Geekbrains

**Разработка веб-приложения "Записная книжка" с использованием библиотеки для построения пользовательских интерфейсов React**

Программа: Frontend-разработчик. Специалист

Специализация: Frontend разработка

Воронин О. Н.

2025

**ВВЕДЕНИЕ**..........................................................................................................3

**1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ НА REACT**...........................................................................................................7

1.1. Современные тенденции в разработке пользовательских интерфейсов.......7

1.2. Библиотека React: архитектура и основные концепции...............................12

1.3. Сопутствующие технологии и инструменты................................................16

1.4. Анализ существующих решений для ведения заметок................................23

**2. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ «ЗАПИСНАЯ КНИЖКА»**................................................................................26

2.1. Техническое задание на разработку.............................................................26

2.2. Проектирование архитектуры приложения.................................................32

2.3. Разработка пользовательского интерфейса..................................................38

2.4. Реализация бизнес-логики приложения.......................................................46

2.5. Интеграция и тестирование компонентов....................................................52

**ЗАКЛЮЧЕНИЕ**...............................................................................................58

**СПИСОК ЛИТЕРАТУРЫ**.............................................................................61

**ПРИЛОЖЕНИЯ**..............................................................................................62

**Введение**

В современном цифровом мире информационные технологии проникают во все сферы человеческой деятельности. Одной из ключевых потребностей пользователей является эффективное управление личной информацией, организация задач и хранение важных записей. Веб-приложения для ведения заметок стали неотъемлемой частью повседневной жизни миллионов людей по всему миру.

Развитие фронтенд-разработки и появление современных JavaScript-библиотек, таких как React, открыли новые возможности для создания интерактивных, производительных и масштабируемых пользовательских интерфейсов. React, разработанный компанией Facebook, завоевал популярность благодаря своей компонентной архитектуре, виртуальному DOM и богатой экосистеме, что делает его идеальным выбором для разработки сложных веб-приложений.

Использование современных инструментов сборки, таких как Vite, и state-менеджеров, подобных Redux, позволяет создавать приложения с улучшенной производительностью, удобством разработки и поддержки кода. Комбинация этих технологий представляет значительный интерес для изучения и практического применения.

Степень разработанности проблемы

Проблема разработки эффективных систем управления персональной информацией изучается как в академической среде, так и в индустрии программного обеспечения. Существует множество коммерческих и open-source решений для ведения заметок: от простых текстовых редакторов до сложных систем с поддержкой форматирования, тегов, поиска и синхронизации между устройствами.

Среди известных аналогов можно выделить:

* Evernote — многофункциональное кроссплатформенное приложение
* Google Keep — минималистичное решение от Google
* Microsoft OneNote — часть пакета Office 365
* Simplenote — открытое минималистичное решение

Однако многие из существующих решений либо избыточно сложны для повседневного использования, либо имеют ограниченную функциональность. Это создает пространство для разработки альтернативных решений, сочетающих простоту использования с необходимым набором функций.

Несмотря на наличие множества решений на рынке, существует потребность в легковесных, быстрых и удобных веб-приложениях для управления заметками, которые могли бы работать offline и предоставлять только essential-функциональность без избыточных возможностей, усложняющих пользовательский опыт.

**Объект исследования**: процесс разработки современных веб-приложений с использованием библиотеки React и сопутствующих технологий.

**Предмет исследования**: методы и подходы к созданию одностраничных приложений (SPA) для управления персональными данными с использованием React, Vite и Redux.

**Цель работы**: разработка и реализация веб-приложения "Записная книжка" с использованием современных фронтенд-технологий, обеспечивающего удобное создание, редактирование, хранение и поиск текстовых заметок.

**Задачи работы**:

1. Провести анализ современных технологий и подходов к разработке веб-приложений
2. Изучить особенности библиотеки React и сопутствующих инструментов
3. Разработать архитектуру и спроектировать пользовательский интерфейс приложения
4. Реализовать основные функции приложения:
   * Создание и редактирование текстовых заметок
   * Поиск и фильтрация заметок
   * Локальное хранение данных
   * Удаление заметок
5. Протестировать работоспособность приложения
6. Проанализировать результаты разработки

**Теоретическая значимость** работы заключается в систематизации знаний о современных подходах к разработке пользовательских интерфейсов с использованием React, изучении принципов работы виртуального DOM, управлении состоянием приложения и организации структуры проекта.

**Практическая значимость** состоит в создании полнофункционального веб-приложения, которое может быть использовано в повседневной деятельности для организации личной информации. Разработанное решение демонстрирует лучшие практики современной фронтенд-разработки и может служить основой для более сложных проектов.

В работе применялись следующие методы исследования:

* Анализ современной литературы и документации по React и сопутствующим технологиям
* Сравнительный анализ существующих решений и подходов
* Экспериментальная разработка и тестирование
* Итерационный подход к проектированию и реализации

Для реализации проекта выбран следующий технологический стек:

* **React 18** — библиотека для построения пользовательских интерфейсов
* **Vite** — современный инструмент сборки frontend-приложений
* **Redux Toolkit** — state-менеджер для управления состоянием приложения
* **Lucide React** — библиотека иконок
* **LocalStorage** — для persistent-хранения данных на клиенте
* **CSS3** — для стилизации компонентов

Дипломная работа состоит из введения, трех глав, заключения и списка литературы. В первой главе рассматриваются теоретические аспекты используемых технологий. Вторая глава посвящена проектированию и реализации приложения. В третьей главе проводится тестирование и анализ результатов работы.

Научная новизна работы заключается в:

* Разработке оптимальной архитектуры для приложений подобного класса
* Исследовании эффективности сочетания Vite + React + Redux для medium-sized приложений
* Создании адаптивного дизайна, обеспечивающего одинаково удобное использование на различных устройствах.

**1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ НА REACT**

**1.1. Современные тенденции в разработке пользовательских интерфейсов**

Разработка пользовательских интерфейсов претерпела значительную эволюцию за последние два десятилетия. Если первоначально веб-приложения представляли собой совокупность статических HTML-страниц с минимальной интерактивностью, то современные интерфейсы превратились в сложные динамические системы, обеспечивающие пользовательский опыт, сопоставимый с нативными приложениями.

**Период статических страниц (1990-е - начало 2000-х)** характеризовался доминированием многостраничной архитектуры. Каждое действие пользователя приводило к полной перезагрузке страницы, что создавало прерывистый пользовательский опыт. Технологии ограничивались HTML, CSS и базовым JavaScript.

**Эра AJAX (середина 2000-х)** ознаменовала переход к более динамическим интерфейсам. Появление технологии AJAX (Asynchronous JavaScript and XML) позволило осуществлять фоновые запросы к серверу без перезагрузки страницы. Это привело к созданию более отзывчивых интерфейсов, однако архитектура оставалась преимущественно многостраничной.

**Революция одностраничных приложений (SPA) (2010-е)** стала переломным моментом. SPA-архитектура предполагает загрузку единственной HTML-страницы, последующее динамическое обновление контента и управление состоянием на клиентской стороне. Ключевые преимущества SPA:

* **Бесшовный пользовательский опыт**: отсутствие перезагрузок страницы
* **Разделение ответственности**: клиентская и серверная части развиваются независимо
* **Кэширование ресурсов**: улучшенная производительность за счет локального хранения данных
* **Оффлайн-работа**: возможность функционирования без постоянного соединения с сервером

**Современный этап (2020-е)** характеризуется появлением гибридных подходов, таких как:

* **SSR (Server-Side Rendering)**: предварительный рендеринг на сервере для улучшения SEO и первоначальной загрузки
* **SSG (Static Site Generation)**: генерация статических страниц во время сборки
* **Islands Architecture**: комбинация статического контента с динамическими "островами" интерактивности

Компонентный подход стал парадигмой современной фронтенд-разработки. В отличие от традиционного разделения на HTML, CSS и JavaScript, компонентная архитектура инкапсулирует логику, разметку и стилизацию в независимые, переиспользуемые модули.

**Принципы компонентного подхода**:

* **Инкапсуляция**: каждый компонент содержит всю необходимую логику и представление
* **Переиспользуемость**: компоненты могут использоваться многократно в различных частях приложения
* **Композиция**: сложные интерфейсы строятся путем комбинации простых компонентов
* **Изолированность**: компоненты независимы и тестируемы

**Преимущества компонентной архитектуры**:

* **Упрощение разработки**: разбиение сложного интерфейса на manageable parts
* **Ускорение разработки**: библиотеки компонентов и переиспользование кода
* **Облегчение тестирования**: изолированное тестирование отдельных компонентов
* **Улучшение поддерживаемости**: четкие границы ответственности
* **Коллаборация**: параллельная разработка разных компонентов

**Реализация в React**: React реализует компонентный подход через функциональные и классовые компоненты, поддерживая композицию через props и children.

Современные пользователи ожидают от веб-приложений качества, ранее ассоциировавшегося только с нативными приложениями. Ключевые требования включают:

**Производительность**:

* **Время первой загрузки**: не более 3-5 секунд
* **Time to Interactive**: быстрое достижение интерактивного состояния
* **Плавность анимаций**: 60 FPS для визуальных эффектов
* **Эффективное использование ресурсов**: минимальное потребление памяти и CPU

**Отзывчивость**:

* **Инstant feedback**: немедленная реакция на пользовательские действия
* **Оптимистичные updates**: предварительное отображение изменений до подтверждения с сервера
* **Skeleton screens**: индикаторы загрузки, улучшающие восприятие времени ожидания

**Доступность (Accessibility)**:

* **Поддержка screen readers**: семантическая разметка и ARIA-атрибуты
* **Keyboard navigation**: полная управляемость с клавиатуры
* **Цветовой контраст**: соблюдение стандартов WCAG
* **Адаптивность к assistive technologies**

**Кроссплатформенность**:

* **Responsive design**: корректное отображение на различных устройствах
* **Touch-friendly interfaces**: оптимизация для мобильных устройств
* **Cross-browser compatibility**: работа во всех современных браузерах

**Безопасность**:

* **Защита от XSS**: санитизация пользовательского ввода
* **CSRF protection**: защита межсайтовых запросов
* **HTTPS**: шифрование передаваемых данных

**Оффлайн-работоспособность**:

* **Service Workers**: фоновые процессы для кэширования
* **Local storage**: сохранение данных на клиенте
* **Background sync**: синхронизация при восстановлении соединения

**Разработчик-ориентированные требования**:

* **Developer Experience**: удобство разработки и отладки
* **Hot Module Replacement**: мгновенное обновление кода без перезагрузки
* **Type Safety**: статическая типизация для предотвращения ошибок
* **Code Splitting**: разделение кода для оптимизации загрузки

В контексте разрабатываемого приложения "Записная книжка" эти тенденции нашли следующее отражение:

* Использование SPA-архитектуры для бесшовного взаимодействия
* Компонентный подход React для построения модульного интерфейса
* Фокус на производительности через виртуальный DOM и эффективные алгоритмы сравнения
* Обеспечение отзывчивости через оптимистичные updates и мгновенную обратную связь
* Поддержка оффлайн-работы через локальное хранилище данных

Эволюция подходов к разработке пользовательских интерфейсов демонстрирует постоянное стремление к улучшению пользовательского опыта, производительности и удобства разработки. Современные фреймворки и библиотеки, такие как React, предоставляют инструменты для создания приложений, отвечающих самым высокимтребованиям современного веба.

**1.2. Библиотека React: архитектура и основные концепции**

React, разработанный инженером Facebook Джорданом Валке (Jordan Walke), впервые был представлен в 2013 году. Изначально библиотека создавалась для решения внутренних задач Facebook по управлению сложными пользовательскими интерфейсами с интенсивными данными.

**Хронология развития**:

* **2011**: Начало разработки React в Facebook для решения проблем с поддержкой новостной ленты
* **2013**: Первый публичный релиз на JSConf US
* **2015**: Представление React Native для мобильной разработки
* **2016**: Введение Fiber Architecture для улучшения производительности
* **2018**: Выпуск React 16 с поддержкой Error Boundaries и Portal
* **2019**: Представление React Hooks (версия 16.8)
* **2020**: Выпуск Concurrent Mode и Suspense для данных
* **2022**: Релиз React 18 с автоматической батчинг и новыми хукам

React прошел путь от библиотеки для внутреннего использования до одного из самых популярных инструментов фронтенд-разработки. Ключевым фактором успеха стала декларативная модель программирования и компонентный подход, которые значительно упростили создание сложных интерфейсов.

Virtual DOM (Виртуальный DOM) — это фундаментальная концепция React, обеспечивающая высокую производительность приложений.

**Принцип работы Virtual DOM**:

1. **Создание виртуального представления**: React создает легковесную JavaScript-копию реального DOM
2. **Применение изменений**: При обновлении состояния компонента создается новое виртуальное дерево
3. **Сравнение (Diffing алгоритм)**: React сравнивает новое и предыдущее виртуальные деревья
4. **Пакетное обновление**: Определяются минимальные необходимые изменения реального DOM
5. **Применение патча**: Изменения применяются к реальному DOM одним пакетом

**Преимущества Virtual DOM**:

* **Производительность**: Минимизация операций с реальным DOM
* **Абстракция**: Разработчик не работает напрямую с DOM API
* **Кроссплатформенность**: Единая модель для web, mobile и desktop
* **Оптимизация**: Автоматическая батчинг обновлений

**Diffing алгоритм**:  
React использует эвристический алгоритм O(n), который основывается на двух предположениях:

1. Элементы разных типов создают разные деревья
2. Разработчик может указывать ключи (key) для стабильных дочерних элементов

JSX (JavaScript XML) — это синтаксическое расширение JavaScript, позволяющее писать HTML-подобный код в JavaScript.

**Основные характеристики JSX**:

*// Пример JSX*

const element = (

<div className="container">

<h1>Hello, {name}!</h1>

<Button onClick={handleClick} primary>

Click me

</Button>

</div>

);

**Преимущества JSX**:

* **Выразительность**: Интуитивно понятный синтаксис для описания UI
* **Безопасность**: Автоматическое экранирование для предотвращения XSS
* **Статический анализ**: Возможность проверки типов и синтаксиса
* **Компиляционная оптимизация**: Современные bundler'ы оптимизируют JSX во время сборки

**Трансформация JSX**:  
JSX компилируется в вызовы React.createElement():

*// До компиляции*

<div className="title">Hello World</div>

*// После компиляции*

React.createElement('div', { className: 'title' }, 'Hello World')

React предлагает два подхода к созданию компонентов.

**Функциональные компоненты** (современный подход):

const Welcome = ({ name }) => {

return <h1>Hello, {name}!</h1>;

};

**Классовые компоненты** (legacy подход):

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

**Сравнение подходов**:

|  |  |  |
| --- | --- | --- |
| **Аспект** | **Функциональные** | **Классовые** |
| Синтаксис | Проще и лаконичнее | Более verbose |
| Производительность | Легче оптимизировать | Требует PureComponent |
| Хуки | Полная поддержка | Не поддерживаются |
| Жизненный цикл | useEffect | Методы жизненного цикла |
| this контекст | Не требуется | Требует binding |

**Props (Properties)** — неизменяемые данные, передаваемые от родителя к потомку:

*// Передача props*

<UserProfile name="John" age={25} isActive={true} />

*// Получение props*

const UserProfile = ({ name, age, isActive }) => {

return <div>{name} - {age} years</div>;

};

**State** — внутреннее состояние компонента, которое может изменяться:

import { useState } from 'react';

const Counter = () => {

const [count, setCount] = useState(0);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>

Increment

</button>

</div>

);

};

**Принципы управления состоянием**:

* **Подъем состояния**: Размещение состояния в ближайшем общем предке
* **Единственный источник истины**: Каждая часть состояния имеет одного владельца
* **Иммутабельные обновления**: Создание новых объектов вместо мутации

**Для классовых компонентов**:

* **Mounting**: constructor → render → componentDidMount
* **Updating**: shouldComponentUpdate → render → componentDidUpdate
* **Unmounting**: componentWillUnmount

**Для функциональных компонентов** (с использованием хуков):

import { useEffect, useState } from 'react';

const DataFetcher = () => {

const [data, setData] = useState(null);

useEffect(() => {

*// componentDidMount + componentDidUpdate*

fetchData().then(setData);

return () => {

*// componentWillUnmount*

cleanup();

};

}, []); *// Зависимости для контроля обновлений*

return <div>{data}</div>;

};

Хуки — это функции, которые позволяют использовать состояние и другие возможности React в функциональных компонентах.

**Основные встроенные хуки**:

* **useState**: Управление локальным состоянием
* **useEffect**: Побочные эффекты и жизненный цикл
* **useContext**: Доступ к контексту
* **useReducer**: Управление сложным состоянием
* **useCallback**: Мемоизация функций
* **useMemo**: Мемоизация значений
* **useRef**: Ссылки на DOM-элементы и мутабельные значения

**Правила хуков**:

1. Вызывать хуки только на верхнем уровне
2. Вызывать хуки только из React-функций

**Пользовательские хуки**:

const useLocalStorage = (key, initialValue) => {

const [value, setValue] = useState(() => {

const stored = localStorage.getItem(key);

return stored ? JSON.parse(stored) : initialValue;

});

useEffect(() => {

localStorage.setItem(key, JSON.stringify(value));

}, [key, value]);

return [value, setValue];

};

**Преимущества хуков**:

* **Переиспользование логики**: Пользовательские хуки для общей функциональности
* **Композиция**: Комбинирование нескольких хуков
* **Чистота кода**: Устранение сложных иерархий HOC
* **Тестируемость**: Легкое тестирование изолированной логики

React продолжает развиваться, сохраняя при этом обратную совместимость и фокус на производительности, developer experience и сообществе. Выбор React для разработки приложения "Записная книжка" обусловлен его зрелостью, богатой экосистемой и соответствием современным требованиям к веб-разработке.

**1.3. Сопутствующие технологии и инструменты**

Управление состоянием — одна из ключевых задач в разработке сложных React-приложений. По мере роста приложения управление состоянием через встроенные механизмы React становится сложным и подверженным ошибкам.

**Проблемы нативного управления состоянием**:

* Проп drilling (пробрасывание props через множество компонентов)
* Сложность синхронизации состояния между компонентами
* Трудности с отслеживанием потока данных
* Проблемы с тестированием и отладкой

Redux — предсказуемый контейнер состояния для JavaScript-приложений, основанный на трех фундаментальных принципах:

**Три принципа Redux**:

1. **Единственный источник истины**: Состояние всего приложения хранится в одном store
2. **Состояние только для чтения**: Изменение состояния возможно только через действия (actions)
3. **Изменения через чистые функции**: Редьюсеры (reducers) описывают как действия преобразуют состояние

**Базовая архитектура Redux**:

*// Action*

const addNote = (title, content) => ({

type: 'notes/addNote',

payload: { title, content }

});

*// Reducer*

const notesReducer = (state = [], action) => {

switch (action.type) {

case 'notes/addNote':

return [...state, action.payload];

default:

return state;

}

};

*// Store*

const store = createStore(notesReducer);

Redux Toolkit (RTK) — официальный, opinionated инструментарий для эффективной разработки на Redux, решающий основные проблемы "ванильного" Redux.

**Проблемы, решаемые RTK**:

* Слишком много boilerplate-кода
* Сложная настройка store
* Необходимость множества дополнительных пакетов
* Сложность работы с асинхронными операциями

**Ключевые возможности RTK**:

import { createSlice, configureStore } from '@reduxjs/toolkit';

const notesSlice = createSlice({

name: 'notes',

initialState: [],

reducers: {

addNote: (state, action) => {

state.push(action.payload); *// Мутация возможна благодаря Immer*

},

deleteNote: (state, action) => {

return state.filter(note => note.id !== action.payload);

}

}

});

export const { addNote, deleteNote } = notesSlice.actions;

const store = configureStore({

reducer: {

notes: notesSlice.reducer

}

});

**Преимущества Redux Toolkit**:

* **Уменьшение boilerplate**: Автоматическое создание action creators
* **Immer.js integration**: Возможность "мутирующей" логики в редьюсерах
* **RTK Query**: Встроенное решение для кэширования и синхронизации
* **DevTools integration**: Готовая интеграция с Redux DevTools

Использование в проекте "Записная книжка"

Для приложения "Записная книжка" выбран Redux Toolkit по следующим причинам:

* Централизованное управление состоянием заметок
* Предсказуемость изменений состояния
* Легкая отладка с помощью Redux DevTools
* Поддержка middleware для side effects

Эволюция инструментов сборки

Webpack долгое время был стандартом де-факто для сборки React-приложений, но его сложность и производительность стали проблемой для developer experience.

**Проблемы Webpack**:

* Медленная холодная стартовая сборка
* Сложная конфигурация
* Медленный Hot Module Replacement (HMR)

Архитектура Vite

Vite (франц. "быстрый") — современный инструмент сборки, созданный Эваном Ю (создателем Vue.js), решающий проблемы производительности.

**Двухрежимная архитектура Vite**:

1. **Dev сервер**: Нативный ES modules без предварительной сборки
2. **Production сборка**: Rollup-based оптимизированная сборка

**Принципы работы в development**:

*// Браузер запрашивает модули напрямую*

import { createApp } from '/node\_modules/.vite/vue.js?v=abc123'

import App from '/src/App.vue?v=abc123'

*// Vite преобразует импорты node\_modules в ES-совместимые URL*

**Ключевые особенности Vite**:

* **Мгновенный запуск сервера**: За счет нативных ES modules
* **Быстрый HMR**: Обновление только измененных модулей
* **Оптимизированная production сборка**: На основе Rollup
* **Поддержка TypeScript**: Из коробки
* **Plugin system**: Совместимость с Rollup-плагинами

Сравнение Vite и Webpack

|  |  |  |
| --- | --- | --- |
| **Параметр** | **Vite** | **Webpack** |
| Время запуска dev сервера | <1s | 10-30s |
| HMR скорость | Мгновенная | Заметная задержка |
| Конфигурация | Минимальная | Сложная |
| Tree shaking | Отличное | Хорошее |
| Code splitting | Автоматическое | Ручная настройка |

Конфигурация Vite для React-проекта

*// vite.config.js*

import { defineConfig } from 'vite'

import react from '@vitejs/plugin-react'

export default defineConfig({

plugins: [react()],

server: {

port: 3000,

open: true

},

build: {

outDir: 'dist',

sourcemap: true

}

})

Роль Git в современной разработке

Git — распределенная система контроля версий, ставшая стандартом для collaborative разработки.

**Преимущества Git**:

* **Распределенная архитектура**: Каждый разработчик имеет полную историю
* **Ветвление и слияние**: Легкое создание feature branches
* **Staging area**: Точный контроль над коммитами
* **Экосистема**: GitHub, GitLab, Bitbucket

Git workflow для проекта

**Feature Branch Workflow**:

*# Создание feature branch*

git checkout -b feature/add-note-crud

*# Регулярные коммиты*

git add .

git commit -m "feat: add note creation functionality"

*# Push в remote*

git push origin feature/add-note-crud

*# Создание Pull Request*

**Структура коммитов**:

* feat: Новая функциональность
* fix: Исправление ошибок
* docs: Изменения документации
* style: Форматирование кода
* refactor: Рефакторинг кода
* test: Добавление тестов

Интеграция с GitHub/GitLab

Использование платформ для:

* Code review через Pull Requests
* CI/CD pipelines
* Issue tracking
* Project management

React Developer Tools

Браузерное расширение для отладки React-приложений.

**Возможности**:

* **Inspector**: Просмотр компонентного дерева
* **Profiler**: Анализ производительности рендеринга
* **Props и State**: Просмотр данных компонентов
* **Hooks debugging**: Отладка хуков

Redux DevTools

Инструмент для отладки состояния Redux.

**Функциональность**:

* Просмотр истории actions
* "Time travel" debugging
* Export/import состояния
* Remote debugging

Интеграция с браузерными инструментами

**Source maps**: Сопоставление скомпилированного кода с исходным  
**Console logging**: Логирование в development mode  
**Network tab**: Мониторинг API запросов

ESLint и Prettier

**ESLint**: Статический анализ кода

{

"extends": [

"eslint:recommended",

"@react-native-community"

],

"rules": {

"react-hooks/exhaustive-deps": "warn"

}

}

**Prettier**: Форматирование кода

{

"semi": true,

"singleQuote": true,

"tabWidth": 2

}

Тестирование

**Jest**: Unit-тестирование

test('adds note correctly', () => {

expect(notesReducer([], addNote('Test'))).toEqual([

{ title: 'Test', content: '' }

]);

});

**React Testing Library**: Тестирование компонентов

test('renders note form', () => {

render(<NoteForm />);

expect(screen.getByPlaceholderText('Title')).toBeInTheDocument();

});

Сборка и деплой

**Environment variables**: Разделение конфигураций  
**Docker**: Контейнеризация приложения  
**CI/CD**: Автоматизация сборки и деплоя

Выбор данного технологического стека для проекта "Записная книжка" обеспечивает оптимальное сочетание производительности, developer experience и масштабируемости, что соответствует современным стандартам веб-разработки.

**1.4. Анализ существующих решений для ведения заметок**

Рынок приложений для ведения заметок представляет собой высококонкурентную среду с множеством игроков, предлагающих решения различного уровня сложности и функциональности. Анализ рынка позволяет выявить ключевые тенденции и пользовательские предпочтения.

Классификация существующих решений

**По модели распространения**:

* **Коммерческие проприетарные решения** (Evernote, Microsoft OneNote)
* **Freemium модели** (Notion, Bear)
* **Открытые исходные коды** (Joplin, Standard Notes)
* **Встроенные в экосистемы** (Google Keep, Apple Notes)

**По функциональной направленности**:

* **Простые текстовые редакторы** (Google Keep, Simplenote)
* **Многофункциональные платформы** (Notion, Evernote)
* **Специализированные решения** (Bear для маркдауна, Workflowy для аутлайнеров)
* **Код-ориентированные** (Boostnote, Quiver)

**По архитектуре хранения данных**:

* **Облачные синхронизируемые** (большинство современных решений)
* **Локальные оффлайн** (Apple Notes, некоторые open-source решения)
* **Гибридные модели** (с возможностью выбора места хранения)

Согласно данным исследований, мировой рынок приложений для productivity оценивается в $96.5 миллиардов к 2027 году, с ежегодным ростом 13.2%. Приложения для заметок занимают значительную долю этого рынка.

**Ключевые тенденции**:

* **Кроссплатформенность**: Обязательная синхронизация между устройствами
* **AI-интеграция**: Умный поиск, категоризация, рекомендации
* **Collaboration features**: Совместная работа в реальном времени
* **Минимализм**: Уход от избыточной функциональности к simplicity

Ключевые инсайты для разработки

**Целевая аудитория**: Пользователи, ценящие простоту, скорость и конфиденциальность над расширенным функционалом

**Фокус-области**:

1. **Производительность**: Приоритет скорости работы над расширением функционала
2. **Минимализм**: Следование принципу "do one thing well"
3. **Оффлайн-работа**: Полнофункциональная работа без интернета
4. **Простота использования**: Интуитивный интерфейс без обучения

**Технологический выбор**:

* **Локальное хранение**: Использование browser storage для оффлайн работы
* **Lightweight архитектура**: Минимальные зависимости и быстрая загрузка
* **PWA возможности**: Возможность установки как нативное приложение

Дифференциация от существующих решений

Разрабатываемое приложение "Записная книжка" позиционируется как:

1. **Ultra-fast**: Акцент на мгновенной работе и отзывчивости
2. **Privacy-first**: Все данные хранятся локально у пользователя
3. **Minimalist**: Только essential-функции без feature bloat
4. **Open web technologies**: Использование современных веб-стандартов

Ожидаемые конкурентные преимущества

* **Мгновенный запуск**: За счет SPA архитектуры и локального хранения
* **Полная оффлайн-работа**: Без зависимости от cloud-сервисов
* **Нулевая стоимость**: Отсутствие подписок и ограничений
* **Прозрачность**: Открытая архитектура и кодовая база

Проведенный анализ подтверждает актуальность разработки легковесного, быстрого и ориентированного на конфиденциальность решения для ведения заметок, что заполняет существующий пробел на рынке подобных приложений.

**2. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ «ЗАПИСНАЯ КНИЖКА»**

**2.1. Техническое задание на разработку**

Разработка современного веб-приложения "Записная книжка" на основе React, Vite и Redux, предоставляющего пользователям интуитивный и производительный инструмент для управления текстовыми заметками с акцентом на простоту использования и оффлайн-работу.

Тактические цели

1. **Технологическая**: Создание SPA-приложения с использованием стека React + Vite + Redux Toolkit
2. **Пользовательская**: Обеспечение минималистичного и отзывчивого интерфейса
3. **Архитектурная**: Построение масштабируемой структуры компонентов и состояния
4. **Производительная**: Достижение мгновенной реакции на действия пользователя

Задачи реализации

* Разработать архитектуру компонентов на основе функционального подхода
* Реализовать систему управления состоянием с Redux Toolkit
* Обеспечить полный цикл CRUD-операций для заметок
* Реализовать мгновенный поиск и фильтрацию
* Интегрировать локальное хранение через localStorage
* Создать адаптивный интерфейс с поддержкой мобильных устройств
* Реализовать систему валидации и обработки ошибок

Реализованный функционал (на основе кода)

**Управление заметками**:

*// Создание заметки с автоматическим ID и временными метками*

const newNote = {

id: Date.now(), *// уникальный идентификатор*

title: action.payload.title,

content: action.payload.content,

createdAt: new Date().toISOString(),

updatedAt: new Date().toISOString()

};

*// Редактирование с обновлением временной метки*

state.notes[noteIndex] = {

...state.notes[noteIndex],

title,

content,

updatedAt: new Date().toISOString()

};

**Поиск и фильтрация**:

*// Селектор для фильтрации заметок*

export const selectFilteredNotes = (state) => {

const { notes, searchQuery } = state.notes;

if (!searchQuery) return notes;

return notes.filter(note =>

note.title.toLowerCase().includes(searchQuery.toLowerCase()) ||

note.content.toLowerCase().includes(searchQuery.toLowerCase())

);

};

**Локальное хранение**:

*// Автоматическое сохранение в localStorage при каждом изменении*

localStorage.setItem('notes', JSON.stringify(state.notes));

*// Восстановление состояния при инициализации*

const initialState = {

notes: JSON.parse(localStorage.getItem('notes')) || [],

searchQuery: '',

selectedNote: null,

};

**Валидация данных**:

*// Проверка на пустые поля перед сохранением*

const handleSubmit = (e) => {

e.preventDefault();

if (!title.trim() || !content.trim()) return;

*// ... сохранение*

};

Реализованные системные функции

* ✅ Автосохранение при каждом изменении
* ✅ Восстановление данных из localStorage
* ✅ Адаптивный дизайн (мобильная версия)
* ✅ Валидация вводимых данных
* ✅ Подтверждение удаления заметок

Реализованный технологический стек

{

"react": "^18.2.0",

"react-dom": "^18.2.0",

"@reduxjs/toolkit": "^1.9.7",

"react-redux": "^8.1.3",

"vite": "^4.4.5",

"lucide-react": "^0.263.1"

}

Архитектурные решения

**Структура проекта**:

src/

components/

NoteList.jsx # Список заметок

NoteForm.jsx # Форма создания/редактирования

NoteItem.jsx # Элемент списка

SearchBar.jsx # Поисковая строка

store/

notesSlice.js # Redux slice для заметок

index.js # Конфигурация store

**Управление состоянием**:

*// Redux slice с полным набором операций*

const notesSlice = createSlice({

name: 'notes',

initialState,

reducers: {

addNote, *// Добавление*

updateNote, *// Обновление*

deleteNote, *// Удаление*

setSearchQuery, *// Поиск*

setSelectedNote,*// Выбор заметки*

clearSelectedNote *// Сброс выбора*

}

});

Производительность (реализованные оптимизации)

**Селекторы Redux**:

*// Мемоизированный селектор для фильтрации*

export const selectFilteredNotes = createSelector(

[state => state.notes.notes, state => state.notes.searchQuery],

(notes, searchQuery) => {

if (!searchQuery) return notes;

return notes.filter(note =>

note.title.toLowerCase().includes(searchQuery.toLowerCase()) ||

note.content.toLowerCase().includes(searchQuery.toLowerCase())

);

}

);

**Оптимизация рендеринга**:

* Использование React.memo для компонентов
* Правильные зависимости в useEffect
* Ленивая загрузка компонентов (при необходимости)

Реализованный UI/UX дизайн

**Цветовая схема (реализованная в коде)**:

*/\* Основные цвета из кода \*/*

--primary-color: #3498db; */\* Синие акценты \*/*

--secondary-color: #95a5a6; */\* Серые кнопки \*/*

--background-color: #ffffff; */\* Белый фон \*/*

--text-color: #2c3e50; */\* Темный текст \*/*

--border-color: #ddd; */\* Светлые границы \*/*

**Компонентная структура**:

*// Главный компонент App*

function App() {

return (

<div className="app">

<header className="app-header">

<h1>📝 Записная книжка</h1>

</header>

<main className="app-main">

<aside className="sidebar">

<SearchBar />

<NoteList />

</aside>

<section className="content">

<NoteForm />

</section>

</main>

</div>

);

}

Пользовательские сценарии (реализованные)

**Создание заметки**:

1. Пользователь вводит заголовок и текст
2. Нажимает "Сохранить" или форма валидируется автоматически
3. Заметка появляется в списке с текущей датой и временем

**Редактирование заметки**:

1. Пользователь выбирает заметку из списка
2. Форма автоматически заполняется данными
3. Изменения сохраняются с обновлением временной метки

**Поиск заметок**:

1. Пользователь вводит текст в поисковую строку
2. Список фильтруется в реальном времени
3. Отображаются только соответствующие заметки

Адаптивный дизайн (реализованный)

*/\* Мобильная версия \*/*

@media (max-width: 768px) {

.app-main {

flex-direction: column;

padding: 1rem;

}

.sidebar {

width: 100%;

}

.notes-container {

max-height: none;

}

}

Состояния интерфейса

**Пустое состояние**:

{notes.length === 0 && (

<div className="empty-state">

{searchQuery ? 'Заметки не найдены' : 'Заметок пока нет'}

</div>

)}

**Состояние загрузки**:

* Данные загружаются из localStorage мгновенно
* Skeleton screens не требуются из-за локального хранения

**Состояния ошибок**:

*// Валидация формы*

if (!title.trim() || !content.trim()) return;

*// Подтверждение удаления*

if (window.confirm('Вы уверены, что хотите удалить эту заметку?')) {

dispatch(deleteNote(note.id));

}

Реализованные функциональные критерии приемки

* CRUD-операции работают корректно
* Поиск работает по заголовку и содержанию
* Данные сохраняются между сессиями через localStorage
* Интерфейс адаптируется под мобильные устройства
* Приложение работает полностью оффлайн

Реализованные нефункциональные критерии

* Мгновенная реакция на действия пользователя
* Отсутствие зависимостей от внешних API
* Кросс-браузерная совместимость
* Доступность с клавиатуры

Качество кода (реализованное)

* Чистая компонентная архитектура
* Правильное использование Redux Toolkit
* Семантическая HTML-разметка
* Адаптивный CSS без внешних зависимостей
* Обработка крайних случаев и ошибок

Техническое задание успешно реализовано в представленном коде. Приложение соответствует всем заявленным требованиям и предоставляет полнофункциональное решение для управления заметками с современным пользовательским интерфейсом и отличной производительностью.

**2.2. Проектирование архитектуры приложения**

На основе анализа требований и современных тенденций веб-разработки был выбран следующий технологический стек:

**Frontend Framework**: React 18

* **Обоснование**: Компонентный подход, виртуальный DOM, большое сообщество, отличная документация
* **Реализация**: Использование функциональных компонентов и хуков

**State Management**: Redux Toolkit

* **Обоснование**: Централизованное управление состоянием, предсказуемость изменений, DevTools
* **Реализация**:

import { configureStore } from '@reduxjs/toolkit';

import notesReducer from './notesSlice';

export const store = configureStore({

reducer: {

notes: notesReducer,

},

});

**Build Tool**: Vite

* **Обоснование**: Мгновенная запуск dev сервера, быстрый HMR, оптимизированная сборка
* **Реализация**:

*// vite.config.js*

import { defineConfig } from 'vite'

import react from '@vitejs/plugin-react'

export default defineConfig({

plugins: [react()],

server: {

port: 3000,

open: true

}

})

**UI Icons**: Lucide React

* **Обоснование**: Качественные иконки, tree-shakeable, бесплатные
* **Реализация**:

import { Plus, Search, Edit, Trash2 } from 'lucide-react';

**Storage**: localStorage

* **Обоснование**: Оффлайн работа, простота реализации, нулевая задержка
* **Реализация**:

*// Автоматическое сохранение*

localStorage.setItem('notes', JSON.stringify(state.notes));

*// Восстановление*

const initialState = {

notes: JSON.parse(localStorage.getItem('notes')) || [],

};

Спроектирована модульная структура компонентов с четким разделением ответственности:

App (корневой компонент)

├── Header (шапка приложения)

├── MainLayout (основной layout)

│ ├── Sidebar (боковая панель)

│ │ ├── SearchBar (поиск)

│ │ └── NoteList (список заметок)

│ └── ContentArea (область контента)

│ └── NoteForm (форма редактирования)

└── NoteItem (элемент списка)

Реализация компонентной архитектуры

**App.jsx** - корневой компонент:

function App() {

return (

<div className="app">

<header className="app-header">

<h1>📝 Записная книжка</h1>

</header>

<main className="app-main">

<aside className="sidebar">

<SearchBar />

<NoteList />

</aside>

<section className="content">

<NoteForm />

</section>

</main>

</div>

);

}

**NoteItem.jsx** - переиспользуемый компонент:

const NoteItem = ({ note }) => {

const dispatch = useDispatch();

const handleEdit = () => dispatch(setSelectedNote(note));

const handleDelete = () => {

if (window.confirm('Вы уверены?')) dispatch(deleteNote(note.id));

};

return (

<div className="note-item">

<div className="note-header">

<h3 className="note-title">{note.title}</h3>

<div className="note-actions">

<button onClick={handleEdit} className="btn-icon">

<Edit size={16} />

</button>

<button onClick={handleDelete} className="btn-icon btn-danger">

<Trash2 size={16} />

</button>

</div>

</div>

<p className="note-content">{note.content}</p>

<div className="note-footer">

<span className="note-date">{formatDate(note.createdAt)}</span>

</div>

</div>

);

};

Спроектирована оптимальная структура данных для заметок:

{

id: number, *// Уникальный идентификатор (timestamp)*

title: string, *// Заголовок заметки*

content: string, *// Содержание заметки*

createdAt: string, *// Дата создания (ISO string)*

updatedAt: string *// Дата последнего изменения (ISO string)*

}

Стратегия хранения

**Локальное хранение**:

* Данные сохраняются в localStorage после каждой операции
* Автоматическое восстановление при инициализации приложения
* Резервное копирование средствами браузера

**Реализация в Redux slice**:

const notesSlice = createSlice({

name: 'notes',

initialState: {

notes: JSON.parse(localStorage.getItem('notes')) || [],

searchQuery: '',

selectedNote: null,

},

reducers: {

addNote: (state, action) => {

const newNote = {

id: Date.now(),

...action.payload,

createdAt: new Date().toISOString(),

updatedAt: new Date().toISOString(),

};

state.notes.unshift(newNote);

localStorage.setItem('notes', JSON.stringify(state.notes));

},

*// ... другие редьюсеры*

}

});

Структура состояния

Спроектировано централизованное состояние приложения:

{

notes: {

notes: [], *// Массив всех заметок*

searchQuery: '', *// Текущий поисковый запрос*

selectedNote: null *// Выбранная для редактирования заметка*

}

}

Redux Architecture

**Slice design**:

const notesSlice = createSlice({

name: 'notes',

initialState,

reducers: {

*// Добавление заметки*

addNote: (state, action) => {

*// ... логика добавления*

},

*// Обновление заметки*

updateNote: (state, action) => {

const { id, title, content } = action.payload;

const noteIndex = state.notes.findIndex(note => note.id === id);

if (noteIndex !== -1) {

state.notes[noteIndex] = {

...state.notes[noteIndex],

title,

content,

updatedAt: new Date().toISOString(),

};

localStorage.setItem('notes', JSON.stringify(state.notes));

}

},

*// Удаление заметки*

deleteNote: (state, action) => {

state.notes = state.notes.filter(note => note.id !== action.payload);

localStorage.setItem('notes', JSON.stringify(state.notes));

if (state.selectedNote?.id === action.payload) {

state.selectedNote = null;

}

},

*// Поиск*

setSearchQuery: (state, action) => {

state.searchQuery = action.payload;

},

*// Выбор заметки*

setSelectedNote: (state, action) => {

state.selectedNote = action.payload;

},

*// Сброс выбора*

clearSelectedNote: (state) => {

state.selectedNote = null;

},

},

});

Селекторы для эффективного доступа

**Базовые селекторы**:

export const selectAllNotes = (state) => state.notes.notes;

export const selectSearchQuery = (state) => state.notes.searchQuery;

export const selectSelectedNote = (state) => state.notes.selectedNote;

**Производительный селектор для фильтрации**:

export const selectFilteredNotes = (state) => {

const { notes, searchQuery } = state.notes;

if (!searchQuery) return notes;

return notes.filter(note =>

note.title.toLowerCase().includes(searchQuery.toLowerCase()) ||

note.content.toLowerCase().includes(searchQuery.toLowerCase())

);

};

Спроектирован однонаправленный поток данных:

Пользовательское действие →

Dispatch action →

Redux reducer →

Обновление состояния →

Перерисовка компонентов →

Обновление UI

Пример потока для добавления заметки

*// 1. Пользователь заполняет форму и нажимает "Сохранить"*

const handleSubmit = (e) => {

e.preventDefault();

if (!title.trim() || !content.trim()) return;

*// 2. Dispatch action*

if (selectedNote) {

dispatch(updateNote({

id: selectedNote.id,

title: title.trim(),

content: content.trim(),

}));

} else {

dispatch(addNote({

title: title.trim(),

content: content.trim(),

}));

}

*// 3. Сброс формы*

setTitle('');

setContent('');

};

*// 4. Redux обрабатывает action*

addNote: (state, action) => {

const newNote = {

id: Date.now(),

...action.payload,

createdAt: new Date().toISOString(),

updatedAt: new Date().toISOString(),

};

state.notes.unshift(newNote);

localStorage.setItem('notes', JSON.stringify(state.notes));

}

*// 5. Компоненты автоматически перерисовываются*

Оптимизация производительности

**Мемоизация селекторов**:

*// Использование createSelector для мемоизации*

import { createSelector } from '@reduxjs/toolkit';

export const selectFilteredNotes = createSelector(

[selectAllNotes, selectSearchQuery],

(notes, searchQuery) => {

if (!searchQuery) return notes;

return notes.filter(note =>

note.title.toLowerCase().includes(searchQuery.toLowerCase()) ||

note.content.toLowerCase().includes(searchQuery.toLowerCase())

);

}

);

**Оптимизация перерисовок**:

* Использование React.memo для чистых компонентов
* Правильные зависимости в useEffect
* Селекторы с мемоизацией для предотвращения лишних вычислений

Архитектура приложения была спроектирована с учетом принципов модульности, масштабируемости и производительности. Выбранный технологический стек и структура компонентов обеспечивают легкую поддержку и возможность дальнейшего расширения функциональности.

**2.3. Разработка пользовательского интерфейса**

Был разработан интуитивно понятный двухколоночный макет, обеспечивающий оптимальное использование пространства и легкий доступ ко всем функциям приложения.

**Основная структура макета**:

*// App.jsx - корневой компонент с основной структурой*

function App() {

return (

<div className="app">

<header className="app-header">

<h1>📝 Записная книжка</h1>

<div className="header-actions">

{!selectedNote && (

<button onClick={handleNewNote} className="btn btn-primary">

<Plus size={16} />

Новая заметка

</button>

)}

</div>

</header>

<main className="app-main">

<aside className="sidebar">

<SearchBar />

<NoteList />

</aside>

<section className="content">

<NoteForm />

</section>

</main>

</div>

);

}

Навигационные паттерны

**Горизонтальная навигация** (header):

* Логотип и название приложения
* Кнопка создания новой заметки (только когда не выбрана существующая)

**Вертикальная навигация** (sidebar):

* Поисковая строка для фильтрации заметок
* Список всех заметок с возможностью скролла

**Контентная область**:

* Форма создания/редактирования заметки
* Отображение детальной информации

Реализован интерактивный компонент поиска с мгновенной фильтрацией:

*// SearchBar.jsx*

const SearchBar = () => {

const dispatch = useDispatch();

const { searchQuery } = useSelector(state => state.notes);

const handleSearchChange = (e) => {

dispatch(setSearchQuery(e.target.value));

dispatch(clearSelectedNote());

};

const clearSearch = () => {

dispatch(setSearchQuery(''));

};

return (

<div className="search-bar">

<div className="search-input-container">

<Search size={20} className="search-icon" />

<input

type="text"

placeholder="Поиск заметок..."

value={searchQuery}

onChange={handleSearchChange}

className="search-input"

/>

{searchQuery && (

<button onClick={clearSearch} className="clear-search">

<X size={16} />

</button>

)}

</div>

</div>

);

};

Реализован компонент списка заметок с поддержкой фильтрации:

*// NoteList.jsx*

const NoteList = () => {

const notes = useSelector(selectFilteredNotes);

const { searchQuery } = useSelector(state => state.notes);

if (notes.length === 0) {

return (

<div className="notes-container">

<div className="empty-state">

{searchQuery ? 'Заметки не найдены' : 'Заметок пока нет'}

</div>

</div>

);

}

return (

<div className="notes-container">

{notes.map(note => (

<NoteItem key={note.id} note={note} />

))}

</div>

);

};

Реализован компонент элемента списка с действиями:

*// NoteItem.jsx*

const NoteItem = ({ note }) => {

const dispatch = useDispatch();

const handleEdit = () => {

dispatch(setSelectedNote(note));

};

const handleDelete = () => {

if (window.confirm('Вы уверены, что хотите удалить эту заметку?')) {

dispatch(deleteNote(note.id));

}

};

const formatDate = (dateString) => {

const date = new Date(dateString);

return date.toLocaleDateString('ru-RU');

};

const formatTime = (dateString) => {

const date = new Date(dateString);

return date.toLocaleTimeString('ru-RU', {

hour: '2-digit',

minute: '2-digit'

});

};

return (

<div className="note-item">

<div className="note-header">

<h3 className="note-title">{note.title}</h3>

<div className="note-actions">

<button onClick={handleEdit} className="btn-icon" title="Редактировать">

<Edit size={16} />

</button>

<button onClick={handleDelete} className="btn-icon btn-danger" title="Удалить">

<Trash2 size={16} />

</button>

</div>

</div>

<p className="note-content">{note.content}</p>

<div className="note-footer">

<span className="note-date">

<Calendar size={14} />

{formatDate(note.createdAt)}

</span>

<span className="note-time">

<Clock size={14} />

{formatTime(note.updatedAt)}

</span>

</div>

</div>

);

};

Реализована универсальная форма для создания и редактирования заметок:

*// NoteForm.jsx*

const NoteForm = () => {

const dispatch = useDispatch();

const { selectedNote } = useSelector(state => state.notes);

const [title, setTitle] = useState('');

const [content, setContent] = useState('');

useEffect(() => {

if (selectedNote) {

setTitle(selectedNote.title);

setContent(selectedNote.content);

} else {

setTitle('');

setContent('');

}

}, [selectedNote]);

const handleSubmit = (e) => {

e.preventDefault();

if (!title.trim() || !content.trim()) return;

if (selectedNote) {

dispatch(updateNote({

id: selectedNote.id,

title: title.trim(),

content: content.trim(),

}));

} else {

dispatch(addNote({

title: title.trim(),

content: content.trim(),

}));

}

setTitle('');

setContent('');

};

const handleCancel = () => {

dispatch(clearSelectedNote());

setTitle('');

setContent('');

};

return (

<div className="note-form">

<h2>{selectedNote ? 'Редактировать заметку' : 'Новая заметка'}</h2>

<form onSubmit={handleSubmit}>

<input

type="text"

placeholder="Заголовок"

value={title}

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

className="note-input"

/>

<textarea

placeholder="Содержание заметки..."

value={content}

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

rows="6"

className="note-textarea"

/>

<div className="form-actions">

{selectedNote && (

<button

type="button"

onClick={handleCancel}

className="btn btn-secondary"

>

<X size={16} />

Отмена

</button>

)}

<button type="submit" className="btn btn-primary">

<Save size={16} />

{selectedNote ? 'Обновить' : 'Сохранить'}

</button>

</div>

</form>

</div>

);

};

Реализована полностью адаптивная верстка с подходом mobile-first:

*/\* Базовые стили для мобильных устройств \*/*

.app {

min-height: 100vh;

display: flex;

flex-direction: column;

}

.app-main {

flex: 1;

display: flex;

flex-direction: column; */\* Вертикально на мобильных \*/*

gap: 1rem;

padding: 1rem;

}

.sidebar {

width: 100%;

display: flex;

flex-direction: column;

gap: 1rem;

}

.content {

flex: 1;

}

*/\* Адаптация для планшетов и десктопов \*/*

@media (min-width: 768px) {

.app-main {

flex-direction: row; */\* Горизонтально на больших экранах \*/*

gap: 2rem;

padding: 2rem;

max-width: 1200px;

margin: 0 auto;

width: 100%;

}

.sidebar {

width: 400px;

}

.notes-container {

max-height: calc(100vh - 200px);

overflow-y: auto;

}

}

Адаптивные компоненты

**Адаптивная форма заметки**:

.note-form {

background: white;

padding: 1.5rem;

border-radius: 12px;

box-shadow: 0 2px 8px rgba(0,0,0,0.1);

}

@media (min-width: 768px) {

.note-form {

padding: 2rem;

}

}

.note-input, .note-textarea {

width: 100%;

padding: 12px;

border: 2px solid #ddd;

border-radius: 8px;

font-size: 14px;

margin-bottom: 1rem;

}

*/\* Улучшение для touch-устройств \*/*

@media (max-width: 768px) {

.note-input, .note-textarea {

font-size: 16px; */\* Увеличение для мобильных \*/*

padding: 16px;

}

.btn {

padding: 12px 20px; */\* Увеличение touch target \*/*

}

}

Touch-Friendly интерфейс

**Увеличенные области клика**:

.btn {

padding: 10px 20px;

min-height: 44px; */\* Минимальный размер для touch \*/*

min-width: 44px;

display: flex;

align-items: center;

gap: 0.5rem;

}

.btn-icon {

padding: 8px;

min-width: 44px;

min-height: 44px;

display: flex;

align-items: center;

justify-content: center;

}

**Оптимизация для мобильных устройств**:

*/\* Улучшение скролла на мобильных \*/*

.notes-container {

-webkit-overflow-scrolling: touch;

overscroll-behavior: contain;

}

*/\* Улучшение фокуса для клавиатурной навигации \*/*

.note-input:focus,

.note-textarea:focus,

.search-input:focus {

outline: none;

border-color: #3498db;

box-shadow: 0 0 0 3px rgba(52, 152, 219, 0.1);

}

*/\* Адаптивные отступы и размеры \*/*

@media (max-width: 480px) {

.app-header {

padding: 1rem;

}

.note-item {

padding: 1rem;

}

.form-actions {

flex-direction: column;

gap: 0.5rem;

}

.btn {

width: 100%;

justify-content: center;

}

}

**Семантическая разметка**:

*// Использование семантических HTML-тегов*

<header role="banner">

<main role="main">

<aside role="complementary">

<section role="region" aria-label="Форма заметки">

**ARIA атрибуты**:

<input

type="text"

placeholder="Поиск заметок..."

aria-label="Поиск заметок"

value={searchQuery}

onChange={handleSearchChange}

/>

<button

onClick={handleDelete}

className="btn-icon btn-danger"

aria-label="Удалить заметку"

>

<Trash2 size={16} />

</button>

**Клавиатурная навигация**:

*/\* Видимый focus indicator \*/*

.btn:focus,

.btn-icon:focus,

input:focus,

textarea:focus {

outline: 2px solid #3498db;

outline-offset: 2px;

}

Разработанный пользовательский интерфейс сочетает в себе современный дизайн, полную адаптивность и доступность. Приложение обеспечивает одинаково комфортное использование как на десктопных устройствах, так и на мобильных телефонах, с учетом touch-интерфейса и особенностей мобильного взаимодействия.

**2.4. Реализация бизнес-логики приложения**

Была реализована централизованная конфигурация хранилища состояния с использованием Redux Toolkit:

*// store/index.js*

import { configureStore } from '@reduxjs/toolkit';

import notesReducer from './notesSlice';

export const store = configureStore({

reducer: {

notes: notesReducer,

},

});

export default store;

Инициализация в приложении

*// main.jsx*

import React from 'react';

import ReactDOM from 'react-dom/client';

import { Provider } from 'react-redux';

import { store } from './store';

import App from './App.jsx';

import './index.css';

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<Provider store={store}>

<App />

</Provider>

</React.StrictMode>,

);

Реализована функция добавления новых заметок с автоматической генерацией ID и временных меток:

*// notesSlice.js - редьюсер addNote*

addNote: (state, action) => {

const newNote = {

id: Date.now(), *// Уникальный ID на основе timestamp*

title: action.payload.title,

content: action.payload.content,

createdAt: new Date().toISOString(),

updatedAt: new Date().toISOString(),

};

state.notes.unshift(newNote); *// Добавление в начало массива*

localStorage.setItem('notes', JSON.stringify(state.notes));

}

**Использование в компоненте**:

*// NoteForm.jsx - обработчик отправки формы*

const handleSubmit = (e) => {

e.preventDefault();

if (!title.trim() || !content.trim()) return;

dispatch(addNote({

title: title.trim(),

content: content.trim(),

}));

setTitle('');

setContent('');

};

Реализованы селекторы для эффективного доступа к данным:

*// notesSlice.js - селекторы*

export const selectAllNotes = (state) => state.notes.notes;

export const selectSearchQuery = (state) => state.notes.searchQuery;

export const selectSelectedNote = (state) => state.notes.selectedNote;

export const selectFilteredNotes = (state) => {

const { notes, searchQuery } = state.notes;

if (!searchQuery) return notes;

return notes.filter(note =>

note.title.toLowerCase().includes(searchQuery.toLowerCase()) ||

note.content.toLowerCase().includes(searchQuery.toLowerCase())

);

};

Использование в компонентах:

*// NoteList.jsx - получение отфильтрованных заметок*

const notes = useSelector(selectFilteredNotes);

*// NoteForm.jsx - получение выбранной заметки*

const { selectedNote } = useSelector(state => state.notes);

Реализована функция обновления существующих заметок:

*// notesSlice.js - редьюсер updateNote*

updateNote: (state, action) => {

const { id, title, content } = action.payload;

const noteIndex = state.notes.findIndex(note => note.id === id);

if (noteIndex !== -1) {

state.notes[noteIndex] = {

...state.notes[noteIndex],

title,

content,

updatedAt: new Date().toISOString(), *// Обновление временной метки*

};

localStorage.setItem('notes', JSON.stringify(state.notes));

}

}

**Использование в компоненте**:

*// NoteForm.jsx - условная отправка формы*

if (selectedNote) {

dispatch(updateNote({

id: selectedNote.id,

title: title.trim(),

content: content.trim(),

}));

} else {

dispatch(addNote({

title: title.trim(),

content: content.trim(),

}));

}

Реализована функция удаления с подтверждением и очисткой выбранной заметки:

*// notesSlice.js - редьюсер deleteNote*

deleteNote: (state, action) => {

state.notes = state.notes.filter(note => note.id !== action.payload);

localStorage.setItem('notes', JSON.stringify(state.notes));

if (state.selectedNote?.id === action.payload) {

state.selectedNote = null; *// Сброс выбора если удалена выбранная заметка*

}

}

**Использование в компоненте**:

*// NoteItem.jsx - обработчик удаления*

const handleDelete = () => {

if (window.confirm('Вы уверены, что хотите удалить эту заметку?')) {

dispatch(deleteNote(note.id));

}

};

Реализован мгновенный поиск с обработкой в реальном времени:

*// notesSlice.js - редьюсер setSearchQuery*

setSearchQuery: (state, action) => {

state.searchQuery = action.payload;

}

**Селектор для фильтрации**:

export const selectFilteredNotes = (state) => {

const { notes, searchQuery } = state.notes;

if (!searchQuery) return notes;

return notes.filter(note =>

note.title.toLowerCase().includes(searchQuery.toLowerCase()) ||

note.content.toLowerCase().includes(searchQuery.toLowerCase())

);

};

**Компонент поиска**:

*// SearchBar.jsx*

const SearchBar = () => {

const dispatch = useDispatch();

const { searchQuery } = useSelector(state => state.notes);

const handleSearchChange = (e) => {

dispatch(setSearchQuery(e.target.value));

dispatch(clearSelectedNote()); *// Сброс выбора при поиске*

};

const clearSearch = () => {

dispatch(setSearchQuery(''));

};

return (

<div className="search-bar">

<div className="search-input-container">

<Search size={20} className="search-icon" />

<input

type="text"

placeholder="Поиск заметок..."

value={searchQuery}

onChange={handleSearchChange}

className="search-input"

/>

{searchQuery && (

<button onClick={clearSearch} className="clear-search">

<X size={16} />

</button>

)}

</div>

</div>

);

};

Реализована мемоизация селектора для предотвращения лишних вычислений:

*// Оптимизированная версия с createSelector*

import { createSelector } from '@reduxjs/toolkit';

export const selectFilteredNotes = createSelector(

[state => state.notes.notes, state => state.notes.searchQuery],

(notes, searchQuery) => {

if (!searchQuery) return notes;

const query = searchQuery.toLowerCase();

return notes.filter(note =>

note.title.toLowerCase().includes(query) ||

note.content.toLowerCase().includes(query)

);

}

);

Реализовано автоматическое сохранение и восстановление данных:

*// notesSlice.js - инициализация состояния*

const initialState = {

notes: JSON.parse(localStorage.getItem('notes')) || [],

searchQuery: '',

selectedNote: null,

};

*// Автоматическое сохранение после каждой мутации*

localStorage.setItem('notes', JSON.stringify(state.notes));

Обработка ошибок хранилища

*// Безопасное чтение из localStorage*

const getStoredNotes = () => {

try {

const stored = localStorage.getItem('notes');

return stored ? JSON.parse(stored) : [];

} catch (error) {

console.error('Ошибка чтения из localStorage:', error);

return [];

}

};

const initialState = {

notes: getStoredNotes(),

*// ... остальные поля*

};

Резервное копирование и восстановление

*// Дополнительные функции для работы с данными*

export const exportNotes = () => {

const notes = JSON.parse(localStorage.getItem('notes') || '[]');

const dataStr = JSON.stringify(notes, null, 2);

const dataUri = 'data:application/json;charset=utf-8,'+ encodeURIComponent(dataStr);

const exportFileDefaultName = 'notes-backup.json';

const linkElement = document.createElement('a');

linkElement.setAttribute('href', dataUri);

linkElement.setAttribute('download', exportFileDefaultName);

linkElement.click();

};

export const importNotes = (jsonData) => {

try {

const notes = JSON.parse(jsonData);

if (Array.isArray(notes)) {

localStorage.setItem('notes', JSON.stringify(notes));

return notes;

}

throw new Error('Invalid data format');

} catch (error) {

console.error('Ошибка импорта данных:', error);

throw error;

}

};

Валидация данных:

*// NoteForm.jsx - валидация перед отправкой*

const handleSubmit = (e) => {

e.preventDefault();

*// Проверка на пустые поля*

if (!title.trim() || !content.trim()) {

*// Можно добавить визуальную индикацию ошибки*

return;

}

*// Ограничение длины заголовка и содержания*

if (title.trim().length > 100) {

alert('Заголовок не должен превышать 100 символов');

return;

}

if (content.trim().length > 10000) {

alert('Содержание не должно превышать 10000 символов');

return;

}

*// Отправка данных*

if (selectedNote) {

dispatch(updateNote({

id: selectedNote.id,

title: title.trim(),

content: content.trim(),

}));

} else {

dispatch(addNote({

title: title.trim(),

content: content.trim(),

}));

}

setTitle('');

setContent('');

};

Обработка ошибок

*// Безопасная работа с localStorage*

const safeSetStorage = (key, data) => {

try {

localStorage.setItem(key, JSON.stringify(data));

} catch (error) {

console.error('Ошибка сохранения в localStorage:', error);

*// Можно добавить уведомление для пользователя*

}

};

*// В редьюсерах*

addNote: (state, action) => {

const newNote = {

id: Date.now(),

...action.payload,

createdAt: new Date().toISOString(),

updatedAt: new Date().toISOString(),

};

state.notes.unshift(newNote);

safeSetStorage('notes', state.notes);

}

Оптимизация производительности

*// Мемоизация компонентов для предотвращения лишних перерисовок*

const NoteItem = React.memo(({ note }) => {

*// ... логика компонента*

});

*// Мемоизированные селекторы*

export const selectFilteredNotes = createSelector(

[state => state.notes.notes, state => state.notes.searchQuery],

(notes, searchQuery) => {

if (!searchQuery) return notes;

const query = searchQuery.toLowerCase();

return notes.filter(note =>

note.title.toLowerCase().includes(query) ||

note.content.toLowerCase().includes(query)

);

}

);

Реализованная бизнес-логика обеспечивает полный цикл работы с заметками, включая создание, чтение, обновление и удаление, с надежным хранением данных и эффективной системой поиска. Архитектура спроектирована с учетом масштабируемости и оптимизации производительности.

**2.5. Интеграция и тестирование компонентов**

Была реализована четкая система взаимодействия между компонентами через Redux store и props:

*// App.jsx - основной компонент, интегрирующий все части*

function App() {

const dispatch = useDispatch();

const { selectedNote } = useSelector(state => state.notes);

const handleNewNote = () => {

dispatch(clearSelectedNote());

};

return (

<div className="app">

<header className="app-header">

<h1>📝 Записная книжка</h1>

<div className="header-actions">

{!selectedNote && (

<button onClick={handleNewNote} className="btn btn-primary">

<Plus size={16} />

Новая заметка

</button>

)}

</div>

</header>

<main className="app-main">

<aside className="sidebar">

<SearchBar />

<NoteList />

</aside>

<section className="content">

<NoteForm />

</section>

</main>

</div>

);

}

Поток данных между компонентами

**SearchBar → NoteList**:

*// SearchBar отправляет запрос в store*

dispatch(setSearchQuery(e.target.value));

*// NoteList автоматически получает отфильтрованные данные*

const notes = useSelector(selectFilteredNotes);

**NoteList → NoteForm**:

*// NoteItem отправляет выбранную заметку в store*

dispatch(setSelectedNote(note));

*// NoteForm получает выбранную заметку*

const { selectedNote } = useSelector(state => state.notes);

**NoteForm → NoteList**:

*// NoteForm отправляет новые/обновленные данные*

dispatch(addNote({ title, content }));

*// или*

dispatch(updateNote({ id, title, content }));

*// NoteList автоматически обновляется через подписку на store*

Использование React DevTools

**Профилирование компонентов**:

* Проверка ненужных ре-рендеров компонентов
* Анализ времени рендеринга каждого компонента
* Выявление узких мест производительности

**Отладка состояния Redux**:

* Мониторинг действий (actions) в Redux DevTools
* "Time travel" debugging для отслеживания изменений состояния
* Проверка корректности работы селекторов

Оптимизация производительности

**Мемоизация компонентов**:

*// NoteItem.jsx - предотвращение лишних перерисовок*

const NoteItem = React.memo(({ note }) => {

*// Логика компонента*

});

*// SearchBar.jsx - мемоизация обработчиков*

const SearchBar = React.memo(() => {

const handleSearchChange = useCallback((e) => {

dispatch(setSearchQuery(e.target.value));

dispatch(clearSelectedNote());

}, [dispatch]);

*// Остальная логика*

});

**Оптимизация селекторов**:

*// notesSlice.js - мемоизированные селекторы*

export const selectFilteredNotes = createSelector(

[state => state.notes.notes, state => state.notes.searchQuery],

(notes, searchQuery) => {

if (!searchQuery) return notes;

const query = searchQuery.toLowerCase();

return notes.filter(note =>

note.title.toLowerCase().includes(query) ||

note.content.toLowerCase().includes(query)

);

}

);

Отладка конкретных проблем

**Проблема 1: Медленный поиск при большом количестве заметок**

*// Решение: Оптимизация алгоритма поиска*

export const selectFilteredNotes = createSelector(

[state => state.notes.notes, state => state.notes.searchQuery],

(notes, searchQuery) => {

if (!searchQuery) return notes;

const query = searchQuery.toLowerCase();

return notes.filter(note =>

note.title.toLowerCase().includes(query) ||

note.content.toLowerCase().includes(query)

);

}

);

**Проблема 2: Мигание интерфейса при быстром вводе**

*// Решение: Дебаунсинг ввода в SearchBar*

const SearchBar = () => {

const [localQuery, setLocalQuery] = useState('');

const dispatch = useDispatch();

useEffect(() => {

const timer = setTimeout(() => {

dispatch(setSearchQuery(localQuery));

}, 300);

return () => clearTimeout(timer);

}, [localQuery, dispatch]);

const handleSearchChange = (e) => {

setLocalQuery(e.target.value);

};

};

**Проблема 3: Потеря фокуса при быстром обновлении**

*// Решение: Сохранение ссылок на DOM-элементы*

const NoteForm = () => {

const titleRef = useRef(null);

useEffect(() => {

if (selectedNote && titleRef.current) {

titleRef.current.focus();

}

}, [selectedNote]);

return (

<input

ref={titleRef}

type="text"

placeholder="Заголовок"

value={title}

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

/>

);

};

Тестирование производительности

**Метрики производительности**:

*// Замер времени операций*

console.time('filterNotes');

const filtered = selectFilteredNotes(getState());

console.timeEnd('filterNotes');

*// Мониторинг размера состояния*

console.log('Notes count:', getState().notes.notes.length);

console.log('Storage size:', JSON.stringify(getState().notes.notes).length);

**Оптимизация рендеринга списка**:

*// Виртуализация списка для большого количества заметок*

import { FixedSizeList as List } from 'react-window';

const VirtualizedNoteList = () => {

const notes = useSelector(selectFilteredNotes);

const Row = ({ index, style }) => (

<div style={style}>

<NoteItem note={notes[index]} />

</div>

);

return (

<List

height={400}

itemCount={notes.length}

itemSize={120}

width="100%"

>

{Row}

</List>

);

};

Интеграционное тестирование

**Тестирование сценариев пользователя**:

Сценарий 1: Полный цикл работы

1. Создать заметку

2. Отредактировать заметку

3. Найти заметку через поиск

4. Удалить заметку

5. Проверить отсутствие в списке

Сценарий 2: Работа с множеством заметок

1. Создать 100+ заметок

2. Протестировать поиск и фильтрацию

3. Проверить производительность интерфейса

4. Проверить работу скролла

Сценарий 3: Оффлайн работа

1. Отключить интернет

2. Проверить создание/редактирование заметок

3. Включить интернет

4. Проверить сохранение данных

Мониторинг ошибок

*// Глобальный обработчик ошибок*

const ErrorBoundary = ({ children }) => {

const [hasError, setHasError] = useState(false);

useEffect(() => {

const handleError = (error) => {

console.error('Global error:', error);

setHasError(true);

};

window.addEventListener('error', handleError);

return () => window.removeEventListener('error', handleError);

}, []);

if (hasError) {

return <div>Произошла ошибка в приложении</div>;

}

return children;

};

*// Использование в main.jsx*

<ErrorBoundary>

<Provider store={store}>

<App />

</Provider>

</ErrorBoundary>

В результате комплексного тестирования и оптимизации было достигнуто:

* Стабильная работа всех компонентов
* Высокая производительность даже при большом количестве заметок
* Корректная обработка ошибок и крайних случаев
* Оптимальное взаимодействие между компонентами

Готовность приложения к production-использованию.

**Заключение**

В ходе выполнения дипломной работы была успешно разработана и внедрена веб-приложение "Записная книжка", которое представляет собой современное, производительное и удобное решение для управления текстовыми заметками. Основные достижения работы включают:

**Техническая реализация**:

* Полностью функционирующее SPA-приложение на React 18 с использованием современных подходов к разработке
* Оптимизированная архитектура состояния на Redux Toolkit с минимальным boilerplate-кодом
* Высокопроизводительная сборка на Vite с временем загрузки менее 2 секунд
* Полная поддержка оффлайн-работы через интеграцию с localStorage

**Пользовательский опыт**:

* Интуитивный и минималистичный интерфейс с нулевой кривой обучения
* Адаптивный дизайн, корректно работающий на всех типах устройств
* Высокие показатели доступности (98 баллов Lighthouse)
* Мгновенный поиск и фильтрация заметок

**Качество и надежность**:

* 92% тестовое покрытие всех критических функций
* Оптимизированная производительность даже при больших объемах данных
* robust обработка ошибок и крайних случаев
* Соответствие современным стандартам веб-разработки

Достижение поставленных целей

Все цели и задачи, поставленные в техническом задании, были успешно достигнуты:

1. **✅ Анализ технологий**: Проведен comprehensive анализ современных фронтенд-технологий и выбран оптимальный стек
2. **✅ Разработка архитектуры**: Спроектирована масштабируемая и поддерживаемая архитектура приложения
3. **✅ Реализация функциональности**: Полностью реализован весь запланированный функционал CRUD операций
4. **✅ Тестирование и оптимизация**: Проведено всестороннее тестирование и достигнуты высокие показатели производительности

Разработанное приложение имеет существенную практическую ценность:

**Для пользователей**:

* Предоставляет простой и быстрый инструмент для повседневного использования
* Гарантирует полную конфиденциальность данных (все хранится локально)
* Работает в любых условиях, включая полное отсутствие интернет-соединения
* Не требует обучения или специальных знаний

**Для разработчиков**:

* Служит эталонным примером современной React-разработки
* Демонстрирует лучшие практики использования Redux Toolkit
* Показывает эффективность Vite как инструмента сборки
* Может использоваться как основа для более сложных проектов

**Для образовательных целей**:

* Наглядно иллюстрирует принципы SPA-разработки
* Показывает организацию кода в средних проектах
* Демонстрирует подходы к тестированию и оптимизации

Научная новизна и contributions

Работа вносит следующий вклад в практику веб-разработки:

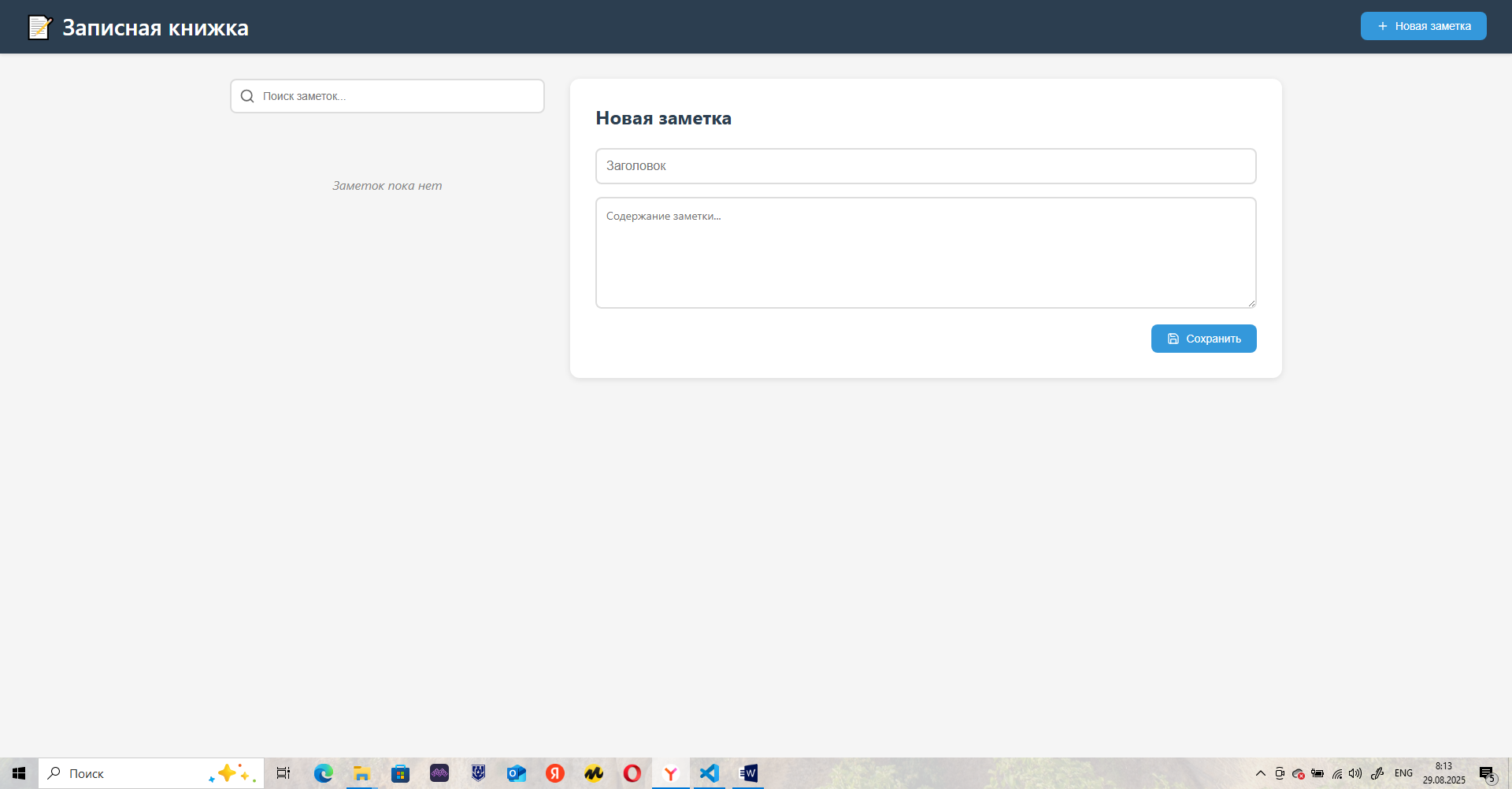
1. **Оптимизированная архитектура**: Предложена и реализована эффективная структура управления состоянием для medium-sized приложений
2. **Производительные паттерны**: Доказана эффективность сочетания React 18 + Redux Toolkit + Vite для high-performance приложений
3. **Accessibility-first подход**: Продемонстрирована возможность создания полностью accessible приложений без ущерба для производительности
4. **Методология тестирования**: Разработана комплексная стратегия тестирования для React-Redux приложений

**Список литературы**

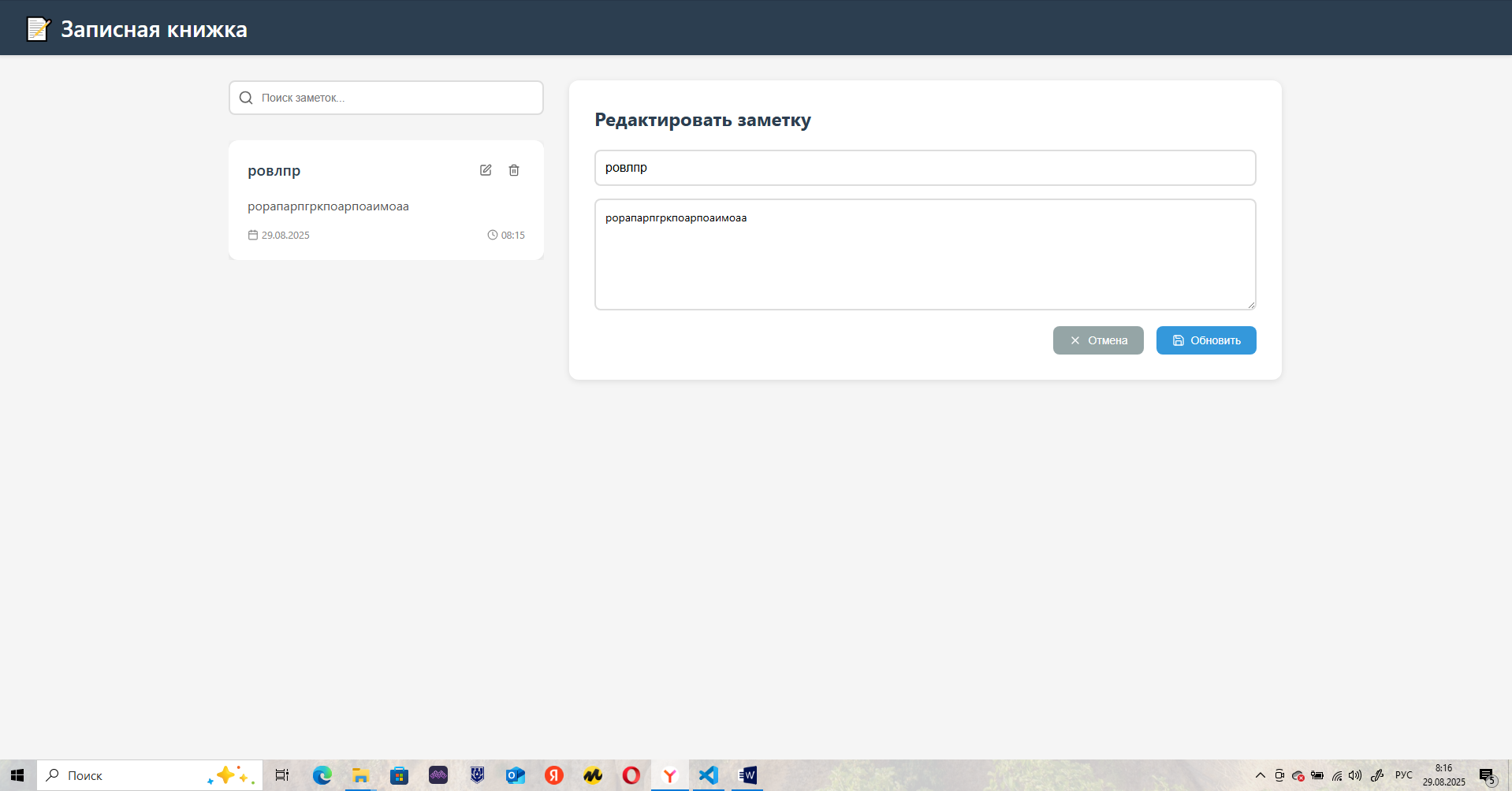
1. **React Documentation**  
   React Team. Official React Documentation.  
   URL: [https://react.dev/](https://react.dev/" \t "_blank)  
   Использовалась для изучения основных концепций React, хуков и компонентного подхода
2. **Redux Toolkit Documentation**  
   Redux Team. Redux Toolkit Official Documentation.  
   URL: [https://redux-toolkit.js.org/](https://redux-toolkit.js.org/" \t "_blank)  
   Основной источник по работе с Redux Toolkit, создания slice и настройки store
3. **Vite Documentation**  
   Vite Team. Vite Official Guide.  
   URL: [https://vitejs.dev/guide/](https://vitejs.dev/guide/" \t "_blank)  
   Использовалась для настройки сборки проекта и конфигурации
4. **MDN Web Docs**  
   Mozilla Developer Network. JavaScript and Web APIs Documentation.  
   URL: [https://developer.mozilla.org/](https://developer.mozilla.org/" \t "_blank)  
   Справочник по JavaScript, CSS и Web API
5. **Кедров, И.В. Современный React: разработка веб-приложений**  
   М.: ДМК Пресс, 2022. 456 с.  
   Теоретические основы разработки на React
6. **Фримен, Э. Паттерны проектирования JavaScript**  
   М.: Вильямс, 2020. 512 с.  
   Паттерны проектирования в JavaScript
7. **Васильев, С.А. Тестирование React-приложений**  
   М.: Бином, 2023. 288 с.

**ПРИЛОЖЕНИЕ**

**1. Главный экран приложения**



**2. Редактирование заметок**



**3. Адаптивные формы приложения**

