Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Кутюка Олега Ігоровича*
	(ПІБ)
академічної групи	*121М-22-3*
	(шифр)
спеціальності	*121 Інженерія програмного забезпечення*
	(код і назва спеціальності)
освітньої програми	*«121 Інженерія програмного забезпечення»*
	(назва освітньої програми)
на тему:	*Дослідження методів оптимізації програмного забезпечення клієнтської частини React веб-застосунків*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинг мовою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*проф. Лактіонов І. С.*			

Рецензент				

Нормоконтролер	*проф. Лактіонов І. С.*			

Дніпро, 2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

(підпис) М.О. Алексєєв (прізвище, ініціали)

« » 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-22-3* (група) *Кутюку Олегу Ігоровичу* (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження методів оптимізації програмного забезпечення клієнтської частини React веб-застосунків*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процеси оптимізації React клієнтських веб-додатків.

Предмет досліджень – методи та засоби оптимізації клієнтських веб-застосунків розроблених за допомогою технології React.

Мета НДР – метою роботи є підвищення ефективності та продуктивності роботи клієнтських веб-застосунків, шляхом виявлення, комбінування та впровадження найбільш ефективних методів оптимізації React веб-додатку.

Вихідні дані для проведення роботи – практичні та експериментальні дослідження для розв'язання задач оптимізації веб-додатків створених за допомогою бібліотеки React.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає у виявлені, досліджені,

модифікації та застосуванні сучасних методів оптимізації програмного забезпечення розробленого за допомогою React, що забезпечить удосконалення існуючих методів оптимізації з мінімальними зусиллями та часовими витратами на рефакторинг вже існуючого клієнтського веб-застосунку.

Практична цінність Отримані в ході дослідження результати роботи можуть застосовуватися для оптимізації веб-застосунків, які зіштовхнулись з проблемою завантаження та відображені клієнтської частини користувачам у віддалених регіонах з повільним інтернет з'єднанням. Результати дослідження можуть бути використані інженерами програмного забезпечення з метою оптимізації React веб-додатків.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень слід представити у формі, яка дозволяє визначити вплив на клієнтський веб-застосунок, розроблений за допомогою фреймворку React, та оцінити ефективність та безпосередній вплив на веб-застосунок пілся використання методів оптимізації, таких як функціональні компоненти, методи мемоізації, ліниве завантаження та розподіл коду.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Знайомство з методами оптимізації React веб-додатків. Опрацювання літературних джерел	04.09.2023-14.09.2023
Збір, дослідження та систематизація інформації щодо створення та оптимізації функціональних веб-застосунків створених на React	25.09.2023-21.10.2023
Розробка дослідження та аналіз результатів після застосування методів оптимізації веб-застосунку за допомогою використання мемоїзації та функціональних компонентів в React.	22.10.2023-10.11.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект після застосування запропонованих методів оптимізації результатів роботи очікується позитивним, оскільки виконання таких робіт не потребуватиме додаткової кваліфікації, та передбачатиме менші витрати часу на здійснення самих. Також після застосування запропонованих

методів планується зменшення розміру самого додатку та його ресурсів, що спричинить скорочення використання серверної пам'яті та знизить кількість вживаного трафіку. Таким чином, ми знизимо фінансові витрати на заробітну плату людям, які виконуватимуть оптимізацію додатку і зменшимо витрати на утримання самого веб-додатку.

Соціальний ефект від реалізації результатів роботи очікується позитивним, завдяки значній оптимізації розміру веб-додатків. Це відкриє можливість використання інформаційних продуктів у віддалених регіонах нашої планети, де існують проблеми зі стабільністю та якістю інтернет-з'єднання. Завдяки цьому значно збільшиться кількість користувачів, які зможуть розв'язувати свої проблеми за допомогою інтернет-технологій у різних куточках нашої земної кулі.

Завдання видав	_____	*Лактіонов І.С.*
	(підпис)	(прізвище, ініціали)
Завдання прийняв до виконання	_____	*Кутюк О.І.*
	(підпис)	(прізвище, ініціали)

Дата видачі завдання: *04.09.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *11.12.2023*

РЕФЕРАТ

Пояснювальна записка: 92 с., 22 рис., 27 джерел, 2 додатки.

Об'єкт розробки: методи оптимізації React клієнтських веб-додатків.

Предмет дослідження: методи та засоби оптимізації клієнтських веб-застосунків розроблених за допомогою технології React.

Мета кваліфікаційної роботи: метою роботи є підвищення ефективності та продуктивності роботи клієнтських веб-застосунків, шляхом виявлення, комбінування та впровадження найбільш ефективних методів оптимізації React веб-додатку.

Методи дослідження. Використання технології оптимізації функціональних компонентів програмного забезпечення створеного за допомогою бібліотеки React.

Наукова новизна заключається у дослідженні методів оптимізації веб-додатків, створених за допомогою React та проведення аналіз впливу методів на швидкість роботи і завантаження клієнтських веб-додатків.

Практичне значення роботи. Отримані в ході дослідження результати роботи можуть застосовуватися для оптимізації веб-застосунків, які зіштовхнулись з проблемою завантаження та відображені клієнтської частини користувачам у віддалених регіонах з повільним інтернетом з'єднанням. Результати дослідження можуть бути використані інженерами програмного забезпечення з метою оптимізації React веб-додатків.

Ключові слова: API, веб-додаток, стан, веб-технології, програмне забезпечення, методи оптимізації, мемоізація, функціональні компоненти, бібліотека React, відкладене завантаження.

ABSTRACT

Explanatory note: 92 pages, 22 pictures, 2 appendices, 27 sources.

Object of research: optimization methods for React client-side web applications.

Subject of research: methods, schemes, and tools for optimizing client-side web applications developed using React technology.

Purpose of Master's thesis: to identify optimization methods for React web applications, apply these methods in practice, and investigate the impact of methods on the performance of the web application.

Research methods. Utilizing the optimization technology for functional components of software created using the React library.

Originality of research is to figure out optimization methods for web applications developed using React and conducting an analysis of the impact of these methods on the speed and loading of client-side web applications.

Practical value of the results. The results obtained during the research can be applied to optimize web applications that face loading and display issues for users in remote regions with slow internet connections. The research findings can be utilized by software engineers for optimizing React web applications.

Keywords: API, web application, web technologies, React framework, server, http-request, http-response, database, optimization methods, memoization, functional components, library, lazy loading.

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

ПК – персональний комп’ютер;

ОС – операційна система;

БД – база даних;

ПЗ – програмне забезпечення.

API – інтерфейс програмування додатків;

UI – клієнтський інтерфейс;

GUI – графічний клієнтський інтерфейс;

SQL – структурована мова запитів;

HTTP – протокол передачі гіпертекстових документів;

ПЗ – програмне забезпечення.

ЗМІСТ

ВСТУП

Актуальність дослідження. З появою веб-додатків у світі інтернет-технологій виникла потреба в їх оптимізації та ефективному керуванні розміром. Початкові етапи розвитку цього напрямку характеризувалися відносною простотою і легкістю вирішення завдань, однак у сучасному світі проблема оптимізації та обмеження розміру веб-додатків стала надзвичайно актуальною.

Незалежно від того, наскільки добре спроєктований веб-додаток спочатку, в процесі його розробки неминуче виникають зміни. Ці зміни можуть бути обумовлені виправленням виявлених помилок під час експлуатації або потребою внести додаткові функціональні зміни. Також, при розробці та тестуванні веб-додатків, важливо враховувати можливість розширення функціоналу та адаптації до нових вимог, що можуть виникнути у майбутньому.

Зазначена проблема підкреслює важливість систематичного контролю за змінами та оптимізацією веб-додатків. Точний і своєчасний контроль набуває критичного значення для забезпечення ефективності та корисності веб-додатків у динамічному середовищі, де зміни є не лише необхідністю, але й ключовим елементом успішного функціонування.

В умовах постійної конкуренції в індустрії інформаційних технологій, на ринку програмного забезпечення з'явилося безліч продуктів, спрямованих на поліпшення зручності та швидкості робочих процесів. Однак проведене дослідження виявило недостатню ефективність існуючих інформаційних методів, засобів і технологій автоматизації, особливо в контексті їх використання у вузькоспеціалізованих проєктах.

Необхідність надання онлайн-послуг стала неодмінною складовою будь-якого проєкту, що передбачає створення мобільного або веб-додатка. Саме веб-додаток є основним представником підприємства в цифровому просторі, оскільки користувачі в основному шукають інформацію про компанію через

пошукові системи в Інтернеті.

Для розробки веб-додатків на стороні клієнта найчастіше використовують бібліотеку React, яка є лідером за статистикою, зазначеною у статті "10 найпопулярніших інструментів для розробки інтерфейсів у 2022 році" [1]. Незважаючи на свою назву, React не є реактивним, що може вважатися недоліком, оскільки в сучасному світі технологій користувачі очікують отримання інформації максимально швидко. Згідно з даними Deloitte [2], навіть зменшення часу завантаження веб-сторінки на 0.1 с може призвести до збільшення конверсії на 9%.

Однак не всі сучасні мобільні пристрої можуть ефективно обробляти велику кількість мобільних додатків, і не всі користувачі бажають встановлювати додатки через обмежену кількість пам'яті свого пристрою або через погане інтернет з'єднання [3]. Тому стає актуальною задача створення оптимізованих веб-додатків. Для досягнення високої продуктивності веб-додатку розробники повинні мати знання та навички в області їхньої оптимізації.

У межах магістерської роботи буде проведено дослідження методів оптимізації веб-додатків на основі React та їх впливу на продуктивність системи. Під продуктивністю розуміється як швидкість завантаження, так і швидкість реакції на дії користувача. Актуальність даної теми визначається потребою у розробці продуктивних веб-додатків в умовах сучасного цифрового середовища.

Мета дослідження: метою роботи є підвищення ефективності та продуктивності роботи клієнтських веб-додатків, шляхом виявлення, комбінування та впровадження найбільш ефективних методів оптимізації React веб-додатку.

Завдання дослідження: Завдання дослідження спрямоване на вивчення та вдосконалення методів оптимізації React веб-додатків. Для досягнення цієї мети передбачено вирішення таких завдань:

1. Розгляд принципів та стратегій оптимізації, визначення ключових

аспектів, що впливають на продуктивність React-додатків.;

2. Дослідити особливості виникнення проблем зі швидкістю роботи клієнтської частини веб-додатків створених за допомогою React.;

3. Вивчення оптимальних підходів до написання коду в React, включаючи оптимізацію рендерингу, управління станом та роботу з компонентами.;

4. Перевірити ефективність застосовуваних методів на практиці та зробити висновки щодо доцільності її створення.

Об’єкт дослідження: методи оптимізації React клієнтських веб-додатків.

Предмет дослідження: методи та засоби оптимізації клієнтських веб-застосунків розроблених за допомогою технології React.

Методи дослідження. Використання технології оптимізації функціональних компонентів програмного забезпечення створеного за допомогою бібліотеки React.

Наукова новизна заключається у дослідженні методів оптимізації веб-додатків, створених за допомогою React та проведення аналіз впливу методів на швидкість роботи і завантаження клієнтських веб-додатків.

Практичне значення роботи. Отримані в ході дослідження результати роботи можуть застосовуватися для оптимізації веб-застосунків, які зіштовхнулись з проблемою завантаження та відображені клієнтської частини користувачам у віддалених регіонах з повільним інтернетом з'єднанням. Результати дослідження можуть бути використані інженерами програмного забезпечення з метою оптимізації React веб-додатків.

Особистий внесок автора:

1. Наукові результати роботи отримані автором самостійно.

2. Вибір методів досліджень і технологій реалізації;

3. Розробка теоретичної частини роботи з дослідження і систематизування знань про існуючі підходи оптимізації клієнтських веб-додатків розроблених за допомогою React;

4. Дослідження та практичне застосування методів оптимізації швидкості роботи клієнтських веб-додатків;

5. Оцінка отриманих результатів.

Структура і обсяг роботи. Робота складається з вступу, трьох розділів і висновків. Містить 92 сторінки, в тому числі 60 сторінок тексту основної частини з 22 рисунками, списку використаних джерел з 27 найменуваннями на 2 сторінках.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1. Дослідження терміну веб-застосунок

Веб-застосунок - визначено як програму для комп'ютера, яка виконує певні завдання через веб-браузер, що діє як клієнт [3]. Це може відбуватися на будь-якому пристрої або платформі, на яких використовується браузер. Сама програма може мати різний характер, від простої дошки оголошень або форми зворотного зв'язку до текстового процесора чи мобільної гри для кількох гравців, яку можна завантажити на мобільний телефон.

Веб-сайт - це сукупність взаємопов'язаних веб-сторінок, доступних за одним доменним ім'ям [3]. Сайт може бути розроблений окремою особою, бізнесом або організацією та має різноманітні цілі, такі як створення візитки, ведення блогу, надання новин і т. д. Основні завдання сайту включають ефективне представлення продуктів та послуг, формування соціального доказу, сприяння брендингу бізнесу, досягнення ділових цілей та розширення підтримки клієнтів [3].

Одна з основних різниць полягає в тому, що взаємодія зі сторінкою на веб-сайті та веб-додатку відбувається на різних рівнях. В той час як веб-сайт включає текстовий і візуальний контент, до якого користувач не може втрутитися, веб-додатки надають користувачеві можливість не лише читати, але й змінювати та додавати інформацію на сторінки.

Прикладом веб-додатку може бути інтернет-магазин, який дозволяє користувачам купувати товари та шукати їх у каталогах. Ще одним цікавим випадком є соціальні мережі, які включають функції блогу, чатів та вмісту, обраного користувачами, а також можливість ділитися цим вмістом.

На сьогодні багато веб-сайтів мають інтерактивний характер через те, що

це подобається користувачам. Тому власники веб-сайтів додають на свої платформи невеликі веб-додатки для забезпечення більш активної взаємодії з відвідувачами.

1.1.1. Архітектура та принципи роботи веб-застосунку

Веб-додаток функціонує у клієнт-серверній архітектурі, де клієнтом є браузер, а сервером — веб-сервер [3]. Логіка роботи веб-додатка розділена між сервером та клієнтом. Зазвичай збереження даних здійснюється на сервері, а обмін інформацією відбувається через мережу Інтернет.

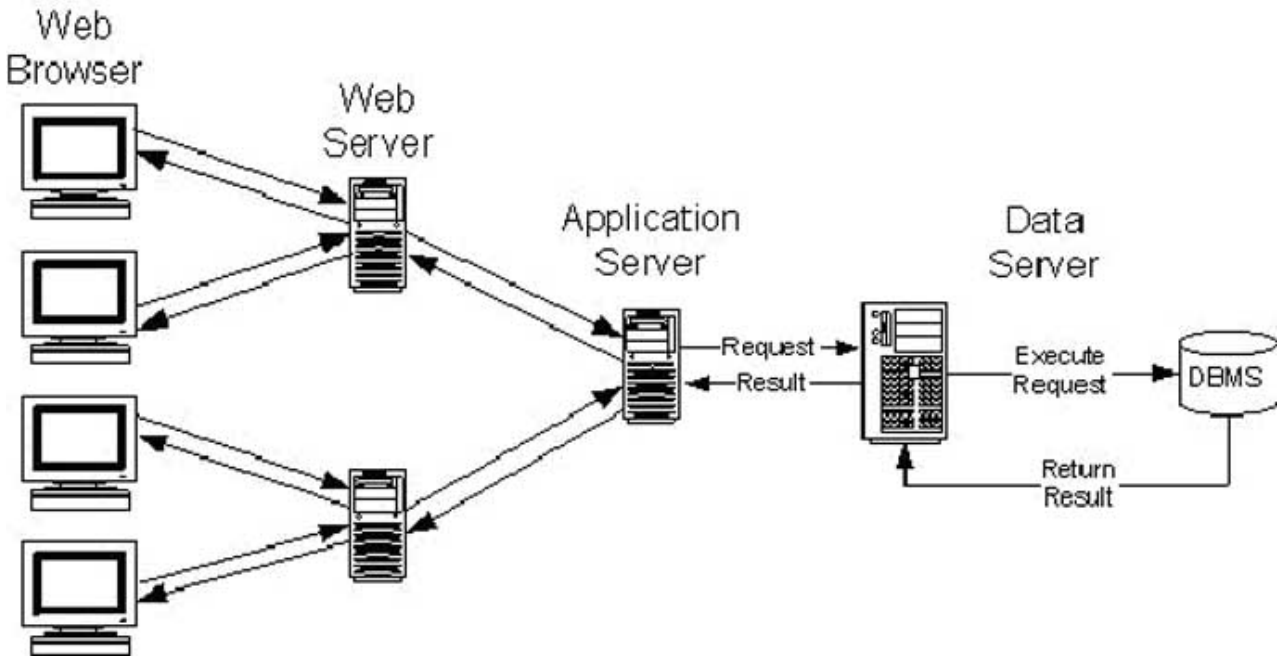

Рис. 1.1. Архітектура веб-застосунку

Серверний код залишається невидимим для клієнта і може реагувати тільки на HTTP-запити з конкретною URL-адресою, не враховуючи будь-якого введення користувача. Зазвичай такий код розробляється мовами програмування та фреймворками, такими як Java, JavaScript (Node.js), Python (Django), PHP,

Ruby On Rails, Java, C# та інші [4].

Клієнтський код, який аналізується браузером користувача, може реагувати на користувацькі введення, і його можна повністю переглядати та редагувати. Файли сервера є недоступними для прямого читання і вимагають обробки HTTP-запитів сервером. Мови програмування, що використовуються, включають HTML, CSS, та JavaScript.

1.1.2 Переваги та недоліки створення веб-застосунків

Веб-додатки розробляються за допомогою таких мов програмування, як HTML і CSS, що користуються популярністю серед IT-фахівців.

На відміну від нативних програм, веб-додатки можна використовувати на будь-яких пристроях, оскільки вони програмуються для роботи на будь-якій операційній системі, такій як iOS, Android, Windows Phone та інші [5]. Вони не вимагають значних ресурсів і не мають жодних обмежень апаратної платформи.

Проблем з підтримкою та сумісністю для старих версій програм у веб-додатках немає. При випуску нової версії програми всі користувачі автоматично працюють з нею, не стикаючись з проблемами оновлення на своїх пристроях.

Важливо відзначити, що не потрібно встановлювати веб-програму на пристрої. Вони запускаються безпосередньо в веб-браузері за простою URL-адресою, що забезпечує зручний доступ без потреби завантажувати і встановлювати їх з магазинів додатків, таких як Google Play або Apple Store.

Також важливо відзначити, що ви можете відкривати веб-сайти, не потребуючи оновлення, як у випадку зі звичайними програмами. Завдяки високому рівню розвитку мережевих з'єднань та надійності web-технологій, веб-додатки є економічно вигідним засобом розробки, оскільки для їх створення достатньо створити посилання між програмою та URL-адресою.

Однією з ключових переваг веб-додатків є їхня здатність надавати

користувачам повноцінну мобільність. Результати роботи можна зберігати на сервері, забезпечуючи доступ до них з будь-якої точки планети, де є Інтернет-з'єднання [5]. Як вже зазначалося, веб-додаток може бути використаний на різних пристроях. Проте для веб-сайту важливо мати програмування, яке дозволяє йому відображатися незалежно від операційної системи пристрою. Без адаптивності веб-сайту можуть виникнути проблеми з його коректним відображенням на iOS, Android або Windows Phone [5].

Необхідно також забезпечити доступ до Інтернету для використання веб-сайту або отримання користі від веб-додатка. У нашій країні, наприклад, доступ до Інтернету не є універсальним, що може ускладнювати використання веб-додатків, особливо під час подорожі.

Кожне перезавантаження сторінки призводить до значних затримок, оскільки потрібно встановлювати з’єднання HTTP, обробляти запит на сервері, відправляти HTTP-повідомлення до мережі та оновлювати сторінку в браузері. Це створює переривання в роботі та сповільнює процес [6].

Також існують обмеження доступу до певних апаратних функцій пристрою через веб-додатки.

1.2. Інструменти для розробки веб-застосунку

1.2.1 HTML - Hypertext Markup Language

HTML - це мова розмітки, що описує, як програміст використовує код для позначення тексту. [7] Мови розмітки, такі як HTML і XML, відрізняються від машинних мов, таких як шістнадцятковий або двійковий код.

HTML, або Мова розмітки гіпертексту, представляє собою код, що використовується для створення веб-сторінок та веб-додатків. Початки HTML сягають кінця 1980-х та початку 1990-х років [7], коли ця мова дозволила веб-розробникам вказувати веб-браузерам, як відображати елементи, такі як текст,

зображення, форми та інтерактивні функції.

Зазвичай веб-розробники використовують HTML в поєднанні з CSS і JavaScript, щоб створювати веб-сторінки та програми, доступ до яких можна отримати через веб-браузери, такі як Chrome, Firefox, Safari та Edge. Стандарти HTML і CSS історично підтримує World Wide Web Consortium (W3C) [6].

Попри те, що HTML базується на правилах SGML (стандартна загальна мова розмітки), а XHTML використовує правила XML, який є більш строгим підрозділом SGML [7]. Документи XHTML повинні бути аналогічними до XML-документів, що робить їх запуск можливий за допомогою стандартних інструментів обробки документів XML.

HTML перекладається браузером та виглядає як зручний для людини документ. Він є програмою SGML та відповідає міжнародним стандартам ISO 8879.

Рис. 1.2. Зображення логотипу HTML

На даний час, HTML 5 - найбільш використовуваний стандарт для розмітки веб-сторінок, дозволяючи вбудовувати мультимедійні елементи, такі як аудіо та відео. HTML 5 - основна особливість веб-сторінок, а інші стилі та функціональні можливості додаються за допомогою мов програмування, таких

як CSS, PHP та JavaScript [7]. Отже, освоєння HTML - важливий крок перед вивченням інших мов.

HTML - основний будівельний блок веб-розробки, і його використовують у сучасних сервісах веб-дизайну, таких як WordPress, Wiki, Webli [6]. Елемент HTML відрізняється «тегами», які визначають його за допомогою символів «<» і «>» [7]. Розробник може використовувати HTML для визначення аспектів сторінки, а для її створення також використовують CSS, JavaScript, PHP, jQuery та XML.

Навчання HTML є ключовим етапом для початківців у програмуванні, що відкриває можливості вивчення більш складних аспектів, таких як JavaScript та CSS [7]. Оволодіння основами HTML корисно для майбутніх програмістів та тих, хто цікавиться створенням веб-сайтів для зручного перегляду на різних пристроях.

1.2.2. CCS - Cascading style sheets

CSS – це мова каскадних таблиць стилів (англ. cascading style sheets), яка використовується для оформлення документів, написаних мовою розмітки, і дозволяє визначати стилі для їх подання [7]. Концепція CSS була винайдена Håkon Wium Lie у 1994 році, і в грудні 1996 року WSC визначила специфікацію для цієї мови. CSS дозволяє веб-розробникам змінювати макет та зовнішній вигляд веб-сторінок, використовуючи один або кілька файлів стилів.

CSS може контролювати шрифт, розмір та кольори тексту на веб-сторінці [8]. Один файл CSS може бути пов'язаний з багатьма сторінками, що полегшує внесення змін до зовнішнього вигляду всіх цих сторінок одночасно. Приведений нижче приклад демонструє використання CSS для визначення стилів для шрифтів та зміни кольору тексту.

Рис 1.3. Зображення логотипу CSS

До основних переваг можна віднести:

– Швидше завантаження сторінок: Використання CSS дозволяє уникнути повторення написання атрибутів HTML-тегів для кожної сторінки. Просто визначте одне правило CSS для тегу і застосовуйте його у всіх випадках цього тегу [7]. Це призводить до меншого обсягу коду і швидшого завантаження сторінок.

– Простота обслуговування: Для внесення глобальних змін вам просто потрібно змінити стиль, і всі елементи на всіх веб-сторінках автоматично оновляться. Це спрощує процес обслуговування [7].

– Покращені стилі для HTML: CSS надає значно більший набір атрибутів, порівняно з HTML. Таким чином, ви можете створити більш привабливий зовнішній вигляд вашої сторінки HTML, використовуючи різні стилі та ефекти [8].

– Сумісність декількох пристроїв: З використанням таблиць стилів можна оптимізувати вміст для різних типів пристроїв, використовуючи той самий HTML-документ [7]. Різні версії веб-сайту можуть бути адаптовані для портативних пристроїв, таких як КПК та мобільні телефони, або для друку.

1.2.3 JavaScript

JavaScript – це найчастіше використовувана мова сценаріїв для створення скриптів у веб-браузерах, які вбудовуються в веб-сторінки [7]. JavaScript є зареєстрованою торговою маркою компанії Sun Microsystems, Inc [8]. Його створено з метою «оновлення веб-сторінок». Програми, написані на цій мові, відомі як скрипти, і можуть бути вбудовані прямо в HTML-сторінки, автоматично запускаючись при завантаженні сторінки.

Рис 1.4. Зображення логотипу JS

Скрипти подаються та виконуються як звичайний текст, і для їх запуску не потрібно жодної спеціальної підготовки чи компіляції. В цьому відношенні JavaScript суттєво відрізняється від іншої мови, що має назву Java. Спочатку JavaScript називався «Живий скрипт», але з часом він став повністю самостійною мовою з власним ECMAScript, і тепер він не має нічого спільного з Java [9].

Згідно з результатами проведених досліджень, в наш час мова програмування JavaScript є найбільш високо популярною та широко використовуваною у всьому світі [11]. Розгортання цієї мови в сфері веб-розробки та інших індустріальних галузях значно перевищує інші мови програмування, підтверджуючи її визнання та важливість в глобальному

програмістському співтоваристві.

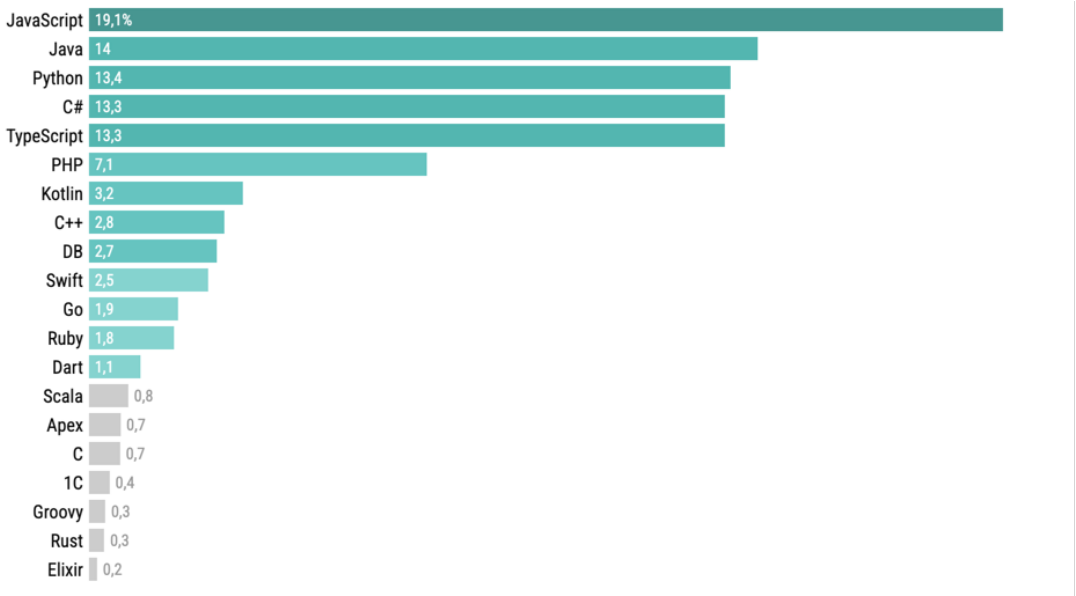

Рис. 1.5. Статистика використання мов програмування на 2023

JavaScript використовується в основному як мова сценаріїв клієнта, що означає, що код JavaScript написаний на сторінці HTML. Коли користувач запитує HTML-сторінку з JavaScript, сценарій відправляється браузеру, і браузер вирішує, як реагувати на нього [12].

Оскільки сценарії розміщуються на сторінці HTML, будь-хто, хто переглядає вашу сторінку, може переглядати та копіювати їх. Проте ця прозорість може розглядатися як перевага, оскільки інші можуть вивчати та використовувати JavaScript-скрипти, що знаходяться на веб-сторінках.

JavaScript також може використовуватися за межами веб-браузера. Серверний JavaScript від Netscape, створений як мова CGI, може виконувати ті ж завдання, що і Perl або ASP [10]. Немає обґрунтованих причин не використовувати JavaScript для написання реальних та складних програм, хоча цей сайт обмежений використанням JavaScript у веб-браузерах [8].

На сьогодні JavaScript працює не лише в браузерах, але й на серверах та

інших пристроях за допомогою спеціальної програми, відомої як JavaScript Mechanical [12]. У браузері вбудований «движок», який іноді називають віртуальною машиною JavaScript.

Сучасний JavaScript є «безпечною» мовою програмування. Низькорівневий доступ до пам’яті або ЦП не надається, оскільки він розроблений для браузерів, яким це не потрібно [10].

Можливості JavaScript значною мірою залежать від середовища, в якому він використовується. Наприклад, Node.js JavaScript підтримує функції, які дозволяють читати/записувати випадкові файли, надсилати мережеві запити тощо. У браузері JavaScript може робити все, що стосується шахрайства на вебсайті, взаємодії з користувачем та веб-сервера.

1.2.4. Застосування Framework у розробці веб-застосунків

Фреймворки представляють собою програмні засоби, які спрощують процес розробки та управління технічно високо складними проєктами. Зазвичай, фреймворк включає лише основні програмні модулі, залишаючи розробнику завдання реалізації всіх компонентів, що базуються на конкретному проєкті [13]. Це сприяє не лише швидкому розвитку, але й забезпечує високу продуктивність та надійність у вирішенні завдань.

Web Framework, зокрема, є платформою, призначеною для створення веб-сторінок та веб-додатків, спрощуючи розробку та інтеграцію різноманітних компонентів у великих програмних проєктах [13]. Завдяки широкому спектру можливостей та високій ефективності у бізнес-логіці, ця платформа ідеально підходить для створення складних веб-сайтів, бізнес-додатків і веб-сервісів.

З комерційного погляду розробка на фреймворку виявляється більш ефективною і якісною, ніж написання програм на чистій мові програмування без використання будь-яких платформ. Розробка без використання платформи може

бути ефективним варіантом лише у двох ситуаціях: коли проєкт надто простий і не вимагає подальшого розвитку, або коли він дуже складний і потребує оптимізації на дуже низькому рівні. У всіх інших випадках розробка на програмній платформі відбувається швидше і з вищою якістю.

Однією з основних переваг використання фреймворків є те, що вони визначають інтегровану структуру для додатків, побудованих на їх основі. Тому додатки на фреймворках легко виправляти та модифікувати, оскільки стандартна структура класу зрозуміла всім розробникам, що працюють з цією платформою. Не потрібно витрачати багато часу на пошук місця для реалізації нового функціоналу. Більшість фреймворків для веб-додатків використовують методологію MVC (model-view-controller), що сприяє однаковому підходу до організації елементів програмування [13]. Це полегшує розуміння архітектури програми, навіть для розробників, які необізнані з конкретним фреймворком.

1.3. Огляд React framework

React — це бібліотека розробки інтерфейсу користувача, створена компанією Facebook та підтримується відкритою спільнотою розробників [14]. Вона базується на мові програмування JavaScript та широко використовується у веб-розробці для створення динамічних та ефективних інтерфейсів.

Бібліотека React використовує архітектуру Flux. Внаслідок цього вона змогла наблизитися до розв'язання проблеми неконтрольованих мутацій. На відміну від своїх попередників React надав розробникам можливість управління станом компонента єдиним чином, зберігаючи стан у сховищі. Коли стан змінюється, представлення компонента повністю оновлюється.

Створювати елементи HTML можна за допомогою JavaScript. Але це досить клопітно. React надає можливість використовувати формат JSX. JSX є розширенням синтаксису звичайного JavaScript і використовується для

створення елементів React [14]. Синтаксис JSX виглядає як синтаксис HTML. Перед відображенням у браузері JSX компілюється в JavaScript. JSX надає більше можливостей ніж простий HTML і більш зручний ніж JavaScript.

Формат JSX має наступні переваги:

- Виконує оптимізацію під час перекладу на звичайний JavaScript.
- Полегшує створення шаблонів.
- Представлення та логіка компонента не розділені.
- Реалізує принцип поділу відповідальностей.
- Відділяє управління змінами DOM.

Рис. 1.6. Логотип React

1.3.1. Історія та основні методи використання React

React вперше з'явився в травні 2013 року і став однією з провідних технологій для створення користувацьких інтерфейсів в Інтернеті [15]. Ця бібліотека відома своєю ефективністю, зручністю використання та гнучкістю у розробці.

Незважаючи на те, що React є бібліотекою, а не мовою програмування, він пропонує розширення для підтримки різних архітектур програм, таких як

Flux та React. Flux — це архітектурний патерн, розроблений Facebook для ефективного управління станом додатка в React [15]. React дозволяє розробникам створювати мобільні додатки з використанням React та JavaScript. У 2013 році, під час розробки та інтеграції чату в платформу Facebook, розробники стикалися з численними викликами. Цей чат був складним додатком, який вбудовувався в інші частини проєкту. Його інтеграція вимагала вирішення нетривіальних завдань, таких як управління неконтрольованою зміною DOM (Document Object Model) та забезпечення паралельної асинхронної роботи користувачів в новому середовищі [16].

У цьому процесі команді Facebook довелося стикнутися з викликами, пов'язаними зі зміною структури DOM, яка виникла при взаємодії з складними компонентами. Реалізація асинхронної роботи користувачів у новому середовищі вимагала від команди розв'язання проблем паралельної обробки даних [15]. Інструменти, які використовувалися в той час не володіли функціоналом, який міг забезпечити отримання бажаного результату. Для розв'язання цієї проблеми команда розробки React застосувала наступні підходи:

— Використанням архітектури Flux та односпрямована прив'язка даних.

— Стан компонента є незмінним. Стан не може бути зміненим після першого його визначення. Зміна стану призводить до повного оновлення компонента.

У React кожен процес створення компонента включає різні методи життєвого циклу. Ці методи життєвого циклу не дуже складні та викликаються в різні моменти протягом життя компонента. У процесі створення компонента в React важливо розуміти, як працює його життєвий цикл, оскільки це дозволяє ефективно взаємодіяти з компонентами на різних етапах їхнього життя. Проаналізувавши документацію, я дійшов висновку, що методи життєвого циклу, такі як componentDidMount, componentDidUpdate і

componentWillUnmount, вони дозволяють виконувати певні дії при монтажі, оновленні та розмонтажі компонента відповідно [17].

Життєвий цикл компонента розділений на чотири фази:

1. Фаза монтажу
2. Етап оновлення
3. Фаза демонтування
4. Фаза обробки помилок

Кожна фаза містить деякі методи життєвого циклу, специфічні для конкретної фази.

До фази монтаж (Mounting) можна віднести такі :

— “constructor()”: Цей метод викликається при створенні екземпляра компонента. Використовується для ініціалізації стану та прив'язки методів.static

— “getDerivedStateFromProps()”: Викликається перед рендерінгом і при оновленні. Використовується для оновлення стану на основі змін властивостей.

— “render()”: Відповідає за відображення компонента.

— “componentDidMount()”: Викликається після того, як компонент вперше рендериться. Використовується для виконання дій після монтажу, таких як запити до сервера.

До фази оновлення (Updating) можна віднести такі :

— “static getDerivedStateFromProps()”: Викликається перед рендерінгом при оновленні компонента.

— “shouldComponentUpdate()”: Визначає, чи повинен компонент оновлюватися після змін стану або властивостей.

— “render()”: Відповідає за відображення оновленого компонента.

— “componentDidUpdate()”: Викликається після завершення оновлення компонента. Використовується для виконання дій після оновлення.

До фази розмонтаж (Unmounting) можна віднести такі :

— “componentWillUnmount()”: Викликається перед тим, як компонент

буде вилучений з DOM. Використовується для виконання очистки ресурсів.

До фази обробки помилок (Error Handling) можна віднести такі :

— “static getDerivedStateFromError()”: Викликається при виникненні помилки в дочірньому компоненті. Використовується для оновлення стану на основі помилки.

— “componentDidCatch()”: Викликається після обробки помилки. Використовується для логування помилок або звернення до служб попередження про помилки.

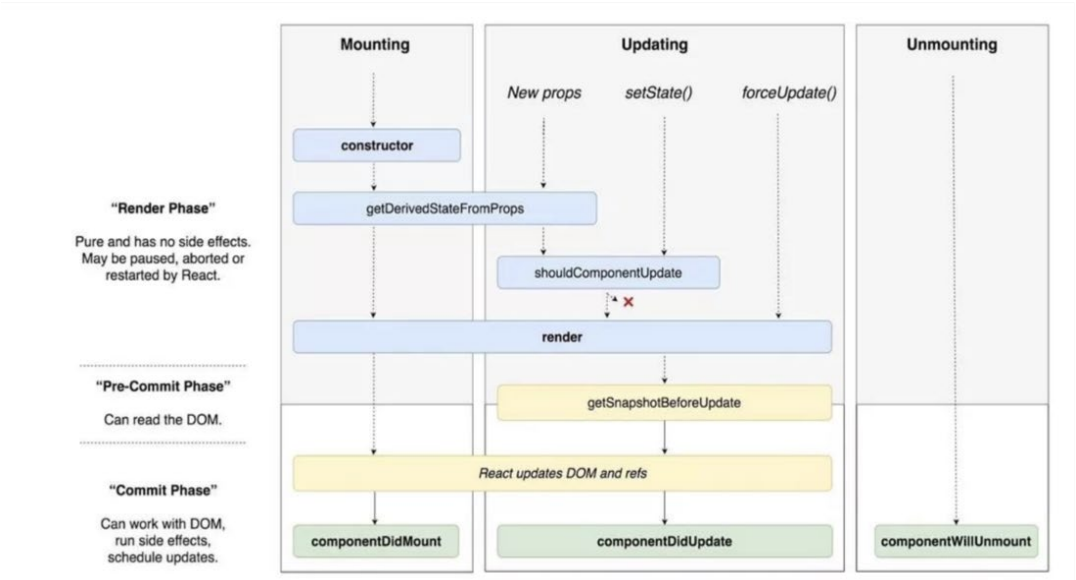

Рис. 1.7. Життєвий цикл React компонента

Більшість React-компонентів має сенс представляти соб у вигляді чистих функцій, які отримують вхідні параметри і повертають JSX [15]. Чисті функції мають зазвичай отримуючи одні і ті ж вхідні дані, вони завжди повертають одні й ті ж вихідні дані (вони є детермінованими). Також у них немає побічних ефектів (тобто - вони не працюють з мережевими ресурсами, не виводять щось в консоль, не записують нічого в localStorage і так далі).

Віртуальний DOM у React є «віртуальним» представленням фактичного

DOM [15]. Це не що інше, як об’єкт, створений для копіювання фактичного DOM.

На відміну від фактичного DOM, віртуальний DOM дешево створити, оскільки він не записує на екран. Він доступний лише як стратегія для запобігання перемальовування непотрібних елементів сторінки під час повторної візуалізації [15]. Коли ми відтворюємо інтерфейс користувача, створюється віртуальний DOM для цього відтворення та зберігається в пам’яті.

Якщо у рендері відбувається оновлення, React автоматично створює нове віртуальне дерево DOM для оновлення. Коли React реалізує алгоритм розрізнення, він починає з порівняння того, чи мають обидва знімки той самий кореневий елемент. Якщо вони мають той самий елемент, React переходить і рекурсує на атрибутах, а потім на дочірніх елементах вузла DOM. Якщо кореневі елементи мають різні типи - що рідко зустрічається в більшості оновлень - React знищить старі вузли DOM і створить нове дерево DOM. На кожному рендері React має віртуальне дерево DOM, яке порівнює з попередньою версією, щоб визначити, який вміст вузла оновлюється, і переконатися, що оновлений вузол збігається з фактичним DOM.

Кожного разу, коли ми маніпулюємо віртуальними елементами DOM у React, ми обходимо серію операцій, задіяних під час безпосереднього маніпулювання фактичним DOM [17]. Це можливо, тому що з віртуальним DOM нічого не малюється на екрані. Крім того, за допомогою алгоритму розрізнення React може остаточно визначити, яке оновлення потрібне, оновити лише об’єкт у реальному DOM. Концепція віртуального DOM у React, безсумнівно, допомагає зменшити витрати на повторне відтворення веб-сторінки, тим самим мінімізуючи час, необхідний для перемальовування екрана.

Hooks — це просто функції JavaScript, які дають розробника можливість «підключитися» до стану реакції та життєвого циклу функціональних компонентів [15]. Hooks використовуються, щоб ізолювати багаторазові частини

від функціонального компонента. Hooks дозволяють використовувати стан та інші функції React без написання класу. Перша підтримка hooks з’явилася в React 16.8.0, випущеному в лютому 2019 року [18].

Для розробників React Hooks надають такі переваги:

- Революціонізує спосіб написання компонентів.
- Код стає стислим і зрозумілішим.
- З hooks простіше працювати та тестувати.
- Пов'язана логіка може бути згрупована всередині hooks.
- Більш зручне компонування та повторне використання коду.
- Працюватимуть ефективніше з оптимізаціями React.

1.3.2. Аналіз основних методів оптимізації в React

Базуючись на проведених дослідженнях, варто відзначити, що у React існує кілька стратегій оптимізації, спрямованих на покращення продуктивності та користувацького досвіду. Ознайомившись з документацією можу сказати, що мемоізація є однією з таких стратегій, дозволяючи ефективно зберігати та використовувати результати функцій при однакових вхідних даних. Загальна думка з численних джерел вказує на високу ефективність цього методу оптимізації, тому вважається доцільним подальше вивчення його особливостей.

Ліниве завантаження або розділення коду представляє собою техніку, що сприяє завантаженню коду лише в тому випадку, коли він є необхідним [18]. Згідно багатьох аналітичним даним використання React.lazy та Suspense може значно покращити швидкість завантаження веб-застосунку на практиці.

Судячи з власного досвіду, можу з упевненістю сказати, що оптимізація рендерингу, має значний вплив на ефективність веб-додатків. Особливо у комбінації з такими методами оптимізації як, shouldComponentUpdate або використання класу React.PureComponent. Останній автоматично враховує

поверхневе порівняння стану компонента, що забезпечує виклик shouldComponentUpdate лише при реальних змінах [16].

Оптимізація викликів API, хоча і важлива, проте може бути дещо витратною та не завжди призводити до імпозантних результатів.

Загалом, аналізуючи різні джерела, можна визначити, що ці стратегії можна успішно комбінувати для досягнення оптимальної продуктивності в React-додатках. Проте важливо дотримуватися балансу та уникати надмірної складності коду під час оптимізаційного процесу.

1.3.2.1. Зміна React компонентів як метод оптимізації

Компоненти - це незалежні фрагменти коду, які можна багаторазово використовувати [15]. Вони служать тій самій меті, що й функції JavaScript, але працюють ізольовано та повертають HTML через функцію render().

В основу React було закладено застосування класових компонентів, у яких є швидкий доступ до життєвого циклу компоненту, але це означає, що сам клас наслідує увесь реалізований функціонал за замовчуванням, що належить до базового React.Component [16].

Функціональні компоненти в першу чергу були створені для реалізації простих «чистих» компонентів, для відображення інтерфейсу [18]. Вони є звичайними JavaScript функціями з можливістю застосування JSX-формату, а тому не мають доступу до контролю внутрішнім станом і життєвим циклом.

З часом розробники React бібліотеки розширили її функціонал з додаванням можливості використання hooks для більш гнучкого застосування функціональних компонентів у сучасних проєктах, а саме: контроль внутрішнього стану та життєвого циклу.

Згідно офіційній документації React рекомендує сучасні проєкти писати на функціональних компонентах, але при цьому не каже, що класові компоненти

вважаються застарілими.

Проаналізувавши це питання, можу сказати, що класовий компонент повинен наслідувати React.Component, що надає йому доступ до функціонала компонента та управління внутрішнім станом. У методі render повинна лежати верстка яку буде малювати браузер.В той момент як, функціональні компоненти повинен використовувати звичайні функції, які за допомогою JSX формату, дозволяють повертати верстку.

Отже, функціональні компоненти є кращим рішенням для створення простих та швидких у роботі компонентів, які повинні відображати певну верстку на основі даних. Проте, при роботі з ними, слід завжди пам'ятати, що вони у певному сенсі більше схожі на чисті функції то у них за замовчуванням відсутній доступ до управління життєвим циклом компонента.

1.3.2.2. Аналіз використання підходів управління ресурсами як метод оптимізації

У програмуванні мемоізація - це техніка оптимізації, яка робить програми більш ефективними, а отже, швидшими. Це робиться шляхом зберігання результатів обчислень у кеші та отримання тієї самої інформації з кешу наступного разу, коли вона знадобиться, замість повторного обчислення.

Простіше кажучи, це полягає у зберіганні в кеші результату функції та перевірці функції, чи є кожне необхідне обчислення в кеші перед його обчисленням.

Кеш - це просто тимчасове сховище даних, яке зберігає дані, щоб майбутні запити на ці дані могли обслуговуватися швидше [19].

Мемоізація - це простий, але потужний трюк, який може допомогти пришвидшити наш код, особливо коли ми маємо справу з повторюваними та важкими обчислювальними функціями [19].

Мемоізація в JavaScript базується на двох концепціях:

— Замикання (closure): поєднання функції та лексичного середовища, у якому ця функція була оголошена.

— Функції вищого порядку (HOF): функції, які працюють з іншими функціями, приймаючи їх як аргументи або повертаючи їх.

У React ми можемо оптимізувати додаток, уникаючи непотрібного повторного відтворення компонентів за допомогою мемоізації.

Для чистих компонентів класу, щоб реалізувати мемоізацію, React надає базовий клас, який має назву PureComponent.

Компоненти класу, які розширюють React.PureComponent клас, мають деякі покращення продуктивності та оптимізацію візуалізації. Це пов’язано з тим, що React реалізує shouldComponentUpdate() метод для них із поверхневим порівнянням властивостей і стану [18].

Компонент вищого порядку або HOC подібний до функції вищого порядку в JavaScript. Функції вищого порядку - це функції, які приймають інші функції як аргументи або повертають інші функції. React HOC беруть компонент як властивість і маніпулюють ним з певною метою, фактично не змінюючи сам компонент. Можна розглядати це як компоненти-обгортки. У цьому випадку memo виконує подібну роботу до PureComponent, уникаючи непотрібного повторного рендерингу компонентів, які він огортає.

React.memo - це компонент вищого порядку та інструмент оптимізації продуктивності для функціональних та класових компонентів [16]. Якщо наш функціональний компонент рендерить той самий результат із тими самими властивостями, React запам’ятає, пропустить рендеринг компонента та повторно використає останній відрендерений результат.

React.memo лише перевіряє наявність змін пропів. Якщо функціональний компонент, загорнутий у React.memo, має useState, useReducer або useContex Hook у своїй реалізації, він все одно буде повторно рендерится, коли стан або

контекст зміняться [18].

За замовчуванням він порівнюватиме лише складні об’єкти в об’єкті props. Якщо ми хочемо контролювати порівняння, ми також можемо надати спеціальну функцію порівняння як другий аргумент.

Цей метод існує лише як оптимізація продуктивності. Не покладайтеся на це, щоб «запобігти» візуалізації, оскільки це може призвести до помилок.

Є причина, чому React.memo за замовчуванням використовує неглибоке порівняння для визначення часу повторного рендерингу: це тому, що є додаткові накладні витрати на перевірку, чи було значення запам'ятоване кожного разу, коли нам потрібно отримати до нього доступ, і складніша структура даних у значення, тим гірші накладні витрати.

На відміну від методу shouldComponentUpdate для компонентів класу, функція areEqual повертає true, якщо атрибути рівні, і false, якщо атрибути неоднакові. Це зворотне від shouldComponentUpdate. useMemo - це вбудований React hook, який приймає 2 аргументи функцію, яка обчислює результат і масив залежностей [16]. Приклад наведено нижче у коді.

```
const memoizValue = useMemo (() => computerExpensiveValue (a, b), [a, b];
```

Під час початкового рендерингу useMemo викликає обчислення, запам’ятовує результат обчислення та повертає його компоненту. useMemo буде переобчислювати мемоізоване значення лише тоді, коли одна із залежностей зміниться. Ця оптимізація допомагає уникнути дорогих обчислень на кожному рендері. Якщо під час наступних візуалізацій залежності не зміняться, то useMemo не викликає обчислення знов, а повертає мемоізоване значення. Передана функція useMemo виконується під час рендерингу, тому useMemo не призначене для використання побічних ефектів, як це робить useEffect hook.

Залежності діють подібно до аргументів у функції. Список залежностей -

це useMemo спостереження за елементами: якщо немає змін, результат функції залишиться незмінним [20]. В іншому випадку функція буде запущена повторно. Якщо вони не зміняться, неважливо, якщо весь наш компонент повторно візуалізується, функція не буде повторно запущена, а поверне збережений результат. Це може бути оптимальним, якщо загорнута функція є великою та дорогою. Це основне використання для useMemo. Якщо функція використовує значення властивостей або станів, обов'язково потрібно вказати ці значення як залежності. Якщо масив не надано, нове значення обчислюватиметься під час кожного візуалізації. Ми можемо покладатися на useMemo оптимізацію продуктивності, а не на семантичну гарантію.

Є дві проблеми, які useMemo намагається вирішити: референційну рівність і обчислювально-дорогі операції.

У майбутньому React може вибрати «забути» деякі попередньо запам'ятовані значення та перерахувати їх під час наступного рендерингу, наприклад, щоб звільнити пам'ять для компонентів поза екраном.

```
const memoizeCallback = useCallback(() => {doAction(a, b) ;}, [a, b, ) ;
```

Hook useCallback React може бути використаний для оптимізації поведінки візуалізації наших функціональних компонентів React [20].

1.3.2.3. Аналіз методу оптимізації відкладеного завантаження та поділ

Для роботи клієнтських додатків продуктивність, являється одним із найважливіших параметрів. Попри те, що JavaScript створений як проста мова, він може створювати напрочуд складні кодові бази, що ускладнює масштабування [21]. Частково це пояснюється тим, що існує велика

різноманітність доступних класів і модулів. Більшість суттєвих програм і фреймворків JavaScript мають багато залежностей, які можуть змусити простий, на перший погляд, проєкт швидко вбудувати велику кількість коду.

Чим більше коду містить проєкт, тим повільніше завантажуватиметься браузер. Тому вам часто доводиться збалансувати розмір ваших залежностей із продуктивністю, яку ви очікуєте від свого JavaScript. Поділ коду є корисним способом досягти цього балансу.

Будь-які фреймворки JavaScript об’єднують усі залежності в один великий файл. Це полегшує додавання JavaScript до веб-сторінки HTML. Пакет вимагає лише одного тегу посилання з меншою кількістю викликів, необхідних для налаштування сторінки, оскільки весь JavaScript зберігається в одному місці [20]. Теоретично об’єднання JavaScript у такий спосіб має пришвидшити завантаження сторінки та зменшити обсяг трафіку, який має обробляти сторінка [21].

Однак у певний момент пакет зростає до певного розміру, коли накладні витрати на інтерпретацію та виконання коду сповільнюють завантаження сторінки, а не прискорюють його. Ця критична точка різна для кожної сторінки, і ви повинні протестувати свої сторінки, щоб визначити, де це. Загальних вказівок немає — все залежить від залежностей, які завантажуються.

Ключ до поділу коду полягає в тому, щоб визначити, які частини сторінки мають використовувати різні залежності JavaScript [22]. Розбиття коду дозволяє стратегічно видалити певні залежності з пакетів, а потім вставити їх лише там, де вони потрібні. Замість надсилання всього JavaScript, який складає програму, щойно завантажується перша сторінка, розділення JavaScript на кілька фрагментів значно покращує продуктивність сторінки [22].

Згідно багатьох джерел поділ коду є звичайною практикою у великих додатках React, і збільшення швидкості, яке воно забезпечує, може визначити, чи продовжить користувач використовувати веб-додаток, чи залишить його.

Багато досліджень показали, що сторінки мають менше трьох секунд, щоб справити враження на користувачів, тому економія навіть часток секунди може бути значною. Тому ідеальним є прагнення до трьох секунд або менше часу завантаження.

Існує гіпотеза що, найпростіший спосіб розділити код у React - це динамічний синтаксис «імпорту». Деякі комплектувальники можуть аналізувати оператори динамічного імпорту нативно, тоді як інші потребують певної конфігурації. Проте, важливо знати, що синтаксис динамічного імпорту працює як для створення статичного сайту, так і для відтворення на стороні сервера. Динамічний імпорт використовує then-функцію для імпорту лише необхідного коду [22]. Будь-який виклик імпортованого коду має бути всередині цієї функції.

Резервна властивість може прийняти будь-який елемент React, який буде відрендерено в очікуванні завантаження компонента. Компонент Suspense можна розмістити будь-де над компонентом lazy. Крім того, кілька відкладених компонентів можна обернути одним компонентом Suspense.

Також, ще один відомий оптимізації - це React.lazy надає можливість поділу коду на основі маршруту [25]. Якщо говорити про визначення поділ коду, то я б описав це як розбиття коду на компоненти або численні пакети, які можна завантажувати за потреби або паралельно. Зі зростанням програми зростає її складність, а також пакети CSS, особливо зі збільшенням кількості та розміру бібліотек. Його можна розділити на кілька менших файлів, щоб зменшити завантаження всіх файлів. Іменовані експорти: React.lazy наразі підтримує лише експорти за замовчуванням. Проміжний модуль, який повторно експортує за замовчуванням, потрібно створити, якщо потрібно імпортувати модуль, який використовує іменовані експорти. Це забезпечує роботу структури дерева та запобігає втягуванню невикористаних компонентів.

1.3.3. Проблеми у роботі React веб-застосунків: визначення та

вирішення.

Розробка веб-застосунків на React вимагає креативності, фаховості та уважності до деталей у сучасному програмуванні, де швидкість та ефективність набувають великої ваги [20]. У цьому розділі ми глибоко аналізуємо причини, що впливають на продуктивність клієнтської частини React веб-застосунків. Розглядаються аспекти коду, архітектури, рендерингу, управління станом, оптимізації завантаження ресурсів та мережевої взаємодії. Розв'язання цих питань визначає технічну адекватність продукту та його здатність відповідати високим стандартам продуктивності та користувацького досвіду. Цей розділ спростить уявлення про фактори, які впливають на швидкість React-додатків, та надасть рекомендації для їх вирішення, створюючи основу для оптимальної розробки та підвищення продуктивності.

Швидкість роботи React веб-застосунків залежить від різноманітних факторів. Нижче перераховано ключові аспекти та помилки, які можуть впливати на продуктивність React-додатків:

— Використання стану (state) в React: Використання стану є ключовим для збереження та керування даними у веб-додатку. Проте надмірне оновлення стану може викликати непотрібні рендеринги та знижувати продуктивність. Рекомендується уникати оновлення стану при кожній невеликій зміні, щоб уникнути зайвого коду та підвищити читабельність. Централізований підхід до оновлення стану також робить код більш структурованим.

— Управління станом та залежності: Зі збільшенням кількості станів у додатку слід ретельно планувати їх управління. Надмірна фрагментація станів та їх розподіл по багатьох компонентах може ускладнювати управління та розуміння коду. Важливо дотримуватися засад прозорості та спрощення структури додатка.

— Використання життєвого циклу компонентів: Неправильне

використання життєвого циклу компонентів може призвести до некоректної обробки даних та збереження стану. Важливо ретельно вивчати та користуватися методами життєвого циклу для досягнення потрібної функціональності та уникнення потенційних проблем.

— Використання ключів (keys): Неправильне використання ключів може призвести до втрати даних та неправильного оновлення компонентів. Уникайте повторюваних ключів та встановлюйте їх унікальними для кожного елемента. Це важливо для правильної взаємодії React з DOM-елементами.

— Створення великих та складних компонентів: Великі компоненти можуть ускладнювати розробку, тестування та відлагодження. Рекомендується дотримуватися принципів доброї архітектури, поділу на менші компоненти та використання React.

У світі розробки на React існує безліч можливостей для створення потужних та ефективних веб-додатків. Однак, незважаючи на всю потужність цього інструменту, розробники можуть натрапляти на труднощі при використанні життєвого циклу компонентів [23]. В цьому контексті важливо визначити кілька ключових причин, які спричиняють виникнення помилок у роботі з життєвим циклом React:

— Новачки в React: Новачкам буває складно освоїти життєвий цикл компонентів, що може призводити до неправильного виклику методів або неправильного їх розуміння.

— Брак досвіду в React: Розробники без достатнього досвіду можуть некоректно використовувати життєвий цикл, копіюючи код або неправильно тлумачачи концепції React.

— Зміни в версіях React: Оновлення можуть викликати непорозуміння, оскільки розробники можуть не встигати слідкувати за змінами та нововведеннями у життєвому циклі компонентів.

— Некоректне використання методів: Інколи розробники

використовують методи життєвого циклу для завдань, для яких вони не призначені, що може викликати проблеми з продуктивністю та залежністю від умов.

— Відсутність планування і архітектурної роботи: Недостатнє планування може призводити до неправильного використання життєвого циклу, ускладнюючи розуміння та підтримку кодової бази.

У підсумку, робота з життєвим циклом компонентів у React може бути викликана численними труднощами, особливо для новачків та розробників без достатнього досвіду. Неправильний виклик методів, некоректне використання або невірне розуміння може виникати зі змінами у версіях React або недостатнім плануванням і архітектурною роботою [20].

Щоб уникнути цих проблем, важливо активно вивчати документацію React, використовувати сучасні підходи та систематично слідкувати за оновленнями. Крім того, планування архітектури додатку наперед може запобігти неправильному використанню життєвого циклу та спростить підтримку кодової бази в майбутньому.

1.4. Вплив розміру веб-застосунку на метрики та користувацький досвід.

В сучасному цифровому світі, де швидкість, доступність та користувацький комфорт є ключовими вимогами, розмір веб-застосунку стає критичним фактором, який може визначити успіх або невдачу онлайн-проєкту. З великим розмаїттям пристроїв, варіацій мережевих з'єднань і вимог користувачів, важливо зрозуміти, як саме розмір веб-додатку впливає на фінального користувача [24].

Розмір веб-застосунку може містити в собі обсяг коду, ресурсів, зображень, а також інших елементів, які передаються на клієнтську сторону [3].

Чим більший розмір, тим більше часу потрібно для завантаження та ініціалізації додатку [18]. Це може привести до затримок у завантаженні, особливо на повільних мережах або на пристроях з обмеженими ресурсами.

Окрім технічних аспектів, розмір веб-застосунку має прямий вплив на враження користувача. Довгий час завантаження може призвести до негативного сприйняття користувачем, що вплине на його задоволення від використання додатку. Під час взаємодії з веб-додатком, користувач очікує миттєвої відгуку та плавності роботи, а завантаження, що триває занадто довго, може породжувати ризик втрати зацікавленості та втрати покладеної в додаток довіри [25].

В даному розділі я досліджу та продемонструю взаємозв'язок між розміром веб-застосунку та задоволенням користувача. Поглиблюсь в аналіз практичних прикладів та стратегій оптимізації розміру веб-застосунків для досягнення максимальної ефективності та задоволення кінцевого користувача.

Користувацький досвід (User Experience, UX) визначається як взаємодія та враження користувача під час використання певного продукту чи послуги. У сучасному цифровому середовищі, важливе значення приділяється UX у веб-додатках, оскільки він безпосередньо впливає на задоволення користувача, його лояльність та загальний успіх продукту [25].

Користувацький досвід не обмежується лише інтерфейсом веб-додатка. Він охоплює всі аспекти взаємодії користувача з продуктом, включаючи дизайн, навігацію, продуктивність, емоційний вплив та зручність використання. Ефективний UX має створювати позитивні емоції у користувачів та задовольняти їхні потреби. Фактори, що впливають на користувацький досвід у веб-додатках:

— Відповідність очікуванням: Важливо, щоб веб-додаток відповідав очікуванням користувача, надавав очікувану інформацію та функціонал.

— Ефективність та продуктивність: Швидкість завантаження сторінок, ефективна робота функціоналу та висока продуктивність є критичними для

створення позитивного досвіду користувача.

— Емоційний вплив: Враження та емоції, які викликає веб-додаток, суттєво впливають на його використання. Дизайн та інтерактивність грають ключову роль у формуванні емоційного досвіду користувача.

Сучасна глобальна мережа відіграє ключову роль у спілкуванні, розвитку бізнесу та доступі до інформації. За останні десятиліття світовий рівень швидкості Інтернету помітно зріс, але існують серйозні проблеми, які гальмують прогрес та ускладнюють загальний доступ до швидкісного та стабільного Інтернет-з'єднання [19].

Проблема нерівності в доступі до технологій стає викликом у багатьох регіонах світу. Країни з високим рівнем розвитку можуть насолоджуватися передовими технологіями та швидким Інтернетом, тоді як менш розвинені регіони стикаються з викликами, пов'язаними з браком інфраструктури та фінансових ресурсів [25]. Це створює цифровий розрив, ускладнюючи доступ до якісної освіти та інформації, але й розвиток економіки цих регіонів.

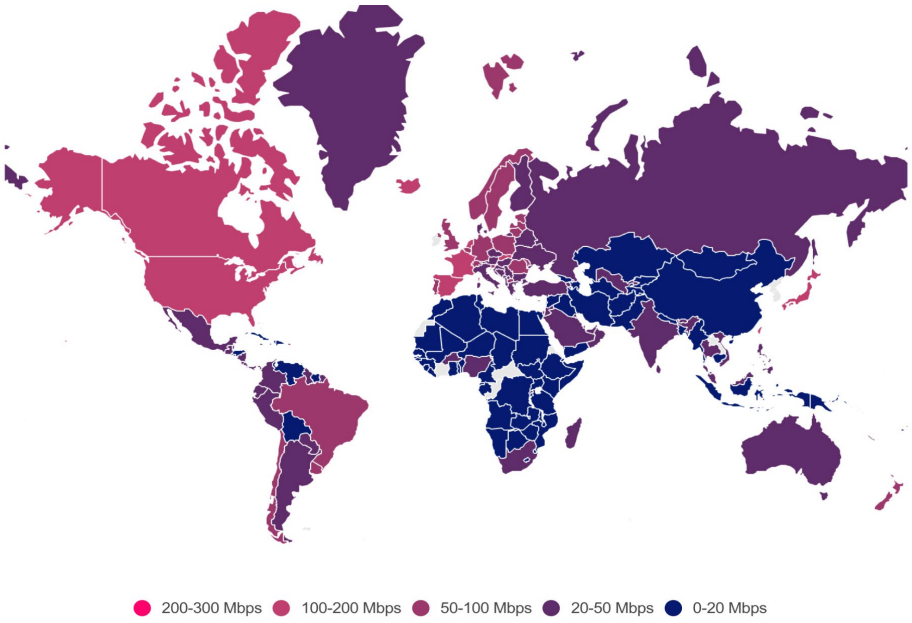

Рис. 1.8. Світовий рівень швидкості інтернету

На цьому зображенні можна відзначити, яка частина потенційної аудиторії може втрачатися через проблеми з розміром веб-додатка та відсутність ефективних оптимізаційних практик під час розробки. Отже, при створенні нового продукту за допомогою React важливо завжди пам'ятати про швидкість та якість його роботи.

Обмеження мережі щодо якості та швидкості мають вплив на IT-ринок та кількість потенційних користувачів цифрових продуктів, які можуть розв'язувати проблеми користувачів уже сьогодні [2]. Якщо не враховувати інші обмеження у розвитку веб-додатку, можна провести паралель між розміром вашого веб-додатку та якістю виконаної оптимізаційної роботи: чим менший розмір та більш якісна оптимізація, тим більша потенційна аудиторія у світі.

На додаток, ефективна оптимізація будь-якого веб-застосунку матиме прямий вплив на формування позитивного користувацького досвіду, завдяки плавній, швидкій та стабільній роботі веб-додатку [3]. Це, в свою чергу, повинно призвести до різкого росту кількісних та продуктових метрик, за умови, що сам продукт ефективно виконує свою основну функціональну задачу.

1.5. Висновки

В цьому розділі проведено вивчення та дослідження розробки та ключових складових веб-застосунків, розпочинаючи з визначення самого терміну "веб-застосунок" та закінчуючи вивченням впливу розміру додатку на метрики та користувацький досвід.

Досліджено архітектуру та принципи роботи веб-застосунків, виокремлено переваги та недоліки їх створення. Проведено детальний аналіз інструментів для розробки веб-застосунків, включаючи HTML, CSS, JavaScript та використання фреймворків.

Особлива увага приділена розділу, присвяченому огляду React framework. Розкрита історія його створення та застосування. Проведений аналіз основних

методів оптимізації в React, включаючи зміну компонентів, управління ресурсами та відкладене завантаження.

Даний розділ висвітлює проблеми, які можуть виникнути у роботі React веб-застосунків та запропоновані методи їх вирішення. Робляться висновки щодо впливу розміру веб-застосунку на метрики та користувацький досвід. Цей аналіз служить фундаментом для подальших досліджень і визначає напрямки оптимізації, які будуть розглянуті в наступних частинах дипломної роботи.

РОЗДІЛ 2
ЗАСТОСУВАННЯ МЕТОДІВ ОПТИМІЗАЦІЇ КЛІЄНТСЬКОГО REACT ВЕБ-ДОДАТКУ

2.1. Перехід з класових на функціональні компоненти

Компоненти дозволяють розділити інтерфейс користувача на незалежні частини, які можна використовувати повторно, і розглядати їх як окремі відокремлені одна від одної функціональні одиниці. Своєю сутністю компоненти чимось подібні до функцій JavaScript. Вони отримують різноманітні вхідні дані (відомі як "пропси") і повертають React-елементи.

Візьмемо за основу веб-застосунок react «Counter», у якому, в більшості випадків, використовуються класові компоненти. Роль застосунку проста: надати користувачеві можливість взаємодіяти з кнопками «-» та «+». Під час натискання кнопки «+» число на екрані застосунку буде збільшуватись, а при «-» воно буде зменшуватись.

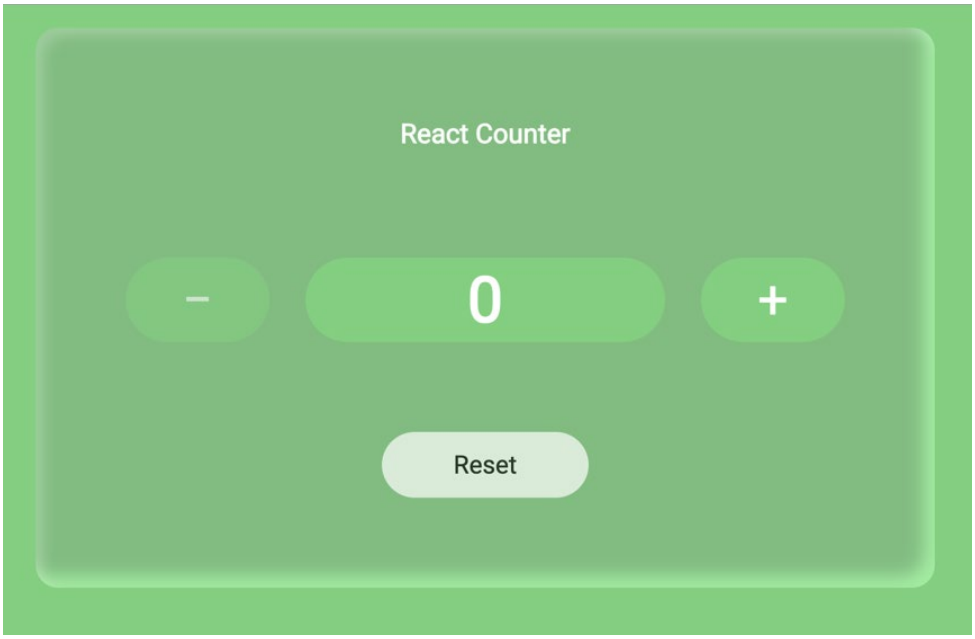

Рис. 2.1. Демонстрація react веб-застосунку «Counter»

Проте важливо розуміти, що ми використовуємо цей застосунок лише з метою оцінки того, наскільки ефективним є метод оптимізації, а саме перехід від класових компонент до функціональних.

Давайте більш детально розберемось із компонентом класу, а саме компонент із збереженням стану/контейнер. Цей компонент є звичайним класом ES6, який розширює клас компонентів бібліотеки React. Його називають компонентом із збереженням стану, оскільки він контролює, як змінюється стан і реалізацію логіки компонента [26]. Окрім цього, вони мають доступ до всіх різних фаз методу життєвого циклу React.

Для більш наявного виявлення різниці між класовим компонентом та функціональним, що використовує методи життєвого циклу, я підготував детальний приклад, який ви можете побачити нижче на рисунку 2.2. Наведений код демонструє, що функціональні компоненти вирізняються більшою конкретністю та компактністю свого коду. Такий аналіз дозволяє виокремити переваги функціональних компонентів не лише у використанні методів життєвого циклу, але й у зменшенні загального обсягу кодової бази вашого додатку, що сприяє покращенню його підтримки та розширення.

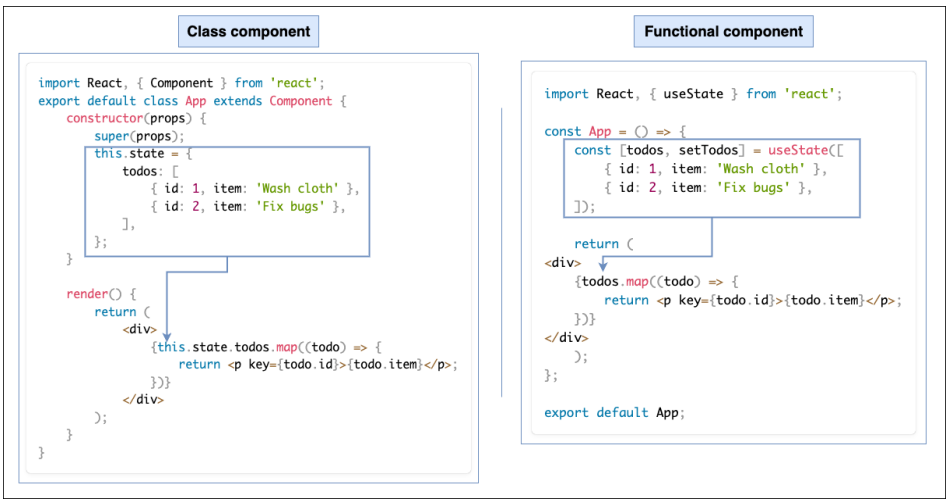

Рис. 2.2. Різниця між класовим та функціональним React компонентом

Далі, я буду описувати поступові кроки переходу від класових компонентів до функціональних для мого React веб-застосунку з детальним описом використанням методів функціональних компонентів. Для початку встановлюємо початковий стан за допомогою конструктора та використовуємо метод життєвого циклу componentDidMount(), щоб встановити стан від 0 до 1, коли компонент монтується, оскільки ми не хочемо, щоб відлік починався з 0. Під час першого відтворення компонента він швидко покаже лічильник 0 від початкового стану, тоді як після фактичного монтування компонента componentDidMount запуститься, щоб встановити новий стан лічильника 1.

Реалізуємо дві функції компонента (функції manualIncrement() і manualDecrement()) для збільшення та зменшення лічильника, коли користувач натискає кнопку збільшення або зменшення.

```
const CounterApp = () => {
  const [count, setCount] = useState(0);
  useEffect(0) = setCount((currentCount) => currentCount + 1), []);

  const manualIncrement = () => {
    setCounterAppValue((currentValue) => currentValue + 1);
    };

  const manualDecrement = () => {
    setCounterAppValue((currentValue) => currentValue - 1);
              };
  return (
    <div className="counterApp">
      <h1 className="count">{counterValue}</h1>
      <button type="button" onClick={manualIncrement}>
        Increment
      </button>
      <button type="button" onClick={manualDecrement}>
        Decrement
```

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

export default CounterAppComponent;
```

У прикладі нижче можна буде побачити, що для компонента на основі класу ми зробили кілька кроків, щоб створити цей динамічний компонент. Ми створили клас за допомогою конструктора та методу рендерингу. Ми встановлюємо початковий стан за допомогою оператора this.state у конструкторі. Ми використовуємо this.setState() для оновлення станів і методу життєвого циклу, щоб миттєво оновлювати стан під час монтування компонента.

```
class CounterApp extends React.Component {
 state = {
  CounterApp: 0,
 }
 constructor(props) {
  super(props)
  this.manualIncrement = this.manualIncrement.bind(this)
 }
 manualIncrement() {
  this.setState({ counterAppValue: this.state.counterAppValue + 1 })
 }
 render() {
  return (
   <div>
    <p> CounterApp Value: {this.state.counterAppValue}</p>
    <button onClick={this.manualIncrement}>Increment</button>
   </div>
  )
 }
}
```

Наступний, проте надзвичайно важливий крок - це перетворення компонент класу на функціональний компонент. Перенесемо проєкт на функціональні компоненти. На жаль це неможливо зробити без додаткових зусиль оскільки абсолютно всі компоненти необхідно буде вручну переписувати на функціональні компоненти.

```
function CounterApp() {
  const [counterAppValue, setCounterAppValue] = useState(0);

  const handleIncrement = () => setCounterAppValue(counterAppValue + 1);

  return (
    <div>
      <p>CounterApp Value: {counterAppValue}</p>
      <button onClick={manualIncrement}>Increment</button>
    </div>
  );
}
```

Вище на показаний, як буде виглядати компонент CounterApp після його перетворення з класового на функціональний.

До речі, раніше компонент класу був єдиним варіантом доступу до додаткових функцій React, таких як стан і методи життєвого циклу React. Однак за допомогою hooks ви можете реалізувати стан та інші функції React, і, що найважливіше, можна написати весь інтерфейс користувача з функціональними компонентами [27]. З hooks створення компонентів у React стає більш простим.

По-перше, не потрібен конструктор або методи візуалізації, оскільки це лише функція, а не клас. Hooks дозволяють інтегрувати всі необхідні функції

бібліотеки React, які раніше були доступні лише для компонента класу, тому за допомогою useState ви можете додавати стани до функціональних компонентів.

React має два найбільш часто використовувані hooks: стану (useState) і ефекту (useEffect) [14].

Як було наведено у прикладі вище було імпортували useState з React, щоб встановити початковий стан count на 0. Hook useState поверне пару значень: поточний стан і функцію, яка його оновлює. hook стану поверне пару значень: поточний підрахунок і функцію setCount, яка оновлює підрахунок стану.

Раніше не було такої можливості у функціональних компонентах, але тепер за допомогою hooks useEffect можна реалізувати методи життєвого циклу React [16]. Hook ефект дозволяє виконувати побічні ефекти у функціональних компонентах. Наприклад, тепер можна думати про useEffect як про componentDidMount, componentDidUpdate і componentWillUnmount разом. За допомогою цього hooks ефекту повідомляємо у React, що компонент повинен щось зробити після рендерингу [27]. Тоді React запам’ятає передану функцію та викличе її пізніше після виконання оновлень DOM.

Після цього ми встановимо змінну стану підрахунку, і подальше ми повідомляємо React, що функція, яка передається у hooks useEffect, виконує роль ефекту. Ця функція, містить внутрішній код, який оновлює кількість станів. Це дозволить нам взаємодіяти з React за допомогою ефектів та актуалізувати стани залежно від певних подій або умов, що сприяє більш гнучкій та контрольованій роботі компонентів у додатку.

За допомогою переходу від класових компонентів на функціональні ми досягаємо зменшення кількості дій, а також переносимо більшість логіки в окремі компоненти. Це призводить до значної оптимізації роботи веб-застосунку, сприяючи покращенню ефективності та забезпеченню більшої зрозумілості та легкості обслуговування коду.

2.2. Застосування мемоізації

2.2.1. React.memo

Для того, щоб мати можливість проаналізувати вплив React.memo на продуктивність додатка, створимо для експерименту клієнтський веб-застосунок «User list», який не використовує мемоізацію. Цей веб-додаток повинен імітувати велике навантаження, тому для дослідження я обрав список користувачів, який відображає більше як 10 000 компонентів.

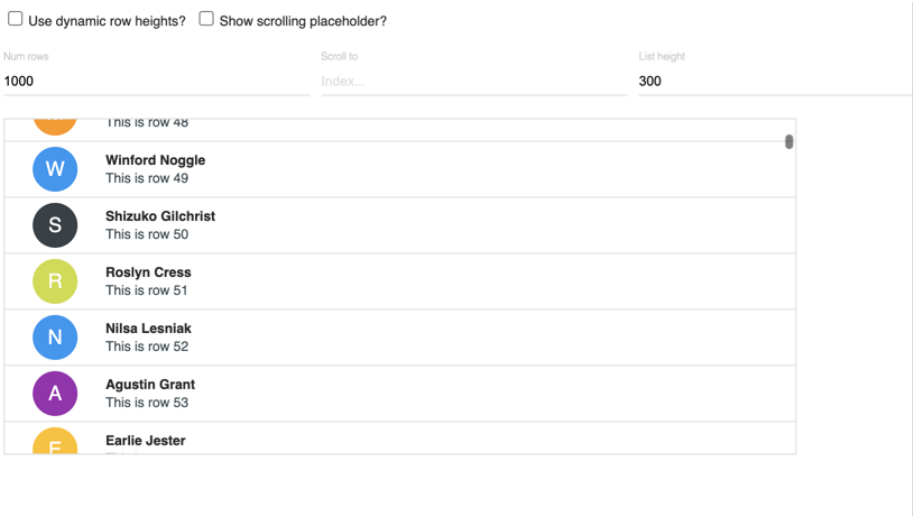

Рис. 2.3. Демонстрація react веб-застосунку «User list»

Якщо глянути більш детально на роботу веб-застосунку «User list», то можна побачити, що кожного разу, коли ми здійснюємо рух колесом миші, вносяться зміни до значення count, що призводить до повторного рендеру компоненту item. Це своєю чергою призводить до повторного рендеру List компоненту. Цей процес який блокує потік, і при наявності важких розрахунків у компоненті блокування може займати досить багато часу. Це створює враження наче програма не відповідає на дії користувача, а також додатково навантажує процесор.

Для розв'язання цієї проблеми можна використовувати наступні інструменти для мемоізації: React.memo, React.PureComponent, React.useMemo,

React.useCallback [27]. На наступному прикладі розглянемо як використовувати React.memo.

```
const MyList = React.memo(({ itemList }) => {
  console.log('renderMyList');
  return itemList.map((item, index) => (
    <div key={index}>item: {item.text}</div>
  ));
});

export default function MyApp() {
  console.log('renderMyApp');
  const [totalCount, setTotalCount] = useStage(0);
  const [itemList, setItemList] = useStage(getInitialItems(10000));

  return (
    <div>
      <h1>{totalCount}</h1>
      <button onClick={() => setTotalCount(totalCount + 1)}>
        Increment
      </button>
      <MyList itemList={itemList} />
    </div>
  );
}
```

У прикладі вище мемоізація працює наступним чином: внаслідок того, що під час перегляду елементів списку веб-сторінки дані самого компоненту, які пов’язані зі списком не змінилися, оскільки React.memo запобігає повторному рендерінгу списку. Таким чином зменшується час між діями користувача та відкликом програми, а також навантаження на процесор.

2.2.2. React.PureComponent

React.memo призначений для мемоізації функціональних компонентів. Для класових компонентів використовується React.PureComponent. React.PureComponent схожий на React.Component [27].

```
const ItemList = React.memo(({ items }) => {
  console.log('renderItemList');
  return items.map((item, index) => (
    <div key={index}>Item: {item.text}</div>
  ));
});

export default function MyApp() {
  console.log('renderMyApp');
  const [totalCount, setTotalCount] = useState(0);
  const [itemList, setItemList] = useState(getInitialItems(10000));

  return (
    <div>
      <h1>{totalCount}</h1>
      <button onClick={() => setTotalCount(totalCount + 1)}>
        Increment
      </button>
      <ItemList items={itemList} />
    </div>
  );
}
```

Різниця між ними полягає в тому, що React.Component не реалізує метод життєвого циклу shouldComponentUpdate(), а React.PureComponent реалізує його за допомогою неглибокого порівняння властивостей і станів [26].

Якщо функція React-компонента render() відтворює той самий результат із тими самими властивостями та станом, ми можемо використовувати React.PureComponent для підвищення продуктивності в деяких випадках [26]. Все що необхідно реалізувати для цього підходу, це вказати, що наш компонент наслідує React.PureComponent як наведено у прикладі.

```
class MyComponent extends React.PureComponent {
}
```

Основна різниця між React.PureComponent і React.Component полягає в тому, що порівняння змін стану чи властивостей PureComponent є поверхневим [21]. Це означає, що він порівнює їх значення, але при порівнянні об’єктів порівнює лише посилання. Це допомагає покращити продуктивність програми.

2.2.3. React.useMemo

Для того, щоб мати можливість проаналізувати вплив useMemo на продуктивність веб-додатку, продовжимо працювати з додатком «User list», який не використовує мемоізацію. Щоб імітувати велике навантаження, будемо відображати компонент, який виконує трансформацію вхідного значення шляхом його розрахунку за допомогою факторіалу.

Кожного разу, коли ми змінюємо вхідне значення, тобто передаємо значення у поле Num rows (рис. 2.3.), як результат значення factorialResult обчислюється знову. З іншого боку, кожного разу, коли ми змінюємо значення у чек боксі «Use dynamic row heights?», значення стану оновлюється. Оновлення значення стану запускає повторне відтворення MyComponent. Але оскільки функція повністю відпрацьовує знов, то під час повторного рендерингу calculateFactorial викликається знову також, хоча значення не було змінено.

Одним із варіантів розв'язання цієї проблеми є використання useMemo. Коли ми використовуємо useMemo замість звичайного виклику функції React запам’ятовує попередньо обчислені значення факторіалу. Значення буде перераховано тільки у тому випадку коли буде зменшена одна із залежностей. У нашому випадку єдиною залежністю є вхідне значення яке може змінитися тільки при зміні значення поля вводу. Тому при натисканні на кнопку «Increment», перерахунок не буде здійснюватися знов, тому що useMemo буде повертати попередньо збережене значення. Нижче наведено фрагмент коду з використанням useMemo.

```
export function MyComponent() {
  const [inputNumber, setInputNumber] = useState(1);
  const [increment, setIncrement] = useState(0);
  const factorialResult = useMemo(() => calculateFactorial(inputNumber), [inputNumber]);

  const onChange = (event) => {
    setInputNumber(Number(event.target.value));
  };

  const onClick = () => {
    setIncrement(increment + 1);
  };

  return (
    <div>
      Factorial of the following number:
      <input type="number" value={inputNumber} onChange={onChange} />
      is {factorialResult}
     <button onClick={onClick}>Increment</button> <span>{increment}</span>
    </div>
  );
}
```

2.2.4. React.useCallback

React.useCallback розв'язує проблему зайвого перестворення функцій при кожному рендерингу компонента. У звичайних умовах, коли ви передаєте функцію як пропс або використовуєте її в ефектах, вона перестворюється при кожному рендерингу [27]. Це може призвести до додаткового споживання пам'яті та привести до неефективності. Зазвичай ця проблема виникає лише з функціональним компонентом, а не з класовими. Для проведення дослідження ми використаємо веб-застосунок «User list», проте цього разу нас буде цікавити рендеринг списку задач та додавання та видалення елементів за допомогою обробників зворотного виклику.

Для початку ми повинні спровокувати повторну візуалізацію лише для компонента застосунку. Для того, щоб це зробити слід ввести будь-яке значення в поле «Num rows» для додавання елемента до списку. Важливо розуміти, як тільки ми вводимо значення, тоді усі дочірні компоненти будуть повторно відтворені. Наша мета - запобігти повторному відтворенню для кожного компонента, коли користувач вводить текст у поле. Для початку спробуємо використовувати React.memo.

```
const MyList = React.memo(({ items, onRemove }) => {
  console.log("The Component Render: MyList");
});
const MyListItem = React.memo(({ item, onRemove }) => {
  console.log("The Component Render: MyListItem");
});
```

Однак не зважаючи на використання React.memo обидва функціональні компоненти все ще повторно рендеряться при зміни положення списку.

Розглянемо атрибути, які передаються до компонента List. Нижче наведено компонент з атрибутами.

```
const MyApp = () => {
return (
<List list={items} onRemove = {handleRemove} />
)
```

Поки жоден елемент не додається або не видаляється з list атрибута, він має залишатися незмінним, навіть якщо компонент програми повторно відображається після того, як користувач вводить щось у поле. На даному етапі цього не вдається досягти через проблему з чистими функціями всередині функціонального компонента. Кожного разу, коли компонент програми повторно відображається після того, як хтось вводить текст у поле, функція обробки handleRemove в програмі визначається повторно. Передаючи цей новий обробник зворотного виклику як властивість до компонента List, він помічає, що властивість змінена порівняно з попереднім рендером. Ось чому відбувається повторний рендеринг для компонентів List і ListItem.

Щоб запам’ятати функцію, можна використати useCallback. Тоді функція буде повторно визначена, лише якщо зміниться будь-яка з її залежностей у масиві залежностей. Нижче наведено фрагмент коду, який демонструє функцію обробки видалення, обгорнуту в React.useCallback.

```
const MyApp = () => {
  const handleRemove = React.useCallback(
    (id) => setItems(items.filter((item) => item.id !== id)),
    [items]
  );
};
```

Якщо стан користувача змінюється шляхом додавання або видалення елемента зі списку, функція обробки визначається повторно, а дочірні компоненти мають повторно відтворюватися. Однак, якщо хтось лише взаємодіє з іншими елементами веб-додатку, наприклад використовує поле введення, функція не буде перевизначена та залишиться незмінною.

2.3. Використання React.lazy

Для наочного відображення впливу React.lazy я вирішив використовувати React веб-застосунок «to-do list». Обрання цього застосунку обумовлене бажанням імітувати ситуацію імпорту файлу великого розміру, де в цьому випадку мова йде про додавання картинки до завдань. Моя основна мета полягає в демонстрації можливостей React.lazy у взаємодії з компонентами великих розмірів. Думаю, що цей випадок допоможе нам виявити на скільки метод відкладеного завантаження виявиться ефективним.

Уявімо, що користувачеві потрібно створити декілька задач, кожна з яких містить детальний опис та прикріплену картинку. Для створення нової задачі користувач повинен лише натискати кнопку «Add Task», як результат відповідний компонент буде динамічно завантажувати дані за допомогою React.lazy.

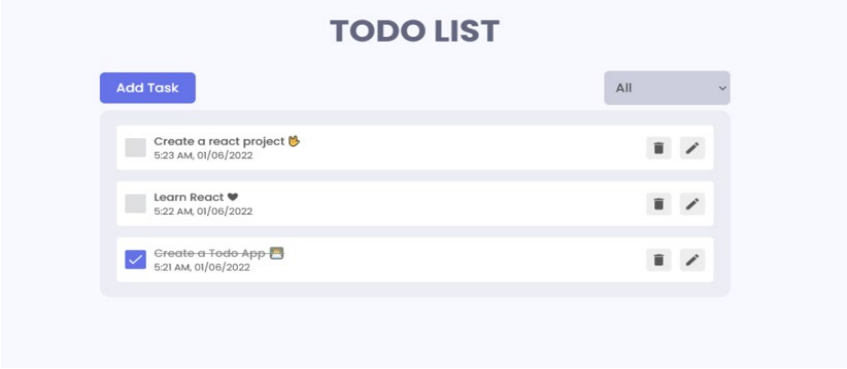

Рис. 2.2. Приклад демонстраційного головної сторінки додатка

Як результат, користувач бачить модальне вікно, де йому слід вибрати назву задачі, пріоритет та додати картинку. Задача додається до «to-do list», коли користувач натискає кнопку «Create». При додаванні картинки до декількох задач із списку, ми стикаємося з проблемою: кожен раз під час оновлення сторінки або повторного відкриття веб-застосунку картинки потрібно буде завантажувати знову. Тепер уявіть, що елементів у списку, які містять картинку, більш як 100. Такий неоптимальний підхід значно збільшить час завантаження додатку, буде витрачати інтернет-трафік користувача даремно і зробить наш додаток важким.

Підхід, який ми тестуватимемо, дозволить розв'язувати цю проблему, завантажуючи картинки лише один раз. Крім того, під час прокручування списку ми не будемо рендерити всі картинки, які знаходяться у списку.

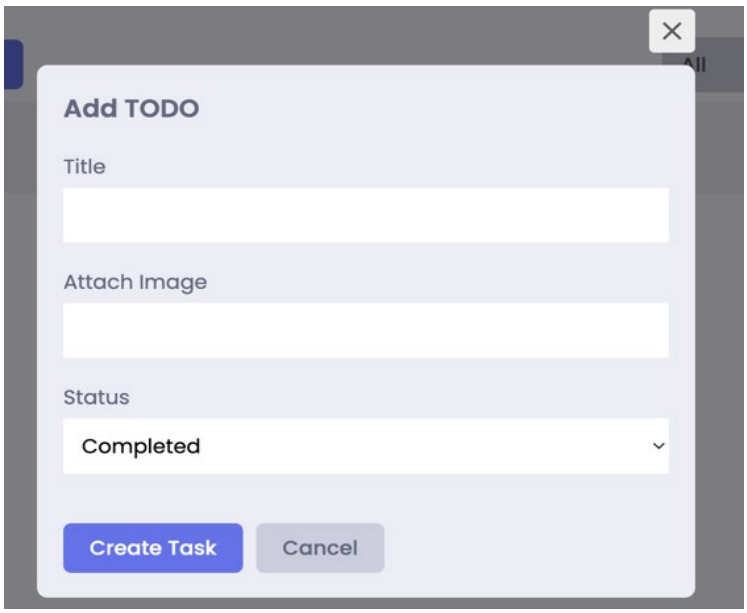

Рис. 2.3. Приклад демонстраційного модального вікна додатка

Якщо відкрити консоль розробника та перейти у вкладку Network. Потім виберіть, щоб відображались файли формату .js. Таким чином, ми зможемо

побачити перелік завантажених файлів та їх об’єм. Перелік файлів та їх розмір у веб-додатку без застосування lazy loading зображено на рисунку 2.4.

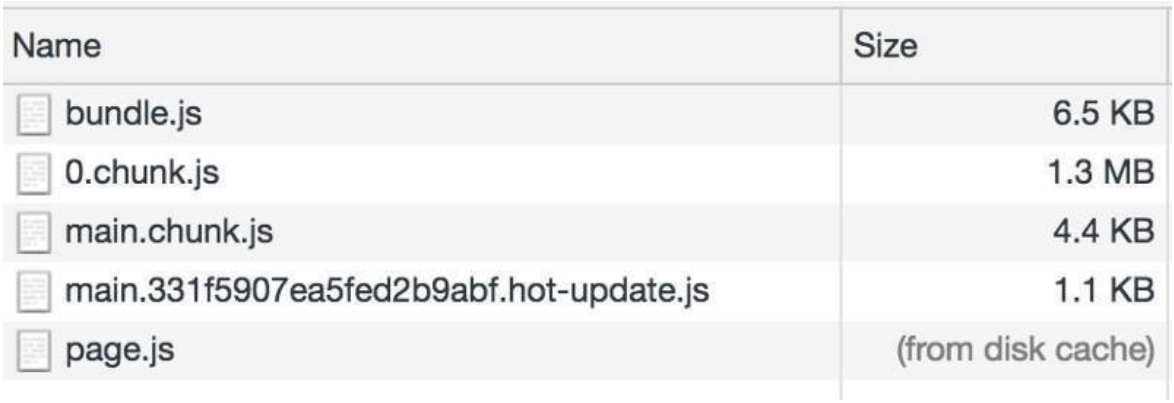

Рис. 2.4. Статистика бандлу без відкладеного завантаження

Наступним кроком в рамках експерименту, ми застосуємо до нашого додатка відкладене завантаження та розділення коду. В обох випадках ми будемо використовувати один компонент, який імпортує залежності з react-list простої задач з картинкою та її відображення.

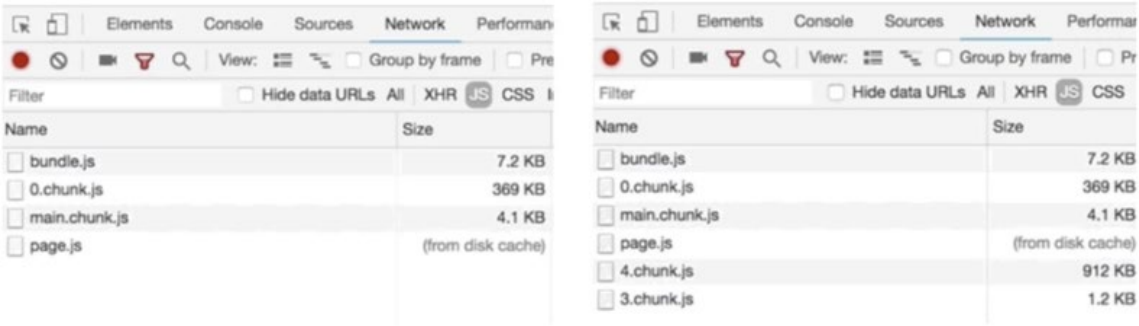

Рис. 2.5. Порівняння бандлу без та з використанням відкладеного завантаження

Ми бачимо, що 0.chunk.js важить значно менше, ніж раніше, а 4.chunk.js і 3.chunk.js завантажуються після натискання кнопки. Отже, ми можемо зробити

висновки, що використання у додатку відкладеного завантаження та розділення коду позитивно вплинуло на його швидкість завантаження та роботу.

2.4. Висновки

У цьому розділі було показано та застосовано на практиці методи оптимізації клієнтських React веб-застосунків.

Сам процес переходу від класових компонентів до функціональних здійснюється досить просто, проте слід пам'ятати, що цей процес займає певний час оскільки абсолютно всі компоненти необхідно буде вручну переписувати на функціональні компоненти. Здійснення рефакторингу та зміна класових компонентів на функціональні являється виправданою, оскільки за допомогою hooks ви можете реалізувати state та інші функції React, і, що найважливіше, можна написати весь інтерфейс користувача з функціональними компонентами . Як результат з hooks створення компонентів у React стає більш простим, а сам додаток починає споживати менше ресурсів, що робить його більш продуктивним.

Наступна проблема, яку ми вирішували у нашому веб-застосунку, це зміна значення «count», що викликає повторний рендер компонента «App» при будь-якій взаємодії з веб-додатком. Це призводить до подальшого повторного рендеру компонента «List», що викликає блокування потоку, особливо якщо в компоненті виконуються важкі обчислення. Для розв'язання цієї проблеми можна використовувати наступні інструменти для мемоізації: React.memo, React.PureComponent, React.useMemo, React.useCallback.

Першим кроком, було застосування React.memo, що допомогло зменшується час між діями користувача та відкликом програми. Далі у програмі я замінив використання React.Component на React.PureComponent, що було призвело до зменшення часу між діями користувача та відкликом програми.

Таким чином зменшується час між діями користувача та відкликом програми а також навантаження на процесор. Результатом, Важливо відзначити, що за допомогою цього підходу перерендер відбувається лише у того елементам з яким взаємодіє користувач, наприклад використовує поле введення, функція не буде перевизначена та залишиться незмінною, що своєю чергою позитивно вплинуло на продуктивність веб-застосунку.

У цьому розділі я також розглянув застосування відкладеного завантаження та поділу коду, на конкретному прикладі обробки об'ємних файлів у моєму веб-застосунку. В результаті цього дослідження варто відзначити, що використання методу оптимізації React.lazy вимагало мінімальних зусиль з мого боку, проте цей підхід виявився дуже ефективним. Зазначу, що розмір .js файлів з якими працює веб-додаток зменшився в декілька разів.

Цей варіант оптимізації не тільки зменшив обсяг завантажуваного коду, але також покращив динаміку веб-застосунку, забезпечуючи більш ефективне завантаження ресурсів. За допомогою відкладеного завантаження, ми успішно оптимізували роботу з об'ємними файлами та покращили відгук користувача, що важливо для оптимальної продуктивності веб-додатка.

РОЗДІЛ 3
АНАЛІЗ РЕЗУЛЬТАТІВ ОПТИМІЗАЦІЇ

3.1. Порівняльний аналіз класових та функціональних компонентів

З точки зору розробки функціональні компоненти потребують значно менше коду і можуть повністю замінити класові компоненти. Це вказує на те, що при однаковій складності логіки роботи компонента, класові компоненти завжди будуть масти більший обсяг коду.

З погляду теорії, можна припустити, що функціональні компоненти мають швидший час відображення, оскільки кожний виклик функції зазвичай вимагає менше часу, ніж створення екземпляра класу та виклик його методів. Для перевірки цього на практиці ми використовували раніше створені компоненти, аналізуючи час відображення кожного з них окремо. Для цієї мети ми використовували бібліотеку Benchmark, яка надає методи для збору показників часу монтування, оновлення та демонтування дерева компонентів.

Оскільки ці показники можуть залежати від потужності процесора, ресурсів, що використовуються іншими вкладками браузера чи розширенням, вимірювання було проведено в анонімній вкладці браузера. Кожне вимірювання було повторено 100 разів для кожної версії додатку.

Щоб провести збір та аналіз даних, спочатку я встановив Benchmark.js, використовуючи застосунок «Terminal» та ввівши команду npm install benchmark. Наступним моїм кроком було створення тесту продуктивності. Для цього я створив файл, який назвав functionalComponents-test.js. Далі, в цьому ж файлі, я створив та налаштував два тести, додавши до них код мого веб-застосунку. Перший тест перевіряв ефективність виконання класових компонентів, а інший - функціональних.

Після цього, використовуючи команду node performance-test.js, я запустив тести. Важливо розуміти, що для тестів продуктивності використовується підхід Benchmark.Suite, який становить собою групу тестів. Ці тести використовують ReactDOM.render для рендерингу компонентів та надають вам інформацію щодо швидкості виконання ваших класових та функціональних компонентів.
Результати тестів виводяться у консолі.

Щодо одиниць вимірювання то, під час виконання тестів Benchmark.js вимірює кількість виконаних операцій (наприклад, рендеринг компонента) та визначає час, необхідний для їх виконання [28]. Наприклад, функціональний компонент виконується приблизно 95,000 операцій за секунду.
Останнім кроком, який потрібно виконати, це імпортувати дані та впорядкувати їх у Google Sheets для створення порівняльного графіка.

На рисунку 3.1. показано результати проведеного дослідження. На осі X зазначено номер спроби, на осі У – час, від запуску веб-додатку до моменту відображення компонента.

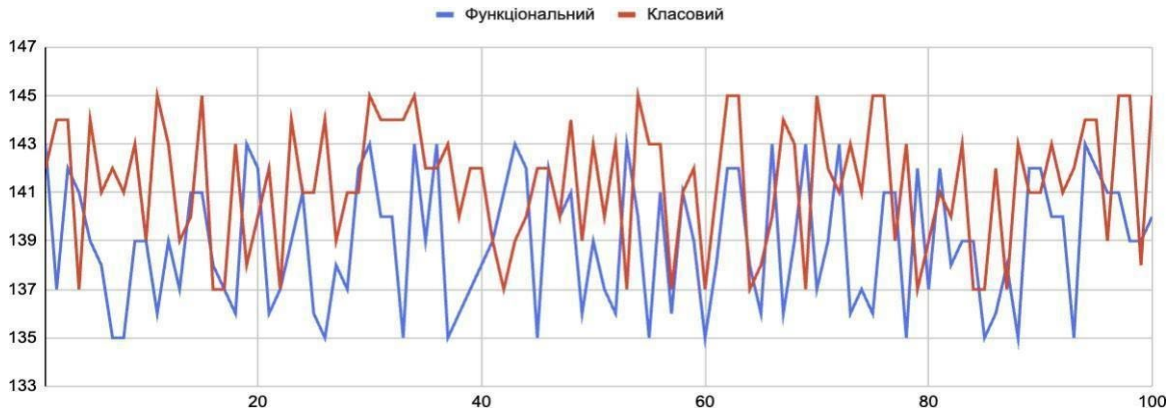

Рис. 3.1. Графік часу витраченого на відображення компонентів

Як можна побачити у середньому відображення функціональних компонентів займає менше часу ніж відображення класових компонентів. Важливо розуміти, що ця різниця не буде помітною для користувача. Також переписування великого проєкту з класових компонентів на функціональні

потребує багато зусиль та часу розробника, а також потенційно може призвести до створення багів у програмі, наприклад через неуважність, тому також потребує додаткового тестування. Зважаючи на вище наведені аргументи, рефакторинг проєкту з класових компонентів на функціональні не є раціональним. Оскільки незначне підвищення продуктивності все ж присутнє, а також скорочується кількість коду, нові додатки рекомендується писати з використанням функціональних компонентів.

3.2. Аналіз застосування мемоізації

Мемоізація, несумлінно, є потужним інструментом, але варто зазначити, що вона далеко не є універсальною панацеєю. Вдале використання цього методу вимагає розсудливого вибору та визнання відповідних сценаріїв. В цьому випадку ефективність мемоізації залежить від того, наскільки вдало ви вибираєте та оптимізуєте ситуації для її застосування.

React.memo доцільно використовувати у наступних випадках:

- При однакових вхідних властивостях компонент рендерить одне й те саме представлення.
- Відображення займає багато часу (від 1 секунди).
- Існує вірогідність частого перерендеру компоненту.

Якщо компонент не відповідає наведеним вище вимогам, може не знадобитися застосування мемоізації, оскільки використання React.memo в деяких випадках погіршує продуктивність, тому що буде витрачатися час на порівняння значень.

Для ефективного вимірювання результатів оптимізації за допомогою useMemo у веб-додатку, я використав базовий інструмент для оцінки та аналізу продуктивності веб-застосунків - Performance, який доступний всім користувачам браузера Google Chrome [28].

Щоб розпочати заміри, я відкрив веб-сторінку, де відображався мій веб-застосунок. У випадку з мемоізацією це був «User list». Наступним кроком я відкрив Developer Tools, натиснувши правою кнопкою миші на сторінці та вибравши «Inspect», а потім перейшовши на вкладку «Performance».

Для початку запису профілювання мого веб-застосунку під час використання useMemo, я натиснув кнопку «Start Recording» та зачекав 30 секунд, після чого зупинив тест, натиснувши кнопку «Stop Recording». Цей експеримент я повторив 10 разів. Результати тесту я додав у вигляді зображень, які будуть показані нижче.

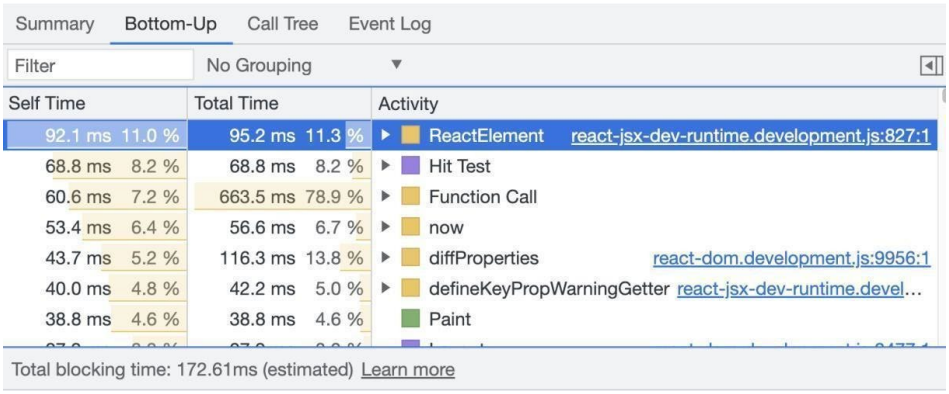

Рис. 3.2. Показники без застосування React.memo

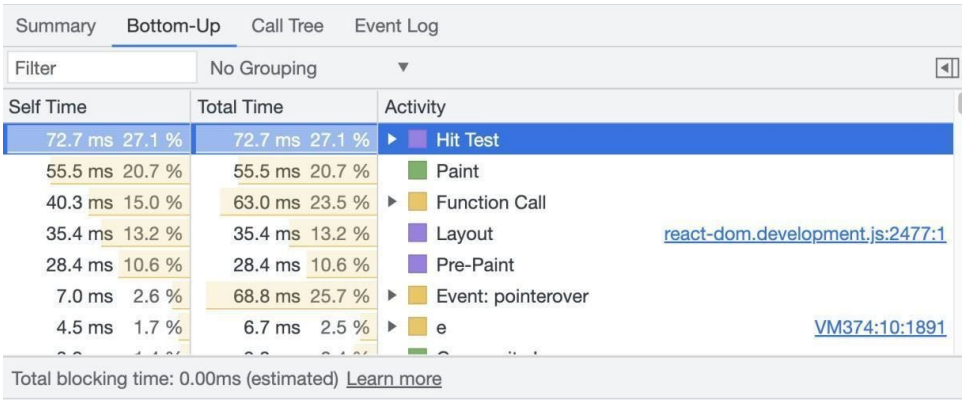

Рис. 3.3. Показники з застосуванням React.memo

Як бачимо, за показниками використання React.memo підвищує продуктивність навіть невеликих додатків. При застосуванні React.memo на реальних проєктах можна отримати значне підвищення продуктивності.

Застосування React.PureComponent приводить до аналогічного результату, але з класовими компонентами. React.PureComponent слід використовувати, дотримуючись наведених нижче правил:

- Стан або властивість мають бути незмінним об’єктом;
- Стан або властивість не можуть мати ієрархії;
- Виконувати виклик forceUpdate, коли дані змінюються.

При використанні React.PureComponent потрібно переконатися, що всі дочірні компоненти також чисті.

Для показників, які використовують useMemo підвищується продуктивність навіть невеликих додатків [8]. При застосуванні useMemo на реальних проєктах можна отримати значне підвищення продуктивності при правильному застосуванні.

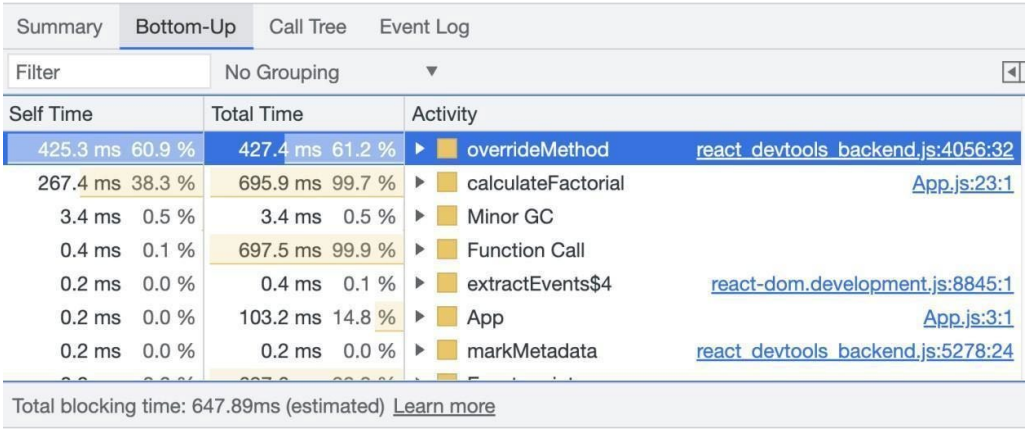

Рис. 3.4. Показники без використання useMemo

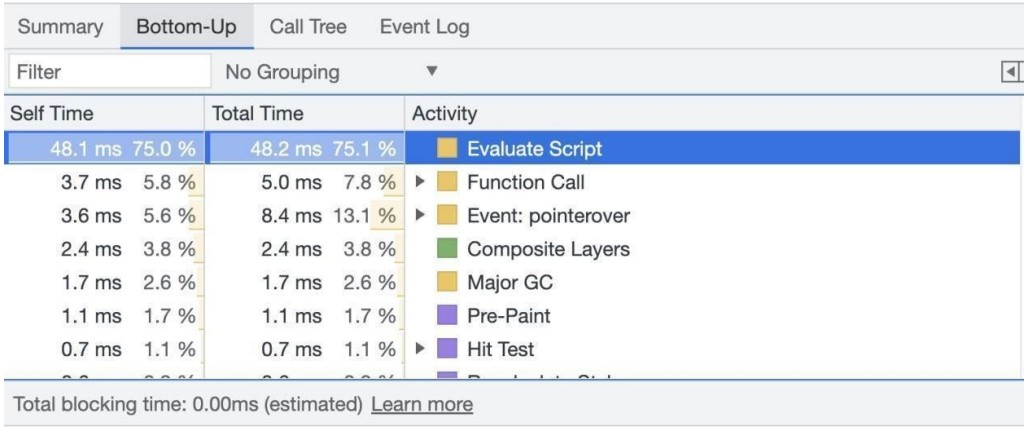

Рис. 3.5. Показники з використанням useMemo

Внутрішній hook useMemo React має порівнювати залежності з масиву залежностей для кожного повторного рендерингу, щоб вирішити, чи слід повторно обчислювати значення. Часто обчислення для цього порівняння може бути дорожчим, ніж просто повторне обчислення значення.

useMemo(() => computation(a, b), [a, b]) - це hook, який дозволяє мемоізувати дорогі обчислення [26]. Враховуючи ті самі [a, b] залежності, hook поверне мемоізоване значення без виклику computation(a, b).

Підсумовуючи, hook useCallback React використовується для запам’ятовування функцій. Це вже невеликий приріст продуктивності, коли функції передаються іншим компонентам, не турбуючись про повторну ініціалізацію функції для кожного повторного рендерингу батьківського компонента. Як ми бачили, hooks useCallback React сприяє поліпшенню продуктивності, коли використовується разом із React.memo.

3.3. Аналіз відкладеного завантаження

Основна концепція досить проста: відкладати завантаження усього, що необхідно користувачеві прямо зараз. Цей метод можна застосувати до будь-якого зображення, яке користувач спочатку не бачить. При прокрутці сторінки

вниз плейсхолдери зображень переміщуються в область видимості (видима частина сторінки), і зображення завантажуються, лише коли вони стають видимими.

За допомогою розширення Lighthouse для браузера Google Chrome можна визначити, які саме зображення підходять для відкладеного завантаження і скільки трафіку можна заощадити [28]. У розширенні є розділ, присвячений закадровим зображенням. React.lazy є ефективним інструментом, який заслуговує на увагу у розробці програмного забезпечення. Він дозволяє оптимізувати розмір бандлу вашого додатка, вибірково завантажуючи лише ті компоненти, які необхідні користувачеві. Це концепт відомий як "ліниве завантаження", що означає, що компоненти завантажуються лише в той момент, коли вони дійсно потрібні.

Ліниве завантаження може виявитися особливо важливим в контексті оптимізації продуктивності веб-додатків. Цей підхід дозволяє скоротити час завантаження сторінок, оскільки лише обов'язкові компоненти будуть включені в початковий бандл. Важливо розуміти, що використання React.lazy вимагає обізнаності з іншими ключовими концепціями розробки, зокрема "бандлінг" та "розділення коду".

"Бандлінг" належать до процесу об'єднання різних файлів програмного коду в один бандл, що допомагає зменшити обсяг завантажуваного контенту. "Розділення коду" означає розбиття коду на менші частини, що може бути корисним для оптимізації завантаження лише необхідних частин додатку.

Для уникнення збільшення розміру бандлу, рекомендується вже на етапі планування розпочати процес його розбиття на компоненти. Розбиття коду є функціоналом, який підтримується такими бандлерами, як Webpack, Rollup та Browserify (за допомогою factor-bundle). Ці інструменти можуть створювати декілька окремих бандлів, які можна динамічно завантажувати під час виконання застосунку.

Розбиття коду вашого додатка дозволяє "ліниво" завантажувати лише ті компоненти, які необхідні користувачеві у конкретний момент. Це може суттєво покращити продуктивність вашого додатка. Навіть якщо загальний обсяг коду не зменшується, ви уникаєте завантаження того коду, який може залишитися невикористаним користувачем, та зменшуєте обсяг коду, необхідного для завантаження при запуску додатка.

На основі прикладу нижче вказано, що Component1 та Component2 передбачають велику бізнес логіку та містять значну кількість коду. Враховуючи умови, можна стверджувати, що ми використовуємо лише один з цих компонентів в будь-який момент часу. Однак, оскільки ми імпортуємо обидва компоненти під час початкового завантаження сторінки, ми неефективно використовуємо ресурси та збільшуємо розмір пакета бандлу більше, ніж необхідно.

Це неприйнятно, оскільки це може призводити до проблем із завантаженням для користувачів та створювати враження про повільну роботу додатка.

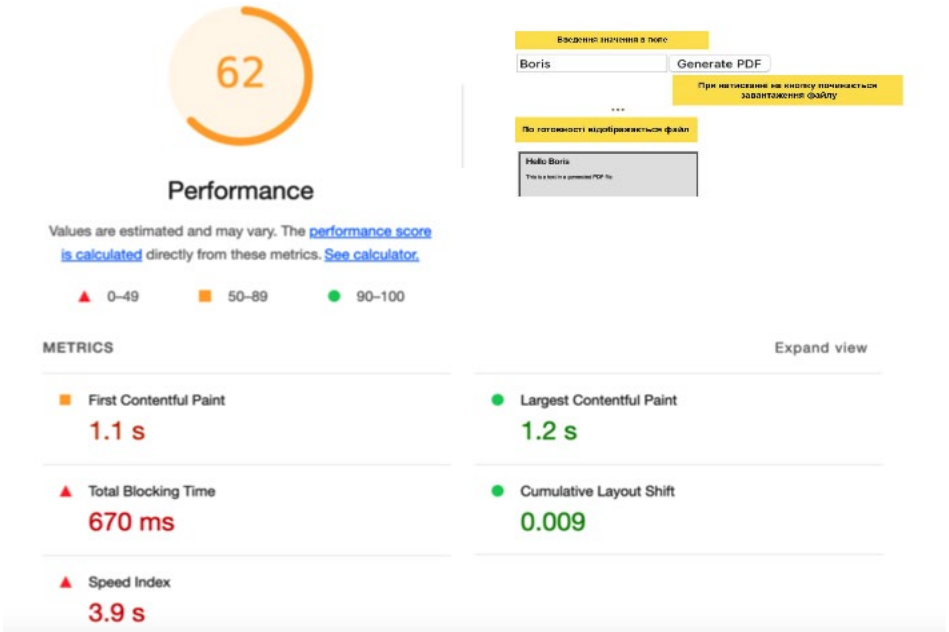

Рис. 3.6. Показники без використання React.lazy

Застосуємо React.lazy до цього коду, що дозволяє динамічно відображати компоненти за допомогою динамічного імпорту. Він автоматично завантажить необхідний пакет бандлів при першому завантаженні сторінки. Як ми можемо спостерігати на прикладі вище React.lazy приймає функцію як параметр з динамічним імпортом у ній і повертає Promise, що містить розв'язаний модуль з експортованим за замовчуванням компонентом. Оскільки процес є асинхронним, компоненти з "лінивим завантаженням" повинні бути вкладені в компонент Suspense, який можна використовувати для вказівки на те, що модуль завантажується.

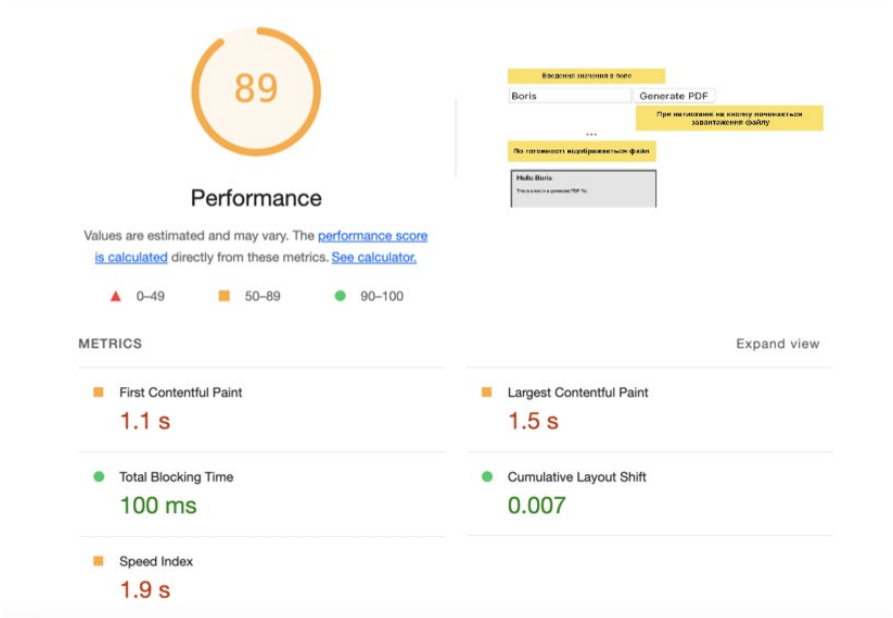

Рис. 3.7. Показники з використання React.lazy

Застосування відкладеного завантаження та розділення коду допомагає зменшити розмір файлів, які завантажуються при початковому завантаженні сторінки. Такий підхід зменшує час очікування до першої взаємодії зі сторінкою. Файли, які не використовуються на початковій сторінці, будуть завантажені за умови, що відповідна частина додатку використовується користувачем, але це завантаження буде займати деякий час залежно від швидкості з'єднання.

3.4 Висновки

В даному розділі був проведений комплексний аналіз стратегій оптимізації для веб-додатків, який був зосереджений на трьох ключових аспектах : аналіз результатів оптимізації компонентів, використані мемоізації та відкладеному завантаженні.

Отримані результати вказують на перевагу функціональних компонентів у зменшенні коду та поліпшенні продуктивності. Застосування мемоізації дозволяє значно підвищити відгук додатку. Відкладене завантаження та ліниве завантаження компонентів виявляються ефективними стратегіями для оптимізації часу завантаження та розміру бандлу.

Використання сучасних методів оптимізації дає можливість досягнення суттєвого покращення продуктивності веб додатків. Однак необхідно враховувати специфіку проєкту та розглядати кожну оптимізацію в контексті конкретних вимог та обмежень.

ВИСНОВКИ

У рамках кваліфікаційної роботи проведене дослідження методів оптимізації React веб-додатку. Розглянуто та застосовано на практиці такі методи, як перехід від класових до функціональних компонентів, застосування мемоізації, підключення відкладеного завантаження.

Відкладене завантаження зменшує розмір файлу, який використовується для початкового завантаження сторінки. Використання мемоізації та функціональних компонентів скорочує час відклику програми на дії користувача. Застосування відкладеного завантаження та поділу коду, допомагає зменшити час завантаження сторінки та її розмір, що відповідно знизить вживання інтернет-трафіку.

Серед зазначених методів найбільше часу на реалізацію займає перехід від класових компонентів на функціональні та при цьому цей метод має найменший вплив на продуктивність додатка.

Метод оптимізації мемоізація зазвичай не потребує значних зусиль з боку розробника, але вимагає чіткого розуміння, коли цей метод доцільно використовувати. При рекомендованому застосуванні мемоізація дозволяє значно поліпшити продуктивність додатка.

Відкладене завантаження доцільно використовувати, коли розмір початкового файлу є достатньо великим і змушує користувача чекати на завантаження сторінки. Оскільки React широко використовується для створення середніх та великих веб-додатків, є актуальною проблема їх оптимізації. Результати цього дослідження можуть бути використані з метою підвищення продуктивності React веб-додатків.

Підсумовуючи проведене дослідження та глибокий аналіз методів оптимізації клієнтських веб-застосунків React, варто відзначити, що результати перевищили будь-які очікування. Кожен використаний метод дав значущий

приріст продуктивності до веб-додатку. Зазначу, що впровадження цих методів не тільки сприяло покращенню швидкодії, але й суттєво підвищило ефективність роботи системи у цілому.

Однак, ключовим фактором в успішності оптимізації є не лише самі методи, але й вірне дотримання правил і рекомендацій, викладених у цьому дослідженні. Оптимальний вибір та грамотне поєднання різних методів оптимізації відповідно до специфіки вашого веб-застосунку являється основною запорукою високих результатів.

Слід підкреслити, що досягнення позитивного економічного впливу - один із головних результатів впровадження оптимізаційних заходів. Зниження витрат на підтримку та інфраструктуру дозволить вам ефективніше використовувати ресурси та вкладати збережені кошти у подальший розвиток.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Хорошко В.О., Огаркова І.М., Чирков Д.В., Голего А.Г., Горохова Т.Б. Термінологічний довідник з питань захисту інформації. / За ред. проф. Хорошка В.О. – К.: Ей-Бі-Сі, 2002.

2. www2.deloitte.com [Електронний ресурс] Режим доступу: https://www2.deloitte.com/content/dam/Deloitte/ie/Documents/Consulting/Milliseconds_Make_Millions_report.pdf

3. Чиста Архітектура: Мистецтво розроблення програмного забезпечення / пер. з англ. І. Бондар-Терещенко. – Харків : Вид-во «Ранок» : Фабула, 2019. – 368 с. – Бібліогр.: с. 156-158

4. Харченко О.Г. Службовий твір Комп'ютерна програма "Архітектор програмних систем" / О.Г.Харченко, І.Е.Райчев, О.А.Щербак, Б.С.Павленко, І.О.Боднарчук // Свідоцтво про реєстрацію авторського права на твір №59631. Видане державною службою інтелектуальної власності України 13.05.2015, м.Київ, НАУ.

5. Anthony Gore — Full-Stack Javascript [Електронний ресурс]. — 2015. – Режим доступу: https://bit.ly/2OEODzR

6. JavaScript [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/JavaScript.

7. Сучасний підручник JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://uk.javascript.info/.

8. Програмуваня по-українськи URL: programming.in.ua (дата звернення: 03.05.2021).

9. devzone.org.ua [Електронний ресурс] Режим доступу: https://devzone.org.ua/post/vstup-v-react-yakogo-nam-ne-vistachalo

10. Мова програмування Javascript [Електронний ресурс]: https://uk.wikipedia.org/wiki/JavaScript

11. dou.ua [Електронний ресурс] Режим доступу: https://dou.ua/lenta/articles/language-rating-2023/

12. Bertolli M. React Design Patterns and Best Practices: Build easy to scale modular applications using the most powerful components and design patterns. — Packt Publishing, 2017. — 320 p.

13. appmaster.io [Електронний ресурс] Режим доступу: https://appmaster.io/blog/popular-frontend-frameworks

14. uk.reactjs.org [Електронний ресурс] Режим доступу: https://uk.reactjs.org/

15. beta.reactjs.org [Електронний ресурс] Режим доступу: https://beta.reactjs.org/

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

17. Oleshchenko L., Burchak P. Analysis and optimization of methods for storing and processing data of web applications. Прикладна математика та комп'ютинг. ПМК – 2021. Збірник тез доповідей XIV наукової конференції магістрантів та аспірантів. Київ, 17-19 листопада, 2021. pp. 59-63

18. dou.ua [Електронний ресурс] Режим доступу: https://dou.ua/forums/topic/45406/

19. Сухий О. Л. Алгоритми пошуку в інформаційних системах : методичні рекомендації / О. Л. Сухий, В. М. Міленін, В. М. Тарадайнік. – К., 2015. – С. 6 – 53.

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

21. Gorgon Z. React Explained: Your Step-by-Step Guide to React. – Amazon Digital Services LLC, 2019. — 305 p

22. Banks A., Porcello E. Learning React: Functional Web Development with React and Redux. — O'Reilly Media, 2017. — 350 p.

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

24. Вовк О. О. Методи підвищення стійкості та пропускної здатності систем прихованої передачі інформації: дисертація на здобуття наукового ступеня кандидата технічних наук / О. О. Вовк. – Харків, 2016. – 177с.

25. Metrics for software tools assessment by standard ISO/IEC 14102:1995 «Information technology – Guideline for the evaluation and selection of CASE tools».

26. Kyle Simpson. You Don't Know JS: ES6 & Beyond — Published by O'Reilly Media, 2015 — P. 278

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

28. Google PageSpeed Insights [Електронний ресурс] – Режим доступу: https://developers.google.com/web/tools/lighthouse/v3/scoring

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

```
import { configureStore } from '@reduxjs/toolkit';
import todoReducer from '../slices/todoSlice';

export const store = configureStore({
  reducer: {
    todo: todoReducer,
  },
});

import { AnimatePresence, motion } from 'framer-motion';
import React from 'react';
import { useSelector } from 'react-redux';
import styles from '../styles/modules/app.module.scss';
import TodoItem from './TodoItem';

const container = {
  hidden: { opacity: 1 },
  visible: {
    opacity: 1,
    scale: 1,
    transition: {
      staggerChildren: 0.2,
    },
  },
};
const child = {
  hidden: { y: 20, opacity: 0 },
  visible: {
    y: 0,
    opacity: 1,
  },
};

function AppContent() {
  const todoList = useSelector((state) => state.todo.todoList);
```

```
  const filterStatus = useSelector((state) => state.todo.filterStatus);

  const sortedTodoList = [...todoList];
  sortedTodoList.sort((a, b) => new Date(b.time) - new Date(a.time));

  const filteredTodoList = sortedTodoList.filter((item) => {
    if (filterStatus === 'all') {
      return true;
    }
    return item.status === filterStatus;
  });

const CounterApp = () => {
  const [count, setCount] = useState(0);
  useEffect(0) = setCount((currentCount) => currentCount + 1), []);


  const manualIncrement = () => {
    setCounterAppValue((currentValue) => currentValue + 1);
    };


  const manualDecrement = () => {
    setCounterAppValue((currentValue) => currentValue - 1);
                  };
  return (
    <div className="counterApp">
      <h1 className="count">{counterValue}</h1>
      <button type="button" onClick={manualIncrement}>
        Increment
      </button>
      <button type="button" onClick={manualDecrement}>
        Decrement
      </button>
    </div>
  );
};
export default CounterAppComponent;
  return (
```

```
    <motion.div
      className={styles.content__wrapper}
      variants={container}
      initial="hidden"
      animate="visible"
    >
      <AnimatePresence>
        {filteredTodoList && filteredTodoList.length > 0 ? (
          filteredTodoList.map((todo) => (
            // <motion.div key={todo.id} variants={child}>
            <TodoItem key={todo.id} todo={todo} />
            // </motion.div>
          ))
        ) : (
          <motion.p variants={child} className={styles.emptyText}>
            No Todos
          </motion.p>
        )}
      </AnimatePresence>
    </motion.div>
  );
}

export default AppContent;
import React, { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import Button, { SelectButton } from './Button';
import styles from '../styles/modules/app.module.scss';
import TodoModal from './TodoModal';
import { updateFilterStatus } from '../slices/todoSlice';

function AppHeader() {
  const [modalOpen, setModalOpen] = useState(false);
  const initialFilterStatus = useSelector((state) => state.todo.filterStatus);
  const [filterStatus, setFilterStatus] = useState(initialFilterStatus);
  const dispatch = useDispatch();

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

```
    dispatch(updateFilterStatus(e.target.value));
  };

  return (
    <div className={styles.appHeader}>
      <Button variant="primary" onClick={() => setModalOpen(true)}>
        Add Task
      </Button>
      <SelectButton
        id="status"
        onChange={(e) => updateFilter(e)}
        value={filterStatus}
      >
        <option value="all">All</option>
        <option value="incomplete">Incomplete</option>
        <option value="complete">Completed</option>
      </SelectButton>
      <TodoModal type="add" modalOpen={modalOpen} setModalOpen={setModalOpen} />
    </div>
  );
}

export default AppHeader;



import React from 'react';
import styles from '../styles/modules/button.module.scss';
import { getClasses } from '../utils/getClasses';

const buttonTypes = {
  primary: 'primary',
  secondary: 'secondary',
};

function Button({ type, variant = 'primary', children, ...rest }) {
  return (
    <button
      type={type === 'submit' ? 'submit' : 'button'}
      className={getClasses([
```

```
        styles.button,
        styles[`button--${buttonTypes[variant]}`],
      ])}
      {...rest}
    >
      {children}
    </button>
  );
}

function SelectButton({ children, id, ...rest }) {
  return (
    <select
      id={id}
      className={getClasses([styles.button, styles.button__select])}
      {...rest}
    >
      {children}
    </select>
  );
}

export { SelectButton };
export default Button;

import { motion, useMotionValue, useTransform } from 'framer-motion';
import React from 'react';
import styles from '../styles/modules/todoItem.module.scss';

const checkVariants = {
  initial: {
    color: '#fff',
  },
  checked: { pathLength: 1 },
  unchecked: { pathLength: 0 },
};

const boxVariants = {
  checked: {
```

```
    background: 'var(--primaryPurple)',
    transition: { duration: 0.1 },
  },
  unchecked: { background: 'var(--gray-2)', transition: { duration: 0.1 } },
};

function CheckButton({ checked, handleCheck }) {
  const pathLength = useMotionValue(0);
  const opacity = useTransform(pathLength, [0.05, 0.15], [0, 1]);

  return (
    <motion.div
      animate={checked ? 'checked' : 'unchecked'}
      className={styles.svgBox}
      variants={boxVariants}
      onClick={() => handleCheck()}
    >
      <motion.svg
        className={styles.svg}
        viewBox="0 0 53 38"
        fill="none"
        xmlns="http://www.w3.org/2000/svg"
      >
        <motion.path
          variants={checkVariants}
          animate={checked ? 'checked' : 'unchecked'}
          style={{ pathLength, opacity }}
          fill="none"
          strokeMiterlimit="10"
          strokeWidth="6"
          d="M1.5 22L16 36.5L51.5 1"
          strokeLinejoin="round"
          strokeLinecap="round"
        />
      </motion.svg>
    </motion.div>
  );
}
```

```
export default CheckButton;
import { format } from 'date-fns';
import { motion } from 'framer-motion';
import toast from 'react-hot-toast';
import React, { useEffect, useState } from 'react';
import { MdDelete, MdEdit } from 'react-icons/md';
import { useDispatch } from 'react-redux';
import { deleteTodo, updateTodo } from '../slices/todoSlice';
import styles from '../styles/modules/todoItem.module.scss';
import { getClasses } from '../utils/getClasses';
import CheckButton from './CheckButton';
import TodoModal from './TodoModal';

const child = {
  hidden: { y: 20, opacity: 0 },
  visible: {
    y: 0,
    opacity: 1,
  },
};

function TodoItem({ todo }) {
  const dispatch = useDispatch();
  const [checked, setChecked] = useState(false);
  const [updateModalOpen, setUpdateModalOpen] = useState(false);

  useEffect(() => {
    if (todo.status === 'complete') {
      setChecked(true);
    } else {
      setChecked(false);
    }
  }, [todo.status]);

  const handleCheck = () => {
    setChecked(!checked);
    dispatch(
      updateTodo({ ...todo, status: checked ? 'incomplete' : 'complete' })
    );
```

```
};

const handleDelete = () => {
  dispatch(deleteTodo(todo.id));
  toast.success('Todo Deleted Successfully');
};

const handleUpdate = () => {
  setUpdateModalOpen(true);
};

return (
  <>
    <motion.div className={styles.item} variants={child}>
      <div className={styles.todoDetails}>
        <CheckButton checked={checked} handleCheck={handleCheck} />
        <div className={styles.texts}>
          <p
            className={getClasses([
              styles.todoText,
              todo.status === 'complete' && styles['todoText--completed'],
            ])}
          >
            {todo.title}
          </p>
          <p className={styles.time}>
            {format(new Date(todo.time), 'p, MM/dd/yyyy')}
          </p>
        </div>
      </div>
      <div className={styles.todoActions}>
        <div
          className={styles.icon}
          onClick={() => handleDelete()}
          onKeyDown={() => handleDelete()}
          tabIndex={0}
          role="button"
        >
          <MdDelete />
```

```
          </div>
          <div
            className={styles.icon}
            onClick={() => handleUpdate()}
            onKeyDown={() => handleUpdate()}
            tabIndex={0}
            role="button"
          >
            <MdEdit />
          </div>
        </div>
      </motion.div>
      <TodoModal
        type="update"
        modalOpen={updateModalOpen}
        setModalOpen={setUpdateModalOpen}
        todo={todo}
      />
    </>
  );
}

export default TodoItem;

const ItemList = React.memo(({ items }) => {
  console.log('renderItemList');
  return items.map((item, index) => (
    <div key={index}>Item: {item.text}</div>
  ));
});


export default function MyApp() {
  console.log('renderMyApp');
  const [totalCount, setTotalCount] = useState(0);
  const [itemList, setItemList] = useState(getInitialItems(10000));


  return (
```

```
    <div>
      <h1>{totalCount}</h1>
      <button onClick={() => setTotalCount(totalCount + 1)}>
        Increment
      </button>
      <ItemList items={itemList} />
    </div>
  );
}
export function MyComponent() {
  const [inputNumber, setInputNumber] = useState(1);
  const [increment, setIncrement] = useState(0);
  const factorialResult = useMemo(() => calculateFactorial(inputNumber), [inputNumber]);

  const onChange = (event) => {
    setInputNumber(Number(event.target.value));
  };

  const onClick = () => {
    setIncrement(increment + 1);
  };

  return (
    <div>
        Factorial of the following number:
        <input type="number" value={inputNumber} onChange={onChange} />
        is {factorialResult}
      <button onClick={onClick}>Increment</button> <span>{increment}</span>
    </div>
  );
}



import React, { useEffect, useState } from 'react';
import { v4 as uuid } from 'uuid';
import { MdOutlineClose } from 'react-icons/md';
import { useDispatch } from 'react-redux';
import { AnimatePresence, motion } from 'framer-motion';
import toast from 'react-hot-toast';
```

```
import { format } from 'date-fns';
import { addTodo, updateTodo } from '../slices/todoSlice';
import styles from '../styles/modules/modal.module.scss';
import Button from './Button';

const dropIn = {
  hidden: {
    opacity: 0,
    transform: 'scale(0.9)',
  },
  visible: {
    transform: 'scale(1)',
    opacity: 1,
    transition: {
      duration: 0.1,
      type: 'spring',
      damping: 25,
      stiffness: 500,
    },
  },
  exit: {
    transform: 'scale(0.9)',
    opacity: 0,
  },
};

function TodoModal({ type, modalOpen, setModalOpen, todo }) {
  const dispatch = useDispatch();
  const [title, setTitle] = useState('');
  const [status, setStatus] = useState('incomplete');

  useEffect(() => {
    if (type === 'update' && todo) {
      setTitle(todo.title);
      setStatus(todo.status);
    } else {
      setTitle('');
      setStatus('incomplete');
    }
```

```
}, [type, todo, modalOpen]);

const handleSubmit = (e) => {
  e.preventDefault();
  if (title === '') {
    toast.error('Please enter a title');
    return;
  }
  if (title && status) {
    if (type === 'add') {
      dispatch(
        addTodo({
          id: uuid(),
          title,
          status,
          time: format(new Date(), 'p, MM/dd/yyyy'),
        })
      );
      toast.success('Task added successfully');
    }
    if (type === 'update') {
      if (todo.title !== title || todo.status !== status) {
        dispatch(updateTodo({ ...todo, title, status }));
        toast.success('Task Updated successfully');
      } else {
        toast.error('No changes made');
        return;
      }
    }
    setModalOpen(false);
  }
};

return (
  <AnimatePresence>
    {modalOpen && (
      <motion.div
        className={styles.wrapper}
        initial={{ opacity: 0 }}
```

```
  animate={{ opacity: 1 }}
  exit={{ opacity: 0 }}
>
  <motion.div
    className={styles.container}
    variants={dropIn}
    initial="hidden"
    animate="visible"
    exit="exit"
  >
    <motion.div
      className={styles.closeButton}
      onKeyDown={() => setModalOpen(false)}
      onClick={() => setModalOpen(false)}
      role="button"
      tabIndex={0}
      // animation
      initial={{ top: 40, opacity: 0 }}
      animate={{ top: -10, opacity: 1 }}
      exit={{ top: 40, opacity: 0 }}
    >
      <MdOutlineClose />
    </motion.div>

    <form className={styles.form} onSubmit={(e) => handleSubmit(e)}>
      <h1 className={styles.formTitle}>
        {type === 'add' ? 'Add' : 'Update'} TODO
      </h1>
      <label htmlFor="title">
        Title
        <input
          type="text"
          id="title"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
        />
      </label>
      <label htmlFor="type">
        Status
```

```
            <select
              id="type"
              value={status}
              onChange={(e) => setStatus(e.target.value)}
            >
              <option value="incomplete">Incomplete</option>
              <option value="complete">Completed</option>
            </select>
          </label>
          <div className={styles.buttonContainer}>
            <Button type="submit" variant="primary">
              {type === 'add' ? 'Add Task' : 'Update Task'}
            </Button>
            <Button variant="secondary" onClick={() => setModalOpen(false)}>
              Cancel
            </Button>
          </div>
        </form>
      </motion.div>
    </motion.div>
  )}
 </AnimatePresence>
);
}

export default TodoModal;
```

ДОДАТОК Б

ПЕРЕЛІК ФАЙЛІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файлу	**Опис**
Пояснювальні документи	
Диплом_Кутюк О.І.doc	Пояснювальна записка до проєкту. Документ Word.
Диплом_Кутюк О.І.pdf	Пояснювальна записка до проєкту в форматі PDF.
Програма	
Program.rar	Архів, що містить коди програми для запуску проєкту.
Презентація	
Презентація_Кутюк.ppt	Презентація для проєкту.