовал популярную систему индексации, которая послужила моделью для обычных книг и вдохновила, по крайней мере, на создание десяти различных опубликованных изданий шаблонов книг в Европе и Америке, включая Карманный ежедневник Белла, сформированый, в основном, на принципах рекомендованных и практикуемых господином Локком (Лондон, 1770 г.).

В современном мире информационных технологий доступ к информации стал одним из ключевых факторов успешной деятельности. В связи с этим возросла потребность в эффективных средствах для организации и управления информацией. В рамках данного проекта было принято решение разработать приложение для ведения заметок, которое представляет собой универсальный инструмент для хранения, организации и обмена информацией.

С развитием цифровых технологий и увеличением объема информации, которую мы ежедневно получаем, возникает необходимость в эффективных инструментах для ее обработки и хранения. В наше время люди сталкиваются с большим объемом информации, будь то личные заметки, задачи, идеи, списки дел или просто фрагменты информации, которые им необходимо сохранить для последующего использования. Стандартные инструменты, такие как бумажные блокноты или ежедневники, уже не всегда удовлетворяют потребности в удобном, быстром и эффективном ведении заметок.

Целью данного проекта является создание удобного и интуитивно понятного приложения для ведения заметок, которое позволит пользователям эффективно организовывать свою информацию, легко находить необходимые записи и удобно работать с ними.

# Составление требований к функционалу приложения

Разрабатываемое приложение должно предоставлять простой и удобный функционал для создания, редактирования и удаления заметок. Интерфейс должен быть интуитивно понятен любому пользователю. Так же необходимо предоставить возможность настройки внешнего вида приложения.

Некоторые особенности функционала:

## Создание заметок

Заметки должны вестись с использованием языка разметки Markdown. Markdown - облегчённый язык разметки, созданный с целью обозначения форматирования в простом тексте, с максимальным сохранением его читаемости человеком, и пригодный для машинного преобразования в языки для продвинутых публикаций (HTML, Rich Text и других).

Язык разметки Markdown широко распространён в вебе. На нём пишут readme-файлы и документацию, а его принципы работы используют для оформления сообщений и публикаций в мессенджерах и социальных сетях.

Markdown легко конвертируется в HTML, открывается во всех текстовых редакторах и легко читается даже в виде исходного кода. При этом писать на нём куда проще, чем на других языках разметки: HTML, XML, TeX и так далее.

Например, следующая разметка, написанная на языке Markdown

**# Введение в браузерные события**

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

**#### События мыши:**

- click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).

- contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.

- mouseover / mouseout – когда мышь наводится на / покидает элемент.

- mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.

- mousemove – при движении мыши.

**#### События на элементах управления:**

- submit – пользователь отправил форму &lt;form&gt;.

- focus – пользователь фокусируется на элементе, например нажимает на &lt;input&gt;.

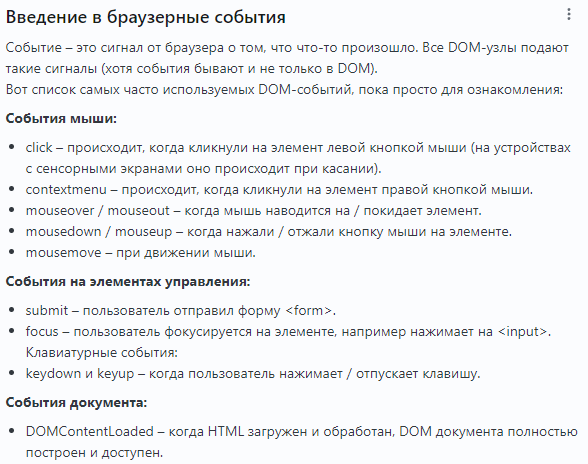
Клавиатурные события:

- keydown и keyup – когда пользователь нажимает / отпускает клавишу.

**#### События документа:**

- DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

будет выглядеть так:



Так же необходимо реализовать подсветку синтаксиса, если в заметку будет добавлен исходный код для улучшения визуального восприятия.

## Закрепление заметок

Необходимо реализовать возможность выделения особенно важных заметок. Такие заметки должны визуально отличаться от остальных. Так же должна быть возможность закреплять такие заметки в верхней части общего списка.

## ToDo-листы

Необходимо реализовать возможность ведения списков задач. В разметке Markdown они выглядят так:

- [x] Design

- [ ] Mockups

- [x] Development

- [ ] Unit testing

- [x] QAi

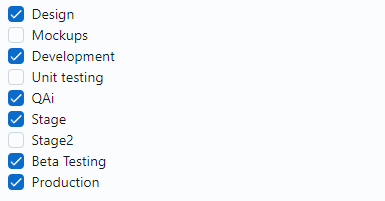
- [x] Stage

- [ ] Stage2

- [x] Beta Testing

- [x] Production

А в результате рендеринга имеют следующий вид:



Важно иметь возможность ставить и снимать отметку о выполнении задачи не редактированием исходного кода Markdown, а нажатием соответствующего пункта непосредственно в интерфейсе приложения.

## Поиск заметок

Требуется реализовать поиск по всем заметкам. Для этого в интерфейсе должно быть поле для ввода текста. По мере ввода текста, общий список заметок должен обновляться, в списке должны оставаться только те заметки, в содержании которых присутствует введенный текст.

## Настройки приложения

Необходимо реализовать возможность изменения следующих параметров приложения:

* Количество колонок с заметками. Возможные варианты: 1, 2 или 3 колонки;
* Закрепление избранных заметок в верхней части общего списка. При отключении этой настройки все заметки будут по умолчанию отсортированы по порядку добавления, но избранные заметки по-прежнему должны иметь визуальное выделение;
* Тема приложения. Возможные варианты: светлая, темная и системная;
* Размер рабочей области приложения: Возможные варианты: маленький, средний и вся доступная ширина экрана.

Так же необходимо реализовать сохранение настроек.

# Выбор технологий для разработки приложения

Для разработки дипломного проекта из множества популярных программных решений были выбраны Node.js и React. Выбор обусловлен высокой производительностью, эффективностью, модульностью и поддержкой активного сообщества разработчиков. Эти технологии предоставляют широкий спектр инструментов для создания современных веб-приложений и обеспечивают хорошие перспективы для будущей карьеры в области веб-разработки. Рассмотрим подробнее:

**1. Популярность и активное сообщество**

Node.js и React являются одними из самых популярных технологий в мире веб-разработки. Node.js, благодаря своей асинхронной природе, позволяет строить масштабируемые сетевые приложения. Он основан на JavaScript, что делает его привлекательным для разработчиков, уже знакомых с этим языком. Сообщество Node.js активно участвует в разработке модулей, обновлениях и поддержке, что делает его надежным выбором для разработки серверной части приложений.

React, в свою очередь, стал одним из самых популярных фреймворков для разработки пользовательского интерфейса. Его компонентный подход позволяет разбивать пользовательский интерфейс на множество маленьких и переиспользуемых компонентов, что делает код более чистым и легко поддерживаемым. Активное сообщество React постоянно предлагает новые идеи, решения и инструменты для улучшения процесса разработки.

**2. Эффективность и производительность**

Node.js обеспечивает высокую производительность за счёт своей асинхронной природы. Он использует однопоточную модель событийного цикла, что позволяет обрабатывать множество запросов без блокирования потока выполнения. Это особенно важно для приложений с большой нагрузкой, таких как социальные сети или онлайн-игры.

React обеспечивает эффективную отрисовку пользовательского интерфейса благодаря виртуальному DOM. Виртуальный DOM позволяет минимизировать количество манипуляций с реальным DOM, что приводит к улучшению производительности приложения.

**3. Односторонний поток данных**

React использует односторонний поток данных, что делает управление состоянием приложения более предсказуемым. Данные передаются от родительских компонентов к дочерним через явно определенные пропсы, а изменения состояния обрабатываются с помощью функций обратного вызова. Это упрощает отслеживание изменений и уменьшает вероятность возникновения ошибок.

**4. Компонентный подход**

React основан на компонентах, что позволяет создавать модульный и масштабируемый код. Компоненты могут быть переиспользованы в различных частях приложения, что упрощает разработку и поддержку кода. Кроме того, компонентный подход способствует легкости тестирования и модификации кода.

**5. Современные технологии**

Использование Node.js и React позволит вам освоить современные технологии разработки, которые востребованы на рынке труда. Знание этих технологий может значительно повысить ваши шансы на трудоустройство в качестве веб-разработчика.

**6. Легкая масштабируемость**

Обе технологии легко масштабируются благодаря своей архитектуре. Node.js позволяет горизонтальное масштабирование за счет возможности создания нескольких экземпляров серверов, а React обеспечивает гибкость при добавлении новых компонентов и функциональности.

# Разработка backend части приложения

Для разработки бэкенд части приложения была выбрана технология Node.js, которая позволяет создавать высокопроизводительные и масштабируемые серверные приложения. Node.js — это кроссплатформенная среда выполнения JavaScript, которая позволяет запускать код на сервере, а не только в браузере. Это позволяет использовать язык программирования JavaScript для разработки как клиентской, так и серверной частей приложения.

Так же был выбран фреймворк Express, который является одним из самых популярных фреймворков для Node.js. Express - это минималистичный и гибкий фреймворк веб-приложений Node.js, который предоставляет набор функций для разработки веб- и мобильных приложений. Он существенно упрощает разработку веб-приложений на базе Node.js, позволяет настроить посредников для ответа на запросы HTTP, определяет таблицу маршрутизации, которая используется для выполнения различных действий на основе метода HTTP и URL-адреса и позволяет динамически создавать HTML-страницы на основе передачи аргументов шаблонам.

Для хранения данных приложения была выбрана система управления базами данных SQLite. SQLite представляет библиотеку, которая написанна на языке C (ANSI-C) и которая реализует движок реляционных баз данных.На сегодняшний день SQLite, возможно, самая используемая система баз данных. Так, ее бд можно найти в каждом устройстве на Android, iOS, Mac, Windows 10/11, ее используются большинство распространенных браузеров - Firefox, Chrome, Safari и т.д.

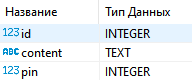
В отличие от других систем баз данных, как MS SQL Server, MySQL, Postgres и т.д., для SQLite не требуется сервер базы данных. SQLite представляет встраиваемый движок базы данных, который обращается напрямую к файлу базы данных на диске. Соответственно для работы с базами данных нам не надо явным образом устанавливать или как-то конфигурировать SQLite.

Серверная часть приложения на Node.js, используя фреймворк Express, предоставляет фронтенду API и маршруты для взаимодействия с базой данных SQLite.

API (Application Programming Interface) - это набор правил и протоколов, которые позволяют разным приложениям обмениваться данными и функциональностью. В контексте нашего приложения API позволяет фронтенду получать и отправлять данные в базу данных SQLite, используя определенные эндпоинты и методы.

Маршрутизация (Routing) - это процесс определения того, какой код должен быть выполнен в ответ на определенный запрос. В Express маршрутизация реализуется с помощью методов, таких как app.get(), app.post(), app.put(), app.delete(), которые определяют, какой код должен быть выполнен в ответ на запрос с определенным методом и URL.

Таблица для хранения заметок имеет следующую структуру:



**id** – Уникальный идентификатор заметки;

**content** – Содержание заметки;

**pin** – флаг, имеющий значение **1** у закрепленных заметок.

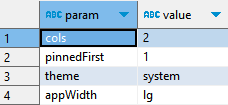
Так же в базе должны храниться настройки приложения. Таблица для их хранения имеет следующую структуру:



**param** – название параметра настроек;

**value** – значение параметра.

По умолчанию в таблице настроек хранятся следующие значения:



Количество колонок – 2;

Закрепление избранных заметок в начале списка;

Тема приложения устанавливается, исходя из настроек системы;

Ширина приложения – средняя.

Инициализация БД и первоначальные настройки выполняются следущим образом:

const db = new sqlite3.Database(DATABASE, (err) => {

    if (err) {

        console.error(err.message);

    } else {

        console.log('Connected to the SQLite database.');

        db.run(`

            CREATE TABLE IF NOT EXISTS notes (

                id INTEGER PRIMARY KEY AUTOINCREMENT,

                content TEXT,

                pin INTEGER DEFAULT (0) NOT NULL

            )

        `);

        db.run(`

            CREATE TABLE IF NOT EXISTS settings (

                param TEXT NOT NULL,

                value TEXT NOT NULL

            )

        `);

        db.run(`

            INSERT INTO settings (param,value) VALUES

                ('cols','2'),

                ('pinnedFirst','0'),

                ('theme','light'),

                ('appWidth','lg');

            `);

    }

});

Сервер должен реализовывать следующие API-методы для работы с заметками: просмотр, добавление, изменение, закрепление, удаление. Реализация этих маршрутов выглядит следующим образом:

## Просмотр заметок

GET-запрос

app.get('/api/notes', (\_req, res) => {

    db.all('SELECT \* FROM notes ORDER BY id DESC', (err, rows) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json(rows);

        }

    });

});

## Добавление заметки

POST-запрос

app.post('/api/notes', (req, res) => {

    let { content } = req.body;

    content = content.replace(

        /```([\s\S]\*?)```|([^`]+)(?=(?:[^`]\*`[^`]\*`)\*[^`]\*$)/g,

        (match, p1, p2) => {

            if (p1) {

                return match;

            } else {

                return ld.escape(p2);

            }

        }

    );

    db.run('INSERT INTO notes (content) VALUES (?)', [content], (err) => {

        if (err) {

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Заметка успешно добавлена' });

        }

    });

});

Для предотвращения возможных XSS-атак к значению **content** применяется регулярное выражение для экранирования специальных символов, таких как символы обрамления кода (```).

Регулярное выражение состоит из двух частей, разделенных | (или). Первая часть \``([\s\S]?)```соответствует строке, обрамленной тремя символами обратной косой черты (```). Внутри этих символов может содержаться любой текст, включая пробелы и переносы строк, поэтому используется[\s\S]?` (любой символ, включая переносы строк, но минимальное количество символов).

Вторая часть регулярного выражения ([^]+)(?=(?:[^]\*[^]\*)\*[^]\*$)/g соответствует любому тексту, который не содержит символ обратной косой черты ([^]+`). Это означает, что эта часть регулярного выражения соответствует любому тексту, который не является частью кода.

Функция-замена принимает два аргумента: match и p1, p2. match соответствует полному совпадению регулярного выражения, а p1 и p2 соответствуют первой и второй части регулярного выражения, соответственно.

Если p1 существует (т.е. если найдена строка кода), функция-замена возвращает match (т.е. оставляет строку кода без изменений). В противном случае, функция-замена экранирует специальные символы в p2 с помощью функции ld.escape и возвращает экранированную строку.

## Изменение заметки

PUT-запрос

app.put('/api/notes/:id/content', (req, res) => {

    const { content } = req.body;

    const id = req.params.id;

    db.run(`UPDATE notes SET content = ? WHERE id = ?`, [content, id], (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Заметка успешно обновлена' });

        }

    });

});

## Закрепление заметки

PUT-запрос

app.put('/api/notes/:id/pin', (req, res) => {

    const { pin } = req.body;

    const id = req.params.id;

    db.run(`UPDATE notes SET pin = ? WHERE id = ?`, [pin, id], (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: `Заметка успешно ${pin ? 'закреплена' : 'откреплена'}` });

        }

    });

});

## Удаление заметки

DELETE-запрос

app.delete('/api/notes/:id', (req, res) => {

    const id = req.params.id;

    db.run('DELETE FROM notes WHERE id = ?', id, (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Заметка успешно удалена' });

        }

    });

});

Для хранения и получения настроек реализуются следующие маршруты:

## Получение настроек

GET-запрос

app.get('/api/settings', (\_req, res) => {

    db.all('SELECT \* FROM settings', (err, rows) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json(rows);

        }

    });

});

## Изменение настроек

PUT-запрос

app.put('/api/settings/', (req, res) => {

    const { param, value } = req.body;

    db.run(`UPDATE settings SET value = ? WHERE param = ?`, [value, param], (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Настройки сохранены' });

        }

    });

});

Полный код серверной части находится в **Приложении 1**.

# Разработка frontend части приложения на React

При выборе технологии для разработки frontend части приложения для организации заметок, было рассмотрено несколько вариантов, включая Angular, Vue.js и React. Однако, после анализа преимуществ и недостатков каждой из этих технологий, было принято решение использовать React.

React - это библиотека JavaScript для создания пользовательских интерфейсов, которая была разработана компанией Facebook. Она позволяет создавать компоненты, которые могут быть легко повторно использованы в различных частях приложения. React известен своей высокой производительностью, легкостью в использовании и обширной экосистемой.

Frontend часть приложения для организации заметок, разработанная на React, имеет следующие особенности:

**Компонентная архитектура**

Приложение состоит из множества маленьких, независимых компонентов, которые могут быть легко повторно использованы в различных частях приложения.

**Использование хуков**

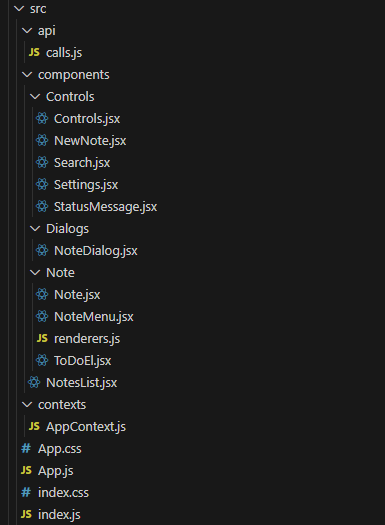
Для управления состоянием компонентов и доступа к функциям React, были использованы хуки, такие как useState, useEffect, useContext и useRef.

**Использование контекста**

Для передачи данных между компонентами, был использован контекст React, который позволяет создавать глобальное хранилище данных.

Использование библиотеки Joy UI: для создания пользовательского интерфейса, была использована библиотека Joy UI, которая предоставляет готовые компоненты и темы для создания приложений.

Структура клиентской части приложения в среде разработки Visual Studio Code выглядит следующим образом:



**App.js** – точка входа в приложение. Здесь производится подключение провайдера контекста, провайдера темы приложения и вызов основных структурных компонентов приложения.

**calls.js** – здесь собраны функции обращения к API сервера для работы с заметками и настройками

**AppContext.js** – в этом файле находится контекст приложения.

Основными компонентами являются:

**NotesList** – в этом компоненте происходит обращение к серверу, получение списка всех заметок и рендеринг каждой заметки;

**Note** – компонент, отрисовывающий каждую отдельную заметку. Из него вызываются **NoteMenu** и **ToDoEl**;

**NoteMenu** – компонент, рендерящий меню заметки;

**ToDoEl** – компонент, рендерящий список задач, если такой встречается в заметке;

**NoteDialog** – компонент, отрисовывающий диалоговое окно, которое используется как для создания новой заметки, так и для редактирования существующей;

**Controls** – компонент, внутри которого размещаются кнопка настроек, кнопка для создания новой заметки и поле для поиска по заметкам;

**Settings** – компонент, в котором меняются настройки приложения;

**Search** – компонент для поиска по заметкам;

**NewNote** – компонент для создания новой заметки;

**StatusMessage** – компонент со всплывающим окном, в котором отображаются сообщения либо об успешном действии с заметкой, либо об ошибке.

Рассмотрим подробнее каждый из компонентов. Полный код каждого компонента находится в **Приложении 2**.

## calls.js

const getNotes = async () => {

    try {

        const response = await fetch('/api/notes');

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        }

        return data;

    } catch (e) {

        throw new Error(e);

    }

};

const addNote = async (content) => {

    try {

        const response = await fetch('/api/notes', {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({ content }),

        });

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        } else {

            return data;

        }

    } catch (e) {

        throw new Error(e);

    }

};

const editNote = async (id, content) => {

    // Код изменения заметки

};

const pinNote = async (id, pin) => {

    // Код закрепления заметки

};

const deleteNote = async (id) => {

    // Код удаления заметки

};

const getSettings = async () => {

    // Код получения настроек

};

const editSettings = async (param, value) => {

    // Код изменения настроек

};

export { getNotes, addNote, editNote, pinNote, deleteNote, getSettings, editSettings };

Технически функции похожи друг на друга: они отправляют запрос на сервер по адресу "**/api/notes**"(обработка таких обращений со стороны сервера будет описана ниже), и получают ответ. Отправка запроса обернута в блок **try/catch** для обработки возможных ошибок, которые, при возникновении, пробрасываются в вызывающий эти функции код:

Разница заключается в том, что функции для получения заметок и настроек используют GET-запрос:

        const response = await fetch('/api/notes');

а функции, которые вносят изменения, используют POST, PUT и DELETE запросы, а также имеют тело запроса, в котором передается полезная нагрузка, как то: новый контент, новые настройки, флаг закрепления заметки или **id** заметки:

        const response = await fetch('/api/notes', {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({ content }),

        });

## AppContext.js

В этом файле создается контекст React (AppContext) и провайдер (AppProvider) для управления глобальными состояниями приложения. Провайдер хранит следующие состояния:

**notes** и **filteredNotes** - для массива всех заметок, или массива отфильтрованных заметок;

**dialogOpen** - для состояния модального окна, служащего для добавления и изменения заметок;

**currentNote** - для текущей заметки;

**apiStatus** - для статуса API-запроса;

**appSettings** для настроек приложения

Также провайдер предоставляет функции для управления этими состояниями:

**openModal** и **closeModal** для открытия и закрытия модального окна;

**setNotes**, **setFilteredNotes**, **setCurrentNote**, **setApiStatus** и **setAppSettings** для обновления соответствующих состояний

Все эти состояния и функции доступны для компонентов приложения через контекст **AppContext**.

## NotesList

В этом файле находится компонент, который отображает список заметок. Компонент использует контекст **AppContext** для доступа к глобальным состояниям приложения.

Компонент имеет следующие функции:

* Получает список заметок с помощью функции **getNotes** из API и сохраняет его в состоянии **notes** контекста **AppContext**;
* Фильтрует массив заметок, если пользователь вводит что-то в строке поиска и сохраняет отфильтрованный массив в состоянии **resultNotes**;

        if (filteredNotes) {

            setResultNotes(filteredNotes);

            return;

        }

* Сортирует массив заметок, если в настройках приложения указано, что избранные заметки должны быть в начале списка и сохраняет отсортированный массив в состоянии **resultNotes**

        if (appSettings.pinnedFirst) {

            const pinedNotes = notes.filter((note) => note.pin);

            const unPinedNotes = notes.filter((note) => !note.pin);

            const sortedNotes = [...pinedNotes, ...unPinedNotes];

            setResultNotes(sortedNotes);

            return;

        }

* Отображает список заметок с помощью компонента **Masonry** из библиотеки **Joy UI**, используя количество столбцов определеное в настройках приложения;
* Отображает каждую заметку с помощью компонента **Note**, передавая ей соответствующие данные.

        <Box pt={2}>

            <Masonry columns={appSettings.cols} spacing={2} sx={{ margin: 0 }}>

                {resultNotes.map((note) => {

                    return <Note key={note.id} note={note}></Note>;

                })}

            </Masonry>

        </Box>

Компонент также использует хуки **useState** и **useEffect** для управления своими состояниями и действиями.

## Note

Этот компонент отображает отдельную заметку. Компонент принимает проп **note**, который содержит информацию о заметке.

Компонент имеет следующие функции:

* Извлекает заголовок и контент заметки из ее содержимого с помощью функции **extractContent**

    const extractContent = (content) => {

        const match = content.match(/^#\s\*(.+)\n/);

        if (match) {

            return [match[1], content.slice(match[0].length)];

        }

        return [`Заметка #${id}`, content];

    };

Она принимает строку **content** в качестве аргумента и возвращает массив из двух элементов: заголовок и содержимое.

Функция использует регулярное выражение /^#\s\*(.+)\n/ для поиска строки, начинающейся с символа "#", за которой следует любое количество пробелов, и затем захватывает любые символы (кроме новой строки) в группе. Если такая строка найдена, функция возвращает массив, содержащий заголовок (группа 1) и содержимое (остаток строки после найденной строки).

Если такая строка не найдена, функция возвращает массив, содержащий заголовок "Заметка #${id}" и оригинальное содержимое;

* Отображает заголовок заметки с помощью компонента **Typography**;
* Отображает контент заметки с помощью компонента **ReactMarkdown**, который парсит все содержимое заметки и, найдя теги Markdown преобразует контент в HTML-разметку. В компонент **ReactMarkdown** передаются рендереры из файла **renderers.js**, которые отдельно описываю рендеринг блоков кода для подсветки синтаксиса и рендеринг элементов **checkbox** для того, чтобы того, чтобы можно было взаимодействовать со списком задач, отмечая или снимая отметку с каждой из них;
* Отображает меню заметки с помощью компонента **NoteMenu**;
* Если заметка закреплена, отображает соответствующий бейдж с помощью компонента **Badge**.

Компонент также использует хук **useState** для управления состоянием меню заметки и хук **useContext** для доступа к контексту **AppContext**.

## NoteMenu

Компонент отображает меню для заметки, которое содержит следующие пункты:

* Редактировать заметку (открывает модальное окно **NoteDialog** для редактирования);
* Закрепить/Открепить заметку;
* Удалить заметку

Меню открывается при клике на иконку в правом верхнем углу заметки. Компонент использует контекст **AppContext** для доступа к функциям **openModal**, **setApiStatus** и **setCurrentNote**.

## NoteDialog

Файл содержит компонент **NoteDialog**, который используется для добавления и редактирования заметок. Компонент использует контекст **AppContext** для доступа к состоянию приложения и функциям для добавления и редактирования заметок.

Компонент отображает модальное окно с формой для ввода текста заметки и кнопкой "Сохранить". Если при открытии окна в **AppContext** есть **currentNote**, то компонент отображает содержимое pfvtnrb в текстовой области и позволяет редактировать его. Если **currentNote** не определена, то компонент отображает пустую текстовую область для ввода новой заметки.

Компонент также использует хуки **useState** и **useEffect** для управления состоянием текстовой области и фокусом на ней. Функции **handleAddNote** и **handleEditNote** используются для отправки запросов на добавление и редактирование заметок соответственно.

## Controls

Этот компонент является контейнером для компонентов настроек, поиска и создания новой заметки: **Settings**, **Search** и **NewNote**. Компонент имеет фиксированное положение вверху экрана и при прокрутке списка заметок всегда остается виден для быстрого доступа к его элементам.

## Settings

Этот компонент отображает панель настроек приложения. Он содержит несколько разделов:

* **Количество колонок**: раздел с группой радиокнопок, позволяющий пользователю выбрать количество колонок для отображения заметок (1, 2 или 3);
* **Избранные заметки сверху**: раздел с переключателем, который позволяет пользователю выбрать, отображать ли избранные заметки вверху списка.
* **Тема приложения**: раздел с группой радиокнопок, позволяющий пользователю выбрать тему приложения (Светлая, Темная или Системная).
* **Размер рабочей области**: раздел с группой радиокнопок, позволяющий пользователю выбрать размер рабочей области приложения (Маленький, Средний или Вся ширина).

Компонент использует **useState** и **useEffect** для управления состоянием открытия/закрытия панели настроек и получения текущих настроек из API. Также он использует **AppContext** для доступа к текущим настройкам и их изменению.

## Search

Компонент представляет собой поле поиска, которое позволяет пользователю искать заметки по их содержимому. Он использует **useContext** для доступа к **AppContext**, который содержит текущие заметки и функции для их фильтрации и установки текущей заметки. Компонент использует **useState** для хранения значения поиска и **useEffect** для фильтрации заметок при изменении значения поиска.

    useEffect(() => {

        if (searchValue.length > 0) {

            const filteredNotes = notes.filter((note) =>

                note.content.toLowerCase().includes(searchValue.toLowerCase())

            );

            setFilteredNotes(filteredNotes);

        } else {

            setFilteredNotes(null);

        }

    }, [searchValue, notes]);

## NewNote

Компонент представляет собой кнопку для открытия модального окна **NoteDialog** в режиме создания новой заметки

## StatusMessage

Этот компонент отображает сообщение о статусе API-запросов в виде всплывающего уведомления. Он использует **AppContext** для доступа к статусу операции. Компонент использует **useState** для хранения состояния открытия/закрытия уведомления и **useEffect** для отслеживания изменений статуса операции. Компонент отображает уведомление с текстом и определенным цветом фона в зависимости от статуса операции (успех или ошибка). Уведомление автоматически закрывается через определенное время.

# Интеграция backend и frontend частей

Для того, чтобы обе части приложения могли взаимодействовать друг с другом, необходим WEB-сервер. В данной работе было решено использовать Nginx.

Nginx - это свободное и открытое программное обеспечение, которое используется как веб-сервер, обратный прокси-сервер, кэш-сервер, load balancer и медиа-стриминговый сервер. Nginx был создан Игорем Сысоевым в 2002 году и с тех пор стал одним из самых популярных веб-серверов в мире.

Nginx известен своей высокой производительностью, надежностью и гибкостью. Он может обрабатывать большое количество запросов одновременно, что делает его идеальным выбором для высоконагруженных веб-приложений.

Его преимущества заключаются в следующем:

* Nginx может работать как проксирующий сервер, который позволяет направлять запросы от frontend части на backend часть, что упрощает интеграцию двух частей приложения.
* Nginx может распределять нагрузку между несколькими серверами, что позволяет масштабировать ваше приложение и обеспечивать высокую доступность.
* Nginx может кэшировать часто запрашиваемые ресурсы, что ускоряет загрузку страниц и уменьшает нагрузку на сервер.
* Nginx имеет встроенные механизмы безопасности, такие как SSL/TLS, что позволяет обеспечивать безопасность данных при передаче между клиентом и сервером.
* Nginx имеет простой и интуитивно понятный конфигурационный файл, что делает его легко настраиваемым даже для начинающих разработчиков.

Для реализации целей этого проекта создаем конфигурационный файл следующего содержания:

server {

        listen                  80;

        listen                  443;

        root                    /var/www/mynotes;

        index                   index.html index.htm;

        server\_name             localhost;

        client\_max\_body\_size    32m;

        error\_page              500 502 503 504  /50x.html;

        location /api {

            proxy\_pass          http://localhost:3003;

            proxy\_set\_header    Host $host;

            proxy\_set\_header    X-Real-IP $remote\_addr;

        }

    }

Далее backend часть собирается с помощью npm-пакета **pkg** в отдельный файл и запускается на сервере.

Frontend часть собирается с помощью **npm run build**, и полученные файлы кладутся на сервере в каталог **/var/www/mynotes**.

После этого приложение становится доступно в локальной сети из браузера по адресу сервера, на котором запущен Nginx.

Скриншоты работающего приложения отображены в **Приложении 3**.

# Перспективы развития приложения

В процессе работы над проектом появились идеи его модернизации и расширения функционала. Среди них:

* Установка тэгов для заметок. Это позволит фильтровать заметки по тематике, например, заметки, связанные с программированием или заметки, содержащие списки задач;
* Авторизация в приложении. На данный момент приложение имеет свободный доступ. Можно ограничиться авторизацией на сервере Nginx, но лучшим способом будет использование jwt-токенов;
* Возможность добавлять изображения, в том числе скриншоты, в заметки;
* Возможность одним кликом из выделенного фрагмента на веб-странице создать заметку;
* Хоть сейчас frontend-часть и является адаптивной, подстраивая размеры блоков к ширине экрана, количество колонок остается неизменным в десктопной, планшетной и мобильной версиях. Эта часть видится так: на планшетной версии в настройках оставлять выбор из 1 или 2 колонок по ширине, а в мобильной версии отключать эту настройку

# Заключение

В данной дипломной работе была разработана архитектура и реализовано приложение для организации заметок на основе Node.js и React. Целью работы было создать эффективное и масштабируемое приложение, которое может помочь пользователям организовывать и хранить свои заметки в удобном и доступном формате.

В ходе работы были рассмотрены различные аспекты разработки приложения, включая выбор технологий, проектирование приложения и базы данных, реализацию backend и frontend частей, а также настройка web-сервера.

Проектирование клиент-серверного приложения включает в себя разработку архитектуры приложения, которая позволяет взаимодействовать между клиентской и серверной частями. В данной дипломной работе была разработана архитектура приложения для организации заметок, которая включает в себя следующие компоненты:

* Backend часть на Node.js, которая отвечает за обработку запросов от клиентской части и взаимодействие с базой данных;
* База данных, которая хранит в себе заметки и настройки приложения;
* Frontend часть на React, которая отвечает за отображение заметок и взаимодействие с пользователем.
* Nginx, который отвечает за маршрутизацию запросов и обеспечение безопасности.

Приложение для организации заметок имеет большую значимость, так как оно позволяет пользователям эффективно управлять своими заметками и улучшать свою продуктивность. Кроме того, оно может быть использовано в различных сферах деятельности, таких как образование, бизнес и развлечения.

В целом, выполнение дипломной работы помогло мне приобрести новые навыки и знания в области разработки приложений, а также понять важность создания эффективных и масштабируемых приложений.

# Список использованных источников

1. <https://react.dev/> - официальный сайт React
2. <https://nodejs.org/> - официальный сайт Node.js
3. <https://learn.javascript.ru/> - учебник по JavaScript
4. <https://www.sqlite.org> - официальный сайт SQLite
5. https://nginx.org/ - официальный сайт nginx
6. <https://stackoverflow.com/>
7. <https://www.google.ru/>
8. https://ya.ru/

# Приложение 1

Исходный код backend-части:

const express = require('express');

const ld = require('lodash');

const sqlite3 = require('sqlite3').verbose();

const app = express();

const PORT = 3003;

const DATABASE = 'notes.db';

const db = new sqlite3.Database(DATABASE, (err) => {

    if (err) {

        console.error(err.message);

    } else {

        console.log('Connected to the SQLite database.');

        db.run(`

            CREATE TABLE IF NOT EXISTS notes (

                id INTEGER PRIMARY KEY AUTOINCREMENT,

                content TEXT,

                pin INTEGER DEFAULT (0) NOT NULL

            )

        `);

        db.run(`

            CREATE TABLE IF NOT EXISTS settings (

                param TEXT NOT NULL,

                value TEXT NOT NULL

            )

        `);

    }

});

app.use(express.json());

app.get('/api/notes', (\_req, res) => {

    db.all('SELECT \* FROM notes ORDER BY id DESC', (err, rows) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json(rows);

        }

    });

});

app.get('/api/notes/:id', (req, res) => {

    const noteId = req.params.id;

    db.get('SELECT \* FROM notes WHERE id = ?', [noteId], (err, row) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else if (row) {

            res.json(row);

        } else {

            res.status(404).json({ error: 'Заметка не найдена' });

        }

    });

});

app.post('/api/notes', (req, res) => {

    let { content } = req.body;

    content = content.replace(

        /```([\s\S]\*?)```|([^`]+)(?=(?:[^`]\*`[^`]\*`)\*[^`]\*$)/g,

        (match, p1, p2) => {

            if (p1) {

                return match;

            } else {

                return ld.escape(p2);

            }

        }

    );

    db.run('INSERT INTO notes (content) VALUES (?)', [content], (err) => {

        if (err) {

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Заметка успешно добавлена' });

        }

    });

});

app.put('/api/notes/:id/content', (req, res) => {

    const { content } = req.body;

    const id = req.params.id;

    db.run(`UPDATE notes SET content = ? WHERE id = ?`, [content, id], (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Заметка успешно обновлена' });

        }

    });

});

app.put('/api/notes/:id/pin', (req, res) => {

    const { pin } = req.body;

    const id = req.params.id;

    db.run(`UPDATE notes SET pin = ? WHERE id = ?`, [pin, id], (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: `Заметка успешно ${pin ? 'закреплена' : 'откреплена'}` });

        }

    });

});

app.delete('/api/notes/:id', (req, res) => {

    const id = req.params.id;

    db.run('DELETE FROM notes WHERE id = ?', id, (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Заметка успешно удалена' });

        }

    });

});

app.get('/api/settings', (\_req, res) => {

    db.all('SELECT \* FROM settings', (err, rows) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json(rows);

        }

    });

});

app.put('/api/settings/', (req, res) => {

    const { param, value } = req.body;

    db.run(`UPDATE settings SET value = ? WHERE param = ?`, [value, param], (err) => {

        if (err) {

            console.error(err.message);

            res.status(500).json({ error: 'Внутренняя ошибка сервера' });

        } else {

            res.json({ message: 'Настройки сохранены' });

        }

    });

});

app.listen(PORT, () => {

    console.log(`Server is running on port ${PORT}`);

});

# Приложение 2

Исходный код frontend-части:

## App.js

import './App.css';

import Box from '@mui/joy/Box';

import { useContext } from 'react';

import { AppProvider } from 'contexts/AppContext';

import AppContext from 'contexts/AppContext';

import NotesList from './components/NotesList';

import Controls from './components/Controls/Controls';

import NoteDialog from './components/Dialogs/NoteDialog';

import StatusMessage from 'components/Controls/StatusMessage';

import { CssVarsProvider } from '@mui/joy/styles';

import CssBaseline from '@mui/joy/CssBaseline';

function App() {

    return (

        <CssVarsProvider disableNestedContext>

            <CssBaseline />

            <div className="App">

                <AppProvider>

                    <AppContent />

                    <NoteDialog />

                    <StatusMessage />

                </AppProvider>

            </div>

        </CssVarsProvider>

    );

}

function AppContent() {

    const { appSettings } = useContext(AppContext);

    return (

        <Box

            sx={{

                pl: 2,

                pr: 2,

                pb: 2,

                m: '0 auto',

                maxWidth: appSettings.appWidth,

            }}

        >

            <Controls />

            <NotesList />

        </Box>

    );

}

export default App;

## calls.js

const getNotes = async () => {

    try {

        const response = await fetch('/api/notes');

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        }

        return data;

    } catch (e) {

        throw new Error(e);

    }

};

const addNote = async (content) => {

    try {

        const response = await fetch('/api/notes', {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({ content }),

        });

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        } else {

            return data;

        }

    } catch (e) {

        throw new Error(e);

    }

};

const editNote = async (id, content) => {

    try {

        const response = await fetch(`/api/notes/${id}/content`, {

            method: 'PUT',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({ content }),

        });

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        } else {

            return data;

        }

    } catch (e) {

        throw new Error(e);

    }

};

const pinNote = async (id, pin) => {

    try {

        const response = await fetch(`/api/notes/${id}/pin`, {

            method: 'PUT',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({ pin }),

        });

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        } else {

            return data;

        }

    } catch (e) {

        throw new Error(e);

    }

};

const deleteNote = async (id) => {

    try {

        const response = await fetch('/api/notes/' + id, {

            method: 'DELETE',

        });

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        } else {

            return data;

        }

    } catch (e) {

        throw new Error(e);

    }

};

const getSettings = async () => {

    try {

        const response = await fetch('/api/settings');

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        }

        return data;

    } catch (e) {

        throw new Error(e);

    }

};

const editSettings = async (param, value) => {

    try {

        const response = await fetch(`/api/settings`, {

            method: 'PUT',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({ param, value }),

        });

        const data = await response.json();

        if (!response.ok) {

            throw new Error(data.error || 'Неизвестная ошибка');

        } else {

            return data;

        }

    } catch (e) {

        throw new Error(e);

    }

};

export { getNotes, addNote, editNote, pinNote, deleteNote, getSettings, editSettings };

## AppContext.js

import React, { createContext, useState } from 'react';

const AppContext = createContext();

export const AppProvider = ({ children }) => {

    const [notes, setNotes] = useState([]);

    const [filteredNotes, setFilteredNotes] = useState([]);

    const [dialogOpen, setDialogOpen] = useState(false);

    const [currentNote, setCurrentNote] = useState(null);

    const [apiStatus, setApiStatus] = useState({ status: '', data: '' });

    const [appSettings, setAppSettings] = useState({

        cols: 2,

        pinnedFirst: true,

        theme: 'system',

        appWidth: 'lg',

    });

    const openModal = (currentNote) => {

        setCurrentNote(currentNote);

        setDialogOpen(true);

    };

    const closeModal = () => {

        setDialogOpen(false);

        setCurrentNote(null);

    };

    return (

        <AppContext.Provider

            value={{

                dialogOpen,

                openModal,

                closeModal,

                notes,

                setNotes,

                filteredNotes,

                setFilteredNotes,

                currentNote,

                setCurrentNote,

                apiStatus,

                setApiStatus,

                appSettings,

                setAppSettings,

            }}

        >

            {children}

        </AppContext.Provider>

    );

};

export default AppContext;

## NotesList

import Masonry from '@mui/lab/Masonry';

import Box from '@mui/joy/Box';

import Note from './Note/Note';

import AppContext from 'contexts/AppContext';

import { getNotes } from 'api/calls';

import { useState, useEffect, useContext } from 'react';

function NotesList() {

    const { apiStatus, notes, setNotes, filteredNotes, appSettings } = useContext(AppContext);

    const [resultNotes, setResultNotes] = useState([]);

    useEffect(() => {

        const fetchNotes = async () => {

            try {

                const data = await getNotes();

                setNotes(data);

            } catch (e) {

                console.error('Ошибка при получении заметок:', e.message);

            }

        };

        fetchNotes();

    }, [apiStatus]);

    useEffect(() => {

        if (filteredNotes) {

            setResultNotes(filteredNotes);

            return;

        }

        if (appSettings.pinnedFirst) {

            const pinedNotes = notes.filter((note) => note.pin);

            const unPinedNotes = notes.filter((note) => !note.pin);

            const sortedNotes = [...pinedNotes, ...unPinedNotes];

            setResultNotes(sortedNotes);

            return;

        }

        setResultNotes(notes);

    }, [filteredNotes, notes, appSettings.pinnedFirst]);

    return (

        <Box pt={2}>

            <Masonry columns={appSettings.cols} spacing={2} sx={{ margin: 0 }}>

                {resultNotes.map((note) => {

                    return <Note key={note.id} note={note}></Note>;

                })}

            </Masonry>

        </Box>

    );

}

export default NotesList;

## Note

import Card from '@mui/joy/Card';

import Badge from '@mui/joy/Badge';

import Typography from '@mui/joy/Typography';

import ReactMarkdown from 'react-markdown';

import remarkGfm from 'remark-gfm';

import remarkBreaks from 'remark-breaks';

import renderers from './renderers';

import NoteMenu from './NoteMenu';

import AppContext from 'contexts/AppContext';

import { useState, useContext } from 'react';

function Note({ note }) {

    const { id, content, pin } = note;

    const extractContent = (content) => {

        const match = content.match(/^#\s\*(.+)\n/);

        if (match) {

            return [match[1], content.slice(match[0].length)];

        }

        return [`Заметка #${id}`, content];

    };

    const [noteTitle, noteContent] = extractContent(content);

    const [isMenuOpen, setIsMenuOpen] = useState(false);

    const { setCurrentNote } = useContext(AppContext);

    return (

        <Card

            size="sm"

            sx={{

                '--Card-padding': '16px',

                '&:hover': { boxShadow: '0px 0px 8px 1px #afafafde' },

                filter: isMenuOpen ? 'blur(5px) brightness(0.5)' : 'none',

            }}

            onClick={() => setCurrentNote(note)}

        >

            {pin ? (

                <Badge

                    badgeContent={''}

                    size="lg"

                    sx={{

                        position: 'absolute',

                        top: 3,

                        left: '50%',

                    }}

                ></Badge>

            ) : (

                ''

            )}

            <Typography level="title-lg">{noteTitle}</Typography>

            <NoteMenu isMenuOpen={isMenuOpen} setIsMenuOpen={setIsMenuOpen} note={note}></NoteMenu>

            <ReactMarkdown

                remarkPlugins={[remarkGfm, remarkBreaks]}

                children={noteContent}

                components={renderers}

            />

        </Card>

    );

}

export default Note;

## renderers.js

import { Prism as SyntaxHighlighter } from 'react-syntax-highlighter';

import { vscDarkPlus } from 'react-syntax-highlighter/dist/esm/styles/prism';

import ToDoEl from './ToDoEl';

const renderers = {

    li(props) {

        const { children, className } = props;

        if (className === 'task-list-item') {

            const [checkbox, ...label] = children;

            const isChecked = checkbox.props.checked;

            return <ToDoEl isChecked={isChecked} label={label} />;

        }

        return <li>{children}</li>;

    },

    code(props) {

        const { children, className, node, ...rest } = props;

        const match = /language-(\w+)/.exec(className || '');

        const language = match ? match[1] : 'bash';

        return (

            <SyntaxHighlighter

                {...rest}

                children={String(children).replace(/\n$/, '')}

                language={language}

                style={vscDarkPlus}

                wrapLongLines={true}

                customStyle={{

                    whiteSpace: 'wrap',

                    wordBreak: 'break-word',

                }}

            />

        );

    },

};

export default renderers;

## ToDoEl

import AppContext from 'contexts/AppContext';

import Checkbox from '@mui/joy/Checkbox';

import { useContext, useEffect, useState } from 'react';

import { editNote } from 'api/calls';

function ToDoEl({ isChecked, label }) {

    const [checked, setChecked] = useState(isChecked);

    const { currentNote, setCurrentNote, setApiStatus } = useContext(AppContext);

    useEffect(() => {

        setChecked(isChecked);

    }, [isChecked]);

    useEffect(() => {

        if (currentNote) {

            const { id, content } = currentNote;

            let todoContent = content;

            const checkIndex = todoContent.indexOf(label[1]) - 3;

            String.prototype.replaceAt = function (index, replacement) {

                return (

                    this.substring(0, index) +

                    replacement +

                    this.substring(index + replacement.length)

                );

            };

            if (checked) {

                todoContent = todoContent.replaceAt(checkIndex, 'x');

            } else {

                todoContent = todoContent.replaceAt(checkIndex, ' ');

            }

            handleEditNote(id, todoContent);

        }

    }, [checked]);

    const handleEditNote = async (id, content) => {

        try {

            const response = await editNote(id, content);

            setApiStatus({ status: 'success', data: response.message });

        } catch (e) {

            setApiStatus({ status: 'error', data: e.message });

            console.log(e.message);

        } finally {

            setCurrentNote(null);

        }

    };

    const handleCheck = () => {

        setChecked(!checked);

    };

    return <li>{<Checkbox size="sm" checked={checked} label={label} onChange={handleCheck} />}</li>;

}

export default ToDoEl;

## NoteMenu

import IconButton from '@mui/joy/IconButton';

import Menu from '@mui/joy/Menu';

import MenuItem from '@mui/joy/MenuItem';

import MoreVert from '@mui/icons-material/MoreVert';

import PinIcon from '@mui/icons-material/BookmarkAddOutlined';

import UnPinIcon from '@mui/icons-material/BookmarkRemoveOutlined';

import EditIcon from '@mui/icons-material/EditOutlined';

import DeleteIcon from '@mui/icons-material/DeleteOutlineOutlined';

import MenuButton from '@mui/joy/MenuButton';

import Dropdown from '@mui/joy/Dropdown';

import AppContext from 'contexts/AppContext';

import { useContext } from 'react';

import { deleteNote, pinNote } from 'api/calls';

function NoteMenu({ isMenuOpen, setIsMenuOpen, note }) {

    const { openModal, setApiStatus, setCurrentNote } = useContext(AppContext);

    const handlePinToggle = async (id, pin) => {

        try {

            const response = await pinNote(id, pin);

            setApiStatus({ status: 'success', data: response.message });

        } catch (e) {

            setApiStatus({ status: 'error', data: e.message });

            console.log(e.message);

        } finally {

            setCurrentNote(null);

        }

    };

    const handleDeleteNote = async (id) => {

        try {

            const response = await deleteNote(id);

            setApiStatus({ status: 'success', data: response.message });

        } catch (e) {

            setApiStatus({ status: 'error', data: e.message });

            console.log(e.message);

        } finally {

            setCurrentNote(null);

        }

    };

    return (

        <Dropdown

            open={isMenuOpen}

            onOpenChange={(\_, isMenuOpen) => {

                setIsMenuOpen(isMenuOpen);

            }}

        >

            <MenuButton

                slots={{ root: IconButton }}

                sx={{

                    position: 'absolute',

                    top: '0.5rem',

                    right: '0.5rem',

                }}

            >

                <MoreVert sx={{ fontSize: 18 }} />

            </MenuButton>

            <Menu

                placement="left"

                size="sm"

                sx={{

                    '--List-padding': '0.25rem',

                    '--ListItem-radius': '5px',

                    '--List-display': 'flex',

                    flexDirection: 'row',

                }}

                slotProps={{ fontSize: 18 }}

            >

                <MenuItem onClick={() => openModal(note)}>

                    <EditIcon />

                </MenuItem>

                <MenuItem onClick={() => handlePinToggle(note.id, !note.pin)}>

                    {note.pin ? <UnPinIcon /> : <PinIcon />}

                </MenuItem>

                <MenuItem onClick={() => handleDeleteNote(note.id)} variant="soft" color="danger">

                    <DeleteIcon />

                </MenuItem>

            </Menu>

        </Dropdown>

    );

}

export default NoteMenu;

## NoteDialog

import { useContext, useState, useEffect, useRef } from 'react';

import AppContext from 'contexts/AppContext';

import Modal from '@mui/joy/Modal';

import Button from '@mui/joy/Button';

import Textarea from '@mui/joy/Textarea';

import DialogTitle from '@mui/joy/DialogTitle';

import Stack from '@mui/joy/Stack';

import FormControl from '@mui/joy/FormControl';

import ModalDialog from '@mui/joy/ModalDialog';

import { addNote, editNote } from 'api/calls';

function NoteDialog() {

    const { dialogOpen, closeModal, currentNote, setApiStatus } = useContext(AppContext);

    const { id, content } = currentNote || { id: '', content: '' };

    const editMode = !!id;

    const [noteContent, setNoteContent] = useState('');

    const textareaRef = useRef(null);

    textareaRef.current?.focus();

    useEffect(() => {

        setNoteContent(content || '');

    }, [currentNote]);

    const handleAddNote = async (content) => {

        try {

            const response = await addNote(content);

            setApiStatus({ status: 'success', data: response.message });

        } catch (e) {

            setApiStatus({ status: 'error', data: e.message });

            console.log(e.message);

        }

    };

    const handleEditNote = async (id, content) => {

        try {

            const response = await editNote(id, content);

            setApiStatus({ status: 'success', data: response.message });

        } catch (e) {

            setApiStatus({ status: 'error', data: e.message });

            console.log(e.message);

        }

    };

    return (

        <Modal open={dialogOpen} onClose={() => closeModal()}>

            <ModalDialog sx={{ width: '90vw', maxWidth: 'md' }}>

                <DialogTitle>

                    {editMode ? 'Редактирование заметки' : 'Добавление заметки'}

                </DialogTitle>

                <form

                    onSubmit={(e) => {

                        e.preventDefault();

                        editMode ? handleEditNote(id, noteContent) : handleAddNote(noteContent);

                        closeModal();

                    }}

                >

                    <Stack spacing={2}>

                        <FormControl>

                            <Textarea

                                value={noteContent}

                                placeholder="Нужно не забыть, что..."

                                slotProps={{ textarea: { ref: textareaRef } }}

                                onChange={(e) => {

                                    setNoteContent(e.target.value);

                                }}

                                sx={{ fontFamily: 'Monospace' }}

                                minRows={5}

                                maxRows={30}

                                required

                            />

                        </FormControl>

                        <Button type="submit">Сохранить</Button>

                    </Stack>

                </form>

            </ModalDialog>

        </Modal>

    );

}

export default NoteDialog;

## Controls

import Stack from '@mui/joy/Stack';

import Settings from './Settings';

import Search from './Search';

import NewNote from './NewNote';

function Controls() {

    return (

        <Stack

            direction="row"

            borderRadius={3}

            justifyContent="space-between"

            position="sticky"

            top={0}

            zIndex={999}

            sx={{

                transition: 'box-shadow 2300ms cubic-bezier(0.4, 0, 0.2, 1) 0ms',

                boxShadow:

                    '0px 2px 4px -1px rgba(0,0,0,0.2), 0px 4px 5px 0px rgba(0,0,0,0.14), 0px 1px 10px 0px rgba(0,0,0,0.12)',

                backgroundColor: 'var(--joy-palette-background-level1)',

                borderBottomLeftRadius: '10px',

                borderBottomRightRadius: '10px',

                p: 1,

                br: 3,

            }}

        >

            <Settings />

            <Search />

            <NewNote />

        </Stack>

    );

}

export default Controls;

## Settings

import IconButton from '@mui/joy/IconButton';

import SettingsOutlinedIcon from '@mui/icons-material/SettingsOutlined';

import { Fragment, useContext, useState, useEffect } from 'react';

import Box from '@mui/joy/Box';

import Drawer from '@mui/joy/Drawer';

import DialogTitle from '@mui/joy/DialogTitle';

import DialogContent from '@mui/joy/DialogContent';

import ModalClose from '@mui/joy/ModalClose';

import Divider from '@mui/joy/Divider';

import FormControl from '@mui/joy/FormControl';

import RadioGroup from '@mui/joy/RadioGroup';

import Radio from '@mui/joy/Radio';

import Sheet from '@mui/joy/Sheet';

import Switch from '@mui/joy/Switch';

import Typography from '@mui/joy/Typography';

import AppContext from 'contexts/AppContext';

import { useColorScheme } from '@mui/joy/styles';

import { getSettings, editSettings } from 'api/calls';

function Settings() {

    const [open, setOpen] = useState(false);

    const { appSettings, setAppSettings } = useContext(AppContext);

    const { setMode } = useColorScheme();

    useEffect(() => {

        fetchSettings();

    }, []);

    useEffect(() => {

        setMode(appSettings.theme);

    }, [appSettings.theme]);

    const fetchSettings = async () => {

        try {

            const settings = await getSettings();

            const newSettings = settings.reduce((acc, obj) => {

                acc[obj.param] = +obj.value || convertToNumber(obj.value);

                return acc;

            }, {});

            setAppSettings(newSettings);

        } catch (e) {

            console.error('Ошибка при получении настроек: ', e.message);

        }

    };

    const convertToNumber = (value) => {

        if (value === '1' || value === 'true') {

            return true;

        } else if (value === '0' || value === 'false') {

            return false;

        } else {

            return value;

        }

    };

    const handleEditSettings = async (param, value) => {

        try {

            await editSettings(param, value);

        } catch (e) {

            console.log(e.message);

        } finally {

            setAppSettings((sets) => ({ ...sets, [param]: value }));

        }

    };

    return (

        <Fragment>

            <IconButton color="primary" size="lg" onClick={() => setOpen(!open)}>

                <SettingsOutlinedIcon sx={{ fontSize: 32 }} />

            </IconButton>

            <Drawer

                open={open}

                onClose={() => setOpen(false)}

                slotProps={{

                    content: {

                        sx: {

                            bgcolor: 'transparent',

                            p: { md: 3, sm: 0 },

                            boxShadow: 'none',

                            maxWidth: '480px',

                        },

                    },

                }}

            >

                <Sheet

                    sx={{

                        borderRadius: 'md',

                        p: 2,

                        display: 'flex',

                        flexDirection: 'column',

                        gap: 2,

                        height: '100%',

                        overflow: 'hidden',

                    }}

                >

                    <DialogTitle>Настройки</DialogTitle>

                    <ModalClose />

                    <Divider sx={{ mt: 'auto' }} />

                    <DialogContent sx={{ gap: 4 }}>

                        <FormControl sx={{ overflow: 'hidden' }}>

                            <Typography level="title-md" fontWeight="bold">

                                Количество колонок

                            </Typography>

                            <RadioGroup

                                sx={{ pl: 2 }}

                                value={appSettings.cols}

                                onChange={(e) => handleEditSettings('cols', e.target.value)}

                            >

                                <Radio value="1" label="1" />

                                <Radio value="2" label="2" />

                                <Radio value="3" label="3" />

                            </RadioGroup>

                        </FormControl>

                        <FormControl orientation="horizontal" sx={{ overflow: 'hidden' }}>

                            <Box sx={{ flex: 1 }}>

                                <Typography level="title-md" fontWeight="bold">

                                    Избранные заметки сверху

                                </Typography>

                            </Box>

                            <Switch

                                checked={!!appSettings.pinnedFirst}

                                onChange={(e) =>

                                    handleEditSettings('pinnedFirst', e.target.checked)

                                }

                            />

                        </FormControl>

                        <FormControl sx={{ overflow: 'hidden' }}>

                            <Typography level="title-md" fontWeight="bold">

                                Тема приложения

                            </Typography>

                            <RadioGroup

                                sx={{ pl: 2 }}

                                value={appSettings.theme}

                                onChange={(e) => {

                                    handleEditSettings('theme', e.target.value);

                                    setMode(e.target.value);

                                }}

                            >

                                <Radio value="light" label="Светлая" />

                                <Radio value="dark" label="Темная" />

                                <Radio value="system" label="Системная" />

                            </RadioGroup>

                        </FormControl>

                        <FormControl sx={{ overflow: 'hidden' }}>

                            <Typography level="title-md" fontWeight="bold">

                                Размер рабочей области

                            </Typography>

                            <RadioGroup

                                sx={{ pl: 2 }}

                                value={appSettings.appWidth}

                                onChange={(e) => handleEditSettings('appWidth', e.target.value)}

                            >

                                <Radio value="md" label="Маленький" />

                                <Radio value="lg" label="Средний" />

                                <Radio value="100%" label="Вся ширина" />

                            </RadioGroup>

                        </FormControl>

                    </DialogContent>

                </Sheet>

            </Drawer>

        </Fragment>

    );

}

export default Settings;

## Search

import Input from '@mui/joy/Input';

import SearchIcon from '@mui/icons-material/SearchOutlined';

import AppContext from 'contexts/AppContext';

import { useContext, useEffect, useState } from 'react';

function Search() {

    const { notes, setFilteredNotes, setCurrentNote } = useContext(AppContext);

    const [searchValue, setSearchValue] = useState('');

    useEffect(() => {

        if (searchValue.length > 0) {

            const filteredNotes = notes.filter((note) =>

                note.content.toLowerCase().includes(searchValue.toLowerCase())

            );

            setFilteredNotes(filteredNotes);

        } else {

            setFilteredNotes(null);

        }

    }, [searchValue, notes]);

    return (

        <Input

            value={searchValue}

            onChange={(e) => setSearchValue(e.target.value)}

            onFocus={() => setCurrentNote(null)}

            sx={{ width: '50%' }}

            endDecorator={<SearchIcon />}

            placeholder="Поиск"

        />

    );

}

export default Search;

## NewNote

import IconButton from '@mui/joy/IconButton';

import AddIcon from '@mui/icons-material/MapsUgcOutlined';

import React, { useContext } from 'react';

import AppContext from 'contexts/AppContext';

function NewNote() {

    const { openModal } = useContext(AppContext);

    return (

        <>

            <IconButton onClick={() => openModal()} color="success" size="lg">

                <AddIcon sx={{ fontSize: 32 }} />

            </IconButton>

        </>

    );

}

export default NewNote;

## StatusMessage

import Snackbar from '@mui/joy/Snackbar';

import AppContext from 'contexts/AppContext';

import { useContext, useEffect, useState } from 'react';

function StatusMessage() {

    const [open, setOpen] = useState(false);

    const { apiStatus } = useContext(AppContext);

    const { status, data } = apiStatus;

    useEffect(() => {

        if (status) {

            setOpen(true);

        }

    }, [apiStatus]);

    return (

        <Snackbar

            autoHideDuration={1000}

            open={open}

            variant="solid"

            size="lg"

            color={status === 'success' ? 'success' : 'danger'}

            onClose={(\_event, reason) => {

                if (reason === 'clickaway') {

                    return;

                }

                setOpen(false);

            }}

            sx={{ justifyContent: 'center' }}

        >

            {data}

        </Snackbar>

    );

}

export default StatusMessage;

# Приложение 3

