СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

Пояснювальна записка

до магістерської дипломної роботи

магістр

(освітньо-кваліфікаційний рівень)

на тему Дослідження методів та засобів підвищення продуктивності клієнтської частини інформаційної системи на базі фреймворку React

Виконав: студент 2 курсу, групи ІСТ-24дм

126 «Інформаційні системи та технології»

(шифр і назва спеціальності)

Вертаєв Є.Є.

(прізвище та ініціали)

Керівник Меняйленко О.С.

(прізвище та ініціали)

Рецензент Ратов Д.В.

(прізвище та ініціали)

Київ – 2025 року

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ ДО МАГІСТЕРСЬКОЇ ДИПЛОМНОЇ РОБОТИ

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

Освітньо-кваліфікаційний рівень магістр

Спеціальність 126 «Інформаційні системи та технології
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТП,
_____д.т.н., проф.Захожай О.І.
(підпис)
«____»_____2025 р.

ЗАВДАННЯ

на магістерську дипломну роботу студенту

Вертаєв Єгор Єгорович
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів та засобів підвищення продуктивності клієнтської частини інформаційної системи на базі фреймворку React
керівник роботи професор, д.т.н. Меняйлено Олександр Сергійович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом університету від « 28 » 11 2025 року №241/17.03
2. Строк подання студентом роботи: 18 грудня 2025р.
3. Вихідні дані до роботи Матеріали науково-дослідницької практики, науково-методична література; дані інтернет-мережі
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1 Вступ

4.2 Аналітичний огляд розробки користувацьких інтерфейсів для інформаційних систем на базі React.

4.3 Аналіз основних методів оптимізації інтерфейсу.

4.4 Аналіз результатів впровадження методів оптимізації інтерфейсу.

4.4 Висновки

4.5 Перелік використаних джерел

5. Перелік графічного матеріалу (з точним значенням обов'язків креслень)_____

_____

_____

_____

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання_____07. 11. 2025р.___

КАЛЕНДАРНИЙ ПЛАН

№ з\п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Одержання завдання на виконання роботи	07.11.2025	виконано
2	Укладання і погодження з керівником плану і етапів виконання роботи	10.11.2025	виконано
3	Узагальнення даних літературних джерел	15.11.2025	виконано
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху виконання завдання	18.11.2025	виконано
5	Аналіз методів та засобів оптимізації	22.11.2025	виконано

6	Реалізація методів та засобів оптимізації та практичної частини завдання	07.12.2025	виконано
7	Укладання, оформлення та погодження пояснювальної записки з керівником	17.12.2025	виконано
8	Надання пояснювальної записки на кафедру	18.12.2025	виконано
9	Підготовка доповіді та презентації	21.12.2025	виконано

Студент _____ Вертаєв Є.Є.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Меняйленко О.С.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Магістерська дипломна робота: 82 стор., 22 рис., 27 джерел.

Об’єкт дослідження – методи та засоби оптимізації клієнтської частини інформаційних систем на базі React.

Мета роботи – дослідження методів оптимізації користувацького інтерфейсу шляхом впровадження та комбінування найбільш ефективних засобів та методів.

Наукова новизна заключається у дослідженні засобів та методів оптимізації інтерфейсів інформаційних систем розроблених на базі React та проведення глибокого аналізу ефективності їх застосування

Розглянуто проблеми продуктивності, визначені ключові аспекти, які впливають на продуктивність React та проаналізовано особливості проблем зі швидкодією роботи клієнтської частини, методи та засоби підвищення продуктивності клієнтської частини інформаційної системи.

Отримані результати можуть бути застосовані для оптимізації та покращення користувацького інтерфейсу, який зіштовхнувся з проблемами продуктивності.

КЛІЄНТСЬКИЙ ІНТЕРФЕЙС, ПРОДУКТИВНІСТЬ, ОПТИМІЗАЦІЯ, ФРЕЙМВОРК, РОЗРОБКА, REACT, ПРОФАЙЛІНГ, МЕМОІЗАЦІЯ

ABSTRACT

Master`s thesis: 82 pages, 22 pictures, 27 sources.

The object of the research is methods and tools for optimizing the client part of information systems based on React.

The purpose of the work is to study methods for optimizing the user interface by implementing and combining the most effective tools and methods.

The scientific novelty lies in the study of tools and methods for optimizing the interfaces of information systems developed on the basis of React and conducting an in-depth analysis of the effectiveness of their application.

Performance problems are considered, key aspects that affect the performance of React are identified and the features of problems with the speed of the client part are analyzed, methods and tools for increasing the performance of the client part of the information system.

The results obtained can be used to optimize and improve the user interface that has encountered performance problems.

CLIENT INTERFACE, PRODUCTIVITY, OPTIMIZATION, FRAMEWORK, DEVELOPMENT, REACT, PROFILING, MEMOIZATION

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

UI – клієнтський інтерфейс;

UX – користувацький досвід;

HTTP – протокол передачі гіпертекстових документів;

CRM – система управління взаємовідносинами з клієнтами;

SPA – односторінковий інтерфейс;

MPA – багатосторінковий інтерфейс;

SSR – рендеринг на стороні сервера;

SSG – статична генерація;

SEO – пошукова оптимізація;

CMS – система керування контентом;

API – програмний інтерфейс.

ЗМІСТ

ВСТУП

Швидкий розвиток технологій призвів до появи великої кількості інформаційних систем різного призначення, клієнтська частина побудована на сучасних фреймворках, яких теж велика кількість та кожні з них мають свої переваги та недоліки.

На сьогодні одним з найпопулярніших фреймворків для створення користувацьких веб-інтерфейсів великих інформаційних систем є React за статистичними даними stateofjs.com [1], створений компанією Meta (раніше Facebook).

Незважаючи на свою історію та назву, React не є реактивним, що на перший погляд може здаватися недоліком, однак це є навмисний архітектурний вибір, який надає багато переваг для розробки стабільних та зрозумілих інтерфейсів для інформаційних систем.

Популярність цього фреймворку зумовлена багатьма перевагами, такими як:

- Компонентний підхід – створення інтерфейсів шляхом декомпозиції елементів, які можна використовувати повторно;
- Віртуальний DOM – який ефективно оновлює інтерфейс завдяки не складному алгоритму, та механізму React Reconciliation;
- Гнучкість та універсальність – надає повну свободу вибору інструментів;
- Екосистема та спільнота – велика популярність означає велику кількість готових рішень практично для будь-яких потреб.

Але наявність всіх цих переваг не дає автоматичної гарантії, що інформаційна система буде мати бажану швидкість робочих процесів та зручність користування. Фреймворк надає лише інструменти, але доцільність та ефективне використання повністю лежить на розробнику або команді. Незалежно від того, як добре спроєктований користувацький інтерфейс на початку, з розвитком інформаційної системи неминуче відбувається масштабування та «Роздування» логіки на клієнтській частині. Це призводить до втрати продуктивності, сповільнення

інтерфейсу, високого споживання ресурсів пристрою на якому працює користувач та підвищує відчуття ненадійності таких систем. Що спричиняє появу негативного досвіду користувача або навіть його втрати, що в умовах постійної конкуренції в сфері цифрових продуктів є критичним.

Сьогодні користувач має нульовий рівень терпимості до повільних, нестабільних інтерфейсів з високою затримкою відгуку.

Зазначена проблема виражає важливість підвищення продуктивності й оптимізації інформаційних систем на стороні клієнта, підкреслює важливість цих методів і засобів для забезпечення ефективності й надійності інформаційної системи в умовах перенасиченого ринку.

Для досягнення високих показників продуктивності фреймворк React має достатню кількість інструментів та технік, таких як: інструменти оптимізації рендерингу, віртуалізація, селектори, композиція і т.д.. Тому розробники мають володіти необхідними знаннями й навичками для якісного застосування та імплементації цих технік.

Об’єктом дослідження є методи та засоби оптимізації клієнтської частини інформаційних систем на базі React.

Предметом дослідження є аналіз сучасних засобів оптимізації фреймворку React.

Метою роботи є дослідження методів оптимізації та виявлення найбільш ефективних.

Для реалізації цієї мети необхідно виконати наступні завдання:

1) Розглянути проблеми продуктивності, визначення ключових аспектів, які впливають на продуктивність React;

2) Проаналізувати особливості появи проблем зі швидкодією роботи клієнтської частини;

3) Проаналізувати методи і засоби підвищення продуктивності клієнтської частини інформаційної системи, зазначити їх переваги та недоліки;

Дана робота присвячена питанням дослідження та застосування методів оптимізації клієнтської частини інформаційної системи, що забезпечують надійну і швидку роботу системи.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Дослідження терміну клієнтський інтерфейс

Термін інтерфейс у контексті програмної інженерії традиційно розглядається як набір засобів, методів і правил взаємодії між компонентами системи [2].

Користувацький інтерфейс (User Interface - UI) – визначено як сукупність методів, засобів та елементів, як візуальних так і звукових, які утворюють точку взаємодії користувача з інформаційною системою.

Це не просто про зовнішній вигляд. Основною метою користувацького інтерфейсу є забезпечити інтуїтивну, приємну і ефективну взаємодію між людиною та системою. В області клієнтської частини інтерфейс є інтерактивною та візуальною складовою, яка виконується в браузерному середовищі і працює в реальному часі. Інтерфейс може виконувати свою роботу на різних операційних системах і різних апаратних платформах, що дозволяє забезпечити однаковий досвід користування.

Спершу розберемось, що таке користувацький інтерфейс, який лежить в основі клієнтської частини та з яких аспектів він складається.

На сьогодні ми маємо велику кількість інформаційних систем які є веб-орієнтовані і мають дуже великі користувацькі інтерфейси, які працюють в браузерах користувачів на різних платформах та пристроях. Прикладом для великих користувацьких інтерфейсів інформаційних систем можуть бути інтернет-магазини, інтерфейси для банківських та фінансових систем, платформи для керування взаєминами з клієнтами (CRM). Ще одним цікавим прикладом виступають соціальні мережі, які включають в себе дуже обширний функціонал.

Супутнім аспектом користувацького інтерфейсу є користувацький досвід який відповідає за архітектуру вражень від взаємодії з інформаційною системою.

Користувацький досвід (User Experience - UX) – це сукупність вражень, сприйняття емоцій, відгуки кінцевого користувача, що виникають в процесі взаємодії та використання інтерфейсу інформаційної системи. Якщо

користувацький інтерфейс відповідає за зовнішній вигляд, то користувацький досвід відповідає за сприйняття та зручність такої системи.

Основні риси користувацького інтерфейсу можна поділити на декілька складових:

- Візуальна складова (дизайн) – включає розташування елементів, кольорову гаму, типографіку, іконки, графіку, а також загальну стилістику додатка, яка формує перше враження користувача та впливає на сприйняття продукту;
- Інтерактивна складова (поведінка) – відповідає за те, як елементи будуть реагувати при взаємодії з інтерфейсом: кліки, наведення курсора, введення даних, тощо;
- Інформаційна архітектура – відповідає за структуру даних і навігацію. Вона допомагає користувачу швидко знайти потрібний йому блок інформації або функціонал;
- Юзабіліті (зручність використання) – це показник того, наскільки інтерфейс інтуїтивно зрозумілий і простий у використанні;
- Адаптивність і доступність – інтерфейс має коректно відображатись на різних пристроях (комп'ютерах, планшетах, смартфонах) та бути доступним для користувачів із обмеженими можливостями. Наприклад, через підтримку клавіатурної навігації чи екранних рідерів.

Сьогодні користувацький інтерфейс складно розглядати ізольовано, він є інтегральною частиною архітектури інформаційної системи та має забезпечувати стабільну роботу в умовах різної продуктивності апаратних платформ, якості мережі, браузерів, технічних характеристик платформи. Проєктування сучасного користувацького інтерфейсу потребує забезпечення сумісності між різними платформами, швидкодії, масштабованості, доступності й енергоефективності [3].

1.1.1 Еволюція клієнтських інтерфейсів та основні поняття

Еволюція інтерфейсів пройшла великий шлях до сьогодні. Якщо раніше це були прості статичні сторінки, які назвати інтерфейсами складно, так як взаємодії

з ними практично не було, то зараз це дуже складні, інтерактивні та високо динамічні системи, які забезпечують не лише відображення, але й виконання бізнес-логіки, персоналізацію та механізми оптимізації. Для зручності розуміння ми можемо розбити еволюцію інтерфейсів на чотири етапи своєї трансформації щоб простежити, які зміни відбувались у взаємодії між користувачем та інтерфейсом, від звичайного відображення до повноцінної взаємодії.

Першим етапом розвитку на період 1990-2000 років була ера статичних сторінок, які слугували засобом навігації та передачі інформації, маючи лише базові набори інструментів, які надавав HTML – мова розмітки. А вся основна логіка обробки даних виконувалась на серверах. Навіть найелементарніший інтерактив обмежувався можливостями HTML. Кожна подія створена користувачем, а саме перехід за посиланням або відправка форми, надсилала запит до серверу, який в свою чергу генерував новий HTML документ та запускав повне оновлення сторінки інтерфейсу, що призводило до великих затримок, а в умовах обмеженого інтернету за трафіком і швидкістю робило взаємодію ресурсозатратною. На цьому етапі з’явилось поняття архітектури «тонкого клієнту», де клієнтом виступав браузер, який відображав статичні сторінки без розрахунків та автономних процесів [4].

Другим етапом вважають період 2000-2010 років, коли в клієнтську частину інтерфейсу почала додаватись динаміка та маніпуляції з статичним документом HTML завдяки інтеграції нових інструментів CSS та мови програмування JavaScript, які надали можливість взаємодії зі структурою сторінки без перезавантажень та динамічно змінювати зовнішній вигляд елементів інтерфейсу. Цей етап став переломним для перетворення інтерфейсу з статичного документа в інтерактивний застосунок і спричинив появу багатьох технологій та бібліотек. Однією з таких переломних технологій став AJAX (Asynchronous JavaScript and XML), який дозволяв у фоновому режимі отримувати та відправляти дані без повного перезавантаження інтерфейсу та став де-факто стандартом у розробці користувацького інтерфейсу. В цей же час паралельно стрімкий розвиток мав CSS, який відповідав за стилі інтерфейсу. Ця технологія почала задіювати складні

методи позиціонування, концепції і впроваджувати надбудови у вигляді препроцесорів Sass та Less для масштабування і підтримки стилів великих інтерфейсів.

Третій етап 2010-2015 років являється ерою SPA – односторінкових додатків та фреймворків, які мали за мету подолати хаос в розробці й масштабуванні великих клієнтських інтерфейсах. В цей період з'явились три повноцінні клієнтські фреймворки такі як: AngularJS в 2010р., React JS в 2013р. та Vue.js в 2014р.. Саме вони започаткували базові принципи та архітектури, на яких мають будуватись масштабовані інтерфейси:

- компонентність – розподіл елементів на незалежні UI-модулі;
- віртуальне дерево – мінімізація прямої взаємодії з документом;
- односторонній потік даних – спрощення передбачуваності стану;
- інкапсуляція стилів та логіки.

З цими змінами зрушився фокус проблем. Якщо раніше основними проблемами були технічні обмеження, які не давали можливості робити динамічні інтерфейси без перезавантажень та великих ресурсозатрат, то тепер головними викликами стали побудова надійних інтерфейсів і керування складністю динаміки елементів.

У великій інформаційній системі існують сотні, а то і тисячі елементів інтерфейсу різного типу та характеру для виконання різних задач. Всі вони мають мати свій стан і логіку для правильного відображення та реакції на дії користувача. Але керувати цими станами і логікою на пряму в документі HTML стало дуже затратно по ресурсам апаратної частини та браузера, на яких відбувається запуск такого інтерфейсу. Тому постали питання не як зробити, а як структурувати такі інтерфейси для подальшого масштабування. Це призвело до появи спеціалізованих бібліотек та архітектурних патернів, які дозволили будувати інтерфейси масштабу Facebook, Netflix, Airbnb та залучати до створення сотні розробників, які могли будувати інтерфейси такого масштабу без хаосу. Важливо підкреслити, що в цей етап фреймворки принесли нові можливості як технічні так і архітектурні. Але, що

важливіше, принесло інженерну дисципліну, перетворивши розробку інтерфейсів з мистецтва на інжерні рішення, які потребують глибокого розуміння тематики.

Четвертий етап ми спостерігаємо з 2015 року і по сьогодні. Індустрія розробки користувацьких інтерфейсів отримала великий поштовх та стрімкий розвиток більш складних архітектур і підходів. Основним поштовхом було запровадження іншого підходу до роботи з інтерфейсом – віртуальний DOM. Який забезпечує оптимізовану роботу інтерфейсу завдяки меншій кількості звернень на зміни структури та стану реального документу HTML, та забезпечує високу швидкість повторної побудови інтерфейсу в тих місцях, де відбулись зміни, а не побудови всього інтерфейсу знову. Важливими поштовхами були ще створення дуже великої кількості спеціалізованих інструментів та бібліотек направлених на подолання архітектурних обмежень за допомогою формування нових архітектурних моделей і надбудов, які переймають парадигми з ООП та інших напрямів технологічних індустрій [5].

Паралельно розвитку всіх технологій розробки поставало багато нових питань і проблем, які потребували рішень. Основними проблемами були масштабування коду та неоптимізовані рішення, в наслідок яких, виросла потреба в рефакторингу та оптимізації клієнтських інтерфейсів, щоб зберегти конкурентоздатність інформаційної системи на ринку.

1.1.2 Архітектурні підходи до побудови клієнтських інтерфейсів

Інформаційна система, інтерфейс якої розроблений на базі сучасних фреймворків, використовує клієнт серверну архітектуру, де клієнтом, як правило, виступає браузер, а сервером – веб-сервер [6]. Логіка роботи таких систем розділена між клієнтом та сервером, як правило сервер відповідає за збереження основних масивів даних, а клієнт за його відображення. Обмін інформацією між ними відбувається мережею Інтернет завдяки постійному обміну запитами та відповідями.

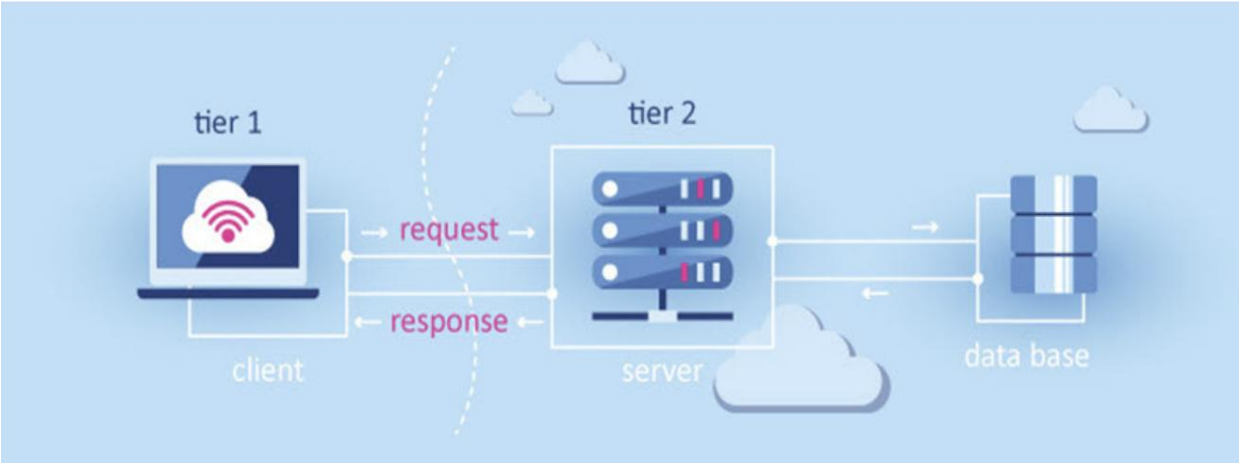

Рис.1.1 – Клієнт-серверна архітектура

Серверна частина функціонує непомітно для кінцевого користувача інтерфейсу і реагує тільки на HTPP-запити, на конкретні URL-адреси, на які відповідає відповідними даними у форматі JSON або HTML. Для розробки серверної частини використовують такі мови програмування та фреймворки: Java (Spring Boot), Python (Django, Flask), JavaScript/TypeScript (Node.js з Express.js, NestJS), PHP (Laravel, Symfony), Ruby (Ruby on Rails), Go (Gin), C# (.NET, ASP.NET Core).

Клієнтська частина коду виконується прямо в браузері й відповідає за інтерактивність, реагуючи в реальному часі на дії користувача. На відміну від захищених серверних файлів, доступних лише через HTTP-запити, код на стороні клієнта є прозорим – його можна вільно переглядати, аналізувати та навіть модифікувати за допомогою інструментів розробника. Основа клієнтської розробки – це тріада веб-технологій: HTML, CSS та JavaScript.

Якщо ж говорити за архітектури самих інтерфейсів ізольовано від інших частин інформаційної системи, то можемо виділити основні сім архітектур: MPA, SPA, SSR, SSG, ISR, CSR, Hybrid Rendering.

SPA (single-page application) – архітектура розробки інтерфейсу, у якій кінцевий користувач отримує з серверу одну сторінку і в подальшому вся робота відбувається на клієнтській частині в браузері за допомогою JavaScript. У такій

архітектурі немає під вантаження кожен раз нового HTML документу, так як всі необхідні файли завантажуються відразу.

MPA (multi-page application) – архітектура розробки інтерфейсу, у якій кожна сторінка інтерфейсу є окремим файлом на серверній частині та завантажується тільки при необхідності. З переваг можна підкреслити можливість для SEO оптимізації, легкість початкового бандлу та плавне масштабування без потреби критичних оптимізації.

SSR (Server-Side Rendering) – архітектурний підхід, який працює за принципом генерації HTML документів на сервері, а потім гідрує його на стороні клієнта. У сучасних фреймворках, таких як next.js, сервер та клієнтська частина може лежати в одному місці та запускатись одночасно у браузері.

CSR (Client-Side Rendering) – архітектурний підхід, який працює в переважній більшості SPA. Все наповнення інтерфесу генерується на клієнті. З переваг можна підкреслити мінімальне навантаження серверу, та гнучкість.

SSG (Static Site Generation) – архітектурний підхід, який генерує HTML документи під час збірки, а не під час запиту до серверу. Фактично не використовує в більшості випадків сервер, що надає перевагу у швидкості та безпеці. Великий недолік такого підходу полягає в тому, що якщо треба часто змінювати наповнення інтерфейсу, то необхідно перезапускати збірку.

ISR (Incremental Static Regeneration) – архітектурний підхід, який комбінує SSR та SSG. В цьому підході інтерфейс статичний, але перегенерація відбувається у фоні. З переваг підкреслимо швидкість та підтримку SEO.

Hybrid Rendering – архітектурний підхід, який комбінує CSR, SSR, SSG, ISR в одному проекті у сучасних фреймворках.

1.1.3 Вимоги до сучасного клієнтського інтерфейсу

Забезпечення дійсно продуктивного та конкурентоспроможного користувацького інтерфейсу зумовлює формування комплексу вимог до нього, що охоплюють продуктивність, адаптивність, масштабованість, безпеку, доступність і

якість відгуку з кінцевим користувачем. Сучасні інтерфейси повинні бути зрозумілими та передбачуваними, швидкими і стабільними. Саме ці основні пункти формують первинне почуття та підкреслюють ефективність взаємодії з інформаційною системою.

Швидкість та продуктивність інтерфейсу є ключовою вимогою за оцінкою Google Web.dev. Затримки більше 200мс у взаємодії з системою, приймаються користувачем як «гальмування» та не плавність користувацького інтерфейсу, а затримка в більш ніж 800мс або 1 секунду, спонукають втраті концентрації та призводять до когнітивного дисонансу [7].

Адаптивність є не менш важливою вимогою, так як інтерфейс має відображатись та працювати коректно на різних апаратних пристроях і браузерах, зберігаючи основну концепцію та головні риси інформаційної системи. Більша частина трафіку в інтернеті на сьогодні припадає на мобільні пристрої, а будувати цілий додаток на смартфон або планшет, який буде окремим проєктом може бути дорого. Тому закладення адаптивності на початку, або реалізація вже в процесі – є наразі необхідністю.

Масштабованість та підтримка в інтерфейсах – це більш про закулісну роботу інформаційної системи, яка кінцевому користувачу не помітна, але є не менш важливою. Зростання кодової бази та роздування, призводять до підвищення складності розвитку інтерфейсів, а також їх покращень. Кожне нове рішення може ламати інші модулі та елементи, або взагалі призводити до непередбачуваної поведінки. Розробка має дозволяти легко вносити зміни, додавати нові можливості і проводити тестування, не ламаючи вже працююче[8].

Безпека клієнтської частини, яка працює з чутливими даними своїх клієнтів, або інформаційної системи, що має доступ до конфіденційних даних підприємств, є на першому місці. Інтерфейс має бути захищеним від зовнішнього впливу, різного роду атак та ін’єкцій, які б могли спричинити витоки даних. Актуальними ризиками є XSS, CSRF, Clickjacking, проблеми зі сторонніми компонентами та небезпечне зберігання автентифікаційних даних [9].

Доступність передбачає створення інклюзивних інтерфейсів. Відповідно до сучасних критеріїв WCAG 2.1, інтерфейс має забезпечувати користування особами з обмеженими можливостями, зокрема тими, хто використовує скрінрідери або альтернативні засоби введення та забезпечуючи повноцінну взаємодію для людей з обмеженнями за зором, слухом або руховими функціями.

1.2 Інструменти для розробки клієнтських інтерфейсів

Створення сучасного користувацького інтерфейсу базується на комплексному технологічному стеку, фундамент якого утворюють три базові складові. Вони посилено відповідають за структуру документу HTML, представлення, візуальних стилів CSS та інтерактивну логіку скриптів JavaScript і нерозривно поєднані між собою. Сучасні фреймворки і бібліотеки розширюють можливості цієї основи, що є стандартом індустрії. Але на кінцевій точці для подання клієнту самого інтерфейсу браузер використовує ці три технології.

Однак, розробка на нативних інструментах, таких як HTML, CSS та JavaScript для складних додатків стає громіздкою. Саме тут на допомогу приходять сучасні фреймворки та бібліотеки, такі як React, Vue.js або Angular. Вони не замінюють базові складові, так звані «три кити» в індустрії розробки, а служать як високорівневі інструментами для їхньої ефективної організації. Весь сучасний код, який створений за допомогою різних бібліотек і фреймворків, транспілюється, оптимізується та компілюється спеціальними рушіями браузера у ту саму тріаду — зрозумілі йому HTML, CSS та JavaScript.

1.2.1 HTML - HyperText Markup Language

HTML - це не мова програмування, а одна з мов розмітки гіпертексту, яка описує, як програміст використовує код для позначення тексту, розділяючи їх на блоки [10]. Мови розмітки HTML і XML, різняться від машинних мов, які мають за собою шістнадцятковий або двійковий код. HTML, представляє собою набір

декларативних інструкцій, що використовується для формування користувацьких інтерфейсів у інформаційних систем, які є веб-орієнтовані та виконують свою роботу в браузері.

Початковий розвиток HTML охоплює кінець 1980-х років та початок 1990-х років, коли ця мова дозволила розробникам вказувати яка мала бути структура документу та її вміст на сторінці інтерфейсу, як відображати елементи, зображення, форми. Переважно під час розробки інтерфейсу дуже рідко використовується тільки HTML. Його використовують в комплексі з CSS і JavaScript, щоб створювати інтерфейси більш розумними та інтерактивними, відображення яких можна отримати через програми, такі як Chrome, Opera, Firefox, Edge та Safari.

Відповідність стандартам HTML і CSS забезпечується World Wide Web Consortium (W3C). Беручи до уваги те, що HTML має фундамент на правилах SGML (яка є стандартною загальною мовою розмітки), а XHTML запозичив правила XML, який є суворим підрозділом SGML [11]. Документи XHTML повинні мати аналогічну інфраструктуру, яка подібна до XML документів, що забезпечує можливість їх запуску за допомогою нативних інструментів обробки XML документів.

HTML зчитується браузером та перетворює його на зручний документ доступний для розуміння людині. Він являється програмою SGML та відповідність до міжнародного стандарту ISO8879.

На сьогодні, HTML 5 – найбільш поширений стандарт для розмітки користувацького інтерфейсу, який в нативному варіанті без надбудов дозволяє інтегрувати мультимедійні елементи, такі як аудіо чи відео, API для роботи з графічними елементами типу canvas, SVG. HTML 5 – є основою для створення скелету інтерфейсу, а інші елементи, такі як стилі та інтерактивні можливості, додаються за допомогою інструментів розробки CSS, PHP та JavaScript [10].

Отже, освоєння HTML – це базовий крок, який необхідний для вивчення інших мов. HTML - основний будівельний блок інтерфейсу, і його задіюють для розробки у сучасних сервісах типу CMS, таких як WordPress, Wiki, Webli [11]. Елементи HTML відрізняються «тегами», які мають специфічний синтаксис у

вигляді символів «<» і «>», які окреслюють початок так кінець тої чи іншої секції чи блоку.

Навчання HTML є важливим етапом для початківців у програмуванні, який прокладає шлях до вивчення більш складних частин розробки інтерфейсів, таких як CSS та JavaScript. Опанування основ HTML має велику користь для майбутніх спеціалістів у розробці систем та тих, хто цікавиться виключно напрямом створенням інтерфейсів.

1.2.2 CCS - Cascading Style Sheets

CSS – це мова каскадних таблиць, основною властивістю якої є оформлення HTML документів. Вона дозволяє задати опис зовнішнього вигляду та надає механізм визначення які стилі будуть мати пріоритет у їх застосуванні.

Основна концепція CSS була запропонована Хоконом Віумом Лі у 1994 році, а вже в кінці 1996 року WSC опублікувала специфікацію для цієї мови [12].

CSS дозволяє під час розробки інтерфейсу змінювати візуальну частину інтерфейсу, використовуючи розподілені частини коду у вигляді одного або декількох файлів стилів. CSS має можливості контролювати шрифти, розміри та кольори тексту, розміри та розташування блоків які були розроблені в HTML. Основна концепція є каскадність, де з одного файлу стилів можуть бути задані інструкції, які будуть виконуватись на всіх сторінках інтерфейсу, що прискорює розробку та полегшує внесення покращень або виправлень до зовнішнього вигляду всіх цих сторінок одночасно.

Каскадність має в основі три принципи: спадкування, порядок оголошення та специфічність. Це надає розробнику інтерфейсів повний контроль над застосуванням стилів то певних елементів без лишніх дублюванню коду [12].

До основних переваг можна віднести:

- швидкодія та ефективність: один файл стилів може мати вплив на керування безлічі сторінок, зменшуючи надмірність кодової бази та прискорюючи завантаження;

- простота обслуговування: при правильному проєктуванні зміни вносяться шляхом редагування одного файлу або модулю після чого оновлюються всі сторінки інтерфейсу;
- можливості оформлення: CSS надає значно більший спектр властивостей для управління зовнішнім виглядом, порівняно з чистим HTML, дозволяючи створювати складні та інтерактивні елементи інтерфейсу;
- кросплатформеність: з використанням одного файлу таблиць стилів, завдяки медіа-запитам, один і той же документ HTML можна представити по різному на різних типах пристроїв.

1.2.3 JavaScript

JavaScript – це мова програмування сценаріїв для створення скриптів, які в свою чергу вносять інтерактивну складову до інтерфейсу, вона додає динаміку поведінки та логіку, перетворюючи статичний HTML на інтерактивний користувацький інтерфейс в браузері.

Згідно з даними проведених опитувань, в наш час JavaScript є фактично найбільш популярною мовою у світі після TypeScript [13]. Але якщо взяти до уваги те, що TypeScript це надбудова над JavaScript, яка доповнює її статичною типізацією, то можна точно сказати, що вона є найбільш використовуваною. Розростання JavaScript в сфері розробки користувацьких інтерфейсів та інших галузях значно перевищує застосування інших мов програмування, що підкреслює її визнання та значимість в світі технологій.

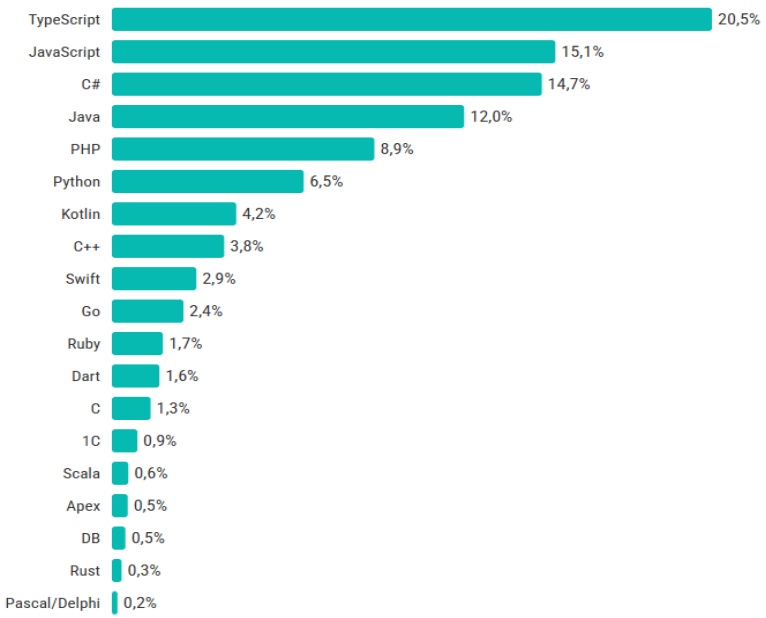

Рис.1.2 – Статистика використання мов програмування на 2025

Сьогодні JavaScript вийшов далеко за межі браузерів та користувацьких інтерфейсів і став самостійною мовою програмування, яка здатна працювати на серверах, мобільних пристроях та інших вбудованих системах. Можливим це стало завдяки влаштованим середовищам виконання, таким як Node.js. В браузерах код виконується завдяки вбудованому рушію V8, SpiderMonkey, що ще іноді називають віртуальною машиною виконання.

Сучасний JavaScript є безпечною мовою програмування. Він не відкриває доступ до пам’яті або процесора, що гарантує ізоляцію коду та логіки в браузерному середовищі й буде безпечною для запуску на пристроях кінцевих користувачів.

Важливим доповненням сучасного JavaScript, враховуючи його спектр використання, є підтримка парадигм об’єктно-орієнтованого та функціонального програмування та розвинутої екосистеми з потужними інструментами, бібліотеками, фреймворками, готовими рішеннями. Що в свою чергу робить його таким популярним та дозволяє будувати цілі інформаційні системи без задіяння інших мов програмування.

1.2.4 Роль фреймворків у розробці клієнтських інтерфейсів

Фреймворки – це свого роду платформи, які надають розробникам комплексний набір інструментів, архітектурних підходів та базових рішень для створення інтерфейсів. Вони значно полегшують процес розробки та забезпечують структурованість коду, що в свою чергу грає роль в продуктивності користувацьких інтерфейсів. Зі стрімким ростом складних інтерфейсів, динамічних оновлень вмісту інтерфейсу та необхідності інтерактивної взаємодії з користувачем, розробка з використанням нативного JavaScript стала не ефективною. Саме фреймворки і бібліотеки дозволили вирішувати проблеми організації коду, оптимізації продуктивності та управління станом.

Популярність фреймворків зумовлена створенням стандартизації підходів до архітектури користувацьких інтерфейсів. Вони зазвичай надають готову певну структурну модель — компонентну, реактивну чи MVC-подібну, що дозволяє розробникам працювати узгоджено в команді.

Компонентний підхід, який став свого роду еталоном після 2015 року, надає можливість розбити інтерфейс на окремі незалежні модулі, компоненти для подальшого повторного використання. Це спрощує структуру UI-елементів, пришвидшує розробку і зменшує кількість дубльованого коду.

Продуктивне управління станом інтерфейсу теж є великим досягненням фреймворків. У складних та масштабних проєктах з великою кількістю пов’язаних елементів інтерфейсу та даних, ручне керування станом стає неефективним та призводить до великої кількості помилок й неконсистентної взаємодії. Інструменти, які розвинулись та стали невід’ємною частиною роботи фреймворків, запобігають виникненню проблем з зайвими затратами ресурсів користувацького інтерфейсу та дозволяють централізовано керувати станом елементів, підвищують передбачувану роботу і прозорість змін в станах.

Важливою перевагою фреймворків з технічної точки зору є спрощена робота з маршрутизацією, організація роботи API для взаємодії з сервером через HTTP запити, інтеграція різних зовнішніх API та управління рендерингом, як на клієнті,

так і на сервері. Такий великий спектр готових та оптимізованих рішень робить фреймворки великими платформами для створення як простих, так і надскладних, високопродуктивних інтерфейсів з великою кількістю бізнес-логіки та індивідуальних запитів від замовника цього інтерфейсу.

Крім великої кількості технічних переваг, порівняно з нативними методами та інструментами розробки, акцентованим пунктом можна виділити велику й розвинену екосистему інструментів, дуже активну спільноту, якісну та офіційну документацію, велику кількість навчальних ресурсів та посібників на різних мовах, що суттєво знижує поріг входу для розробників таких інтерфейсів. Враховуючи те, що різні фреймворки мають кожні свої тонкощі й аспекти, перехід з одного на інший за рахунок великої кількості матеріалів стає більш плавний та позбавлений надлишкового стресу.

Недоліки у розробці інтерфейсів на базі фреймворків важливо теж підкреслити, хоча переваг набагато більше. Основним недоліком є зростання складності проєктів, великої кількості абстракцій та парадигм, які використовуються в комбінаційних варіантах. Використання фреймворків неминуче створює велику кількість залежності від сторонніх бібліотек та різних екосистем, які потребують постійних оновлень та рефакторингу. Надмірне застосування абстракцій й службового коду негативно впливають на швидкість, продуктивність інтерфейсів та на загальні метрики ресурсозатратності [14].

1.3 Ознайомлення з фреймворком React

React – це в першу чергу бібліотека, а не повноцінний MVC-фреймворк, яка набула великої популярності завдяки своїм архітектурним підходам до розробки динамічних користувацьких інтерфейсів. Вона має свою дуже велику екосистему та аудиторію, що дозволяє прирівнювати її до повноцінного фреймворку. Бібліотека React зайняла домінуючу позицію серед своїх аналогів, яка стала де-факто стандартом для створення інтерактивних та динамічних користувацьких інтерфейсів

Філософія цієї бібліотеки є унікальною та зосередженою на декларативному методі розробки, компонентності, односторонньому напрямі потоку даних і асинхронної моделі оновлення, що кардинально спрощує створення складних інтерфейсів та пропонує відмінний підхід, відмовляючись від типової моделі шаблонів MVC на користь архітектури, яка є компонентно-орієнтованою.

React змінив основу принципів взаємодії розробника та дерево подібного об'єкту, який представляє інтерфейс – DOM.

1.3.1 Історія та основні концепції фреймворку React

Фреймворк React був створений інженером Facebook, зараз Meta, Джорданом Волке, та вперше був представлений на публіку у 2013 році як проста бібліотека для зручного і ефективного створення інтерфейсів та керування ними, та яка стала провідною технологіє в розробці.

На той час існувало достатньо інструментів, які виконували схожу роботу, але будувались на основі імперативних підходів, що не давало повноцінно та ізольовано створювати інтерфейси без додаткових інструментів і глибоких знань кожного з них.

Запропонувавши новий архітектурний патерн – FLUX, створений розробниками Facebook для продуктивного управління станом інтерфейсу [15], що дозволило на сьогодні ефективно створювати не тільки веб-орієнтовані інтерфейси але й ще мобільні додатки.

В кінці 2013 року, команда розробників Facebook розробила й інтегрувала чат в платформу Facebook, після чого зіштовхнулась з чисельними викликами. Інтеграція вимагала нетривіальних рішень, таких як: управління надмірною та неконтрольованою зміною DOM (Document Object Model); надання можливості паралельної асинхронної роботи користувачів в новій екосистемі [16].

Інструменти для реалізації такого проєкту на той період не мали такий функціонал, який би забезпечив отримання потрібного результату. Для подолання цих викликів команда розробників React почала застосовувати наступні рішення:

- FLUX архітектура для односпрямованого потоку даних в інтерфейсі;
- незмінність стану компонента. Стан після першого визначення не може бути змінений на пряму для уникнення повного оновлення компонента інтерфейсу.

У 2018-2020 роках концепція бібліотеки React зазнала значної еволюції після появи React Hooks, які надали можливість керувати станом та побічними ефектами у функціональних компонентах, що зумовило перехід від класових компонентів до функціональних. Hooks дозволили виконувати повторно певні частини логіки без дублювання кодової бази, що спричиняло зростання продуктивності та уникнення розростанню складності компонентів.

В 2022 році вийшов React 18, який мав у собі нову концепцію конкурентного рендерингу (Concurrent Rendering) та новий механізм планування задач — React Fiber. Це дало можливість більш гнучко керувати інтерфейсом на базовому рівні, переривати рендеринг, оновлювати елементи за пріоритетом та робити інтерфейс більш плавним для користувачів, які мають апаратну платформу зі слабким CPU, нестабільною мережею інтернет [17].

1.3.2 Аналіз механізмів оновлення інтерфейсу

У React майже кожен процес при відображенні елемента інтерфейсу викликає роботу методів життєвого циклу цих елементів та під час його існування. Методи самі по собі не є складними і мають цілком інтуїтивні для розуміння назви. Розуміння життєвих циклів й того, що кожна зміна елементів інтерфейсу може зумовити повне оновлення всіх дочірніх елементів та перемалювати повністю інтерфейс і викликати інші сторонні ефекти.

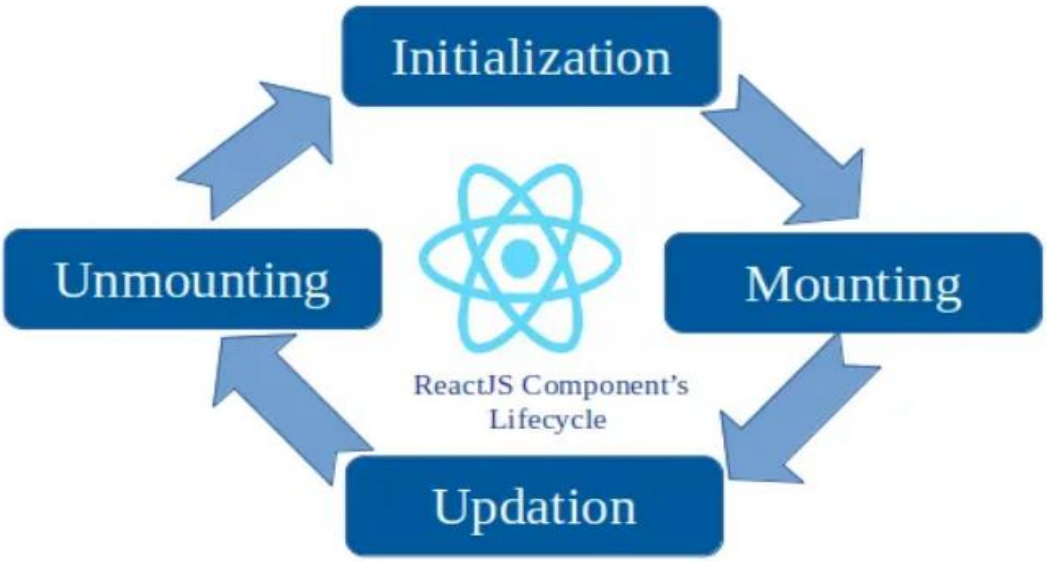

Рис.1.3 – Життєвий цикл компонентів React

Життєвий цикл компонента поділений на 4 фази:

1. Монтування (Mounting);
2. Оновлення (Updating);
3. Розмонтування (Unmounting);
4. Обробка помилок (Error Handling).

На кожній з цих фаз існують свої методи, які є специфічним для неї і мають свою функціональність та принципи взаємодії.

На фазі монтування (Mounting) доступні такі методи:

- Constructor() – викликається при створенні та ініціалізації стану;
- getDerivedStateFromProps() – викликається перед рендерингом;
- render() – викликається для відображення елементу інтерфейсу;
- componentDidMount() – викликається після того як елемент вперше відмалювався.

На фазі оновлення (Updating) доступні наступні методи:

- getDerivedStateFromProps() – викликається перед оновленням;

- shouldComponentUpdate() – викликається перед оновленням для оцінки чи повинен елемент після змін перемалюватись;
- render() – викликається для відображення елементу інтерфейсу;
- componentDidUpdate() – викликається після оновлення.

На фазі розмонтування (Unmounting) доступні такі методи:

- componentWillUnmount() – викликається перед тим як елемент буде видалений з DOM.

На фазі обробка помилок (Error Handling):

- getDerivedStateFromError() – викликається при наявності помилок в дочірніх елементах;
- componentDidCatch() – викликається після обробки помилок для їх подальшого логування або завершення роботи логіки.

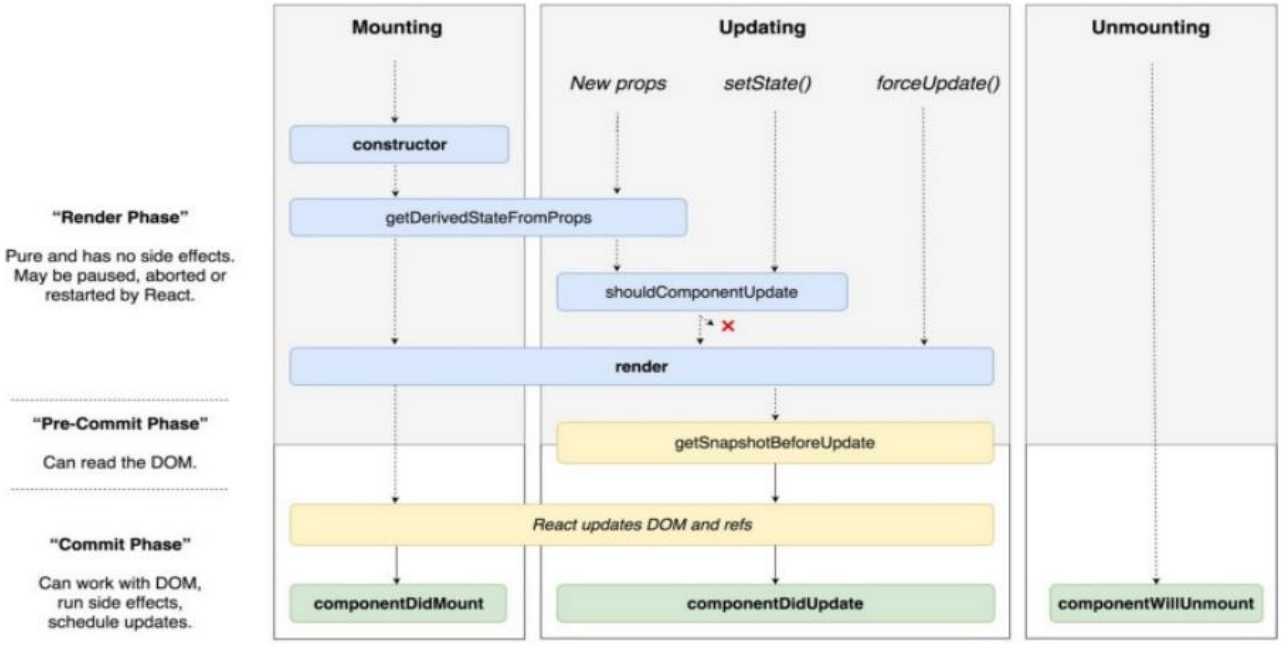

Рис. 1.4. Методи життєвого циклу компонента

В основі популярності бібліотеки React лежить його ефективність до оновлення користувацького інтерфейсу, який базується на концепціях віртуального DOM, механізму узгодження (Reconciliation) та обчислюванні оптимізації рендерингу.

Віртуальний DOM (VDOM) – це абстрактне представлення інтерфейсу у вигляді деревоподібної структури, яка дозволяє максимально ефективно виконувати оновлення структури реального DOM в браузері [18].

Кожен елемент інтерфейсу є об’єктом, що описує атрибути, тип та всі дочірні елементи цього вузла. Створення навіть великої кількості таких об’єктів є швидкою операцією, якщо порівнювати зі створення реальних вузлів браузера.

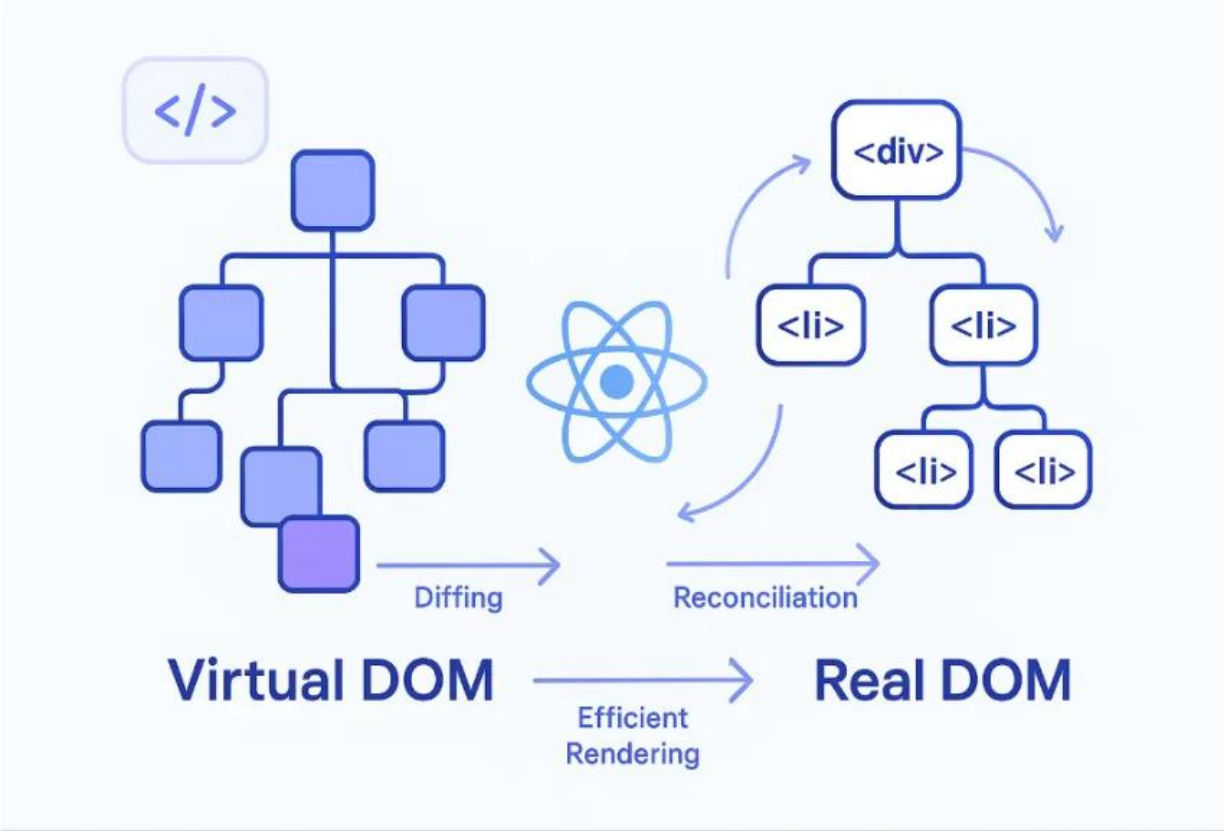

Рис.1.5 – Схема роботи віртуального DOM

Механізм роботи:

1. При зміні властивостей або стану елементів інтерфейсу створюється нове дерево Virtual DOM;

2. Спрацьовує алгоритм порівняння який порівнює нове дерево зі старим та обчислюється різниця вузлів;

3. Процес синхронізації який забезпечує алгоритм узгодження. Віртуальне дерево стає реальним.

Операції по зміні DOM в браузері є затратними по ресурсу і, якщо маніпулювати ним на пряму та без потреб, може викликати падіння продуктивності

інформаційної системи на клієнтській частині. Тому в додаток до віртуального DOM додається один з ключових механізмів – алгоритм узгодження (Reconciliation Algorithm).

Алгоритм узгодження (Reconciliation Algorithm) – це алгоритм, який виконує порівняння поточного стану віртуального DOM із його попередньою версією. На базі цього порівняння алгоритм визначає, яку саме частинку інтерфейсу треба оновити, після чого змінює реальний DOM браузера лише в необхідних для цього місцях та тільки ті елементи, які були змінені.

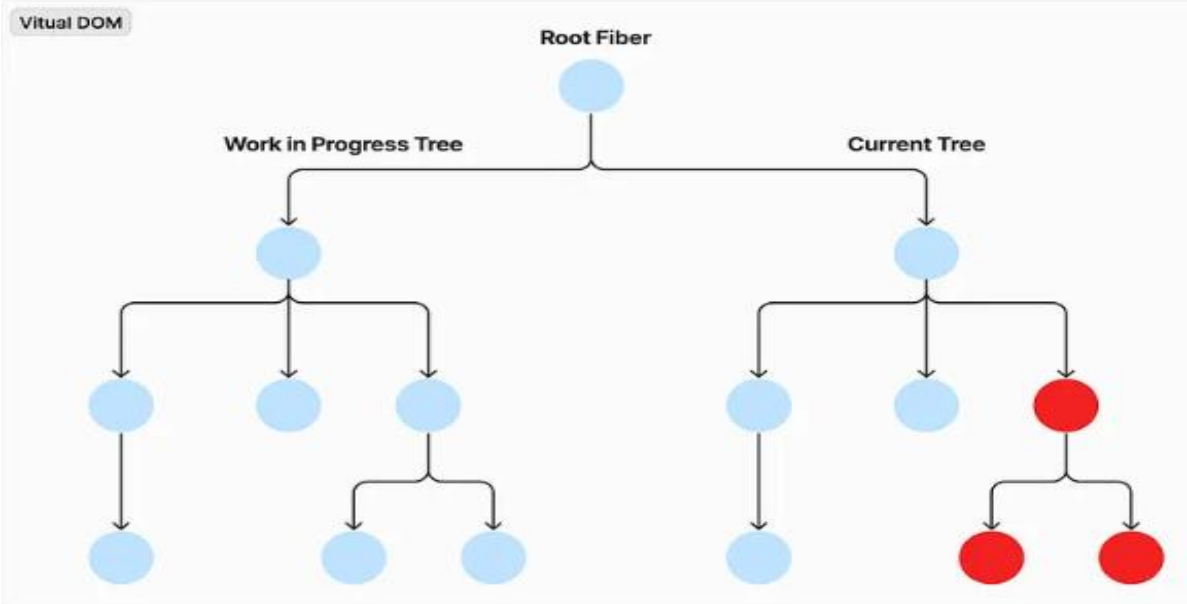

Рис.1.6 – Схема роботи react reconciliation

Алгоритм має алгоритмічну складність порядку *O(n),* що являється лінійною складністю та є ефективнішою та менш ресурсозатратною порівняно з класичним порівнянням двох довільних дерев (Tree Edit Distance), де складність становить *O(n^3), n* – кількість елементів дерева.

Розробники React застосували евристичний алгоритм, який в основі має два припущення, що являється істиною для інтерфейсів:

- елементи різних типів будуть створювати різні дерева;
- розробники інтерфейсів можуть вказувати на те, які дочірні елементи будуть стабільними для запобігання лишніх рендерів, використовуючи атрибут key.

Якщо основні елементи двох дерев мають різний тип наприклад, div змінюється на span, або компонент Header на Footer, React повністю стирає старе дерево та формує нове з нуля. Це рішення є радикальним і дозволяє уникнути складних порівнянь несумісних структур [19].

1.3.3 Екосистема React та архітектурні доповнення

Екосистема React на сьогодні є однією з найбільших та динамічних систем у сфері розробки користувацьких інтерфейсів. React як бібліотека є мінімалістичною та сфокусована лише на побудові інтерфейсів. Однак, великі інформаційні системи потребують набагато більше ніж просто представлення елементів інтерфейсу. Їм необхідно налаштувати роботу з даними, маршрутизацією, управління станами та сховищами, кешуванням, тестування та багато чого іншого. Всі ці потреби і доповнює екосистема React, яка налічує багато користувачів, інструментів та офіційних бібліотек. Складністю такої великої екосистеми є правильний вибір інструментів та архітектурних доповнень, які надає ця екосистема в умовах великого надлишку схожих інструментів та їх використання разом з підтримкою від розробників цих доповнень.

З основних та важливих елементів екосистеми можна виокремити базові елементи, які необхідні для створення користувацького інтерфейсу складного рівня: бібліотеки управління станом, маршрутизація, тестування, інструменти збірки, метафреймворки.

Бібліотеки для управління станом мають значну перевагу за рахунок структурованого підходу до менеджменту станів. Хоча React в своїй базі має інтегровані механізми управління станом React.Context, для великих інтерфейсів цього інструменту не достатньо, тому перевагу надають таким бібліотекам: Redux, MobX, Zustand, Recoil. Всі вони мають свої переваги та недоліки один перед одним та концепції роботи можуть відрізнятись своїми архітектурними рішеннями і підходами до роботи. Вибір може вплинути на продуктивність, тому вивчення та розуміння базових аспектів необхідні, щоб зробити вірний вибір інструменту.

За маршрутизацію в React відповідає доповнення React Router для забезпечення навігації без перезавантаження сторінки та отримання нового HTML документу з серверу. React Router на сьогодні є стандартом для роботи з маршрутизацією та немає конкурентних аналогів такого доповнення, який надавав би такий же спектр декларативних API для роботи [20].

Одними з найскладніших частин користувацького інтерфейсу є робота з даними та управління асинхронними операціями. Ці процеси відповідають за ядро всього інтерфейсу та включають: запити до серверу, синхронізацію та оновлення даних, кешування. Бібліотеки TanStack Query та RTK Query, які спеціалізуються на цій частині та значно перевершують за можливостями нативні та влаштовані інструменти fetch і useEffect в React є найпопулярнішими в своєму сегменті за рахунок великої підтримки та постійного оновлення.

Тестування та розробка в цілому теж мають доволі багато аналогів в екосистемі, але є інструменти які суттєво вириваються вперед за всіма параметрами. Для розробки найпопулярнішими інструментами збірки є Webpack та Vite, які зараз використовуються для написання тестів, Jest та React Testing Library для тестування елементів з точки зору користувача. Для інспектування, профілювання та дебагу використовується спеціалізоване розширення браузера React DevTools, яке значно спрощує розробку інтерфейсів.

Централізованим рішенням для проблем з SEO в інтерфейсах SPA архітектури стало появою метафреймворків, які їх вирішували за допомогою архітектурного розширення React та надання серверних можливостей. Найпопулярнішим метафреймворком є Next.js, який надає можливості інтеграцій різних стратегій рендерингу та дає можливість інтегрувати SEO інфраструктуру [21].

1.3.4 Аналіз оптимізації продуктивності на рівні ядра React

На рівні ядра React має низку просунутих технологій оптимізації окрім механізму virtual DOM, які націлені на мінімізацію проблем з продуктивністю,

мінімізацію основного потоку виконання та процесів перемальовування. Архітектура React Fiber на сьогодні має оновлену структуру роботи. Відбувся перехід з рекурсивної роботи на ітеративну з використанням структури даних у вигляді дерева, де кожен вузол зв’язаний через двобічні зв’язки.

Основними вбудованими методами оптимізації які доступні можна виділити:

- мемоізація компонентів та значень;
- оптимізація списків за допомогою ключів;
- код-спліттинг;
- ледаче завантаження;
- пакетування оновлень стану;
- транзиції;

Найпоширенішою причиною падіння продуктивності користувацького інтерфейсу є зайві перемальовування інтерфейсу, коли компонент створюється заново без реальної необхідності та виконує логіку з обчисленням кожен зайвий рендеринг стає важким [22]. Для усунення цих проблем React пропонує такі механізми мемоізації:

- React.memo() – компонентна обгортка яка мемоізує результати рендерингу. Якщо стан та властивості не змінились React буде повторно використовувати останні результати;
- useMemo() – спеціальний хук для мемоізації обчислень. Використовуючи кешування він зберігає результати важких обчислень поки не будуть змінені залежності;
- useCallback() – хук який дозволяє мемоізувати функції зворотного виклику.

Для оптимізації списків React додав атрибут key, який необхідно додавати в кожен елемент списку. Завдяки цьому атрибуту React розуміє який саме елемент списку змінився і який треба перемалювати.

Код-спліттинг та ледаче завантаження за мету має зменшити початковий розмір JavaScript бандлу, що покращує показники швидкості завантаження.

Пакетування (Batching) це механізм, який виконує групове оновлення стану в один єдиний цикл перерендеру. Якщо в одному з обробнику подій відбулося декілька підряд викликів змін стану, React не запустить після кожного новий рендер, а дочекається виконання всіх викликів та виконає лише одне оновлення інтерфейсу [22].

Транзиції з’явились зовсім недавно з останніми версіями React. Вони дозволяють оновлювати стан без блокування основного потоку, тобто асинхронно. Це робить користувацький інтерфейс плавним та стабільним навіть при виконанні важких обчислень чи операцій [22].

1.3.5 Вплив підходів управління ресурсами як оптимізація інтерфейсу

Продуктивність клієнтського інтерфейсу залежить не лише від швидкості рендерингу елементів цього інтерфейсу, але й ефективного управління ресурсами такими як: мережеві запити, обчислювальна потужність апаратної частини клієнта та пам’яттю. Екосистема React надає інструменти для керування цими ресурсами, які в свою чергу мають значний вплив основні параметри швидкодії інтерфейсу.

Під час роботи з запитими на сервер є декілька технік які покращують керування мережевими ресурсами та покращують показники користувацького досвіду (UX). Одною з таких технік є попереднє завантаження (Prefetching) – ця стратегія спрямована на оптимізацію взаємодії користувача та інформаційної системи шляхом завантаження даних або ресурсів заздалегідь які, ймовірно, будуть потрібні користувачу. Ще Одною такою технікою є оптимістичні оновлення (Optimistic Updates), яка оновлює інтерфейс завжди успішно при взаємодії користувача з системою, (видалення, додавання елементів списку), ще до отримання успішної відповіді від серверу. Це створює ілюзію неймовірно швидкого користувацького інтерфейсу. Якщо запит буде відхиллено, стан повернеться до попереднього значення [23].

Інтерфейси створені на базі React особливо схильні до витоків пам’яті за рахунок своїх особливостей та особливостей мови програмування. Важливо

правильно підчищати побічні ефекти, підписки на події та задання інтервалів у роботі логіки, які будуть накопичувати непотрібні об’єкти або інші структури даних навіть після демонтажу елементів інтерфейсу.

Також важливо при роботі з великими списками, які можуть мати тисячі елементів цього списку використовувати віртуалізацію як спосіб оптимізації цих списків. Віртуалізація буде відображати лише ті елементи списку, які будуть потрапляти у видиму область користувача [24].

Операції які інтенсивно використовують ресурс CPU такі як: сортування або перебирання великих масивів, обробка зображень, складні анімації, можуть викликати блокування основного потоку та появи підвисання інтерфейсів. В таких випадках винесення важких обчислень у окремі потоки завдяки Web workers дасть змогу запобігти блокуванню.

1.4 Аналіз проблем продуктивності клієнтської частини на базі React

Розробка користувацьких інтерфейсів на React потребує креативності, глибоких знань та уважності до деталей у сучасній розробці інтерфейсів, де швидкість розробки й ефективність інтерфейсу мають велику вагу. У цьому розділі ми поглибимось у аналіз факторів, що мають вплив на продуктивність клієнтської частини React інтерфейсів. Розглядаються аспекти коду, рендерингу, архітектури, оптимізації завантаження ресурсів, управління станом та мережевої взаємодії. Розв'язання цих проблем визначає технічну якість інтерфейсу та здатність забезпечувати високий рівень продуктивності і користувацького досвіду. Цей розділ допоможе краще зрозуміти фактори, які впливають на швидкість та стабільність інтерфейсів на базі React, та запропонує рекомендації для їх подолання, створюючи підґрунтя для оптимальної та ефективної розробки з підвищенням продуктивності.

Продуктивність роботи інтерфейсу React залежить від багатьох факторів. Далі перераховано основні чинники та ключові аспекти, які можуть впливати на продуктивність інтерфейсів на базі:

1. Використання стану (state) в React: стан є ключовим механізмом для збереження та керування даними у користувацькому інтерфейсі. Водночас надмірне або часте оновлення стану викликають непотрібні рендеринги, які в свою чергу знижують продуктивність та швидкодію. Рекомендується мінімізувати кількість оновлення стану при кожній незначній зміні, щоб зменшити навантаження, обсяг зайвого коду і підвищити читабельність. Централізований підхід з використанням спеціалізованих інструментів для керування та оновлення стану робить код більш структурованим, а його роботу стабільною.

2. Управління станом й залежності: зростання кількості станів та підвищення складності потребує більш ретельного планування та управління. Надмірна та необґрунтована фрагментація станів між чисельними компонентами призводить до ускладнювання розуміння коду й управління ним. Важливо дотримуватися централізації та прозорості структури інтерфейсу.

3. Взаємодія з життєвим циклом компонентів: некоректне використання механізмів життєвого циклу елементів інтерфейсу може спричиняти помилки, некоректну обробку даних та збереження попереднього стану. Важливо мати глибоке розуміння принципів та коректно користуватися методами життєвого циклу для забезпечення необхідної функціональності й запобігання потенційних проблем.

4. Використання ключів (keys): некоректне задання ключів може спричиняти втрату даних або помилки під час оновлення компонентів. Для коректної роботи механізму зіставлення елементів варто уникати дублювання ключів та забезпечувати їх унікальність для кожного елемента інтерфейсу. Це дозволить React коректно працювати з DOM.

5. Створення складних та великих компонентів: надмірно великі й складні компоненти ускладнюють розробку, підтримку, відлагодження та тестування. Доцільно дотримуватися принципів якісної архітектури, розділяючи логіку на менші елементи та використання React.

Незважаючи на те, що React в базовій конфігурації має достатньо методів та засобів оптимізації, неправильне використання або недосконала архітектура

можуть призводити до неефективного використання ресурсів та методів життєвого циклу компонентів [25]. Важливо підкреслити декілька ключових причин, що призводять до виникнення помилок під час роботи з React:

- Новачки в React: недостатній рівень знань у початківців спонукає до проблем з освоєнням механізмів та архітектури React, що може спричиняти до неправильного використання методів або використання їх з відсутністю розуміння.
- Нестача досвіду в React: недостатній практичний досвід розробника часто сприяє застосуванню шаблонних рішень, звичайного копіювання коду без розуміння його роботи та помилкові тлумачення концепції React.
- Зміни в версіях React: оновлення можуть викликати труднощі в міграційних процесах та конфлікти, які в більшості випадів не завжди можна одразу визначити. Досвід роботи з оновлень та переходу на більш нові версії спрощує цей процес та може зарадити появі проблем.
- Некоректне використання методів: використання методів життєвого циклу не за призначенням негативно впливає на продуктивність користувацького інтерфейсу та створює небажані залежності та непередбачувану поведінку.
- Відсутність архітектурного планування і роботи: недостатня увага до планування часто призводить до ускладнення, розуміння та підтримку кодової бази.

Робота з механізмами React може бути ускладнена та пов’язана численними труднощами, особливо для початківці та розробників з недостатнім досвідом [26].

Для уникнення цих проблем, важливо уважно вивчати матеріали та документацію React, задіювати сучасні підходи та слідкувати за оновленням концепцій.

1.5 Висновки до розділу

В даному розділі було проаналізовано та досліджено предметну область розробки користувацьких інтерфейсів та його ключові складові, розпочинаючи з

визначення терміну "інтерфейс", і закінчуючи ключовими причинами, які призводять до зниження продуктивності та якості самого інтерфейсу.

Проаналізовано архітектурні рішення та базові принципи роботи користувацьких інтерфейсів які створені на базі фреймворку React, проведено аналіз інструментів які застосовуються для розробки інтерфейсів включаючи JavaScript, HTML та CSS. Особлива увага була спрямована на ознайомлення з фреймворком React. Розкрита його історія та застосування, основні концепції та його екосистему.

Проведений аналіз базових методів оптимізації на рівні ядра та вплив управління ресурсами, які підкреслюють свою вагу в розрізі продуктивності на надійності користувацького інтерфейсу.

Даний розділ дозволив сформувати повне розуміння проблематики, яка може виникнути під час роботи з React екосистемою. Цей аналіз буде використаний фундаментом для подальшого дослідження та буде визначати напрямок оптимізації, які будуть розглядатись в наступних розділах кваліфікаційної роботи.

РОЗДІЛ 2. РОЗРОБКА ТА МЕТОДИ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ КЛІЄНТСЬКОЇ ЧАСТИНИ НА БАЗІ REACT

2.1 Оптимізація списків за допомогою ключів

Для того, щоб наглядно побачити та проаналізувати вплив оптимізації списків на продуктивність користувацького інтерфейсу для експерименту створимо невеликий інтерфейс, який матиме два компоненти зі списками користувачів. Ці два компоненти будуть імітувати просту роботу списку елементів, які будуть мати функціональність додавання користувача, видалення та поле для додавання інформації по кожному користувачу. Також був доданий лічильник рендерингу до кожного зі списку для того, щоб наочно показати розбіжність.

Ключовою відмінністю в цих списках буде значення атрибуту key. В першому компоненті списку значенням буде виступати індекс масиву, а в другому – це буде унікальний ідентифікатор.

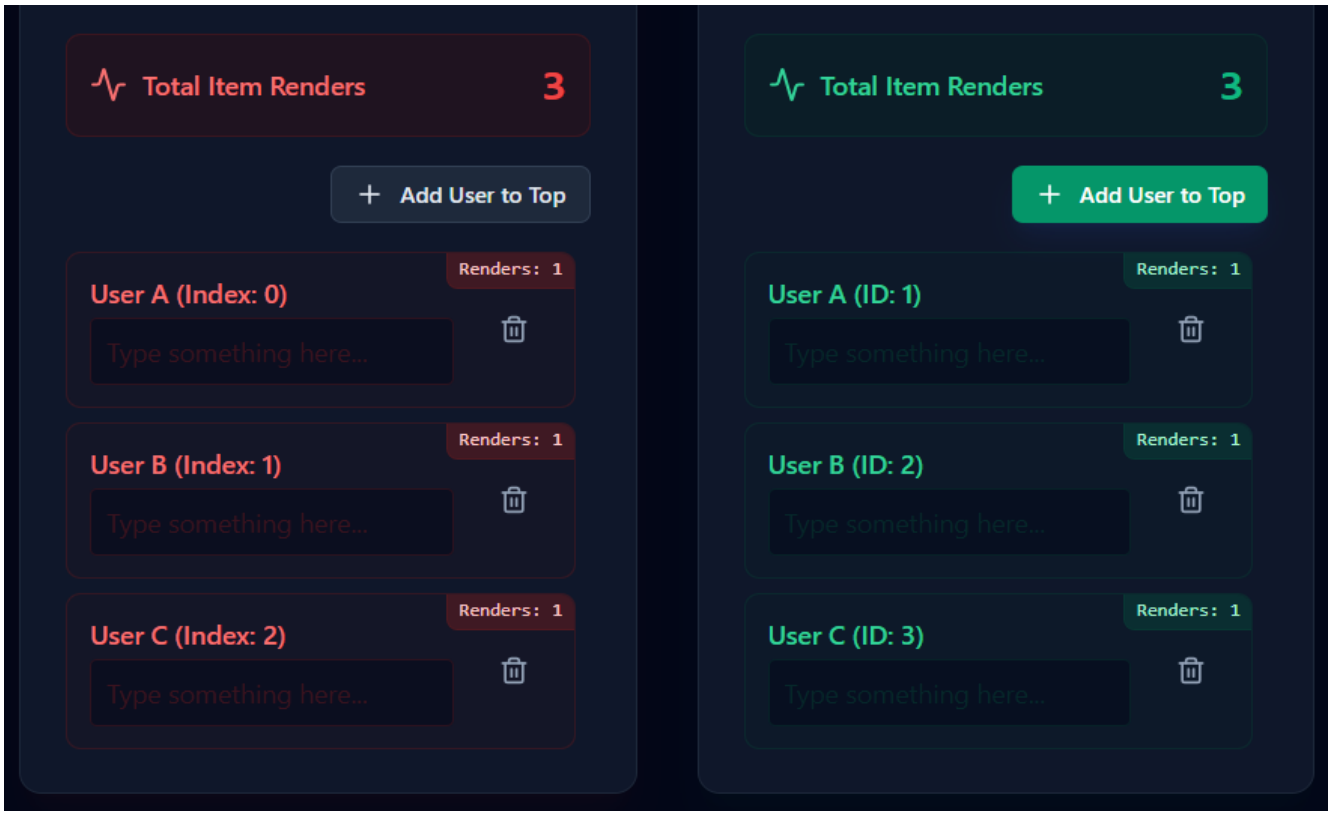

Рис.2.1 – Демонстрація інтерфейсу з двома списками

Якщо поглянути більш детально на роботу списку червоного кольору, де ключі мають значення індексу, то ми можемо спостерігати, що при додаванні або видаленні користувача у списку, всі існуючі елементи списку змінюють свій індекс, що призводить до перемалювання всіх елементів у цьому списку. React вважає, що всі елементи списку зазнали змін.

Якщо додатково заповнити інформацію про кожного користувача, почати додавати та видаляти елементи в червоному списку можна спостерігати некоректну поведінку інтерфейсу і значну кількість рендерів на лічильнику. Додані елементи отримують інформацію створених, а вже створені очищуються. Відбувається це за рахунок того, що React використав існуючі DOM-вузли для інших елементів інтерфейсу.

Для оптимізації цього списку використаємо атрибут key та присвоїмо йому значення унікального ідентифікатора.

```
<div className="space-y-2 max-h-[400px] overflow-y-auto pr-2 custom-scrollbar">
        {items.map((item) => (
         <ListItem
           key={item.id}
           name={`${item.text} (ID: ${item.id.substr(0, 4)})`}
           onDelete={() => deleteItem(item.id)}
         />
        ))}
        {items.length === 0 && (
         <div className="text-center py-8 text-slate-600 text-sm italic">
           List is empty
         </div>
        )}
</div>
```

У прикладі наведеному вище відображається код, який належить списку зеленного кольору, в якому кожен елемент має стабільне та унікальне значення key.

Таким чином додавання та видалення користувачів не викликає зайвих оновлень на лічильнику, а також відсутні порушення поведінки інших елементів. React коректно ідентифікує, які саме елементи інтерфейсу були видалені або додані та які зазнали змін.

2.2 Застосування React.lazy

Для наочного відображення роботи React.lazy та його впливу на початковий розмір JavaScript коду, який завантажує собі користувач, створимо React інтерфейс зі списком задач, в якому буде доступна низка можливостей: додавання і видалення задач, редагування, завантаження зображень різних форматів. Обрання цього застосунку обумовлене бажанням імітувати ситуацію імпорту файлу великого розміру, де в цьому випадку мова йде про додавання картинки до завдань. Основною метою є демонстрація спроможностей React.lazy у спільній дії з компонентами великих розмірів. Саме цей приклад дозволить нам побачити та визначити ефективність методу відкладеного завантаження.

Припустимо, що користувачу необхідно створити кілька задач, кожна з яких повинна містити в собі деталізований опис та прикріплене зображення. Щоб створити нову задачу, користувачеві необхідно натиснути лише на кнопку «Add Task». Результатом буде динамічне завантажування даних за допомогою React.lazy, відповідним компонентом.

Рис.2.2 – Демонстрація інтерфейсу списку задач

Як наслідок, перед користувачем з’являється модальне вікно, в якому йому необхідно обрати назву задачі, пріоритет, а також додати зображення. При натисканні користувачем на кнопку «Create», задача додається до списку. Під час додавання зображення до кількох завдань зі списку, виникає наступна проблема: кожного разу, при повторному відкритті інтерфейсу, зображення необхідно буде завантажувати наново. А тепер припустимо, що елементів, які мають зображення, у списку більше 100.

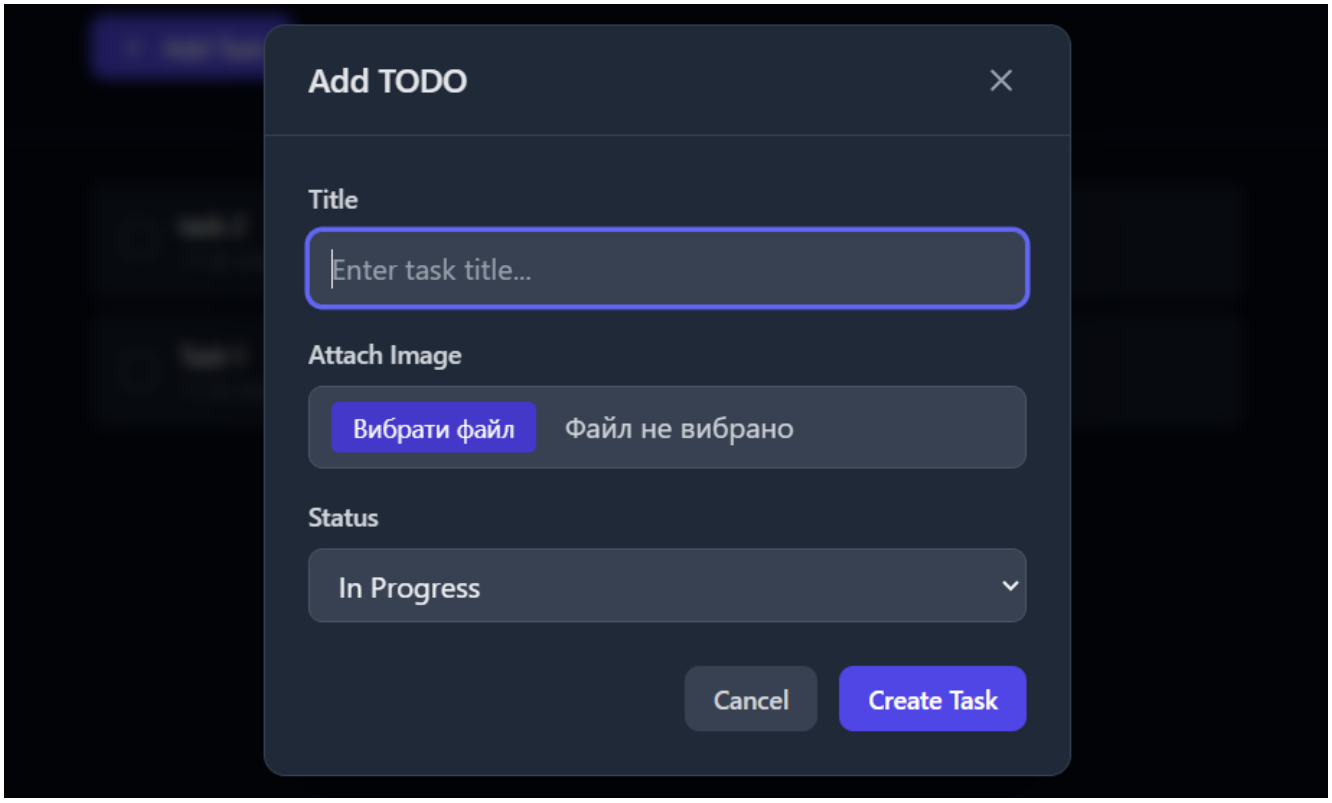

Рис.2.3 – Демонстрація інтерфейсу додавання задач

Стає зрозуміло, що подібний неоптимальний підхід спричинить збільшення часу на завантаження користувацького інтерфейсу, даремно витрачатиме ресурси користувача та зробить клієнтську частину важкою. Метод, який ми будемо застосовувати, вирішить проблему шляхом одноразового завантаження зображень. До того ж, при прокручуванні списку, не буде потреби відмальовувати всі зображення, що містяться в ньому. Відкривши консоль розробника, перейшовши на вкладку Network, після чого обравши відображення файлів формату .js, перед

нами з’явиться список завантажених файлів із зазначенням їх об’єму.

Name	Size
bundle.js	6.5 KB
0.chunk.js	1.3 MB
main.chunk.js	4.4 KB
main.331f5907ea5fed2b9abf.hot-update.js	1.1 KB
page.js	(from disk cache)

Рис.2.4 – Результати завантаження без React.lazy

Наступним етапом ми використаємо інструмент відкладеного завантаження та розділення коду до інтерфейсу. Для обох підходів буде застосовано один елемент, який буде імпортувати залежності з зображенням та його відображенням.

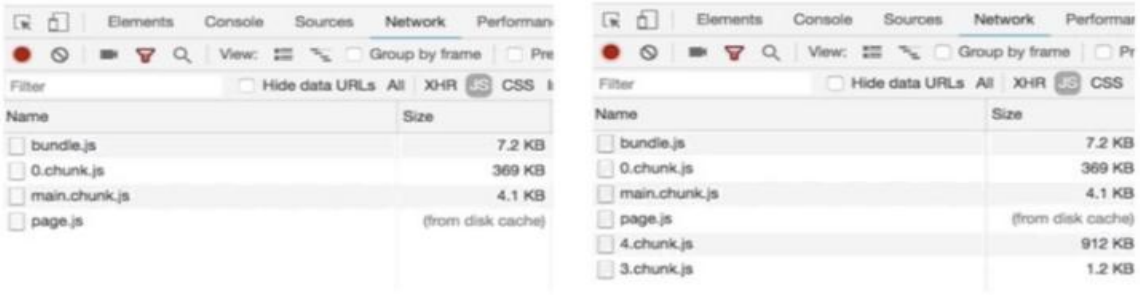

Рис.2.5 – Результати завантаження з React.lazy

В результаті бачимо, що 0.chunk.js має значно менший розмір, ніж було раніше, а завантаження 3.chunk.js та 4.chunk.js відбувається після натискання кнопки. Отож, можемо прийти до висновку, що дані методи та інструменти позитивно вплинули на швидкість завантаження інтерфейсу та його роботу.

2.3 Використання Мемоізації

Щоб проаналізувати вплив інструментів React.useMemo, React.useCallback, React.memo, які виступають засобами мемоізації елементів та які впливають на продуктивність інтерфейсу, візьмемо елемент, який буде нам відмальовувати список продуктів та робити важкі обчислення. Метою мемоізації є уникнення

надмірних рендерів та обмеження повторних обчислень у випадку коли дані для цих обчислень залишились ті самі.

2.3.1 React.useMemo

Для імітації навантаження список будемо брати з 20 000 записами та робити фільтрацію по ціні від 1 000 та можливістю сортування.

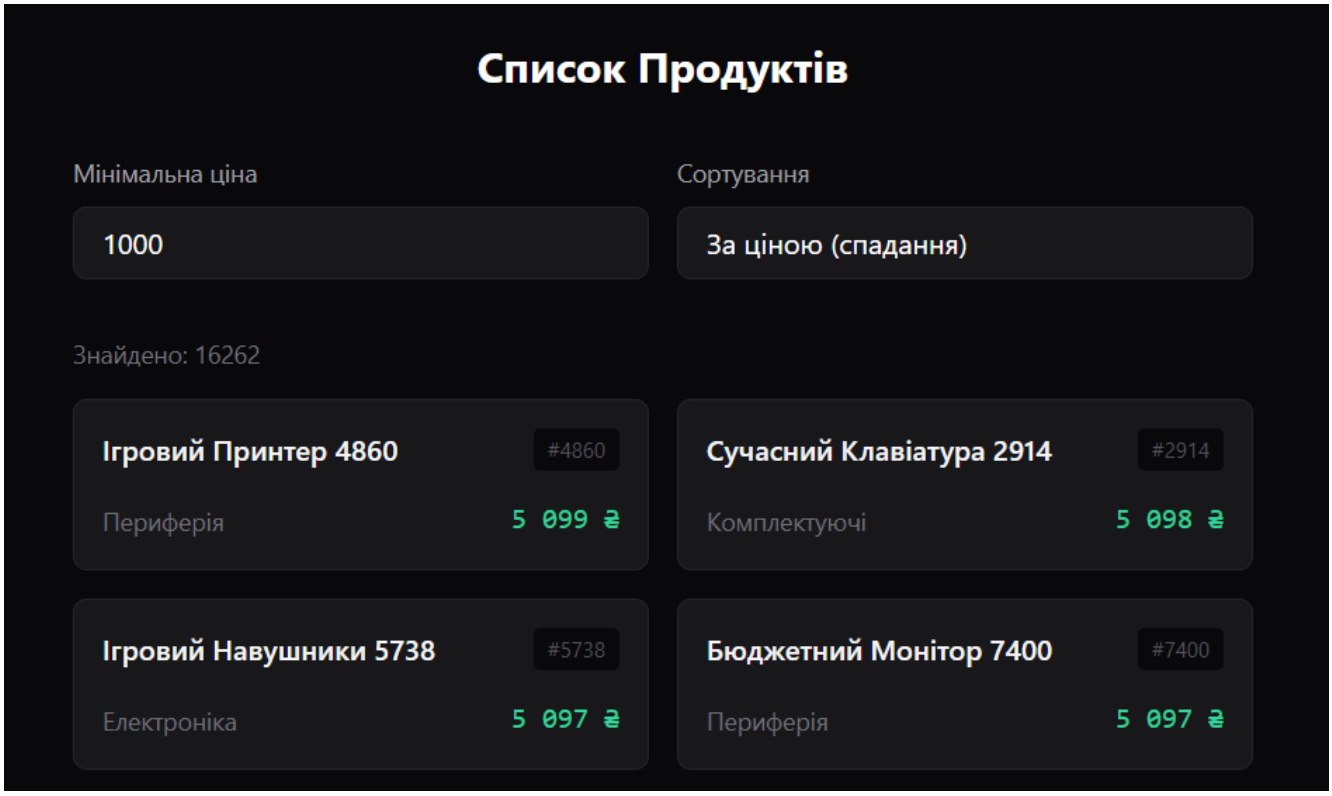

Рис.2.6 – Демонстрація інтерфейсу зі списком продуктів

В даному прикладі список продуктів буде повністю перемальовуватись при кожній зміні стану елементу інтерфейсу. Також буде проводитись фільтрація від 1 000 навіть якщо список залишився абсолютно таким самим та при зміні параметра сортування буде відбуватись обчислення. Фільтрація блокує основний потік виконання, тому наявність чисельних розрахунків у компонентів викликає затримку, яка може займати досить багато часу. У результаті програма не реагує на дії користувача, крім того витрачає ресурси апаратної частини.

Для покращення роботи інтерфейсу можемо використати useMemo для

функції фільтрування. Після кожного рендерингу фільтрація не буде відбуватись якщо не змінився сам список. useMemo кешує результат фільтрації і повторно виконує обчислення лише коли змінюється products. Це зменшує навантаження на JavaScript-потік і дозволяє уникнути надлишкових операцій обчислення та рендерингу. Нижче наведено демонстраційний фрагмент коду з застосування React.useMemo.

```
function ProductList({ products }) {
  const expensiveCalculation = useMemo(() => {
    return products.filter(p => p.price > 1000);
  }, [products]);

  return (
    <div>
      {expensiveCalculation  .map((p) => (
        <div
          key={p.id}
              className="bg-zinc-900 border border-zinc-800 p-4 rounded-lg hover:border-zinc-700 transition-colors"
        >
          <div className="flex justify-between items-start mb-2">
            <h3 className="text-zinc-100 font-medium">{p.name}</h3>
            <span className="text-xs text-zinc-600 bg-zinc-950 px-2 ">
              #{p.id}
            </span>
          </div>
          <div className="flex justify-between items-center mt-4">
            <span className="text-zinc-500 text-sm">{p.category}</span>
            <span className="text-emerald-400 font-mono font-medium">
              {p.price.toLocaleString('uk-UA')} ₴
            </span>
```

```
        </div>
      </div>
  );
}
```

2.3.2 React.memo

Після оптимізації важкого обчислення за рахунок інструменту React.useMemo, питання з надлишковим перемалювання елементів інтерфейсу залишається актуальним. При кожному рендерингу батьківського компонента всі дочірні також повторно оновлюються. React.memo запобігає надлишковому рендерингу саме окремого елементу списку за рахунок обгортання елементу в елемент вищого порядку. Для усунення такої проблеми використаємо React.memo, який дозволить нам виконати мемоізацію самого компонента і він буде перемальовуватись лише в тому випадку коли змінились вхідні дані.

Нижчезазначений фрагмент коду містить приклад розділення логіки важкого обчислення та обгортання елементу в React.memo.

```
function ProductList({ products }) {
  const filteredProducts = useMemo(() => {
    return products.filter(p => p.price > 1000);
  }, [products]);

  return (
    <div className="grid gap-4">
      {filteredProducts.map(product => (
        <ProductItem key={product.id} product={product} />
      ))}
    </div>
  );
```

Після додавання нового елементу ProductItem, який містить в собі ту ж саму структуру елементу, що і до цього, ми обгортаємо його в React.memo

```
const ProductItem = React.memo(function ProductItem({ product }) {
 return (
  <div
   className="bg-zinc-900 border border-zinc-800 p-4 rounded-lg hover:border-zinc-700 transition-colors"
  >
   <div className="flex justify-between items-start mb-2">
    <h3 className="text-zinc-100 font-medium">{product.name}</h3>
    <span className="text-xs text-zinc-600 bg-zinc-950 px-2">
     #{product.id}
    </span>
   </div>

   <div className="flex justify-between items-center mt-4">
    <span className="text-zinc-500 text-sm">{product.category}</span>
    <span className="text-emerald-400 font-mono font-medium">
     {product.price.toLocaleString('uk-UA')} ₴
    </span>
   </div>
  </div>
 );
});
```

2.3.3 React.useCallback

За певних умовах, список не може уникнути постійного рендеру, при відсутності змін і це є нормальною поведінкою. useCallback розв'язує проблему зайвого перестворення функцій при кожному рендерингу компонента [27]. Функція

handleClick створюється заново при кожному рендері, через що React повторно ререндерить усі дочірні компоненти ProductItem, навіть якщо їхній стан не змінився. Це викликає надмірне використання ресурсів і знижує продуктивність.

У фрагменті коду наведеному нижче ми додаємо функцію handleClick, яка при виконанні буде виводити в консоль унікальний ідентифікатор продукту, в якому вона була викликана, та передаємо її у вигляді атрибуту в компонент ProductItem. Саму функцію обгорнемо в хук useCallback, який мемоізує її та збереже одне і теж саме посилання на функцію. Це дозволить уникнути повторного створення функції, відповідно, зменшить надмірний рендеринг.

```
function ProductList({ products }) {
  const filteredProducts = useMemo(() => {
    return products.filter(p => p.price > 1000);
  }, [products]);

  const handleClick = useCallback((id) => {
    console.log('Selected product:', id);
  }, []);

  return (
    <div className="grid gap-4">
      {filteredProducts.map(product => (
        <ProductItem
          key={product.id}
          product={product}
          onClick={handleClick}
        />
      ))}
    </div>
  );
```

2.4 Використання useTransition

Якщо все ж таки не вдається уникнути великих обчислень на стороні клієнтського інтерфейсу і не зменшити навантаження на нього, застосування транзицій допоможе вирішити це питання. Наприклад фільтрація або сортування великих списків може виконуватися з меншим пріоритетом ніж, наприклад, введення тексту користувачем в поле пошуку. Без застосування транзицій React всі оновлення стану з однаковим пріоритетом.

Інструмент useTransition дає можливість задати певним оновленням стану низький пріоритет. У прикладі коду, наведеному нижче, додано поле введення для пошуку та додана функція для запуску пошуку. Введення тексту залишається миттєвим для користувача, тоді як сортування або фільтрація буде виконуватись у фоновому режимі.

```
import { useState, useMemo, useTransition } from 'react';

function ProductList({ products }) {
  const [search, setSearch] = useState('');
  const [isPending, startTransition] = useTransition();

  const filteredProducts = useMemo(() => {
    return products.filter(
      p =>
        p.price > 1000 &&
        p.name.toLowerCase().includes(search.toLowerCase())
    );
  }, [products, search]);

  const handleSearchChange = (e) => {
    const value = e.target.value;
```

```
    startTransition(() => {
      setSearch(value);
    });
  };

  return (
    <div>
      <input
        type="text"
        onChange={handleSearchChange}
        placeholder="Пошук продукту"
        className="mb-4 p-2 border rounded"
      />

      {isPending && (
        <p className="text-sm text-zinc-500 mb-2">
          Оновлення списку...
        </p>
      )}

      {filteredProducts.map(product => (
        <ProductItem key={product.id} product={product} />
      ))}
    </div>
  );
}
```

В той час як React виконує обробку транзицій, користувацький інтерфейс залишається інтерактивним і доступним для користувача. Шляхом відокремлення

критичних для користувача дій від важких обчислювальних операцій забезпечується стабільність та плавна робота користувацького інтерфейсу.

2.5 Висновки

У цьому розділі було продемонстровано та застосовано на практиці засоби та методи оптимізації користувацьких інтерфейсів на базі React.

Процес оптимізації списків завдяки ключам здійснюється доволі просто за рахунок використання унікальних ідентифікаторів. Проте слід пам’ятати, що ефективність даного засобу залежить від структури даних з якими працюють списки. За відсутності таких унікальних ідентифікаторів необхідно вдаватися до додаткових засобів, які забезпечать створення таких ідентифікаторів на стороні користувача.

Також у цьому розділі були задіяні інструменти поділу коду та відкладеного завантаження React.lazy. В результаті дослідження варто підкреслити, що застосування цього інструменту потребувало зовсім не великих зусиль, але результат виявився дуже ефективним. Розмір файлів з якими працює користувацький інтерфейс зменшився в рази.

Наступне дослідження було вирішення проблем з надлишковими рендерами інтерфейсу зі списком продуктів у кількості 20 000. Ця проблема призводила до нестабільної роботи інтерфейсу або великого часу відгуку, а в деяких випадках до повного припинення роботи інтерфейсу. Розв’язання цих проблем за рахунок інструментів мемоізації показали значний результат. Застосування React.useMemo, React.memo, React.useCallback зменшило навантаження на апаратну частину та зменшило час відгуку між інтерфейсом та користувачем. Важливо додати, що процес застосування цих інструментів не завжди може бути легким і може потребувати значної кількості часу та розуміння аспектів в контексті елементів інтерфейсу, де є необхідність робити оптимізацію.

Результати дослідження з застосуванням транзицій показали, що навіть при наявності великих обчислень та навантаження на інтерфейс, можна залишити продуктивність і його надійність на високому рівні.

РОЗДІЛ 3. АНАЛІЗ РЕЗУЛЬТАТІВ ОПТИМІЗАЦІЇ

3.1 Аналіз оптимізації списків за допомогою ключів

Додавання ключів для динамічних списків в React є гарною практикою для продуктивної роботи користувацького інтерфейсу, хоча і не є обов'язковою. Використання стабільних та унікальних ключів покращує роботу React та забезпечує точне визначення елементів, які саме зазнали змін або були видалені чи додані до списку.

Ключі доцільно додавати у наступних випадках:

1. Під час рендерингу списків які мають динаміку та інтерактив.
2. При додаванні, видаленні або зміні послідовності елемента.
3. У випадках коли необхідно зберігати стан елементу списку.

Якщо список не статичний і не потребує взаємодії чи динаміки, ключ можна не додавати.

Для проведення аналізу ефективності оптимізації списків за допомогою ключів та профілювання необхідно запустити інтерфейс в браузері та відкрити панель розробника Developer Tools. Далі ми переходимо на вкладку Performance, яка надає можливість аналізувати швидкодію і поведінку інтерфейсу під час взаємодії з ним за певний період часу. Натиснувши на кнопку «Record», необхідно провести типові дії користувача, наприклад додавання користувача, заповнення інформації та видалення протягом 10 секунд. Після профілювання ми натискаємо кнопку «Stop». Цей тест проводимо 5 разів для списку червоного кольору, який використовує в якості ключів нестабільні індекси, та 5 разів для списку зеленого кольору, який в якості ключів має унікальні ідентифікатори.

Дані які необхідно проаналізувати знаходяться на вкладці Summary, де відображені такі основні параметри:

- Scripting – час виконання JavaScript
- Rendering – час на обчислення стилів
- Painting – час відмалювання

- System – внутрішня робота ОС

Нижче наведено результати профілювання у вигляді зображень.

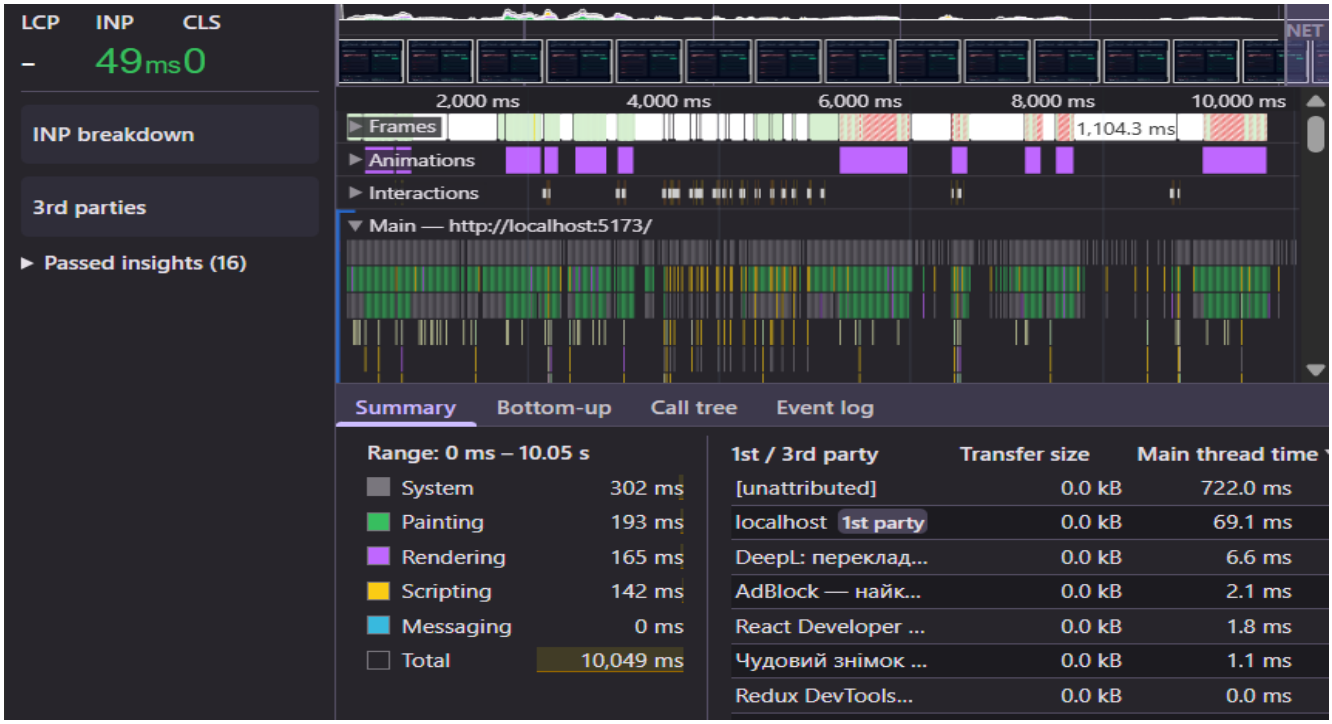

Рис.3.1 – Показники з ключами у вигляді індексів

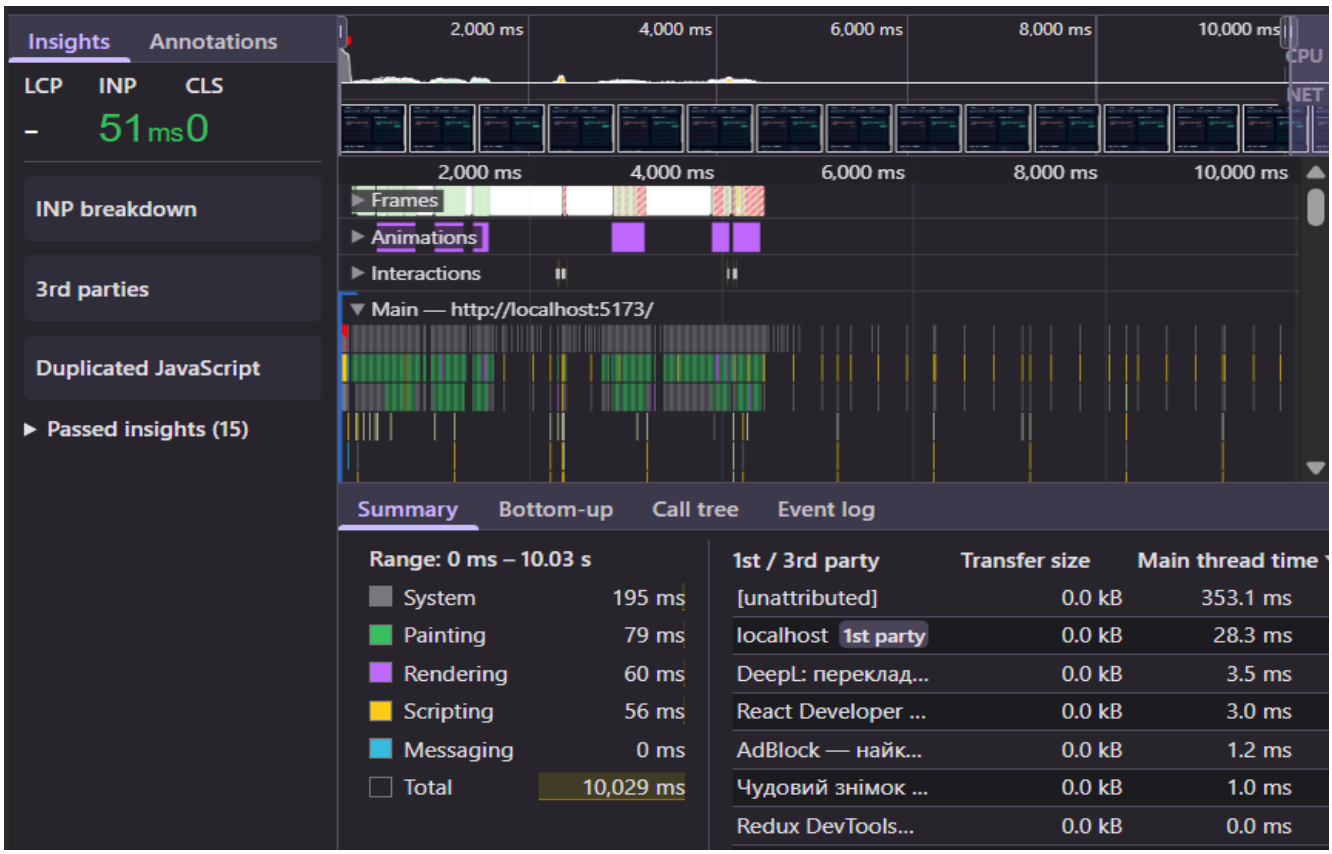

Рис.3.2 – Показники з ключами у вигляді унікальних ідентифікаторів

Як бачимо з результатів профілювання, показники часу виконання JavaScript (Scripting), рендерингу (Rendering) та малювання інтерфейсу (Painting) значно зменшились, що свідчить про ефективність даного методу оптимізації та позитивний вплив на роботу користувацького інтерфейсу.

3.2 Аналіз застосування React.lazy

Концепція даного методу оптимізації досить проста: відкласти завантаження усього, що потребує користувач прямо зараз. Цей метод можна використати до зображень, які користувач не бачить. При пролистуванні сторінки вниз, зображення будуть завантажені коли вони будуть в області видимості користувача.

Відкладене завантаження або ліниве завантаження, як інструмент оптимізації може мати великий вплив на продуктивність користувацького інтерфейсу. Цей підхід дозволяє зменшити час завантаження інтерфейсу та швидко надати можливість користувачеві взаємодіяти з ним. Варто звернути увагу на те, що використання цього методу потребує знання концепцій, які є ключовими для роботи, зокрема «розділення коду» та «бандлінг».

«Розділення коду» означає декомпозицію основної частини коду на невеликі частинки, що можуть існувати окремо одна від одної. «Бандлінг» - це процес об’єднання всього необхідного для роботи з інтерфейсу в одну абстракцію.

Для зменшення розміру бандлу або недопускання його розростанню, рекомендується на етапі проєктування робити декомпозицію елементів інтерфейсів для розділення їх на компоненти. Зробивши якісну декомпозицію, інструменти збірки, такі як Webpack або Vite, будуть створювати декілька бандлів, які будуть підвантажуватись динамічно в процесі взаємодії користувача з потрібними частинами інтерфейсу.

Розбиття інтерфейсу на компоненти дозволяє зручно керувати завантаженням цих елементів, які необхідні у конкретний момент користувачу. Це суттєво впливає на продуктивність інтерфейсу шляхом уникнення завантаження

тих елементів інтерфейсу які, можливо, навіть і не будуть використовуватись під час взаємодії.

Для проведення аналізу оптимізації шляхом React.lazy та розділення коду необхідно запустити інтерфейс в браузері та відкрити панель розробника Developer Tools. Далі ми переходимо на вкладку Lighthouse та натискаємо кнопку «Analyze page load», після чого отримуємо звіт з основними параметрами та їх значеннями.

Результати звітів у вигляді зображення представлені нижче.

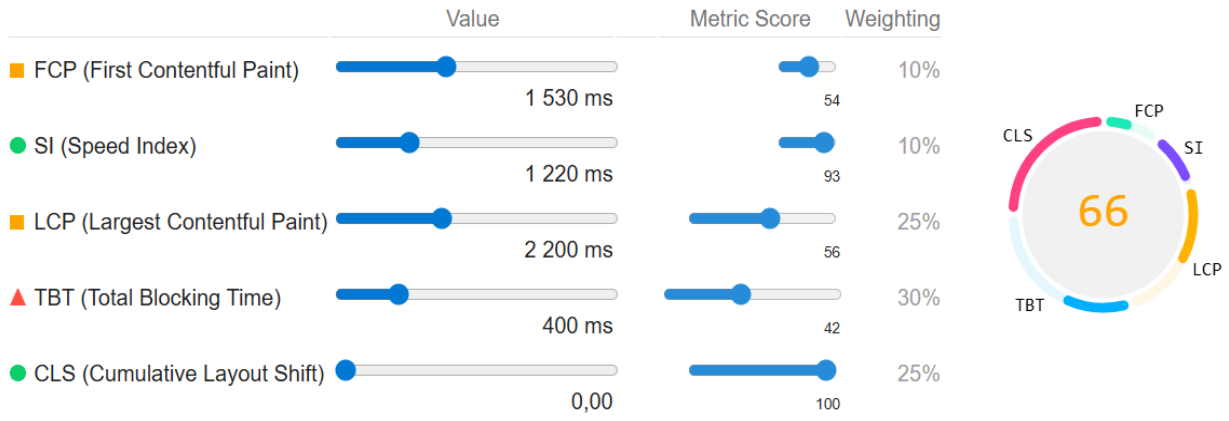

Рис.3.3 – Звіт без використання React.lazy

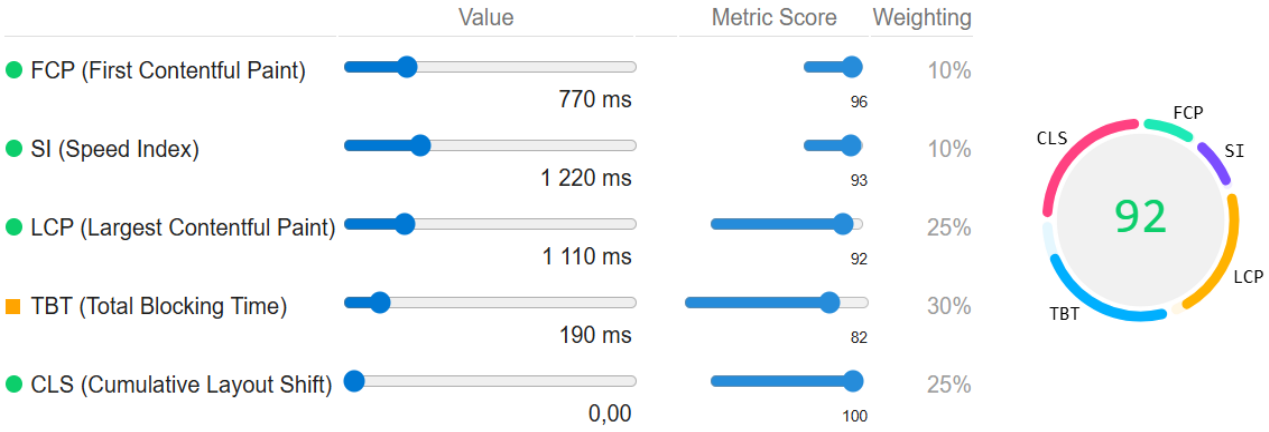

Рис.3.4 – Звіт з використання React.lazy

Як видно з даних представлених вище, застосування механізму React.lazy продемонструвало позитивний вплив на загальний показник Performance Score, збільшивши його з 66 до 92, що є суттєвим покращенням після оптимізації. Показники які зазнали покращень:

- First Contentful Paint (FCP) – час за який користувач побачить перші елементи інтерфейсу.
- Largest Contentful Paint (LCP) – час за який користувач побачить всю основну частину інтерфейсу
- Total Blocking Time (TBT) – час блокування головного потоку виконання JavaScript.

Зменшення показників FCP, LCP та TBT підкреслює ефективність оптимізації, а також позитивний вплив на продуктивність і швидкість користувацького інтерфейсу.

3.3 Аналіз застосування мемоізації

Мемоізація без сумніву є потужним методом підвищення продуктивності, але варто підкреслити, що вона не є універсальним інструментом, який просто вирішує всі проблеми. Невдале використання може призвести до погіршення показників продуктивності, тому варто виважено та розсудливо підходити до використання цього методу. Також треба взяти до уваги те, що сам процес мемоізації не є «безкоштовною» операцією і потребує певних ресурсів для своєї роботи.

Мемоізацію доцільно застосовувати у наступних випадках:

1. Компонент виконує важкі обчислення під час рендерингу.
2. Якщо батьківський елемент створюється постійно заново, а дочірні не змінюються.
3. Якщо дочірні компоненти мають залежність у вигляді функцій.
4. Є необхідність кешування складних структур даних у властивостях які передаються компонентам.

Для ефективного профілювання результатів оптимізації React.useMemo, React.memo, React.useCallback було використано інструмент Performance, який доступний з панелі розробника в браузері.

Для початку запису профілювання натискаємо «Record» і починаємо виконувати користувацькі дії з інтерфейсом, а саме змінюємо сортування, що буде

викликати у неоптимізованого інтерфейсу процес фільтрації при кожному рендерингу. Після завершення профілювання натискаємо кнопку «Stop». Повторюємо тест по 5 разів для оптимізованого та неоптимізованого інтерфейсу протягом 10 секунд.

Результати профайлингу додані у вигляді зображень нижче.

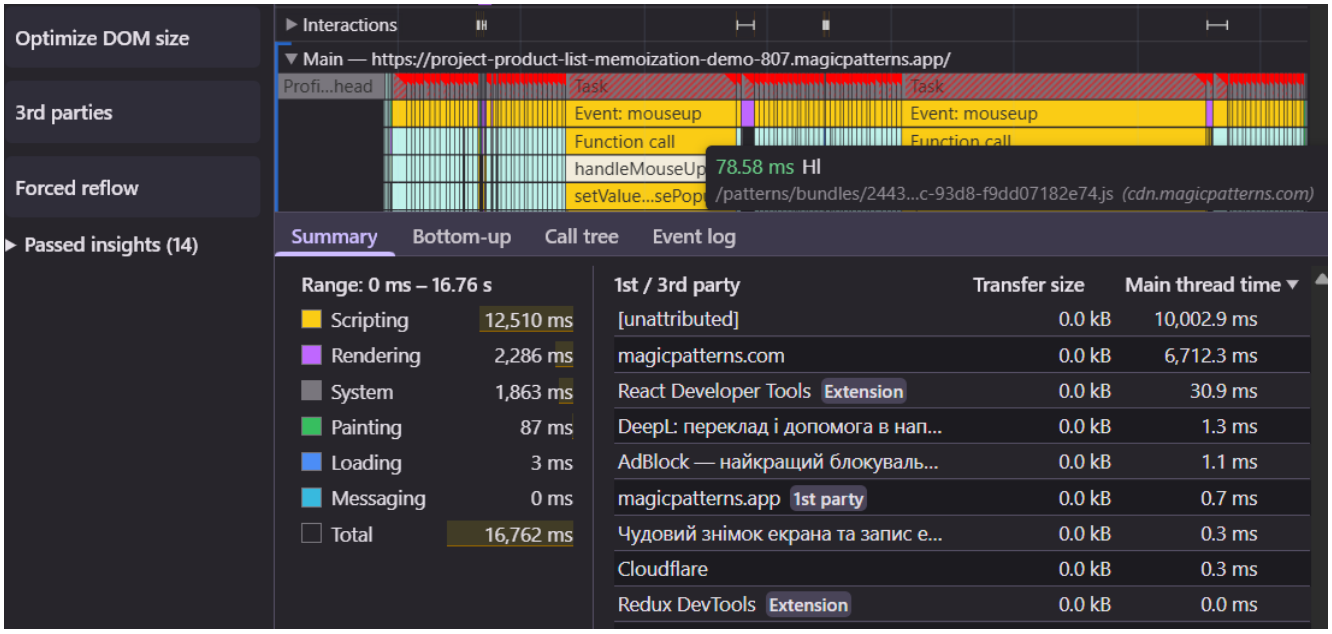

Рис.3.5 – Показники без використання React.useMemo

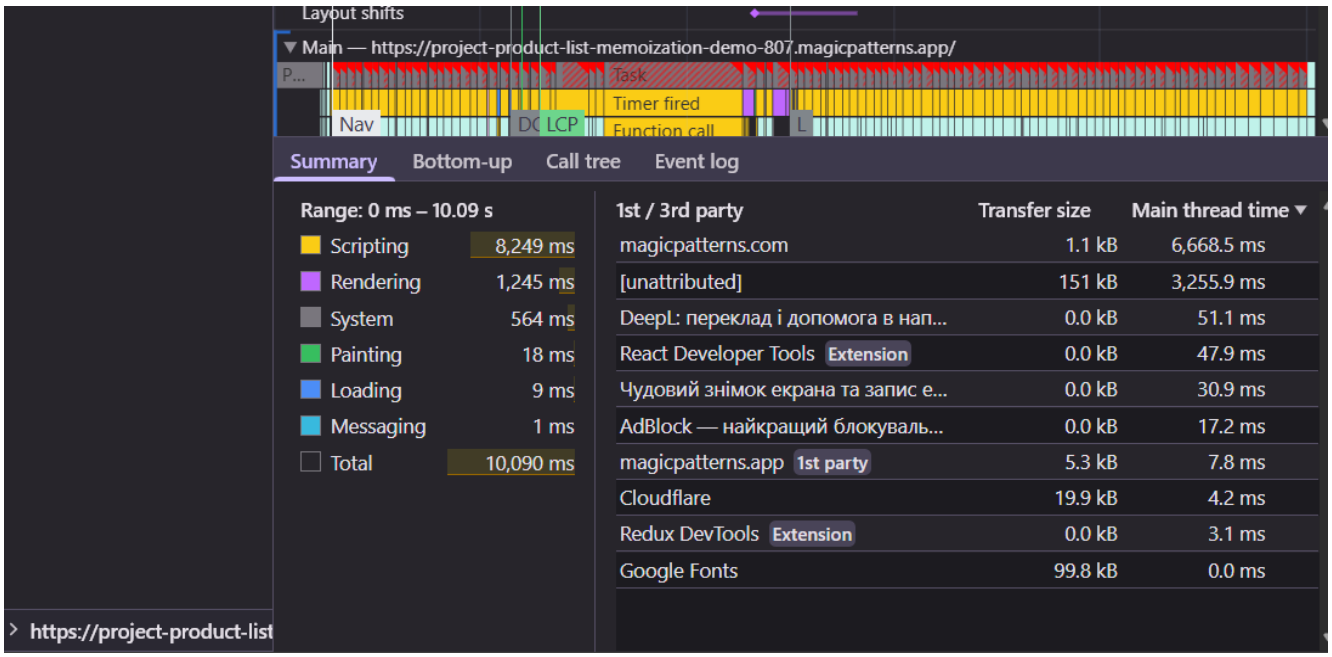

Рис.3.6 – Показники з використання React.useMemo

Як бачимо з результатів профілювання, застосування React.useMemo суттєво вплинуло на показники Rendering, System, Scripting, що свідчить про зменшення

часу виконання JavaScript та часу на рендеринг.

Повторимо профілювання вже з застосуванням React.useCallback для обгортання функції та React.memo. Ми обгорнемо елементи списку продуктів для уникнення надлишкових оновлень інтерфейсу. Знову натискаємо «Record» та повторюємо ті самі дії, що і в попередньому дослідженні, після чого натискаємо «Stop».

Результати профайлингу додані у вигляді зображень нижче.

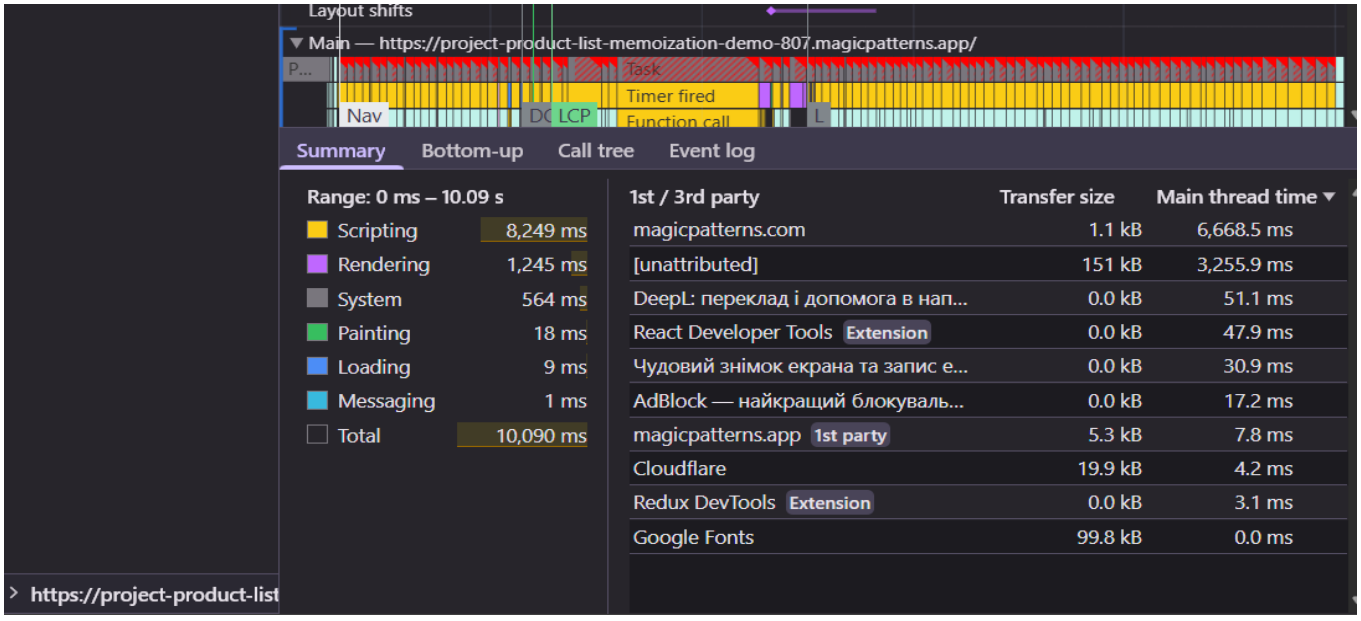

Рис.3.7 – Показники без використання React.memo та React.useCallback

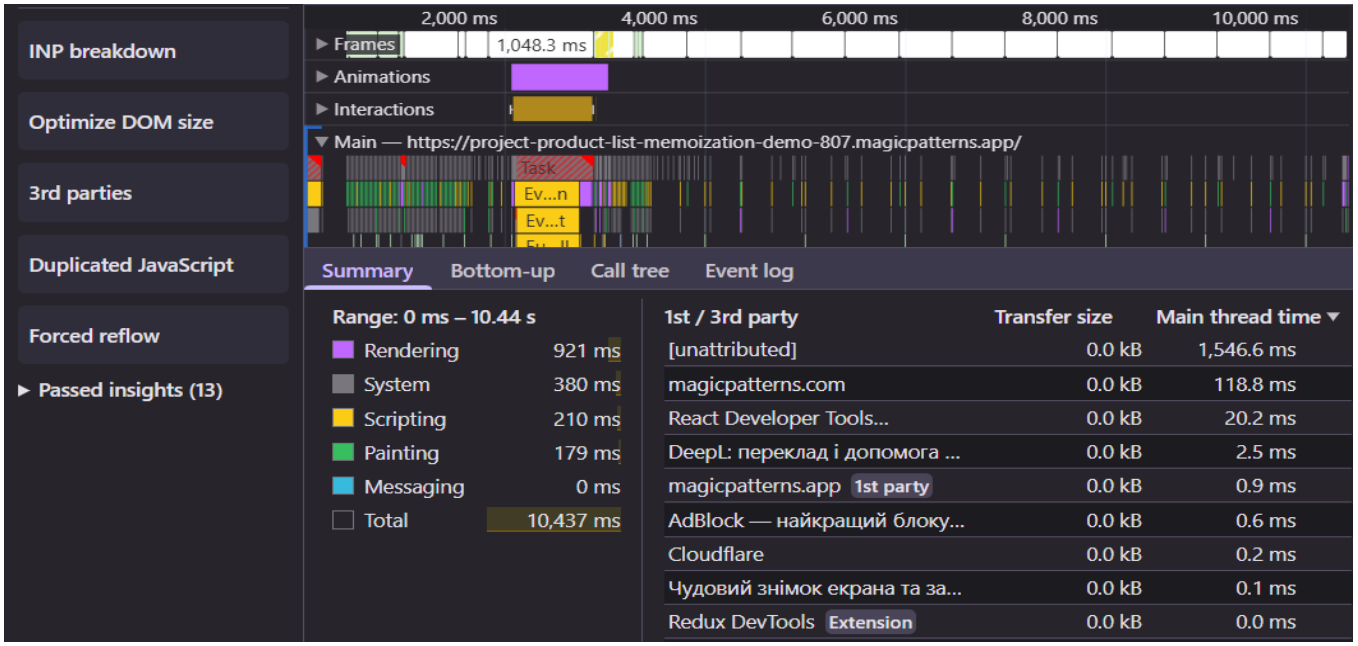

Рис.3.8 – Показники з використання React.memo та React.useCallback

Як можна побачити, застосування інструменту React.memo в парі з React.useCallback і винесенням елементу інтерфейсу в окремий компонент, суттєво знизило показник Scripting. React.useCallback допоміг уникнути створення нових посилань на функцію, яку ми передавали в компонент, а React.memo в свою чергу запобіг повторному рендерингу елемента. Таке поєднання інструментів значно вплинули на показники виконання потоку JavaScript коду.

3.4 Аналіз застосування useTransition

Основною ідеєю застосування useTransition є розділення оновлень користувацького інтерфейсу на термінові та нетермінові, що забезпечить збереження стабільної та плавної роботи клієнтської частини інформаційної системи.

useTransition доцільно застосовувати у наступних випадках:

1. Коли оновлення стану викликає оновлення великої кількості елементів інтерфейсу.
2. Коли під час взаємодії з інтерфейсом, інтерактив повинен приховати складні обчислення.
3. Якщо є складний механізм, який викликає зміну сторінок або режимів з відображенням великої кількості елементів.

Для аналізу ефективності застосування useTransition ми додали поле пошуку, яке ми будемо заповнювати і змінювати фільтри, щоб під час взаємодії виконувались складні обчислення.

Результати звітів у вигляді зображення представлені нижче.

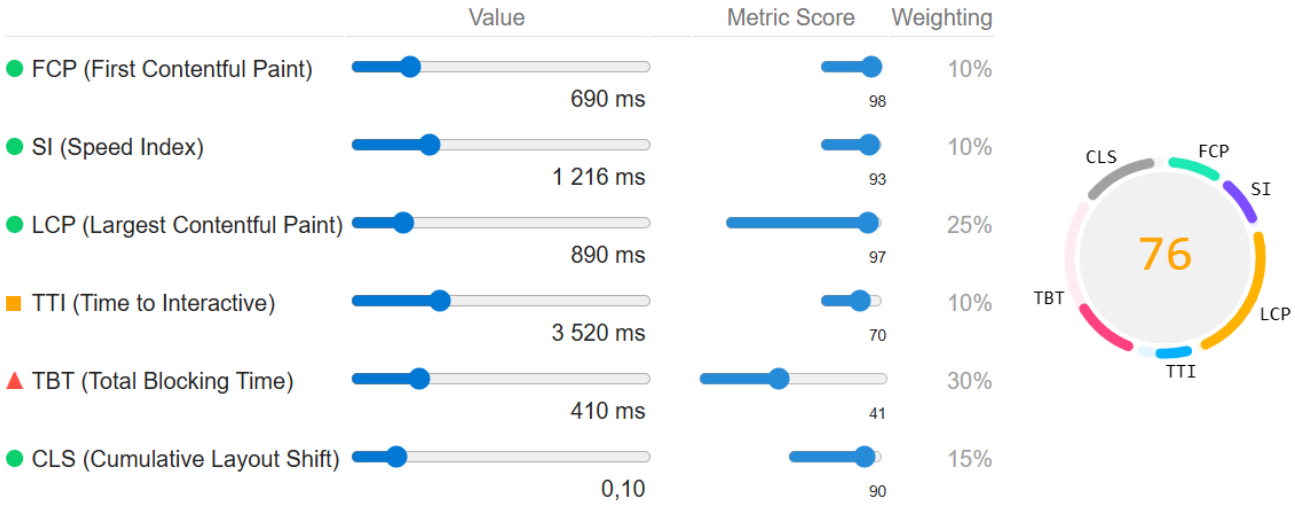

Рис.3.9 – Звіт без використання useTransition

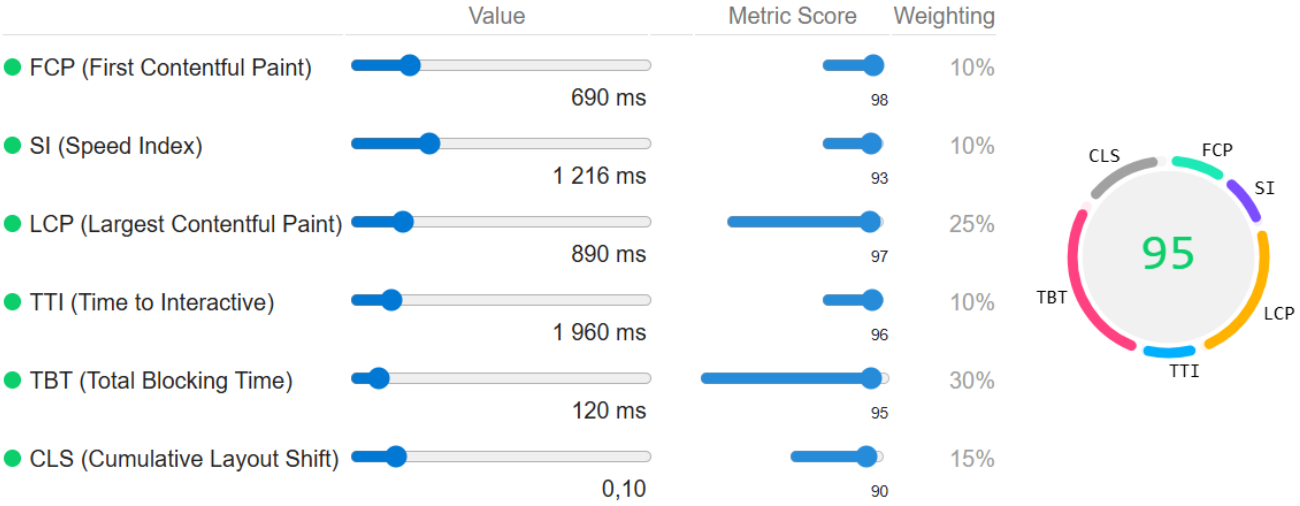

Рис.3.10 – Звіт з використання useTransition

Як видно з даних представлених вище, застосування механізму useTransition зменшило показники TTI (Time to Interactive) та TBT (Total Blocking Time). Зменшення часу блокування і готовності до взаємодії, призвело до покращення взаємодії користувача з інтерфейсом та збільшення швидкодії. React отримав можливість відкласти важкі обчислення та в першу чергу забезпечити взаємодію, а ресурсозатратні операції виконав у фоновому режимі.

3.5 Висновки

Даний розділ був присвячений комплексному аналізу методів і засобів оптимізації користувацького інтерфейсу, який фокусувався на таких основних методах та засобах: аналіз результатів оптимізації списків за допомогою ключів, використання відкладеного завантаження, використання мемоізації та розподілення пріоритетів оновлення.

Отримані результати підкреслюють значну перевагу оптимізації для покращення швидкодії та продуктивності користувацького інтерфейсу. Застосування стабільних ключів у динамічних списках значно зменшує кількість зайвих операцій рендерингу. Механізм відкладеного завантаження сприяв зменшенню початкового розміру бандлу, що безпосередньо вплинуло на покращення ключових метрик First Contentful Paint, Largest Contentful Paint та Total Blocking Time.

Використання методів мемоізації при правильному та доцільному застосуванні допомогли уникнути зайвих важких обчислень і повторних оновлень інтерфейсу. Транзиції допомогли уникнути погіршення швидкодії та зберегти виконання складних операцій непомітними для користувача.

Оптимізація з використанням сучасних та просунутих інструментів і методів дає можливість суттєво покращити продуктивність користувацьких інтерфейсів. Однак, необхідно брати до уваги специфіку інформаційної системи та розглядати кожен процес оптимізації в контексті конкретних завдань і можливостей.

ВИСНОВКИ

У рамках кваліфікаційної роботи було проведено дослідження методів і засобів оптимізації клієнтської частини, розробленої на базі React. Проаналізовано та застосовано на практиці такі методи та засоби, як: оптимізація списків за допомогою ключів, застосування відкладеного завантаження, застосування мемоізації, реалізація транзицій.

Застосування стабільних ключів у динамічних списках значно зменшує кількість зайвих операцій рендерингу, що дійсно важливо для продуктивності користувацького інтерфейсу. Відкладене завантаження в рази зменшує початковий розмір файлу, що прискорює завантаження інтерфейсу та скорочує час до першої взаємодії. Використання мемоізації забезпечує стабільну роботу та допомагає уникати надмірного навантаження на апаратну частину клієнта, шляхом уникнення зайвих рендерів та важких обчислень. Транзиції підвищили користувацький досвід у випадку, коли під час роботи інтерфейсу не можливо було уникнути вагомих обчислень та зміни інтерфейсу.

Серед перелічених методів і засобів, застосування транзицій та мемоізації, були найбільш затратними по часу та вимагали чіткого розуміння області застосування, але при цьому мали найбільший вплив на покращення продуктивності. Застосування відкладеного завантаження та реалізація оптимізації списків за допомогою ключів, не потребували великих зусиль, але вимагали певних засад для реалізації.

Глибокий аналіз проведеного дослідження методів оптимізації користувацького інтерфейсу підкреслив великий приріст продуктивності інтерфейсу, що перевищило будь-які очікування. Отримані результати можуть бути застосовані для оптимізації та покращення користувацького інтерфейсу як великих інформаційних систем, так і маленьких, які зіштовхнулися з проблемами продуктивності або ефективності.

Незважаючи на те, що React в базовій конфігурації має достатньо методів та засобів оптимізації, неправильне використання або недосконала архітектура

можуть призводити до неефективного використання ресурсів. Варто зазначити, що основними факторами успішної оптимізації є не тільки самі методи та засоби, але й чітке дотримання рекомендацій і правил, наданих в цьому дослідженні. Зважене та раціональне поєднання різних інструментів оптимізації відповідно до специфіки інформаційної системи та мети інтерфейсу, являється основою для досягнення високих результатів.

Слід підкреслити економічну складову оптимізації інтерфейсу, якою є зменшення витрат на подальшу підтримку та масштабування. Оскільки правильна архітектура і оптимізована система легше супроводжується, розширюється, адаптується до нових вимог. У довгостроковій перспективі такі підходи до покращення користувацького інтерфейсу можуть значно скоротити фінансове навантаження на розвиток інформаційної системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Співвідношення фреймворків для клієнтської частини з часом [Електронний ресурс]. – 2024. – Режим доступу: https://2024.stateofjs.com/ua-UA/libraries/front-end-frameworks/#front_end_frameworks_ratios (дата звернення: 17.11.2025).

2. Sommerville, I. Software Engineering. – 10th ed. – Pearson, 2015. — 816 p.

3. Pérez J., Garrigós I., Mazón J.-N. Modern Web Engineering: Design, Development and Measurement. — Springer, 2017. — 267 p.

4. Grigorik I. High Performance Browser Networking. — O'Reilly Media, 2018 (updated edition). — 420 p.

5. Kosmakov A. Front-End Architecture: A Modern Blueprint for Scalable and Maintainable Web Applications. – O'Reilly Media, 2020. — 412 p.

6. Пер. з англ. І. Бондар-Терещенко. – Харків, 2019. Чиста Архітектура: Мистецтво розроблення програмного забезпечення. – 368 с

7. Google. Core Web Vitals Overview. – Mountain View: Google Web.dev. [Електронний ресурс] – 2023. URL: https://web.dev/learn-core-web-vitals/ (дата звернення: 19.11.2025).

8. Пер. з англ. І. Бондар-Терещенко. – Харків, 2019. Чиста Архітектура: Мистецтво розроблення програмного забезпечення. – 368 с

9. OWASP Foundation. OWASP Top 10: The Ten Most Critical Web Application Security Risks. [Електронний ресурс]. URL: https://owasp.org/www-project-top-ten/Application_Security_Risks (дата звернення: 21.11.2025).

10. Mozilla Developer Network. HTML: HyperText Markup Language. – MDN Web Docs. [Електронний ресурс] – 2024. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (дата звернення: 21.11.2025).

11. JavaScrip. [Електронний ресурс] – URL: https://uk.wikipedia.org/wiki/JavaScript (дата звернення: 22.11.2025).

12. Mozilla Developer Network (MDN). CSS documentation. – Mozilla Foundation, [Електронний ресурс] – 2024. URL: https://developer.mozilla.org/en-

US/docs/Web/CSS (дата звернення: 23.11.2025).

13. Рейтинг мов програмування 2025.[Електронний ресурс] – 2025. URL: https://dou.ua/lenta/articles/language-rating-2025 (дата звернення: 24.11.2025).

14. Sullivan C., Hudson A. Front-End Development: The Big Nerd Ranch Guide. – New York: Big Nerd Ranch, 2020. – 312 p.

15. React library. [Електронний ресурс] – URL: https://beta.reactjs.org/ (дата звернення: 24.11.2025).

16. Ethan Holmes, Tom Bray. Getting Started with React Native — Published by Packt Publishing Ltd, 2015 — P. 129-172

17. Meta Platforms Inc. React Documentation. [Електронний ресурс] – 2024. URL: https://react.dev/ (дата звернення: 24.11.2025).

18. Understanding Reacts Virtual Dom. [Електронний ресурс] – 2025. URL: https://medium.com/@nishchay340/understanding-reacts-virtual-dom (дата звернення: 24.11.2025).

19. Meta Platforms Inc. React Documentation: Rendering and Reconciliation. [Електронний ресурс] – 2022. URL: https://react.dev/learn/render-and-commit (дата звернення: 25.11.2025).

20. Jackson R., Pierce M. React Router Documentation. — Remix Software. [Електронний ресурс] – 2023. URL: https://reactrouter.com (дата звернення: 25.11.2025).

21. Jstify community React. [Електронний ресурс] – 2022. URL: https://medium.com/@jstify.community/react (дата звернення: 04.12.2025).

22. Meta Open Source. Optimizing Performance – React Documentation – Menlo Park: Meta. [Електронний ресурс] – 2023. URL: https://react.dev/learn/optimizing-performance (дата звернення: 04.12.2025).

23. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. – Sebastopol: O'Reilly Media, 2020. – 350 p.

24. React-window Documentation. (2024). Overview. [Електронний ресурс]. – 2024. URL: https://react-window.vercel.app/#/examples/list/fixed-size (дата звернення: 11.12.2025).

25. Козлюк П. в. Розробка ефективного дискретного перетворення для потокової обробки / П. В. Козлюк // Прогресивні інформаційні технології в науці та освіті. Збірник наукових праць. – Вінниця, 2007.

26. Carlos R. React Cookbook: Create dynamic web apps with React using Redux, Webpack, Node.js, and GraphQL. — Packt Publishing, 2018. — 580 p.

27. Kyle Simpson. You Don't Know JS: Async & Performance — Published by O’Reilly Media, 2015 — P. 150-223 17. Shelley Powers. Learning JavaScript, 2nd Edition — Published by O’Reilly Media, 2008 — P. 120-154

ДОДАТОК А

ЛІСТИНГ КОДУ

```
import React from 'react';
import { ListOptimizationDemo } from './components/ListOptimizationDemo';
export function App() {
  return <div className="min-h-screen bg-slate-950 text-slate-200 font-sans selection:bg-blue-500/30">
    <ListOptimizationDemo />
  </div>;
}

import React, { Component } from 'react';
import { UnoptimizedList } from './UnoptimizedList';
import { OptimizedList } from './OptimizedList';
import { Info, Server, Zap, Activity } from 'lucide-react';
export function ListOptimizationDemo() {
 return <div className="max-w-7xl mx-auto p-4 md:p-8 space-y-12">

    <div className="text-center space-y-6 max-w-4xl mx-auto">



      <div className="bg-slate-900 border border-slate-800 rounded-xl p-6 md:p-8 relative overflow-hidden">
        <div className="absolute top-0 right-0 w-64 h-64 bg-blue-500/5 rounded-full blur-3xl -mr-16 -mt-16 pointer-events-none" />

        <div className="grid md:grid-cols-3 gap-8 relative z-10">
         <div className="space-y-3">
```

```
          <div className="space-y-3">
            <div className="flex items-center gap-2 text-slate-100 font-semibold">
              <Zap className="h-5 w-5 text-emerald-500" />
              <h3>The Solution</h3>
            </div>
            <p className="text-slate-400 text-sm leading-relaxed">
              Stable <code>unique IDs</code> allow React to precisely identify
              changed items. Only the specific DOM nodes that changed are
              updated, keeping the UI buttery smooth.
            </p>
          </div>
        </div>
      </div>


      <div className="grid md:grid-cols-2 gap-8 items-start">
        <UnoptimizedList />
        <OptimizedList />
      </div>


    </div>;
}
import React, { useState } from "react";
import { Trash2, Plus, CheckCircle } from "lucide-react";
import { Button } from "./ui/Button";
import { Card } from "./ui/Card";
const ListItem = ({
```

```
  name,
  onDelete,
 }: {
  name: string;
  onDelete: () => void;
 }) => {
  return (
   <div className="flex items-center gap-3 p-3 bg-emerald-950/10 border border-emerald-900/30 rounded-lg animate-in fade-in slide-in-from-bottom-2 duration-300 group hover:border-emerald-900/50 transition-colors">
    <div className="flex-1">
     <div className="font-medium text-emerald-400 mb-1">{name}</div>

     <input
      className="w-full text-sm p-2 rounded border border-emerald-900/30 bg-slate-950/50 text-slate-200 placeholder:text-emerald-900/40 focus:outline-none focus:border-emerald-500/50 focus:ring-1 focus:ring-emerald-500/20 transition-all"
      placeholder="Type something here..."
     />
    </div>
    <Button
     variant="ghost"
     size="sm"
     onClick={onDelete}
     className="text-emerald-500 hover:bg-emerald-950/30 hover:text-emerald-400"
     aria-label="Delete item"
    >
     <Trash2 className="h-4 w-4" />
    </Button>
```

```
    </div>
  );
};
export function OptimizedList() {
  const [items, setItems] = useState([
    {
      id: "1",
      text: "User A",
    },
    {
      id: "2",
      text: "User B",
    },
    {
      id: "3",
      text: "User C",
    },
  ]);
  const addItem = () => {
    const newId = Math.random().toString(36).substr(2, 9);
    setItems((prev) => [
      {
        id: newId,
        text: `User ${String.fromCharCode(65 + prev.length)}`,
      },
      ...prev,
    ]);
  };
  const deleteItem = (idToDelete: string) => {
    setItems((prev) => prev.filter((item) => item.id !== idToDelete));
```

```
  };
  return (
    <Card
      title="✅ Optimized List (Unique IDs)"
      description="Using stable IDs for precise item identification."
      className="h-full border-emerald-900/30 shadow-emerald-900/5"
    >
      <div className="space-y-4">


        <div className="flex justify-end">
          <Button
            onClick={addItem}
            variant="primary"
            size="sm"
              className="bg-emerald-600 hover:bg-emerald-500 focus-visible:ring-
emerald-600"
            leftIcon={<Plus className="h-4 w-4" />}
          >
            Add User to Top
          </Button>
        </div>

        <div className="space-y-2 max-h-[400px] overflow-y-auto pr-2 custom-
scrollbar">
          {items.map((item) => (
            <ListItem
              key={item.id}
              name={`${item.text} (ID: ${item.id.substr(0, 4)})`}
              onDelete={() => deleteItem(item.id)}
```

```
            />
          ))}
          {items.length === 0 && (
            <div className="text-center py-8 text-slate-600 text-sm italic">
              List is empty
            </div>
          )}
        </div>
      </div>
    </Card>
  );
}

import React, { useState } from "react";
import { Trash2, Plus, AlertTriangle } from "lucide-react";
import { Button } from "./ui/Button";
import { Card } from "./ui/Card";
const ListItem = ({
  name,
  onDelete,
}: {
  name: string;
  onDelete: () => void;
}) => {
  return (
    <div className="flex items-center gap-3 p-3 bg-red-950/10 border border-red-900/30 rounded-lg animate-in fade-in slide-in-from-bottom-2 duration-300 group hover:border-red-900/50 transition-colors">
      <div className="flex-1">
        <div className="font-medium text-red-400 mb-1">{name}</div>
```

```
            <input
              className="w-full text-sm p-2 rounded border border-red-900/30 bg-slate-950/50 text-slate-200 placeholder:text-red-900/40 focus:outline-none focus:border-red-500/50 focus:ring-1 focus:ring-red-500/20 transition-all"
              placeholder="Type something here..."
            />
          </div>
          <Button
            variant="ghost"
            size="sm"
            onClick={onDelete}
            className="text-red-500 hover:bg-red-950/30 hover:text-red-400"
            aria-label="Delete item"
          >
            <Trash2 className="h-4 w-4" />
          </Button>
        </div>
      );
    };
    export function UnoptimizedList() {
      const [items, setItems] = useState(["User A", "User B", "User C"]);
      const addItem = () => {
        setItems((prev) => [
          `User ${String.fromCharCode(65 + prev.length)}`,
          ...prev,
        ]);
      };
      const deleteItem = (indexToDelete: number) => {
        setItems((prev) => prev.filter((_, index) => index !== indexToDelete));
```

```
  };
  return (
    <Card
      title="❌ Unoptimized List (Index Keys)"
      description="Simulating a large list where index is used as key."
      className="h-full border-red-900/30 shadow-red-900/5"
    >
      <div className="space-y-4">


        <div className="flex justify-end">
          <Button
            onClick={addItem}
            variant="secondary"
            size="sm"
            leftIcon={<Plus className="h-4 w-4" />}
          >
            Add User to Top
          </Button>
        </div>

        <div className="space-y-2 max-h-[400px] overflow-y-auto pr-2 custom-scrollbar">
          {items.map((item, index) => (
            <ListItem
              key={index}
              name={`${item} (Index: ${index})`}
              onDelete={() => deleteItem(index)}
            />
          ))}
```

```
            {items.length === 0 && (
              <div className="text-center py-8 text-slate-600 text-sm italic">
                List is empty
              </div>
            )}
          </div>
        </div>
      </Card>
    );
}

import React from 'react';
interface CardProps {
  children: React.ReactNode;
  className?: string;
  title?: string;
  description?: string;
  footer?: React.ReactNode;
}
export function Card({
  children,
  className = '',
  title,
  description,
  footer
}: CardProps) {
    return <div className={`bg-slate-900 rounded-xl border border-slate-800 shadow-xl overflow-hidden ${className}`}>
        {(title || description) && <div className="px-6 py-4 border-b border-slate-800">
```

```
                    {title && <h3 className="text-lg font-semibold text-slate-100">{title}</h3>}
                    {description && <p className="mt-1 text-sm text-slate-400">{description}</p>}
                  </div>}
                <div className="p-6">{children}</div>
                {footer && <div className="px-6 py-4 bg-slate-900/50 border-t border-slate-800">
                    {footer}
                  </div>}
              </div>;
            }
            import React, { useId } from 'react';
            interface InputProps extends React.InputHTMLAttributes<HTMLInputElement> {
              label?: string;
              error?: string;
            }
            export function Input({
              className = '',
              label,
              error,
              id,
              ...props
            }: InputProps) {
              const inputId = id || useId();
              return <div className="w-full">
                  {label && <label htmlFor={inputId} className="block text-sm font-medium text-slate-400 mb-1">
                      {label}
```

```
            </label>}
          <input id={inputId} className={`flex h-10 w-full rounded-md border border-slate-700 bg-slate-950 px-3 py-2 text-sm text-slate-100 placeholder:text-slate-600 focus:outline-none focus:ring-2 focus:ring-blue-500 focus:border-transparent disabled:cursor-not-allowed disabled:opacity-50 transition-colors ${error ? 'border-red-500 focus:ring-red-500' : ''} ${className}`} {...props} />
            {error && <p className="mt-1 text-xs text-red-400">{error}</p>}
          </div>;
        }
        import React from 'react';
        import { Loader2 } from 'lucide-react';
        interface ButtonProps extends React.ButtonHTMLAttributes<HTMLButtonElement> {
          variant?: 'primary' | 'secondary' | 'danger' | 'ghost';
          size?: 'sm' | 'md' | 'lg';
          isLoading?: boolean;
          leftIcon?: React.ReactNode;
        }
        export function Button({
          className = '',
          variant = 'primary',
          size = 'md',
          isLoading = false,
          leftIcon,
          children,
          disabled,
          ...props
        }: ButtonProps) {
           const baseStyles = 'inline-flex items-center justify-center rounded-md font-medium transition-all focus-visible:outline-none focus-visible:ring-2 focus-visible:ring-
```

```
offset-2 focus-visible:ring-offset-slate-950 disabled:opacity-50 disabled:pointer-events-none';
        const variants = {
          primary: 'bg-blue-600 text-white hover:bg-blue-500 focus-visible:ring-blue-600 shadow-lg shadow-blue-900/20',
          secondary: 'bg-slate-800 text-slate-200 hover:bg-slate-700 focus-visible:ring-slate-500 border border-slate-700',
          danger: 'bg-red-600 text-white hover:bg-red-500 focus-visible:ring-red-600 shadow-lg shadow-red-900/20',
          ghost: 'hover:bg-slate-800 text-slate-400 hover:text-slate-200'
        };
        const sizes = {
          sm: 'h-8 px-3 text-xs',
          md: 'h-10 px-4 py-2 text-sm',
          lg: 'h-12 px-6 text-base'
        };
        return <button className={`${baseStyles} ${variants[variant]} ${sizes[size]} ${className}`} disabled={disabled || isLoading} {...props}>
            {isLoading && <Loader2 className="mr-2 h-4 w-4 animate-spin" />}
            {!isLoading && leftIcon && <span className="mr-2">{leftIcon}</span>}
            {children}
          </button>;
      }
```