БАКАЛАВРСЬКА РОБОТА

БР. ІП - 90.00.00.000 ПЗ

Група ІП-21-3

Юрків Мар`ян

2025

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Юрків Мар`ян Михайлович

(прізвище, ім’я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка public-блогу засобами React
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня Юрків Мар`ян Михайлович
(підпис, ініціали та прізвище здобувача)

Науковий керівник Саманів Любов Василівна, асистент
(підпис, прізвище, ім’я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____202 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Юрків Мар`ян Михайлович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Розробка public-блогу засобами React"

керівник проекту (роботи) к.т.н., асистент Саманів Любов Василівна

затверджені наказом вищого навчального закладу від **“** 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1 Теоретичні основи

2 Бекенд або серверна сторона

3 Фронтенд або клієнтська сторона

4 Впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Спектр фронтенду (рис. 1.1:,ст.10)

2. Зв'язок між createElement та DOM-елементом (рис 1.2, ст. 14).

3. Розбиття інтерфейсу користувача на компоненти. (рис 1.3, ст. 15).

4. Інтерес до React та Angular з плином часу (рис. 1.4, ст. 19)

5. Щотижневі завантаження React проти Angular (рис. 1.5, ст.20).

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	15.02.2025	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	25.02.2025	виконано
3	Побудова моделі або алгоритму власного рішення	15.03.2025	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	25.04.2025	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2025	виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Дипломна робота містить 71 сторінка, 32 рисунки, 2 таблиці, список використаних джерел із 30 найменуваннями.

Метою роботи є розробка public-блогу засобами React для забезпечення ефективного створення та управління контентом із використанням сучасних веб-технологій.

Об'єкт дослідження: процеси розробки та функціонування веб-додатку для створення public-блогу.

Предмет дослідження: методи, технології та інструменти розробки веб-додатку для блогу, включаючи аналіз фронтенд- і бекенд-технологій, проектування архітектури, інтеграцію з Notion API, створення адаптивного інтерфейсу, управління станом і тестування системи.

Результати дослідження: створено веб-додаток для public-блогу з використанням React, Node.js, Express і Notion API, який забезпечує адаптивність, швидке завантаження та зручне управління контентом.

У першому розділі аналізуються фронтенд-екосистема, принципи React, декларативне та імперативне програмування, також порівняння React і Angular.

У другому розділі формулюються завдання бекенду, досліджуються типи рендерингу та використання Node.js і Express.

У третьому розділі описано проектування фронтенду, включаючи адаптивний дизайн і оптимізацію продуктивності.

У четвертому розділі розглянуто практичну реалізацію блогу, зокрема інтеграцію з Notion API, управління станом через React-хуки та навігацію.

Висновок: розробка веб-додатку успішно реалізована, відповідаючи сучасним вимогам до блог-платформ і забезпечуючи високу якість користувацького досвіду.

КЛЮЧОВІ СЛОВА: PUBLIC-БЛОГ, REACT, NODE.JS, EXPRESS, NOTION API, АДАПТИВНИЙ ДИЗАЙН, REACT-ХУКИ, ФРОНТЕНД, БЕКЕНД, ВЕБ-РОЗРОБКА.

ABSTRACT

The bachelor’s thesis comprises 65 pages, 32 figures, 2 tables, and a reference list with 30 entries.

The aim of the work is to develop a public blog using React, focused on providing convenient content creation, management, and display through modern web technologies.

Object of study: the processes of developing and operating a web application for a public blog, aimed at ensuring effective interaction between authors and readers.

Subject of study: methods, technologies, and tools for developing a blog web application, including analysis of the frontend ecosystem, design of server-side and client-side logic, integration with Notion API, as well as state management, testing, and system efficiency evaluation.

Research results: a web application for a public blog was developed using React, Node.js, Express, and Notion API, ensuring adaptive design and fast content loading.

The first section analyzes the frontend ecosystem, React principles, declarative and imperative programming, and a comparison of React with Angular.

The second section formulates backend tasks, explores rendering types, and the use of Node.js and Express.

The third section describes frontend development, including adaptive design and performance optimization.

The fourth section details the practical implementation of the blog with Notion API integration, state management, and navigation via React hooks.

Conclusion: the web application development was successfully implemented, meeting modern web development standards and providing a high-quality user experience.

KEYWORDS: PUBLIC BLOG, REACT, NODE.JS, EXPRESS, NOTION API, FRONTEND, BACKEND, ADAPTIVE DESIGN, REACT HOOKS, TESTING.

ЗМІСТ

					БР.ІП –90.00.00.000 ПЗ			
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розроб.		Юрків М.М.			Розробка public-блогу засобами React **Пояснювальна записка**	*Літ.*	*Арк.*	*Акрушів*
Перевір.		Саманів Л.В.					*8*	
Реценз.		Гобир Л.М.				**ІФНТУНГ ІП-21-3**		
Н. Контр.		Піх М.М.						
Затверд.		Бандура В. В.						

ВСТУП

У сучасному цифровому світі блоги відіграють ключову роль у поширенні інформації, просуванні ідей та взаємодії з аудиторією, що зумовлює зростання попиту на створення зручних, продуктивних і адаптивних веб-додатків для управління контентом. React, як одна з найпопулярніших бібліотек для створення користувацьких інтерфейсів, забезпечує розробку динамічних і масштабованих веб-додатків, що робить її оптимальним вибором для реалізації public-блогів. Інтеграція React із сучасними бекенд-технологіями, такими як Node.js і Express, а також використання API, наприклад Notion API, відкриває нові можливості для автоматизації управління контентом, оптимізації продуктивності та забезпечення якісного користувацького досвіду.

Актуальність теми обумовлена швидким розвитком веб-технологій і потребою в персоналізованих блог-платформах, які відповідають сучасним стандартам швидкості, адаптивності та гнучкості. Існуючі рішення, такі як WordPress, Medium чи Ghost, часто мають обмеження щодо кастомізації, швидкості завантаження сторінок або інтеграції з передовими інструментами, що підкреслює необхідність створення нових платформ на основі React.

Метою роботи є розробка public-блогу засобами React, який забезпечить ефективне створення, управління та відображення контенту з акцентом на продуктивність і зручність використання**.**

Завданнями дослідження є вивчення фронтенд-екосистеми та принципів React, аналіз бекенд-технологій і типів рендерингу, розробка адаптивного фронтенду з оптимізацією продуктивності, а також реалізація блогу з інтеграцією бази знань через Notion API, управлінням станом за допомогою React-хуків і тестуванням системи.

Об’єктом дослідження є процеси розробки програмного забезпечення для створення public-блогу, що охоплюють аналіз вимог, проектування, реалізацію, інтеграцію та тестування системи. Предметом дослідження є методи, технології та інструменти, що застосовуються під час розробки блогу,

зокрема React для клієнтської сторони, Node.js і Express для серверної логіки, Notion API для управління контентом, а також методи адаптивного дизайну, оптимізації продуктивності й тестування. Для досягнення мети використано методи аналізу літературних джерел, порівняльного аналізу технологій, моделювання архітектури, емпіричного програмування, модульного та інтеграційного тестування, а також оцінки продуктивності й користувацького досвіду. Розроблений блог передбачає створення та управління постами, адаптивне відображення контенту, швидке завантаження сторінок завдяки серверному рендерингу та оптимізації, а також інтеграцію з Notion API для автоматизації імпорту контенту, що спрощує роботу авторів і покращує взаємодію з читачами. Очікується, що впровадження блогу сприятиме підвищенню доступності контенту, полегшенню управління інформацією та встановленню нових стандартів для блог-платформ.

Бакалаврська робота містить 71 сторінок, 32 рисунки, 2 таблиці, список використаних джерел із 30 найменуваннями.

РОЗДІЛ 1. СУЧАСНА ФРОНТЕНД-РОЗРОБКА: ІНСТРУМЕНТИ, ПІДХОДИ ТА ТЕХНОЛОГІЇ

1.1 Екосистема фронтенд-розробки: основи та інструмен

Фронтенд-розробка зосереджена на клієнтському сайті, тобто на тому, що відбувається у браузері. Основними технологіями для фронтенд-розробки є HTML, CSS та JavaScript. Однак в останні роки фронтенд стрімко розвивається, і винаходиться все більше інструментів для обробки різних частин екосистеми.

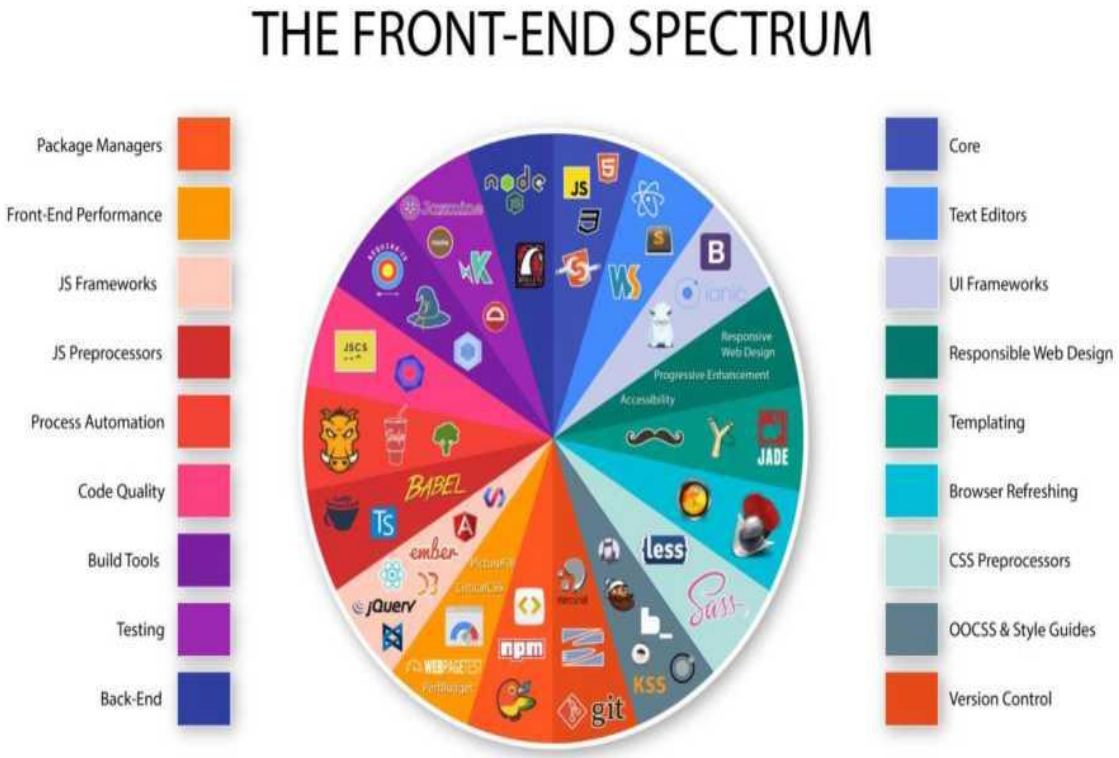

Рисунок 1.1 - Спектр фронтенду

Як показано на рисунку 1.1, екосистема фронтенду поділена на різні сектори, і кожен сектор виконує певне завдання. Примітно, що в шаблонах для HTML є handlebars та Jade. Для обробки стилів, окрім основного CSS, існують препроцесори, такі як Less та Sass. Відповідними препроцесорами для JavaScript є CoffeeScript, TypeScript та Babel. Одним з найбільш конкурентних та

розвиваючихся секторів є бібліотеки та фреймворки JavaScript з такими помітними представниками, як React, jQuery, Angular, Vue, Ember, D3.

Окрім основних HTML, CSS та JavaScript, екосистема пропонує багато інструментів для інших аспектів. Npm та Yarn є найпомітнішими інструментами для керування пакетами. Так само, для якості коду є ESLint, JSCS та Prettier. Інструмент збірки є важливою частиною сучасних веб-технологій. Найпопулярнішими інструментами збірки є Webpack, Browserify та Parcel.

1.2 Декларативне та імперативне програмування

Ів Порчелло та [5] стверджували, що імперативне програмування отримує кінцеві результати, надаючи інструкції застосунку крок за кроком за допомогою коду. З іншого боку, декларативне програмування структурує застосунок таким чином, що він оголошує логіку, а не визначає, як вона відбувається. Візьмемо, наприклад, просте завдання зробити рядок зручним для URL. Наступний код описує, як отримати результат за допомогою імперативного програмування.

```
1.  var string = "This is the midday show with Cheryl Waters";
2.  var urlFriendly = "";
3.
4.  for (var i=0; i<string.length; i++) {
5.    if (string[i] === " ") {
6.      urlFriendly += "-";
7.    } else {
8.      urlFriendly += string[i];
9.    }
10. }
11.
12. console.log(urlFriendly);
```

Лістинг 1.1 - Маніпулювання рядками за допомогою імперативного підходу

В імперативному підході кожен символ рядка перебирається в циклі, і якщо він є пробілом, він замінюється дефісом. Цикл for та оператори if використовуються для керівництва програмою для виконання завдання. Щоб зрозуміти, що робить код, в імперативному програмуванні потрібно багато коментарів. Декларативний підхід набагато простіший для виконання:

```
1.  const string = "This is the mid day show with Cheryl Waters"
2.  const urlFriendly = string.replace(/ /g, "-")
3.
4.  console.log(urlFriendly)
```

Лістинг 1.2 - Маніпулювання рядками за допомогою декларативного підходу

Вивчення коду чітко показує читачеві, якого результату хоче досягти програма. Програма описує, що рядок має замінити пробіл на дефіс, не знаючи, як це робить функція replace. Код простий та зрозумілий завдяки декларативному програмуванню. Логіка абстрагована до багатьох крихітних функцій з гарними назвами. Оскільки код не потребує пояснень та його легко слідувати, розробляти великі застосунки легше. [7].

```
1.  // imperative jQuery
2.  var button = $('<button class="btn red">Change color</button>')
3.  $(button).on('click', function () {
4.    if ($(this).hasClass('red')) {
5.      $(this).removeClass('red')
6.      $(this).addClass('blue')
7.    } else {
8.      $(this).removeClass('blue')
9.      $(this).addClass('red')
10.   }
11. })
12. // declarative React
13. const Button = () => {
14.   const [color, setColor] = React.useState('red')
15.   const handleClick = () => {
16.     setColor(prevColor => prevColor === 'red' ? 'blue' : 'red')
17.   }
18.   return <button className={`btn ${color}`} onClick={handleClick}>Change color
    </button>
19. }
```

Лістинг 1.3 - Імперативне та декларативне програмування з використанням веб-технологій

У лістингу 3 описано відмінності між імперативним та декларативним підходами в jQuery та React відповідно при створенні кнопки, яка може перемикатися між червоним та синім кольорами. У підході jQuery програмі крок за кроком, від створення кнопки, надається інструкція для обробки обробника

кліку. Натомість, React оголошує компонент кнопки, в якому визначено стан кольору та обробник кліку для зміни стану. Програма React не стосується того, як кнопка візуалізується або як вона перемикає кольори. Фрагмент коду React не тільки коротший та легший для розуміння, але й версію кнопки React можна використовувати в кількох місцях програми без повторного оголошення.

Декларативне програмування стає стандартом сучасних веб-технологій, приклади з яких можна знайти в React та Angular. Імперативне програмування, таке як використання jQuery, ускладнює розуміння та підтримку застосунків. Завдяки декларативному підходу в сучасних веб-технологіях розробники можуть створювати зручні та масштабовані застосунки.

1.3 Аналіз архітектурних підходів у React та Angular

React — це фронтенд-бібліотека, випущена та підтримувана Facebook у 2013 році. React зосереджується на рівні перегляду застосунку шляхом створення та керування компонентами. З моменту свого випуску React закріпив міцні позиції у фронтенд-спільноті. Більше того, React використовується не лише для розробки веб-застосунків, React також орієнтований на інші платформи, такі як нативні мобільні пристрої та віртуальна реальність [3].

Об'єктна модель документа або DOM, яка побудована за допомогою HTML або XML, є представленням веб-сторінки, за допомогою якої програми або скрипти можуть маніпулювати структурою, стилем або вмістом сторінки (Mozilla, 2019). Для зміни DOM JavaScript використовує DOM API, наприклад document.createElement, document.appendChild, і цей процес є відносно простим [4] Однак ефективне керування сторінкою за допомогою DOM API та JavaScript є дуже складним, особливо під час створення великого застосунку. React - одна з бібліотек, яка надає рішення за допомогою віртуального DOM. Згідно з документацією React (2019), віртуальний DOM (VDOM) являє собою інтерфейс користувача (UI), який зберігається в пам'яті. Після зміни стану інтерфейсу користувача, VDOM синхронізується з реальним DOM через процес узгодження.

Розробники можуть працювати з декларативним API React замість DOM API, а React бере на себе маніпуляції з DOM та обробку подій.

Елементи DOM використовуються для побудови DOM браузера, так само як елементи React використовуються для створення React VDOM. Елементи React відрізняються від елементів DOM браузера, оскільки вони є незмінними простими об'єктами, а оновлення об'єктів набагато швидше, ніж безпосереднє використання DOM за допомогою DOM API (Porcello & Banks. 2017V React.ci

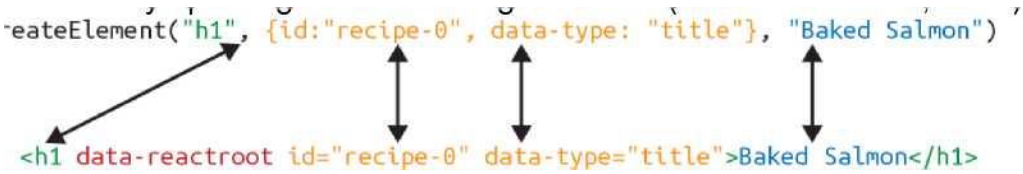

Рисунок 1.2 - Зв'язок між createElement та DOM-елементом.

Як показано на рисунку 1.2, властивості елемента React додаються до тегу DOM як атрибути, а дочірній текст додається як текст вузла елемента DOM.

```
// without JSX
React.createElement("ul", {"className": "list"},
  React.createElement("li", {"className": "list-item"},"Item text"),
  React.createElement("li", {"className": "list-item"},"Item text"),
  React.createElement("li", {"className": "list-item"},"Item text"),
  React.createElement("li", {"className": "list-item"},"Item text"),
  React.createElement("li", {"className": "list-item"},"Item text"),
  React.createElement("li", {"className": "list-item"},"Item text")
);

// JSX
<ul className="list">
  <li className="list-item">Item Text</li>
  <li className="list-item">Item Text</li>
  <li className="list-item">Item Text</li>
  <li className="list-item">Item Text</li>
  <li className="list-item">Item Text</li>
  <li className="list-item">Item Text</li>
</ul>
```

Лістинг 1.4 - Реагувати з JSX та без нього

У специфікації JSX від Facebook (2014) зазначено, що «JSX — це XML-подібне розширення синтаксису ECMAScript без будь-якої визначеної семантики». JSX — це альтернативний та рекомендований спосіб написання

елементів React. Код, написаний на JSX, перетворюється на звичайний JavaScript компілятором, таким як Babel. Мета JSX — надати простіший синтаксис для побудови складного дерева DOM та зробити його більш читабельним, подібним до HTML [17].

У лістингу 1.4 показано різницю між написанням React з використанням JSX та без нього. JSX робить код чистішим та зрозумілішим для читання; отже, це спрощує роботу з обслуговування. Розробник може писати складні вкладені структури DOM без зручності для читання, оскільки JSX допомагає складати код подібно до HTML.

Компонент - це частина інтерфейсу користувача застосунку, велика чи мала, наприклад, панель навігації, заголовок або розділ. Компоненти складаються з елементів. Розбиття інтерфейсу користувача на менші компоненти допомагає швидше розробляти, легше підтримувати та краще використовувати код повторно (Facebook, 2019).

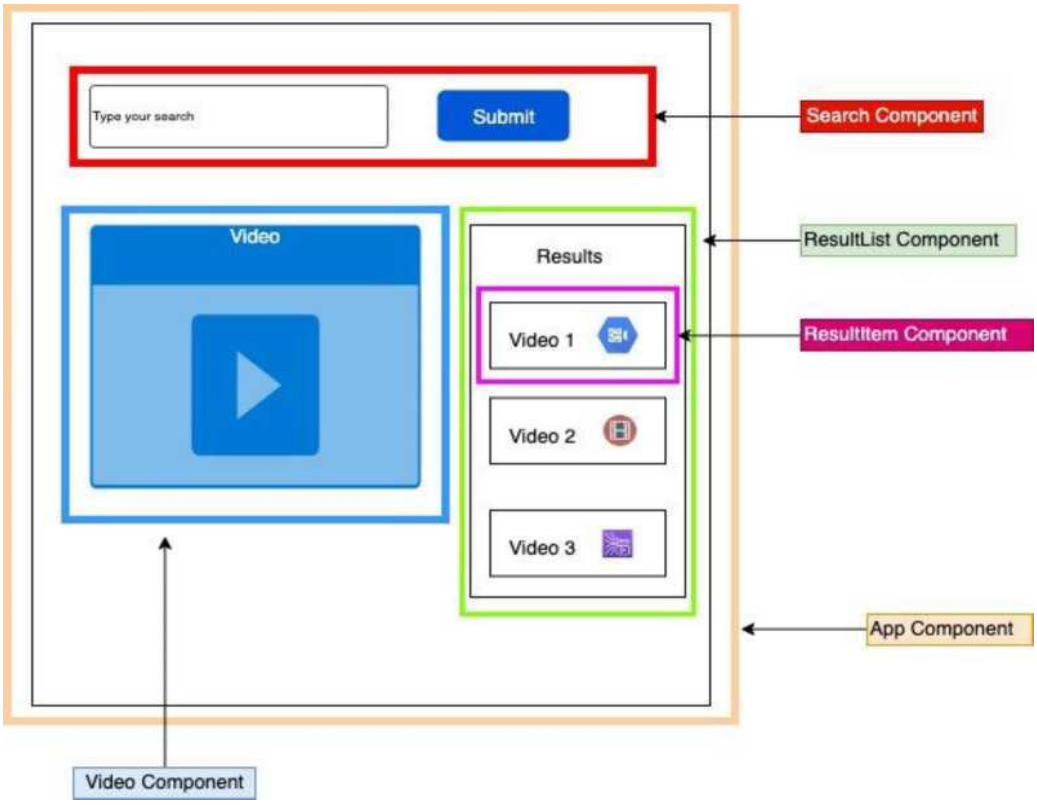

Рисунок 1.3 - Розбиття інтерфейсу користувача на компоненти

На рисунку 1.3 зображено інтерфейс користувача простого застосунку, який приймає введені користувачем дані та відображає результати пошуку відео. Інтерфейс користувача поділено на чотири великі компоненти:

- Компонент програми – це ціла програма.
- Компонент пошуку приймає пошуковий запит користувача.
- Компонент ResultList відображає список результатів пошуку.
- Відеокомпонент містить відеоплеєр.

Крім того, деякі компоненти можна розділити на ще менші компоненти для повторного використання. Наприклад, компонент пошуку складається з поля для введення тексту та кнопки, які є компонентом і можуть бути використані повторно в інших місцях програми. Існує два типи компонентів: функціональні та класові.

У React 16.8 були представлені хуки, які надають стан та інші функції функціональним компонентам. Абрамов (2018) стверджував, що створення компонентів за допомогою хуків є простим та зручнішим для користувача. React надає різні вбудовані хуки, включаючи useState та useEffect, які допомагають оголошувати стан та побічні ефекти для функціональних компонентів. Усі варіанти використання для компонентів класу можуть бути охоплені хуками, але хуки також забезпечують повторне використання коду, вилучення та тестування з більшою гнучкістю. Таким чином, хуки слугують майбутнім баченням React. Проект випадку в цій дисертації був складений з використанням хуків для обробки стану замість використання компонентів класу. Загалом, існує два суворих правила використання хуків:

- Викликайте хуки лише на верхньому рівні. Не викликайте хуки всередині циклів, умов або вкладених функцій.
- Викликайте хуки лише з функцій React. Не викликайте хуки зі звичайних функцій JavaScript.

React сам по собі є бібліотекою керування станом, а контекст — один із методів для цього. React має низхідний механізм передачі даних від батьківського до дочірнього компонента за допомогою props, що є простим, але

цей механізм також створює проблеми, такі як деталізація prop. Проблема деталізації prop виникає, коли розробники повинні передавати або деталізувати prop на кількох рівнях компонентів, що створює труднощі з рефакторингом. Контекст React допомагає легко керувати станом та обмінюватися даними без явного передавання prop через кожен рівень дерева компонентів [25].

React Router

Історично маршрутизація веб-сайтів здебільшого реалізується на сервері, де зберігається низка файлів, що представляють веб-сторінки. Коли користувач переходить на веб-сторінку, сервер повертає файл у цьому місці, що відображається в адресному рядку браузера. Сучасні веб-технології пропонують односторінкові застосунки, в яких JavaScript завантажує дані та змінює інтерфейс користувача без необхідності маршрутизації на стороні сервера. В екосистемі React, React Router є найвідомішим рішенням для маршрутизації на стороні клієнта для React-застосунків [11].

Angular — це проєкт з відкритим кодом та фреймворк, випущений Google у 2016 році для створення односторінкових додатків (SPA). Фреймворк Angular повністю переписаний ¬з AngularJS або Angular 1 тією ж командою з Google. Окрім веб-додатків, за допомогою Angular можна розробляти додатки для різних платформ, таких як мобільні та настільні. [15].

Angular має пов'язаний продукт під назвою Angular CLI. Angular CLI — це інструмент командного ¬рядка , розроблений командою Angular, який допомагає розробникам позбутися навантаження під час налаштування нового проекту Angular. Проект, згенерований за допомогою Angular CLI, має необхідні структурні блоки, такі як сервер розробки, засіб виконання тестів, Webpack, лінтинг та кроки збірки.

Компонент

Компонент визначає частину інтерфейсу користувача на екрані програми, таку як панель інструментів з посиланнями навігації, список елементів, поле введення. Компонент Angular — це клас JavaScript або TypeScript, декорований декоратором @Component, який визначає метадані для компонента. Метадані

можуть містити шаблон для визначення представлення, селектор для виклику компонента в шаблоні HTML, URL-адреси стилів для застосування стилів до представлення HTML та постачальників для надання послуг.

```
1. @Component({
2.   selector:    'app-hero-list',
3.   templateUrl: './hero-list.component.html',
4.   providers:  [ HeroService ]
5. })
6. export class HeroListComponent implements OnInit {
7. /* . . . */
8. }
```

Лістинг 1.5 - Приклад Angular компонента

Як показано в лістингу 1.5, декоратор @Component розміщується перед визначенням класу HeroListComponent, щоб надати компоненту необхідні метадані, такі як селектор, templateUrl та провайдери. Тільки класи, декоровані декоратором @Component, як показано вище, стають компонентом Angular.

В ідеальному випадку компонент повинен обробляти лише користувацький досвід і виступати посередником між представленням і логікою програми, надаючи властивості та методи для прив'язки даних. Крім того, компонент може використовувати сервіси для обробки таких завдань, як вибірка даних, перевірка вводу або ведення журналу консолі. Ці сервіси можна впроваджувати через впровадження залежностей, що надається Angular, тому вони доступні для всієї програми.

Ін'єкція залежностей – це шаблон проектування програмування, який дозволяє класу використовувати сервіси поза своєю областю видимості шляхом ін'єкції їх як залежностей. Ін'єкція залежностей підвищує гнучкість, ефективність, тестованість, зручність обслуговування та модульність Angular-застосунку [6].

1.4 Порівняння React та Angular

React та Angular – дві найпотужніші технології для розробки фронтенду. Angular – це фреймворк, який надає повноцінний інструментарій для створення застосунку. Пакети для http-запитів, маршрутизації, шаблонів тощо доступні в основному фреймворку Angular. Однією з найбільших переваг Angular є узгодженість у різних проектах, оскільки всі інструменти вже створені для використання розробниками, а Angular визначає, як розробники структурують проект. З іншого боку, важко включити нову бібліотеку в застосунок Angular, оскільки бібліотека повинна бути сумісною з TypeScript та обгорнутою в сервіс, щоб взаємодіяти з компонентами. Крім того, Angular має упереджений підхід, що означає, що існує спосіб реалізації на Angular.

На відміну від Angular, React — це бібліотека, яка зосереджена на рівні перегляду застосунку. Через свою зосередженість на побудові інтерфейсу користувача, React потребує інших бібліотек для розробки повноцінного застосунку. Через це імпорт та використання сторонньої бібліотеки в React надзвичайно просте за допомогою npm. React має велику екосистему, яка надає численні рішення для різних рівнів застосунку, таких як маршрутизація, вибірка даних, управління станом. Таким чином, React пропонує розробникам гнучкість у структуруванні та розробці застосунків.

React та Angular – це великі проекти, які користуються великою популярністю серед спільноти розробників з відкритим кодом. Існує багато факторів для порівняння популярності, таких як тренди, використання, зростання спільноти тощо.

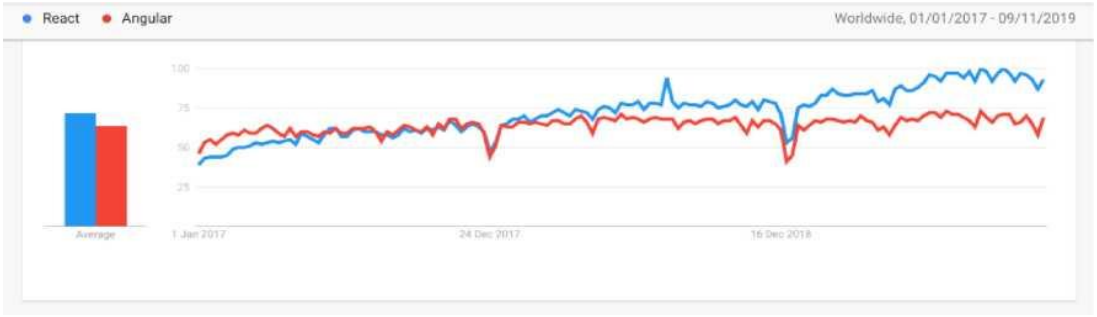

Рисунок 1.4 - Інтерес до React та Angular з плином часу з січня 2017 року по листопад 2019 року (Google Trends, 2019)

Як показано на рисунку 1.4, який показує тенденції React та Angular у пошуковій системі Google протягом останніх трьох років, на початку 2017 року Angular лідирував; проте з середини 2017 року React послідовно випереджає Angular. Більше того, React поступово збільшує розрив з часом. Тенденції в Google частково доводять, що з часом все більше людей цікавляться React.

Рисунок 1.5 - Щотижневі завантаження React проти Angular з травня 2017 року по листопад 2019 року

Згідно з npmtrends.com, кількість завантажень React та Angular значно відрізняється. Протягом останніх двох років Angular завантажували близько мільйона разів на тиждень, і ця цифра постійно подвоювалася з 750 000 у травні 2017 року до 1350 000 у листопаді 2019 року. З іншого боку, з травня 2017 року React вже мав понад два мільйони завантажень на тиждень. За два роки ця цифра потроїлася до понад шести мільйонів на кінець 2019 року. Щотижневі завантаження як Angular, так і React значно зростають за два роки; проте React продемонстрував більшу траєкторію зростання.

Веб-опитування – це потужний інструмент для вимірювання ставлення розробників до інструментів і технологій. Щорічне опитування розробників Stack Overflow – одне з найкращих опитувань щодо технологій загалом.

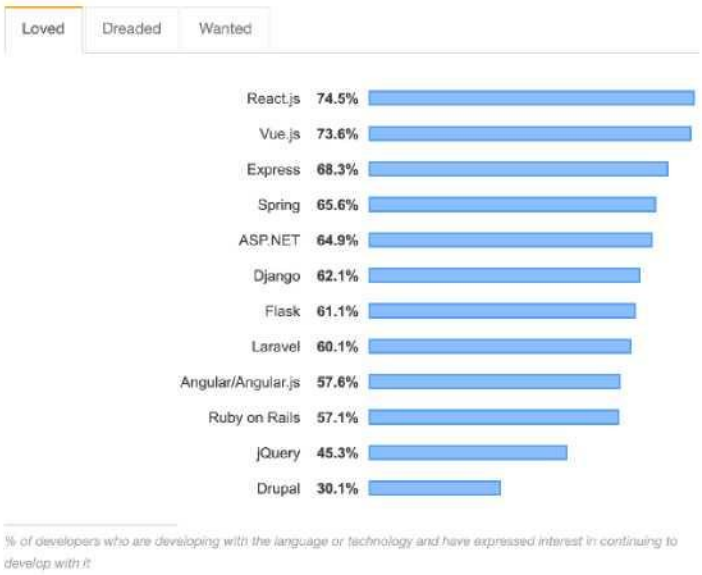

Рисунок 1.6 - Найпопулярніші технології (Stack Overflow, 2019)

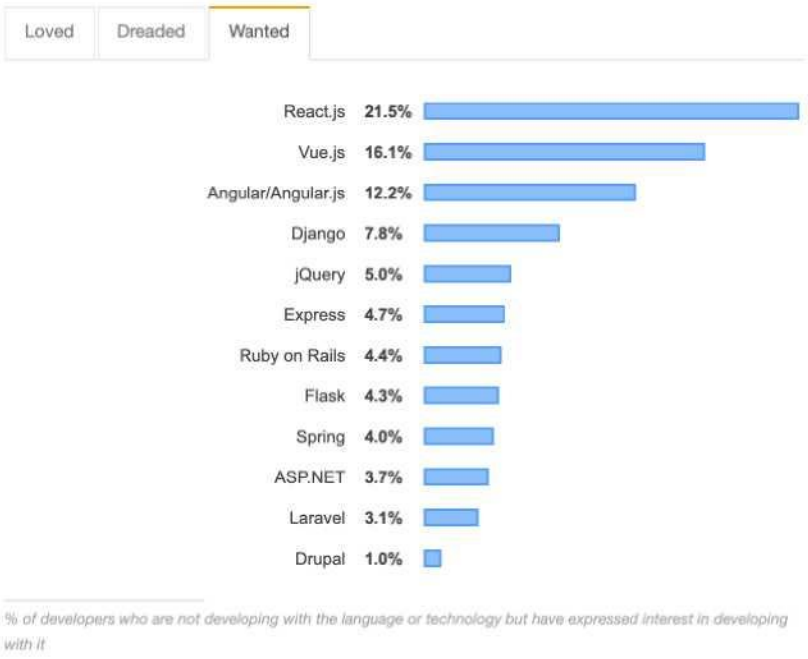

Рисунок 1.7 - Найбільш затребувані технології (Stack Overflow, 2019)

Як показано на рисунках 1.6 та 1.7, спостерігається позитивна тенденція щодо React, оскільки бібліотека очолює таблиці найулюбленіших та найбажаніших технологій. Троє з чотирьох розробників висловлюють величезний інтерес до React, оскільки хочуть продовжувати розробку з використанням бібліотеки. Натомість, лише понад половина розробників мають таку ж думку щодо Angular.

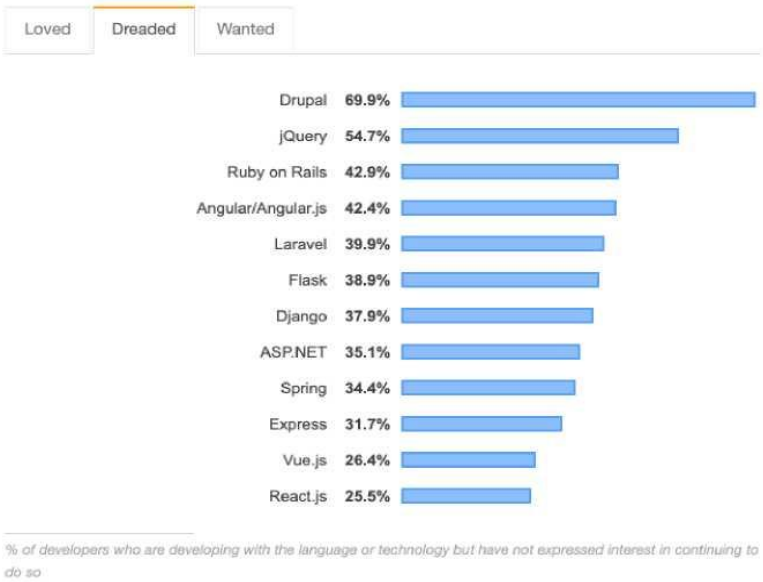

Рисунок 1.8 - Найбільш страшні технології [9]

В іншому опитуванні щодо найстрашніших технологій Angular посідає четверте місце в таблиці, де 42,4% розробників вважають, що не продовжуватимуть використовувати Angular для своєї майбутньої розробки. Для React цей показник становить 25,5%, що значно нижче. Опитування Stack Overflow показують, що розробники краще ставляться до React, ніж до Angular. Однак , Angular все ще залишається однією з найбільш затребуваних технологій поряд з React та Vue у 2019 році.

1.5 Висновки до розділу

Як React, так і Angular – надзвичайно відомі та широко використовувані технології для створення веб-додатків. Однак React лідирує з точки зору зростання спільноти та позитивного ставлення з боку розробників. Kodit, як стартап, по-перше, хоче швидко розробляти та випускати продукти. По-друге, Kodit потрібно залучати ширше коло фронтенд-розробників, щоб швидко розвиватися. Завдяки легшій кривій навчання та більшій спільноті, React є кращим рішенням для Kodit, ніж Angular. Тому в Kodit технічна команда вирішила використовувати React для створення внутрішніх інструментів, а також веб-сайту компанії.

РОЗДІЛ 2. БЕКЕНД АБО СЕРВЕРНА СТОРОНА

2.1 Обов'язки бек-енду

Веб- або мобільний застосунок працює завдяки зв'язку між фронтенд-частиною застосунку (зазвичай вона запускається на основі дій користувача, клієнта) та серверною частиною застосунку, що дозволяє йому поводитися бажаним чином на основі послідовності дій. Бекенд також зберігає та обробляє дані, що використовуються в рамках застосунку.

Коли користувач взаємодіє з додатком, запит у форматі HTTP надсилається до серверної частини: вона потім обробляє цей HTTP-запит і надсилає відповідь клієнту. Обробка запиту включає взаємодію із серверами баз даних (керування отриманням даних та їх модифікацією), мікросервісами (виконання підмножини завдань, запитуваних користувачем) та сторонніми інтерфейсами прикладного програмування (зазвичай їх називають «API», які збирають додаткову інформацію або виконують додаткові завдання) (AWS, 2024).

Добре написаний та структурований бекенд має вирішальне значення для забезпечення безперебійної роботи застосунку, оскільки він є його ядром. Ось деякі з його основних функцій, згідно з Гранджеоном (2023):

Керування базами даних : зберігання та отримання даних з бази даних. Потреби залежать від використання програми, але зазвичай включають створення, зміну та видалення даних.

Керування користувачами та автентифікацією : важливо мати захищений застосунок, а автентифікація користувачів є одним з основних аспектів, яким керує серверна частина. Це також охоплює сеанси користувачів, зашифровані паролі та інші аспекти, пов'язані з безпекою.

Бізнес-логіка: бекенд відповідає за визначення часу проведення операцій, таких як обчислення та транзакції, а також за відповідні бізнес-правила, визначення завдань та способів їх виконання для виконання цих операцій.

Обробка запитів : бекенд найчастіше тісно пов'язаний з розробкою фронтенду , оскільки бекенд отримує та обробляє запити від фронтенду. Зв'язок між цими двома сторонами програми базується на здатності бекенду визначати, як відповідати на запити, пов'язані з фронтендом, надсилаючи йому необхідні дані для відображення.

Управління продуктивністю та масштабованістю : висока продуктивність також є одним із обов'язки серверної частини. Його необхідно оптимізувати, щоб гарантувати зручність використання програми, навіть коли багато користувачів одночасно отримують до неї доступ. Висока продуктивність досягається за допомогою стратегій кешування, оптимізації бази даних та управління ресурсами сервера.

Бекенд, безсумнівно, є дуже важливою частиною для належного функціонування застосунків будь-якого масштабу.

2.2 Типи рендерингу

Рендеринг – це дія перетворення фрагмента коду в те, що користувач може бачити під час переходу до веб-застосунку. Якщо існує кілька типів рендерингу, є два основних, які виділяються при пошуку правильного методу рендерингу. Вони називаються серверним рендерингом (SSR) та клієнтським рендерингом (CSR): обидва пропонують різні переваги, тому вибір одного з них залежить від мети застосунку та цілей, які компанія хоче досягти.

Хоча рендеринг часто асоціюється з фронтенд-розробкою, вибір правильного типу рендерингу має значний вплив на бекенд-частину застосунку.

Рендеринг на стороні сервера (SSR)

Як випливає з назви, серверний рендеринг – це тип рендерингу, в якому процес рендерингу відбувається на стороні сервера. Сервер генерує вичерпну HTML-сторінку та надсилає її до браузера клієнта. Потім браузер відображає відрендерену сторінку повністю, без виконання будь-якого JavaScript-коду, що походить з боку клієнта.

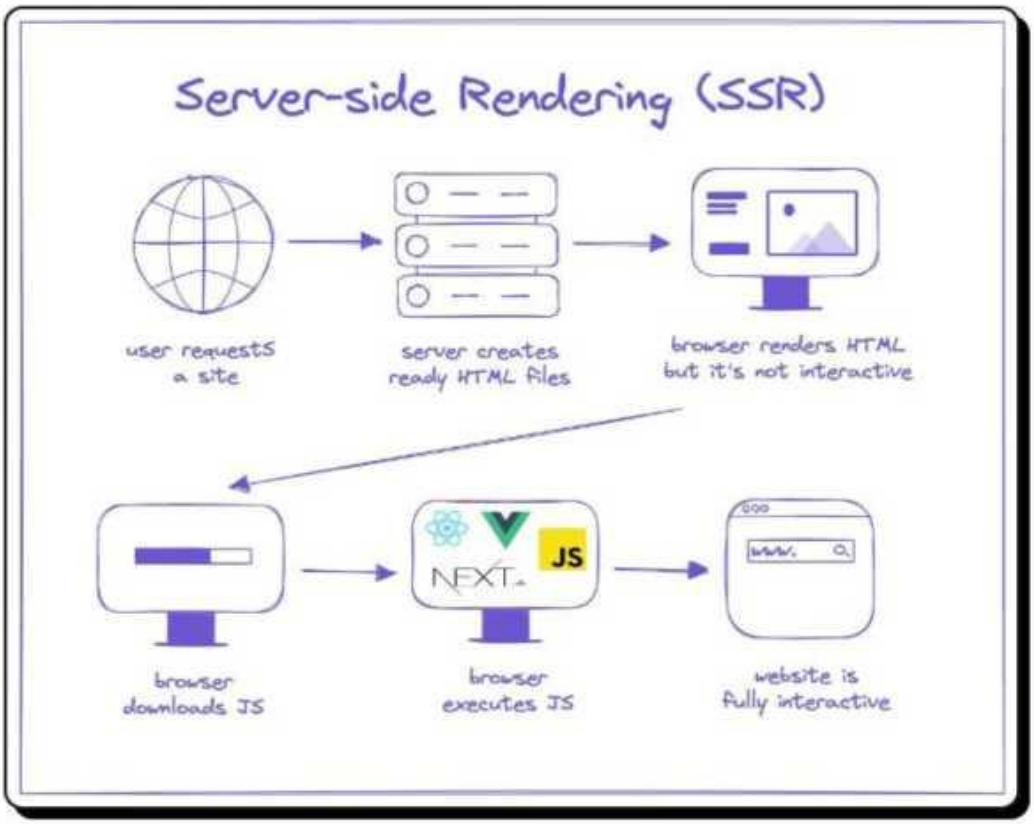

Рисунок 2.1 - Ескіз логіки рендерингу на стороні сервера.

На рисунку 2.1 детально описано, як реалізується рендеринг на стороні сервера: сервер отримує запит на веб-сторінку. Потім він отримує дані для цієї конкретної сторінки та заповнює їх у шаблон HTML. Згодом сервер генерує розмітку HTML, відображає її вміст та застосовує різні стилі. Після цього кроку сервер повністю надсилає сторінку до браузера для перегляду користувачем та взаємодії з нею.

SSR – це гарний варіант для веб-застосунку, який потребує оптимізації для пошукових систем (SEO), і забезпечує швидше завантаження веб-сайтів. SSR краще розглядати для статичних веб-сайтів, оскільки інтерактивність веб-сайту знижується (через рендеринг, що відбувається на сервері, що робить будь-які зміни в інтерфейсі користувача залежними від затримок). Серверний рендеринг також споживає більше серверних ресурсів: збільшується використання пам'яті та процесора, що також збільшує витрати.

Рендеринг на стороні клієнта (CSR)

У випадку рендерингу на стороні клієнта, сервер надсилає HTML-файл, заповнений лише мінімальним вмістом. Клієнтський JavaScript-код отримує дані із сервера та відтворює повний інтерфейс користувача у браузері.

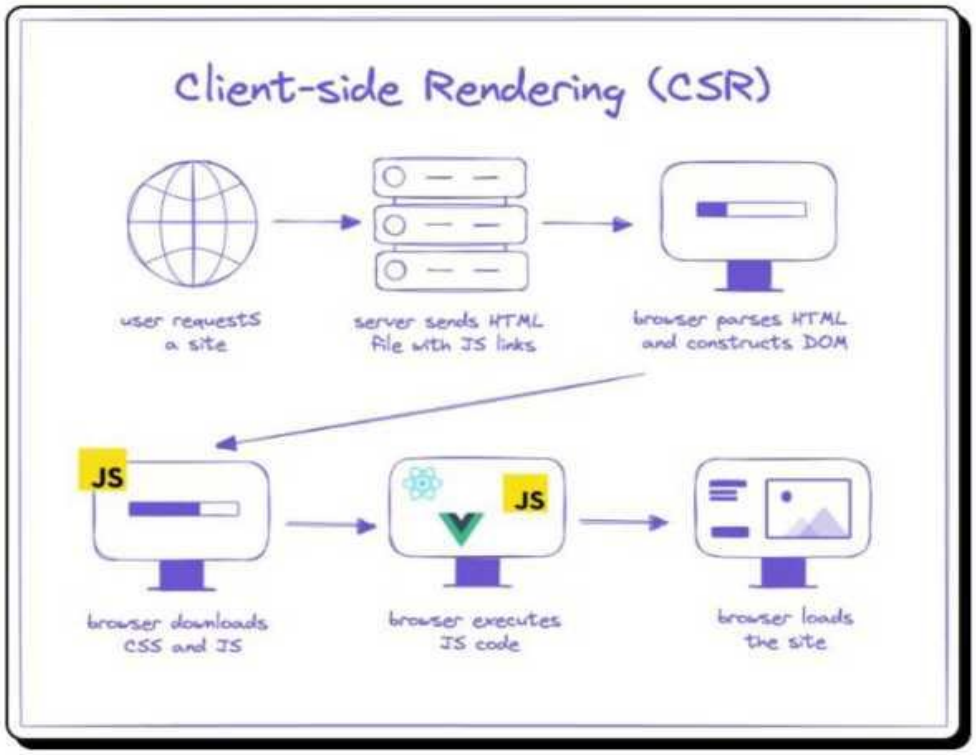

Рисунок 2.2 - Ескіз логіки рендерингу на стороні клієнта.

Як показано на рисунку 2.2 , у випадку клієнтського рендерингу сервер передає браузеру HTML-файл, що містить посилання на файли CSS та JavaScript. Браузер аналізує його та будує DOM-дерево сторінки (модель об'єктів документа). Після завантаження ресурсів CSS та JavaScript браузер рендерить веб-сторінку, включаючи всі необхідні елементи (текст, зображення, кнопки, стилі). Якщо є JavaScript, браузер також виконує його для рендерингу динамічного вмісту, форм та отриманих даних через API. Браузер виконує повторний рендеринг та оновлення на основі взаємодії з користувачем.

Завдяки рендерингу на стороні клієнта навантаження на сервер зменшується, а інтерактивність веб-застосунку стає доступнішою. Однак важливо враховувати, що початковий час завантаження довший, а також деякі обмеження SEO-оптимізації. Крім того, користувач побачить порожню сторінку,

якщо JavaScript вимкнено.

Зрештою, вибір найкращого типу рендерингу залежить від різних факторів. Які основні вимоги до продуктивності? Наскільки важлива SEO-оптимізація? Який рівень складності з точки зору розробки та підтримки є прийнятним? Чи визначені вимоги до масштабованості? Відповідаючи на ці питання, стає зрозумілим, який тип рендерингу обрати для конкретної програми.

2.3 Node.JS та Express

Щоб налаштувати серверну частину застосунку найефективнішим чином, розробники мають у своєму розпорядженні багато різних інструментів. Деякі інструменти мають унікальне призначення, а інші пропонують набір різних інструментів, які є у розпорядженні розробника. Node.JS та Express – це два популярні інструменти, що широко використовуються для налаштування серверного середовища.

Node.JS

Node.js — це середовище виконання та бібліотека з відкритим вихідним кодом, що працює на різних платформах. Воно використовується для серверного програмування, здебільшого для розгортання традиційних веб-сайтів та бекенд-сервісів API. Node.js — це не фреймворк і не бібліотека, а середовище виконання JavaScript, яке зазвичай є місцем виконання програми або застосунку на стороні сервера [16].

Node.js працює в одному потоку, що означає, що він може обробляти тисячі циклів подій, що відбуваються одночасно. Однопоточна архітектура дозволяє виконувати події швидше та ефективніше, ніж багатопотокові налаштування.

Node.js складається з багатьох різних частин: обробка помилок, глобальні елементи, кластер, консоль, модулі, відладчик, DNS, домен, буфер та потокова передача. Всі ці окремі елементи, об'єднані разом, утворюють Node.js і допомагають обробляти кожну деталь, необхідну для якісної розробки додатків.

Але Node.js — це ще більше, ніж просто інструменти, які можуть підтримувати подальшу розробку додатків. Express — один з них.

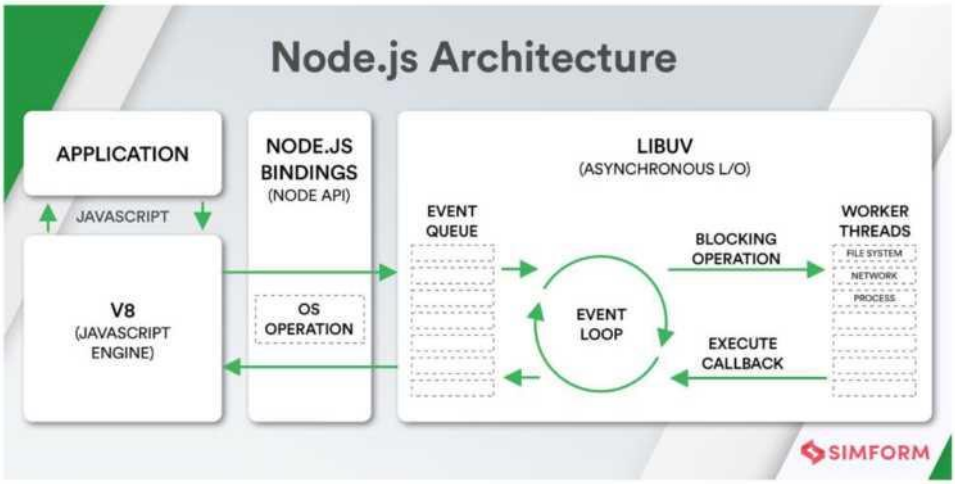

Рисунок 2.3 - Архітектура node.Js (sinform.Com, через simplilearn, 2024).

Експрес

Express — це неупереджений JavaScript-фреймворк: він гнучкий і не вимагає чіткого структурування застосунку. Він використовується поверх шару Node.js і допомагає керувати серверами та маршрутами, розробляти мобільні веб-застосунки, а також API. Express не можна використовувати без Node.js [20].

Express — це багатий та модульний фреймворк, простий у розумінні та використанні, який дозволяє розробникам додавати або видаляти функції відповідно до своїх потреб. Express також пропонує повний набір проміжного програмного забезпечення. Це дозволяє розробникам легко виконувати поширені завдання (наприклад, аналіз тіл запитів, зменшення розміру відповідей або обробку файлів cookie) та мати більше контролю над системою маршрутизації. Окрім набору проміжного програмного забезпечення, Express пропонує велику кількість плагінів та модулів, що дозволяє розробникам легко створювати масштабовані програми, а також деякі функції налагодження [14].

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World!');
});
server.listen(port, hostname, () => {
    console.log('Server running at http://${hostname}:${port}/');
});
```

Рисунок 2.4 - Приклад коду простої програми, написаної за допомогою express.

Express — це дуже універсальний фреймворк, який можна використовувати для створення API для односторінкових додатків (SPA), а також для додатків реального часу, потокового передавання та фінтех-додатків, а також звичайних API.

2.4 Висновки go розділу

У цьому розділі було розглянуто ключові функції серверної частини веб-застосунку, що відповідає за обробку запитів користувача, взаємодію з базами даних, логіку авторизації та генерацію контенту. Висвітлено два основні типи рендерингу – серверний (SSR) та клієнтський (CSR), кожен із яких має свої переваги залежно від потреб проєкту. SSR забезпечує швидке завантаження контенту та краще індексується пошуковими системами, тоді як CSR дозволяє створювати більш інтерактивні та гнучкі інтерфейси. Окрему увагу приділено Node.js – сучасному середовищу для виконання JavaScript на сервері – та Express, як легкому, зручному у використанні фреймворку для розробки API та маршрутизації. Ці технології є потужною основою для створення масштабованих і продуктивних веб-додатків.

РОЗДІЛ 3. ФРОНТ-ЕНД АБО КЛІЄНТСЬКА СТОРОНА

Веб- чи мобільний застосунок не може бути повноцінним без бекенду чи фронтенду. Якщо бекенд займається більше технічними аспектами розробки застосунку, фронтенд не менш важливий.

Розробка частини застосунку, яку користувач бачитиме та з якою взаємодіятиме, є важливим кроком у процесі розробки. Фронтенд-розробка відповідає за загальний візуальний аспект та користувацький досвід, те, як відбувається навігація, як дизайн адаптується до різних розмірів екранів, а також за продуктивність застосунку. Загалом, фронтенд-частина проєкту є таким же важливим аспектом процесу розробки, як і бекенд-частина.

3.1 Адаптивний дизайн

Наявність веб-сайту, який чудово виглядає на екрані комп'ютера, недостатньо, і вже деякий час цього недостатньо. Окрім екранів настільних комп'ютерів, більшість веб-сайтів також доступні на менших екранах, таких як планшети або ще менші, смартфони. Адаптивний дизайн – це дизайн, який дозволяє веб-сайту добре виглядати на будь-якому екрані.

З точки зору фронтенду, адаптивний дизайн досягається за допомогою медіа-запитів CSS, які є точками зупинки для кожного розміру екрана, що дозволяють змінювати деякі елементи для певного розміру екрана без необхідності змінювати весь код.

Адаптивність – це питання того, як веб-сайт обробляється на стороні фронтенду розробки, але відправною точкою є етап проектування. Адаптивний дизайн враховує весь макет, зображення, текстові блоки та інші компоненти. З цієї причини пріоритет мобільних пристроїв як відправної точки для розробки став важливим способом роботи з адаптивним дизайном, і про це не слід забувати на початковому етапі роботи на етапі проектування. Існує багато front-end практик, які можна використовувати для легкої обробки адаптивності. Медіа-

запити є однією з технік CSS, що використовуються, і вони виглядають наступним чином:

```css
@media only screen and (max-width: 600px) {
  body {
    background-color: blue;
  }
}
```

Рисунок 3.1 - Приклад медіа-запиту.

@media містить блок властивостей CSS, до яких буде доступ , якщо певна умова буде істинною. Медіа-запити найкраще використовувати з різними точками зупинки. Кожна точка зупинки представляє розмір екрана, допомагаючи створити дизайн, який працюватиме по-різному залежно від розміру. Наразі існують різні розміри екранів, тому спроби орієнтуватися на кожну можливу висоту та ширину пристрою стають складними. W3Schools рекомендує орієнтуватися на ці п'ять груп (рис. 3.2):

```css
/* Extra small devices (phones, 600px and down) */
@media only screen and (max-width: 600px) { }

/* Small devices (portrait tablets and large phones, 600px and up) */
@media only screen and (min-width: 600px) { }

/* Medium devices (landscape tablets, 768px and up) */
@media only screen and (min-width: 768px) { }

/* Large devices (laptops/desktops, 992px and up) */
@media only screen and (min-width: 992px) { }

/* Extra large devices (large laptops and desktops, 1200px and up) */
@media only screen and (min-width: 1200px) { }
```

Рисунок 3.2 - Рекомендовані точки зупинки медіа-запитів.

У межах @media код може змінювати поведінку елементів. Це означає керування розмірами шрифтів, кольорами, полями, відступами, а також напрямком руху деяких елементів. За допомогою різних груп медіа-запитів (

рис. 3.3) кожен розмір екрана можна оптимізувати та отримувати контент, який відповідає доступному простору, забезпечуючи оптимальний користувацький досвід.

Деякі методи CSS-макетування також цікаві для полегшення роботи з адаптивністю. За допомогою Flexbox Layout легко розміщувати, вирівнювати та розподіляти простір між елементами, присутніми в контейнері, навіть якщо розмір цих елементів невідомий або динамічний. Це дозволяє елементам у контейнері заповнювати доступний простір, змінюючи їх висоту та ширину (розширюючи або стискаючи їх). За допомогою Flexbox Layout можна обробляти як контейнер (батьківський), так і елемент у ньому (дочірні елементи) за допомогою різних властивостей.

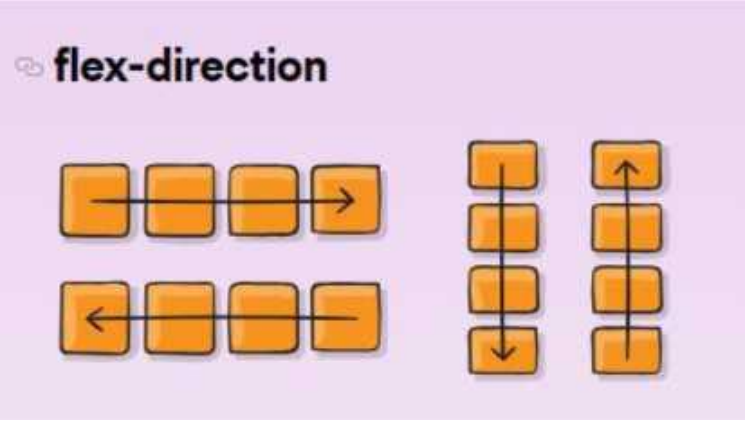

Рисунок 3.3 - Приклад обробки батьківського компонента за допомогою flexbox layout.

За допомогою властивості .flex-direction вміст можна вирівняти вертикально або горизонтально, використовуючи значення row , row-reverse , column або column-reverse .

У випадку з рисунком 3.4 використовується властивість .flex-wrap , яка дозволяє елементам переноситися за потреби залежно від розміру екрана. Більш конкретно, ця властивість дозволяє елементам, які за замовчуванням намагаються поміститися в одному рядку, динамічно переміщуватися на кілька

рядків - зверху вниз або знизу вгору, використовуючи значення nowrap , wrap або wrap-reverse .

Рисунок 3.4 - Приклад використання flexbox layout для адаптивності веб-сайту. Веб-сайт (угорі), планшет (ліворуч внизу), смартфон (праворуч внизу)

3.2 Оптимізація продуктивності

Якщо бекенд може допомогти з оптимізацією продуктивності, це також є функцією фронтенду застосунку. Оптимізація продуктивності має вирішальне значення для будь-якого застосунку, оскільки користувачі швидко втрачають свою зацікавленість, якщо застосунок працює недостатньо швидко або містить

деякі недоліки, що заважають їхньому досвіду роботи з ним. Фактично, згідно з дослідженням, проведеним The [24], «сайт, який завантажується за 1 секунду, має коефіцієнт конверсії в 3 рази вищий, ніж сайт, який завантажується за 5 секунд».

Завдяки різним покращенням та методам, фронтенд-розробник може зробити веб-сайти завантаженими швидше та працювати ефективніше:

Таблиця 3.1

Приклади фронтенд-технік для ефективного веб-сайту.

Техніка	**Опис**
Лаконічний та ефективний HTML-текст	Для швидшого та легшого процесу сканування.
Стиснення файлів	Зменшує обсяг місця, яке займає файл. Це дозволяє вебсайту швидко обробляти файли та швидше завантажуватися.
Оптимізація зображень та відео	Можна досягти за допомогою різних методів: стиснення, зміна розміру, ліниве завантаження, і це лише деякі з них.
Зменшення зовнішніх HTTP-запитів	Шляхом видалення нерелевантних функцій (непотрібних зображень або JavaScript, додаткового CSS або невикористаних плагінів).
Використання відповідної структури, що підходить для типу проекту	Найкраще знати обсяг проєкту на початку, щоб уникнути вибору фреймворку з безліччю додаткових функцій та можливостей, які залишаться невикористаними.
Кешування	Тобто тимчасове зберігання даних, щоб уникнути їх постійного отримання (включаючи кешування браузера, кешування CDN та кешування на стороні сервера).

Також можна розглянути більше технічних підходів, згаданих [18] у статті, і представлених нижче.

Розділення пакетів або коду

Цей метод особливо цікавий для програм з великими кодовими базами або складними залежностями. Він працює шляхом розділення великої кодової бази на менші частини та завантаження цих частин лише за необхідності.

Динамічний імпорт та відкладене завантаження

Динамічний імпорт відбувається, коли код імпортується на основі взаємодії, видимості або навігації користувача. Так само, лінивий код затримує ініціалізацію об'єкта, щоб відобразити його лише тоді, коли цей об'єкт потрібен користувачеві.

Попереднє завантаження

Попередня вибірка, як випливає з назви, завантажує різні ресурси до їх запиту, у фоновому режимі. Завдяки попередній вибірці сприйнятий час очікування та затримка зменшуються, а швидкість реагування покращується – це можливо завдяки моделям поведінки або прогностичним алгоритмам. Існує більше методів оптимізації веб-сайтів і додатків, і кожен з них має свої переваги та недоліки. Важливо врахувати, які з них підходять для конкретного типу проекту, і чи виправдовують результати оптимізації вкладені в них зусилля.

3.3 Фронтенд-розробка

Сьогодні доступно багато інструментів для допомоги в дизайні, отримання певних шрифтів та спрощення роботи зі стилізацією, що робить розробку фронтенду більш плавною та простою. Фреймворки є більш повним рішенням, ніж бібліотеки. Фреймворк складається з цілої екосистеми та архітектури для створення застосунку. Фреймворк також має певний спосіб організації свого коду. З іншого боку, головною метою бібліотеки є надання модулів коду, які можна повторно використовувати для конкретних завдань. Вона розроблена для використання певної функціональності без попередньо визначеної структури.

React

React — це бібліотека JavaScript, розроблена META, як для веб-сайтів (ReactJS), так і для мобільних пристроїв (React Native). React допомагає створювати великомасштабні додатки. Вона проста, швидка та масштабована, а також дозволяє змінювати дані без перезавантаження сторінки.

React базується на компонентах. Різні компоненти створюються окремо та можуть використовуватися разом для побудови застосунку. Цей модульний підхід дозволяє легко повторно використовувати різні компоненти та допомагає підтримувати чіткий та лаконічний код. Хуки також є важливою функцією React. Хуки дозволяють розробнику використовувати стан, а також інші функції React без написання компонентів класу. Деякі популярні хуки - це usestate (для керування станом у компоненті), useEffect (для виконання вторинних ефектів, таких як отримання даних) та useContext (для доступу до значень контексту з будь-якої точки дерева компонентів) (React).

React-хуки

React-хуки є однією з найвідоміших та найспецифічніших функцій бібліотеки. У цьому розділі детальніше розглядаються три з них, цікаві з точки зору навігації : useParams , useNavigate та useLocation .

Щоб зрозуміти контекст цих гачків, необхідне знайомство з React Router DOM. Коротко кажучи, react-router-dom – це пакет, який дозволяє імплантувати динамічну маршрутизацію, що дає змогу відображати сторінки та навігувати користувачам по них. За допомогою React Router навігація між сторінками здійснюється без перезавантаження в будь-який момент. Це забезпечує швидку навігацію сторінками, покращуючи користувацький досвід.

DOM маршрутизатора React складається з кількох компонентів, включаючи Router, Routes, Route та Link .

```
<BrowserRouter>
    <Routes>
        <Route path="/ " element={<Home/>} />
        <Route path="/products" element={<Products/>} />
        <Route path="/about" element={<About/>} />
    </Routes>
</BrowserRouter>
```

Рисунок 3.5 - Приклад фрагмента коду використання dom react router.

Router (зазвичай імпортується як BrowserRouter) – це батьківський компонент, який зберігає всі інші компоненти. Маршрути (Routes) визначають

шляхи навігації, вони визначають, які компоненти відображати, на основі поточної URL-адреси або розташування програми. Компонент Route перевіряє поточну URL-адресу та відображає пов'язаний компонент. Нарешті, компонент Link використовується для створення посилань на різні маршрути [12].

useParams повертає динамічну частину URL-адреси та дозволяє доступ до неї. Цей гачок дуже корисний під час створення динамічних маршрутів, де можуть змінюватися лише частини URL-адреси.

useNavigation

Функція хука useNavigate доповнює інші хуки React для спрощення процесу зміни URL-адреси. Раніше вона була відома як useHistory . Після виклику функція navigate() приймає рядок як аргумент для переходу до правильного шляху. У прикладі, функція викликається, коли викликається handleClick() – далі в операторі return: <button onClick={handleClick}>Перейти на іншу сторінку</button> . Після натискання кнопки користувач переходить до маршруту « /other-page ».

Функція navigate() може приймати другі параметри, які можна використовувати для керування поведінкою навігації (React Router): options.replace , options.state , options.preventScrollRest , options.relative , options.unstable_flushSync та options.unstable_viewTransition . React Router визначає другий аргумент наступним чином: «Якщо вказати replace: true, навігація замінить поточний запис у стеку історії замість додавання нового».

Використовується для доступу до інформації про поточне розташування URL-адреси у формі об'єкта та може містити властивості, такі як pathname (поточний шлях URL-адреси, наприклад, /cat/cinnamon), пошук (частина рядка запиту URL-адреси, наприклад, ?breed=ragdoii), хеш (частина фрагмента хешу URL-адреси, наприклад, #top) та стан (будь-які додаткові дані про стан, пов'язані з місцем розташування).

Notion — це веб-застосунок для роботи з робочим простором. Користувачі Notion можуть писати, планувати та співпрацювати над різними проектами. Notion працює з різними блоками, які можна додавати одним кліком

на головній сторінці. Блоки включають заголовки (1, 2 та 3), абзаци, марковані списки, нумеровані списки, зображення, відео, вбудовані посилання, список справ, перемикальний список, стовпці та таблиці, серед іншого. Сторінки Notion можуть бути типовими сторінками, а також тим, що Notion API визначає як «бази даних»: таблиці, дошки, діаграми, часові шкали або календарі.

Ці різні типи блоків та файлів роблять Notion дуже універсальним та легким для розуміння інструментом, ідеальним для створення нотаток, планування проектів та написання текстів, наприклад. REST API Notion було випущено у 2022 році та дозволяє користувачам використовувати Notion двома різними способами: інтегрувати існуючий контент Notion у власний застосунок або вставляти контент у свій проект Notion.

Різні кінцеві точки дозволяють маніпулювати різними об'єктами Notion, зокрема:

Таблиця 3.2

Кінцеві точки notion.

Елемент	*Опис*
Сторінки	Створення, оновлення та видалення сторінок, які можуть містити текст, зображення, бази даних та інший контент.
Бази даних	Створення та керування базами даних, які можуть зберігати та впорядковувати дані структурованим чином.
Блоки	Робота з окремими блоками на сторінках, такими як текстові блоки, блоки коду та блоки зображень.
Користувачі	Керування користувачами та їхніми дозволами в робочому просторі Notion.
Файли	Завантаження та керування файлами, пов'язаними зі сторінками та базами даних.

Початок роботи з Notion API відбувається за тим самим шаблоном, що й робота з іншими API. Користувач отримує свій токен Notion API для автентифікації своїх запитів (його можна згенерувати в налаштуваннях Notion), вибирає мову для роботи (Notion API підтримує Python, JavaScript та Ruby, серед інших) та здійснює виклики API, надсилаючи запити до кінцевих точок Notion

API. Документація API містить детальну інформацію про доступні кінцеві точки та їх параметри.

3.4 Висновки по розділу

У цьому розділі зосереджено увагу на клієнтській частині застосунку, яка забезпечує зручну, адаптивну та ефективну взаємодію з користувачем. Описано принципи створення адаптивного дизайну, що дозволяє коректно відображати інтерфейс на різних типах пристроїв та екранів. Розглянуто основні методи оптимізації продуктивності, включаючи зменшення обсягів ресурсів, асинхронне завантаження та кешування. Основним інструментом розробки обрано React – популярну бібліотеку для побудови компонентної архітектури інтерфейсу, що дозволяє ефективно оновлювати сторінку без повного перезавантаження. Застосування React-хуків, таких як useState, useEffect тощо, дозволяє зручно працювати зі станом компонентів і життєвими циклами, забезпечуючи гнучкість і масштабованість при реалізації динамічних інтерфейсів.

РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ ВЕБ-ПРИСТОСУНКУ: REACT, NOTION API ТА АРХІТЕКТУРА

Два основні проекти, які будуть розглянуті, – це впровадження бази знань та блогу. Вміст обох сторінок створюється та розміщується в Notion, потім витягується та відображається в React-застосунку за допомогою Notion REST API. Notion, для цілей цього проекту, використовується як система керування контентом (CMS).

База знань містить важливу інформацію для різних користувачів та зацікавлених сторін інструменту Peili Vision. База знань використовується для відображення статичної інформації: якщо має бути контент, що містить динамічні елементи, в іншому випадку він розміщується.

З реалізацією блогу процес подвійний: відображення карток, які містять інформацію, необхідну для ідентифікації кожної статті, та отримання звичайної сторінки Notion, що містить вміст статті. Також було вирішено зробити так, щоб кнопка, яка веде до статті, також підтримувала гіперпосилання, оскільки Peili Vision може потребувати посилання на дослідницьку роботу або будь-яку іншу зовнішню статтю. Наразі невідомо, чи буде цей блог зрештою використовуватися, але було вирішено все ж продовжити його реалізацію.

REST API Notion працює таким чином, що кожна сторінка ідентифікується певним ідентифікатором (який зазвичай називають «ідентифікатором сторінки»), присутнім у вихідній URL-адресі Notion. Остання комбінація цифр і літер утворює ідентифікатор сторінки.

Основний механізм отримання даних полягає в отриманні правильного ідентифікатора сторінки для відображення в застосунку React. Після визначення та відображення ідентифікатора сторінки різні елементи сторінки вибираються у вигляді блоків з Notion. Кожен блок представляє елемент дизайну (наприклад, заголовок, таблицю, зображення, маркований список).

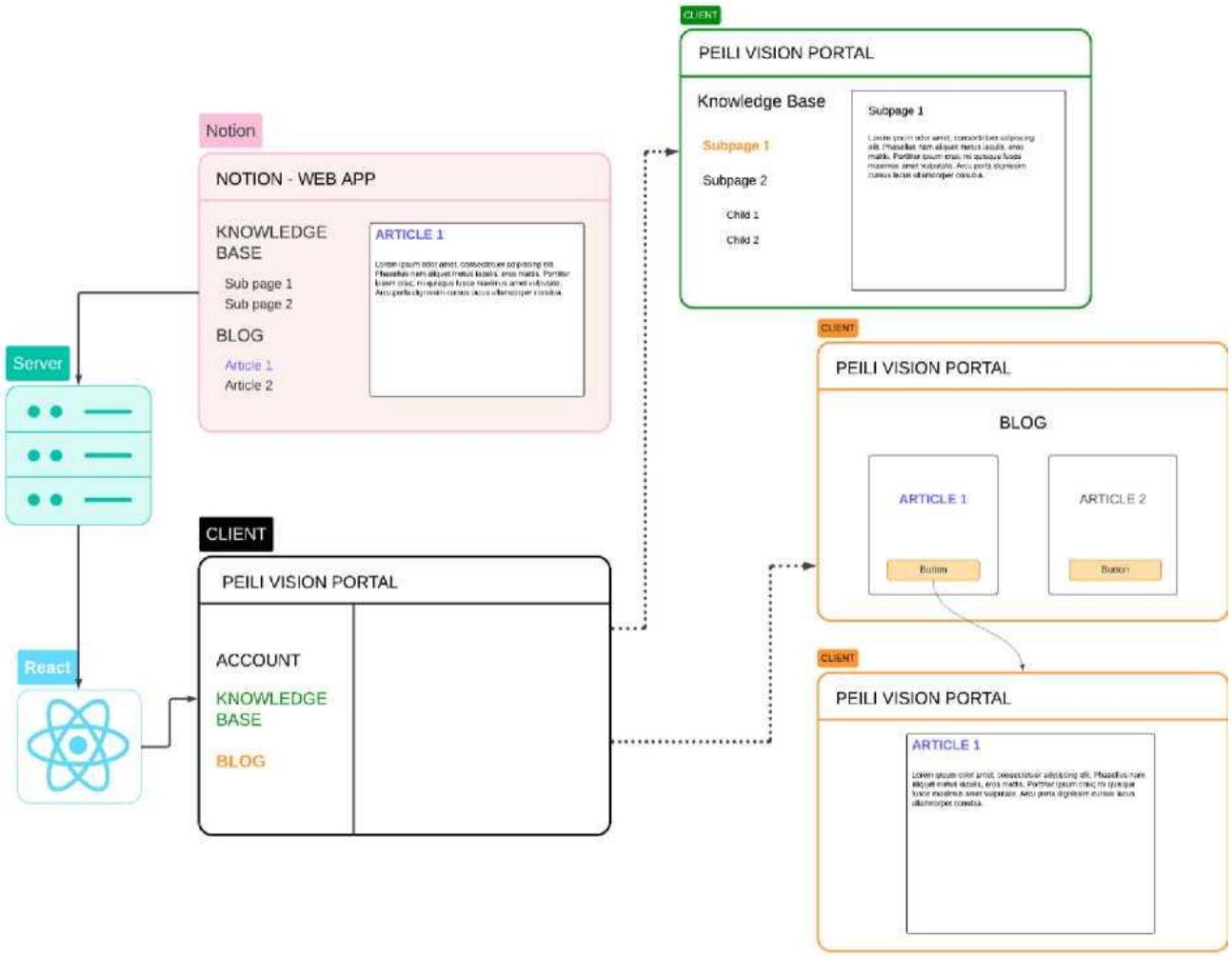

Рисунок 4.1 - Діаграма проекту, що показує різні взаємозв'язки між інструментами та елементами.

Notion дозволяє користувачеві створювати сторінки (часто звані «звичайними сторінками»), а також бази даних, графіки та таблиці. Важливо вказати та зрозуміти цю різницю, оскільки API обробляє їх по-різному. Якщо База знань, представлена вище, створюється за зразком звичайної сторінки в Notion, то Блог створюється як База даних. Ці два проекти розглядалися окремо.

4.1 Архітектура та реалізація інтеграції Notion API у React-додаток

Серверна частина Notion REST API

Для цього проєкту використовувався Express. Те, що спочатку було сервером та окремо побудованим React-додатком, було перетворено на CSR-

додаток за допомогою Webpack.

Код визначає маршрут для обробки GET-запитів до URL-шляху /fetchDataFromNotion/ :pageid з динамічним параметром :pageid , який отримує ідентифікатор сторінки з URL-адреси. Відповідь від fetchPageData деструктурується на заголовок (який отримує заголовок сторінки) та блоки (які є масивом, що містить усі об'єкти блоків, включаючи основний вміст).

Потім код перебирає кожен об'єкт блоку та намагається знайти певну функцію-обробник для поточного типу блоку (block.type) в об'єкті blockProcessor.blockHandlers . Якщо певний обробник не існує (наприклад , paragraph: this.processParagraph.bind(this) , toggle: this.processToggle.bind(this)), він повернеться до обробника за замовчуванням.

Використання Promise.all гарантує, що вся обробка блоків буде завершена перед подальшим виконанням.

Клієнтська сторона

На стороні клієнта дані отримуються з кількох місць, але фрагмент коду, взято зі сторінки, що отримує скелет Бази знань (KnowledgeBase.js), з якої викликається SubPage.js як компонент, що отримує та відображає дочірні сторінки. Для отримання даних з Notion було використано хук React useEffect . У цьому конкретному прикладі він базується на властивості country . Хук UseEffect також містить логіку кешування.

Рендеринг застосунку

Як згадувалося раніше, цей проект React-додатку та Express-сервер були створені та запущені окремо. Для потреб поточної архітектури та середовища розробки це було нежиттєздатним. Після деяких досліджень було вирішено перетворити існуючу логіку на клієнтський (CSR) додаток. Для досягнення бажаного результату було використано пакет модулів Webpack та завантажувач Babel. Після деяких коригувань існуючого коду, CSR-додаток було завершено: він зрештою обслуговує простий HTML та пакет React, який рендерить React у додатку.

У цьому випадку використання CSR є гарним варіантом з кількох

причин. Перша полягає в тому, що немає потреби підтримувати занадто великий трафік. Друга — потреби в SEO невеликі. Цей додаток створено таким чином, щоб бути доступним лише для людей, які вже працюють з інструментами Peili Vision.

Третя причина – це терміни впровадження. Цікаво відзначити, що База знань була створена окремо перед інтеграцією в кінцевий додаток (портал), і що логіка рендерингу React-додатку обговорювалася та визначалася, коли База знань вже повністю функціонувала. Реалізувати SSR без фронтенд-фреймворку складно, і оскільки проект є виключно React-додатком, більшу частину проекту довелося б продумати ще раз, а це було неможливо. Next.js широко рекомендувався під час досліджень для допомоги у безперебійному створенні SSR, але часу, що залишився до запланованого випуску Бази знань, було недостатньо для початку таких налаштувань.

Для ефективного рендерингу різних блоків було застосовано об'єктно-орієнтоване програмування (ООП). У цьому конкретному випадку ООП було необхідно в першу чергу для рендерингу блоків у застосунку React, оскільки вони були спочатку розташовані в Notion.

Спочатку код був створений без використання ООП для ознайомлення з ним, але його дуже швидко почали використовувати, оскільки порядок блоків був би не таким, як потрібно, і було потрібне рішення. Наприклад, якщо в Notion елементи розташовувалися б у такому порядку: заголовок, потім абзац, зображення та ще один абзац під ним, то в застосунку React це відображалося б за типом елемента: заголовок, обидва абзаци та зображення останніми, оскільки саме в такому порядку вибиралися елементи.

Реалізація ООП починається з визначення класу BiockProcessor , що містить конструктор biockHandiers , який ініціалізує кожен необхідний блок з Notion. Тут демонструється один із принципів ООП, «інкапсуляція», шляхом розкриття лише необхідних методів, без показу внутрішньої реалізації їхніх деталей.

Кожен метод окремо отримує певний блок, а в деяких випадках були

необхідні асинхронні функції: деякі блоки (включаючи стовпці, блоки перемикання та таблиці) складаються з батьківського та дочірнього компонентів. Асинхронні функції були необхідні спочатку для отримання ідентифікатора блоку батьківського компонента, а потім для отримання ідентифікатора дочірнього компонента. Різний вміст потім поміщається в масив і перехоплюється на стороні застосунку React в операторі Switch.

Рисунок 4.2 - Приклад коду відповіді processcolumn() у server.Js.

has_chiidren: true вказує на те, що дійсно існують дочірні блоки для отримання всього вмісту блоку стовпців. Дочірні блоки можуть обробляти як текстові (абзаци) блоки, так і зображення.

Щоразу, коли блок містить текст, він обробляється у два кроки, за допомогою виклику функції processRichText . Наприклад, processParagraph(biock) приймає об'єкт блоку як вхідні дані та припускає, що він містить властивість, яка сама містить вміст форматованого тексту (у цьому конкретному випадку, абзац). Потім він викликає process sRichText для подальшої обробки: це дозволяє тексту підтримувати будь-які анотації (текст виділений жирним шрифтом, курсивом, обведений або іншим кольором), а також

містити будь-яку потенційну інформацію про посилання. Це приклад поліморфізму (ще один принцип ООП) через динамічне зв'язування: processParagraph викликає processRichText , який виконується залежно від типу середовища виконання аргументу richTextArray .

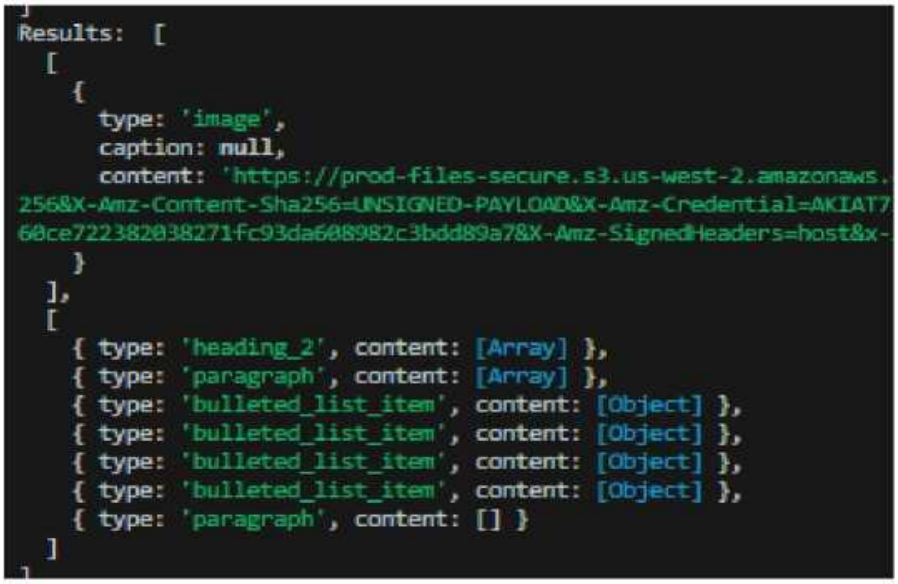

Рисунок 4.3 - Приклад коду відповіді processcolumn() у server.Js.

У випадку processimage, функція витягує URL-адресу зображення, а також будь-який доданий підпис (вона повертає null, якщо його не надано). Відповідь блоку зображення можна побачити на рисунку 4.3. На стороні клієнта більша частина рендерингу даних обробляється в операторі Switch — короткий огляд цього оператора наведено нище. Кожен тип блоку обробляється в операторі окремо, і саме в ньому можна також опрацьовувати деякі деталі. Наприклад, випадок ' buiiet_iist_item ' оператора Switch, що відповідає за рендеринг елементів блоку маркованого списку, побудований таким чином, що підтримує різні рівні маркованих списків. Також в цьому операторі Switch блоковий елемент 'toggle' отримує свій стиль перемикання панелей.

Оптимізація продуктивності та швидкості реагування

Щоб розпочати оптимізацію застосунку React, розділення коду виконується за допомогою React.lazy , що дозволяє відображати динамічний

імпорт як звичайні компоненти.

```
import Navigation from './components/navigation.jsx';
import Blog from './pages/Blog.js';
import AboutUs from './pages/AboutUs.js';
import KnowledgeBase from './pages/KnowledgeBase.js';
import Article from './components/article.js';
```

Рисунок 4.4 - Приклад коду імпорту звичайної сторінки.

На рисунку 4.4 показано оригінальні оператори імпорту різних сторінок, що використовуються в логіці React Router застосунку React, до використання React.lazy .

```
const Navigation = lazy(() => import( './components/navigation.jsx'));
const Blog = lazy(() => import('./pages/Blog.js'));
const AboutUs = lazy(() => import('./pages/AboutUs.js'));
const KnowledgeBase = lazy(() => import('./pages/KnowledgeBase.js'));
const Article = lazy(() => import('./components/article.js'));
```

Рисунок 4.5 - Приклад коду імпорту сторінки за допомогою react.Lazy.

Рисунок 4.5 – це оригінальний імпорт, змінений таким чином, що початковий рендеринг компонента запускатиме завантаження пакета, наприклад, Navigation , що гарантуватиме його готовність до використання за потреби.

Далі в App.js, Suspense використовується в операторі React Router , глобізуючи Routes , щоб можна було відобразити резервний оператор під час очікування завантаження лінивих компонентів (Рис. 4.6). У цьому випадку повідомлення, яке має відображатися, — «Очікування...».

Рисунок 4.6 - Приклад коду використання react suspense.

Початкове завантаження сторінки (до використання React.lazy) було досить мінімальним, тому React.lazy ще не довів свою радикальність, хоча іноді можна побачити резервний варіант, а це означає, що він певною мірою корисний. React.lazy легко та швидко впроваджувати, що робить його гарною звичкою. Особливо у випадку, коли База знань все ще перебуває на самому початку свого розвитку, а це означає, що він може стати важливим у майбутньому.

React.lazy та Suspense також були реалізовані в головному компоненті програми. Оскільки цей компонент містить кілька підкомпонентів, ліниве завантаження має бути корисним. Використовуючи React.lazy та Suspense , підкомпоненти можуть зменшити складність сторінки та розмір пакета, оскільки дані вибираються динамічно, а деякі елементи інтерфейсу відображаються умовно.

Найпоширенішими перевагами його використання в цьому випадку є покращений користувацький досвід (UX) та скорочений час початкового завантаження, оскільки завантажуються лише необхідні компоненти для початкового рендерингу. Додаткові компоненти (chiidPage , KnowiedgeBaseintro) завантажуються лише за потреби. React.lazy та Suspense не використовуються більше в коді, оскільки жоден інший компонент не є таким складним, як

попередній (KnowledgeBase). Використання React.lazy та Suspense не мало б сенсу , і також добре використовувати їх економно.

Кешування за допомогою хука useRef на React

Ідея Бази знань полягає у відображенні інформації, розміщеної в Notion, та її доступності кількома мовами. З практичних міркувань було вирішено, що в Notion будуть створені папки (які насправді є сторінками, що містять підсторінки, але для ясності їх будемо називати «папками») з однаковим вмістом, перекладеним потрібними мовами. Наприклад, буде папка, що містить сторінки з вмістом англійською мовою, та інша папка з таким самим вмістом, перекладена фінською або будь-якою потрібною мовою. Переклад основних даних React-додатку обробляється за допомогою i18next, а зміна ідентифікатора папки обробляється окремою логікою, присутньою в документі JavaScript, що обробляє основний прийом ідентифікатора сторінки.

Коротко кажучи, логіка працює наступним чином: користувач може вибрати бажану мову за допомогою селектора мов, розташованого у верхньому правому куті програми. Вибір мови призначає різний ідентифікатор сторінки, що відповідає правильному ідентифікатору папки. Якщо логіка спочатку працювала належним чином, під час першої зміни мови виникала затримка, і ця затримка була постійною після повторних змін, враховуючи, що перезавантаження не виконувалося. Ця затримка була спричинена викликом API при кожному перемиканні мови – найімовірніше, через використання хука useEffect для обробки отримання сторінки.

Якщо час завантаження під час першого перемикання мови наразі не викликає занепокоєння, кешування було необхідним, щоб уникнути постійного виклику API для кожного перемикання мови. Для продовження було використано хук React useRef , оскільки його метою є збереження значень між рендерами. useRef ініціалізується за допомогою оператора: const dataRef = useRef({}) .

Рисунок 4.7 - Фрагмент коду використання хука react useref для кешування.

На рисунку 4.7 показано застосування хука useRef для отримання та відображення потрібної сторінки через dataRef, який використовується як кеш для зберігання отриманих даних на основі pageId. Коли компонент вперше рендериться або коли змінюється країна, хук useEffect запускає функцію fetchData. Ця функція перевіряє, чи дані для поточного pageId вже кешовані в dataRef.current .

Якщо дані кешуються, вони встановлюються як стан fetchedData , що дозволяє уникнути мережевого вибору. Якщо дані не кешовані, функція отримує дані з вказаної URL-адреси та оновлює стан fetchedData отриманими даними. Вона також кешує отримані дані в dataRef.current , використовуючи pageid як ключ. Це гарантує, що наступні запити для того самого pageid використовуватимуть кешовані дані. Після цієї реалізації було помічено позитивні зміни. Якщо час завантаження після першого перемикання мови все ще триває, дані кешуються правильно та дозволяють плавно перемикатися між мовами після цього.

Покращення динаміки рендерингу сторінок

Відображення різних сторінок показано на наступному рисунку 4.8 : на

лівій панелі знаходиться зміст усіх доступних сторінок, деякі без дочірніх сторінок, а деякі містять підсторінки (головною папкою є «База знань»).

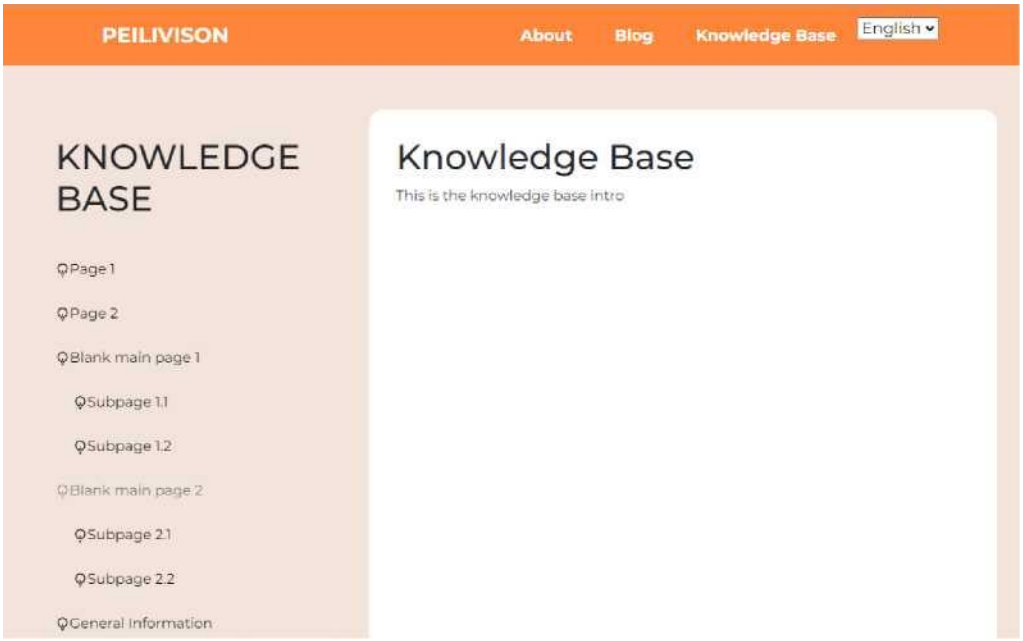

Рисунок 4.8 - Знімок екрана бази знань.

У цьому прикладі на лівій панелі Бази знань розташовані Сторінка 1 та Сторінка 2, які, як відомо, не містять дочірніх сторінок і не є порожніми. Пуста головна сторінка 1 та Пуста головна сторінка 2, як випливає з їхньої назви, – це дві порожні сторінки, обидві з яких містять дві дочірні сторінки: Підсторінку 1.1 та 1.2; та Підсторінку 2.1 та Підсторінку 2.2 (кожна з чотирьох містить вміст).

У Notion кожна дочірня сторінка (або підсторінка) залишає слід і є окремими блоковими елементами. Як показано на рисунку 4.9 , щоразу, коли головна сторінка містить дочірній елемент, створюється елемент, який вказує на наявність дочірньої сторінки та робить можливим доступ до цієї дочірньої сторінки.

Наявність блоків, що представляють дочірні сторінки, дозволяє створювати зміст та його реалізовувати безперешкодно. У розділі коду, що керує змістом (KnowiedgeBase.js), ними керує компонент <chiidPage/> , який також обробляє колір підсвічування вибраної сторінки (що робить навігацію зрозумілішою).

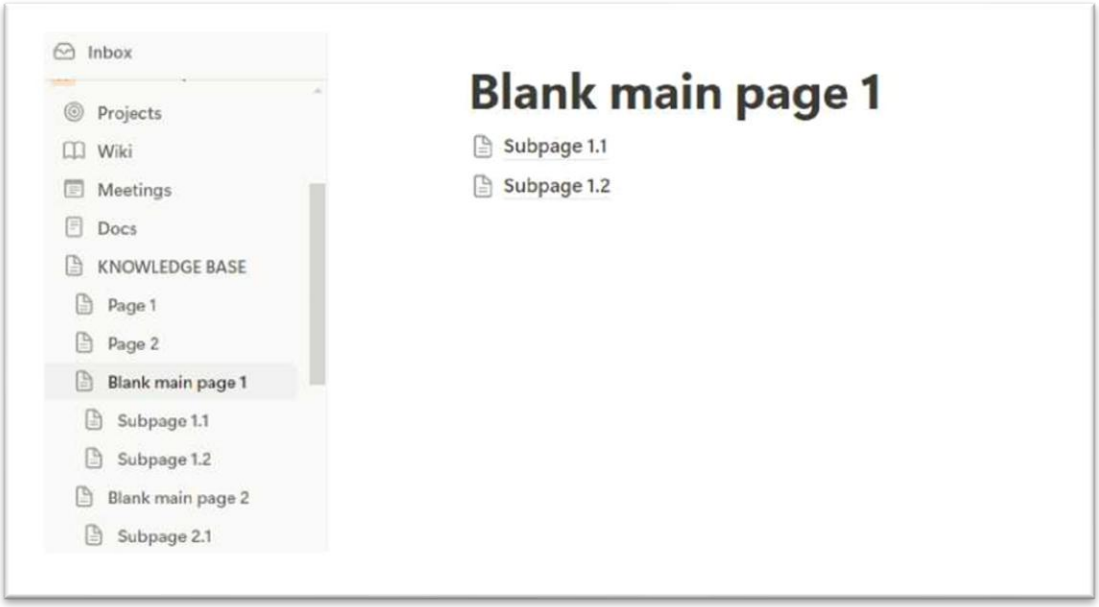

Рисунок 4.9 - Понятійний вигляд порожньої головної сторінки 1, що містить блокові елементи, які вказують на те, що вона містить підсторінки.

Однак було вирішено не використовувати той самий компонент <chiidPage/> у розділі коду, який керує відображенням кожної сторінки окремо (що можна побачити праворуч на рисунку 4.8). На практиці це означає, що той самий вигляд Notion, зображений на рисунку 4.9, виглядає так, як показано на рисунку 4.10: якщо сторінка порожня в Notion, вона все ще містить блокові елементи, що матеріалізують наявність дочірніх сторінок, на відміну від Бази знань, де сторінка порожня без будь-якої ознаки наявності підсторінок.

Для зручності користування було вирішено зробити так, щоб, якщо головна сторінка не містить жодного вмісту, відображався лише вміст першої підсторінки (також званої «першою дочірньою сторінкою»).

Для досягнення цього зміни відбудуться в компоненті <subPage/> , який спочатку відповідає за відображення вмісту звичайної сторінки.

Рисунок 4.10 - Вигляд веб-застосунку пустої головної сторінки 1.

Основні зміни для початку обробки відображення першої підсторінки на порожній головній сторінці полягають у введенні нового стану chiidPageData . Цей новий стан зберігає дані з дочірніх сторінок. fetchPageData отримує дані з певних дочірніх сторінок (активується biock.id).

функцію renderChildPage, яка призначена для обробки блокових елементів типу "child_page": block.type === 'child_page' . Ця функція спочатку перевіряє, чи доступний childPageData. Якщо ні, вона отримує необхідні дані під час входу в систему та повідомлення про завантаження. Якщо childPageData доступний, функція відтворює заголовок дочірньої сторінки, а також оброблений вміст за допомогою renderBlock (який спочатку існував для відображення різних блокових елементів, присутніх на сторінці). На стороні рендерингу, хоча fetchedData.processedContent було перенаправлено для рендерингу кожного блоку за допомогою renderBlock, його було замінено функцією renderContent . renderContent визначає, як відображати вміст головної сторінки. Спочатку перевіряється, чи доступні fetchedData та його оброблений вміст. Якщо всі блоки є дочірніми сторінками, рендериться перша дочірня сторінка за допомогою renderchiidPage . Якщо ні, вміст перенаправляється та використовується функція renderBlock для рендерингу інших типів блоків.

Ця перша спроба показала деякі багатообіцяючі перші результати, оскільки вміст першої підсторінки справді відображався на головній порожній

сторінці 1. Однак, цей самий вміст також відображався на головній порожній сторінці 2. Фактично, щоразу, коли нова головна порожня сторінка додавалася для тестування, завжди відображався той самий вміст підсторінки 1.1. На конкретному прикладі це означає, що якщо перша підсторінка - «Розуміння зору Peili», то саме ця конкретна сторінка відображалася на головній порожній сторінці 1 та головній порожній сторінці 2 (а також на гіпотетичній головній порожній сторінці 3 та інших).

Потім стало зрозуміло, що код виконував саме те, що від нього вимагалося: відображав вміст першої підсторінки на головній сторінці, хоча остання складалася лише з блоку типу " child_page ". Фактично, він відображав вміст підсторінки 1.1 (першої підсторінки, що існує у змісті) на кожній порожній головній сторінці (у цьому конкретному випадку, порожня головна сторінка 1 та порожня головна сторінка 2).

Тонкість полягала в тому, що насправді потрібно було виконати: коли головна сторінка містить лише елементи блокового типу " child_page ", має відображатися вміст першої підсторінки цієї конкретної головної сторінки, а не першої підсторінки загалом. Щоб досягти потрібного сценарію, знадобилися деякі зміни для обробки дочірніх сторінок на головній сторінці Notion. Першою значною модифікацією є введення useEffect, що глобізує fecthChildData , яка раніше була окремою функцією. У той час як у першій версії коду fetchChildPageData викликалася вручну, fetchChildPageData тепер викликається автоматично, коли стан fetchData змінюється - це означає, що fetchChildPageData спрацьовує лише тоді, коли завантажено головну сторінку, а всі блоки є дочірніми сторінками.

Коротко кажучи, цей useEffect гарантує, що дані дочірньої сторінки будуть отримані лише тоді, коли вміст головної сторінки складається з дочірніх сторінок. Якщо це так, він отримує першу дочірню сторінку та попередньо отримує її дані. Ця перша зміна підвищує ефективність коду, оскільки тепер він викликається лише за необхідності.

Друга суттєва зміна з'являється у функції renderContent . У той час як

перша версія перевіряє, чи всі блоки є дочірніми сторінками, і чи є хоча б один блок у processedContent , друга версія перевіряє, чи всі блоки є дочірніми сторінками та чи існує chiidPageData. Це дозволяє уникнути рендерингу дочірньої сторінки, якщо вона не містить необхідних даних. Щодо логіки рендерингу, перша версія безпосередньо рендерить першу дочірню сторінку, тоді як друга версія коду рендерить дочірню сторінку за допомогою childPageData .

Загалом, друга версія є легшою та спрощенішою. Підхід є ефективнішим завдяки використанню другого useEffect, який керує отриманням даних дочірньої сторінки. Він також працює краще та дозволяє досягти бажаного результату: коли головна сторінка порожня, він відтворює вміст своєї першої підсторінки (або дочірньої сторінки).

4.2 Побудова блогу: запити до бази, керування станом і навігація через React-хуки

Розділ блогу на вебсайті також базується на сторінці Notion. У цьому випадку елемент Notion не є звичайною сторінкою, а тим, що Notion називає «Базою даних». Бази даних поводяться інакше, ніж звичайні сторінки, і вимагають іншого підходу як під час роботи з Notion, так і під час роботи з Notion API.

Як показано на рисунку 4.11 , модель бази даних виглядає та поводиться інакше, ніж звичайна сторінка, яка містить лише блоки елементів, такі як абзаци, списки, стовпці тощо. Бази даних – це більш просунутий тип сторінок, який дозволяє користувачеві використовувати та відображати інформацію більш структуровано. У наведеному вище прикладі стиль дошки дозволяє класифікувати та розділяти статті на основі їхнього статусу: «Опубліковано» (готово до публікації в блозі) або «Чернетка» (все ще в режимі розробки). Тут створено третій стовпець – «Без статусу», але дошка може містити більше або менше стовпців, оскільки її легко налаштувати залежно від потреб.

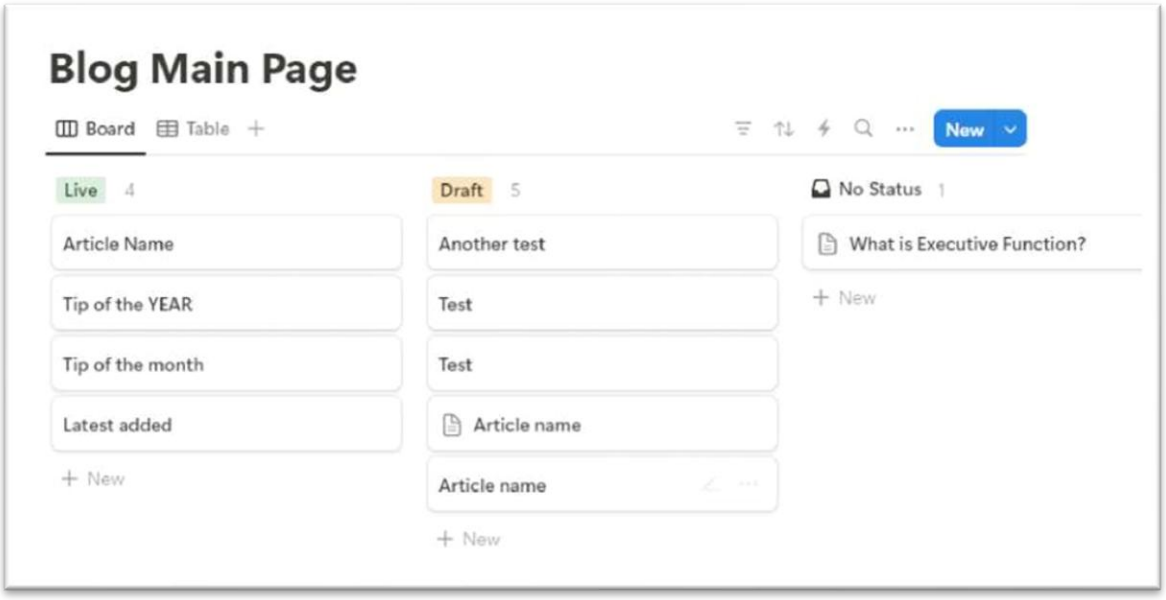

Рисунок 4.11 - Вигляд бази даних з notion.

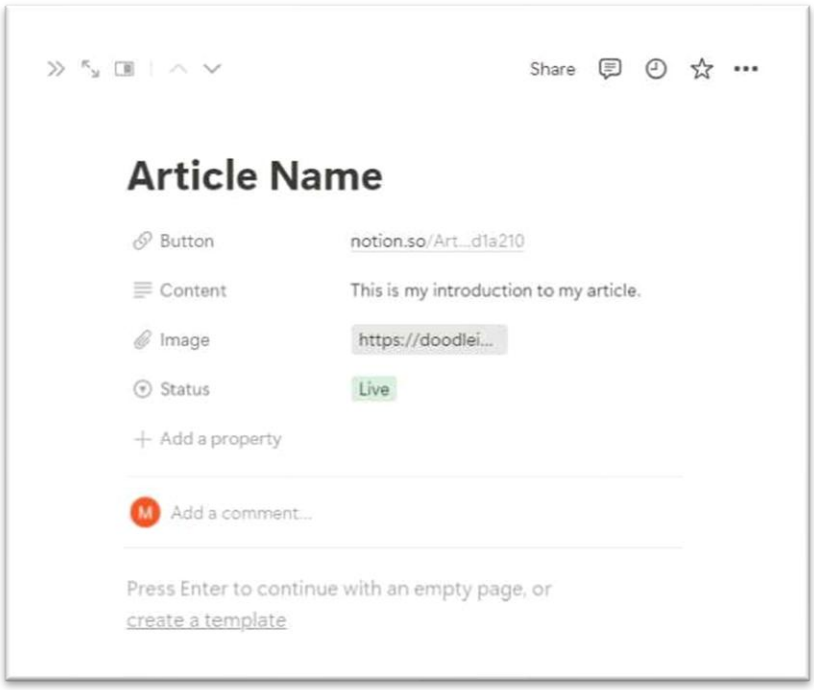

Рисунок 4.12 - Елемент картки з бази даних notion.

Кожна картка зі стовпця містить інформацію, вибрану користувачем (рис. 4.11). У цьому випадку картка містить поле для назви статті, розділ контенту, що дозволяє користувачеві написати короткий вступ, заповнювач для зображення, а також кнопку (у цьому випадку заповнювач посилання) та елемент «Статус», необхідний для класифікації картки на дошці.

Рисунок 4.13 - Стаття, написана на тему notion.

Ці окремі картки також містять слот для написання будь-якого бажаного тексту (це видно на рисунку 4.12, де відображається наступний текст: «Натисніть Enter, щоб продовжити з пустої сторінки або створіть шаблон»), проте API Notion не надає жодної кінцевої точки для отримання цієї конкретної області. З цієї причини було вирішено, що статті будуть написані на окремій звичайній сторінці Notion (рис. 4.13), і що кожна з цих статей буде пов’язана з відповідною карткою блогу, присутньою в Базі даних, шляхом додавання власної URL-адреси до заповнювача кнопки картки. Хоча спочатку ця ідея здавалася непрактичною, на практиці такий спосіб добре себе показав. Цікаво відзначити, що кнопка картки також може містити зовнішнє посилання.

На стороні клієнта логіка навігації така: користувач заходить до Блогу, де відображаються картки (рис 4.14), що містять зображення, заголовок, короткий вступ та кнопку з посиланням (інформація, що міститься в картках (рис. 4.12), присутніх на дошці, показаній на рис. 4.11).

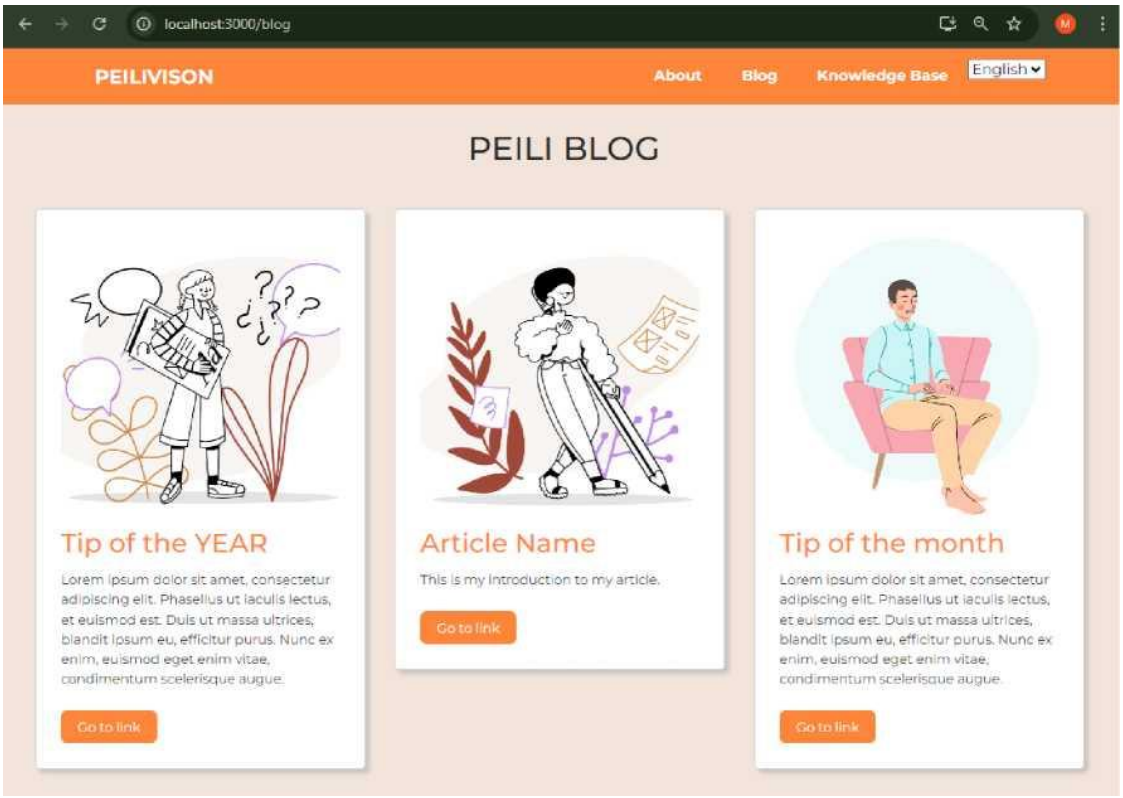

Рисунок 4.14 - Вигляд блогу з застосунку react.

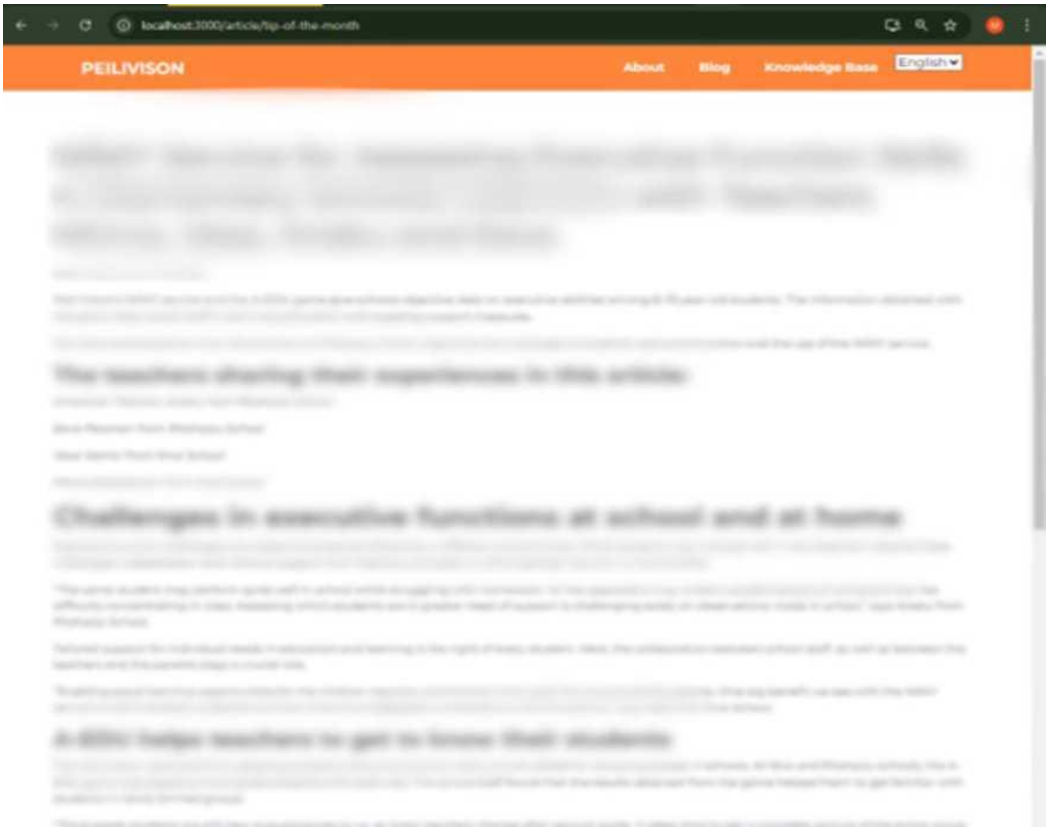

Рисунок 4.15 - Вигляд статті після доступу до неї через картку з головної сторінки блогу.

Після натискання кнопки користувач буде перенаправлено до відповідної статті (рис 4.15 - стаття розміщена в Notion, як показано на рис. 4.13 - або будь-яке інше посилання, присутнє в кнопці. Якщо кнопка не містить посилання, картка взагалі не відображається.

Отримання бази даних

На стороні сервера ця база даних має власну функцію вибірки, яка відрізняється від функції вибірки для отримання звичайної сторінки. Як показано, функція fetching також містить фільтр, який дозволяє React-стороні відображати лише ті елементи, що знаходяться у стовпці "Live".

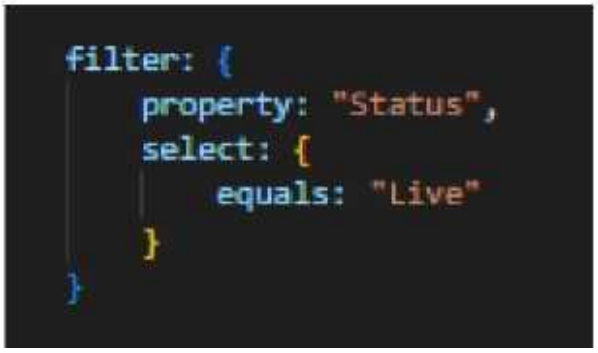

Рисунок 4.16 - Фільтрування для відображення лише елементів у стовпці «активний».

Будь-який елемент, присутній у розділах «Чернетка» або «Без статусу», не зможе відображатися у застосунку React.

Практичне використання React-хуків: useParams, useNavigate та useLocation

Початкова логіка полягала в тому, щоб витягти ідентифікатор сторінки із загальної URL-адреси сторінки статті Notion, використати метод useParams() з компонента React у Biog.js та передати ідентифікатор як властивість компоненту Articie.js , який обробляє рендеринг вмісту статті. Це передбачає невелику модифікацію оригінальної маршрутизації.

На рисунку 4.17 для навігації було додано маршрут /article/ :id . Він складається з динамічного сегмента :id, який є заповнювачем для ідентифікатора

сторінки Notion. Це гарантуватиме, що коли користувач переходить за URL-адресою /articie/exampie_Notion_page_iD , буде відображено правильну статтю. Ідентифікатор доступний як у компонентах Blog, так і в компонентах.

```
<Routes>
    <Route path="/blog" element={<Blog/>} />
    <Route path="/article/:id" element={<Article/>} />
     [...]
</Routes>
```

Рисунок 4.17 - Фрагмент коду маршрутизації.

Оператор if на початку фрагмента коду, наведеного нижче, обробляє випадок порожньої кнопки, яка не містить жодного посилання. Це гарантує, що не буде викинуто жодних помилок, і що користувач зможе використовувати програму в будь-якому випадку: картка, яка не містить посилання, просто не буде відображатися, навіть якщо вона знаходиться в розділі "Live" бази даних. Саме тому в базі даних Notion є чотири картки (рис. 4.11), але в застосунку React відображаються лише три (рис. 4.14).

Як створюється новий об'єкт URL З ВИКОРИСТАННЯМ item.button як значення (item.button – це посилання, присутнє в картці). Якщо ім'я хоста містить "notion", використовується компонент Link (з React Router) для створення маршруту до /articie/${id} . Цей ідентифікатор відокремлюється від URL-адреси, де остання частина, розділена дефісами, витягується за допомогою pop() . Якщо ім'я хоста не містить "notion", то посилання обробляється як звичайне зовнішнє посилання. Отже, коли користувач натискає на кнопку, він або перенаправляється на пов'язану статтю, розміщену на Notion (але відображену в застосунку React), або на кінцеву точку зовнішнього посилання, якщо це так.

Тепер, коли компонент Blog для рендерингу встановлено, час переконатися, що при натисканні на кнопку здійснюється доступ до правильної статті. Як показано, параметр ID витягується з URL-адреси за допомогою гачка

useParams() . Спочатку pageid визначався лише мовою (через параметр країни), а відтепер визначається або параметром ID , або мовою. Гачок useEffect гарантує, що дані будуть отримані щоразу, коли змінюється ID або країна .

Ці елементи дозволяють переходити з картки блогу до відповідної статті. Наприклад, URL-адреса переходить від /blog до /article/4e921be7771c43788b3a7698286da76b. Однак, навіть якщо це поточне рішення працює належним чином, той факт, що публічна URL-адреса складається з ідентифікатора сторінки Notion, створює певні проблеми. В ідеалі кінцева URL-адреса статті мала б відображати назву статті, а не її ідентифікаційний номер. Це також покращило б її з точки зору UX.

Щоб використовувати інший параметр для відображення в кінцевій URL-адресі статті, водночас використовуючи параметр ID, який є важливим для отримання та відображення правильної статті, знадобилися деякі корективи. Першим кроком є зміна маршруту в системі маршрутизації з <Route path="/article/:id" element={<Article/>} /> на "/article/:name" . Це забезпечить кращу зрозумілість. У компоненті Blog введено гачок useNavigate .

Хук імпортується одразу на початку фрагмента за допомогою const navigation = useNavigate() . За допомогою функції handleNavigation обробляється навігація на основі наданого об'єкта item . ІДЕНТИФІКАТОР витягується аналогічно до попереднього, за допомогою pop() . Нова URL-адреса для внутрішньої навігації створюється, а encodeURIComponent допомагає створити дійсний сегмент URL-адреси. item.name також стилізовано під нижній регістр та розділене дефісами. Потім функція викликає navigate зі створеною URL-адресою та об'єктом стану опції . Об'єкт стану містить витягнутий ІДЕНТИФІКАТОР, який буде передано разом з навігацією.

Код перевіряє в операторі return , чи є item.button дійсною URL-адресою, використовуючи isValidUrl . Якщо ні, реєструється помилка, і повертається значення null, що фактично пропускає рендеринг цього елемента. Якщо URL-адреса дійсна, новий елемент div рендериться з унікальним ключовим оператором prop -, який вже присутній в оригінальній версії коду.

Наступна частина оператора досить схожа на оригінальну версію, з невеликою зміною, оскільки тепер елемент button відображається з обробником onClick, встановленим на handleNavigation(item) . Це запускає внутрішню навігацію за допомогою React Router та передає ID у стані. В іншому випадку відображається елемент anchor з атрибутом href , встановленим на item.button . Це створює посилання на зовнішню URL-адресу.

У компоненті Article було використано новий гачок: useLocation . Цей гачок забезпечує доступ до об'єкта стану, переданого з компонента Blog, під час навігації, та витягує ідентифікатор з об'єкта стану. Щоб отримати дані, як можна побачити, хук useParams витягує параметр name з URL-адреси, подібно до того, як він робив це раніше з параметром ID . Потім хук useLocation витягує ID зі стану (const id = location.state?.id). Зрештою, хук useEffect використовує або id , або параметр name (якщо ID недоступний) для отримання даних статті з API. Отримані дані потім використовуються для відображення вмісту статті.

Використовуючи ці перехоплювачі разом, компонент Article може динамічно отримувати та відображати правильну статтю на основі URL-адреси та стану, переданих компонентом Blog. Після цих налаштувань логіка блогу працює належним чином. Щойно користувач опиняється на головній сторінці блогу з URL-адресою /blog , він може натиснути кнопку, щоб отримати доступ до зовнішнього посилання або перенаправитися на потрібну статтю (приклад URL-адреси: /стаття/назва-статті).

4.3 Огляд Notion API

Як і для будь-якого проекту такого типу, після того, як було встановлено основну логіку, були необхідні зміни та вдосконалення. Сам по собі Notion – це чудовий інструмент із простим, але ефективним дизайном, легким для розуміння та використання. Головна мета цієї Бази знань – бути дематеріалізованим посібником для різних сторін, які використовують інструмент Peili Vision. Це означає, що контент також необхідно створювати.

Після заповнення Бази знань фактичним кінцевим контентом, на поверхню випливло кілька проблем, що ще більше розкрило, як керувати Notion API на практиці. Дійсно, якщо Notion дозволяє легко об'єднувати різні блоки шляхом простого перетягування їх на сторінку, то API Notion не такий простий, оскільки він обробляє різні блоки окремо. Наприклад, під час створення на сервері функції для отримання блоку Column з Notion та його відображення на стороні клієнта, вона автоматично відображає лише текстовий вміст. Якщо користувач бажає створити колаж на своїй сторінці Notion із зображенням у лівій частині колонки та заголовком, текстом та маркованим списком у правій частині колонки, кожен із цих блоків також має бути оброблений у функції Column, присутній на сервері.

Приклад показано чудово, оскільки можна побачити, як він перевіряє, чи містить вміст колонки блоки абзаців, заголовки, зображення або блок маркованого списку. Це означає, що якщо в майбутньому колонка має відображати нумерований список або цитату, то до методу processColumnList потрібно буде включити функцію, яка керує цим блоком.

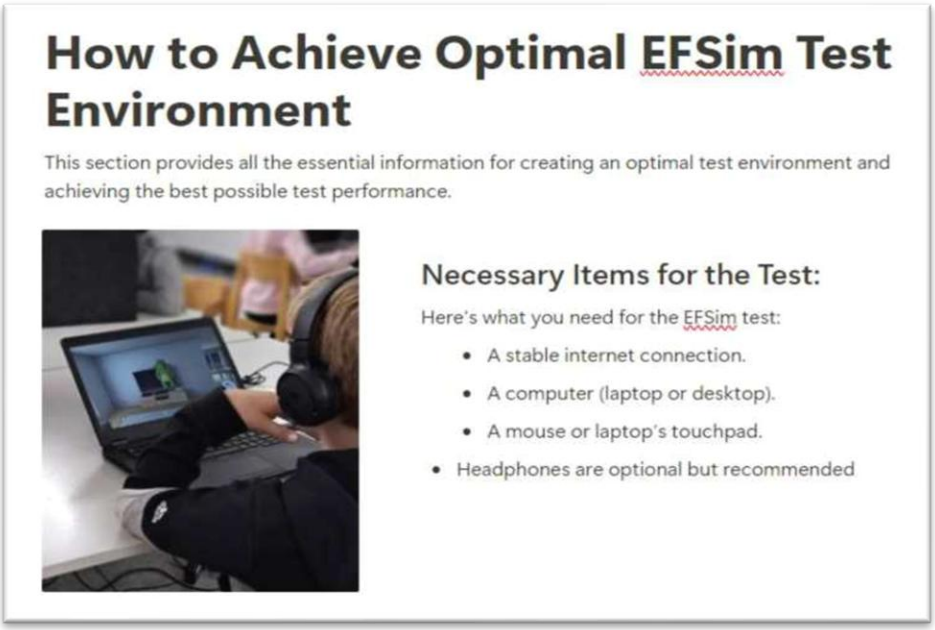

Рисунок 4.18 - Приклад проблеми з відступами в notion, вигляд з notion.

Робота з Notion API також вимагає точності та делікатності з оригінальним форматуванням у Notion. Фактично, коли вперше виникла потреба відобразити маркований список у блоці стовпця, блок маркованого списку не можна було побачити ніде в консолях чи в браузері, тоді як у Notion він був присутній у стовпці. Потім було виявлено, що відступ, який був присутній у маркованому списку (і непомічений спочатку), зробив його нечитабельним для Notion API. Цей відступ, найімовірніше, був спричинений жестом копіювання та вставки з оригінального документа, де маркований список був вперше записаний у текст Notion (рис 4.18). Перші три пункти мають відступ, що запобігає їх видимості, а четвертий і останній пункт містить правильний відступ, що робить його видимим у застосунку React.

Аналогічна проблема виникала також під час відображення на першій підсторінці, коли головна сторінка порожня. Код перевіряє наявність biock.type === 'chiid_page' , і якщо є лише блоки дочірніх сторінок, порожня головна сторінка відображає вміст своєї першої підсторінки. Однак Notion дозволяє порожні пробіли, які вважаються блоками "абзаців" через Notion API.

Тоді користувачеві потрібно бути обережним, маючи справу з нібито порожніми основними сторінками, оскільки будь-який небажаний порожній простір завадить відображенню вмісту першої підсторінки. Дійсно, якщо буде лише один зайвий пробіл, код зчитуватиме сторінку як таку, що не має лише дочірніх сторінок, і нічого не відображатиме, оскільки дочірні сторінки повинні повертати null у звичайному налаштуванні відображення сторінки.

Рисунок 4.19 - Приклад головної сторінки в notion, що містить пробіли.

На рисунку 4.19 показано приклад порожньої головної сторінки, що містить лише блоки підсторінок у Notion, але з деякими білими пробілами. Ці порожні пробіли можна розпізнати за текстом «Напишіть щось...», а також за знаком «+». Це завадить порожній головній сторінці поводитись належним чином у застосунку React.

Рисунок 4.20 - Приклад головної сторінки в notion, що не містить пробілів.

На рисунку 4.20 показано правильний спосіб форматування сторінки Notion, щоб код працював належним чином, тобто відображав вміст першої підсторінки. Документація Notion REST API також у деяких випадках не містить детальної інформації, що дозволяє користувачеві покладатися на метод спроб і помилок. Хоча Notion як веб-інструмент простий у розумінні , універсальний і дозволяє створювати складні архітектури сторінок, його API не завжди такий простий.

Загалом, Notion та Notion API – це хороші інструменти, які дозволяють створювати вичерпні сторінки та інші бази даних. Однак, є деякі обмеження, оскільки для належної роботи Notion API потрібне певне стилізування, що не залишає користувачеві можливості бездумно пропустити будь-яку деталь. Більше того, якщо Notion дозволяє користувачеві досліджувати свою творчість під час оформлення сторінок, це не так просто реалізувати через Notion API, що може створити певні розчарування.

Таким чином, Notion та Notion API – це хороші інструменти, які мають деякі обмеження, що їх необхідно враховувати, і які можуть обмежити їх використання для більших та складніших проектів.

4.4 Висновки по розділу

Розділ присвячений практичній реалізації розробленого програмного рішення. Описано інтеграцію з Notion API для роботи з базами знань, використання об’єктно-орієнтованого програмування для організації структури застосунку, а також техніки оптимізації продуктивності та покращення адаптивності інтерфейсу. Значну увагу приділено динаміці рендерингу сторінок, що забезпечує кращу взаємодію користувача з додатком. Окремим підрозділом представлено реалізацію блогу – включаючи отримання даних з бази, застосування контексту та React-хуків для реалізації навігації й динамічного контенту. У результаті реалізовано гнучку, масштабовану та сучасну веб-систему, що відповідає вимогам продуктивності, зручності та технічної ефективності.

ВИСНОВКИ

База знань була створена для того, щоб забезпечити кінцевим користувачам Peili Vision кращий та легший доступ до банку інформації щодо процесу оцінювання, результатів оцінювання, а також іншої корисної інформації та порад. Раніше цей контент був доступний лише у форматі PDF, що ускладнювало його оновлення. Завдяки всій цій інформації, яка тепер доступна на Notion, набагато легше змінювати, додавати, оновлювати та видаляти контент, а також створювати версії різними мовами за потреби. Більше того, оскільки мовні версії не пов'язані одна з одною, це дає змогу мати дещо відмінний та локалізований контент на основі мов за потреби.

Блог – це додатковий елемент цього порталу, який все ще був частиною проєкту, оскільки його створення було запропоновано, хоча й не було пріоритетним – тому його могли ввести в експлуатацію лише пізніше. Після роботи над цими проектами, База знань була інтегрована в портал та введена в експлуатацію. Контент ще не повністю написаний, і бажано, щоб різні його частини були динамічно доступні різним сторонам (сім'ям, працівникам освіти та охорони здоров'я). Ця логіка не була частиною завдань, які мені було доручено.

Робота над Базою знань та Блогом була дуже цікавим та конструктивним проектом. Notion API з'явився зовсім недавно, як згадувалося раніше, і брак інформації часом був складним завданням, оскільки документація була обмеженою, а допомога від інших розробників також була обмеженою. Під час цього проекту було багато спроб і помилок, але зрештою все було того варте. Також було дуже цікаво мати справу з багатьма різними аспектами фронтенд-розробки, що часто передбачало відкладення звичайного дизайну та CSS-форматування, оскільки кінцевий дизайн не мав бути виконаний мною самостійно. React — дуже цікава та проста у використанні бібліотека, але протягом цього проєкту я неодноразово читав, що іноді стратегічно цікавіше почати з фреймворку, а потім глибше зануритися в проєкт. Одного разу це сталося, коли я читав про способи реалізації та роботи з серверним рендерингом.

Поточний портал — це проєкт на React, який містить Базу знань, яка також є проєктом на React сама по собі. Обговорюючи це з колегами, я дізнався, що впровадження Бази знань на порталі та підключення її до серверної частини Azure також є виснажливим процесом. Тоді було розглянуто можливість використання фреймворку (наприклад, Next.js) у майбутньому.

Зрозуміло, чому було легше створити Базу знань на моєму локальному комп'ютері, а потім інтегрувати її з порталом, оскільки ми не знали, як вона розвиватиметься. Але, озираючись назад, задаюся питанням, чи не було б легше працювати з нею вже з рештою порталу, як ми робили б, наприклад, для інших проектів, що включають елементи, розроблені та створені іншими людьми. Це могло б запобігти проблемам з інтеграцією її з порталом після його готовності та створило б відчуття більш інклюзивної командної роботи.

В іншому випадку, я вважаю, що цей проєкт пройшов досить добре та гладко, і було дуже цікаво працювати з ним. Я здобув багато знань і тепер впевненіше почуваюся у своєму способі пошуку інформації та читання документації. Хоча я стикався з випадками, коли мій спосіб мислення ускладнював мою роботу більше, ніж потрібно, це також був важливий крок до того, щоб навчитися бути простішим та ефективнішим у кодуванні.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Banks, A., & Porcello, E. (2020). *Learning React: Functional web development with React and Redux* (2nd ed.). O'Reilly Media. https://www.oreilly.com/library/view/learning-react/9781492051718/

2. Boduch, A., & Derks, R. (2023). *React and React Native* (4th ed.). Packt Publishing.https://www.packtpub.com/product/react-and-react-native/9781803245683

3. Chinnathambi, K. (2024). *React basics for beginners*. SitePoint. https://www.sitepoint.com/premium/books/react-basics-for-beginners

4. Freeman, A. (2023). *Pro React 18*. Apress. https://www.apress.com/gp/book/9781484296288

5. Gatsby Documentation. (2025). *gatsbyjs.com*. https://www.gatsbyjs.com/docs/

6. Haverbeke, M. (2018). *Eloquent JavaScript: A modern introduction to programming* (3rd ed.). No Starch Press. https://eloquentjavascript.net/

7. Jest Documentation. (2025). *jestjs.io*. https://jestjs.io/docs/getting-started

8. MDN Web Docs. (2025). React and JavaScript best practices. *developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing /Client-side_JavaScript_frameworks/React_getting_started

9. Next.js Documentation. (2025). *nextjs.org*. https://nextjs.org/docs

10. Node.js Documentation. (2025). *nodejs.org*. https://nodejs.org/en/docs/

11. React Documentation. (2025). *react.dev*. https://react.dev/

12. React Router Documentation. (2025). *reactrouter.com*. https://reactrouter.com/en/main

13. Redux Documentation. (2025). *redux.js.org*. https://redux.js.org/

14. Tailwind CSS Documentation. (2025). *tailwindcss.com*. https://tailwindcss.com/docs

15. Testing Library Documentation. (2025). *testing-library.com*. https://testing-library.com/docs/react-testing-library/intro/

16. Wieruch, R. (2024). *The road to React: Your journey to master React.js*. Leanpub. https://www.roadtoreact.com/

17. Wilson, C. (2023). *Modern web development with React and TypeScript*. Packt Publishing. https://www.packtpub.com/product/modern-web-development-with-react-and-typescript/9781803240879

18. Bhatt, P., & Sharma, S. (2024). Scalable blog architectures with Next.js and React. *Journal of Web Engineering*, 23(2), 189–204. https://doi.org/10.13052/jwe1540-9589.2324

19. Chen, J. (2023). Server-side rendering with React: Performance optimization for blogs. *ACM Transactions on Web*, 17(3), 1–18. https://doi.org/10.1145/3591234

20. Gupta, R., & Kumar, A. (2024). Headless CMS integration with React for public blogs. *IEEE Transactions on Software Engineering*, 50(5), 1123–1135. https://doi.org/10.1109/TSE.2024.3367890

21. Khan, M. (2025). SEO optimization for React-based blogs using Gatsby. *Web Development Journal*,12(1), 45–60. https://doi.org/10.1007/s42979-024-02345-6

22. Amazon Web Services. (2025). Hosting React applications with AWS Amplify. *aws.amazon.com*. https://aws.amazon.com/amplify/

23. Contentful. (2024). Building blogs with Contentful and React. *contentful.com*. https://www.contentful.com/developers/docs/react/

24. Google. (2025). Web performance fundamentals for React applications. *web.dev*. https://web.dev/react/

25. GraphQL. (2025). GraphQL for React blog development. *graphql.org*. https://graphql.org/learn/

26. Netlify. (2024). Deploying React blogs with Netlify. *netlify.com*. https://www.netlify.com/platform/react/

27. Strapi. (2025). Building APIs for React blogs with Strapi. *strapi.io*. https://docs.strapi.io/developer-docs/latest/developer-resources.html

28. Vercel. (2025). Deploying Next.js blogs with Vercel. *vercel.com*. https://vercel.com/docs/frameworks/nextjs

29. Osmani, A. (2023). *Learning JavaScript design patterns* (2nd ed.). O'Reilly Media. https://www.oreilly.com/library/view/learning-javascript-design/9781098139865/

30. Zakas, N. C. (2016). *Understanding ECMAScript 6: The definitive guide for JavaScript developers*. No Starch Press. https://leanpub.com/understandinges6

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: «Розробка public-блогу засобами React».

Обсяг пояснювальної записки: 70 аркуш

Дата закінчення дипломної роботи «10» червня 2025р.

Підпис студента _____