Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів оптимізації та архітектурних рішень для підвищення продуктивності та масштабованості застосунків на основі React
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-1

Анатолій ФІЛІПЕНКО
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Наталія ГОЛЯН
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачу Філіпенку Анатолію Вадимовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів оптимізації та архітектурних рішень для підвищення продуктивності та масштабованості застосунків на основі React»

Затверджена наказом по університету від 15.04. 2025р. № 290 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 19.06.2025
3. Вихідні дані до роботи дослідження методів оптимізації рендерингу компонентів у React, аналіз архітектурних підході, впровадження сучасних технологій Next.js для реалізації SSR, SSG, CSR, а також оцінка ефективності оптимізацій за допомогою метрик продуктивності і порівняння продуктивності між оптимізованою та не оптимізованою версіями застосунку
4. Перелік питань, що потрібно опрацювати в роботі
огляд існуючих архітектурних рішень, дослідження технік оптимізації рендерингу, вибір релевантних інструментів для реалізації та оцінки оптимізацій, розробка двох варіантів тестових застосунків з оптимізаціями та без

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	15.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	26.04.2025	*виконано*
3	Огляд й аналіз літературних, наукових джерел	01.05.2025	*виконано*
4	Теоретичне дослідження	10.05.2025	*виконано*
5	Проведення експериментального дослідження	20.05.2025	*виконано*
6	Підготовка до апробації результатів дослідження. Публікація матеріалів	03.06.2025	*виконано*
7	Підготовка пояснювальної записки	04.06.2025	*виконано*
8	Підготовка презентації та доповіді	08.06.2025	*виконано*
9	Перевірка на плагіат	11.06.2025	*виконано*
10	Нормоконтроль	11.06.2025	*виконано*
11	Рецензування	15.06.2025	*виконано*
12	Попередній захист	16.06.2025	*виконано*
13	Занесення диплома в електронний архів	17.05.2025	*виконано*
14	Допуск до захисту у зав. кафедри	18.05.2025	*виконано*

Дата видачі завдання 15 квітня 2025р.

Студент (ка) _____ Анатолій ФІЛІППЕНКО
(підпис)

Керівник роботи _____ доц. Наталія ГОЛЯН
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 84 с., 42 рис., 1 табл., 16 джерел.

ВІРТУАЛІЗАЦІЯ СПИСКІВ, МЕМОІЗАЦІЯ, ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ, РОЗРОБКА ВЕБ-ЗАСТОСУНКІВ, УПРАВЛІННЯ СТАНОМ, CSR, LAZY LOADING, LIGHTHOUSE, NEXT.JS, REACT, SSG, SSR, TANSTACK QUERY, WEB VITALS.

Об’єктом дослідження є застосунки на основі React з використанням фреймворку Next.js.

Метою роботи є аналіз та застосування на практиці методів оптимізації та архітектурних рішень застосунків на основі React, виявлення їхніх переваг і недоліків, а також визначення випадків, у яких вони застосовуються для підвищення продуктивності та масштабованості застосунків.

Методи дослідження – аналіз існуючих методів оптимізації продуктивності веб-застосунків (мемоізація, віртуалізація списків, lazy loading, SSR/CSR/SSG), дослідження інструментів керування станом і мережевими. Розробка двох версій веб-застосунку з і без оптимізацій для порівняння. Впровадження динамічного імпорту компонентів, віртуалізації великих списків, оптимізації запитів та архітектурних рішень на основі Next.js. Проведення тестування з використанням інструменту Lighthouse та аналіз метрик Web Vitals для об’єктивної оцінки ефективності застосованих оптимізацій.

У результаті кваліфікаційної роботи було проаналізовано архітектурні рішення та методи оптимізації застосунків на базі React, які включають мемоізацію, віртуалізацію списків, lazy loading, різні стратегії рендерингу, такі як SSR, SSG та CSR, а також методи дослідження продуктивності веб-застосунків, застосовано ці рішення та методи на практиці й експериментально підтверджено їхню ефективність.

VIRTUALIZATION OF LISTS, MEMOIZATION, PERFORMANCE OPTIMIZATION, WEB APPLICATION DEVELOPMENT, STATE MANAGEMENT, CSR, LAZY LOADING, LIGHTHOUSE, NEXT.JS, REACT, SSG, SSR, TANSTACK QUERY, WEB VITALS.

The object of study is React-based applications using the Next.js framework.

The purpose of the study is to analyze and apply in practice optimization methods and architectural solutions for React-based applications, identify their advantages and disadvantages, and determine the cases in which they are used to improve application performance and scalability.

Research methods - analysis of existing methods for optimizing the performance of web applications (memoization, list virtualization, lazy loading, SSR/CSR/SSG), research of state and network management tools. Development of two versions of the web application with and without optimizations for comparison. Implementation of dynamic component import, large list virtualization, query optimization, and architectural solutions based on Next.js. Conducting testing using the Lighthouse tool and analyzing Web Vitals metrics to objectively evaluate the effectiveness of the applied optimizations.

As a result of the research and experiments, we analyzed architectural solutions and methods for optimizing React-based applications, including memoization, list virtualization, lazy loading, various rendering strategies such as SSR, SSG, and CSR, as well as methods for studying the performance of web applications, applied these solutions and methods in practice, and experimentally confirmed their effectiveness.

Завідувачу кафедри ПІ
проф. Кирилу СМЕЛЯКОВУ

ЗАЯВА

щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Філіпенко Анатолій Вадимович, здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-1 кафедра програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження методів оптимізації та архітектурних рішень для підвищення продуктивності та масштабованості застосунків на основі React», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

11.06.2025  Анатолій ФІЛІПЕНКО

ЗМІСТ

ВСТУП

Сучасні веб-застосунки стають дедалі складнішими через постійне зростання вимог до їхньої функціональності, продуктивності та зручності для кінцевого користувача. У цьому контексті бібліотека React відіграє важливу роль завдяки своїй гнучкості, продуктивності та широкому застосуванню у розробці інтерфейсів користувача. Водночас збільшення масштабів веб-застосунків, ускладнення їх архітектури та необхідність забезпечення швидкодії ставлять перед розробниками численні виклики, що зумовлюють актуальність пошуку і впровадження оптимізаційних методів та архітектурних рішень.

Основною метою роботи є аналіз та застосування на практиці методів оптимізації та архітектурних рішень застосунків на основі React, виявлення їхніх переваг і недоліків, а також визначення випадків, у яких вони застосовуються для підвищення продуктивності та масштабованості застосунків.

Будуть розглянуті такі методи оптимізації як мемоізація, ліниве завантаження, віртуалізація списків і динамічне завантаження модулів. Важливим також є дослідження сучасних архітектурних підходів, які сприяють підвищенню продуктивності та ефективності розробки, включаючи розподіл функціональності між компонентами та використання сучасних інструментів керування станом. Крім того, значну увагу приділено фреймворку Next.js, який реалізує серверний рендеринг (SSR), статичну генерацію сайтів (SSG) та клієнтський рендеринг (CSR).

Об’єктом дослідження є застосунки на основі React, до яких буде застосовано оптимізацію рендерингу та які використовуватимуть сучасні архітектурні підходи і швидку обробку великих даних.

Предметом дослідження є методи оптимізації продуктивності та архітектурні рішення для розробки сучасних веб-застосунків на основі бібліотеки React, з урахуванням використання фреймворку Next.js та інших технологій, що дозволяють зменшити час завантаження, оптимізувати використання ресурсів і підвищити масштабованість застосунків.

Для досягнення поставленої мети будуть використовуватися різноманітні методи дослідження. Теоретична частина включатиме аналіз літератури та

сучасних наукових публікацій на тему оптимізації веб-застосунків і архітектурних підходів. Практична частина базуватиметься на емпіричних перевірках і дослідженнях продуктивності двох версій застосунків – із впровадженими оптимізаціями та без них. Аналіз результатів буде виконуватися за допомогою інструменту Lighthouse, який дозволяє отримати точні дані про ключові показники роботи застосунку. Крім того, можуть проводитися експерименти для оцінки впливу оптимізацій на швидкість завантаження, споживання ресурсів і стабільність роботи.

У результаті роботи будуть визначені переваги та недоліки кожного з методів оптимізації, описано в якому випадку який з методів оптимізації використовувати, а також визначено оптимальний архітектурний підхід для побудови застосунку. Використання цих методів та архітектурних підходів дозволить підвищити швидкодію застосунків, знизити затрати на їхню підтримку та зробити їх більш зручними для кінцевих користувачів. Результати дослідження можуть бути використані у створенні веб-продуктів різного масштабу, від невеликих інформаційних сайтів до складних корпоративних систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

У сучасних веб-застосунках відчутно зростає потреба в ефективному керуванні даними, підвищенні продуктивності інтерфейсу та забезпеченні швидкого часу відгуку через високу швидкість росту технологій і вимог. Однією з провідних бібліотек спрямованих на створення інтерфейсів користувача є React. Ця JavaScript-бібліотека, розроблена компанією Meta (раніше Facebook) [1], не лише змінила звичні підходи до менеджменту стану і рендерингу, але й дала поштовх до формування цілої екосистеми різноманітних супутніх інструментів та архітектурних патернів.

React набрав популярність завдяки своїй концепції компонентно-орієнтованого підходу. Компоненти в React представляють собою логічно та візуально відокремлені частини інтерфейсу, що дозволяє розробникам розбивати складні інтерфейси на невеликі, легші для розуміння та підтримки блоки коду. Такий підхід давно застосовується в інших фреймворках, проте у випадку з React він був поєднаний із принципом декларативності та використанням віртуального DOM. Virtual DOM – це технологія, що використовується в React для оптимізації оновлення інтерфейсу, створюючи його копію в пам’яті та змінюючи лише ті елементи, які зазнали оновлення. Це дало змогу мінімізувати кількість дорогих маніпуляцій з DOM, що, в свою чергу, сприяло оптимізації продуктивності у великих проєктах.

Порівняно з іншими популярними фреймворками та бібліотеками, такими як Angular чи Vue, React є більш гнучким та менш монолітним. Якщо Angular пропонує відразу повноцінне готові рішення, наприклад, шаблонізацію, роутинг, модулі, DI тощо, то React дає лише ядро для створення інтерфейсу. Ця відсутність жорстких рамок дозволяє вибирати інструменти під конкретні потреби, однак вона ж і породжує виклики пов’язані з формуванням однієї цілісної архітектури застосунку. Так само Vue часто розглядають як більш простий у засвоєнні підхід завдяки його інтегрованим можливостям та чіткій структурі. React же, який є

бібліотекою, надає більше свободи, а разом із тим і більше відповідальності за вибір архітектурних та оптимізаційних патернів.

Оптимізація React-застосунків є багатогранним завданням. З одного боку, існує рівень оптимізації на рівні компонентів, де йдеться про уникнення зайвих перерендерів, мемоізацію функцій та значень, а також правильне використання ключів у списках. З іншого боку, існує оптимізація на рівні архітектури – вибір менеджера стану, розподіл логіки між компонентами, використання серверного рендерингу (SSR) або статичної генерації (SSG) для поліпшення швидкості першого відображення сторінки. Тут особливу роль відіграють фреймворки по типу Next.js, які базуються на React, але надають додаткові можливості для оптимізації, зокрема попереднє завантаження сторінок, статичну генерацію або змішану архітектуру рендерингу.

Next.js, як надбудова над React, став яскравим прикладом тенденції до спрощення оптимізаційних рішень. Замість того, щоб вручну конфігурувати webpack, який є інструментом для збору та оптимізації фронтенд-коду, налаштовувати серверний рендеринг і турбуватися про SEO, розробникам пропонується готове рішення з підтримкою гібридних методів рендерингу [2]. Такий підхід значно полегшує інтеграцію із системами кешування, CDN, а також підвищує продуктивність для кінцевого користувача. Це відповідає загальній тенденції надавати більш узгоджені, гнучкі та зручні у використанні середовища, які спрощують складні процеси оптимізації.

Не можна оминути увагою й сучасні тенденції у напрямку мікрофронтендів (Microfrontends), які також впливають на архітектурні рішення React-застосунків. Вони дозволяють розділяти великі, монолітні застосунки на дрібніші, автономні частини. Це підвищує масштабованість проєкту, полегшує поступову міграцію між технологіями та пришвидшує час розробки завдяки можливості роботи окремих команд над відокремленими частинами інтерфейсу. Однак реалізація мікрофронтендів за допомогою React вимагає дотримання певних архітектурних патернів: чіткого розмежування меж застосунку, організації передавання стану та залежностей, а також забезпечення узгодженого користувацького досвіду між

компонентами, які можуть бути реалізовані на різних фреймворках чи різних версіях React.

Щодо підходів до оптимізації, серед найбільш поширених варто відзначити мемоізацію та покращення стратегій розподілу логіки. Іноді оптимізація може полягати в застосуванні бібліотек для віртуалізації списків з метою рендерингу великої кількості елементів без непотрібного навантаження на DOM. Іншим методом є code splitting – розбиття коду застосунку на окремі бандли, тобто окремі частини коду, що завантажуються за потребою. Це дозволяє прискорити початковий час завантаження сторінки та зменшити обсяг даних, необхідний для початку роботи з інтерфейсом [3].

Lazy loading є ще одним важливим механізмом оптимізації React-застосунків, що дозволяє відкладати завантаження частин коду або ресурсів до моменту, коли вони дійсно знадобляться. За допомогою такого підходу забезпечується скорочення початкового часу завантаження застосунку, оскільки користувач отримує лише мінімальний набір необхідного коду, а додаткові модулі завантажуються динамічно у процесі взаємодії. Lazy loading також ефективний для оптимізації зображень, відео та інших важких ресурсів, які можуть бути завантажені лише тоді, коли користувач прокручує сторінку до відповідного місця. Недоліком цього підходу може бути затримка у відображенні контенту через додатковий час на завантаження ресурсів. Утім, використання попереднього завантаження (prefetching) та місцевого кешування може зменшити цей негативний ефект і забезпечити більш плавний користувацький досвід.

Також важливе місце у оптимізації займає серверний рендеринг (SSR). Застосування SSR покращує індексацію сторінок пошуковими системами, скорочує час до першого відображення контенту та робить застосунки більш доступними. Next.js спрощує використання SSR, дозволяючи поєднувати його з клієнтським рендерингом та забезпечуючи високий рівень продуктивності без необхідності ручної оптимізації кожного етапу. Окрім того, серверний рендеринг відкриває шлях до використання гібридних технік, таких як інкрементальна

статична генерація (ISR), що дає змогу постійно оновлювати статично згенеровані сторінки без повного перегенерування усього застосунку.

У контексті архітектури та оптимізації важливим аспектом стає керування станом. Використання React Hooks та Redux дозволяє ефективно оптимізувати роботу із даними, зменшуючи кількість повторних рендерів та забезпечуючи чітке розмежування логіки роботи із станом [4]. Хоча Redux тривалий час був найрозповсюдженішим та виступав стандартом, останнім часом з'являються альтернативи, такі як Recoil, Zustand або Jotai, що пропонують більш простий підхід до управління даними. Як і Redux, вони дозволяють зменшити кількість повторних рендерів, поліпшити масштабованість та чіткіше розмежувати логіку роботи із даними. Нові менеджери стану часто легші для розуміння й інтеграції, проте уподобання розробника сильно впливають на вибір технології керування станом. Він значною мірою залежить від розміру проєкту, його специфіки та вимог, адже кожен інструмент має свої переваги і сфери застосування.

Перспективи розвитку React та суміжних технологій пов'язані з подальшою стандартизацією інструментів, появою нових підходів до побудови інтерфейсів, розширенням можливостей фреймворків типу Next.js, а також з інтеграцією сучасних платформ для розгортання та хостингу. Виклики що постають – це оптимізація рендерингу на рівні підкомпонентів, автоматичний аналіз продуктивності та впровадження інноваційних рішень. Слід очікувати збільшення інновацій у сфері менеджментів стану, можливо, із більш прямим використанням потоків даних (Streams) або реактивних парадигм.

1.2 Актуальність дослідження

Серед проблем, з якими стикаються розробники React-застосунків, можна виділити складнощі з продуктивністю при великій кількості компонентів, часто змінюваних даних та складних обчисленнях. Навіть віртуальний DOM не може повністю компенсувати нераціональну логіку оновлення стану. Надмірна кількість перерендерів, викликана необережним використанням контекстів або невірно застосованими хуками, може призвести до відчутної втрати продуктивності. Також

проблемою часто стає неправильне структурування даних: коли стан зберігається на занадто високому рівні і кожна незначна зміна змушує перемальовувати великі частини інтерфейсу, час відгуку зростає, а взаємодія з користувачем уповільнюється.

Загалом, сучасний стан React свідчить про те, що він продовжує еволюціонувати, стаючи більш гнучким та універсальним інструментом. Паралельно з цим зростають вимоги до продуктивності, тому оптимізація не може бути просто другорядним завданням. Тенденція створення складних багатофункціональних застосунків змушує розробників більш усвідомлено обирати архітектурні патерни, а також піклуватися про те, щоб кожна взаємодія з користувачем була максимально швидкою та інтуїтивною. Водночас на ринку з’являються інструменти, спрямовані на автоматизацію деяких оптимізаційних рішень, як, наприклад, покращені збірники (build tools), загальнодоступні пакети із готовими оптимізованими компонентами та просунуті засоби аналізу продуктивності.

Першочергово постає задача проєктування архітектури. Тут можна виділити ряд критеріїв, яким повинен відповідати застосунок:

- масштабованість: при зростанні функціональних можливостей виникає потреба в додаванні нової логіки. Для цього архітектура повинна бути розроблена таким чином, щоб легко інтегрувати нові функціональні елементи без порушення вже існуючих;
- тестованість: застосунок має бути побудований так, щоб його компоненти можна було легко тестувати, як за допомогою модульних тестів, так і інтеграційних. Це забезпечує виявлення та виправлення помилок ще на ранніх етапах розробки, що значно економить час і ресурси;
- гнучкість: архітектура повинна бути достатньо адаптивною, щоб дозволяти легко вносити зміни в бізнес-логіку або візуальну частину без значних перебудов проєкту. Гнучкість дозволяє швидко реагувати на зміну вимог користувачів або ринку.

Оптимізація застосунку є ще одним важливим аспектом, який безпосередньо впливає на досвід користувачів. Вона стосується як ефективного використання ресурсів, так і швидкодії інтерфейсу. Для цього важливо враховувати оптимальне завантаження даних, мінімізацію затримок, а також використання сучасних інструментів і технологій для покращення продуктивності. Оптимізований застосунок забезпечує не лише кращу взаємодію з користувачами, а й зменшує витрати на серверні ресурси.

2 ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Огляд основних джерел

Джерела були відібрані за наступними критеріями:

- авторитетність: перевага надавалась офіційним ресурсам або авторитетним вебсайтам, що мають тривалу історію, активно оновлюються та є широко відомими в середовищі розробників;
- актуальність: обрані ресурси, що відображають сучасний стан технологій (React, Next.js, оптимізаційних технік);
- об'єктивність: надано перевагу матеріалам, які подають інформацію без явної комерційної зацікавленості, мають технічний і неупереджений характер;
- достовірність: використано офіційну документацію, статті та блоги визнаних компаній, команд розробників або експертів у галузі, які відомі своєю репутацією та якістю подання матеріалу.

Обрані джерела можна згрупувати за певним набором тем. Першим є джерела, пов'язані з React, а саме офіційна документація, базові поняття, найкращі практики:

- React (software) – Wikipedia [1]: тут наведено базовий огляд популярної JavaScript-бібліотеки React для розробки інтерфейсів користувача. Ресурс містить історію створення, основні ідеї, особливості і ключові поняття;
- React.dev [5]: офіційна навчальна документація з React, що містить уроки, гайди, приклади та найкращі практики. Це оновлена версія офіційних матеріалів після випуску останніх версій React;
- React Lazy [6]: офіційна сторінка документації React з описом функції React.lazy для динамічного імпорту компонентів і оптимізації продуктивності шляхом розбиття коду (code splitting);
- Redux Documentation [7]: офіційна документація бібліотеки для орагнізації стейт менеджера Redux, яка тісно пов’язана з React. Тут містяться концепції, найкращі практики, приклади;

– TanStack Query [8]: офіційна документація з TanStack Query, бібліотеки для керування станом серверних даних у React. Ця бібліотека робить оптимізацію продуктивності під час роботи з асинхронними запитами.

Далі можна згрупувати джерела, які стосуються Next.js та серверним рендерингом, які включають офіційну документацію та блогові пости про особливості Next.js у поєднанні з React:

– Next.js and React [2] – Embrox Blog: пост, що описує особливості використання Next.js разом з React, переваги, приклади застосувань;
– Next.js – Офіційна документація [9]: офіційний сайт Next.js, фреймворку для React з підтримкою серверного рендерингу, статичних сторінок та оптимізації. Містить документацію та приклади використання.

Наступною темою є оптимізація продуктивності та метрики завантаження. Сюди входять ресурси, присвячені способам прискорення React-додатків, вимірюванню продуктивності та оптимізації UX:

– React Performance Optimization Techniques – MaybeWorks Blog [3]: пост, присвячений різним методам оптимізації React-додатків, таким як мемоізація, динамічне завантаження компонентів та використання ефективних патернів;
– основні метрики швидкості завантаження сайту – CityHost Blog [10]: Стаття про ключові веб-метрики продуктивності: TTFB, FCP, LCP та інші. Важлива для розуміння, як виміряти ефективність React/Next.js застосунків;
– web.dev [11]: офіційний ресурс від Google, присвячений оптимізації продуктивності веб-додатків, застосункам, Core Web Vitals та іншим сучасним практикам;
– Patterns.dev – Vanilla Virtual Lists [12]: опис підходів до віртуалізації списків, що є однією з ключових технік оптимізації продуктивності під час відображення великої кількості елементів у React. Даний матеріал подає патерни та техніки, які можуть бути застосовані і в React, і в інших фреймворках.

– Usability Models for Site Optimization – CEUR Workshop Proceedings [13]: стаття про використання моделей оцінки юзабіліті на етапі проєктування для покращення продуктивності вебсайтів. Розглядаються методи, які допомагають підвищити швидкість завантаження, якість UX та ефективність роботи сайту.

Окремо було розглянуто вже існуючі дослідження, присвячені оптимізації React-застосунків, зокрема публікації з тематичних форумів та технічних блогів, що описують загальні принципи підвищення продуктивності:

– DOU.ua Forum Topic – Оптимізація продуктивності React-застосунків [14]: у цій публікації розглядаються базові методи оптимізації (мемоізація, розбиття коду, віртуалізація), але відсутні реальні production-кейси, кількісні метрики до/після змін, глибинний аналіз SSR/SSG, а також не враховано сучасні підходи до керування станом, кешування даних і роботи з великими обсягами інформації в реальних умовах використання.
– Bezverhiy O. I., Kutsenko O. I. – Шляхи оптимізації кросплатформених додатків із використанням бібліотеки React та фреймворку React Native [15]: у цій публікації докладно описано оптимізацію на рівні компонентів (React.memo, React.lazy/Suspense), управління станом (Redux, MobX, Context), кешування та оптимізацію запитів API, розділення коду та використання Web Workers, а також поєднання синхронного та асинхронного рендерингу; проте відсутні реальні production-кейси, немає кількісних замірів Web Vitals (FCP, LCP, TTI) до й після оптимізації, не розглянуто SSR/SSG-стратегії, що обмежує практичну та наукову цінність роботи.

2.2 Аналіз літератури

Офіційна документація (React, Next.js, Redux, TanStack Query) встановлює стандарти, які допомагають створювати інтерфейси та керувати станом застосунків. Вона пропонує перевірені рішення та приклади, які можна легко

використовувати у своїх проєктах. Документація постійно оновлюється, додаючи нові найкращі практики та досвід спільноти.

Блоги та технічні статті додають до офіційної документації більше практичних прикладів із реальних проєктів. У них можна знайти ідеї для вирішення специфічних проблем, порівняння інструментів і технік, а також практичні поради. Такі джерела допомагають адаптувати теорію до конкретних практичних потреб. Ключовими висновками, що постали з аналізу, є:

- поділ додатків на компоненти робить код простішим у розробці та підтримці;
- використання бібліотек для управління станом робить застосунки більш стабільними та спрощує роботу з асинхронними запитами;
- поєднання клієнтського і серверного рендерингу прискорює завантаження сторінок і покращує SEO;
- оптимізація продуктивності та використання метрик підвищують зручність для користувачів, прискорюючи роботу додатків.

Методи та підходи, використані в оглянутих джерелах:

- офіційна документація: Пропонує покрокові інструкції, приклади коду та чіткі описи API. Вона допомагає уникати помилок і забезпечує стабільність роботи;
- блоги та статті: Зазвичай базуються на практичному досвіді. Автори аналізують ефективність інструментів і методів, показують їхні переваги та недоліки, використовуючи реальні кейси. Це дозволяє швидко знаходити рішення для поширених проблем;
- метрики продуктивності: Джерела, які фокусуються на метриках (наприклад, web.dev), використовують кількісні показники для оцінки ефективності застосунків. За допомогою цього, можна оцінити ефективність застосованих оптимізацій та виявити слабкі місця. Загалом ці підходи допомагають знаходити слабкі місця та перевіряти, чи працюють зміни на краще.

Попередні дослідження створили міцну базу для розробки веб-застосунків із використанням React та схожих технологій. Вони визначили основні принципи, запропонували інструменти для управління станом, серверного рендерингу та оптимізації продуктивності. Поєднання офіційної документації, блогів, статей і метрик дозволяє використовувати як перевірені стандарти, так і практичний досвід. Це допомагає ефективно розвивати галузь і покращувати якість продуктів.

2.3 Оцінка актуальності та новизни.

Інформація, представлена в офіційній документації, блогах та аналітичних статтях, залишається актуальною, адже ці джерела постійно оновлюються та реагують на зміни в технологіях. Вони враховують сучасні тенденції, впроваджують нові підходи та інструменти, а також адаптуються до останніх версій фреймворків і бібліотек. Така актуальність робить їх цінними для розробників, які прагнуть оперативно застосовувати інноваційні рішення у своїх проєктах.

Наукова новизна джерел полягає в пошуку та запровадженні кращих практик, оптимізації процесів та вдосконаленні користувацького досвіду. Вона проявляється у використанні прогресивних підходів до управління станом, прискорення завантаження сторінок, покращення взаємодії з користувачем та підвищення якості кінцевого продукту. Завдяки цьому джерела мають позитивний вплив на розвиток дослідження, адже вони вказують на напрямки подальших покращень, пропонують сучасні методи оцінки ефективності та сприяють формуванню більш надійних та швидкодіючих веб-застосунків.

2.4 Висновки з огляду

На основі проведеного огляду літератури можна зробити висновок, що сучасні джерела, які стосуються React, Next.js, Redux, TanStack Query та оптимізаційних методик, створили міцний фундамент для розвитку веб-застосунків. Офіційна документація забезпечує надійні стандарти та покрокові інструкції, пропонуючи фундаментальні принципи та стабільні підходи до

побудови інтерфейсів і керування станом. У той самий час блоги та статті з реальними випадками доповнюють ці стандарти, надаючи розробникам практичний досвід, допомагаючи їм швидко знаходити рішення для складних проблем, порівнювати інструменти та обирати найефективніші методи.

Разом з тим, при всій різноманітності джерел та їх постійному оновленні, можна помітити певні прогалини. Зокрема, не завжди існує достатньо детальних, покрокових прикладів застосування нових інструментів у специфічних або нетипових випадках. Інколи бракує порівняльних оглядів складних підходів у реальних масштабованих проєктах, які б дали змогу краще зрозуміти їхню довгострокову ефективність та зручність. Крім того, сфера технологій постійно змінюється, тож нові інструменти, методи чи фреймворки потребують дослідження і інтеграції з існуючими підходами.

Такі прогалини дають змогу говорити о потребі в подальших дослідженнях. Потрібно продовжувати аналізувати та порівнювати різні інструменти, методи та стратегії оптимізації. Дослідження повинні охоплювати не лише стандартні сценарії, а й складні, унікальні або експериментальні випадки. Це допоможе краще зрозуміти, наскільки ефективні різні рішення, подолати обмеження поточних підходів, знайти нові способи використання сучасних технологій і покращити якість продуктів. Таким чином, підсумкові висновки вказують на вагомий внесок існуючої літератури та її цінність для актуальних реалій, а також підтверджують необхідність і перспективність подальших досліджень.

3 ПОСТАНОВКА ЗАДАЧІ

Задача полягає у систематичному дослідженні, порівнянні та застосуванні ефективних методів оптимізації й архітектурних рішень для застосунків на основі React з метою підвищення їхньої продуктивності та поліпшення користувацького досвіду. На практичному рівні передбачається реалізація прототипу застосунку, побудованого на базі React з використанням фреймворку Next.js, а також створення його альтернативної версії без оптимізаційних покращень. Подальше порівняння отриманих результатів дозволить визначити ефективність застосованих підходів.

Щоб забезпечити повноцінне дослідження, необхідно чітко визначити послідовність дій та інструменти, що будуть задіяні. Вихідним пунктом стане розробка майбутнього застосунку, який не міститиме цілеспрямованих оптимізацій. Застосунок має демонструвати обробку даних (наприклад, відображення списків, фільтрація або пошук), роботу з зовнішніми API (для імітації реальних умов), а також роботу з графічними ресурсами (зображеннями). Цей початковий варіант буде слугувати контрольним прикладом для подальшого впровадження й оцінювання оптимізаційних рішень.

Після створення базової версії застосунку буде впроваджено низку методів оптимізації. Перш за все, оптимізацію рендерингу компонентів: використання мемоізаційних технік, що допоможе уникнути зайвих перерендерів при зміні стану або пропсів, тобто переданих компоненту даних, не пов’язаних із цим конкретним компонентом. Далі планується застосувати віртуалізацію списків за допомогою відповідних бібліотек для ефективної роботи з великими обсягами даних у вигляді таблиць чи галерей аби не рендерити всі елементи одночасно, а лише ті, що перебувають у видимій області вікна. Наступним кроком буде впровадження lazy loading, тобто завантаження лише потрібних частин коду під час їх використання, для асинхронного завантаження компонентів, модулів та окремих ресурсів, що зменшить початковий обсяг коду, необхідний для відображення сторінки, і знизить час її стартового рендерингу.

Ще одним аспектом є оптимізація використання мережевих ресурсів. Тут передбачається застосування стратегії зменшення кількості запитів до серверу

шляхом попереднього кешування результатів запитів, використання відповідних бібліотек, таких як TanStack Query, для локального кешування та повторного використання вже завантажених даних. Важливо перевірити ефективність цих підходів на реальних або тестових даних, імітуючи умови складних застосунків, які часто роблять багатократні звернення до сторонніх служб чи внутрішніх API.

Також увагу буде приділено архітектурним рішенням. Реалізація компонентно-орієнтованої архітектури з розподілом на дрібні функціональні компоненти повинна забезпечити легку підтримку та масштабування коду. Такий підхід дозволить спростити розширення функціональності та підвищити продуктивність застосунку загалом. Окрім того, планується проаналізувати використання статичної генерації сторінок чи гібридних рішень.

Для створення застосунків буде використаний фреймворк Next.js, який базується на React і значно полегшує впровадження таких підходів, як SSR (Server-side Rendering), SSG (Static Site Generation) та CSR (Client-side Rendering). Next.js також містить низку внутрішніх оптимізаційних механізмів, таких як автоматична підтримка code splitting, можливості пререндерингу та інтелектуального завантаження ресурсів. Таким чином, використання даного фреймворку дасть можливість в повній мірі проаналізувати всі можливі методи оптимізації та отримати більш чітко визначену загальну архітектуру застосунку завдяки вже існуючому набору правил.

Для оцінювання результатів буде використано низку метрик, зокрема First Contentful Paint (FCP), Largest Contentful Paint (LCP), загальну продуктивність (Performance), а також інші показники з Lighthouse[4]. Це дозволить порівняти початкову версію застосунку з оптимізованою, оцінити вплив конкретних змін на кінцеві показники продуктивності та визначити ефективність кожного застосованого методу окремо. Очікується, що після впровадження оптимізацій користувачі відчуватимуть більш швидку реакцію інтерфейсу, зниження часу завантаження й покращене загальне враження від роботи із застосунком.

Для вимірювання продуктивності планується використовувати Lighthouse, оскільки цей інструмент є надійним та ефективним програмним забезпеченням.

Крім того, потрібна буде певна кількість тестових даних та ресурсів для імітації реальної роботи застосунку. Йдеться про підбір або генерацію великих списків елементів, зображень різного формату, а також доступ до тестових або реальних API для завантаження даних у динамічному режимі.

Обмеження, які можуть виникнути у процесі проведення дослідження, пов’язані з обсягом реалізованого функціоналу: не всі оптимізаційні підходи будуть однаково показовими в межах обраного прототипу. Наприклад, якщо застосунок не використовує великі списки даних, то віртуалізація списків не дасть вираженого результату. Подібно до цього, якщо використання зображень буде обмеженим, то оптимізація форматів та їх стиснення не продемонструють значного впливу. Тому під час розробки майбутнього застосунку слід продумати випадки, які дадуть змогу оцінити ці аспекти.

До необхідних ресурсів також можна віднести документацію та довідкові матеріали: офіційну документацію React [5], Next.js [9], Redux [7] або інших менеджерів стану, матеріали з оптимізації від Google [11], а також статті та блоги, які описують найкращі практики та нові тенденції у сфері оптимізації React-застосунків. Знання про налаштування білду, застосування TypeScript для кращої підтримуваності коду, а також про використання інструментів розробника (Chrome DevTools, Lighthouse) необхідні для ефективного аналізу виникаючих проблем і пошуку найбільш оптимальних рішень.

Враховуючи ці фактори, у ході проєкту буде використано підхід, коли спочатку формується базова версія застосунку зі стандартною архітектурою та без оптимізацій. Потім послідовно будуть додані методи оптимізації: мемоізація, Tanstack Query, lazy loading, віртуалізація списків, оптимізація ресурсів та інтеграція SSR, SSG чи гібридних підходів від Next.js. На кожному етапі проводитиметься вимірювання продуктивності та аналіз показників. Така ітеративність дозволить визначити конкретний вплив кожного методу на кінцевий результат і аргументувати вибір оптимізаційних стратегій.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

4.1 Огляд архітектури (архітектурних рішень)

Першим ключовим аспектом є вибір архітектури програмного забезпечення, яке буде створено на базі React. Він сам по собі не нав’язує жорстких архітектурних рамок, що дає велику гнучкість, але створює необхідність свідомого вибору підходу. Зазвичай рекомендується застосовувати компонентно-орієнтовану архітектуру з односпрямованим потоком даних. Це означає, що на верхньому рівні ми маємо кореневий компонент, який отримує дані ззовні (від сервера чи зі сховища стану), і передає їх “вниз” крізь дерево компонентів. Така модель спрощує розуміння потоку даних та пошук джерела потенційних проблем.

Вибір архітектури залежить від розміру та складності проєкту, а також від його вимог. Неправильна структура може призвести до зайвих витрат ресурсів або до суттєвих обмежень у розширенні функціональності. У випадку невеликих проєктів, створення занадто продуманої та масштабної архітектури часто є надмірним. Для таких проєктів велика кількість директорій, складна модульна організація або надлишкове використання патернів може не лише ускладнити розробку, але й потребуватиме більше часу на реалізацію. З іншого боку, у великих проєктах, які мають багато фіч, модулів і складну бізнес-логіку, використання простої архітектури може стати серйозною проблемою. Якщо структуру не продумати заздалегідь, розробники зіткнуться із хаотичним кодом, що важко підтримувати, тестувати та масштабувати.

Запропонована структура проєкту (див. рис. 4.1) є універсальною і підходить як для невеликих, так і для досить великих проєктів. Її основна перевага полягає у гнучкості та модульності. Для невеликих проєктів можна реалізувати тільки базові частини структури. Для великих проєктів структура легко розширюється. Вона забезпечує чітку модульну структуру, яка дозволяє ефективно організувати код, логіку, стилі та інші ресурси додатку. Ключовими принципами, що лежать в основі цієї структури, є компонентно-орієнтований підхід, розділення відповідальностей та повторне використання коду. Застосунок має форму "дерева" директорій, де кожна гілка відповідає за свою функціональну частину.

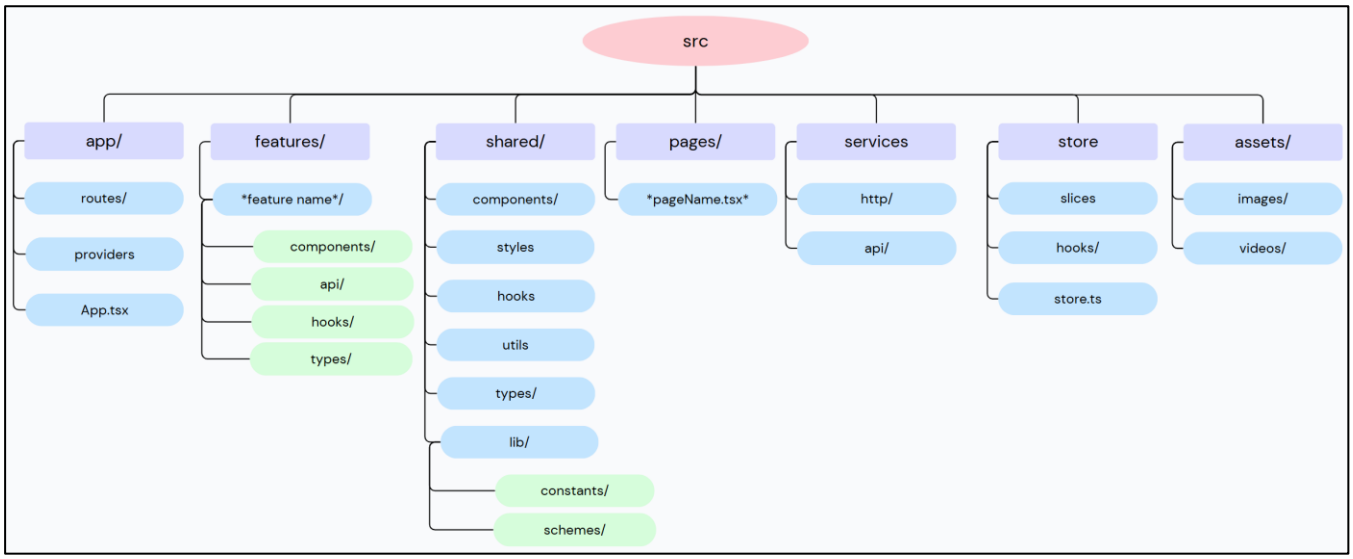


Рисунок 4.1 – Приклад архітектури застосунку(рисунок виконаний самостійно)

Почнемо із зовнішнього рівня структури: корнева директорія src/ містить увесь вихідний код додатку. Завдяки цьому, під час білду чи тестування ми чітко розуміємо, де знаходиться бізнес-логіка, компоненти та решта сутностей. Структура охоплює кілька ключових директорій: app/, features/, shared/, pages/, services/, store/, assets/. Кожна з них призначена для певної категорії завдань.

Папка app/ – це стартова точка додатку, яка містить головний компонент App.tsx та супутні конфігурації на рівні додатку. Тут розташовується логіка ініціалізації, наприклад підключення глобальних провайдерів, таких як сховище Redux, провайдери локалізації, а також провайдери маршрутизації (React Router). Структура app/ може виглядати так:

– app/routes/ – містить файли, пов’язані з маршрутизацією. Наприклад, це можуть бути конфігураційні файли з масивом маршрутів, які описують, які компоненти відповідають за певні URL-шляхи. Можна створити файл, наприклад, routesConfig.ts, де визначаються усі маршрути додатку в одному місці. Це дасть змогу легко керувати навігацією, централізовано додавати або змінювати маршрути без розпорошення логіки по всьому проєкту;

– app/providers/ – тут розташовуються провайдери різного типу. Наприклад, ReduxProvider.tsx для ініціалізації Redux Store, I18nProvider.tsx

для налаштування мультимовності, ThemeProvider.tsx для темізації додатку. Ідея полягає в тому, щоб усі провайдери були зібрані в одному місці та загортали App.tsx. Таким чином, в App.tsx ми просто імпортуємо усі ці провайдери та обгортаємо застосунок, отримуючи єдиний "композиційний шар", відповідальний за глобальну конфігурацію;

– App.tsx – основний компонент, що слугує коренем додатку. Він підключає маршрутизатор, провайдери стану та інші глобальні контексти. Це точка, з якої починається рендеринг усього застосунку. Залежно від обраного архітектурного підходу, App.tsx може містити механізми SSR/SSG інтеграції з Next.js, або ж просто бути точкою входу для CSR (Client-Side Rendering).

Перейдемо до features/. Ця директорія служить для розміщення логіки, що стосується певних функціональних можливостей застосунку. Кожна фіча (модуль) матиме власну директорію, наприклад, auth/ для автентифікації. Ідея полягає у відокремленні логіки за доменними зонами (тобто за конкретними частинами застосунку, які відповідають певній бізнес-логіці чи задачам). Тут зосереджені компоненти, які безпосередньо реалізують функціонал фічі, її API-запити, хуки, які розв’язують специфічні завдання, та типи даних (інтерфейси, enum, типи, пов’язані саме з цією фічею). Таким чином, коли розробник відкриває features/auth/, він бачить повністю зосереджений тут функціональний блок: UI-компоненти, хуки для перевірки стану користувача, а також типи користувача у types.ts.

Наступна важлива директорія – shared/. Вона містить загальні компоненти, стилі та утиліти, які можуть бути використані в різних місцях додатку. Наприклад, у shared/components/ui розмістяться базові елементи інтерфейсу – Button.tsx, Modal.tsx, Spinner.tsx. Ці компоненти незалежні від конкретних фіч і можуть бути використані в будь-якому місці. У shared/styles/ знаходяться глобальні стилі, змінні теми, reset-стилі чи глобальні стилі шрифтів. shared/hooks/ містить загальні хуки, які можуть застосовуватися в різних контекстах. shared/utils/ – місце для загальних утиліт, наприклад, форматування дат, перетворення рядків, хелпери для виклику API. shared/constants/ – для статичних констант, таких як URL сторонніх сервісів,

рядки локалізації за замовчуванням або конфігураційні змінні. shared/types/ – для глобальних типів, які можуть бути важливими в усьому проєкті (наприклад, загальні інтерфейси відповіді від сервера).

pages/ – це директорія, де зберігаються файли сторінок. Кожна сторінка організована за маршрутом. Наприклад, pages/login/ містить LoginPage.tsx, який відповідає за рендеринг сторінки входу. Якщо сторінка складніша й має власні специфічні компоненти, їх можна помістити в pages/login/components/. Розподіл на сторінки дозволяє легко орієнтуватися в маршрутах і зрозуміти, які компоненти відповідають за той чи інший URL. На відміну від features/, де ми фокусуємося на функціоналі, у pages/ ми фокусуємося відображенні вже готових компонентів у конкретні користувацькі сценарії.

Наступний розділ – services/. Це місце, де розташовуються інтеграційні шари додатку. До них належить налаштування HTTP-клієнта (наприклад, services/http/ з файлом axiosInstance.ts, який створює і налаштовує інстанс axios), а також services/api/, де розміщені файли з описом API-запитів (наприклад, userApi.ts для отримання або оновлення інформації про користувача). Якщо проєкт використовує сторонні сервіси, такі як Firebase, то в services/firebase/ будуть зберігатися конфігурації, ініціалізація та хелпери для роботи з Firebase. Ця архітектура дозволяє чітко відділити логіку взаємодії з зовнішніми системами та API від основного коду компонентів.

Директорія store/ – це центр управління глобальним станом додатку. Тут ми можемо створити store.ts, де буде ініціалізовано Redux Store або Zustand Store, а також директорію slices/ (для Redux) чи іншу структуру для стану, в якій ми створюємо "зрізи" (slices) – окремі частини глобального стану, наприклад userSlice.ts. store/hooks/ містить спеціальні хуки для взаємодії зі станом, такі як useAppSelector(), useAppDispatch(), які абстрагують стандартну логіку Redux (або інструмента менеджменту стану) і роблять її більш зрозумілою при використанні у компонентах. Такий підхід дозволяє централізувати логіку зміни стану і легко розширювати його за необхідності.

Директорія assets/ містить статичні файли: зображення, іконки, шрифти, відео. Тут ми можемо організувати їх за піддиректоріями (images/, fonts/, videos/) для зручного використання у компонентах та стилях. Цей підхід впорядковує всі мультимедійні ресурси в одному місці, що полегшує їх оновлення і пошук.

Наостанок у корені src/ знаходиться index.tsx – точка входу додатку, з якої веб-пакет або інший збирач починає процес побудови. Тут ми підключаємо кореневий App.tsx, відомий зі структури app/, застосовуємо провайдери та рендеримо застосунок у кореневий елемент HTML (root), який визначений у public/index.html.

Описана архітектура є гнучкою та масштабованою. Вона знижує зв'язаність коду, оскільки кожна частина (фіча, сторінка, загальні компоненти) мають окреме призначення, а глобальні аспекти (стан, сервіси, ресурси) розташовані у відповідних директоріях, що полегшує розвиток та підтримку проєкту. У перспективі, якщо проєкт зростатиме, можна легко додавати нові фічі, оскільки все є прозорим та зрозумілим. Це полегшує орієнтацію нових розробників у проєкті, спрощує навігацію в коді та підвищує якість розробки загалом.

Важливо зазначити, що ця архітектура природньо поєднується з підходами до оптимізації: при чіткому поділі на фічі та загальні компоненти, простіше впроваджувати lazy loading (наприклад, динамічно підвантажувати сторінки чи фічі), застосовувати мемоізацію для окремих компонентів, впроваджувати віртуалізацію списків у конкретних фіча-директоріях, а також ефективно керувати завантаженням статичних ресурсів, оскільки зображення або інші великі файли сконцентровані в assets/.

4.2 Огляд методів оптимізації

4.2.1 Техніки мемоізації у React

Мемоізація – це ключова техніка оптимізації, яка використовується для зменшення кількості повторних обчислень або перерендерів у React-застосунках. У контексті React мемоізація дозволяє зберігати результат функції або компонента і повторно використовувати його, якщо вхідні дані не змінилися. Це особливо

важливо у випадках, коли обчислення займають багато ресурсів або рендеринг компонентів викликається занадто часто.

React пропонує кілька вбудованих інструментів для мемоізації. Наприклад, React.memo дозволяє обгортати функціональні компоненти, щоб вони не перерендерювалися, якщо пропси залишаються незмінними. Це простий у використанні інструмент, який підходить для компонентів, що отримують багато даних і не залежать від глобального стану. У випадках, коли потрібно мемоізувати складні обчислення всередині компонентів, використовується хук useMemo. Він обчислює значення лише тоді, коли змінюються залежності, і зберігає це значення між рендерами. Наприклад, якщо у компоненті потрібно відсортувати великий масив даних, useMemo гарантує, що сортування відбуватиметься лише тоді, коли змінюється сам масив, а не при кожному рендері.

Ще одним важливим інструментом є хук useCallback, який працює аналогічно useMemo, але використовується для мемоізації функцій. Це корисно у випадках, коли функції передаються як пропси в дочірні компоненти, і їхній незмінний стан важливий для уникнення зайвих перерендерів. Наприклад, у компонентах з великою кількістю подій useCallback запобігає створенню нових екземплярів функцій при кожному рендері.

У зв’язці з useCallback треба використовувати React.memo, який запобігає повторному рендерингу функціонального компонента, якщо його пропси залишаються незмінними. Якщо вони не змінювалися, компонент не буде перерендерено, навіть якщо батьківський компонент оновився. Така комбінація забезпечує мінімальне використання ресурсів і покращує продуктивність, особливо в компонентах, які знаходяться високо в ієрархії або повторюються у великій кількості.

Мемоізація є ефективною тільки тоді, коли застосовується у потрібних місцях. Її надмірне використання може призвести до зворотного ефекту, оскільки перевірка залежностей і збереження мемоізованих значень теж споживають ресурси.

4.2.2 Віртуалізація списків у веб-застосунках

Віртуалізація списків – це техніка, яка використовується для оптимізації рендерингу великих списків або таблиць, зменшуючи навантаження на браузер. У традиційному підході React завантажує усі елементи списку одночасно, незалежно від того, чи знаходяться вони у видимій області екрану. Це може призвести до значного споживання пам’яті та зниження продуктивності у випадках, коли список містить тисячі елементів. Завдяки віртуалізації рендеряться лише ті елементи, які розташовані поруч із видимою зоною. Це досягається за допомогою обчислення позицій елементів у DOM і заміни "невидимих" елементів порожніми контейнерами. У React реалізація віртуалізації найчастіше здійснюється за допомогою бібліотек, таких як react-window або react-virtualized [12].

Наприклад, у випадку використання списку з 10 000 елементів, віртуалізація дозволяє рендерити лише 20-30 елементів одночасно, залежно від розміру вікна перегляду. Бібліотека react-window надає готові компоненти, такі як FixedSizeList, які дозволяють ефективно рендерити лише видимі елементи списку та значно спрощують впровадження цієї техніки. Все, що потрібно зробити – це вказати висоту кожного елемента та загальний розмір списку.

4.2.3 Лініве завантаження (Lazy Loading) в React

Lazy loading (лініве завантаження) дозволяє відкладати завантаження частин застосунку або ресурсів до того моменту, коли вони дійсно потрібні. У React це досягається за допомогою динамічного імпорту компонентів або модулів. Наприклад, замість того, щоб включати всю сторінку одразу в бандл, тобто зібраний файл з кодом і ресурсами, її компоненти завантажуються тільки тоді, коли користувач переходить за відповідним маршрутом.

React має вбудовану підтримку lazy loading через функцію React.lazy() у поєднанні з компонентом Suspense [6]. Це дозволяє розбити застосунок на частини, які можуть завантажуватися асинхронно. Наприклад, у pages/ можна реалізувати лінійне завантаження сторінок. Якщо користувач заходить на сторінку логіну, інші сторінки (dashboard/) не завантажуються доти, доки вони не потрібні. Це значно

зменшує розмір початкового бандла, прискорює завантаження сторінки та покращує Time to Interactive (TTI).

Lazy loading застосовується не лише до компонентів, але й до зображень, відео та інших ресурсів. Це реалізовується за допомогою додавання атрибуту «loading» зі значенням «lazy». Однак, деякі браузери, наприклад Safari або Firefox, не підтримують даний атрибут. Тому існують спеціальні бібліотеки. Одна з таких це react-lazy-load-image-component. Вона дозволяє завантажувати зображення лише тоді, коли вони з'являються у видимій області екрана, зменшуючи кількість одночасно завантажених ресурсів і покращуючи швидкість завантаження сторінки. Замість використання стандартного HTML-елемента <img>, ця бібліотека надає компонент LazyLoadImage, який автоматично підключає механізм лінійного завантаження. Компонент працює із використанням Intersection Observer API , тобто вбудованого інструменту для відстеження елементів у полі зору. Він також підтримує такі корисні функції, як попереднє завантаження (placeholder) або ефекти розмиття, що дозволяють створити більш плавний і привабливий візуальний досвід для користувачів.

У фреймворку Next.js альтернативою для lazy loading компонентів є функція dynamic, яка забезпечує асинхронне завантаження компонентів без додаткових налаштувань. Для роботи з зображеннями Next.js має інтегрований компонент Image, який за замовчуванням підтримує lazy loading і додаткові оптимізації зображень.

4.2.4 Оптимізація мережевих запитів за допомогою TanStack Query

Для оптимізації мережевих запитів використовується бібліотека TanStack Query [8]. Вона створена для управління серверними запитами та станом даних у React-застосунках. Її основна мета – спростити процес отримання, кешування, оновлення та синхронізації даних з сервером, забезпечуючи високу продуктивність і оптимальний користувацький досвід. Бібліотека містить цілий ряд переваг:

– кешування даних: TanStack Query автоматично кешує відповіді від серверу, що дозволяє повторно використовувати вже отримані дані без

необхідності здійснювати нові запити. Це значно зменшує кількість мережевих запитів, підвищуючи продуктивність і знижуючи затримки. Наприклад, коли користувач повертається на сторінку, яка вже була завантажена, дані можуть миттєво відображатися з кешу;

- синхронізація даних у реальному часі: бібліотека пропонує автоматичне оновлення кешу у фоновому режимі через механізм рефетчингу (refetching). Це означає, що дані залишаються актуальними, навіть якщо вони змінюються на сервері. Завдяки цьому користувачі завжди бачать свіжу інформацію без потреби ручного оновлення;
- контроль станів запитів: TanStack Query автоматично управляє станами запитів, такими як loading, success, error, що спрощує побудову UI і дозволяє розробникам зосередитися на логіці, а не на створенні вручну механізмів обробки запитів;
- механізми повторних спроб (Retries): якщо запит до серверу зазнав невдачі, бібліотека автоматично виконає повторні спроби, використовуючи експоненційний інтервал. Це покращує стійкість застосунку, особливо у випадках тимчасових збоїв мережі;
- оптимістичне оновлення (Optimistic Updates): бібліотека дозволяє передбачати результат операцій (наприклад, додавання або редагування даних) і тимчасово оновлювати UI ще до отримання підтвердження від серверу. Це покращує користувацький досвід, створюючи відчуття миттєвої реакції додатку.

4.2.5 Техніки оптимізації SSR, SSG та CSR у Next.js

У Next.js техніки SSR (Server-Side Rendering), SSG (Static Site Generation) та CSR (Client-Side Rendering) є основними підходами до рендерингу сторінок, які надають гнучкі можливості для оптимізації продуктивності, SEO та користувацького досвіду. Ці механізми дозволяють ефективно поєднувати як серверне, так і клієнтське рендеринг для різних частин застосунку залежно від вимог. Завдяки можливості комбінування цих підходів у межах одного проєкту,

Next.js дозволяє досягти оптимального балансу між швидкодією, актуальністю контенту та гнучкістю архітектури застосунку.

Server-Side Rendering (SSR) – це техніка рендерингу сторінок на стороні сервера перед тим, як вони будуть доставлені клієнту. У Next.js SSR реалізується через функцію getServerSideProps, яка викликається на сервері щоразу, коли запитуємо сторінку. Ця функція збирає необхідні дані, генерує HTML і передає його на клієнт разом із початковим станом. Можна виділити такі переваги даного методу:

- SEO-оптимізація: Оскільки сторінки рендеряться на сервері, пошукові системи отримують готовий HTML, що підвищує видимість у пошуковій видачі;
- актуальність даних: Кожен запит до сторінки генерує новий HTML із найсвіжішими даними, що забезпечує користувачів актуальним контентом;
- покращення часу до першого відображення (FCP): HTML завантажується одразу без необхідності виконання JavaScript.

Але існують і мінуси у використанні SSR, основним з яких є високе навантаження на сервер і залежність від його продуктивності. Оскільки сторінка генерується на сервері для кожного запиту, це значно підвищує навантаження на сервер, особливо при великому трафіку. У випадку масштабних проєктів із великою кількістю відвідувачів це може вимагати значних ресурсів для підтримки стабільної роботи серверної інфраструктури.

SSR доцільно використовувати в ситуаціях, де важлива актуальність даних у реальному часі, наприклад, у панелях адміністратора, інформаційних табло чи платформах електронної комерції, де товарні запаси змінюються швидко.

Static Site Generation (SSG) – це техніка, при якій сторінки рендеряться під час білду додатку. У Next.js SSG реалізується через функцію getStaticProps, яка викликається під час генерації сторінки. Сторінки генеруються як статичні HTML-файли і доставляються клієнту без додаткових запитів до сервера. Використання SSG має наступні переваги:

- висока швидкість завантаження: Статичні сторінки розміщуються на CDN, тобто в мережі взаємопов'язаних серверів для прискорення завантаження сторінок, і доставляються миттєво, що забезпечує надзвичайно швидкий відгук;
- низьке навантаження на сервер: Після білду сервер не бере участі в генерації HTML, що знижує витрати на хостинг;
- покращення SEO: Готовий HTML добре індексується пошуковими системами.

SSG найкраще підходить для контенту, який рідко змінюється, наприклад, для блогів, документації, сторінок портфоліо чи лендінгів. Для оновлення даних використовується механізм регенерації (ISR – Incremental Static Regeneration), який дозволяє оновлювати окремі сторінки без повного ребілду сайту.

Client-Side Rendering (CSR) – це традиційний підхід у React-застосунках, коли HTML генерується на клієнтській стороні. У Next.js CSR використовується за замовчуванням у компонентах, які не мають специфічних функцій для SSR чи SSG. Дані отримуються через JavaScript після завантаження сторінки, а початковий контент може бути порожнім або містити базовий стан. Використання даного підходу має наступні переваги:

- інтерактивність: весь рендеринг і оновлення відбуваються на клієнтській стороні в браузері, що дозволяє створювати багатофункціональні веб-застосунки;
- гнучкість: CSR не залежить від сервера для рендерингу, що робить його універсальним для динамічних застосунків.

Однак, існують і певні проблеми при використанні CSR. Основні з них включають:

- SEO-проблеми: пошукові системи можуть погано індексувати динамічні сторінки, тобто не зберігати або не враховувати вміст таких сторінок у своїй базі даних, оскільки початковий HTML не містить контенту;
- погіршення UX при повільному завантаженні: користувачі можуть бачити порожній екран, доки JavaScript не завантажиться.

CSR підходить для ситуацій, коли потрібно працювати з даними що постійно оновлюються, коли є інтерактивність.

Усі три підходи є унікальними та створені для конкретних цілей (див. табл. 4.1).

Таблиця 4.1 – Порівняльний аналіз SSR, SSG, CSR (таблиця виконана самостійно)

Параметр	SSR	SSG	CSR
Рендеринг	Рендеринг на сервері перед доставкою сторінки клієнту	Рендеринг відбувається під час білду	Рендеринг відбувається на клієнтській стороні після завантаження сторінки
Швидкість завантаження	Середня (залежить від часу обробки на сервері)	Дуже висока (статичні файли доставляються миттєво)	Низька (початкове завантаження довше через JavaScript)
SEO-оптимізація	Висока, готовий HTML одразу доступний	Висока, готовий HTML одразу доступний	Низька, HTML генерується на клієнті
Оновлення даних	Дані оновлюються при кожному запиті	Дані оновлюються лише при новому білді або через ISR	Дані оновлюються динамічно на клієнті
Навантаження на сервер	Високе, оскільки сервер обробляє кожен запит	Мінімальне, сторінки вже згенеровані	Низьке, сервер лише доставляє JavaScript
Застосування	Підходить для динамічних сторінок, які потребують актуальних даних	Підходить для статичних сторінок і контенту, що рідко змінюється (блоги, документація)	Підходить для інтерфейсів, які потребують високої інтерактивності

Next.js дозволяє комбінувати SSR, SSG і CSR в одному проєкті, що робить його універсальним для різних вимог. Наприклад, статичні сторінки можна генерувати з SSG, актуальні дані завантажувати через SSR, а інтерфейси, які потребують складної динаміки, реалізувати через CSR. Це дозволяє створювати оптимальні рішення залежно від потреб проєкту.

Починаючи з версії Next.js 13+, концепції SSR (Server-Side Rendering) та SSG (Static Site Generation) були значно спрощені завдяки впровадженню серверних та клієнтських компонентів. У новій архітектурі App Router Next.js автоматично визначає, який метод рендерингу застосовувати залежно від контексту компонента. Серверні компоненти використовуються для виконання важких операцій, таких як отримання даних або обробка логіки, оскільки вони рендеряться на сервері. Натомість клієнтські компоненти відповідають за інтерактивність на стороні клієнта і рендеряться вже в браузері. Сам Next.js інтелектуально комбінує SSR та SSG для оптимального результату: статичні сторінки генеруються під час білду, а динамічний контент обробляється на сервері під час запитів. Це дозволяє зосередитися на логіці застосунку, оскільки Next.js автоматично обирає найефективніший метод рендерингу для кожного випадку. Однак слід розуміти, що хоча ці концепції і були спрощені, але існує велика кількість проєктів яка користається старою концепцією.

4.2.6 Оцінка продуктивності веб-застосунків

Оцінка продуктивності веб-застосунків є ключовим етапом для забезпечення швидкого завантаження, зручного користувацького досвіду та відповідності сучасним стандартам веб-розробки. Web Vitals – це певний набір стандартизованих метрик від Google, який використовується для оцінки якості користувацького досвіду під час роботи з веб-сторінками. First Contentful Paint (FCP), Time to Interactive (TTI), Largest Contentful Paint (LCP) та Cumulative Layout Shift (CLS) – це основні метрики, які використовуються для вимірювання швидкодії та ефективності веб-додатків (див. рис. 4.2). Ці показники разом з інструментом

Lighthouse надають глибоке розуміння роботи сторінок і допомагають виявити вузькі місця у продуктивності.

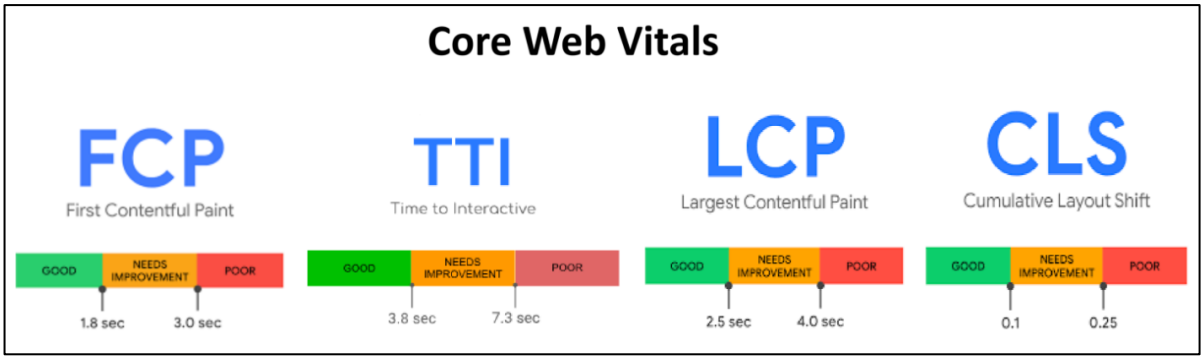


Рисунок 4.2 – Метрики для вимірювання продуктивності (рисунок виконаний самостійно)

First Contentful Paint (FCP) вимірює час, який потрібен браузеру для відображення першого елементу контенту на екрані після завантаження сторінки. Це може бути текст, зображення, SVG або інші елементи DOM. Значення FCP менше 1.8 секунд вважається хорошим, а більше 3 секунд – потребує оптимізації. Для покращення FCP зазвичай роблять наступні дії:

– мінімізують розмір CSS і JavaScript файлів;
– використовують critical CSS (вбудовують найважливіші стилі в HTML);
– пріоритезують завантаження ресурсів, що з’являються у видимій зоні;
– використовують сучасні формати зображень, як-от WebP.

Time to Interactive (TTI) – це час, за який сторінка стає повністю інтерактивною, тобто коли корисні ресурси завантажені, основний потік не зайнятий тривалими задачами і сторінка реагує на користувацькі дії, такі як кліки чи прокручування. Високе значення TTI створює враження, що сторінка зависла і недоступна. Значення TTI менше 3.8 секунд є хорошим, а більше 5 секунд потребує оптимізації. Для зниження TTI використовуються такі методи:

– відкладання виконання неважливих скриптів за допомогою lazy loading;
– оптимізація JavaScript (зменшення розміру бандла, code splitting);
– використання Web Workers для обробки важких завдань у фоновому режимі;

– зниження часу блокування основного потоку (Main Thread).

Largest Contentful Paint (LCP) вимірює час від початку завантаження сторінки до моменту, коли найбільший елемент контенту стає видимим на екрані. LCP є критичним для сприйняття швидкодії сторінки користувачем. Чим швидше відображається головний контент, тим кращий досвід отримує користувач. Значення LCP менше 2.5 секунд вважається хорошим, а більше 4 секунд – потребує оптимізації. Для покращення LCP:

– використовуються CDN для швидкої доставки ресурсів;
– пріоритезуються великі елементи (зображення, відео) для завантаження;
– оптимізуються зображення через сучасні формати (WebP, AVIF);
– використовується lazy loading для вторинних зображень;
– застосовується Server-Side Rendering (SSR) або Static Site Generation (SSG) для швидкого рендерингу сторінки.

Cumulative Layout Shift (CLS) вимірює кількість несподіваних зсувів елементів сторінки під час її завантаження. Це стосується ситуацій, коли контент рухається на екрані через завантаження зображень, шрифтів або рекламних блоків. Несподівані зміщення контенту можуть дратувати користувачів та призвести до помилкових кліків. Значення CLS менше 0.1 є хорошим, а більше 0.25 потребує оптимізації. Для зменшення CLS:

– визначаються фіксовані розміри для зображень та відео;
– резервуються місця для рекламних блоків;
– використовуються шрифти з font-display: swap, щоб уникнути зсувів при завантаженні шрифтів;
– застосовуються CSS-анімації з мінімальним впливом на layout.

Lighthouse та Web Vitals використовуються для оцінки та покращення продуктивності, доступності та якості веб-додатків. Lighthouse є автоматизованим інструментом від Google, який аналізує веб-сторінки за кількома ключовими критеріями: продуктивність (Performance), доступність (Accessibility), найкращі практики (Best Practices), SEO-оптимізація та відповідність стандартам прогресивних веб-додатків (PWA). Завдяки Lighthouse і Web Vitals можна

об'єктивно оцінювати користувацький досвід, усувати вузькі місця продуктивності та забезпечувати відповідність сторінок сучасним стандартам швидкодії та якості.

5 ОПИС ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

5.1 Розробка програмного забезпечення

У ході дослідження було створено два варіанти веб-застосунку: неоптимізований, який не містить жодних покращень продуктивності, та оптимізований, де було застосовано різні методи оптимізації. Для обох версій було проведено вимірювання продуктивності за допомогою Lighthouse, що дозволило оцінити їхню швидкодію, використання ресурсів та ефективність рендерингу.

Для реалізації програмного застосунку було створено серверну та клієнтську частини. Обидві частини побудовані у вигляді моноліту, що спрощує початковий етап розробки, забезпечує централізоване управління кодом і дозволяє легше контролювати всі процеси. Застосунок одразу розроблявся на Next.js версії 13+, оскільки цей фреймворк є сучасним стандартом для створення високопродуктивних React-застосунків з підтримкою гібридного рендерингу. Також, було створено простий серверний застосунок, задача якого віддавати дані клієнтській частині.

5.1.1 Використання CSR та SSR

На першому етапі була створена неоптимізована версія застосунку. Для отримання реалістичних та наближених до реальних умов тестування показників продуктивності був розроблений дизайн без зайвих анімацій чи складних UI-ефектів. Ситуація, коли треба робити багато запитів на сервер, отримувати велику кількість даних і динамічно їх відмальовувати є дуже розповсюдженою. Сам запит робиться всередині useEffect, а потім отримані результати відображаються у вигляді карток. При такому підході, якщо кількість користувачів є невеликою, то застосунок працює стабільно, і помилок та проблем з продуктивністю, як правило, не виникає. Однак, при зростанні обсягу даних, наприклад, коли користувачів більше двохсот, може виникати проблеми з продуктивністю.

Крім базового відображення списку в застосунку також була реалізована можливість сортування користувачів за різними критеріями та функція пошуку, що додало додаткове навантаження на клієнтську частину, оскільки вся обробка даних

виконується виключно на стороні браузера(див. рис. 5.1). У неоптимізованій версії ця логіка була реалізована без додаткових оптимізацій, що призводило до зайвих обчислень і повторних ререндерів компонентів при кожній зміні стану, наприклад, при введенні символу в полі пошуку чи при зміні критерію сортування. Оскільки список користувачів щоразу змінювався і оброблявся без кешування чи мемоізації, це створювало зайве навантаження на браузер і могло призводити до втрати продуктивності, особливо при великій кількості записів. Кожне введення символу у поле для пошуку чи зміна типу сортування викликало повторне створення масиву користувачів та заново виконувало сортування, що не є ефективним з точки зору продуктивності.

```tsx
type UsersListClientProps = { initialUsers?: User[] };
const MemoizedUserCard : React.NamedExoticComponent<Use... = memo(UserCard);
const UsersListClient = ({initialUsers}: UsersListClientProps) => {  Show usages  Anatolii_Filipenko *
    const [users : User[] , setUsers : React.Dispatch<React.SetStateA... ] = useState<User[]>( initialState: initialUsers || []);
    const [sortBy : "age" | "firstName" , setSortBy : React.Dispatch<React.SetStateA... ] = useState<"age" | "firstName">( initialState: "age");
    const [search : string , setSearch : React.Dispatch<React.SetStateA... ] = useState( initialState: "");
    const router : AppRouterInstance = useRouter();
    const sortedUsers : User[] = useMemo( factory: () => {
        return [...users]
            .filter((user : User ) => user.firstName.toLowerCase().includes(search.toLowerCase()))
            .sort( compareFn: (a : User , b : User ) : number => (sortBy === "age" ? a.age - b.age : a.firstName.localeCompare(b.firstName)));
    }, deps: [users, search, sortBy]);

    const handleUserClick = useCallback( callback: (id: number) : void => {
        router.replace( href: `/${id}`);
    }, deps: []);
    const handleSearchChange = (e: React.ChangeEvent<HTMLInputElement>) : void => {  Show usages  Anatolii_Filipenko
        setSearch(e.target.value);
    };

    return (
        <div className="p-3 bg-blue-200">
            <input
                type="text"
                placeholder="Пошук за ім'ям..."
                value={search}
                onChange={handleSearchChange}
                className="p-2 mb-4 w-full border rounded"
            />
            <div className="flex items-center gap-5">
                <button
                    onClick={() => setSortBy( value: sortBy === "age" ? "firstName" : "age")}
                    className="mb-4 p-2 bg-blue-500 text-white rounded"
                >
                    Sort by {sortBy === "age" ? "name" : "age"}
                </button>
                <div>
                    <PurchasesModal/>
                </div>
            </div>
            <div className="grid grid-cols-5 gap-4">
                {sortedUsers.map(user : User => (
                    <MemoizedUserCard onClick={handleUserClick} user={user} key={user.id}/>
                ))}
            </div>
        </div>
    );
};
```

Рисунок 5.1 – Код неоптимізованої версії головної сторінки (рисунок виконаний самостійно)

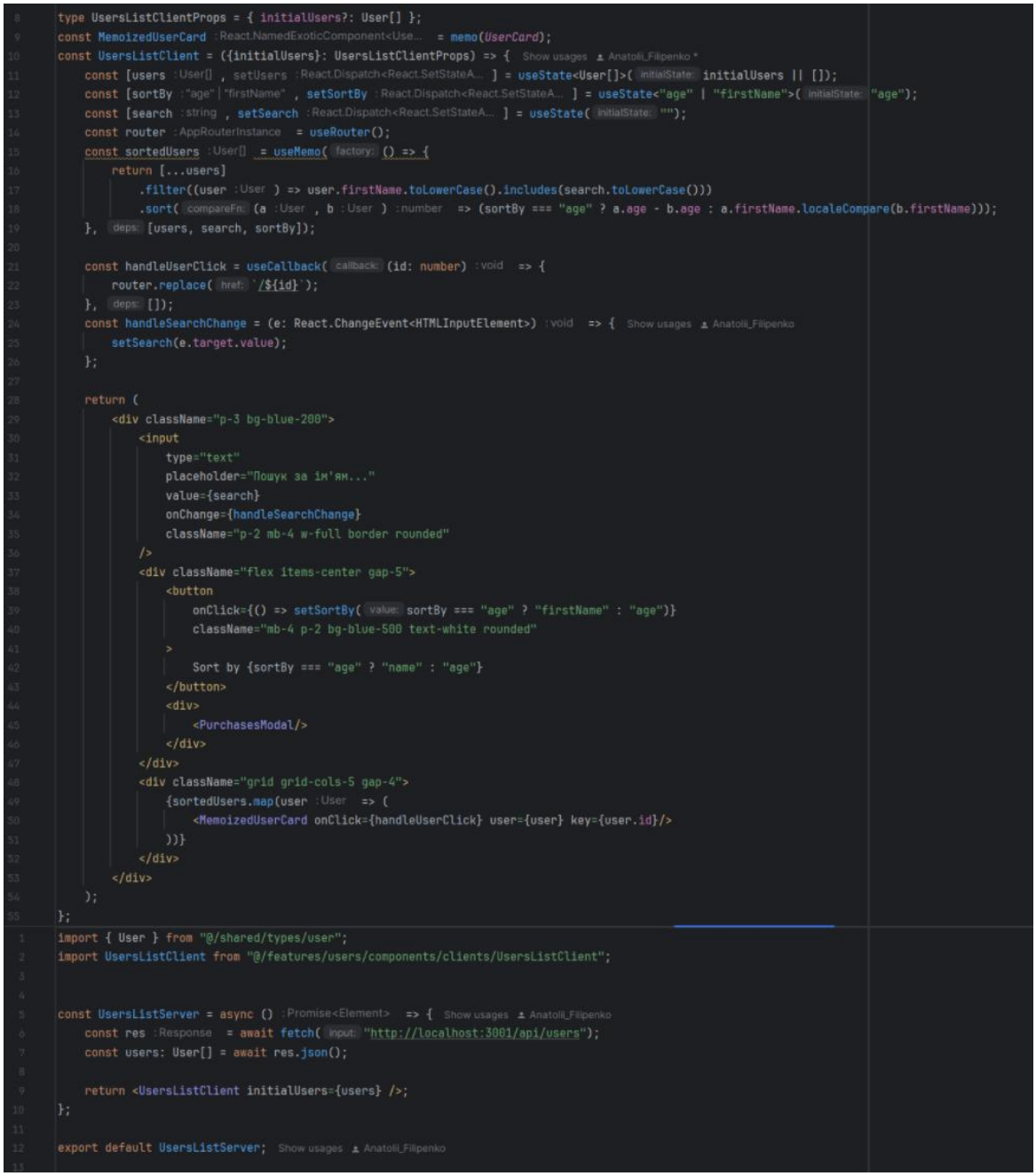
Щоб уникнути розбіжностей під час гідратації було застосовано підхід розділення компонентів на серверні (SSR) та клієнтські (CSR). Це дозволило забезпечити узгодженість даних між серверним і клієнтським рендерингом та зменшити навантаження на клієнта. Також, щоб зменшити вплив зайвих обчислень на швидкодію, було створено оптимізовану версію(див. рис. 5.2), у якій використано React-хуки useMemo, useCallback і React.memo.

Фільтрація та сортування користувачів були обгорнуті в useMemo, що дозволило кешувати результати цих обчислень і повторно їх виконувати лише у випадку зміни вихідних даних, тобто при оновленні списку користувачів, зміні критеріїв сортування або змінах у полі пошуку. Це дало змогу значно зменшити кількість викликів функції сортування та фільтрації, що позитивно вплинуло на продуктивність. Крім того, функція обробки кліку по користувачу була мемоізована за допомогою useCallback, який запобіг її повторному створенню при кожному ререндері, що є важливим аспектом при передачі функцій як пропсів дочірнім компонентам.

Також компонент UserCard, який відповідає за відображення окремого користувача, був обгорнутий у React.memo, що дозволило запобігти його непотрібному повторному ререндеру в тих випадках, коли його пропси не змінювалися. Завдяки цим змінам оптимізована версія значно ефективніше працює з великими списками користувачів, мінімізуючи кількість непотрібних обчислень та оновлень інтерфейсу, що дозволяє досягти кращого користувацького досвіду та зменшити навантаження на браузер.

Коли серверний компонент вкладений у клієнтський, його логіка виконується на сервері ще до того, як клієнтський почне рендеритись у браузері, але за умови, якщо серверний компонент складений як children. Якщо ж просто вкласти серверний компонент в клієнтський, то серверний компонент автоматично перетвориться на клієнтський. Серверний компонент повертає згенерований HTML, який вбудовується у клієнтський. У результаті, під час завантаження сторінки користувач одразу бачить зміст, отриманий із сервера, а клієнтський компонент лише додає інтерактивність, таку як обробка подій, оновлення стану

тощо. Завдяки цьому підходу сторінка швидко відображається в готовому вигляді ще до завантаження та виконання клієнтських скриптів.



```tsx
type UsersListClientProps = { initialUsers?: User[] };
const MemoizedUserCard : React.NamedExoticComponent<Use...  = memo(UserCard);
const UsersListClient = ({initialUsers}: UsersListClientProps) => {  Show usages  Anatolii_Filipenko *
    const [users : User[] , setUsers : React.Dispatch<React.SetStateA... ] = useState<User[]>( initialState: initialUsers || []);
    const [sortBy : "age" | "firstName" , setSortBy : React.Dispatch<React.SetStateA... ] = useState<"age" | "firstName">( initialState: "age");
    const [search : string , setSearch : React.Dispatch<React.SetStateA... ] = useState( initialState: "");
    const router : AppRouterInstance  = useRouter();
    const sortedUsers : User[]  = useMemo( factory: () => {
        return [...users]
            .filter((user : User ) => user.firstName.toLowerCase().includes(search.toLowerCase()))
            .sort( compareFn: (a : User , b : User ) : number  => (sortBy === "age" ? a.age - b.age : a.firstName.localeCompare(b.firstName)));
    },  deps: [users, search, sortBy]);

    const handleUserClick = useCallback( callback: (id: number) : void  => {
        router.replace( href: `/${id}`);
    },  deps: []);
    const handleSearchChange = (e: React.ChangeEvent<HTMLInputElement>) : void  => {  Show usages  Anatolii_Filipenko
        setSearch(e.target.value);
    };

    return (
        <div className="p-3 bg-blue-200">
            <input
                type="text"
                placeholder="Пошук за ім'ям..."
                value={search}
                onChange={handleSearchChange}
                className="p-2 mb-4 w-full border rounded"
            />
            <div className="flex items-center gap-5">
                <button
                    onClick={() => setSortBy( value: sortBy === "age" ? "firstName" : "age")}
                    className="mb-4 p-2 bg-blue-500 text-white rounded"
                >
                    Sort by {sortBy === "age" ? "name" : "age"}
                </button>
                <div>
                    <PurchasesModal/>
                </div>
            </div>
            <div className="grid grid-cols-5 gap-4">
                {sortedUsers.map(user : User  => (
                    <MemoizedUserCard onClick={handleUserClick} user={user} key={user.id}/>
                ))}
            </div>
        </div>
    );
};
```

```tsx
import { User } from "@/shared/types/user";
import UsersListClient from "@/features/users/components/clients/UsersListClient";


const UsersListServer = async () : Promise<Element>  => {  Show usages  Anatolii_Filipenko
    const res : Response  = await fetch( input: "http://localhost:3001/api/users");
    const users: User[] = await res.json();

    return <UsersListClient initialUsers={users} />;
};

export default UsersListServer;  Show usages  Anatolii_Filipenko
```

Рисунок 5.2 – Код оптимізованої версії головної сторінки з використанням серверного та клієнтського компонентів (рисунок виконаний самостійно)

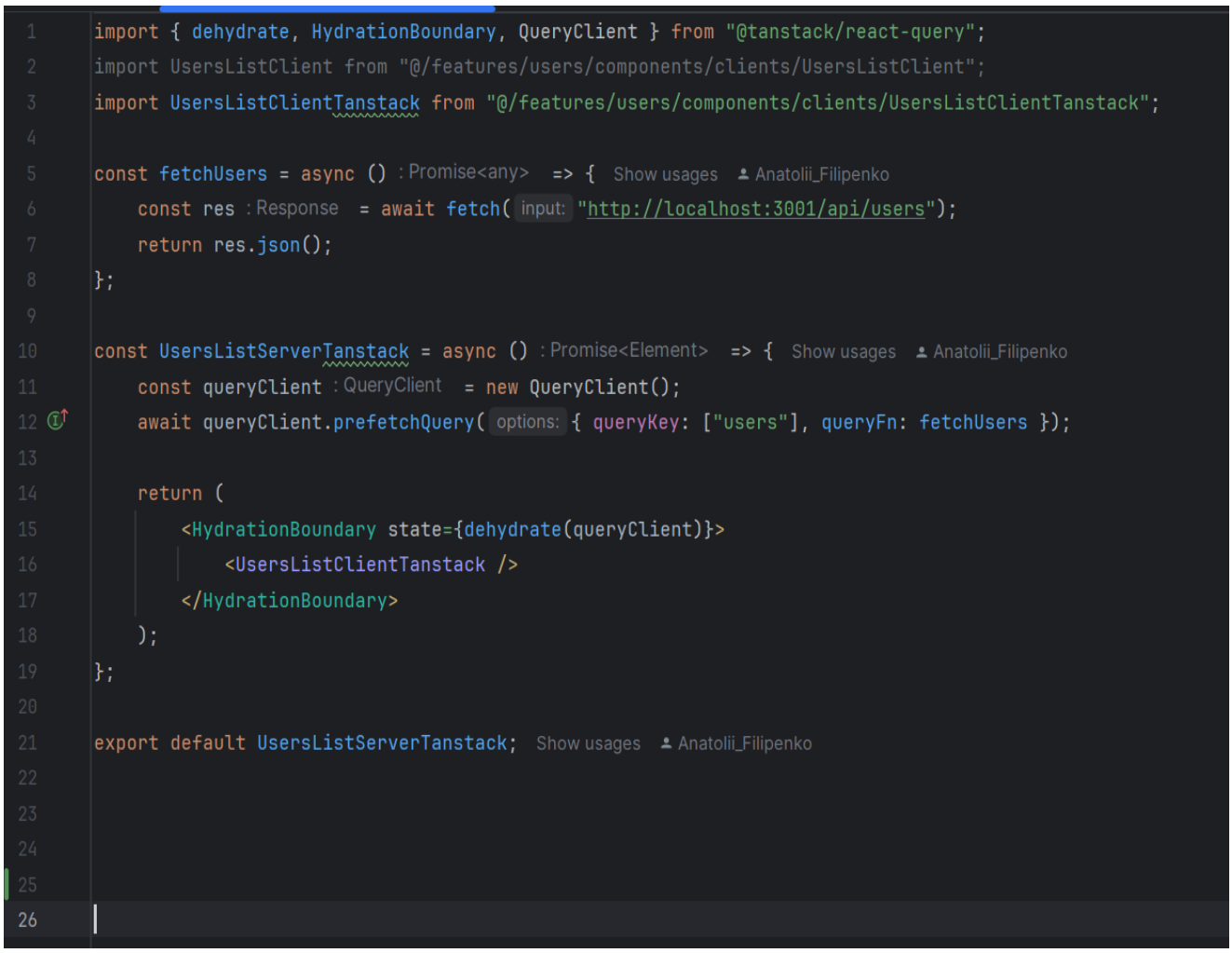
Зворотний випадок, коли клієнтський компонент вкладений в серверний, працює трохи інакше. Спочатку серверний компонент отримує всі необхідні дані і рендерить статичний HTML-код, що містить клієнтський. У цьому випадку клієнтський компонент отримує початкові значення як props, що передаються з

серверного компонента. Потім, коли сторінка завантажується у браузері, клієнтський компонент починає працювати вже на клієнті, додаючи інтерактивні елементи, оновлення стану та обробку подій. Таким чином забезпечується оптимальне використання можливостей серверного рендерингу для зменшення часу завантаження сторінки, а клієнтський компонент використовується для забезпечення динамічності та взаємодії з користувачем.

5.1.2 Оптимізація за допомогою TanStack Query

Якщо до підходу з використанням клієнтського компонента всередині серверного додати TanStack Query, то можна ще більше оптимізувати рендеринг. Серверний компонент може виконати запит до сервера та передати кешовані дані клієнтському компоненту, який використовуватиме їх як початковий стан. Після завантаження сторінки клієнтський компонент зможе автоматично перевіряти, чи не змінилися дані на сервері, і оновлювати їх у разі потреби. Завдяки цьому можна уникнути додаткових запитів на сервер під час першого завантаження сторінки, водночас зберігаючи можливість динамічного оновлення інформації. Такий підхід поєднує переваги серверного рендерингу, що забезпечує швидке завантаження сторінки, із гнучкістю клієнтського рендерингу, який дозволяє працювати з актуальними даними без необхідності перезавантажувати сторінку. Це робить застосунок швидшим, більш інтерактивним і зручним для користувачів.

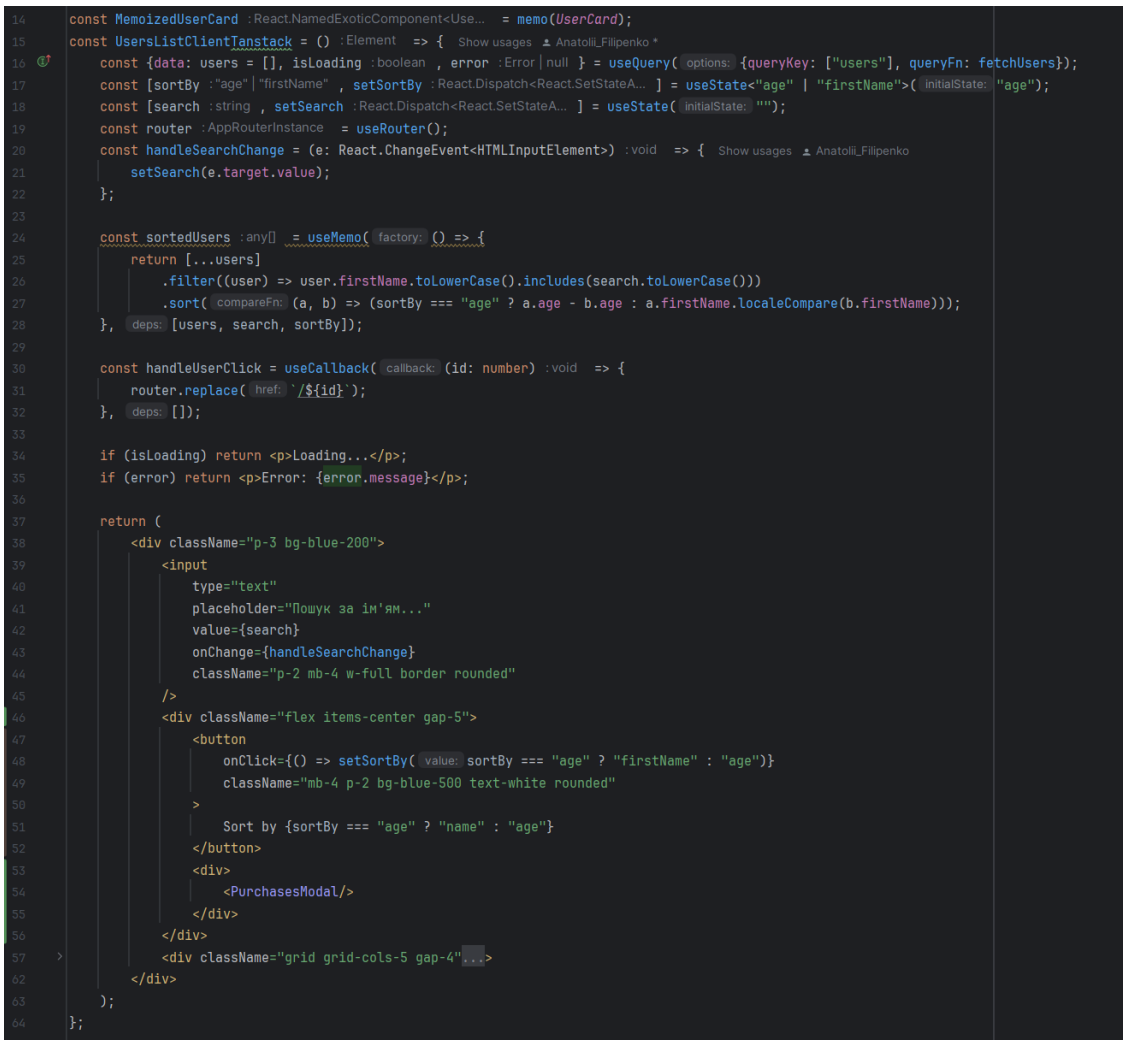
У практичній реалізації(див. рис. 5.3), серверний компонент UsersListServer використовує спеціальний метод prefetchQuery, який заздалегідь отримує дані з API ще до того, як HTML-код сторінки буде сформовано та надіслано браузеру. Після отримання цих даних серверний компонент зберігає результат запиту у спеціальному кеші, який передається до клієнта через компонент HydrationBoundary. Таким чином, при першому завантаженні сторінки користувач отримує готовий HTML, у якому вже присутні всі необхідні дані, а клієнтські компоненти можуть миттєво їх використати без додаткових мережевих запитів.



```tsx
import { dehydrate, HydrationBoundary, QueryClient } from "@tanstack/react-query";
import UsersListClient from "@/features/users/components/clients/UsersListClient";
import UsersListClientTanstack from "@/features/users/components/clients/UsersListClientTanstack";

const fetchUsers = async () : Promise<any>  => {  Show usages  Anatolii_Filipenko
    const res : Response  = await fetch( input: "http://localhost:3001/api/users");
    return res.json();
};

const UsersListServerTanstack = async () : Promise<Element>  => {  Show usages  Anatolii_Filipenko
    const queryClient : QueryClient  = new QueryClient();
    await queryClient.prefetchQuery( options: { queryKey: ["users"], queryFn: fetchUsers });

    return (
        <HydrationBoundary state={dehydrate(queryClient)}>
            <UsersListClientTanstack />
        </HydrationBoundary>
    );
};

export default UsersListServerTanstack;  Show usages  Anatolii_Filipenko
```

Рисунок 5.3 – Код оптимізованої серверної версії головної сторінки з використанням TanStack Query (рисунок виконаний самостійно)

У практичній реалізації клієнтського компонента(див. рис. 5.4) UsersListClient використовується хук useQuery від TanStack Query, який отримує початкові дані з цього самого кешу. Якщо, з якихось причин, дані не було отримано, то функція fetchUsers виконається на клієнті, що гарантує отримання даних користувачем у будь-кому разі. Якщо згодом інформація на сервері оновлюється, клієнтська частина автоматично виконує фоновий запит і оновлює свій стан без необхідності перезавантаження сторінки. Також TanStack Query додатково обробляє стани завантаження (isLoading) та помилок (error), що дозволяє зручно керувати інтерфейсом залежно від статусу отримання даних.



```
const MemoizedUserCard :React.NamedExoticComponent<Use... = memo(UserCard);
const UsersListClientTanstack = () :Element => {  Show usages  Anatolii_Filipenko *
    const {data: users = [], isLoading :boolean , error :Error | null } = useQuery( options: {queryKey: ["users"], queryFn: fetchUsers});
    const [sortBy :"age" | "firstName" , setSortBy :React.Dispatch<React.SetStateA... ] = useState<"age" | "firstName">( initialState: "age");
    const [search :string , setSearch :React.Dispatch<React.SetStateA... ] = useState( initialState: "");
    const router :AppRouterInstance = useRouter();
    const handleSearchChange = (e: React.ChangeEvent<HTMLInputElement>) :void => {  Show usages  Anatolii_Filipenko
        setSearch(e.target.value);
    };

    const sortedUsers :any[] = useMemo( factory: () => {
        return [...users]
            .filter((user) => user.firstName.toLowerCase().includes(search.toLowerCase()))
            .sort( compareFn: (a, b) => (sortBy === "age" ? a.age - b.age : a.firstName.localeCompare(b.firstName)));
    }, deps: [users, search, sortBy]);

    const handleUserClick = useCallback( callback: (id: number) :void => {
        router.replace( href: `/${id}`);
    }, deps: []);

    if (isLoading) return <p>Loading...</p>;
    if (error) return <p>Error: {error.message}</p>;

    return (
        <div className="p-3 bg-blue-200">
            <input
                type="text"
                placeholder="Пошук за ім'ям..."
                value={search}
                onChange={handleSearchChange}
                className="p-2 mb-4 w-full border rounded"
            />
            <div className="flex items-center gap-5">
                <button
                    onClick={() => setSortBy( value: sortBy === "age" ? "firstName" : "age")}
                    className="mb-4 p-2 bg-blue-500 text-white rounded"
                >
                    Sort by {sortBy === "age" ? "name" : "age"}
                </button>
                <div>
                    <PurchasesModal/>
                </div>
            </div>
            <div className="grid grid-cols-5 gap-4"...>
        </div>
    );
};
```

Рисунок 5.4 – Код оптимізованої клієнтської версії головної сторінки з використанням TanStack Query (рисунок виконаний самостійно)

Таким чином, застосування TanStack Query для оптимізації рендерингу поєднує переваги серверного та клієнтського підходів, забезпечуючи швидке завантаження сторінки, зниження навантаження на сервер, а також автоматичне оновлення даних у фоновому режимі. Це дозволяє створити більш ефективний, швидкий та зручний у використанні застосунок, покращуючи загальний користувацький досвід.

5.1.3 Оптимізація за допомогою віртуалізації списків

Для подальшого зменшення часу відображення та уникнення надмірного завантаження клієнтської частини під час роботи з великими обсягами даних було впроваджено механізм віртуалізації списків за допомогою бібліотеки react-

window(див. рис. 5.5). Суть цього методу полягає в тому, що у видимій частині екрану відображаються виключно ті елементи, які наразі потрапляють у поле зору користувача, а решта не рендеряться у DOM до моменту, коли користувач прокручує сторінку. У межах роботи було застосовано компонент FixedSizeGrid, що дає змогу відтворити поведінку, аналогічну фреймворку Tailwind CSS з його grid-сіткою. Спеціальні параметри, такі як columnCount, columnWidth, rowCount та rowHeight, визначають сітку з певною кількістю колонок і задають точні розміри кожного відображуваного елементу. Завдяки цьому навіть коли список містить сотні записів, застосунок не перевантажується при відтворенні всього обсягу даних одночасно, а обмежується лише поточним вікном видимості.

```
const VirtualizedUsersGrid = () : Element  => {  Show usages  Anatolii_Filipenko *
    const { data: users = [], isLoading : boolean , error : Error | null  } = useQuery( options: { queryKey: ["users"], queryFn: fetchUsers });
    const [sortBy : "age" | "firstName" , setSortBy : React.Dispatch<React.SetStateA... ] = useState<"age" | "firstName">( initialState: "age");
    const [search : string , setSearch : React.Dispatch<React.SetStateA... ] = useState( initialState: "");
    const router : AppRouterInstance  = useRouter();

    const handleUsersPurchasesClick = () : void  => {...}
    const handleSearchChange = useCallback( callback: (e: React.ChangeEvent<HTMLInputElement>) : void  => {
        setSearch(e.target.value);
    }, deps: []);
    const sortedUsers : any[]  = useMemo( factory: () => {...}, deps: [users, search, sortBy]);
    const handleUserClick = useCallback( callback: (id: number) : void  => {...}, deps: []);
    if (isLoading) return <p>Loading...</p>;
    if (error) return <p>Error: {error.message}</p>;
    const rowCount : number  = Math.ceil( x: sortedUsers.length / COLUMN_COUNT);
    const Cell = ({ columnIndex, rowIndex, style }: any) : null | Element  => {  Show usages  Anatolii_Filipenko
        const index = rowIndex * COLUMN_COUNT + columnIndex;
        const user = sortedUsers[index];
        if (!user) return null;
        return (
            <div style={style} className="p-2">
                <MemoizedUserCard
                    user={user}
                    onClick={handleUserClick}
                />
            </div>
        );
    };
    return (
        <div className="p-3 bg-blue-200">
            <button onClick={handleUsersPurchasesClick} className="my-3 text-purple underline cursor-pointer">Users
                purchases
            </button>
            <input
                type="text"
                placeholder="search for name"
                value={search}
                onChange={handleSearchChange}
                className="p-2 mb-4 w-full border rounded"
            />
            <div className="flex items-center gap-5"...>
            <Grid
                columnCount={COLUMN_COUNT}
                columnWidth={CARD_WIDTH - 10}
                height={794}
                rowCount={rowCount}
                rowHeight={CARD_HEIGHT}
                width={CARD_WIDTH * COLUMN_COUNT}
            >
                {Cell}
            </Grid>
        </div>
    );
```

Рисунок 5.5 – Код оптимізованої клієнтської версії головної сторінки з використанням віртуалізації за допомогою бібліотеки react-window (рисунок виконаний самостійно)

Для уніфікації процесу отримання та відображення даних, як і в попередніх прикладах, використовується TanStack Query, завдяки якому серверний компонент спочатку виконує prefetchQuery для отримання списку користувачів. Після цього збережені в кеші результати передаються на клієнт через HydrationBoundary, щоб користувач уже на момент першого завантаження сторінки міг миттєво взаємодіяти з готовим інтерфейсом. На клієнті компонент VirtualizedUsersGrid отримує ці дані та, враховуючи задані параметри висоти та ширини, рендерить елементи списку у вигляді сітки. При прокручуванні сторінки react-window розраховує, які саме комірки мають бути зараз у DOM, а які можна проігнорувати, заощаджуючи обчислювальні ресурси й уникнувши надмірних ререндерів. Крім цього, попередньо реалізовані оптимізації за допомогою useMemo та useCallback і надалі гарантують, що функції, відповідальні за сортування, фільтрацію та обробку подій, не викликатимуться повторно при кожному оновленні стану, а компонент UserCard, обгорнутий у React.memo, не оновлюватиметься без потреби.

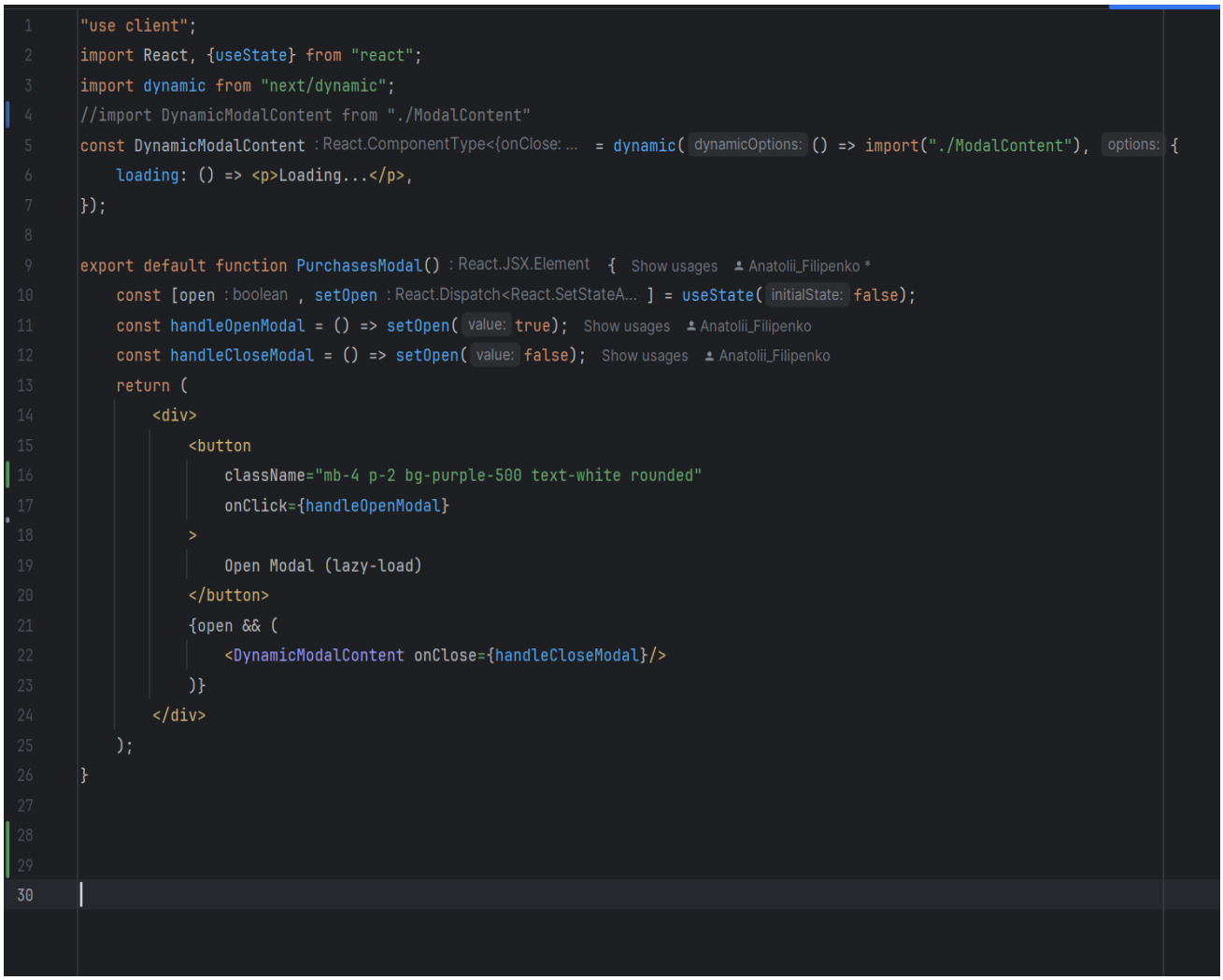
5.1.4 Оптимізація за допомогою lazy loading

Додатково до вже реалізованих оптимізацій було створено модальне вікно PurchasesModal(див. рис. 5.6), яке призначена для демонстрації списку останніх покупок користувачів. Дані про покупки завантажуються локально для імітації важкої компоненти яка містить багато контенту. Відповідно, отриманий список покупок користувачів містить зображення товарів, назви, дату покупки та ідентифікатор користувача.

Через значний обсяг зображень та інформації, що має бути відображена у модальному вікні, було застосовано метод lazy loading, який суттєво знижує навантаження на клієнтський додаток і скорочує час первинного завантаження сторінки. У контексті роботи з Next.js цей підхід вже інтегрований у компонент Image, який автоматично завантажує зображення тільки тоді, коли вони потрапляють у зону видимості користувача (viewport).

Для подальшої оптимізації та прискорення роботи сторінки було використано механізм dynamic import, наданий бібліотекою Next.js через функцію

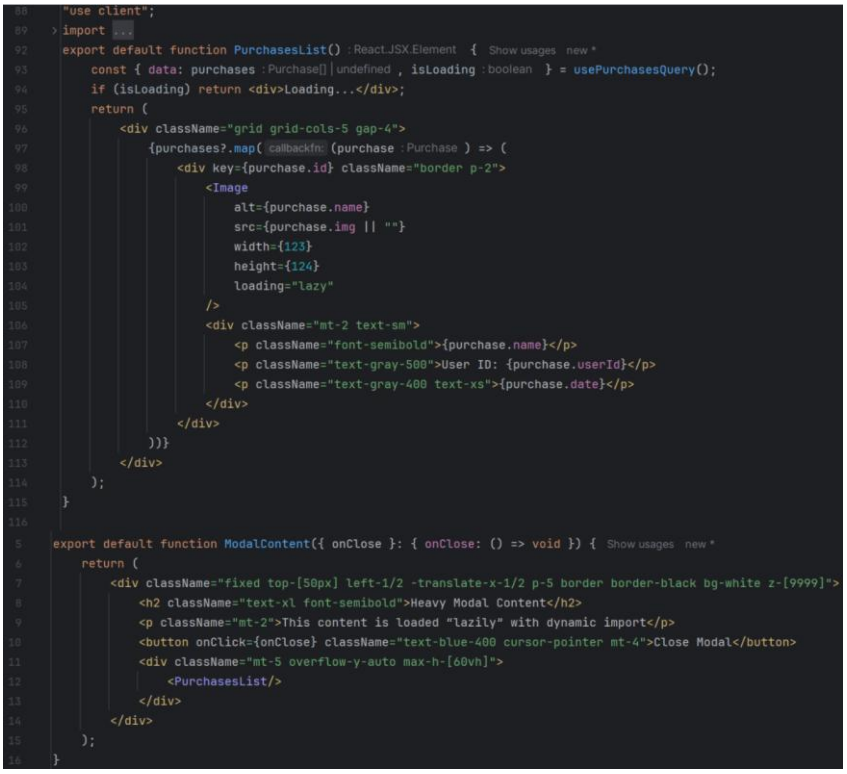
dynamic(). Цей підхід є зручною альтернативою класичним React.lazy та Suspense, що дозволяє динамічно завантажувати компоненти на вимогу, тобто лише тоді, коли користувач дійсно взаємодіє з певним елементом інтерфейсу.



```
"use client";
import React, {useState} from "react";
import dynamic from "next/dynamic";
//import DynamicModalContent from "./ModalContent"
const DynamicModalContent : React.ComponentType<{onClose: ... = dynamic( dynamicOptions: () => import("./ModalContent"), options: {
    loading: () => <p>Loading...</p>,
});

export default function PurchasesModal() : React.JSX.Element { Show usages Anatolii_Filipenko *
    const [open : boolean , setOpen : React.Dispatch<React.SetStateA... ] = useState( initialState: false);
    const handleOpenModal = () => setOpen( value: true); Show usages Anatolii_Filipenko
    const handleCloseModal = () => setOpen( value: false); Show usages Anatolii_Filipenko
    return (
        <div>
            <button
                className="mb-4 p-2 bg-purple-500 text-white rounded"
                onClick={handleOpenModal}
            >
                Open Modal (lazy-load)
            </button>
            {open && (
                <DynamicModalContent onClose={handleCloseModal}/>
            )}
        </div>
    );
}
```

Рисунок 5.6 – Код оптимізованої версії компоненти модального вікна PurchasesModal з використанням lazy loading (рисунок виконаний самостійно)

У даному випадку компонент модального вікна ModalContent(див. рис. 5.7), що містить список покупок користувачів, завантажується лише після натискання кнопки Open Modal. Таким чином, сторінка Users Purchases завантажується швидше, оскільки важкий модальний контент, що містить зображення та інші дані, не завантажується заздалегідь, а тільки в момент, коли користувач ініціює його перегляд.



```
"use client";
import ...
export default function PurchasesList() : React.JSX.Element  {  Show usages  new *
    const { data: purchases : Purchase[] | undefined , isLoading : boolean  } = usePurchasesQuery();
    if (isLoading) return <div>Loading...</div>;
    return (
        <div className="grid grid-cols-5 gap-4">
            {purchases?.map( callbackFn: (purchase : Purchase ) => (
                <div key={purchase.id} className="border p-2">
                    <Image
                        alt={purchase.name}
                        src={purchase.img || ""}
                        width={123}
                        height={124}
                        loading="lazy"
                    />
                    <div className="mt-2 text-sm">
                        <p className="font-semibold">{purchase.name}</p>
                        <p className="text-gray-500">User ID: {purchase.userId}</p>
                        <p className="text-gray-400 text-xs">{purchase.date}</p>
                    </div>
                </div>
            ))}
        </div>
    );
}

export default function ModalContent({ onClose }: { onClose: () => void }) {  Show usages  new *
    return (
        <div className="fixed top-[50px] left-1/2 -translate-x-1/2 p-5 border border-black bg-white z-[9999]">
            <h2 className="text-xl font-semibold">Heavy Modal Content</h2>
            <p className="mt-2">This content is loaded “lazily” with dynamic import</p>
            <button onClick={onClose} className="text-blue-400 cursor-pointer mt-4">Close Modal</button>
            <div className="mt-5 overflow-y-auto max-h-[60vh]">
                <PurchasesList/>
            </div>
        </div>
    );
}
```

Рисунок 5.7 – Код оптимізованої версії сторінки PurchasesList з використанням Image які мають lazy loading(рисунок виконаний самостійно)

Такий підхід дає значну перевагу в оптимізації продуктивності додатку, особливо в ситуаціях, коли модальні вікна або інші компоненти мають значний обсяг додаткових даних чи інтерактивних елементів. Завдяки реалізації динамічного імпорту та вбудованим механізмам lazy loading зображень Next.js вдалося досягти високої швидкості роботи застосунку, оптимізувати витрати ресурсів на клієнтській стороні та забезпечити якісний користувацький досвід без небажаних затримок при завантаженні.

5.1.5 Опис структури застосунків

У фінальній, оптимізованій версії застосунку(див. рис. 5.8) в папці app розміщено основні маршрути Next.js разом із глобальними стилями та компонентами. Для створення нових маршрутів, Next.js вимагає створювати папки всередині app, назва яких і буде використовуватись у якості маршруту. Щоб визначити контент для відображення, створюється файл з назвою page.tsx, в якому відображаються потрібні компоненти.

Статичні ресурси, такі як зображення, знаходяться у папці assets. У features згруповано логіку за окремими модулями й відповідними компонентами. Папка providers містить провайдери (наприклад React Query або Redux Provider для підключення глобального стану), а в services розташована логіка для отримання даних з сервера. Папка shared включає повторно використовувані компоненти, хуки, типи та утиліти. Завдяки такій структурі легше впроваджувати оптимізації й тримати все організовано.

Для управління глобальним станом застосовано Redux Toolkit, який знаходиться у папці store, де зберігаються слайси, кастомні хуки та файл ініціалізації store. Наприклад, у store/slices/usersSlice.ts зберігається логіка керування користувачами, а ReduxProvider.tsx в app/providers відповідає за обгортання усього застосунку Redux-провайдером. Такий підхід дозволяє централізовано керувати станом, повторно використовувати логіку і легко масштабувати застосунок.

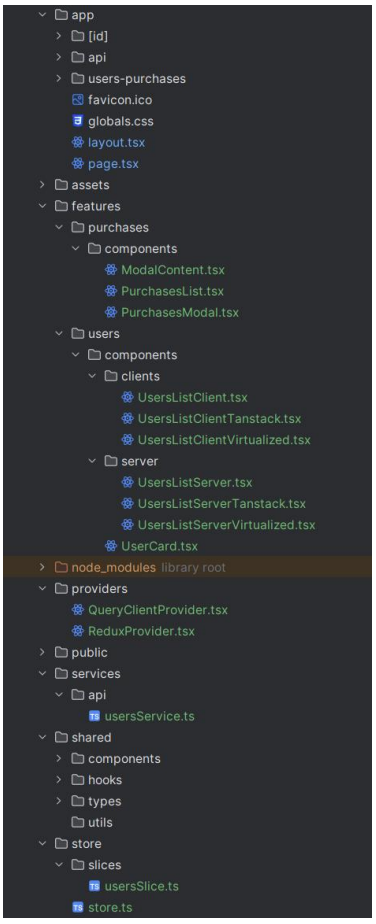


Рисунок 5.8 – Структура оптимізованої версії застосунку (рисунок виконаний самостійно)

У неоптимізованій версії застосунку(дис. рис. 5.9) структура проєкту є спрощеною та менш модульною. Всі компоненти зібрані в одну загальну папку components, і всередині розділені за категоріями(наприклад users або purchases). Це є часта практика, але при зростанні проекту ця папка дуже швидко заповнюється та стає незручно орієнтуватись всередині неї. Папка app містить маршрути та базову структуру Next.js, включно з глобальними стилями і компонуванням (layout.tsx, page.tsx). Серверні запити розміщені в папці services, зокрема файл usersService.ts відповідає за отримання даних про користувачів. Статичні ресурси застосунку зберігаються в assets. Є також окремі папки shared для спільного коду, store для стану та types для типізації. Такий підхід зручний на ранніх етапах, але в умовах росту застосунку швидко призводить до змішування відповідальностей та ускладнює масштабування і оптимізацію.

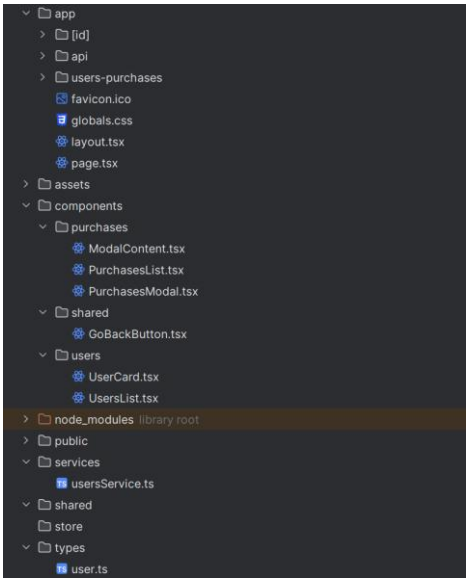


Рисунок 5.9 – Структура неоптимізованої версії застосунку (рисунок виконаний самостійно)

Отже, можна зробити висновок, що в більшості випадків, структура в оптимізованій версії застосунку є більш універсальною та зручною.

5.2 Проведення тестування програмного забезпечення

Для тестування веб-застосунку було обрано інструмент Lighthouse, що є вбудованим рішенням для аналізу продуктивності веб-сторінок від компанії

Google. Вибір саме цього інструмента був зроблений через його комплексний підхід до оцінювання ключових показників продуктивності (performance), доступності (accessibility), найкращих практик (best practices) та оптимізації для пошукових систем (SEO).

Тестування проводилося для головної сторінки веб-застосунку, де з серверу відразу завантажувалося 225 записів користувачів. Це рішення було прийняте для того, щоб максимально навантажити застосунок та виявити різницю між оптимізованою та неоптимізованою версіями як мобільної, так і десктопної версій.

Спочатку було проведено тестування для неоптимізованої версії(див. рис. 5.10). Результати показали, що неоптимізована десктопна версія демонструє не високий рівень продуктивності – 86 балів зі 100. Загальний час блокування складає 290 мілісекунд. Найбільший контентний елемент завантажується за 1,2 секунди, хоча перший контент відображається досить швидко – за 0,2 секунди. Інші параметри залишаються на прийнятному рівні.

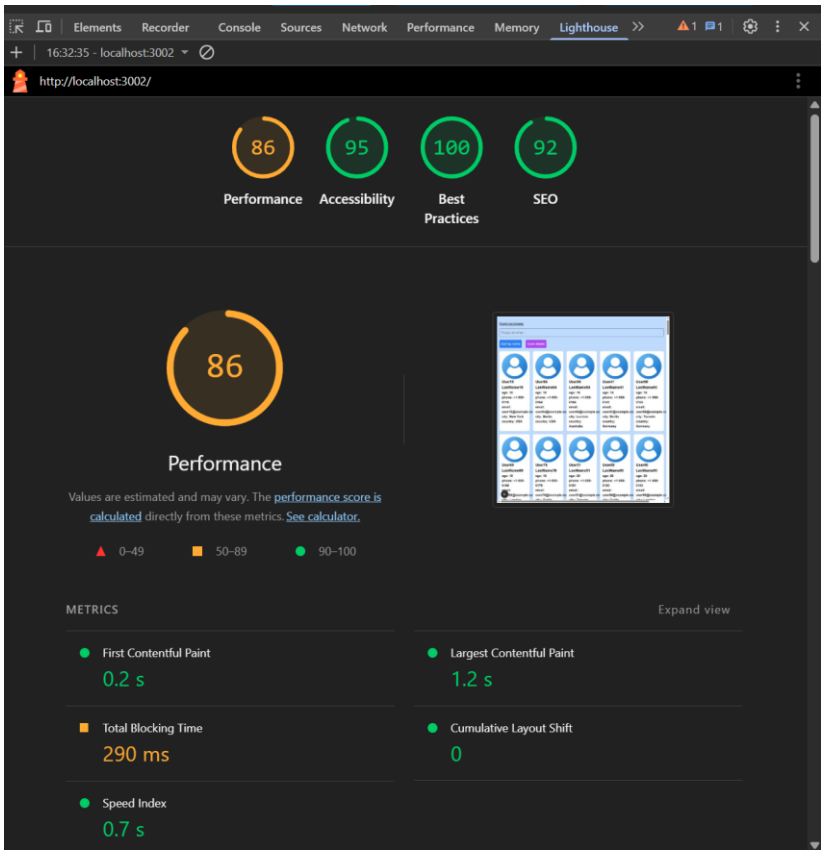


Рисунок 5.10 – Результати тестування за допомогою Lighthouse десктопної неоптимізованої версії застосунку (рисунок виконаний самостійно)

Суттєве погіршення продуктивності спостерігається у неоптимізованій мобільній версії(див. рис. 5.11) – 53 балів. Час завантаження найбільшого контентного елемента збільшився до 5,2 секунди. Загальний час блокування критично виріс до 1930 мілісекунд. Інші показники теж значно погіршились, перший контент відображається за 0,8 секунд, а індекс швидкості – 1,4 секунди. Це свідчить про критичну необхідність оптимізації мобільної версії для комфортної роботи користувачів.

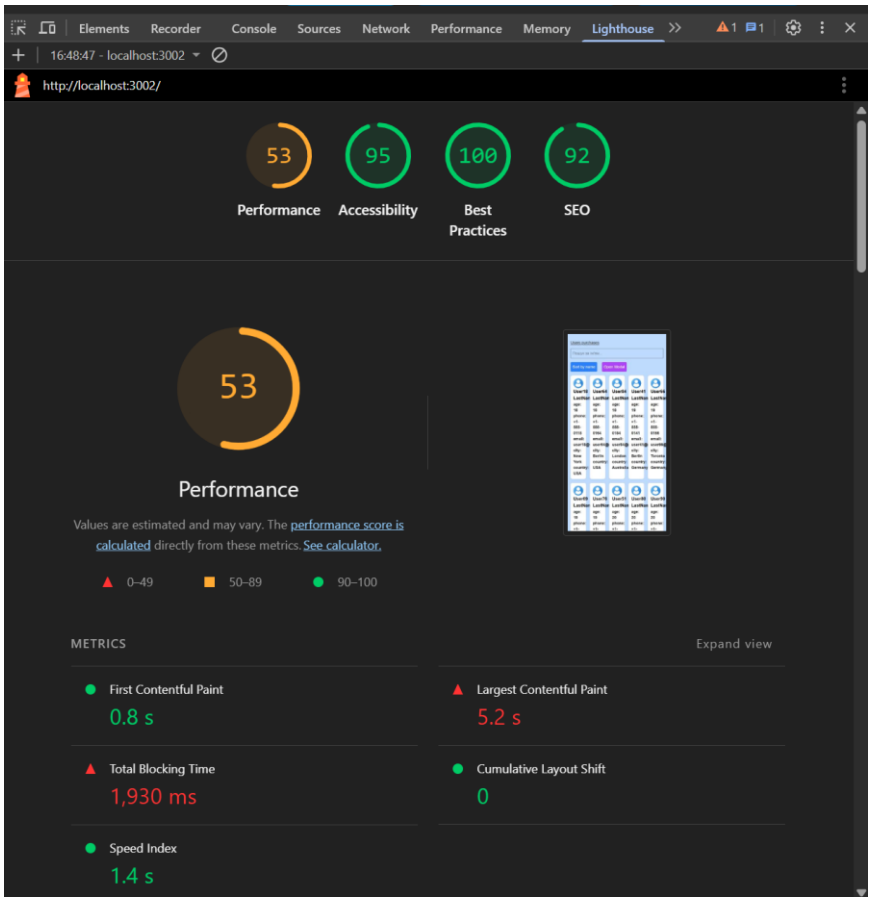


Рисунок 5.11 – Результати тестування за допомогою Lighthouse мобільної неоптимізованої версії застосунку (рисунок виконаний самостійно)

Далі, для кращого розуміння результатів оптимізацій кожного методу, було проведено поступове тестування. Спочатку, було проведено тестування для версії застосунку, в якому реалізовано мемоізацію та чітке розбиття на клієнтські та серверні компоненти. Для десктопної версії отримали набагато кращі результати продуктивності – 98 балів зі 100(див. рис. 5.12). Значно кращим став показник загального часу блокування – 50 мілісекунд. Показник FCP став 0,3 секунди, а час

завантаження найбільшого контентного елемента став 1 секунду. Отже, завдяки додаванню SSR, тобто створенню серверної компоненти яка робить запит на бекенд, і віддає дані через props клієнтській компоненті, було досягнуто значних оптимізацій. Крім того слід враховувати, що завдяки мемоізації, кожен раз при зміні стану, наприклад, відкритті та закритті модального вікна, не відбувається повторного рендерингу усього списку компонентів, що дає надзвичайну перевагу під час користування веб-застосунком.

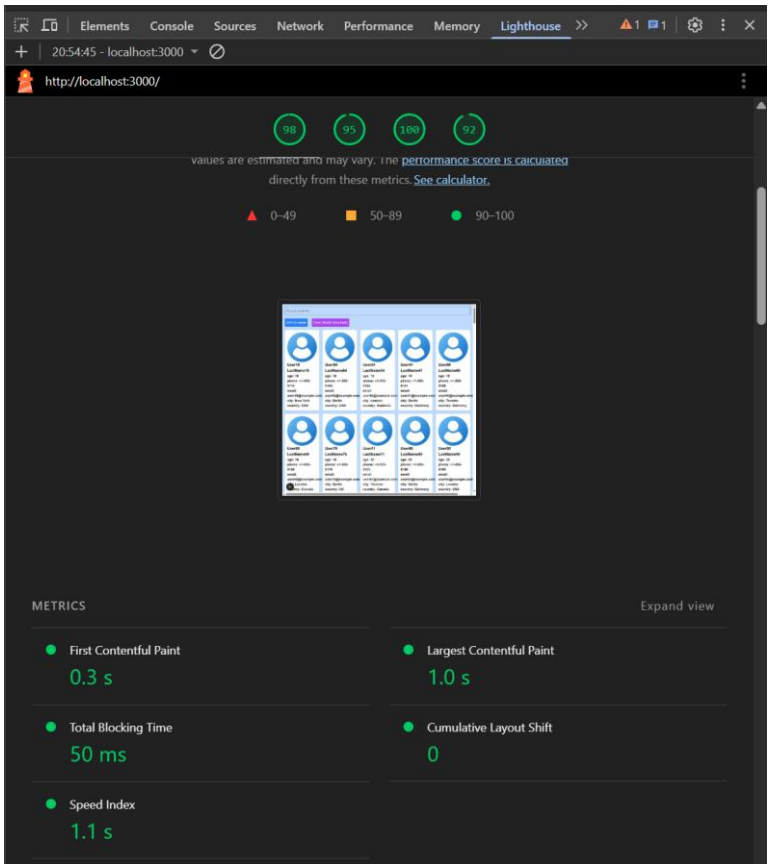


Рисунок 5.12 – Результати тестування за допомогою Lighthouse десктопної оптимізованої версії застосунку з додаванням SSR та мемоізації (рисунок виконаний самостійно)

Суттєве покращення продуктивності спостерігається у мобільній оптимізованій версії застосунку (див. рис. 5.13), в якій було реалізовано розділення компонентів на серверні та клієнтські, а також застосовано мемоізацію. Показник продуктивності збільшився до 88 балів, що є значним стрибком у порівнянні з неоптимізованою версією.

Перший контентний елемент (FCP) завантажується за 0,9 секунди, а індекс швидкості (Speed Index) склав 1,9 секунди, що є хорошим результатом для мобільних пристроїв. Значно зменшився загальний час блокування (Total Blocking Time) – до 230 мілісекунд, що покращує взаємодію користувача з інтерфейсом. Хоча показник завантаження найбільшого контентного елемента (LCP) ще становить 3,3 секунди, це суттєво краще за результат неоптимізованої версії. Завдяки використанню SSR, дані були підготовлені ще на сервері, що зменшило час до появи візуального контенту. А мемоізація компонентів забезпечила уникнення непотрібних повторних рендерів під час взаємодії зі сторінкою. Усе це свідчить про ефективність обраних методів оптимізації також і для мобільної версії застосунку.

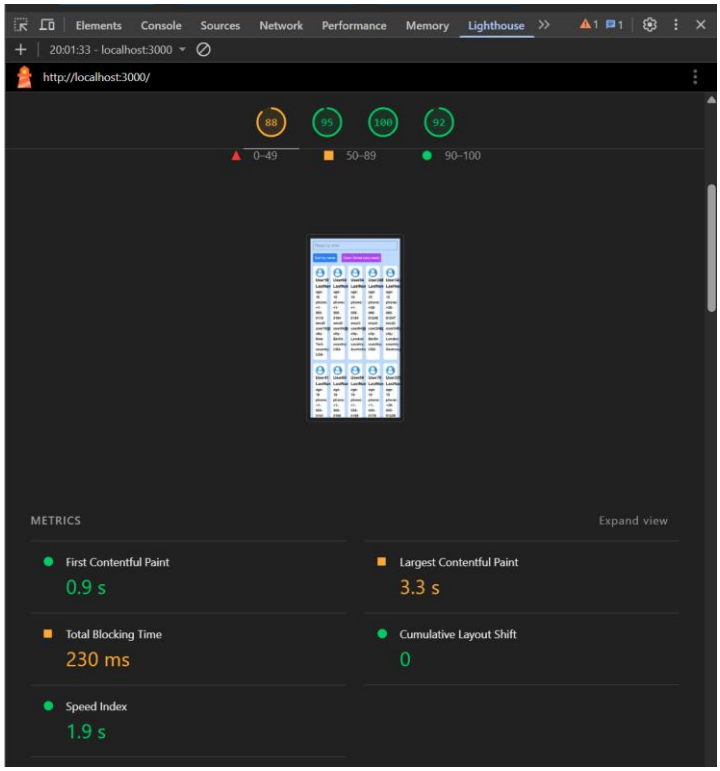


Рисунок 5.13 – Результати тестування за допомогою Lighthouse мобільної оптимізованої версії застосунку з додаванням SSR та мемоізації (рисунок виконаний самостійно)

Наступним кроком стало проведення тестування для десктопної версії з інтеграцією бібліотеки TanStack Query для оптимізації роботи з даними (див. рис. 6.14). Результати тестування показали, що показники продуктивності залишилися

майже незмінними – 99 балів, що є найвищим результатом серед усіх попередніх версій. Перший контентний елемент з'являється вже за 0,2 секунди, найбільший – за 0,9 секунди, а загальний час блокування скоротився до мінімальних 10 мілісекунд. Це свідчить про високий рівень продуктивності та швидкої взаємодії з інтерфейсом навіть при великому обсязі даних.

Попри те, що у звіті Lighthouse не відображено суттєвого приросту продуктивності порівняно з попередньою оптимізацією, додавання TanStack Query має важливу практичну цінність. Бібліотека забезпечує автоматичне кешування даних, повторне використання запитів без необхідності рендеру, а також оновлення інформації у фоновому режимі. Це значно покращує досвід користувача під час реального використання застосунку, особливо у випадках повторного відвідування сторінок або взаємодії з великими обсягами динамічних даних. Таким чином, навіть без суттєвого впливу на синтетичні метрики, TanStack Query підвищує стабільність та ефективність застосунку у повсякденному використанні.

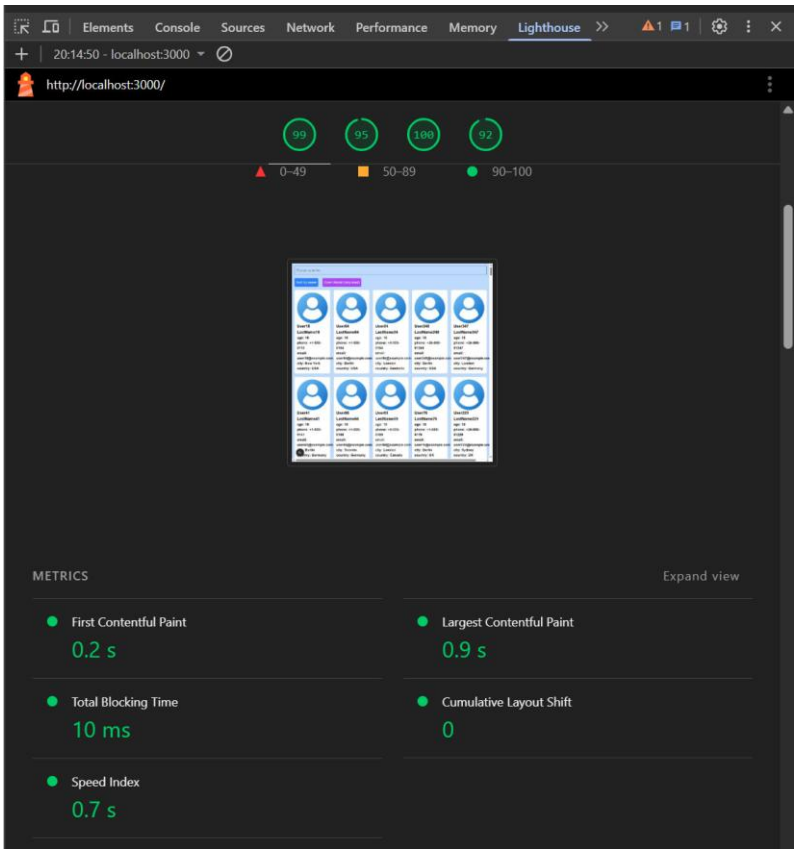


Рисунок 5.14 – Результати тестування за допомогою Lighthouse десктопної оптимізованої версії застосунку з додаванням TanStack Query (рисунок виконаний самостійно)

Далі було отримано результати тестування мобільної версії застосунку після впровадження бібліотеки TanStack Query(дис. рис. 5.16). Показник продуктивності залишився практично на тому ж рівні – 89 балів, як і у попередній версії з SSR та мемоізацією. Час першого контентного елемента склав 0,9 секунди, а індекс швидкості – 1,8 секунди. Загальний час блокування знизився до 130 мілісекунд, а час завантаження найбільшого контентного елемента (LCP) становив 3,6 секунди.

Попри невелике зниження деяких метрик, загальна стабільність інтерфейсу залишилася на хорошому рівні, а переваги від TanStack Query, такі як кешування, передзавантаження та уникнення зайвих запитів, будуть особливо помітні саме під час реального користування застосунком.

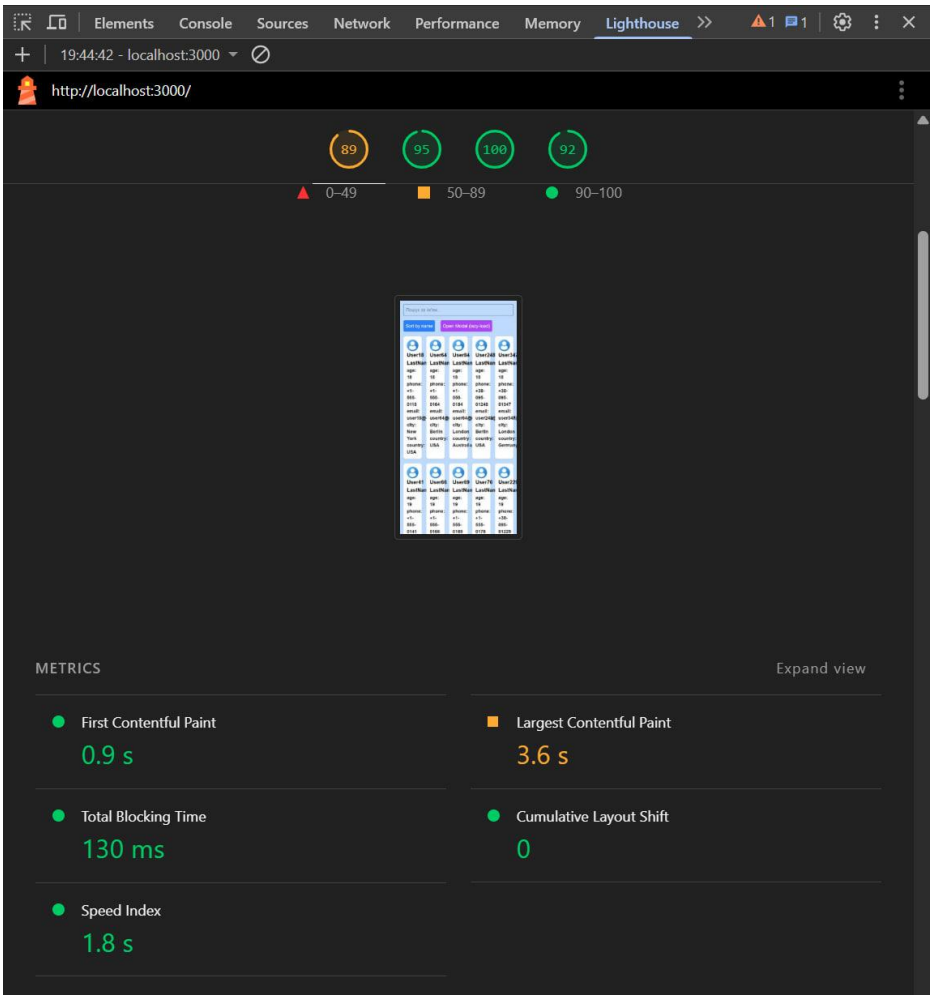


Рисунок 5.15 – Результати тестування за допомогою Lighthouse мобільної оптимізованої версії застосунку з додаванням TanStack Query (рисунок виконаний самостійно)

Наступним етапом оптимізації стало проведення тестування з реалізацією віртуалізації списків у веб-застосунку (див. рис. 5.16). Завдяки цьому підходу досягнуто найвищих показників продуктивності – 99 балів із 100. Перший контентний елемент (FCP) з'являється вже через 0,3 секунди, індекс швидкості (Speed Index) також становить лише 0,3 секунди, а найбільший контентний елемент (LCP) завантажується за 0,9 секунди. Загальний час блокування (Total Blocking Time) взагалі відсутній – 0 мілісекунд, що свідчить про миттєву реакцію інтерфейсу без затримок.

Віртуалізація дозволяє рендерити лише ті елементи списку, які безпосередньо видимі на екрані користувача, а не всі 225 записів одночасно. Це значно знижує навантаження на браузер під час першого рендеру, покращує плавність прокручування та скорочує кількість обчислювальних операцій. Таким чином, навіть при роботі з великим обсягом даних, інтерфейс залишається максимально швидким і чутливим до дій користувача.

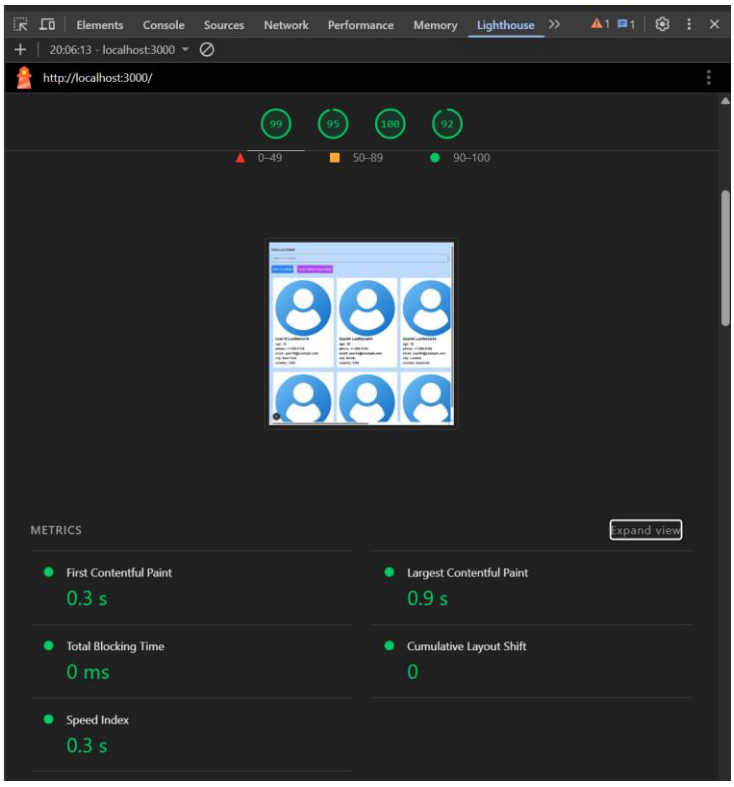


Рисунок 5.15 – Результати тестування за допомогою Lighthouse десктопної оптимізованої версії застосунку з додаванням віртуалізації списків (рисунок виконаний самостійно)

Для мобільної версії з додаванням віртуалізації списків показник продуктивності досягнув 92 балів, що свідчить про високу оптимізацію навіть на мобільних пристроях(див. рис. 5.16).

Перший контентний елемент (FCP) завантажується за 0,9 секунди, індекс швидкості (Speed Index) також склав 0,9 секунди, що є відмінним результатом. Загальний час блокування скоротився до 70 мілісекунд, а найбільший контентний елемент (LCP) завантажується за 3,3 секунди.

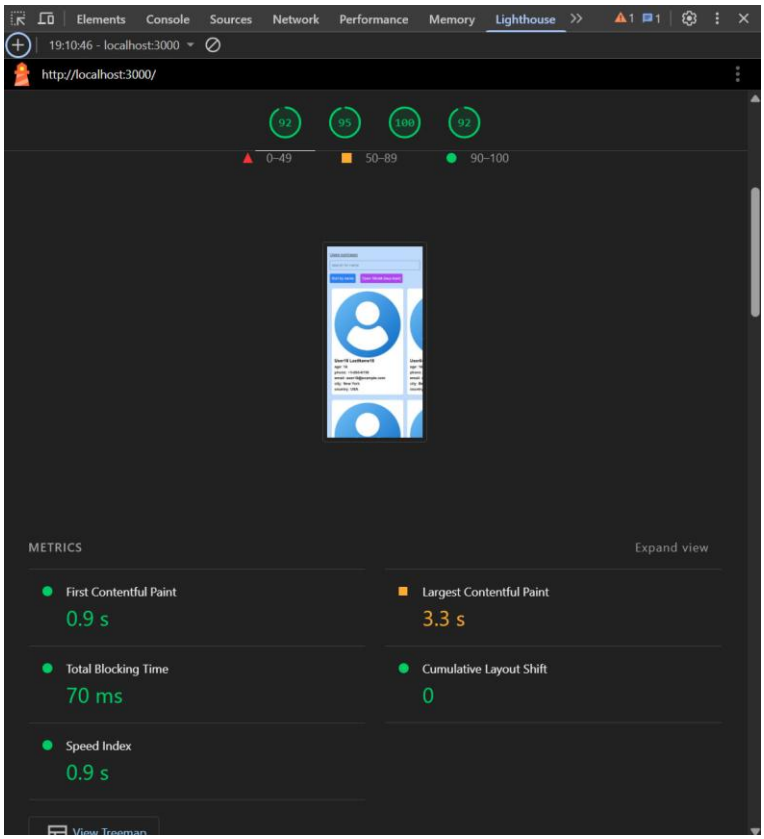


Рисунок 5.16 – Результати тестування за допомогою Lighthouse мобільної оптимізованої версії застосунку з додаванням віртуалізації списків (рисунок виконаний самостійно)

Окрім того, в оптимізованих версіях було додано динамічний імпорт модального вікна, що означає, що код цієї компоненти не включається до початкового JavaScript-бандлу. Завдяки цьому зменшується обсяг початкових ресурсів, які потрібно завантажити браузеру, що позитивно впливає на час рендерингу головної сторінки.

Крім того, зображення, які використовуються в модальному вікні, також завантажуються ліниво (lazy load), тобто лише у момент відкриття модального вікна, а не одразу при завантаженні сторінки. Це дозволяє зменшити навантаження на мережу і забезпечити швидше завантаження початкового інтерфейсу для користувача, особливо на мобільних пристроях або при повільному інтернет-з'єднанні. На рисунку 5.17 видно, що в першому випадку після відкриття модального вікна підвантажуються частина фотографій та JavaScript-код компоненти. У другому випадку, відразу після відкриття модального вікна, завантажуються усі фотографії, та не завантажується JavaScript-код, оскільки він був завантажений вже до цього при першому відкритті сторінки.

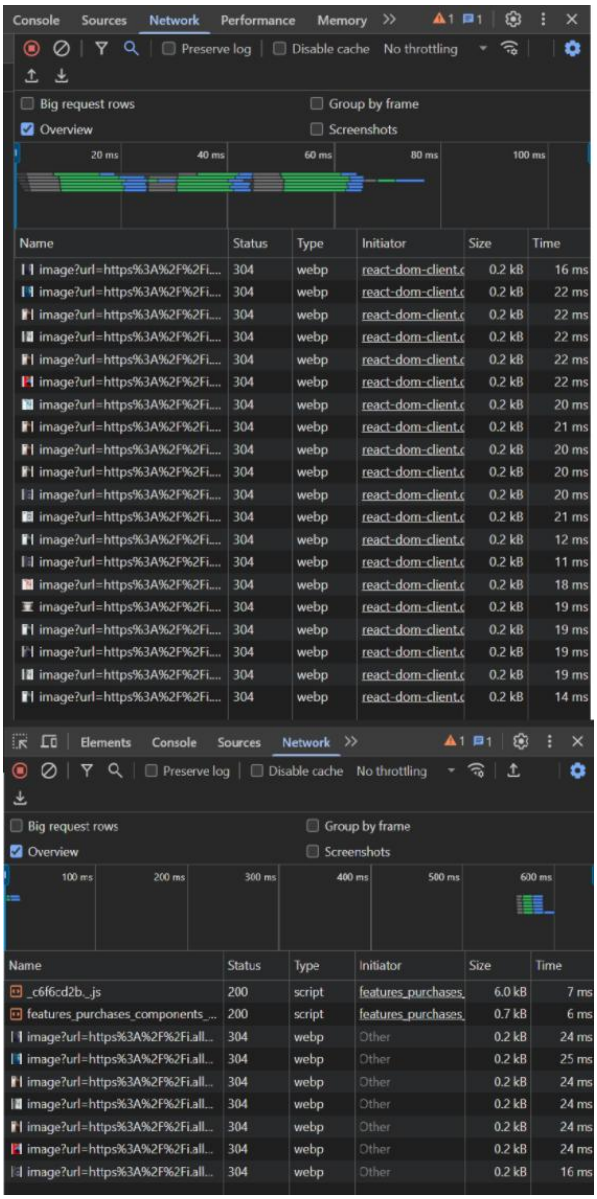


Рисунок 5.17 – Демонстрація роботи lazy loading (рисунок виконаний самостійно)

Проведене тестування чітко демонструє важливість та ефективність оптимізації продуктивності веб-застосунку. Під час завантаження великої кількості даних (225 записів користувачів) оптимізована версія забезпечує швидку реакцію, комфортне використання та стабільність як на десктопних, так і на мобільних пристроях. Неоптимізована версія суттєво програє за всіма показниками продуктивності, особливо на мобільних пристроях. Тому проведення оптимізаційних заходів є критично важливим етапом розробки веб-застосунків для забезпечення зручності користування та зменшення негативного впливу на кінцевих користувачів.

ВИСОВКИ

У результаті виконання кваліфікаційної роботи та отриманих результатів з дослідження, можна зробити висновок, що використання сучасних технологій на базі React, Next.js та пов’язаних інструментів, а також впровадження оптимізаційних методів, має принципове значення для забезпечення високої продуктивності, масштабованості та зручного користувацького досвіду сучасних веб-застосунків. Ключовими елементами успішної архітектури є чітке структурування проєкту, розумне планування його компонентної архітектури, впровадження інструментів для ефективного управління станом та використання засобів для зниження часу завантаження сторінок та перерендерингу інтерфейсів.

Важливою складовою у результаті аналізу стало розуміння ролі архітектури застосунку. React, як основна бібліотека для створення інтерфейсів, надає гнучкість у виборі структурних підходів, проте саме свідоме планування і розподіл обов’язків між компонентами, сторінками, модулями та сервісами визначає ступінь ефективності та масштабованості рішення. Застосування компонентно-орієнтованого підходу та дотримання принципу односпрямованого потоку даних дозволяють зменшити ймовірність виникнення труднощів із підтримкою коду. Чітка структура директорій та чітке визначення відповідальностей кожного із цих структурних елементів забезпечують прозорий та надійний підхід до розвитку застосунку. Це у перспективі дасть змогу легше впроваджувати нові функціональні можливості, розширювати команду розробників, а також знижувати ризик появи надмірно складних залежностей між частинами коду.

Крім того, отримані висновки вказують на те, що для ефективної розробки сучасних веб-застосунків на базі React та Next.js важливо звертати увагу на оптимізацію продуктивності. Використання різних підходів до завантаження сторінок (SSR, SSG, CSR), грамотна організація коду за допомогою чіткої архітектури й компонентно-орієнтованого підходу, а також застосування технік мемоізації, віртуалізації списків, лінивого завантаження та оптимізації мережевих запитів дозволяють значно покращити час завантаження, чуйність інтерфейсу та стабільність роботи. Це, у свою чергу, позитивно впливає на користувацький

досвід, роблячи взаємодію зі сторінками швидшою, простішою та більш передбачуваною.

Результати проведеного теоретичного аналізу підтвердили, що сучасні технологічні стеки та архітектурні підходи здатні суттєво вплинути на якість кінцевого продукту. Правильний вибір структурних рішень, продумане використання оптимізаційних технік та інструментів, регулярний моніторинг метрик продуктивності – усе це формує комплексний підхід, який дозволяє забезпечувати більш стабільну, швидку та якісну роботу веб-застосунків.

Застосування отриманих висновків на практиці, що підтвердили отримані результати після проведення практичного експериментального дослідження, дозволяє суттєво скоротити час завантаження сторінок, покращити швидкість реакції інтерфейсу, знизити використання ресурсів, забезпечити високу масштабованість та позитивний користувацький досвід.

Таким чином, можна стверджувати, що обраний шлях комплексного використання кращих практик архітектури, управління станом та оптимізацій на базі React та Next.js є виправданим і перспективним. Такий підхід забезпечує гнучкість, ефективність та адаптивність, дозволяючи веб-застосункам відповідати усім сучасним вимогам та задовольняти потреби як розробників, так і кінцевих користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. React (software) / Wikipedia. URL: https://en.wikipedia.org/wiki/React_(software) (дата звернення: 03.10.2024).

2. Advantages of Next.js for SEO Optimization Over React / Embrox Solution. URL: https://embrox.com/blog/Nextjs-react (дата звернення: 10.10.2024).

3. React Performance Optimization: Best Techniques and Ways to Implementation / Maybe Works. URL: https://maybe.works/blogs/react-performance-optimization-techniques (дата звернення: 15.10.2024).

4. Comparison of Redux and React Hooks Methods in Terms of Performance / CEUR Workshop Proceedings, Pronina D., Kyrychenko I., 2022, 3171, стр. 791–8000.

5. Learning React / React Official Documentation. URL: https://react.dev/learn (дата звернення: 22.10.2024).

6. React.lazy: APIS lazy / React Official Documentation. URL: https://react.dev/reference/react/lazy (дата звернення: 02.11.2024).

7. Getting Started with Redux / Redux Official Documentation. URL: https://redux.js.org/introduction/getting-started (дата звернення: 09.11.2024).

8. React Query Overview / TanStack Query Documentation. URL: https://tanstack.com/query/latest/docs/framework/react/overview (дата звернення: 15.11.2024).

9. Next.js: The React Framework for the Web / Next.js Official Website. URL: https://nextjs.org/ (дата звернення: 24.11.2024).

10. Основні метрики швидкості завантаження сайту: TTFB, FCP, LCP та інші / CityHost Blog. URL: https://cityhost.ua/uk/blog/osnovnye-metriki-skorosti-zagruzki-sayta-ttfb-fcp-lcp-i-drugie.html (дата звернення: 01.12.2024).

11. Web Performance Tools and Insights / web.dev. URL: https://web.dev/ (дата звернення: 03.12.2024).

12. Performance pattern: List Virtualization / Patterns.dev. URL: https://www.patterns.dev/vanilla/virtual-lists/ (дата звернення: 06.12.2024).

13. Analysis of Models Usability Methods Used on Design Stage to Increase Site Optimization / CEUR Workshop Proceedings, Gruzdo I., Kyrychenko I., Tereshchenko

G., Shanidze O., 2023, 3403, стр. 387–409.

14. Оптимізація продуктивності в React-застосунку / dou.ua. URL: https://dou.ua/forums/topic/45406/ (дата звернення: 10.12.2024).

15. O. I. Bezverhiy and O. I. Kutsenko, «Шляхи оптимізації кросплатформенних додатків із використанням бібліотеки React та фреймворку React Native», стр. 30 – 35, 2024, doi: 10.3272/2521-6643-2024-1-67.5.

16. Репозиторій GitHub [Електронний ресурс] – 2025. – URL: https://github.com/Gelan007/diploma.git.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

4. Comparison of Redux and React Hooks Methods in Terms of Performance / CEUR Workshop Proceedings, Pronina D., Kyrychenko I., 2022, 3171, стр. 791–8000.

13. Analysis of Models Usability Methods Used on Design Stage to Increase Site Optimization / CEUR Workshop Proceedings, Gruzdo I., Kyrychenko I., Tereshchenko G., Shanidze O., 2023, 3403, стр. 387–409.