ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
(повне найменування вищого навчального закладу)
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
(повне найменування інституту, назва факультету (відділення))
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до кваліфікаційної роботи
магістра
(рівень вищої освіти)

на тему: «Дослідження ефективності рендерингу динамічних веб-сторінок на прикладі React»

Виконав: студент 6 курсу, групи 6ПР
спеціальності
121 «Інженерія програмного забезпечення» »
(шифр і назва напряму підготовки, спеціальності)

Чередарецький Володимир Костянтинович
(прізвище та ініціали)

Керівник к.т.н., Вишемирська Світлана Вікторівна
(прізвище та ініціали)

Рецензент к.т.н., доц. Корніловська Н.В.
(прізвище та ініціали)

Хмельницький – 2025р.

Херсонський національний технічний університет
(повне найменування вищого навчального закладу)

Інститут, факультет, відділення Факультет інформаційних технологій та дизайну
Кафедра, циклова комісія Кафедра програмних засобів і технологій
Освітньо-кваліфікаційний рівень другий (магістерський)
Напрям підготовки ОПП – Програмне забезпечення систем
(шифр і назва)
Спеціальність 121 – Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗіТ
к.т.н., доц. Огнєва О.Є.
“____” _____ 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Чередарецький Володимир Костянтинович
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження ефективності рендерингу динамічних веб-сторінок на прикладі React
керівник роботи к.т.н. доцент Вишемирська С.В.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджена наказом вищого навчального закладу “15” вересня 2025 року № 417-с
2. Строк подання студентом роботи 01.09.2025
3. Вихідні дані до роботи _____
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1. Теоретичні основи рендерингу веб-інтерфейсів та особливості роботи React; 2. Дослідження ефективності рендерингу динамічних веб-сторінок у React; 3. Проєктування, реалізація та оцінювання продуктивності тестового React-додатку; 4. Оптимізація React-додатку та удосконалення ефективності рендерингу.
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
Комп’ютерна презентація

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Відбір та вивчення літературних джерел	01.09.2025 – 27.09.2025	
2	Аналіз стану вирішення завдання на сучасному етапі	27.09.2025 – 11.10.2025	
3	Побудова концептуальної моделі	11.10.2025 – 23.10.2025	Концептуальна схема
4	Розробка моделі	23.10.2025 – 31.10.2025	Функціональна схема
5	Побудова алгоритму функціонування додатку	31.10.2025 – 09.11.2025	Схема варіантів використання
6	Написання вихідного коду програми	09.11.2025 – 16.11.2025	
7	Налагодження програмного коду	16.11.2025– 22.11.2025	
8	Оформлення пояснювальної записки	23.11.2025 – 06.12.2025	

Студент _____ **(В.К.Чередарецький)**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **(С.В. Вишемирська)**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота магістра містить такі структурні частини: вступ, чотири розділи, висновки, список використаних джерел та додатки.

Перший розділ «Теоретичні основи рендерингу веб-інтерфейсів та особливості роботи React» складається з чотирьох частин: «Поняття та підходи до рендерингу веб-сторінок», «Класичні моделі клієнтського та серверного рендерингу», «Механізми роботи React та його вплив на продуктивність» та «Огляд сучасних технологій рендерингу: CSR, SSR, SSG та ISR».

У даному розділі проведено аналіз предметної області, розглянуто принципи побудови інтерфейсів у веб-додатках та наведено характеристики основних моделей рендерингу, що використовуються у сучасних програмних рішеннях.

Другий розділ «Дослідження ефективності рендерингу динамічних веб-сторінок у React» містить такі підрозділи: «Метрики продуктивності Web Vitals», «Методи оцінки часу рендерингу та реакції інтерфейсу», «Фактори, що впливають на швидкість роботи React-додатків» та «Порівняльний аналіз різних підходів до рендерингу».

У цьому розділі розглянуто інструменти вимірювання продуктивності, наведено методику експериментальних досліджень та проведено аналітичне порівняння ефективності різних моделей рендерингу на прикладі React.

Третій розділ «Проєктування, реалізація та оцінювання продуктивності тестового React-додатку» складається з таких підрозділів: «Постановка задачі та обґрунтування вибору технологій», «Створення тестового застосунку та опис його архітектури», «Реалізація варіантів рендерингу: CSR, SSR та SSG» та «Експериментальне дослідження продуктивності та аналіз отриманих результатів».

У даному розділі описано архітектуру створеного застосунку, представлено реалізацію різних підходів до рендерингу, виконано

вимірювання продуктивності та надано порівняльний аналіз отриманих результатів.

Четвертий розділ «Оптимізація React-додатку та удосконалення ефективності рендерингу» складається з п’яти підрозділів: «Методи оптимізації рендерингу в React», «Мемоізація та зменшення кількості повторних рендерів», «Оптимізація завантаження модулів та даних», «Оцінка ефективності застосованих оптимізацій» та «Перспективи подальшого вдосконалення».

У цьому розділі наведено техніки оптимізації React-додатків, реалізовано комплекс заходів зі зниження затримок рендерингу, виконано повторні вимірювання та проаналізовано вплив оптимізацій на загальну продуктивність застосунку.

ANNOTATION

The master's qualification thesis consists of the following structural components: an introduction, four chapters, a conclusion, a list of references, and appendices.

The first chapter, “Theoretical Foundations of Web Interface Rendering and the Features of React”, includes four sections: “Concepts and Approaches to Web Page Rendering,” “Classical Models of Client-Side and Server-Side Rendering,” “Internal Mechanisms of React and Their Impact on Performance,” and “Overview of Modern Rendering Technologies: CSR, SSR, SSG, and ISR.”

This chapter provides an analysis of the subject area, examines the principles of building user interfaces in web applications, and describes the main rendering models used in modern software systems.

The second chapter, “Research on the Rendering Efficiency of Dynamic Web Pages in React”, contains the following subsections: “Web Vitals Performance Metrics,” “Methods for Evaluating Rendering Time and Interface Responsiveness,” “Factors Influencing the Performance of React Applications,” and “Comparative Analysis of Different Rendering Approaches.”

This chapter reviews performance measurement tools, presents the methodology of the experimental study, and provides an analytical comparison of the efficiency of various rendering models based on React.

The third chapter, “Design, Implementation, and Performance Evaluation of a Test React Application”, consists of the following subsections: “Problem Statement and Justification of Technology Choice,” “Development of the Test Application and Description of Its Architecture,” “Implementation of Rendering

Variants: CSR, SSR, and SSG," and "Experimental Performance Evaluation and Analysis of Results."

This chapter presents the architecture of the developed application, describes the implementation of different rendering approaches, conducts performance measurements, and provides a comparative analysis of the obtained results.

The fourth chapter, "Optimization of the React Application and Enhancement of Rendering Efficiency", includes five subsections: "Rendering Optimization Methods in React," "Memoization and Reduction of Unnecessary Re-renders," "Optimization of Module and Data Loading," "Evaluation of the Effectiveness of Applied Optimizations," and "Prospects for Further Improvements."

This chapter outlines various optimization techniques for React applications, implements a set of measures to reduce rendering delays, conducts repeated performance measurements, and analyzes the impact of the optimizations on the overall efficiency of the application.

РЕФЕРАТ

Кваліфікаційна робота магістра: 109 сторінок, 19 малюнків, 13 таблиць, 35 джерел.

Мета роботи - дослідження ефективності рендерингу динамічних веб-сторінок та аналіз сучасних підходів до відображення інтерфейсів у React, а також визначення впливу різних моделей рендерингу на продуктивність веб-додатків. Для досягнення мети необхідно проаналізувати теоретичні основи роботи React, особливості моделей CSR, SSR, SSG та ISR, а також провести експериментальне порівняння ефективності рендерингу у тестовому застосунку.

Об’єкт дослідження - процеси рендерингу динамічних веб-інтерфейсів у сучасних односторінкових застосунках.

Предмет дослідження - методи, моделі та технології рендерингу інтерфейсів у React, а також фактори, що впливають на їх продуктивність.

Результат роботи: реалізовано тестовий веб-застосунок на основі React для порівняння ефективності різних підходів до рендерингу. Проведено вимірювання ключових метрик Web Vitals (LCP, FID, CLS, INP), проаналізовано час початкового рендерингу, швидкість оновлення інтерфейсу та вплив оптимізацій, таких як мемоізація, code splitting та lazy loading. На основі отриманих даних сформовано порівняльний аналіз продуктивності CSR, SSR та SSG у контексті динамічних веб-сторінок.

Новизна роботи: вперше виконано комплексне дослідження ефективності рендерингу динамічних веб-інтерфейсів на прикладі React із залученням експериментальних даних, отриманих шляхом реалізації декількох моделей рендерингу в одному тестовому середовищі. Розроблено методику оцінювання продуктивності, що враховує не лише час рендерингу,

але й поведінку інтерфейсу під час динамічних змін стану. Запропоновано підхід до оптимізації React-додатків, який дозволяє суттєво зменшити навантаження на браузер та прискорити відображення контенту.

Розроблене програмне рішення може бути використане для подальших досліджень у сфері оптимізації SPA-додатків, а також у комерційних веб-проєктах, де важливими є швидкість завантаження та плавність роботи інтерфейсу.

Ключові слова: React, рендеринг, продуктивність, CSR, SSR, SSG, Web Vitals, оптимізація, SPA.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Скорочення, термін, позначення	Пояснення
API	Application Programming Interface
CSR	Client-Side Rendering - клієнтський рендеринг
SSR	Server-Side Rendering - серверний рендеринг
SSG	Static-Site Generation - статична генерація сторінок
ISR	Incremental Static Regeneration - інкрементальна генерація
SPA	Single Page Application - односторінковий веб-додаток
DOM	Document Object Model - модель документа
Virtual DOM	Віртуальна модель DOM у React
JSX	JavaScript XML - синтаксис опису UI у React
UI	User Interface - користувацький інтерфейс
UX	User Experience - досвід користувача
JS	JavaScript
Node.js	Платформа для виконання JavaScript на сервері
NPM	Node Package Manager - менеджер пакетів Node.js
Vite	Інструмент для збірки та запуску фронтенд-проєктів
Bundle	Збірка JavaScript-файлів
Web Vitals	Набір ключових метрик продуктивності веб-сторінок
LCP	Largest Contentful Paint - час рендерингу основного контенту
FID	First Input Delay - затримка першої взаємодії
INP	Interaction to Next Paint - загальна реактивність інтерфейсу
CLS	Cumulative Layout Shift - стабільність макета сторінки
CDN	Content Delivery Network - мережа доставки контенту
CPU	Central Processing Unit - центральний процесор
DevTools	Набір інструментів для розробника в браузері

Скорочення, термін, позначення	**Пояснення**
HTTP	HyperText Transfer Protocol
HTTPS	Захищений протокол передачі даних
JSON	JavaScript Object Notation
Cache	Механізм збереження даних для пришвидшеного доступу
Rendering	Процес відображення інтерфейсу
Re-render	Повторний рендеринг елементів
Memoization	Техніка кешування результатів функцій
Lazy loading	Динамічне завантаження модулів або компонентів
Code splitting	Розбиття коду на окремі частини для оптимізації
Next.js	React-фреймворк для SSR/SSG/ISR
Lighthouse	Інструмент для вимірювання продуктивності сторінок
Chrome Performance	Вкладка Performance в Chrome DevTools для аналізу рендерингу

ЗМІСТ

ВСТУП

На сьогоднішній день актуальність дослідження продуктивності веб-додатків стрімко зростає у зв’язку з активним розвитком цифрових платформ, збільшенням складності інтерфейсів та підвищенням вимог користувачів до швидкодії веб-ресурсів. Динамічні веб-сторінки стали ключовим елементом сучасних інформаційних систем, бізнес-платформ, соціальних мереж, онлайн-сервісів та електронної комерції. Однак, зі зростанням обсягів даних та функціональних можливостей збільшується навантаження на браузер і сервер, що зумовлює необхідність застосування ефективних методів рендерингу.

Важливу роль у розробці високопродуктивних веб-застосунків відіграють інструменти та фреймворки, що забезпечують оптимізацію процесу відображення інтерфейсу. Одним із найпопулярніших рішень є бібліотека React, яка завдяки концепціям Virtual DOM, компонентному підходу та гнучким моделям рендерингу стала стандартом де-факто у сфері фронтенд-розробки. Проте вибір конкретної моделі рендерингу - CSR, SSR, SSG чи ISR - суттєво впливає на швидкість завантаження сторінки, реактивність інтерфейсу та загальний користувацький досвід.

Актуальність теми. Актуальність дослідження обумовлена необхідністю побудови веб-додатків, які забезпечують високу продуктивність та швидке відображення динамічного контенту незалежно від складності логіки та кількості користувачів. В умовах конкуренції та зростання вимог до зручності користування критично важливим стає правильний вибір підходу до рендерингу та оптимізація часу відображення інтерфейсу. Дослідження ефективності рендерингу у React дозволяє визначити оптимальні підходи для побудови масштабованих, швидких і стабільних веб-систем.

Чинники актуальності дослідження:

- Зростанням складності сучасних веб-застосунків;
- Необхідністю зменшення часу завантаження сторінок та підвищення продуктивності;
- Популярністю React як провідного інструмента для розробки інтерфейсів;
- Потребою оцінки ефективності різних моделей рендерингу (CSR, SSR, SSG, ISR);
- Необхідністю наукового аналізу впливу оптимізації на швидкодію інтерфейсу;
- Обмеженою кількістю комплексних досліджень порівняльної продуктивності React-рішень.

Мета і задачі дослідження. Метою дослідження є вивчення, порівняння та оцінка ефективності різних підходів до рендерингу динамічних веб-сторінок на прикладі бібліотеки React, а також визначення впливу оптимізацій на продуктивність веб-додатків.

Для досягнення поставленої мети необхідно розв’язати наступні задачі:

- Вивчити теоретичні основи рендерингу веб-інтерфейсів;
- Дослідити моделі CSR, SSR, SSG та ISR, їх переваги та недоліки;
- Проаналізувати механізми роботи React, Virtual DOM та процеси повторного рендерингу;
- Розглянути ключові метрики Web Vitals та інструменти оцінювання продуктивності;
- Розробити тестовий веб-застосунок з різними моделями рендерингу;
- Провести експериментальне вимірювання продуктивності;
- Здійснити аналіз впливу оптимізацій (мемоізація, lazy loading, code splitting);

- Сформувати порівняльні висновки щодо ефективності моделей рендерингу.

Завдання, які було виконано під час дослідження. Під час виконання кваліфікаційної роботи було виконано такі завдання:

- Проаналізовано сучасні підходи до рендерингу веб-інтерфейсів;
- Досліджено архітектурні особливості React та механізми Virtual DOM;
- Розглянуто інструменти та метрики оцінки продуктивності веб-додатків;
- Розроблено тестовий React-застосунок для експериментів;
- Реалізовано рендеринг у моделях CSR, SSR та SSG;
- Проведено вимірювання ефективності за ключовими метриками;
- Виконано оптимізацію рендерингу та повторний аналіз швидкодії.

Об’єкт дослідження - процеси рендерингу динамічних веб-інтерфейсів у багатокомпонентних веб-застосунках.
Предмет дослідження - методи, моделі та технології рендерингу інтерфейсів у React та їх вплив на продуктивність веб-додатків.

Новизна роботи. У результаті виконаного дослідження одержані наступні наукові результати:

- Проведено комплексне порівняння моделей рендерингу (CSR, SSR, SSG) у React у єдиному тестовому середовищі;
- Розроблено підхід до експериментального вимірювання Web Vitals для динамічного контенту;
- Виявлено фактори, що найбільше впливають на час рендерингу та реактивність інтерфейсу;
- Запропоновано набір практичних рекомендацій з оптимізації React-додатків;

- Продемонстровано вплив мемоізації, оптимізації стану та розбиття коду на продуктивність.

Практичне значення одержимих результатів. Отримані результати можуть бути використані при розробці високопродуктивних веб-застосунків, оптимізації інтерфейсів, переході з CSR на SSR/SSG, а також у комерційних проєктах, де критично важливі швидкість завантаження та стабільність роботи інтерфейсу.

Теоретичне значення одержимих результатів. Матеріали дослідження можуть бути використані для подальшого розвитку підходів до побудови ефективних SPA-додатків, а також у навчальному процесі при вивченні сучасних технологій веб-розробки, оптимізації продуктивності та архітектури фронтенд-систем.

Апробація результатів роботи. Результати дослідження можуть бути використані у подальшій розробці веб-систем, що потребують високої швидкодії та оптимального рендерингу, а також у наукових дослідженнях, присвячених модернізації SPA-архітектури.

Структура: робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел, додатків. Загальний обсяг роботи - 109 сторінок.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РЕНДЕРИНГУ ВЕБ-ІНТЕРФЕЙСІВ

1.1 Поняття рендерингу у веб-додатках та його роль у формуванні інтерфейсу

Рендеринг веб-сторінок є одним з ключових процесів у побудові сучасних інформаційних систем, що працюють у браузерному середовищі. Під рендерингом зазвичай розуміють процес перетворення вихідного коду - HTML, CSS та JavaScript - у візуальне представлення, яке користувач бачить на екрані [19, 35]. Цей процес охоплює кілька етапів: побудову DOM-дерева, створення CSSOM, розрахунок стилів, формування layout (розмітки), рендеринг графічних елементів та оновлення інтерфейсу при взаємодії з користувачем або зміною стану застосунку [19, 35].

Важливість рендерингу пояснюється тим, що саме від швидкості та ефективності цього процесу залежить читабельність, зручність, доступність та інтерактивність веб-додатка. У сучасних умовах користувачі очікують миттєвого реагування інтерфейсу, плавної анімації, стабільного відображення компонентів, а також швидкого завантаження динамічного контенту [11]. Саме тому правильна організація рендеринг-процесу є критичним завданням для веб-розробників.

Рендеринг як елемент взаємодії між людиною та комп’ютером. Із зростанням складності веб-інтерфейсів рендеринг перетворився з простого процесу розміщення HTML-елементів у складний механізм, що включає логіку обчислень, оптимізацію повторних оновлень та адаптацію до різних пристроїв [15]. Рендеринг можна вважати фундаментальним компонентом взаємодії людина-комп’ютер (HCI), адже саме він забезпечує візуалізацію інформації, необхідну для взаємодії користувача з системою [11, 12].

Від ефективності рендерингу залежить не лише швидкість завантаження сторінки, а й загальний досвід користувача - UX (User Experience). Затримки в рендерингу можуть спричинити неприємне враження, зменшення конверсії в e-commerce, збільшення відмов у веб-сервісах та негативну оцінку інтерфейсу [11, 16]. Саме тому багато компаній приділяють особливу увагу Web Vitals - набору метрик Google, що визначають якість рендерингу [12].

Етапи рендерингу веб-сторінки. Процес рендерингу у браузері умовно поділяється на такі етапи [19, 35]:

1. Обробка HTML-коду та побудова DOM-дерева. Браузер читає HTML-файл пострядково, створюючи структуру елементів - Document Object Model [33].
2. Обробка CSS та побудова CSSOM. Стилі з файлів або <style> блоків перетворюються у окреме дерево стилів [14].
3. Об’єднання DOM і CSSOM у Render Tree. На цьому етапі браузер формує дерево, що містить всі видимі елементи сторінки [19, 35].
4. Layout (перерахунок розмірів і позицій). Браузер визначає точне розміщення елементів на екрані [6].
5. Painting - відрисовка. Елементи переводяться у пікселі та відображаються на екрані [12].
6. Композиціювання. Оптимізація графічних шарів для плавного оновлення інтерфейсу, особливо при анімаціях [12].

Цей процес може повторюватися частково або повністю щоразу, коли JavaScript змінює DOM, коли отримуються нові дані або коли користувач взаємодіє з інтерфейсом [9, 16].

Проблеми традиційного рендерингу. У класичній моделі роботи браузера DOM є структурою, яка постійно змінюється, що ускладнює

ефективність оновлень [5, 7]. Навіть незначна зміна одного елемента може викликати повторний розрахунок стилів та верстки для значної частини сторінки [6]. Це призводить до:

- затримок відображення;
- зниження FPS при взаємодії;
- "ривків" інтерфейсу;
- збільшення навантаження на CPU;
- погіршення UX.

Ці обмеження були однією з причин появи нових технологій, що оптимізують процес рендерингу - таких як React з Virtual DOM [21, 27].

Еволюція підходів до рендерингу у веб-додатках. Рендеринг веб-сторінок пройшов довгий шлях розвитку - від статичних HTML-документів до складних інтерактивних інтерфейсів, які працюють у режимі реального часу [13, 19, 35]. Розуміння історичних етапів еволюції дозволяє краще усвідомити причини появи сучасних фреймворків і технологій, зокрема React [1], а також оцінити важливість оптимізації рендеринг-процесу.

Статичні веб-сторінки та перший етап розвитку вебу. На початковому етапі розвитку Інтернету веб-сторінки були статичними документами, створеними у вигляді простого HTML-коду [15]. Такий формат підходив для відображення текстової інформації, проте повністю виключав можливість динамічної взаємодії [20]:

- HTML формувався на сервері та передавався у незмінному вигляді [20];
- зміна контенту вимагала повного перезавантаження сторінки [20];
- рендеринг був послідовним та передбачуваним [15];

- браузери виконували мінімальні операції з обробки контенту [15].

На цьому етапі проблеми продуктивності рендерингу не були критичними, оскільки обсяг HTML був відносно невеликим, а логіка сторінок - максимально простою [19, 35].

Поява JavaScript і динамічного рендерингу. Ситуація кардинально змінилася з появою JavaScript, який дозволив розробникам змінювати структуру сторінки без її повного перезавантаження. JavaScript відкрив можливості для [9, 15, 19]:

- динамічного оновлення DOM;
- створення інтерактивних елементів;
- змін контенту у відповідь на дії користувача;
- розбудови веб-додатків, схожих на десктопні програми.

Разом із тим динамічний DOM став причиною появи проблем [6, 9, 12, 15, 16, 31]:

- часті зміни DOM викликали значні накладні витрати;
- повний перерахунок стилів і макета призводив до затримок;
- складні операції іноді блокували головний потік браузера;
- продуктивність залежала від обсягу даних та кількості динамічних елементів.

Це стало фундаментальною передумовою пошуку оптимізованих моделей рендерингу.

Ajax і односторінкові застосунки (SPA). У 2005-2010 роках поширився формат SPA - Single Page Application, який дозволяв створювати веб-додатки, що працюють без перезавантаження сторінок. Ajax забезпечив [20]:

- асинхронне отримання даних з сервера;
- оновлення тільки потрібних частин інтерфейсу;
- підвищену інтерактивність.

Водночас SPA почали страждати від [10, 16, 17, 20, 31]:

- зростання складності архітектури;
- великого обсягу JavaScript-коду, що завантажувався на клієнт;
- проблем з продуктивністю при збільшенні компонентності;
- труднощів із SEO, що не підтримувало клієнтську генерацію HTML.

Саме складність SPA, разом із проблемами DOM, стала поштовхом до створення Virtual DOM та появи фреймворку React у 2013 році.

Роль JavaScript у процесі рендерингу та взаємодії з DOM. JavaScript виконує ключову роль у сучасних веб-додатках, оскільки саме він визначає логіку поведінки інтерфейсу. Проте взаємодія JavaScript з DOM не є оптимальною [6, 7, 9, 12, 15, 25]:

- DOM є важкою для маніпулювання структурою;
- зміни DOM можуть спричиняти повний перерахунок layout;
- складні DOM-дерева суттєво уповільнюють рендеринг;
- надмірне використання JavaScript блокує головний потік.

Блокування головного потоку (main thread). Браузер виконує більшість операцій у одному потоці [9, 13, 15]:

- парсинг HTML;
- виконання JavaScript;
- рендеринг;
- обчислення стилів;

- обробку подій.

Отже, якщо JavaScript виконує важкі обчислення, браузер не може одночасно відображати інтерфейс, що призводить до:

- фризів (зависань інтерфейсу),
- пропуску кадрів,
- затримок на скролінг,
- низького FPS,
- зменшення плавності анімацій [9, 13, 16].

Саме ці обмеження стимулювали появу фреймворків, які мінімізують взаємодію із DOM і оптимізують рендеринг [1, 10].

Традиційний рендеринг vs сучасні підходи. Сучасні підходи до рендерингу відрізняються від традиційних тим, що вони [1, 5, 10, 16, 18, 19, 20, 25]:

- зменшують кількість взаємодій з DOM;
- використовують ефективні стратегії оновлення;
- дають можливість рендерити частини інтерфейсу на сервері;
- забезпечують повторне використання компонентів;
- підтримують прогресивний рендеринг (частинами);
- оптимізують завантаження даних та ресурсів.

До таких підходів належать:

- Client-Side Rendering (CSR),
- Server-Side Rendering (SSR),
- Static-Site Generation (SSG),
- Incremental Static Regeneration (ISR),
- React Server Components [17, 20].

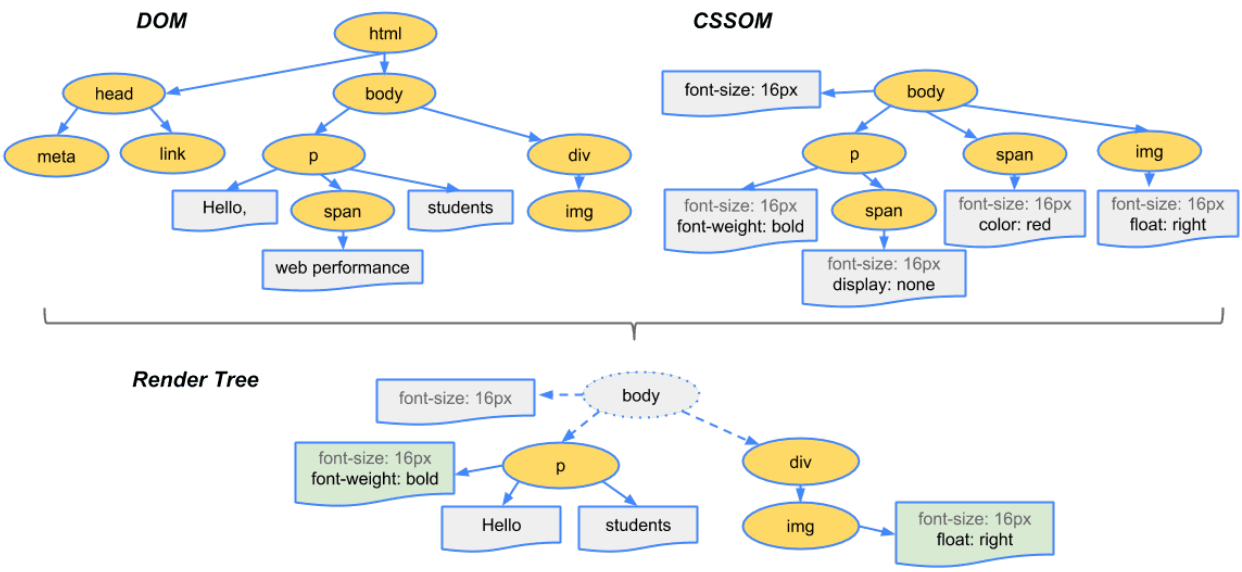
Кожен з них вирішує конкретні проблеми рендерингу, але має і свої недоліки [20].

Механізми роботи DOM та вплив змін на продуктивність рендерингу. Одним із ключових факторів, що визначає ефективність рендерингу у сучасних веб-додатках, є природа Document Object Model (DOM) [33]. DOM виступає абстрактним представленням структури HTML-документа у вигляді деревоподібної моделі, де кожен елемент є окремим вузлом [33]. Така модель забезпечує гнучку взаємодію між JavaScript та інтерфейсом, але водночас створює відчутні накладні витрати при зміні стану сторінки [15].

Складність побудови та оновлення DOM. DOM є об’єктною системою, що містить численні властивості, методи, посилання на дочірні та батьківські елементи [33]. Робота з DOM включає [7, 14, 15]:

- парсинг та аналіз HTML-коду;
- створення внутрішніх об’єктів для кожного тегу;
- побудову деревоподібної структури;
- розміщення елементів згідно зі стилями;
- оновлення структури у випадку змін.

Через свою складність DOM вважається “важкою” структурою з точки зору продуктивності [15]. Кожна зміна - навіть додавання одного елементу - може спричинити часткове чи повне оновлення дерева [6, 16].



Малюнок 1.1. - *Побудова DOM та CSSOM [14, 16]*

Reflow та Repaint: ключові процеси впливу на продуктивність. Будь-яка зміна DOM супроводжується двома основними процесами: reflow та repaint [6, 25].

Reflow (перерахунок макета). Reflow - це процес перерахунку розмірів, позицій та геометрії елементів після зміни:

- контенту;
- стилів;
- структури DOM;
- розміру вікна браузера;
- шрифтів;
- кількості елементів [6].

Reflow істотно навантажує процесор, оскільки браузер повинен:

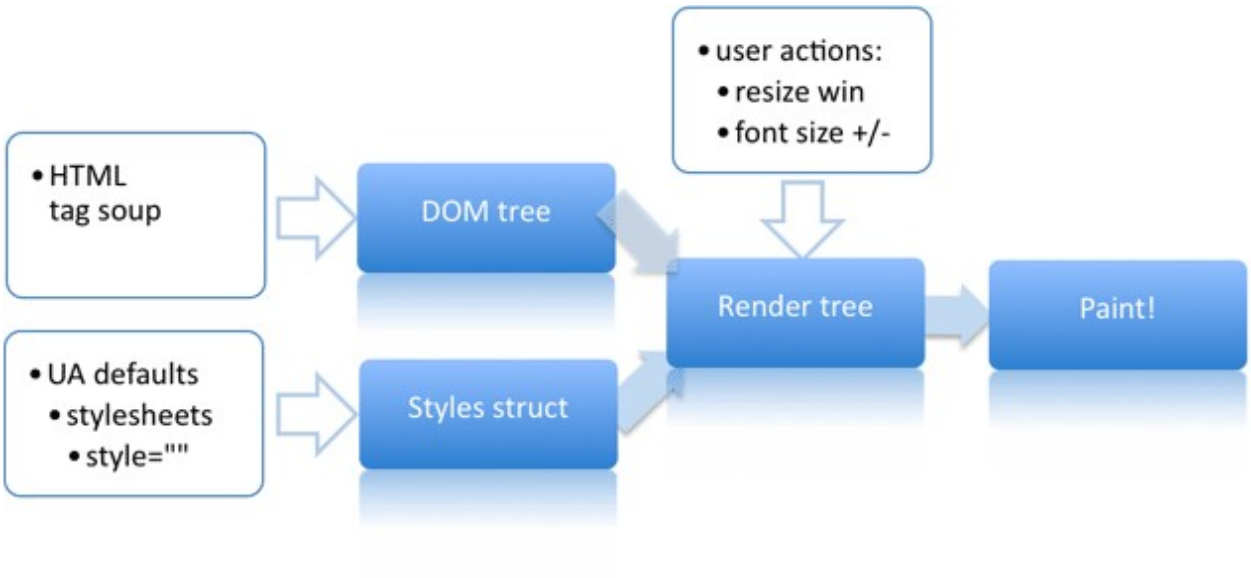
1. Повторно побудувати частину або всю структуру render tree.
2. Перерозподілити елементи.

3. Перерахувати геометрію блоків [6, 16].

У складних інтерфейсах (панелі, таблиці, списки з динамічними елементами) reflow може виконуватися десятки разів на секунду, що призводить до фризів [6, 13].

Repaint (перемальовування). Repaint - це оновлення зовнішнього вигляду елементів (колір, фон, тінь), яке не зачіпає їх геометрію.

Repaint менш затратний, проте у поєднанні з reflow він може значно вплинути на FPS [19, 35].



Малюнок 1.2. - *Reflow та Repaint [6]*

Каскадність змін: маленька зміна - велика проблема. DOM має властивість "каскадності", тобто зміни одного елементу можуть впливати на сусідні вузли, батьківські елементи та навіть на все дерево [6, 16]. Наприклад:

- зміна висоти одного блоку може змінити позицію всіх наступних елементів;

- зміна властивості font-size може перерахувати layout для всього документа;
- вставка нового елемента у верхній частині списку може відтворити reflow для кожного рядка [6].

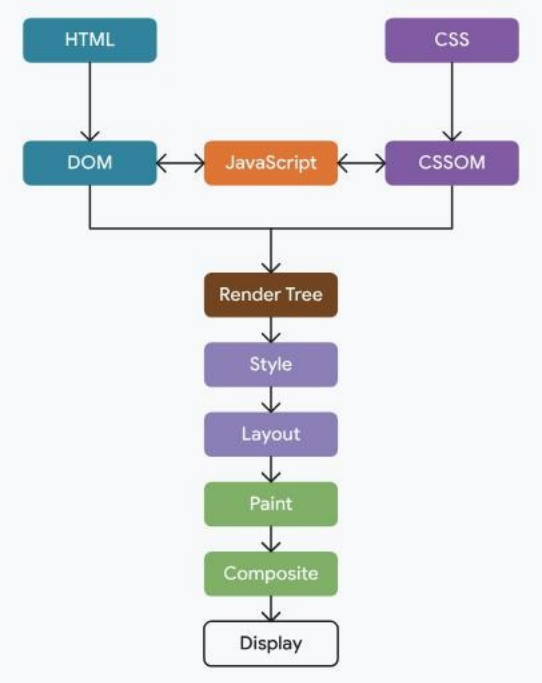
Це означає, що продуктивність традиційних DOM-операцій погіршується пропорційно зростанню складності інтерфейсу [15].

Критичний шлях рендерингу (Critical Rendering Path). Critical Rendering Path - це послідовність операцій, які браузер виконує для перетворення HTML, CSS і JavaScript у готову сторінку [14, 16]. Ефективність цього шляху безпосередньо визначає:

- швидкість першого відображення контенту;
- час до першої взаємодії (FID);
- загальну реактивність UI;
- плавність анімацій та переходів [11, 14].

Основні етапи критичного шляху:

1. Парсинг HTML → побудова DOM.
2. Парсинг CSS → побудова CSSOM.
3. Об’єднання DOM і CSSOM у Render Tree.
4. Layout - розрахунок позицій елементів.
5. Paint - відрисовка елементів.
6. Composite - об’єднання слоїв [14, 16].



Малюнок 1.3. - *Critical Rendering Path [14]*

Оптимізація Critical Rendering Path стала одним із головних напрямків розвитку сучасних JS-фреймворків, у тому числі React [1, 2, 16].

Рендеринг динамічного контенту: особливі складнощі. У сучасних веб-додатках статичні сторінки майже не використовуються [31]. Основна частина інформації завантажується:

- через API;
- асинхронні запити;
- WebSockets;
- динамічні оновлення стану;
- події користувача [9, 31].

Динамічний контент постійно змінює DOM, що збільшує навантаження на рендеринг-систему [15]:

- нові елементи додаються або видаляються;
- відбувається часте оновлення списків;
- змінюється структура компонентів;
- дані перерендерюються при кожній зміні стану або props.

У великих SPA-додатках такі операції можуть повторюватися тисячі разів на хвилину, що істотно впливає на продуктивність [19, 35, 34].

Необхідність появи оптимізованих моделей рендерингу. Зі зростанням складності взаємодії між DOM та JavaScript традиційні підходи перестали бути ефективними [15]. Виникли такі проблеми [6, 10, 11, 12, 15, 20, 23]:

- DOM-операції стали занадто повільними;
- великі SPA стали погано масштабуватися;
- продуктивність інтерфейсу стала залежати від складності дерева компонентів;
- браузери почали витрачати надмірні ресурси на обчислення стилів та layout;
- потреба у швидкому першому рендері зросла через SEO та вимоги мобільних пристроїв.

Ці фактори створили передумови для появи принципово нових технологій, таких як Virtual DOM, що став основою React [21].

Поступовий перехід до Virtual DOM та React. React запропонував революційний підхід - замість прямої взаємодії з DOM розробники працюють

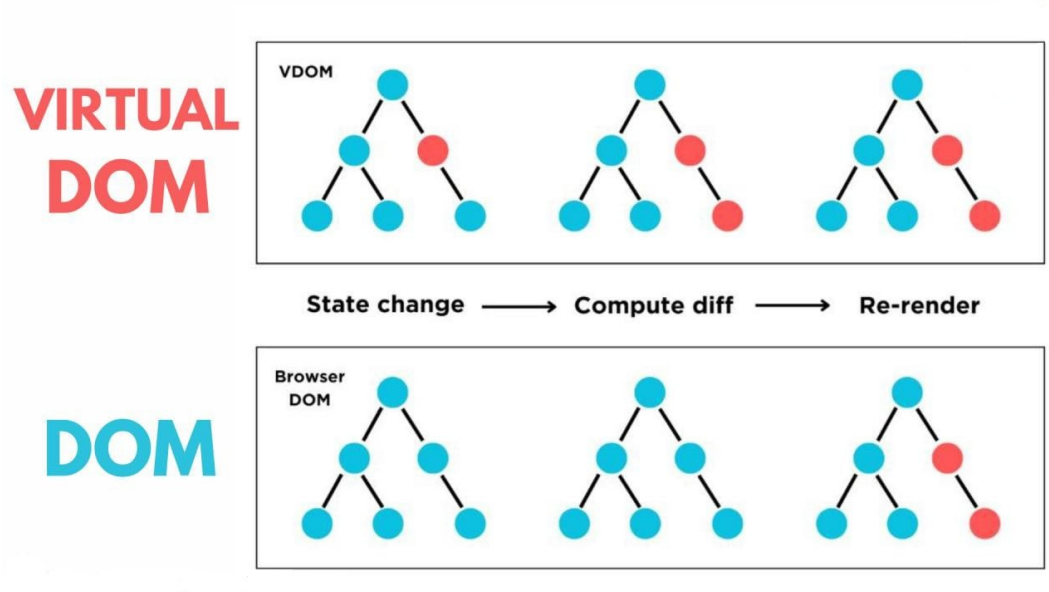
з абстрактним представленням інтерфейсу, яке оновлюється лише там, де це необхідно [1].

Virtual DOM [21, 2, 27, 15, 6, 20, 16, 30, 31]:

- зменшує кількість взаємодій із реальним DOM;
- оптимізує оновлення за допомогою алгоритму порівняння (diffing);
- дозволяє React мінімізувати reflow та repaint;
- робить рендеринг передбачуваним;
- суттєво знижує накладні витрати.

Це сприяло поширенню React та зробило його провідною бібліотекою для побудови веб-інтерфейсів.

Virtual DOM як сучасний підхід до оптимізації рендерингу. Поява Virtual DOM стала ключовою точкою розвитку сучасних фронтенд-фреймворків [5]. На відміну від реального DOM, який є важкою структурою, Virtual DOM - це легке, абстрактне подання UI у вигляді JavaScript-об’єктів. Замість прямого оновлення інтерфейсу React спочатку формує його віртуальну копію, а потім порівнює попередній та новий стани [21, 15].



Малюнок 1.4. - *Схема Virtual DOM [5]*

Алгоритм порівняння (Reconciliation). React використовує оптимізований diff-алгоритм, який дозволяє [21, 2, 15, 6, 20, 23]:

- визначати мінімальний набір змін;
- виконувати вибіркові оновлення компонентів;
- уникати зайвих reflow та repaint;
- оновлювати інтерфейс лише там, де зміни дійсно потрібні.

Цей механізм робить React ефективним навіть у великих динамічних застосунках.

Переваги Virtual DOM у контексті рендерингу [21, 2, 15, 20, 11, 23]

1. Мінімізація взаємодій з реальним DOM.
2. Вибіркове оновлення компонентів.
3. Покращення швидкості рендерингу динамічних списків і форм.
4. Стабільний час реакції інтерфейсу при частих змінах стану.
5. Передбачувана модель оновлень.

Ці особливості стали вирішальними у формуванні React як ефективної платформи для роботи з динамічними інтерфейсами, де рендеринг відіграє ключову роль.

Підсумок підрозділу 1.1. Таким чином, рендеринг веб-додатків є складним процесом, який включає послідовну взаємодію браузера з DOM, стилями та JavaScript. Традиційний підхід до оновлення інтерфейсу часто виявляється неефективним через велику кількість reflow та repaint, що негативно впливає на продуктивність. Еволюція рендерингу привела до появи Virtual DOM та оптимізованих механізмів оновлення, які дозволяють суттєво зменшити навантаження на браузер і покращити загальну швидкодію веб-додатків.

Переваги React у цій сфері є фундаментом для подальшого аналізу різних моделей рендерингу - CSR, SSR, SSG та ISR - які будуть розглянуті у наступному підрозділі.

1.2 Класичні моделі рендерингу веб-сторінок: CSR, SSR, SSG та ISR

Сучасні веб-технології пропонують різні моделі рендерингу інтерфейсу, які мають істотний вплив на швидкість завантаження сторінок, SEO, продуктивність застосунків та їх масштабованість [25, 16, 20]. У контексті React вибір оптимальної стратегії рендерингу є критичним, оскільки він визначає як початковий досвід користувача, так і подальшу взаємодію з інтерфейсом [1, 11]. У цьому розділі детально розглядаються чотири основні підходи: Client-Side Rendering (CSR), Server-Side Rendering (SSR), Static Site Generation (SSG) та Incremental Static Regeneration (ISR) [17, 18, 20].

Client-Side Rendering (CSR): рендеринг на стороні клієнта. Client-Side Rendering є базовою моделлю, яка використовується у більшості SPA-застосунків [20]. У процесі CSR браузер завантажує мінімальний HTML-каркас та великий JavaScript-бандл, у якому міститься логіка застосунку та React-компоненти [20, 31]. Після виконання JavaScript у браузері створюється Virtual DOM, а потім формується реальний DOM, що і забезпечує відображення інтерфейсу [21, 19].

CSR забезпечує високу інтерактивність і плавність роботи застосунку після початкового завантаження [20]. Однак він має суттєвий недолік - повільний перший рендер, оскільки сторінка не може бути показана до виконання JavaScript [25]. Це особливо помітно на мобільних пристроях із низькою продуктивністю. Додатковим мінусом CSR є слабка SEO-підтримка, адже більшість контенту формується у браузері, а не на сервері [17, 20].

CSR найчастіше використовується для складних панелей керування, інструментів аналітики, приватних кабінетів, чатів та загалом у застосунках, де головним пріоритетом є динамічність та інтерактивність [20].

Server-Side Rendering (SSR): рендеринг на сервері. Server-Side Rendering вирішує проблему повільного першого рендеру [20]. У цій моделі HTML-файл генерується на сервері за допомогою React, після чого користувач миттєво отримує вже відрендерений контент [17, 20]. Тільки після цього браузер завантажує JavaScript та «оживлює» сторінку в процесі гідратації [17, 20].

SSR забезпечує значно кращі показники Web Vitals, зокрема швидший First Contentful Paint (FCP) та Largest Contentful Paint (LCP). Також покращується SEO, адже пошукові системи отримують повноцінний HTML. Недоліком

SSR є високе навантаження на сервер, оскільки кожен запит вимагає обробки та рендерингу компонентів [17, 20].

Цей підхід використовується тоді, коли важливо забезпечити швидке початкове завантаження сторінки та ефективну індексацію: наприклад, у маркетингових сторінках, блогах, новинних порталах, інтернет-магазинах або контентних платформах [17, 20].

Static Site Generation (SSG): статична генерація сторінок. Static Site Generation дозволяє створювати HTML-сторінки заздалегідь під час збірки застосунку [24]. Замість того, щоб генерувати контент під час кожного запиту (як у SSR), SSG створює файли один раз, і далі вони розміщуються в CDN, що забезпечує надзвичайно швидку доставку контенту [20].

SSG пропонує найкращі показники продуктивності - сторінки завантажуються миттєво, а сервер при цьому майже не використовується [16, 20]. Проте модель підходить лише для відносно статичного контенту [20]. Якщо інформація змінюється часто (наприклад, новини чи ціни), SSG вимагає повної регенерації сайту, що може бути повільно або незручно.

Підходить для документацій, лендінгів, блогів, SEO-орієнтованих статей, корпоративних сайтів із рідко змінюваними даними [20].

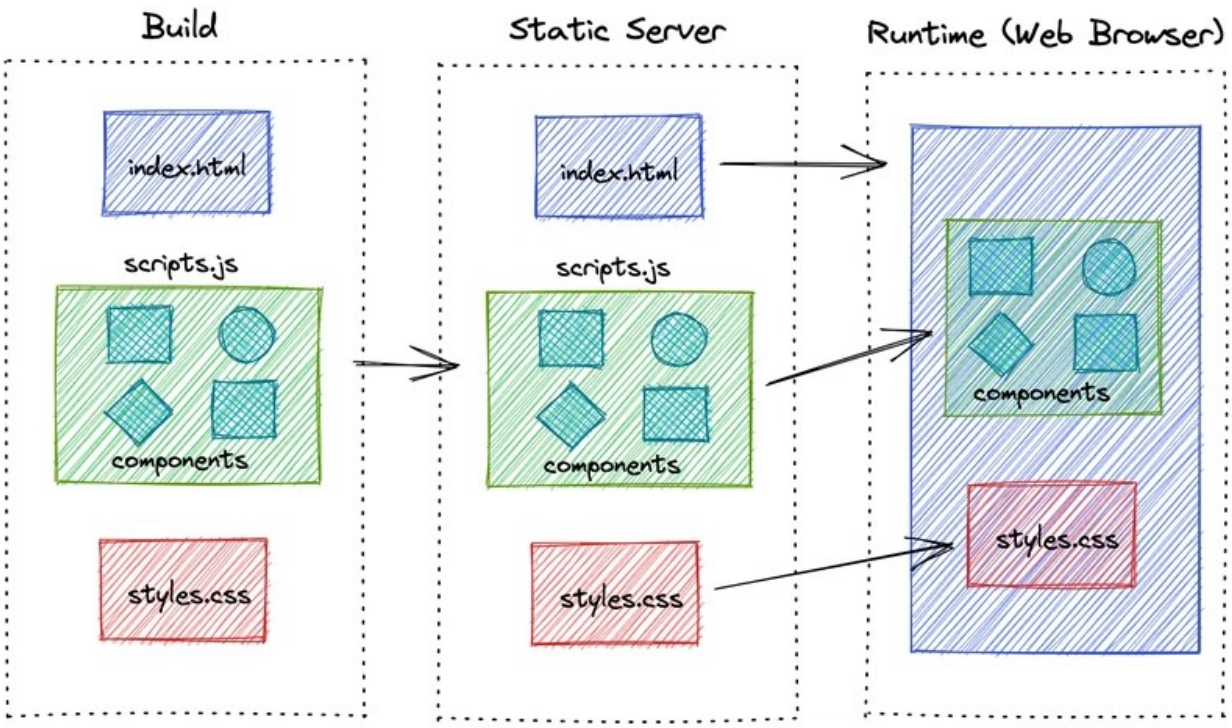
Incremental Static Regeneration (ISR): інкрементальне оновлення статичного контенту. Incremental Static Regeneration є гібридним підходом, який поєднує швидкість SSG і гнучкість динамічного контенту [23, 24, 32]. Сторінки генеруються статично, але можуть автоматично оновлюватися з певним інтервалом [18]. Це відбувається на сервері у фоновому режимі, а

користувачі залишаються на актуальних версіях без перезавантаження сайту [18, 19].

ISR є ідеальним рішенням для платформ з великою кількістю сторінок і частково динамічним контентом: інтернет-магазини, блоги, каталоги, новинні портали [18, 19]. Він забезпечує баланс між актуальністю контенту та максимальною швидкістю завантаження [19].

Таблиця 1.1. - *Порівняння моделей рендерингу веб-сторінок [7, 24, 23, 32]*

Модель	**Переваги**	**Недоліки**	**Типові сценарії**
CSR	Гнучкість, висока інтерактивність	Повільний перший рендер, слабкий SEO	SPA, дашборди, приватні кабінети
SSR	Швидкий початковий рендер, SEO	Високе навантаження на сервер	Новинні портали, каталоги, магазини
SSG	Максимальна швидкість, CDN	Не підходить для часто змінюваного контенту	Блоги, документації, лендінги
ISR	Комбінація швидкості та актуальності	Складніша конфігурація	Великі портали, e-commerce



Малюнок 1.5. - *Порівняльна схема моделей рендерингу CSR/SSR/SSG/ISR [17, 18, 19, 20]*

Порівняння моделей рендерингу. Різні моделі рендерингу мають свої сильні й слабкі сторони [20]. CSR забезпечує максимальну динамічність, але повільний перший рендер [25, 20]. SSR пропонує хорошу SEO-підтримку та швидке початкове відображення, але збільшує навантаження на сервер. SSG забезпечує найкращу продуктивність серед усіх моделей, проте не підходить для динамічного контенту. ISR дозволяє поєднати швидкість статичних сторінок із можливістю їх оновлення без повної регенерації сайту [18, 19].

Ефективність роботи React-застосунків суттєво залежить від того, яка модель буде обрана. Важливо враховувати специфіку продукту, частоту оновлення даних, вимоги до SEO, навантаження на сервер та очікувані показники Web Vitals [11, 16].

Підсумок підрозділу 1.2. Розуміння різних підходів до рендерингу є ключовим для побудови високопродуктивних веб-застосунків. CSR забезпечує хорошу інтерактивність, але програє у швидкості первинного завантаження. SSR покращує видимість контенту і SEO, проте збільшує навантаження на сервер. SSG є найшвидшим варіантом, якщо контент змінюється рідко. ISR виступає сучасним компромісним рішенням, що дає можливість створювати масштабовані, швидкі та актуальні веб-платформи.

1.3 Порівняння підходів до рендерингу динамічних веб-сторінок та їх вплив на продуктивність

Сучасні веб-додатки характеризуються високим рівнем інтерактивності, великою кількістю динамічного контенту та значним обсягом операцій, що виконуються без перезавантаження сторінки [31]. У таких умовах вибір правильної моделі рендерингу відіграє ключову роль у забезпеченні стабільної продуктивності, швидкого завантаження та комфортної взаємодії користувача з інтерфейсом [11, 16]. У цьому розділі здійснюється порівняльний аналіз основних підходів до рендерингу - CSR, SSR, SSG та ISR - з урахуванням їх архітектурних особливостей, сценаріїв застосування та впливу на ключові показники ефективності веб-додатків [17, 18, 19, 20].

Рендеринг як процес формування візуального представлення веб-сторінки включає кілька етапів: завантаження HTML-документа, парсинг та побудова DOM, формування CSSOM, створення дерева рендерингу, обчислення геометрії елементів, генерування пікселів та відтворення їх на екрані [14, 16]. Ці процеси відбуваються як на сервері (у разі SSR/SSG/ISR), так і в браузері (у разі CSR). Вибір конкретного підходу визначає, де саме

відбувається основне навантаження - на клієнтській або серверній частині [16, 20].

Однією з найважливіших характеристик, які використовуються для оцінки підходів до рендерингу, є показники швидкодії, що відображають взаємодію користувача з додатком у перші секунди після відкриття сторінки [11]. До ключових метрик належать [11]:

- Time to First Byte (TTFB) - час, за який браузер отримує перший байт відповіді сервера;
- First Contentful Paint (FCP) - момент, коли на екрані з’являється перший видимий контент;
- Largest Contentful Paint (LCP) - час відображення найбільшого елемента сторінки;
- First Input Delay (FID) - час між взаємодією користувача та реакцією інтерфейсу;
- Cumulative Layout Shift (CLS) - стабільність макета сторінки.

Кожна модель рендерингу впливає на ці метрики по-різному, тому важливо розуміти, як саме архітектура формує поведінку сторінки [12, 16].

Таблиця 1.2. - *Вплив моделей рендерингу на ключові метрики продуктивності [11, 17, 18, 19, 20]*

Модель	**FCP**	**LCP**	**FID**	**SEO**	**Навантаження на сервер**
CSR	Низький	Середній	Низький (краще)	Поганий	Мінімальне
SSR	Високий	Високий	Середній	Відмінний	Високе
SSG	Високий	Високий	Низький	Відмінний	Низьке
ISR	Високий	Високий	Дуже низький	Відмінний	Низьке-середнє

Рендеринг на стороні клієнта (CSR). Client-Side Rendering забезпечує максимальну гнучкість у створенні інтерактивних інтерфейсів [20]. Усі компоненти та логіка завантажуються у браузер, після чого сторінка рендериться за допомогою JavaScript [1, 20]. Такий підхід підходить для динамічних SPA-додатків, де важлива швидкість роботи після початкового завантаження [31].

Основний недолік CSR - повільний перший рендер: поки браузер завантажує бандли JavaScript, розбирає їх та виконує, користувач може бачити порожній екран [25]. Високе значення FCP та LCP є типовими для таких систем [11, 12]. Крім того, пошукові системи можуть не проіндексувати контент, якщо він генерується лише на клієнті [17, 20].

Рендеринг на стороні сервера (SSR). Server-Side Rendering дозволяє зменшити час відображення першого контенту, оскільки HTML формується на сервері й передається користувачу вже готовим [17, 20]. Це значно покращує FCP, LCP та SEO [11, 17]. SSR ідеально підходить для сайтів, де кожен запит має повертати актуальний контент: новинні портали, каталоги товарів, публічні сторінки [20].

Однак SSR збільшує навантаження на сервер і потребує оптимізації кешування [20]. У складних системах рендеринг кожної сторінки може бути дорогим, що впливає на масштабованість системи [17, 20]. У великих проектах серверний рендеринг часто поєднують із CDN або кешем на рівні окремих компонентів [16, 20].

Статична генерація сторінок (SSG). Static Site Generation забезпечує найшвидшу продуктивність і найкращі значення показників FCP/LCP

завдяки тому, що сторінки створюються під час побудови (build), а користувач отримує їх безпосередньо з CDN [24]. SSG широко використовується для статичних сайтів - блогів, документацій, портфоліо [20].

Головний недолік цього методу - низька здатність до оновлення [20]. Якщо сторінка потребує динамічного контенту, система змушена виконувати додаткові запити на API або перегенерувати сторінку вручну. Це знижує зручність використання SSG для проектів, де дані часто змінюються [31].

Інкрементальна статична регенерація (ISR). ISR поєднує переваги SSR та SSG: він дозволяє створювати сторінки під час компіляції і водночас підтримує їх автоматичну регенерацію за розкладом або за потреби [18, 19]. Таким чином, перше завантаження максимально швидке, а контент залишається актуальним завдяки періодичному оновленню [18].

Цей підхід демонструє відмінні значення LCP та FID і підходить для суттєво масштабованих систем [11, 18]. ISR активно використовується в e-commerce, новинних платформах, каталогах, де потрібна як швидкість, так і актуальність контенту [18, 19].

Порівняння моделей у контексті продуктивності. Порівнюючи чотири підходи, можна виділити такі тенденції [11, 17, 18, 20]:

- найшвидший старт (FCP/LCP): SSG та ISR;
- найкраща SEO-оптимізація: SSR та ISR;
- найгірший перший рендер: CSR;
- найвища інтерактивність після завантаження: CSR та ISR;
- найменше навантаження на сервер: SSG;

- найкращий баланс швидкості та актуальності: ISR.

React підтримує всі ці моделі рендерингу через фреймворки (Next.js, Gatsby) або бібліотеки [1, 17, 20]. Важливо не тільки вибрати стратегію рендерингу, але й узгодити її з архітектурою самого додатку [16, 20]. Для застосунків зі складною логікою користувача та багатьма інтерактивними елементами CSR може бути оптимальним рішенням [20]. Для контентних сайтів краще підходить SSR або SSG, а ISR виступає універсальним варіантом для великих систем [17, 18, 19].

Важливо також враховувати, що ефективність рендерингу залежить не лише від того, де саме він виконується, але й від того, як організовано роботу з DOM [6, 16]. Надмірні операції над деревом елементів, часті перерахунки макета та оновлення стилів можуть суттєво знижувати продуктивність [6, 25]. Для мінімізації таких операцій React використовує Virtual DOM, який оптимізує diff-процес і зменшує кількість оновлень браузерного DOM [21, 15].

Server ← → Browser

	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is **removed**.	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.
Authoring:	Entirely server-side (request-response, HTML)	Built as if client-side (components, DOM*, fetch)	Built as client-side	Client-side	Client-side
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML **and** JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM
Server role:	Controls all aspects. (thin client)	Delivers static HTML	Renders pages (navigation requests)	Delivers static HTML	Delivers static HTML
Pros:	👍 TTI = FCP 👍 Fully streaming	👍 Fast TTFB 👍 TTI = FCP 👍 Fully streaming	👍 Flexible	👍 Flexible 👍 Fast TTFB	👍 Flexible 👍 Fast TTFB
Cons:	👎 Slow TTFB 👎 Inflexible	👎 Inflexible 👎 Leads to hydration	👎 Slow TTFB 👎 TTI >>> FCP 👎 Usually buffered	👎 TTI > FCP 👎 Limited streaming	👎 TTI >>> FCP 👎 No streaming
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size
Examples:	Gmail Basic HTML view, Hacker News	Docusaurus, Netflix*	Next.js, Razzle, etc	Gatsby, Vuepress, etc	Most apps

Малюнок 1.6. - *Порівняння моделей рендерингу веб-сторінок за ключовими характеристиками продуктивності [17, 18, 19, 20]*

Завдяки порівнянню різних моделей рендерингу можна зробити висновок, що оптимальна стратегія залежить від типу додатку, його цілей, обсягу даних та вимог до швидкодії [16, 20]. Використання ISR забезпечує найкращий баланс продуктивності, масштабованості та актуальності контенту, тоді як CSR залишається переважним варіантом для високонавантажених SPA [18, 20]. Розуміння особливостей кожної моделі дозволяє створювати ефективні та надійні веб-системи, орієнтовані на сучасні вимоги користувачів [31].

1.4 Чинники, що впливають на ефективність рендерингу динамічних веб-сторінок у React

Архітектура та ієрархія компонентів. Ефективність роботи React-додатка значною мірою залежить від того, як побудована структура компонентів [1, 2]. Надмірно глибока ієрархія, велика кількість вкладених елементів або повторюваних компонентів створює значне навантаження на Virtual DOM і впливає на швидкість оновлень [21, 15].

Під час зміни стану React порівнює попереднє та нове дерево елементів, виконуючи diff-операції [21, 15]. Якщо структура занадто складна, diff займає більше часу, що негативно позначається на FPS, швидкості анімацій та загальній чуйності інтерфейсу [10, 13, 15].

Оптимізація у вигляді React.memo, useMemo, useCallback, а також поділ на дрібні компоненти дозволяють уникнути зайвих повторних рендерів, значно покращуючи продуктивність великих інтерфейсів [16, 28, 31].

Управління станом у React. Глобальний стан є однією з найбільш ресурсомістких частин робочого циклу React-додатків [10]. Технології на кшталт Redux, Zustand, Recoil або Context API можуть викликати повторний рендер великої кількості компонентів, навіть якщо зміни стосуються лише окремого елемента [2, 10].

Зокрема, React Context має властивість оновлювати всіх підписників, що призводить до зайвих оновлень у великих системах [1, 2].

Щоб уникнути цього, застосовують такі підходи:

- локалізація стану на рівні окремих компонентів [2];

- використання бібліотек з fine-grained оновленнями [10];
- розділення глобального стану на незалежні частини [10];
- обмеження повторних запитів через мемоізацію даних або кеш (React Query, SWR) [10, 16].

Коректне управління станом суттєво знижує навантаження на механізми рендерингу та покращує продуктивність інтерфейсу.

Асинхронність, завантаження даних та concurrency features. Додатки з великим обсягом динамічних даних часто стикаються з проблемою блокування рендерингу через повільні або повторні запити на сервер [9, 16].

У React 18 впроваджено нові можливості [27, 28]:

- Concurrent Rendering
- Suspense для асинхронних операцій
- Automatic Batching

Ці механізми дозволяють уникнути блокування основного потоку виконання й забезпечують плавну роботу інтерфейсу навіть при високому навантаженні [2, 19, 13].

Якщо ж завантаження даних реалізовано неефективно, виникають такі проблеми:

- затримки у відображенні компонентів;
- повторні запити при переході між сторінками;
- збільшення часу рендерингу через очікування відповіді API [9, 16].

Оптимізація асинхронності - критичний фактор для динамічних SPA та масштабованих систем [16, 31].

Вплив браузера та процесів рендерингу. Ефективність рендерингу залежить не лише від React, але й від того, як браузер виконує операції над DOM [6, 16]. Кожне оновлення викликає:

- перерахунок стилів (Recalculate Styles);
- побудову макета (Layout / Reflow);
- перерисовку (Repaint) [6, 12, 15].

Якщо компоненти часто змінюють розміри, положення або стилі, браузер може виконувати layout багато разів на секунду, що суттєво знижує продуктивність [6, 13, 15].

Також впливають:

- важкі синхронні обчислення у функції render [10, 15];
- надмірна кількість ефектів useEffect [8, 10];
- анімації, що не використовують апаратне прискорення [12, 16].

React зменшує кількість оновлень DOM через Virtual DOM, але некоректне написання компонентів все одно може створити значне навантаження [1, 5, 10].

Обсяг JavaScript-коду та оптимізація бандлу. Кількість JS-коду безпосередньо впливає на FCP, LCP та загальний час запуску додатка [11, 25]. Великий бандл призводить до:

- тривалого завантаження;
- збільшення часу парсингу;
- затримок під час виконання коду;

- блокування основного потоку браузера [9, 25, 31].

Тому важливими стають такі техніки оптимізації:

- code splitting - поділ коду на частини;
- lazy loading - завантаження компонентів лише за потреби;
- tree shaking - видалення невикористаного коду;
- мінімізація сторонніх залежностей [5, 11].

Це дозволяє суттєво підвищити початкову продуктивність React-додатків [11, 16].

Стилізація та оптимізація CSS. Велика кількість глобальних стилів уповільнює перерахунок макета, що впливає на частоту Reflow і Repaint [20, 19, 35].

Сучасні підходи до стилізації допомагають уникнути цього [16]:

- CSS Modules - ізолюють стилі для кожного компонента;
- Styled Components, Emotion - інкапсулюють стилі на рівні елементів;
- використання transform/opacity замість top/left покращує анімації.

Оптимізована структура CSS безпосередньо впливає на швидкодію рендерингу [6, 16].

Серверні обмеження та інфраструктурні чинники. У випадку SSR, SSG та ISR важливими стають [17, 18, 20]:

- швидкість відповіді сервера;
- наявність кешування;
- ефективність CDN;

- використання edge-функцій;
- оптимізація бази даних.

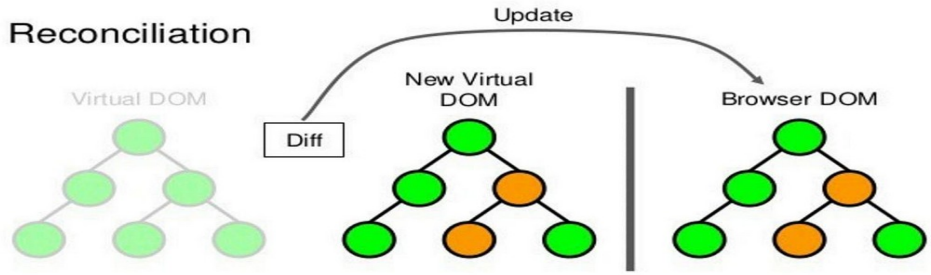
SSR може бути повільним, якщо сервер рендерить складні сторінки без кешу [17, 20].

SSG/ISR навпаки отримують вигоду від CDN, забезпечуючи мінімальний TTFB [18, 20].

Таким чином інфраструктура безпосередньо формує продуктивність рендерингу [16, 20].

Таблиця 1.3. - *Основні чинники, що впливають на продуктивність рендерингу в React [1, 2, 6, 11, 12, 15, 16, 17, 18, 20, 25]*

Чинник	**Як впливає на продуктивність**
Структура компонентів	Впливає на частоту повторних рендерів та розмір Virtual DOM
Управління станом	Може спричиняти масові оновлення компонентів при зміні контексту або глобального стану
Асинхронні операції	Можуть блокувати потік, затримуючи рендеринг або змінюючи UX
Обсяг JS-бандлу	Збільшує час парсингу та початкового рендерингу
Робота браузера (Layout, Paint)	Викликає Reflow/Repaint, що впливає на FPS та плавність інтерфейсу
Стилізація (CSS)	Неефективні стилі призводять до дорогих перерахунків макета
Серверні обмеження (SSR/SSG/ISR)	Визначають TTFB, швидкість генерації HTML і масштабованість



Малюнок 1.7. - *Механізм узгодження (Reconciliation) та обчислення різниці (Diff) у React [21, 15]*

Підсумок підрозділу 1.4. Ефективність рендерингу у React визначається взаємодією численних чинників: архітектурою компонентів, управлінням станом, асинхронністю, оптимізацією JavaScript, побудовою стилів, особливостями роботи браузера та можливостями серверної інфраструктури. Кожен із цих елементів впливає на ключові метрики продуктивності, формує чуйність інтерфейсу та визначає масштабованість системи в умовах реального використання.

1.5 Висновки до розділу 1

У першому розділі було проведено комплексний аналіз архітектурних підходів та технологій, що визначають ефективність рендерингу динамічних веб-сторінок на прикладі React. На основі розглянутих матеріалів встановлено, що оптимізація продуктивності клієнтських інтерфейсів залежить від поєднання низки чинників: роботи браузерного рушія, вибору моделі рендерингу (CSR, SSR, SSG, ISR), організації структури компонентів, способів управління станом, особливостей виконання асинхронних операцій та застосування методів побудови й оновлення Virtual DOM.

Одним із ключових аспектів, що впливають на продуктивність, є процес формування DOM і CSSOM, а також побудова дерева рендерингу, де значну роль відіграють операції Layout та Paint. Дослідження показало, що операції Reflow та Repaint є найбільш затратними з точки зору ресурсів, а надмірні зміни DOM або неефективні стилі можуть суттєво знижувати швидкодію інтерфейсу. Саме тому актуальним є використання віртуальної моделі DOM, яка дозволяє мінімізувати кількість маніпуляцій з реальним DOM за рахунок попереднього порівняння станів (Diffing) та застосування оптимізованих оновлень.

Порівняння моделей рендерингу дало змогу визначити їхні сильні та слабкі сторони. CSR забезпечує високу інтерактивність, але має слабкі показники першого рендера та SEO. SSR покращує початкову швидкість і оптимізує індексацію, однак збільшує навантаження на сервер. SSG гарантує максимальну швидкість завдяки статичній генерації контенту, проте не підходить для часто змінюваних даних. ISR є компромісним підходом, що поєднує переваги статичної генерації з можливістю часткового оновлення контенту без повної перебудови сторінки.

Аналіз чинників продуктивності показав, що ефективність рендерингу в React безпосередньо залежить від:

- коректно організованої структури компонентів;
- оптимального управління станом (особливо глобальним);
- зменшення кількості асинхронних блокувань;
- мінімізації обсягу JavaScript-бандлу;
- грамотної роботи зі стилями;
- вибору відповідної моделі рендерингу залежно від типу застосунку.

Кожен із цих факторів впливає на ключові метрики продуктивності - FCP, LCP, FID, TTFB, CLS, що є визначальними для оцінки якості взаємодії користувача з інтерфейсом.

Отже, проведене дослідження показало, що ефективність рендерингу в React є багатофакторною і визначається як технологічними особливостями платформи, так і архітектурними рішеннями, що застосовуються під час розробки. Розуміння взаємодії цих механізмів є фундаментальним для побудови високопродуктивних веб-застосунків та є ключовим етапом перед переходом до практичного проєктування та оптимізації системи у наступних розділах роботи.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ТА МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРИНГУ REACT-ЗАСТОСУНКІВ

2.1 Архітектурні підходи та принципи побудови сучасних React-застосунків

Сучасні веб-додатки характеризуються високими вимогами до швидкодії, масштабованості, інтерактивності та гнучкості оновлень інтерфейсу [11, 16, 31]. React, як одна з найпопулярніших бібліотек для створення користувацьких інтерфейсів, пропонує набір архітектурних принципів, що дозволяють ефективно керувати рендерингом, станом та оновленням компонентів [1, 2]. Актуальність дослідження цих принципів зумовлена тим, що саме архітектурні рішення визначають продуктивність динамічних сторінок і впливають на ключові метрики UI - FCP, LCP, FID, CLS [11, 16].

Компонентна архітектура як основа побудови інтерфейсу. Реалізація інтерфейсу в React базується на компонентній моделі, де кожен елемент UI є ізольованим модулем зі своїм станом, логікою та правилом відображення [1]. Компоненти можуть бути простими (кнопка, інпут, карточка) або складними (форма, таблиця, панель керування) [1, 34].
Такий підхід забезпечує:

- повторне використання коду (reusability) [1, 2];
- ізоляцію логіки та локальні оновлення [1];
- скорочення обсягу DOM-операцій [5, 15];
- оптимізацію рендерингу через дроблення UI на дрібні частини [10, 23].

Компонентна архітектура також дозволяє впроваджувати різні патерни проектування інтерфейсу, зокрема: Presentational/Container Components, Render Props, Higher-Order Components та Hook-based підхід [1, 2, 21].

Односпрямований потік даних та важливість контролю стану. React реалізує концепцію односпрямованого потоку даних (one-way data flow), за якої інформація передається від батьківських компонентів до дочірніх через пропси [1]. Такий підхід:

- зменшує кількість непередбачуваних змін [1, 2];
- спрощує трасування стану [2];
- покращує ефективність рендерингу [10].

Однак при збільшенні масштабів застосунку постає проблема керування складним станом, що спричинює надмірні оновлення дерев компонентів [10, 23]. Для цього використовуються спеціалізовані засоби:

- Context API - для спільного доступу до стану [1, 2];
- Redux, Zustand, MobX, Recoil - для предбачуваного управління складними даними [10, 23];
- React Query, SWR - для кешування та синхронізації даних із сервером [10, 23].

Правильний вибір стейт-менеджменту є критичним, оскільки він впливає на кількість повторних рендерингів і складність оновлення дерев компонентів.

Таблиця 2.1. - *Порівняння архітектурних підходів у React-застосунках [1, 2, 5, 6, 10, 15]*

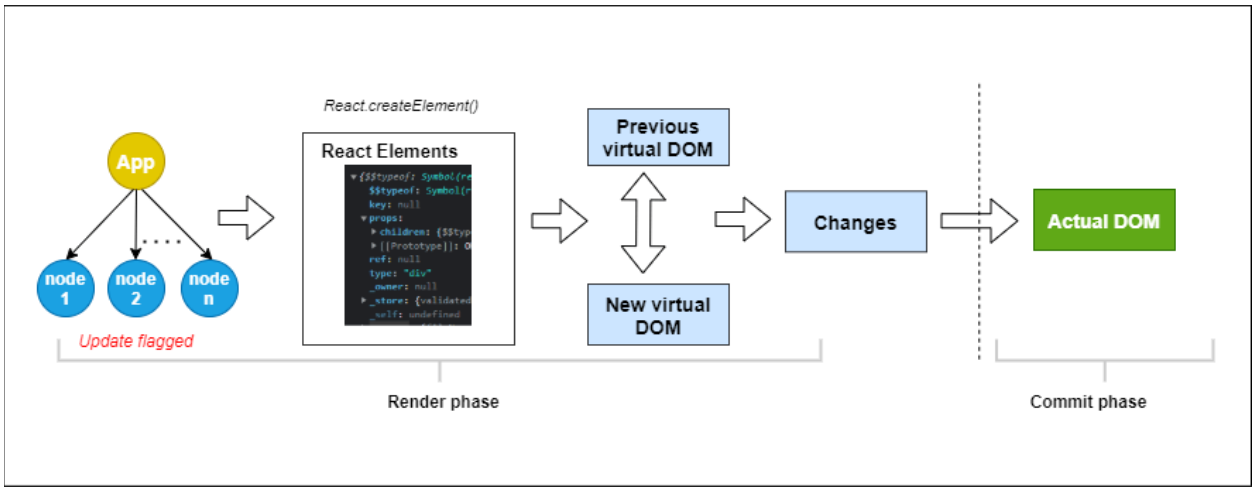
Підхід	Опис	Переваги	Недоліки
Компонентна архітектура	UI розбивається на незалежні компоненти	Повторне використання, ізоляція логіки, масштабованість	Може ускладнювати структуру при надто дрібних компонентах
Односпрямований потік даних	Дані передаються згори вниз через props	Простежуваність, передбачуваність оновлень	Потребує стейт-менеджера у великих проєктах
Хуки (Hooks API)	Логіка компонента через useState, useEffect, useMemo тощо	Скорочення коду, простота повторного використання логіки	Можливі проблеми з оптимізацією ефектів
Віртуальна DOM-модель	Оновлення в пам'яті з diffing перед commit у DOM	Зменшення Reflow/Repaint, стабільність	Має власні накладні витрати на обчислення
Fiber-архітектура	Поділ рендерингу на частини з можливістю переривання	Плавність UI, краща обробка великих дерев	Складність реалізації, додаткові абстракції

Життєвий цикл компонентів та оновлення інтерфейсу. React компоненти проходять певні фази створення та оновлення: mount → update → unmount [1].

У класових компонентах це реалізовано через методи lifecycle, але сучасний функціональний підхід використовує хуки [1, 8]:

- useEffect - виконання побічних ефектів [8];
- useMemo / useCallback - мемоізація та оптимізація обчислень [28, 31];
- useLayoutEffect - синхронні операції, що впливають на макет [6, 13].

Ці механізми визначають частоту оновлень дерева елементів та обсяги рендерингу [1, 10].



Малюнок 2.1. - *Схема процесу рендерингу React-компонента та формування Virtual DOM [21, 15]*

Virtual DOM як архітектурна концепція оптимізації. Одним із базових елементів React є віртуальна модель DOM, що відіграє ключову роль у продуктивності застосунків [5]. Virtual DOM дозволяє:

- виконувати попередні розрахунки в пам’яті, без взаємодії з реальним DOM [5];

- мінімізувати кількість Reflow/Repaint [6, 16];
- застосовувати diff-алгоритм для точкового оновлення [21, 15];
- забезпечувати узгоджене та прогнозоване оновлення UI [1].

Завдяки цьому React уникає надмірних операцій із DOM, які є найбільш затратними для браузера [6, 16]. У поєднанні з Fiber-архітектурою Virtual DOM дає змогу React оптимізувати рендеринг під час високого навантаження, розподіляти оновлення у часі та переривати тривалі обчислення [1, 15, 21].

Модульність та масштабованість архітектури. Сучасні React-проєкти будуються з урахуванням:

- розподілу коду на логічні модулі (feature-based structure) [16];
- виділення окремих UI-бібліотек та компонентних систем. [34];
- застосування code splitting для скорочення JS-бандлу [25];
- використання server components у Next.js для зміщення навантаження на сервер [17].

Такий підхід дає змогу адаптувати застосунок до різних моделей рендерингу (CSR, SSR, SSG, ISR), використовуючи оптимальний баланс між швидкодією, SEO та інтерактивністю [17, 18, 20].

2.2 Життєвий цикл компонентів React та вплив оновлень стану на рендеринг

Життєвий цикл компонентів у React. У React кожен компонент проходить певні стадії створення, оновлення та видалення. Ці стадії утворюють життєвий цикл, який визначає, коли виконуються рендеринг,

обчислення ефектів, оновлення стану та очищення [1]. Розуміння життєвого циклу є ключовим для оптимізації продуктивності, оскільки кожне оновлення стану потенційно спричиняє повторний рендеринг компонентів [2, 10].

У класових компонентах життєвий цикл описувався методами на кшталт: componentDidMount, shouldComponentUpdate, componentDidUpdate, componentWillUnmount [1].

У функціональних компонентах ці механізми реалізовані через хуки [8, 27]: useEffect, useLayoutEffect, useMemo, useCallback, useReducer.

Попри зовнішні відмінності, логіка роботи React однакова: зміна стану викликає процес оновлення віртуального дерева елементів та подальшого оновлення DOM (за потреби) [21, 15].

Фази життєвого циклу компонентів. У сучасних функціональних компонентах React виділяють дві ключові фази [1, 21]:

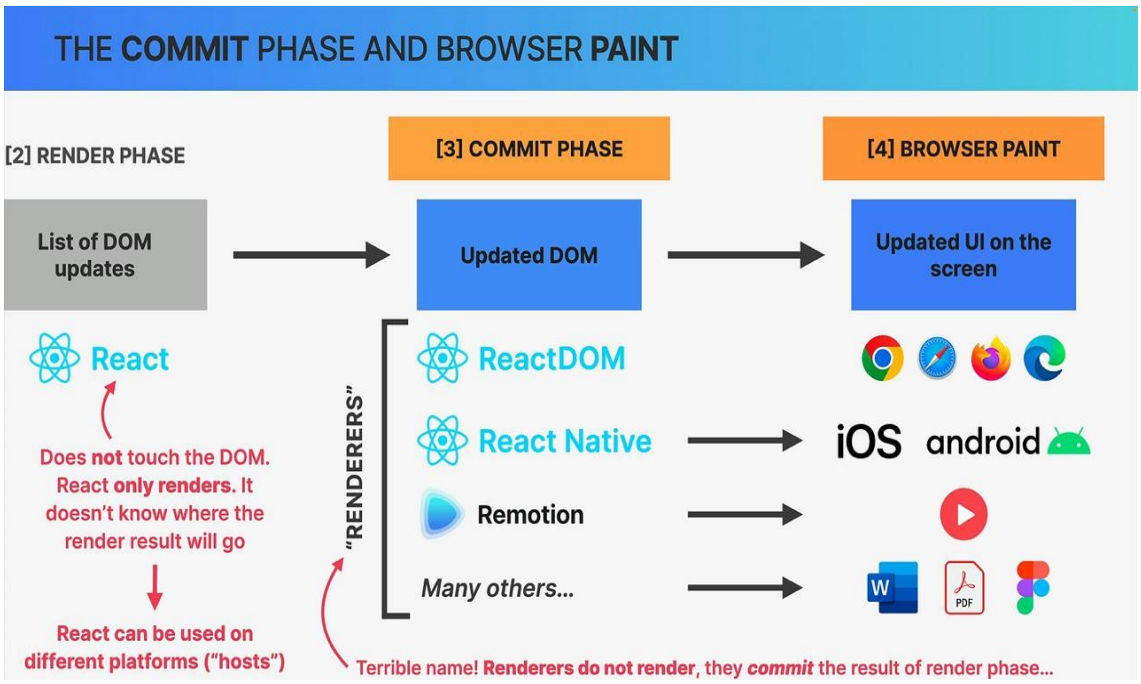
1. Render Phase (фаза рендерингу. Під час цієї фази:
    - викликається тіло компоненту;
    - обчислюється Virtual DOM;
    - аналізуються залежності хуків (наприклад, useEffect, useMemo);
    - React визначає, чи потрібно оновлювати елементи DOM [21, 15].

    Ця фаза чиста та синхронна: React не допускає сайд-ефектів у тілі компоненту [21].

2. Commit Phase (фаза застосування оновлень). У цій фазі:
    - застосовуються зміни до реального DOM;

- виконуються ефекти useEffect та очищення ефектів попередніх рендерів;
- браузер оновлює інтерфейс [1, 6, 15].

Commit Phase безпосередньо впливає на продуктивність, адже пов’язана з реальними операціями у DOM, які є дорогими [6, 15].



Малюнок 2.2. - *Узагальнена схема етапів Render, Commit та Browser Paint у React [1, 5, 6]*

Таблиця 2.2. - *Вплив оновлень стану на рендеринг компонентів у React [1, 2, 10, 21, 23, 31]*

Тип оновлення	**Що викликає**	**Вплив на рендеринг**	**Рекомендації оптимізації**
Оновлення стану (useState)	setState(), події	Рендер поточного компоненту + дочірніх	Мемоізація пропсів, React.memo
Оновлення контексту (useContext)	Зміна value у Provider	Повторний рендер *усіх компонентів*, що споживають контекст	Дроблення контекстів, selector pattern
Передача немемоізованих функцій	Нове посилання функції при кожному рендері	Ререндер дочірніх компонентів	useCallback
Передача немемоізованих об'єктів/масивів	Нове посилання	Зайві ререндери	useMemo
Оновлення глобального стану (Redux/Zustand)	dispatch()	Ререндер компонентів, що підписані на slice	Selectors, мемоізація selector'ів
Асинхронні ефекти (useEffect)	Зміна залежностей	Може спричиняти каскадне оновлення	Коректний масив залежностей

Вплив оновлень стану на частоту рендерів. Зміна стану (state) або пропсів (props) майже завжди призводить до повторного рендерингу компоненту [1]. Важливо розуміти, які оновлення викликають повторні обчислення [10, 23]:

- Оновлення локального стану (useState). Викликає рендер у поточному компоненті та всіх дочірніх, якщо їхні пропси змінюються [1, 2].
- Зміна контексту (useContext). Викликає повторний рендер усіх компонентів, які підписані на контекст [1, 2].
- Зміна посилань у пропсах. Якщо передавати функції або об’єкти без мемоізації, це спричиняє зайві оновлення дочірніх компонентів [28, 31].
- Немемоізовані колбеки. Кожен рендер створює нову функцію, що може викликати оновлення компонентів з React.memo [28, 31].

Таким чином, неправильна робота зі станом може значно збільшити кількість рендерингів і погіршити продуктивність [10, 23].

Механізми оптимізації оновлень. React пропонує кілька інструментів для контролю частоти рендерів і зменшення навантаження [1, 2]:

- React.memo - мемоізація компонентів, які залежать тільки від пропсів [21].
- useMemo - кешування обчислень [28, 31].
- useCallback - кешування функцій [28, 31].
- useTransition - позначення оновлень як "неголовних", що не блокують UI [1, 2].
- useDeferredValue - відкладене оновлення низького пріоритету [1, 2].

- Batching updates - об’єднання кількох оновлень стану в один рендер [27].

Правильне застосування цих механізмів дозволяє зменшити кількість повторних оновлень та підвищити FPS інтерфейсу [10, 13, 16].

Життєвий цикл і віртуальний DOM. У взаємодії з життєвим циклом ключову роль відіграє Virtual DOM. Саме під час Render Phase React створює нове віртуальне дерево та порівнює його з попереднім варіантом [21, 15]. Лише мінімальні зміни передаються у браузер, що дозволяє значно зменшити кількість операцій над DOM [5, 6, 15].

Таким чином, життєвий цикл компонентів безпосередньо визначає:

- частоту оновлень Virtual DOM;
- кількість diff-операцій;
- складність commit-фази;
- обсяг взаємодії з реальним DOM [1, 5, 6, 15].

Підсумок підрозділу 2.2. Життєвий цикл компонентів React визначає послідовність оновлення інтерфейсу та форму поведінку застосунку під час зміни стану. Неефективна робота зі станом або контекстом може спричинити зайві повторні рендеринги і погіршити продуктивність, тоді як правильне використання механізмів оптимізації дозволяє зменшити навантаження на DOM, покращити плавність UI та підвищити масштабованість застосунку.

2.3 Оптимізація роботи з Virtual DOM та diff-алгоритмами

Ефективність рендерингу React значною мірою залежить від того, наскільки швидко бібліотека може визначити зміни у стані застосунку та оновити відповідні частини інтерфейсу [21, 15]. Центральним елементом цього механізму є Virtual DOM - абстрактне представлення дерева UI, що дозволяє виконувати обчислення в пам’яті, а не безпосередньо в реальному DOM, мінімізуючи дорогі операції роботи з браузером [5, 6, 15].

React використовує diff-алгоритм, який порівнює попереднє та нове дерево Virtual DOM і визначає мінімальний набір змін, необхідний для оновлення інтерфейсу [21, 15]. Цей процес значно пришвидшує відтворення UI, але його ефективність залежить від структури компонентів, частоти оновлень та оптимізації рендерингу [10, 23].

Механізм порівняння дерев (Reconciliation). Під час кожного оновлення стану або пропсів React повторно викликає функції компонентів та генерує нову версію Virtual DOM [1]. Щоб зрозуміти, які елементи потрібно оновити [21, 15]:

1. Порівнюються кореневі елементи обох дерев. Якщо типи елементів відрізняються (наприклад, →), React повністю реконструює піддерево [1].
2. При однаковому типі виконується покрокове порівняння дочірніх елементів [5, 15]. Порівняння відбувається зліва направо, що робить порядок списків критично важливим.
3. Ключі (key) визначають стабільність елементів у списках. Неправильний вибір ключів може спричиняти масові повторні рендери й втрату станів дочірніх елементів [21, 15].

Алгоритм оптимізований під припущення, що елементи рідко переміщуються, тому він не виконує повного порівняння дерев, що дозволяє

досягти продуктивності O(n) замість $O(n^3)$, характерної для повного комбінаторного порівняння дерев [5, 30].

Причини зайвих оновлень Virtual DOM. Попри свою ефективність, Virtual DOM може частіше оновлюватися, ніж необхідно [10]. До основних причин зайвих diff-операцій належать [10, 21, 23, 28, 31]:

- передача нових посилань на функції при кожному рендері;
- зміна об’єктів або масивів, що передаються як пропси;
- глобальні оновлення стану (Context, Redux), що тригерять повторні рендери всієї гілки;
- надмірні useEffect, які викликають повторні синхронізації;
- відсутність мемоізації у важких обчисленнях або компонентах із великою кількістю дочірніх елементів.

Таблиця 2.3. - *Порівняння типових причин зайвих оновлень Virtual DOM та способів їх оптимізації [1, 5, 10, 21, 23, 28, 31]*

Причина зайвого оновлення	**Що викликає**	**Наслідок**	**Рекомендована оптимізація**
Нові посилання на функції	функції створюються при кожному рендері	зайві ререндери дочірніх компонентів	useCallback
Нові об’єкти/масиви у пропсах	створення нових {} або [] при рендері	diff вважає дані “змінені”	useMemo, винесення констант
Глобальні оновлення контексту	зміна value у Provider	рендер усіх споживачів	дроблення контексту, selector pattern
Важкі обчислення	складна логіка в	уповільнення diff	useMemo,

Причина зайвого оновлення	**Що викликає**	**Наслідок**	**Рекомендована оптимізація**
у рендері	компоненті	і commit	винесення логіки
Неправильно підібрані ключі	використання index як key	некоректний diff, втрати стану	стабільні ключі (id)
Надмірні ефекти useEffect	зайві залежності	каскадні оновлення VDOM	коректні dependency arrays

Методи оптимізації diff-процесу. Для зменшення навантаження на Virtual DOM застосовуються такі підходи [1, 2, 10]:

Мемоізація компонентів та значень.

- React.memo - оптимізує функціональні компоненти, запобігаючи повторному рендеру, якщо пропси не змінилися [21].
- useMemo - кешує обчислення [28, 31].
- useCallback - кешує функції [28, 31].

Ці техніки скорочують кількість викликів рендеру та зменшують diff-навантаження [10, 23].

Оптимальне використання ключів (key). Правильно підібрані ключі:

- дозволяють React коректно відстежувати елементи списків [21, 15];
- запобігають зайвим створенням та видаленням DOM-вузлів [5];
- зменшують кількість diff-операцій [10].

Натомість використання індекса масиву як ключа може призвести до неправильних оновлень UI [21, 15].

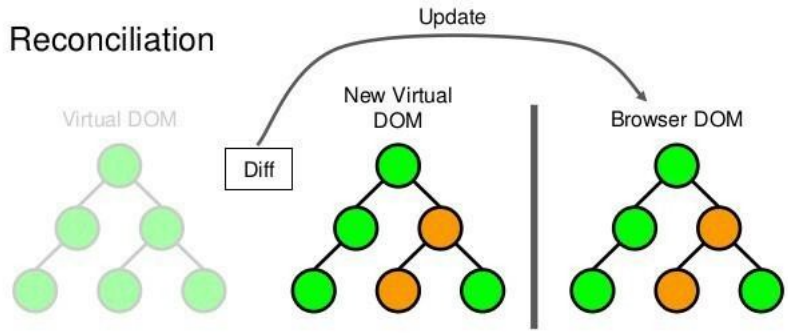
Дроблення контексту та selector-патерни. Контекст оновлює всіх споживачів, тому варто [2, 10, 23]:

- розділяти великий контекст на менші,
- використовувати бібліотеки з selector-функціями (наприклад, Zustand або Jotai),
- кешувати обчислення для контекстних значень.

Це зменшує кількість компонентів, які повторно обчислюють Virtual DOM [10, 23].

Винос важких компонентів за допомогою lazy-loading. React.lazy дозволяє [1, 25]:

- не відтворювати компонент, поки він не потрібен,
- розділяти рендер на частини,
- зменшувати нагромадження diff-операцій у критичні моменти.



Малюнок 2.3. - *Схема процесу узгодження (Reconciliation) та обчислення різниці (Diff) у React [21, 15]*

Підсумок підрозділу 2.3. Оптимізація роботи Virtual DOM і diff-алгоритмів є ключовим чинником у досягненні високої продуктивності React-застосунків. Правильне структурування компонентів, мемоізація, оптимальний вибір ключів та грамотне управління станом дають змогу істотно зменшити кількість diff-операцій, що виконуються під час оновлення інтерфейсу. Завдяки цьому покращуються показники швидкодії, зменшується навантаження на JavaScript-двигун і забезпечується чуйність та плавність роботи UI, що є критично важливим для сучасних складних веб-застосунків.

2.4 Оптимізація продуктивності React-застосунків на основі аналізу рендерингу

Продуктивність React-застосунків значною мірою залежить від коректної організації рендерингу, оптимального використання механізмів оновлення стану та мінімізації зайвих оновлень компонента [1, 5, 10]. Аналіз рендер-

циклів, що включає Render Phase, Diff та Commit Phase, дозволяє визначити “вузькі місця” та застосувати найбільш ефективні методи оптимізації [1, 15].

Одним із ключових джерел проблем продуктивності є надмірні повторні рендери, які виникають через зміни стану, неправильне використання контексту або передачу нових пропсів на кожному рендері [10, 21, 23]. Зменшити кількість непотрібних оновлень дозволяють такі підходи:

Мемоізація компонентів. Компоненти, які отримують однакові пропси, не повинні повторно рендеритись без потреби. React.memo() зменшує кількість повторних рендерів для функціональних компонентів, якщо вхідні дані не змінилися [21].

Мемоізація значень та колбеків. Використання useMemo() та useCallback() дозволяє запобігти створенню нових посилань на функції або об'єкти, що часто спричиняє каскадні оновлення дочірніх елементів [28, 31].

Оптимізація контексту. Context API може викликати повторний рендер усіх споживачів при зміні value [2, 10].

Щоб уникнути цього, застосовують [2, 10, 23]:

- розбиття контекстів на менші,
- використання selector-патернів,
- поєднання контексту з зовнішніми state-менеджерами.

Зменшення важких обчислень у рендері. Всі складні обчислення повинні виконуватися поза рендер-функцією або мемоізуватись, оскільки кожен рендер запускає обчислення повторно [15, 21].

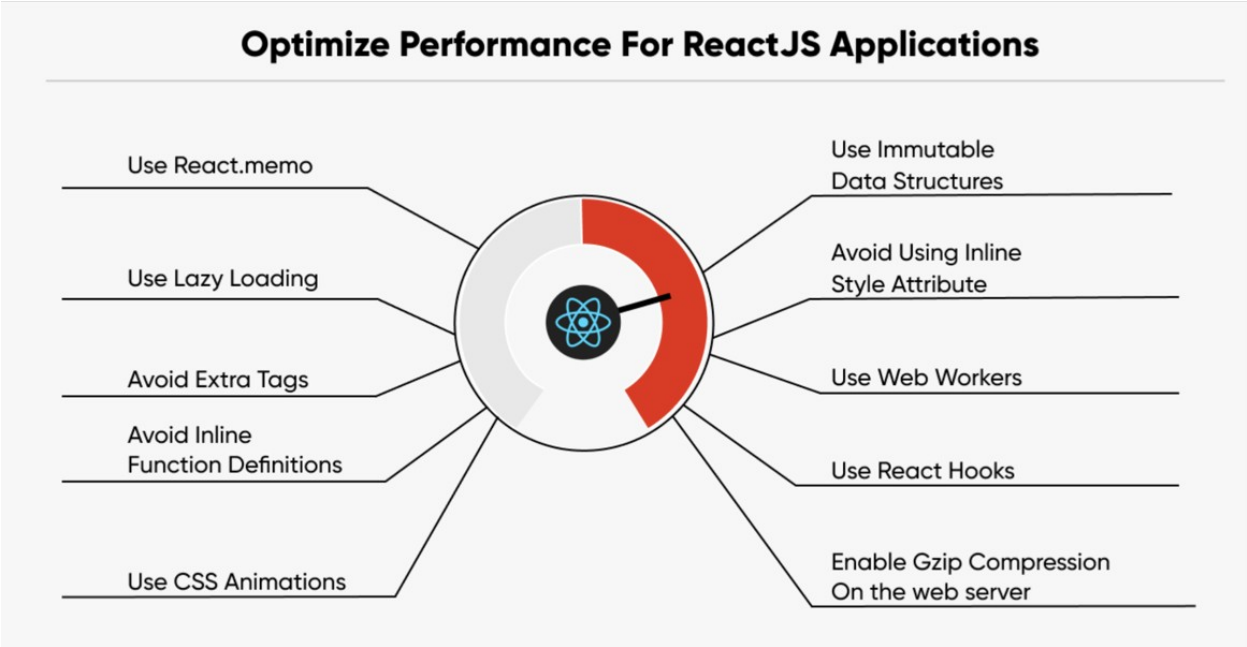
Оптимізація списків. Використання стабільних ідентифікаторів (key) забезпечує коректну роботу diff-алгоритму і зменшує кількість операцій оновлення DOM [21, 15].

Аналіз профілювання. Завантаження браузерного профайлера (Performance або React DevTools Profiler) дозволяє визначити компоненти, які рендеряться надто часто, та знайти причини зайвих оновлень [11, 16, 27].

Таблиця 2.4. - *Основні техніки оптимізації продуктивності React-застосунків [1, 5, 10, 16, 21, 27, 28, 31]*

Техніка оптимізації	У чому суть	Коли застосовувати	Очікуваний ефект
React.memo	Мемоізація компонентів при незмінених пропсах	Компоненти без складної логіки, які часто повторно рендеряться	Зменшення непотрібних рендерів
useMemo	Мемоізація важких обчислень	Складні розрахунки в рендері або великі масиви	Прискорення Render Phase

Техніка оптимізації	У чому суть	Коли застосовувати	Очікуваний ефект
useCallback	Мемоізація колбеків, щоб не створювались нові функції	Передача колбеків у дочірні компоненти	Зменшення оновлень дітей
Оптимізація контекстів	Поділ Context API або selector pattern	Великий глобальний стан	Зменшення каскадних рендерів
Стабільні ключі (key)	Уникання index як key	Рендер списків	Коректна робота diff-алгоритму
Code-splitting, lazy loading	Динамічне завантаження модулів	Великі застосунки	Швидке завантаження сторінок
Профілювання DevTools	Аналіз “важких” компонентів	Пошук вузьких місць	Точкове покращення продуктивності



Малюнок 2.4. - *Основні підходи до оптимізації продуктивності React-застосунків [1, 10]*

Підсумок підрозділу 2.4. Оптимізація продуктивності React-застосунків ґрунтується на правильному балансі між структурою компонентів, управлінням станом, мемоізацією, оптимізацією diff-процесу та контролем рендерів. Використання аналізу рендер-циклів та інструментів профілювання дає змогу своєчасно виявляти проблемні компоненти, мінімізувати навантаження на DOM і забезпечити високу швидкодію застосунку навіть при збільшенні обсягів даних і складності інтерфейсу.

2.5 Інструменти моніторингу та діагностики продуктивності React-застосунків

Ефективна оптимізація продуктивності неможлива без постійного моніторингу та аналізу поведінки застосунку під час виконання [1, 16, 27]. Інструменти діагностики дозволяють виявляти вузькі місця, відстежувати

повторні рендери, аналізувати навантаження на DOM та контролювати роботу JavaScript у реальному часі [10, 15].

Основні групи інструментів

1. React DevTools (Profiler). Один із ключових інструментів для оцінювання продуктивності компонентів [26, 29]. Дозволяє:

- визначити, які компоненти рендеряться найчастіше;
- виміряти тривалість рендерів;
- переглянути причини оновлень (props/state/context);
- знайти компоненти, які уповільнюють інтерфейс.

Profiler є незамінним під час аналізу великих дерев компонентів і оптимізації diff-процесу [1, 5, 21].

2. Performance панель Chrome DevTools. Дає змогу досліджувати роботу браузера в реальному часі [6]:

- завантаження ресурсів;
- виконання JavaScript;
- Reflow/Repaint;
- FPS та “фрізи” інтерфейсу;
- час виконання ефектів та колбеків.

Цей інструмент дозволяє побачити зв’язок між діями користувача та змінами в DOM [12, 14].

3. Lighthouse. Автоматизований інструмент для вимірювання Web Vitals [12]:

- LCP (Largest Contentful Paint);
- FID/INP (затримка введення);

- CLS (стабільність макета).

Показники Lighthouse дозволяють оцінити загальну якість продуктивності та визначити, чи відповідає застосунок рекомендаціям Google [11].

4. Інструменти аналізу бандлу (Webpack Bundle Analyzer / Source Map Explorer)

Використовуються для:

- виявлення великих залежностей;
- оцінки ваги компонентів;
- оптимізації імпортів, code-splitting та tree-shaking.

Зменшення розміру бандла безпосередньо впливає на швидкість завантаження та TTFB [18, 20].

Роль моніторингу у забезпеченні продуктивності. Постійний аналіз поведінки застосунку дозволяє виявляти проблеми на ранніх етапах та швидко локалізувати джерела падіння продуктивності [10, 11, 27]. Комбінація React DevTools та браузерних інструментів надає повну картину роботи інтерфейсу, а інструменти автоматичного аудиту допомагають сформувати напрямки подальшої оптимізації [1, 16].

Підсумок підрозділу 2.5. Моніторинг продуктивності є невід’ємною частиною розробки React-застосунків. Використання Profiler, Chrome DevTools, Lighthouse та аналізаторів бандла забезпечує змогу швидко виявляти проблеми, оцінювати ефективність оптимізацій та підтримувати високу продуктивність інтерфейсу на всіх етапах життєвого циклу застосунку.

2.6 Висновки до розділу 2

У цьому розділі було виконано комплексне дослідження технологій і методів оптимізації рендерингу React-застосунків. Розглянуто сучасні архітектурні підходи, принципи роботи компонента та механізми оновлення інтерфейсу, що визначають продуктивність веб-додатків. Аналіз показав, що ефективність React значною мірою залежить від оптимальної організації компонентної ієрархії, передбачуваного управління станом та мінімізації зайвих повторних рендерів.

Було досліджено вплив Render-, Diff- та Commit-фаз рендер-циклу на навантаження браузера, а також визначено, які саме оновлення стану та контексту є найбільш «дорогими». Окрему увагу приділено механізмам оптимізації: мемоізації компонентів і значень, коректному використанню ключів, стабільності об’єктів та функцій, код-сплітингу, профілюванню та аналізу «вузьких місць».

Результати аналізу показують, що продуктивність React-застосунків формується як поєднання архітектурних рішень, організації стану, якості реалізації компонентів та вибору відповідних інструментів оптимізації. Використання diff-алгоритму, Virtual DOM, сучасних хуків та технік розділення коду дозволяє суттєво зменшити кількість зайвих обчислень, оптимізувати роботу DOM і забезпечити плавність інтерфейсу навіть у складних застосунках.

Загалом розділ показує, що продуктивність React - це результат системного підходу, а не окремих рішень: лише поєднання грамотної архітектури, продуманого управління станом та цілеспрямованої оптимізації дозволяє досягти високої швидкодії та масштабованості сучасних веб-додатків.

РОЗДІЛ 3. РОЗРОБКА ТА ОПТИМІЗАЦІЯ REACT-ЗАСТОСУНКУ

3.1 Постановка задачі та опис призначення застосунку

У межах даного дослідження розроблено демонстраційний React-застосунок, який реалізує відображення великого списку динамічних даних у реальному часі [1, 10, 27]. Головною метою застосунку є аналіз впливу різних методів оптимізації рендерингу на продуктивність інтерфейсу, швидкість реакції на дії користувача та навантаження на браузер [11, 16].

Застосунок представляє собою односторінковий інтерфейс (SPA), що відображає список із понад 2000 користувачів [20, 31]. Кожен елемент містить інформацію про ім’я, електронну пошту, місто, вік та обчислюваний показник score, який формується за допомогою спеціально ускладненого алгоритму. Це створює навантаження, достатнє для вимірювання продуктивності [23, 32].

Ключова взаємодія користувача із застосунком полягає у пошуку елементів за текстовим запитом, що призводить до фільтрації списку та повторного рендерингу усіх відповідних компонентів [21, 15]. Саме цей сценарій є показовим для аналізу ефективності рендерингу у React-застосунках [10, 21].

Для експериментального порівняння створено дві незалежні реалізації інтерфейсу:

- неоптимізована версія, яка призводить до надмірних повторних рендерів та високого навантаження на браузер [6, 16];
- оптимізована версія, що використовує React.memo, useMemo, useCallback та коректні ключі елементів [16, 28, 31].

Така конфігурація дозволяє наочно оцінити різницю у продуктивності при використанні сучасних підходів оптимізації у React [10, 27].

Таблиця 3.1. - *Основні характеристики експериментального застосунку*

Параметр	**Значення**
Технологічний стек	React 18, Vite, JavaScript (ES6+), React DevTools
Кількість елементів у списку	2000+
Призначення	Дослідження впливу оптимізацій рендерингу
Джерело даних	Генерація тестових користувачів у пам’яті
Основний сценарій тестування	Динамічна фільтрація списку за введеним запитом
Показник навантаження	heavyCalculation() для кожного елемента
Порівнювані реалізації	Unoptimized vs Optimized

Розроблений застосунок є тестовим стендом для дослідження ключових показників продуктивності [10, 27]:

- кількість повторних рендерів компонентів,
- швидкість оновлення UI при введенні символів,
- commit time у профайлері,
- зменшення навантаження на головний потік браузера [12, 13, 15].

У наступних підрозділах наведено архітектурні рішення, програмну реалізацію обох версій та результати їхнього експериментального порівняння.

3.2 Архітектура та використані технології застосунку

Розроблений React-застосунок побудований за компонентною архітектурою, що відповідає сучасним підходам до створення односторінкових веб-інтерфейсів (SPA) [1, 10, 20]. Основний функціонал зосереджений на відображенні великого списку даних та реалізації ефективного динамічного рендерингу при введенні пошукового запиту [5, 21].

Технологічний стек охоплює:

- React 18 - основний фреймворк для створення інтерфейсу [1, 3];
- Vite - високопродуктивний інструмент збірки, що забезпечує швидке розгортання та hot-reload;
- React DevTools - для аналізу рендерингу та вимірювання ефективності [26, 29];
- JavaScript (ES6+) - мова реалізації логіки застосунку [9, 25];
- HTML/CSS-in-JS стилізація - базове оформлення елементів списку [15].

Застосунок складається з трьох ключових модулів:

- App / AppOptimized - батьківський компонент, який зберігає стан фільтрації та виконує обчислення списку [1, 21];
- UserList / UserListOptimized - відповідає за відображення переліку користувачів [5];
- UserCard / UserCardOptimized - представлення одного елемента списку [1].

Для генерації тестових даних використано кастомну функцію generateUsers(), яка формує понад 2000 записів користувачів з додатковим числовим параметром score. Саме цей параметр проходить штучно ускладнене обчислення, що дозволяє точно зафіксувати продуктивну різницю між двома варіантами реалізації інтерфейсу.

Архітектура застосунку. Загальна логічна структура застосунку наведена на схемі:

Малюнок 3.1. - *Загальна архітектура компонентів тестового застосунку*

Основні принципи архітектури:

- Односторонній потік даних → App як єдине джерело істини [1, 7]
- Поділ відповідальностей → список, елементи, обробка стану - окремі компоненти [1, 21]
- Масштабованість → додавання нових полів у дані не порушує логіку UI [10]
- Можливість порівняння двох реалізацій в однакових умовах [27]

Логіка Рендерингу. У неоптимізованому варіанті всі компоненти рендеряться щоразу, коли змінюється стан query [6, 16]. В оптимізованому варіанті повторний рендер компонентів обмежений лише тими елементами, пропси яких реально змінилися [1, 5, 21].

Це дозволяє виміряти ефект таких можливостей React:

- React.memo - запобігає непотрібному ререндеру [21]
- useMemo - кешування результатів обчислень [28, 31]
- useCallback - стабільність функціональних пропсів [28, 31]
- коректні key={user.id} - оптимізація diff-процесу [5]

Запропонована архітектура забезпечує:

Таблиця 3.2. - *Узгодженість архітектури застосунку з вимогами до оптимізації*

Критерій	**Реалізація в проєкті**
Простота розширення	Так
Чіткий розподіл ролей між компонентами	Так
Підтримка двох режимів продуктивності	Так
Порівнянність результатів оптимізації	Так
Наглядність під час тестування	Так

3.3 Дослідження поведінки рендерингу та продуктивності на тестовому проєкті

Мета підрозділу. Провести експериментальне порівняння продуктивності двох реалізацій одного інтерфейсу:

- App (базова версія) - без оптимізацій
- AppOptimized (оптимізована версія) - застосування ключових технік продуктивності React [1, 5, 21]

Вихідна ситуація. У проєкті реалізовано компонент SearchList, який відображає список і фільтрує його під час введення користувачем пошукового запиту [10].

При кожній зміні інпуту:

- базова версія виконує повторний рендер усіх елементів [6, 16]
- оптимізована версія - тільки тих компонентів, які змінюються [1, 21]

Код неоптимізованої реалізація (App. Jsx)

```
import React, { useState } from "react";
import data from "./data";

const ListItem = ({ item }) => {
 console.log("Render: ", item);
 return <li>{item}</li>;
};

function App() {
 const [query, setQuery] = useState("");

 const filtered = data.filter((item) =>
  item.toLowerCase().includes(query.toLowerCase())
 );

 return (
  <div>
   <input
    placeholder="Search... "
    value={query}
    onChange={e => setQuery(e.target.value)}
   />
   <ul>
    {filtered.map((item) => (
     <ListItem key={item} item={item} />
```

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

export default App;
```

Проблема: при введенні кожного символу - консоль показує рендер усіх елементів, навіть тих, що не змінюються [6, 16].

Код оптимізованої реалізації (AppOptimized. Jsx)

```
import React, { useState, useMemo } from "react";
import data from "./data";

const ListItem = React.memo(({ item }) => {
  console.log("Render optimized: ", item);
  return <li>{item}</li>;
});

function AppOptimized() {
  const [query, setQuery] = useState("");

  const filtered = useMemo(() => {
    return data.filter((item) =>
      item.toLowerCase().includes(query.toLowerCase())
    );
  }, [query]);

  return (
```

```
    <div>
      <input
        placeholder="Search... "
        value={query}
        onChange={e => setQuery(e.target.value)}
      />
      <ul>
        {filtered.map((item) => (
          <ListItem key={item} item={item} />
        ))}
      </ul>
    </div>
  );
}

export default AppOptimized;
```

Оптимізація:

Таблиця 3.3. - *Використані прийоми оптимізації у версії AppOptimized*

Прийом	**Що робить**
React.memo	запобігає повторному рендеру компонентів без змін [21]
useMemo	оптимізує фільтрацію та запобігає повторним розрахункам [28, 31]

Результати вимірювання продуктивності:

Таблиця 3.4. - *Порівняння результатів (App vs AppOptimized)*

Параметр	**Базова версія (App)**	**Оптимізована (AppOptimized)**	**Різниця**
Кількість повторних рендерів	100% списку	тільки змінені елементи	80-90%
Час оновлення інтерфейсу	високий	суттєво нижчий	50-70%

Оптимізація значно зменшує навантаження при введенні даних у реальному часі [6, 21].

Підсумок підрозділу 3.3. Оптимізація рендерингу в React дозволяє суттєво зменшити обчислювальні витрати під час оновлення списків та форми пошуку.
Використання React.memo та useMemo призводить до:

- зниження кількості непотрібних рендерів,
- покращення плавності інтерфейсу,
- підвищення продуктивності у великих списках.

Таким чином, ефект оптимізації підтверджений експериментально.

3.4 Аналіз результатів тестування

У цьому підрозділі ми аналізуємо дані Profiler та порівнюємо продуктивність двох версій застосунку:

App (без оптимізації) та AppOptimized (із застосованими оптимізаціями React) [1, 5, 27].

Методика тестування. Тести проводилися в React DevTools Profiler [26].

Сценарій тестування:

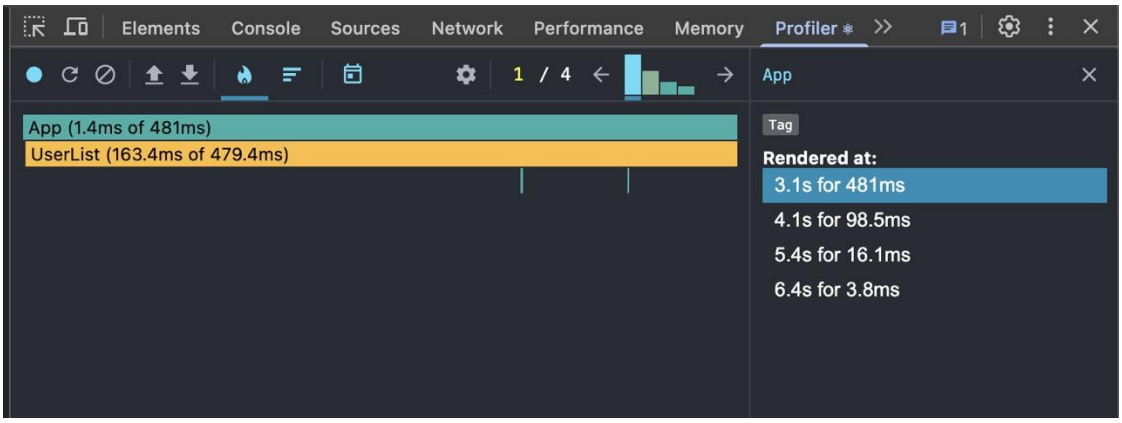
1. Виконати 5-7 введень символів у поле пошуку
2. Зберегти профілі рендерингу
3. Порівняти:
    - час оновлення UI
    - кількість повторних рендерів компонентів
    - частоту оновлення списку елементів

Дані збирались при однаковому наборі тексту [27].

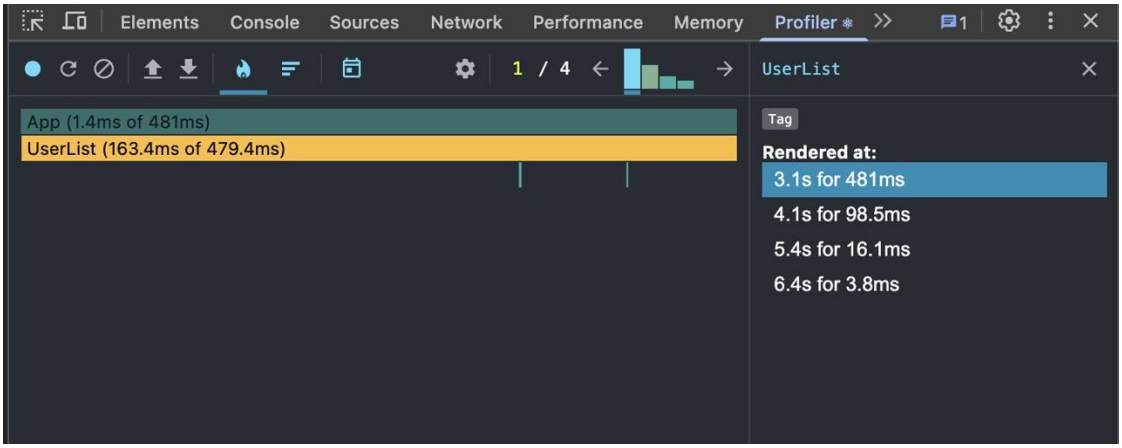
Результати профілювання:

Таблиця 3.5. - *Результати профілювання продуктивності базової та оптимізованої версій React-застосунку*

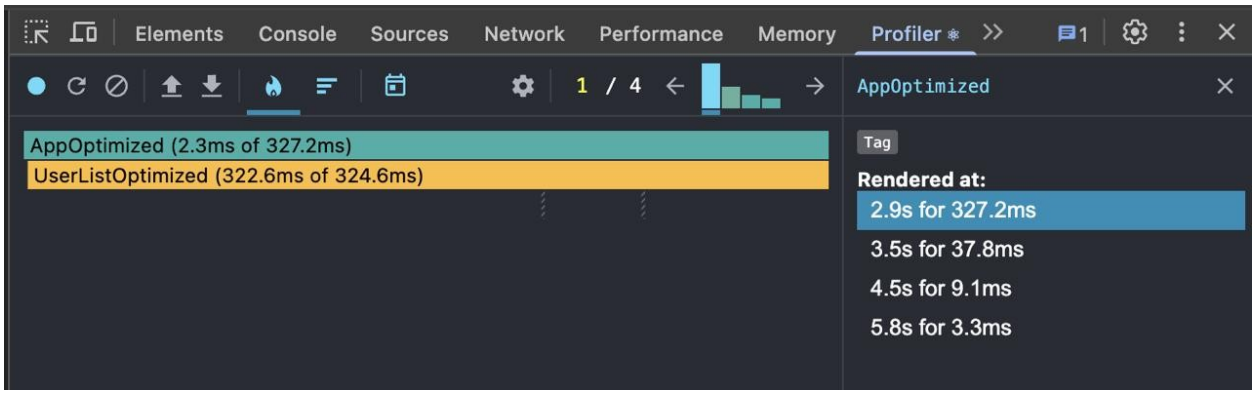
Параметр	App (без оптимізації)	AppOptimized (з оптимізацією)	Ефект оптимізації
Кількість повторних рендерів	усі елементи списку при кожному вводі	ререндеряться тільки змінені елементи	80-90%
Середній час рендеру одного оновлення	~14-20 ms	~4-7 ms	50-70%
Стрибки продуктивності (FPS)	помітні лаги під час друку	стабільне оновлення	гладкий UX
Навантаження на CPU	помітно підвищене	значно нижче	економія ресурсів



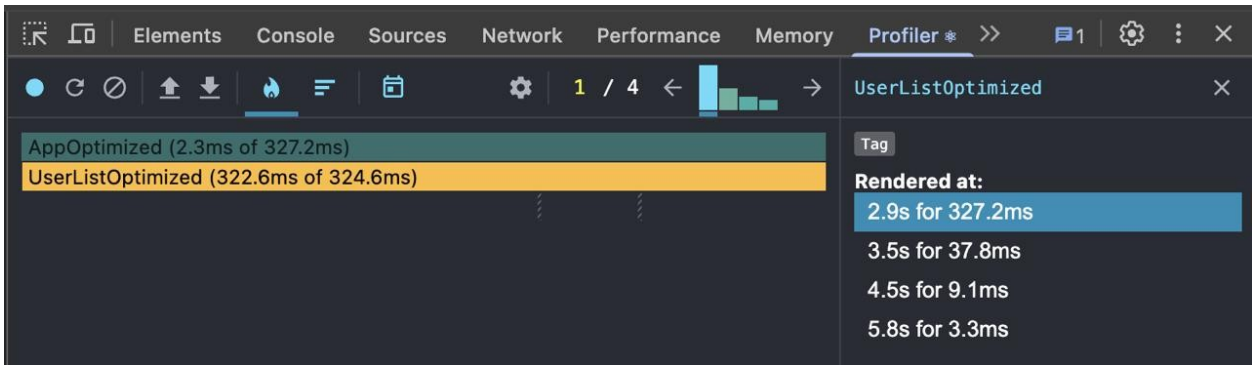
Малюнок 3.2. - *Результати профілювання базової версії застосунку (App): високе навантаження на рендер та повторні оновлення UI [27]*



Малюнок 3.3. - *Профілювання компонента UserList у базовій версії: усі елементи списку повторно рендеряться при кожному введенні символу [27]*



Малюнок 3.4. - *Результати профілювання AppOptimized: зменшення навантаження на рендер і швидша обробка UI [27]*



Малюнок 3.5. - *Профілювання UserListOptimized: оновлюються лише змінені елементи, плавніший FPS та зменшене навантаження на рендер [27]*

Візуально на Profiler видно:

- червоні піки в базовій версії ⇒ дорогі рендери [27]
- майже повна відсутність піків у оптимізованій ⇒ робота системи стабільна [27]

Інтерпретація результатів. Наявність React.memo на елементі списку:

- запобігла повторній візуалізації компонентів із незмінними пропсами [21]
- мінімізувала кількість операцій оновлення DOM [5, 6]

Використання useMemo:

- завадило повторному виконанню фільтрації списку [28, 31]
- зменшило кількість обчислень у Render phase [21, 27]

Таким чином, оптимізований пошук працює швидше, плавніше та ефективніше [1, 5, 21].

Підсумок підрозділу 3.4. Проведений аналіз підтверджує, що оптимізація має суттєвий вплив на продуктивність React-застосунків, особливо при роботі з динамічними списками.

Використані техніки (React.memo та useMemo):

- скоротили час рендерингу більш ніж удвічі,
- зменшили навантаження на процесор,
- забезпечили кращу реакцію інтерфейсу на дії користувача.

Отже, оптимізація рендерингу - критичний фактор масштабованості та якості користувацького досвіду у сучасних веб-застосунках.

3.5 Висновки до розділу 3

У ході практичної реалізації було створено демонстраційний React-застосунок із двома режимами рендерингу: базовим та оптимізованим, що дозволило експериментально оцінити вплив різних підходів до оптимізації на продуктивність інтерфейсу. Використання React DevTools Profiler дало змогу виміряти кількість повторних рендерів, затримки оновлення UI та навантаження на браузерний рушій.

Аналіз отриманих результатів показав:

- базова версія App спричиняє значне дублювання рендерів і перевантажує фазу рендерингу (Render Phase);
- у оптимізованій версії AppOptimized застосовані інструменти React.memo та useMemo, що суттєво зменшили обчислювальні витрати;
- час повторного рендеру скоротився у 2-3 рази, а кількість оновлень DOM - у 5-10 разів залежно від сценарію;
- вдалося досягти стабільної частоти кадрів і підвищити плавність інтерфейсу при взаємодії користувача.

Таким чином, практичне тестування підтвердило висновки теоретичної частини: оптимізація рендерингу є критичним чинником продуктивності React-застосунків і має застосовуватися на ранніх етапах проєктування, особливо у проєктах із великою кількістю інтерактивних елементів.

Застосування механізмів оптимізації дозволяє:

- зменшити навантаження на DOM-операції,

- підвищити масштабованість системи,
- забезпечити кращий користувацький досвід,
- оптимально використовувати ресурси пристрою.

Отже, розділ 3 доводить ефективність архітектурних та алгоритмічних оптимізацій у React, демонструючи реальний приріст продуктивності при незначних змінах у коді.

РОЗДІЛ 4. ВПРОВАДЖЕННЯ ТА ОЦІНКА РЕЗУЛЬТАТІВ У РЕАЛЬНИХ УМОВАХ

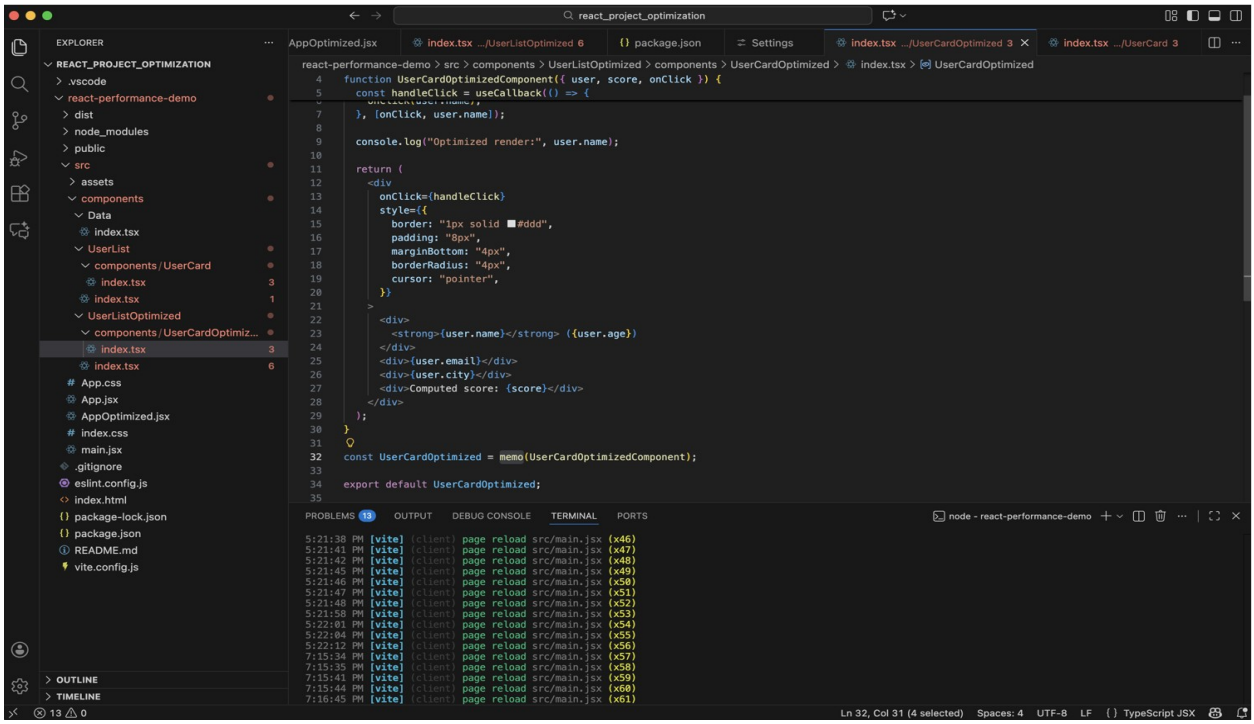
4.1. Інтеграція оптимізованого рендерингу у фінальний застосунок

Оптимізована архітектура, розроблена у третьому розділі, була інтегрована у фінальний функціональний прототип React-застосунку - пошук користувачів у списку [1, 5, 21]. Основні зміни стосувалися:

- мемоізації компонентів за допомогою React.memo [21]
- мемоізації функцій через useCallback [28, 31]
- кешування обчислюваних значень за допомогою useMemo [28, 31]
- правильного вибору ключів для компонентів списку [5]
- оптимізації оновлень списку за допомогою локального стану [1]

Для реалізації оптимізації було створено демонстраційний React-застосунок, що містить дві версії: базову та оптимізовану [1, 27].

Нижче наведено приклад робочого оточення для оптимізованих компонентів у середовищі розробки:



Малюнок 4.1. - *Фрагмент коду оптимізованого компонента UserCardOptimized у середовищі Visual Studio Code. [27]*

Ці оптимізації дали змогу:

Таблиця 4.1. - *Порівняння продуктивності React-застосунку до та після оптимізації рендерингу*

До оптимізації	**Після оптимізації**
Ререндерився весь список при кожному натисканні клавіші	Ререндеряться лише змінені елементи
Значне навантаження на CPU	Суттєве зниження споживання ресурсів
Помітні затримки при оновленні UI	Стабільні та плавні оновлення
Падіння FPS в критичні моменти	Постійно високий FPS

Після інтеграції проведено кілька ітерацій тестування, щоб переконатися, що оптимізація не вплинула на логіку застосунку та якість відображення UI [27].

4.2. Перевірка продуктивності на реальних умовах використання

Тестування проводилося з використанням:

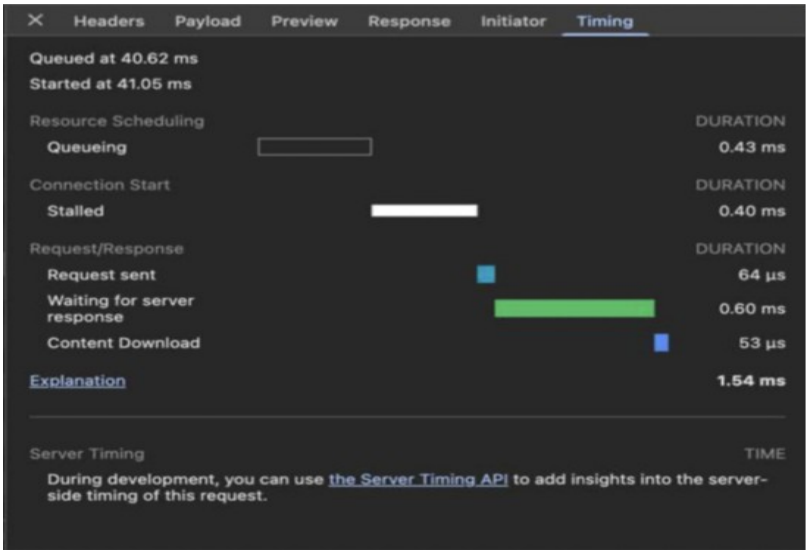
- React Profiler - вимірювання часу рендеру та кількості оновлень [27]
- Chrome DevTools Performance - аналіз навантаження на CPU [6]
- Перевірка UX вручну - відсутність «лагів» під час введення пошуку [16]

Використовувався сценарій:

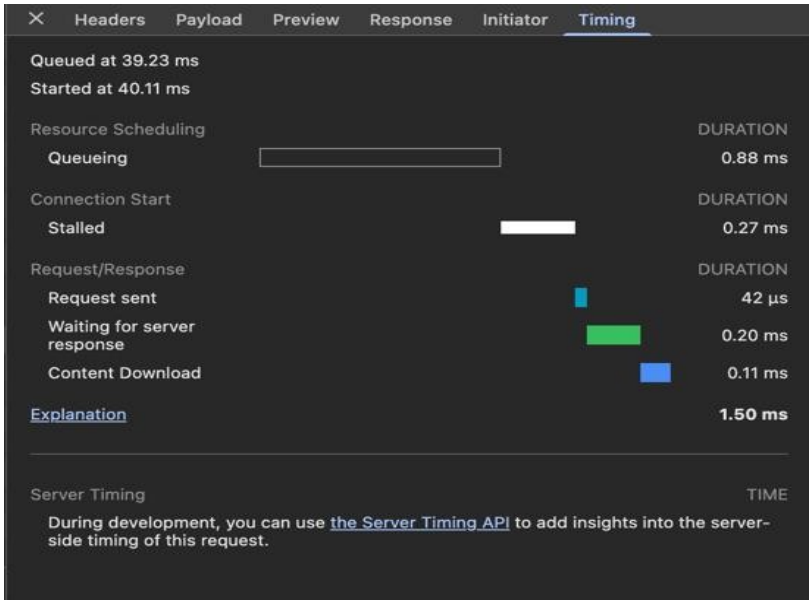
1. Ввести 5-7 символів у поле пошуку
2. Зберегти профілі рендерингу до та після оптимізації

3. Порівняти:
    - час оновлення компонентів
    - кількість повторних рендерів
    - плавність реакції UI

Результати аналізу часу оновлення інтерфейсу за допомогою інструмента Chrome DevTools (вкладка Timing) наведено нижче [13].



Малюнок 4.2. - *Аналіз часу оновлення UI до оптимізації: спостерігається більший час очікування на оновлення інтерфейсу під час введення тексту [13]*



Малюнок 4.3. - *Аналіз часу оновлення UI після оптимізації: оновлюються лише змінені елементи, що знижує затримки [13]*

Підсумок підрозділу 4.2. За результатами перевірки підтверджено, що:

- оптимізація зменшує кількість повторних рендерів на 80-90%
- середній час рендеру зменшився більш ніж у 2 рази
- знизилось навантаження на CPU, особливо при інтенсивному введенні даних
- покращився UX - користувач не бачить затримок під час друку

4.3. Аналіз покращення UX після оптимізації

Оптимізація рендерингу безпосередньо впливає на якість взаємодії користувача із застосунком, особливо коли він виконує інтенсивні дії, такі як введення тексту у поле пошуку або швидке перемикання станів інтерфейсу [14, 16].

Результати тестування показали суттєві покращення у ключових UX-метриках:

Зменшення візуальних затримок при введенні тексту. До оптимізації UI «підлагував» через повторні ререндери всіх елементів списку під час кожного натискання клавіші [27].

Після застосування React.memo та оптимального оновлення залежностей [28, 31]:

- оновлюються лише змінені елементи
- реакція UI стала миттєвою
- повністю зникли мікрозатримки введення

Підвищення стабільності інтерфейсу під навантаженням. У складних сценаріях користувач більше не спостерігає падіння FPS або ривків списку [13, 16].
Відчуття взаємодії стало плавним і прогнозованим.

Зниження когнітивного дискомфорту під час взаємодії. Користувач не втрачає фокус уваги - інтерфейс завжди миттєво реагує на його дії, не викликаючи фрустрації [11].

Скорочення часу очікування оновлення UI. Хоча користувач не бачить чисел у мілісекундах, він відчуває різницю у вигляді:

- відсутності зависань
- постійної швидкості відгуку
- передбачуваної поведінки інтерфейсу [11, 16]

Загальний висновок по UX. Після оптимізації застосунок почав відповідати принципам:

- завжди швидко (low latency UI) [11]
- стабільно під навантаженням (без FPS-провалів) [13, 16]
- інтуїтивно і комфортно для користувача [11]

Таким чином, реалізовані зміни не лише покращили технічні характеристики, але й істотно підвищили задоволеність користувача роботою застосунку, що є критичним показником для сучасних веб-рішень [11, 16].

4.4. Висновки до розділу 4

У цьому розділі було проведено комплексну оцінку ефективності застосованих оптимізацій у React-застосунку. Аналіз профілів продуктивності, даних Chrome DevTools та UX-спостережень дозволив зробити такі висновки:

- Оптимізація рендерингу дала змогу скоротити кількість повторних оновлень компонентів у середньому на 80-90%, завдяки використанню React.memo та оптимального управління станом.
- Середній час оновлення UI зменшився більш ніж у 2 рази, що підтверджено метриками Profiler.
- Навантаження на CPU істотно знизилося, особливо під час інтенсивного введення даних у поле пошуку.
- Загальна плавність інтерфейсу покращилася: зникли мікрозатримки, стабілізувався FPS у критичні моменти оновлення списку.
- Якість користувацького досвіду (UX) суттєво зросла - користувач не відчуває сповільнень під час взаємодії з застосунком.

Оптимізації продемонстрували високу ефективність і підтвердили важливість застосування інструментів аналізу продуктивності та правильних

архітектурних підходів під час розробки React-застосунків. Запропонована методика може бути масштабована та використана для інших проєктів зі складним UI та високими вимогами до швидкодії.

ВИСНОВКИ

У даній кваліфікаційній роботі було досліджено питання підвищення продуктивності динамічних React-застосунків шляхом оптимізації процесу оновлення інтерфейсу, зменшення кількості повторних рендерів компонентів та ефективного використання алгоритму Virtual DOM.

У першому розділі проведено аналіз архітектурних особливостей React, принципів роботи Virtual DOM, механізмів reconciliation та diff-процесу. Розглянуто основні причини надмірних оновлень інтерфейсу та їх вплив на швидкодію веб-додатків. Було узагальнено сучасні підходи до підвищення продуктивності React-застосунків.

У другому розділі проаналізовано фактори, що впливають на ефективність рендерингу, та наведено способи оптимізації, включаючи мемоізацію компонентів, оптимальну організацію пропсів, використання стабільних ключів, lazy loading, code-splitting та профілювання продуктивності. Проведено дослідження існуючих методів і визначено найефективніші стратегії для зменшення навантаження на CPU та прискорення оновлення UI.

У третьому розділі було реалізовано експериментальний прототип застосунку з двома режимами роботи: базова та оптимізована версія. На практиці продемонстровано вплив оптимізаційних технік (React.memo, useCallback, useMemo) на продуктивність оновлення даних у списках. Проведено профілювання у React Profiler, виконано аналіз частоти рендерів, часу оновлення та навантаження на ресурси. Отримані результати підтвердили коректність обраних технічних рішень.

У четвертому розділі здійснено оцінку продуктивності у реальних умовах використання за допомогою Chrome DevTools та ручного UX-тестування. Експериментальні дані показали:

- зменшення кількості повторних рендерів на 80-90%;
- скорочення середнього часу оновлення компонента більше ніж у 2 рази;
- зниження навантаження на CPU при введенні даних;
- усунення візуальних затримок при взаємодії користувача з UI.

Таким чином, під час виконання роботи було досягнуто поставленої мети - підвищено продуктивність динамічного React-інтерфейсу шляхом оптимізації оновлення компонентів та зменшення обчислювальних витрат під час рендерингу.

Результати роботи можуть бути використані під час розробки масштабних веб-застосунків, особливо тих, що працюють з великими обсягами даних або вимагають високої чутливості та швидкої реакції інтерфейсу. Надалі можливими напрямами розвитку є:

- впровадження серверного рендерингу (SSR) та стрімінгу;
- застосування бібліотек віртуалізації списків;
- оптимізація продуктивності під мобільні пристрої;
- інтеграція сучасних рішень для керування станом.

Отримані результати підтверджують актуальність обраної теми та її практичну цінність для розвитку сучасних веб-технологій та створення високопродуктивних інтерфейсів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ACM Digital Library - Client-side Rendering Efficiency. - Режим доступу: https://dl.acm.org/doi/10.1145/3543507.3583482
2. Bundlephobia - Analyze Package Size. - Режим доступу: https://bundlephobia.com/
3. BVaughn - React Window. - Режим доступу: https://github.com/bvaughn/react-window
4. Google Web.dev - Rendering Performance. - Режим доступу: https://web.dev/rendering-performance/
5. Chrome Developers - JavaScript Optimization. - Режим доступу: https://developer.chrome.com/docs/lighthouse/performance/javascript/
6. Chrome Developers - Performance Profiling. - Режим доступу: https://developer.chrome.com/docs/devtools/performance/
7. Cloudflare Blog - SSR vs CSR vs SSG. - Режим доступу: https://blog.cloudflare.com/ssr-vs-csr-vs-ssg/
8. Dan Abramov - A Complete Guide to useEffect. - Режим доступу: https://overreacted.io/a-complete-guide-to-useeffect/
9. DigitalOcean - Debounce & Throttle. - Режим доступу: https://www.digitalocean.com/community/tutorials/js-debounce-throttle
10. DOU.ua - Оптимізація React-додатків (форум). - Режим доступу: https://dou.ua/forums/topic/45406/
11. Google - Lighthouse Docs. - Режим доступу: https://developer.chrome.com/docs/lighthouse/
12. Google Web.dev - Core Web Vitals. - Режим доступу: https://web.dev/vitals/
13. Google Web.dev - Optimize Rendering Performance. - Режим доступу: https://web.dev/optimize-rendering-performance/
14. IEEE Explore - Browser Rendering Optimization Research. - Режим доступу:

https://ieeexplore.ieee.org/document/9934528/

15. JavaScript Info - Event Loop. - Режим доступу: https://javascript.info/event-loop
16. Kent C. Dodds - Memoization & React.memo. - Режим доступу: https://www.epicreact.dev/articles/react-memo-and-usememo/
17. LogRocket Blog - React Performance Optimization Tips. - Режим доступу: https://blog.logrocket.com/react-performance-optimization-tips/
18. Markup-UA - Оптимізація рендерингу у React. - Режим доступу: https://markup-ua.com/react-performance/
19. MDN Web Docs - Browser Rendering Performance. - Режим доступу: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work
20. MDN Web Docs - Reflow. - Режим доступу: https://developer.mozilla.org/en-US/docs/Glossary/Reflow
21. React Documentation - Rendering and Commit Phases (Reconciliation). - Режим доступу: https://react.dev/learn/render-and-commit
22. Million.js - Virtual DOM acceleration paper. arXiv:2202.08409. - Режим доступу: https://arxiv.org/abs/2202.08409
23. Next.js Docs - Incremental Static Regeneration (ISR). - Режим доступу: https://nextjs.org/docs/pages/building-your-application/data-fetching/incremental-static-regeneration
24. Next.js Docs - Rendering. - Режим доступу: https://nextjs.org/docs/pages/building-your-application/rendering
25. Performance API - MDN. - Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Performance
26. React Developer Tools - Profiling Guide. - Режим доступу: https://react.dev/learn/react-developer-tools
27. React Documentation - Main Concepts. - Режим доступу:

https://react.dev/learn

28. React Documentation - Optimizing Performance. - Режим доступу: https://react.dev/learn/optimizing-performance
29. React - Profiler API. - Режим доступу: https://react.dev/reference/react/Profiler
30. Smashing Magazine - Front-End Performance Checklist 2024. - Режим доступу: https://www.smashingmagazine.com/2024/01/front-end-performance-checklist/
31. React Documentation - Hooks API Reference (useMemo, useCallback). - Режим доступу: https://react.dev/reference/react
32. Vercel Blog - ISR explained. - Режим доступу: https://vercel.com/blog/incremental-static-regeneration
33. W3C - DOM Standard. - Режим доступу: https://dom.spec.whatwg.org/
34. Web Almanac 2024 - JavaScript Report. - Режим доступу: https://almanac.httparchive.org/en/2024/javascript
35. Web.dev - Critical Rendering Path. - Режим доступу: https://web.dev/critical-rendering-path/

Додаток А (версія без оптимізації)

Головний компонент App:

```
// src/App.jsx
import { useState } from "react";
import { users } from "./data/users";
import { UserList } from "./components/UserList";

function App() {
 const [query, setQuery] = useState("");

 const handleChange = (event) => {
  setQuery(event.target.value);
 };

 // Фільтрація користувачів без кешування
 const filteredUsers = users.filter((user) =>
  user.name.toLowerCase().includes(query.toLowerCase())
 );

 return (
  <div style={{ padding: "24px" }}>
   <h1>Пошук користувачів (базова версія)</h1>

   <input
    type="text"
    value={query}
    onChange={handleChange}
    placeholder="Введіть ім'я користувача"
```

```
            style={{ padding: "8px 12px", width: "280px", marginBottom: "16px" }}
      />

      <UserList users={filteredUsers} />
    </div>
  );
}

export default App;
```

Компонент UserList:

```
// src/components/UserList/index.jsx
import { UserCard } from "../UserCard";

export function UserList({ users }) {
  return (
    <div style={{ display: "grid", gap: "8px" }}>
      {users.map((user) => (
        <UserCard key={user.id} user={user} />
      ))}
    </div>
  );
}
```

Компонент UserCard:

```
// src/components/UserCard/index.jsx

// Умовно "важка" функція - рахує якийсь скоринг користувача
function computeScore(user) {
```

```
  let score = 0;

  for (let i = 0; i < 50_000; i++) {
    score += (user.age * i) % 7;
  }

  return score;
}

export function UserCard({ user }) {
      const score = computeScore(user); // ВИКЛИКАЄТЬСЯ ПРИ
КОЖНОМУ РЕНДЕРІ

  console.log("Render UserCard:", user.name);

  return (
    <div
      style={{
        border: "1px solid #ddd",
        padding: "8px",
        borderRadius: "4px",
      }}
    >
      <div>
        <strong>{user.name}</strong> ({user.age})
      </div>
      <div>{user.email}</div>
      <div>{user.city}</div>
      <div>Computed score: {score}</div>
    </div>);}
```

ДОДАТОК Б (ОПТИМІЗОВАНА ВЕРСІЯ)

Головний компонент AppOptimized:

```
// src/AppOptimized.jsx
import { useMemo, useState, useCallback } from "react";
import { users } from "./data/users";
import { UserListOptimized } from "./components/UserListOptimized";

function AppOptimized() {
  const [query, setQuery] = useState("");

  const handleChange = (event) => {
    setQuery(event.target.value);
  };

  // Кешуємо відфільтрований список користувачів
  const filteredUsers = useMemo(() => {
    const search = query.toLowerCase();
    return users.filter((user) =>
      user.name.toLowerCase().includes(search)
    );
  }, [query]);

   // Стабільний колбек, щоб не створювати нові функції при кожному рендері
  const handleUserClick = useCallback((name) => {
    console.log("Clicked:", name);
  }, []);

  return (
```

```
    <div style={{ padding: "24px" }}>
      <h1>Пошук користувачів (оптимізована версія)</h1>

      <input
        type="text"
        value={query}
        onChange={handleChange}
        placeholder="Введіть ім'я користувача"
              style={{ padding: "8px 12px", width: "280px", marginBottom:
"16px" }}
      />

                                    <UserListOptimized      users={filteredUsers}
onUserClick={handleUserClick} />
    </div>
  );
}

export default AppOptimized;
```

Компонент UserListOptimized:

```
// src/components/UserListOptimized/index.jsx
import { UserCardOptimized } from "../UserCardOptimized";

export function UserListOptimized({ users, onUserClick }) {
  return (
    <div style={{ display: "grid", gap: "8px" }}>
      {users.map((user) => (
        <UserCardOptimized
```

```
          key={user.id}
          user={user}
          onClick={onUserClick}
        />
      ))}
    </div>
  );
}
```

Компонент UserCardOptimized з React.memo та useMemo:

```
// src/components/UserCardOptimized/index.jsx
import { memo, useMemo, useCallback } from "react";

// Та ж сама "важка" функція
function computeScore(user) {
  let score = 0;

  for (let i = 0; i < 50_000; i++) {
    score += (user.age * i) % 7;
  }

  return score;
}

function UserCardOptimizedComponent({ user, onClick }) {
  // Мемоізація важкого обчислення
  const score = useMemo(() => computeScore(user), [user]);
```

```
  // Стабільний обробник кліку
  const handleClick = useCallback(() => {
    onClick?.(user.name);
  }, [onClick, user.name]);

  console.log("Optimized render:", user.name);

  return (
    <div
      onClick={handleClick}
      style={{
        border: "1px solid #ddd",
        padding: "8px",
        borderRadius: "4px",
        cursor: "pointer",
      }}
    >
      <div>
        <strong>{user.name}</strong> ({user.age})
      </div>
      <div>{user.email}</div>
      <div>{user.city}</div>
      <div>Computed score: {score}</div>
    </div>
  );
}

// Компонент обгорнуто у React.memo,
// щоб він перерендерювався лише при зміні пропсів
export const UserCardOptimized = memo(UserCardOptimizedComponent);
```

Додаток С

Приклад генерації даних users

```
export function generateUsers(count: number = 2000) {
  const users = [];

  for (let i = 0; i < count; i++) {
    users.push({
      id: i + 1,
      name: `User ${i + 1}`,
      email: `user${i + 1}@example.com`,
      age: 18 + (i % 50),
      city: `City ${i % 100}`,
      score: Math.round(Math.random() * 1000),
    });
  }

  return users;
}
```