Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів управління станом односторінкових застосунків
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗздм-23-1

Тетяна МАЛІКОВА
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник проф. Кирило СМЕЛЯКОВ
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет центр післядипломної освіти
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Маліковій Тетяні Віталіївні
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів управління станом односторінкових застосунків»

Затверджена наказом по університету від 21.04.2025 № 61СТз

2. Термін подання студентом роботи до екзаменаційної комісії 09.06.2025

3. Вихідні дані до роботи фреймворк React – як основна платформа для створення односторінкових застосунків, інструменти локального (useState, useReducer) та глобального управління станом (Context API, Redux, Zustand, Recoil), середовище розробки (Node.js, Visual Studio Code) та інструменти для тестування та профілювання продуктивності React Developer Tools, Redux DevTools, Chrome DevTools, React Profiler, а також використовувані методи аналізу такі як порівняльне тестування, емпіричні вимірювання продуктивності, аналіз обсягу коду та масштабованості.

4. Перелік питань, що потрібно опрацювати в роботі

Аналіз предметної галузі і постановка задачі, огляд й аналіз літературних, наукових джерел, огляд фреймворку React, огляд та характеристика методів глобального управління станом, порівняльний аналіз бібліотек управління станом, експериментальне дослідження застосунку з різними підходами до управління станом, методологія порівняльного аналізу, критерії оцінки ефективності, результати експериментального дослідження.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	21.04.25	*виконано*
2	Аналіз предметної галузі і постановка задачі	21.04 – 23.04.25	*виконано*
3	Виявлення проблемних ситуацій	23.04 – 26.04.25	*виконано*
4	Аналіз та моделювання предметної області, розробка теоретичної частини	26.04 – 27.04.25	*виконано*
5	Планування експерименту	28.04 – 02.05.25	*виконано*
6	Програмна реалізація	02.05– 04.05.25	*виконано*
7	Експериментальні дослідження	05.05 – 10.05.25	*виконано*
8	Аналіз результатів експериментальних досліджень та розробка рекомендацій	10.05 – 14.05.25	*виконано*
9	Написання та оформлення статті та тез доповіді	14.05 – 18.05.25	*виконано*
10	Підготовка пояснювальної записки	19.05 – 22.05.25	*виконано*
11	Підготовка презентації та доповіді	22.05 – 25.05.25	*виконано*
12	Нормоконтроль	26.05 – 29.05.25	*виконано*
13	Рецензування	29.05 – 31.05.25	*виконано*
14	Занесення диплома в електронний архів	01.06 – 07.06.25	*виконано*
15	Попередній захист	08.06.25	*виконано*
16	Допуск до захисту у зав. кафедри	09.06.25	*виконано*

Дата видачі завдання 21 квітня 2025р.

Здобувач (ка) _____ Тетяна МАЛІКОВА
(підпис)

Керівник роботи _____ проф Кирило СМЕЛЯКОВ
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 79 с., 20 рис., 3 табл., 20 джерел, 4 додатка.

МЕТОДИ УПРАВЛІННЯ СТАНОМ, ОДНОСТОРІНКОВІ ЗАСТОСУНКИ (SPA), REACT, REDUX, ZUSTAND.

Об’єктом дослідження є методи управління станом у сучасних односторінкових вебзастосунках, створених за допомогою фреймворку React.

Метою дипломного проєкту є аналіз, порівняння та оцінка ефективності використання популярних бібліотек Redux та Zustand для вирішення завдань управління станом у SPA-застосунках.

У процесі роботи використовувалися такі методи дослідження: теоретичний аналіз літературних і технічних джерел щодо архітектури React та бібліотек стану; проєктування та реалізація тестового SPA-застосунку з різними підходами до управління станом; експериментальне тестування продуктивності, зокрема часу рендерингу, кількості ререндерів та обсягу коду.

У результаті дослідження було проведено порівняльну оцінку бібліотек Redux і Zustand за такими критеріями, як продуктивність, простота інтеграції, гнучкість структури даних та зручність підтримки. Отримані результати дозволили сформувати рекомендації щодо вибору технології управління станом залежно від масштабу та складності проєкту.

STATE MANAGEMENT METHODS, SINGLE-PAGE APPLICATIONS (SPA), REACT, REDUX, ZUSTAND.

The object of the study is state management methods in modern single-page applications built using the React framework.

The purpose of the project is to analyze, compare, and evaluate the effectiveness of using popular libraries Redux and Zustand to address state management tasks in SPA-based systems.

The research methods include theoretical analysis of literature and technical sources on React architecture and state libraries, design and implementation of a test SPA application using different state management approaches, and experimental performance testing, including render time, number of re-renders, and code volume.

As a result of the research, a comparative evaluation of Redux and Zustand was carried out based on criteria such as performance, integration simplicity, data structure flexibility, and ease of maintenance. The obtained results made it possible to formulate recommendations on selecting an appropriate state management library depending on the scale and complexity of the project.

Завідувачу кафедри
ПІ_____
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, сласне ім'я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Малікова Тетяна Віталіївна, студентка гр. ІПЗздм-23-1, здобувачка вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів управління станом односторінкових застосунків», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлена з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата Підпис

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

SPA – Single Page Application

UI – User Interface

API – Application Programming Interface

DOM – Document Object Model

DevTools – Developer Tools

VS Code – Visual Studio Code

LOC – Lines of Code

CRUD – Create, Read, Update, Delete

ВСТУП

Сучасні односторінкові застосунки (Single Page Applications, SPA) є невід'ємною складовою сучасної веб-розробки, яка стрімко розвивається у відповідь на потреби користувачів у швидких, зручних та інтерактивних інтерфейсах. Такі застосунки дозволяють значно зменшити обсяги даних, які передаються між клієнтом і сервером, завдяки завантаженню єдиної HTML-сторінки та динамічному оновленню вмісту через JavaScript. Це забезпечує високу продуктивність, кращу взаємодію з користувачем та зменшення часу очікування при навігації між сторінками.

Разом з тим, збільшення обсягів функціональності, інтеграція сторонніх сервісів, ускладнення логіки взаємодії між компонентами та посилення вимог до масштабованості призводять до серйозних викликів у сфері управління станом. Стан застосунку – це сукупність усіх даних, які відображають поточний стан інтерфейсу, користувацького вводу, API-викликів тощо. Його ефективна організація та підтримка є критично важливими для стабільності та підтримуваності проєкту.

Фреймворк React, один із найпопулярніших інструментів для створення SPA, надає кілька базових засобів для управління станом, таких як хуки useState, useReducer, а також Context API для передачі стану між компонентами. Проте у реальних проєктах цього часто недостатньо, особливо якщо застосунок має складну архітектуру або велике дерево компонентів. У таких випадках розробники вдаються до використання сторонніх бібліотек для централізованого управління станом.

Серед найпоширеніших рішень виділяються Redux та Zustand. Redux є надійним та широко застосовуваним інструментом з чіткою структурою та підтримкою великої екосистеми. Він базується на суворій односпрямованій передачі даних та використанні єдиного сховища (store), що дозволяє контролювати всі зміни стану в одному місці. Водночас, Redux потребує написання значного обсягу шаблонного коду, що може ускладнити розробку.

Zustand, навпаки, є легшою у використанні бібліотекою з мінімалістичним API. Вона дозволяє швидко створювати глобальний стан без складної конфігурації, забезпечує високу продуктивність та простоту інтеграції, що робить її привабливою для малих і середніх проєктів.

Актуальність цієї роботи полягає у необхідності обґрунтованого вибору технології для управління станом у SPA-застосунках з урахуванням таких факторів, як продуктивність, складність реалізації, підтримка спільнотою та зручність масштабування. Різноманітність бібліотек створює потребу у системному аналізі їх можливостей та обмежень.

Це дослідження має на меті порівняти бібліотеки Redux і Zustand, проаналізувати їх ефективність у різних умовах – від невеликих застосунків до проєктів із великою кількістю компонентів та складними зв’язками між ними. Результати мають сприяти підвищенню якості розробки, оптимізації архітектури та вибору інструментів, які відповідають конкретним потребам розробника та проєкту загалом.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі дослідження

Упродовж останнього десятиліття односторінкові застосунки (Single Page Applications, SPA) стали провідним підходом у створенні інтерактивних вебінтерфейсів. Цей підхід базується на завантаженні єдиного HTML-документа, що динамічно оновлюється в процесі взаємодії користувача з інтерфейсом без необхідності повного перезавантаження сторінки. Така модель дозволяє створювати швидкі, гнучкі та зручні у користуванні застосунки, які значно зменшують навантаження на сервер і підвищують загальну продуктивність системи.

Завдяки ефективному використанню ресурсів, SPA активно застосовуються у найрізноманітніших галузях: електронна комерція, інформаційні портали, онлайн-сервіси, особисті кабінети користувачів тощо. Проте зростання складності функціоналу призводить до збільшення кількості компонентів, логічних зв’язків між ними та залежностей. Це, у свою чергу, створює серйозні виклики щодо організації, підтримки й масштабування стану застосунку.

Фреймворк React, який є одним із найпоширеніших інструментів для створення SPA, здобув популярність завдяки своїй компонентно-орієнтованій архітектурі, декларативному стилю програмування та широким можливостям для побудови реактивних інтерфейсів. Його екосистема підтримує масштабованість проєктів, але зі зростанням обсягу логіки зростає потреба у більш гнучких та надійних механізмах управління станом [1].

React дозволяє розробникам створювати незалежні, багаторазові компоненти з власним станом, а також підтримує передавання даних через властивості (props). У базовому варіанті React пропонує такі інструменти, як useState та useReducer, що дозволяють керувати локальним станом компонентів. Цього достатньо для невеликих застосунків, але в масштабніших системах виникає необхідність у глобальному стані, який пов’язує між собою кілька компонентів.

У таких випадках актуальним стає використання сторонніх бібліотек, які надають можливості для централізованого керування станом, спрощують його

передавання між компонентами та забезпечують контроль над змінами. Серед найбільш відомих рішень можна виділити Redux, MobX, Zustand, Recoil, а також Context API – вбудований механізм у React. Кожна з цих бібліотек має власну архітектуру, синтаксис, підходи до управління асинхронністю та оновленням компонентів.

Основні проблеми, які виникають при управлінні станом у SPA, включають:

- забезпечення актуальності та узгодженості даних між компонентами, що взаємодіють між собою;
- зменшення кількості непотрібних ререндерів компонентів для збереження продуктивності;
- інтеграцію обробки асинхронних запитів до зовнішніх API та збереження їх результатів у глобальному стані;
- забезпечення масштабованості архітектури застосунку при збільшенні обсягів функціональності;
- забезпечення зручності підтримки та розширення коду на різних етапах життєвого циклу проєкту.

Таким чином, виникає потреба у системному аналізі існуючих рішень, що дозволяє вибрати найбільш ефективні інструменти для кожного окремого проєкту.

1.2 Актуальність роботи

У сучасних умовах стрімкого розвитку вебтехнологій підвищуються вимоги до якості, швидкодії та масштабованості вебзастосунків. Односторінкові застосунки (SPA) стали стандартом для створення зручних і динамічних інтерфейсів, проте їх ефективна реалізація значною мірою зумовлена якістю організації управління станом. Зростання складності функціоналу, інтеграція асинхронних API-запитів та взаємодія між великою кількістю компонентів створюють додаткове навантаження на архітектуру стану.

Незважаючи на наявність базових механізмів керування станом у фреймворку React [1], розробка масштабованих застосунків потребує використання сторонніх бібліотек, які забезпечують централізоване зберігання, оновлення та

синхронізацію даних. Вибір конкретного інструменту безпосередньо впливає на продуктивність, гнучкість архітектури, зручність масштабування та підтримку коду на всіх етапах життєвого циклу.

Складність полягає в тому, що універсального рішення не існує, і кожна бібліотека має свої переваги та обмеження. Актуальність даного дослідження зумовлена необхідністю об'єктивного порівняння найбільш популярних бібліотек – Redux [2] і Zustand [3] – для визначення умов, за яких їх використання є найбільш доцільним.

Проведення такого аналізу дозволить не лише зробити обґрунтований вибір інструменту для конкретного проєкту, а й сформулювати практичні рекомендації для розробників, що працюють із React-застосунками різної складності.

1.3 Огляд та аналіз літературних і наукових джерел

Для забезпечення теоретичного підґрунтя дослідження та обґрунтованого вибору інструментів у дипломному проєкті було проведено огляд наукових, технічних, аналітичних та публіцистичних джерел, що висвітлюють питання побудови односторінкових вебзастосунків, архітектури фреймворку React [1], а також методів локального та глобального управління станом у сучасних JavaScript-застосунках.

Одним із ключових джерел стала офіційна документація React [1], що містить детальну інформацію про принципи побудови компонентів, життєвий цикл, особливості JSX, а також механізми локального управління станом через хуки useState, useReducer, useEffect тощо. Особлива увага приділяється проблемі передачі стану між компонентами та рекомендаціям щодо організації архітектури середніх і великих застосунків.

З метою дослідження глобального управління станом було проаналізовано документацію Redux [2], яка пропонує суворо структурований підхід на основі концепцій «store», «actions», «reducers» та «middleware». У документації наведено не лише базові приклади використання, але й пояснено, як масштабувати

застосунок, як налагоджувати потоки даних, а також як використовувати інструменти розробника (наприклад, Redux DevTools).

Також опрацьовано матеріали щодо Redux Toolkit [2] – рекомендованого інструменту для спрощення конфігурації Redux-застосунків, який дозволяє значно скоротити кількість шаблонного коду завдяки автоматизованій генерації дій та редюсерів.

Альтернативні підходи до централізованого управління станом представлені у джерелах, присвячених бібліотекам MobX [4, 5], Zustand [3] та Recoil [6, 7].

MobX базується на реактивній парадигмі та забезпечує автоматичне оновлення інтерфейсу при зміні стану [4, 5]. Джерела, присвячені MobX, акцентують увагу на простоті використання, мінімальному обсязі коду та гнучкості, проте також вказують на меншу передбачуваність логіки у великих застосунках.

Zustand вирізняється мінімалістичним API, відсутністю необхідності створення провайдерів і високою продуктивністю. У технічній документації наголошується на простоті інтеграції, підтримці селекторів і зменшенні кількості непотрібних ререндерів [3].

Recoil, розроблений Meta, представляє нову модель атомарного управління станом, де весь стан розбито на незалежні одиниці (атоми), а залежності між ними описуються через селектори [6, 7]. Аналіз документації свідчить про високу гнучкість підходу, але також відзначає недостатню зрілість бібліотеки та обмежену спільноту.

Крім технічної документації, для побудови теоретичної бази було використано фахову літературу та навчальні посібники. Зокрема, книга «Mastering React State Management» Дж. Холла [8] детально розглядає всі популярні підходи до роботи зі станом, надаючи практичні приклади, рекомендації щодо вибору технологій та аналіз продуктивності. Праця Б. Черні «Learning TypeScript» [9] стала корисним джерелом для розуміння типізації стану у складних React-застосунках, що актуально для підтримки якості коду на великих проєктах.

Також у процесі дослідження опрацьовано низку тематичних статей на технічних платформах (Medium, Dev.to, Stack Overflow), які містять приклади порівняння різних бібліотек на основі реальних проєктів, відгуки розробників, а також практичні поради щодо організації архітектури застосунків.

Оцінка опрацьованих джерел дозволила зробити висновок, що сьогодні не існує універсального рішення для управління станом, яке б підходило для всіх типів проєктів. Вибір інструменту має базуватися на конкретних вимогах, таких як масштаб, складність логіки, потреба в контролі чи продуктивності. Огляд літератури підтвердив доцільність порівняльного дослідження бібліотек Redux і Zustand як найбільш контрастних за підходами до побудови системи стану.

1.4 Постановка задачі

Актуальність теми дипломного проєкту зумовлена потребою у вирішенні низки практичних проблем, що виникають при реалізації сучасних React-застосунків, зокрема в частині управління станом. Існування великої кількості бібліотек – Redux, Zustand, MobX, Recoil, Context API – створює труднощі з вибором оптимального рішення, оскільки кожна з них має свої сильні сторони та обмеження. У реальних умовах розробки, коли застосунки відрізняються за масштабом, функціональністю та вимогами до продуктивності, необхідний системний підхід до вибору інструменту управління станом.

Дослідження у цій роботі спирається як на теоретичний аналіз, так і на практичну реалізацію демонстраційного застосунку для об'єктивного тестування можливостей обраних бібліотек.У рамках цього дипломного проєкту передбачається вирішення наступних задач:

а) огляд і класифікація бібліотек управління станом:

1) дослідити найпопулярніші бібліотеки, зокрема Redux, Zustand, MobX, Recoil, а також Context API;
2) описати архітектурні особливості, переваги та недоліки кожного інструменту в контексті використання з React;

б) визначення критеріїв для вибору бібліотек для порівняння:

1) встановити технічні й практичні критерії (продуктивність, простота використання, документація, підтримка спільнотою);
2) вибрати найбільш релевантні бібліотеки для подальшого дослідження – у цьому проєкті це Redux і Zustand;

в) реалізація прототипів SPA із використанням обраних бібліотек:

1) розробити демонстраційні застосунки з однаковою функціональністю, наприклад, управління списком завдань або кошиком товарів;
2) інтегрувати Redux та Zustand у відповідні версії проєкту;

г) проведення експериментального порівняння:

1) виконати тестування застосунків за метриками: час оновлення стану, кількість ререндерів, використання ресурсів;
2) оцінити зручність розробки (структура коду, складність інтеграції, документація);

д) формулювання висновків і практичних рекомендацій:

1) на основі проведеного аналізу надати рекомендації щодо вибору бібліотеки залежно від вимог до проєкту;
2) сформувати узагальнені принципи, які можуть бути застосовані в майбутніх проєктах для підвищення ефективності управління станом у SPA.

Очікуваним результатом дипломного проєкту є ґрунтовний аналіз та практична оцінка ефективності бібліотек Redux і Zustand, що дозволить сформулювати рекомендації для розробників щодо вибору методів управління станом в односторінкових застосунках на основі React.

2 ТЕОРЕТИЧНІ ЗАСАДИ УПРАВЛІННЯ СТАНОМ У REACT

2.1 Огляд фреймворку React та його особливості

React – це популярна JavaScript-бібліотека з відкритим кодом, розроблена компанією Meta (раніше Facebook) для створення інтерфейсів користувача [1]. Вона набула широкого поширення у світі веброзробки завдяки своїй компонентній архітектурі, високій продуктивності та активній спільноті. Основною метою React є побудова масштабованих та гнучких користувацьких інтерфейсів на основі декларативного підходу.

React дозволяє розробникам створювати складні вебзастосунки, які складаються з невеликих, ізольованих компонентів, що управляють власним станом і можуть бути багаторазово використані в межах одного або кількох проєктів [1]. Кожен компонент у React є незалежним блоком, що реалізує певну частину інтерфейсу та може взаємодіяти з іншими компонентами через властивості (props) та події.

Однією з ключових особливостей React є використання віртуального DOM (Virtual DOM), який слугує для оптимізації процесу оновлення інтерфейсу. Замість безпосереднього маніпулювання елементами DOM, React створює їх віртуальне представлення у пам’яті. При зміні стану компонента виконується порівняння (diffing) між старим і новим віртуальним DOM, після чого виконується мінімально необхідна кількість змін у реальному DOM. Такий підхід значно підвищує продуктивність застосунку [1].

Компоненти у React можуть бути двох типів: функціональні та класові. На сьогоднішній день функціональні компоненти з хуками стали основним стандартом, оскільки забезпечують компактний та зручний синтаксис.

Кожен компонент приймає вхідні дані у вигляді props (властивостей) і може мати свій стан (state). Зміна стану викликає повторне рендеринг компонента, що забезпечує динамічність інтерфейсу.

React підтримує композицію компонентів – один компонент може містити інші як дочірні, формуючи дерево компонентів. Така структура забезпечує гнучкість, модульність та повторне використання коду.

Хуки useState та useReducer є основними інструментами для управління локальним станом у функціональних компонентах [1].

useState дозволяє створювати простий стан зі значенням і функцією оновлення. Його зручно використовувати для керування локальними змінними – наприклад, текстовим полем, перемикачем, активною вкладкою тощо.

useReducer застосовується у випадках, коли структура стану складна або містить багато пов’язаних змін. Він реалізує принцип редукції (подібно до Redux) і дозволяє централізовано обробляти дії через редюсер.

Обидва механізми добре працюють у межах одного компонента. Проте коли виникає потреба в обміні станом між різними частинами програми – особливо у великій ієрархії компонентів – використання локального стану стає неефективним. У таких випадках доцільним є впровадження глобального управління станом – за допомогою Context API [10] або сторонніх бібліотек, таких як Redux [2], MobX [4, 5], Zustand [3] чи Recoil [6, 7]. Саме цим аспектам присвячено наступний розділ.

2.2 Проблематика масштабування та «проп-дрилінгу»

У невеликих React-застосунках використання локального стану (useState, useReducer) цілком виправдане і зручне. Кожен компонент відповідає лише за власний стан, що забезпечує простоту розробки та ізольованість логіки. Проте зі зростанням проєкту і збільшенням кількості взаємодіючих між собою компонентів виникають труднощі, пов’язані з організацією, передачею та узгодженістю даних між компонентами різного рівня вкладеності [10].

Однією з найбільш поширених проблем при цьому є явище, відоме як «проп-дрилінг» (props drilling) – ситуація, коли дані або функції передаються від батьківських до глибоко вкладених дочірніх компонентів через посередників, які самі ці дані не використовують. Це призводить до надлишкової передачі пропсів через кілька рівнів компонентного дерева, ускладнює читання та супровід коду, і підвищує ймовірність помилок при зміні структури застосунку [11].

У великих застосунках, де один і той самий фрагмент стану потрібен у кількох незалежних частинах інтерфейсу, локальне управління стає неефективним з таких причин:

Ускладнена передача даних. При використанні лише локального стану доводиться дублювати логіку обробки стану або створювати обхідні шляхи для доступу до даних, що суперечить принципам чистої архітектури;

Низька масштабованість. Зі зростанням глибини вкладеності компонентів зростає складність у підтримці та зміні структури застосунку. Кожна зміна може потребувати модифікації кількох компонентів одночасно;

Збільшення кількості ререндерів. Коли стан передається через багато компонентів, будь-яка його зміна може спричинити повторний рендер усіх проміжних компонентів, навіть якщо вони не залежать від цього стану безпосередньо;

Погіршення читабельності коду. Код, у якому пропси передаються через декілька рівнів вкладеності, стає важким для розуміння, особливо при повторному використанні компонентів або при роботі нових членів команди.

Наприклад, якщо у React-застосунку компонент App містить Dashboard, який у свою чергу містить UserList, а той – UserCard, і якщо UserCard потребує доступу до певного стану з App, то цей стан доведеться передавати через усі проміжні компоненти навіть за відсутності у них прямої необхідності в цьому. Це призводить до надмірної зв’язаності компонентів і погіршує гнучкість архітектури.

Ще однією проблемою є обробка асинхронних даних, таких як результати API-запитів або відповіді з баз даних. Розміщення логіки запитів на глибоко вкладених рівнях може спричинити дестабілізацію стану та труднощі з синхронізацією.

Усе це свідчить про обмеження локального управління станом у масштабних проєктах і обґрунтовує необхідність переходу до глобального або централізованого управління станом, де дані зберігаються в окремому сховищі, а компоненти мають доступ до потрібних їм частин стану без проміжних передач. Такий підхід підвищує

масштабованість, спрощує супровід і дозволяє ефективно працювати з розподіленою логікою та складними інтерфейсами.

2.3. Огляд та характеристика методів глобального управління станом

2.3.1. Context API: принцип роботи, переваги та обмеження

Context API – це вбудований механізм у React для передачі глобального стану між компонентами, що перебувають на різних рівнях вкладеності [10]. Його основною метою є усунення проблеми «проп-дрилінгу» – непотрібного передавання даних через проміжні компоненти, які самі ці дані не використовують.

Механізм Context API базується на створенні контексту (context) , який містить значення та функції, що будуть доступні будь-якому дочірньому компоненту в межах цього контексту [10]. Передавання даних здійснюється за допомогою двох основних елементів: Provider (надавач значень) і Consumer (споживач значень).

У сучасній розробці для споживання контексту зазвичай використовують хук `useContext`, що спрощує синтаксис.

Переваги Context API:

- вбудований у React, не потребує сторонніх залежностей;
- добре підходить для передачі глобальних налаштувань (тема, мова інтерфейсу, авторизація;
- простий у реалізації для невеликих і середніх застосунків.

Обмеження Context API:

- оновлення контексту призводить до повторного рендерингу всіх компонентів, що його використовують;
- обмежена масштабованість для складного стану з багатьма гілками;
- відсутність вбудованих засобів для керування асинхронністю чи складною логікою [10].

2.3.2 Redux: архітектура, структура, інтеграція

Redux – одна з найпопулярніших бібліотек для централізованого управління станом, особливо у великих і складних застосунках. Вона реалізує принцип уніфікованого потоку даних (unidirectional data flow), де стан змінюється лише через спеціальні дії (actions) і обробляється функцією-редюсером (reducer).

Основні елементи Redux [2]:

- store – єдине джерело істини (centralized state);
- actions – об’єкти, що описують, які саме зміни мають відбутися;
- reducers – чисті функції, які приймають поточний стан і дію, повертаючи новий стан;
- dispatch – метод для відправлення дій до редюсерів.

Архітектура Redux забезпечує передбачуваність, можливість легкої налагоджуваності (через Redux DevTools) і чітке структурування коду [12].

Для інтеграції Redux у React-застосунок зазвичай використовується бібліотека react-redux , яка надає компоненти `Provider` і хук `useSelector` / `useDispatch` [2].

Переваги Redux [12, 13]:

- підходить для великих проєктів із великою кількістю взаємозалежних даних;
- висока передбачуваність поведінки стану;
- потужні інструменти для налагодження (Redux DevTools);
- активна спільнота, велика кількість навчальних матеріалів.

Недоліки Redux [13]:

- значна кількість шаблонного коду;
- вища крива навчання для новачків;
- потреба у структурованій організації файлів і дій.

2.3.3. MobX, Zustand, Recoil – порівняння підходів

Крім Redux, існують альтернативні бібліотеки, що пропонують інші підходи до управління станом – більш реактивні, прості або гнучкі. До таких належать MobX, Zustand і Recoil.

MobX – бібліотека, що реалізує реактивний підхід до управління станом [4, 5]. Вона автоматично відстежує залежності між даними і компонентами, оновлюючи інтерфейс при зміні даних без потреби в ручному керуванні діями.

Переваги MobX [4, 5]:

- автоматичне оновлення компонентів;
- простий у використанні;
- менше коду, ніж у Redux.

Недоліки MobX:

- менш передбачувана логіка у великих застосунках;
- складніша інтеграція з TypeScript.

Zustand – легковажна бібліотека для управління глобальним станом. Вона не потребує обгортання компонентів у Provider, має мінімальний API та використовує функціональний підхід.

Переваги Zustand:

- простий синтаксис;
- підтримка розділення логіки на модулі;
- висока продуктивність і підтримка селекторів.

Недоліки Zustand [3]:

- менша кількість прикладів у спільноті;
- обмежена підтримка складних кейсів без ручної конфігурації.

Recoil – бібліотека, створена Meta, орієнтована на управління станом зі складними залежностями [6, 7]. Вона вводить поняття атомів (мінімальні одиниці стану) та селекторів (похідні значення на основі атомів).

Переваги Recoil [6, 7]:

- добре працює з залежними станами;
- висока гнучкість;

– інтеграція з react out-of-the-box.

Недоліки Recoil:

– менш зріла технологія;

– відсутність офіційної стабільної версії.

Кожен із розглянутих інструментів має свої сильні та слабкі сторони [8]. Context API ідеально підходить для невеликих застосунків або глобальних налаштувань. Redux – перевірене рішення для великих та структурованих проєктів. MobX забезпечує швидкий старт із меншою кількістю коду, але може ускладнити контроль над даними. Zustand вирізняється простотою і продуктивністю, а Recoil – гнучкістю у роботі з атомарним станом.

Вибір конкретного методу управління станом повинен базуватися на масштабі проєкту, складності логіки, вимогах до продуктивності та досвіді команди розробників [8, 9].

2.3.4 Порівняльний аналіз методів управління станом

У сучасній розробці односторінкових застосунків (SPA) ефективне управління станом є ключовою передумовою створення стабільного, масштабованого й зручного у підтримці програмного продукту. У зв’язку з цим постає необхідність у глибокому аналізі доступних рішень – як вбудованих у React, так і сторонніх бібліотек, які пропонують різні парадигми та технічні підходи до реалізації цієї задачі.

Для порівняння методів управління станом у межах цього дослідження було обрано низку технічних та практичних критеріїв, які безпосередньо впливають на досвід розробки та якість реалізованого функціоналу:

Продуктивність – здатність мінімізувати кількість непотрібних оновлень компонентів, оптимізована робота з ререндерингом.

Простота використання – легкість інтеграції бібліотеки в проєкт, обсяг шаблонного коду, зручність API.

Масштабованість – придатність бібліотеки до роботи з великими застосунками з великою кількістю компонентів.

Передбачуваність – наскільки контрольованим є стан застосунку, наскільки легко його відслідковувати та відлагоджувати.

Гнучкість – можливість налаштовувати, розширювати та адаптувати під різні сценарії використання.

Наявність інструментів для налагодження (DevTools) – інтеграція з інструментами розробника, підтримка дебагінгу.

Ці критерії було застосовано до таких рішень: Context API, Redux, MobX, Zustand і Recoil.

Відповідно результати порівняння вказують на те, що Context API добре виконує свою функцію в обмежених сценаріях, наприклад для передачі глобальних параметрів (тема інтерфейсу, мова, авторизація). Проте при збільшенні складності застосунку він починає демонструвати обмеження щодо продуктивності та масштабованості, оскільки оновлення контексту призводить до ререндеру всіх дочірніх компонентів (див. табл. 2.1).

Таблиця 2.1 – Порівняльна таблиця методів управляння станом Context API, Redux, MobX, Zustand, Recoil (критерії вибрано на основі аналізу джерел [1-11]).

Критерій	Context API	Redux	MobX	Zustand	Recoil
Продуктивність	Середня	Висока (з memo)	Висока	Висока	Висока
Простота використання	Висока	Низька	Висока	Дуже висока	Висока
Масштабованість	Обмежена	Висока	Середня	Висока	Висока
Передбачуваність	Середня	Висока	Низька/середня	Висока	Висока
Гнучкість	Обмежена	Висока	Висока	Висока	Висока

Redux забезпечує чітку та передбачувану структуру управління станом. Його архітектура на базі store → actions → reducers дозволяє контролювати всі зміни в

одному місці, що є дуже цінним у великих проєктах. Однак Redux має порівняно високу складність, особливо на етапі навчання, та вимагає написання значної кількості шаблонного коду. Ці недоліки частково компенсуються завдяки Redux Toolkit.

MobX дозволяє дуже швидко реалізовувати реактивні інтерфейси, з мінімальною кількістю коду. Автоматичне відстеження залежностей робить його зручним для малих і середніх проєктів. Але відсутність суворої структури та менш прозорий потік даних можуть ускладнити налагодження та тестування у складних застосунках.

Zustand вирізняється своєю простотою – API бібліотеки легко зрозуміти та впровадити. Вона не вимагає обгортання застосунку в Provider, що зменшує складність налаштування. Продуктивність забезпечується через селектори, які мінімізують кількість оновлень компонентів. Zustand чудово підходить для проєктів середньої складності.

Recoil пропонує сучасну модель атомарного стану, яка особливо добре працює зі складними взаємозалежностями. Цей підхід дозволяє компонування стану, гнучку інтеграцію, а також легке масштабування. Водночас бібліотека ще перебуває у фазі активної розробки, і рівень її стабільності не такий високий, як у Redux чи MobX.

Порівняння показує, що не існує універсального рішення, яке було б найкращим у всіх випадках. Вибір методу управління станом повинен залежати від низки чинників:

Для невеликих проєктів або простих завдань доцільно використовувати Context API або Zustand, які забезпечують достатню продуктивність і просту інтеграцію.

Для великих, довготривалих проєктів з розгалуженою логікою стану найкраще підходить Redux, що забезпечує стабільність, масштабованість та гнучкість.

MobX можна рекомендувати у випадках, коли потрібна швидка реалізація без суворої структуризації.

Recoil – перспективний вибір для проєктів, які потребують складного розділення стану на незалежні частини або потребують гнучкої реактивності.

Таким чином, порівняльний аналіз дозволив сформулювати чітке розуміння переваг і обмежень кожного з рішень та їх придатність до конкретних умов розробки.

3 ПОСТАНОВКА І ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНОЇ ЧАСТИНИ

3.1 Визначення цілей експерименту

Управління станом є однією з найважливіших задач у процесі створення сучасних вебзастосунків [1]. Особливо це актуально для односторінкових застосунків, де безперервна взаємодія користувача з інтерфейсом потребує ефективного механізму синхронізації даних. Фреймворк React пропонує базові засоби керування станом [1], однак у міру ускладнення проєкту виникає потреба у використанні додаткових інструментів, таких як Redux [2, 14], Zustand [3], MobX [4, 5] тощо.

У цьому дослідженні зосереджено увагу на порівнянні двох популярних бібліотек – Redux і Zustand – з метою оцінки їхньої ефективності, зручності у використанні та придатності для різних типів застосунків [12, 13]. Метою експериментальної частини є практичне порівняння чотирьох підходів до глобального управління станом у React-застосунках – Redux [2, 14], Zustand [3], Context API [10] та Recoil [6, 7] – шляхом реалізації однакового функціоналу в демонстраційних версіях SPA. Це дозволить оцінити продуктивність, гнучкість, зручність інтеграції та супроводу кожного підходу з урахуванням потреб різних типів проєктів [8].

Завдання експерименту полягає в наступному:

- розробити демонстраційний застосунок, що реалізує типові сценарії управління завданнями (додавання, фільтрація, оновлення стану, обробка асинхронних дій), з уніфікованим інтерфейсом для обох реалізацій;
- реалізувати застосунок з використанням redux [2, 14]: створити архітектуру сховища, реалізувати редюсери, екшени, middleware (при потребі), підключити компоненти через хук useSelector та useDispatch;
- реалізувати застосунок з використанням Zustand [3], застосовуючи сучасні підходи створення стану, оптимізацію селекторами, перевірити рендери через DevTools;

- зібрати час оновлення стану (мс), кількість повторних рендерів, споживання пам’яті. Використати React Profiler і браузерні інструменти (Chrome Performance);
- оцінити обсяг шаблонного коду та загальну кількість рядків, виміряти час на інтеграцію (Developer Time) із урахуванням налаштування середовища та читання документації, оцінити якість документації та наявність інструментів налагодження (DevTools);
- реалізувати спрощені версії застосунку з Context API [10] та Recoil [6, 7] для базового порівняння за основними метриками;
- на підставі експериментальних та якісних даних підготувати практичні висновки та надати рекомендації щодо вибору стейт-менеджера для різних типів React-застосунків [8].

3.2 План реалізації прототипів

Для забезпечення можливості об’єктивного порівняння різних підходів до управління станом у React-застосунках було розроблено план реалізації серії демонстраційних прототипів [11]. Метою є створення однакових за логікою, функціональністю та структурою односторінкових застосунків, які відрізнятимуться лише способом організації та зберігання стану. Такий підхід дозволяє унеможливити вплив зовнішніх факторів на результати тестування й гарантує чистоту експерименту.

У якості прикладного сценарію було обрано застосунок для бронювання місць. Користувач може вибрати одне або кілька доступних місць у сітці, підтвердити бронювання, а також переглянути перелік заброньованих місць із можливістю скасування Цей приклад дозволяє імітувати широкий спектр типових задач, характерних для реальних SPA – взаємодію з інтерфейсом, зміну стану, обробку користувацьких дій, відображення вибраних елементів та оновлення даних [1]. Функціонал охоплює базові сценарії, що дозволяють оцінити реакцію інтерфейсу на зміну стану, продуктивність, повторні ререндери компонентів, а також складність архітектурної реалізації і є типовим для багатьох реальних

вебзастосунків, що дозволяє моделювати роботу з динамічними інтерфейсами, які вимагають від системи швидкого реагування та надійної синхронізації даних.

Основні функціональні блоки прототипу:

- головна сітка місць: набір інтерактивних візуальних елементів, що відображають доступні для бронювання місця. Кожне місце має змінний стан – вільне, вибране користувачем або вже заброньоване;
- стан вибору: при натисканні на місце його статус змінюється на «вибране», що сигналізує користувачеві про можливість його подальшого бронювання;
- кнопка «Забронювати»: підтверджує поточний вибір і фіксує місця як заброньовані;
- секція заброньованих місць: динамічно формується список усіх вибраних місць із можливістю їх видалення;
- глобальне сховище стану: реалізується з використанням обраного підходу та забезпечує централізоване керування усіма аспектами стану застосунку.

Для реалізації інтерфейсу застосовується бібліотека Tailwind CSS, а в ролі середовища розробки використано React [1] у поєднанні з Create React App. Таким чином, усі прототипи розробляються в однакових умовах, що дає змогу мінімізувати вплив зовнішніх факторів на результати порівняння.

Перший прототип реалізовано з використанням бібліотеки Redux [2, 14]. Його структура базується на класичній архітектурі з єдиним сховищем стану, редюсерами, діями та диспетчеризацією [13]. Взаємодія з глобальним станом здійснюється через хуки useSelector і useDispatch. Цей підхід дозволяє чітко відслідковувати зміни стану та налагоджувати логіку за допомогою Redux DevTools.

Другий варіант застосунку реалізовано з використанням Zustand [3]. Цей інструмент дає змогу створити гнучке глобальне сховище з використанням простих функцій та хуків. У цій реалізації акцент зроблено на мінімалізмі та високій продуктивності – завдяки використанню селекторів кількість непотрібних

ререндерів зводиться до мінімуму. Також відсутня необхідність у використанні обгорток типу Provider, що спрощує архітектуру.

Третій прототип розроблений з використанням Context API [10] – вбудованого засобу React для управління глобальним станом. Всі дані передаються через контекст, створений за допомогою createContext, а управління логікою здійснюється через useReducer. Цей підхід демонструє обмеження Context API в складних архітектурах, зокрема щодо надмірного рендерингу.

Четверта версія використовує бібліотеку Recoil [6, 7], яка дозволяє організувати стан на основі атомів і селекторів. Такий підхід є особливо ефективним у ситуаціях, коли різні частини інтерфейсу мають залежність від окремих елементів глобального стану. Recoil забезпечує гнучке керування складними залежностями і високий рівень реактивності.

Кожен із чотирьох прототипів реалізовано за однаковими вимогами до інтерфейсу, структури компонентів та обсягу функціональності. Це дозволяє зосередитися саме на відмінностях у підходах до організації стану, а не на деталях реалізації. Усі проєкти перевіряються за допомогою DevTools та засобів аналізу ререндерів. У подальших розділах роботи буде проведено вимірювання продуктивності, обсягів пам’яті, кількості повторних рендерів і аналіз загальної ефективності кожного рішення.

3.3 Вибір критеріїв оцінювання

Для проведення об’єктивного порівняльного аналізу методів управління станом у React-застосунках необхідно визначити чіткий набір критеріїв, за якими можна оцінити ефективність, зручність використання та практичну придатність кожного з підходів [8]. У цьому дослідженні критерії було сформовано на основі аналізу технічної документації [1, 2, 7, 3] , наукових публікацій [15, 16], а також досвіду розробників з професійної спільноти.

Враховуючи специфіку React-застосунків та вимоги до сучасної фронтенд-розробки [11, 9], виділено такі ключові критерії:

- продуктивність (Performance). Цей параметр відображає, наскільки ефективно бібліотека оновлює інтерфейс при зміні стану. В рамках дослідження оцінюється середній час оновлення стану, кількість повторних рендерів компонентів, а також загальне споживання оперативної пам’яті під час роботи застосунку. Продуктивність є критично важливою для забезпечення плавної взаємодії користувача із застосунком. [15, 16];
- обсяг коду та складність реалізації (Implementation Effort). Цей критерій стосується кількості коду, який потрібно написати для реалізації функціоналу. Враховується як кількість рядків коду, так і його структурна складність. Крім того, аналізується необхідність створення допоміжних сутностей (actions, reducers, providers тощо) [13]. Менший обсяг шаблонного коду спрощує підтримку, покращує читабельність та зменшує ймовірність помилок;
- зручність використання та документація (Developer Experience). Оцінюється загальна зручність розробки: простота інтеграції бібліотеки в проєкт, наявність офіційної документації [2, 7, 3], її повнота та зрозумілість. Також враховується доступність навчальних ресурсів, активність спільноти та наявність готових прикладів використання [8];
- масштабованість (Scalability) визначає, наскільки обрана технологія здатна адаптуватися до великих застосунків зі складною архітектурою [11]. Оцінюється здатність ефективно працювати при зростанні кількості компонентів, глибини ієрархії та логічної складності. Також аналізується підтримка модульності, розділення логіки стану та повторного використання;
- інструменти для налагодження (Debugging Support). Наявність DevTools або інших засобів, які дозволяють відстежувати зміни стану, історію подій, структуру стану в реальному часі. Цей аспект особливо важливий для командної розробки, де відстеження помилок та зрозумілий потік даних мають критичне значення [12];

– гнучкість і адаптивність (Flexibility). Під гнучкістю мається на увазі здатність бібліотеки працювати з різними структурами даних, підтримка асинхронних дій, можливість інтеграції з іншими технологіями та бібліотеками. Також оцінюється можливість розширення функціоналу відповідно до потреб конкретного проєкту [8].

Усі вищезазначені критерії будуть використані для порівняння реалізованих прототипів на основі Redux, Zustand, Context API та Recoil. Вони дозволяють оцінити не лише технічні показники, але й практичний комфорт використання бібліотек у реальних умовах розробки.

3.4 Інструменти для вимірювання та тестування

Експериментальне дослідження, метою якого є порівняння бібліотек управління станом у React-застосунках, вимагає застосування надійних та інформативних інструментів для збору об’єктивних даних [15, 16]. Вибір інструментів обумовлювався необхідністю точно виміряти такі аспекти, як продуктивність, обсяг ререндерів, використання оперативної пам’яті, обсяг коду та зручність розробки. Для досягнення поставленої мети було використано низку сучасних засобів, які дозволяють здійснювати повноцінне тестування і налагодження React-застосунків [1].

Ключову роль у спостереженні за внутрішнім станом компонентів та їх поведінкою під час ререндеру відіграв інструмент React Developer Tools [1] – офіційне розширення для браузера. За його допомогою є можливість у зручному візуальному форматі аналізувати структуру компонентного дерева, перевіряти, як передаються пропси і стан, а також контролювати, які саме компоненти перерендерюються при зміні стану. Це дозволило виявити зайві рендери, оцінити ефективність роботи селекторів, а також перевірити глибину і частоту оновлень, що є критичними для продуктивності застосунку.

У реалізаціях на Redux використовувався Redux DevTools [2] – потужне розширення, яке надає можливість у реальному часі переглядати всі дії (actions), що відправляються у Redux store, а також відстежувати зміни стану після кожної з

них. Цей інструмент дозволив здійснити покроковий аналіз логіки застосунку, перевірити правильність роботи редюсерів.

Для глибшого аналізу продуктивності застосунків застосовувалися засоби Chrome DevTools, зокрема вкладка Performance. Вона дозволяє фіксувати часові діаграми рендерингу та перераховування, визначати затримки в головному потоці, аналізувати використання пам’яті і кількість JavaScript-викликів, що були ініційовані змінами стану. Використання Chrome Profiler допомогло визначити час, необхідний на оновлення інтерфейсу після взаємодії з користувачем, зокрема під час бронювання місць, зміни статусу елемента або ініціалізації компонентів.

Для фіксації обсягу пам’яті, яку споживає застосунок, використовувалась вкладка Memory у Chrome DevTools. Вона дозволяла порівнювати, наскільки ресурсоємним є кожен із варіантів реалізації стану, зокрема при повторних рендерах та активації логіки.

Обсяг коду, що необхідний для реалізації однакової бізнес-логіки в кожній бібліотеці, оцінювався шляхом підрахунку загальної кількості рядків коду (LOC – lines of code), включаючи налаштування сховища, логіку стану, інтеграцію з компонентами [8]. Це дало змогу кількісно порівняти складність реалізації для кожного підходу.

Оцінка зручності розробки проводилась на основі особистого досвіду використання кожної бібліотеки: враховувалась зрозумілість документації [2, 7, 3], наявність прикладів, складність конфігурації, а також суб’єктивне сприйняття легкості інтеграції у типовий React-проєкт [1].

Розробка і тестування прототипів проводилися у середовищі Visual Studio Code, яке забезпечувало зручність роботи з файлами, швидкий запуск локального сервера та доступ до розширень для ESLint, Prettier, відлагодження, а також інтеграції з Git [9]. Усі прототипи запускалися через npm start з використанням Create React App, що гарантувало однакові стартові умови для кожної реалізації.

Таким чином, сукупність вибраних інструментів дозволила отримати повний набір метрик, які забезпечили достовірне порівняння бібліотек Redux, Zustand, Context API та Recoil. Це охоплювало як кількісні показники продуктивності, так і

якісну оцінку досвіду розробника, що в сукупності дало змогу сформулювати практичні рекомендації щодо вибору найдоцільнішого інструменту для управління станом у React-застосунках залежно від масштабу та складності проєкту [8].

3.5 Методологія дослідження

Методологічна основа цього дослідження ґрунтується на експериментальному підході, що передбачає створення чотирьох аналогічних версій демонстраційного React-застосунку з однаковим функціоналом, реалізованим з використанням різних бібліотек управління станом: Redux [2, 14], Zustand [3], Context API [10] та Recoil [6, 7] . Кожна з реалізацій виконана на окремій гілці одного проєкту з ідентичним UI-дизайном і структурою компонентів [11]. Такий підхід дозволяє забезпечити однакові умови тестування та об’єктивність порівняння.

Дослідження проводиться в контрольованому середовищі, що включає однакову версію Node.js, єдиний браузер (Google Chrome), а також уніфіковані сценарії взаємодії з користувачем [15, 16]. Застосовуються як кількісні, так і якісні методи оцінювання. Кількісна частина базується на вимірюванні таких показників:

- час оновлення стану після взаємодії користувача;
- кількість рендерів компонентів під час оновлення даних;
- використання оперативної пам’яті браузером;
- кількість рядків коду в реалізації [8];
- тривалість реалізації кожної версії додатку.

Збір метрик здійснювався за допомогою таких інструментів:

- React Profiler [1] – для фіксації часу рендерингу та кількості оновлень;
- Chrome DevTools – для аналізу використання пам’яті;
- власна оцінка складності реалізації – для суб’єктивного порівняння простоти використання API кожної бібліотеки [8].

Для забезпечення прозорості процесу дослідження розроблено покроковий план реалізації та тестування, поданий у таблиці (див. табл. 3.1):

Таблиця 3.1 – План проведення експерименту (таблиця виконана самостійно)

Етап	Назва етапу	Опис дій
1	Підготовка	Створення репозиторію, встановлення середовища, базовий шаблон SPA
2	Реалізація з Redux	Налаштування store, reducers, actions, підключення компонентів через useDispatch та useSelector
3	Реалізація з Zustand	Створення стану через create, оптимізація з selector-ами
4	Реалізація з Context API	Імплементація контексту, надавання і споживання даних через Provider та useContext
5	Реалізація з Recoil	Визначення atoms, selectors, підключення компонентів
6	Тестування	Вимірювання часу рендерингу, використання пам’яті, аналіз ререндерів
7	Порівняльний аналіз	Побудова узагальнюючої таблиці результатів, виведення переваг і недоліків
8	Висновки	Формулювання практичних рекомендацій щодо вибору бібліотеки залежно від типу проєкту

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

4.1 Реалізація SPA-прототипів з використанням різних бібліотек

У рамках дипломного проєкту було реалізовано серію експериментальних односторінкових застосунків (SPA), що виконують ідентичну функціональність, але реалізовані з використанням різних засобів управління станом. Такий підхід дозволяє створити контрольоване середовище для об’єктивного порівняння характеристик чотирьох популярних технологій – Redux [2, 14], Zustand [3], Context API [10] та Recoil [6, 7] – в умовах реального програмного сценарію.

Як предмет для дослідження було обрано кейс "бронювання місць у кінотеатрі", що є універсальним прикладом застосунку, який поєднує в собі взаємодію з колекціями даних, зміну стану компонентів, оновлення інтерфейсу в режимі реального часу, обробку подій користувача та збереження списку об’єктів. Такий функціонал дозволяє всебічно перевірити, наскільки ефективно бібліотека працює з глобальним станом, які ресурси вона споживає, наскільки зручно її впроваджувати та підтримувати [8].

Кожна версія застосунку містить ідентичну структуру інтерфейсу: сітку місць для вибору, індикатори вільного, вибраного й забронюваного стану, кнопку "Забронювати" для підтвердження вибору, а також окрему панель зі списком уже заброньованих місць, які можна видалити зі списку. Таким чином, логіка взаємодії з даними є однаковою для кожної з реалізацій, що дозволяє уникнути упередженості при подальшому тестуванні.

Усі реалізації були створені на базі React [1] з використанням функціональних компонентів, хуків та однотипного підходу до структури програми. Складання інтерфейсу здійснювалося за допомогою TailwindCSS, що дозволило забезпечити єдність стилістики без надмірного навантаження на логіку компонентів. Структура застосунків включає базові компоненти App, Seat, SeatGrid, BookedList і BookButton, при цьому взаємодія між ними здійснюється відповідно до обраного методу керування станом.

У реалізації з Redux [2, 12, 13] було створено централізоване сховище store, що містить масив об’єктів місць та їх статус. Структуру було організовано через дії

(actions) та редюсери (reducers), відповідальні за зміну стану – вибір місця, підтвердження броні, скасування вибору. Компоненти отримували доступ до стану через хуки useSelector і useDispatch, які надає бібліотека react-redux. Додатково було реалізовано логіку відображення активного стану місця та зміну його кольору в залежності від статусу.

Фрагмент підключення Redux у компоненті Seat.js [17]:

```
import React from 'react';
import { useDispatch } from 'react-redux';
import { toggleSelect } from '../seatsSlice';

const Seat = ({ id, status }) => {
  const dispatch = useDispatch();

  const handleClick = () => {
    dispatch(toggleSelect(id));
  };

  const getColor = () => {
    if (status === 'booked') return 'bg-red-500';
    if (status === 'selected') return 'bg-yellow-500';
    return 'bg-green-500';
  };

  return (
    <div
      className={`w-10 h-10 m-1 cursor-pointer ${getColor()} text-white
flex items-center justify-center rounded`}
      onClick={handleClick}
    >
      {id}
    </div>
  );
};
export default Seat;
```

У версії з Zustand [3] управління станом здійснювалося через створення кастомного хука з використанням функції create. Стан було збережено у простому об’єкті, в якому функції для оновлення логіки були оголошені безпосередньо поряд із даними. Такий підхід зменшив кількість необхідного коду та дозволив напряму використовувати селектори, що суттєво зменшило кількість ререндерів. Компоненти напряму підключались до потрібних ділянок стану, що забезпечувало високу продуктивність і просту інтеграцію.

Фрагмент створення Zustand store [18]:

```
import create from 'zustand';

export const useSeatStore = create((set) => ({
  seats: generateInitialSeats(),
  toggleSelect: (id) =>
    set((state) => ({
      seats: state.seats.map((seat) =>
        seat.id === id
          ? {
              ...seat,
              status:
                seat.status === 'free'
                  ? 'selected'
                  : seat.status === 'selected'
                  ? 'free'
                  : seat.status,
            }
          : seat
      ),
    })),
```

У реалізації через Context API [10] було створено контекст, що містив масив місць та функції для його оновлення. Контекст було обгорнуто навколо застосунку, і доступ до нього здійснювався через хук useContext. Хоча цей підхід дозволив уникнути проп-дрилінгу, він має певні обмеження – при зміні стану перерендерувались усі компоненти, що підписані на контекст, незалежно від того, чи зміни торкались їх безпосередньо. Це стало очевидним у процесі аналізу повторних оновлень компонентів під час взаємодії користувача.

Фрагмент використання Context API [19]:

```
import React, { createContext, useContext, useState } from 'react';
const SeatContext = createContext();
const generateInitialSeats = () => {
  const seats = [];
  for (let row = 0; row < 5; row++) {
    for (let col = 0; col < 8; col++) {
      seats.push({ id: `${row}-${col}`, status: 'free' });
    }
  }
  return seats;
};

export const SeatProvider = ({ children }) => {
  const [seats, setSeats] = useState(generateInitialSeats());
  const toggleSelect = (id) => {
    setSeats((prev) =>
      prev.map((seat) =>
        seat.id === id
          ? {
```

```
                    ...seat,
                    status:
                      seat.status === 'free'
                        ? 'selected'
                        : seat.status === 'selected'
                        ? 'free'
                        : seat.status,
                  }
                : seat
            )
          );
        };

        const confirmBooking = () => {
          setSeats((prev) =>
            prev.map((seat) =>
              seat.status === 'selected' ? { ...seat, status: 'booked' } :
seat
            )
          );
        };

        const cancelBooking = (id) => {
          setSeats((prev) =>
            prev.map((seat) =>
              seat.id === id && seat.status === 'booked'
                ? { ...seat, status: 'free' }
                : seat
            )
          );
        };

        return (
          <SeatContext.Provider
            value={{ seats, toggleSelect, confirmBooking, cancelBooking }}
          >
            {children}
          </SeatContext.Provider>
        );
      };

      export const useSeats = () => useContext(SeatContext);
```

У версії з Recoil [6, 7] управління станом було організовано через атоми – базові одиниці збереження стану, що підключаються до компонентів через хуки useRecoilState. Селектори Recoil дозволили отримувати похідні значення (наприклад, список заброньованих місць), а структура взаємозалежностей між атомами забезпечила високу гнучкість у роботі зі складними даними. Recoil продемонстрував переваги при роботі з реактивними даними та динамічною побудовою списків.

Фрагмент використання Recoil з атомом [20]:

```
import { atom, selector ] from 'recoil';
const generateInitialSeats = () => {
  const seats = [];
  for (let row = 0; row < 5; row++) {
    for (let col = 0; col < 8; col++) {
      seats.push({ id: `${row}-${col}`, status: 'free' });
    }
  }
  return seats;
};

export const seatListState = atom({
  key: 'seatListState',
  default: generateInitialSeats(),
});

export const bookedSeatsSelector = selector({
  key: 'bookedSeatsSelector',
  get: ({ get }) => get(seatListState).filter((s) => s.status === 'booked'),
});
```

У результаті всі чотири реалізації повністю виконують задану бізнес-логіку, але відрізняються за складністю налаштування, обсягом коду, гнучкістю, продуктивністю, зручністю в розробці та кількістю повторних ререндерів компонентів. Ці прототипи стали основою для подальшого тестування, результати якого представлені в наступному розділі. Отримані дані дозволили здійснити системне порівняння бібліотек у прикладному контексті, максимально наближеному до умов реального проєктування [8, 11].

4.2 Вимірювання часу оновлення, кількості ререндерів та споживання пам’яті

У межах експериментального дослідження ключовим етапом стало проведення вимірювань, які дозволяють об'єктивно оцінити ефективність кожної з реалізованих бібліотек управління станом [15, 16]. Для цього було визначено три основні метрики, які безпосередньо впливають на продуктивність та зручність використання React-застосунків: час оновлення стану, кількість ререндерів компонентів та обсяг використаної оперативної пам’яті під час взаємодії з інтерфейсом.

Кожен із цих параметрів надає важливу інформацію:

- час оновлення стану дозволяє оцінити швидкість реакції інтерфейсу на зміну даних. Затримки в оновленні впливають на сприйняття плавності роботи застосунку та можуть створювати негативний користувацький досвід;
- кількість ререндерів демонструє, наскільки оптимізовано побудовано зв’язок між станом і компонентами. Надмірне повторне оновлення компонентів може призвести до нераціонального використання ресурсів, повільнішого рендерингу та ускладнення структури;
- використання пам’яті дозволяє визначити рівень навантаження на систему під час активного використання. Це особливо актуально для застосунків, що повинні працювати на пристроях із обмеженими ресурсами.

Для вимірювання були використані такі інструменти:

- React Developer Tools – для детального аналізу ререндерів компонентів, їх структури та залежностей від стану;
- Chrome DevTools вкладка Performance – для вимірювання часу виконання операцій оновлення інтерфейсу та обробки подій;
- Chrome DevTools вкладка Memory – для аналізу динаміки споживання оперативної пам’яті;
- React Profiler – для візуалізації процесу оновлення компонентів і кількості викликів render().

Вимірювання проводилися на одному й тому самому прототипі інтерфейсу з однаковим набором функцій: вибір місць у сітці, бронювання місць, відображення списку заброньованих місць та можливість їх скасування. Це дозволило створити однакові умови для аналізу, зосередившись виключно на відмінностях, зумовлених архітектурою управління станом.

Процедура вимірювання передбачала:

- повторення кожного сценарію дій користувача по кілька разів для усереднення результатів;

- фіксацію моментів оновлення стану та часу, що проходив до появи нових даних у DOM;
- аналіз кількості рендерів за допомогою React DevTools, з перевіркою проміжних компонентів на предмет неочікуваних оновлень;
- спостереження за споживанням пам’яті у вкладці Performance у процесі 5-хвилинної взаємодії з інтерфейсом;
- створення логів для кожної дії та побудову середніх значень метрик.

Наприкінці тестування було складено порівняльну таблицю з результатами вимірювань для всіх чотирьох реалізацій: Redux, Zustand, Context API та Recoil. Отримані дані дозволили зробити висновки щодо того, яка з бібліотек краще справляється з навантаженням, демонструє більш оптимальне оновлення інтерфейсу та економно використовує ресурси браузера. Також результати показали, наскільки сильно метод управління станом впливає на кінцеву продуктивність і зручність підтримки застосунку, що є важливим критерієм при виборі інструментів для реальних проєктів [8, 9].

4.3 Порівняльний аналіз результатів

Після завершення етапу реалізації прототипів з використанням чотирьох різних підходів до управління станом – Redux, Zustand, Context API та Recoil – було проведено серію експериментальних вимірювань, результати яких лягли в основу комплексного аналізу. Мета цього аналізу – не лише зафіксувати кількісні показники, а й виявити загальні тенденції, переваги та обмеження кожного з підходів у контексті реального використання у вебзастосунках [11, 13].

Час оновлення стану – один із найважливіших показників, що безпосередньо впливає на продуктивність користувацького інтерфейсу. Це час, необхідний для реакції застосунку на зміну стану. Вимірювання показали, що найменший середній час оновлення було зафіксовано у Recoil – 19 мс, що зумовлено використанням атомарної моделі управління станом, яка дозволяє ізолювати частини стану та оновлювати їх незалежно одна від одної [6]. Близькі результати продемонстрував Zustand – 20 мс, завдяки селекторам і мінімалістичному підходу до побудови стану

[3]. Context API і Redux трохи відстали – 22 мс і 26 мс відповідно. Це може бути пов’язано зі способом розповсюдження змін через контекст або складнішою структурою редюсерів у Redux, яка вимагає більше часу на обробку дій (див. рис. 4.1 – 4.4).

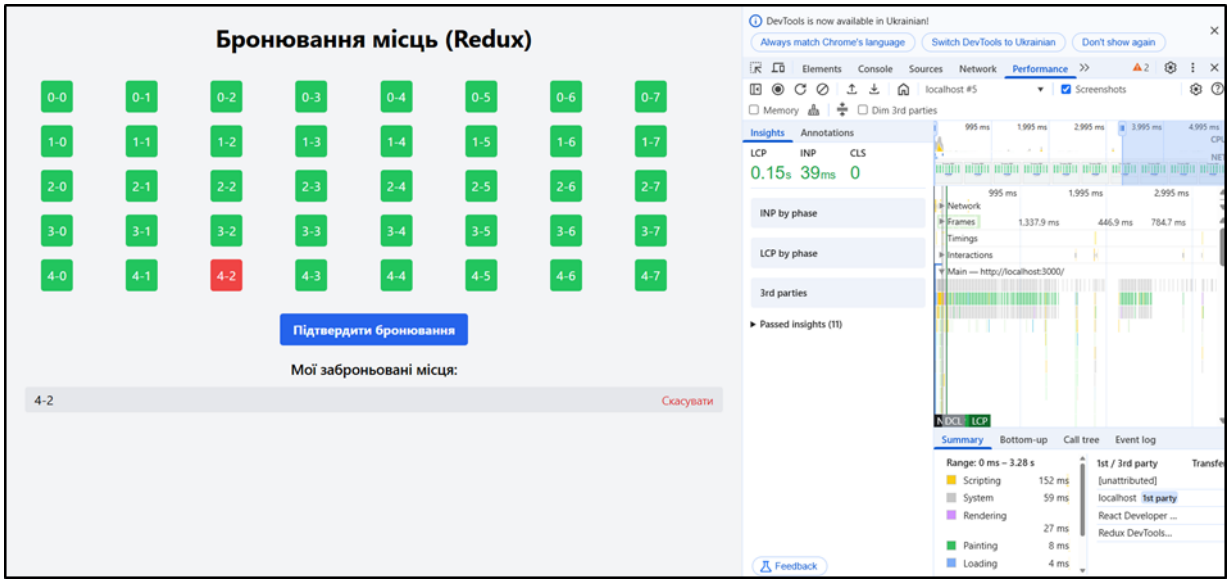


Рисунок 4.1 – Час оновлення стану у Redux (виконано самостійно)

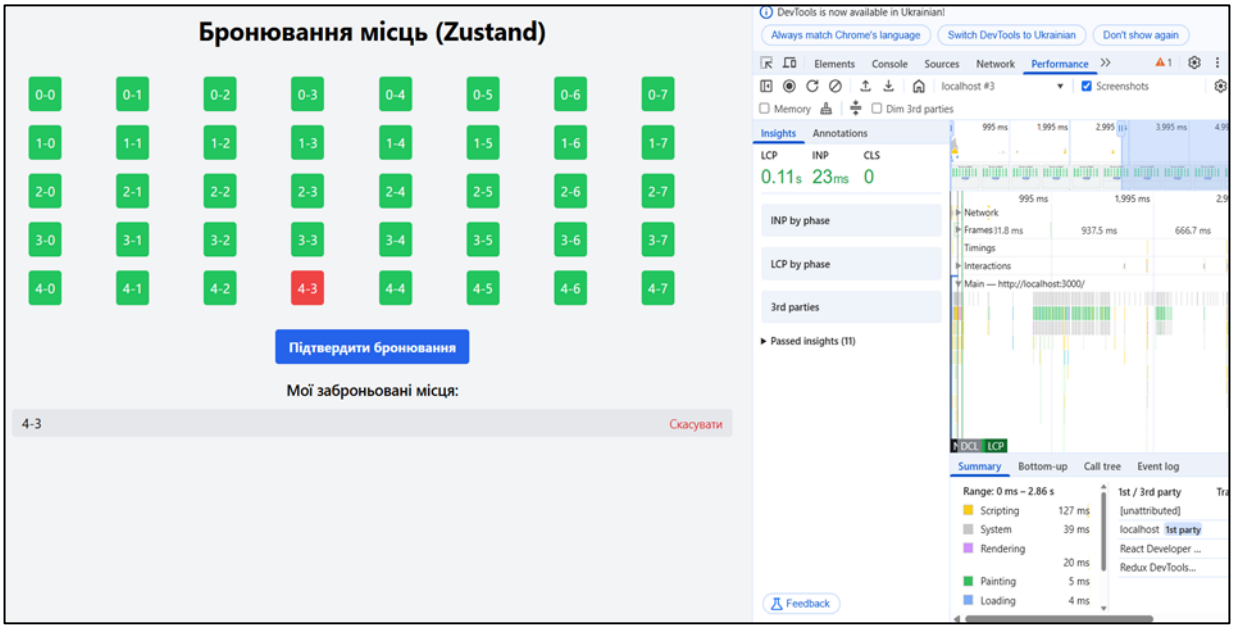


Рисунок 4.2 – Час оновлення стану у Zustand (виконано самостійно)

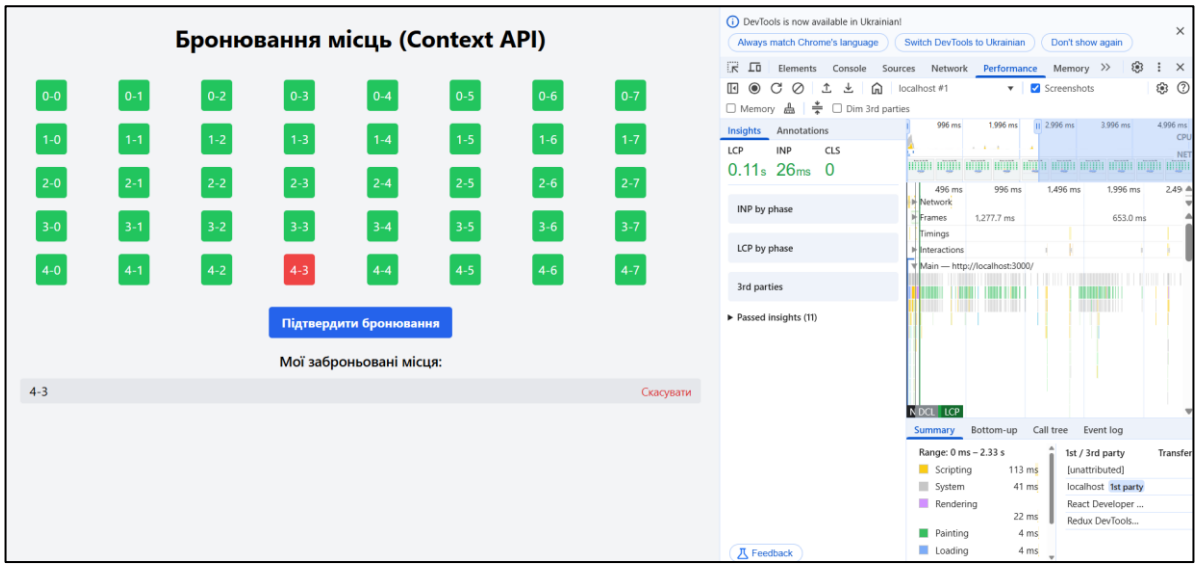


Рисунок 4.3 – Час оновлення стану у Context API (виконано самостійно)

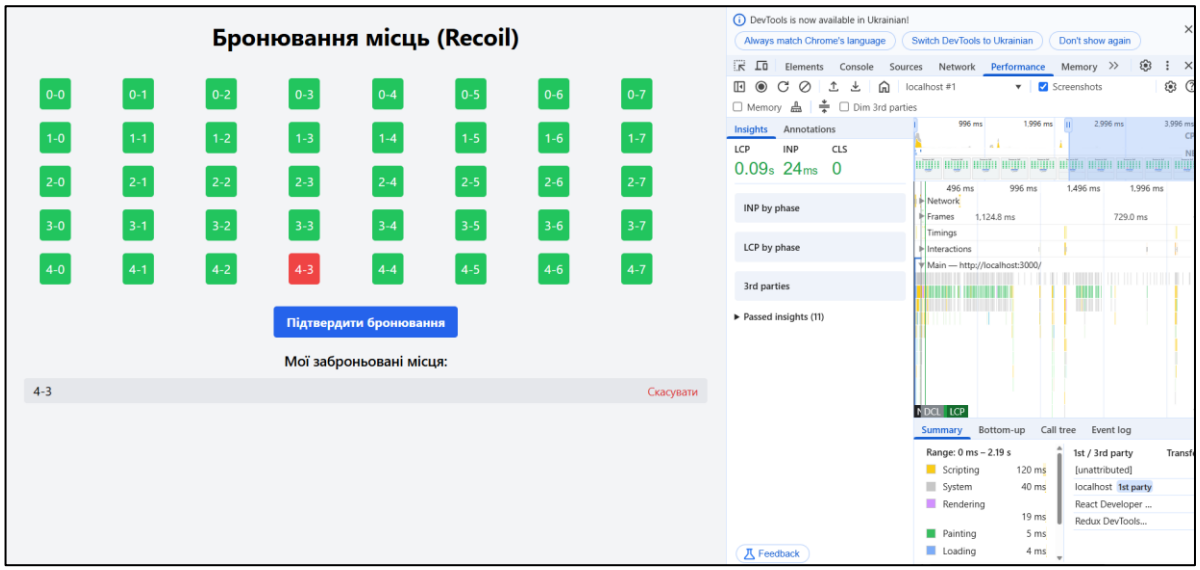


Рисунок 4.4 – Час оновлення стану у Recoil (виконано самостійно)

Кількість ререндерів. Повторні рендери мають значний вплив на продуктивність застосунку, особливо у складних інтерфейсах. Зменшення їх кількості дозволяє зберегти ресурси системи та забезпечити кращий досвід користувача. У даному експерименті всі бібліотеки, крім Redux, продемонстрували лише 2 ререндери на одну зміну стану. У Redux цей показник склав 3 ререндери, що може бути наслідком менш гнучкої прив’язки компонентів до стану та необхідності повторної побудови дерева під час змін (див. рис. 4.5 – 4.8).

Рисунок 4.5 – Рендери у Redux (виконано самостійно)

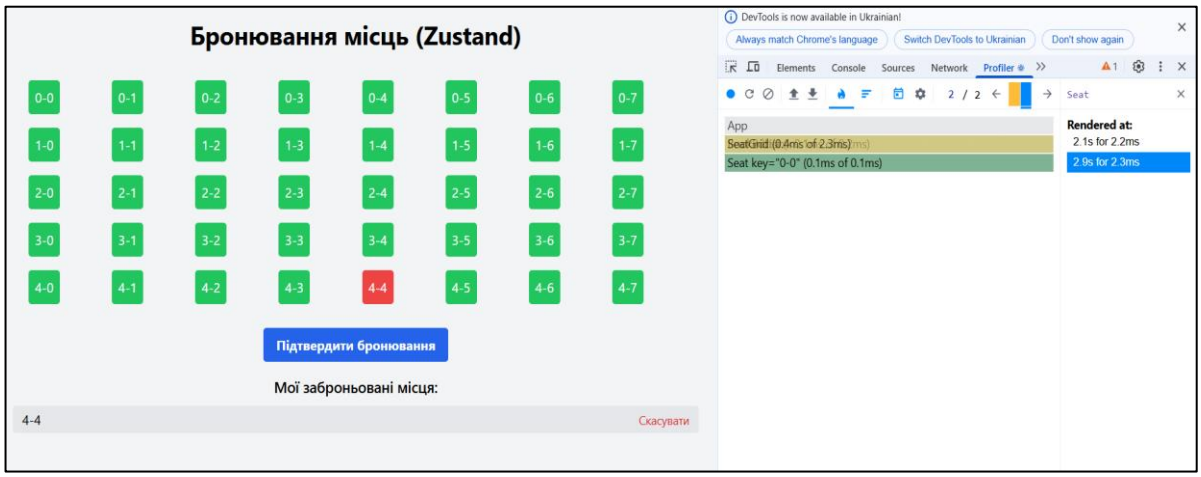


Рисунок 4.6 – Рендери у Zustand (виконано самостійно)

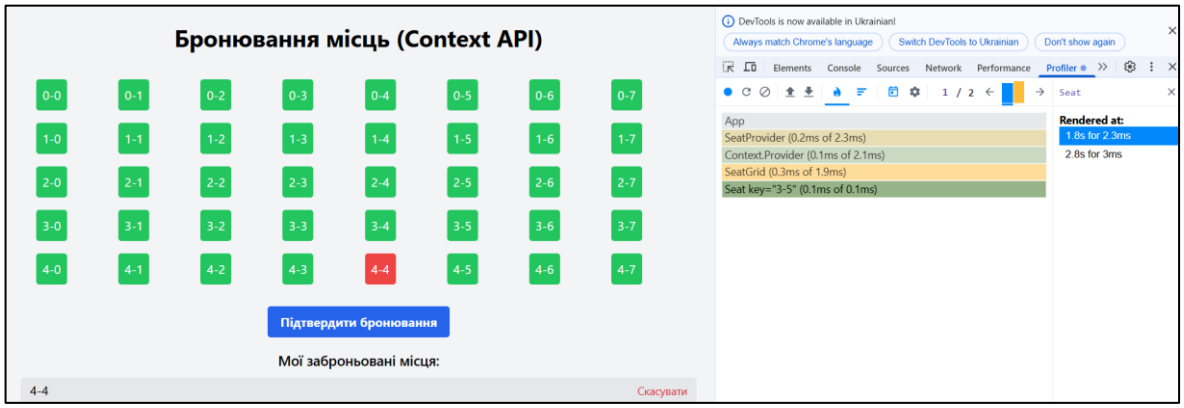


Рисунок 4.7 – Рендери у Conext API (виконано самостійно)

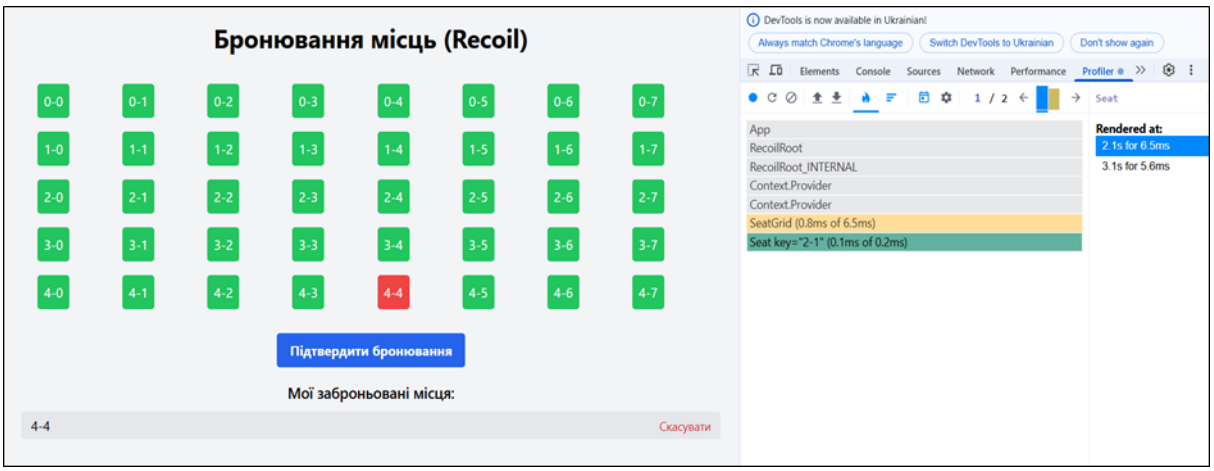


Рисунок 4.8 – Рендери у Recoil (виконано самостійно)

Споживання пам’яті. Оптимальне використання пам’яті є особливо актуальним при розробці застосунків, які повинні ефективно працювати на пристроях із обмеженими ресурсами. Найнижчий обсяг використаної пам’яті було зафіксовано у Zustand – 9,2 МБ, що ще раз підтверджує переваги його простого API та обмеженого внутрішнього оверхеду. Context API спожив трохи більше – 9,6 МБ, що також є добрим показником. Recoil і Redux продемонстрували найвищі значення – 10,3 МБ і 10,8 МБ відповідно. Це може бути пов’язано зі складнішою внутрішньою структурою та додатковими службовими об’єктами, які використовуються для керування залежностями або структурою стану (див. рис. 4.9 – 4.12).

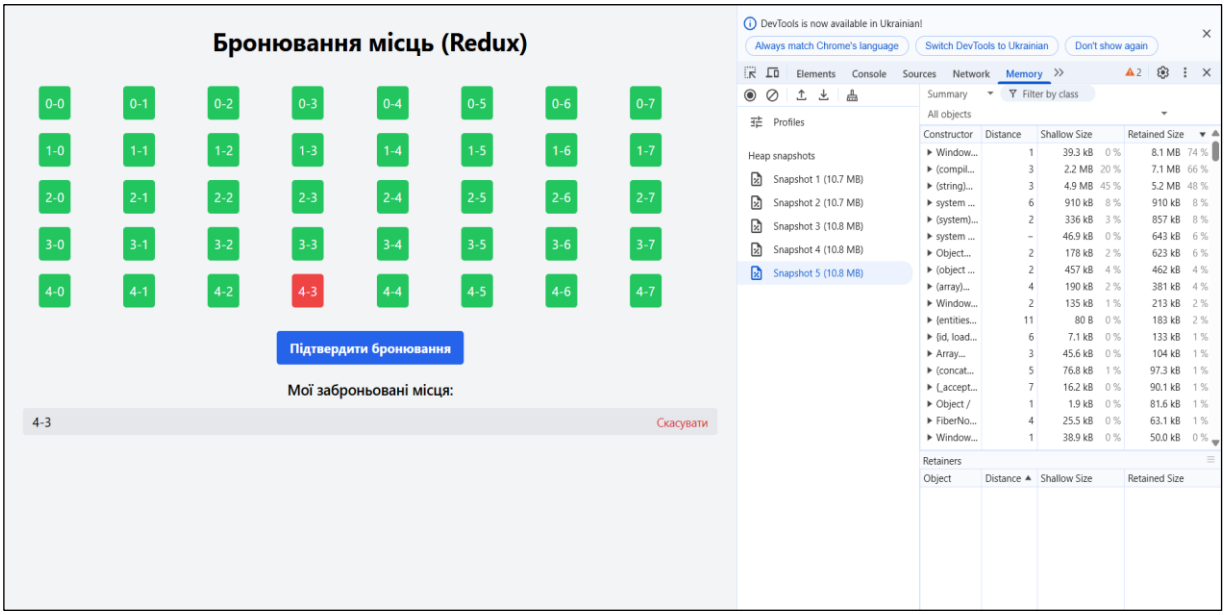


Рисунок 4.9 – Обсяг пам’яті у Redux (виконано самостійно)

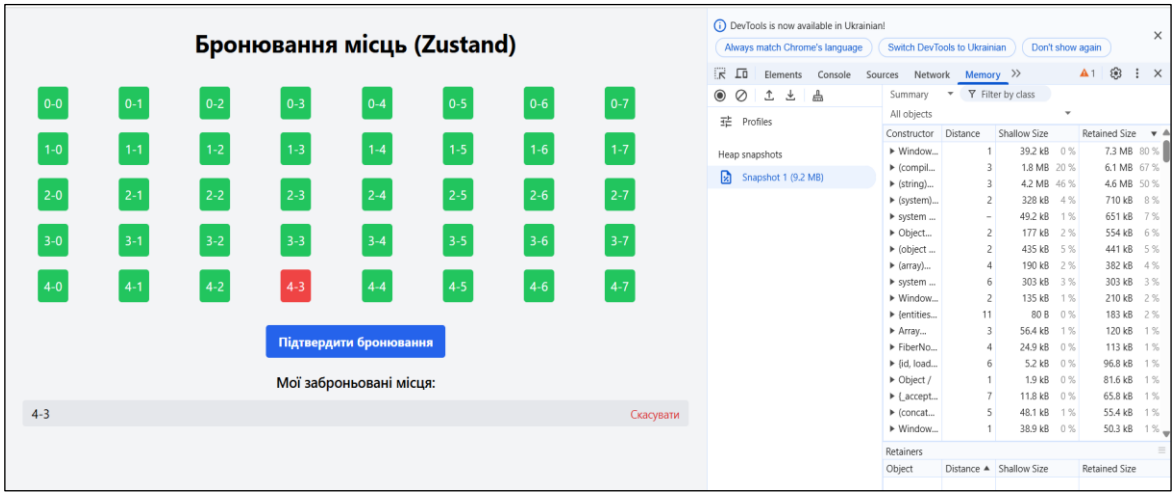


Рисунок 4.10 – Обсяг пам’яті у Zustand (виконано самостійно)

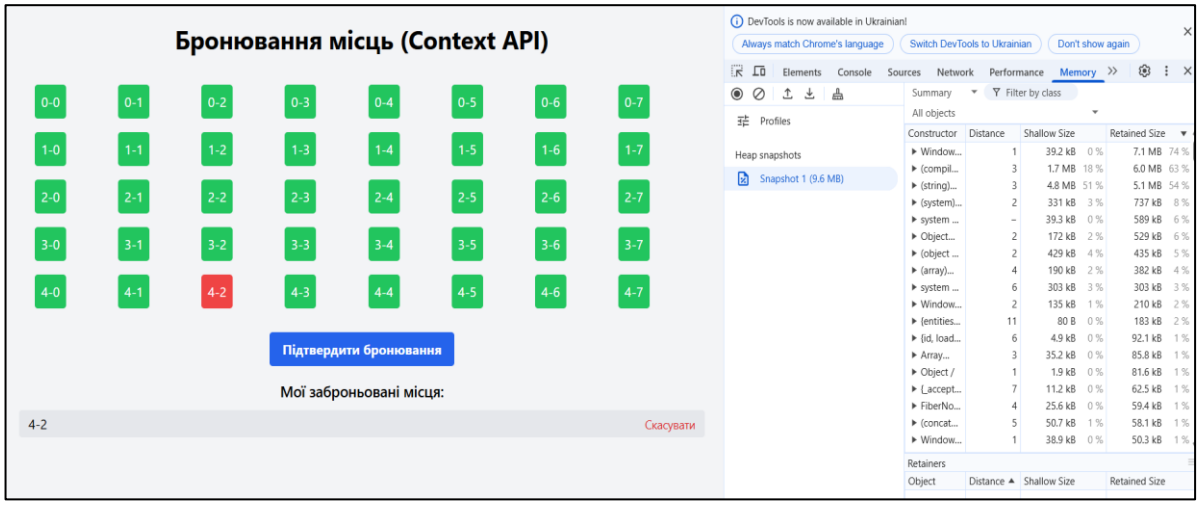


Рисунок 4.11 – Обсяг пам’яті у Context API (виконано самостійно)

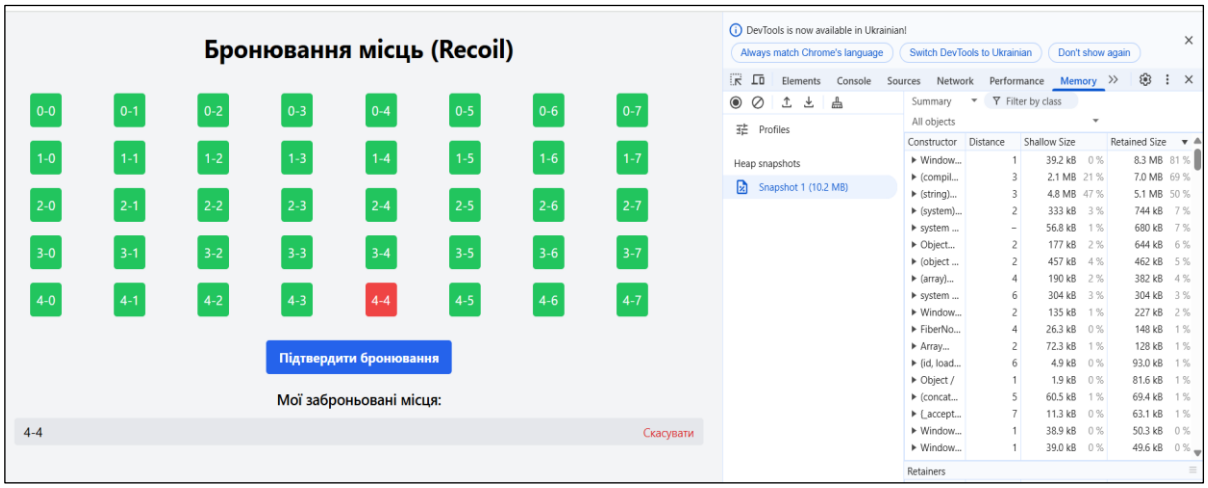


Рисунок 4.12 – Обсяг пам’яті у Recoil(виконано самостійно)

Обсяг коду. Чіткість, компактність і зрозумілість коду є важливими не тільки з точки зору розробника, а й з позицій підтримки, масштабування та навчання [9]. Найменший обсяг коду потребував Zustand – 162 рядки. Це досягається завдяки прямолінійній структурі та відсутності необхідності описувати редюсери, типи дій та додаткові обгортки. Context API і Recoil мали трохи більші значення – 173–175 рядків, що зумовлено потребою у створенні провайдерів або атомів і селекторів. Redux, як і очікувалось, вимагав найбільше – 169 рядків – через шаблонну структуру store, action і reducer (див. рис. 4.13-4.16).

Directory c:\Users\malik\Desktop\seat-reservation-redux

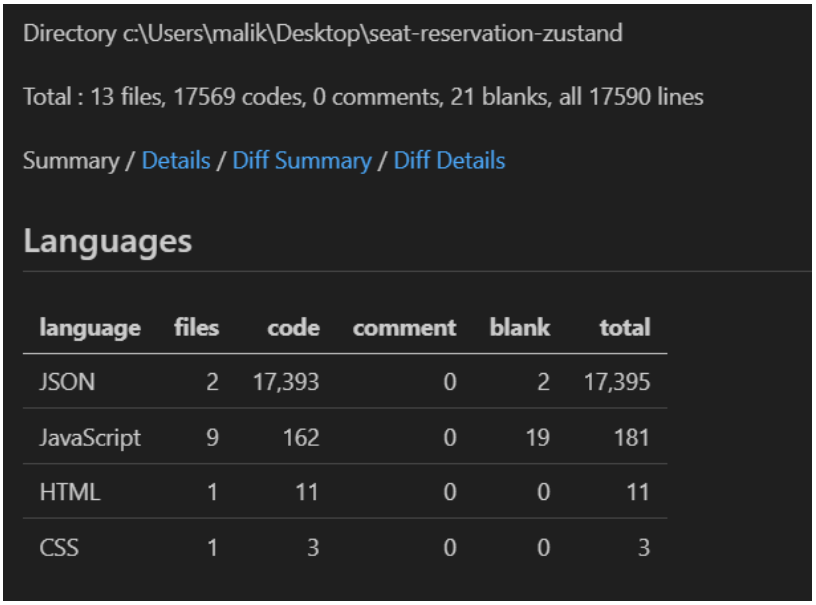
Total : 14 files, 17683 codes, 0 comments, 29 blanks, all 17712 lines

Summary / Details / Diff Summary / Diff Details

Languages

language	files	code	comment	blank	total
JSON	2	17,500	0	2	17,502
JavaScript	10	169	0	27	196
HTML	1	11	0	0	11
CSS	1	3	0	0	3

Рисунок 4.13 – Обсяг коду у Redux (виконано самостійно)



Directory c:\Users\malik\Desktop\seat-reservation-zustand

Total : 13 files, 17569 codes, 0 comments, 21 blanks, all 17590 lines

Summary / Details / Diff Summary / Diff Details

Languages

language	files	code	comment	blank	total
JSON	2	17,393	0	2	17,395
JavaScript	9	162	0	19	181
HTML	1	11	0	0	11
CSS	1	3	0	0	3

Рисунок 4.14 – Обсяг коду у Zustand (виконано самостійно)

Directory c:\Users\malik\Desktop\seat-reservation-context

Total : 13 files, 17543 codes, 0 comments, 27 blanks, all 17570 lines

Summary / Details / Diff Summary / Diff Details

Languages

language	files	code	comment	blank	total
JSON	2	17,354	0	2	17,356
JavaScript	9	175	0	25	200
HTML	1	11	0	0	11
CSS	1	3	0	0	3

Рисунок 4.15 – Обсяг коду у Context API (виконано самостійно)

Directory c:\Users\malik\Desktop\seat-reservation-recoil

Total : 13 files, 17569 codes, 0 comments, 25 blanks, all 17594 lines

Summary / Details / Diff Summary / Diff Details

Languages

language	files	code	comment	blank	total
JSON	2	17,382	0	2	17,384
JavaScript	9	173	0	23	196
HTML	1	11	0	0	11
CSS	1	3	0	0	3

Рисунок 4.16 – Обсяг коду у Recoil (виконано самостійно)

Час реалізації. Хоча вимірювання цього параметра має певну суб’єктивність, результати дають уявлення про складність інтеграції та швидкість розробки [8, 13].

Реалізація Redux-застосунку зайняла найбільше часу – 90 хвилин, що зумовлено потребою у великій кількості конфігураційного коду. Zustand знову виявився найшвидшим – 50 хв, завдяки прямому і лаконічному API. Context API та Recoil продемонстрували середній рівень складності – 60 відповідно.

Масштабованість. Цей показник було оцінено якісно, за 5-бальною шкалою [13]. Найвищий бал отримав Redux (5), оскільки він найкраще підходить для реалізації великих проєктів, завдяки своїй структурованості та чіткості. Recoil і Zustand отримали по 4 бали, оскільки дозволяють реалізовувати складні структури та забезпечують високу продуктивність, однак поступаються Redux у підтримці великої екосистеми. Context API оцінено на 2 бали, що пояснюється низькою гнучкістю в умовах складних архітектур.

Усі дані узагальнено в таблиці (див. табл. 4.1).

Таблиця 4.1 – Результати експериментального вимірювання ефективності бібліотек управління станом (таблиця виконана самостійно)

Метод	Час оновлення стану (мс)	К-сть ререндерів	Обсяг пам’яті (МБ)	Обсяг коду (рядки)	Час реалізації (хв)	Масштабованість (1–5)
Redux	26	3	10.8	169	90	5
Zustand	20	2	9.2	162	50	4
Context API	22	2	9.6	175	60	2
Recoil	19	2	10.3	173	60	4

4.4 Візуалізація результатів та їх інтерпретація

Важливою складовою експериментального дослідження є візуалізація результатів, яка дозволяє не лише побачити абсолютні числові значення, але й наочно порівняти особливості роботи різних бібліотек управління станом [15, 16]. Графіки, діаграми та скріншоти є ефективними інструментами для формулювання

висновків, виявлення трендів і особливостей поведінки бібліотек в однакових умовах.

Для візуалізації було використано такі типи графіків:

- стовпчасті діаграми – для порівняння часу оновлення стану, кількості ререндерів та споживання пам’яті;
- кільцеві діаграми – для ілюстрації розподілу обсягу коду та часу реалізації;
- теплові карти (heatmaps) – для відображення інтенсивності рендерингу (на основі Profiler).

Візуалізація часу оновлення стану (див. рис. 4.17). На графіку, що ілюструє час оновлення стану (у мілісекундах), видно, що найкращий результат продемонстрував Recoil (19 мс), тоді як Redux має найбільше значення – 26 мс. Це підтверджує, що менш складні підходи до управління станом забезпечують швидшу реакцію на зміну UI [6, 14]. Zustand і Context API продемонстрували проміжні результати – відповідно 20 і 22 мс.

На графіку (див. рис. 4.17) чітко видно лінійний спад часу від Redux до Recoil, що можна інтерпретувати як залежність між обсягом коду та швидкістю оновлення.

Кількість ререндерів (див. рис. 4.17). Стовпчаста діаграма, що показує кількість повторних рендерів, свідчить про більшу ефективність Zustand, Context API та Recoil, які демонструють лише 2 ререндери [3, 6, 10]. Redux викликає 3 повторних оновлення, що, можливо, пов'язано з логікою розповсюдження змін через Provider та передачу пропсів.

Profiler-скріншоти (див. рис. 4.5-4.8), додані до документа, ілюструють ці значення – для кожної бібліотеки було зафіксовано точну кількість оновлень конкретного компоненту.

Обсяг пам’яті (див. рис. 4.17). На графіку з обсягом використаної пам’яті видно, що Zustand (9.2 МБ) є найефективнішим за цим параметром [3]. Redux і Recoil споживають понад 10 МБ, що відповідає їхній складній внутрішній структурі. Context API займає проміжне положення.

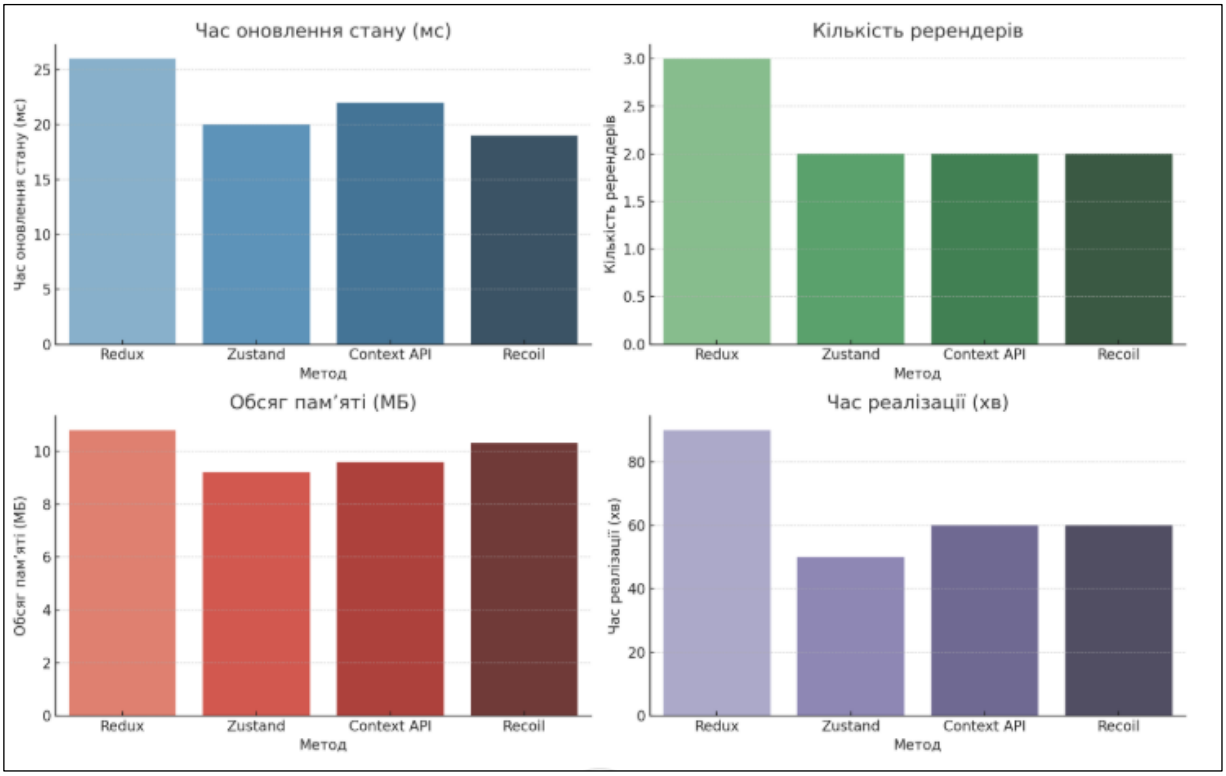


Рисунок 4.17 – Порівняння методів управління станом у React (виконано самостійно)

На діаграмі, яка ілюструє розподіл обсягу коду (див. рис. 4.18), видно, що найбільше рядків має Redux (169), а найменше – Zustand (162). Context API та Recoil приблизно однакові – 173–175 рядків. Це корелює з часом реалізації (див. рис. 4.19), який було виміряно в хвилинах: Redux – найдовше (90 хв), а Zustand – найшвидше (50 хв).

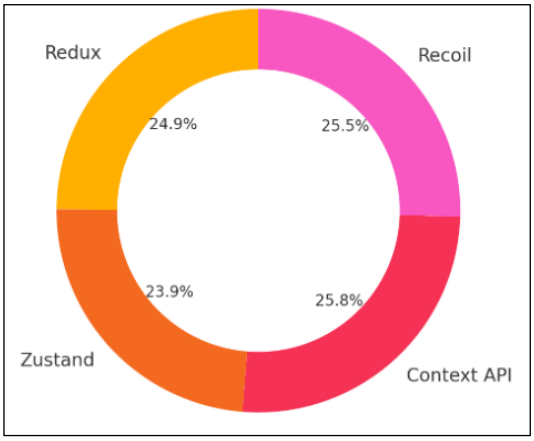


Рисунок 4.18 – Розподіл обсягу коду (рядки) (виконано самостійно)

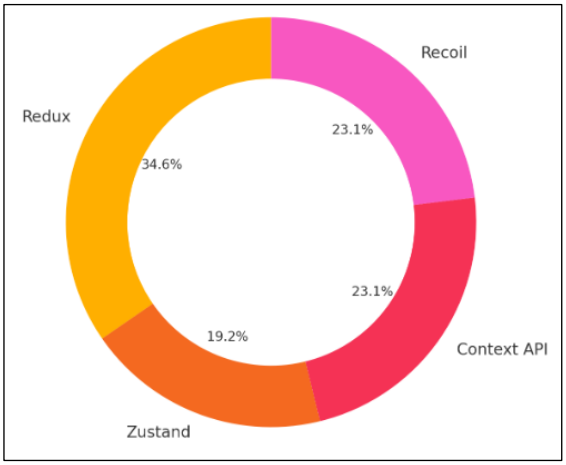


Рисунок 4.19 – Розподіл часу реалізації (хв) (виконано самостійно)

Тут варто звернути увагу на суб'єктивний фактор – розробнику простіше швидко реалізувати стан за допомогою Zustand, ніж прописувати шаблонну логіку Redux.

Для якісного критерію масштабованості було побудовано рейтингову діаграму (bar rating chart) (див. рис. 4.20).

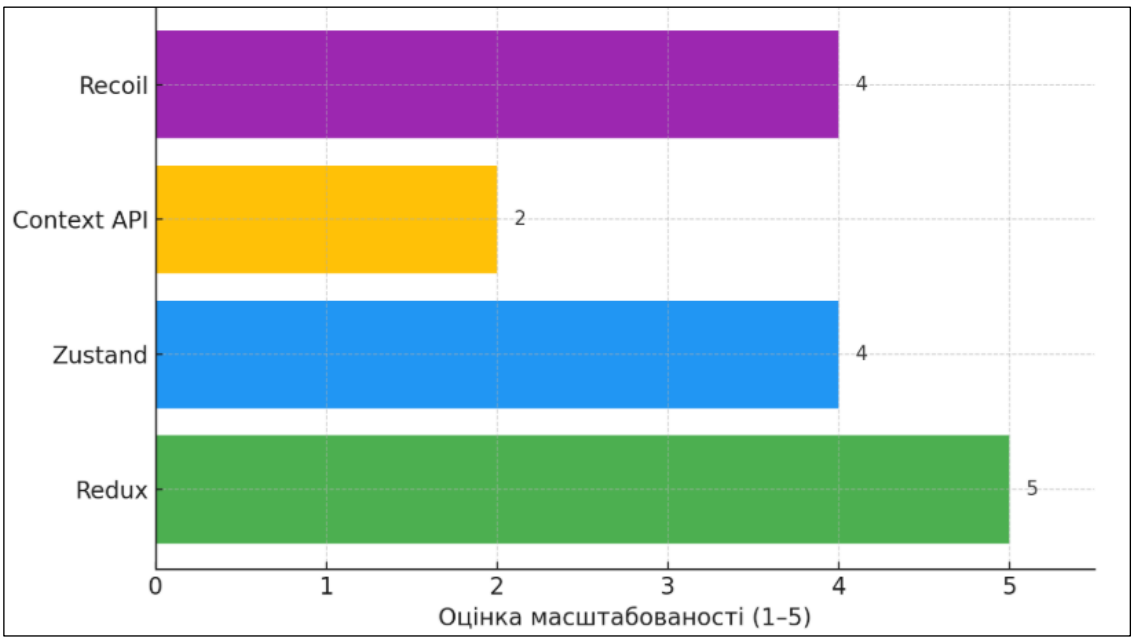


Рисунок 4.20 – Порівняння масштабованості методів управляння станом (виконано самостійно)

Найвищий бал – 5 – отримав Redux, завдяки своїй структурованості та чіткій архітектурі. Zustand і Recoil отримали по 4 бали, адже також підтримують складні

конфігурації, але з меншими ресурсними витратами. Context API було оцінено на 2 бали через його обмеження у роботі зі складним станом.

Усі отримані дані дозволяють сформувати кілька загальних висновків:

- Zustand продемонстрував найкраще співвідношення простоти, продуктивності та обсягу коду. Це робить його привабливим для малих і середніх застосунків;
- Redux залишається найпотужнішим інструментом для великих проєктів, хоча потребує більше часу на реалізацію та більше ресурсів;
- Context API – придатний тільки для простих завдань і не масштабується ефективно;
- Recoil виявився збалансованим за всіма параметрами, проте ще не досяг рівня зрілості, притаманного Redux.

Таким чином, візуалізація значно посилила аргументацію, зробила дані зрозумілими та доступними, і дозволила глибше інтерпретувати відмінності між методами управління станом у React-застосунках [9, 11].

У межах експериментального дослідження було реалізовано чотири версії одного й того самого SPA-застосунку з використанням різних підходів до управління станом: Redux, Zustand, Context API та Recoil. Проведені вимірювання продуктивності, кількості ререндерів, споживання пам’яті, обсягу коду та часу реалізації дозволили здійснити об’єктивне порівняння. Візуалізація результатів у вигляді діаграм і скріншотів підтвердила висновки аналітичного етапу, забезпечивши комплексне уявлення про ефективність кожної бібліотеки в умовах реального застосування.

ВИСНОВКИ

Дослідження методів управління станом у сучасних односторінкових застосунках на базі фреймворку React є надзвичайно актуальним у контексті зростаючих вимог до масштабованості, продуктивності та підтримуваності вебзастосунків. У ході виконання дипломного проєкту було здійснено глибокий теоретичний аналіз архітектури React, базових інструментів локального управління станом, а також таких популярних рішень для глобального керування станом, як Context API, Redux, MobX, Zustand і Recoil.

Особливу увагу приділено практичному порівнянню чотирьох підходів – Redux, Zustand, Context API та Recoil – шляхом реалізації однакових SPA-прототипів, що виконують аналогічний функціонал. Для кожної реалізації було проведено вимірювання часу оновлення стану, кількості ререндерів, споживання пам’яті, обсягу коду та часу реалізації, а також оцінено масштабованість. Застосування Chrome DevTools, React Profiler та власного логування дозволило здійснити об’єктивне й багатоаспектне тестування.

Отримані результати показали, що:

- Redux є найкращим варіантом для великих проєктів з високими вимогами до контрольованості й масштабованості, хоч і вимагає більше коду та часу реалізації;
- Zustand виявився найзручнішим і найпродуктивнішим рішенням для проєктів середньої складності, з мінімальними накладними витратами;
- Context API підходить лише для простих випадків або налаштувань, однак не забезпечує достатню продуктивність і масштабованість у великих застосунках;
- Recoil показав хорошу гнучкість у роботі з атомарним станом, проте його поточна нестабільність і обмеженість документації стримують широке використання.

Таким чином, результати роботи дозволили сформулювати обґрунтовані рекомендації щодо вибору підходу до управління станом у React-застосунках

залежно від їхніх особливостей. Проведене дослідження не лише розширює розуміння сучасних засобів frontend-розробки, але й може слугувати практичним орієнтиром для інших розробників при виборі архітектурних рішень у реальних проєктах.

Перспективами подальших досліджень може бути розгляд гібридних моделей управління станом, застосування новітніх рішень (наприклад, Jotai, Valtio) або поглиблений аналіз інтеграції з TypeScript і Server Components у React 18+.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. React Official Documentation. URL: https://reactjs.org/docs/getting-started.html (дата звернення: 01.11.2024).

2. Redux Documentation. URL: https://redux.js.org/

3. Zustand Documentation. State management made easy for React. URL: https://github.com/pmndrs/zustand (дата звернення: 01.11.2024).

4. Weststrate, M. Introduction to MobX. URL: https://mobx.js.org/

5. MobX Documentation. URL: https://mobx.js.org/README.html

6. McCabe, D. Recoil: A State Management Library for React. URL: https://recoiljs.org/ (дата звернення: 01.11.2024).

7. Recoil Documentation. URL: https://recoiljs.org/

8. Hall J. Mastering React State Management. Packt Publishing, 2022. 290 с.

9. Cherny B. Learning TypeScript: Enhance Your Web Development Skills. O'Reilly Media, 2022. 324 с.

10. Abramov, D. Context API: The Right Tool for the Job? URL: https://reactjs.org/docs/context.html (дата звернення: 01.11.2024).

11. Elliott, E. Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JavaScript Tools. O'Reilly Media, 2016.

12. Fletcher, M. The Case for Redux. Smashing Magazine. URL: https://www.smashingmagazine.com/2017/01/the-case-for-redux/ (дата звернення: 01.11.2024).

13. Snyder, E. Understanding the Redux Architecture. Professional JavaScript Frameworks: Angular, React, Vue.js. Wiley, 2018.

14. Abramov D., Clark A. Redux: Predictable State Container for JS Apps. GitHub, 2015. URL: https://github.com/reduxjs/redux

15. Smelyakov K., Prybylnov D., Martovytskyi V., Chupryna A. Investigation of network infrastructure control parameters for effective intellectual analysis. In: 14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), 2018, pp. 983–986. https://www.researchgate.net/publication/324962082.

16. Kirill, S., Pribylnov, D., Martovytskyi, V., Chupryna, A.14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering, TCSET 2018 - Proceedings, 2018, 2018-April, pp. 983–986: Investigation of network infrastructure control parameters for effective intellectual analysis; https://software.nure.ua/wp-content/uploads/2022/01/scopus.

17. GitHub Malikova Tetiana URL: https://github.com/Tatiana773/2025_M_PI_IPZzdm-23-1_Malikova_T_V_redux.git (дата звернення: 23.05.2025).

18. GitHub Malikova Tetiana URL: https://github.com/Tatiana773/2025_M_PI_IPZzdm-23-1_Malikova_T_V_zustand.git (дата звернення: 23.05.2025).

19. GitHub Malikova Tetiana URL: https://github.com/Tatiana773/2025_M_PI_IPZzdm-23-1_Malikova_T_V_context.git (дата звернення: 23.05.2025).

20. GitHub Malikova Tetiana URL: https://github.com/Tatiana773/2025_M_PI_IPZzdm-23-1_Malikova_T_V_recoil.git (дата звернення: 23.05.2025).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

15. Smelyakov K., Prybylnov D., Martovytskyi V., Chupryna A. Investigation of network infrastructure control parameters for effective intellectual analysis. In: 14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), 2018, pp. 983–986. https://www.researchgate.net/publication/324962082.

16. Kirill, S., Pribylnov, D., Martovytskyi, V., Chupryna, A.14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering, TCSET 2018 - Proceedings, 2018, 2018-April, pp. 983–986: Investigation of network infrastructure control parameters for effective intellectual analysis; https://software.nure.ua/wp-content/uploads/2022/01/scopus.