МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: «ОПТИМІЗАЦІЯ РЕНДЕРИНГУ REACT КОМПОНЕНТІВ»

Виконав: студент 2 курсу, групи 8.1212-іпз-1
спеціальність 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
Д.С. Кравченко
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Горбенко В.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп’ютерних наук, доцент, к.т.н. Борю С.Ю.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної
інженерії, к.ф.-м.н., доцент

________________ Лісняк А.О.
(підпис)

“ _____ ” ___________ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Кравченку Данилу Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Оптимізація рендерингу React компонентів

керівник роботи Горбенко Віталій Іванович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » травня 2023 року № 642-с

2. Строк подання студентом роботи 27.11.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
    1. Постановка задачі.
    2. Основні теоретичні відомості.
    3. Оптимізація рендерингу React компонентів.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) _______
презентація

6. Консультанти розділів роботи

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

7. Дата видачі завдання 03.05.2023 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	17.05.2023	
2.	Збір вихідних даних.	07.06.2023	
3.	Обробка методичних та теоретичних	28.06.2023	
	джерел.		
4.	Розробка першого та другого розділу.	30.08.2023	
6.	Розробка третього розділу.	08.11.2023	
7.	Оформлення та нормоконтроль	20.11.2023	
	кваліфікаційної роботи магістра.		
8.	Захист кваліфікаційної роботи.	14.12.2023	

Студент ______________ Д.С. Кравченко
(підпис) (ініціали та прізвище)

Керівник роботи ______________ В.І. Горбенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер ______________ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Оптимізація рендерингу React компонентів»: 55 с., 26 рис., 18 джерел, 4 додатки.

ОПТИМІЗАЦІЯ РЕНДЕРІНГУ, CSR, REACT, SSR, SSG, VIRTUAL DOM.

Об’єкт дослідження – процес оптимізації React додатків.

Мета роботи: дослідити роботу рендера компонентів React та вивчити механізми, які використовує React для оптимізації клієнтських додатків. На прикладі сайту по продажу авто проаналізувати, які проблеми з оптимізацією вебдодатків вирішує React.

Метод дослідження: у процесі дослідження використовувалися методичні вказівки, включаючи аналіз літературних джерел, експерименти та практичні дослідження, зокрема, використання інструментів для профілювання та вимірювання продуктивності React додатків.

Об’єкт дослідження даної кваліфікаційної роботи – це робота рендера компонентів React у контексті оптимізації клієнтських вебдодатків. Тобто, об’єктом дослідження є сам процес рендера компонентів, який використовується в React, та його можливості для покращення продуктивності та оптимізації вебдодатків.

SUMMARY

Master's qualifying paper «Optimization of the React Components Rendering»: 55 pages, 26 figures, 18 references, 4 supplements.

RENDER OPTIMIZATION, CSR, REACT, SSR, SSG, VIRTUAL DOM.

Research object is the process of optimizing React applications.

Objective of the work: to investigate the rendering of React components and study the mechanisms that React uses to optimize client applications. Using the example of a car sales website, analyze the optimization problems that React solves for web applications.

Research method: the research process involved the use of methodological guidelines, including the analysis of literary sources, experiments, and practical investigations, including the use of tools for profiling and measuring the performance of React applications.

The object of research in this qualification work is the rendering of React components in the context of optimizing client web applications. In other words, the research object is the rendering process of components used in React and its capabilities for improving productivity and optimizing web applications.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Аналіз та порівняння типів рендерінгу у веброзробці: CSR, SSR, SSG 9
1.1 Client-Side Rendering (CSR) 9
1.2 Server-Side Rendering (SSR) 11
1.3 Static Site Generation (SSG) 12
1.4 Аналітика рішень на ринку 14
2 Принципи використання та задачі React 16
2.1 Проблеми, які вирішує Virtual DOM (Застосування Virtual DOM) . 16
2.2 Принципи роботи React 17
2.3 Сценарій оновлення 19
2.4 Алгоритм порівняння дерев 20
2.5 Пріоритизація 23
3 Варіанти вирішення проблем та оптимізація 24
3.1 Покращення продуктивності при використанні реекспорту 25
3.2 Профілювання додатку React для виявлення вузьких місць 27
3.3 Зберігати стан компонентів на місцевому рівні, де це необхідно ... 29
3.4 Запам'ятовування React-компонентів, щоб уникнути непотрібних повторних рендерів 30
3.5 Розділення коду в React за допомогою динамічного імпорту 33
3.6 Lazy loading зображень у React 34
3.7 Використання незмінних структур даних (immutable data structures) 37
3.8 Застосування web-workers у React 38
3.9 React ViewPort List 41

Висновки 42
Перелік посилань.. 43
Додаток А Синтаксис експортів 45
Додаток Б Компонент CarSelectionForm 47
Додаток В Компонент SelectComponent 50
Додаток Г Приклад використання react-viewport-list 54

ВСТУП

У сучасному світі інформаційних технологій розробка вебдодатків стала вельми актуальною та широко застосовуваною галуззю програмування. Велика частина цих додатків побудована на основі бібліотек та фреймворків, серед яких React від Facebook є одним із найпопулярніших. Однак, зростання складності та функціональності вебдодатків призводить до збільшення навантаження на клієнтську сторону, що може вплинути на продуктивність та ресурсоємність додатків. Тому тема оптимізації рендеру React компонентів є надзвичайно важливою та актуальною для розробників.

Мета цієї дипломної роботи полягає в дослідженні роботи рендера компонентів React та вивченні механізмів, які використовує React для оптимізації клієнтських додатків. Наша мета – розібратися у принципах роботи React і з’ясувати, яким чином він може допомогти вирішити проблеми з оптимізацією вебдодатків. Для досягнення цієї мети ми взяли за основу конкретний приклад – сайт по продажу автомобілів. Це надає нам можливість детально проаналізувати проблеми та можливості оптимізації, які стосуються реального вебдодатку.

У подальших розділах дипломної роботи буде розглянуто та проаналізовано роботу React, різні методи оптимізації та їх застосування на прикладі сайту по продажу авто. Ми також розглянемо вплив цих оптимізаційних заходів на продуктивність та ефективність вебдодатку.

Ця дипломна робота спрямована на збагачення знань в галузі веброзробки та розуміння принципів оптимізації React додатків, що може бути корисним як для розробників, так і для користувачів сучасних вебдодатків.

1 АНАЛІЗ ТА ПОРІВНЯННЯ ТИПІВ РЕНДЕРІНГУ У ВЕБРОЗРОБЦІ: CSR, SSR, SSG

В сучасному світі веброзробка стає все більш складною та різноманітною завдяки зростанню вимог до ефективності, швидкодії та оптимізації вебдодатків. Однією з ключових відмінностей у веброзробці є вибір типу рендерінгу – стратегії відображення контенту на сторінці вебдодатку. У цьому контексті розглядаються та порівнюються три основних підходи: Client-Side Rendering (CSR), Server-Side Rendering (SSR) та Static Site Generation (SSG).

Метою цього розділу є ретельний аналіз кожного типу рендерінгу та їх впливу на ефективність, швидкодію та користувацький досвід вебдодатків. Досліджуються переваги та недоліки кожного методу, а також враховується їх відповідність конкретним вимогам та сценаріям використання. Зроблено акцент на виборі оптимального типу рендерінгу для розв’язання конкретних завдань веброзробки.

Розділ розпочинається з аналізу актуальності проблеми та надає вступне уявлення про важливість правильного вибору стратегії рендерінгу для досягнення максимальної ефективності та задоволення потреб користувачів. Цей розділ допоможе розкрити та висвітлити важливі аспекти веброзробки, спростить процес вибору стратегії рендерінгу та надасть підстави для подальших досліджень у цій області.

1.1 Client-Side Rendering (CSR)

Client-Side Rendering (CSR) [1] є підходом до рендерінгу вебсторінок, при якому весь процес відбувається на стороні клієнта, тобто в браузері користувача. Кроки створення сторінки зображені на малюнку нижче (див. рис. 1.1). В такому популярному ресурсі, як Rendering Pattern [15], написано, що основною

особливістю CSR є те, що браузер отримує мінімальний HTML-код та порожній контент при завантаженні сторінки, і весь вміст формується динамічно за допомогою JavaScript.

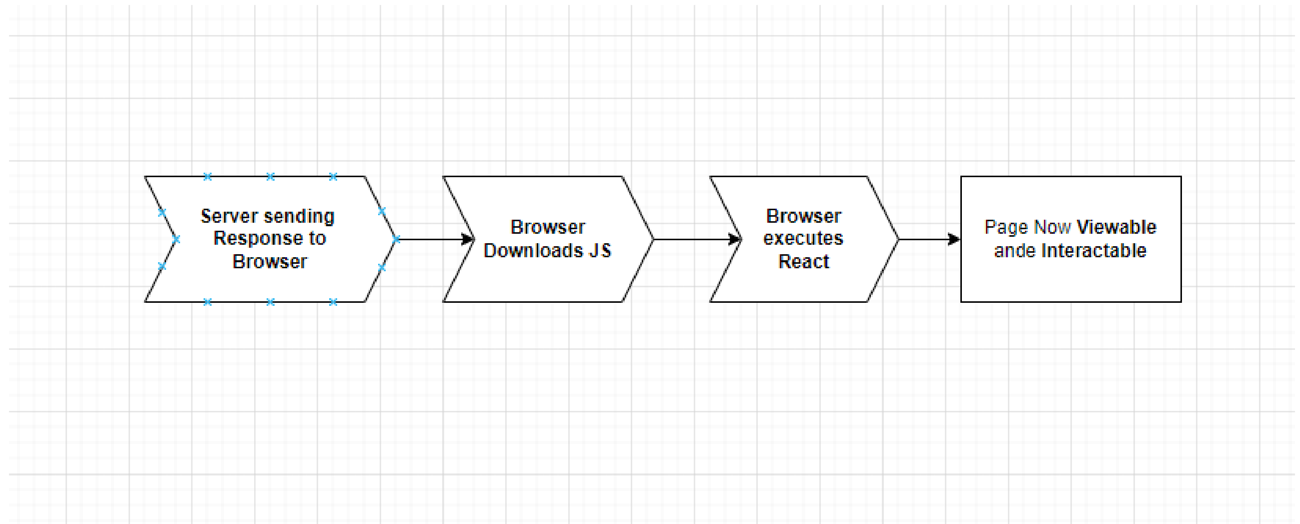

Рисунок 1.1 – Client-Side Rendering (CSR)

Переваги CSR:

– інтерактивність: CSR надає більшу інтерактивність вебсторінок, оскільки весь контент формується на боці клієнта, що дозволяє швидше реагувати на користувацькі події;
– легкість розробки: розробка за допомогою CSR може бути більш простою, оскільки розробникам не потрібно докладати зусиль для налаштування серверного рендерінгу;
– спрощення серверної інфраструктури: потреба в серверному рендерінгу зменшується, що може полегшити обслуговування серверної інфраструктури.

Недоліки CSR:

– SEO-неоптимальність: через те, що ініціюючий HTML-код при CSR містить мінімальну інформацію, це може впливати на SEO-оптимізацію сторінок, оскільки пошукові системи можуть мати обмежений доступ до контенту;
– помітна затримка: користувач може відзначити помітну затримку в

завантаженні сторінки, особливо при великому обсязі даних, оскільки контент формується динамічно.

CSR є популярним вибором для односторінкових додатків (Single Page Applications – SPA), де важлива швидкодія та висока реактивність інтерфейсу. Однак вирішення про використання CSR повинно базуватися на конкретних вимогах та характеристиках проєкту.

1.2 Server-Side Rendering (SSR)

Server-Side Rendering (SSR) [2] представляє собою метод рендерінгу вебсторінок, при якому весь процес відбувається на сервері перед тим, як контент буде відправлений на клієнтський браузер. Послідовність рендерінгу вебсторінок зображені на малюнку нижче (див. рис. 1.2). Основна ідея полягає в тому, що сервер генерує HTML-код із вмістом сторінки, включаючи дані та стилі, і надсилає цей готовий HTML-код на браузер. В контексті React інструментом є бібліотека Next.js [3].

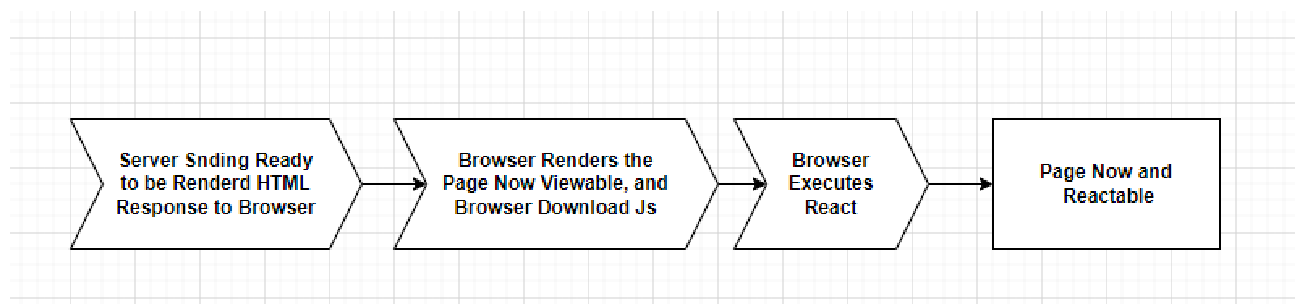

Рисунок 1.2 – Server-Side Rendering (SSR)

Переваги SSR:

- SEO-оптимізація: однією з ключових переваг SSR є його позитивний вплив на SEO (пошукові системи отримують повний та готовий HTML-код, що поліпшує індексацію сторінок);
- зменшення часу завантаження: користувач отримує повний вміст сторінки

вже при її завантаженні, що може зменшити час завантаження та покращити загальний користувацький досвід;

– кешування сторінок: велику роль грає можливість кешування готових HTML-сторінок, що сприяє подальшому прискоренню завантаження.

Недоліки SSR:

– збільшена навантаженість сервера: виконання рендерінгу на сервері може створити додаткове навантаження на сервер, особливо при великому обсязі запитань;
– обмежена інтерактивність: оскільки перший варіант HTML коду відправляється на браузер, подальші зміни або оновлення вимагають додаткових запитів до сервера, що може зменшити інтерактивність додатка.

SSR є часто використовуваним вибором для вебдодатків, які залежать від високої інтерактивності та SEO оптимізації, але варто враховувати його обмеження у випадку великої кількості динамічного контенту.

1.3 Static Site Generation (SSG)

Static Site Generation (SSG) є методом рендерінгу вебсторінок, при якому сторінки генеруються попередньо, на етапі розробки або при зміні контенту, та зберігаються у вигляді статичних HTML файлів. Послідовність рендерінгу вебсторінок зображені на малюнку нижче (див. рис. 1.3). Основна ідея полягає в тому, щоб вже на етапі збірки (build time) мати готові HTML файли для кожної сторінки, які потім можуть бути розділені клієнтам.

Переваги SSG:

– швидкодія: статичні файли можуть бути безпосередньо доставлені до клієнтів без додаткового оброблення на сервері, що робить їх надзвичайно швидкими у вигляді завантаження сторінок;
– безпека: оскільки немає необхідності виконувати код на сервері чи клієнті,

ризик потенційних атак зменшується, що підвищує безпеку додатка;

– простота хостингу: статичні файли можуть бути легко розгорнуті на різних платформах для хостингу, що полегшує процес розгортання.

Недоліки SSG:

– обмежена динамічність: динамічний контент, який змінюється в реальному часі, може виявитися складним для реалізації за допомогою SSG;

– складність управління великим обсягом даних: у великих проєктах може бути важко ефективно управляти та підтримувати великий обсяг статичних файлів.

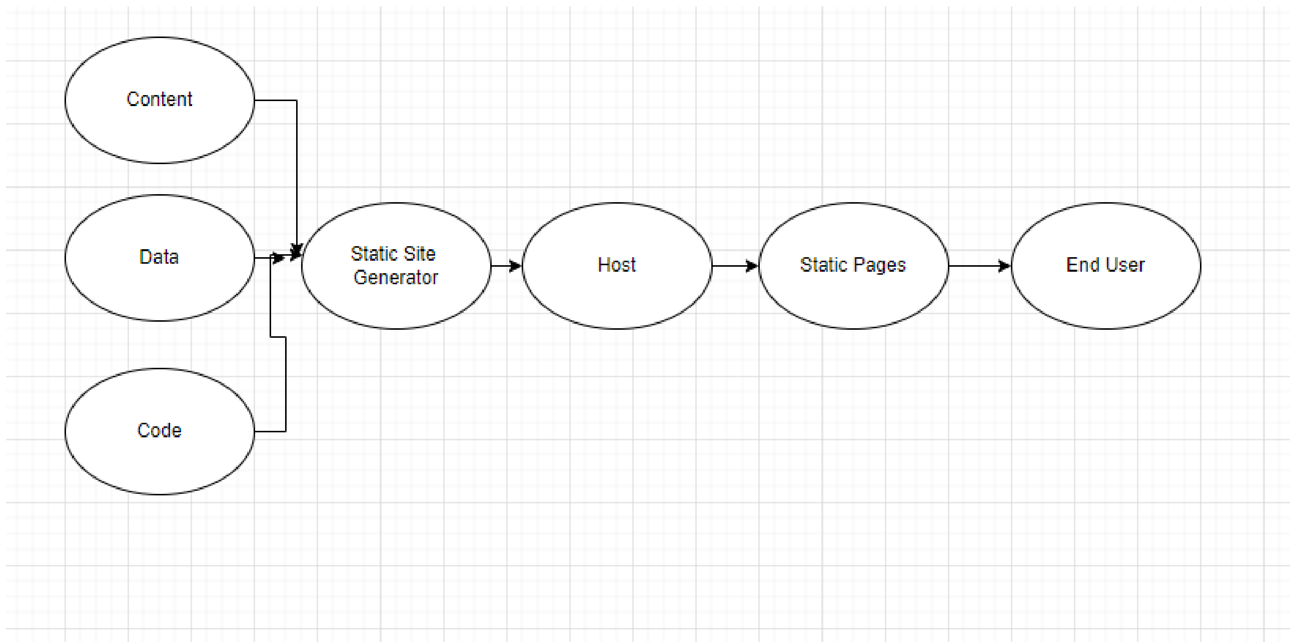

Рисунок 1.3 – Static Site Generation (SSG)

SSG добре підходить для вебсайтів, де контент майже статичний або рідко змінюється. Для проєктів, де частіше відбуваються зміни, можуть бути використані гібридні рішення, де деякі частини вебсайту генеруються статично, а інші використовують CSR або SSR. Вивчення SSG дозволить зрозуміти його сферу застосування та визначити його переваги та обмеження у порівнянні з іншими стратегіями рендерінгу.

1.4 Аналітика рішень на ринку

Аналіз ринку рішень у сфері веброзробки для вибору типу рендерінгу є ключовим етапом дослідження. На ринку існує ряд популярних фреймворків та інструментів, які підтримують різні підходи до рендерінгу, такі як CSR, SSR, та SSG. Аналіз цих рішень дозволяє визначити переваги та недоліки кожного, а також вибрати оптимальний інструмент для конкретного проєкту.

До фреймворків та бібліотек CSR відносяться такі популярні до даним аналітики які представив Alex Ivanovs в статті “The Most Popular Front-end Frameworks” [18] фреймворки, як React, Angular, та Vue.js.

Розглянемо фреймворки та бібліотеки, що відносяться до SSR. Аналізуються фреймворки, такі як Next.js (для React), Nuxt.js (для Vue.js), та Angular Universal (для Angular), які спеціалізуються на SSR. Вивчаються їхні можливості у покращенні SEO та зменшенні часу завантаження сторінок.

Генератори статичних сайтів розглядають інструменти, такі як Gatsby, Hugo, та Jekyll. Вони надають можливості створення статичних сайтів з великим обсягом контенту та додатковою можливістю використання динамічного рендерінгу.

На початку роботи над сайтом, на основі проведеного аналізу формулюються рекомендації щодо вибору оптимального типу рендерінгу для конкретного проєкту враховуючи його вимоги, потреби та особливості. Переваги та недоліки кожного підходу розглядаються в контексті конкретної задачі та цілей веброзробки.

Приклад проєкту з використанням CSR (Client-Side Rendering): онлайн-магазин зі складним інтерфейсом та багатою анімацією. Онлайн-магазин, де користувач може швидко фільтрувати товари, додавати їх у кошик, та переглядати рекомендації на основі попередніх виборів. Такий проєкт вимагає високої інтерактивності, і CSR дозволяє забезпечити швидку відповідь на дії користувача без необхідності перезавантаження сторінки.

Приклад проєкту з використанням SSR (Server-Side Rendering): новинний

портал з акцентом на SEO оптимізацію. Новинний портал, де важлива якість індексації контенту для пошукових систем. SSR генерує повний HTML код на сервері, що позитивно впливає на SEO-оптимізацію та забезпечує користувачам швидкий доступ до актуальної інформації.

Приклад проєкту з використанням SSG (Static Site Generation): особистий блог чи корпоративний сайт-візитка. Сайт, який має статичний характер та не часто змінюється. Наприклад, особистий блог або сайт-візитка для компанії. SSG дозволяє попередньо згенерувати сторінки, зменшити навантаження на сервер і забезпечити швидкий час завантаження сторінок для користувачів.

Ці приклади відображають, як вибір конкретного типу рендерінгу може залежати від конкретних потреб та характеристик проєкту, дозволяючи досягти оптимальної продуктивності та відповідати специфікації вимог веброзробки.

2 ПРИНЦИПИ ВИКОРИСТАННЯ ТА ЗАДАЧІ REACT

Основною проблемою WEB є оновлення DOM дерева і одним із його популярних рішень є Віртуальний DOM.

Якщо є необхідним зробити 500 оновлень на сторінці, не використовуючи технологію Virtual DOM, буде зроблено 500 оновлень і це не дуже подобається користувачу, бо очам людини важко сприйняти багато оновлень одночасно і це займає багато ресурсів. Тому, для оптимізації React використовує Virtual DOM, який оновиться 500 раз, а в реальному домі відобразиться всього 60 оновлень кадку в секунду, що буде краще сприймається оком і займе менше ресурсів.

React використовує та покращує примнив Virtual DOM в контексті своєї бібліотеки.

Оновлення React Virtual DOM є необхідним у зв'язку з різноманітними подіями та змінами в стані, які виникають під час взаємодії користувача або отримання нових даних.

2.1 Проблеми, які вирішує Virtual DOM (Застосування Virtual DOM)

Virtual DOM [4] (віртуальний DOM) є концепцією, що використовується в бібліотеках та фреймворках для реактивного оновлення інтерфейсу користувача. Основні проблеми, які вирішує Virtual DOM, включають (До основних проблем, які вирішує Virtual DOM, відносяться):

– ефективність оновлення;
– швидкість оновлення;
– кроссбраузерність.

Основною проблемою в ефективності оновлення є безпосередньо оновлення реального DOM для кожного змінного елемента виникає значна кількість операцій з перерисовкою, навіть для невеликої кількості оновлень.

Рішенням цього є те, що Virtual DOM дозволяє React обчислити та зберегти зміни у віртуальному дереві. Потім відбувається порівняння нового та попереднього віртуального DOM, і лише фактично змінені частини переносяться на реальний DOM.

Проблемою зі швидкість оновлення було те, що при частих оновленнях реального DOM може виникнути «флікеринг» або відчуття мерехтіння сторінки через багатошвидкісні маніпуляції з вмістом.

Рішенням цього також став Virtual DOM, який дозволяє React збирати групи оновлень та застосовувати їх одним махом, зменшуючи кількість змін та покращуючи загальний візуальний досвід.

Проблемою з кросбраузерністю стало те, що різні браузери можуть володіти різною інтерпретацією та реалізацією операцій з реальним DOM.

Рішенням став Virtual DOM який дозволяє React абстрагувати роботу з реальним DOM через єдиний інтерфейс. Це спрощує підтримку кроссбраузерності та зменшує ймовірність виникнення різних проблем на різних браузерах.

2.2 Принципи роботи React

За суттю React є бібліотекою JavaScript, що використовується для розробки інтерфейсів користувача. Вона дозволяє створювати вебдодатки, які мають динамічний інтерфейс і забезпечують ефективне управління станом додатку. До основних концепцій React відносяться: компоненти, віртуальний DOM [6], управління станом, JSX, реактивні властивості (Props), життєвий цикл компонентів.

Функціональні компоненти є функції, які приймають вхідні дані (props) і повертають React-елементи, які визначають структуру та вигляд компонента. Класові компоненти є класи, які розширюють базовий клас React.Component. Вони мають власний внутрішній стан і методи життєвого циклу.

React використовує віртуальний DOM для оптимізації оновлення інтерфейсу. Замість того, щоб безпосередньо змінювати реальний DOM кожного разу, коли змінюється стан компонента, React спершу вносить зміни до віртуального DOM. Потім він порівнює віртуальний DOM з реальним і вносить лише необхідні зміни, щоб ефективно оновити інтерфейс.

Стан – це дані, які визначають, як виглядає інтерфейс і як взаємодіє з користувачем додаток. React дозволяє компонентам мати внутрішній стан, який може бути змінений за допомогою методу setState(). При зміні стану React автоматично перерендерює компонент, оновлюючи відображення відповідно до нового стану.

React надає синтаксис JSX. JSX – це розширення синтаксису JavaScript, яке дозволяє писати код, який виглядає подібно HTML (див. рис. 2.1).

```
const element = <h1>Hello, World!</h1>;
```

Рисунок 2.1 – Результат HTML елементу

Реактивні властивості (Props) представляє собою можливість компонентів отримувати дані від своїх батьків через властивість props. Це дозволяє передавати дані вниз по ієрархії компонентів.

Життєвий цикл у класових компонентах залишається з методами, такими як componentDidMount, componentDidUpdate, і componentWillUnmount. У функціональних компонентах використовуються хуки, такі як useEffect, для вирішення аналогічних завдань.

React почав активно підтримувати функціональні компоненти з введенням хуків у версії React 16.8. Ця версія була випущена в лютому 2019 року. Хуки, такі як useState і useEffect, надають можливість використовувати стан і життєвий цикл в функціональних компонентах. Завдяки цим змінам, функціональні компоненти стали більш потужними і зручними для використання, та їх використання стало стандартом у розробці React-додатків.

2.3 Сценарій оновлення

Сценарій оновлення дерева компонентів в React [5] складається з п'яти кроків, починаючи з створення та закінчюю останнім оновленням дерева.

Першим кроком є складання дерева з вузлів, в якому кожен вузол є node-об'єктом з інформацією про компонент. Таке дерево має назву "Current tree" (див. рис. 2.2).

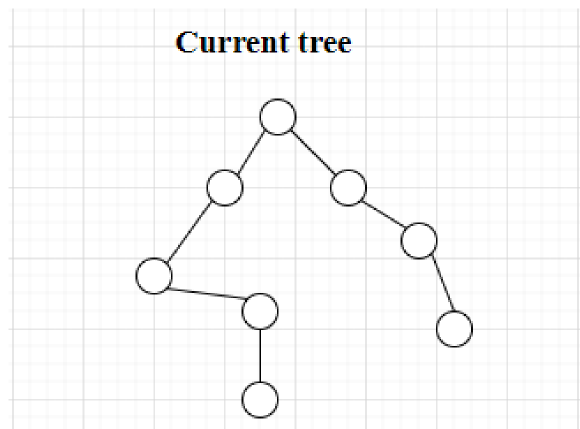

Рисунок 2.2 – Current tree

На другому кроці React передаємо Current tree до Rendering environment в якому воно перетворюється в набір DOM операцій які пререотизуються. Для кращого розуміння Rendering environment зображено на малюнку нижче (див. рис. 2.3).

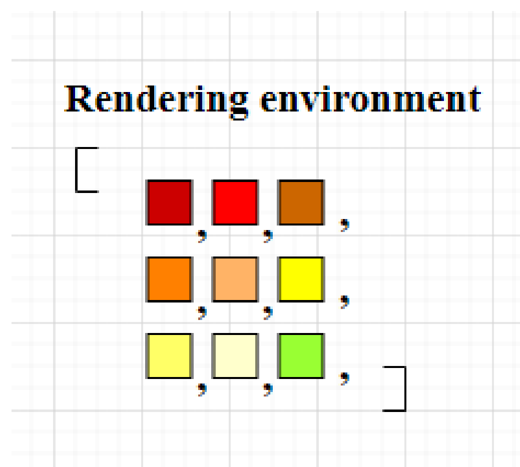

Рисунок 2.3 – Rendering environment

Третім кроком є виконання всіх операцій в залежності від пріоритету. Після виконання всіх операцій результатом вже буде сторінка в браузері.

Четвертій крок вже є роботою React після якихось дій на сторінці (наприклад: натискання на кнопку або ввід тексту) створюється ще одне дерево Work-In-Progress tree (див. рис. 2.4) зі всіма змінами.

На п’ятому кроці починаєть операція порівняння двух дерев Current tree та Work-In-Progress tree. На цьому етапі знаходяться всі зміни, які передаються до Rendering environment і знову виконуються дкурій на третій кроки. Тому Work-In-Progress tree стає Current tree і це перетворюється в цикл доки користувач не перестане працювати з сторінкою.

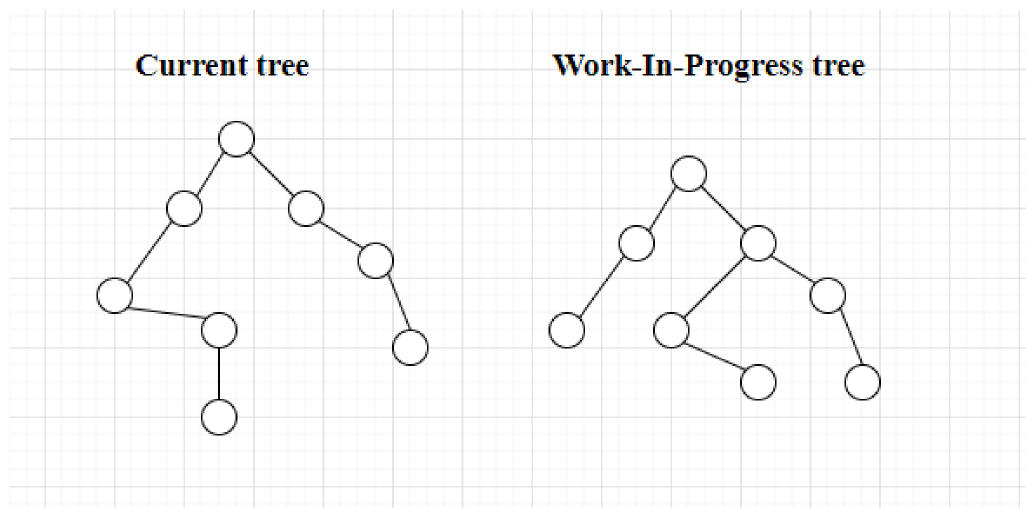

Рисунок 2.4 – Work-In-Progress tree

2.4 Алгоритм порівняння дерев

React надає декларативний API, тому не доведеться турбуватися про те, що саме змінюється під час кожного оновлення. Це значно спрощує написання додатків, але може бути не очевидно, як це реалізовано в React.

При використанні React, на перший час render() здається функцію яка створеннє дерева елементів React. При наступному оновленні стану ця render() функція поверне інше дерево елементів React. Потім React необхідно з’ясувати,

як ефективно оновити користувацький інтерфейс, щоб сторінка відповідала найостаннішому дереву.

Існує кілька загальних розв’язків цієї алгоритмічної, проблеми створення мінімальної кількості операцій для перетворення одного дерева в інше. Однак сучасні алгоритми, за даними опитування які представленні в роботі Філіпа Білле представника Копенгагенський університет інформаційних технологій [16] мають складність порядку $O(n^3)$, де n – кількість елементів у дереві.

Розробники React в свой документації [17] кажуть якби в React використовували сучасні алгоритми, для відображення 1000 елементів знадобилося б близько одного мільярда порівнянь. Це занадто дорого. Замість цього React реалізує евристичний алгоритм зі складністю O(n).

Heuristic $O(n)$ алгоритм – це алгоритм вирішення задачі правильність рішення якої для всіх можливих випадків не доведена, але про який відомо що він дає доволі правильне рішення в більшості випадках. мається на увазі, що React залишає допущення для того щоб заощадити час.

Допущеннями е поява нового типу та нового ключа.

Новий тип – нове дерево. Коли йде порівняння дерев і алгоритм знаходить різницю в типах, дерево видаляється та створюється нове. Наприклад: в блоці div був ваш компонент і ви змінили div на section.

Новий key – нове дерево. Це допущення більше схоже на інструмент який додає ключ до компонента і запобігають його видаленню з дерева, навіть якщо тип компоненту перед ним змінився. Але не випадку якщо ви змінили div на section.

Щоразу, коли кореневі елементи мають різні типи, React знищує старе дерево і будує нове з нуля. Переходи від <a> до <img> або від <Article> до <Comment> або від <Button> до <div> призведуть до повної перебудови. При знищенні дерева старі DOM-вузли видаляються. Примірники компонента одержують componentWillUnmount(). При побудові нового дерева нові DOM-вузли вставляються в DOM. Примірники компонента отримують componentWillMount(), а потім componentDidMount(). Будь-який стан,

пов’язаний зі старим деревом, втрачається. Будь-які компоненти, що лежать нижче кореневого, також розмонтуються, а їхній стан знищиться. Translated with DeepL.com (free version). При цьому старий Counter знищиться, а новий змонтується.

Коли компонент оновлюється, його екземпляр залишається тим самим, тому його стан зберігається між рендерами. React оновлює пропси базового екземпляра компонента для відповідності новому елементу та викликає componentWillReceiveProps() та componentWillUpdate() на базовому екземплярі.

Далі викликається метод render() та алгоритм порівняння рекурсивно оминає попередній та новий результати.

За умовчанням при рекурсивному обході дочірніх елементів DOM-вузла React проходить по обидва списки нащадків одночасно і створює мутацію, коли знаходить відмінність.

Коли відсутнє унікальне значення, ви можете додати нову властивість ідентифікатора у вашу модель або прохешувати дані, щоб згенерувати ключ. Ключ має бути унікальним лише серед його сусідів, а не глобально.

У крайньому випадку ви можете передати індекс елемента масиву як ключ. Це добре працює у випадку, якщо елементи ніколи не змінюють порядок. Перестановки елементів викликають уповільнення.

При використанні ключів перестановки можуть викликати проблеми зі станом компонента. Примірники компонента оновлюються та повторно використовуються на основі їх ключів. Переміщення елемента змінює його, якщо ключ є індексом. В результаті стан компонента для таких речей, як некеровані <input>, може змішатися та оновитися несподіваним чином.

Важливо пам’ятати, що алгоритм узгодження це деталь реалізації. React може повторно рендерувати весь додаток на кожну дію, кінцевий результат буде тим самим. Для ясності повторний рендер у цьому контексті означає виклик функції render для всіх компонентів, але це не означає, що React розмонтує та змонтує їх заново. Він застосує відмінності тільки дотримуючись правил, які були позначені в попередніх розділах.

React регулярно вдосконалюємо евристику, щоб прискорити варіанти використання, що часто зустрічаються. У поточній реалізації ви можете висловити факт того, що піддерево зрушило серед його сусідів, але ви не можете сказати, що воно зрушило кудись в інше місце. Алгоритм повторно відрендерить все піддерево.

React покладається на евристику, отже якщо припущення, на яких вона заснована, не дотримані, постраждає продуктивність.

Алгоритм не намагатиметься зіставити піддерева компонентів різних типів. Якщо ви помітите за собою, що намагаєтесь чергувати компоненти різних типів з дуже схожим висновком, то бажано зробити їх компонентами одного типу. На практиці ми не виявили із цим проблем.

Ключі мають бути стабільними, передбачуваними та унікальними. Нестабільні ключі (наприклад, зроблені за допомогою Math.random()) викликають необов’язкове перестворення багатьох екземплярів компонента та DOM-вузлів, що може спричинити погіршення продуктивності та втрату стану у дочірніх компонентів.

2.5 Пріоритизація

Коли треба працювати з великою кількістю задач як React повинен зрозуміти що важливіше і не робити всі задачі одночасно, що це не перетворилось у тремтячу сторінку. Коли React отримує якусь кількість змін (batching) від розбиває цей батчь на малі операції та пріоритизує. У React є алгоритм пріоритизації цих операцій і цей алгоритм вирішує що важливіше, відкладає відрисовку менш важливих задач на потім та непомітно відривовує в кінці, мається на увазі що браузер отримує ці операції порціями.

В ідеалі, кожні 16мс підтягувати новий кадр, але якщо треба обробити кадр на 240 мс, React бере цей кадр і розбиває його на малі операції, незалежні один від одного та пршоритизує. Для кращого уявлення цей процес зображено на малюнку нижче (див. рис. 2.5).

Рисунок 2.5 – Кількість кадрів в секунду

В основі пріоритизації використовується лише дві функції: window.requestAnimationFrame та window.requestIdleCallback.

Window.requestAnimationFrame(Callback) – на кожний кадр React викликає цю функцію і завдяки Callback функції розуміє що вже відмальовано і по пріоритету віддає нові кадри.

Window.requestIdleCallback(callback[, options]) – після того як відмалювались більш важливі кадри, це метод починає відмальовувати менш пріоритетні кадри(наприклад ті елементи які ще не бачить користувач).

3 ВАРІАНТИ ВИРІШЕННЯ ПРОБЛЕМ ТА ОПТИМІЗАЦІЯ

Знання про оптимізацію мають стратегічне значення для розробників, допомагаючи їм ефективніше вирішувати виклики оптимізації та гарантувати високу ефективність їхніх проєктів. Завдяки розгляду та застосуванню різноманітних методів, розробники можуть не лише досягти великого рівня продуктивності своїх додатків, але і забезпечити високий рівень користувацького задоволення.

3.1 Покращення продуктивності при використанні реекспорту

Реекспорт багатьох компонентів з одного файлу є популярним шаблон, якого дотримуються розробники для організації своєї кодової бази. Це дозволяє легко імпортувати ці компоненти в інші файли [7].

Приклад реекспорту представлений на малюнку нижче (див. рис. 3.1). У прикладі файл index.js у папці "UI" імпортує, а потім експортує всі компоненти з каталогу 'UI'. Тепер при імпорти компонентів безпосередньо з цього UI/index.js файлу, до вашого файла з компоненот будуть імпортовані всі компоненти з UI/index.js.

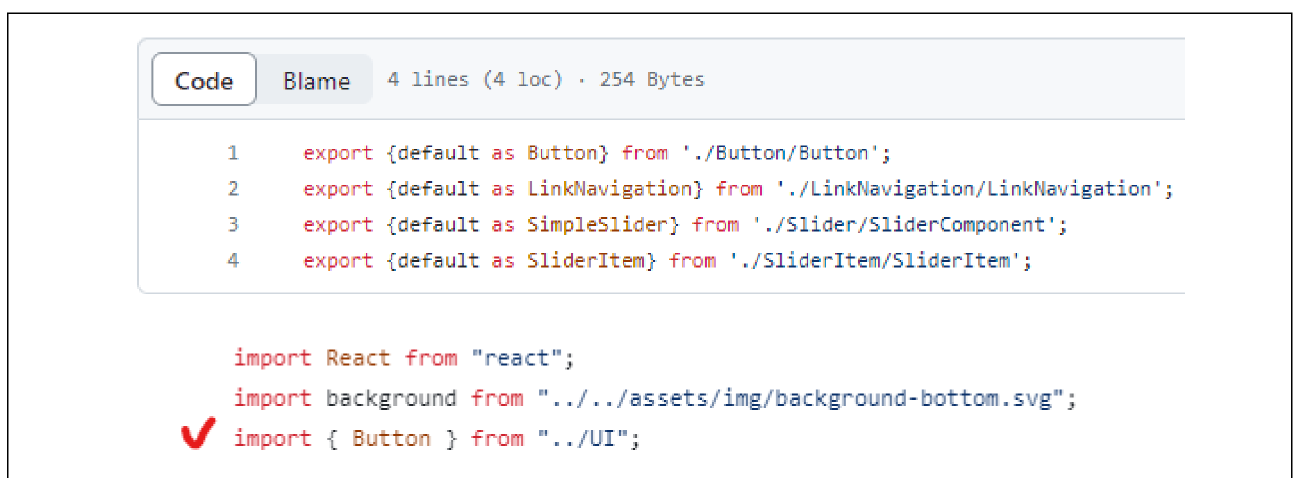

Рисунок 3.1 – Приклад реекспорту

Реекспорт погано впливає на продуктивність сайту, оскільки навіть якщо імпортується один компонент, всі інші невикористані компоненти додаються до цього фрагмента.

Tree shaking – це термін, який зазвичай використовується в контексті JavaScript для усунення «мертвого» коду. Він ґрунтується на статичній структурі синтаксису модулів ES2015, тобто імпорті та експорті. Назва та концепція були популяризовані завдяки згортанню пакунків модулів ES2015.

React збирається за допомогою Webpack і використовує Tree shaking. Але воно видаляє лише ті компоненти, які не використовуються у всьому додатку.

Було знайдено проблему при реекспорті Tree shaking не працює і бандл має багато коду який не використовується.

Оголошення експорту[A] використовується для експорту значень з модуля JavaScript. Експортовані значення можна імпортувати в інші програми за допомогою оголошення імпорту або динамічного імпорту. Значення імпортованого зв'язування може бути змінено в модулі, який його експортує – коли модуль оновлює значення зв'язування, яке він експортує, оновлення буде видно в імпортованому значенні.

Щоб використовувати декларацію експорту у вихідному файлі, файл має бути інтерпретований середовищем виконання як модуль. У HTML це робиться шляхом додавання type="module" до тегу <script> або шляхом імпорту іншим модулем. Модулі автоматично інтерпретуються в строгому режимі.

Ідентифікатор для експорту (щоб його можна було імпортувати за допомогою імпорту в іншому скрипті). Якщо використовується псевдонім з as, фактичне експортоване ім'я можна вказати як рядковий літерал, який може не бути дійсним ідентифікатором.

Кожен модуль може мати два різних типи експорту: іменований експорт та експорт за замовчуванням. Ви можете мати декілька іменованих експортів для одного модуля, але лише один експорт за замовчуванням.

Першим рішенням, але не дуже хорошим рішенням було уникнути її (див. рис. 3.2). Видаливши цей файл реекспорту і використовуйте абсолютний

імпорт(іменований) для кожного компонента. Це було просте рішення, але воно збільшувало довжину операторів імпорту і було не дуже масштабованим.

Другим рішенням було перетворення оператора імпорту під час компіляції за допомогою babel-plugin-transform-imports.

```
import { Button } from "../../UI/Button/Button"; ✔
import { ServicesTerms } from "../ServicesTerms/ServicesTerms";
import background from "../../assets/img/background-bottom.svg";
import { Button } from "../UI";✘
import bartender from "../../assets/img/home/bl-bartender.jpg";
```

Рисунок 3.2 – Перше рішення зайвого реекспорту

Це було хорошим рішенням для простого імпорту. Але як тільки файл реекспорту стає складнішим, плагін babel-plugin-transform-imports стає складним у підтримці.

3.2 Профілювання додатку React для виявлення вузьких місць

React дозволяє вимірювати продуктивність додатків за допомогою профайлера в React Developer Tools. Такоє можемо збирати інформацію про продуктивність кожного разу, коли додаток рендериться. Профайлер записує, скільки часу потрібно компоненту для рендерингу, чому компонент рендериться тощо. Звідси можливо дослідити відповідний компонент і забезпечити необхідну оптимізацію.

Щоб використовувати профайлер, треба встановити React DevTools для браузера. Якщо його не встановлено, треба перети на сторінку розширення та встановіть його. Тепер, працюючи над React-проєктом, маємо можливість побачити вкладку профайлера.

Спочатку треба увімкнути функцію для відображення рендеру, як на малюнку нижче (див. рис. 3.3). Під час профілювання додатку, відображається поведінка рендеру компонентів як на рисунку (див. рис. 3.4).

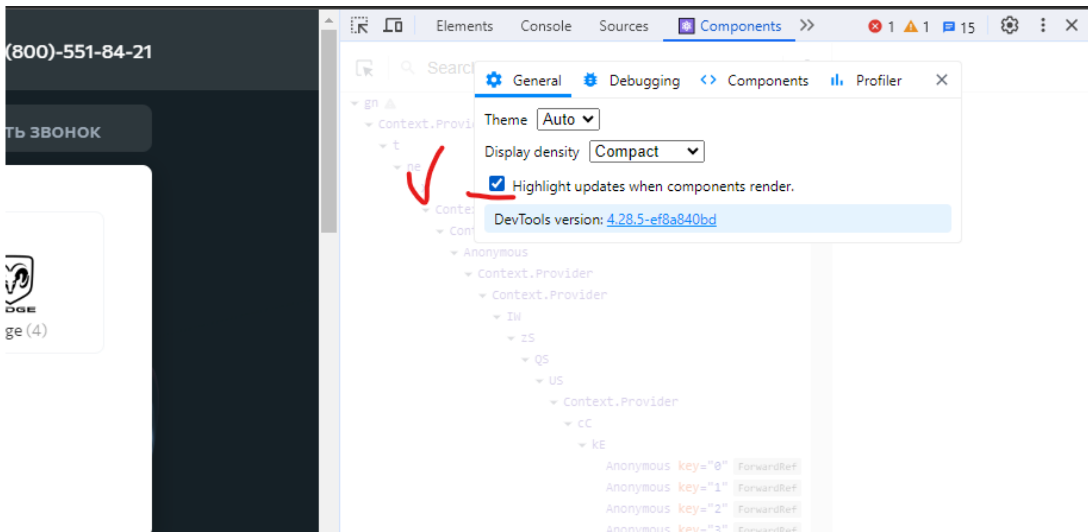

Рисунок 3.3 – Увімкнути функцію для відображення рендеру

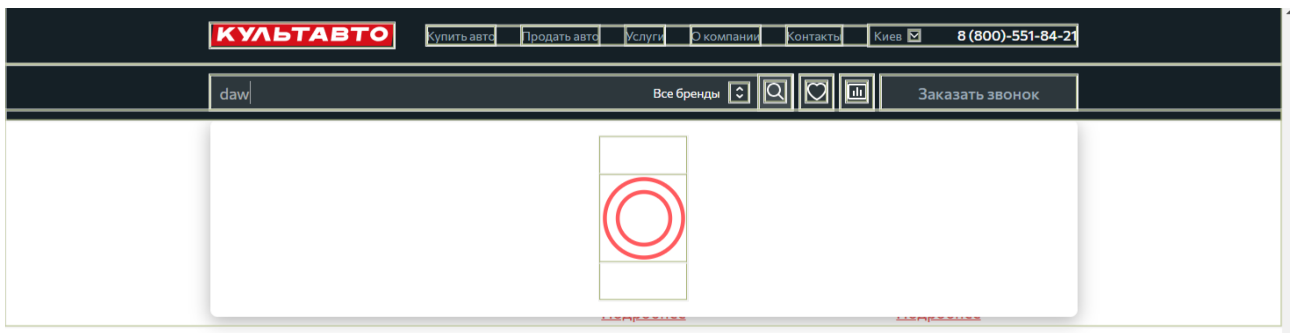

Рисунок 3.4 – Відображення рендеру

Профайлер React DevTools підсвічує кожен відрендерений компонент, поки оновлюється текстове поле вводу, і відображається кожна деталь з відрендерених компонентів. На діаграмі нижче зображено, скільки часу зайняв рендеринг компонентів і чому рендериться компонент App (див. рис. 3.5).

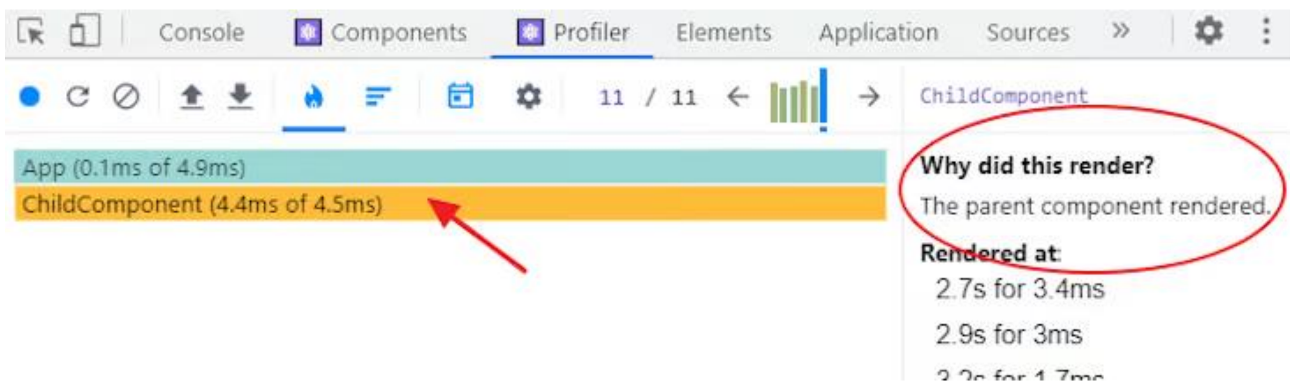

Рисунок 3.5 – Діаграма рендеру

3.3 Зберігати стан компонентів на місцевому рівні, де це необхідно

Оновлення стану в батьківському компоненті призводить до повторного рендерингу батьківського та дочірніх компонентів.

Для того, щоб перерендеринг компонента відбувався тільки тоді, коли це необхідно, треба виділити частину коду, яка відповідає за стан компонента, і зробити його локальним для цієї частини коду. На прикладі сторінки Каталогу можемо побачити компомпонент CarsSection (див. рис. 3.6), який не залежить від батьківського стану [Б].

```
1   import React from "react";
2   import FilterComponent from "../../../components/Main/CarFilter/FilterComponent/FilterComponent";
3   import style from "./Catalog.module.scss";
4   import { Breadcrumbs } from "../../../components/Breadcrumbs/Breadcrumbs";
5   import { CatalogContactForm } from "../../../components/Catalog/CatalogContactForm/CatalogContactForm";
6   import { About } from "../../../components/Main/About/About/About";
7   import { CarsSection } from "../../../components/Catalog/CarsSection/CarsSection";
8   import { MainWrapper } from "../../../components/UI/MainWrapper/MainWrapper";
9   function Catalog() {
10    return (
11      <MainWrapper>
12        <div className={style["catalog-filter"]}>
13          <FilterComponent />
14        </div>
15        <section className={style.catalog} id="catalog">
16          <div className={style.catalog__inner}>
17            <Breadcrumbs />
18            <CarsSection />
19          </div>
20        </section>
21        <CatalogContactForm />
22        <section className={style.catalog}>
23          <About />
24          <div className={style.catalog__inner}></div>
25        </section>
26      </MainWrapper>
27    );
28  }
29
30  export { Catalog };
```

Рисунок 3.6 – Батьківськiй компонент

Це гарантує, що рендериться лише той компонент, який дбає про стан. У нашому коді тільки поле вводу піклується про стан. Отже, якщо перенести стан з батьківського компонента до компонента CarsSection, зробивши його дочірнім компонентом(ChildComponent).

Але іноді майже неможливо уникнути наявності стану в батьківському компоненті і із-за бізнес логікою треба передавати його дочірнім компонентам як проп. У такому випадку, теж можна уникнути повторного рендерингу незчеплених дочірніх компонентів використовуючи інструменти для запам'ятовування React-компонентів.

3.4 Запам'ятовування React-компонентів, щоб уникнути непотрібних повторних рендерів

На відміну від попереднього методу підвищення продуктивності, де рефакторинг коду дає приріст продуктивності, в цьому випадку обмінюється простір пам'яті на час. Отже, потрібно запам'ятовувати компонент лише тоді, коли це необхідно [8].

Запам'ятовування(Memoization) – це стратегія оптимізації, яка кешує операцію рендерингу компонента, зберігає результат у пам'яті та повертає кешований результат для тих самих вхідних даних. По суті, якщо дочірній компонент отримує проп, запам'ятований компонент за замовчуванням поверхнево порівнює проп і пропускає повторний рендеринг дочірнього компонента, якщо проп не змінився.

React.memo [9] – це компонент вищого порядку, який використовується для обгортання суто функціонального компонента, щоб запобігти повторному рендерингу, якщо пропси, отримані в цьому компоненті, ніколи не змінюються. На прикладі компонента Select можемо побачити його синтаксис[В].

React.memo() добре працює, коли отримує передаємо примітивні значення, такі як число. Э таке поняття як посилальна рівность. У світі JavaScript можна

порівнювати значення, використовуючи примітивні типи, як рядки або числа, або порівнювати посилання під час роботи з об’єктами. Порівняння за вартістю каже про те, що, якщо два значення рівні, то логічне порівняння повертає true. Це працює для найпоширеніших примітивних типів JavaScript (рядки, числа та логічні значення). Порівняння за посиланням э типом порівняння який вираховує, де в пам’яті розташований об’єкт. Якщо два об’єкти вказують на одне й те саме місце, вони рівні, в іншому разі вони різні. Навіть якщо два об’єкти мають однакові властивості з однаковими значеннями, вони не будуть рівними, якщо тільки не знаходяться в одній і тій самій позиції пам’яті.

Примітивні значення завжди є посилально рівними і повертають true, якщо значення ніколи не змінюються.

З іншого боку, не примітивні значення, такі як об’єкт (object), до яких відносяться масиви і функції, завжди повертають false між повторними рендерингами, оскільки вони вказують на різні ділянки в пам’яті. Коли передається об’єкт, масив або функцію як проп, запам’ятовуваний компонент завжди повторно рендериться.

Цей код фокусується також на функції, які передається до нашого компонента. Коли батьківський компонент перевизначається, навіть якщо кнопка не натиснута, функція перевизначається, що призводить до перевизначення компонента.

Щоб запобігти постійному перевизначення функції, використається хук useCallback, який повертає запам’ятовувану версію зворотного виклику між рендерингами.

З хуком useCallback функція incrementCount перевизначається тільки тоді, коли змінюється масив залежностей count (див. рис. 3.7).

```
const incrementCount = React.useCallback(() => setCount(count + 1),
[count]);
```

Рисунок 3.7 – useCallback

Коли проп, який передається дочірньому компоненту, є масивом або об’єктом, можна використовувати хук useMemo для запам’ятовування значення між рендерингами. Ці значення вказують на різні ділянки в пам’яті і є абсолютно новими значеннями.

Також можна використовувати хук useMemo, щоб уникнути повторного обчислення одного і того ж дорогого значення в компоненті. Він дозволяє запам’ятовувати ці значення і повторно обчислювати їх лише у разі зміни залежностей. Подібно до useCallback, хук useMemo також очікує функцію і масив залежностей.

Давайте подивимось, як застосувати хук useMemo для покращення продуктивності React-додатку. Погляньте на наступний код, який ми навмисно затримали, щоб він працював дуже повільно (див. рис. 3.8).

```
import React, { useState } from "react";

const expensiveFunction = (count) => {
// artificial delay (expensive computation)
for (let i = 0; i < 1000000000; i++) {
} return count * 3;
};
export default function App() {
 // ...
const myCount = expensiveFunction(count);
 return (
<div> {/* ... */}
<h3>Count x 3: {myCount}</h3>
 <hr />
 <ChildComponent count={count} onClick={incrementCount} />
 </div> );
 }
const ChildComponent = React.memo(
function ChildComponent({ count, onClick }) {
 // …
 });
```

Рисунок 3.8 – useMemo

Після відтворення цей код, відчувається затримку в роботі додатку, коли вводиться текст у поле введення і коли натискається кнопка «Приріст». Це відбувається тому, що кожного разу, коли компонент App рендерить, він викликає expensiveFunction і уповільнює роботу додатку.

Функцію expensiveFunction слід викликати лише тоді, коли натискається кнопка Increment, а не тоді, коли вводиться текст у поле вводу. React може запам'ятати повернуте значення функції за допомогою хука useMemo, щоб вона повторно обчислювала функцію тільки тоді, коли це потрібно, наприклад, коли натискається кнопка Increment. Для цього у нас буде щось на кшталт того що е на рисунку нижче (див. рис. 3.9).

```
const myCount = React.useMemo(
        () => {
                return expensiveFunction(count);
        }, [count]);
```

Рисунок 3.9 – React.useMemo приклад

Тепер, якщо ще раз протестувати додаток, користувач більше не відчує затримок при введенні даних у полі введення.

3.5 Розділення коду в React за допомогою динамічного імпорту

Розбиття коду [10] – ще одна важлива техніка оптимізації для React-додатків. За замовчуванням, коли React-додаток рендериться у браузері, користувачеві завантажується і надається одразу весь код додатку. Цей файл генерується шляхом злиття всіх файлів коду, необхідних для роботи вебдодатку.

Ідея бандлінгу корисна, оскільки вона зменшує кількість HTTP-запитів, які може обробити сторінка. Однак зі зростанням додатку розміри файлів збільшуються, що призводить до збільшення пакувального файлу. У певний момент це безперервне збільшення файлу сповільнює початкове завантаження

сторінки, знижуючи задоволеність користувачів.

Завдяки розбиттю коду, React дозволяє нам розбити великий файл бандлу на декілька частин за допомогою динамічного імпорту() з подальшим лінивим завантаженням цих частин на вимогу за допомогою React.lazy. Ця стратегія значно покращує продуктивність сторінки складного React-додатку.

Щоб реалізувати розділення коду, ми трансформуємо звичайний React-імпорт (див. рис. 1.10).

```
import Home from "./components/Home";
import About from "./components/About";

const Home = React.lazy(() => import("./components/Home"));
const About = React.lazy(() => import("./components/About"));

<React.Suspense fallback={<p>Loading page...</p>}>
        <Route path="/" exact> <Home /> </Route>
         <Route path="/about"> <About /> </Route>
</React.Suspense>
```

Рисунок 3.10 – React.useMemo приклад

Suspense дозволяє нам відображати завантажувальний текст або індикатор як запасний варіант, поки React чекає на рендеринг лінивого компонента в інтерфейсі.

3.6 Lazy loading зображень у React

Якщо для прикладу взяти додаток, в якому рендериться кілька рядів елементів на сторінці. Незалежно від того, чи відображається якийсь з елементів у вікні браузера, вони відображаються в DOM і можуть вплинути на продуктивність нашого додатку.

За допомогою концепції вікон(Windowing) надає можливість відрендерити

в DOM лише видиму для користувача частину. Потім, при прокручуванні, решта елементів списку відображаються, замінюючи елементи, які виходять з області перегляду. Ця техніка може значно покращити продуктивність рендерингу великого списку. І react-window, і react-virtualized – дві популярні бібліотеки вікон, які можуть реалізувати цю концепцію.

Щоб оптимізувати додаток, який складається з декількох зображень, є можливість уникнути рендерингу всіх зображень одночасно, щоб зменшити час завантаження сторінки. Завдяки лінивому завантаженню [11] можна зробити затримку, поки кожне з зображень з'явиться у вікні перегляду, перш ніж відрендерити їх у DOM.

Подібно до концепції вікон, згаданої вище, ліниве завантаження зображень запобігає створенню непотрібних вузлів DOM, підвищуючи продуктивність нашого React-додатку. React-lazyload та react-lazy-load-image-component – популярні бібліотеки лінивого завантаження, які можна використовувати у React-проєктах.

Ось приклад лінивого завантаження за допомогою react-lazy-load-image-component (див. рис. 3.11).

```
import Home from "./components/Home";
import About from "./components/About";

const Home = React.lazy(() => import("./components/Home"));
const About = React.lazy(() => import("./components/About"));
<React.Suspense fallback={<p>Loading page...</p>}>
        <Route path="/" exact> <Home /> </Route>
         <Route path="/about"> <About /> </Route>
</React.Suspense>
```

Рисунок 3.11 – Приклад використання react-lazy-load-image-component

Але зробити це самому не важно і якщо бібліотека перестане підтримуватись, ви завжди будете впевнені що все працює. Вікорисповючи useObserver (див. рис. 3.12) я відтворив туж саму логіку роботи lazy-load-image (див. рис. 3.13).

```
import React, { useEffect, useRef, useState } from "react";

export function useObserver() {
  const [isView, onView] = useState(false);
  const indicatorRef = useRef();
  var intersectionObserver = new IntersectionObserver((entries) => {
    onView(entries[0].isIntersecting);
  });
  useEffect(() => {
    if (!indicatorRef) {
      return;
    }
    intersectionObserver.observe(indicatorRef.current);
  }, [indicatorRef]);

  return { indicatorRef, isView };
}
```

Рисунок 3.12 – хук useObserver

```
import React, { useEffect, useState } from "react";
import { useObserver } from "../../../hooks/useObserver";
function LazyImage({ className, imageAlt, imageSrc }) {
  const { indicatorRef, isView } = useObserver();
  const [dataSrc, setDataSrc] = useState(
    "https://cataas.com/cat?width=300&i=5"
  );
  useEffect(() => {
    setDataSrc(imageSrc);
  }, [isView]);

  return (
    <img
      ref={indicatorRef}
      className={className}
      src={dataSrc}
      data-src={"https://cataas.com/cat?width=300&i=5"}
      alt={imageAlt}
    />
  );
}
export { LazyImage };
```

Рисунок 3.13 – власній компонент використовуючи підхід lazy-load

3.7 Використання незмінних структур даних(immutable data structures)

Замість того, щоб безпосередньо вносити зміни в об’єкт, який містить складні дані, треба зробити копію об’єкта, який було оновлено з урахуванням змін.

В такому випадку можна легко порівняти посилання оригінального об’єкта і нового, щоб визначити зміни і запустити оновлення інтерфейсу [12]. Стан React слід розглядати як незмінний, і розробник ніколи не повинен безпосередньо мутувати його. Давайте подивимося приклад використання нижче (див. рис. 3.14), як це працює на практиці.

```
export default function App() {
  const [bookInfo, setBookInfo] = useState({
    name: "A Cool Book",
    noOfPages: 28
  });

  const updateBookInfo = () => {
   bookInfo.name = 'A New title'
  };
  return (
    <div className="App">
      <h2>Update the book's info</h2>
      <pre>
        {JSON.stringify(bookInfo)}
      </pre>
      <button onClick={updateBookInfo}>Update</button>
    </div>
  );
}
```

Рисунок 3.14 – Приклад використання immutable data structures

В прикладі нижче (див. рис. 3.15) зображено безпосередньо оновлення стану bookInfo у функції updateBookInfo. Це спричинить деякі проблеми з

продуктивністю, оскільки React не може відстежити цю зміну і оновити інтерфейс відповідно. Це можна виправити, розглядаючи стан bookInfo як незмінну структуру даних, замість того, щоб намагатися змінювати її безпосередньо.

```
const updateBookInfo = () => {
  const newBookInfo = { ...bookInfo };
  newBookInfo.name = "A Better Title";
  setBookInfo(newBookInfo);
};
```

Рисунок 3.15 – updateBookInfo

Замість того, щоб безпосередньо оновлювати стан в updateBookInfo, потрібно зробити копію bookInfo, оновлюючи її і передаючи це нове значення в setBookInfo. Таким чином, React може належним чином відстежувати будь-які зміни стану, що відбуваються, і відповідно оновлювати інтерфейс користувача. Налаштувати незмінність самостійно можна зробити самостійно, або скористатися бібліотеками, такими як Immer та Immutable.js.

3.8 Застосування web-workers у React

Використовуючи web-worker [13] для запуску скриптів паралельно з основним потоком програми. Це надає можливість виконувати довгі та інтенсивні процеси в окремому потоці, не сповільнюючи рендеринг інтерфейсу. Давайте подивимося, як це працює (див. рис. 3.16), змоделювавши заблокований інтерфейс.

У наведеному фрагменті коду створено простий додаток з двома станами, noOfApples і noOfTomatoes. Кількість noOfTomatoes збільшується, коли натискається кнопка. Однак noOfApples не може збільшуватися через цикл, який, у свою чергу, блокує інтерфейс користувача.

```
export default function App() {
  const [noOfTomatoes, setNoOfTomatoes] = useState(0);
  const [noOfApples, setNoOfApples] = useState(0);

  const addApples = () => {
    const start = Date.now();
    while (Date.now() < start + 5000) { }
    setNoOfApples(noOfApples + 1);
  }
  return (
    <main>
      <p> Tomato: {noOfTomatoes} | Apple: {noOfApples} </p>
      <div>
        <button onClick={() => setNoOfTomatoes(noOfTomatoes + 1)}>
          Tomato
        </button>
        <button onClick={() => addApples()}>Apple</button>
      </div>
    </main>
  );
}
```

Рисунок 3.16 – web-worker

Давайте виправимо ефект блокування за допомогою web-worker, який буде обробляти функціонал addApples (див. рис. 3.17).

```
self.onmessage = async ($event) => {
    if ($event && $event.data && $event.data.msg === 'increaseAppleCount') {
        const newCounter = addApples($event.data.noOfApples);
        self.postMessage(newCounter);
    }
};

function addApples(noOfApples) {
    const start = Date.now();
    while (Date.now() < start + 5000) {
    }
    return noOfApples + 1;
}
```

Рисунок 3.17 – web-worker

Onmessage є точкою входу для web-worker, і це слухач, який буде запускатись в додатку. Якщо є подія з деякими даними, яка містить відповідне msg – в даному випадку «increaseAppleCount» – викликає функцію addApples, яка збільшує кількість яблук.

Тепер повертається значення підрахунку в додаток за допомогою postMessage. Наступним кроком використовується працівника apple, який був створений в інтерфейсі (див. рис. 3.18).

```
function App() {
  const [noOfTomatoes, setNoOfTomatoes] = useState(0);
  const [noOfApples, setNoOfApples] = useState(0);

  useEffect(() => {
    appleWorker.onmessage = ($event) => {
      if ($event && $event.data) {
        setNoOfApples($event.data);
      }
    };
  },
 []);

  function addApples() {
    appleWorker.postMessage({
      msg: "increaseAppleCount",
      noOfApples: noOfApples,
    }
);
  }
  return (
    <main> ... </main>
  );}
```

Рисунок 3.18 – Web-worker

У useEffect зареєстрований слухача, який оновлює noOfApples, коли web-worker видає результат. И тепер потрібно оновити функцію addApples для

виклику web-worker. Тепер додаток може запускати кілька процесів одночасно, не блокуючи рендеринг інтерфейсу.

3.9 React ViewPort List

Як і реактивно-віртуалізований, React ViewPort List [Г] використовує техніку, що називається віконним рендерингом, яка рендерить лише частину списку за раз, значно зменшуючи час, необхідний для повторного рендерингу компонентів, а також кількість створюваних DOM-вузлів. React ViewPort List [14] має деякі цікаві можливості, такі як:

– підтримка вертикальних та горизонтальних списків;
– підтримка прокрутки для індексації;
– вирівнювання Flexbox;
– динамічна висота та ширина для області перегляду.

Компонент Viewport List приймає проп item, який призначається набору даних для списків. Потім список відображається у вигляді рядків з відступом 8px і мінімальним розміром висоти 40px.

При роботі з великим списком важливо не рендерити всі дані одразу, щоб уникнути перевантаження DOM-дерева.

Найкращий підхід до підвищення продуктивності залежить від конкретного випадку використання. Якщо ви віддаєте перевагу відображенню всіх даних в одному місці, найкращим варіантом буде нескінченна прокрутка або віконна техніка. В іншому випадку, ви можете віддати перевагу пагінації, яка сегментує дані на різних сторінках.

ВИСНОВКИ

Проведений аналіз функціональності рендера компонентів у фреймворку React з використанням різних типів рендерінгу (CSR, SSR, SSG) підкреслив важливість обраного методу в контексті конкретного вебпроєкту. Визначено, що ефективність різних типів рендерінгу залежить від конкретного сценарію використання, та відбувається обрання оптимального підходу для покращення продуктивності в конкретних умовах.

Глибоке вивчення внутрішньої структури React виявило ключові моменти, такі як використання Virtual DOM та алгоритми порівняння дерев, що визначають успішність рендера. Виявлено, що використання Virtual DOM сприяє ефективному оновленню, швидкості рендера та кросбраузерності, що є критичними чинниками у веброзробці сучасних додатків.

Досліджено практичні аспекти роботи React, включаючи вирішення проблем за допомогою Virtual DOM, пріоритизацію та алгоритми порівняння дерев. Акцент робиться на тому, як різні методи оптимізації можуть впливати на продуктивність та ресурсоємність в реальному вебдодатку.

Розглянуто варіанти вирішення проблем та оптимізації відображення React компонентів. Застосування реекспорту, профілювання додатку, зберігання стану компонентів, ліниве завантаження та інші методи розглянуто у контексті їхнього впливу на продуктивність в реальному вебпроєкті.

Встановлено що ефективне поєднання різноманітних методів оптимізації може суттєво підняти якість та продуктивність React додатків.

ПЕРЕЛІК ПОСИЛАНЬ

1. Client-Side Rendering (CSR). W3C. URL: https://www.w3.org/WAI/WCAG22/quickref/?versions=2.1#client-side-rendering (дата звернення: 09.07.2023).
2. MDN Web Docs. Server-Side Rendering (SSR). URL: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction (дата звернення: 12.07.2023).
3. Next.js Documentation. Static Site Generation (SSG). URL: https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts#static-generation-recommended (дата звернення: 12.07.2023).
4. React Documentation. Virtual DOM. URL: https://legacy.reactjs.org/docs/faq-internals.html (дата звернення: 13.10.2023).
5. React Documentation. React Internals – Reconciliation Algorithm. URL: https://react.dev/learn/preserving-and-resetting-state (дата звернення: 13.07.2023).
6. Understanding React’s Virtual DOM. URL: https://blog.logrocket.com/virtual-dom-react/ (дата звернення:14.08.2023).
7. Оптимізація продуктивності за допомогою реекспорту. URL: https://tech.groww.in/improve-web-performance-if-you-are-using-re-exports-daaabb0aa422 (дата звернення: 14.08.2023).
8. Зберігання стану компонентів на місцевому рівні в React. URL: https://react.dev/reference/react/hooks (дата звернення: 20.08.2023).
9. Запам’ятовування React-компонентів для уникнення непотрібних повторних рендерів. URL: https://react.dev/reference/react/memo (дата звернення: 20.08.2023).
10. Розділення коду в React за допомогою динамічного імпорту. URL: https://legacy.reactjs.org/docs/code-splitting.html (дата звернення: 20.08..2023).
11. Lazy loading зображень у React. URL:

https://www.freecodecamp.org/news/how-to-lazy-load-images-in-react/ (дата звернення: 13.09.2023).

12. Використання незмінних структур даних (immutable data structures) в React. URL: https://immutable-js.com/ (дата звернення: 13.09.2023).
13. Застосування web-workers у React. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (дата звернення: 13.09.2023).
14. React ViewPort List. URL: https://blog.logrocket.com/rendering-large-lists-react-virtualized/ (дата звернення: 13.09.2023).
15. Client-side Rendering. URL: https://www.patterns.dev/react/client-side-rendering/#:~:text=In%20Client%2DSide%20Rendering%20(CSR,executes%20in%20the%20browser%2Fclient (дата звернення: 03.10.2023).
16. A Survey on Tree Edit Distance and Related Problems Philip Bille∗ The IT University of Copenhagen Glentevej 67, DK-2400 Copenhagen NV, Denmark. URL: https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf (дата звернення: 03.10.2023).
17. Reconciliation. URL: https://legacy.reactjs.org/docs/reconciliation.html (дата звернення: 26.10.2023).
18. The Most Popular Front-end Frameworks. URL: https://stackdiary.com/front-end-frameworks/ (дата звернення: 10.10.2023).

ДОДАТОК А

Синтаксис експортів

```
export let name1, name2/*, … */; // also var
export const name1 = 1, name2 = 2/*, … */; // also var, let
export function functionName() { /* … */ }
export class ClassName { /* … */ }
export function* generatorFunctionName() { /* … */ }
export const { name1, name2: bar } = o;
export const [ name1, name2 ] = array;
// Export list export { name1, /* …, */ nameN };
export { variable1 as name1, variable2 as name2, /* …, */ nameN };
export { variable1 as "string name" };
export { name1 as default /*, … */ };
// Default exports
export default expression;
 export default function functionName() { /* … */ }
 export default class ClassName { /* … */ }
export default function* generatorFunctionName() { /* … */ }
export default function () { /* … */ }
export default class { /* … */ }
export default function* () { /* … */ }
// Aggregating modules
export * from "module-name";
export * as name1 from "module-name";
export { name1, /* …, */ nameN } from "module-name";
export { import1 as name1, import2 as name2, /* …, */ nameN } from
"module-name";
export { default, /* …, */ } from "module-name";
```

```
export { default as name1 } from "module-name";
import React, { useState } from "react";
```

ДОДАТОК Б

Компонент CarSelectionForm

```
import style from "./CarSelectionForm.module.scss";
import { Input } from "../../UI/Input/Input";
import { Loading } from "../../UI/Loader/Loading";
import { InputNumber } from "../../UI/InputNumber/InputNumber";
import { RangeUI } from "../../UI/Range/Range";
import { Textarea } from "../../UI/Textarea/Textarea";
import { SentenceLink } from "../../SentenceLink/SentenceLink";
import { Button } from "../../UI/Button/Button";
import { useForm } from "react-hook-form";
import { yupResolver } from "@hookform/resolvers/yup";
import schemaContactForm from "../../../helpers/yup/yup";
import { usePostData } from "../../../hooks/usePostData";
function CarSelectionForm() {
  const [initialPrize, setInitialPrize] = useState(4140000);
  const [text, setText] = useState("");
  const {
    register,
    handleSubmit,
    formState: { errors },
    reset,
  } = useForm({ resolver: yupResolver(schemaContactForm) });
  const onSubmit = (data) =>
    submitForm({ ...data, description: text, initial_fee: initialPrize });
  const { error, loading, postingData } = usePostData();
  function submitForm(data) {
    postingData("declaration_car_selection", data);
```

```
  !error && reset();
}


return (
  <>
    {loading ? (
      <Loading height={30} width={500} />
    ) : (
      <form
        className={style["selected-form"]}
        onSubmit={handleSubmit(onSubmit)}
      >
        <h4 className={style["selected-form__title"]}>Ваши контакты</h4>
        <div className={style["input-section"]}>
          <Input
            id="user_name"
            {...register("user_name")}
            isValid={!errors.user_name}
            placeholder="Имя"
            mode="input--light-gray"
          />
          <Input
            id="phone"
            {...register("phone")}
            isValid={!errors.phone}
            placeholder="Телефон"
            mode="input--light-gray"
          />
        </div>
```

```
              <h4 className={style["selected-form__title"]}>
                Первоначальный взнос
              </h4>
              <div className={style["input-section-bottom"]}>
                <RangeUI
                  MIN={150000}
                  MAX={6000000}
                  center={4140000}
                  currency="₽"
                  setRange={setInitialPrize}
                />
                <InputNumber
                  mode="--light-gray"
                  currency="₽"
                  value={initialPrize}
                />
              </div>
              <h4 className={style["selected-form__title"]}>Ваш
комментарий</h4>
              <Textarea onChange={(e) => setText(e.target.value)} />
              <SentenceLink />
              <Button type="submit">Отправить заявку</Button>
            </form>
          )}
        </>
      );
    }


    export { CarSelectionForm };
```

ДОДАТОК В

Компонент SelectComponent

```
import React, { useState, useRef, memo } from "react";
import style from "./Select.module.scss";
import classNames from "classnames";
import { ReactComponent as Arrow } from "../../../assets/images/arrow.svg";
import { useOnClickOutside } from "../../../hooks/useOnClickOutside";
import PropTypes from "prop-types";
import { useEffect } from "react";
import { useSelector } from "react-redux";
import {
  selectCars,
  selectFilteredCar,
} from "../../../store/cars/carSelectors";
function SelectComponent({ ...props }) {
  const cars = useSelector(selectCars);
  const filteredCar = useSelector(selectFilteredCar);
  const [isSelect, setSelect] = useState(false);
  const [s, sets] = useState("");
  const [f, setF] = useState("");
  const {
    selectData = { name: "", data: [""] },
    mode = "",
    setInfo = sets,
    field = "",
    setDataField = setF,
    setDataWithIndex = setF,
    isEmpty = false,
```

```
  index = "1",
} = props;
const [title, setTitle] = useState(selectData.name);
const [isInvalid, onInvalid] = useState(false);
useEffect(() => {
  onInvalid(isEmpty);
}, [isEmpty]);


function onSelect(item) {
  setTitle(item);
  setSelect(false);
  setInfo(selectData.name, item);
  setDataField(field, item);
  setDataWithIndex(index, field, item);
  onInvalid(false);
}


useEffect(() => {
  if (selectData.name != "Марка") {
    setTitle(selectData.name);
  }


  if (!field && cars.length == filteredCar.length) {
    setTitle(selectData.name);
  }
}, [selectData]);
```

```
const ref = useRef();
useOnClickOutside(ref, () => setSelect(false));
return (
  <div
    className={classNames(
      style.select,
      style["select" + mode],
      isInvalid && style["select--empty"]
    )}
    ref={ref}
  >
    <div className={style.select__title} onClick={() => setSelect(!isSelect)}>
      <p
        className={classNames(
          style.select__name,
          isSelect && style["active-name"]
        )}
      >
        {title}
      </p>


      <Arrow
        className={classNames(
          style["select-arrow"],
          isSelect && style["select-arrow-active"]
        )}
      />
    </div>
```

```
      <ul
        className={classNames(
          style["select-list"],
          style["select-list" + mode],
          isSelect && style["active"]
        )}
      >
        {selectData.data.map((item, index) => (
          <li
            onClick={() => onSelect(item)}
            className={style["select-list__item"]}
            key={index}
          >
            {item}
          </li>
        ))}
      </ul>
    </div>
  );
}
SelectComponent.propTypes = {
  mode: PropTypes.string,
  selectData: PropTypes.object,
};
export const Select = memo(SelectComponent);
```

ДОДАТОК Г

Приклад використання react-viewport-list

```
import React from "react";
import { faker } from "@faker-js/faker";
import { useRef } from "react";
import ViewportList from "react-viewport-list";



const App = () => {
  const ref = useRef(null);
  const items = new Array(1000).fill().map((value, index) => ({
    id: index,
    name: faker.name.firstName(5),
    body: faker.lorem.paragraph(8),
  }));



  return (
    <div className="scroll-container" ref={ref}>
      <ViewportList viewportRef={ref} items={items} itemMinSize={40}
margin={8}>
        {(item) => (
          <div key={item.id} className="post">
            <h3>
              {item.name} - {item.id}
            </h3>
            <p>{item.body}</p>
          </div>
```

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