Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____Інформаційних управляючих систем_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____

Дослідження методів управління станом у веб-орієнтованих додатках на основі фреймворку React для інформаційних систем електронної комерції_____
(тема)

Виконав:
здобувач_2_ року навчання, групи ІУСТм-24-1
_____Андрій ДВУГРОШЕВ_____
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність _122 Комп'ютерні науки_____
(код і повна назва спеціальності)
Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)
Освітня програма _Інформаційні управляючі системи та технології_____
(повна назва освітньої програми)

Керівник ___доц. каф. ІУС Олена МІХНОВА_
(посада, власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту

Завідувач кафедри ІУС _____ ___Костянтин ПЕТРОВ__
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерних наук_____

Кафедра_____Інформаційних управляючих систем_____

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність_____122 Комп'ютерні науки_____
(код і повна назва)

Тип програми _____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма___Інформаційні управляючі системи та технології_____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«_24__»__листопада__ 20 25_ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві_____Двугрошеву Андрію Олексійовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи ___«Дослідження методів управління станом у веб-орієнтованих додатках на основі фреймворку React для інформаційних систем електронної комерції»_____
_____
затверджена наказом університету від _24_ __листопада__2025_ р. № _1055Ст_
2. Термін подання здобувачем роботи до екзаменаційної комісії _18_ __грудня__ 2025 р.
3. Вихідні дані до роботи Матеріали науково-дослідної практики, огляд науково-_____
технічних джерел щодо методів управління станом, інтернет ресурси щодо теми_____
кваліфікаційної роботи._____
_____
_____
_____
_____
_____
_____
4. Перелік питань, що потрібно опрацювати в роботі _Аналіз сучасних методів управління станом у веб-орієнтованих додатках. Визначення вимог до управління станом в_____
інформаційних системах електронної комерції. Порівняльний аналіз переваг і недоліків__
існуючих рішень. Апробація запропонованого комбінованого методу._____
_____
_____
_____
_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної області, особливостей методів управління станом для ІС електронної комерції	24.11.2025 – 28.11.2025	Виконано
2	Постановка мети і завдання дослідження	29.11.2025 – 31.11.2025	Виконано
3	Аналіз існуючих методів оцінки управління станом	01.12.2025 – 03.12.2025	Виконано
4	Розробка комбінованого методу управління станом	04.12.2025 – 08.12.2025	Виконано
5	Практична апробація розробленого методу	08.12.2025 – 11.12.2025	Виконано
6	Оформлення пояснювальної записки	12.12.2025 – 13.12.2025	Виконано
7	Оформлення графічної частини та презентаційних матеріалів	14.12.2025 – 15.12.2025	Виконано
8	Представлення на рецензування	16.12.2025 – 17.12.2025	Виконано
9	Представлення атестаційної роботи до ЕК	18.12.2025	Виконано

Дата видачі завдання __24_ ____листопада___2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ __доц. каф. ІУС Олена МІХНОВА_____
(підпис) (посада, власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 87 с., 29 рис., 14 табл., 34 джерела.

ВЕБ-ЗАСТОСУНКИ, ГЛОБАЛЬНИЙ СТАН, ЕЛЕКТРОННА КОМЕРЦІЯ, СИНХРОНІЗАЦІЯ ДАНИХ, УПРАВЛІННЯ СТАНОМ, CONTEXT API, MOBX, REACT, RECOIL, REDUX, ZUSTAND.

Об’єктом дослідження є процес управління станом у веб-орієнтованих додатках інформаційних систем електронної комерції, створених на основі фреймворку React.

Метою роботи є дослідження, порівняльний аналіз та вдосконалення сучасних методів управління станом у React-застосунках з метою підвищення ефективності обробки даних, стабільності та масштабованості інформаційних систем електронної комерції.

У процесі дослідження застосовано методи системного аналізу, порівняльний аналіз архітектур клієнтських застосунків, експериментальне моделювання, а також аналітичні методи оцінки продуктивності різних підходів до управління станом. Проведено експериментальне порівняння продуктивності різних методів управління станом у тестовому e-commerce застосунку.

У результаті дослідження було розроблено комбінований метод управління станом, що поєднує переваги легковагової бібліотеки Zustand для роботи з зовнішніми даними та API та реактивної моделі MobX для високопродуктивного оновлення клієнтського інтерфейсу.

ABSTRACT

The explanatory note of the qualification thesis: 87 pages, 29 figures, 14 tables, 34 sources.

CONTEXT API, DATA SYNCHRONIZATION, E-COMMERCE, GLOBAL STATE, MOBX, REACT, RECOIL, REDUX, STATE MANAGEMENT, WEB APPLICATIONS, ZUSTAND.

The object of the research is the state management process in web-oriented applications of e-commerce information systems developed using the React framework.

The aim of the thesis is to investigate, compare, and improve modern state management methods in React applications in order to enhance data processing efficiency, stability, and scalability of e-commerce information systems.

The research employs methods of system analysis, comparative analysis of client-side application architectures, experimental modeling, as well as analytical methods for evaluating the performance of different state management approaches. Experimental performance comparison of various state management methods was carried out in a test e-commerce application.

As a result of the research, a combined state management method was developed, integrating the advantages of the lightweight Zustand library for working with external data and APIs with the reactive MobX model for high-performance client-side interface updates.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ІС – інформаційна система

API – Application Programming Interface (прикладний програмний інтерфейс)

CRUD – Create, Read, Update, Delete (створення, читання, оновлення, видалення)

CSS – Cascading Style Sheets (каскадні таблиці стилів)

DOM – Document Object Model (об’єктна модель документа)

DX – Developer Experience (досвід розробника)

GC – Garbage Collector (збирач сміття)

HTML – HyperText Markup Language (мова розмітки гіпертексту)

IDE – Integrated Development Environment (інтегроване середовище розробки)

JSON – JavaScript Object Notation (об’єктна нотація JavaScript)

SEO – Search Engine Optimization (пошукова оптимізація)

SSR – Server-Side Rendering (рендеринг на стороні сервера)

SWR – Stale-While-Revalidate (стратегія повторної валідації застарілих даних)

UI – User Interface (інтерфейс користувача)

UX – User Experience (користувацький досвід)

ВСТУП

У сучасному цифровому світі веб-орієнтовані додатки стали основним засобом взаємодії користувача з інформаційними системами. Особливо це стосується сфери електронної комерції, де від ефективності та стабільності роботи веб-додатка безпосередньо залежить рівень довіри клієнтів, швидкість обробки транзакцій і, як наслідок, конкурентоспроможність підприємства. Зростання складності веб-застосунків, збільшення обсягів даних і необхідність забезпечення безперервної взаємодії між компонентами призвели до підвищення значущості управління станом (state management) як ключового аспекту архітектури сучасних інформаційних систем.

Станом на сьогодні провідні науковці та розробники зосереджують увагу на пошуку ефективних методів управління станом у клієнтських додатках. У науково-дослідних центрах таких компаній, як Meta (Facebook), Google та Microsoft, активно розвиваються бібліотеки та фреймворки, орієнтовані на оптимізацію передачі та синхронізації даних між компонентами інтерфейсу. Зокрема, фреймворк React, створений компанією Meta, є однією з найпопулярніших технологій для розробки динамічних веб-додатків. Його екосистема підтримує широкий набір інструментів для управління станом, таких як Redux, MobX, Zustand, Recoil та Context Application Programming Interface (API), які реалізують різні концепції архітектурних підходів до зберігання, оновлення та передачі стану.

Дослідження вчених у галузі комп’ютерних наук підтверджують, що правильний вибір методу управління станом суттєво впливає на продуктивність додатка, його масштабованість та зручність у підтримці. У прикладних розробках для електронної комерції особливо актуальними є підходи, що забезпечують узгодженість даних між клієнтською та серверною частинами, стабільну роботу при високому навантаженні та можливість швидкої адаптації до змін бізнес-логіки.

На глобальному рівні простежується тенденція до стандартизації підходів до управління станом у рамках компонентних архітектур. Провідні світові IT-компанії активно впроваджують реактивні моделі управління станом, які ґрунтуються на концепціях унідірекційного потоку даних, іммутабельності стану та детермінованих ред'юсерів. У галузі електронної комерції все більшого поширення набувають інтегровані підходи, що поєднують Redux або Context API із серверними технологіями, такими як Next.js чи GraphQL, для забезпечення єдиного стану між клієнтом і сервером [1].

Також набуває популярності використання бібліотек нового покоління – Zustand, MobX, які дозволяють розробникам створювати легковагові рішення без складних конфігурацій, зберігаючи при цьому контроль над логікою оновлення стану [2]. Це відображає рух до підвищення ефективності розробки та зниження технічної складності великих систем.

Актуальність дослідження визначається необхідністю вдосконалення підходів до управління станом у веб-орієнтованих додатках, зокрема для інформаційних систем електронної комерції. Зі збільшенням кількості користувачів та складності бізнес-логіки стає недостатньо класичних підходів до роботи зі станом, таких як передача даних через властивості компонентів або використання простих локальних станів. Необхідно забезпечити ефективну синхронізацію даних, швидку реакцію інтерфейсу на зміни та мінімізацію конфліктів між компонентами.

Своєчасність проведення дослідження обумовлена активним розвитком ринку електронної комерції, що вимагає створення масштабованих, продуктивних і зручних у підтримці систем. Розробка сучасних підходів до управління станом дозволить підвищити стабільність роботи інформаційної системи (ІС), скоротити час реакції системи та поліпшити користувацький досвід.

Метою дипломної роботи є дослідження та порівняльний аналіз сучасних методів управління станом у веб-орієнтованих додатках на основі фреймворку

React з метою вдосконалення процесів обробки та синхронізації даних у інформаційних системах електронної комерції.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ОСОБЛИВОСТЕЙ ТА НАЯВНИХ РІШЕНЬ УПРАВЛІННЯ СТАНОМ У ВЕБ-ОРІЄНТОВАНИХ ДОДАТКАХ

1.1 Аналіз бізнес-процесів ІС електронної комерції, що потребують управління станом на рівні веб-додатків (клієнтська частина)

Електронна комерція залишається надзвичайно актуальною з огляду на стійку зміну поведінки споживачів у бік онлайн-покупок, широку доступність мобільного інтернету та розвиток платіжної інфраструктури. Сучасні ринки дедалі більше орієнтуються на поєднання веб-порталів, мобільних застосунків і соціальних платформ, що дозволяє бізнесу досягати більшої аудиторії та знижувати витрати на утримання фізичної мережі продажів. Пандемія та інші макротенденції прискорили перехід частини торгових операцій в онлайн, а також сприяли розвитку персоналізованих сервісів, швидкої доставки і підписних моделей, що робить інвестиції в надійні та масштабовані ІС електронної комерції критично важливими для конкурентоздатності.

Управління станом у веб-орієнтованих додатках є ключовим елементом для забезпечення передбачуваного, швидкого та безпечного користувацького досвіду. Коректна організація стану дозволяє синхронізувати локальні дії користувача з серверними даними, реалізувати оптимістичні оновлення, ефективне кешування та відкат у разі помилок, а також зменшити кількість зайвих перерендерів і відповідно підвищити продуктивність [3]. У контексті e-commerce це безпосередньо впливає на бізнес-метрики – час на оформлення замовлення, відсоток завершених покупок і утримання клієнтів – тому вибір підходу до управління станом має технічне й економічне значення: він визначає масштабованість, стійкість до мережевих збоїв, можливості для відлагодження та швидкість розробки нових функцій.

У клієнтській частині інформаційної системи електронної комерції управління станом забезпечує коректну поведінку інтерфейсу, узгодженість

даних із сервером та належний користувацький досвід за високих навантажень і при нестабільному мережевому з’єднанні. Нижче наведено перелік бізнес-процесів і вимог до управління їх станом:

– кошик покупок та проміжний стан замовлення;

– багатокроковий процес оформлення замовлення та оркестрація кроків;

– аутентифікація, авторизація та синхронізація сесій між кількома вкладками і пристроями;

– перегляд каталогу, фільтрація, сортування і кешування результатів пошуку;

– оновлення в реальному часі: запаси, статуси замовлень, повідомлення;

– персоналізація, рекомендації та збережені налаштування інтерфейсу;

– обробка платежів та транзакційні сценарії з відкатом;

– складні взаємодії інтерфейсу: модальні вікна, багатосторінкові форми, вкладені операції;

– офлайн-режим і політика повторної синхронізації.

Кошик – динамічний клієнтський домен із частими локальними змінами: додавання й видалення товарів, зміна кількостей, застосування купонів. Такі операції потребують миттєвого відгуку інтерфейсу, збереження проміжного стану під час навігації й між сесіями, а також правильної синхронізації із сервером для запобігання конфліктам. Архітектурно це вимагає розмежування локального стану інтерфейсу і механізмів, що відповідають за узгодження з серверною частиною і повернення до попереднього стану у разі помилок.

Багатокроковий процес оформлення замовлення та оркестрація кроків складається з послідовних кроків із валідаціями, резервуванням товарів, вибором способу доставки та платіжною логікою. Клієнтська частина повинна забезпечувати детерміністичну модель переходів між станами, підтримувати відновлення після помилок та передбачувану поведінку при асинхронних подіях. Для таких сценаріїв важливі можливості трасування переходів станів і їх відлагодження.

При аутентифікації, авторизації та синхронізації сесій між кількома вкладками і пристроями, клієнтська частина зберігає і поширює інформацію про вхід користувача, ролі та права доступу; обробляє оновлення токенів і стан «пам’яті» користувача. Це потребує узгодженості стану між вкладками та пристроями, коректного очищення стану при виході, безпечного зберігання даних аутентифікації та мінімізації ризику несанкціонованого доступу.

Перегляд каталогу, фільтрація, сортування і кешування результатів пошуку вимагають збереження стану фільтрів, номера сторінки і параметрів сортування, а також ефективного кешування запитів для зменшення затримок при повторних переходах. Клієнтська частина має керувати інвалідізацією кешу при зміні даних і поєднувати локальні налаштування інтерфейсу з механізмами кешування серверного стану.

Оновлення в реальному часі: запаси, статуси замовлень, повідомлення вимагають від клієнтської частини правильної агрегації потокових даних і оперативного оновлення локального стану без погіршення взаємодії з користувачем. Сценарії конкурентних змін потребують механізмів вирішення конфліктів на клієнті та координації з серверною логікою.

Персоналізація, рекомендації та збережені налаштування інтерфейсу поєднує локальні переваги користувача та результати серверних моделей рекомендацій. Важлива синхронізація між сесіями і пристроями та коректне застосування оновлень від сервера без руйнування локальних налаштувань.

Обробка платежів та транзакційні сценарії з відкатом потребують чіткого інформування про стан транзакції та механізми повернення локального стану у разі помилок. Клієнтська частина має опрацьовувати асинхронні зворотні виклики платіжних шлюзів і узгоджувати видимий стан із реальною транзакцією на сервері.

Складні взаємодії інтерфейсу: модальні вікна, багатосторінкові форми, вкладені операції потребують централізованих механізмів координації стану, щоб уникнути неконсистентних відображень і неправильних переходів між підстанами інтерфейсу.

Офлайн-режим і політика повторної синхронізації клієнтська частина має підтримувати локальне збереження дій, чергувати запити, виконувати повторну синхронізацію при відновленні зв’язку і вирішувати конфлікти змін з урахуванням бізнес-правил

E-commerce-додатки зазвичай мають високу динамічність: змінюються кількість товарів, ціни, наявність, застосовуються фільтри, змінюється мова чи валюта – усе це призводить до численних змін Document Object Model (DOM) [4]. Якщо ці зміни не координуються централізовано через state management, можливі:

– надмірні перерендери;

– конфлікти станів (наприклад, кількість товару в кошику в DOM не відповідає стану в пам’яті);

– затримки у відповіді User Interface (UI).

Була спроєктована організаційна структура інформаційної системи електронної комерції, що забезпечує комплексну підтримку процесів продажу товарів, управління замовленнями, обробки платежів і взаємодії з користувачами в онлайн-середовищі. Схема організаційної структури підприємства електронної комерції наведена на рисунку 1.1.

Інформаційна система електронної комерції є комплексом взаємопов’язаних підсистем, що забезпечують автоматизацію основних бізнес-процесів підприємства, пов’язаних із продажем товарів та послуг через мережу Інтернет. Організаційна структура ІС e-commerce включає як технічні компоненти (серверна, клієнтська та інтеграційна частини), так і функціональні підсистеми, що відповідають за окремі напрями діяльності компанії. На рисунку А.1-А.3 у додатку А подано структурну схему ІС електронної комерції, що відображає зв’язки між її компонентами та рівнями архітектури з декомпозицією 2 рівнів.

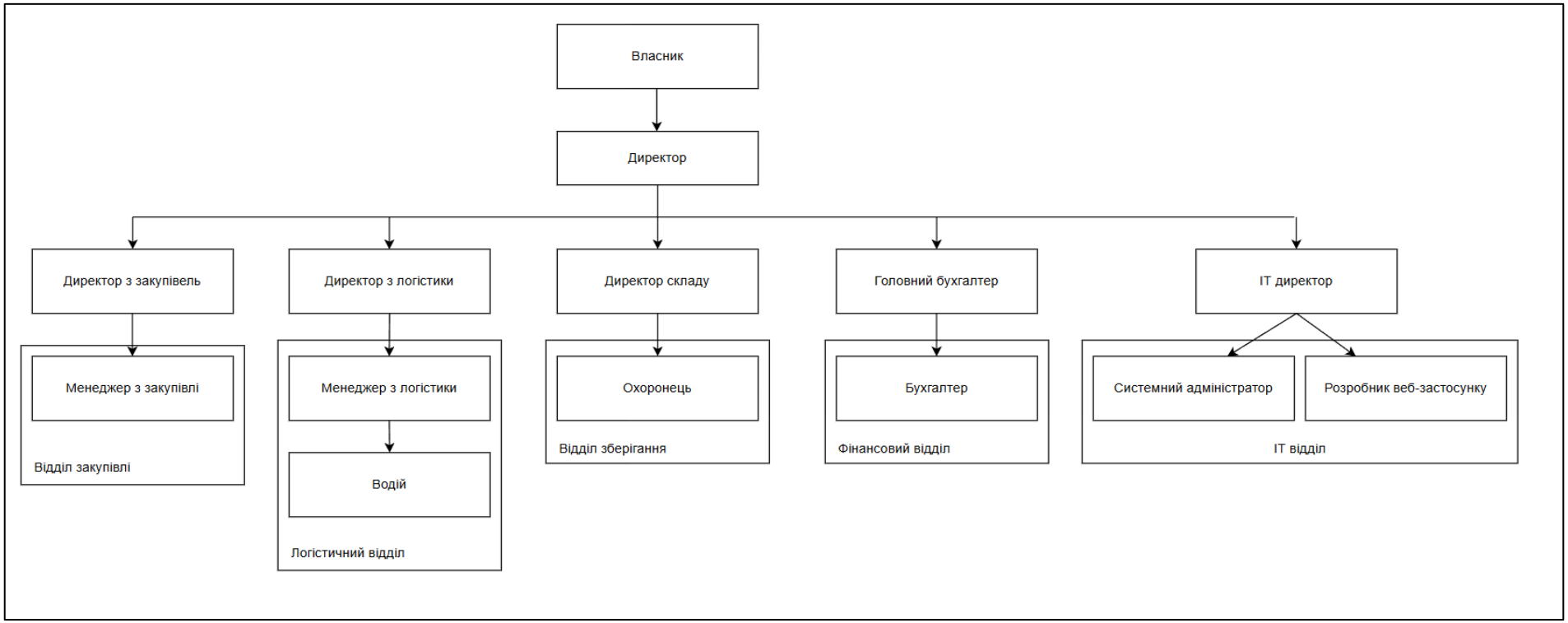

Рисунок 1.1 – Організаційна структура підприємства електронної комерції

Управління станом у клієнтській частині e-commerce має одночасно вирішувати кілька взаємопов’язаних завдань: забезпечення низької затримки інтерфейсу й оперативного відгуку для користувача, гарантування консистентності даних із сервером, стійкість до мережевих збоїв і мультисесійних сценаріїв, а також мінімізація витрат на продуктивність. Архітектурні рішення повинні передбачати чітке розмежування локального і серверного стану, механізми кешування й інвалідизації, політики передбачуваного оновлення і відкату, а також спеціалізовані моделі для оркестрації багатокрокових процесів. Таке розуміння бізнес-вимог є вихідною точкою для обґрунтованого вибору підходів і засобів управління станом у наступних розділах дослідження.

1.2 Наявні рішення управління станом у веб-орієнтованих додатках на основі фреймворку React

Управління станом у веб-орієнтованих додатках (особливо в таких, як React) – це процес контролю за змінами даних і їх оновленням у інтерфейсі користувача, щоб забезпечити стабільну та зручну роботу додатка. Це важливий аспект розробки, оскільки наявність ефективної системи управління станом дозволяє уникнути проблем з відображенням даних і полегшує підтримку масштабованості додатка.

Стан (state) – це об'єкти або змінні, що зберігають дані, які можуть змінюватися в процесі роботи додатка. У React, стан використовують для відображення інтерфейсу, зміни якого залежить від даних, що містяться у стані [5].

Управління станом у веб-орієнтованих додатках стало необхідним через обмеження класичних веб-технологій. У традиційних веб-додатках, побудованих за допомогою HyperText Markup Language (HTML), Cascading Style Sheets (CSS)

та JavaScript, кожен запит до серверу зазвичай призводить до повного перезавантаження сторінки. В результаті, стан інтерфейсу, який залежить від взаємодії з користувачем або даними, неможливо було оновлювати динамічно без перезавантаження.

З розвитком веб-технологій та появою JavaScript-фреймворків, таких як React, стало можливим відмовитись від повного перезавантаження сторінки. У React кожна зміна стану компонента може призвести до ререндерингу тільки тих частин інтерфейсу, які зазнали змін, без необхідності перезавантажувати всю сторінку. Це дозволяє створювати більш швидкі та динамічні веб-додатки, які можуть ефективно реагувати на зміни даних, взаємодію користувача та інші події. Принцип роботи управління станом у веб-додатках наведено у рисунку 1.2.

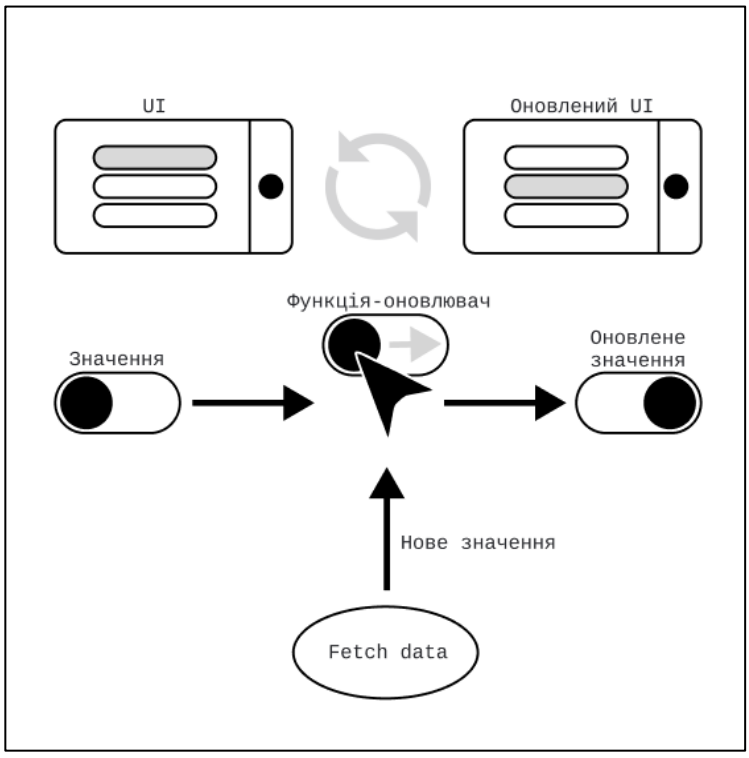

Рисунок 1.2 – Принцип роботи управління станом у веб-додатках

Основні принципи управління станом:

– локальний стан компонентів – кожен компонент у React може мати власний стан, який визначає, як цей компонент виглядатиме та як він буде реагувати на дії користувача (наприклад, натискання кнопки, введення тексту);

– глобальний стан – в деяких випадках необхідно зберігати загальні дані, доступні для кількох компонентів або навіть усієї програми, для цього використовуються стани, які зберігаються поза компонентами, в глобальному контексті;

– підняття стану (Lifting state up) – коли кілька компонентів повинні обмінюватися даними, цей стан піднімається до їх спільного батьківського компонента, це дозволяє централізовано управляти даними.

Фреймворк React забезпечує компонентно-орієнтовану архітектуру розробки веб-додатків, у межах якої управління станом є ключовим аспектом підтримання узгодженості даних та стабільної роботи інтерфейсу користувача. У сучасній практиці виділяють кілька основних підходів до організації управління станом, що відрізняються рівнем абстракції, масштабованістю та призначенням.

Локальний стан компонентів – це базовий механізм React, який забезпечує збереження і зміну стану безпосередньо всередині окремих компонентів. Такий підхід є доцільним для ізольованих частин інтерфейсу або нескладних сценаріїв, де взаємодія між компонентами мінімальна. Його перевагами є простота реалізації та висока продуктивність, проте він ускладнює підтримку великих застосунків через відсутність централізованого сховища даних.

Класичний state-менеджер у React на основі useState складається з трьох ключових елементів (рисунок 1.3): самого стану – змінної, яка зберігає поточні дані компонента; функції для оновлення стану – спеціальної функції, що дозволяє змінювати значення стану і автоматично викликає повторний рендер компонента; та початкового значення стану, яке задається під час ініціалізації і визначає стан до будь-яких змін [5]. Ці три елементи взаємодіють так, що зміна стану через updater-функцію приводить до реактивного оновлення інтерфейсу без перезавантаження сторінки.

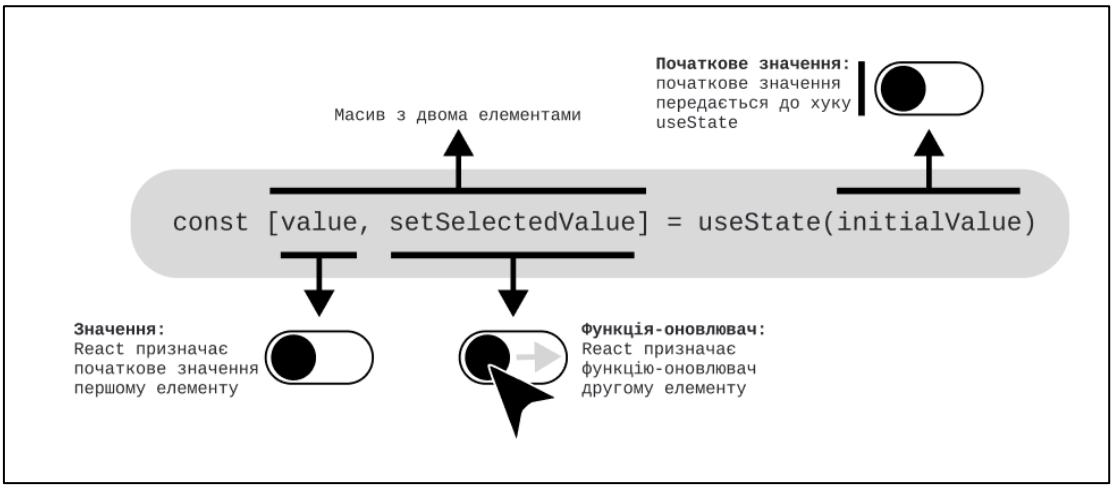

Рисунок 1.3 – Класичний state-менеджер на основі useState

Context API у поєднанні з useReducer або спеціалізованими провайдерами дає змогу передавати стан у глибоко вкладені компоненти без необхідності багаторівневої передачі властивостей. Цей підхід доцільний для додатків середньої складності, наприклад, для реалізації кошика покупок або налаштувань користувача. Разом з тим, при неефективній структурі контекстів можливе надмірне повторне рендерування компонентів, що потребує оптимізації селекторів та мемоізації даних [6].

Як ілюструє схема архітектури цього підходу (рисунок 1.4), при поєднанні Context API з хуком useReducer логіка стану чітко структурується: Reducers (редюсери) виступають як чисті функції, що детерміновано обчислюють новий стан на основі поточного стану та отриманої Actions (дії) – об'єкта, що описує подію [7]. Створений React Context через свій компонент Provider направляє цей стан та функцію dispatch (яка відправляє дії) дереву компонентів. Для отримання доступу до цих даних компоненти-споживачі використовують хук useContext [8]. Або, у застарілому підході, компонент Context.Consumer, що забезпечує їх автоматичне оновлення при зміні стану [9].

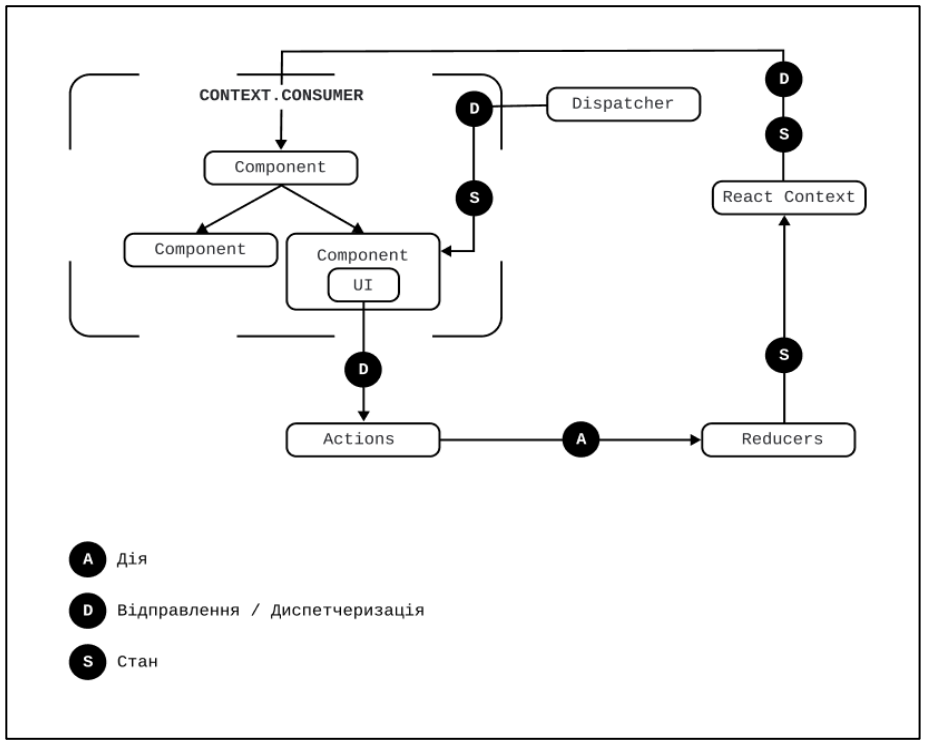

Рисунок 1.4 – Діаграма архітектури Context API

Redux / Redux Toolkit реалізує концепцію єдиного централізованого сховища стану з використанням детерміністичних ред`юсерів [10]. Інструментарій Redux Toolkit значно скорочує обсяг шаблонного коду та спрощує патерни роботи зі станом, що робить його доцільним вибором для великих корпоративних систем із розгалуженою бізнес-логікою [11]. Серед ключових переваг – прозорість, прогнозованість змін стану та розвинена екосистема засобів розробки.

Як детально ілюструє діаграма архітектури (рисунок 1.5), в основі системи лежить Store (сховище) – об'єкт, що інкапсулює весь стан додатку. Для інтеграції з React використовується компонент Redux Provider, який робить цей Store доступним для всієї ієрархії компонентів. Щоб отримати дані зі стану, компоненти інтерфейсу використовують хук useSelector, який підписується на оновлення та витягує потрібні дані. Зміна стану ініціюється виключно через виклик функції Dispatcher (диспетчера), зазвичай через хук useDispatch(). Цей диспетчер приймає Actions (дії) – прості об'єкти, що описують подію, яка відбулася та передає їх до Reducers. Редюсери є чистими функціями, які,

базуючись на поточному стані та отриманій дії, повертають новий стан store. У Redux Toolkit логіка редюсерів, генератори дій та початковий стан зручно групуються у Slice (зрізи), що значно спрощує структуру коду. Коли Store оновлюється новим станом, useSelector у відповідних компонентах виявляє зміни, що призводить до їх повторного рендерингу з актуальними даними.

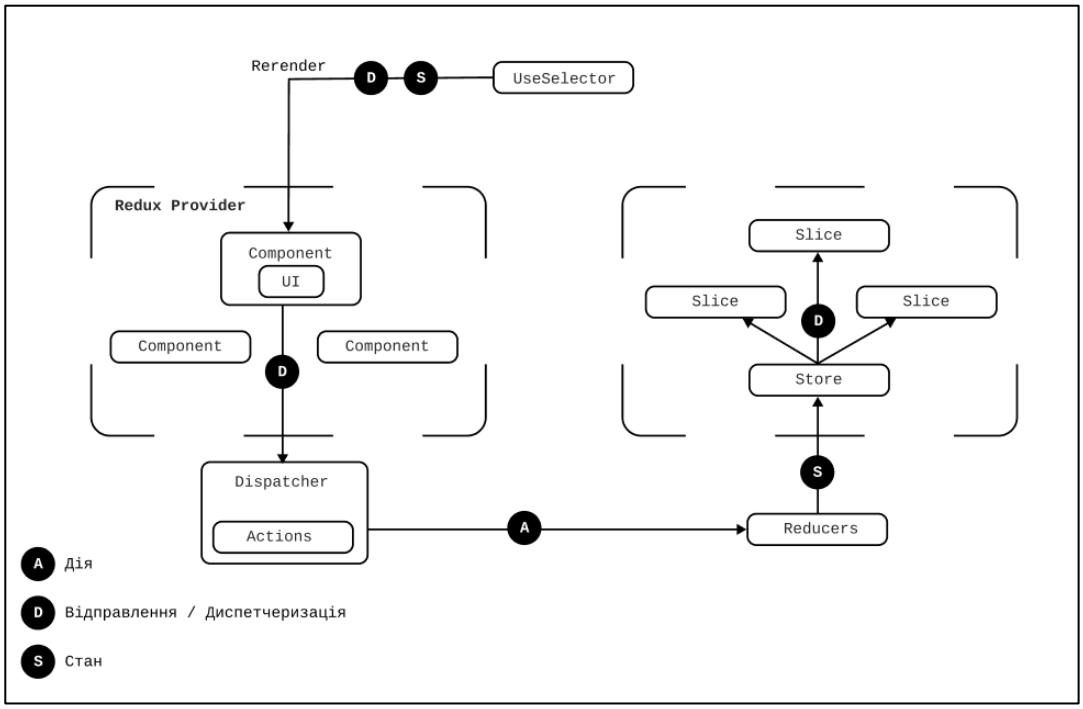

Рисунок 1.5 – Діаграма архітектури Redux / Redux Toolkit

MobX базується на реактивному підході до управління станом і реалізує патерн спостерігача. Система автоматично відстежує залежності між даними та компонентами, що мінімізує кількість зайвих оновлень інтерфейсу. Це рішення є ефективним у сценаріях, де необхідна висока реактивність та мінімальний обсяг коду для синхронізації стану [12].

Як детально показано на відповідній діаграмі архітектури (рисунок 1.6), в основі цього підходу лежить концепція деривацій (derivations). Фундаментальними елементами MobX є observable (спостережувані) дані – це стан додатку, який MobX відстежує за домопогою зовнішнього провайдеру змін. Компоненти React, які мають реагувати на ці зміни, оголошуються як observer (спостерігач). Коли такий компонент виконує рендеринг, MobX автоматично

фіксує, які саме observable дані були прочитані, і створює реактивний зв’язок. Модифікація стану повинна відбуватися виключно в Action (діях) – спеціально позначених функціях, що гарантує атомарність та прогнозованість змін. Окрім стану, MobX дозволяє створювати computed (обчислювані) значення, які автоматично кешуються і оновлюються лише тоді, коли змінюються observable дані, від яких вони залежать [13]. Таким чином, коли action змінює observable стан, MobX миттєво ідентифікує, які computed значення та observer-компоненти залежать саме від цих даних, і запускає їх оновлення (або повторний рендеринг) автоматично.

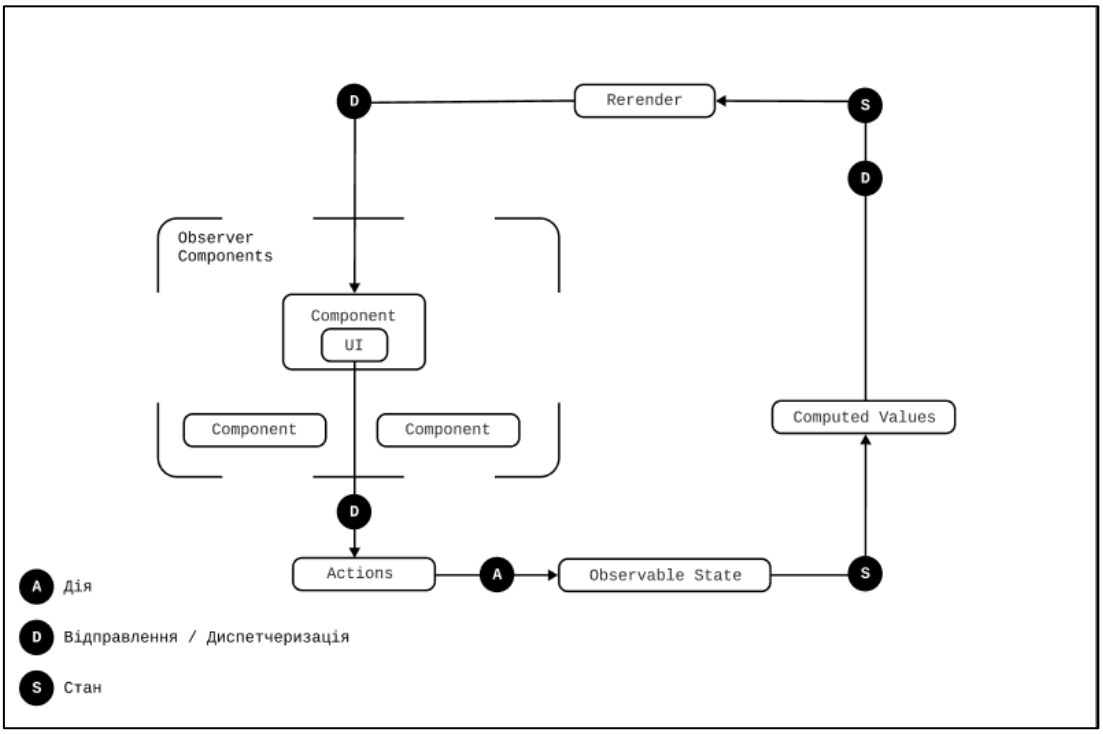

Рисунок 1.6 – Діаграма архітектури MobX

Zustand представляє сучасні легкі бібліотеки для управління станом, орієнтовані на спрощення розробки, модульність і продуктивність [14]. Вони забезпечують більш гнучку архітектуру, поєднуючи можливості локального та глобального збереження стану. Такі інструменти часто використовуються у швидкорозроблюваних або середніх за масштабом проєктах, де важливі низькі затримки та простота інтеграції. На відміну від Redux та Context API, бібліотека

Zustand не використовує механізм контексту React. Доступ до стану можна отримати безпосередньо у будь-якій частині застосунку – у сервісах [15].

Як ілюструє схема архітектури (рисунок 1.7), в основі Zustand лежить функція create, яка використовується для побудови модульного сховища (store). Ця функція приймає колбек, що отримує функцію set (для оновлення стану) та, за потреби, функцію get (для читання поточного стану всередині дій). Стан та функції для його зміни (actions) визначаються разом в одному об'єкті, що спрощує їх зв’язок. На відміну від Redux, set виконує поверхове злиття (shallow merge) стану, подібно до setState у класових компонентах React. Функція create повертає єдиний хук (наприклад, useStore), який є основним інтерфейсом для React-компонентів. Важливо, що повернений об'єкт useStore також надає методи .getState() та .setState(), які дозволяють маніпулювати станом напряму з будь-якого місця коду, включно з сервісами або обробниками подій поза межами React-компонентів.

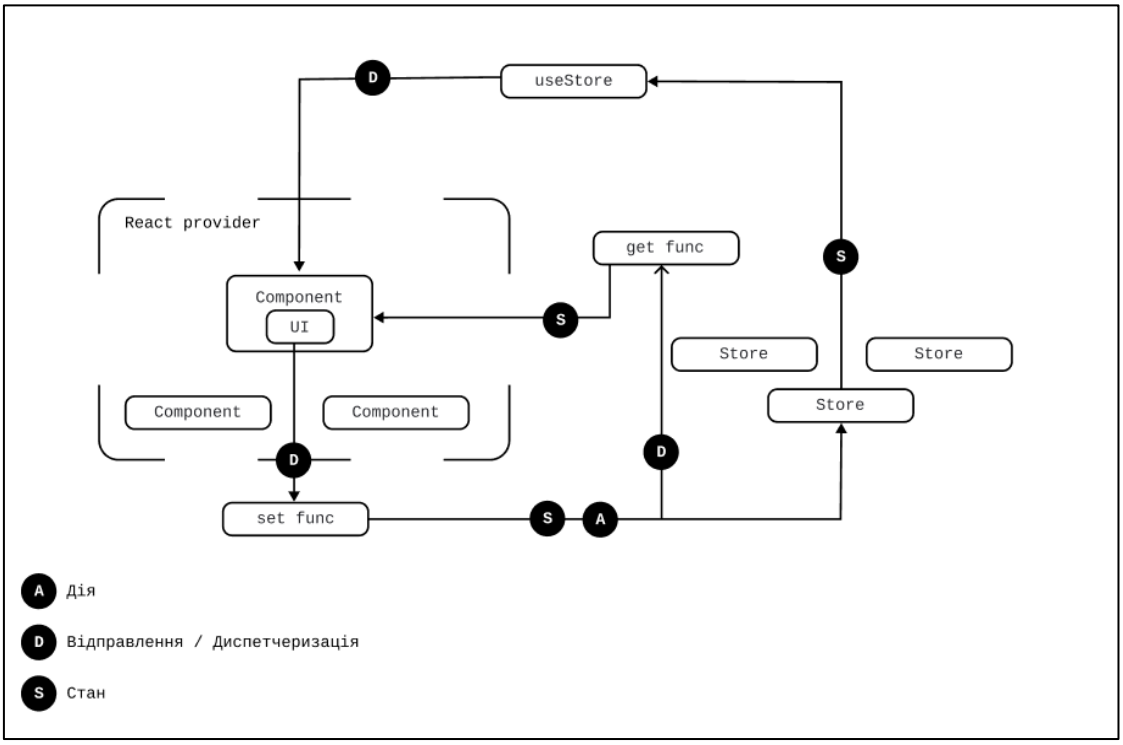

Рисунок 1.7 – Діаграма архітектури Zustand

XState реалізує підхід управління станом на основі машин станів і діаграм переходів, що забезпечує формальне описання поведінки системи [16]. Такий підхід є особливо корисним для моделювання складних процесів, зокрема оформлення замовлення або платіжних сценаріїв, де важливо забезпечити детермінованість і передбачуваність переходів між станами [17].

Як показано на відповідній діаграмі архітектури (рисунок 1.8), центральним елементом є Machine (машина) – конфігурація, що формально визначає скінченну множину станів системи (наприклад, idle, loading, success, error). Перехід між цими станами детерміновано контролюється через відправку Events (подій), які виступають єдиним тригером для змін. Коли Machine у поточному стані отримує подію, вона перевіряє наявність Transitions (переходів), що відповідають цій події. Важливою особливістю є Guards (умови переходу) – функції, які мають повернути true, щоб дозволити конкретний перехід. Для зберігання кількісних даних (наприклад, ідентифікатора замовлення або тексту помилки). Побічні ефекти, такі як запити до API або оновлення сховища, інкапсулюються в Actions (дії), які виконуються при зміні стану. Для запуску машини, взаємодії, рендеру сторінки використовується useMachine інтерпретатор.

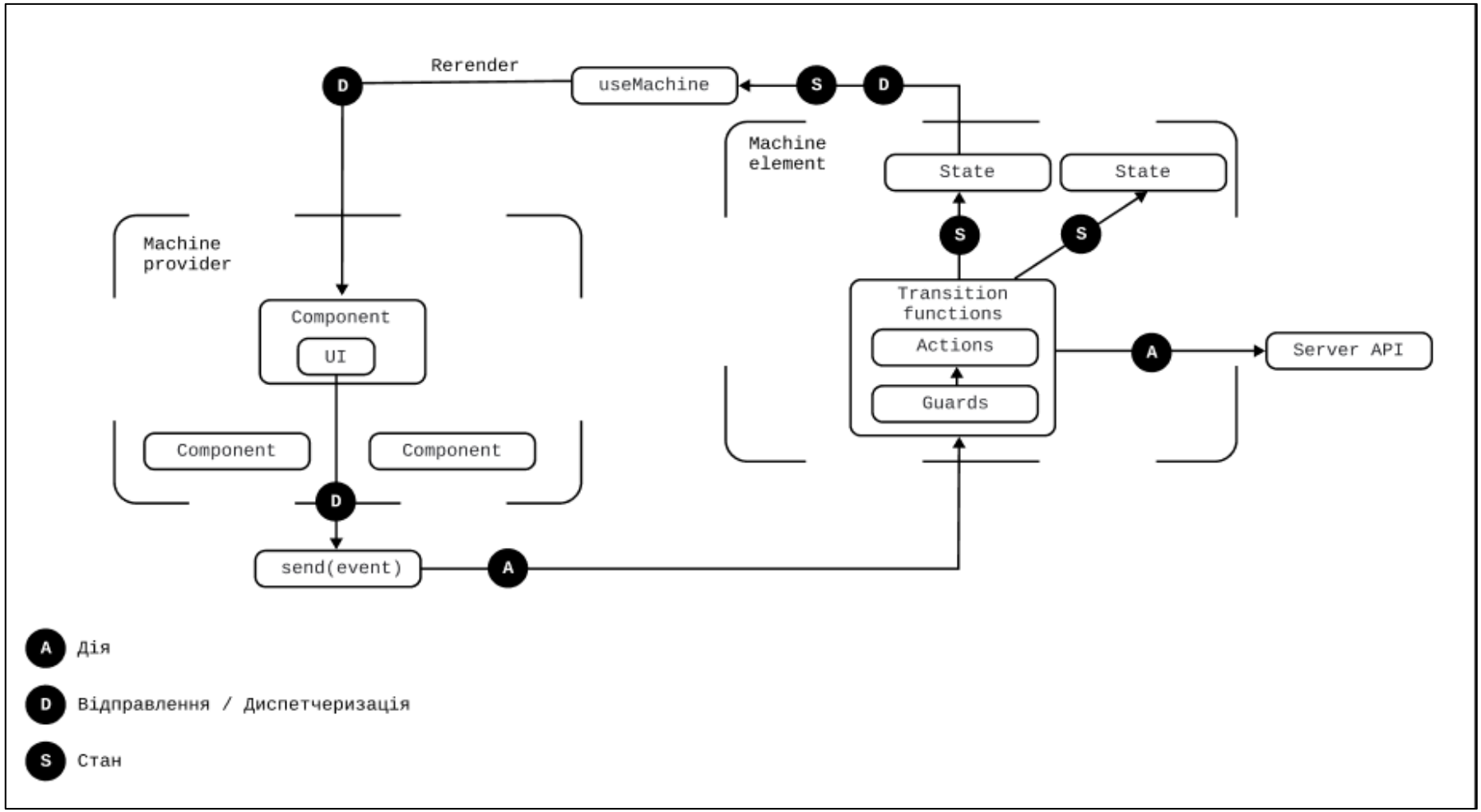

Рисунок 1.8 – Діаграма архітектури XState

Сучасна екосистема React пропонує широкий спектр рішень для управління станом – від простих локальних механізмів до складних моделей централізованого контролю й реактивної синхронізації даних. Вибір конкретного підходу визначається складністю бізнес-процесів, обсягом даних, вимогами до продуктивності та архітектурними принципами побудови інформаційної системи електронної комерції.

1.3 Особливості використання наявних рішень управління станом у веб-орієнтованих додатках на основі фреймворку React в ІС електронної комерції

Застосування механізмів управління станом у веб-додатках електронної комерції має свої особливості, пов’язані з високими вимогами до продуктивності, узгодженості даних і стабільності користувацької взаємодії. У цьому контексті поєднання локального, глобального та серверного стану вимагає ретельного проєктування архітектури та вибору оптимальних інструментів:

– поділ локального та серверного стану;

– оптимістичні оновлення;

– продуктивність і мінімізація повторних рендерів;

– серверний рендеринг, Search Engine Optimization (SEO) та гідрація стану;

– синхронізація інвентаря та транзакційність операцій;

– розмір бандла та час завантаження.

У системах електронної комерції дані, пов’язані з товарами, цінами, запасами чи замовленнями, належать до категорії серверного стану, який має бути консистентним і синхронізованим із базою даних. Для його обробки доцільно використовувати кешуючі бібліотеки, такі як React Query або Stale-While-Revalidate (SWR), що забезпечують автоматичне оновлення, повторні

запити та контроль за актуальністю інформаці. Водночас локальний стан доцільно зберігати для елементів інтерфейсу користувача – відкритих модальних вікон, стану форм або параметрів сортування. Такий поділ зменшує навантаження на центральне сховище та підвищує стабільність інтерфейсу.

Для покращення користувацького досвіду застосовується стратегія оптимістичного оновлення, коли зміни в інтерфейсі відображаються негайно, ще до отримання підтвердження від сервера. У випадку помилки система виконує відкат стану до попереднього значення. Такий підхід особливо ефективний для сценаріїв на кшталт додавання товарів у кошик або зміни кількості одиниць. Сучасні бібліотеки для роботи із серверним станом, зокрема TanStack Query, містять вбудовані механізми для реалізації цього патерну [18].

Надмірне оновлення компонентів є однією з найпоширеніших проблем при роботі з глобальними сторами, особливо при неефективному використанні Context API або неправильно спроєктованому Redux-сховищі. Для підвищення ефективності рекомендується застосовувати селектори, мемоізацію та розподіл стану на доменні підсховища, що зменшує кількість непотрібних оновлень компонентів та покращує загальну продуктивність застосунку [19].

Більшість сучасних e-commerce платформ використовують серверний рендеринг, наприклад у середовищі Next.js, для підвищення швидкості завантаження та покращення індексації сторінок пошуковими системами. У таких випадках стан додатка має коректно гідруватися на клієнтській стороні, а кешування серверного стану повинно бути узгоджене між середовищами рендерингу [20]. React-орієнтовані бібліотеки пропонують спеціальні патерни для синхронізації стану в умовах Server-Side Rendering (SSR), що забезпечує узгодженість даних між сервером і клієнтом.

У системах електронної комерції надзвичайно важливо забезпечити цілісність транзакцій та синхронізацію запасів у режимі реального часу. Під час оформлення замовлення або резервування товарів можуть виникати ситуації конкурентного доступу, особливо при одночасних запитах із кількох пристроїв чи вкладок. Для запобігання помилкам використовуються поєднання

оптимістичних патернів на клієнтському рівні та гарантованих підтверджень на стороні сервера. Такі підходи дозволяють забезпечити надійність транзакцій і коректне оновлення стану замовлень.

Вибір інструментів управління станом безпосередньо впливає на розмір збірки клієнтського коду. Для мобільних користувачів або при повільному мережевому з'єднанні надмірна вага бібліотек може істотно погіршити користувацький досвід. Тому важливо досягти балансу між функціональністю та легкістю рішення – наприклад, обираючи Zustand або Recoil для невеликих проєктів замість повнофункціонального Redux.

Впровадження механізмів управління станом у системах електронної комерції потребує врахування як технічних аспектів, так і бізнес-вимог. Ефективна інтеграція цих рішень забезпечує стабільність роботи системи, підвищує надійність бізнес-процесів і формує позитивний користувацький досвід.

1.4 Постановка задачі дослідження методів управління станом у веб-орієнтованих додатках на основі фреймворку React для ІС електронної комерції

Мета дослідження – здійснити систематичне порівняння та практичну оцінку методів управління станом у React-додатках з урахуванням специфічних вимог інформаційних систем електронної комерції, зокрема швидкодії інтерфейсу, консистентності даних, масштабованості, складності розробки та оптимізації розміру клієнтської частини.

Завдання дослідження:

– провести систематичний огляд існуючих підходів і бібліотек для управління станом (Hooks/Context, Redux Toolkit, MobX, Recoil, Zustand, XState) та класифікувати їхні властивості за архітектурними принципами, моделями

даних, підтримкою серверного рендерингу та наявністю механізмів оптимістичних оновлень;

– розробити набір типових сценаріїв для електронної комерції (додавання товару до кошика, багатокроковий процес оформлення замовлення з резервуванням товарів, фільтрація каталогу, оновлення даних у реальному часі) і створити експериментальні прототипи з використанням різних підходів до управління станом;

– здійснити вимірювання та порівняння ключових показників ефективності: часу реакції інтерфейсу, кількості повторних рендерів, споживання пам’яті та процесорних ресурсів, розміру клієнтського пакета, складності реалізації, читабельності коду та стійкості системи до помилок під час оновлення стану;

– розробити практичні рекомендації та архітектурні шаблони вибору технології управління станом залежно від масштабу інформаційної системи електронної комерції (малий інтернет-магазин, середня платформа, великий маркетплейс), а також визначити підхід до інтеграції глобального сховища стану з механізмами серверного кешування даних.

Методологія дослідження – поєднання систематичного аналізу наукових і технічних джерел, експериментального моделювання (створення прототипів і вимірювання показників продуктивності) та аналізу прикладних кейсів з реальної практики електронної комерції; у процесі буде використано інструменти профілювання React-додатків і засоби логування браузера для збору та аналізу метрик.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ОЦІНКИ РІШЕНЬ УПРАВЛІННЯ СТАНОМ У ВЕБ-ОРІЄНТОВАНИХ ДОДАТКАХ

2.1 Критерії оцінки

2.1.1 Продуктивність системи

Продуктивність системи – один із ключових критеріїв оцінки підходів до управління станом у веб-орієнтованих додатках. Вона визначає, наскільки швидко та ефективно інтерфейс реагує на зміни даних, а отже прямо впливає на користувацький досвід і бізнес-метрики. До складу критерію продуктивності доцільно включити такі підпоказники:

– час оновлення стану: час між викликом механізму оновлення та фактичним відображенням змін у DOM, практичне вимірювання – середній та перцентильні показники для набору типових дій у додатку;

– частота та обсяг повторних ререндерингів: кількість компонентів, що перерендерюються в результаті однієї дії, та загальна кількість ререндерів за одиницю часу, висока частота ререндерів негативно впливає на відчутну швидкість інтерфейсу та енергоспоживання на клієнті [19];

– споживання пам’яті: оперативна пам’ять, зайнята об’єктами стану, кешами та допоміжними структурами, для деяких бібліотек пам’ять може зростати значно більше, ніж для легковагих рішень;

– пропускна здатність і масштабованість під навантаженням: здатність системи обробляти велику кількість одночасних подій/запитів без деградації часу відповіді, включає оцінку поведінки при зростанні числа компонентів, кількості записів у сховище та розмірів даних;

– детермінованість і відновлюваність: наскільки передбачувано виконуються оновлення стану та як швидко система може відкотитися або відновити консистентний стан після збоїв.

Для оцінки використовуються комбінації профілювання у браузері, API продуктивності, а також автоматизовані навантажувальні тести, що повторюють

типові користувацькі сценарії у контрольованому середовищі. Результати слід усереднювати по кількох запусках з урахуванням «холодного» та «теплого» стану кешу [21].

У підсумку, комплексна оцінка продуктивності повинна поєднувати кількісні метрики і якісні аспекти та виконуватися на наборі репрезентативних сценаріїв, релевантних для e-commerce додатків.

2.1.2 Зручність розробки

Зручність розробки – критерій, що оцінює наскільки швидко, комфортно й безпечно команда може реалізовувати функціонал, підтримувати й розширювати кодову базу при використанні конкретного підходу до управління станом. Цей критерій включає як суб'єктивні фактори, так і об'єктивні: кількість шаблонного коду, наявність інструментів для налагодження та тестування.

Ключові підпоказники зручності розробки:

– простота та ясність API: наскільки природно й детерміновано бібліотека відображує модель домену; чи інтуїтивні назви методів/хуків; чи легко читати й передбачати побічні ефекти;

– обсяг шаблонного коду: кількість кроків і файлів, які потрібно створити для реалізації типової задачі;

– навчальна крива та поріг входу: час, необхідний новому розробнику, щоб почати продуктивно працювати з підходом; наявність навчальних матеріалів і прикладів;

– інструменти налагодження і візуалізації стану: наявність інструментів розробника, логування, можливостей перегляду дій, легкої інтеграції з Integrated Development Environment (IDE);

– підтримка типізації та якість типів: наскільки просто отримати коректні типи, чи допомагає типізація в запобіганні класу помилок;

– тестованість і модульність: наскільки легко писати unit/integration тести для бізнес-логіки і селекторів стану; чи можна тестувати логіку поза React компонентами;

– документація та екосистема: якість офіційної документації, прикладів, шаблонів; наявність middleware, persist-рішень, шаблонів архітектури;

– можливість рефакторингу й масштабування коду: наскільки просто змінювати структуру стану, розбивати на модулі, додавати нові фічі без розриву існуючих контрактів.

Методи вимірювання та інструменти:

– емпіричні тест-кейси: виконати типовий набір завдань і виміряти час розробки, кількість змінених файлів/рядків;

– опитування розробників і оцінка задоволеності.

Зручність розробки є не другорядним, а одним з головних критеріїв при виборі підходу до управління станом: вона безпосередньо впливає на швидкість впровадження функцій, якість коду, вартість підтримки та ризики технічного боргу.

2.1.3 Масштабованість і стійкість

Масштабованість у контексті механізмів керування станом – це здатність обраного підходу підтримувати ефективну роботу при збільшенні кількості компонентів, обсягу даних і рівня паралельності взаємодій без непропорційного зростання затримок або складності підтримки. Стійкість означає здатність системи зберігати консистентний стан і коректну роботу під навантаженням, при часткових відмовах або в умовах помилок.

Ключові показники масштабованості і стійкості:

– архітектурна модульність – наскільки просто розбити стан на автономні доменні підмодулі, ізольовані від інших частин системи; важлива для командної розробки та паралельного розвитку, для Redux існують усталені підходи, що допомагають контролювати складність у великих кодових базах;

– контроль цілісності і предиктивність оновлень – модель із централізованим потоком дій спрощує трасування й відлагодження, що позитивно впливає на стійкість при масштабуванні;

– розподіл навантаження і кількість підключень – як добре механізм дозволяє локалізувати оновлення та підтримувати велику кількість одночасних компонентів/підписників, підходи з мінімальною абстракцією часто мають менше шаблонів і можуть ефективно локалізувати оновлення, але потребують дисципліни в організації стану при рості проєкту;

– розширюваність інфраструктури – наявність зрілої екосистеми для керування побічними процесами, збереженням стану та засобів моніторингу є критичною для масштабних систем;

– відновлюваність і безпека даних – можливість відкотити зміни, відтворити послідовність дій, або обмежити доступ до певних операцій – важливі для фінансових/корпоративних систем; централізований і іммутабельний підхід полегшує реалізацію таких механізмів.

Методи вимірювання:

– функціональне тестування під навантаженням: аналіз часу обробки великої кількості одночасних дій, вимірювання перцентилів затримок під час масових оновлень стану;

– інструментальне тестування відтворення інтерфейсу: підрахунок кількості повторних відтворень компонентів під час типових сценаріїв збільшення кількості елементів користувацького інтерфейсу;

– аналіз структури коду: оцінювання залежностей між модулями стану, визначення рівня розширюваності системи – наскільки просто можна додати новий модуль без зміни наявних інтерфейсів;

– тестування на відмовостійкість: моделювання збоїв для перевірки механізмів відновлення стану системи.

Масштабованість і стійкість механізмів керування станом безпосередньо залежать від архітектурної дисципліни, прозорості потоків даних і зрілості супровідної екосистеми. Важливо поєднувати архітектурні принципи з систематичним тестуванням, щоб гарантувати надійність і відновлюваність стану на всіх рівнях застосунку.

2.1.4 Стійкість та керування постійним станом

Критерій «стійкість та керування постійним станом» охоплює здатність системи керування станом коректно зберігати та відновлювати дані між сесіями, забезпечувати надійну персистентність (на локальному носії або у базі даних), а також належну обробку помилок під час серіалізації та відновлення стану. Важливою складовою є також підтримка механізмів версіювання і міграції даних при зміні формату стану [22].

Основні показники:

– підтримка постійного збереження «з готових рішень» – наявність у бібліотеці або фреймворку стандартних засобів для збереження і автоматичного відновлення стану, а також елементів для відкладеного відтворення інтерфейсу до завершення процесу відновлення;

– гнучкість у виборі сховищ і масштабуванні обсягів даних – можливість використовувати різні сховища та враховувати вплив вибору сховища на швидкодію;

– обробка помилок під час серіалізації та відновлення – наявність засобів для виявлення пошкоджених або несумісних даних, а також безпечного відновлення після таких помилок, для цього можуть використовуватись функції резервного відновлення або спеціальні обробники подій;

– синхронізація між вкладками та усунення конфліктів – здатність системи підтримувати узгоджений стан між кількома відкритими вікнами чи вкладками одного застосунку, а також правильно реагувати на конфлікти при одночасному оновленні даних;

– безпека та робота з конфіденційними даними – наявність засобів для запобігання запису у сховище конфіденційної інформації або можливість її попереднього шифрування, важливою функцією також є очищення збережених даних після виходу користувача із системи.

Методи оцінювання та тестування стійкості:

– перевірка коректності відновлення стану – попереднє збереження контрольного набору даних, перезапуск застосунку і перевірка повноти відновлення стану;

– імітація пошкоджених або застарілих даних – навмисне спотворення збереженого стану для перевірки коректності реакції системи, наприклад виконання резервного відновлення або оновлення формату;

– тестування обмеження обсягу сховища – моделювання перевищення допустимого розміру локального сховища для оцінки поведінки системи в умовах нестачі пам’яті;

– перевірка одночасних оновлень у кількох вкладках – симуляція паралельного запису даних у різних вікнах та аналіз розв’язання конфліктів;

– оцінювання продуктивності – вимірювання часу зчитування і запису великих обсягів даних для визначення впливу обраного сховища на швидкодію системи.

Якісне керування постійним станом є важливою складовою стійкості та надійності сучасних вебзастосунків. Наявність перевірених механізмів персистентності, гнучкість у виборі сховищ і контроль над процесами відновлення даних визначають здатність системи зберігати коректний стан у довготривалій роботі та під час непередбачуваних збоїв.

2.2 Метрики та інструменти вимірювання

2.2.1 Час оновлення стану

Показник часу оновлення стану визначається як інтервал часу між моментом ініціалізації зміни і моментом, коли ці зміни стають видимими в документі – тобто фактичним завершенням фази commit у циклі рендерингу React. Ця метрика характеризує реальний час реакції інтерфейсу на дію користувача або внутрішню подію і прямо впливає на користувацький досвід. Тривалість оновлення, що перевищує поріг сприйняття, призводить до відчуття «затримки» або «пригальмовування» інтерфейсу, що може знизити задоволеність користувачів і конверсію в e-commerce-застосунках.

Для точних і відтворюваних вимірювань доцільно поєднувати два підходи:

– профілювання React та інструменти розробника браузера – дозволяють відстежувати час рендерингу компонентів і момент commit [22], це корисно для локалізації «важких» компонентів та візуального аналізу вкладених оновлень [23];

– API продуктивності браузера – забезпечує точні часові мітки в мілісекундах та зручний механізм для автоматизованого збору даних у коді, цей підхід зручний для серійних прогонів, тестування на різних пристроях і збереження результатів для подальшого статистичного аналізу.

Нижче наведено спрощений приклад використання API продуктивності в компоненті React.

Лістинг 2.1 – Приклад використання API продуктивності в компоненті React

```
import { useLayoutEffect, useRef } from 'react';
export function useStateUpdateMeasurement(label) {
  const idRef = useRef(0);
```

```
      const startNameRef = useRef(null);
      function markStart() {
        const id = ++idRef.current;
        const startName = `${label}-start-${id}`;
        performance.mark(startName);
        startNameRef.current = startName;
        return startName;
      }
      function markEnd() {
        const startName = startNameRef.current;
        if (!startName) return null;
        const endName = `${startName}-end`;
        performance.mark(endName);
        const measureName = `${label}-measure-
${startName.split('-').pop()}`;
        performance.measure(measureName, startName, endName);
        const measures =
performance.getEntriesByName(measureName);
        return measures;
      }
      useLayoutEffect(() => {
      });
      return { markStart, markEnd };
    }
```

Для отримання репрезентативних даних слід дотримуватися наступних правил:

– вибір сценаріїв: локальні оновлення, оптимістичні оновлення, масові оновлення, груповані операції;

– набір пристроїв і середовищ: тестувати на різних класах пристроїв – потужний десктоп, слабший ноутбук, мобільний пристрій; проводити тести у різних браузерах;

– контроль фонових факторів: виконувати виміри в «чистому» середовищі, фіксувати стан кешу й умови навантаження процесора.

Результати доцільно представляти таблицями з основними статистиками та графіками: ящикові діаграми або гістограми. Такі візуалізації допомагають виявити асиметрію розподілу, наявність викидів і характерні патерни, що підказують джерела затримок.

Час оновлення стану є ключовою кількісною метрикою при оцінці підходів до управління станом у React-застосунках. Для обґрунтованого

порівняння рішень потрібно поєднати інструментальні вимірювання з профілюванням React і провести статистично достовірні експерименти на репрезентативних сценаріях і пристроях.

2.2.2 Частота перерендерингів компонентів

Показник частота повторних рендерів компонентів характеризує кількість виконаних операцій рендерингу окремих компонентів у відповідь на зміни стану, властивостей або контексту протягом певного інтервалу часу або у відповідь на одну дію користувача [24]. Ця метрика важлива, оскільки надмірна кількість повторних рендерів може призводити до зайвого використання процесора, збільшення споживання пам’яті та погіршення відчутної швидкодії інтерфейсу, особливо у випадках великої кількості одночасних компонентів або на слабких клієнтських пристроях.

Для повноти оцінки рекомендується збирати такі підпоказники:

– кількість повторних рендерів на компонент за одиницю дії;

– сукупна кількість повторних рендерів за сценарій;

– частка «непотрібних» рендерів – рендери, що відбулися без зміни вхідних даних або релевантного стану компонента;

– розподіл рендерів по ієрархії компонентів – які компоненти генерують найбільший відсоток рендерів;

– періодичність і тривалість піків рендерів – коли й чому виникають сплески.

Для вимірювання частоти рендерів доцільно використовувати комбінацію інструментів:

– React Profiler – дозволяє бачити час рендерингу та кількість викликів рендер-функцій компонентів під час запису сесії профілювання, профілювання

зберігає трасування, яке потім можна аналізувати для виявлення «важких» компонентів;

– вбудовані лічильники в коді – прості й відтворювані методи: користувацькі хуки для підрахунку кількості рендерів і логування або збір метрик у масив для подальшої агрегації;

– API продуктивності – дозволяють зберігати події й зіставляти їх із записами рендерів, особливо корисні для автоматизованих тестів.

Нижче наведено два утилітні приклади: хук для підрахунку кількості рендерів і хук для логування змін властивостей компонента.

Лістинг 2.2 – Хук для підрахунку кількості рендерів і хук для логування змін властивостей компонента

```
import { useRef, useEffect } from 'react';
export function useRenderCount(name = 'Component') {
  const countRef = useRef(0);
  countRef.current += 1;
  useEffect(() => {
    console.log(`${name} render #${countRef.current}`);
  });
  return countRef.current;
}
import { useRef, useEffect } from 'react';
import isEqual from 'lodash.isequal';
export function useWhyDidUpdate(name, props) {
  const previousProps = useRef();
  useEffect(() => {
    if (previousProps.current) {
      const allKeys =
Object.keys({ ...previousProps.current, ...props });
      const changes = {};
      allKeys.forEach(key => {
        if (!isEqual(previousProps.current[key], props[key]))
{
          changes[key] = {
            from: previousProps.current[key],
            to: props[key]
          };
        }
      });
      if (Object.keys(changes).length) {
        console.log('[why-did-update]', name, changes);
      }
```

```
    }
    previousProps.current = props;
  });
}
```

Ці хуки дають змогу оперативно ідентифікувати компоненти з великою кількістю повторних рендерів та один із перших кроків у виявленні причин – зміну властивостей або контекстних значень.

Для репрезентативного оцінювання частоти повторних рендерів необхідно:

– визначити сценарії – вибрати типові користувацькі сценарії: навігація, фільтрація каталогу, додавання у кошик, масові оновлення списків, відкриття модальних вікон; для кожного сценарію описати початковий стан і кроки;

– вибрати метрики збору – кількість рендерів на компонент, частка непотрібних рендерів, сукупний час, кількість рендерів на дію;

– зберігати й агрегувати дані – логувати виміри у форматі JavaScript Object Notation (JSON) для подальшого аналізу;

– контролювати змінні – відключати зайві розширення, фіксувати навантаження на центральний процесор, однакові умови мережі, використовувати «чисті» профілі браузера для відтворюваності.

При інтерпретації слід врахувати контекст: у деяких випадках часті рендери можуть бути виправданими, однак для більшості взаємодій у e-commerce надмірні рендери – ознака неефективної організації стану або неправильної структури компонентів.

Результати слід представляти у вигляді таблиць та графіків: стовпчикові діаграми з кількістю рендерів за компонентами, ящикові діаграми для розподілу рендерів за сценаріями, теплові карти ієрархії компонентів з інтенсивністю рендерів.

Частота повторних рендерів є критичною метрикою продуктивності React застосунків, особливо в умовах e-commerce, де швидкість інтерфейсу впливає на конверсію та задоволення користувачів. Систематичне вимірювання

повторних рендерів у поєднанні з аналізом причин дозволяє цілеспрямовано знижувати навантаження та підвищувати ефективність застосунку.

2.2.3 Споживання пам’яті

Показник споживання пам’яті оцінює обсяг оперативної пам’яті, яку займають об’єкти додатка під час виконання: об’єкти стану, кеші, буфери, DOM вузли та супутні структури [25]. Для клієнтських односторінкових застосунків важливими є два аспекти: постійне споживання – кількість пам’яті під час нормальної роботи, та пікове споживання – максимальний обсяг пам’яті під час навантаження або складних операцій. Невдале керування пам’яттю призводить до зростання затримок, частого очищення пам’яті збирачем сміття, підвищеного споживання енергії та потенційних падінь у слабких пристроях.

Для повноти оцінки слід фіксувати:

– середнє і пік споживання пам’яті під час сценарію;

– темп росту пам’яті під час тривалого навантаження – дозволяє виявити витоки пам’яті;

– кількість від’єднаних DOM-вузлів і їхній утримуваний розмір;

– частота й тривалість очищень пам’яті – вплив на продуктивність.

Рекомендується поєднувати інструменти браузера з програмними вимірами:

– інструменти розробника;

– автоматизовані інструменти й аудити;

– вбудовані API й лічильники;

– моніторинг у реальному середовищі.

Приклади практичних вимірювань:

– Для браузеру наведені у лістингу В.1

– Для фреймворку Node.js у лістингу В.2

Щоб отримати відтворювані результати, необхідно дотримуватися такої процедури:

– початкове встановлення бази: перезапустити браузер/сервер, очистити кеш, закрити зайві вкладки; зафіксувати початкове значення пам’яті;

– визначити сценарії: короткі операції, середні, важкі;

– запуск серій прогонів: для кожного сценарію виконати багаторазові прогони у «теплому» та «холодному» стані, записуючи значення пам’яті з інтервалом;

– знімки купи: робити знімки купи на початку, під час і в кінці сценарію; порівнювати знімки для виявлення незвільнених об’єктів;

– тривале навантаження: проводити тест тривалістю 10-60 хвилин для виявлення повільних витоків пам’яті; обчислювати темп росту;

– фіксація Garbage Collector подій (GC-подій): у записі продуктивності звертати увагу на події збирання сміття й їхній вплив на латентність.

При аналізі знімків необхідно звертати увагу на:

– Retained size – обсяг пам’яті, який буде звільнений лише після видалення даного об’єкта; великі retained-size об’єкти часто вказують на джерела витоку;

– Shallow size – розмір самого об’єкта;

– замкнені змінні – функції або замикання, які утримують великі масиви або об’єкти.

Споживання пам’яті – одна з ключових нефункціональних метрик для оцінки якості управління станом у веб-застосунках. Систематичні вимірювання у поєднанні з інструментальним аналізом дозволяють виявити та усунути витоки, оптимізувати кеші та архітектуру додатка. Для e-commerce-систем, де стабільність і передбачуваність роботи на різних пристроях критичні, контроль пам’яті має стати обов’язковою частиною процесу розробки, тестування й моніторингу.

2.2.4 Оцінка досвіду розробника

Оцінка досвіду розробника відображає, наскільки швидко, комфортно й безпечно команда може розробляти, тестувати та підтримувати функціонал з використанням конкретного підходу до управління станом і суміжних інструментів. Developer Experience (DX) безпосередньо впливає на швидкість реалізації функцій, наявність помилок у продукції, вартість підтримки та мотивацію команди.

Для повної картини варто розглядати як об’єктивні, так і суб’єктивні показники. Перелік об’єктивних показників наведено нижче:

– час реалізації типового завдання: час від початку роботи до першого робочого коміту функціоналу або до закриття завдання в системі відстеження;

– час на налаштування середовища: час, необхідний новому розробнику, щоб запустити проєкт локально;

– розмір шаблонного коду: кількість файлів і рядків коду, необхідних для реалізації типового CRUD-випадку (CRUD – Create, Read, Update, Delete);

– час побудови й тестування: тривалість виконання збірки та тестів у системі безперервної інтеграції;

– кількість відкритих питань у процесі розробки: дефекти, питання архітектури, запити на додаткові пояснення.

Суб’єктивні показники наведено нижче:

– задоволеність API/інструментами: наскільки інтуїтивний та зручний інтерфейс бібліотеки/фреймворку;

– якість документації та прикладів.;

– поріг входження: час, потрібний новому розробнику, щоб стати продуктивним із даним підходом.

Для порівняння підходів до управління станом потрібно створити набір типових задач:

– типове CRUD-завдання: додати новий тип даних, реалізувати створення, редагування, видалення та відображення списку;

– синхронізація між вкладками: реалізувати збереження стану кошика у локальному сховищі й синхронізацію між двома вкладками;

– міграція стану: змінити структуру сховища і виконати міграцію з мінімальними змінами у коді.

Оцінка досвіду розробника – багатовимірний процес, що поєднує кількісні метрики продуктивності розробки та якісні оцінки задоволеності й зручності. Результати аналізу повинні стати підґрунтям для практичних рекомендацій щодо вибору інструментів і процедур у проєктах електронної комерції.

3 МЕТОДИКА РОЗРОБКИ ТЕСТОВИХ ВЕБ-ДОДАТКІВ ТА ОЦІНКИ РІШЕНЬ УПРАВЛІННЯ СТАНОМ У E-COMMERCE

3.1 Розробка тестового застосунку для моделювання функціоналу та оцінки ефективності стейт-менеджерів

Для проведення дослідження ефективності різних методів управління станом у веб-додатках було створено тестовий веб-застосунок на основі фреймворку React. Метою розробки було забезпечити платформу для порівняння продуктивності та стабільності рішень для управління станом, у контексті електронної комерції. Для зручності та швидкої розробки інтерфейсу використовувалася бібліотека Ant Design, яка дозволяє створювати сучасний, оптимізований та зрозумілий користувацький інтерфейс із готовими компонентами, такими як таблиці, кнопки, модальні вікна та форми.

Сторінка товарів (рисунок 3.1) є ключовим компонентом застосунку, оскільки саме вона відображає основний функціонал e-commerce системи. Користувач може переглядати перелік продуктів, отриманих з тестового Node.js API, у вигляді таблиці або карток товарів. Для покращення продуктивності та оцінки реактивності стану реалізована пагінація, що дозволяє завантажувати обмежену кількість товарів на сторінку. Кожен товар містить основні характеристики: назву, категорію та наявність на складі, а також інтерактивні кнопки для додавання у кошик та перегляду детальної інформації.

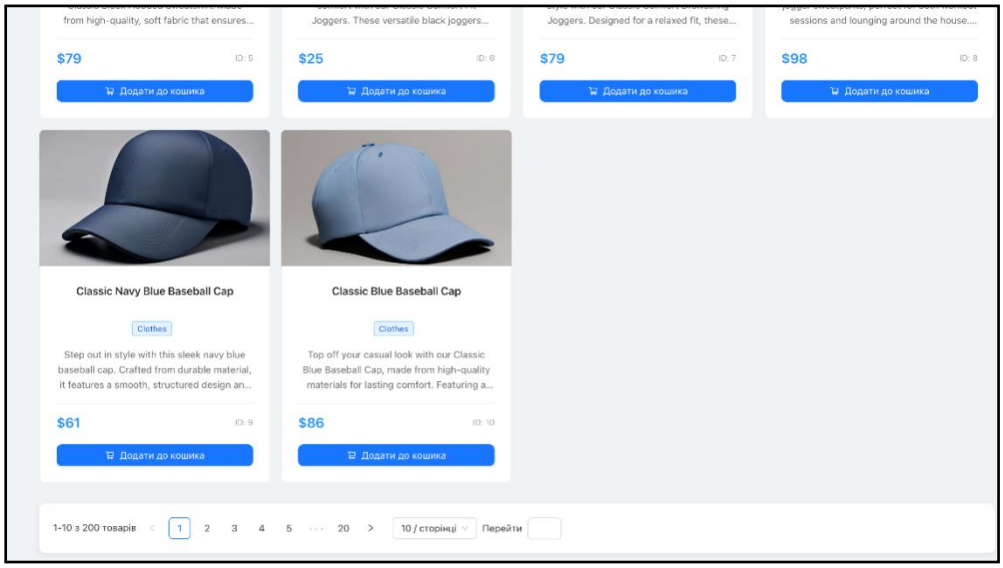

Рисунок 3.1 – Інтерфейс товарної сітки та пагінації у додатку

Кошик реалізовано як окремий компонент із використанням Drawer з Ant Design (рисунок 3.2). Користувач може додавати товари у кошик з будь-якої сторінки, збільшувати або зменшувати кількість товару та видаляти товари з кошика. Стан кошика зберігається між переходами сторінок, що дозволяє тестувати стабільність та швидкодію різних рішень управління станом при одночасній взаємодії з багатьма компонентами. Підсумкова ціна та кількість товарів у кошику оновлюються динамічно, що демонструє реактивність системи.

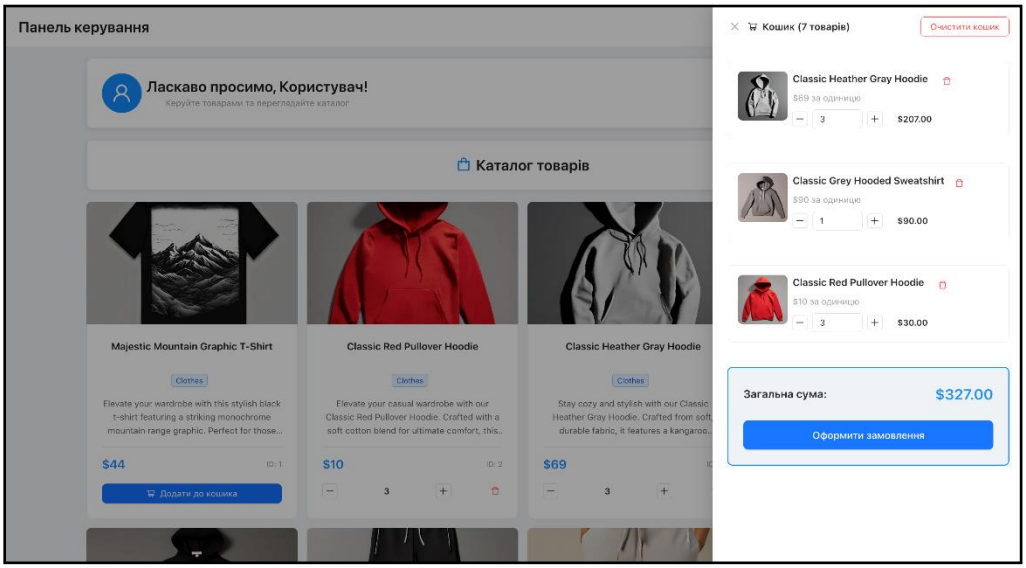

Рисунок 3.2 – Інтерфейс товарного кошика у додатку

Сторінка авторизації користувача (рисунок 3.3) забезпечує базову перевірку ідентифікаційних даних. Користувач вводить email та пароль у форму, після чого відбувається запит до тестового API. У разі успішної авторизації стан користувача зберігається у глобальному сховищі стану та використовується на всіх сторінках застосунку. У разі помилки відображається повідомлення, що інформує користувача про некоректні дані. Цей функціонал дозволяє оцінювати, як методи управління станом обробляють асинхронні запити та оновлення стану користувача.

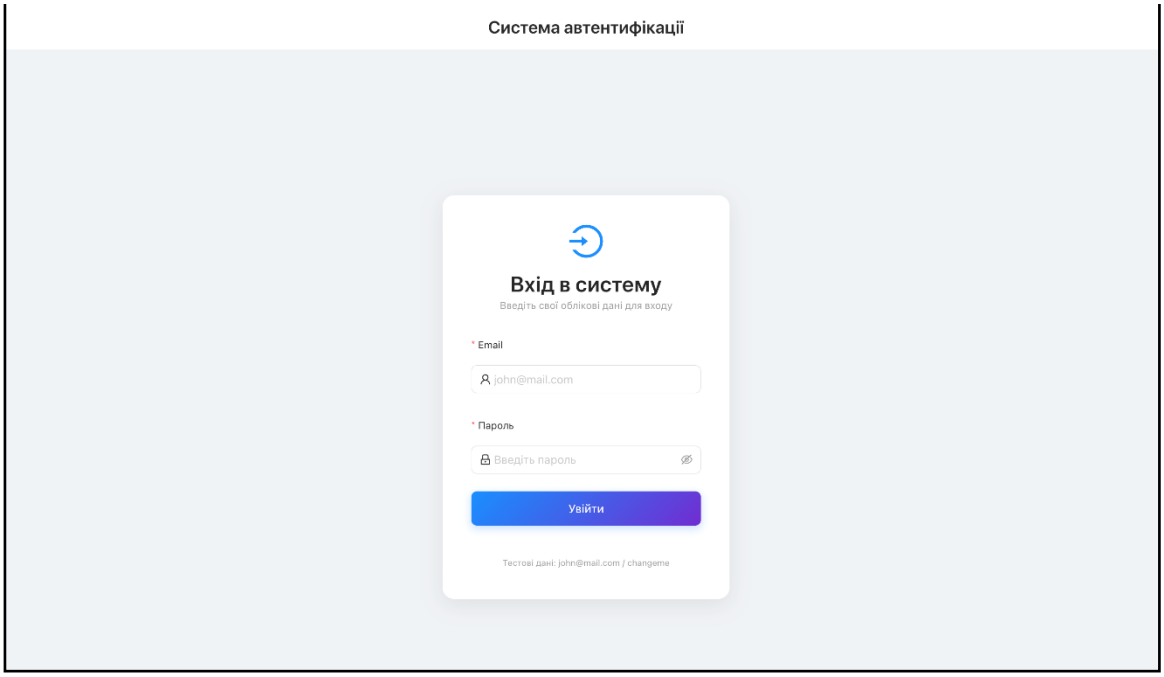

Рисунок 3.3 – Інтерфейс сторінки авторизації

Для взаємодії із сервером використано Axios, що забезпечує асинхронні запити до тестового Node.js API [26]. Endpoints включають отримання списку товарів, додавання товарів у кошик та авторизацію користувача. Така інтеграція дозволяє оцінювати швидкість оновлення стану та реактивність інтерфейсу при різних сценаріях використання, включно з багатократними запитами та одночасним оновленням стану декількох компонентів.

Застосунок також оптимізовано для кращого користувацького досвіду: використано lazy loading для компонентів сторінок, що зменшує час

початкового завантаження, а компоненти побудовані модульно, що дозволяє легко підміняти методи управління станом для тестів. Інтерфейс адаптовано під різні роздільні здатності екрану, що дозволяє перевіряти, як стабільність стану впливає на адаптивність та візуальну цілісність інтерфейсу.

Особливу увагу під час розробки приділено модульній структурі проєкту. Усі основні функціональні частини – товари, кошик, авторизація, пагінація та API-запити – були реалізовані як незалежні модулі з чітко визначеними інтерфейсами та областями відповідальності. Завдяки цьому стало можливим створення еквівалентних копій застосунку, у яких змінювалася лише логіка управління станом. Для кожного з досліджуваних рішень, була створена окрема версія проєкту з однаковим інтерфейсом і структурою компонентів. Такий підхід забезпечив чесне порівняння методів управління станом, оскільки всі інші фактори (візуальна частина, API, обсяг даних, логіка компонентів) залишалися незмінними.

У підсумку, розроблений тестовий веб-застосунок забезпечує повний функціонал для оцінки ефективності методів управління станом у веб-додатках електронної комерції, дозволяючи аналізувати швидкодію, стабільність стану та взаємодію користувача з різними елементами інтерфейсу. Інтерфейс застосунку готовий до візуалізації у дипломній роботі за допомогою скріншотів сторінок та компонентів, що ілюструє реалізацію ключових функцій.

3.2 Збір та аналіз даних для оцінки ефективності рішень управління станом

Для об’єктивної оцінки ефективності різних підходів до управління станом у веб-застосунках було проведено серію експериментів із тестовим e-commerce застосунком, описаним у попередньому підрозділі. У процесі тестування фіксувалися кількісні показники, що відображають швидкодію,

стабільність, використання ресурсів та масштабованість застосунку під час типових користувацьких дій.

Зібрані дані систематизуються у вигляді таблиць, де для кожного рішення наведено результати вимірювань за однакових умов експерименту. Для кожного сценарію виконувалася серія тестів, після чого обчислювалися середні значення, пікові навантаження та кількість повторних рендерингів компонентів. Усі тести проводилися в однаковому середовищі браузера при незмінному обсязі даних і структурі компонентів, щоб забезпечити порівняльну достовірність.

Час оновлення стану (ms), наведено у таблиці 3.1, відображає середній час, необхідний для оновлення стану після виконання певної дії користувача. Для вимірювання обрано базовий сценарій – додавання одного товару до кошика (single item). Після натискання кнопки “Додати в кошик” фіксується проміжок часу між викликом дії (dispatch/setState) та фактичним оновленням інтерфейсу (відображення зміненого стану). Менше значення свідчить про вищу швидкодію системи управління станом.

Таблиця 3.1 – Час оновлення стану (ms)

Метод	mean (ms)	p50 (ms)	p95 (ms)
Redux	0.47	0.40	0.95
Zustand	0.28	0.24	0.60
Context API	0.95	0.80	2.10
MobX	0.33	0.27	0.70
XState	0.55	0.48	1.30

Частота ререндерингів (на 1 дію), наведено у таблиці 3.2, показує скільки разів React-компоненти були повторно відрендерені під час виконання однієї дії користувача. Надлишкові ререндеринги можуть свідчити про неефективну роботу стану або неправильну структуру підписки компонентів на зміни.

Менша кількість ререндерингів вказує на більш оптимальне оновлення інтерфейсу та зниження навантаження на браузер.

Таблиця 3.2 – Час оновлення стану (ms)

Сценарій	Метод	avg рендерів на дію
Додавання в кошик	Redux	3.2
Додавання в кошик	Zustand	2.1
Додавання в кошик	Context API	8.7
Додавання в кошик	MobX	1.8
Додавання в кошик	xState	3.2
Масове оновлення (1000)	Redux	450
Масове оновлення (1000)	Zustand	220
Масове оновлення (1000)	Context API	980
Масове оновлення (1000)	MobX	12
Масове оновлення (1000)	xState	45

Споживання пам’яті (JS Heap, MB), наведено у таблиці 3.3. Для кожного варіанта управління станом фіксується середнє та пікове споживання пам’яті під час виконання дій. Цей показник дозволяє оцінити, наскільки ефективно фреймворк або бібліотека працює з пам’яттю при динамічному оновленні стану.

Таблиця 3.3 – Споживання пам’яті

Сценарій / Фреймворк	Redux	Zustand	Context API	MobX	XState
Стандартна сесія (1 хв) Average / Max (JS Heap, MB)	~ 45 / ~ 55	~ 42 / ~ 50	~ 48 / ~ 60	~ 40 / ~ 52	~ 43 / ~ 53
Тривале навантаження (5 хв). Темп росту (MB в 1 хвилину)	+0.30	+0.20	+0.55	+0.25	+0.28

Надмірне зростання JS Heap може свідчити про витоки пам’яті або неоптимальну роботу механізмів збереження даних.

Пропускна здатність, наведено у таблиці 3.4 – здатність системи коректно обробляти одночасні зміни стану при зростанні кількості даних або користувацьких дій. У межах тестування цей показник вимірювався під час симульованого додавання кількох десятків товарів у кошик поспіль. Аналіз дозволяє оцінити, як кожне рішення реагує на збільшення навантаження: чи зберігає стабільність швидкодії, чи не виникають затримки або втрати стану.

Таблиця 3.4 – Пропускна здатність

Метод	Throughput (actions/sec)
Redux	18 000–22 000
Zustand	28 000–35 000
Context API	7 000–10 000
MobX	22 000–30 000
XState	14 000–18 000

Стійкість, відновлюваність та персистентність, наведено у таблиці 3.5 – критерії відображає надійність збереження стану після оновлення сторінки або при повторному відкритті застосунку. Перевіряється здатність системи підтримувати дані кошика та авторизації користувача після перезавантаження.

Таблиця 3.5 – Стійкість, відновлюваність та персистентність

Метод	Час відкату / rollback (ms)
Redux	2.4 ms
Zustand	1.6 ms
Context API	8.0 ms
MobX	2.7 ms
XState	6.0 ms

Рішення, що забезпечують механізм локального збереження (наприклад, через localStorage або middleware), демонструють вищий рівень стійкості для кінцевого користувача.

3.3 Визначення оптимальних рішень та формулювання висновків

Для визначення оптимального підходу до управління станом було проведено інтегральний аналіз за всіма критеріями: швидкодією оновлення (mean, p50, p95), частотою ререндерингів, використанням пам'яті, пропускною здатністю та відновлюваністю. Оскільки кожен із показників має різний вплив на кінцеву продуктивність, доцільно застосувати узагальнену метрику ефективності.

Для кожного підходу розраховано індекс ефективності за формулою:

$$Score = W_1 S_{speed} + W_2 S_{renders} + W_3 S_{memory} + W_4 S_{throughput} + W_5 S_{rollback}$$

де $Score$ – це індекс ефективності, обчислений як зважена сума нормованих метрик продуктивності

S_I – нормований показник (0…1, де 1 – найкраще);

W_I – вагові коефіцієнти;

S_{speed} – нормований показник швидкості;

S_{memory} – нормований показник використання пам'яті;

$S_{throughput}$ – нормований показник пропускної здатності;

$S_{rollback}$ – нормований показник відновлюваності.

Оскільки для реальних веб-додатків критичною є швидкодія, пропускна здатність і ререндеринги, обрано ваги:

– швидкодія – 0.30;

– ререндеринги – 0.25;

– пропускна здатність – 0.20;

– пам’ять – 0.15;

– відновлюваність – 0.10.

Для показників, де менше значення є кращим, усі значення метрик було нормовано за нижченаведеною формулою:

$$S = \frac{minvalue}{currentvalue} \quad ,$$

де S – нормоване значення показника;

$minvalue$ – мінімальне значення показника серед усіх досліджуваних варіантів;

$currentvalue$ – поточне значення показника для конкретного варіанта.

Для показників, де більше значення є кращим, усі значення метрик було нормовано за нижченаведеною формулою:

$$S = \frac{currentvalue}{minvalue} \quad .$$

У цьому випадку більші значення показника приводять до більших значень S, що відображає кращу якість. Результати наведені у таблиці 3.6.

Таблиця 3.6 – Результати інтегральної оцінки

Метод	Швидкодія	Ререндеринги	Пропускна здатність	Пам’ять	Відновлюваність
Zustand	1.00	0.49	0.92	1.00	1.00
MobX	0.85	1.00	1.00	0.86	0.59
Redux	0.59	0.22	0.82	0.63	0.67
XState	0.51	0.008	0.86	0.51	0.27
Context API	0.29	0.002	0.66	0.28	0.20

На основі отриманих даних була побудована гістограма (рисунок 3.4), що дозволяє наочно порівняти продуктивність окремих підходів та їхніх комбінацій. Візуалізація робить відмінності між методами більш очевидними, демонструючи як загальні тенденції, так і локальні переваги конкретних рішень. Такий графічний аналіз допомагає швидше і точніше інтерпретувати результати, підтверджуючи перевагу найбільш ефективних методів.

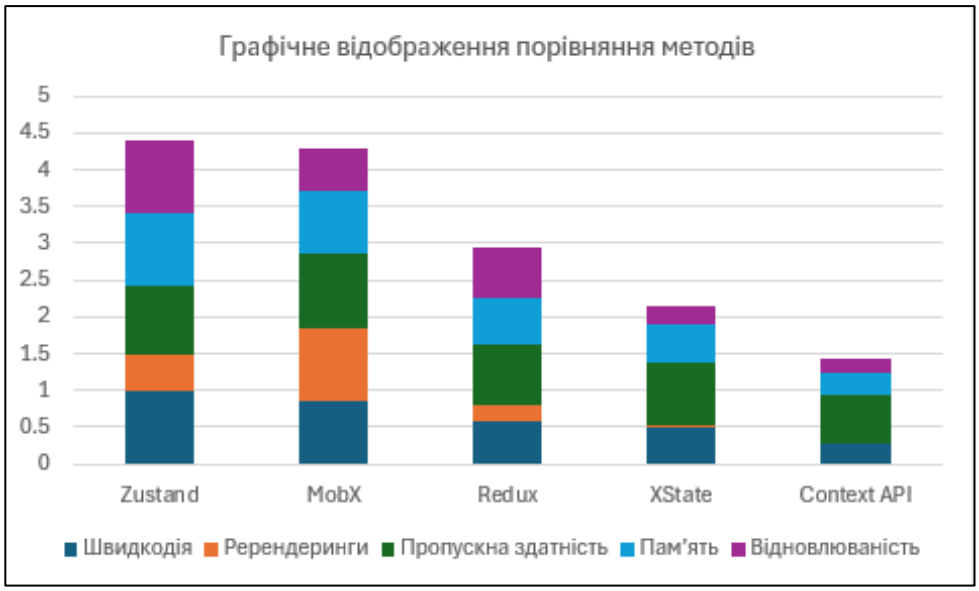

Рисунок 3.4 – Порівняльна гістограма методів

Таблиця 3.7 – Підсумкові результати інтегральної оцінки

Метод	Підсумковий Score
Zustand	0.90
MobX	0.83
Redux	0.54
XState	0.32
Context API	0.18

На основі інтегральної оцінки результат якого наведений у таблиці 3.7, експериментальних даних та нормованого порівняння можна зробити висновок, що Zustand є найбільш універсальним та продуктивним рішенням для сучасних

веб-застосунків, тоді як MobX забезпечує найкращу реактивність та мінімальні ререндеринги при роботі з великими обсягами даних.

Для складних e-commerce застосунків, де поєднуються часті оновлення інтерфейсу, масові операції з даними та висока потреба у стабільній персистентності стану, оптимальним є комбінований метод Zustand + MobX, що дозволяє досягти максимальної продуктивності системи.

4. АПРОБАЦІЯ ЗАПРОПОНОВАНОЇ КОМБІНОВАНОЇ АРХІТЕКТУРИ ZUSTAND / MOBX

4.1. Архітектурна модель тестового веб-додатка з комбінованим управлінням станом

Тестовий веб-застосунок, повторює інтерфейс та функціонал інших тестових додатків які використовуються для дослідження. Додаток має типовий e-commerce інтерфейс і складається з трьох основних функціональних модулів: авторизації користувача, каталогу товарів та модуля управління кошиком. Структура додатка побудована за компонентно-орієнтованою архітектурою React, де окремі частини інтерфейсу мають власні зони відповідальності та взаємодіють із різними рівнями стану відповідно до їхніх потреб.

У контексті комбінованої архітектури оптимальний розподіл виглядає так: Zustand використовується для глобального стану, а MobX – для локальних реактивних моделей каталогу та списків товарів. Глобальний стан включає дані авторизації, параметри інтерфейсу (active modal, theme, фільтри), структуру кошика та загальні метадані застосунку. Ці дані змінюються порівняно часто й мають бути доступними всім частинам додатка, тому застосування легковагового та швидкого Zustand є найбільш виправданим. Він забезпечує миттєві оновлення, селектори вузької спрямованості та мінімальну кількість ререндерингів при зміні глобальних значень, завдяки чому кошик та модуль авторизації працюють стабільно та передбачувано.

Модуль каталогу товарів побудований на MobX, оскільки саме тут відбуваються найважчі з обчислювальної точки зору операції: сортування, фільтрація, зміна властивостей окремих товарів, оновлення цін, кількості, статусів наявності тощо. MobX забезпечує fine-grained реактивність, коли зміна навіть однієї властивості одного товару не викликає повторних ререндерингів усієї сторінки. Завдяки цьому користувач може працювати з великими наборами товарів без втрати продуктивності, а каталог реагує на зміну стану

плавно та без затримок. Каталог реалізований як окрема реактивна модель у MobX, яка містить масив товарів, обчислювані значення (computed) для сортування та фільтрації, а також окремі action-методи для модифікацій.

Взаємодія модулів здійснюється за допомогою мінімального API: каталог MobX лише надсилає дані про вибраний товар, тоді як глобальний Zustand-кошик відповідає за збереження цього вибору, перерахунок вартості та управління кількістю. Такий поділ відповідальності забезпечує структурну чіткість: MobX відповідає за реактивність та оптимізацію роботи з великими масивами даних, тоді як Zustand – за швидке та легке управління ключовими глобальними об’єктами. У результаті застосунок отримує максимально ефективну архітектуру, де кожен інструмент використовується саме в тій частині логіки, для якої він технологічно найкраще підходить.

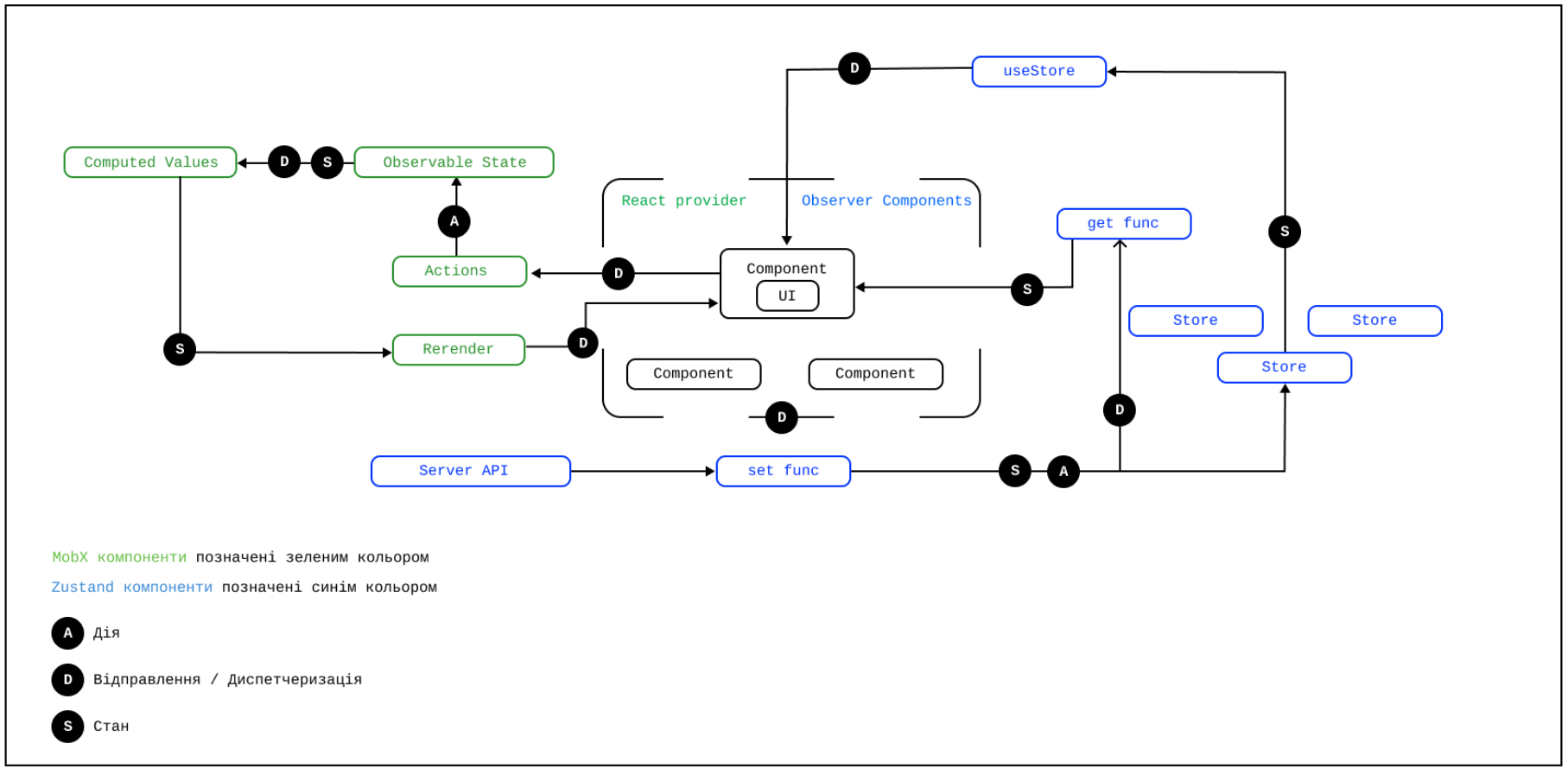

Рисунок 4.1 – Принцип роботи управління станом комбінованого методу

Оптимальність поєднання Zustand та MobX пояснюється тим, що ці дві технології компенсують слабкі місця одна одної та покривають різні рівні логіки веб-застосунку. На рисунку 4.1 продемонстрований новий принцип роботи взаємодії комбінованого методу. Zustand забезпечує надзвичайно швидку роботу з глобальним станом, де важливі мінімальні накладні витрати, прості оновлення і висока пропускна здатність – саме ті операції, які домінують у більшості e-commerce сценаріїв (додавання в кошик, зміни UI, робота з авторизацією). Завдяки селекторам і відсутності необхідності глибокого копіювання об’єктів бібліотека демонструє найкращі часові показники оновлення стану та низьке споживання пам’яті. Це робить Zustand оптимальною основою для глобального шару даних, який часто оновлюється, але повинен залишатися легким та передбачуваним.

У той же час MobX доповнює систему там, де необхідна реактивність та ефективна робота з великими або складно структурованими наборами даних. Завдяки fine-grained реактивності зміна одного поля в одному елементі списку не викликає оновлення всього списку, як це відбувається у Redux чи Context API [27]. Це критично для каталогів товарів, фільтрації, сортування, реалізації складних обчислюваних властивостей. MobX дозволяє оновлювати інтерфейс точно і точково, що забезпечує рекордно низьку кількість ререндерингів навіть при масових змінах [28]. Таким чином, у поєднанні Zustand створює швидку й мінімалістичну основу, а MobX додає реактивність і оптимізацію для важких локальних структур. Разом ці інструменти формують архітектуру, що забезпечує найвищу продуктивність, масштабованість та економію ресурсів у реальних веб-застосунках [29].

4.2. Оцінка продуктивності моделі тестового веб-додатка з комбінованим управлінням станом

Було проведено повторну оцінку продуктивності тестового веб-додатка, у якому реалізовано комбінований метод до управління станом на основі Zustand та MobX. Основною метою аналізу є визначення того, як поєднання цих інструментів впливає на швидкість оновлення стану порівняно з їх окремим використанням.

Таблиця 4.1 – Час оновлення стану (ms)

Метод	mean (ms)	p50 (ms)	p95 (ms)
Zustand	0.28	0.24	0.60
MobX	0.33	0.27	0.70
Zustand + MobX	0.27	0.25	0.63

Отримані результати по тестуванню часу оновлення (таблиця 4.1) демонструють, що комбіноване використання Zustand і MobX забезпечує найменший середній час оновлення стану (0.27 ms), перевершуючи як окреме застосування Zustand (0.28 ms), так і MobX (0.33 ms). При цьому значення p50 і p95 залишаються на рівні або незначно кращими порівняно з альтернативними підходами, що вказує на стабільність реакції навіть під час близьких до пікових навантажень. Отже, інтеграція двох бібліотек дозволяє досягти невеликого, але відчутного покращення продуктивності, що може бути критичним для застосунків, де потрібні часті оновлення стану та оперативна взаємодія з інтерфейсом.

Таблиця 4.2 – Час оновлення стану (ms)

Сценарій	Метод	avg рендерів на дію
Додавання в кошик	Zustand	2.1
Додавання в кошик	MobX	1.8
Додавання в кошик	Zustand + MobX	1.8
Масове оновлення (1000)	Zustand	220
Масове оновлення (1000)	MobX	8
Масове оновлення (1000)	Zustand + MobX	9

Різниця у кількості ререндерів (таблиця 4.2) між Zustand та MobX пояснюється відмінностями їхніх внутрішніх механізмів роботи зі станом. Zustand використовує просту модель підписок і не виконує глибокого аналізу залежностей, тому під час масових оновлень кожна зміна може викликати зайві ререндери компонентів [30]. Це не є проблемою для легких сценаріїв, де змінюється лише невелика частина стану, однак у випадку інтенсивного навантаження (наприклад, 1000 оновлень) такий метод призводить до значного зростання кількості повторних рендерів.

MobX, на відміну від Zustand, реалізує реактивну модель з точним відстеженням залежностей між даними та компонентами. Завдяки цьому оновлюються лише ті частини інтерфейсу, які справді змінюються, що дозволяє суттєво зменшити кількість ререндерів навіть під час великих серій подій. Комбінований метод, де MobX використовується поверх Zustand, успадковує ці переваги, що пояснює близькі до MobX результати в обох тестових сценаріях.

Таблиця 4.3 – Споживання пам’яті

Сценарій / Фреймворк	Zustand	MobX	Zustand + MobX
Стандартна сесія (1 хв) Average / Max (JS Heap, MB)	~ 42 / ~ 50	~ 40 / ~ 52	~ 42 / ~ 54
Тривале навантаження (5 хв). Темп росту (MB в 1 хвилину)	+0.20	+0.25	+0.28

Під час тривалого навантаження (5 хв) найбільше зростання споживання пам’яті характерне для комбінованого підходу Zustand + MobX – до +0.28 МБ за хвилину (таблиця 4.3). Це пояснюється тим, що при поєднанні двох систем стану утворюється додатковий шар обгорток, підписок, реактивних спостерігачів та службових структур, які дублюють частину функцій одна одної. Через це в системі накопичується більше проміжних об’єктів, а GC (збирач сміття) очищує їх менш ефективно. Саме надлишкові реактивні залежності та механізми синхронізації між Zustand і MobX спричиняють поступове збільшення споживання пам’яті з часом. Тоді як Zustand і MobX окремо демонструють рівномірніший та помірний темп росту (+0.20 та +0.25 відповідно), комбінований метод показує, що одночасне використання двох парадигм управління станом створює додаткове навантаження на оперативну пам’ять, що у деяких випадках може стати істотним недоліком, але в інформаційних системах електронної комерції мало коли є ключовим через малий обсяг бізнес-логіки на клієнтскій частині додатку.

Таблиця 4.4 – Пропускна здатність

Метод	Throughput (actions/sec)
Zustand	30050
MobX	26320
Zustand + MobX	29400

Комбінований метод демонструє пропускну здатність на рівні 29400 дій/сек (таблиця 4.4), що практично збігається з показником Zustand (30050 дій/сек). Це пояснюється тим, що в даному функціоналі саме Zustand виконує основну роль у керуванні станом на рівні клієнта, тоді як MobX задіяний лише для реактивних обчислень і не бере участі у високочастотній обробці дій. У результаті більшість операцій проходять через легковагову архітектуру Zustand, що мінімізує накладні витрати та забезпечує високу пропускну здатність. Додаткові механізми MobX створюють певний, але невеликий оверхед, тому

Throughput у комбінованому варіанті трохи нижчий, але загалом залишається майже таким же швидким, як у чистого Zustand.

Таблиця 4.5 – Стійкість, відновлюваність та персистентність

Метод	Час відкату / rollback (ms)
Zustand	1.6 ms
MobX	2.7 ms
Zustand + MobX	1.5 ms

Отримані результати показують, що найшвидший середній час (таблиця 4.5) відкату стану демонструє комбінований метод Zustand + MobX – 1.5 ms, що навіть трохи швидше за чистий Zustand (1.6 ms) і суттєво швидше за MobX (2.7 ms). Це пояснюється тим, що у цій конфігурації саме Zustand відповідає за механізм історії станів, персистентність і операції rollback, тоді як MobX використовується здебільшого для реактивних оновлень і не впливає на логіку збереження попередніх версій стану. У результаті відкат виконується через легковагову й просту структуру Zustand, без залучення складних реактивних залежностей.

4.3. Аналіз та порівняння комбінованого методу з класичним

Для отримання інтегральних показників були повторно виконані розрахунки відповідно до методології, детально описаної у пункті 3.3 та записано до таблиці 4.6.

Таблиця 4.6 – Результати інтегральної оцінки

Метод	Швидкодія	Ререндеринги	Пропускна здатність	Пам’ять	Відновлюваність
Zustand	0.9643	0.0364	0.9524	1	0.9375
MobX	0.8182	1	1	0.8759	0.5556
Zustand + MobX	1	0.8889	0.9524	0.9784	1

Як видно з отриманих значень в підсумковій таблиці 4.7, комбінований метод демонструє найвищу узгодженість показників одразу у кількох ключових категоріях, зберігаючи баланс між швидкодією, пропускною здатністю та стабільністю відновлення стану. Однак для повного розуміння ефективності важливо розглянути, як інтегральні значення впливають на загальний підсумковий результат.

Таблиця 4.7 – Підсумкові результати інтегральної оцінки

Метод	Підсумковий Score
Zustand	0.7350
MobX	0.8762
Zustand + MobX	0.9608

Інтегральний аналіз підтверджує попередні якісні висновки: Zustand + MobX – найвигідніша архітектура з погляду сукупної ефективності для тестового e-commerce додатка. Вона поєднує:

– дуже низьку латентність глобальних операцій (перевага Zustand);

– практично мінімальні ререндери при роботі з великими колекціями (перевага MobX);

– високу пропускну здатність із компромісами в плані пам’яті.

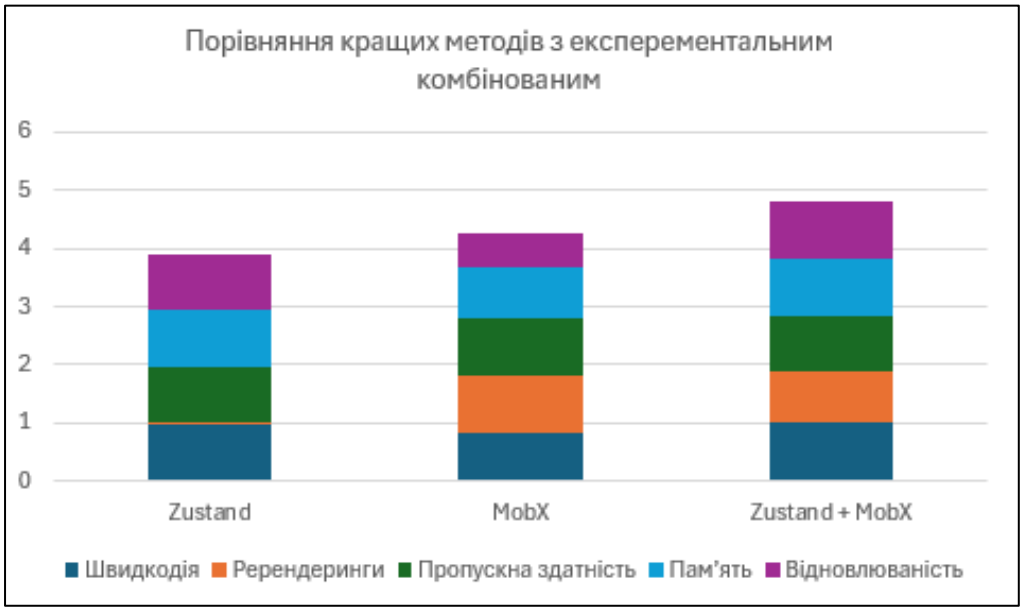

Рисунок 4.2 – Порівняльна гістограма кращих методів з комбінованим

Як показує графік (рисунок 4.2) рекомендовано використовувати цю комбіновану схему, якщо пріоритет – швидкість і User Experience (UX) при роботі з великими списками; якщо ж обмеження по пам’яті критичні (наприклад, дуже слабкі клієнти або жорсткі ліміти), слід розглянути оптимізацію (усунення надлишкових підписок, більш агресивна серіалізація локальних моделей або тримання деяких даних на сервері), або відмову від повного поєднання у «важких» ділянках.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи магістра було досліджено проблематику ефективності функціонування клієнтської частини інформаційних систем електронної комерції шляхом удосконалення методів управління станом у веб-орієнтованих додатках на базі фреймворку React. На основі проведеного аналізу, моделювання та експериментальних досліджень отримано наступні результати.

У межах роботи було проведено порівняльний аналіз існуючих сучасних методів управління станом за критеріями продуктивності, споживання пам’яті та масштабованості. Розглянуто такі бібліотеки, як Redux Toolkit, MobX, Zustand, Recoil та XState. Встановлено, що жодне з монолітних рішень не забезпечує ідеального балансу між швидкодією та зручністю підтримки для високонавантажених інтерфейсів інтернет-магазинів.

На основі всебічного аналізу та оцінки обраних методів було розроблено комбінований метод управління станом (Zustand + MobX), що поєднує переваги кожного з розглянутих інструментів та нівелює їхні недоліки. Запропонована архітектура передбачає чіткий розподіл відповідальності: використання Zustand для легковагого глобального стану та MobX для забезпечення дрібнозернистої реактивності складних доменних моделей.

Розроблений метод було застосовано до тестового проєкту, який моделює типовий функціонал E-commerce системи (каталог товарів, кошик, авторизація, фільтрація). На базі цього прототипу проведено повторний експериментальний аналіз та навантажувальне тестування, які підтвердили ефективність запропонованого підходу. Результати апробації засвідчили, що комбінована архітектура дозволяє зменшити час оновлення стану та мінімізувати кількість повторних рендерингів компонентів (re-renders) порівняно з класичними підходами. Зокрема, досягнуто високих показників відновлюваної здатності

системи та стабільності роботи інтерфейсу при маніпуляціях з великими масивами даних.

Наукова новизна роботи полягає в тому, що тема оцінки та оптимізації методів управління станом у веб-додатках електронної комерції досі залишається недостатньо вивченою в наукових публікаціях, попри її високу практичну значущість. У роботі вперше запропоновано систематизований підхід до поєднання декількох парадигм управління станом у межах однієї клієнтської архітектури, що дозволяє підвищити продуктивність та гнучкість інтерфейсу без втрати керованості та масштабованості коду. Крім того, проведене експериментальне дослідження формує підґрунтя для подальших наукових робіт у напрямі адаптивних та гібридних систем управління станом, орієнтованих на високонавантажені додатки електронної комерції.

За результатами дипломної роботи опубліковано тези [31].

Пояснювальна записка до кваліфікаційної роботи оформлена згідно ДСТУ та методичними вказівками щодо розробки та оформлення кваліфікаційної роботи [32-34].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Apollo Client: State Management. Apollo GraphQL. URL: https://www.apollographql.com/docs/react/local-state/local-state-management (дата звернення: 02.11.2025).

2. Jotai: Primitive and flexible state management for React. Jotai Documentation. URL: https://jotai.org/ (дата звернення: 02.11.2025).

3. Josh W. Comeau. Why React Re-Renders. URL: https://www.joshwcomeau.com/react/why-react-re-renders/ (дата звернення: 01.11.2025).

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

5. Narayn H. Just React!: Learn React the React Way. Apress L. P., 2022.

6. useContext – React. URL: https://react.dev/reference/react/useContext (дата звернення: 22.10.2025).

7. useReducer – React. URL: https://react.dev/reference/react/useReducer (дата звернення: 02.11.2025).

8. useContext – React. URL: https://react.dev/reference/react/useContext (дата звернення: 22.10.2025).

9. Context API – React. React.dev. URL: https://react.dev/learn/passing-data-deeply-with-context (дата звернення: 02.11.2025).

10. Introduction | Redux. URL: https://redux.js.org/introduction (дата звернення: 24.10.2025).

11. Byers D. An Introduction to Redux's Core Concepts. URL: https://www.digitalocean.com/community/tutorials/redux-redux-intro (дата звернення: 10.12.2025).

12. Updating state – Zustand. URL: https://zustand.docs.pmnd.rs/guides/updating-state (дата звернення: 24.10.2025).

13. Introduction to MobX. URL: https://mobx.js.org/README.html (дата звернення: 02.11.2025).

14. Managing React state with Zustand – LogRocket Blog. URL: https://blog.logrocket.com/managing-react-state-zustand/ (дата звернення: 14.12.2025).

15. Defining data stores · MobX. URL: https://mobx.js.org/defining-data-stores.html (дата звернення: 24.10.2025).

16. What are state machines and statecharts? | Stately. URL: https://stately.ai/docs/state-machines-and-statecharts (дата звернення: 24.10.2025).

17. Why React Re-Renders • Josh W. Comeau. URL: https://www.joshwcomeau.com/react/why-react-re-renders/ (дата звернення: 01.11.2025).

18. Stately: XState Documentation. Stately.ai. URL: https://stately.ai/docs/ (дата звернення: 02.11.2025).

19. Overview | TanStack Query React Docs. URL: https://tanstack.com/query/latest/docs/react/overview (дата звернення: 02.11.2025).

20. useMemo – React. URL: https://react.dev/reference/react/useMemo (дата звернення: 02.11.2025).

21. React Hooks for Data Fetching. *SWR*. URL: https://swr.vercel.app/ (дата звернення: 02.11.2025).

22. Take heap snapshot | Developer Guide | Nightwatch.js. URL: https://nightwatchjs.org/guide/running-tests/take-heap-snapshot.html (дата звернення: 10.12.2025).

23. Chrome for Developers. Record heap snapshots | Chrome DevTools. URL: https://developer.chrome.com/docs/devtools/memory-problems/heap-snapshots (дата звернення: 01.11.2025).

24. useContext vs Redux: Choosing the Right State Management Solution | by Love Trivedi | ZestGeek | Medium. URL: https://medium.com/zestgeek/usecontext-vs-redux-choosing-the-right-state-management-solution-5d7ee71d248b (дата звернення: 01.11.2025).

25. Introduction to web APIs - Learn web development | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_APIs/Introduction (дата звернення: 10.12.2025).

26. Getting Started | Axios Docs. URL: https://axios-http.com/docs/intro (дата звернення: 02.11.2025).

27. useContext vs Redux: Choosing the Right State Management Solution | by Love Trivedi | ZestGeek | Medium. URL: https://medium.com/zestgeek/usecontext-vs-redux-choosing-the-right-state-management-solution-5d7ee71d248b (дата звернення: 01.11.2025).

28. Defining data stores · MobX. URL: https://mobx.js.org/defining-data-stores.html (дата звернення: 24.10.2025).

29. MDN Web Docs. Performance: memory property - Web APIs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/memory (дата звернення: 01.11.2025).

30. Introduction – Zustand. URL: https://zustand.docs.pmnd.rs/ (дата звернення: 02.11.2025).

31. Двугрошев А.О., Міхнова О.Д. Порівняння методів управління станом у веб-орієнтованих додатках при застосуванні фреймворку React. 10th International scientific and practical conference “Science and technology: challenges, prospects and innovations” (May 22-24, 2025) CPN Publishing Group, Osaka, Japan. 2025. С. 260-263.

32. ДСТУ 3008:2015 "Звіти у сфері науки і техніки. Структура та правила оформлення". Київ: Держстандарт України, 2017. 31 с.

33. ДСТУ 8302:2015 «Бібліографічне посилання. Загальні положення та правила складання». Київ: Держстандарт України, 2017. 20 с.

34. Методичні вказівки щодо розробки та оформлення кваліфікаційної роботи (для студентів усіх форм навчання другого (магістерського) рівня програми «Інформаційні управляючі системи та технології») / Упоряд.: Петров К.Е., Левикін В.М., Чалий С.Ф., Євланов М.В., Саєнко В.І., Міхнов Д.К., Міхнова А.В., Чала О.В. Харків: ХНУРЕ, 2021. 30 с.