**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**
**КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ**

# КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження методів оптимізації веб-застосунків, на базі технології React»

на здобуття освітнього ступеня магістра
зі спеціальності 126 Інформаційні системи та технології
*(код, найменування спеціальності)*
освітньо-професійної програми 126 Інформаційні системи та технології
*(назва)*

*Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело*

_____ Рогов Олександр
*(підпис)* *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав:
здобувач вищої освіти Рогов Олександр
група ІСДМ-62

Керівник: Каміла Сторчак
*науковий ступінь, вчене звання* к.т.н., професор

Рецензент:
*науковий ступінь, вчене звання* _____
Ім'я, ПРІЗВИЩЕ

**Київ 2023**

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

**Навчально-науковий інститут інформаційних технологій**

Кафедра Інженерії програмного забезпечення автоматизованих систем

Ступінь вищої освіти Магістр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

**ЗАТВЕРДЖУЮ**
Завідувач кафедрою ІПЗАС

_____ Каміла СТОРЧАК
«_____» _____ 2023 р.

**ЗАВДАННЯ**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Рогов Олександр Олександрович
*(прізвище, ім'я, по батькові здобувача)*

1. Тема кваліфікаційної роботи: Дослідження методів оптимізації веб-застосунків, на базі технології React
керівник кваліфікаційної роботи Каміла Сторчак КТН Професор,
*(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)*
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023р.

3. Вихідні дані до кваліфікаційної роботи:
Наукова-технічна література; Нормативні матеріали
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд React та його фреймворків;
2. Аналіз методів оптимізації React застосунків;
3. Практичне застосування методів оптимізації до React застосунку

5. Перелік ілюстративного матеріалу: *презентація*
6. Дата видачі завдання «19» жовтня 2023 р.

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Огляд React | 19.10-05.11.23 | Виконано |
| 2 | Огляд React фреймворків | 05.11-12.11.23 | Виконано |
| 3 | Можливості розробки React Native | 13.11-19.11.23 | Виконано |
| 4 | Життєвий цикл React | 20.11-25.11.23 | Виконано |
| 5 | React Хуки | 27.11-03.12.23 | Виконано |
| 6 | Методи оптимізації react застосунків | 04.12-10.12.23 | Виконано |
| 7 | Оптимізація продуктивності react-застосунків: підходи та інструменти | 11.12-20.12.23 | Виконано |
| 8 | Вступ, висновок, | 21.12-29.12.23 | Виконано |

Здобувач(ка) вищої освіти _____ Олександр Рогов
*(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Каміла Сторчак
*(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

# РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 81 стор., 22 рис., 50 джерел.

*Мета роботи* – дослідження методів оптимізації веб-застосунків, на базі технології React.

*Об'єкт дослідження* – React-застосунки, зосереджені на виявленні й реалізації оптимальних методів для вирішення проблем, пов'язаних із продуктивністю.

*Предмет дослідження* – конкретні методи, бібліотеки та підходи, які дозволяють забезпечити високу ефективність та швидкодію React-застосунків.

*Короткий зміст роботи*: Дипломна робота присвячена оптимізації React-застосунків через використання хуків, таких як `useCallback` та `useMemo`, а також врахування властивості ключа при роботі з компонентами списків. Робота надає конкретні поради для поліпшення продуктивності та користувацького досвіду у розробці React-застосунків.

КЛЮЧОВІ СЛОВА: REACT ОПТИМІЗАЦІЯ, ПРОДУКТИВНІСТЬ REACT-ЗАСТОСУНКІВ, ХУКИ REACT

## ABSTRACT

Text part of the master's qualification work:
81 pages, 22 pictures, 0 table, 50 sources.

*The purpose of the work* - explore methods for optimizing web applications built with React technology.

*Object of research* – React applications, focusing on identifying and implementing optimal methods to address performance-related challenges..

*Subject of research* – Specific methods, libraries, and approaches that enable achieving high efficiency and speed in React applications.

*Summary of the work* - This thesis delves into the optimization of React applications by leveraging hooks such as useCallback and useMemo, and emphasizes the importance of considering the key property when working with list components. The work provides practical advice for enhancing productivity and user experience in the development of React applications.



KEYWORDS: REACT OPTIMIZATION, PRODUCTIVITY OF REACT APPLICATIONS, REACT HOOKS

ЗМІСТ

# ВСТУП

В сучасному світі інформаційних технологій, коли вимоги користувачів до продуктивності та швидкодії застосунків несприйнятливо зростають, оптимізація є ключовим аспектом розробки програмного забезпечення. Оптимізовані застосунки не лише забезпечують користувачам комфортну роботу, але й сприяють зниженню навантаження на сервери та покращенню ефективності використання ресурсів.

Ця дипломна робота присвячена вивченню та розгляду питань оптимізації React-застосунків — технології, яка стала однією з основних у веб-розробці. Реалізація оптимальної продуктивності та швидкодії великих та складних застосунків на React має вирішальне значення для стабільності та задоволення користувачів.

В процесі дослідження ми докладно розглянули ключові інструменти та підходи, які надає React для оптимізації, такі як `React.memo`, `PureComponent`, `shouldComponentUpdate`, `useCallback`, та `useMemo`. Посутньою частиною дослідження була аналіз використання ключів у React-застосунках та їх вплив на продуктивність.

Це дослідження не лише присвячене розумінню технічних аспектів оптимізації React-застосунків, але й має на меті забезпечити розробникам практичні інструменти та знання для поліпшення продуктивності своїх проектів. Важливість оптимізації застосунків у світі високих технологій важко переоцінити, і ця робота надає конкретні рекомендації та приклади, які можна використовувати в реальних розробках.

Дослідження надає розгорнуті висновки, які дозволяють узагальнити та застосувати отримані знання в практиці. Його результати можуть бути особливо корисними для розробників, які стикаються з розробкою великих та розширених React-застосунків, де оптимізація відіграє важливу роль у забезпеченні стабільної та швидкої роботи застосунків.

# 1 ОГЛЯД JAVASCRIPT ТА ФРЕЙМВОРКІВ

[1] JavaScript - це мова сценаріїв або програмування, яка дозволяє реалізувати складні функції на веб-сторінках - кожного разу, коли веб-сторінка відображає не статичну інформацію: оновлення контенту, інтерактивні карти, анімовані 2D/3D графіку, прокручувані відео і т.д. Це третій рівень технологій стандартної веб-структури, два з яких (HTML та CSS).

[2] Мова JavaScript була створена Бренданом Айхом у 1995 році для Netscape 2. У 1996 році Netscape і Брендан Айх передали JavaScript до міжнародної організації стандартів ECMA, для розробки мови було створено технічний комітет (TC39). ECMA-262 Edition 1 було випущено в червні 1997 року. Коли комітет TC39 зібрався в Осло в 2008 році, щоб узгодити ECMAScript 4, вони розділилися на 2 дуже різні табори:

Табір ECMAScript 3.1: Microsoft і Yahoo, які хотіли отримати поетапне оновлення з ES3. Та табір ECMAScript 4: Adobe, Mozilla, Opera та Google, які хотіли масштабного оновлення ES4. 13 серпня 2008 року Брендан Айх написав в електроній пошті:

*Ні для кого не секрет, що орган стандартів JavaScript, Технічний відділ Ecma Комітет 39 був розколотий більше року, з деякими членами перевагу ES4, основного четвертого видання ECMA-262 та інших пропагування ES3.1 на основі існуючого ECMA-262 Edition 3 (ES3) специфікація. Тепер, я радий повідомити, що розлучення закінчилося.* Рішенням було працювати разом:

- ECMAScript 4 було перейменовано на ES5
- ES5 має бути поступовим оновленням ECMAScript 3.
- Функції ECMAScript 4 слід використовувати в пізніших версіях.
- TC39 має розробити новий основний випуск, більший за обсягом, ніж ES5.

Запланований новий випуск (ES6) отримав кодову назву "Harmony" (Через розкол, який він створив?). ES5 мав величезний успіх. Він був випущений у 2009 році, і всі основні браузери (включно з Internet Explorer) були переведені на цю версію до липня 2013 року:

| | | | | |
|---|---|---|---|---|
| Chrome 23 | IE10 / Edge | Firefox 21 | Safari 6 | Opera 15 |
| Nov 2012 | Sep 2012 | May 2013 | Jul 2012 | Jul 2013 |

Рисунок 1.1 Підтримка браузерами JavaScript

Версія ES6 також мала також величезний успіх. Вона була випущена у 2015 році, і всі основні браузери були повністю сумісні до березня 2017 року:

| | | | | |
|---|---|---|---|---|
| Chrome 51 | Edge 14 | Firefox 52 | Safari 10 | Opera 38 |
| May 2016 | Aug 2016 | Mar 2017 | Sep 2016 | Jun 2016 |

Рисунок 1.2 Підтримка браузерами JavaScript

*Використання JavaScript*

Початкові версії мови сценаріїв були лише для внутрішнього використання. Після того, як Netscape подав його до ECMA International як стандартну специфікацію для веб-браузерів, JavaScript став піонером у випуску ECMAScript. Це була мова сценаріїв загального призначення для забезпечення взаємодії веб-сторінок у різних браузерах і пристроях.

JavaScript продовжує розвиватися разом із веб-переглядачами, такими як Firefox та Chrome відтоді. Останній навіть розпочав розробку першого сучасного двигуна JavaScript під назвою V8, який компілює байт-код у рідний машинний код. Сьогодні JavaScript має багато фреймворків і бібліотек для спрощення складних проектів, таких як AngularJS, jQuery та ReactJS.

Спочатку реалізація JavaScript запускалася на стороні клієнта, але після розробки Node.js була розгорнута на стороні сервера в крос-платформному серверному середовищі, створене на механізмі Google Chrome JavaScript V8.

Хоча V8 найбільше підходить для веб-програм, функції програмування JavaScript мають інші реалізації в різних областях. Нижче наведено кілька основних способів використання JavaScript.

*Веб і мобільні програми*. Розробка фреймворків JavaScript, що складаються з бібліотек коду JavaScript, дозволяє розробникам використовувати попередньо написаний код JavaScript у своїх проектах. Це економить їх час і зусилля від необхідності кодувати функції програмування з нуля.

Кожен фреймворк JavaScript має функції, спрямовані на спрощення процесу розробки та налагодження. Наприклад, зовнішні фреймворки JavaScript, такі як jQuery та ReactJS, покращують ефективність проектування. Вони дозволяють розробникам повторно використовувати й оновлювати компоненти коду, не впливаючи один на одного, функціонуючи чи цінно.

Тим часом, фреймворки розробки програм для мобільних пристроїв, такі як Cordova та Titanium дозволяють створювати власні або гібридні програми.

Реалізація коду JavaScript у Node.js також відіграє важливу роль у веб-розробці. Node.js може скоротити час відповіді сервера завдяки своїй однопотоковій природі та неблокуючій архітектурі та усунути затримки.

Node.js також достатньо легкий, щоб служити масштабованим інструментом для мікросервісів, дозволяючи вам розробляти єдину програму, що складається з невеликих служб з окремими процесами.

*Створення веб-серверів і серверних програм*. Завдяки Node.js JavaScript дозволяє розробникам створювати веб-сервери та серверну інфраструктуру, заощаджуючи час і зусилля на створення веб-сервера.

Вбудований модуль HTTP дозволяє розробити базовий HTTP-сервер, який відображає звичайний текст під час доступу користувачів до веб-сторінки. Ви можете використовувати власний веб-сервер Node.js, Node-OS,

або сторонні сервери, такі як Microsoft Internet Information Services (IIS) і для обробки запитів HTTP.Apache

*Інтерактивна поведінка на веб-сайтах*. Однією з основних функцій JavaScript є додання динамічності веб-сторінкам. Це включає відображення анімації, зміну видимості тексту та створення спадних меню.

Хоча для створення веб-сайту можна використовувати лише код HTML і CSS, він матиме лише статичне відображення. За допомогою JavaScript користувач може взаємодіяти з веб-сторінками та мати кращий досвід перегляду.

Крім того, JavaScript дозволяє змінювати вміст HTML і значення атрибутів без попереднього перезавантаження веб-сторінки.

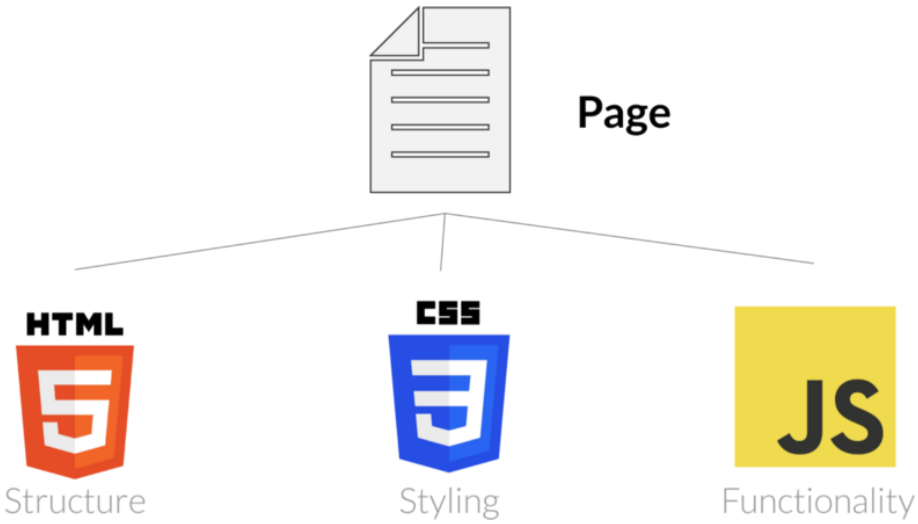



Рисунок 1.3 Використання стеку JavaScript

*Розробка гри*. JavaScript може допомогти створити гру, якщо використовувати його з HTML5 та інтерфейсом програмування додатків (API), як-от WebGL. Існує Багато ігрових механізмів на основі JavaScript, як-от Phaser, GDevelop та Kiwi.js Приклади ігр створених на JavaScript Angry Birds, The Wizard та 2048.

*Що робить JavaScript чудовим?*

[3] JavaScript має низку переваг, які роблять його кращим вибором, ніж його конкуренти. Нижче наведено кілька переваг використання JavaScript:

- Простота – завдяки простій структурі JavaScript легше вивчати та застосовувати, а також він працює швидше, ніж деякі інші мови. Помилки також легко виявити та виправити.
- Швидкість – JavaScript виконує сценарії безпосередньо у веб-переглядачі без попереднього підключення до сервера чи компілятора. Крім того, більшість основних браузерів дозволяють JavaScript компілювати код під час виконання програми.
- Універсальність – JavaScript сумісний з іншими мовами, такими як PHP, Perl і Java. Це також робить науку про дані та машинне навчання доступними для розробників.
- Популярність – доступно багато ресурсів і форумів, які допоможуть новачкам із обмеженими технічними навичками та знаннями JavaScript.
- Навантаження на сервер – ще одна перевага роботи на стороні клієнта полягає в тому, що JavaScript зменшує кількість запитів, надісланих на сервер. Перевірку даних можна виконати за допомогою веб-переглядача, а оновлення стосуються лише певних розділів веб-сторінки.
- Оновлення – Команда розробників JavaScript і ECMA International постійно оновлюють і створюють нові фреймворки та бібліотеки, забезпечуючи їх актуальність у галузі.

*Які недоліки JavaScript?*

Як і будь-яка інша мова програмування, JavaScript має кілька обмежень, які потрібно враховувати. Нижче наведено деякі з недоліків використання JavaScript:

- Сумісність веб-переглядача – різні веб-браузери по-різному інтерпретують код JavaScript, що спричиняє неузгодженість. Тому вам слід протестувати свій сценарій JavaScript у всіх популярних веб-браузерах, включно з їхніми старими версіями, щоб уникнути шкоди для взаємодії з користувачем.

- Безпека – код JavaScript, який виконується на стороні клієнта, вразливий до використання безвідповідальними користувачами.
- Налагодження – хоча деякі редактори HTML підтримують налагодження, вони менш ефективні, ніж інші редактори. Оскільки веб-переглядачі не показують жодних попереджень про помилки, виявити проблему може бути складно.

## 1.1 Фреймворки JavaScript

[4] Фреймворки JavaScript є важливою частиною сучасної зовнішньої веб-розробки, надаючи розробникам перевірені інструменти для створення масштабованих інтерактивних веб-додатків. Багато сучасних компаній використовують фреймворки як стандартну частину свого інструментарію, тому для багатьох завдань із розробки інтерфейсу необхідний досвід роботи з фреймворками. У цьому наборі статей ми прагнемо дати вам зручну відправну точку, щоб допомогти вам розпочати вивчення фреймворків. Фреймворк

*JavaScript* — це набір попередньо написаного коду для багаторазового використання, який служить шаблоном або планом, щоб допомогти розробникам створювати веб-додатки ефективніше. Фреймворки розроблені для створення та керування веб-додатками за допомогою JavaScript.

Вони забезпечують структурований спосіб організації коду та містять вбудовані функції та інструменти для типових завдань, що означає, що можна зосередитися на унікальних аспектах свого проекту замість того, щоб починати з нуля. По суті, вони спрощують процес веб-розробки, забезпечуючи швидше, послідовніше та безпомилкове кодування за допомогою JavaScript.

*Основні характеристики фреймворків JavaScript:*

*Багаторазове використання коду:* Фреймворки надають готовий код, який можна використовувати повторно для різних частин проекту. Це сприяє підвищенню швидкості розробки та забезпечує консистентність.

*Шаблони та плани:* Вони включають в себе шаблони та плани, які служать основою для створення веб-додатків. Це робить розробку більш прогнозованою та допомагає уникнути великої кількості повторюваного коду.

Структура коду: Фреймворки надають чітку структуру для організації коду, що полегшує управління та розвиток великих проектів. Це забезпечує легшу читабельність та обслуговуваність.

*Вбудовані функції та інструменти:* Містять у собі вбудовані функції та інструменти для розв'язання типових задач, таких як маршрутизація, керування станом додатку, взаємодія з сервером тощо.

*Швидше та безпомилкове кодування:* Забезпечують швидший розвиток проекту завдяки готовим рішенням та стандартизованому підходу. Це дозволяє уникнути багатьох помилок та прискорює вирішення завдань.

*Angular* — це потужна структура веб-додатків із відкритим вихідним кодом, яку очолює команда Angular у Google і спільнота розробників-ентузіастів. Цей популярний фреймворк JavaScript є комплексним рішенням для створення ефективних і складних односторінкових програм (SPA).

Angular.js виділяється багатим набором функцій, таких як двостороннє зв’язування даних, модульна структура розробки, обробка форм, маршрутизація та впровадження залежностей, які спрощують процес розробки та підвищують продуктивність.

На відміну від бібліотек, які зосереджені виключно на рівні перегляду, Angular надає повну структуру, включаючи інструменти для тестування, розробки та розгортання програм. Він також використовує TypeScript для забезпечення узгодженості коду, покращеної читабельності та надійної перевірки типу. Компонентна архітектура Angular дозволяє багаторазово

використовувати код і ефективні практики розробки. Цей фреймворк особливо популярний у додатках корпоративного масштабу завдяки його масштабованості, зручності обслуговування та сильному акценту на тестуванні та найкращих практиках кодування.

*Vue.js* відомий своєю простотою та гнучкістю, і його часто використовують для створення інтерфейсів користувача та односторінкових програм.

Створений для поступового впровадження, розробники можуть інтегрувати Vue у свої існуючі проекти по частинах або використовувати його для створення складних програм з нуля. Основна бібліотека Vue зосереджена на рівні перегляду, що полегшує підбір та інтеграцію з іншими бібліотеками чи існуючими проектами. Це також добре підходить для складних SPA у поєднанні з допоміжними бібліотеками. Його компоненти зв’язування даних і реактивні компоненти пропонують ефективний і простий спосіб керування станом програми та створення динамічних інтерфейсів користувача.

Екосистема Vue включає Vue Router для маршрутизації, Vuex для управління станом і Vue CLI для проекту, що робить його комплексним рішенням для розробників, які шукають баланс між продуктивністю, гнучкістю та простотою використання.

*Node.js*. Будучи потужним міжплатформним середовищем виконання з відкритим кодом, Node.js дозволяє розробникам виконувати код JavaScript на стороні сервера.

Створений за допомогою двигуна Chrome V8 JS, Node дозволяє розробляти швидкі та масштабовані мережеві програми. Він також добре відомий своєю неблокуючою архітектурою, керованою подіями, що робить його дуже ефективним із програмами, що інтенсивно обробляють дані, і програмами в реальному часі на розподілених пристроях.

Це зробило Node.js популярним для розробки широкого спектру додатків, від веб-додатків до RESTful API і навіть для пристроїв IoT. Якщо це передбачає рендеринг на стороні сервера, Node ідеально підходить.

*Ember.js* — це надійна платформа JavaScript із відкритим вихідним кодом, розроблена для створення амбітних веб-додатків.

Він відомий своєю "конвенцією над конфігурацією" Філософія, яка оптимізує розробку, надаючи набір типових поведінок і найкращих практик, зменшуючи кількість шаблонного коду, який мають писати розробники.

Ember також має потужний механізм створення шаблонів і автоматично оновлює DOM при зміні основних даних, що робить його чудовим для розробки складних інтерактивних інтерфейсів користувача.

Також приємно, що він включає вбудований маршрутизатор, керування станом і Ember CLI для генерації проектів, створення скелетів, розробки та інструментів конвеєрної побудови. Ця комплексна екосистема робить Ember.js особливо придатним для великомасштабних, багатофункціональних веб-додатків, де зручність обслуговування та продуктивність є критичними.

*Meteor* — це скоріше повноцінна платформа JavaScript, ніж фреймворк JavaScript, оскільки він включає як клієнтські, так і серверні компоненти.

Це означає, що він надає повний набір інструментів для створення та розгортання веб- та мобільних додатків.

Деякі з його видатних функцій включають повну інтеграцію з MongoDB і його здатність поширювати зміни даних клієнтам без необхідності писати код синхронізації. Це робить Meteor дуже ефективним для створення додатків у режимі реального часу, таких як програми для чату або інтерфейси з живим оновленням.

Він також пропонує багатий набір функцій, включаючи гаряче перезавантаження коду, миттєве оновлення веб-сторінки та єдине мовне середовище розробки, що дозволяє розробникам використовувати JavaScript як на стороні клієнта, так і на стороні сервера.

Екосистема Meteor також включає широкий набір пакетів та інструментів, що спрощує процес розробки від ініціації проекту до розгортання. Він також може інтегруватися з різними популярними зовнішніми фреймворками та бібліотеками JS, такими як React, Angular і Vue.

## 1.2 Вибір фреймворку React

Сьогодні інтерфейсні фреймворки та бібліотеки стають невід'ємною частиною сучасної веб-розробки. React.js — це інтерфейсна бібліотека, яка поступово стала основним фреймворком для сучасної веб-розробки в спільноті JavaScript.

[5] React.js — це бібліотека JavaScript з відкритим вихідним кодом, ретельно розроблена Facebook, яка спрямована на спрощення складного процесу створення інтерактивних інтерфейсів користувача. Уявіть собі інтерфейс користувача, створений за допомогою React, як набір компонентів, кожен з яких відповідає за виведення невеликого багаторазового фрагмента HTML-коду.

У наш час світ не може жити без мобільних і веб-додатків. Все оцифровано, від бронювання таксі до замовлення їжі для здійснення банківських операцій. Завдяки ефективним структурам, які забезпечують безперебійну роботу користувача. Однією з таких надійних інтерфейсних бібліотек є React. Цей підручник про «що таке React» допоможе вам зрозуміти основи бібліотеки та працювати з простою демонстрацією.

React — це структура, яка використовує Webpack для автоматичної компіляції коду React, JSX і ES6 під час обробки префіксів файлів CSS. React — це бібліотека розробки інтерфейсу користувача на основі JavaScript. Хоча

React є бібліотекою, а не мовою, він широко використовується у веб-розробці. Бібліотека вперше з’явилася в травні 2013 року і зараз є однією з найбільш часто використовуваних інтерфейсних бібліотек для веб-розробки.

React пропонує різні розширення для підтримки всієї архітектури програми, наприклад Flux і React Native, крім простого інтерфейсу користувача.

У порівнянні з іншими технологіями на ринку, React є новою технологією. Джордан Волке, інженер-програміст Facebook, заснував бібліотеку в 2011 році, давши їй життя. Подібні до XHP, простому фреймворку HTML-компонентів для PHP, впливають на React. Новинна стрічка React була його дебютною програмою в 2011 році. Пізніше Instagram підбирає її та додає до своєї платформи.

Популярність React сьогодні затьмарила популярність усіх інших інтерфейсних фреймворків розробки. Ось чому:

[6] *Просте створення динамічних додатків*: React полегшує створення динамічних веб-додатків, оскільки вимагає менше кодування та пропонує більше функціональних можливостей, на відміну від JavaScript, де кодування часто стає складним дуже швидко.

*Покращена продуктивність*: React використовує Virtual DOM, завдяки чому швидше створюються веб-додатки. Віртуальний DOM порівнює попередні стани компонентів і оновлює лише ті елементи в реальному DOM, які були змінені, замість того, щоб оновлювати всі компоненти знову, як це роблять звичайні веб-програми.

*Багаторазові компоненти*: компоненти є будівельними блоками будь-якої програми React, і одна програма зазвичай складається з кількох компонентів. Ці компоненти мають свою логіку та елементи керування, і їх можна повторно використовувати в усій програмі, що, у свою чергу, значно скорочує час розробки програми.

*Односпрямований потік даних*: React слідує за односпрямованим потоком даних. Це означає, що під час розробки програми React розробники часто вкладають дочірні компоненти в батьківські. Оскільки дані передаються в одному напрямку, стає легше виправляти помилки та знати, де виникає проблема в програмі в даний момент.

*Невелика крива навчання:* React легко освоїти, оскільки він здебільшого поєднує базові концепції HTML та JavaScript з деякими корисними доповненнями. Проте, як і у випадку з іншими інструментами та фреймворками, вам доведеться витратити деякий час, щоб правильно зрозуміти бібліотеку React.

Його можна використовувати для розробки як веб, так і мобільних програм: ми вже знаємо, що React використовується для розробки веб-програм, але це ще не все, що він може зробити. Існує фреймворк під назвою React Native, похідний від самого React, який надзвичайно популярний і використовується для створення красивих мобільних додатків. Отже, насправді React можна використовувати для створення як веб, так і мобільних програм.

*Спеціальні інструменти* для легкого налагодження: Facebook випустив розширення Chrome, яке можна використовувати для налагодження програм React. Це робить процес налагодження веб-додатків React швидшим і простішим.

У React ви розробляєте свої програми, створюючи повторно використовувані компоненти, які можна розглядати як незалежні блоки Lego. Ці компоненти є окремими частинами кінцевого інтерфейсу, які, будучи зібраними, утворюють весь інтерфейс користувача програми.

Основна роль React у додатку полягає в тому, щоб обробляти рівень перегляду цього додатка так само, як V у шаблоні «model-view-controller» (MVC), забезпечуючи найкраще та найефективніше виконання візуалізації. Замість того, щоб мати справу з усім інтерфейсом користувача як єдиним

блоком, React.js заохочує розробників розділити ці складні інтерфейси користувача на окремі багаторазово використовувані компоненти, які утворюють будівельні блоки цілого інтерфейсу користувача. При цьому фреймворк ReactJS поєднує швидкість і ефективність JavaScript з більш ефективним методом маніпулювання DOM для швидшого відтворення веб-сторінок і створення високодинамічних і адаптивних веб-додатків.

[7] Ще в 2011 році Facebook мав величезну базу користувачів і зіткнувся з непростим завданням. Він хотів запропонувати користувачам більш багатий досвід користувача, створивши більш динамічний і більш чуйний інтерфейс користувача, який був швидким і високопродуктивним.

Джордан Волке, один із інженерів програмного забезпечення Facebook, створив React саме для цього. React спростив процес розробки, забезпечивши більш організований і структурований спосіб побудови динамічних та інтерактивних інтерфейсів користувача з компонентами, які можна використовувати повторно.

Першою його використала стрічка новин Facebook. Завдяки своєму революційному підходу до маніпулювання DOM та користувальницькими інтерфейсами React кардинально змінив підхід Facebook до веб-розробки та швидко став популярним в екосистемі JavaScript після випуску для спільноти з відкритим кодом.

*Як працює React.js*

Зазвичай коли найсилається запит на веб-сторінку, ввівши її URL-адресу у веб-переглядач,  він надсилає запит на сторінку, яку відображає. Якщо користувач клацнете на посилання на веб-сторінці, щоб перейти на іншу, на сервер надсилається новий запит для отримання цієї нової сторінки.

Такі переходи відбуваються щоразу для кожної нової сторінки чи ресурсу, до якого користувач намагаєтеся отримати доступ. Такий підхід до завантаження веб-сайтів працює чудово, але розгляньте веб-сайт, який дуже

повільно завантажує данні. Попереднє завантаження повної веб-сторінки було б зайвим і створювало б погану взаємодію з користувачем.

Крім того, коли дані змінюються в традиційній програмі JavaScript, для відображення цих змін потрібно ручне маніпулювання DOM. Ви повинні визначити, які дані змінилися, і оновити DOM, щоб відобразити ці зміни, що призведе до повного перезавантаження сторінки.

React використовує інший підхід, дозволяючи створювати так звану односторінкову програму (SPA). Односторінкова програма завантажує лише один документ HTML за першим запитом. Потім він оновлює певну частину, вміст або тіло веб-сторінки, які потребують оновлення, за допомогою JavaScript.

Цей шаблон відомий як маршрутизація на стороні клієнта, оскільки клієнту не потрібно перезавантажувати повну веб-сторінку, щоб отримати нову сторінку кожного разу, коли користувач робить новий запит. Натомість React перехоплює запит і лише отримує та змінює ті розділи, які потрібно змінити, без необхідності ініціювати повне перезавантаження сторінки. Цей підхід забезпечує кращу продуктивність і більш динамічний досвід користувача.

React покладається на віртуальний DOM, який є копією фактичного DOM. Віртуальний DOM React негайно перезавантажується, щоб відобразити цю нову зміну щоразу, коли відбувається зміна стану даних. Після цього React порівнює віртуальний DOM із фактичним DOM, щоб зрозуміти, що саме змінилося.

Після цього React знаходить найлегший спосіб виправити фактичний DOM за допомогою цього оновлення без відтворення фактичного DOM. Як наслідок, компоненти та інтерфейс React дуже швидко відображають зміни, оскільки вам не потрібно перезавантажувати всю сторінку кожного разу, коли щось оновлюється.

## 1.3 Можливості розробки React Native

[8] React Native (також відомий як RN) — це популярний фреймворк мобільних програм на основі JavaScript, який дозволяє створювати нативно відтворені застосунки. Фрейморк має ту саму кодову базу, що і React. Фреймворк дозволяє створювати програми для iOS та Android

React Native був уперше випущений Facebook як проект із відкритим вихідним кодом у 2015 році. Лише за пару років він став одним із найкращих рішень для мобільної розробки. Розробка React Native використовується для роботи Instagram, Facebook і Skype. Є кілька причин глобального успіху React Native.

По-перше, використовуючи React Native, компанії можуть створити код лише один раз і використовувати його для роботи своїх додатків для iOS і Android. Це означає величезну економію часу та ресурсів.

По-друге, React Native було створено на основі React – бібліотеки JavaScript, яка вже була надзвичайно популярною, коли було випущено мобільний фреймворк. По-третє, фреймворк дав змогу розробникам зовнішніх інтерфейсів, які раніше могли працювати лише з веб-технологіями, створювати надійні, готові до використання програми для мобільних платформ.

Коли Facebook уперше вирішив зробити свою послугу доступною на мобільних пристроях, замість розробляючи рідну програму як і багато провідних технічних гравців того часу, вони вирішили запустити мобільну веб-сторінку на основі HTML5. Однак рішення не витримало випробування часом, залишаючи багато можливостей для покращення інтерфейсу користувача та продуктивності. Справді, у 2012 році Марк Цукерберг визнав,

що «найбільша помилка, яку ми припустили як компанія, полягала в тому, що ми зробили занадто багато ставки на HTML, а не на нативні застосунки.

Незабаром, у 2013 році, розробник Facebook Джордан Волке зробив новаторське відкриття – він знайшов метод генерації елементів інтерфейсу користувача для програм iOS за допомогою JavaScript. Це викликало пожежу, і був організований спеціальний хакатон, щоб дізнатися, наскільки багато мобільних розробок можна зробити за допомогою (поки що традиційно веб-рішень) JavaScript. Усього через три роки React Native уже був другим за величиною проектом на GitHub за кількістю учасників. У 2019 році він посів шосте місце з понад 9100 співавторами.

*React та React Native*

Простіше кажучи, React Native не є «новішою» версією React, хоча React Native використовує її. React (також відомий як ReactJS) — це бібліотека JavaScript, яка використовується для створення інтерфейсу веб-сайту. Подібно до React Native, його також розробила команда інженерів Facebook.

Тим часом React Native, який працює на базі React, дозволяє розробникам використовувати набір компонентів інтерфейсу користувача для швидкої компіляції та запуску програм для iOS і Android.

І React, і React Native використовують суміш JavaScript і спеціальної мови розмітки JSX. Однак синтаксис, який використовується для візуалізації елементів у компонентах JSX, відрізняється в програмах React і React Native. Крім того, React використовує деякі HTML і CSS, тоді як React Native дозволяє використовувати рідні елементи мобільного інтерфейсу користувача.

*Кросплатфомна розробка*

Кросплатформна розробка — це практика створення програмного забезпечення, сумісного з більш ніж одним типом апаратної платформи. Кросплатформна програма може працювати на Microsoft Windows, Linux і

macOS або лише на двох із них. Хорошим прикладом кросплатформної програми є веб-браузер або Adobe Flash, які працюють однаково, незалежно від комп'ютера чи мобільного пристрою, на якому вони запущені.

Кросплатформенність розглядається як Святий Грааль розробки програмного забезпечення – ви можете створити свою кодову базу один раз, а потім запустити її на будь-якій платформі, на відміну від програмного забезпечення, створеного для конкретної платформи. Розробники можуть використовувати інструменти, якими вони володіють, як-от JavaScript або C#, для створення застосунків на непідтримувані ОС. Власники програмного забезпечення також зацікавлені в цьому, оскільки розробка продукту з точки зору часу виходу на ринок і витрат скорочується вдвічі.

## 1.4 Життєвий цикл React

[9] У React життєвий цикл компонентів починається з його ініціалізації та закінчується, коли його відмонтовано з DOM. Існує кілька методів, визначених для різних фаз життєвого циклу компонента. Кожен компонент React має власний життєвий цикл, життєвий цикл компонента можна визначити як ряд методів, які викликаються на різних етапах існування компонента. Визначення досить просте, але що ми маємо на увазі під різними етапами? Компонент React може пройти чотири етапи свого життя, як описано нижче.

- *Ініціалізація*: це етап, на якому компонент створюється з заданими Props і стандартним станом. Це робиться в конструкторі класу компонентів.
- *Монтування*: Монтування – це етап візуалізації JSX, який повертає сам метод рендерингу.

- *Оновлення*: Оновлення – це етап оновлення стану компонента та перефарбування програми.
- *Демонтування*: як випливає з назви, демонтування є останнім етапом життєвого циклу компонента, на якому компонент видаляється зі сторінки.

React надає розробникам набір попередньо визначених функцій, які, якщо вони присутні, викликаються навколо певних подій протягом життя компонента. Передбачається, що розробники перевизначають функції з бажаною логікою для відповідного виконання. Ми проілюстрували суть на наступній діаграмі.

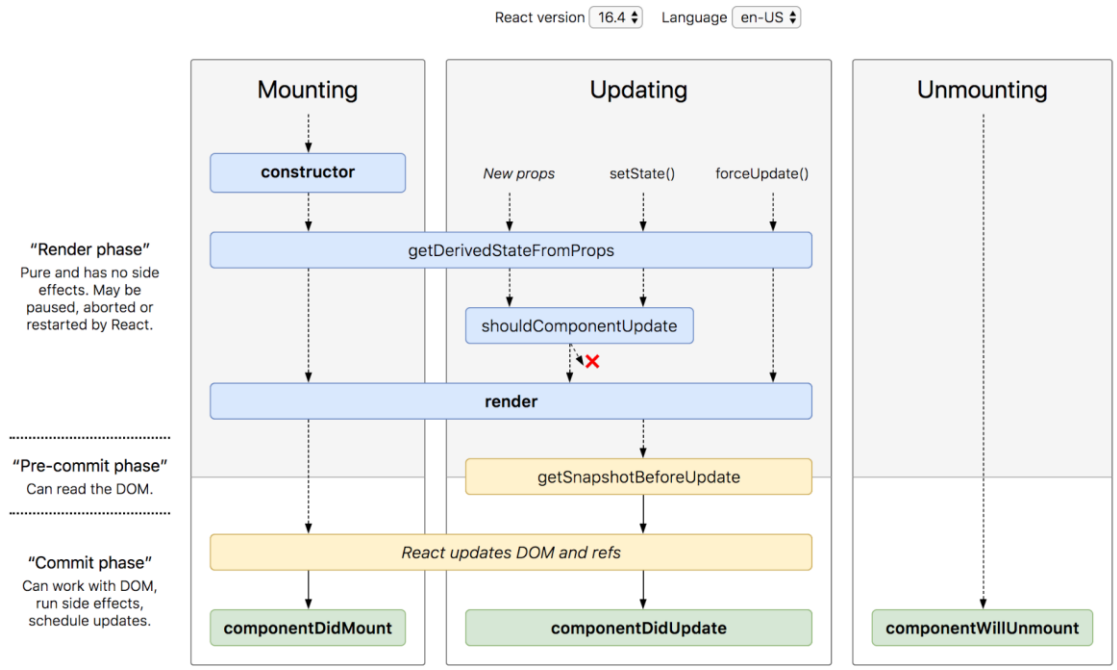



Рисунок 1.4 Життєвий цикл React

*Ініціалізація:* На цьому етапі розробник має визначити атрибути та початковий стан компонента, як правило, це робиться в конструкторі компонента. Наступний фрагмент коду описує процес ініціалізації.

```
    class Clock extends React.Component {
constructor(props)
{
```

```
        // Виклик конструктора
                // Батьківський клас React.Component

    super(props);

    // Встановлення початкового стану
    this.state = { date : new Date() };
  }
}
```

*Монтаж* — це фаза життєвого циклу компонента, коли ініціалізацію компонента завершено та компонент монтується в DOM та вперше відображається на веб-сторінці. Тепер React дотримується процедури за замовчуванням у Конвенціях про іменування цих попередньо визначених функцій, де функції, що містять «Will», представляють до певної фази, а «Did» представляють після завершення цієї фази. Етап монтування складається з двох таких попередньо визначених функцій, як описано нижче.

- Ініціалізація даних та станів в конструкторі
- Функція ComponentDidMount(): ця функція викликається відразу після монтування компонента в DOM, тобто ця функція викликається один раз після першого виконання функції render().

*Оновлення*: React — це бібліотека JS, яка допомагає легко створювати активні веб-сторінки. Зараз активні веб-сторінки – це конкретні сторінки, які поводяться відповідно до свого користувача. Наприклад, візьмемо веб-сторінку онлайн редактора коду, веб-сторінка діє по-різному з кожним користувачем. Користувач А може написати деякий код мовою С у світлій темі, тоді як інший користувач може одночасно написати код Python у темній темі. Ця динамічна поведінка, яка частково залежить від самого користувача,

робить веб-сторінку активною. Як це може бути пов’язано з оновленням? Оновлення — це фаза, на якій стани та властивості компонента оновлюються, а потім відбуваються певні дії користувача, такі як клацання, натискання клавіші на клавіатурі тощо. Нижче наведено описи функцій, які викликаються на різних етапах фази оновлення.

- getDerivedStateFromProps: getDerivedStateFromProps(props, state) — це статичний метод, який викликається безпосередньо перед методом render() як на етапі монтування, так і на етапі оновлення в React. Він приймає оновлені властивості та поточний стан як аргументи.

```
    static getDerivedStateFromProps(props, state) {
  if(props.name !== state.name){
    //Зміна в props
    return{
      name: props.name
    };
  }
  return null; // No change to state
}
```

- Функція setState(): це не функція життєвого циклу, і її можна викликати явно в будь-який момент. Ця функція використовується для оновлення стану компонента.
- Функція shouldComponentUpdate(): за замовчуванням кожне оновлення стану або реквізитів повторно відображає сторінку, але це не завжди може бути бажаним результатом, іноді бажано оновити сторінку не буде перефарбовуватися. Функція shouldComponentUpdate() відповідає вимогам, повідомляючи React

про те, чи вплине оновлення на вихід компонента чи ні. shouldComponentUpdate() викликається перед рендерингом уже змонтованого компонента, коли надходять нові властивості або стани. Якщо повернуто false, наступні кроки рендерингу виконуватися не будуть. Цю функцію не можна використовувати у випадку forceUpdate(). Функція приймає нові властивості та новий стан як аргументи та повертає, чи потрібно повторно -виводити чи ні.

- метод getSnapshotBeforeUpdate(): метод getSnapshotBeforeUpdate() викликається безпосередньо перед відтворенням DOM. Він використовується для збереження попередніх значень стану після оновлення DOM.
- Функція ComponentDidUpdate(): так само ця функція викликається після повторного відтворення компонента, тобто ця функція викликається один раз після того, як функція render() виконується після оновлення стану або Props .

*Демонтування:* Це остання фаза життєвого циклу компонента, тобто фаза демонтування компонента з DOM. Наступна функція є єдиним членом цієї фази.

- Функція ComponentWillUnmount(): ця функція викликається перед тим, як компонент остаточно демонтується з DOM, тобто ця функція викликається один раз перед видаленням компонента зі сторінки, і це означає, що кінець життєвого циклу.

## 1.5 React Хуки

[10] Хуки React — це функції, надані React, які дозволяють розробнику підключатися до функцій React із функціонального компонента. Хуки React принесли багато переваг екосистемі розробки, яка вирішує проблему, пов’язану з компонентами класу.

Хуки — це нові React API, додані до React 16.8. Вони дозволили функціональним компонентам React використовувати функції React, які раніше були доступні лише в компонентах класу React. Це функції, які передають потужність компонентів класу React функціональним компонентам, створюючи більш простий спосіб їх поєднання.

Функції, які починаються зі слова use, називаються хуками. Хуки є більш обмежувальними, ніж інші функції. Хуки викликаються лише у верхній частині ваших компонентів. До появи хуків до 16 лютого 2019 року функціональні компоненти та компоненти класу React виконували різні функції. Функціональні компоненти використовувалися лише для презентації.

*Функціональні компоненти*

Функціональний компонент — це проста функція JavaScript, яка приймає props як аргумент і повертає елемент React. React припускає, що кожен написаний вами компонент є чистою функцією, що означає, що написані компоненти React повинні завжди повертати той самий JSX за однакових вхідних даних. Функціональні компоненти не дозволяють setState() і чому вони називаються компонентами без стану. Коли потрібен стан, створюється компонент класу або стан піднімається до батьківського компонента, і стан передається через властивості до функціонального компонента. Функціональний компонент не може використовувати хуки життєвого циклу, оскільки вони походять від React.component, який розширено з компонентів класу. Функціональні компоненти не відстежують внутрішній стан і не знають життєвого циклу компонента. Тому їх називали «тупими компонентами».

*Компоненти класу*

Компоненти класу відстежують внутрішній стан компонента та дозволяють виконувати операції протягом кожної фази за допомогою методів життєвого циклу. Один із способів побачити компонент класу в дії — це зробити запит на вибірку до зовнішнього API після монтування компонента, оновлення, оновлення стану відповідно до інтерактивності користувача та скасування підписки на магазин після демонтування компонента. Це стало можливим завдяки тому, що компонент класу відстежує свій внутрішній стан і життєвий цикл. Хуки React були додані, щоб вирішити пекло обгорток, величезні компоненти та заплутані класи серед інших, з якими стикаються розробники, використовуючи компоненти класів. Однак деякі з цих проблем не пов’язані безпосередньо з React, а скоріше зі способом розробки нативних класів JavaScript.

*Проблеми в компонентах класу React*

Компоненти класу React є нативним класом JavaScript. Він успадкував проблеми класів JavaScript, включаючи роботу з ключовим словом «this», методи явного прив’язування, докладний синтаксис тощо.

```
        import React, { Component } from 'react';

class ExampleComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { name: 'John Doe' };
    this.changeName = this.changeName.bind(this);
  }

  changeName() {
    this.setState({ name: 'Abel Doe' });
```

```
  }

  render() {
    return (
      <div>
        <p>My name is {this.state.name}.</p>
        <button onClick={this.changeName}>Change name</button>
      </div>
    );
  }
}
```

Наведений вище код відображає простий компонент класу, який передає стан імені в інтерфейс користувача та надає кнопку для зміни імені. Як видно, вам потрібно зв’язати «this», викликати super(props) у базовому конструкторі та завжди додавати префікс «this» до свого стану чи методів, щоб отримати до них доступ. Наведені вище вирази є функцією того, як розроблено класи ES6, і є одними з поширених причин помилок у програмах React.

*Детальний синтаксис*

Компоненти класу React мають докладний синтаксис, який часто може призвести до дуже великих компонентів; компоненти з великою кількістю логіки, розділеної на методи життєвого циклу, які важко читати та дотримуватися. API методу життєвого циклу змушує вас повторювати логіка в різних методах життєвого циклу по всьому компоненту.

Хуки вирішують усі перелічені вище проблеми, пов’язані з класом. Вони також дають змогу писати чистіший, компактніший і зручніший код.

Функціональні компоненти в React — це просто старі функції JavaScript! Отже, якщо функції мають можливість компонування, компоненти React також можуть мати компонування. Це означає, що ми можемо використовувати (компонувати) один або кілька компонентів в інший компонент, як показано на зображенні нижче:

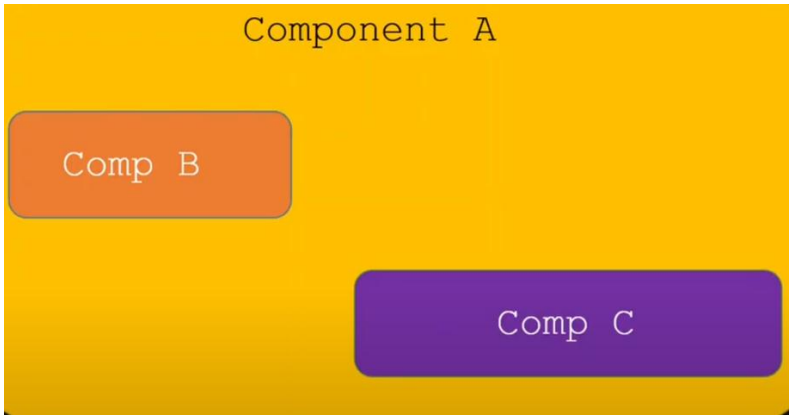



Рисунок 1.5 компонування елементів

Компоненти в React можуть мати статус або без нього. Компонент із збереженням стану оголошує та керує локальним станом у ньому.

*Компонент без стану* — це чиста функція, яка не має локального стану та побічних ефектів, якими потрібно керувати.

*Чиста функція* — це функція без будь-яких побічних ефектів. Це означає, що функція завжди повертає той самий вихід для того самого введення.

Якщо ми вилучимо логіку стану та побічних ефектів із функціонального компонента, ми отримаємо компонент без стану. Крім того, логіку стану та побічних ефектів можна повторно використовувати в іншому місці програми. Тому має сенс максимально ізолювати їх від компонента.


Рисунок 1.6 *Компонент із збереженням стану*

За допомогою React Hooks ми можемо ізолювати логіку стану та побічні ефекти від функціонального компонента. Хуки — це функції JavaScript, які керують поведінкою стану та побічними ефектами, ізолюючи їх від компонента.

Отже, тепер ми можемо виокремити всю логіку стану в хуках і використовувати (компонувати їх, оскільки хуки також є функціями) у компоненти.

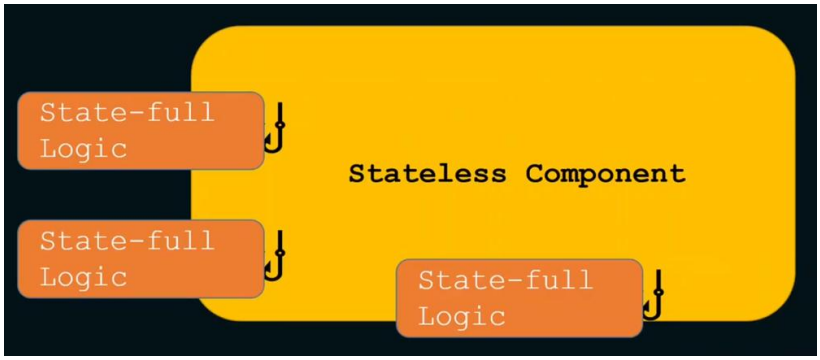



Рисунок 1.7 Ізольована логіка із збереженням стану в хуках

Питання в тому, що це за логіка станів? Це може бути будь-що, що потребує локального оголошення та керування змінною стану. Наприклад, логіка отримання даних і керування даними в локальній змінній є станом. Ми також можемо захотіти повторно використовувати логіку отримання в кількох компонентах.

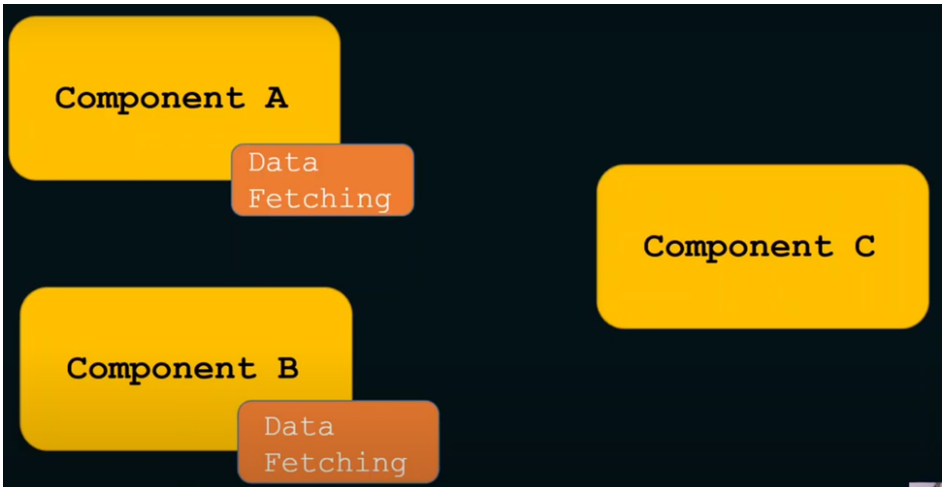



Рисунок 1.8 повторне використання логіки в кількох компонентах

*Хуки React*

Хуки React — це прості функції JavaScript, за допомогою яких ми можемо відокремити повторно використовувану частину від функціонального компонента. Хуки можуть зберігати стан і керувати побічними ефектами.

React надає купу стандартних вбудованих хуків:

- useState: Керувати станами. Повертає значення стану та функцію оновлення для його оновлення.
- useEffect: для керування побічними ефектами, такими як виклики API, підписки, таймери, мутації тощо.
- useContext: щоб повернути поточне значення для контексту.
- useReducer: useState альтернатива для допомоги у складному управлінні станом.
- useCallback: Він повертає запам’ятану версію зворотного виклику, щоб допомогти дочірньому компоненту не повторно відтворюватися без потреби.

- useMemo: повертає мемоізоване значення, яке допомагає оптимізувати продуктивність.
- useRef: повертає об’єкт ref із властивістю .current. Об’єкт ref є змінним. В основному він використовується для імперативного доступу до дочірнього компонента.
- useLayoutEffect: запускається в кінці всіх мутацій DOM. Найкраще використовувати useEffect якомога більше над цим, оскільки useLayoutEffect спрацьовує синхронно.
- useDebugValue: Допомагає відобразити мітку в React DevTools для користувацьких хуків.

Також можна створювати власні хуки для унікальних випадків використання, таких як вибірка даних, реєстрація на диск, таймери та багато іншого.

## 1.6 Реальний та віртуальний DOM

*Справжній DOM*

[11] DOM перекладається як "Об’єктна модель документа". Він представляє весь інтерфейс користувача веб-програми як структуру даних дерева. Простіше кажучи, це структурне представлення елементів HTML веб-програми.

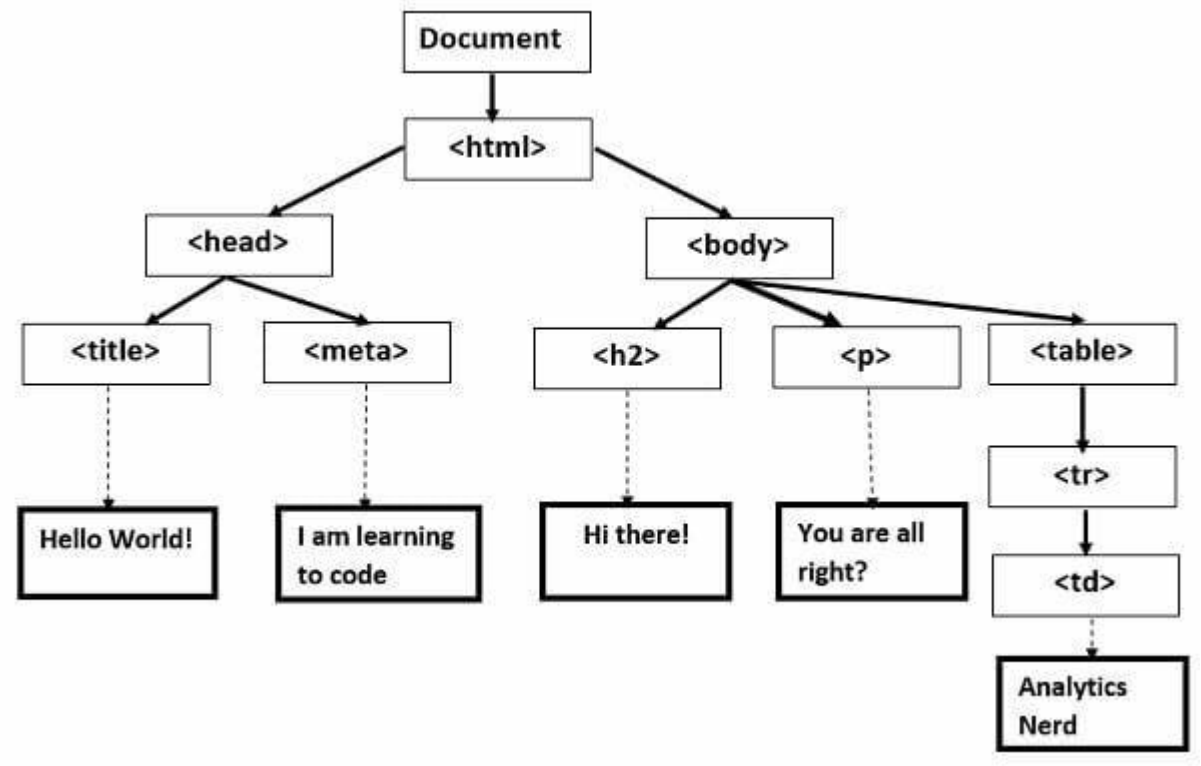



Рисунок 1.9 Дерево елементів сайту

Кожен раз, коли змінюється вигляд програми, Document Object Model (DOM) оновлюється. Це може негативно впливати на продуктивність та призводити до сповільнення роботи веб-сайту, оскільки кожна зміна в DOM вимагає повторного відображення та представлення цієї зміни.

Отже, зі збільшенням кількості компонентів у користувацькому інтерфейсі та складною структурою DOM, вартість оновлення DOM зростає. Це тому, що з кожною зміною потрібно виконати додаткові маніпуляції з DOM та знову відобразити його.

*Причини сповільнення роботи з DOM*

DOM представлено у вигляді деревовидної структури даних. Через це зміни та оновлення DOM відбуваються швидко. Але після зміни оновлений елемент і його дочірні елементи потрібно відобразити повторно, щоб оновити інтерфейс програми. Повторне відтворення або перемальовування інтерфейсу користувача робить його повільним. Таким чином, чим більше у вас компонентів інтерфейсу користувача, тим дорожчими можуть бути

оновлення DOM, оскільки їх потрібно буде повторно відтворювати для кожного оновлення DOM.

*Віртуальний DOM*

Віртуальний DOM (VDOM) є абстракцією реального DOM, створеного та підтримуваного бібліотеками JavaScript, такими як React. Віртуальний DOM — це полегшена копія справжнього DOM, яка забезпечує швидші оновлення та покращує продуктивність. DOM є лише віртуальним представленням DOM. Кожного разу, коли стан нашої програми змінюється, віртуальний DOM оновлюється замість реального.

«Хіба віртуальний DOM не робить те саме, що і справжній DOM, це звучить як подвійна робота?» Як це може бути швидше, ніж просто оновлення реального DOM?»

Віртуальний DOM набагато швидший і ефективніший, ось чому. Коли нові елементи додаються до інтерфейсу користувача, створюється віртуальний DOM, який представлений у вигляді дерева. Кожен елемент є вузлом цього дерева. Якщо стан будь-якого з цих елементів змінюється, створюється нове віртуальне дерево DOM. Потім це дерево порівнюється або «розрізняється» з попереднім віртуальним деревом DOM.

Після цього віртуальна DOM розраховує найкращий можливий метод внесення цих змін до реальної DOM. Це гарантує мінімальну кількість операцій над реальним DOM. Отже, зниження витрат на продуктивність оновлення реального DOM.

На зображенні нижче показано віртуальне дерево DOM і процес зміни.

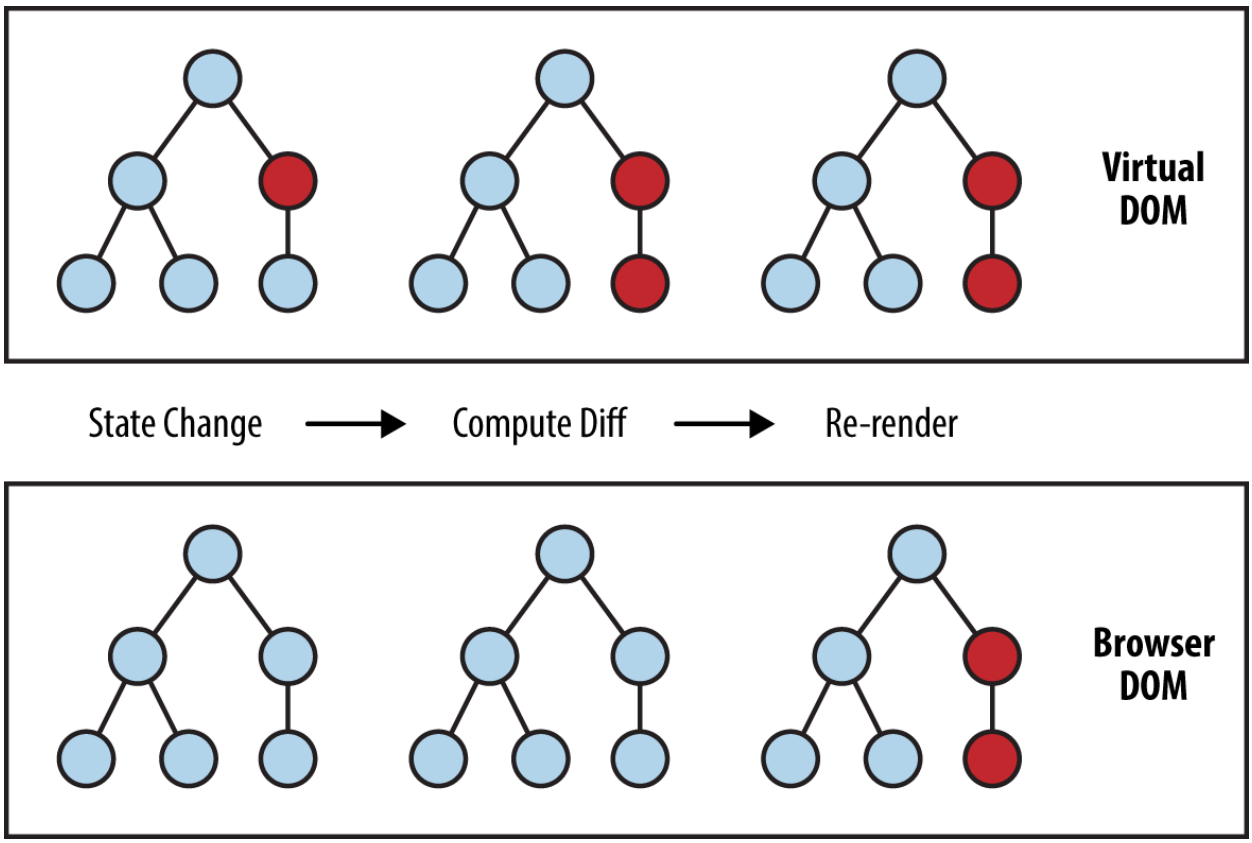



Рисунок 1.10 Віртуальне обчислення DOM

Червоні кружечки позначають вузли, які змінилися. Ці вузли представляють елементи інтерфейсу користувача, стан яких було змінено. Потім обчислюється різниця між попередньою версією віртуального DOM дерева і поточним віртуальним DOM деревом. Потім все батьківське піддерево повторно відображається, щоб отримати оновлений інтерфейс користувача. Це оновлене дерево потім пакетно оновлюється до справжнього DOM.

*Як React використовує Virtual DOM*

У React кожна частина інтерфейсу є компонентом, і кожен компонент має стан. React слідує спостережуваному шаблону та відстежує зміни стану. Коли стан компонента змінюється, React оновлює віртуальне дерево DOM. Після оновлення React порівнює поточну версію віртуальної DOM з попередньою VDOM. Цей процес називається «різниця».

Якщо React дізнається, які віртуальні об’єкти DOM змінилися, React оновлює лише ці об’єкти в реальному DOM. Це робить продуктивність набагато кращою порівняно з безпосередньою маніпуляцією реальним DOM. Це робить React видатним як високопродуктивна бібліотека JavaScript.

Простими словами, ви вказуєте React, у якому стані ви хочете, щоб інтерфейс користувача був, і він гарантує, що DOM відповідає цьому стану. Великою перевагою тут є те, що розробнику, не потрібно задумуватись про маніпулювання атрибутами, обробку подій або ручне оновлення DOM. Все це відбувається за лаштунками.

Усі ці деталі абстрагуються від розробників React. Все, що вам потрібно зробити, це оновити стан вашого компонента за потреби, а React подбає про інше. Це забезпечує чудовий досвід розробника під час використання React.

*Функція React render().*

Функція render() є точкою входу, де створюється дерево елементів React. Коли стан або реквізит у компоненті оновлюється, React негайно виявляє зміну стану та повторно відтворює компонент.setState()  поверне інше дерево елементів React. Якщо ви використовуєте render() React з’ясовує, як ефективно оновити інтерфейс користувача відповідно до останніх змін дерева.

*Пакетне оновлення*

React використовує механізм пакетного оновлення для оновлення реального DOM, що призводить до підвищення продуктивності. Це означає, що оновлення справжнього DOM надсилаються пакетами, замість того, щоб надсилати оновлення для кожної окремої зміни стану.

# 2 МЕТОДИ ОПТИМІЗАЦІЇ REACT ЗАСТОСУНКІВ

## 2.1 Використання незмінних структур даних

[12] Незмінність даних не є архітектурою чи шаблоном проектування; це конкретний спосіб написання коду. Це змушує вас задуматися про те, як ви будуєте потік даних вашого додатку. На мою думку, незмінність даних — це практика, яка базується на строгому однобічному потоці даних.

Незмінність даних, яка походить із світу функціонального програмування, може бути застосована до проектування веб-додатків. Вона може мати багато переваг, таких як:

- Відсутність побічних ефектів;
- Незмінні об'єкти даних простіше створювати, тестувати та використовувати;
- Допомагає уникнути часового зв'язку;
- Легше відстежувати зміни.

У світі React ми використовуємо концепцію компонента для зберігання внутрішнього стану компонентів, і зміни в стані можуть спричинити повторний рендеринг компонента.

React будує та утримує внутрішнє представлення відображеного інтерфейсу (віртуальний DOM). Коли змінюються властивості чи стан компонента, React порівнює новий елемент з попереднім відображеним. Якщо вони не рівні, React оновить DOM. Тому ми повинні бути обережними при зміні стану.

Давайте розглянемо приклад компонента "Список користувачів":

```
state = {
        users: []
    }

    addNewUser = () =>{
        /**
         *  OfCourse not correct way to insert
         *  new user in user list
         */
        const users = this.state.users;
        users.push({
            userName: "robin",
            email: "email@email.com"
        });
        this.setState({users: users});
    }
```

Нейтральність тут полягає в тому, що ми додаємо нових користувачів до змінної `users`, яка є посиланням на `this.state.users`.

Порада від експерта: Стан React повинен розглядатися як незмінний. Ми ніколи не повинні змінювати `this.state` безпосередньо, оскільки подальше викликання `setState()` може замінити зроблену мутацію.

Отже, що не так із зміною стану безпосередньо? Допустимо, ми перевизначили `shouldComponentUpdate` і перевіряємо `nextState` в порівнянні з `this.state`, щоб переконатися, що ми перерендерюємо компоненти лише при змінах у стані.

```
shouldComponentUpdate(nextProps, nextState) {
    if (this.state.users !== nextState.users) {
      return true;
    }
    return false;
  }
```

Навіть якщо відбуваються зміни в масиві користувачів, React не перерендерить інтерфейс користувача, оскільки це те саме посилання.

Найпростіший спосіб уникнути цього типу проблем - уникати мутацій у властивостях чи стані. Таким чином, метод addNewUser може бути переписаний за допомогою concat:

```
addNewUser = () => {
  this.setState(prevState => ({
    users: prevState.users.concat({
      userName: "robin",
      email: "email@email.com"
    })
  }));
}
```

Використання concat дозволяє створити новий масив, додавши до нього нового користувача, і при цьому не змінює оригінальний масив users. Це дозволяє React правильно визначити, що стан змінився, і спричиняє перерендеринг компоненту.

[13] Для роботи зі змінами у стані чи властивостях компонентів React можна враховувати наступні підходи до незмінності:

Для масивів: використовуйте `[].concat` або ES6 `[ ...params]`.

Для об'єктів: використовуйте `Object.assign({}, ...)` або ES6 `{...params}`.

Ці два методи добре працюють при впровадженні незмінності у ваш код.

Однак часто краще використовувати оптимізовану бібліотеку, яка надає набір незмінних структур даних. Ось деякі бібліотеки, які ви можете використовувати:

1. Immutability Helper: Це хороша бібліотека для мутації копії даних без зміни джерела.

2. Immutable.js: Це популярна бібліотека, яка надає різноманітні стійкі незмінні структури даних, включаючи List, Stack, Map, OrderedMap, Set, OrderedSet і Record.

3. Seamless-immutable: Ця бібліотека пропонує незмінні структури даних для JavaScript, які сумісні зі звичайними масивами та об'єктами.

4. React-copy-write: Незмінна бібліотека управління станом React з простим мутабельним API, мемоізованими селекторами та структурною часткою.

Метод `setState` у React є асинхронним. Це означає, що, замість того щоб безпосередньо мутувати `this.state`, `setState()` створює відкладений перехід до стану. Якщо ви отримуєте доступ до `this.state` після виклику цього методу, він, можливо, поверне існуюче значення. Щоб цього уникнути, використовуйте функцію зворотного виклику `setState`, щоб виконати код після завершення виклику.

2. Функційні/безстандартні компоненти та React.PureComponent

Функційні та безстаничні (stateless) компоненти та React.PureComponent в React надають два різних способи оптимізації додатків React на рівні компонентів.

Функційні компоненти запобігають створенню екземплярів класів, при цьому зменшуючи розмір пакета, оскільки вони мініфікуються краще, ніж класи.

[14] З іншого боку, для оптимізації оновлень інтерфейсу користувача ми можемо розглядати конвертацію функційних компонентів в клас PureComponent (або клас із власним методом shouldComponentUpdate). Проте, якщо компонент не використовує стан та інші методи життєвого циклу, час початкового рендерингу є трошки складнішим у порівнянні з

функційними компонентами, які можуть мати потенційно швидше оновлення.

1. Коли слід використовувати React.PureComponent?

[15] React.PureComponent використовує поверхневе порівняння при зміні стану. Це означає, що він порівнює значення для примітивних типів даних та порівнює посилання для об'єктів. У зв'язку з цим ми повинні переконатися, що дві умови виконуються при використанні React.PureComponent:

1. Стан/властивості компонента є незмінним об'єктом.

2. Стан/властивості не повинні мати об'єктів з багаторівневою вкладеністю.

Усі дочірні компоненти React.PureComponent також повинні бути чистими (Pure) або функціональними компонентами.

3. Множинні файли часток

Ваша програма завжди розпочинається з декількох компонентів. Ви починаєте додавати нові функції та залежності, і перш ніж ви це усвідомите, у вас вийде великий виробничий файл.

Ви можете розглядати можливість мати два окремих файли, розділивши вендорний, або код бібліотек третіх сторін від коду вашого додатку, скориставшись CommonsChunkPlugin для webpack. В результаті ви отримаєте файли vendor.bundle.js та app.bundle.js. Розділивши файли, ваш браузер рідше кешує та паралельно завантажує ресурси для скорочення часу очікування завантаження.

Якщо ви використовуєте останню версію webpack, ви також можете врахувати використання SplitChunksPlugin.

4. Використання прапорця режиму в Webpack

Якщо ви використовуєте webpack 4 як збірник модулів для свого додатку, ви можете розглядати встановлення параметра режиму на

production. Це в основному говорить webpack використовувати вбудовану оптимізацію:

```
javascript
module.exports = {
  mode: 'production'
};
```

Альтернативно, ви можете передати його як аргумент командного рядка:

webpack --mode=production

Це обмежить оптимізації, такі як мініфікація або вилучення коду тільки для розробників, для бібліотек. Це не викладатиме вихідний код, шляхи до файлів та багато іншого.

5. Оптимізація залежностей

При розгляді оптимізації розміру пакета додатка важливо перевірити, скільки коду ви фактично використовуєте залежностей. Наприклад, ви можете використовувати Moment.js, який включає локалізовані файли для підтримки багатьох мов. Якщо вам не потрібна підтримка різних мов, ви можете розглядати використання moment-locales-webpack-plugin для видалення невикористаних локалей з кінцевого пакету.

Інший приклад - це lodash. Допустимо, ви використовуєте лише 20 з 100+ методів, тоді наявність всіх додаткових методів у кінцевому пакеті не є оптимальною. Таким чином, ви можете використовувати lodash-webpack-plugin для видалення невикористовуваних функцій.

6. Використання React.Fragments для уникнення додаткових обгорток HTML-елементів

React.Fragments дозволяють групувати список дочірніх елементів без додавання додаткового вузла.

```
jsx
```

```
class Comments extends React.PureComponent {
  render() {
    return (
      <React.Fragment>
        <h1>Comment Title</h1>
        <p>comments</p>
        <p>comment time</p>
      </React.Fragment>
    );
  }
}
```

Є альтернативний та більш лаконічний синтаксис за допомогою React.Fragments:

```
      jsx
class Comments extends React.PureComponent {
  render() {
    return (
      <>
        <h1>Comment Title</h1>
        <p>comments</p>
        <p>comment time</p>
      </>
    );
  }
}
```

7. Уникаємо визначення вбудованої функції у функції рендерингу.

Оскільки функції у JavaScript є об'єктами ({} !== {}), вбудована функція завжди не пройде перевірку різниці властивостей, коли React робить

перевірку різниці. Крім того, стрілкова функція створює новий екземпляр функції при кожному рендерингу, якщо вона використовується в властивості JSX. Це може створити багато роботи для сборщика сміття.

Приклад використання вбудованої функції:

```
        jsx
default class CommentList extends React.Component {
   state = {
      comments: [],
      selectedCommentId: null
   }

   render(){
      const { comments } = this.state;
      return (
        comments.map((comment)=>{
           return <Comment onClick={(e)=>{
               this.setState({selectedCommentId:comment.commentId})
           }} comment={comment} key={comment.id}/>
        })
      )
   }
}
```

Замість визначення вбудованої функції для властивостей, ви можете визначити стрілкову функцію:

```
        jsx
default class CommentList extends React.Component {
   state = {
```

```
    comments: [],
    selectedCommentId: null
  }

  onCommentClick = (commentId)=>{
    this.setState({selectedCommentId:commentId})
  }

  render(){
    const { comments } = this.state;
    return (
      comments.map((comment)=>{
        return <Comment onClick={this.onCommentClick}
         comment={comment} key={comment.id}/>
      })
    )
  }
}
```

8. Регулювання та затримка виконання подій в JavaScript

[16] Частота спрацювання подій - це кількість разів, коли обробник подій викликається протягом певного періоду часу.

Загалом, клацаннями миші викликаються менше подій, ніж прокрутка та наведення миші. Вища частота спрацювання подій іноді може призвести до зависання вашого додатка, але це можна контролювати.

Розглянемо деякі техніки.

Спочатку ідентифікуйте обробник подій, який виконує дорогу роботу. Наприклад, запит XHR або маніпуляції DOM, що виконує оновлення

користувацького інтерфейсу, обробляє велику кількість даних або виконує великі обчислення. У таких випадках техніки затримки та регулювання можуть бути великим врятівником без змін в слухачі подій.

Регулювання (Throttling)

Взагалі кажучи, регулювання означає відкладення виконання функції. Замість того, щоб виконати обробник подій/функцію негайно, ви додаєте кілька мілісекунд затримки при спрацьовуванні події. Це може бути корисно при реалізації безкінечної прокрутки, наприклад. Замість отримання наступного набору результатів при прокручуванні користувача, ви можете відкласти

Деактивація (Debouncing)

На відміну від регулювання, деактивація - це техніка для запобігання занадто частому спрацьовуванню події. Якщо ви використовуєте lodash, ви можете обгорнути функцію, яку ви хочете викликати, в функцію lodash's debounce.

Ось демо-код для пошуку коментарів:

```
jsx
import debounce from 'lodash.debounce';

class SearchComments extends React.Component {
 constructor(props) {
  super(props);
  this.state = { searchQuery: "" };
 }

 setSearchQuery = debounce(e => {
  this.setState({ searchQuery: e.target.value });
```

```
  // Виклик API або маніпуляція коментарями на стороні клієнта
 }, 1000);

 render() {
  return (
    <div>
      <h1>Search Comments</h1>
      <input type="text" onChange={this.setSearchQuery} />
    </div>
  );
 }
}
```

Якщо ви не використовуєте lodash, ви можете використовувати стиснуту функцію de bounce для її впровадження в JavaScript.

javascript

function debounce(a,b,c){var d,e;return function(){function h(){d=null,c||(e=a.apply(f,g))}var f=this,g=arguments;return clearTimeout(d),d=setTimeout(h,b),c&&!d&&(e=a.apply(f,g)),e}}

9. Уникайте використання індекса як ключа для map

Часто бачите, що індекси використовуються як ключ, коли відображається список.

```
jsx
{
   comments.map((comment, index) => {
      <Comment
        {..comment}
        key={index} />
   })
```

```
}
```

Проте використання ключа як індексу може призвести до відображення вашому додатку некоректних даних, оскільки він використовується для ідентифікації елементів DOM. Коли ви додаєте або видаляєте елемент зі списку, якщо ключ такий самий, як раніше, React припускає, що елемент DOM представляє той самий компонент.

Завжди рекомендується використовувати унікальну властивість як ключ, або, якщо ваші дані не мають унікальних атрибутів, ви можете подумати про використання модуля shortid, який генерує унікальний ключ.

```
        jsx
import shortid from  "shortid";

{
   comments.map((comment, index) => {
      <Comment
         {..comment}
         key={shortid.generate()} />
   })
}
```

Однак, якщо у даних є унікальна властивість, така як ID, то краще використовувати цю властивість.

```
        jsx
{
   comments.map((comment, index) => {
```

```
      <Comment
        {..comment}
        key={comment.id} />
   })
}
```

У певних випадках використання індексу як ключа - це зовсім прийнятно, але тільки якщо виконується нижче зазначена умова:

- Список та елементи статичні

- Елементи у списку не мають ID, і список ніколи не буде переупорядкований або відфільтрований

- Список є незмінним

10. Уникайте використання Props у початкових станах

Ми часто потребуємо передавати початкові дані з props до компонента React для встановлення початкового значення стану.

```
        jsx
class EditPanelComponent extends Component {

   constructor(props){
      super(props);

      this.state ={
         isEditMode: false,
         applyCoupon: props.applyCoupon
      }
   }

   render(){
      return <div>
```

```
            {this.state.applyCoupon &&
            <>Enter Coupon: <Input/></>}
          </div>
    }
}
```

[17] Якщо props змінюються без оновлення компонента, нове значення props ніколи не буде присвоєно applyCoupon стану. Це тому, що функція конструктора викликається лише при створенні EditPanelComponent вперше.

Щоб цитувати документацію React:

Використання props для ініціалізації стану в функції конструктора часто призводить до дублювання "джерела правди", тобто де фактично знаходяться реальні дані. Це тому, що функція конструктора викликається лише при створенні компонента.

Обхідний шлях:

- Не ініціалізуйте стан за допомогою props, які можуть бути змінені пізніше. Замість цього використовуйте props безпосередньо в компоненті.

```
        jsx
class EditPanelComponent extends Component {

   constructor(props){
      super(props);

      this.state ={
         isEditMode: false
      }
   }

   render(){
```

```
        return <div>{this.props.applyCoupon &&
         <>Enter Coupon:<Input/></>}</div>
    }
}
```

- Ви можете використовувати componentWillReceiveProps для оновлення стану при зміні props.

jsx

```
class EditPanelComponent extends Component {

    constructor(props){
        super(props);

        this.state ={
            isEditMode: false,
            applyCoupon: props.applyCoupon
        }
    }

    // скидання стану, якщо змінено вихідну властивість
    componentWillReceiveProps(nextProps){
        if (nextProps.applyCoupon !== this.props.applyCoupon) {
            this.setState({ applyCoupon: nextProps.applyCoupon })
        }
    }

    render(){
        return <div>{this.props.applyCoupon &&
```

```
        <>Enter Coupon: <Input/></>}</div>
    }
}
```

# 3 ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ REACT-ЗАСТОСУНКІВ: ПІДХОДИ ТА ІНСТРУМЕНТИ

[18] З віртуальним DOM React-додатки вже мають високу продуктивність, але оптимізація React-додатків все ще є необхідною. До прикладу компонент, в основному, схожий на ексельну сітку, і на сторінці є список таких компонентів. Якщо користувач працює з великими даними і вставляє рядок, сторінка стає повільною і завмирає на кілька секунд. Після того, як користувач перезавантажив сторінку і виконав ті ж самі дії, сторінка все ще не працює. Причина цього випадку полягає в тому, що React генерує віртуальний DOM, рендерить всі компоненти, а потім виконує налагодження і оновлює реальний DOM.

У деяких випадках налагодження витрачає ресурси. Ідеальною ситуацією є випадок, коли дочірній компонент потребує оновлення, React повинен лише відновити і порівняти всі компоненти на шляху від кореня до дочірнього компонента в віртуальному DOM. Однак React за замовчуванням відновлює всі компоненти в дереві віртуального DOM, а потім порівнює згенероване дерево віртуального DOM із попереднім деревом віртуального DOM. Додатку потрібен додатковий час для відновлення компонентів, які не мають змін.

[19] У прикладі React Example 1 в Додатку В сторінка відображає this.state.value на екрані і має кнопку для виклику функції handleClick, яка викликає setState для збільшення this.state.value на 1. Після натискання користувача на кнопку, this.state.value збільшиться на 1. Функція render

поверне новий елемент React із оновленим this.state.value як новий вміст, який буде повторно відображено на екрані.

Однак якщо this.state.value оновлюється за допомогою setState із тим самим значенням, що і попереднє значення, також буде викликана функція render. У прикладі React 2 в Додатку В, після натискання користувача на кнопку, функція handleClick буде викликана і викличе setState для оновлення this.state.value на 1, що є значенням за замовчуванням для this.state.value. Як показано на рисунку, результат профілювання показує, що компонент App перерендерюється за 1.7 секунди на 1.9 мс після натискання кнопки. Стан не має змін, але викликається функція render для одного компонента. Якщо функція setState викликається лише тоді, коли потрібно змінити стан, перерендерювання компонента можна уникнути. Однак перерендерювання батьківського компонента також призведе до перерендерювання дочірніх компонентів, які не мають змін.

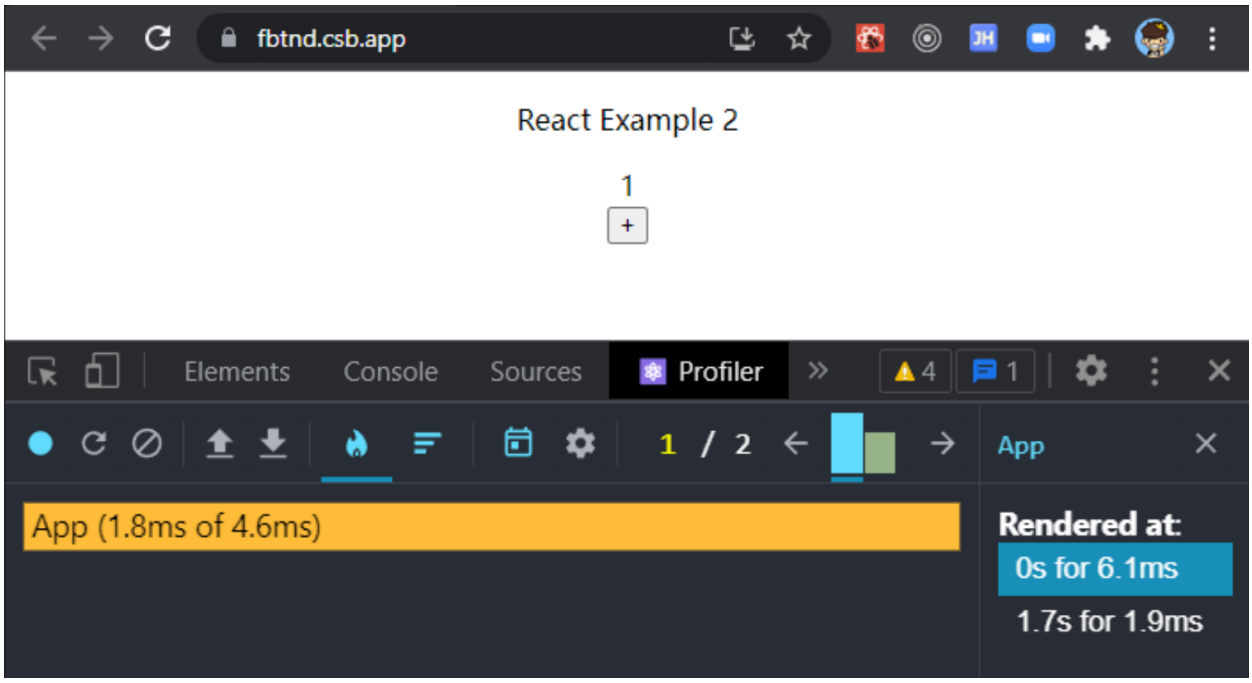



Рисунок 3.1 Результат профілювання для прикладу React 3 у Додатку В

[20] У прикладі React 3 з Додатка В батьківський компонент, Parent, має два дочірні компоненти, ChildOne і ChildTwo. ChildOne отримує дані зі стану Parent як властивість і відображає ці дані на екрані. ChildTwo не

отримує жодних даних від Parent, цей компонент лише відображає повідомлення на екрані. Після натискання користувачем кнопки викликається функція handleClick, стан Parent не змінюється. Однак обидва ChildOne і ChildTwo рендеряться знову. Згідно з рисунком, результат профайлера показує, що всі компоненти рендеряться за 1,8 мс після натискання кнопки.

Отже, незалежно від того, чи отримує дочірній компонент дані від батьківського компонента чи ні, дочірній компонент буде знову рендеритися, якщо батьківський компонент рендериться.

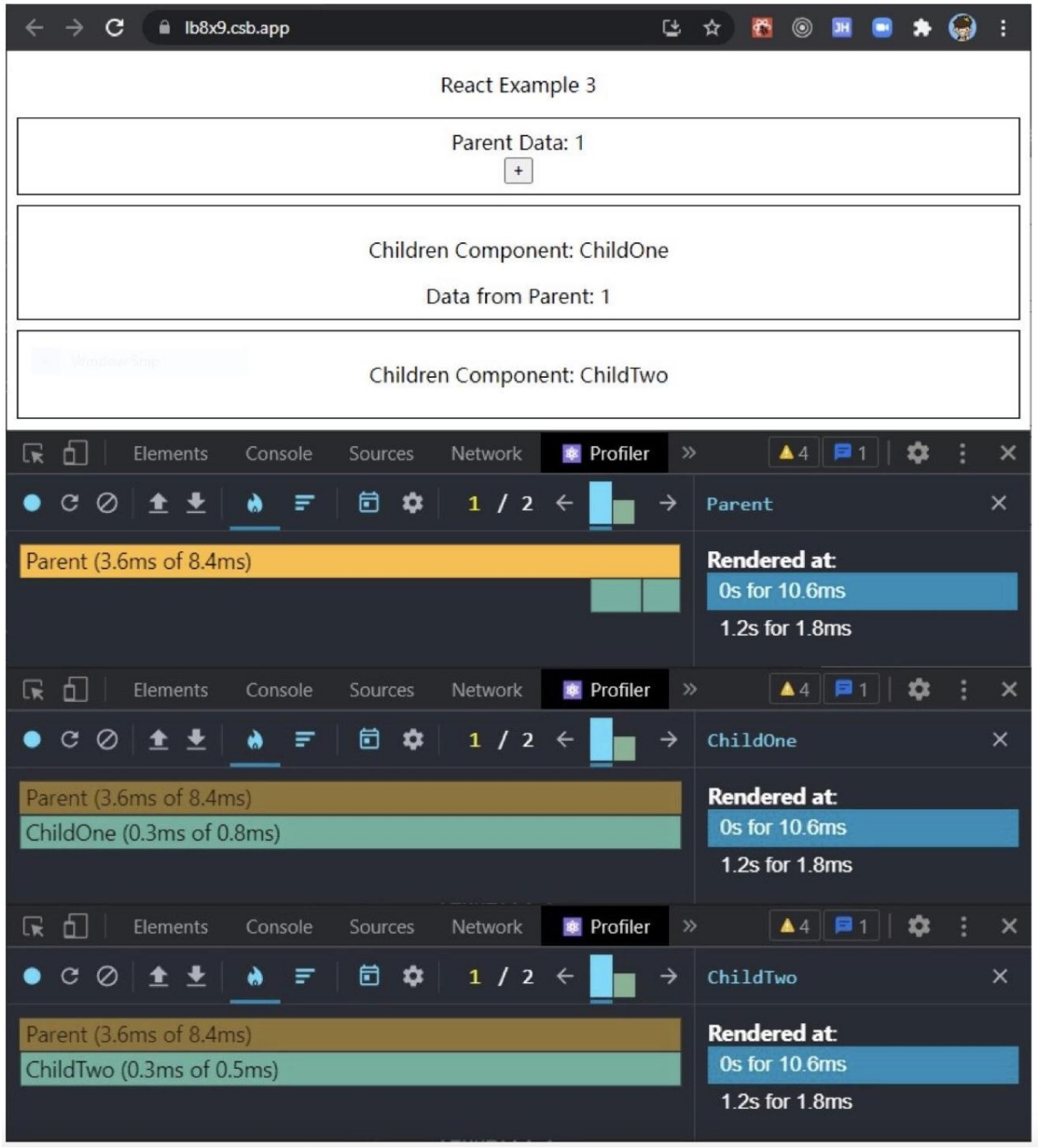



Рисунок 3.2 Результат профілювання для прикладу React 3 у Додатку В

Це важлива проблема для додатка. Коли бізнес-логіка стає складніше, а додаток стає великим, кількість компонентів зростає, і глибина дерева DOM стає все більшою. Непотрібний рендеринг призводить до більше проблем з продуктивністю. Таким чином, центральною концепцією оптимізації додатків у React є зменшення непотрібного перерендерингу компонентів після зміни стану компонента.

*PureComponent, React.memo*

[21] Як показано у результаті профайлера на рисунку вище, лише батьківський компонент піддався оновленню стану, проте всі властивості, передані від батьківського компонента до дочірнього компонента, не були змінені. Це оновлення стану призведе до перерендерингу дочірнього компонента. З точки зору декларативної філософії проектування React, якщо властивості та стан дочірнього компонента не змінені, то структуру DOM і згенеровані ефекти не повинні змінюватися. Коли дочірній компонент відповідає концепції декларативного дизайну, процес рендерингу повинен бути пропущений у цей момент. PureComponent та React.memo відповідають на цей сценарій. PureComponent виконує поверхневе порівняння властивостей та стану для класового компонента, а React.memo робить поверхневе порівняння властивостей для функційного компонента.

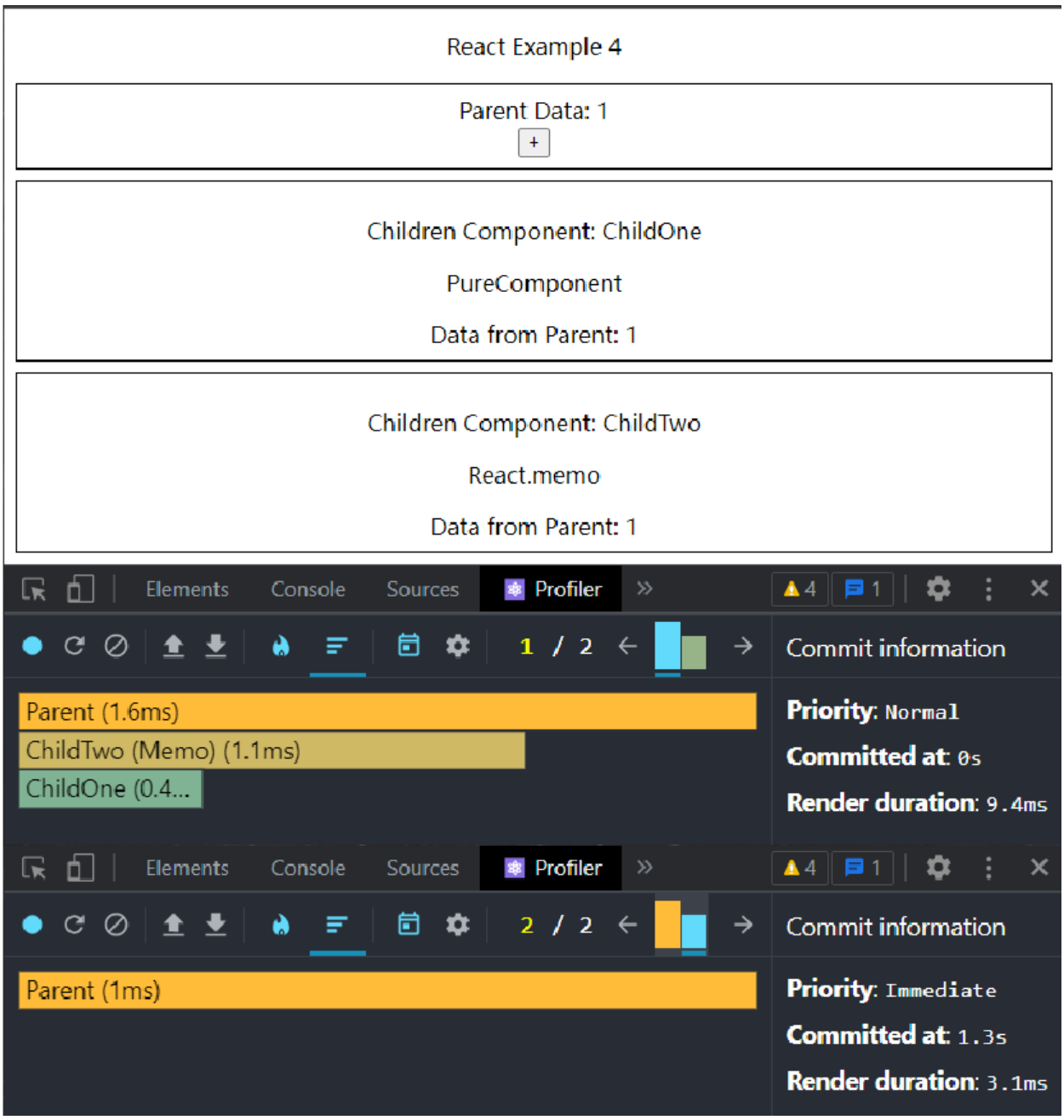



Рисунок 3.3 - результат профайлера для реалізації Прикладу 4 у Додатку В

[22] У цьому прикладі створюється батьківський компонент, Parent, який успадковує звичайний компонент React. У батьківського компонента є два дочірні компоненти, ChildOne та MemorizedChildTwo. ChildOne успадковується від PureComponent. MemorizedChildTwo - це кешований компонент ChildTwo за допомогою React.memo. ChildTwo - це функційний компонент із тією ж логікою, що і ChildOne. Після натискання кнопки буде перерендерено лише Parent. Як показано на Рисунку 6, перший рендеринг

для всіх компонентів, Parent, ChildOne та ChildTwo, був 0с на 9,4 мс, і лише Parent був перерендерений через 1,3 с на 3,1 мс після натискання кнопки. Перед кожним перерендерингом як чистий компонент, так і кешований компонент за допомогою React.memo виконують поверхневе порівняння властивостей та стану, щоб вирішити, чи виконувати перерендеринг.

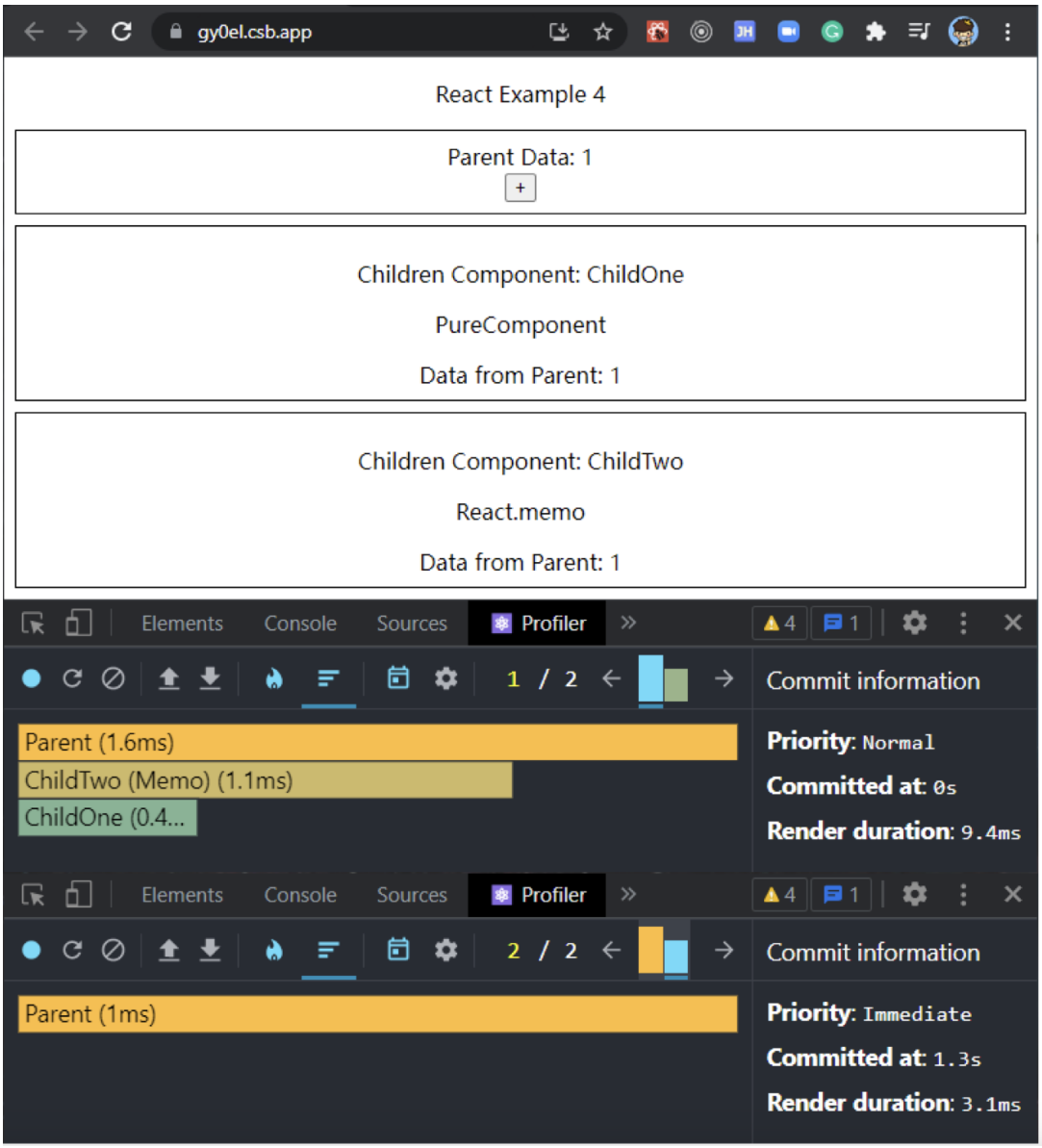



Рисунок 3.4 Результат профілювання для прикладу React 4 у Додатку В

*shouldComponentUpdate*

[23] У реальній розробці розробники зазвичай передають великі об'єкти від батьківського до дочірнього компонента як властивості. Коли властивість великого об'єкта оновлюється, але не використовується в дочірньому

компоненті, це оновлення викличе процес рендерингу дочірнього компонента. Як показано у Прикладі 5 React у Додатку В, обидва дочірні компоненти, ChildOne і ChildTwo, успадковані від PureComponent. this.state.value - це об'єкт з властивістю числового типу, названою "number", і Parent передає this.state.value до ChildOne. ChildOne відображає props.value.number на сторінці. Рисунок 7 показує, що після натискання кнопки перерендеряться лише ChildOne та Parent.

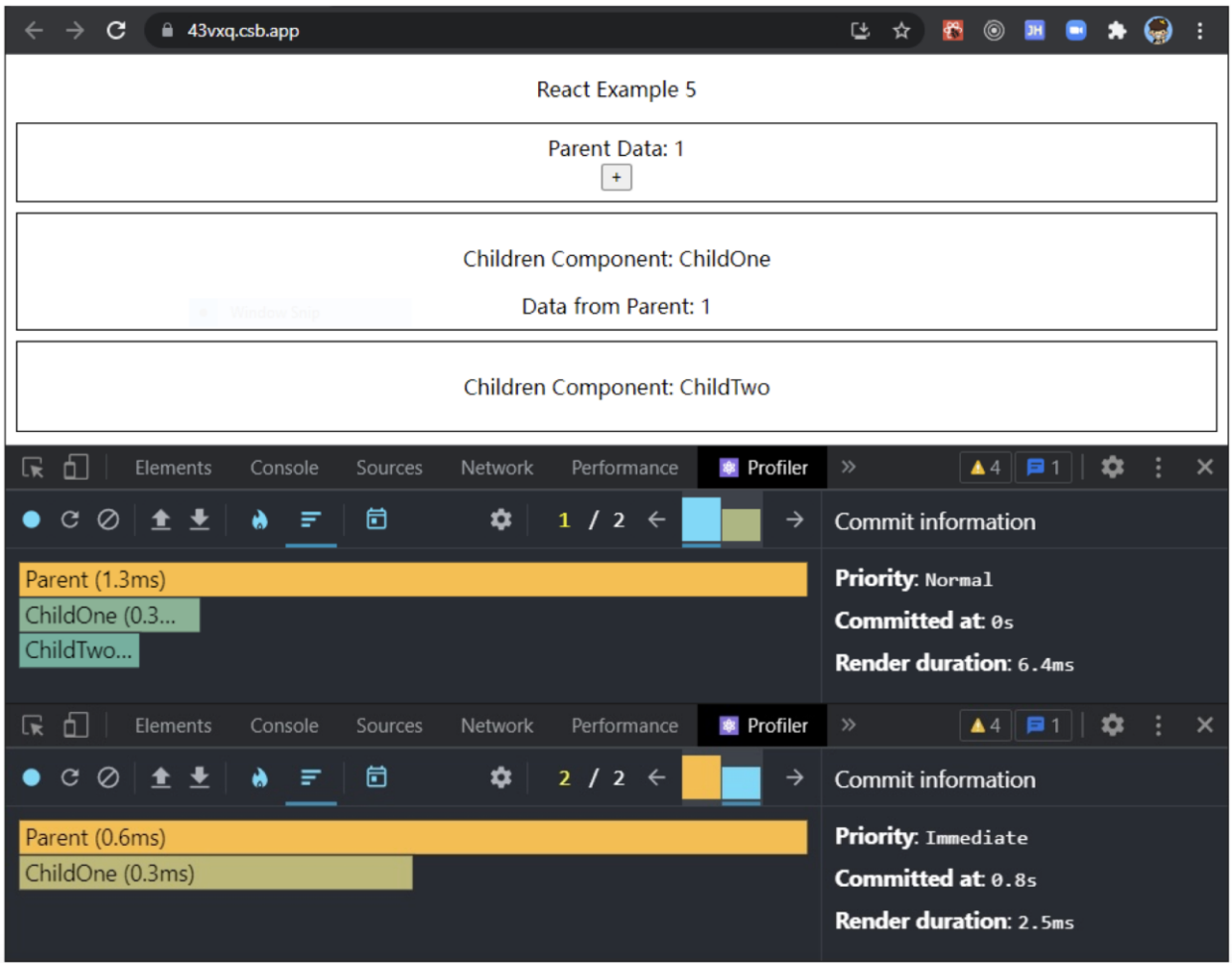



Рисунок 3.5 Результат профілювання для прикладу React 5 у Додатку В

[24] У такому випадку розробники можуть перезаписати метод shouldComponentUpdate з власним алгоритмом глибокого порівняння для властивостей та стану. Таким чином, shouldComponentUpdate може порівнювати ті властивості props, які використовуються дочірнім компонентом для контролю перерендерингу чи його пропуску. У прикладі

React 6 у Додатку В, ChildOne успадковується від звичайного компонента React, але метод shouldComponentUpdate ChildOne реалізований для порівняння this.props.value.number і nextProps.value.number. Рисунок 8 показує, що обидва дочірні компоненти, ChildOne і ChildTwo, не перерендерюються після натискання кнопки.

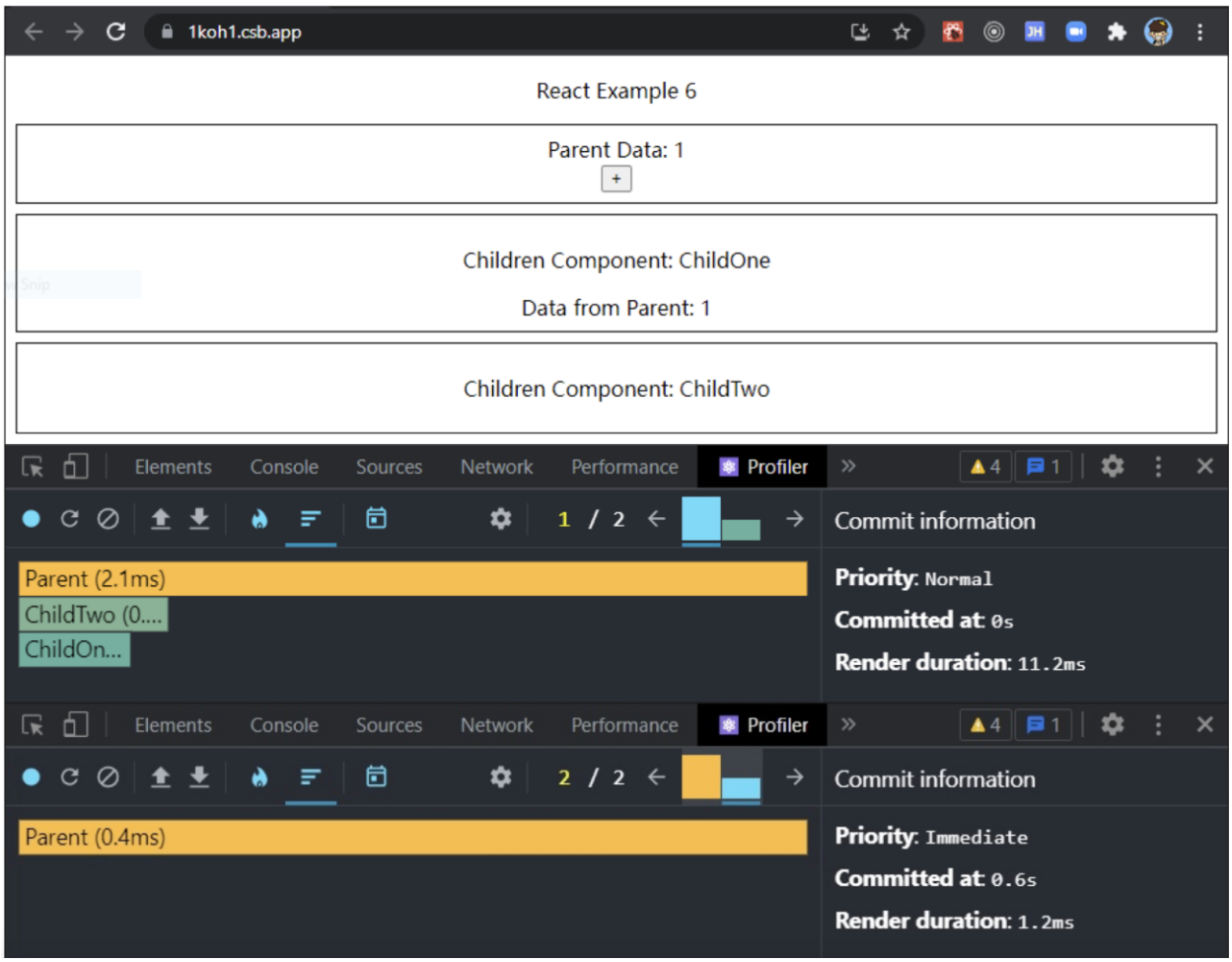



Рисунок 3.6 Результат профілювання для прикладу React 6 у Додатку В

Хоча shouldComponentUpdate допомагає пропускати непотрібний перерендеринг компоненту, в нього є певний недолік. Якщо властивості чи стан компонента мають великі дані, shouldComponentUpdate потрібно витрачати час на виконання порівняння кожного разу, коли компонент намагається перерендеритися. Це також може знизити продуктивність додатка.

*useCallback*

Кожного разу, коли батьківський компонент оновлюється, вивідна функція матиме новий посилання. Отже, стратегії PureComponent і React.memo будуть невдалими. У прикладі React 7 у Додатку B, MemoedIncrement та MemoedMultiply - це кешовані версії компонентів Increment і Multiply. Коли клікається MemoedIncrement або MemoedMultiply, викликається handleIncrement або handleMultiply для збільшення incrementValue на 1 або множення multiplyValue на 3. Рисунок нижче показує результати профайлера для трьох етапів. По-перше, сторінка була завантажена, всі три компоненти були рендерені. По-друге, було натиснуто кнопку збільшення, всі три компоненти були перерендерені. По-третє, було натиснуто кнопку множення, всі три компоненти знову були перерендерені. Результати показують, що посилання на функції оновлюються кожного разу, коли батьківський компонент оновлюється. Це призводить до того, що дочірній компонент, який отримує функції як властивості від батьківського компонента, буде перерендерений, навіть якщо дочірній компонент кешований за допомогою React.memo.

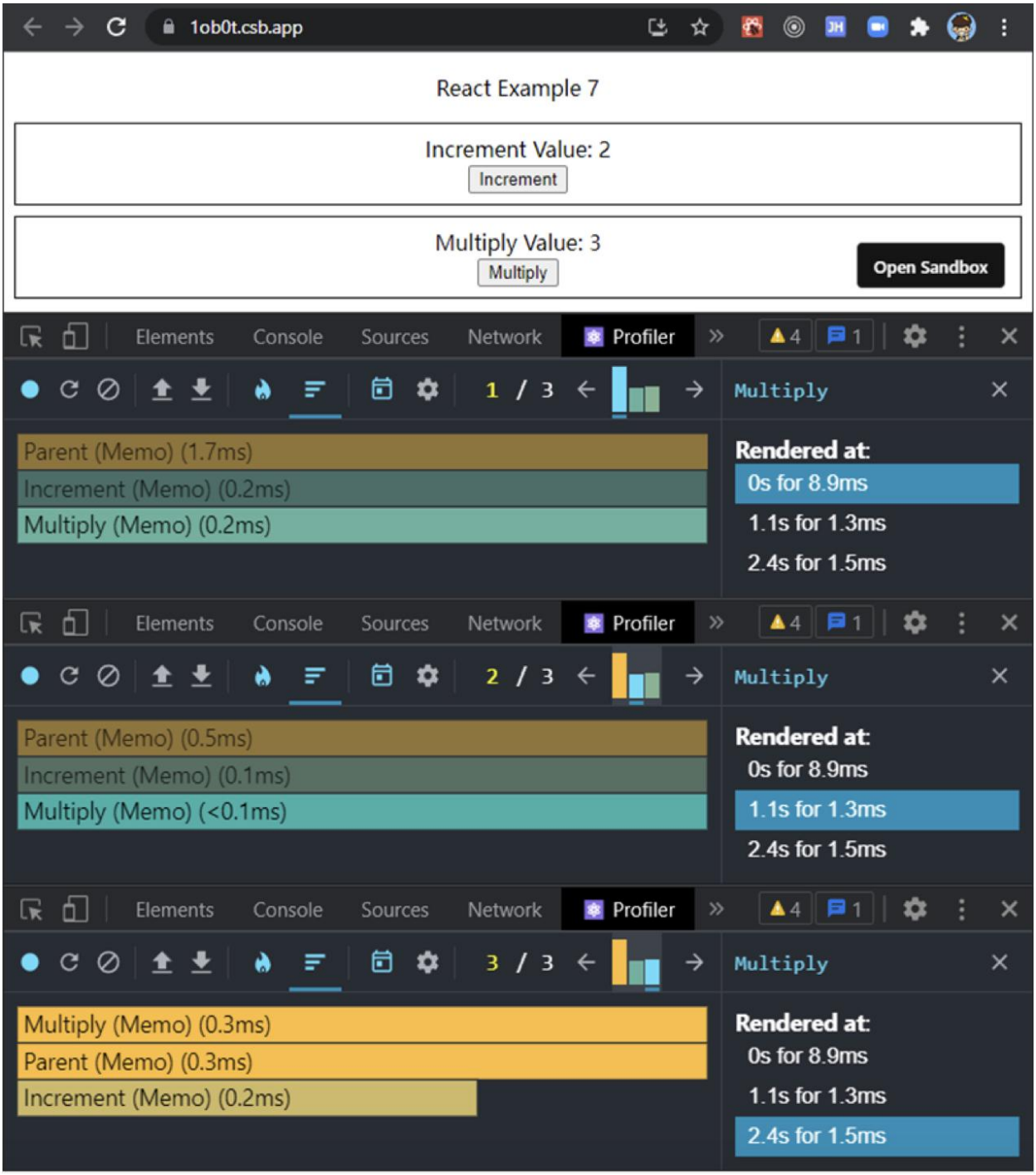



Рисунок 3.7 Результат профілювання для прикладу React 7 у Додатку В

[25] useCallback - це Hook у React, який допомагає генерувати кешовану версію функцій. useCallback приймає функцію та масив залежностей в якості параметрів. Меморизація є механізмом кешування; useCallback запам'ятовує та кешує функцію, яка передається як перший параметр useCallback. І useCallback повертає кешовану версію функції.[2]

Кешована версія функції оновлюється лише тоді, і тільки тоді, коли змінилася одна залежність.

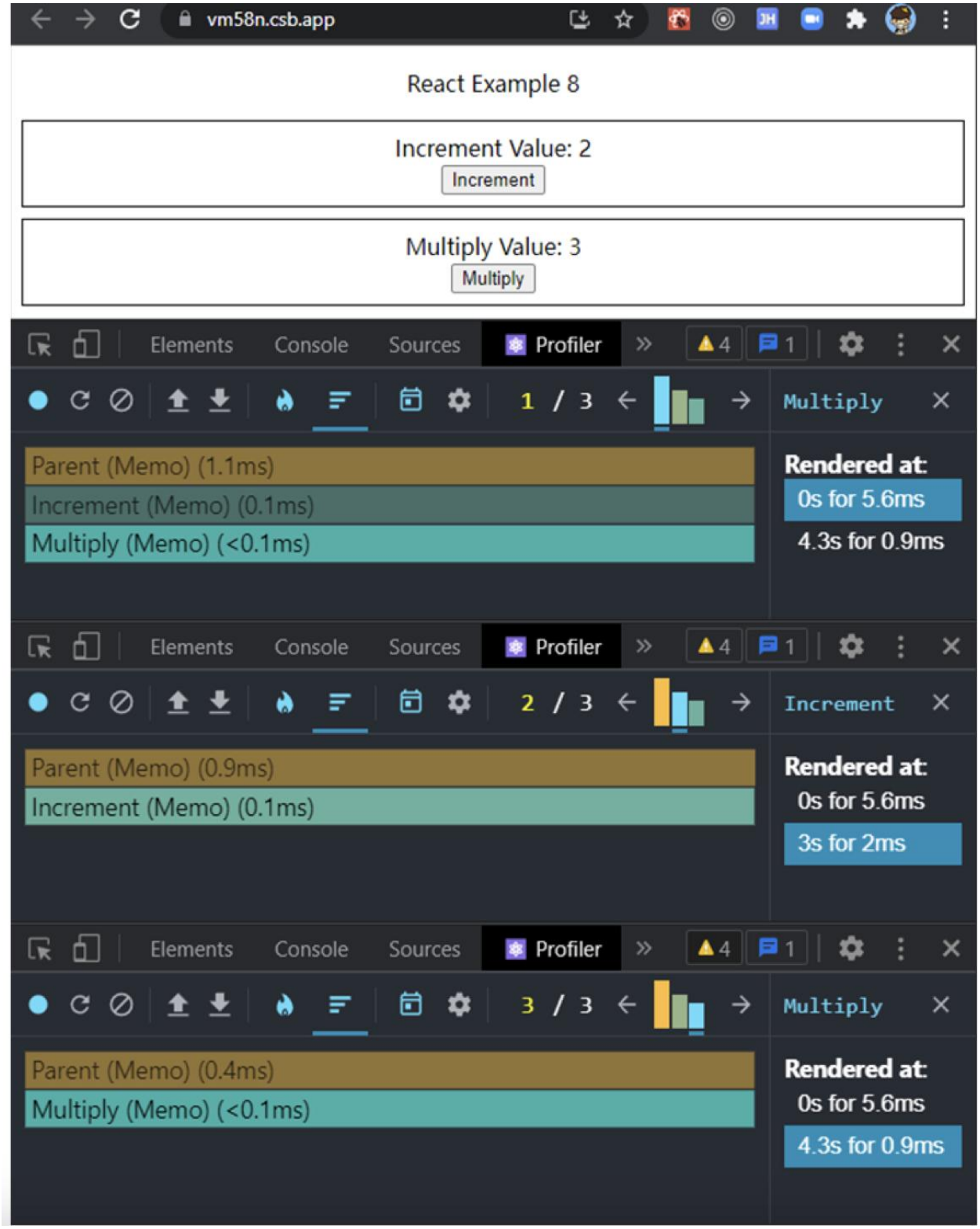



Рисунок 3.8 Результат профілювання для прикладу React 8 у Додатку В

Як показано у React Прикладі 8 у Додатку В, як handleIncrement, так і handleMultiply є кешованими версіями функцій і залежать від incrementValue та multiplyValue, тому посилання на handleIncrement або handleMultiply буде змінено лише в разі оновлення incrementValue чи multiplyValue. Рисунок 10 показує результати профайлера для трьох кроків. По-перше, сторінка була завантажена, всі три компоненти були рендерені. По-друге, було натиснуто кнопку збільшення. Батьківський компонент та компонент Increment були перерендерені, але не Multiply. По-третє, було натиснуто кнопку множення. Батьківський компонент та компонент Multiply були перерендерені, але не Increment. Таким чином, useCallback можна використовувати для генерації стабільної зворотного виклику, щоб уникнути непотрібного перерендерення дочірніх компонентів у батьківському компоненті.

*useMemo*

[26] useMemo - це вбудований Hook у React для кешування результату обчислення між викликами функції та між рендерами. useMemo приймає функцію, яка повертає результат обчислення, як перший параметр, і масив залежних змінних як другий параметр. Результат обчислення буде переобчислений лише тоді, і тільки тоді, коли зміниться одна залежність. Ця оптимізація призначена для пропускання дорогих обчислень кожного рендера.

Як показано у React Прикладі 9 у Додатку В, є дві кнопки "Increment" та "Додати число". Якщо натискана кнопка збільшення, incrementValue збільшиться на 1. Якщо натискана кнопка "Додати число", функція pushNumber додасть число до nums, і largestNum буде переобчислено за допомогою функції getLargestNum.

Фігура 11 показує логи React Прикладу 9 у Додатку В. Після завантаження сторінки функція getLargestNum була викликана і повернула найбільше число масиву nums у largestNum. Потім було натискано кнопку Increment, і функція getLargestNum викликалася для повторного обчислення

largestNum. Після цього було натискано кнопку "Додати число", до масиву nums було додано число. Функція getLargestNum знову викликалася для обчислення нового largestNum для нового nums. У React Прикладі 9 є проблема з продуктивністю: largestNum перераховується, коли масив не був оновлений.

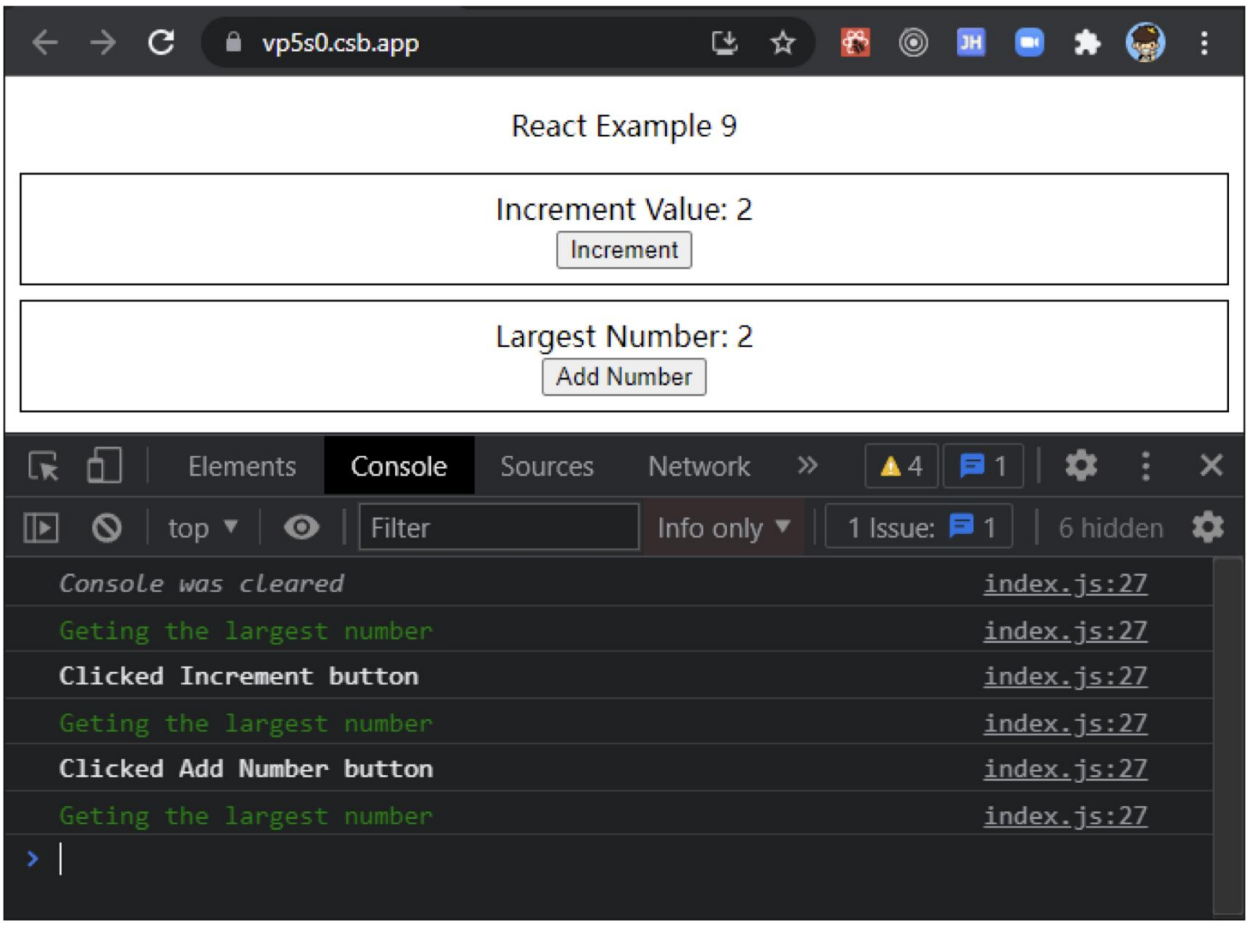



Рисунок 3.9 Логи React Прикладу 9 у Додатку В

Для уникнення проблеми повторного обчислення, приклад може бути вдосконалений за допомогою useMemo, як показано у React Прикладі 10 у Додатку В. Різниця між React Прикладом 9 та React Прикладом 10 полягає в тому, що largestNum є кешованим значенням, яке повертається з useMemo. useMemo приймає функцію, яка повертає результат функції getLargestNum як перший параметр, і масив, який містить масив nums, як другий параметр. largestNum буде оновлено в залежності від масиву nums. Фігура 12, яка є логами React Прикладу 10, показує, що функція getLargestNum не була

викликана після натискання кнопки збільшення. Однак після натискання кнопки "Додати число" масив nums було оновлено. Функція getLargestNum була викликана. Іншими словами, largestNum перераховується лише тоді, коли масив оновлюється. Таким чином, useMemo може допомогти уникнути непотрібних повторних обчислень.

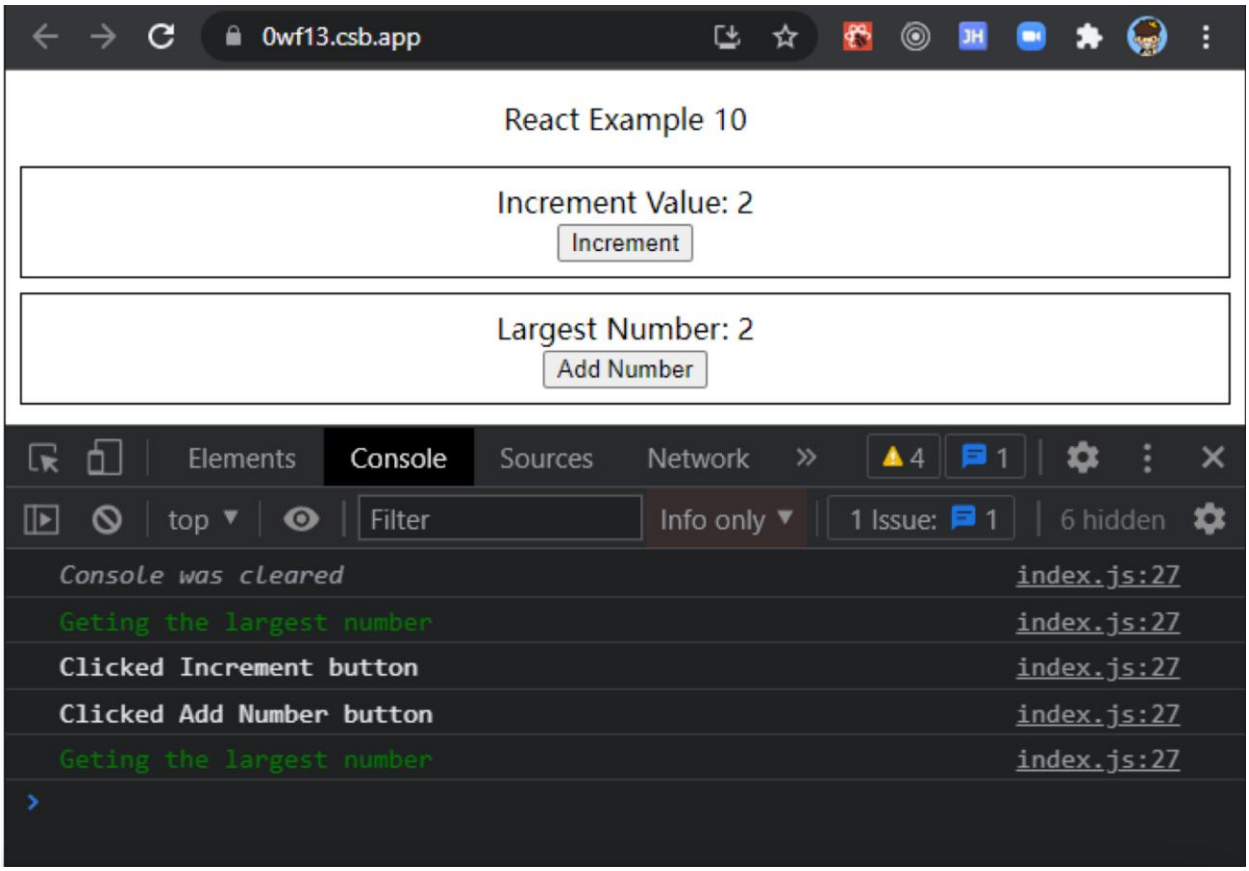



Рисунок 3.10 Логи React Прикладу 10 у Додатку В

**Важливість ключової властивості списку**

[27] Під час реальної розробки розробники часто мають потребу відображення списку компонентів на сторінці. Наприклад, якщо потрібно відобразити дані в таблиці, ми можемо використовувати метод `map` для створення масиву компонентів рядків та їх відображення на сторінці. У React Прикладі 11 при натисканні кнопки "unshift" додаватиметься довжина поточного списку до його початку. Метод `list.map` створить масив `MemorizedBlock`, який відображає числовий блок на сторінці. На Рисунку 13 видно, що при кожному натисканні кнопки "unshift" весь список

компонентів `MemorizedBlock` перерендерюється. Це є серйозною проблемою продуктивності. Після останнього натискання кнопки "unshift", коли було додано новий елемент, React виконав два рази оновлення DOM та один раз створення DOM. Припускаючи, що рендеринг елемента займає 5 мс, і кількість елементів у списку - 200, після натискання кнопки "unshift" React повинен виконати 200 раз оновлення DOM та один раз створення DOM. Отже, додавання нового елемента на сторінку обійдеться додатковими 200 × 5 мс = 1 000 мс = 1 с.

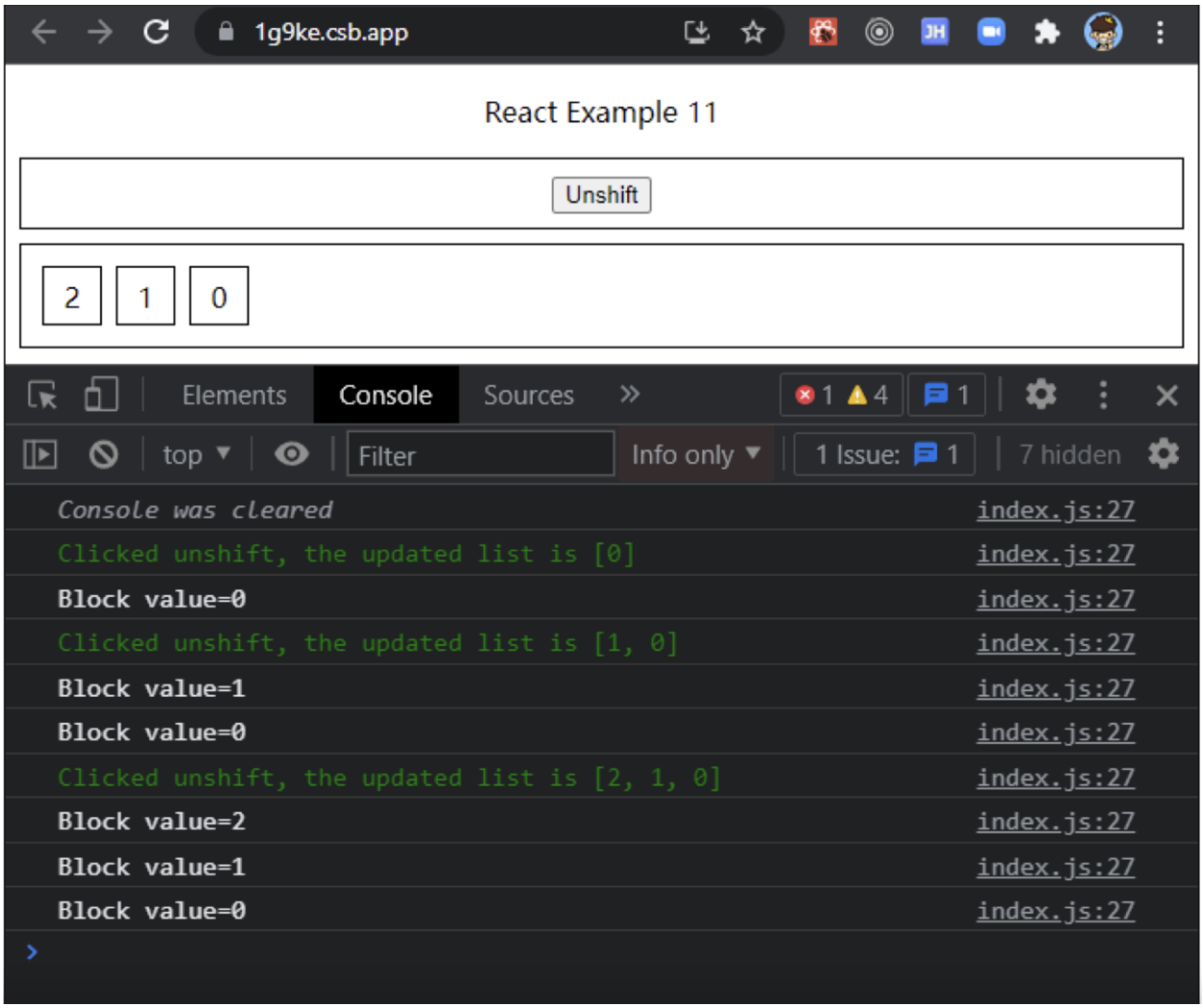



Рисунок 3.11 Логи React Прикладу 11 у Додатку В

Оптимізація цих проблем продуктивності полягає в використанні властивості ключа. Для кожного елемента у списку ми можемо створити унікальний ідентифікатор і використовувати його як властивість ключа для

кожного компонента елемента. У React Прикладі 12 для кожного елемента у списку є унікальний ідентифікатор, який генерується за допомогою Date.now(), що повертає кількість мілісекунд, пройдених з 1 січня 1970 року UTC. Унікальний ідентифікатор призначається властивості ключа для кожного компонента `MemorizedBlock`. На Рисунку 14 видно, що при кожному натисканні кнопки "unshift" перерендерюється лише компонент нового елемента.

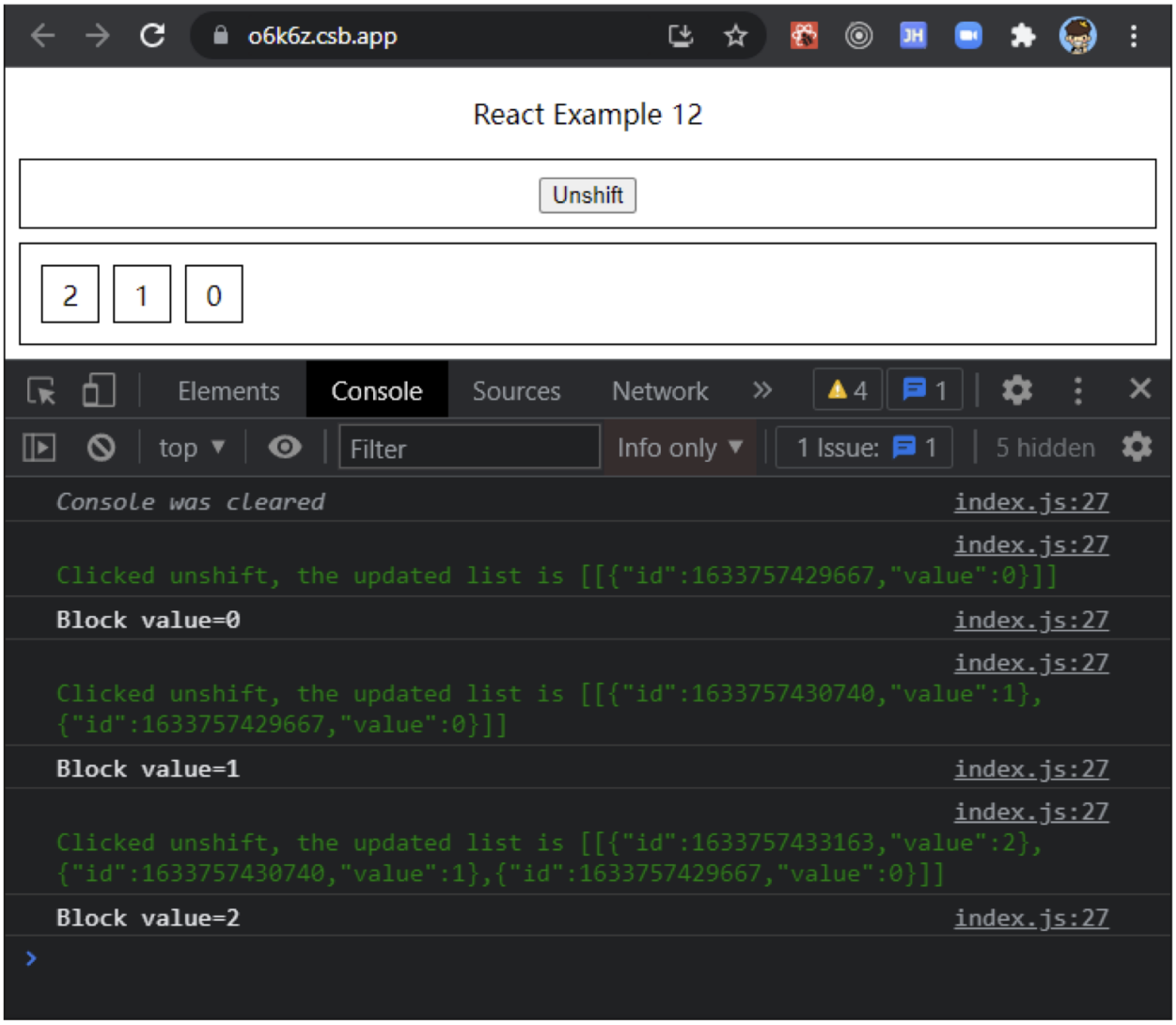



Рисунок 3.12 Логи React Прикладу 12 у Додатку В

[28] Властивість ключа допомагає React ідентифікувати, який елемент був змінений. Після останнього натискання кнопки "unshift" рендерився лише новий елемент із значенням 2. Під час етапу узгодження React виконує алгоритм порівняння віртуального DOM. React виявив, що є два вузла

віртуального DOM з ключовими значеннями 1633757429667 та 1633757430740, які не були змінені і не потребують оновлення. Також React виявив, що вузол віртуального DOM із ключовим значенням 1633757433163 не існує, тому React потрібно було створити вузол віртуального DOM лише для ключового значення 1633757433163. Порівняно з React Прикладом 11, де не використовується властивість ключа, використання властивості ключа дозволяє заощадити операції оновлення для існуючих елементів та витратити операції створення лише для нових елементів.

React офіційно рекомендує використовувати ідентифікатор кожного елемента як властивість ключа компонента для досягнення вищезазначеної оптимізації. Також React не рекомендує використовувати індекс кожного елемента як ключ. При використанні індексу як ключа, якщо ми додаємо новий елемент до початку списку, весь список компонентів буде перерендерений, оскільки індекс нового елемента дорівнює 0, і індекси інших елементів збільшуються на 1. Таким чином, використання індексу як ключа не може досягти оптимізаційної мети.

Однак не завжди краще використовувати ідентифікатор, ніж індекс у всіх сценаріях. У звичайному списку зі сторінкованою навігацією, ідентифікатори елементів списку на першій сторінці відрізняються від ідентифікаторів на другій сторінці. Припустимо, що кожна сторінка відображає три елементи списку, як показано на Рисунку 15, і кожен елемент списку має ключ із

унікальним ідентифікатором.

```
<!-- First Page List Items in Virtual DOM -->
<li key="a">dataA</li>
<li key="b">dataB</li>
<li key="c">dataC</li>

<!-- Second Page List Items in Virtual DOM -->
<li key="d">dataD</li>
<li key="e">dataE</li>
<li key="f">dataF</li>
```

Рисунок 3.13 Віртуальний DOM списку зі сторінкованою навігацією елементів із використанням ID як ключа

[29] Коли перемикається на другу сторінку, React видалить всі вузли DOM для першої сторінки і створить вузли для другої сторінки, оскільки всі теги `li` мають різні значення ключів. Таким чином, використання унікального ідентифікатора як властивості ключа для кожного елемента списку не може досягти оптимізаційної мети. У цьому сценарії використання індексу як ключа є кращим, як показано на Рисунку 16. При перемиканні на другу сторінку, React повинен оновити всі вузли DOM для списку зі сторінкованою навігацією, оскільки всі теги `li` мають однакові значення ключів. Порівняно з прикладом на Рисунку 15, приклад на Рисунку 16 коштує на 3 операції менше.

```
<!-- First Page List Items in Virtual DOM -->
<li key="0">dataA</li>
<li key="1">dataB</li>
<li key="2">dataC</li>

<!-- Second Page List Items in Virtual DOM -->
<li key="0">dataD</li>
<li key="1">dataE</li>
<li key="2">dataF</li>
```

Рисунок 3.14 Віртуальний DOM списку зі сторінкованою навігацією елементів із використанням індексу як ключа

Незважаючи на вищезазначений сценарій, React офіційно рекомендує використовувати ID як значення ключа для кожного елемента. Є дві причини такої рекомендації:

1. При видаленні, вставці або сортуванні елементів у список ефективніше використовувати властивість ключа із унікальним ID. Операції перелистування зазвичай супроводжуються запитами до API, і операції DOM вимагають набагато менше часу, ніж запити до API. Тому вплив використання ID або індексу як властивості ключа на користувацький досвід у цьому сценарії майже не відчутний.

[30] 2. Властивість ключа із унікальним ID може зберігати стан компонента елемента списку, який відповідає ID. Наприклад, кожен рядок у таблиці має два стани: "ідеальний" та "редагування". Спочатку всі рядки знаходяться в стані "ідеальний". Користувач натискає кнопку "редагування" для першого рядка та переходить у стан "редагування". Потім користувач перетягує другий рядок і переміщає його на першу позицію в таблиці. Якщо ця таблиця використовує індекс як ключ, стан першого рядка залишиться в стані "редагування". Однак користувач хоче редагувати другий рядок, що не відповідає очікуванням користувача. Хоча цю проблему можна вирішити, додавши властивість для ідентифікації стану елемента до об'єкта даних елемента та передаючи цю властивість у компонент як `props`, це простіше вирішити, використовуючи унікальний ID як ключ.

# ВИСНОВКИ

У ході даного дослідження було вивчено та висвітлено ключові аспекти оптимізації React застосунків. Розглянувши різноманітні підходи та інструменти, які надає React для підвищення продуктивності, ми визначили найефективніші методи, що можуть суттєво поліпшити досвід користувача та знизити навантаження на візуальний шар застосунку.

Один із ключових висновків полягає в тому, що використання `React.memo`, `PureComponent`, та `shouldComponentUpdate` може значно зменшити кількість непотрібних перерендерень компонентів, забезпечуючи більшу швидкодіючу інтерактивність. Використання `useCallback` та `useMemo` дозволяє ефективно керувати кешуванням функцій та результатів обчислень, що також впливає на продуктивність додатку.

Особливий акцент був зроблений на правильному використанні ключів у React. Використання унікального ідентифікатора (ID) як значення ключа для кожного елемента списку дозволяє уникнути зайвого оновлення DOM та забезпечити ефективні операції вставки, видалення та сортування елементів у списку.

Ця робота надає практичні поради та конкретні приклади, які можна використовувати під час розробки React-застосунків. Впровадження вивчених підходів дозволить створювати продуктивні та високоефективні додатки, що особливо важливо у вимогливому світі веб-розробки.

Загалом, ця робота не лише спрямована на розуміння технічних аспектів оптимізації React-застосунків, а й покликана надати розробникам конкретні інструменти та знання для покращення продуктивності своїх проектів. У майбутньому це дослідження може знайти застосування в розробці великої кількості проектів, забезпечуючи оптимальну ефективність та задоволення користувачів. Його результати можуть виявитися особливо корисними для тих, хто працює над великими та розширеними React-

застосунками, де оптимізація грає ключову роль у забезпеченні стабільної та швидкої роботи додатків.

# ПЕРЕЛІК ПОСИЛАНЬ


1. What is JavaScript? - Learn web development | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript
2. https://www.w3schools.com/js/js_history.asp
3. https://codeinstitute.net/global/blog/advantages-of-javascript/
4. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks
5. https://www.techmagic.co/blog/why-we-use-react-js-in-the-development/
6. https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs
7. https://blog.hubspot.com/website/react-js
8. https://www.netguru.com/glossary/react-native
9. https://programmingwithmosh.com/javascript/react-lifecycle-methods/
10. https://legacy.reactjs.org/docs/hooks-overview.html
11. https://www.geeksforgeeks.org/difference-between-virtual-dom-and-real-dom/
12. https://www.codementor.io/blog/react-optimization-5wiwjnf9hj
13. https://medium.com/@housecor/handling-state-in-react-four-immutable-approaches-to-consider-d1f5c00249d5
14. https://www.freecodecamp.org/news/handling-state-in-react-four-immutable-approaches-to-consider-d1f5c00249d5/
15. https://ireadyoulearn.info/2020/09/13/learn-about-react-pure-components-shallow-comparison-and-common-pitfalls/
16. https://javascript.info/mouse-events-basics
17. https://ru.react.js.org/docs/render-props.html
18. https://elitex.systems/blog/react-vs-vue-javascript-framework/
19. https://stackoverflow.com/questions/42364838/incrementing-state-value-by-one-using-react
20. https://www.merixstudio.com/blog/redux-vs-context-vs-local-component-state/
21. reactjs - Why React.Memo() keeps rendering my component - Stack Overflow
22. https://legacy.reactjs.org/docs/react-api.html
23. https://www.g2i.co/blog/understanding-the-objects-are-not-valid-as-a-react-child-error-in-react
24. https://dev.to/rthefounding/optimizing-re-renders-with-shouldcomponentupdate-5862
25. https://dev.to/spukas/how-to-usecallback-2nai
26. https://dev.to/lorenzojkrl/react-usememo-3faf
27. https://scrimba.com/articles/react-list-array-with-map-function/

28. https://ia801507.us.archive.org/26/items/GameProgramming/obect-oriented-programming-3rd-edition.pdf
29. https://sentry.io/answers/understanding-unique-keys-for-array-children-in-react-js/
30. https://www.freecodecamp.org/news/full-stack-project-tutorial-create-a-notes-app-using-react-and-node-js/

ДОДАТОК А

```
//Function Component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

```
//Class Component
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

```
// Function Component UseState
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, []);

  const handleClick = () => {
    setCount((prevCount) => prevCount + 1);
  };
  return (
```

```
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
      </button>
    </div>
  );
}
```

```
//Function Component useEffect
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, []);

  const handleClick = () => {
    setCount((prevCount) => prevCount + 1);
  };

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
```

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

ДОДАТОК Б

Приклад 1

```
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]);

  const handleClick = () => {
    setCount((prevCount) => prevCount + 1);
  };

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
      </button>
    </div>
  );
}
```

Приклад 2

```
class App extends React.Component {
  state = {
    value: 1
  };

  handleClick = () => {
    this.setState((state) => {
      return {
        value: state.value + 1
      };
    });
  };

  render() {
    console.log("Trick render");
    return (
      <div className="App">
        <p>Example 00</p>
        <div className="box">
          <div>{this.state.value}</div>
        </div>
        <button onClick={this.handleClick}>+</button>
      </div>
    );
  }
}
```

Приклад 3

```
import React from "react";

class ChildOne extends React.Component {
  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p>Children Component: ChildOne</p>
        <div>Data from Parent: {value}</div>
      </div>
    );
  }
}

class ChildTwo extends React.Component {
  render() {
    return (
      <div className="box">
        <p>Children Component: ChildTwo</p>
      </div>
    );
  }
}

class Parent extends React.Component {
  state = { value: 1 };
```

```
  handleClick = () => {
    this.setState({ value: 1 });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 3</p>
        <div className="box">
          <div>Parent Data: {this.state.value}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
        <ChildOne value={this.state.value} />
        <ChildTwo />
      </div>
    );
  }
}

export default Parent;
```

Приклад 4

```
import React from "react";

class Childone extends React.PureComponent {
  render() {
    const { value } = this.props;
```

```
    return (
      <div className="box">
        <p>Children Component: Childone</p>
        <p>PureComponent</p>
        <div>Data from Parent: {value}</div>
      </div>
    );
  }
}

const ChildTwo = ({ value }) => (
  <div className="box">
    <p>Children Component: ChildTwo</p>
    <p>React memo</p>
    <div>Data from Parent: {value}</div>
  </div>
);

const MemorizedChildTwo = React.memo(ChildTwo);

class Parent extends React.Component {
  state = { value: 1 };

  handleClick = () => {
    this.setState({ value: this.state.value + 1 });
  };

  render() {
```

```
    return (
      <div className="App">
        <p>React Example 4</p>
        <div className="box">
          <div>Parent Data: {this.state.value}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
        <Childone value={this.state.value} />
        <MemorizedChildTwo value={this.state.value} />
      </div>
    );
  }
}

export default Parent;
```

Приклад 5

```
import React from "react";

class ChildOne extends React.PureComponent {
  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p>Children Component: ChildOne</p>
        <div>Data from Parent: {value.number}</div>
      </div>
```

```
    );
  }
}

class ChildTwo extends React.PureComponent {
  render() {
    return (
      <div className="box">
        <p>Children Component: ChildTwo</p>
      </div>
    );
  }
}

class Parent extends React.Component {
  state = { value: { number: 1 } };

  handleClick = () => {
    this.setState({ value: { number: this.state.value.number + 1 } });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 5</p>
        <div className="box">
          <div>Parent Data: {this.state.value.number}</div>
          <button onClick={this.handleClick}>+</button>
```

```
        </div>
        <ChildOne value={this.state.value} />
        <ChildTwo />
      </div>
    );
  }
}

export default Parent;
```

Приклад 6

```
import React from "react";

class Parent extends React.Component {
  state = { value: { number: 1 } };

  handleClick = () => {
    this.setState({ value: { number: this.state.value.number + 1 } });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 6</p>
        <div className="box">
          <div>Parent Data: {this.state.value.number}</div>
          <button onClick={this.handleClick}>+</button>
```

```
        </div>
        <ChildOne value={this.state.value} />
        <ChildTwo />
      </div>
    );
  }
}

class ChildOne extends React.Component {
  shouldComponentUpdate(nextProps) {
    return this.props.value.number !== nextProps.value.number;
  }

  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p>Children Component: ChildOne</p>
        <div>Data from Parent: {value.number}</div>
      </div>
    );
  }
}

class ChildTwo extends React.PureComponent {
  render() {
    return (
      <div className="box">
```

```
        <p>Children Component: ChildTwo</p>
      </div>
    );
  }
}

export default Parent;
```

# Дослідження методів оптимізації веб-застосунків, на базі технології React

Рогов Олександр

## Мета роботи

Дослідження методів оптимізації веб-застосунків, на базі технології React

## Об'єкт дослідження

React-застосунки, зосереджені на виявленні й реалізації оптимальних методів для вирішення проблем, пов'язаних із продуктивністю.

## Предмет дослідження

Методи, бібліотеки та підходи, які дозволяють забезпечити високу ефективність та швидкодію React-застосунків.

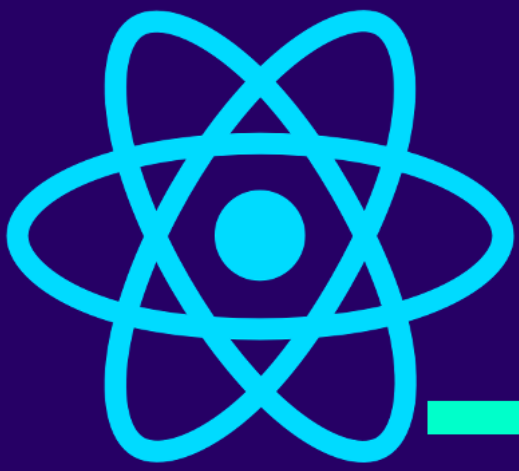
## Наукова новизна

Робота розкриває новий погляд до оптимізації React-застосунків через ефективне використання хуків та ключів.

JS

JavaScript - це мова сценаріїв або програмування, яка дозволяє реалізувати складні функції на веб-сторінках - кожного разу, коли веб-сторінка відображає не статичну інформацію: оновлення контенту, інтерактивні карти, анімовані 2D/3D графіку, прокручувані відео і т.д.

## REACT . JS



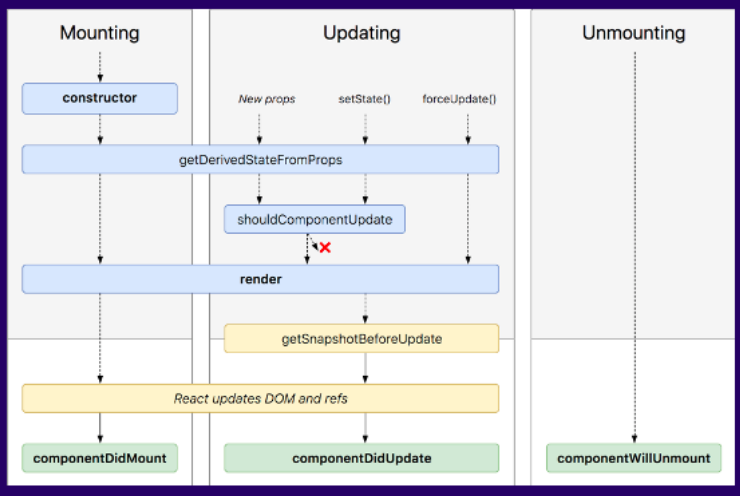
React - це бібліотека JavaScript для розробки інтерфейсів користувача. Вона дозволяє створювати високоефективні та зручні для розробки веб-додатки, основані на компонентах. React розроблений Facebook і використовується для побудови великої кількості веб-сайтів та додатків.

# Як працює React.js

React використовує односторінкову архітектуру, що дозволяє оновлювати лише потрібні частини веб-сторінки за допомогою віртуального DOM, уникнувши повних перезавантажень і забезпечуючи ефективніше відображення змін у динамічному інтерфейсі користувача.

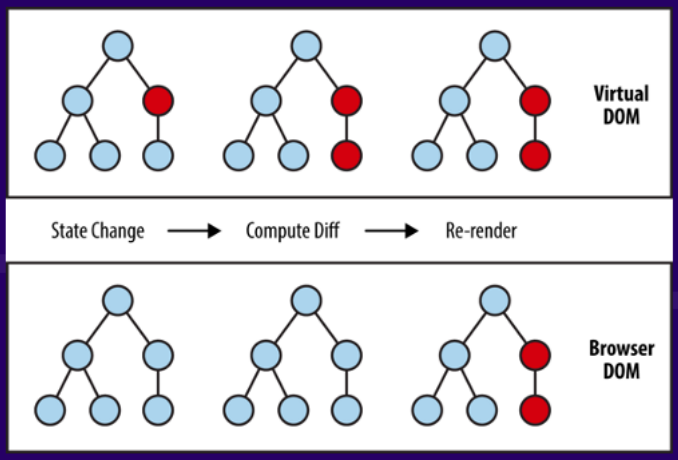
# Життєвий цикл React

У React життєвий цикл компонентів складається з чотирьох основних етапів: ініціалізація, монтування, оновлення і демонтування. Кожен компонент пройде ці етапи під час свого існування. Це дозволяє налаштовувати поведінку компонента на різних етапах його життєвого циклу відповідно до потреб розробників.



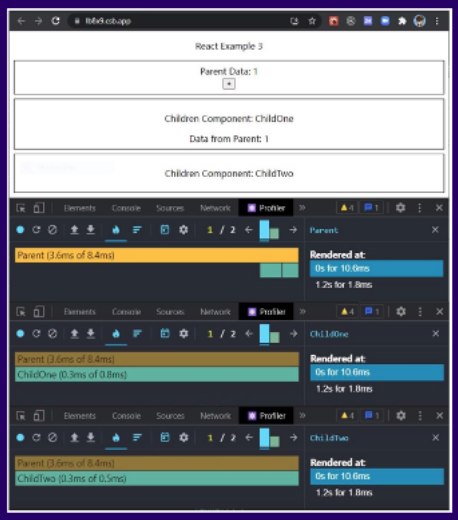
# Реальний та віртуальний DOM

DOM (Об'єктна модель документа) є структурним представленням елементів HTML веб-програми. Зміни в DOM можуть впливати на продуктивність сайту, оскільки кожна зміна вимагає повторного відображення його елементів. Віртуальний DOM (VDOM) використовується для оптимізації цього процесу в бібліотеках, таких як React. Він створює полегшену копію справжнього DOM, яка ефективно оновлюється та покращує продуктивність. Реалізація React використовує функцію `render()` та механізм пакетного оновлення для оптимізації взаємодії з реальним DOM, забезпечуючи ефективні та швидкі зміни інтерфейсу користувача.




# Проблема рендерингу

Центральною концепцією оптимізації додатків у React є зменшення непотрібного перерендерингу компонентів після зміни стану. Коли бізнес-логіка стає складніше і додаток стає великим, ця проблема може стати ще більш значущою. Тому важливо ефективно керувати рендерингом компонентів для забезпечення оптимальної продуктивності.

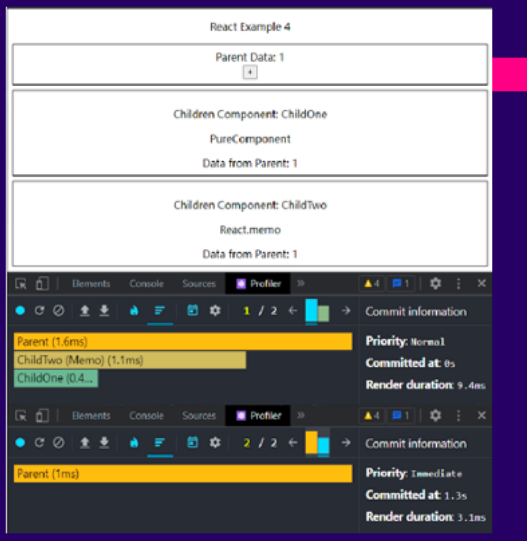


# PureComponent, React.memo

PureComponent та React.memo - оптимізаційні засоби в React для уникнення зайвих перерендерів компонентів.

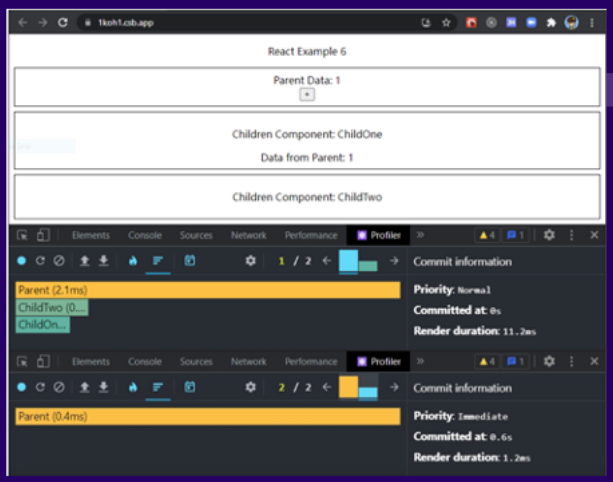
PureComponent використовує поверхневе порівняння властивостей та стану для класових компонентів. Якщо вони не змінені, компонент не перерендерюється.

React.memo застосовує цю логіку до функційних компонентів, здійснюючи поверхневе порівняння властивостей. Якщо вони не змінені, компонент не перерендерюється.




# shouldComponentUpdate

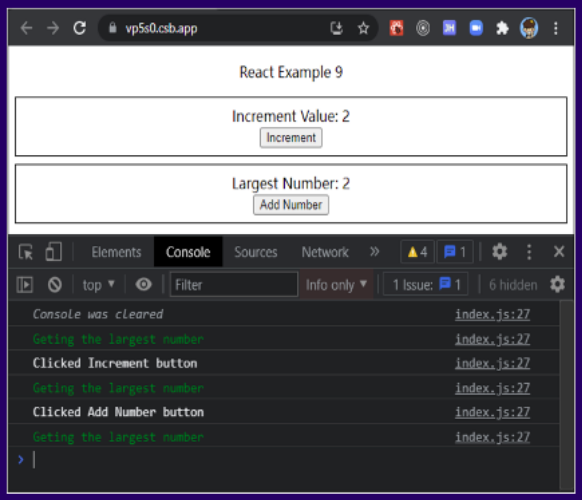
Розробники уникають непотрібних рендерингів дочірніх компонентів, передаючи великі об'єкти через властивості. Метод `shouldComponentUpdate` вирішує цю проблему, порівнюючи необхідні властивості перед рендерингом. У прикладі React 6 ручне управління цим методом забезпечує ефективну оптимізацію, яка уникає зайвого перерендерингу. Однак, цей підхід може знизити продуктивність з великою кількістю об'єктів.



# useMemo

У React 7, кешовані компоненти MemoedIncrement і MemoedMultiply, навіть застосовуючи стратегії PureComponent і React.memo, оновлюють функції при кожному оновленні батьківського компонента. Результат - непотрібний перерендеринг

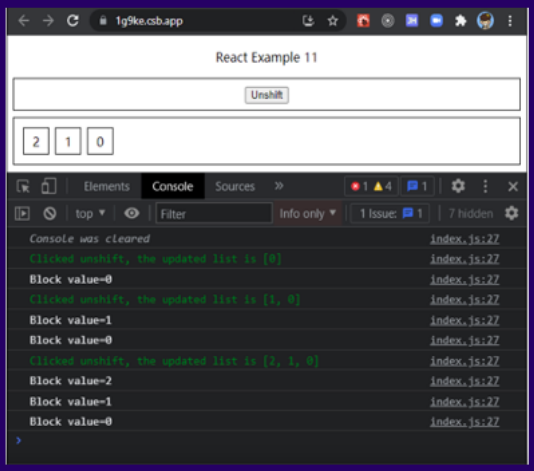
В React 8 використовується useCallback для кешування функцій відповідно до їхніх залежностей. Це дозволяє уникнути непотрібного перерендерення дочірніх компонентів при оновленні батьківського




# ОПТИМІЗАЦІЯ СПИСКІВ

В React ключі важливі при відображенні списків компонентів. У Прикладі метод `map` при додаванні нового елемента призводить до зайвого перерендерингу всіх компонентів.

Оптимізація: використання унікального ідентифікатора кожного елемента як ключа уникне зайвих перерендерингів та покращить продуктивність. Рекомендація React - використовувати ідентифікатор елемента як ключ.



# ВИСНОВКИ

Це дослідження висвітлює ключові аспекти оптимізації React-застосунків, надаючи конкретні методи, такі як використання `React.memo`, `PureComponent`, `useCallback`, та правильне використання ключів для поліпшення продуктивності та досвіду користувача. Робота не лише зосереджена на технічних аспектах, а й надає розробникам інструменти та практичні поради для створення ефективних React-застосунків.

# АПРОБАЦІЯ

ВСЕУКРАЇНСЬКА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ «TELECOMMUNICATION: PROBLEMS AND INNOVATION

ВИЗНАЧЕННЯ ПРОБЛЕМИ ОПТИМІЗАЦІЇ ВЕБ-ЗАСТОСУНКІВ, ЩО БАЗУЮТЬСЯ НА ТЕХНОЛОГІЇ REACT

ЖИТТЄВИЙ ЦИКЛ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ JIRA

Дякую за увагу!