Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження продуктивності та ефективності state-менеджерів для веб-додатків на Angular
(тема)

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

Володимир КОРОБЕЙНИК
(Власне ім'я, ПРІЗВИЩЕ)

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

Керівник проф. Ігор ШУБІН
(посада, Власне ім'я, ПРІЗВИЩЕ)

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

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
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

Студентові Коробейнику Володимиру Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження продуктивності та ефективності state-менеджерів для веб-додатків на Angular»

Затверджена наказом по університету від 15.04.2025р. № 290 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 11.06.2025

3. Вихідні дані до роботи методи управління станом, OS MacOS, мова програмування JavaScript/TypeScript, фреймворк Angular, середовище розробки WebStorm

4. Перелік питань, що потрібно опрацювати в роботі

мета роботи, аналіз предметної галузі і постановка задачі, огляд та аналіз літературних джерел з дослідження, теоретичне дослідження, практичне дослідження.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	20.04.2025	*виконано*
3	Огляд й аналіз літературних джерел	25.04.2025	*виконано*
4	Постановка задачі	28.05.2025	*виконано*
5	Теоретичне дослідження	01.05.2025	*виконано*
6	Практичне дослідження	10.05.2025	*виконано*
7	Підготовка до апробації результатів дослідження. Публікація матеріалів	17.05.2025	*виконано*
8	Підготовка пояснювальної записки	20.05.2025	*виконано*
9	Підготовка презентації та доповіді	25.05.2025	*виконано*
10	Перевірка на плагіат	01.06.2025	*виконано*
11	Нормоконтроль	02.06.2025	*виконано*
12	Рецензування	03.06.2025	*виконано*
13	Попередній захист	09.06.2025	*виконано*
14	Занесення диплома в електронний архів	10.06.2025	*виконано*
15	Допуск до захисту у зав. кафедри	11.06.2025	*виконано*

Дата видачі завдання 15 квітня 2025р.

Студент _____ Володимир КОРОБЕЙНИК
(підпис)

Керівник роботи _____ проф. Ігор ШУБІН
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 76 с., 18 рис., 6 табл. 21 джерело, 4 додатки.

ВЕБ-ДОДАТКИ, ЕФЕКТИВНІСТЬ, СТЕЙТ-МЕНЕДЖЕРИ, AKITA, ANGULAR, NGRX, NGX-BASE-STATE, RxJS.

Об’єктом дослідження є ефективність і продуктивність управління станом у веб-додатках на платформі Angular.

Метою роботи є порівняння продуктивності та ефективності різних стейт-менеджерів, таких як NgRx, Akita, легковісної бібліотеки Ngx-base-state.

У результаті роботи проведено аналіз існуючих досліджень у сфері управління станом, виконано огляд найпоширеніших стейт-менеджерів для Angular, а також експериментальне порівняння їхньої продуктивності за ключовими метриками, включно з використанням ресурсів, швидкодією та простотою інтеграції.

WEB APPLICATIONS, EFFICIENCY, STATE MANAGERS, AKITA, ANGULAR, NGRX, NGX-BASE-STATE, RxJS.

The object of the research is the efficiency and performance of state management in web applications built on the Angular platform.

The purpose of the work is to compare the performance and efficiency of various state managers, including NgRx, Akita, and the lightweight library Ngx-base-state.

As a result of the work, an analysis of existing studies in the field of state management was conducted, a review of the most common state managers for Angular was performed, and an experimental comparison of their performance was carried out based on key metrics, including resource usage, speed, and ease of integration.

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, Коробейник Володимир Сергійович, студент(ка) гр. ІПЗм-23-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження продуктивності та ефективності state-менеджерів для веб-додатків на Angular», 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 ознайомлений(на) 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 – односторінковий застосунок

API – Application Programming Interface – інтерфейс прикладного програмування.

IDE – Integrated Development Environment – інтегроване середовище розробки.

UI/UX – User Interface and user experience – інтерфейс користувача та користувальницький досвід

ВСТУП

Розробка веб-додатків є однією з найважливіших галузей сучасної програмної інженерії. Для забезпечення високої продуктивності, зручності у використанні та гнучкості розробки одним з важливих аспектів є управління станом додатку. Angular, як один із провідних front-end фреймворків, надає широкі можливості для створення динамічних веб-додатків, але вибір підходу для ефективного управління станом залишається складним завданням для розробників.

Існує кілька популярних підходів до управління станом в Angular-додатках. Серед них NgRx, Akita та легковісна бібліотека Ngx-base-state. Кожен із цих підходів має свої переваги й недоліки, які впливають на швидкодію, споживання ресурсів і зручність інтеграції. Водночас, на сьогодні недостатньо досліджень, які систематично порівнюють ці підходи за ключовими метриками, такими як продуктивність та ефективність.

Управління станом відіграє ключову роль у створенні масштабованих та стабільних веб-додатків. Однак, нераціональний вибір стейт-менеджера може негативно вплинути на продуктивність системи, що є критичним фактором як для користувачів, так і для бізнесу.

Актуальність цієї роботи обумовлена необхідністю поглибленого аналізу ефективності та продуктивності різних підходів до управління станом в Angular-додатках. Це дозволить розробникам обирати оптимальні рішення для своїх проєктів, а також сприятиме розвитку методів інженерії програмного забезпечення.

Метою роботи є дослідження продуктивності та ефективності стейт-менеджерів NgRx, Akita та Ngx-base-state для веб-додатків на Angular.

Для досягнення поставленої мети, у подальшому дослідженні, необхідно виконати такі завдання:

– провести аналіз існуючих підходів до управління станом в Angular;

- визначити ключові метрики для оцінювання продуктивності та ефективності стейт-менеджерів;
- розробити тестовий веб-додаток для проведення експериментів;
- здійснити експериментальне порівняння NgRx, Akita та Ngx-base-state за визначеними метриками;
- надати рекомендації щодо використання кожного із підходів залежно від специфіки проєкту.

Об’єктом дослідження є управління станом у веб-додатках на платформі Angular.

Предметом дослідження є стейт-менеджери NgRx, Akita та Ngx-base-state, а також їхній вплив на продуктивність та ефективність роботи веб-додатків.

Методами дослідження є порівняльний а також експериментальний аналіз продуктивності за допомогою метрик, таких як споживання пам’яті, швидкодія та складність інтеграції. Дослідження також включає аналіз існуючої літератури, проектування та розробка тестового додатку для проведення дослідження.

Результати цієї роботи можуть бути використані для прийняття рішень щодо вибору оптимального стейт-менеджера в Angular-додатках, а також слугувати основою для подальших досліджень у цій галузі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Аналітичний огляд

З кожним роком веб-додатки стають все складнішими [1], однак розвиток технологій і поява нових інструментів дозволяють значно спростити їх розробку. Сучасні бібліотеки та фреймворки пришвидшують процес проєктування та впровадження складних систем, даючи змогу зосередитися на бізнес-логіці. Водночас, зростання функціональності додатків супроводжується підвищенням вимог до управління станом, що впливає як на їх продуктивність, так і на зручність підтримки.

Традиційна архітектура багатосторінкових веб-додатків (MPA), яка базується на оновленні всієї сторінки при зміні даних, поступово втратила актуальність через низьку швидкодію та відсутність плавності роботи. Її замінила концепція односторінкових веб-додатків (SPA) [2], що забезпечують завантаження даних без перезавантаження сторінки. SPA дозволяють створювати інтерактивні та динамічні інтерфейси, проте для створення швидкодіючих SPA додатків необхідно ретельно підходити до управління станом, який є важливим для забезпечення стабільності системи.

Angular [3], як один із найпопулярніших фронт-енд фреймворків, забезпечує розробників потужним інструментарієм для створення веб-додатків. Але навіть із таким широким набором інструментів, вибір ефективного підходу управління станом залишається складним завданням, особливо у великих додатках. Коли зростає кількість компонентів та взаємодій між ними, розробники стикаються з потребою обирати найбільш оптимальний підхід до управління станом, який поєднує продуктивність, масштабованість та зручність інтеграції.

1.2 Огляд існуючих підходів

Базово, у Angular для реактивності та керування станом використовується бібліотека RxJs [4]. Це потужний інструмент для роботи з асинхронними потоками даних, що дозволяє ефективно обробляти події, запити до API та взаємодію між компонентами. RxJs є основою для багатьох state-менеджерів,

включаючи NgRx, Akita та Ngx-base-state, які забезпечують вбудовану підтримку реактивних патернів програмування. Далі розглянемо кожний з наведених стейт-менеджерів більше детальніше.

NgRx [5] є одним із найпотужніших інструментів для управління станом у Angular-додатках. Цей state-менеджер базується на патерні Redux, який використовує централізовану архітектуру для управління станом (див. рис. 1.1).

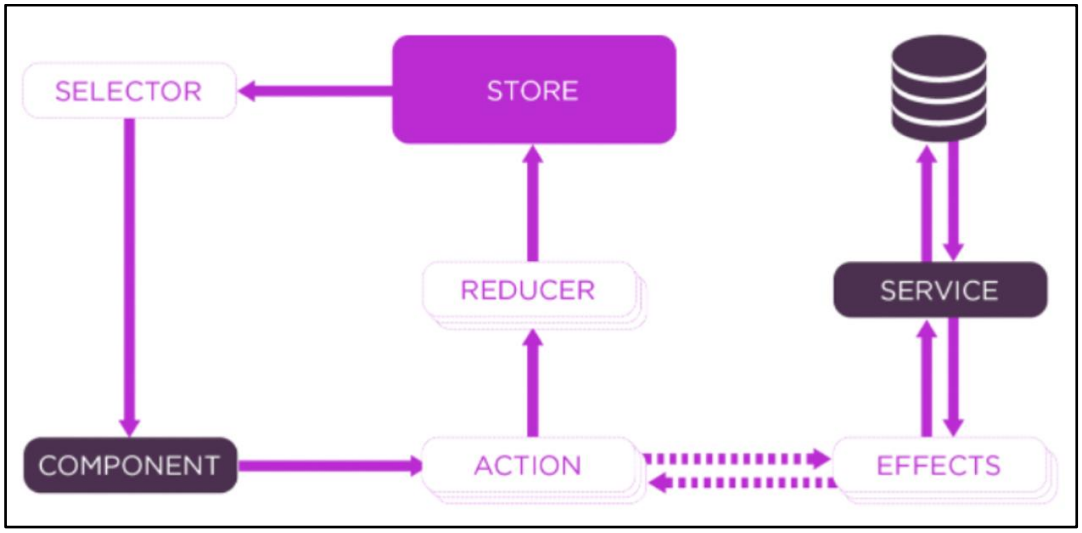

Рисунок 1.1 – Діаграма потоку даних в NgRx (за даними [5])

У NgRx усі зміни стану відбуваються через чітко визначений потік даних: компоненти генерують дії (actions), які обробляються редюсерами (reducers) для оновлення стану (state). Такий підхід забезпечує контрольованість і передбачуваність змін, що є особливо важливим для великих додатків із численними взаємозалежними компонентами.

NgRx пропонує багатий функціонал, включаючи інтеграцію з DevTools, яка дозволяє відстежувати історію змін стану, що спрощує налагодження додатка. Однак цей інструмент вимагає значного обсягу шаблонного коду, що може збільшити складність і час на початкову інтеграцію. Він добре підходить для масштабованих проєктів із великими командами розробників, але для невеликих додатків його використання може бути надмірним.

Akita [6] – це реактивний state-менеджер, створений для спрощення роботи з даними в Angular. Він базується на RxJs і пропонує зручний API для управління

станом. Akita дозволяє створювати окремі сховища (stores) для різних частин додатка, що забезпечує модульність і гнучкість у роботі зі станом (див. рис. 1.2).

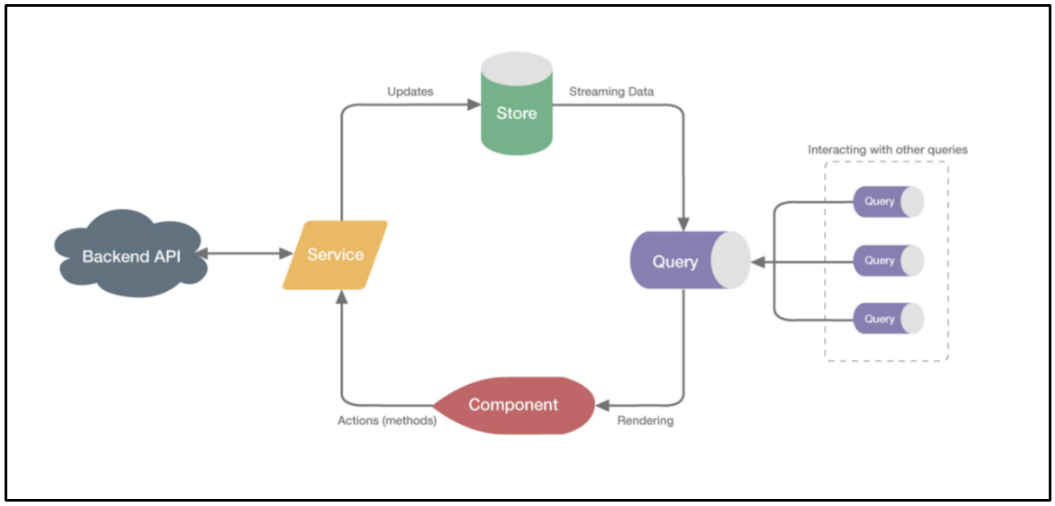

Рисунок 1.2 – Діаграма потоку даних в Akita (за даними [6])

Однією з ключових переваг Akita є зменшення обсягу шаблонного коду в порівнянні з NgRx, що робить його більш доступним для команд, які не мають великого досвіду роботи з Redux-подібними підходами. Akita також підтримує реактивну природу даних, що дозволяє зручно працювати з потоками й забезпечує автоматичне оновлення компонентів при зміні стану. Цей state-менеджер є хорошим вибором для середніх і великих додатків, де важлива модульність і простота використання. Проте в деяких випадках його універсальність може стати недоліком для вузькоспеціалізованих завдань, що вимагають централізованого контролю, як у NgRx.

Ngx-base-state [7] – це легковагова бібліотека для управління локальним станом компонентів у Angular. Вона забезпечує простий і ефективний спосіб роботи зі станом без потреби в централізованій архітектурі. Ngx-base-state дозволяє створювати локальні стейти для окремих частин проекту, що робить його ідеальним для невеликих проектів (див. рис. 1.3).

Головною перевагою цієї бібліотеки є її простота та мінімальний час інтеграції. Вона не вимагає складного налаштування, що дозволяє швидко розпочати роботу. Однак ця простота є і її обмеженням: ngx-base-state не забезпечує масштабованості, необхідної для великих додатків із численними

взаємозалежними компонентами. Ngx-base-state підходить для невеликих команд і проєктів, де основний акцент зроблений на швидкість розробки, а не на складність архітектури. У великих системах його використання може призвести до дублювання логіки та ускладнень при зростанні додатка.

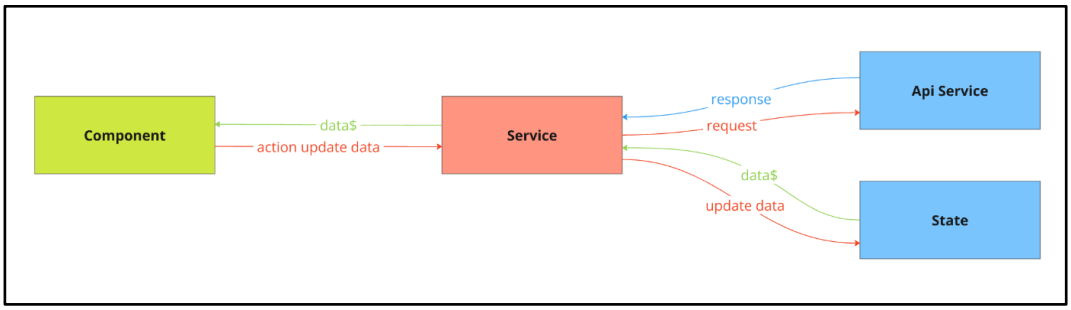

Рисунок 1.3 – Діаграма потоку даних в Ngx-base-state (рисунок створено самостійно)

Всі ці підходи активно використовуються, і порівнюються у контексті продуктивності, складності інтеграції та використання ресурсів. Вибір підходу до управління станом в Angular є важливим етапом розробки, адже продуктивність веб-додатка залежить не лише від фреймворка, але й від того, наскільки ефективно організована робота з даними. У масштабованих додатках, де структура складається з багатьох взаємопов'язаних компонентів, розробники стикаються з викликом забезпечення узгодженості та коректності стану системи. Це завдання вимагає зваженого підходу до вибору інструментів, які б поєднували гнучкість, продуктивність і зручність у підтримці.

1.3 Виявлення проблем та актуалізація рішень

Одним із ключових викликів при розробці сучасних односторінкових веб-додатків (SPA) є вибір підходу для управління станом. Складність цього процесу зростає зі збільшенням масштабів додатка та кількості взаємодій між його компонентами. Розробники стикаються з низкою проблем, пов'язаних із ефективним управлінням станом. До таких проблем належать:

– підвищення складності структури додатка. Сучасні веб-додатки часто мають багатокомпонентну архітектуру, де кожен компонент взаємодіє з

іншими через загальний стан. У великих додатках кількість таких взаємодій може значно зростати, що ускладнює забезпечення узгодженості даних і створює ризики появи помилок. Наприклад, некоректна синхронізація стану між різними частинами системи може призводити до некоректного відображення даних або збоїв у роботі функцій;

– високі витрати ресурсів на впровадження state-менеджерів. Інтеграція state-менеджерів, таких як NgRx або Akita, часто супроводжується необхідністю вивчення документації, написання великого обсягу шаблонного коду та проведення тестування. Це значно збільшує час і ресурси, необхідні для запуску проєкту. Для малих команд або проєктів із обмеженим бюджетом це може стати серйозним бар'єром;
– складність вибору оптимального інструменту для управління станом для проекту. Розробникам доступний широкий вибір інструментів для управління станом, кожен із яких має свої переваги й обмеження, через що часто важко обрати, що краще використовувати під конкретний проєкт.

Ці проблеми ускладнюють розробку та підтримку веб-додатків, особливо для команд із обмеженими ресурсами або у великих проєктах із багатьма взаємозалежними модулями.

У Angular розробникам доступний широкий вибір інструментів для управління станом, серед яких популярністю користуються NgRx, Akita та Ngx-base-state. Проте вибір оптимального рішення залишається складним завданням через відмінності у продуктивності, споживанні ресурсів та зручності інтеграції.

Огляд існуючих підходів демонструє, що кожен інструмент має свої переваги та недоліки. Наприклад, NgRx, що базується на принципах Redux, пропонує строгі правила управління станом і забезпечує масштабованість для великих додатків, але додає складності в інтеграції. Akita спрощує роботу з реактивними даними, проте може бути занадто загальним для вузькоспеціалізованих рішень. Ngx-base-state добре підходить для швидкої

розробки простих додатків, однак його можливості обмежуються потребами невеликих команд або проектів.

Актуальність цього дослідження обумовлена необхідністю оцінки продуктивності та ефективності популярних state-менеджерів, таких як NgRx, Akita та Ngx-base-state, для визначення їх оптимального використання залежно від масштабу проєкту та вимог до системи. Вибір state-менеджера має враховувати не лише продуктивність, але й потреби команди, вимоги до швидкодії, а також ресурси, доступні для інтеграції та підтримки. Неправильний вибір може призвести до зниження продуктивності, збільшення витрат на підтримку та ускладнення архітектури додатка. Тому важливо розробити рекомендації, які допоможуть розробникам і командам оптимально впроваджувати state-менеджери в своїх проєктах.

Результати роботи допоможуть:

- розроблені рекомендації допоможуть розробникам обирати оптимальні інструменти для управління станом, враховуючи специфіку проєкту, масштаб команди та вимоги до продуктивності;
- знизити складність впровадження state-менеджерів у проєктах різних масштабів, надаючи чіткі рекомендації щодо інтеграції та найкращих практик використання кожного з інструментів;
- забезпечити розробникам більш чітке розуміння, як state-менеджери впливають на масштабованість і стабільність систем, що особливо важливо для проєктів, які мають розширюватися в майбутньому.

Таким чином, дослідження продуктивності та ефективності state-менеджерів є важливим кроком у напрямку підвищення якості веб-додатків та їх конкурентоспроможності.

2 ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Огляд основних джерел

Для аналізу літератури [8], що стосується продуктивності та ефективності state-менеджерів у веб-додатках на Angular, використовувалися такі критерії відбору:

- авторитетність. Розглянуті джерела включають статті розробників, книги та офіційну документацію до бібліотек і фреймворків (Angular, RxJs, NgRx, Akita, ngx-base-state);
- актуальність. Пріоритет надавався матеріалам, опублікованим за останні 5 років, що забезпечує відповідність сучасним тенденціям у веб-розробці;
- об'єктивність. Включені джерела, що надають неупереджені оцінки та містять порівняльний аналіз інструментів управління станом;
- достовірність. Перевага надавалася роботам, що базуються на практичних експериментах, перевірених методологіях або аналізі реальних кейсів.

Для зручності аналізу всі джерела були згруповані за ключовими тематиками, що стосуються дослідження.

Основи Angular та управління станом:

- офіційна документація Angular описує базові принципи роботи з фреймворком та його інтеграцію зі state-менеджерами;
- книга “RxJS in Action” (Paul P. Daniels, Luis Atencio) [9] розглядає фундаментальні концепції роботи з потоками даних, які є основою для більшості state-менеджерів у Angular;
- документація RxJS надає практичні приклади та теоретичні основи для роботи з реактивними потоками даних.

Порівняння state-менеджерів:

- стаття “Deep Comparison of State Management Solutions in Angular” [10] проводить порівнняння NgRx, Akita та іншої легковісної бібліотеки для управління станом;

– стаття “State Management in Angular: Exploring Strategies for Effective Application Management” [11] пропонує порівняння NgRx, Akita та базового підходу із використанням Behavior Subjects з бібліотеки RxJs.

Продуктивність та масштабованість:

– офіційна документація NgRx описує архітектурні рішення, які забезпечують масштабованість і продуктивність великих додатків;
– документація Akita зосереджується на ефективності управління станом із використанням менш складного API.

Легковагові рішення:

– документація Ngx-base-state описує легкий і простий у використанні підхід для управління станом у невеликих додатках.

2.2 Аналіз літератури

Аналіз розглянутих джерел дозволив виділити ключові концепції та підходи до управління станом в Angular-додатках. Основні моделі управління станом, які аналізувалися, включають централізоване управління (NgRx), об'єктно-орієнтовані підходи з акцентом на зручність (Akita) та легковагові рішення по типу (ngx-base-state).

NgRx базується на принципах Redux і забезпечує уніфіковане централізоване управління станом, використовуючи потоки даних RxJs. Цей підхід гарантує передбачуваність стану, але вимагає значних зусиль на налаштування.

Akita пропонує більш простий API для роботи зі станом, що спрощує інтеграцію, зберігаючи високий рівень реактивності. Це підходить для середніх та великих проєктів, де важливе балансування між простотою і продуктивністю.

Ngx-base-state реалізує легкий підхід до управління станом на основі Behavior Subjects, що забезпечує мінімальну складність для невеликих додатків.

Результати попередніх досліджень:

– стаття "Deep Comparison of State Management Solutions in Angular" аналізує три бібліотеки для управління станом: NgRx, Akita та іншу

легковісну бібліотеку ng-state. Автор наголошує на перевагах ng-state для роботи з вкладеним станом, легкості використання Akita та сумісності NgRx із великими проєктами. NgRx залишається лідером для складних додатків, тоді як Akita забезпечує швидку інтеграцію в середніх масштабах, а ng-state вирізняється ефективністю при роботі з глибоко вкладеними структурами;

– стаття "State Management in Angular: Exploring Strategies for Effective Application Management" розглядає альтернативний підхід із використанням Behavior Subjects у порівнянні з NgRx та Akita. Висновки демонструють, що такі легковагові рішення, як базовий підхід з сервісами, добре підходять для невеликих команд із простими вимогами;
– книга “RxJS in Action” надає глибокий огляд роботи з потоками даних, що є основою для всіх розглянутих state-менеджерів. Вона висвітлює ключові переваги реактивного програмування для масштабованості та продуктивності;
– офіційна документація NgRx, Akita, ngx-base-state забезпечує розуміння специфічних механізмів реалізації state-менеджменту. NgRx наголошує на складній архітектурі з підвищеною продуктивністю, Akita – на зручності використання, а ngx-base-state – на простоті і легкості інтеграції.

Методи, які використовувалися у вивчених джерелах, включають практичні експерименти, порівняльний аналіз та теоретичний підхід. Практичні експерименти полягали у створенні реальних або тестових додатків для аналізу ефективності різних state-менеджерів. Порівняльний аналіз зосереджувався на оцінці переваг та недоліків кожного підходу, зокрема продуктивності, використання ресурсів і складності інтеграції. Теоретичний аналіз використовував концепції RxJS для пояснення реактивної природи state-менеджменту. Ефективність кожного підходу оцінювалася їхньою здатністю задовольнити вимоги різних сценаріїв. NgRx виявився найефективнішим для великих і масштабованих додатків завдяки строгій структурі та потужним інструментам для дебагінгу. Akita продемонстрував гнучкість і простоту, що робить його

оптимальним для середніх проєктів. Стосовно Ngx-base-state, базуючись на документації, він швидкий у впровадженні за рахунок його простоти, проте він має обмежені можливості для великих і складних додатків, Попередні дослідження акцентують увагу на важливості вибору state-менеджера залежно від масштабу та складності проєкту. RxJs є фундаментом для побудови ефективних state-менеджерів у Angular-додатках, а оцінка продуктивності, зручності інтеграції та освоєнням командою є критичними чинниками при прийнятті рішення.

2.3 Оцінка актуальності та новизни

Актуальність використаних джерел зумовлена швидким розвитком веб-технологій та постійним оновленням інструментів для управління станом у Angular-додатках. Більшість розглянутих джерел опубліковані протягом останніх п’яти років, що забезпечує їх відповідність сучасним тенденціям у веб-розробці. Зокрема, офіційна документація Angular, NgRx, Akita та ngx-base-state регулярно оновлюється, щоб включати останні зміни в архітектурі та функціоналі бібліотек.

Наукова новизна аналізованих джерел полягає в порівнянні різних підходів до управління станом, які враховують специфіку реактивного програмування, сучасні вимоги до продуктивності, гнучкості та масштабованості. Наприклад, стаття "Deep Comparison of State Management Solutions in Angular " пропонує детальний огляд трьох популярних бібліотек і надає об'єктивний аналіз їхніх сильних та слабких сторін. Важливим внеском є дослідження "State Management in Angular: Exploring Strategies for Effective Application Management", яке розглядає окрім крупних стейтменеджерів альтернативний підхід із використанням Behavior Subjects, що є цінним для невеликих команд та проєктів із мінімальними вимогами до складності.

Книга “RxJS in Action” робить вагомий внесок у розуміння основ реактивного програмування, що є критично важливим для ефективної роботи state-менеджерів. Її матеріали мають практичне значення для розробників, оскільки дають змогу краще зрозуміти роботу з потоками даних у великих масштабованих проєктах.

Попередні дослідження та документація підкреслюють ключові аспекти розробки:

– ngRx забезпечує строгість і продуктивність, що робить його ідеальним для великих додатків;
– akita орієнтована на спрощення роботи зі станом і баланс між продуктивністю та простотою, що підходить для середніх проєктів;
– ngx-base-state пропонує легковагове рішення для невеликих проєктів, яке можна швидко інтегрувати та легко освоїти.

2.4 Висновки з огляду

На основі проведеного аналізу літератури було сформовано кілька ключових висновків, що відображають сучасні тенденції та підходи до управління станом у веб-додатках на Angular.

По-перше, вибір інструмента для state-менеджменту залежить від масштабу, складності та вимог конкретного проєкту. NgRx пропонує строгість архітектури, високу продуктивність і централізоване управління станом, що робить його оптимальним для великих і масштабованих додатків. Такі проєкти часто вимагають ретельної структури, чіткої взаємодії між компонентами та детального відстеження змін у стані. Однак, ця бібліотека потребує зусиль для вивчення і впровадження, що може стати викликом для команд із меншим досвідом. Akita забезпечує баланс між простотою і функціональністю. Її легша архітектура дозволяє швидше розпочати розробку, що робить її більш придатною для середніх за розміром проєктів, які потребують достатньої гнучкості без ускладнення кодової бази. Akita підтримує об’єктно-орієнтований підхід до управління станом і надає можливість швидкого навчання завдяки інтуїтивно зрозумілому API. З іншого боку, ngx-base-state вирізняється легковаговістю та простотою. Вона розрахована на невеликі проєкти, де складність вимог мінімальна, а швидкість розробки і впровадження є основним критерієм. Її переваги включають мінімальний шаблонний код і зручність інтеграції, що дозволяє розробникам зосередитись на бізнес-логіці, а не на складнощах налаштування інструменту.

По-друге, практичне значення реактивного програмування в контексті роботи з потоками даних у веб-додатках підтверджується численними дослідженнями та прикладами. Книга “RxJS in Action” детально розглядає принципи роботи з потоками даних і підкреслює, як реактивне програмування сприяє підвищенню продуктивності та масштабованості. Використання бібліотек, таких як RxJS, дозволяє ефективно обробляти асинхронні дані, що є важливим аспектом для сучасних веб-додатків.

По-третє, більшість існуючих досліджень зосереджені на порівнянні популярних state-менеджерів, таких як NgRx та Akita, через їхню популярність і широке використання у великих проєктах. Однак, такі бібліотеки, як ngx-base-state, залишаються поза увагою при прямому порівнянні. Це створює можливу прогалину в аналізі, оскільки легковагові нові прості рішення, такі як ngx-base-state, можуть бути ідеальними для ряду сценаріїв, де важливими факторами є мінімізація складності, низький поріг входження та швидкість впровадження. Легковагові бібліотеки можуть значно скоротити час розробки в невеликих проєктах або стартапах, де розробники прагнуть швидко створити прототип або функціональний продукт. Їхній вплив на продуктивність додатка також важливий: простота архітектури та відсутність зайвого шаблонного коду можуть мінімізувати час завантаження сторінки та ресурси, необхідні для її обробки.

Таким чином, необхідність подальших досліджень полягає у детальному аналізі впливу state-менеджменту на продуктивність у реальних проєктах, проведенні прямих порівнянь із легковаговими рішеннями, такими як ngx-base-state, а також у розробці рекомендацій щодо оптимального вибору інструмента залежно від вимог до складності, продуктивності та ресурсоємності. Розглянуті джерела формують надійну базу для майбутніх досліджень, спрямованих на вдосконалення підходів до управління станом у веб-додатках та підвищення ефективності їх розробки.

3 ПОСТАНОВКА ЗАДАЧІ

3.1 Формулювання задачі

Метою даної роботи є проведення всебічного аналізу трьох популярних state-менеджерів для Angular: NgRx, Akita та ngx-base-state, з метою визначення найоптимальнішого для використання у залежності від особливостей проєкту, зручності використання та швидкодії. Завдання, які необхідно виконати:

- порівняння продуктивності. Виконати вимірювання швидкості оновлення стану, використання пам'яті та швидкості відгуку інтерфейсу. Проаналізувати результати для різних сценаріїв навантаження;
- оцінка складності інтеграції. Провести огляд необхідних кроків для налаштування кожного інструменту. Врахувати час на вивчення документації, написання коду та тестування;
- аналіз результатів. Узагальнити отримані результати для надання рекомендацій щодо вибору state-менеджера залежно від вимог проєкту, зручності використання та швидкодії.

Виконання всіх поставлених завдань дозволить мати достатню інформаційну базу для того, щоб обґрунтовано рекомендувати вибір state-менеджера для Angular-додатків, який буде оптимальний та ефективний для розробки того чи іншого програмного забезпечення. А також сприятиме кращому розумінню особливостей кожного інструменту, та стане основою для подальших досліджень у сфері оптимізації управління станом у веб-додатках.

3.2 Методи дослідження та інструменти

Виконання поставлених завдань вимагає використання комплексного підходу до дослідження, який враховує як технічні аспекти роботи інструментів, так і зручність їх застосування для розробників. Основна увага приділяється практичному тестуванню, аналізу інтеграції, що дозволяє сформувати всебічну оцінку кожного інструменту. Для реалізації поставлених завдань передбачається використання таких методів дослідження:

– експериментальний метод. Створення трьох тестових додатків із однаковим функціоналом, реалізованих за допомогою NgRx, Akita та ngx-base-state. Аналіз продуктивності буде проводитися на основі показників, таких як час оновлення стану, використання пам'яті та швидкість відгуку інтерфейсу;
– аналіз складності інтеграції. Оцінка часу та зусиль, необхідних для налаштування кожного state-менеджера, включаючи необхідність вивчення документації, написання коду та тестування;
– порівняльний аналіз. Узагальнення отриманих даних для виявлення сильних і слабких сторін кожного підходу.

Для проведення аналізу state-менеджерів важливо забезпечити належне середовище розробки та інструменти, які дозволять отримати точні результати. Було визначено програмне забезпечення, бібліотеки, обладнання та ресурси, необхідні для виконання поставлених завдань. Вибрані інструменти дозволять створити ефективні умови для тестування продуктивності та зручності кожного інструменту, а також для об'єктивного порівняння їх сильних і слабких сторін.

Для виконання роботи буде використано:

– програмне забезпечення. Для створення тестових додатків буде використано мову TypeScript [12] у поєднанні з Angular (версія 17+) для фронтенду. Для бекенду буде використано фреймворки Express або Nest для Node JS, які забезпечать обробку запитів та роботу із СУБД PostgreSQL. Інструменти для розробки включають IDE WebStorm, IntelliJ IDEA, а також Postman для тестування API-запитів. Для оцінки продуктивності роботи веб-додатку Chrome Dev Tools, WebPack Bundle Analyzer, Lighthouse, Jasmine [13] / Karma [14] / Cypress [15];
– бібліотеки. Базова бібіліотека RxJS для реалізації реактивних потоків, серед state-менеджерів будуть використовуватися NgRx, Akita та ngx-base-state, кожен із яких інтегрується з Angular для управління станом додатка;

– обладнання. У якості пристрою для розробки та тестування буде використовуватися персональний комп'ютер із процесором Apple M3 Pro, оперативною пам'яттю 36 ГБ і SSD на 512 ГБ;
– інформаційні ресурси. Використання офіційних документацій Angular, NgRx, Akita та ngx-base-state, а також статей і наукових досліджень, що стосуються state-менеджерів.

3.3 Очікувані результати

Очікується, що результати роботи дозволять об’єктивно порівняти state-менеджери NgRx, Akita та Ngx-base-state за ключовими критеріями, такими як продуктивність, складність інтеграції та зручність використання. Зібрані дані дадуть змогу сформувати практичні рекомендації щодо вибору найбільш відповідного інструменту залежно від потреб конкретного проєкту. Зокрема, передбачається:

– надання рекомендацій щодо вибору оптимального інструменту, що базуються на проведених тестах і аналізі сценаріїв використання;
– сприяння створенню більш продуктивних і швидкодіючих веб-додатків на основі Angular, для покращення досвіду користувачів.

Очікувані результати також сприятимуть подальшим дослідженням у сфері state-менеджменту та підтримуватимуть поширення кращих практик у розробці веб-додатків.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

4.1 Аналіз досліджуваних підходів

Для ефективного управління станом в Angular-застосунках сучасні розробники використовують найбільш популярні state-менеджерів такі як NgRx, Akita та ngx-base-state. Кожен із цих підходів реалізує певні патерни, які мають свої переваги та обмеження залежно від конкретних вимог до застосунку.

NgRx (Redux-патерн) реалізує строгий унідекційний підхід до управління станом, заснований на патерні Redux. У цій архітектурі стан додатка централізовано зберігається в глобальному сховищі. Зміни стану ініціюються через “Actions” (декларації дій), які обробляються “Reducers” – чистими функціями, що приймають поточний стан і дію, повертаючи новий стан. Для отримання даних із стану використовуються “Selectors”, які дозволяють оптимізувати доступ до потрібних частин стану через реактивний підхід. NgRx також підтримує “Effects”, що дає змогу управляти асинхронними операціями, такими як API-запити. Ця сувора структура допомагає розділяти бізнес-логіку від представлення, забезпечуючи передбачуваність і легкість відстеження змін. Однак значна кількість шаблонного коду, необхідного для налаштування, може уповільнити початкові етапи розробки, особливо для невеликих команд чи проєктів.

Akita (Об’єктно-орієнтована модель) використовує архітектурний підхід, орієнтований на об’єктно-орієнтовані принципи, і комбінує його з реактивним програмуванням на основі RxJS. Управління станом здійснюється через “Stores” (сховища), які представляють собою колекції об’єктів із доступними методами для оновлення даних. Для доступу до стану використовуються “Queries” (запити), що дозволяють отримувати дані через Observables, спрощуючи інтеграцію з Angular компонентами. Дії (Actions) виконуються через методи сторів, а не через окремі редуктори, як у NgRx, що знижує кількість шаблонного коду. Цей підхід забезпечує гнучкість і полегшує обробку великих структурованих даних. Завдяки меншій суворості архітектури, Akita дозволяє розробникам швидко адаптуватися

до вимог проєкту. Однак це також може створювати труднощі для підтримки єдиного підходу у великих командах, що працюють над складними додатками.

Ngx-base-state (Реактивний мінімалізм) побудований навколо ідеї мінімізації шаблонного коду, його ідея це відійти від нагромадженого паттерну redux та спростити проектування процесу управління станом. Він використовує реактивний підхід на основі RxJS, дозволяючи створювати прості й ефективні сховища стану за кілька рядків коду. Архітектура побудована на легковагових реактивних службах, які управляють станом через спостережувані потоки даних. Це дозволяє уникнути дублювання коду та забезпечує швидке впровадження. Ngx-base-state особливо корисний для невеликих проєктів, прототипів або додатків із простими бізнес-вимогами. Однак його мінімалістична архітектура може бути недостатньою для складних додатків із великою кількістю бізнес-логіки чи інтеграцій, оскільки відсутність суворої структури обмежує масштабованість.

На архітектурному рівні NgRx забезпечує строгість і контроль за допомогою унідекційного потоку даних, що особливо корисно в складних і довготривалих проєктах. Akita надає більшу гнучкість і знижує навантаження на розробника завдяки об'єктно-орієнтованому підходу, що ідеально підходить для проєктів із великими колекціями даних. Ngx-base-state, у свою чергу, забезпечує легкість у використанні й швидкість розробки, але може не відповідати вимогам до масштабованості в складних сценаріях.

4.2 Вимоги до тестового додатку

Щоб провести дослідження та оцінити вплив різних підходів до управління станом на швидкодію веб-додатка, буде розроблено тестовий односторінковий додаток типу “To-Do List” з використанням Angular. Цей додаток дозволить перевірити продуктивність, складність інтеграції та зручність використання різних state-менеджерів у контексті реального застосунку. Можна виділити такі функціональні вимог для тестового додатка:

– створення нового завдання;

– редагування існуючого завдання;
– видалення завдання;
– маркування завдання як виконаного;
– відображення списку завдань;
– можливість фільтрувати завдання за статусом (виконані/невиконані);
– пошук за назвою групи.

Також виділимо нефункціональні вимоги:

– інтуїтивно зрозумілий інтерфейс для роботи з завданнями;
– забезпечення ідентичності вигляду та взаємодії з інтерфейсом при тестуванні незалежно від обраного підходу до управління станом;
– використання одного формату та структури даних для завдань у кожному тестовому сценарії;
– інтеграція з інструментами моніторингу продуктивності, такими як Chrome DevTools та інші, для збору метрик під час тестування.

Після того як було визначено всі функціональні вимоги до додатка, для більшої наочності була побудована Smart Use case діаграма, за допомогою інструменту Miro [16], щоб візуалізувати як виглядатиме взаємодія користувача з системою (див. рис. 4.1).

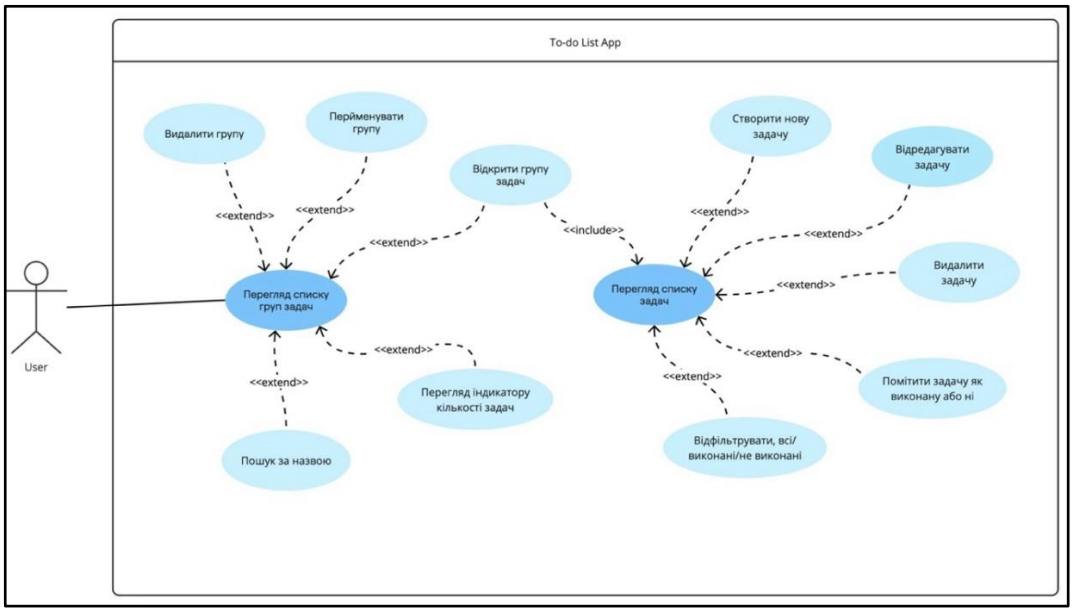

Рисунок 4.1 – Smart Use case діаграма для тестового додатку To-do list (рисунок створено самостійно)

Реалізація зазначених вимог дозволить створити універсальний тестовий додаток, який буде використовуватися для порівняння продуктивності трьох state-менеджерів: NgRx, Akita та ngx-base-state. Фіксація однакових умов і функціональності забезпечить об’єктивність та коректність результатів дослідження.

4.3 Архітектура та проектування ПЗ

Для тестового додатку To Do List було обрано та спроектовано 3-рівневу клієнт-серверну архітектуру, яка складається з клієнтського рівня, серверного рівня та рівня бази даних. (див. рис. 4.2)

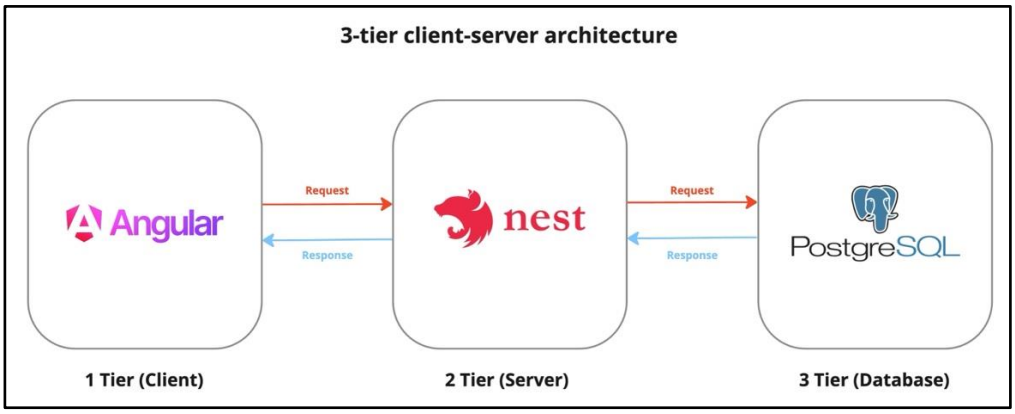

Рисунок 4.2 – 3-рівнева клієнт-серверна архітектура додатку to-do list (рисунок створено самостійно)

Клієнтський рівень буде реалізований за допомогою фреймворку Angular. Цей рівень відповідає за взаємодію з користувачем, відображення даних, обробку подій та передачу даних на сервер. На цьому рівні будуть інтегровані різні state-менеджери, такі як NgRx, Akita та ngx-base-state, для управління станом додатку, забезпечуючи реактивність і зручність роботи. Серверний рівень буде побудований на платформі Node.js з використанням фреймворку Nest.js. Він виконуватиме роль посередника між клієнтом і базою даних, забезпечуючи обробку бізнес-логіки та взаємодію з рівнем даних. Сервер також надає API для виконання CRUD-операцій із завданнями, необхідних для функціонування To Do List. Рівень даних базується на системі управління базами даних PostgreSQL. Цей

рівень відповідає за надійне зберігання даних про групи задач та самі задачі, забезпечуючи швидкий доступ до них та цілісність даних.

Клієнтська частина буде розроблюватися з використанням компонентної архітектури яка притаманна всім веб-фреймворкам включаючи Angular. Для того щоб мінімізувати дублювання компонентів та зручно керувати залежностями буде створений монорезпозиторій з використанням системи збірки Nx, в якому будуть створені 3 проекти для кожного стейтменеджеру та бібліотека shared з ui компонентами та необхідними утилітами.

Після того як були визначені функціональні вимоги до системи, та побудована smart use case діаграма, на базі цього можна побудувати приблизну архітектуру клієнтської частини, виділяючи основні компоненти та сутності. Для більшої наочності була побудована діаграма компонентів, яка ілюструє дерево компонентів веб-додатку (див. рис. 4.3).

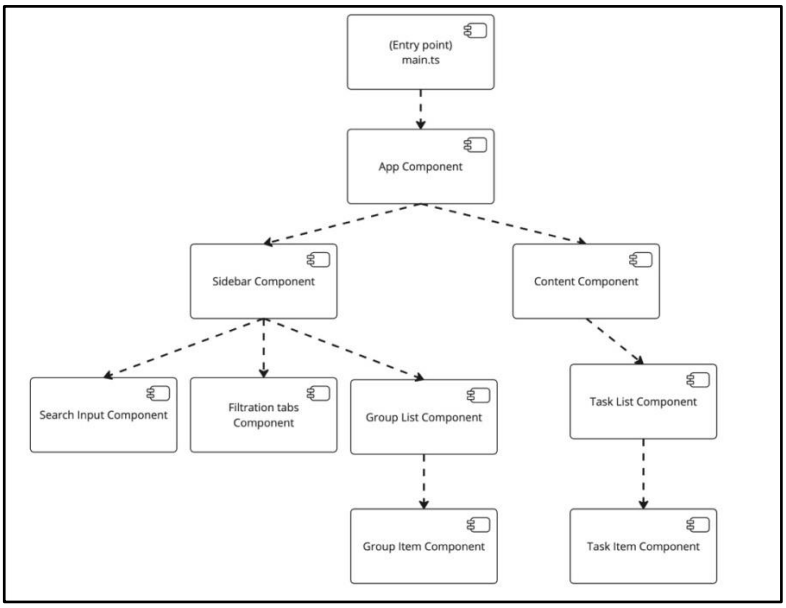

Рисунок 4.3 – Дерево компонентів тестового застосунку To Do List (рисунок створено самостійно)

Серверна частина тестового додатку буде розроблена з використанням Node.js у поєднанні з фреймворком Nest.js, який забезпечує гнучку платформу для створення RESTfull API. Реалізовуватися серверна частина буде за шаровою архітектурою, тобто шар контролерів, сервісів та репозиторіїв. Контролери

відповідатимуть за обробку HTTP-запитів від клієнта, отримуючи дані із запитів, передаючи їх у відповідні сервіси та формуючи HTTP-відповіді на основі результатів. Сервіси будуть реалізовувати бізнес-логіку додатку, працюючи як посередники між контролерами та репозиторіями, обробляючи дані перед передачею їх на зберігання або отримання з бази даних. Репозиторії, у свою чергу, відповідатимуть за взаємодію з базою даних і реалізовуватимуть CRUD-операції, використовуючи відповідну ORM для роботи з базами даних у моєму випадку Sequelize. Для функціонування тестового додатку, необхідно буде створити 2 групи ендпоінтів для сутностей “Задач” (Task), та “Груп” (Group), для яких потрібно буде реалізувати базовий CRUD, для того, щоб успішно проводити тестування.

Для зберігання даних у тестовому додатку буде використано реляційну базу даних PostgreSQL. Структура бази даних буде побудована відповідно до сутностей “Задача” (Task) і “Група” (Group), які мають зв’язок “один-до-багатьох”, тобто одна група може містити багато завдань. Для більшої наочності побудуємо ER-діаграму, щоб візуалізувати сутності та зв’язки між ними (див. рис. 4.4).

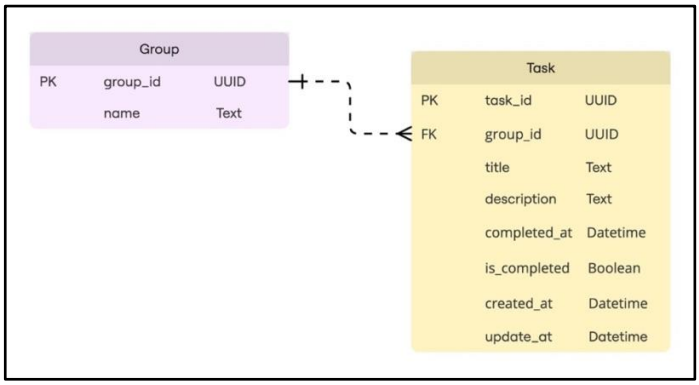

Рисунок 4.4 – ER діаграма тестового застосунку To Do List (рисунок створено самостійно)

Таблиця “Group” зберігає інформацію про групи, кожна з яких має унікальний ідентифікатор (group_id) та назву (name).

Таблиця “Task” використовується для зберігання даних про завдання. Вона містить унікальний ідентифікатор завдання (task_id), посилання на групу через

зовнішній ключ (group_id), а також такі поля, як назва задачі (title), опис (description), статус виконання (is_completed), дата завершення (completed_at), дата створення (created_at) і останнього оновлення (update_at).

Зв’язок між таблицями забезпечується через зовнішній ключ group_id у таблиці Task, що дає змогу визначити, до якої групи належить кожне завдання.

4.4 UI/UX дизайн

Після того як була спроектована архітектура тестового додатку, спроектуємо як буде виглядати візуальна частина додатку. Для розробки дизайну використовувався інструмент Figma [17]. Процес розробки дизайну проведений у два етапи: створення прототипу та на його основі розробка кінцевого дизайну. Такий підхід дозволяє створити інтуїтивно зрозумілий інтерфейс, який забезпечує ефективну взаємодію користувача.

Першим кроком було створення прототипу інтерфейсу (див. рис. 4.5). Він відображає основну структуру програми, побудовану на базі діаграми компонентів. У прототипі виділено ключові функціональні блоки: поле пошуку, панель фільтрації завдань за станом, список груп задач та основний простір для перегляду списку задач у вибраній групі. Прототип акцентує увагу на функціональних можливостях без деталізації стилів чи кольорової схеми, що дозволяє зосередитися на логіці взаємодії елементів.

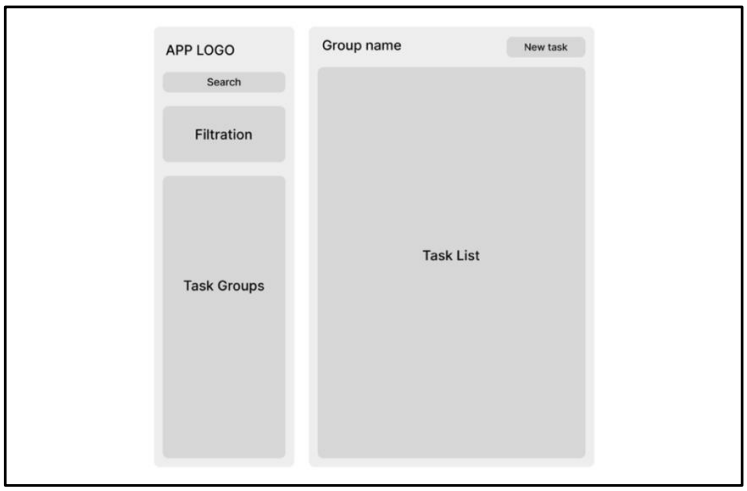

Рисунок 4.5 – Прототип інтерфейс тестового додатку To Do List (рисунок створено самостійно)

Після створення прототипу на його основі був розроблений кінцевий дизайн, у якому використано всі сучасні принципи UI/UX-дизайну, що робить додаток не лише функціональним, але й візуально привабливим. Для створення дизайну застосовано світлу кольорову схему із використанням акцентних елементів, що додають мінімалістичності та сучасності. Акцентними кольорами виділено важливі функції, як-от стан завдань та кнопка “New Task”. Обрано простий і читабельний шрифт, який змінюються залежно від значущості інформації. Головний екран логічно розділено на дві області: одна з них містить панель навігації з пошуком, фільтрацією та списком груп задач, тоді як друга зосереджена на перегляді задач і можливості додавання нових (див. рис. 4.6).

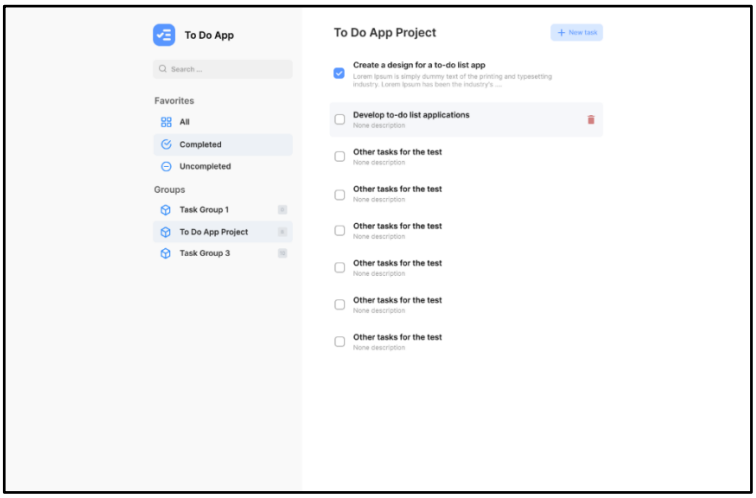

Рисунок 4.6 – Кінцевий дизайн інтерфейсу тестового додатку To Do List (рисунок створено самостійно)

У результаті отримуємо інтерфейс який забезпечує всі необхідні функції для тестування, такі як: пошук груп, фільтрації за станом (усі, завершені, незавершені), зручна робота з групами задач та перегляду списків задач. Завдяки такому підходу дизайн гармонійно поєднує естетичність, зручність використання та функціональність.

4.5 Оцінка ефективності та тестування

Для оцінки ефективності роботи досліджуваних стейт-менеджерів NgRx, Akita та ngx-base-state було розроблено детальний план тестування, спрямований

на аналіз продуктивності, зручності використання та їхньої здатності обробляти навантаження у реальних умовах. Основною метою цього етапу є створення плану оцінювання, який дозволить об’єктивно порівняти переваги й недоліки кожного з обраних інструментів.

Для тестування були визначенні ключові показники, які впливають на продуктивність стейт-менеджерів. До таких показників належать:

– час оновлення стану, яка визначає, наскільки швидко стейт-менеджер може обробляти зміни та відображати їх у додатку;
– використання пам’яті, що показує оптимізацію роботи з ресурсами;
– затримка рендера інтерфейсу користувача після зміни стану;
– показники розміру коду;
– вимірювання розміру вихідного коду;
– розрахунок відсотку дублювання коду.

Для забезпечення об'єктивності тестування були визначені ключові сценарії тестування. Вони включають:

– роботу з великим обсягом даних, що включає операції з масивами від 100 до 10 000 елементів;
– інтенсивні зміни стану, що передбачають часте оновлення даних у реальному часі.

Для проведення аналізу ефективності досліджуваних стейт-менеджерів під час тестування будуть використані сучасні інструменти для збору даних, які забезпечують точний і детальний аналіз продуктивності застосунків. Далі розглянемо кожний інструмент більш детальніше.

Chrome DevTools – потужний набір інструментів для розробників, який дозволяє виконувати моніторинг споживання пам’яті, аналіз часу рендерингу компонентів та відстеження FPS (кадрів за секунду) під час роботи додатку. За допомогою Chrome DevTools можна точно виміряти вплив різних стейт-менеджерів на продуктивність і ресурси системи.

Lighthouse – автоматизований інструмент, який використовується для оцінки затримок інтерфейсу, загальної продуктивності додатку, а також

ефективності рендерингу сторінки. Він дозволяє порівняти час відгуку інтерфейсу користувача під час різних сценаріїв взаємодії зі стейт-менеджером.

Jasmine/Karma – є ключовими інструментами для тестування Angular-застосунків. Jasmine використовується для створення юніт-тестів, дозволяє структурувати тестові сценарії та перевіряти очікувані результати. Karma, у свою чергу, виконує роль тестового раннера, запускаючи тести у реальних браузерах і забезпечуючи сумісність додатку з різними середовищами.

Метрики продуктивності [18], які будуть вимірюватися під час тестування, охоплюють найважливіші аспекти роботи додатку та стейт-менеджера:

– час оновлення стану (T_u) – це час, необхідний для виконання змін у глобальному стані додатку. Даний показник дозволяє оцінити швидкість реакції стейтменеджеру на оновлення (формула 4.1):

$$T_{update} = T_{end} - T_{start} \tag{4.1}$$

де, T_{update} – час оновлення стану,

T_{end} – час завершення оновлення стану,

T_{start} – час тригеру зміну стану;

– затримка рендерингу (L) – час між завершенням оновлення стану і моментом, коли ці зміни стають видимими у користувацькому інтерфейсі. Цей показник визначає, наскільки швидко додаток реагує на дії користувача (формула 4.2):

$$L = T_{render\_end} - T_{state\_update} \tag{4.2}$$

де, L – затримка рендерингу,

T_{render_end} – час завершення рендерингу,

T_{state_update} – час оновлення стану;

– середнє використання пам’яті (M_{avg}) – середній обсяг оперативної пам’яті, який використовується додатком під час виконання тестів. Ця метрика є важливою для оцінки ресурсоємності стейт-менеджера (формула 4.3):

$$M_{usage} = M_{usage\_after} - M_{usage\_before} \quad (4.3)$$

де M_{usage}– середнє використання пам'яті на операцію,

M_{usage_before} – загальне споживання пам'яті до виконання операції,

M_{usage_after}– споживання після виконання операції.

Для проведення дослідження була розроблена експериментальна методика для тестування, яка складається з таких етапів:

– інтеграція стейт-менеджера у тестовий додаток. На цьому етапі в тестовий застосунок реалізується функціонал To Do List з використанням кожного стейт-менеджера. Це забезпечує єдиний базис для порівняння;

– виконання тестових сценаріїв. Додаток тестується за допомогою типових операцій, таких як додавання, редагування, видалення завдань, а також виконання пошуку й фільтрації. Ці сценарії імітують стандартну поведінку користувача в реальних умовах;

– збір даних**.** Під час виконання тестових сценаріїв за допомогою Chrome DevTools, Lighthouse збираються метрики продуктивності. Зібрані дані охоплюють час оновлення стану, затримку рендерингу, використання пам’яті та інші показники;

– аналіз отриманих результатів**.** Зібрані дані аналізуються з використанням математичних формул, що дозволяє обчислити середні значення, стандартні відхилення та інші статистичні показники для кожного стейт-менеджера.

Даний підхід до тестування забезпечує повну прозорість процесу оцінки ефективності завдяки чіткому розмежуванню кожного етапу тестування та

використанню сучасних інструментів для збору та аналізу даних. Інтеграція стейт-менеджерів у тестовий додаток дозволяє створити єдину платформу для їхнього порівняння, усуваючи можливість впливу сторонніх чинників на результати. Виконання типових сценаріїв користувача, таких як додавання, редагування та видалення завдань, забезпечує моделювання реальних умов роботи додатків, що робить отримані результати релевантними для практичного використання.

Збір об’єктивних метрик продуктивності, включаючи час оновлення стану, затримку рендерингу та споживання пам’яті, гарантує, що оцінювання базується на кількісних показниках, а не суб’єктивних спостереженнях. Аналіз отриманих даних із застосуванням математичних методів, таких як обчислення середніх значень та стандартних відхилень, дозволяє зробити висновки про стабільність і ефективність кожного інструменту.

Завдяки такому підходу результати дослідження є репрезентативними, оскільки вони враховують різні аспекти роботи стейт-менеджерів у типових сценаріях використання. Це дозволяє обґрунтовано рекомендувати оптимальний стейт-менеджер для різних типів проєктів на Angular залежно від їхніх вимог до продуктивності, масштабу та складності інтеграції.

5 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ

5.1 Опис проведення досліджень

У межах даної роботи було проведено два типи вимірювань: вимірювання продуктивності та аналіз коду. Метою цих вимірювань була оцінка ефективності використання різних state-менеджерів в Angular-додатках на основі об’єктивних метрик [19] та визначення оптимальних підходів для їх інтеграції залежно від специфіки проєкту.

Для оцінки продуктивності стейт-менеджерів (NgRx, Akita, ngx-base-state) були проаналізовані наступні показники:

– час оновлення стану, визначає наскільки швидко state-менеджер може обробляти зміни. Вимірювався час між ініціацією зміни стану та його оновленням;
– використання пам’яті – оцінює, як кожен state-менеджер працює з ресурсами, включаючи споживання оперативної пам’яті під час виконання типових операцій (створення, оновлення, видалення стану);
– затримка рендеру інтерфейсу, визначає скільки часу проходить між зміною стану та його відображенням у візуальному інтерфейсі.

Крім продуктивних характеристик, також було проведено аналіз вихідного коду для оцінки складності та підтримуваності кожного state-менеджера:

– показники розміру коду. Аналізувався обсяг коду, необхідного для реалізації state-менеджменту, включаючи додаткові файли, класи, сервіси та екшени;
– вимірювання розміру вихідного коду. Порівнювався загальний обсяг коду, який додається в проєкт при використанні того чи іншого state-менеджера;
– розрахунок відсотку дублювання коду. Аналізувалася кількість повторюваних шаблонних блоків, що впливають на складність підтримки додатку.

Для збору та аналізу отриманих даних були використані інструменти Chrome DevTools, Lighthouse, Esbuild Analyze [20] а також статичний аналіз коду.

Для вимірювання метрики T_s (час оновлення стану) було проведено серію експериментів. Дослідження включало заміри часу для різних обсягів даних: 100, 1000 та 10000 елементів. Для кожного обсягу даних виміри проводилися 10 разів, після чого було розраховано середнє значення. Отримані результати представлені в таблиці. (див. табл 5.1)

Таблиця 5.1 – T_{update} (Час оновлення стану) на різних розмірах даних

Розмірність даних	NgRx	Akita	NgxBaseState
100	0.322 мс	0.097 мс	0.033 мс
1000	0.577 мс	0.114 мс	0.070 мс
10000	3.119 мс	0.446 мс	0.346 мс

Розрахунок цієї метрики виконувався за такою формулою 5.1:

$$T_{update} = T_{end} - T_{start} \tag{5.1}$$

Для вимірювання даної метрики було використано вбудовані інструменти JavaScript (console.time/console.timeEnd). Щоб мінімізувати похибку, точки початку та завершення вимірювання були обрані однакові для всіх підходів. У випадку NgxBaseState таймер запускається безпосередньо перед рядком виклику оновлення стану, і завершується одразу після виконання цієї операції, що дозволяє отримати точний час оновлення. В Akita аналогічний підхід, вимірювання починається перед місцем тригеру оновлення стану і завершується після його виконання. Для NgRx, враховуючи використання потоків дій (Actions) та ефектів (Effects), точка запуску вимірювання знаходиться перед передачею даних в екшен у ефекті, а завершення фіксується у момент отримання оновлених даних через селектор у компоненті. Таким чином було проведено максимально коректне вимірювання часу оновлення стану для кожного з state-менеджерів.

Для вимірювання метрики L (Затримка рендерингу) також було проведено серію експериментів із різними обсягами даних: 100, 1000 та 10 000 елементів. Затримка рендерингу визначає час, який проходить між завершенням оновлення стану та фактичним відображенням змін у графічному інтерфейсі користувача. Розрахунок даної метрики виконувався за такою формулою 5.2:

$$L = T_{render\_end} - T_{state\_update} \tag{5.2}$$

Дослідження проводилося шляхом використання Chrome DevTools, де аналізувалася часова шкала рендерингу (Rendering) для фіксації моменту оновлення віртуального DOM та оновлення інтерфейсу. (див. рис. 5.1)

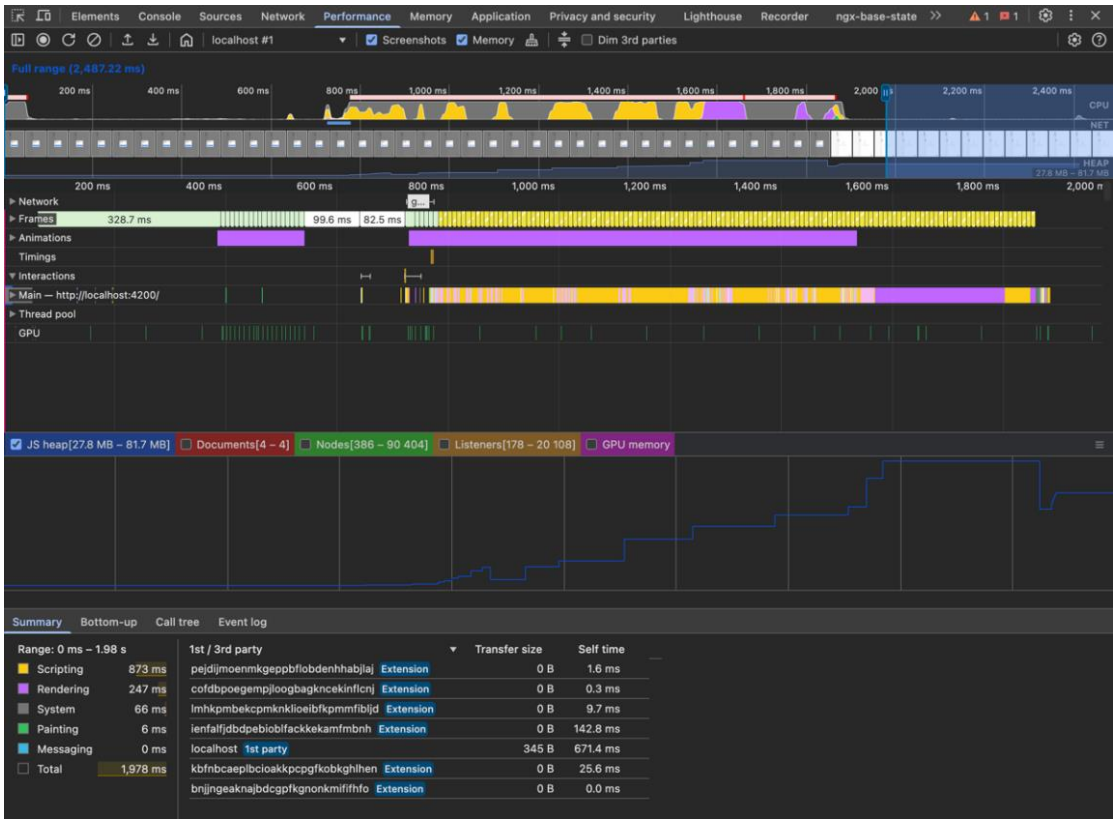

Рисунок 5.1 – Вкладка Performance в Chrome Dev Tools (рисунок створено самостійно)

Вимірювання виконувалися 10 разів для кожного розміру даних, після чого було розраховано середнє значення. Отримані результати наведені в таблиці. (див. табл 5.2)

Таблиця 5.2 – L (Затримка рендерингу) на різних розмірах даних

Розмірність даних	NgRx	Akita	NgxBaseState
100	4 мс	4 мс	4 мс
1000	23 мс	21 мс	21 мс
10000	240 мс	232 мс	228 мс

Для вимірювання метрики M_{usage} (Використання пам’яті) було також проведено серію експериментів із різними обсягами даних, та розраховано середнє значення для кожного з стейтменеджерів. Розрахунок цієї метрики виконувався за такою формулою 5.3:

$$M_{usage} = M_{usage\_after} - M_{usage\_before} \tag{5.3}$$

Замір метрики проводився шляхом використання Chrome DevTools (вкладки “Performance” та “Memory”), де аналізувалося споживання пам’яті (JS Heap size) у різних сценаріях виконання. Для кожного розміру даних проводилися знімки стану пам’яті (snapshot) до та після виконання операції додавання записів у state-менеджері. (див. рис. 5.2)

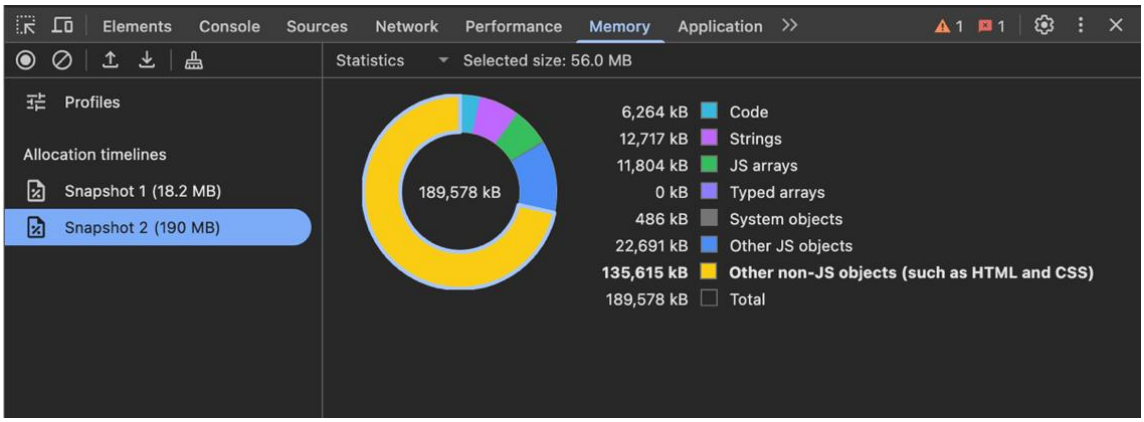

Рисунок 5.2 – Вкладка Memory в Chrome Dev Tools (рисунок створено самостійно)

Вимірювання виконувалися 10 разів для кожного розміру даних, після чого було розраховано середнє значення. Отримані результати наведені в таблиці (див. табл. 5.3).

Таблиця 5.3 – M_{usage} (Використання пам’яті) на різних розмірах даних

Розмірність даних	NgRx	Akita	NgxBaseState
100	2.8 МБ	2.8 МБ	2.7 МБ
1000	18.4 МБ	18.6 МБ	18.3 МБ
10000	171.8 МБ	170.8 МБ	171.3 МБ

Після того як було зібрано показники продуктивності стейт-менеджерів, далі було виконано аналіз коду який був необхідний для реалізації керування станом у цих тестових додатків.

Для вимірювання метрики “Показники розміру коду” було проведено аналіз частини коду тестових додатків. Метою цього вимірювання було визначення обсягу коду, необхідного для впровадження state-менеджменту у проєкті. У кожному тестовому додатку були визначені файли, що безпосередньо стосуються реалізації керування станом додатку. Аналіз охоплював усі сутності проєкту (групи та задачі), включаючи файли store, state, actions, reducers, selectors, queries, services та інші допоміжні елементи в залежності від архітектури стейт-менеджера. Такий підхід дозволяє оцінити, наскільки кожен state-менеджер впливає на складність проєкту. Отримані результати наведені в таблиці. (див. табл. 5.4)

Таблиця 5.4 – Показники розміру коду

Стейт-мендежер	Кількість файлів	Загальна кількість рядків (LoC)
NgRx	16	861
Akita	8	406
NgxBaseState	8	280

Для отримання показників був проведений підрахунок кількості файлів, що містять код, необхідний для роботи state-менеджера, а також загальної кількості рядків коду (LoC). Для цього використовувалися команда wc -l у терміналі (для підрахунку рядків у файлах) та інструментарій самої IDE WebStorm. Після збору

даних було отримано загальну кількість файлів та рядків коду для кожного state-менеджера.

Для вимірювання метрики “Показники розміру вихідного коду” необхідного для керування станом, було проведено аналіз як початкового коду перед збіркою, так і його розміру у фінальному бандлі після оптимізації. Це дозволило визначити, скільки місця займає код state-менеджера у загальному розмірі додатку, а також його вплив на кінцевий розмір бандлу.

Спочатку було виконано підрахунок показників розміру вихідного коду до збірки, включаючи всі файли, що відповідають за керування станом. Далі було проведено збірку кожного з проєктів за допомогою вбудованих інструменів Nx, після чого виконано аналіз фінального бандлу. Для аналізу бандлу використовувався інструмент Esbuild Analyze, який дозволяє переглянути структуру бандлу та визначити, який обсяг займає код кожного state-менеджера у фінальному JS-файлі. (див. рис. 5.3)

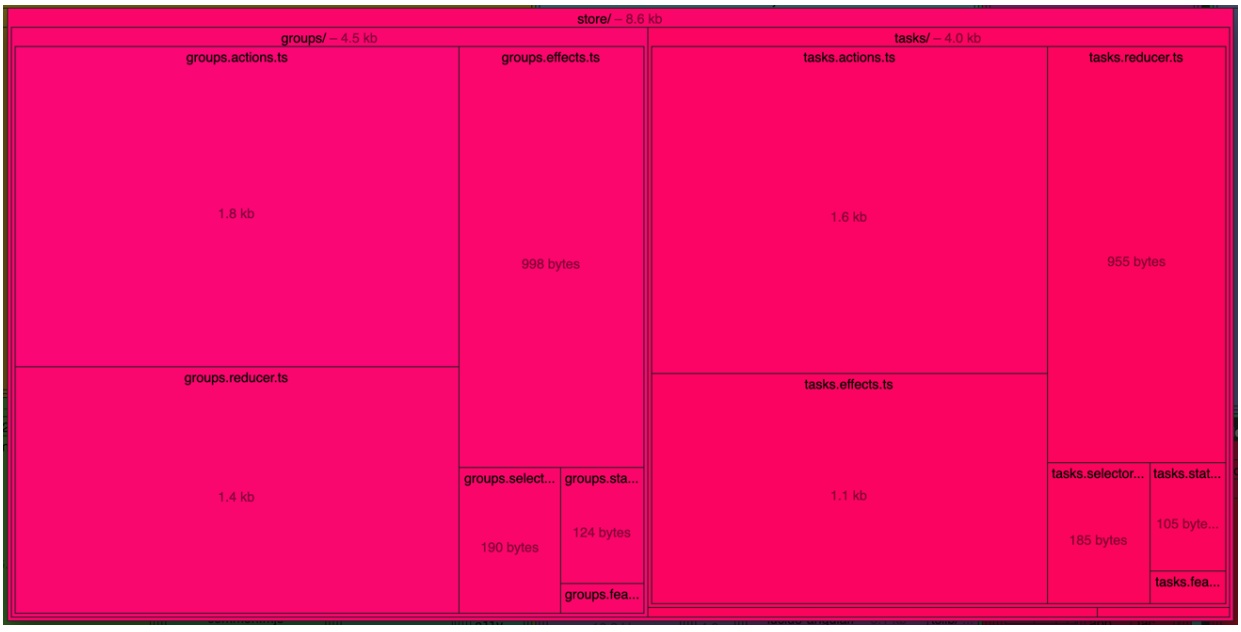

Рисунок 5.3 – Схема бандлу store в Esbuild Analyze (рисунок створено самостійно)

Було визначено розмір коду керування станом після збірки, а також розраховано відсоток цього коду від загального розміру бандлу, що дозволяє оцінити, наскільки кожен state-менеджер впливає на кінцевий розмір додатку. Отримані результати наведені у таблиці нижче. (див. табл. 5.5)

Таблиця 5.5 – Показники розмірів вихідного коду керування станом

Стейт-менеджер	Початковий код (до бандлу)	Розмір у бандлі (після збірки)	Відсоток від усього бандлу
NgRx	34 КБ	10.8 КБ	2.75%
Akita	18 КБ	5.9 КБ	1.59%
NgxBaseState	7.9 КБ	4.4 КБ	0.99%

Для вимірювання метрики “Відсотку дублювання шаблонного коду” в реалізації керування станом було проведено аналіз коду кожного state-менеджера з метою визначення повторюваних структур. Це дозволило оцінити, наскільки кожен із підходів вимагає написання однотипного коду, що може впливати на підтримку та масштабованість проєкту.

Аналіз проводився в кілька етапів. Спочатку код було розбито на категорії відповідно до архітектури state-менеджера, виділяючи такі сутності як actions, reducers, effects, queries, stores, services тощо. Далі для кожної категорії підраховувалася загальна кількість рядків коду, після чого виділялися шаблонні частини – рядки, які повторюються з мінімальними змінами (наприклад, декларація екшенів у NgRx, базові структури сторів в Akita, реактивні сервіси в NgxBaseState). Після того як було виділено шаблонні рядки розраховувався відсоток дублювання, який визначався як відношення кількості шаблонних рядків до загальної кількості рядків, помножене на 100 для отримання відсоткового значення. Такий підхід дозволяє оцінити рівень шаблонного коду для кожного стейтменеджеру та порівняти, наскільки кожен state-менеджер сприяє повторюванню схожих структур. Отримані результати наведені у таблиці нижче. (див. табл. 5.6)

Таблиця 5.6 – Відсоток дублювання шаблонного коду

Метрика	NgRx	Akita	NgxBaseState
Відсоток дублюючого шаблонного коду	59.8%	39.4%	35.9%

5.2 Аналіз результатів досліджень

У ході дослідження було проведено порівняння 3-х стейтменеджерів (NgRx, Akita, NgxBaseState) з точки зору продуктивності, ефективності та зручності використання. Для отримання об’єктивної оцінки кожного з них було проведено серію експериментів та вимірювань за різними метриками [21] які дозволяють оцінити вплив вибраного підходу на продуктивність додатка, складність реалізації та підтримку коду.

Насамперед було проаналізовано час оновлення стану для кожного зі стейт-менеджерів, оскільки ця метрика безпосередньо впливає на продуктивність додатка та плавність його роботи. (див. рис. 5.4)

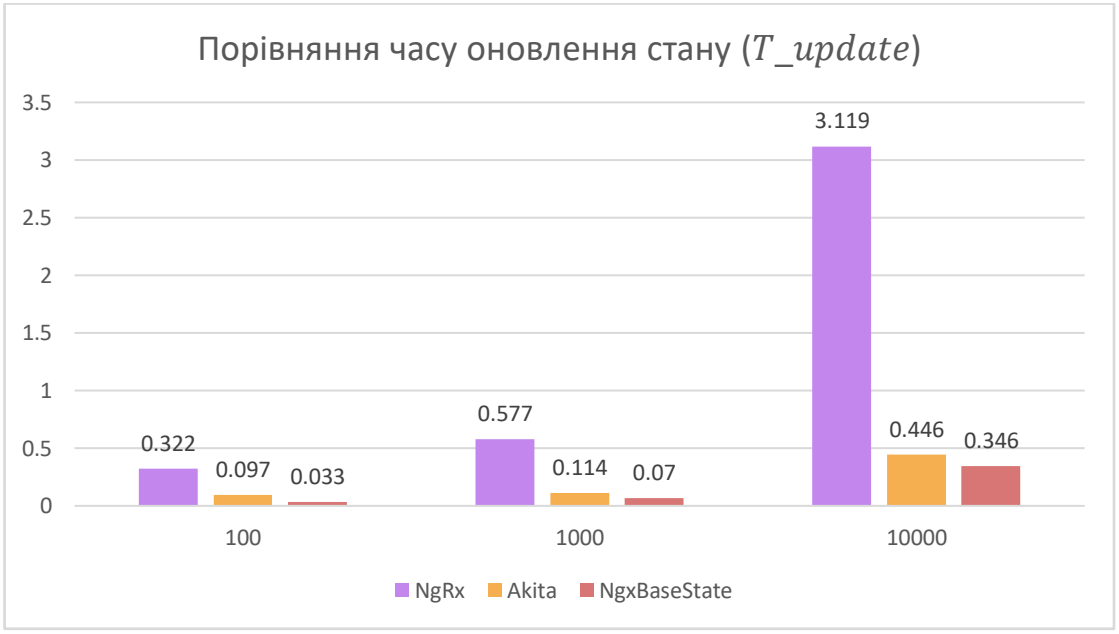

Рисунок 5.4 – Графік порівняння часу оновлення стану стейтменеджерів (рисунок створено самостійно)

Аналіз результатів показав, що NgRx демонструє значно гіршу продуктивність, що, ймовірно, пов’язано з його архітектурою, заснованою на потоковому підході з використанням Redux-подібного зберігання стану. Ця модель передбачає створення нових об’єктів стану після кожної зміни, що може призводити до значних витрат пам’яті та зниження швидкодії при роботі з

великими наборами даних. Крім того, використання RxJS та складної системи ефектів ускладнює оптимізацію оновлень.

Akita та NgxBaseState демонструють кращі результати, оскільки вони мають більш гнучкий підхід до управління станом. Akita використовує менш складну архітектуру, уникаючи зайвих перетворень стану, що дозволяє зменшити затримки при оновленні даних. NgxBaseState, у свою чергу, забезпечує ефективну роботу завдяки легковагому підходу до управління станом, мінімізуючи кількість змін та операцій над об’єктами.

При збільшенні розміру даних розрив між NgRx і конкурентами стає ще більш помітним, що свідчить про його недостатню масштабованість без додаткової оптимізації. Водночас Akita та NgxBaseState зберігають відносно стабільну швидкодію завдяки ефективнішому механізму оновлення стану.

Ще не менш важливим параметром, що впливає на сприйняття швидкодії веб-застосунків, є затримка рендерингу (L). Ця метрика відображає, скільки часу проходить від моменту зміни стану до повного оновлення інтерфейсу користувача. Чим нижче значення цієї метрики, тим швидше користувач отримує візуальний відгук на свої дії, що напряму впливає на досвід користувача. (див. рис. 5.5)

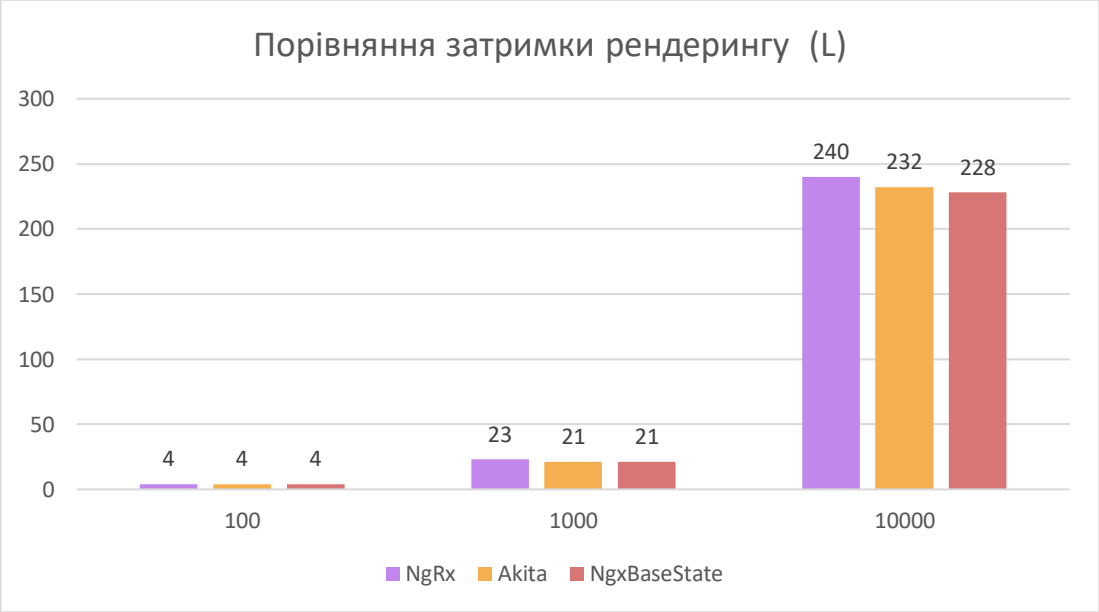

Рисунок 5.5 – Графік порівняння затримки рендерингу стейтменеджерів (рисунок створено самостійно)

Дивлячись на результати можна сказати, що на малих обсягах даних (100 записів) всі стейт-менеджери працюють однаково швидко, демонструючи мінімальну затримку рендерингу. Однак зі збільшенням розміру даних починають проявлятися помітні відмінності. При 1000 записах затримка рендерингу зростає, але все ще залишається відносно низькою: NgRx має найгірший показник (23 мс), у той час як Akita та NgxBaseState демонструють дещо кращу продуктивність (21 мс).

Найбільш значущі відмінності помітні при 10 000 записах. В усіх трьох випадках спостерігається значне зростання затримки рендерингу, що пов’язано з обробкою великої кількості змін у DOM. NgRx показує найвищу затримку (240 мс), що, ймовірно, зумовлено його підходом до обробки оновлень через потоки даних та необхідність створення нових об’єктів стану при кожному оновленні. Akita (232 мс) та NgxBaseState (228 мс) виявилися трохи ефективнішими, що може бути наслідком їхньої більш оптимізованої роботи з оновленням стану без зайвих перетворень а також простішою архітектурою.

Загалом результати свідчать про те, що всі три стейт-менеджери забезпечують прийнятну продуктивність на малих обсягах даних. Однак із ростом кількості записів вплив затримки рендерингу стає більш помітним. NgRx демонструє найгіршу масштабованість у цьому аспекті, тоді як Akita та NgxBaseState мають незначну перевагу, знижуючи затримки при оновленні інтерфейсу.

Наступною важливою метрикою продуктивності веб-застосунків є ефективне використання оперативної пам’яті (M_{usage}). Високе споживання пам’яті може призводити до уповільнення роботи браузера, збільшення часу реакції застосунку та підвищеного навантаження на систему. (див. рис. 5.6)

Аналізуючи отримані результати, можна побачити, що при малій кількості записів (100) усі три стейт-менеджери показали майже однаковий рівень використання пам’яті: NgRx та Akita – по 2,8 МБ, а NgxBaseState – 2,7 МБ. Це свідчить про те, що на малих обсягах даних ефективність управління пам’яттю у

всіх розглянутих рішень є приблизно однаковою. Зі збільшенням обсягу даних до 1000 записів використання пам’яті значно зросло.

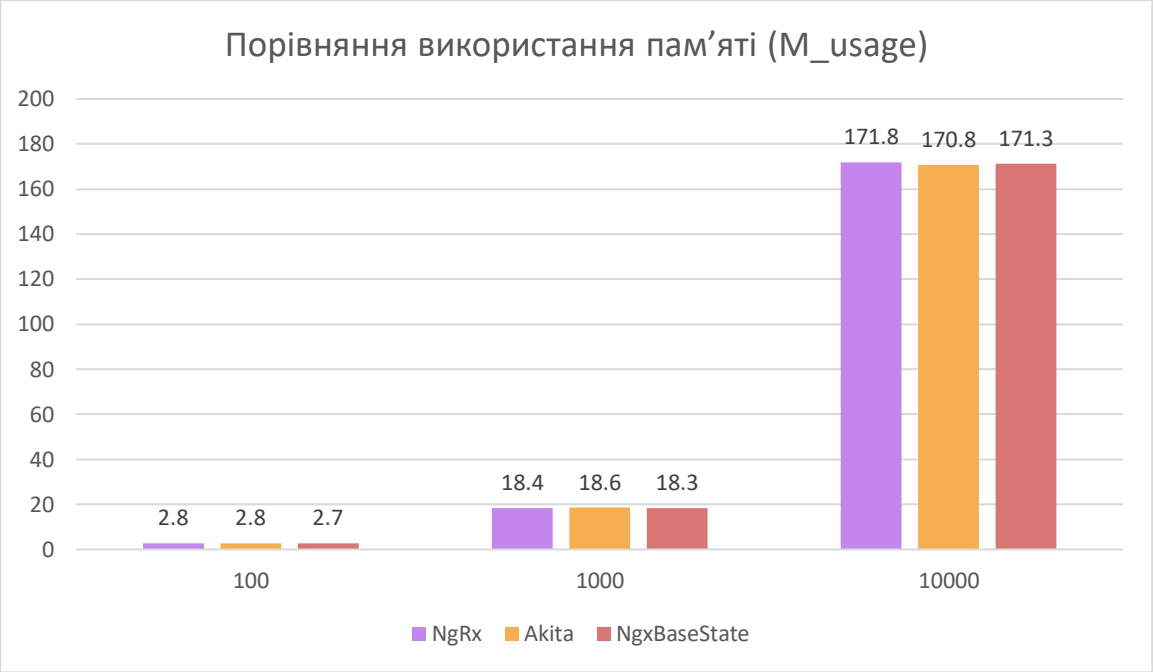

Рисунок 5.6 – Графік порівняння використання пам’яті (рисунок створено самостійно)

Найменше пам’яті використовує NgxBaseState (18,3 МБ), тоді як у NgRx цей показник складає 18,4 МБ, а в Akita – 18,6 МБ. Незначна різниця між значеннями вказує на те, що всі три стейт-менеджери досить добре справляються з управлінням пам’яттю на середніх обсягах даних. Найбільші відмінності проявляються при роботі з великими обсягами інформації (10 000 записів). Тут споживання пам’яті зростає в рази: у NgRx воно досягає 171,8 МБ, у Akita – 170,8 МБ, а у NgxBaseState – 171,3 МБ. Хоча різниця між цими показниками є мінімальною, варто зазначити, що NgRx використовує трохи більше пам’яті, що може мати значення для високонавантажених застосунків.

Загалом, результати свідчать про те, що всі три стейт-менеджери мають подібні показники використання пам’яті, і їх вибір у реальних проєктах має ґрунтуватися не лише на цьому показнику, а й на інших аспектах, таких як зручність реалізації, швидкодія оновлення стану та відповідність архітектурним вимогам застосунку.

Тепер перейдемо до аналізу вихідного коду, який описує реалізацію керування станом у тестових додатках. А саме метрика розміру коду, дана метрика дозволяє оцінити, скільки файлів і рядків коду (LoC) необхідно для повної реалізації стейт-менеджменту в межах двох фіч тестового додатку. (див. рис. 5.7) Аналіз цієї характеристики є важливим, оскільки більший обсяг коду ускладнює підтримку, розширення функціоналу та може впливати на швидкість розробки.

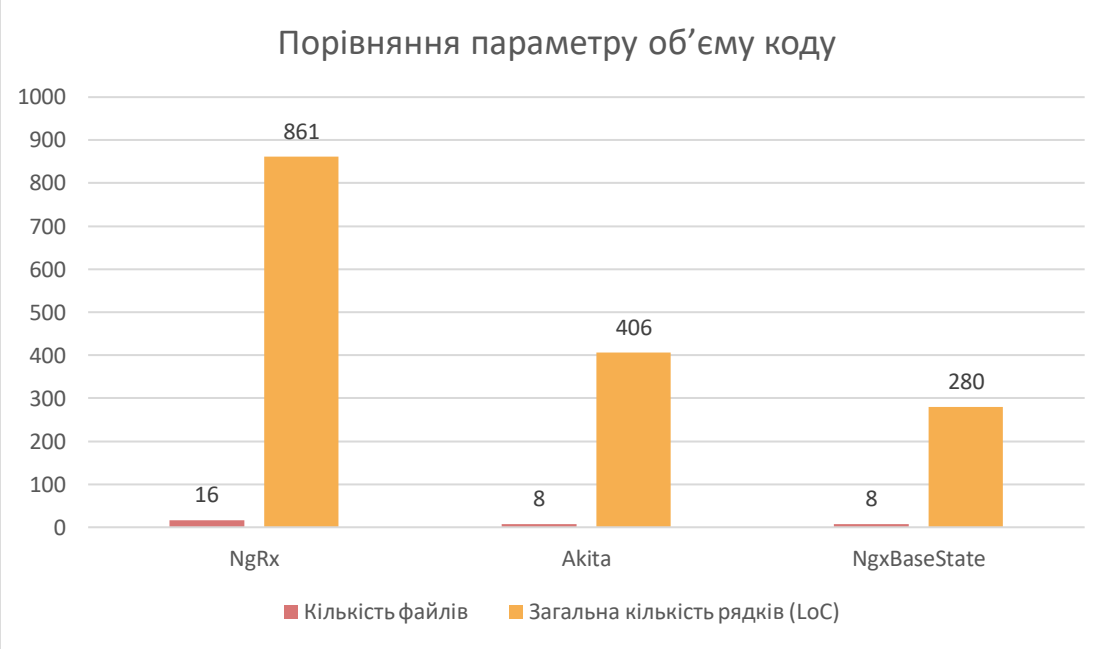

Рисунок 5.7 – Графік порівняння параметру об’єму коду (рисунок створено самостійно)

Аналізуючи отримані дані, можна побачити значні відмінності між трьома стейт-менеджерами. NgRx виявився найбільш об’ємним, потребуючи 16 файлів і загалом 861 рядок коду. Це пояснюється його декларативним підходом, який вимагає створення численних додаткових структур, таких як редюсери, ефекти, дії та селектори. Akita є менш громіздким у реалізації, використовуючи лише 8 файлів і 406 рядків коду. Завдяки своїй більш лаконічній API та меншій потребі у шаблонному коді, Akita дозволяє зменшити обсяг вихідного коду, зберігаючи при цьому зручність у роботі зі станом. Найменше коду потребує NgxBaseState – усього 8 файлів і 280 рядків. Така ефективність пояснюється його спрощеним

підходом до управління станом, мінімальною кількістю шаблонного коду, що зменшує кількість необхідних файлів і рядків коду без втрати гнучкості.

Загалом, отримані результати показують, що NgRx є найбільш складним у реалізації через велику кількість коду, тоді як Akita та особливо NgxBaseState дозволяють досягти аналогічного результату з меншими витратами коду. Це може відігравати важливу роль при виборі стейт-менеджера залежно від потреб проєкту.

Далі перейдемо до метрики “Параметри розміру вихідного коду” для стейт-менеджерів. Дана метрика дозволяє оцінити, скільки місця займає початковий код до збірки, його розмір після оптимізації та який відсоток він становить від загального розміру бандлу. Аналіз цієї характеристики є також важливим, оскільки більший розмір коду може негативно впливати на швидкість завантаження додатку та його продуктивність у браузері. (див. рис. 5.8)

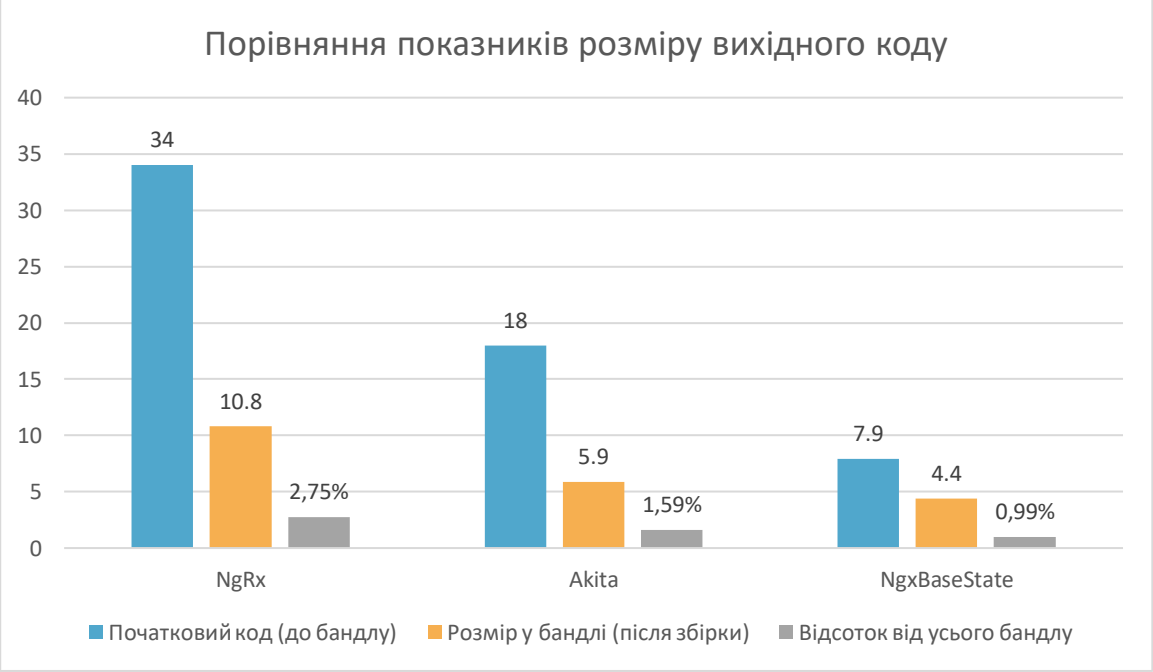

Рисунок 5.8 – Графік порівняння показників розміру вихідного коду керування станом (рисунок створено самостійно)

Аналізуючи отримані результати, можна помітити, що NgRx має найбільший обсяг початкового коду – 34 КБ, який після збірки оптимізується до 10.8 КБ, що становить 2.75% від загального розміру бандлу. Такий великий

початковий розмір пояснюється складною архітектурою NgRx, що передбачає використання множинних файлів для редюсерів, дій та ефектів.

Akita, у свою чергу, потребує значно менше коду – 18 КБ, а після збірки його розмір зменшується до 5.9 КБ, що становить 1.59% від усього бандлу. Менший розмір можна пояснити більш лаконічним API та відсутністю потреби в створенні додаткових файлів для управління станом, як це відбувається в NgRx.

Найменший розмір має NgxBaseState – лише 7.9 КБ у початковому вигляді, який після збірки стискається до 4.4 КБ, що становить всього 0.99% від загального бандлу. Це свідчить про ефективність підходу цього стейт-менеджера, з точки зору розміру, який мінімізує кількість шаблонного коду та простішою архітектурою без втрати функціональності.

Загалом, результати показують, що NgRx, хоча й є потужним рішенням, має найбільший вплив на розмір бандлу, що може негативно позначатися на продуктивності веб-додатку. Akita пропонує більш збалансоване рішення, зменшуючи розмір коду, зберігаючи при цьому достатню гнучкість. NgxBaseState має менший розмір за рахунок мінімальної кількості шаблонного коду, що робить його ефективним рішенням з точки зору оптимізації розміру вихідного коду.

Далі перейдемо до метрики “Відсоток дублювання шаблонного коду” для стейт-менеджерів. Дана метрика дозволяє оцінити, наскільки багато коду повторюється в рамках реалізації керування станом. Високий рівень дублювання шаблонного коду може ускладнювати підтримку та розвиток додатка, а також збільшувати загальну складність коду. Аналіз цієї характеристики є важливим, оскільки зменшення дублювання сприяє покращенню читабельності, підтримуваності та масштабованості проєкту. (див. рис. 5.9)

Аналізуючи отримані результати, можна побачити, що найбільший відсоток дублювання шаблонного коду спостерігається у NgRx (59.8%), що свідчить про значну повторюваність коду при реалізації керування станом. Це пояснюється тим, що NgRx вимагає багато шаблонного коду для оголошення екшенів, редюсерів та ефектів, що призводить до збільшення обсягу дублікатів у кодовій базі. Akita демонструє кращий показник – 39.4%, що означає зменшення кількості

повторюваних конструкцій у коді. Це пов’язано з більш зручним API, яке дозволяє збалансувати шаблонний код і зробити його менш громіздким у порівнянні з NgRx.

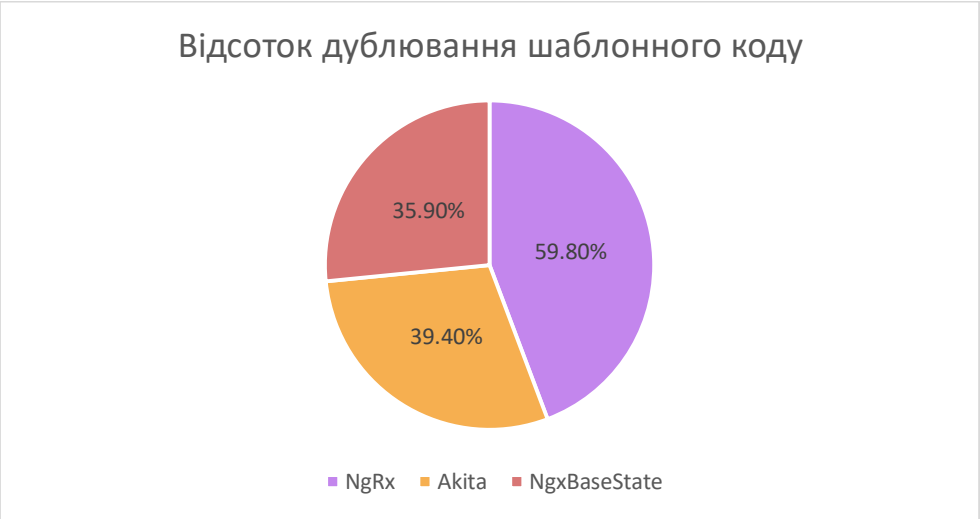

Рисунок 5.9 – Графік порівняння показників дублювання шаблонного коду (рисунок створено самостійно)

Найменший рівень дублювання шаблонного коду демонструє NgxBaseState (35.9%), що вказує на його перевагу з точки зору мінімізації повторюваних конструкцій. Завдяки цьому розробники можуть швидше реалізовувати бізнес-логіку без надмірного коду. Проте, хоча спрощена архітектура NgxBaseState сприяє зменшенню шаблонного коду, у складних проєктах вона може викликати труднощі з масштабованістю та проектуванням структури застосунку, що потребує додаткового аналізу перед вибором цього підходу.

5.3 Висновки та рекомендації

Після проведення серії експериментів та аналізу отриманих результатів вдалося оцінити продуктивність та ефективність використання трьох state-менеджерів для Angular: NgRx, Akita та ngx-base-state. Дослідження включало вимірювання ключових показників продуктивності, аналіз кодової бази, визначення рівня шаблонного коду, а також оцінку загальної зручності та придатності кожного підходу для різних типів проєктів.

Якщо говорити про продуктивність, то Ngx-base-state забезпечує найшвидше оновлення стану, що робить його ефективним для додатків із частими змінами даних, тоді як NgRx та Akita показали трохи нижчу швидкодію через складніші механізми обробки змін. Усі три state-менеджери демонструють схожі результати за затримкою рендерингу, але ngx-base-state має перевагу у випадках роботи з великими обсягами даних завдяки простішому підходу до оновлення стану. Використання пам’яті між підходами різниться незначно, хоча Ngx-base-state виявився трохи ефективнішим при високих навантаженнях, тоді як NgRx може споживати більше ресурсів через складні механізми керування станом.

Щодо обсягу вихідного коду, NgRx має найбільший розмір через необхідність створення значної кількості додаткових файлів і шаблонного коду. Akita забезпечує баланс між структурованістю та простотою, займаючи проміжне місце за складністю та обсягом коду. Ngx-base-state має найменший розмір вихідного коду, що спрощує його підтримку та інтеграцію. Дублювання шаблонного коду є найбільшим у NgRx через складну структуру та необхідність створювати додаткові елементи, тоді як Akita зменшує цей обсяг завдяки простішому підходу. Ngx-base-state має мінімальний рівень дублювання, що позитивно впливає на швидкість розробки та підтримку коду.

На основі отриманих результатів можна сформулювати такі загальні рекомендації щодо вибору state-менеджера залежно від типу проєкту.

NgRx демонструє високу ефективність у проектах, де критично важлива централізована архітектура та контроль за змінами. Завдяки строго визначеній структурі, заснованій на патерні Redux, NgRx забезпечує передбачуваність стану, можливість відслідковувати історію змін та контролювати бізнес-логіку навіть у великих і складних додатках. Це особливо актуально для корпоративних систем або проектів з великою кількістю взаємодій між компонентами, де важлива централізація даних і суворе розмежування відповідальності. Однак через значний обсяг шаблонного коду та складність налаштування, NgRx може стати надмірним для невеликих проектів або додатків, де основним пріоритетом є швидкість розробки та гнучкість архітектурних рішень.

Akita виявляється хорошим компромісом між продуктивністю та простотою використання. Його структура дозволяє ефективно управляти станом при меншому обсязі шаблонного коду, зберігаючи при цьому певну централізованість і структурованість даних. У експериментах Akita показала достатню гнучкість, забезпечуючи швидке оновлення стану без надмірного дублювання коду. Це робить його оптимальним вибором для середніх проектів, де важливо підтримувати баланс між ефективністю та простотою розробки, а також для додатків із динамічною взаємодією компонентів, де потрібно швидко впроваджувати зміни без суттєвого збільшення складності системи.

Ngx-base-state відзначається високою продуктивністю завдяки мінімалістичному підходу до управління станом. Він забезпечує найшвидше оновлення даних завдяки оптимізованій обробці змін, що є критичним для SPA-додатків і систем із частими оновленнями інтерфейсу. Крім того, мінімальний обсяг вихідного коду та низький рівень дублювання шаблонного коду значно спрощують інтеграцію та подальшу підтримку проекту. Цей state-менеджер є ідеальним вибором для невеликих проектів, де важливо досягти максимальної швидкодії з мінімальними витратами на розробку, а архітектурна складність має бути зведена до мінімуму.

Підсумовуючи, вибір state-менеджера повинен базуватися на конкретних потребах проєкту:

- для масштабних додатків із складною бізнес-логікою та високими вимогами до централізованого управління станом доцільно використовувати NgRx;
- для проектів середнього рівня складності, де потрібна гнучкість та оптимальний баланс між структурованістю та простотою, більш придатним є Akita;
- для невеликих додатків або систем із високою частотою оновлень, де ключовим фактором є швидкість розробки та мінімізація обсягу коду, оптимальним вибором стане ngx-base-state.

Таблиця 5.7 – Вибір state-менеджерів Angular на основі експериментальних порогових значень

Метрика	Ngx-base-state	Akita	NgRx
Час оновлення стану	< 0.4 мс	0.4 – 0.7 мс	> 0.7 мс
Затримка рендерингу	< 230 мс	230 – 240 мс	> 240 мс
Використання пам’яті	< 170 МБ	170 – 172 МБ	> 172 МБ
Розмір коду (LoC)	< 300	300 – 450	> 800
Розмір у бандлі (КБ)	< 5 КБ	5 – 6 КБ	> 10 КБ
Відсоток дублювання шаблонного коду	< 38%	38 – 45%	> 50%

Більш деталізовані рекомендації на основі сценаріїв та метрик (див. табл. 5.7):

- для високошвидкісних та компактних SPA/реактивних додатків Якщо ваш проєкт потребує максимальної швидкості обробки стану (< 0.4 мс), низької затримки рендерингу (< 230 мс), мінімального використання пам’яті (< 170 МБ), компактного коду (< 300 LoC) та невеликого внеску в бандл (< 5 КБ), найкращим вибором буде ngx-base-state. Це підходить для простих застосунків, MVP, інтерактивних UI-компонентів, систем із частими оновленнями даних або обмеженими ресурсами. Низький рівень дублювання шаблонного коду (< 38%) також забезпечує легшу підтримку й швидке масштабування без зростання складності;
- для середніх проєктів, де важливий баланс між продуктивністю та архітектурною організацією Якщо час оновлення стану в межах 0.4 – 0.7 мс, рендеринг – 230 – 240 мс, а розмір коду становить до 450 рядків, тоді доцільно використовувати Akita. Він забезпечує гнучку архітектуру з помірним розміром коду, помірним навантаженням на пам’ять (до 172 МБ), та кращу підтримку типізації та структури. Akita доцільно використовувати в SaaS-рішеннях, адміністративних панелях, внутрішніх

системах із не надто складною логікою та 3–10 модулями взаємодії. Це забезпечить стабільність при збереженні помірної швидкості розробки та підтримки;

– для великих корпоративних або легасі систем із високими вимогами до централізованого управління Якщо метрики перевищують критичні пороги (> 0.7 мс оновлення, > 240 мс рендеринг, > 172 МБ пам’яті, > 800 LoC), а також важливі підтримка Redux-подібного патерну, історії змін та повний контроль над потоком даних – NgRx є логічним вибором. Це особливо актуально для ERP/CRM-систем, проєктів із десятками взаємозалежних модулів, багаторівневою логікою та командною розробкою. Попри високий рівень дублювання шаблонного коду (> 50%) і збільшене навантаження, NgRx забезпечує найвищу масштабованість і контроль.

Отримані результати цього дослідження дозволяють краще зрозуміти переваги та обмеження кожного зі стейт-менеджерів у контексті реальних потреб проєкту. Завдяки кількісному порівнянню ключових метрик розробники можуть не лише обрати найбільш підходящий інструмент, а й передбачити його поведінку в умовах різного навантаження. Це особливо важливо на етапах планування архітектури, коли кожне рішення впливає на масштабованість і стабільність. Практичне застосування результатів сприятиме скороченню часу на інтеграцію, зменшенню помилок та підвищенню якості коду. Крім того, врахування особливостей кожного інструмента дозволяє краще адаптувати їх до внутрішніх процесів команди розробки. Сформовані рекомендації дозволять розробникам приймати більш обґрунтовані рішення при виборі стейт-менеджеру, що в цілому сприятиме підвищенню ефективності розробки веб-додатків на Angular.

ВИСНОВКИ

У результаті виконання науково-дослідницької практики за темою «Дослідження продуктивності та ефективності state-менеджерів для веб-додатків на Angular» було проведено комплексний аналіз існуючих рішень та практичне дослідження. Основна мета дослідження полягала у визначенні оптимального підходу до управління станом у веб-додатках враховуючи складність проєкту, зокрема при використанні NgRx, Akita та ngx-base-state.

Було здійснено огляд предметної галузі та проаналізовано сучасні тенденції у сфері управління станом у SPA-додатках. Визначено, що ефективний state-менеджмент є критично важливим для забезпечення продуктивності, масштабованості та зручності підтримки сучасних веб-додатків. Дослідження підтвердило, що вибір state-менеджера безпосередньо впливає на швидкодію та стабільність системи.

Також було проведено детальний огляд трьох state-менеджерів. NgRx виявився потужним, проте вимагав значних ресурсів для інтеграції через сувору структуру. Akita забезпечив баланс між простотою та гнучкістю, що зробило його оптимальним вибором для середніх за масштабом проєктів. Ngx-base-state продемонстрував легковаговість та високу продуктивність у невеликих проєктах, мінімізуючи складність реалізації.

На основі попередніх досліджень було підтверджено залежність вибору state-менеджера від складності проєкту. Водночас у попередніх роботах ngx-base-state не розглядався, що зумовило необхідність проведення дослідження.

Для тестування продуктивності state-менеджерів було спроектовано та розроблено тестовий додаток «To-Do List», який був реалізований трьома різними підходами до управління станом. Була визначена трирівнева клієнт-серверна архітектура для додатку, для збереження даних спроєктовано відповідну базу. Також було створено UI/UX-дизайн, що забезпечив уніфікованість інтерфейсу при інтеграції різних state-менеджерів.

Оцінювання ефективності state-менеджерів проводилося за такими ключовими метриками, як час оновлення стану, затримка рендерингу,

використання пам’яті, обсяг коду та рівень дублювання. Для цього було спроєктовано тестові сценарії, що включали роботу з великими обсягами даних і часті зміни стану, а також розроблено математичні моделі для стандартизованого аналізу отриманих результатів.

Результати експериментів показали, що ngx-base-state забезпечує найкращу продуктивність при роботі з невеликими та середніми обсягами даних, значно випереджаючи NgRx за швидкістю оновлення стану. Akita, хоча і демонструє баланс між продуктивністю та функціональністю, має складніший API, що збільшує час інтеграції. NgRx, через громіздку систему дій, редюсерів та ефектів, виявився найменш продуктивним, особливо при обробці великих масивів даних. Щодо використання пам’яті, ngx-base-state також став лідером, особливо у високонавантажених сценаріях. Akita показав стабільні результати, тоді як NgRx споживав більше ресурсів при аналогічних умовах. Це свідчить про доцільність використання ngx-base-state або Akita у проєктах із жорсткими обмеженнями на ресурси. Аналіз складності коду вказав, що ngx-base-state є найбільш простим у використанні, тоді як NgRx потребує значних зусиль для впровадження та підтримки. Його застосування виправдане лише у великих корпоративних системах, де важлива сувора узгодженість даних. Akita займає проміжне положення, пропонуючи структурованість із відносною простотою впровадження.

В результаті роботи, були сформовані рекомендації по вибіру state-менеджера в залежності від складності проєкту. NgRx є найкращим рішенням для великих систем, Akita – для середніх проєктів, де потрібен баланс між продуктивністю та зручністю розробки, а ngx-base-state ідеально підходить для невеликих застосунків та прототипів.

Перспективи подальших досліджень включають розширення експериментальної бази та порівняння цих state-менеджерів із іншими підходами, що дозволить отримати ще точніші висновки про їх ефективність у реальних проєктах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. I. Shubin, Z. Dudar , V. Skovorodnikova, S. Litvin, Current Trends in Communication and Information Technologies /Research of Ways to Increase the Efficiency of Functioning Between Firewalls in the Protection of Information Web-Portals in Telecommunications Networks, Springer Nature Switzerland AG., pp 272-292

2. Single Page Web Applications / Michael S. Mikowski, Josh C. Powell, 2013. 433 с. (дата звернення: 24.12.2024)

3. Angular web framework documentation [Електронний ресурс] URL: https://angular.dev/overview (дата звернення: 24.12.2024).

4. Reactive Extensions Library for JavaScript documentation. [Електронний ресурс] URL: https://rxjs.dev/guide/overview (дата звернення: 24.12.2024)

5. NgRx reactive state manager documentation. [Електронний ресурс] URL: https://ngrx.io/docs (дата звернення: 24.12.2024)

6. Akita reactive state manager documentation. [Електронний ресурс] URL: https://opensource.salesforce.com/akita/ (дата звернення: 24.12.2024)

7. Ngx-base-state reactive state manager documentation. [Електронний ресурс] URL: https://github.com/Nillcon248/ngx-base-state (дата звернення: 24.12.2024)

8. І.В. Кириченко, Інформаційний пошук навчального контенту у веб-просторі, Тези доповідей VII Міжнародно-практичної конференції «Проблеми та перспективи розвитку IT-індустрії», 28-29 квітня 2018 р. – Х. : ХНЕУ ім. Семена Кузнеця, 2018– С. 45.

9. RxJs in Action / Paul P. Daniels Luis Atencio, 2017, 354 с. (дата звернення: 14.02.2024)

10. Pranskunas V. Deep Comparison Of State Management Solutions for Angular [Електронний ресурс] / Vytautas Pranskunas – 2019 – URL: https://medium.com/@vpranskunas/deep-comparison-of-state-management-solutions-in-angular (дата звернення: 24.12.2024)

11. Levanenia A. State Management in Angular: Exploring Strategies for Effective Application Management [Електронний ресурс] / Aliaksandr Levanenia –

2024. – URL: https://medium.com/@schaman762/state-management-in-angular-exploring-strategies-for-effective-application-management (дата звернення: 24.12.2024)

12. TypeScript programming language documentation [Електронний ресурс] URL: https://www.typescriptlang.org/docs/ (дата звернення: 24.12.2024).

13. Jasmine testing JS framework documentation [Електронний ресурс] URL: https://jasmine.github.io/ (дата звернення: 24.12.2024).

14. Karma web testing space documentation [Електронний ресурс] URL: https://www.npmjs.com/package/karma (дата звернення: 24.12.2024).

15. Cypress - Test your modern applications with our open-source app [Електронний ресурс] URL: https://www.cypress.io (дата звернення: 24.12.2024).

16. Miro - Meet the Innovation Workspace, the AI-powered collaboration platform that helps your team build the right thing faster. [Електронний ресурс] URL: https://miro.com (дата звернення: 24.12.2024).

17. Figma helps design and development teams build great products, together [Електронний ресурс] URL: https://www.figma.com/ (дата звернення: 24.12.2024).

18. Козюбера М.В., "МЕТОД ПОБУДОВИ МЕТРИКИ ОЦІНКИ СКЛАДНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.". Інформаційні технології і автоматизація – 2024 / Матеріали XVII міжнародної науково-практичної конференції. Одеса, 31 жовтня - 1 листопада 2024 р. - Одеса, Видавництво ОНТУ, 2024 р. С. 130 - 132"

19. О.С. Ашурова, Дослідження метрик програмного забезпечення у розробці адаптивних навчальних систем, Матеріали 8-ї Міжнародної науково-технічної конференції , 9-14 вересня 2019 р., Коблеве-Харків, Україна

20. EsBuild Bundle Size Analyzer [Електронний ресурс] URL: https://esbuild.github.io/analyze/ (дата звернення: 24.12.2024).

21. Skovorodnikova, V., Kozyriev, A., Pitiukova, M., Mining methods for adaptation metrics in e-learning, CEUR Workshop Proceedings, 2019, 2362

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

1. I. Shubin, Z. Dudar , V. Skovorodnikova, S. Litvin, Current Trends in Communication and Information Technologies /Research of Ways to Increase the Efficiency of Functioning Between Firewalls in the Protection of Information Web-Portals in Telecommunications Networks, Springer Nature Switzerland AG., pp 272-292

8. І.В. Кириченко, Інформаційний пошук навчального контенту у веб-просторі, Тези доповідей VII Міжнародно-практичної конференції «Проблеми та перспективи розвитку ІТ-індустрії», 28-29 квітня 2018 р. – Х. : ХНЕУ ім. Семена Кузнеця, 2018– С. 45.

18. Козюбера М.В., "МЕТОД ПОБУДОВИ МЕТРИКИ ОЦІНКИ СКЛАДНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.". Інформаційні технології і автоматизація – 2024 / Матеріали XVII міжнародної науково-практичної конференції. Одеса, 31 жовтня - 1 листопада 2024 р. - Одеса, Видавництво ОНТУ, 2024 р. С. 130 - 132"

19. О.С. Ашурова, Дослідження метрик програмного забезпечення у розробці адаптивних навчальних систем, Матеріали 8-ї Міжнародної науково-технічної конференції , 9-14 вересня 2019 р., Коблеве-Харків, Україна

21. Skovorodnikova, V., Kozyriev, A., Pitiukova, M., Mining methods for adaptation metrics in e-learning, CEUR Workshop Proceedings, 2019, 2362