Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження впливу методів і засобів контролю глобального стану компонентів на ефективність розробки, виконання та супроводу React додатків
(тема)

Виконав: студент 2 курсу, групи ПЗСм-19-1
спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-професійної програми Програмне забезпечення систем
Пироженко С.С.

Керівник проф. Лєсна Н.С.

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2020 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121-Інженерія програмного забезпечення
(код і повна назва)

Освітньо-професійна програма Програмне забезпечення систем
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«_____» _____ 20 ____ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Пироженку Сергію Станіславовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: Дослідження впливу методів і засобів контролю глобального стану компонентів на ефективність розробки, виконання та супроводу React додатків
затверджена наказом університету від «30» жовтня 2020р № 1490Ст.
2. Термін подання студентом роботи до екзаменаційної комісії «14» 12 2020р.
3. Вихідні дані до роботи: *методи й засоби контролю глобального стану, OS Windows, мова програмування TypeScript, бібліотека React, середовище розробки Visual Studio Code.*
4. Перелік питань, що потрібно опрацювати в роботі: *аналіз предметної області й постановка задачі, дослідження критеріїв ефективності та розробка методики оцінювання ефективності методів та засобів контролю глобального стану додатку, опис програмної системи для дослідження ефективності методів та засобів контролю глобального стану, дослідження ефективності методів та засобів контролю глобального стану React додатків.*
5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій: *предметна область, існуючі методи та засоби, критерії ефективності розробки програмного забезпечення, критерії ефективності роботи програмного забезпечення, методика оцінки ефективності, розроблена програмна система, результати моделювання, рекомендації по використанню, висновки.*

6 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	проф. Лєсна Н.С.		14.12.2020р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	Аналіз предметної області та постановка задачі	06.11.2020р.	
2.	Дослідження критеріїв ефективності та розробка методики оцінювання ефективності методів та засобів контролю глобального стану додатку	11.11.2020р.	
3.	Розробка програмної системи для дослідження ефективності методів та засобів контролю глобального стану	15.11.2020р.	
4.	Дослідження ефективності методів та засобів контролю глобального стану React додатків	23.11.2020р.	
5.	Оформлення результатів дослідження	28.11.2020р.	
6.	Підготовка пояснювальної записки	30.11.2020р.	
7.	Підготовка презентації та доповіді	07.12.2020р.	
8.	Перевірка роботи на антиплагіат	09.12.2020р.	
9.	Нормоконтроль	10.12.2020р.	
10.	Рецензування	10.12.2020р.	
11.	Занесення атестаційної роботи в електронний архів	12.12.2020р.	
12.	Попередній захист	14.12.2020р.	
13.	Допуск до захисту у зав. кафедри	15.12.2020р.	

Дата видачі завдання 02 листопада 2020 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Лєсна Н.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 130 с., 30 рис., 5 табл., 21 джер., 5 додатків.

АРХІТЕКТУРА, ГЛОБАЛЬНИЙ СТАН, ІММУТАБЕЛЬНІСТЬ, ЕФЕКТИВНІСТЬ, КОМПОНЕНТ, РЕДЬЮСЕР, СТАН REACT ДОДАТКУ, CONTEXT API, JAVASCRIPT, MOBX, REACT, REDUX, TYPESCRIPT.

Об’єктом дослідження є процес взаємодії React компонентів та глобальних сховищ.

Метою роботи є дослідження впливу методів та засобів контролю глобального стану компонентів на ефективність розробки, виконання та супроводу React додатків.

У результаті роботи були визначені критерії, створена методика оцінювання ефективності методів та засобів контролю глобального стану React додатків. Розроблена програмна система та проведене дослідження ефективності методів та засобів контролю глобального стану. Розроблені рекомендації щодо використання.

ARCHITECTIRE, COMPONENT, CONTEXT API, EFFECTIVENESS, GLOBAL STATE, IMMUTABILITY, JAVASCRIPT, MOBX, REACT, REACT COMPONENT STATE, REDUCER, REDUX, TYPESCRIPT.

The object of the research is an interaction process between React components and global storages.

The purpose of the work is to identify the impact of different methods and means of global state management on the efficiency of development, execution, and maintenance of React application.

As a result of work criteria were defined and the technique of an estimation of efficiency of methods and means of global state management of React application was created. The software system was developed and the research of efficiency of methods and means of global state management was carried out. Recommendations for use were created.

ЗМІСТ

ВСТУП

За останні роки кількість веб-сайтів, які розроблюються, збільшилась у декілька разів. Зараз кожна компанія, не зважаючи на її розмір, повинна мати свій сайт, не кажучи вже про те, що багато людей хочуть мати свій невеликий веб-сайт у мережі.

На ранніх етапах розвитку веб-технологій переважно створювалися сайти, які мають одну сторінку. Пізніше з'явилася потреба у більш складних додатках, які працюють з даними з серверу, виводять на екран багато елементів з різноманітною анімацією й мають велику кількість сторінок. Разом з цим браузери за останні 5 – 10 років стали набагато потужнішими й мають можливість виконувати JavaScript, що обумовило появу нового методу розробки веб-додатків – single page application метод. Це призвело до швидкого розвитку великої кількості JavaScript бібліотек та фреймворків. Хоча single page application підхід має велику кількість переваг, основною з яких є перехід між сторінками без перезавантажування додатку, даний метод не позбавлений недоліків. Проблемою цього методу є те, що тепер HTML не зберігає у собі усі дані, які потрібні користувачу по поточному URL й перше завантаження сторінки може зайняти деякий час, не кажучи вже про старі або не досить потужні смартфони. Через було створено SSR (server side rendering) підхід, котрий дозволяє візуалізувати усю сторінку з даними на сервері та віддавати її браузеру, на відміну від CSR (client side rendering), який отримує пусту html сторінку від серверу й наповнює її даними після завантаження JavaScript скриптів.

Сьогодні найбільш популярною бібліотекою, яка дозволяє створювати single page application й підтримує SSR є бібліотека React.js. І хоча використання SSR ще не є обов'язковою умовою для розробки гарного веб додатку, React.js набула великої популярності через широкий спектр можливостей і гнучкості при розробці сайтів, а також, що не менш важливо, можливості розробляти мобільні додатки. Перевагою додатків, написаних за допомогою даної бібліотеки, є розподілення додатку на велику кількість компонентів, які можуть використовуватись як у

декількох місцях одного веб-сайту, так й зовсім у різних проектах. Це значно прискорює розробку й покращує архітектуру проекту шляхом наближення розробки до принципу SRP – single responsibility principle, котрий стверджує, що кожний модуль програми повинен відповідати за одну задачу.

Окрім переваг компонентного підходу, який використовує React.js, він все ж таки має й недоліки. Основним недоліком є складність двонаправленої передачі даних між компонентами, так як бібліотека розроблялася лише з підтримкою однонаправленої передачі. Таким чином, передача даних між великою кількістю компонентів різних ієрархій та рівнів стає досить складною задачею на будь-якому етапі роботи. Найкращим рішенням даної проблеми є створення деякого глобального сховища для даних, яке надалі може використовуватися будь-якими компонентами у дереві компонентів. На даний момент є багато засобів, які використовують відокремлене сховище для даних. Є реалізації, які використовують певну архітектуру й мають більш суворі правила роботи з ними, інші ж надають більше гнучкості й залишають проектування архітектури додатку на самому розробнику.

Актуальність теми атестаційної роботи обумовлена наявністю проблеми вибору методів та засобів реалізації глобального сховища React додатків. Такий вибір постає перед розробниками на початку роботи над проектом і може вплинути як на саму розробку, її складність та тривалість цієї розробки, так і на подальшу роботу додатку, а саме: стабільність роботи, продуктивність, масштабування й супровід.

Метою роботи є дослідження впливу методів та засобів контролю глобального стану компонентів на ефективність розробки, виконання та супроводу React додатків.

Об’єктом дослідження є процес взаємодії React компонентів та глобальних сховищ.

Предметом дослідження є методи та засоби контролю глобального стану React додатків.

Задля дослідження ефективності застосовувались теоретичні (моделювання) та емпіричні (експеримент, дослідження, вимірювання) наукові методи.

Наукова новизна. Вперше запропонована методика оцінювання ефективності методів та засобів контролю глобального стану React додатків, розроблені процедури моделювання продуктивності роботи компонентів внутрішнього устрою методів та засобів контролю глобального стану React додатку і продуктивності взаємодії React компонентів з глобальним сховищем.

Практичне значення. Результати оцінювання ефективності та розроблені рекомендації щодо вибору методів та засобів контролю глобального стану React додатків дозволять пришвидшити розробку React додатків, зменшивши її складність та збільшити продуктивність і стабільність роботи додатків, а також зменшити затрати на їх супровід.

Апробація роботи. Результати дослідження, проведеного в атестаційній роботі, були опубліковані в статті «Дослідження продуктивності методів та засобів контролю глобального стану компонентів React додатків» міжнародного наукового електронного журналу «Наука онлайн» та оприлюднені на XXIV міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті» (тези доповіді «Критерії вибору стратегії керування глобальним станом React додатків»).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ Й ПОСТАНОВКА ЗАДАЧІ

1.1 Компонентна архітектура React додатків

При розробці як веб-додатків, так і мобільних додатків на React, розробник поділяє додаток на велику кількість компонентів, які можуть бути будь-чим: починаючи від простої кнопки й закінчуючи цілою сторінкою, яка об’єднує у собі інші компоненти.

Компоненти формують ієрархію у якій на верхньому рівні є компонент додатку, який надалі складається з множини менших компонентів – сторінок, форм, списків, таблиць, тексту та ін.

Кожен компонент містить у собі об’єкт, який відповідає за інформацію, яку компонент відображає кінцевому користувачу й те, як він реагує на дії користувача, тобто його динамічну поведінку. У термінології React даний об’єкт називається «state», далі у роботі надалі буде використовуватися визначення «стан компоненту» або «стан».

При розробці компонентів постійно виникає необхідність передачі стану від одного компоненту до іншого. При передачі вниз по ієрархії не виникає жодних проблем, ця функціональність реалізована ефективно як з точки зору швидкості роботи, так і з сторони написання коду. Прикладом такої передачі даних може бути отримання даних з серверу у компоненті сторінки й передача їх до компоненту, наприклад, таблиці для відображення. Передача ж даних від дочірнього компонента до компонента вищого по ієрархії або передача даних між компонентами, які знаходяться на одному рівні ієрархії є не зовсім тривіальною задачею, котра вирішується за допомогою можливостей JavaScript. Найпростішим прикладом ситуації, у якій необхідно передати дані між двома компонентами одного рівня є додаток, у якому форма для створення задач і список задач розміщені на одній сторінці та мають спільний компонент – саму сторінку. Для реалізації оновлення списку при створення задачі необхідно передати дані спочатку до батьківського компоненту, а потім до компоненту списку. Для

створення такої функціональності, розробнику потрібно передавати вниз по ієрархії функцію, виклик якої дочірнім компонентом буде оновлювати стан батьківського компонента. При цьому, якщо дочірній компонент, який повинен вносити зміни у батьківський компонент, знаходиться глибоко у ієрархії, розробник повинен власноруч передавати цю функцію через кожен проміжний компонент. І хоча ми отримаємо потрібний нам результат, даний підхід не є оптимальним зі сторони зусиль, які потрібні для його реалізації.

Компоненти React можуть бути написані двома способами:

– як функціональні компоненти, де компоненти є простими JavaScript функціями, які можуть повертати код для відображення;

– як класові компоненти, де кожен компонент є класом, який наслідує клас Component. Даний підхід наближує React до стилю програмування, який використовується у об’єктно-орієнтованих мовах програмування.

Починаючи з версії React 16.6 були значно розширені можливості функціональних компонентів [1], що дало поштовх до розвитку нових засобів контролю як глобального, так і локального станів. Також це призвело до того, що деякі вже існуючі бібліотеки почали змінюватися у сторону функціонального підходу програмування й, таким чином, це значно вплинуло як на їх спосіб реалізації, так і на ефективність їхнього використання.

Через вищезазначені зміни є сенс розглядати лише версії бібліотек, які використовуються в нових 16.6+ версіях React додатків.

Основною метою розробників бібліотек по керуванню глобальними даними додатків було знайти можливість зменшити складність передачі даних між компонентами. Таким чином, з’явилась велика кількість засобів, основними ознаками яких були:

– спрощення передачі даних до компонентів, які знаходяться глибоко в ієрархії. Прикладом може бути використання даних користувача, автентифікованого в додатку, у компонентах сторінок для, наприклад, блокування деяких компонентів користувацького інтерфейсу;

– передача даних від дочірнього до батьківського компонента, включаючи передачу даних при глибокій ієрархії;

– спрощення використання спільних даних декількома компонентами та створення єдиної функціональності оновлення цих даних.

1.2 Аналіз існуючих методів і засобів контролю глобального стану

На сьогодні, за даними менеджеру пакетів npm, який використовується для встановлення бібліотек у React додатках, а також пошуку у Google, найпопулярнішими методами та засобами контролю глобального стану є наступні: Flux, Redux, MobX, Unstated, Apollo Link State. Також, слід звернути увагу на Context API + React Hooks, який є вбудованою функціональністю React і, навіть, є основою, що використовується у інших засобах.

Усі вищенаведені методи й засоби спрощують передачу даних між компонентами та іноді надають інші переваги. Хоча кожна з бібліотек має свої переваги, їх використання не позбавлене проблем. Ці проблеми пов’язанні зі способами внутрішньої реалізації зберігання, зміни й передачі даних.

Зауважимо, що перехід від одного засобу до іншого на пізніх стадіях розробки проекту майже неможливий, бо потребує великої кількості часу через необхідність зміни усіх компонентів, які використовують глобальні дані.

Також слід зазначити, що наявна велика кількість бібліотек, які були розроблені не великою командою розробників, а декількома або, навіть, однією людиною для власного використання. Такі бібліотеки не будуть розглядатися у роботі, так як не мають достатнього розповсюдження й, звісно, недостатньо протестовані на проектах.

1.2.1 Архітектура Flux та бібліотека Redux

Бібліотека Redux за даними npm на даний момент є найпопулярнішою бібліотекою для контролю глобального стану React додатків, має майже 5 мільйонів завантажень щотижня й використовується у половині проектів.

Redux був створений на основі архітектури Flux [2], яку запропонував Facebook для роботи з React додатками. Ця архітектура використовує паттерн observer-observable (спостерігач та спостерігаючий) й підтримує передачу даних в одному напрямі.

Flux архітектура базується на чотирьох сегментах, які організовані в однонаправленому потоці: action (дія), dispatcher, store (сховище даних), view (у даному дослідженні - React компонент).

Dispatcher може бути лише один на весь додаток, а кількість дій та сховищ необмежена.

Компоненти можуть викликати дії, котрі мають унікальний тип та можуть передавати дані. Після цього dispatcher повідомляє всім наявним сховищам про дію, виконану компонентом і одне сховище, яке має запрограмовану реакцію на цю дію, оновлюється разом із компонентом, який, у свою чергу, прослуховує дане сховище. Таким чином, ми маємо однонаправлений поток даних, реалізація якого дозволяє оновити компоненти, які використовують глобальні дані з будь-якого іншого компонента чи функції.

Сховища у даній архітектурі відповідають не лише за зберігання даних, а також за їх оновлення.

Слід зазначити, що архітектура Flux є доволі популярною, що привело до того, що вона у тому чи іншому вигляді використовується не лише у React, а й у більшості сучасних бібліотек та фремворків для фронтенд розробки, таких як: Vue, Angular, Riot.js.

Так NGRX, який використовується у Angular фреймворку був натхнений Redux та архітектурою Flux, а Vue має бібліотеку VueX, за основу якої взяті ідеї Flux.

На рисунку 1 зображено схему роботи Flux.

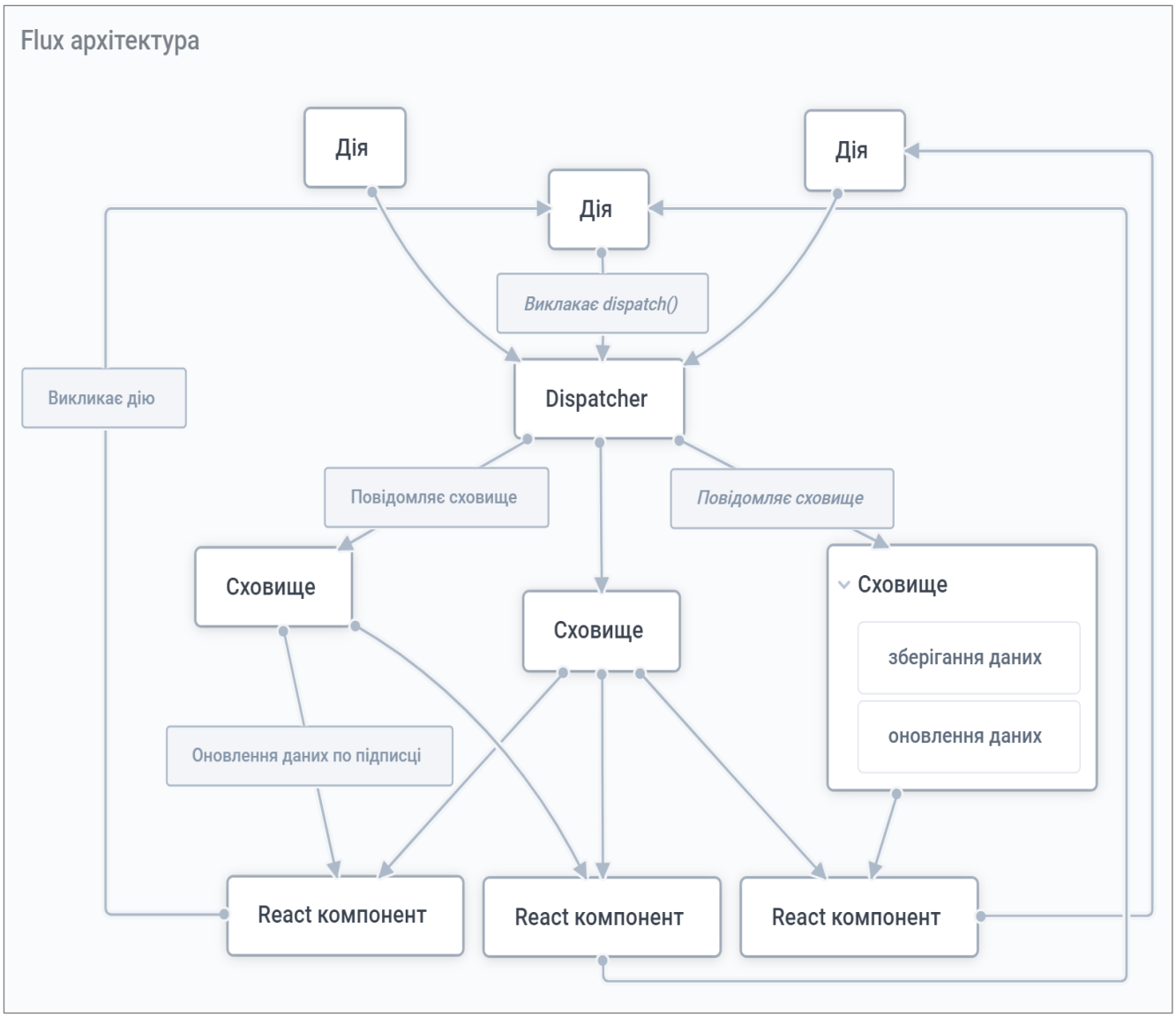

Рисунок 1 – Схема роботи Flux

Так як Flux є лише архітектурою, а не конкретною бібліотекою, на даний момент у React використовується саме Redux.

Основною відмінністю в Redux є перехід від множини сховищ до одного єдиного. Такий крок надав можливість позбавитись від явного використання

dispatcher через відсутність необхідності розподіляти actions між сховищами, хоча сам dispatcher все ще є інструментом для виклику необхідних дій.

Узагальнена схема роботи Redux зображена на рисунку 2.

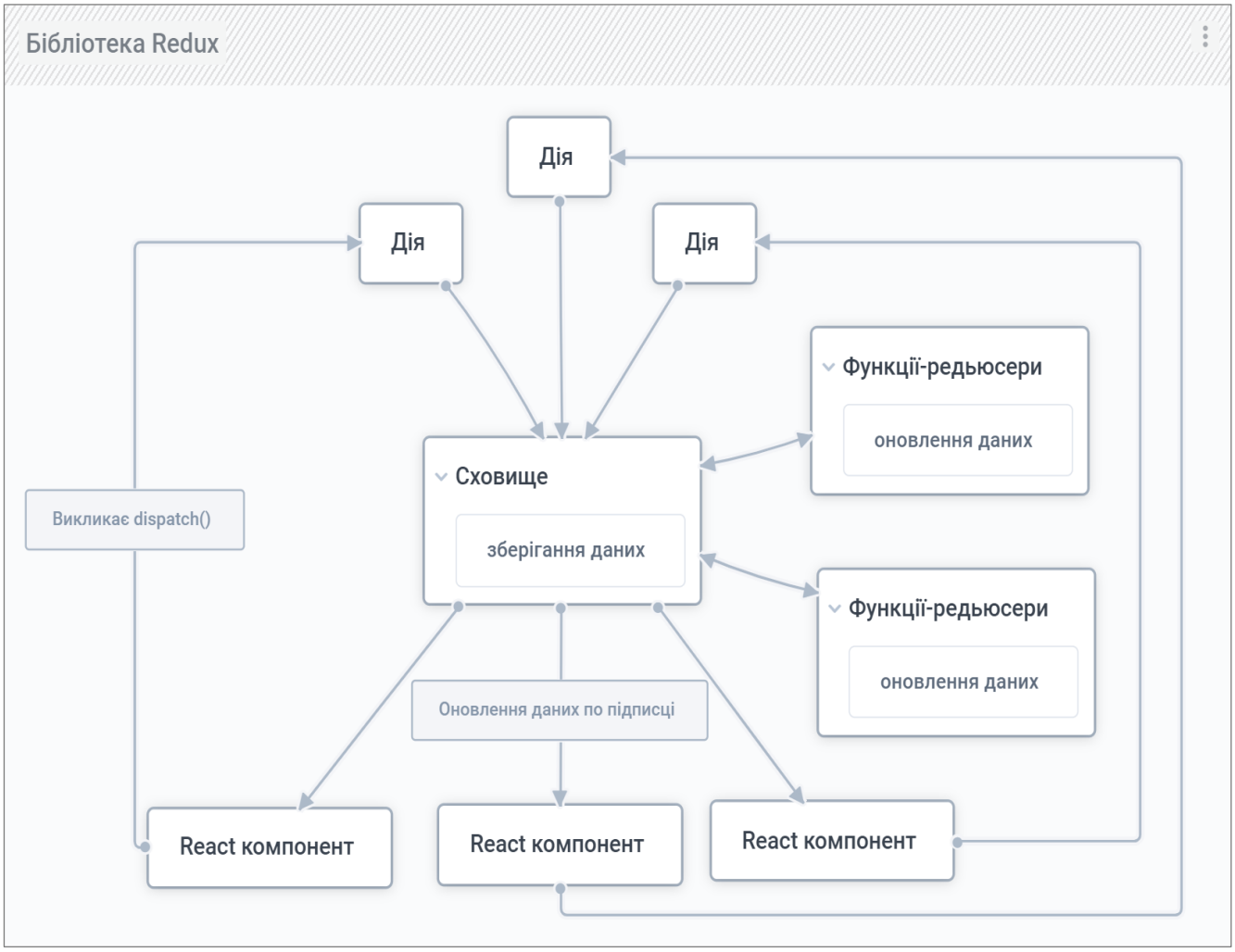

Рисунок 2 – Схема роботи бібліотеки Redux

Як видно з представленої вище схеми (див. рис. 2), другою важливою відмінністю в порівнянні з архітектурою Flux є зменшення відповідальності сховища – тепер воно лише зберігає дані. Уся логіка зміни даних була передана зі сховища до так званих функцій-редьюсерів (reducers). Це «чисті» функції, які, базуючись на типі й даних дій, повертають або незмінений стан, або новий стан, який повністю заміщує попередній стан необхідної частини сховища.

Узагалі поняття «чистих» функцій використовується не лише у React, а є більш широким поняттям, яке описує функції, котрі задовольняють наступним

вимогам: по-перше, функція повертає однаковий результат при однакових вхідних даних, по-друге, функція не має побічних ефектів, наприклад, не робить запитів до серверу та ніяк не впливає на середу її виконання.

Такі функції полегшують тестування додатку й є однією з причин широкого вибору інструментів для тестування бібліотеки Redux.

Бібліотека заохочує розробників використовувати певні архітектурні рішення, що веде до можливості контролю кожного кроку виконання й дозволяє швидке налагодження коду, навіть у проектах, які розробляють не перший рік. Це було досягнуто шляхом незмінності даних у глобальних сховищах, що означає неможливість змінити будь-яке поле об’єкту за бажанням, а лише замінити весь об’єкт повністю. Втім, через такий підхід, багато розробників скаржаться на потребу написання великої кількості шаблонного коду, а також поступове погіршення продуктивності додатку з його зростанням.

Слід зазначити, що Redux знаходиться на ринку досить великий час і за цей період була створена ціла екосистема [3], що складається з допоміжних бібліотек та інструментів для різних задач, наприклад, контролю комплексних форм, обробки структур даних, збереження даних при перезавантаженні сторінки.

Утім, найважливіше місце серед допоміжних бібліотек Redux посідають бібліотеки для виконання побічних ефектів, таких як запити до бази даних. Необхідність у таких інструментах з’явилася через неможливість виконання асинхронних дій у Redux. Найпопулярнішими бібліотеками, які дозволяють використовувати асинхронні дії, є Redux-Saga та Redux-Thunk.

Вищенаведені бібліотеки хоча й виконують однакові функції, мають цілком різні реалізації. Так Redux-Thunk використовується як частина action і дозволяє повертати функцію замість простого об’єкту. Це, у свою чергу, дає можливість використовувати асинхронні дії у цій функції, після чого повертати потрібний об’єкт-дію, який використовується у функції-редьюсері. Redux-Saga натомість використовується як окремий шар асинхронних дій і може бути аналогом Data Access Layer. Реалізація Redux-Saga є більш трудомісткою та складною, бо використовує нововведені у JavaScript функції-генератори для виконання

асинхронного коду. Втім складність використання цілком виправдовується додатковими можливостями по супроводу, масштабованості, а також розширеним функціоналом і кращою архітектурою додатку з можливістю відокремлення логіки у додаткового шар.

Схема роботи з Redux-Saga зображена на рисунку 3.

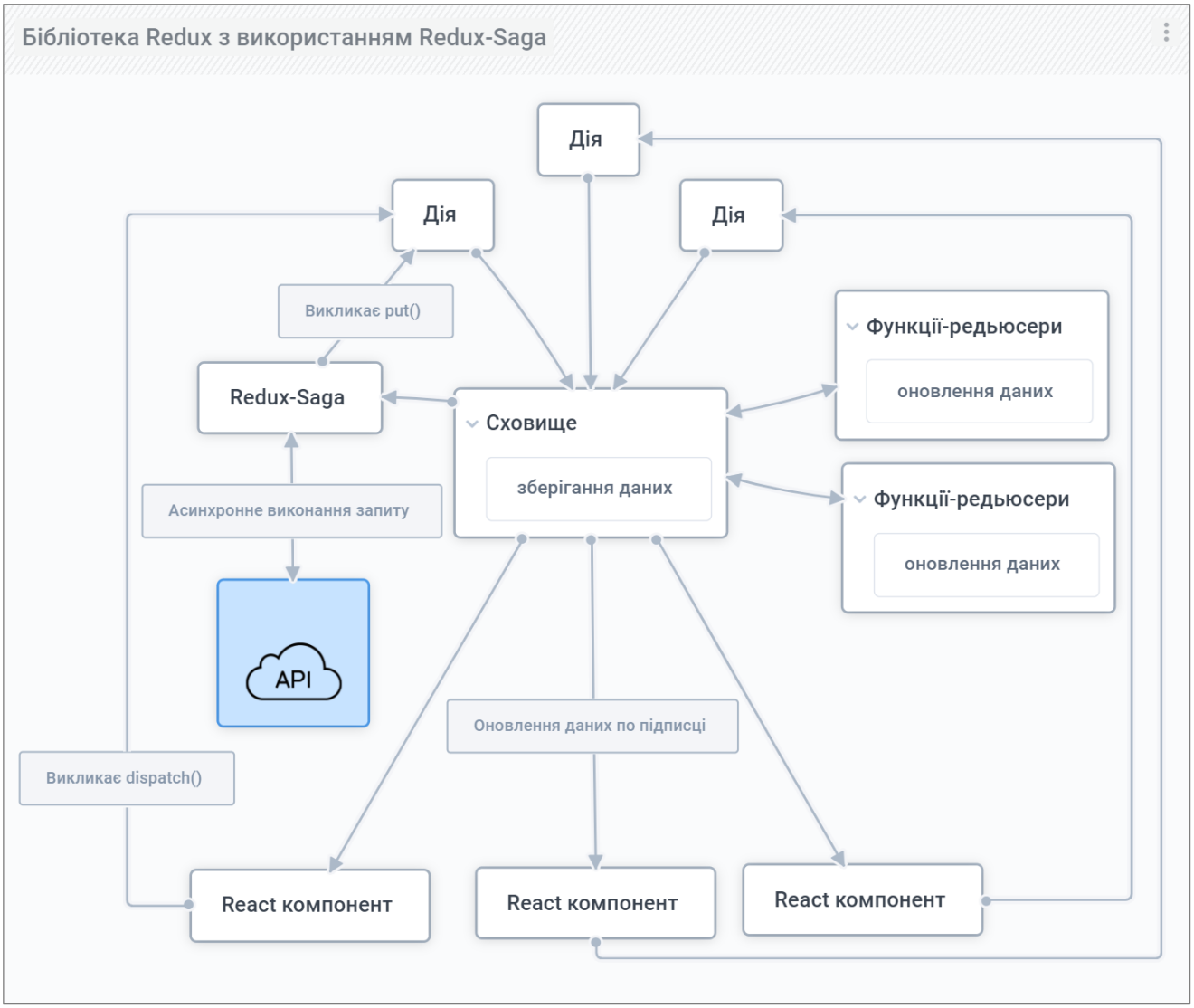

Рисунок 3 – Схема взаємодії Redux та Redux-Saga

Виклик «put()» у даному разі є альтернативою «dispatch()» для виконання дій.

У подальших дослідженнях, для виконання асинхронних запитів буде використовуватися саме Redux-Saga.

1.2.2 MobX

Бібліотека MobX використовує у багатьох речах протилежний до Redux підхід. По-перше, усі дані, які зберігаються у одному чи декількох сховищах, на відміну від Redux, можуть бути змінені, а не тільки замінені. Більш за те, ці зміни можуть бути виконані автоматично.

Основною причиною, через яку розробники обирають MobX, а не Redux, є спрощення багатьох аспектів у порівнянні зі строгою архітектурою Redux та Flux. Наприклад, окрім прямого оновлення даних, ми можемо навіть автоматично обчислювати одні дані на основі інших. Усе це дозволяє значно скоротити час на розробку через написання меншої кількості коду, але, звичайно, має свої недоліки.

Слід також зазначити, що MobX, навідміну від Redux, який пішов у сторону функціонального програмування, використовує ООП підхід, що передбачає створення сховищ за допомогою класів.

Увесь MobX побудовано на так званих декораторах (decorators), які по своїй суті є функціями-обгортками, котрі надають додатковий функціонал іншим функціям й, відповідно, JavaScript класам. До версії React 16.6, використання MobX супроводжувалося використанням класових компонентів і, зазвичай, мови TypeScript, яка є надбудовою звичайного JavaScript. Використання MobX разом з функціональними компонентами було можливе, але потребувало використання додаткових бібліотек або самостійно написаних компонентів-обгорток. При цьому декоратори, які використовуються в MobX, не є частиною ядра JavaScript і тому не доступні для використання у браузері за замовчуванням. Для їх використання без TypeScript необхідна конфігурація IDE, а також використання транспайлерів, котрі перетворюють новий функціонал JavaScript у код, який може бути інтерпретований браузером. Прикладом такого транспайлера може бути Babel. Однак після введення React 16.6 і суттєвого розширення можливостей функціональних компонентів, використання MobX стало значно простіше.

У останніх версіях React для використання сховища у функціональному компоненті React достатньо створити «контекст» MobX класу за допомогою вбудованої у React функції «CreateContext» і використовувати цей контекст у функціональному компоненті за допомогою «useContext».

CreateContext є частиною досліджуємого Context API та дозволяє надавати дані до множини компонентів без необхідності явної передачі.

UseContext є одним з так званих React Hooks (далі – «хук»), котрі є функціями, що дозволяють використовувати стан та функції класових компонентів (у тому числі функції, прив’язані до етапів життєвого циклу компонента) у функціональних компонентах. У цьому випадку «useContext» дозволяє використовувати контекст і викликає оновлення компоненту кожен раз (за відсутності інших налаштувань), коли дані були змінені.

У більшості випадків, при розробці за допомогою MobX, у додатку наявні декілька сховищ, які розділені за сутностями, сторінками або будь-яким іншим способом.

У основу MobX сховища покладені чотири елементи:

– директива @observable – стан, який може бути зміненим у будь-який час та служити основою для @computed значень;

– @computed – значення, які підраховуються за допомогою функцій та базуються на @observable й самі, по своїй суті, є @observable, що означає можливість підписки React компонентів на їх оновлення;

– функції-реакції, які схожі на @computed, але натомість повертають не @observable значення, а виконують деякі сторонні ефекти, наприклад, функції з запитами до серверу, логування даних або навіть виклики @action;

– директива @action, що може бути додана до будь-якої функції та вводить функціонал транзакцій. Це може бути використано при необхідності оновлення багатьох @observable зміних, коли необхідно оновити їх як одне ціле та викликати одне оновлення. Якщо не використовувати директиву @action, то кожне оновлення @observable змінної спричинить оновлення компоненту, який відстежує цю змінну.

Для реагування компонентів на @observable дані, React використовує допоміжну бібліотеку «mobx-react», основна мета якої – додавання функції обгортки до компонентів, яка дозволяє відстежувати зміни у контексті. Дані функції-обгортки реалізують логіку підписок, які надають можливість компонентам відстежувати @observable дані, які використовуються під час відмальовування користувацького інтерфейсу.

На рисунку 4 зображена загальна схема роботи бібліотеки MobX з компонентом.

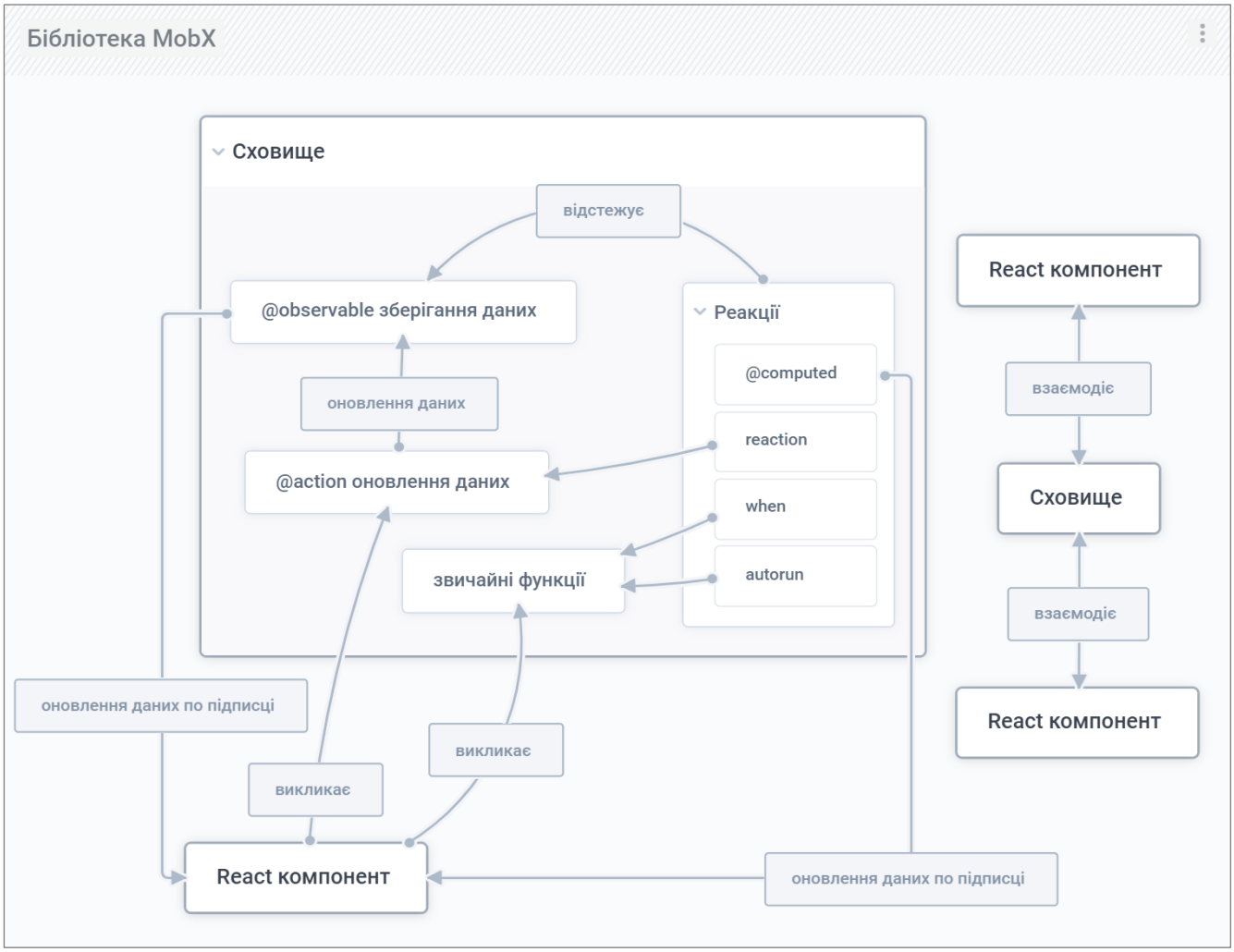

Рисунок 4 – Схема роботи бібліотеки MobX

Як видно зі схеми (див. рис. 4), у разі використання MobX, додаток може мати безліч сховищ, кожне з яких може взаємодіяти з будь-якою кількістю React компонентів.

Уся логіка зберігання й оновлення даних, а також звичайні функції для, наприклад, запитів та логування знаходиться у самому сховищі.

Підписки по своїй суті дозволяють об’єкту, який спостерігається, надсилати усім спостерігачам (компонентам) повідомлення про те, що дані його дані застарілі, після чого вони будуть заново підраховані й змінені усередині компоненту. MobX самостійно контролює як підписку, так і її скасування та відновлення.

Слід зазначити, що реалізація MobX є більш абстрактною за Redux, що призводить до того, що розробник не може контролювати кожен крок оновлення даних у додатку. Це може спричинити помилки, які дуже важко відстежити при недостатньому розумінні роботи бібліотеки, а також за відсутністю детального інструменту для відладки, який наявний у Redux.

1.2.3 Context API та React Hooks

Даний метод створення глобальних сховищ суттєво відрізняється від попередніх, бо є не бібліотекою або архітектурою, а вбудованим функціоналом React, який почав використатися з версії React 16.6.

Context API дозволяє розробникам створювати будь-яку кількість сховищ-контекстів, які надалі можна передавати будь-якому компоненту, обгорнувши його в провайдер з потрібним контекстом. Використання Context API було можливо й до React версії 16.6, але за відсутності такого функціоналу як React Hooks, використання контексту було досить незручним.

Початкові дані для контексту можуть бути додані або у самому контексті, або у компоненті, який є провайдером цього контексту.

Використання Context API дозволяє уникнути такого поняття у React, як «props drilling» - передача стану від батьківського компонента до дочірнього через велику кількість проміжних компонентів.

Взагалі, проблему «prop drilling» вирішує не тільки Context API, а будь-який з вищезазначених методів та засобів. Різниця у передачі даних між «prop drilling» та використанням Context API показана на рисунку 5.

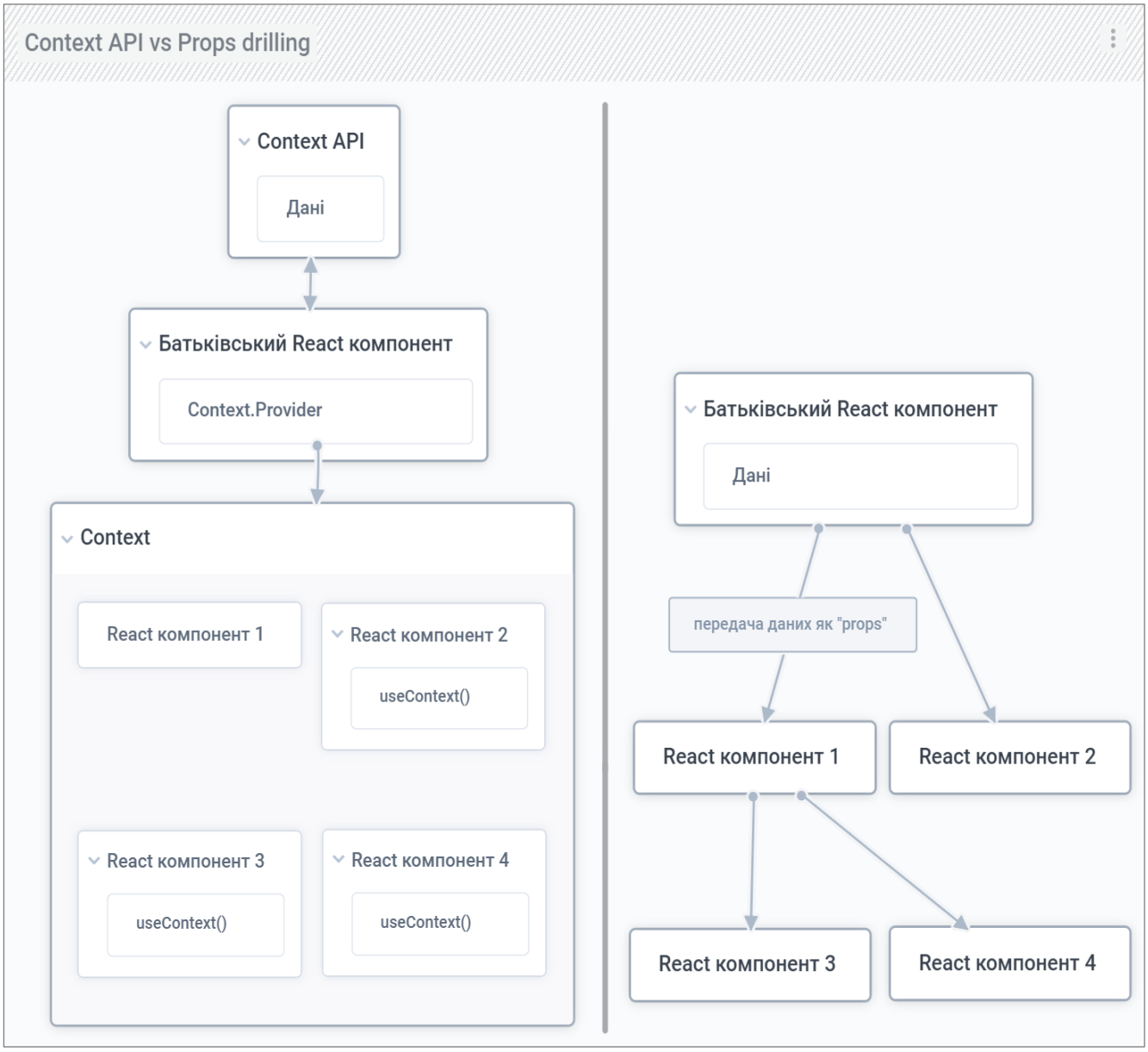

Рисунок 5 – Схема передачі даних без Context API та з його використанням

Реалізація глобального сховища за допомогою Context API та React Hooks у більшості випадків схожа комбінацію на реалізацій Redux та Flux. Це спричинено тим, що архітектурні підходи Redux та Flux є заслужено популярними рішеннями, які надає масштабування й супровід.

Схема роботи одного сховища (Context API дозволяє реалізацію безлічі сховищ) зображена на рисунку 6.

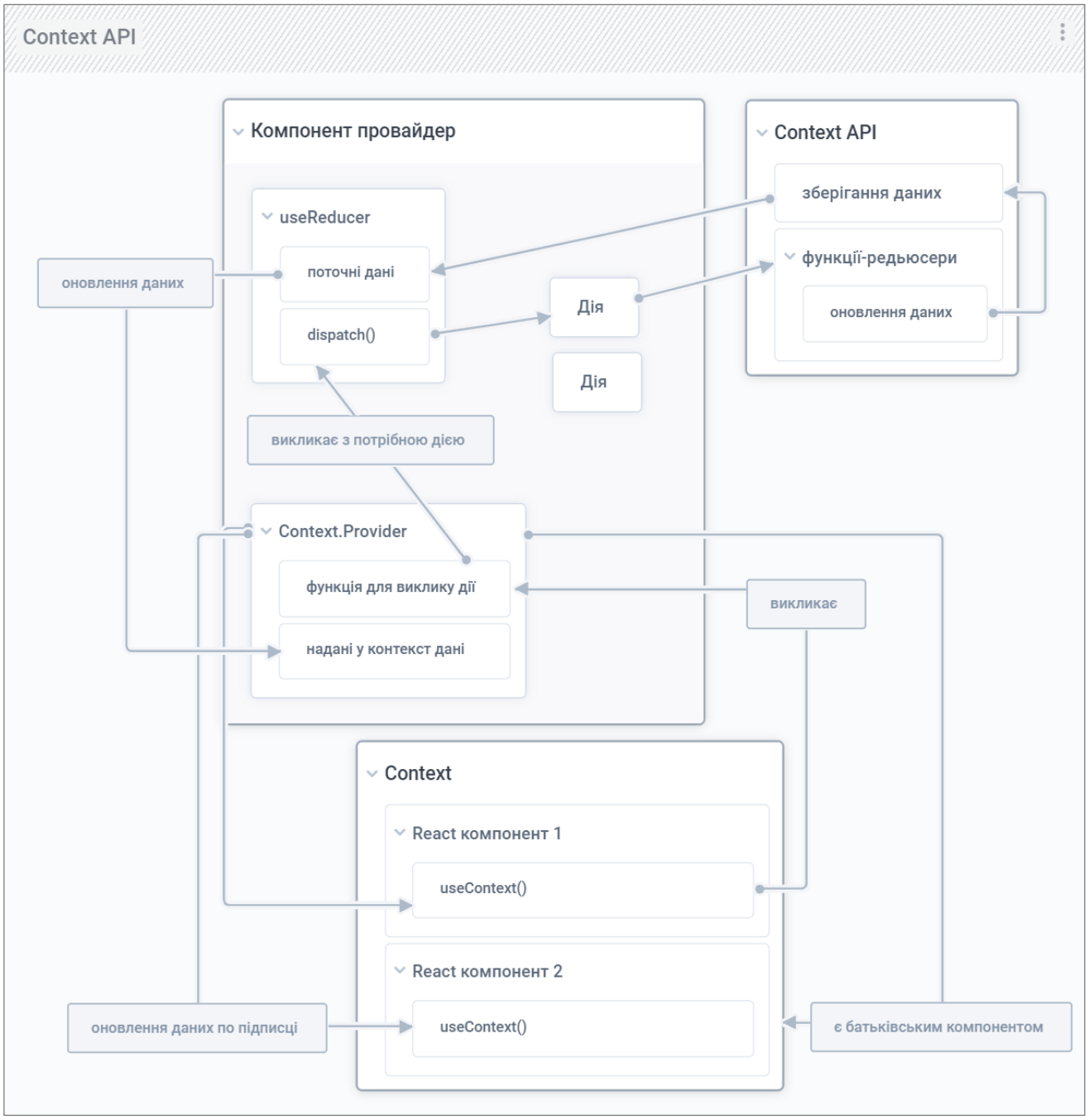

Рисунок 6 – Схема роботи Context API

При вищезазначеній реалізації (див. рис. 6), найбільш важливою частиною є «useReducer» хук, який зв’язує провайдер контексту, який використовується у компонентах та саме сховище (контекст) . Дані з контексту передаються до «useReducer» кожен раз, коли вони оновлюються у сховищі. «useReducer» у свою

чергу передає їх до провайдеру, котрий далі за допомогою «useContext» хуку може бути використаним будь-яким компонентом, який знаходиться нижче по ієрархії ніж компонент провайдер. «useReducer» хук також надає «dispatch()» функцію, яка схожа на аналогічну у бібліотеці Redux та може оновлювати дані у контексті шляхом виклику потрібної дії та її подальшої обробки у функції-редьюсері.

Компонент провайдер надає компонентам функції для виклику «dispatch()», які можуть бути отримані ними, так само як і дані, за допомогою «useContext». Слід зазначити, що навідміну від Redux, кожне сховище може мати лише один набір функцій-редьюсерів. Таким чином, Context API дозволяє створювати безліч сховищ, кожен з яких має свої об’єкти-дії та функції-редьюсери.

1.2.4 Unstated

Бібліотека Unstated надає доступ до даних за допомогою створення контейнеру, який є схожим на React компонент за тим виключенням, що має в собі лише логіку та дані й не містить ніякого відмальовування інтерфейсу. Надалі даний контейнер може бути використаний будь-яким компонентом. Також компоненти, які будуть використовувати контейнери, повинні бути заздалегідь обгорнуті компонентом-провайдером, який входить до складу бібліотеки й надає необхідні функції для роботи.

Головною відмінністю Unstated від інших бібліотек є те, що вона змінює підхід, який використовується для компонентів. Дана бібліотека дозволяє розділити логіку й відображення шляхом розміщення усієї логіки та даних у контейнер, а відображення у окремий компонент, який відповідає за вивід інформації користувачу. Компонент відображення підписується на однин чи декілька контейнерів і може використовувати дані та методи для зміни цих даних.

1.2.5 Apollo Link State

Дана бібліотека відрізняється від попередньо розглянутих засобів у першу чергу тим, що вона має більш специфічну область використання.

Apollo – це платформа, яка є реалізацією GraphQL та використовується для передачі даних між сервером та інтерфейсом користувача, реалізованого у даному випадку за допомогою React. Тому, якщо в роботі використовується GraphQL та Apollo, деякі розробники обирають саме Apollo Link State для менеджменту даних React додатків.

Apollo Link State використовується для централізації локальних даних додатків React. При цьому використання Apollo Link State дозволяє простіше працювати з Apollo Client (проміжна ланка між сервером та клієнтським додатком). Apollo Link State можна описати як єдине сховище, яке зберігає кешовані дані, отримані від Apollo Client.

1.3 Переваги і недоліки найбільш перспективних методів та засобів

Хоча будь-який метод контролю глобальних даних, розглянутий у попередньому підрозділі, може виконувати свою основну функцію – спрощувати передачу даних між компонентами, його використання не завжди виправдане у проекті. У першу чергу слід звернути увагу на спектр використання, розповсюдженість, а також на слабкі й сильні сторони кожного з цих методів та засобів.

Бібліотека Apollo Link State, хоча й має досить гарну реалізацію й функціонал, є вузькоспеціалізованою, бо може застосовуватися лише при використанні у проекті GraphQL замість REST API. І хоча такий підхід до передачі

даних набирає популярності, у порівнянні з REST API його розповсюдження досить низьке.

Бібліотека Unstated, у свою чергу, ще не набула достатньої популярності у великих проектах і тому її використання може привести до неочікуваних результатів при довгостроковому використанні.

Таким чином, будуть розглядатися три засоби: бібліотека Redux, бібліотека MobX, а також комбінація Context API та React Hooks. Ці засоби використовуються доволі часто й кожен з них має свої переваги і недоліки, які наведені у таблиці 1.

Таблиця 1 – Переваги і недоліки обраних засобів

Засіб	Переваги	Недоліки
Redux	– жорстка архітектура, яка спрощує роботу у команді; – verbosity – «багатослівність», яка спрощує тестування, супровід і масштабування додатку; – широкі можливості по розширенню функціоналу за допомогою бібліотек; – відмінний інструментарій для тестування та відладки прямо у браузері; – можливість створення окремого шару для роботи з даними (DAL).	– жорстка архітектура, яка обмежує розробника; – verbosity – «багатослівність», яка збільшує час на розробку й є надмірною для невеликих додатків; – не підтримує асинхронність «з коробки»; – у теорії, має погану продуктивність і потребує додаткового налаштування компонентів для оптимальної роботи з Redux сховищем.
Context API та React Hooks	– не потрібно додаткових бібліотек; – можливість створення багатьох сховищ зі своїми функціями-редьюсерами та діями.	– погана продуктивність у порівняні з іншими засобами; – відсутність можливості розширення функціоналу; – слабкі можливості з відладки та тестування.

Продовження таблиці 1

Засіб	Переваги	Недоліки
MobX	– багато функціоналу працює автоматично; – швидкість роботи «з коробки»; – швидкість реалізації; – мінімальна необхідність написання шаблонного коду; – незначна можливість розширення за допомогою бібліотек.	– багато функціоналу працює автоматично й недоступно для розробника, що може призводити до неочікуваних результатів і неможливості відстеження кожного кроку роботи; – відсутність зручних інструментів для тестування; – потрібні конфігурації IDE, а також TypeScript або Babel для використання декораторів.

Деякі з цих недоліків і переваг (див. табл 1) будуть надалі перевірені під час дослідження та моделювання.

1.4 Постановка задачі

Виходячи з проведеного аналізу предметної області, метою роботи стає визначення впливу різних засобів та методів контролю глобального стану на ефективність роботи React додатків, а саме – його компонентів. Для проведення дослідження повинні бути сформовані критерії ефективності, які можуть бути застосовані з кожним із обраних методів та засобів, і визначена методика оцінювання ефективності.

Поставлена задача може бути розділена на наступні частини:

– аналіз існуючих методів та засобів контролю глобального сховища;

– вибір найбільш перспективних методів та засобів для їх подальшого дослідження;

– визначення поняття ефективності та дослідження критеріїв ефективності;

– визначення метрик для оцінювання критеріїв ефективності;

– розробка методики оцінювання ефективності методів та засобів контролю глобального стану React додатків;

– розробка процедури моделювання продуктивності роботи компонентів внутрішнього устрою методів та засобів контролю глобального стану додатку;

– розробка процедури моделювання продуктивності взаємодії React компонентів з глобальним сховищем;

– розробка програмної системи для дослідження ефективності методів та засобів контролю глобального стану додатку;

– розробка рекомендацій по використанню методів та засобів контролю глобального стану React додатків за результатами оцінювання їх ефективності.

2 ДОСЛІДЖЕННЯ КРИТЕРІЇВ ЕФЕКТИВНОСТІ ТА РОЗРОБКА МЕТОДИКИ ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ТА ЗАСОБІВ КОНТРОЛЮ ГЛОБАЛЬНОГО СТАНУ ДОДАТКУ

2.1 Визначення поняття ефективності та критеріїв її оцінювання

Для вибору критеріїв оцінювання ефективності, перш за все ми маємо чітко сформулювати поняття «ефективність» у сфері даного дослідження.

При розробці програмного забезпечення за допомогою бібліотеки React поняття «ефективність» можна розділити на дві категорії: ефективність розробки програмного забезпечення та ефективність виконання розробленого програмного забезпечення. Використання різних методів і засобів контролю глобального стану React додатків має прямий вплив як на першу, так і на другу категорію.

Визначення критеріїв ефективності є важливим як для розробників програмного забезпечення, так і для замовника, основна мета якого балансування між часом розробки та його якістю.

За основу критеріїв ефективності розробки програмного забезпечення були взяті нефункціональні вимоги до програмного забезпечення запропоновані Карлом Вігерсом [4], які надалі були модифіковані та розширені для цілей досягнення дослідження та розробки програмного забезпечення на React.

2.2 Дослідження критеріїв ефективності розробки програмного забезпечення

До ефективності розробки програмного забезпечення відносяться ті моменти, які напряму впливають на час, складність та якість розробки програмного забезпечення.

У даному дослідженні були обрані наступні критерії цієї категорії:

– масштабованість (scalability);

– можливості налагодження (debugging);

– сумісність (compatibility);

– крива навчання (learning curve);

– ширина інструментарію.

Під масштабованістю розуміють можливість програмного забезпечення стабільно працювати та адаптуватися до стрімких змін деяких показників, розширення, зміни, підвищення вимог. Тобто цією характеристикою визначають можливості адаптації окремих компонентів та системи у цілому під час її тривалої розробки та підтримки. Зазвичай про масштабування системи потрібно замислитися на початку розробки програмного забезпечення, планування використання бібліотек та проектних рішень є однією з частин цього процесу.

Вивченню та розробкою підходів по створенню додатків, які гарно масштабуються замаються багато розробників незалежно від обраної технології. Тому не дивно, що така популярна технологія як React має багато ресурсів, які можуть допомогти розробникам створити архітектуру, яка може гарно масштабуватися.

Серед вивчених ресурсів найбільш видатним є путівник для побудови масштабованих та супроводжуємих React додатків написаний Artemij Fedosejev [5], який покриває теми ефективної розробки компонентів, використання різних бібліотек та архітектур, у тому числі архітектури Flux та бібліотеки Redux. Щодо самої бібліотеки Redux, її архітектуру, а також вбудовування у вже існуючі проекти гарно описані у книжці Marc Garreau [6].

Можливості налагодження додатку є важливим критерієм розробки, бо пошук проблем у коді при розробці програмного забезпечення є задачею, яка трапляється на регулярній основі. Таким чином, при розробці ПЗ розробник повинен розуміти, які можливості та інструментарій має та чи інша бібліотека та технологія. У разі, коли обрана технологія бракує такого інструментарію її використання може призвести до небезпечних наслідків.

Під сумісністю програмного у контексті даного дослідження розуміється можливість того чи іншого методу/засобу безперебійно працювати в умовах різного оточення. У більшості випадків цей критерій залежить від команди підтримки, що супроводжує й оновлює бібліотеку. Так, при недостатніх ресурсах для підтримки бібліотеки, може втратитися стабільна робота у нових версіях. Також слід зазначити, що деякі бібліотеки можуть конфліктувати одна з одною, таким чином ставлячи усю систему під загрозу.

Крива навчання, яка може бути об’єднана з когнітивною складністю коду, на перший погляд, може здатися не досить важливим критерієм, але швидкість, з якою команда або розробник можуть освоїти засіб, його окремі частини та нововведення напряму впливає на якість отримуємого програмного забезпечення та час його розробки. Так при недостатніх знаннях усіх «тонкощів» обраного засобу можна не лише не отримати усіх його переваг, а і зробити критичні помилки при розробці програмного забезпечення, які призведуть до зниження ефективності його роботи.

Ширина інструментарію деякої бібліотеки чи технології (засобу) – це обсяг можливостей, які надаються розробнику. Так при використанні функціоналу лише однієї бібліотеки у декількох місцях, де у протилежному разі могли використовуватися декілька інших, призводить до покращення розуміння коду, його зв’язності та зменшує час на розробку. Чим більший інструментарій у деякої технології, тим вищий показник успіху і менший час на адаптації при появі нових задач. Прикладами такого інструментарію є інструменти для тестування, пошуку помилок, інструменти для роботи з асинхронними запитами та можливості по їх обробці.

Усі вищеозначені критерії є важливими при розробці програмного забезпечення та без істотних змін можуть бути застосовані й до програмного забезпечення, написаного на бібліотеці React з використанням різних методів та засобів контролю глобального стану додатків.

2.3 Дослідження критеріїв ефективності роботи програмного забезпечення

До ефективності виконання програмного забезпечення можна віднести наступні критерії:

– продуктивність – вплив використання засобу на загальну швидкість роботи додатку або на швидкість роботи його окремих частин. У багатьох випадках саме цей критерій стає основним під час вибору методу реалізації глобального сховища, але слід також зазначити, що не для всіх типів додатків висока продуктивність є ключовою вимогою, у такому разі є сенс звернути увагу на інші критерії. У більшості випадків на швидкість роботи бібліотеки впливають такі фактори, як дані, їх тип та структура, яка використовується для зберігання, а також способи реалізації бібліотеки. Так знання бібліотеки та її правильна реалізація можуть компенсувати проблеми у продуктивності;

– вплив обраного методу/засобу на фінальний розмір додатку при його збірці та подальшому розміщення на сервері у виді статичних файлів. Ці статичні файли далі віддаються сервером на запит користувача, який намагається завантажити сторінку веб-сайту у своєму браузері. Розмір цієї сторінки напряму впливає на швидкість даного завантаження й, відповідно, на задоволеність користувача.

Вищенаведені критерії є критичними при розробці сучасного програмного забезпечення, особливо веб-сайтів. Так як у даному дослідженні ми працюємо саме з React бібліотекою, є сенс визначити критерій «продуктивність» у контексті саме цієї бібліотеки. Таким чином, для можливості більш детального аналізу ефективності роботи React у комбінації із засобами глобального контролю стану, критерій продуктивності повинен бути розділений на 2 частини:

– продуктивність роботи внутрішнього устрою елементів глобального сховища. До цього критерію входить вимірювання продуктивності роботи функцій та методів, які використовуються у глобальному сховищі для маніпуляції над даними;

– продуктивність взаємодії глобального сховища з React компонентами, які потребують доступ до даних даного сховища. До цього критерію входить вимірювання продуктивності роботи React компонентів, а також усього додатку узагалі при отриманні оновлених даних від глобального сховища.

2.4 Метрики оцінювання критеріїв ефективності

Визначення метрик для критеріїв оцінювання ефективності розробки програмного забезпечення без використання жодного контексту майже неможливо. Таким чином, було вирішено вимірювати дані критерії при порівнянні засобів контролю глобального стану між собою. Це дозволить виконати порівняння за критеріями навіть без введення чітких метрик. Негативним аспектом такого підходу є те, що його використання можливо лише при розробленому тестовому програмному забезпеченні, котре використовує той чи інший засіб. Тому, для проведення порівняння будуть змодельовані додатки для обраних методів та засобів контролю глобального стану.

Говорячи про критерії оцінювання ефективності роботи програмного забезпечення, швидкість роботи додатку, далі у роботі – продуктивність, є найбільш значущим критерієм, який помітний як користувачу, так і розробнику та може спостерігатися без допомоги додаткових інструментів. Важливість даного критерію зростає з розміром додатку – чим більший додаток, тим з більшою кількістю компонентів і даних він працює, що призводить до більш складних операцій, виконуємих браузером і, відповідно, платформою користувача. Дане питання вивчається протягом багатьох років. Під час виконання дослідження було звернено увагу на велику кількість робіт на дану тему, особливо на статті, які націлені на вирішення проблем швидкодії веб-додатків, наприклад, [7] та [8].

Для визначення метрик необхідно розглянути продуктивність у двох контекстах – роботи програмного забезпечення написаного на React, а також роботи засобів контролю глобального стану.

При визначенні продуктивності програмного забезпечення зазвичай використовують такі показники, як навантаження на центральний процесор та мережу, використання пам’яті, а також час на виконання запитів і час реакції на дії користувача. І хоча навантаження на мережу і процесор є досить важливими показниками, особливо для мобільних додатків, найбільший вплив на них мають використані структури та алгоритми мови JavaScript або TypeScript, що робить дані показники несуттєвими при визначенні ефективності методів реалізації контролю глобального стану.

Через вищезазначену причину у контексті глобального стану залишаються три метрики:

– використання пам’яті додатком та його окремими компонентами;

– швидкість обробки дій користувача;

– час виконання запитів.

У сучасних реаліях кількість оперативної пам’яті при коректному функціонуванні програмного забезпечення не є проблемою.

Час виконання запитів не залежить від використаного методу реалізації контролю глобальних даних додатка, тому не є суттєвою метрикою у даному дослідженні.

Таким чином, метрика швидкості обробки дій користувача є найважливішим показником. У контексті додатків React та особливо глобального стану цих додатків, дана метрика повинна бути розділена на наступні частини:

– швидкість реагування елементів глобального сховища на дії користувача, що буде вимірюватися у мілісекундах (ms) та означати час, який необхідний додатку на опрацювання дії користувача по завантаженню й оновленню даних використовуючи глобальні сховища;

– кількість оновлень викликаних зміною стану всередині глобальних сховищ, час кожного оновлення та загальний час усіх оновлень будуть вимірюватися відповідно у кількості (одиниці) оновлених компонентів, мілісекундах (ms) на одне оновлення та мілісекундах (ms) витрачених на повне оновлення;

– латентність при виконанні високочастотних оновлень, яка буде вимірюватися у мілісекундах та визначає затримку, що може виникнути при частому оновлені даних у глобальному сховищі та їх відображенні у підписаних до сховища компонентах у порівнянні з джерелом.

Додатковою метрикою, яка позначає другий критерій ефективності роботи програмного забезпечення, а саме вплив обраного методу/засобу на фінальний розмір додатку при його збірці, вимірюється у кілобайтах (Кб) та означає на скільки багато місця займає додаток при використанні тих чи інших методів реалізації глобального сховища.

Таким чином, критерії ефективності роботи програмного забезпечення, які наведені у розділі 2.3 визначаються чотирма метриками, а критерії ефективності розробки програмного забезпечення будуть визначатися порівнянням засобів та методів контролю глобального стану між собою після проведення їх моделювання.

2.5 Розробка методики оцінювання ефективності методів та засобів контролю глобального стану React додатку

2.5.1 Методика оцінювання ефективності

Методика оцінювання ефективності методів та засобів контролю глобального стану React додатків полягає у виконанні наступних дій:

– моделювання продуктивності роботи елементів внутрішнього устрою обраних методів та засобів та формулювання результатів за метриками;

– моделювання продуктивності роботи при взаємодії глобального сховища з React компонентами та формулювання результатів за метриками;

– аналіз отриманого коду, вихідних файлів додатку та можливостей кожної з отриманих реалізацій з метою здійснення порівняння за критеріями ефективності розробки програмного забезпечення;

– розробка рекомендацій по використанню обраних методів та засобів;

– здійснення вибору найбільш ефективного методу, засобу контролю глобального стану React додатків.

Моделювання продуктивності для обраних методів, засобів здійснюється за допомогою розробленої програмної системи, яка дозволяє проведення тестів з різним набором даних та умов.

2.5.2 Процедура моделювання продуктивності роботи компонентів внутрішнього устрою методів та засобів контролю глобального стану додатку

Для моделювання продуктивності роботи внутрішніх елементів обраних методів та засобів контролю глобального стану має бути розроблена програмна система, котра дозволяє вимірювати час, витрачений саме на виконання внутрішніх функцій та методів розробленого глобального сховища. Таким чином, повинні бути виключені усі побічні ефекти, такі як: запити до серверу, виконання складних обчислень, а також виклик функцій, які не мають прямого відношення до поточного методу/засобу контролю стану глобального сховища.

Слід зазначити, що для точності результатів та виключення можливості проблем несумісності, для кожного обраного методу/засобу з контролю глобального сховища повинна бути розроблена окрема програмна система у вигляді клієнтського додатку React, яка має лише фунціонал, необхідний для конкретної реалізації глобального сховища відповідно до документацій, які

наведені у [9-11]. Втім передбачається, що кожна програмна система буде використовувати однакові React компоненти.

Кожен метод та засіб контролю глобального сховища може мати одну чи декілька альтернативних реалізацій, які приведені у документаціях, форумах або існують як можливі рішення проблем базових реалізацій.

Для отримання повних результатів час виконання методів контролю стану глобального сховища повинен бути прив’язаний до розміру даних, над якими проводилися операції. Таким чином, передбачається проведення декількох моделювань з різними вибірками даних, кожне з яких буде виконане не менш за 20 разів для усунення можливих випадкових величин та помилок. І хоча двадцяти тестувань не достатньо для повного усунення випадкових величин, така кількість дозволить значно зменшити їх кількість, чого буде достатьо при використанні середнього значення з усіх тестів. Зважачи на те, що повна автоматизація під час даного моделювання нездійснена, проведення більшої кількості тестів не є можливим через брак часу та ресурсів.

Моделюванню підлягатимуть наступні дії:

– початкове завантаження даних з серверу з подальшим збереженням їх у глобальному сховищі та передання до компонентів;

– оновлення даних, які знаходяться у глобальному сховищі, зумовлені діями користувача у компонентах.

Моделювання продуктивності роботи внутрішніх елементів є важливою складовою для отримання результатів оцінювання продуктивності роботи самого глобального сховища у середі, яка максимально відокремлена від самих React компонентів та шляху їх розробки.

Незважаючи на це, проаналізовані дослідження, наприклад, [12-14], були націлені на взаємодію засобу контролю глобального стану з компонентами React, а не на продуктивність роботи самого методу/засобу.

2.5.3 Процедура моделювання продуктивності взаємодії React компонентів з глобальним сховищем

Моделювання продуктивності роботи внутрішнього устрою є важливою складовою, але не дозволяє оцінити поведінку й продуктивність роботи додатку в цілому. Так як будь-яке глобальне сховища створюється для його використання у безлічі різних компонентів, моделювання продуктивності взаємодії між цими компонентами і глобальним сховищем дозволить отримати результати, які можуть або підкріпити переваги швидкої роботи внутрішніх елементів, або повність знехтувати наявністю через зазначені недоліки у їх інтеграції.

Для моделювання вищезазначеної продуктивності будуть використовуватися вже розроблені додатки. Для вимірювання продуктивності взаємодії компонента та глобального сховища будуть використовуватись як профілювання за допомогою інструментів, наявних у браузері, так і додатково розроблені компоненти React.

Профілювання браузеру дозволить отримати дані про загальну кількість оновлень, їх час, а також надасть інформацію щодо тих оновлень, які були пропущені через відсутність необхідності їх оновлення.

При моделювання продуктивності взаємодії компонентів з глобальним сховищем основна увага повинна бути приділена відмальовуванню цих компонентів.

Відмальовування компонентів – далі «рендеринг», при першому відмальовуванні компонента, та «ререндеринг», при подальших відмальовуваннях – є дуже важливим моментом у розробці на React. Усі React компоненти мають життєвий цикл, який відповідає за їх поведінку й те, які дані та у якому вигляді вони відображуються на сторінці. При створенні компоненту відбувається його перший рендеринг, який, зазвичай, займає найбільший час. У разі, коли нам необхідно оновити які-небудь дані, інтерфейс чи стилі, ми повинні спричинити ререндеринг компоненту.

У додатках з великою кількістю компонентів особливу увагу потрібно приділити тому, у яких ситуаціях виконується рендеринг та чи потрібен він взагалі. Це важливо, бо велика кількість зайвих перемалювань може призвести до значного зниження швидкості роботи додатку.

У додатках, які використовують глобальні сховища, відстежування зайвих рендерингів є навіть більшою проблемою, бо у більшості випадків їх причина може бути не досить очевидною, а наслідки навіть більш небезпечними, наприклад, у разі, коли одні дані використовуються багатьма компонентами.

Дослідження продуктивності взаємодії між компонентами та глобальними сховищами у такому разі зводиться до відстеження наслідків оновлення даних у сховищах, а саме до пошуку зайвих рендерингів і, якщо можливо, пошуку реалізацій, які можуть їх усунути.

Таким чином, передбачається дослідження початкового рендерингу та подальших рендерингів компонентів для кожної з реалізацій, отриманих у результаті моделювання продуктивності внутрішнього устрою компонентів, відповідно до опису, наведеного у розділі 2.5.2.

3 РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ТА ЗАСОБІВ КОНТРОЛЮ ГЛОБАЛЬНОГО СТАНУ

3.1 Опис обраних технологій

У зв’язку з тим, що предметом дослідження є методи та засоби контролю глобального стану React додатків, єдиною доступною технологією для клієнтської частини є сама бібліотека React.

React бібліотека передбачає використання мови програмування JavaScript, але, у останні роки, широкого розповсюдження набувають React додатки, які використовують TypeScript.

Порівнюючи з JavaScript, основною перевагою TypeScript є підтримка статичної типізації. За задумом розробників, такий підхід має підвищити можливості рефакторингу, повторного використання коду, а також значно зменшити кількість помилок завдяки їх автоматичному пошуку на етапі розробки.

Не менш важливою перевагою TypeScript є доступ до функціоналу, який ще не входить до ядра JavaScript. У разі використання JavaScript та необхідності доступу до вищезазначеного функціоналу, розробнику потрібно використовувати транспайлер, наприклад, Babel. Більш за те, використання TypeScript надає доступ до широкого функціоналу, який може пришвидшити та спростити розробку, але взагалі не присутній у JavaScript.

Додатковою причиною вибору мови програмування TypeScript для розробки програмної системи для дослідження ефективності методів та засобів контролю глобального стану є залежність бібліотеки MobX від декораторів, робота яких була описана у розділі 1.2.2. Якщо TypeScript має їх нативну підтримку, то JavaScript потребує додаткових налаштувань IDE та використання транспайлеру, наприклад, вищезазначеного Babel.

За середовище розробки (або IDE) була обрана Visual Studio Code, яка надає широкі можливості по розробці як серверних, так і клієнтських додатків. Visual

Studio Code надає розробникам підтримку більшості популярних мов програмування, а також дозволяє використовувати велику кількість розширень. Дані розширення додають до інтегрованого середовища розробки підтримку структур різних мов програмування, бібліотек та технологій, у тому числі й React, що дозволяє спростити розробку, відлагодження та запуск додатків.

Окрім зазначених переваг, Visual Studio Code має зручний інтерфейс та термінал для керування запуском багатьох додатків, що є невід'ємною частиною даного дослідження.

Для самого запуску додатків буде використовуватися середовище виконання Node.js. Дане середовище є невід'ємною частиною як серверної частини, яка буде описана нижче, так і клієнтських додатків на React. Node.js та його менеджер пакетів npm дозволяє зручно встановлювати та оновлювати пакети (бібліотеки, фреймворки та інші інструменти), до яких входить як сам React, так і досліджуємі бібліотеки Redux та MobX.

Для розробки серверу додатку було обрано вже зазначений Node.js та веб-фреймворк Express.js. Даний фреймворк має широкі можливості для розробки веб-додатків та створення швидких, безпечних та надійних API. Сервер додатку буде використовуватися для обробки запитів від клієнтських додатків, роботи з базою даних та запуску збудованих React додатків. Node.js сервер разом з клієнтською частиною можуть бути розміщення майже на будь-якому хостингу.

Для роботи з базою даних буде використовуватися MongoDB, а саме ODM (object data modeling) mongoose.

MongoDB є документо-орієнтованою базою даних. Перевагою цієї бази даних, як й інших нереляційних баз даних, порівняння яких влучно подано у статті [15], є збереження часу на опис таблиць та їх взаємодії. Усі дані зберігаються у форматі JSON, що дозволяє зберегти будь-які дані у вигляді звичайного рядка й, у свою чергу, надає гнучкості даним. Утім, такий підхід не є досить безпечним через відсутність будь-якого контролю структури. Використання бібліотеки mongoose, у свою чергу, надає такий контроль.

Mongoose дозволяє визначати суворо типізовану схему для об’єкту, який представляє документ у базі даних. Такий підхід надає можливість розробнику самому визначати степінь контролю типів об’єктів у базі даних і, у комбінації з мовою TypeScript, дозволяє досить швидко створити стійку систему, яка легко відстежує помилки, не обмежуючи розробника, як це роблять реляційні бази даних.

Для збірки клієнтських додатків був обраний Webpack, який є найбільш популярним засобом для збірки React додатків і навіть внутрішньо використовується у шаблоні від розробників під назвою create-react-app. Хоча даний шаблон є досить популярним та гарно організованим, було вирішено використовувати власноруч написані конфігурації. Такий підхід надає багато переваг, серед яких основними є:

– відсутність модулів та пакетів, які не використовуються у додатку;

– повний контроль роботи додатку;

– можливість написання конфігурацій Webpack з нуля.

Найважливішою перевагою відмови від шаблону create-react-app є саме можливість написання Webpack конфігурацій з нуля. Це дозволяє:

– обирати як саме будуть оброблюватися окремі типи файлів, бібліотеки;

– налаштовувати метод побудови додатків для їх подальшого використання у express.js;

– налаштовувати вхідні дані для додатків;

– оптимізувати окремі типи вихідних файлів.

Така гнучкість необхідна для моделювання роботи великої кількість реалізацій методів та засобів контролю глобального сховища.

Для контролю версій буде використовуватися Git, який наразі є найбільш ефективним та надійним інструментом серед своїх аналогів. Використання Git у даному досліджені важливо через необхідність роботи як з додатками, кожна дія яких контролюється та вимірюється сервісом для відправки на сервер, так і з їх полегшеними версіями, які мають лише необхідний функціонал для роботи React компонентів та засобів контролю глобального стану. Перехід між даними версіями

додатків повинен бути швидким, що з легкістю досягається за допомогою створення декількох «гілок» за допомогою Git. Також Git використовується для швидкого завантаження додатків на віддалений сервер.

3.2 Програмна реалізація

Для дослідження ефективності методів та засобів контролю глобального стану React додатків потрібна програмна система, яка дозволяє змоделювати роботу кожного з обраних інструментів та керувати їх роботою.

Першим етапом побудови програмної системи є визначення її архітектури. При розробці програмного забезпечення на React, додаток зазвичай складається з багатьох компонентів, які можна розподілити на наступні типи:

– компоненти-контейнери, які агрегують у собі інші компоненти та, зазвичай, можуть містити у собі сторінки веб-сайту, навігацію, деяку глобальну логіку та інше;

– компоненти-контейнери, які представляють деяку конкретну сторінку або велику частину додатку зі значною кількістю логіки для компонентів, які відповідають за користувацький інтерфейс;

– “presentational” компоненти – компоненти, які відповідають за користувацький інтерфейс, зазвичай не мають у собі великої кількості логіки, але відповідають за рендеринг інтерфейсу, базуючись на даних, отриманих від компонентів-контейнерів. Presentational компоненти можуть містити в собі багато інших дочірніх presentational компонентів.

У більшості випадків React розробники намагаються уникати операцій з даними у presentational компонентах розміщуючи усю логіку у компонентах-контейнерах, тому самі presentational компоненти не повинні залежати від реалізації глобального сховища, а лише відображувати ті дані, які їм були надані компонентами-контейнерами.

Усі з вищезазначених типів компонентів будуть використовуватися у програмній системі. Слід зазначити, що важливо розробити такі компоненти, функціонал яких можливо використовувати у реальних додатках або, якнайкраще, вже використовується повсюди у тому чи іншому видах.

Спираючись на власний досвід розробки додатків на React, додаток для моделювання роботи засобів та методів контролю глобального стану буде мати наступні структурні елементи:

– presentational компоненти, які представляють собою елемент списку, у нашому випадку – елемент списку блогу. Дані компоненти є загальними для будь-якого з обраних засобів, так як не залежить від реалізації глобального сховища;

– presentational компоненти, які представляють собою елемент фільтру для списку. Як і в попередньому випадку, дані компоненти є загальними для кожної реалізації;

– компоненти-контейнери, які працюють з глобальним сховищем, передають дані до вищезазначених presentational компонентів та оброблюють дії користувача. Реалізації даних компонентів будуть виконані окремо для кожного з обраних реалізації глобального сховища;

– глобальне сховище, яке є специфічним для кожного з обраних методів та засобів, а також може мати декілька реалізацій;

– сервіс для вимірювання метрик ефективності засобу. Даний сервіс повинен мати можливість використання у будь-якому розробленому компоненті та глобальному сховищі. Сервіс буде використовуватися для вимірювання швидкості роботи внутрішніх функцій глобального сховища, швидкості обробки дій ініційованих користувачем і системою та швидкості передачі цієї інформації від сховища до цільового компоненту;

– сервер та база даних для агрегації даних від сервісу, а також можливості одночасного запуску збудованих React додатків;

– компоненти-контейнери та presentational компоненти для представлення агрегованих даних з серверу. Дані компоненти повинні надавати можливість

порівняти кожний засіб та отримати агреговані дані для представлення у даному дослідженні.

На рисунку 7 зображено архітектуру програмної системи для моделювання роботи засобів та методів реалізації глобального сховища.

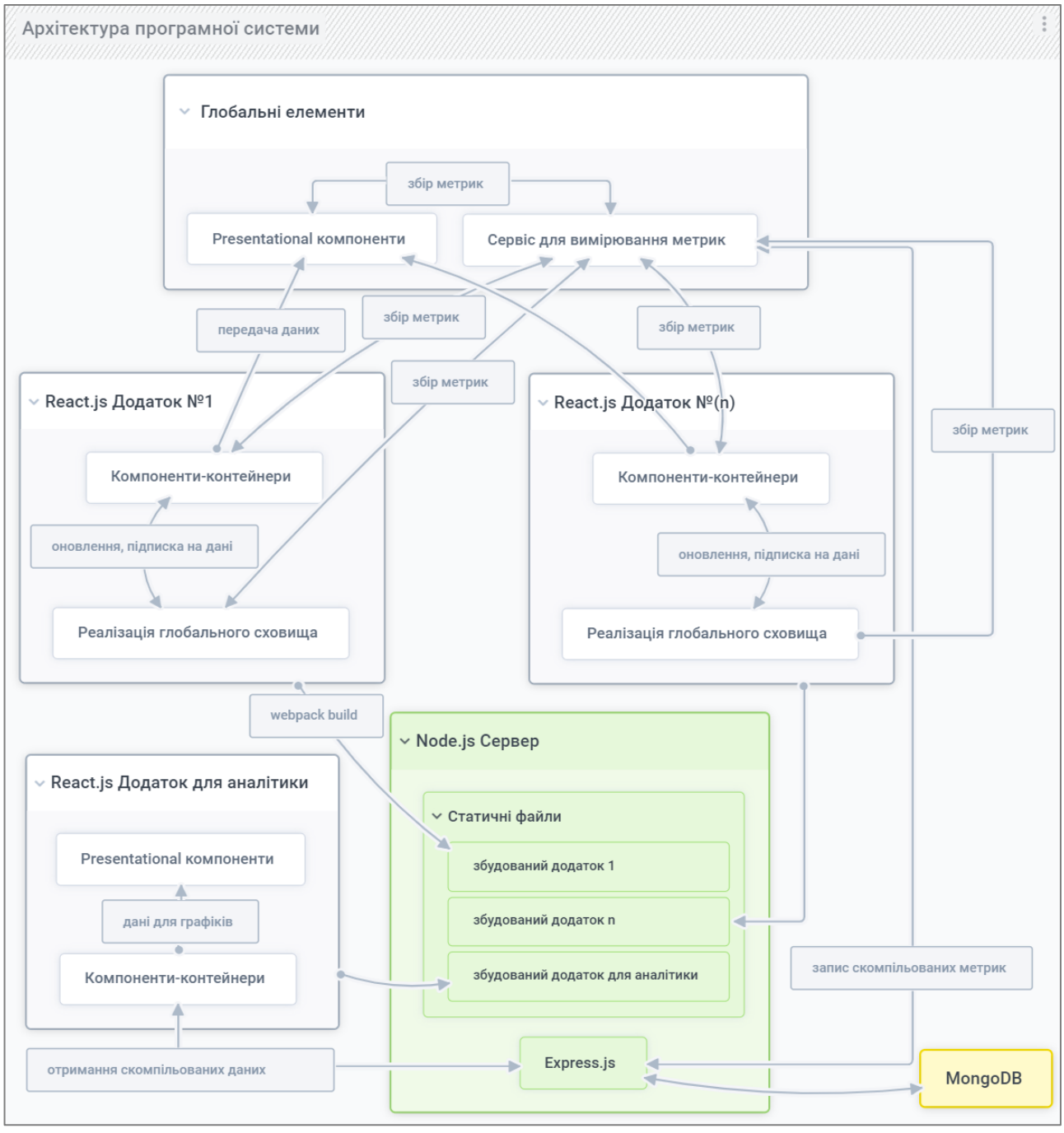

Рисунок 7 – Діаграма архітектури програмної системи

Як видно з вищенаведеної діаграми (див. рис. 7), для тестування ефективності буде розроблена множина додатків, кожен з яких буде використовувати різний

засіб реалізації глобального сховища або його окремих елементів. При цьому усі додатки будуть використовувати глобальні компоненти для вимірювання метрик ефективності та presentational компоненти, що представляють частини попередньо-визначеного функціоналу. Слід зауважити, що частина presentational компонентів може буде модифікована для задоволення вимог тієї чи іншої реалізації. Передбачається, шо усі компоненти, які використовуються для вимірювання метрик будуть ідентичними у кожному додатку. У кожному з розроблених додатків будуть створені свої компоненти-контейнери, які будуть реалізувати взаємодію з глобальним сховищем. Кількість додатків буде дорівнювати кількості виділених реалізацій.

На рисунку 8 зображено приклад розроблених компонентів додатку з однією з реалізацій глобального сховища.

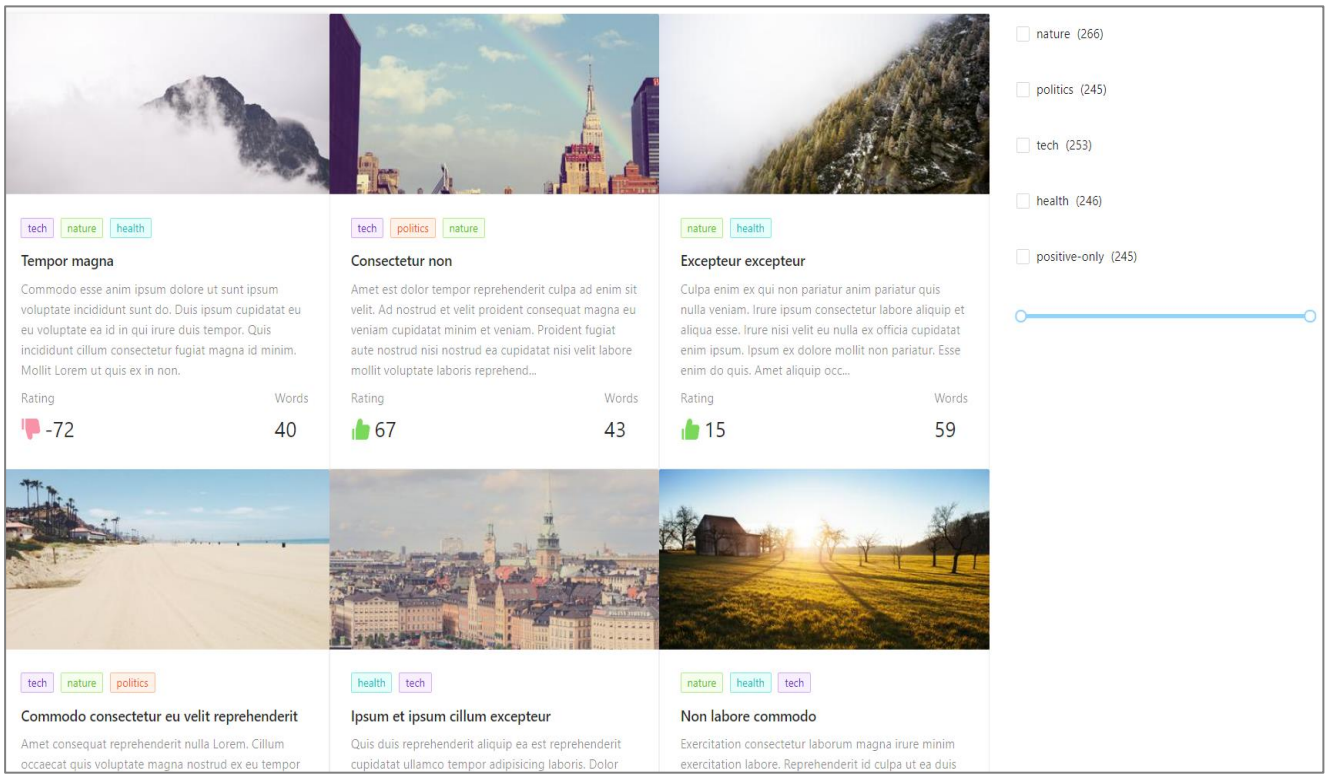

Рисунок 8 – Приклад розроблених компонентів React додатка

При завантаженні сторінки та при оновленні даних на ній за допомогою компоненту фільтрів, сервіси будуть збирати інформацію й передавати її на сервер.

Node.js сервер буде використовувати Express.js фреймворк для створення роутингу. Також будуть створенні обробники, які будуть викликатися з React додатків для отримання, збереження, модифікації даних у базі даних.

Додаток для аналітики буде отримувати дані сервісів, які були скомпільовані на сервері та видавати їх у графічному вигляді. Приклад одного з можливих результатів, отриманий додатком зображено на рисунку 9.

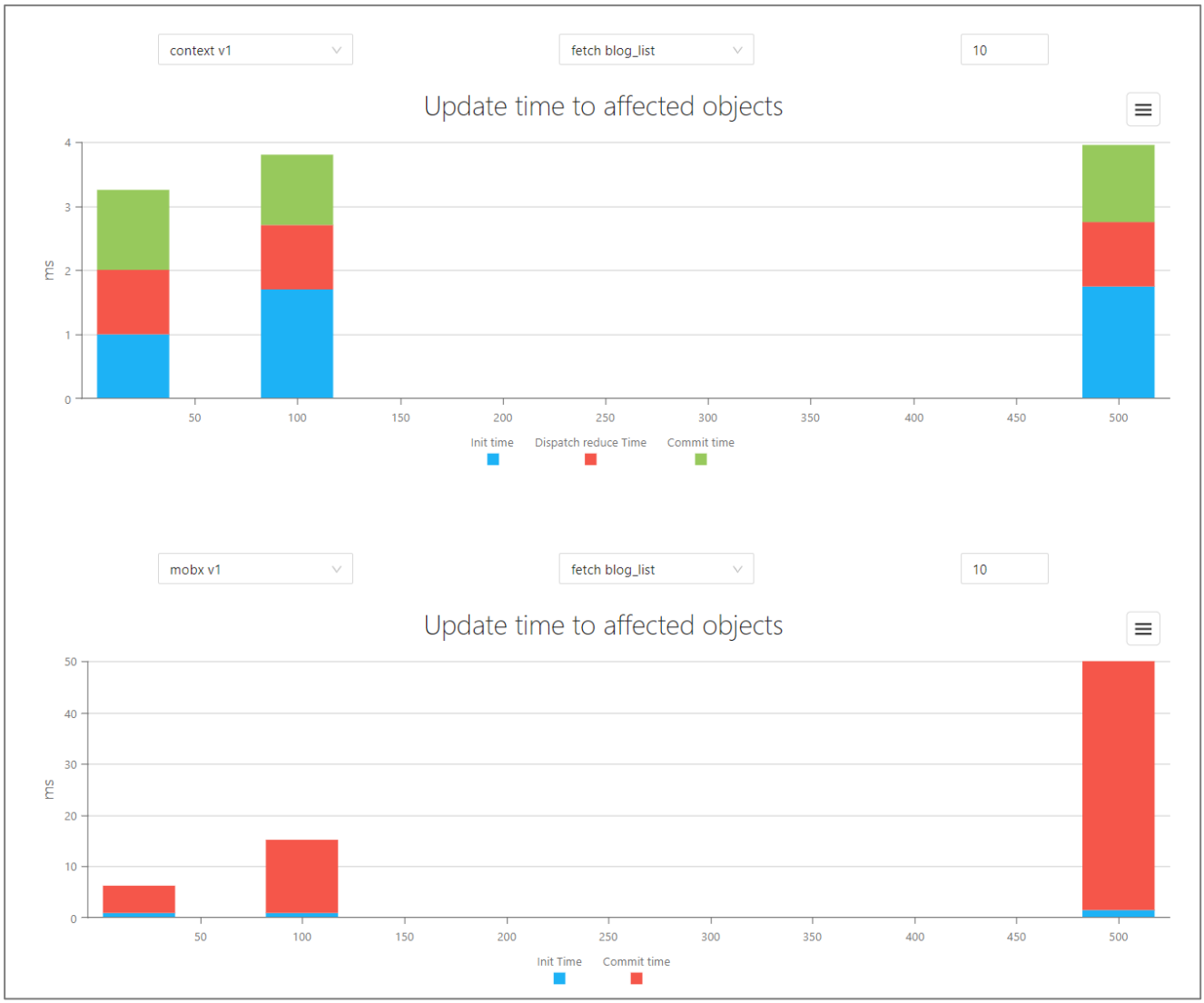

Рисунок 9 – Приклад роботи додатку аналітики

Графічні результати будуть використовуватися у наступному розділі для аналізу ефективності засобів та методів контролю глобального стану.

3.3 Тестування програмної системи

Тестування програмної системи є важливим етапом, який дозволяє бути впевненим, що отриманий додаток відповідає всім поставленим вимогам, працює правильно і стабільно.

Приймаючи до уваги той факт, що неможливо покрити усі тестові випадки, стратегія тестування полягає у проведені можливих тестів за доступний час та ресурси.

Для тестування розробленої програмної системи було вирішено обрати функціональне тестування. Передумовою для проведення такого типу тестування є наявність функціональних вимог до програмного забезпечення.

Таким чином, до системи висунуті наступні вимоги:

– ендпоінт «/api/blogs» при запиті типу GET без передачі параметру повинен повертати список усіх блогів, які наявні у базі даних;

– ендпоінт «/api/blogs» при запиті типу GET з передачею числового параметру «limit» повинен повертати визначену цим параметром кількість блогів, які наявні у базі даних;

– ендпоінт «/api/tracker/redux» при запиті типу POST з JSON даними, які відповідають TypeScript інтерфейсу для Redux операції, за відсутності запису з однаковими параметрами, повинен створювати новий запис Redux операції у базі даних, записувати метрики часу у масив та встановлювати кількість зразків у значення «1». Відповіддю ендпоінту є створений запис з відміткою «new»;

– ендпоінт «/api/tracker/redux» при запиті типу POST з JSON даними, які відповідають TypeScript інтерфейсу для Redux операції, за присутності запису з однаковими параметрами, повинен оновлювати запис Redux операції у базі даних, додавати метрики часу у масив, збільшувати кількість зразків на «1» та оновлювати середнє значення часу. Відповіддю ендпоінту є оновлений запис з відміткою «updated»;

– ендпоінт «/api/tracker» при запиті типу GET з параметрами «source» та «action» повинен повертати список усіх елементів Tracker у базі даних, поля яких відповідають переданим параметрам;

– ендпоінт «/api/tracker» при запиті типу GET з параметрами «source», «action» та «limit» повинен визначену параметром «limit» кількість елементів Tracker, поля яких відповідають переданим параметрам;

– ендпоінт «/api/profiler» при запиті типу POST з JSON даними, які відповідають TypeScript інтерфейсу для Profiler, за відсутності запису з однаковими параметрами, повинен створити новий запис у базі даних, записати метрики часу у масив та встановлювати кількість зразків у значення «1». Відповіддю ендпоінту є створений запис з відміткою «new»;

– ендпоінт «/api/profiler» при запиті типу POST з JSON даними, які відповідають TypeScript інтерфейсу для Profiler, за присутності запису з однаковими параметрами, повинен оновлювати запис у базі даних, записувати метрики часу у масив, збільшувати кількість зразків на «1» та підраховувати середнє значення часу. Відповіддю ендпоінту є створений запис з відміткою «updated».

На основі перелічених функціональних вимог можна створити тестові випадки для подальшого функціонального тестування.

Деякі тестові випадки можуть мати передумови, які необхідно виконати для отримання правильних результатів.

Далі наведені результати тестування за допомогою тестових випадків з порівнянням очікуваного результату з фактичним результатом тесту та наочним зображенням результатів у вигляді рисунків.

Розглянемо перший тестовий випадок. Передумови: база даних має хоча б один запис блогу. Очікуваний результат: GET запит до ендпоінту «/api/blogs» без передачі параметру поверне список усіх блогів, які наявні у базі даних. Фактичний результат: GET запит до ендпоінту «/api/blogs» без передачі параметру отримав

результат у JSON форматі, який має список усіх блогів, наявних у базі даних. Результат тестування наведено на рисунку 10.

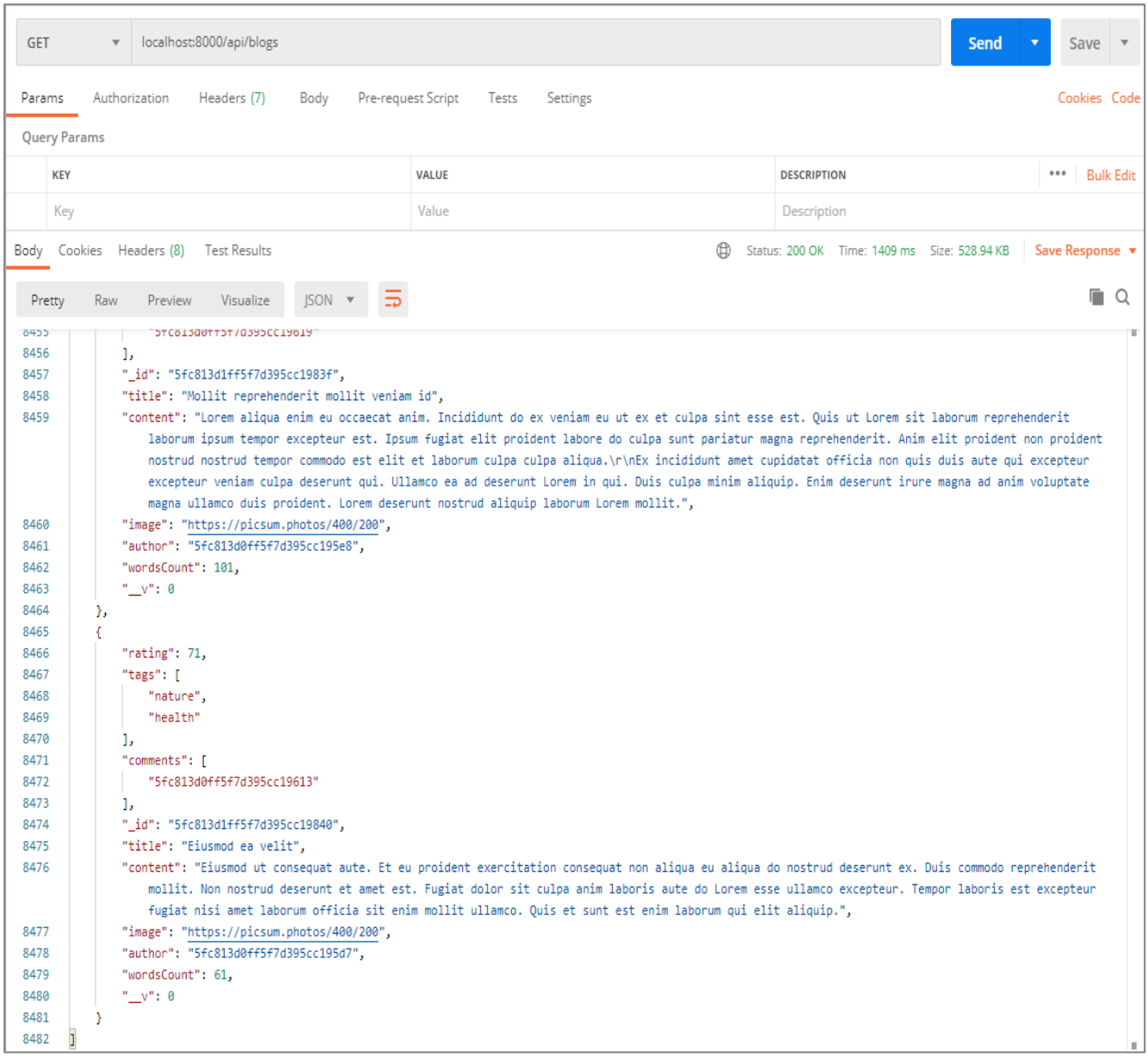

Рисунок 10 – Відповідь GET запиту до ендпоінту «/api/blogs» без передачі параметру

Розглянемо другий тестовий випадок. Передумови: база даних має хоча б один запис блогу. Очікуваний результат: GET запит до ендпоінту «/api/blogs» з передачею параметру «limit» поверне визначену цим параметром кількість блогів, які наявні у базі даних. Фактичний результат: GET запит до ендпоінту «/api/blogs»

з передачею параметру «limit» повернув результат у JSON форматі, який має масив блогів розміром у параметр «limit». Результат тестування наведено на рисунку 11.

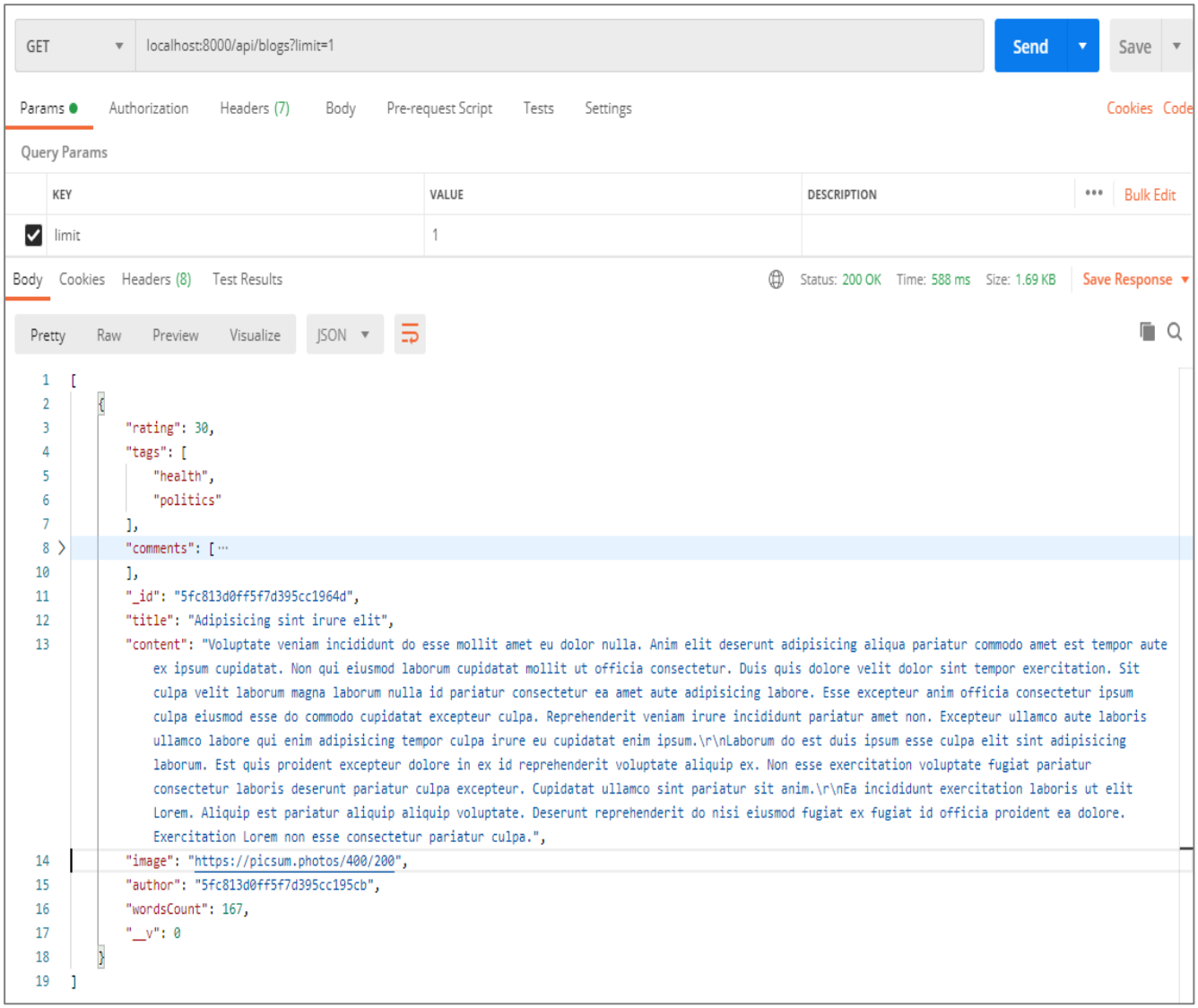

Рисунок 11 – Відповідь GET запиту до ендпоінту «/api/blogs» з передачею параметру «limit»

Розглянемо третій тестовий випадок. Передумови: немає. Очікуваний результат: POST запит до ендпоінту «/api/tracker/redux» з відповідними даними, створить новий запис Redux операції у базі даних у який буде записано метрики часу та встановлено кількість зразків у значення «1». Створений об’єкт з відміткою «new» повернуто як результат запиту. Фактичний результат: POST запит до ендпоінту «/api/tracker/redux» повернув об’єкт з відміткою «new», який має масив з

часом та кількість зразків якого дорівнює «1». Результат тестування наведено на рисунку 12.

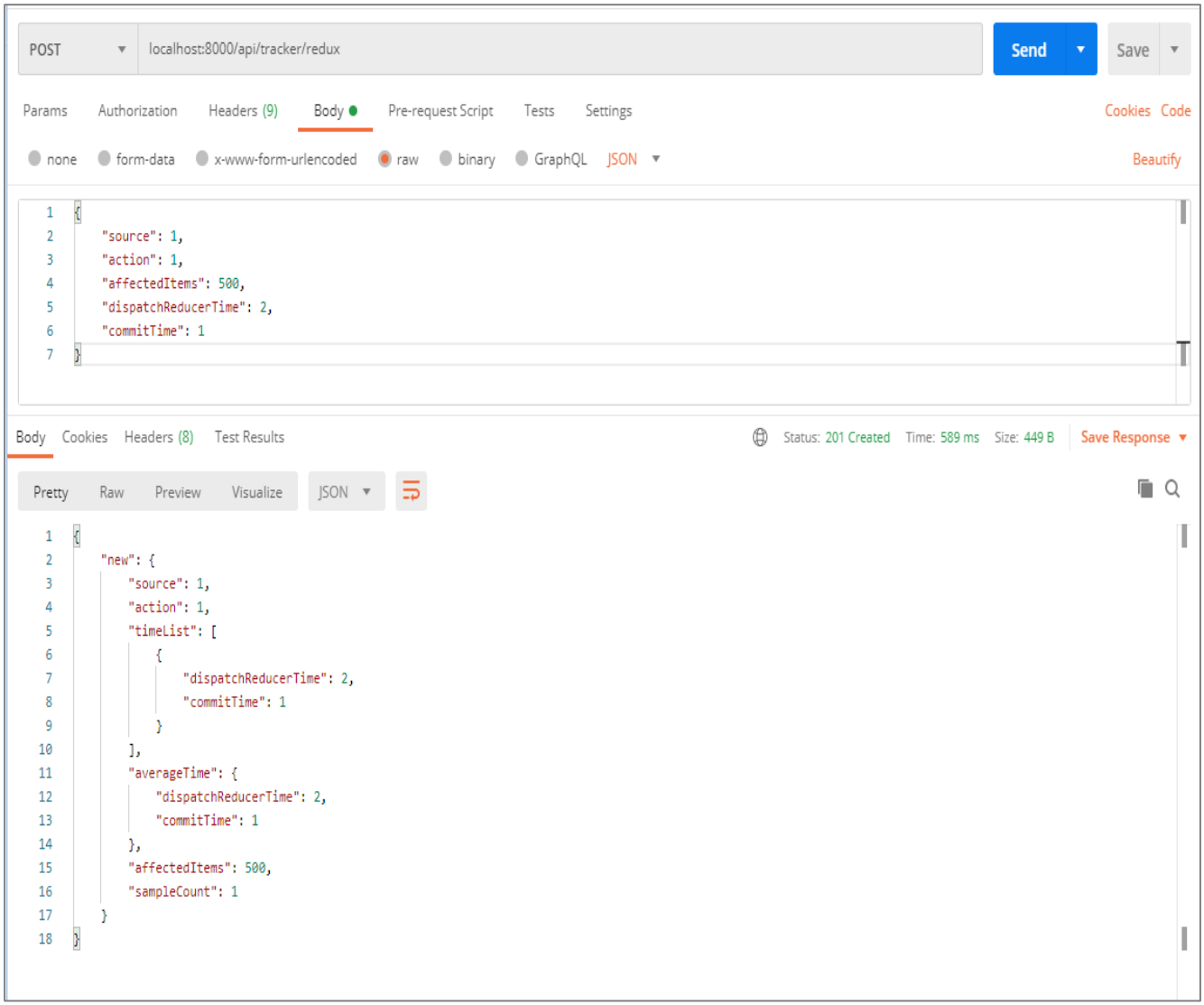

Рисунок 12 – Відповідь POST запиту до ендпоінту «/api/tracker/redux» за відсутності запису з однаковими параметрами

Розглянемо четвертий тестовий випадок. Передумови: база даних зберігає Redux операцію з параметрами, які співпадають з параметрами поточного запиту. Очікуваний результат: POST запит до ендпоінту «/api/tracker/redux» з відповідними даними, оновить запис Redux операції у базі, додасть нові метрики часу до масиву, збільшить кількість зразків на «1» та підрахує новий середній час. Оновлений об’єкт з відміткою «updated» повернуто як результат запиту. Фактичний результат: POST запит до ендпоінту «/api/tracker/redux» повернув об’єкт з відміткою

«updated», який має оновлений масив з часом, кількість зразків якого дорівнює «2» та який має оновлений середній час. Результат тестування наведено на рисунку 13.

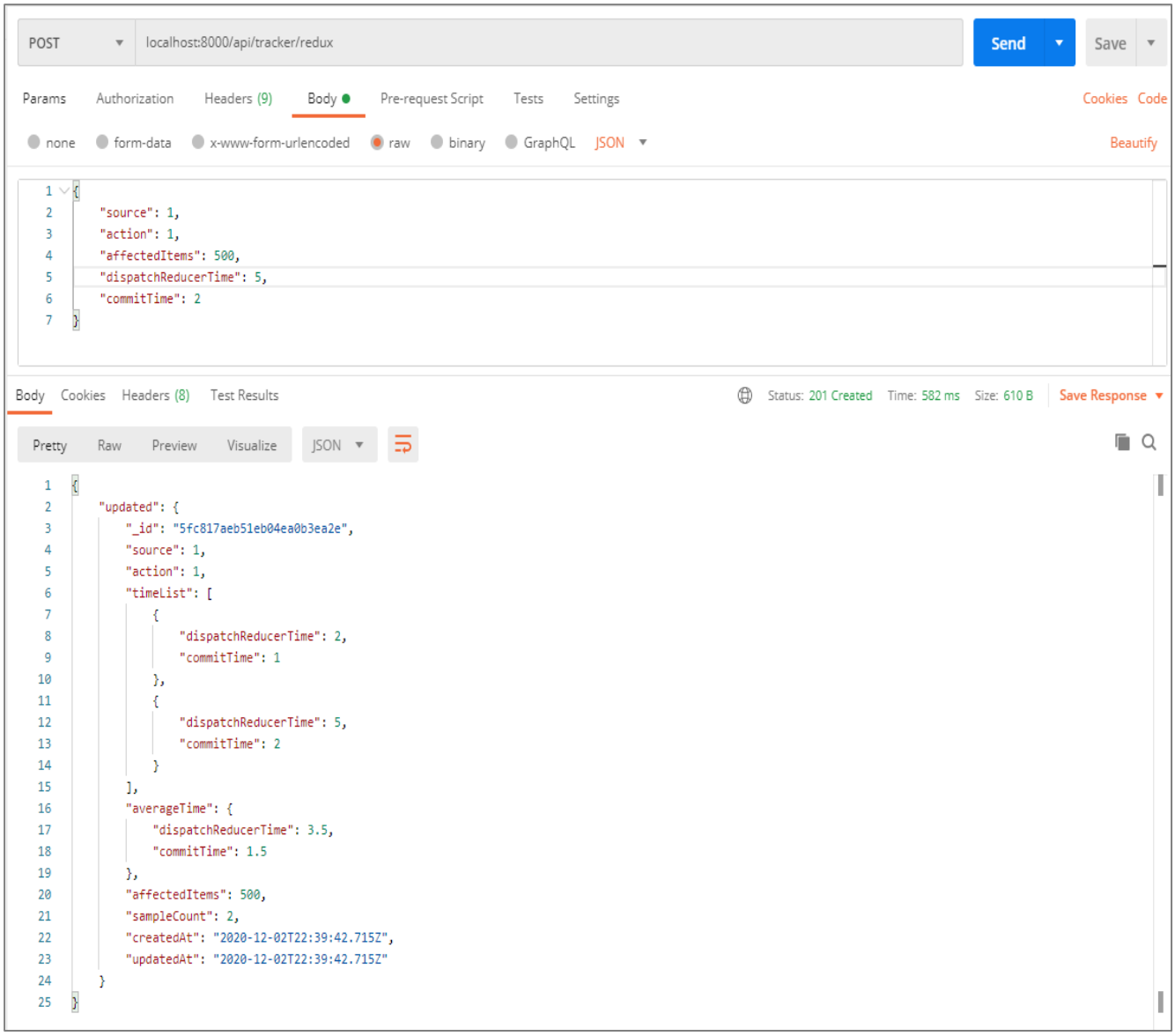

Рисунок 13 – Відповідь POST запиту до ендпоінту «/api/tracker/redux» за присутності запису з однаковими параметрами

Розглянемо п’ятий тестовий випадок. Передумови: база даних має хоча б один запис з параметрами поточного запиту. Очікуваний результат: GET запит до ендпоінту «/api/tracker» з параметрами «source» та «action» поверне список усіх елементів Tracker, поля яких відповідають переданим параметрам. Фактичний результат: GET запит до ендпоінту «/api/tracker» з параметрами «source» та «action» повернув результат у вигляді JSON, який містить масив усіх елементів Tracker,

поля яких відповідають переданим параметрам. Результат тестування зображено на рисунку 14.

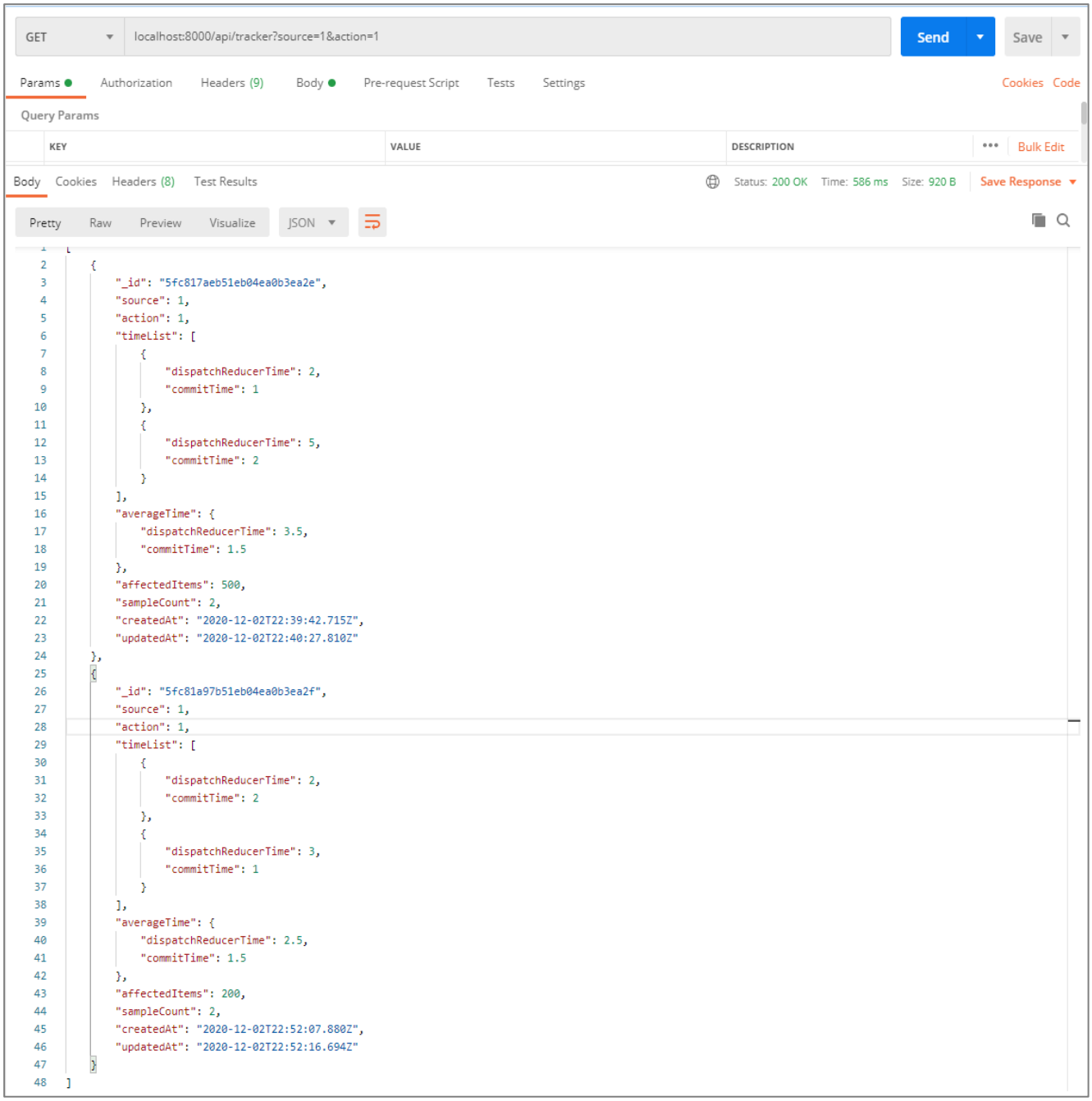

Рисунок 14 – Відповідь GET запиту до ендпоінту «/api/tracker» з параметрами «source» та «action»

Розглянемо шостий тестовий випадок. Передумови: база даних має хоча б один запис з параметрами поточного запиту. Очікуваний результат: GET запит до ендпоінту «/api/tracker» з параметрами «source», «action» та «limit» поверне визначену параметром «limit» кількість елементів Tracker, поля яких відповідають

переданим параметрам. Фактичний результат: GET запит до ендпоінту «/api/tracker» з параметрами «source» та «action» повернув результат у вигляді JSON, який має масив елементів Tracker розмір якого дорівнює параметру «limit». Результат тестування наведено на рисунку 15.

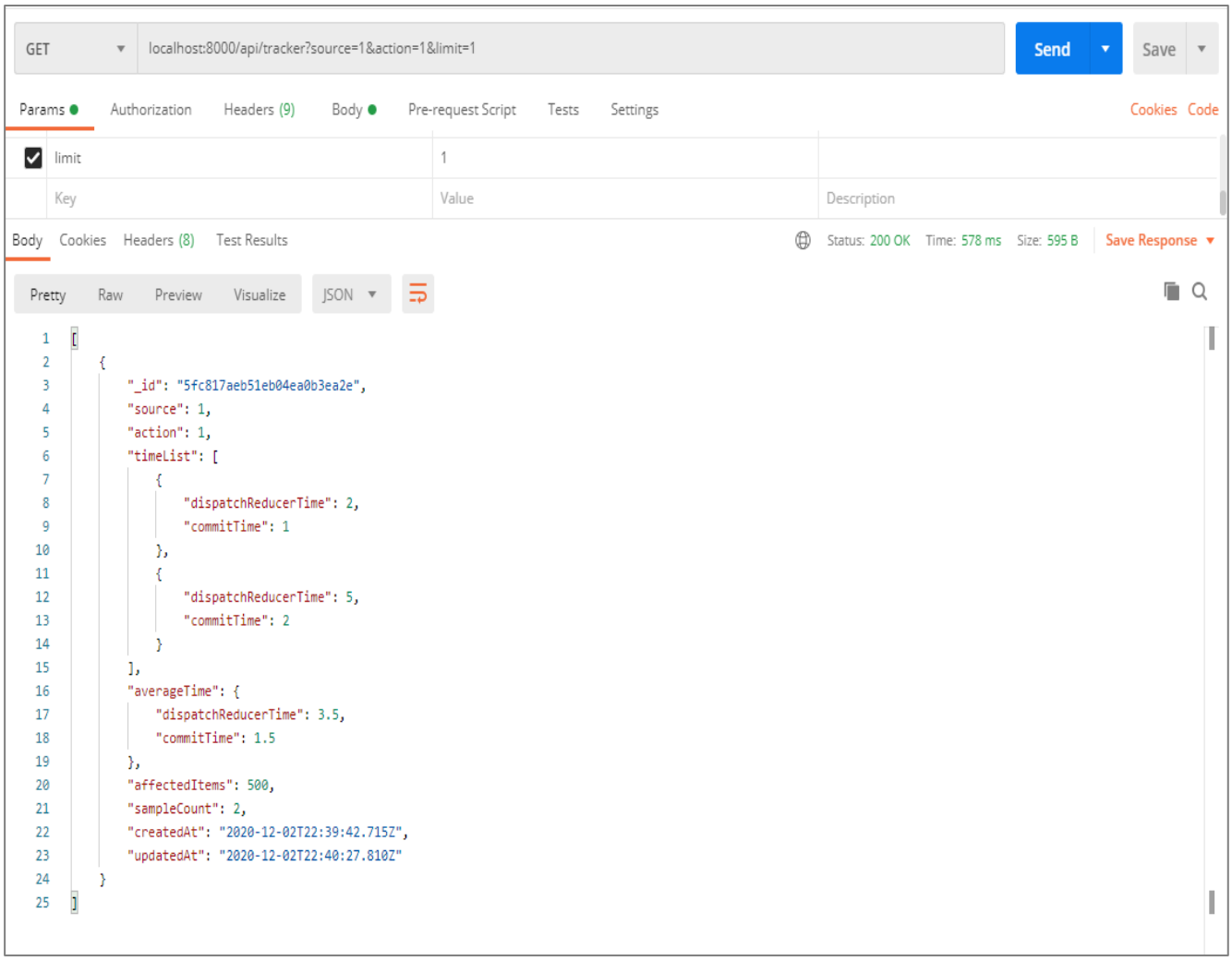

Рисунок 15 – Відповідь GET запиту до ендпоінту «/api/tracker» з параметрами «source», «action» та «limit»

Розглянемо сьомий тестовий випадок. Передумови: немає. Очікуваний результат: POST запит до «/api/profiler» з відповідними даними, створить новий запис у базі даних у який буде записано метрики часу та встановлено кількість зразків у значення «1». Створений об’єкт з відміткою «new» повернуто як результат запиту. Фактичний результат: POST запит до ендпоінту «/api/profiler» повернув

об’єкт з відміткою «new», який має масив з часом та кількість зразків якого дорівнює «1». Результат тестування наведено на рисунку 16.

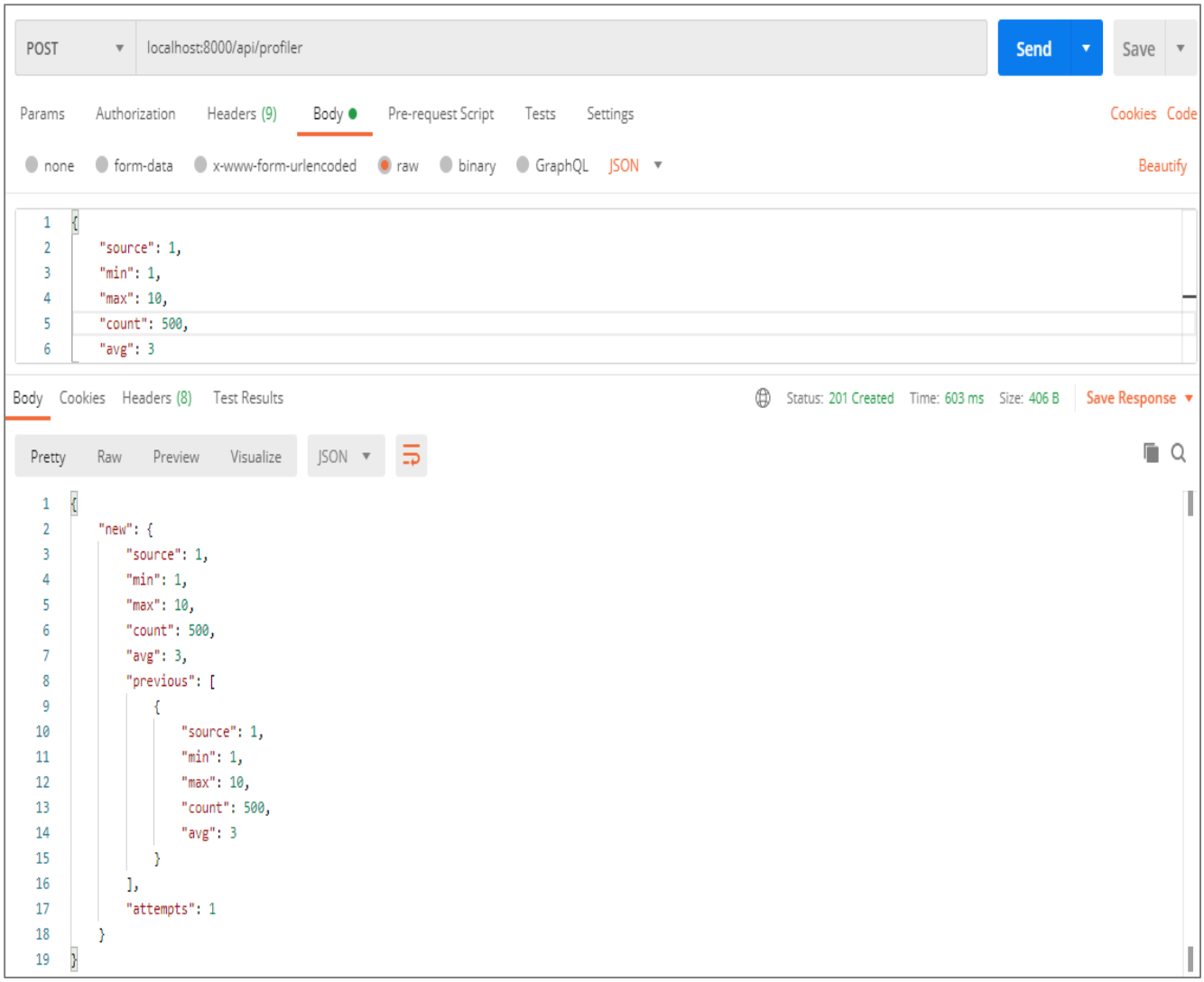

Рисунок 16 – Відповідь POST запиту до ендпоінту «/api/profiler» за відсутності запису з однаковими параметрами

Розглянемо восьмий тестовий випадок. Передумови: база даних зберігає Profiler запис з параметрами, які співпадають з параметрами поточного запиту. Очікуваний результат: POST запит до «/api/profiler» з відповідними даними оновить запис Profiler у базі, додасть нові метрики часу до масиву, збільшить кількість зразків на «1» та підрахує новий середній час. Оновлений об’єкт з відміткою «updated» повернуто як результат запиту. Фактичний результат: POST запит до ендпоінту «/api/profiler» повернув об’єкт з відміткою «updated», який має

оновлений масив з часом, кількість зразків якого дорівнює «2» та який має оновлений середній час. Результат тестування наведено на рисунку 17.

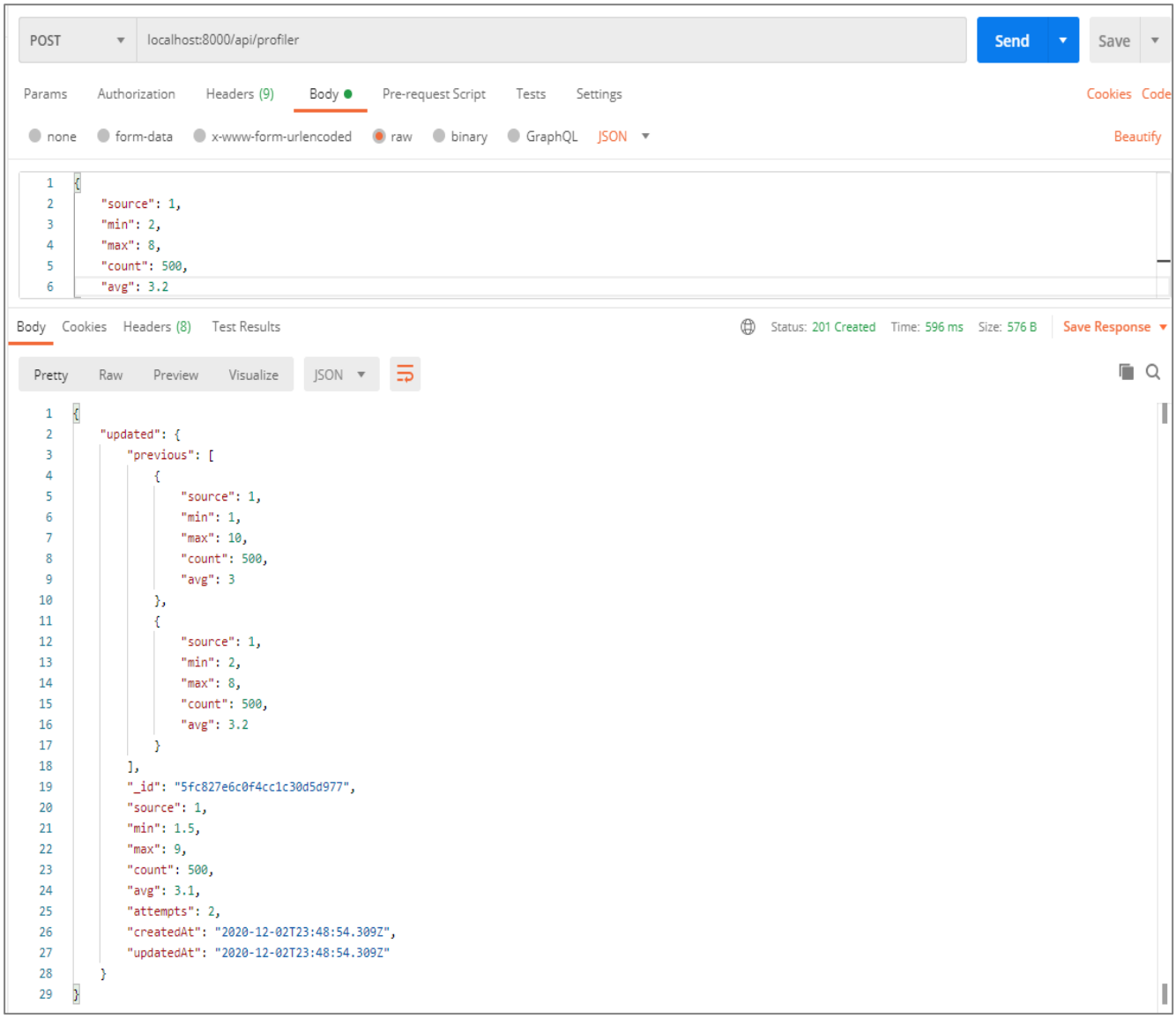

Рисунок 17 – Відповідь POST запиту до ендпоінту «/api/profiler» за присутності запису з однаковими параметрами

З результатів проведеного функціонального тестування по тестовим випадкам можна зробити висновок, що програмна система повністю відповідає поставленим до неї вимогам та може використовуватися за призначенням.

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ТА ЗАСОБІВ КОНТРОЛЮ ГЛОБАЛЬНОГО СТАНУ REACT ДОДАТКІВ

4.1 Дослідження продуктивності роботи компонентів внутрішнього устрою методів та засобів контролю глобального стану додатку

Базуючись на описі методики оцінювання ефективності та процедурі моделювання, наведених у розділі 2.5, для оцінювання продуктивності компонентів внутрішнього устрою обраних методів та засобів контролю глобального стану React додатків були виділені критичні точки для кожного з них. Для бібліотеки MobX – це час ініціації дії від компоненту та час фіксації дії (час, потрібний для отримання оновлених даних у компонентах). Для бібліотеки Redux при асинхронному завантаженні даних з серверу – це час від ініціалізації дії, викликаної компонентом, до обробки цієї дії бібліотекою Redux-Saga [16], яка використовується для асинхронних дій; час передачі дії від Redux-Saga до Redux редьюсера [17]; час від фіксації оновлених даних до отримання їх компонентом. Для бібліотеки Redux при синхронному оновленні даних, ми не використовуємо Redux-Saga, тому вимірюються лише ініціація дії від компонента до Redux редьюсера та час фіксації дії у зворотному напряму. Для Context API вимірюємо час ініціалізації, час ініціалізації дії у Redux-подібній функції-редьюсері та час фіксації. Таким чином, вимірюється лише час роботи інструментів самого засобу, виключаючи усі додаткові розрахунки й запити в мережі.

Для вимірювання часу виконання був розроблений сервіс, який отримує від компонентів та засобів контролю наступні дані: поточний засіб, критична точка, дія, стан (початок або завершення дії), а також витрачений на дію час і розмір (кількість) заторкнутих даних. Час, який є нижчим за 1мс, округлюється до 1мс.

Для моделювання початкового завантаження були опрацьовані вибірки з 20, 100 та 500 елементів. Кожен тестовий випадок було виконано не менш 20 разів та отримано середнє значення. Моделювання оновлення кількості елементів залежить

від набору фільтрів. Кількість початково збережених елементів при оновленні дорівнює 500 елементам.

Першим є моделювання базової реалізації Redux, яка наведена у документації та є найбільш розповсюдженою. На рисунку 18 наведені результати, які були отримані при завантаженні даних різної величини.

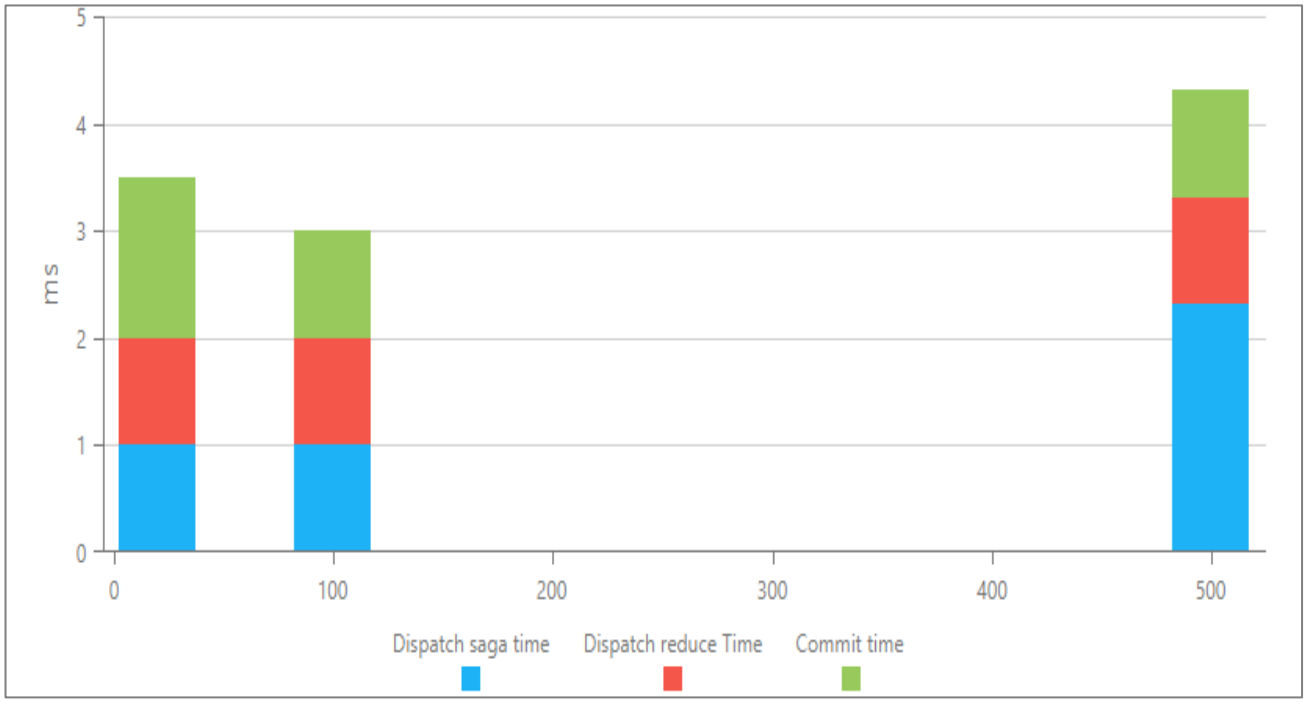

Рисунок 18 – Результати реалізації Redux №1. Завантаження даних

Як видно з рисунку (див. рис. 18), при початковому завантаженні даних для бібліотеки Redux розмір даних не має значення. Було проведено моделювання трьох варіацій розміру й не було виявлено жодної залежності часу виконання від розміру зберігаємих даних. Мінімальна затримка між виконанням тестувань дорівнювала 5 секундам та виконувалася після повного перезавантаження сторінки у браузері. Після проведення 10 тестів для 20, 100, 500 елементів, завантаження даних через глобальне сховище Redux займає у середньому 3.5мс, що є несуттєвим значенням. Максимальний час, який був витрачений на завантаження даних та сповіщення про це компонента, дорівнює 6 секундам.

Для проведення оновлення початкова кількість даних при моделюванні є максимальною й дорівнює 500 елементам.

Як і у випадку з завантаженням даних, усі тестування були виконанні з мінімальним інтервалом у 5 секунд.

На рисунку 19 наведений графік залежності кількості даних, які оновлюються, від часу, який необхідний для виконання повного циклу – від ініціалізації дії до отримання компонентом сповіщення про нові дані.

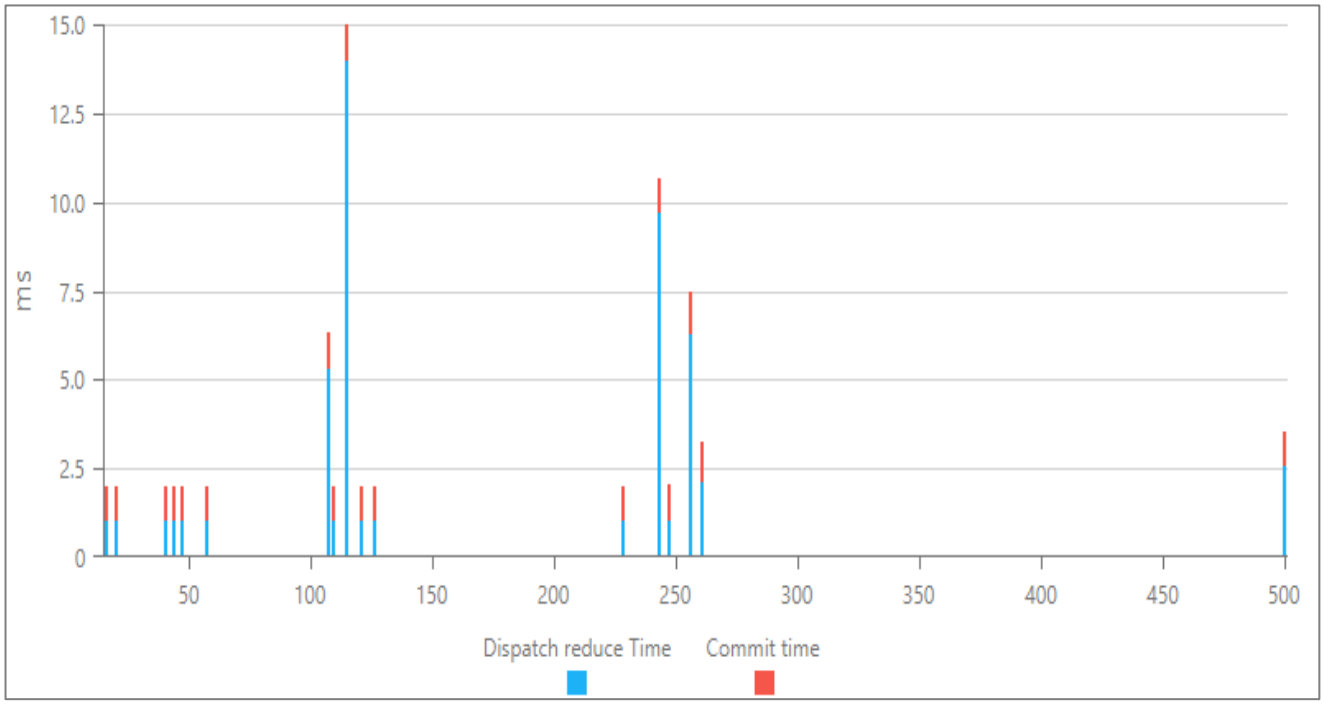

Рисунок 19 – Результати реалізації Redux №1. Оновлення даних

Як видно з рисунку (див. рис. 19), як й у випадку з початковим завантаженням даних, немає залежності між кількістю елементів та часом виконання.

Різні середні значення для «Dispatch», тобто часу ініціалізації дії від компонента до сховища Redux, обумовлені тим, що Redux вміє кешувати ініціювання однакових дій протягом деякого часу.

Це кешування працює досить нестабільно й зберігається у пам’яті недовгий час. Таким чином, при ініціалізації дії з кешу час її виконання займає не більше 1 мілісекунди. Якщо ж дія вже була видалена з кешу, її ініціація буде займати від 10 до 18мс. Різниця у часі між різними вибірками пов’язана з різним інтервалом між їх виконанням.

Другим є моделювання реалізації Redux, яка не була суттєво змінена у порівнянні з першою, але була суттєво масштабована в ширину, що означає

збільшення кількості можливих дій та кількісті функцій-редьюсерів. Така реалізація, за багатьма скаргами розробників, призводить до того, що швидкість ініціалізації суттєво зменшується, в той час, як власники бібліотеки заперечують це й наголошують, що навіть 200 функцій-редьюсерів, що є достатньою кількістю майже для будь-якого великого проекту, не мають жодного впливу на продуктивність роботи redux.

Саме кількість у 200 додаткових тестових функцій-редьюсерів і була обрана для проведення другого моделювання. Під тестовими функціями-редьюсерами маються на увазі, редьюсери, кожен з яких має відповідну дію-ідентифікатор, на ініціалізацію якої він викликається.

Результати початкового завантаження даних і їх фіксації на стороні компоненту зображені на рисунку 20.

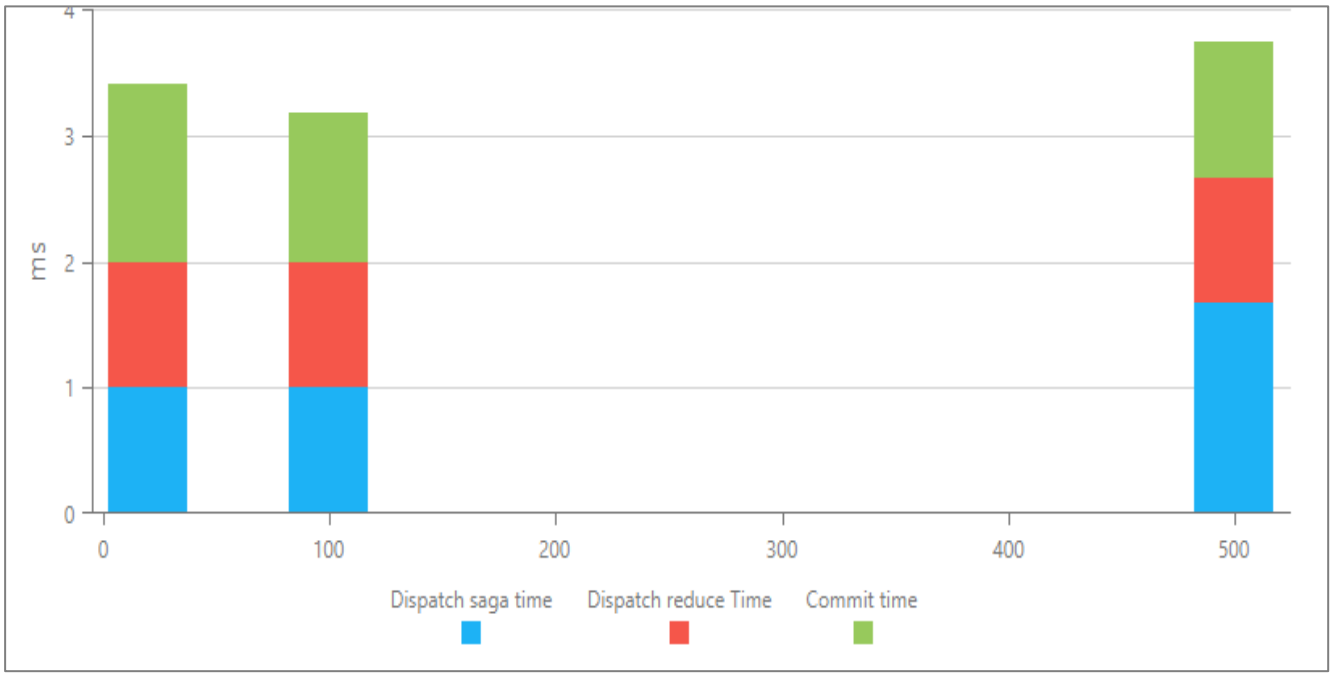

Рисунок 20 – Результати реалізації Redux №2. Завантаження даних

Як видно з рисунку (див. рис. 20), при порівнянні результатів другої реалізації з першою, додавання великої кількості функцій-редьюсерів не привнесло суттєвих змін у час виконання, який все ще дорівнює близько 3.5мс у середньому.

Для моделювання оновлення даних з додатковими редьюсерами, як і в попередньому випадку, була обрана кількість у 500 елементів, результати якого зображені на рисунку 21.

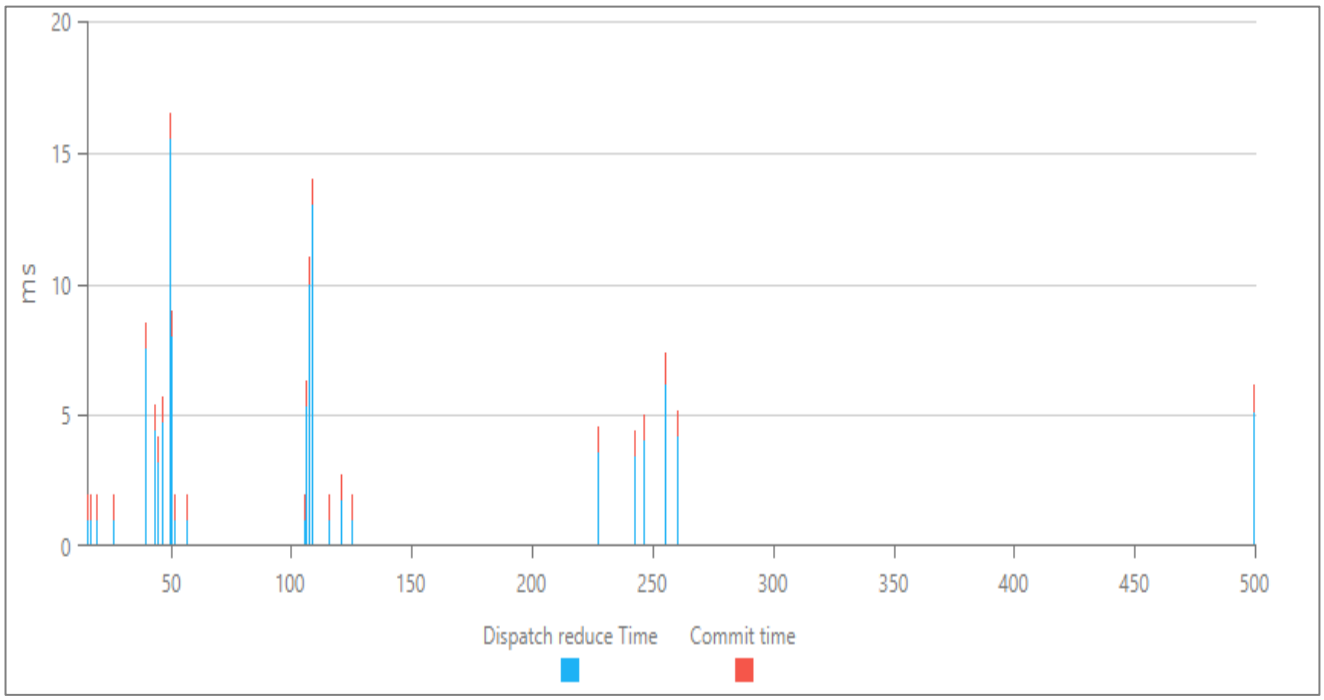

Рисунок 21 – Результати реалізації Redux №2. Оновлення даних

Як видно з вищенаведеного рисунку (див. рис. 21), особливих змін у порівнянні з оновленням даних в першій реалізації не відбулося.

Таким чином, можна сказати, що при використанні Redux ініціалізація різних дій, які вже були видалені із кешу, тобто були викликані з великим проміжком у часі, призводить до збільшення часу на їх обробку, а саме на пошук необхідного редьюсеру. Втім, у разі, коли ці функції викликаються доволі рідко, розробнику не потрібно турбуватися з приводу їх швидкодії, так як навіть у найгіршому випадку, коли швидкодія складає 18мс, це значення не є суттєвим при виклику функції, наприклад, один раз у 5 секунд. При частому оновленні й використанні закешованих редьюсерів, час ініціалізації є меншим за 1мс, що є відмінним значенням для додатків з необхідністю у високочастотних оновленнях.

Третім є моделювання реалізації глобального сховища, розробленого за допомогою бібліотеки MobX.

Основною зміною у порівнянні з Redux є ініціювання дії, яке тепер є простим викликом функції. Таким чином, можна передбачити низький час у ініціюванні дій незалежно від частоти їх виклику. Результати початкового завантаження даних наведені на рисунку 22.

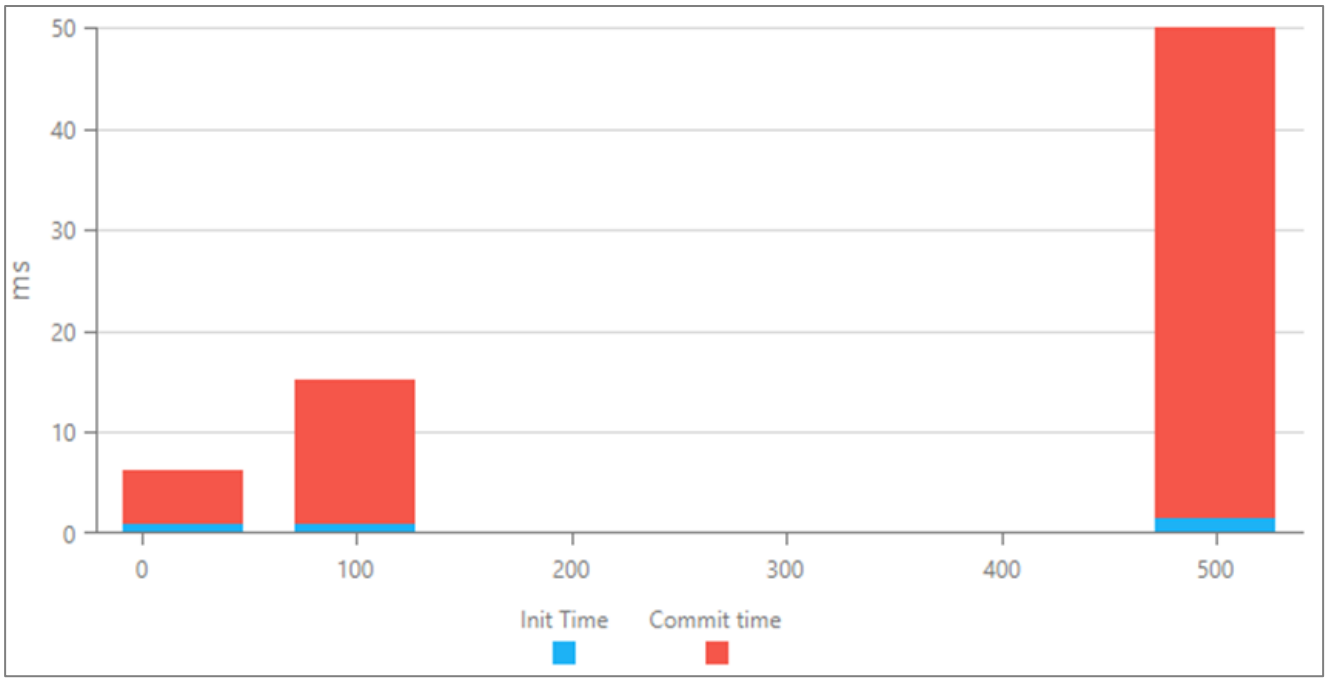

Рисунок 22 – Результати реалізації MobX №1. Завантаження даних

Як видно з рисунку (див. рис. 22), при завантаженні даних час ініціалізації залишається майже незмінним й залежить лише від навантаження на браузер. Час, що необхідний для передачі даних від сховища до компонента, значно зростає зі збільшенням кількості цих даних і досягає 50мс у середньому для 500 елементів списку, що може бути відчутною затримкою.

Слід зазначити, що у даному випадку при початковому завантаженні даних до сховища окрім самого списку даних додається їх копія для подальшого використання фільтрації, тобто оновлення даних, а також оригінальні фільтри та їх копія, якими через їх невеликий розмір, можна знехтувати. Втім, якщо в Redux при максимальних 500 елементах списку й також 500 елементах-копіях, це не викликало жодних змін у швидкості роботи, для MobX збільшення кількості даних є суттєвою проблемою.

Результати оновлення даних у глобальному сховищі на бібліотеці MobX зображено на рисунку 23.

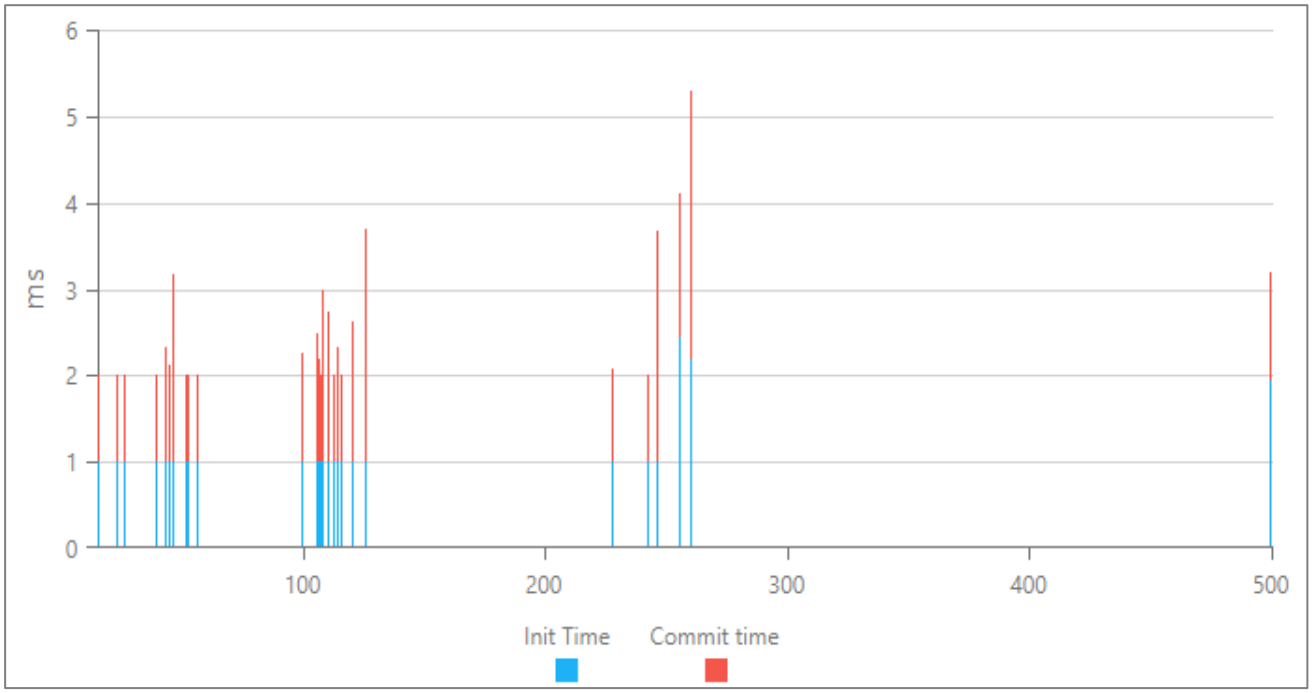

Рисунок 23 – Результати реалізації MobX №1. Оновлення даних

При оновленні даних різниця у розмірі даних відрізняється не так суттєво, бо оновлюється лише один список даних в той час, коли при початковому завантаженні оновлювалися чотири списки. Утім, якщо будуть виконуватися послідовні оновлення з великою кількістю даних, ця різниця буде відчутною в порівнянні з Redux, який може кешувати дії і не має залежності часу виконання від розміру зберігаємих даних.

Четвертим є моделювання зміненої реалізації MobX, що полягає в переході від прямого оновлення списку за допомогою @observable [18] до підрахунку значення за допомогою директиви @computed [19], який ініційований оновленням залежностей (фільтрів).

Такий перехід дозволяє зменшити кількість даних, які одночасно оновлюються у сховищі та відстежуються компонентами. Це можливо через те, що директива @computed дозволяє підраховувати новий результат, коли будь-які з залежних даних були змінені.

Таким чином, це дозволило повністю перемістити обробку оновлення оригінальних елементів до @computed, яка буде реагувати на зміну фільтрів та підраховувати новий результат.

Результати початкового завантаження даних вищезазначеної реалізації MobX наведені на рисунку 24.

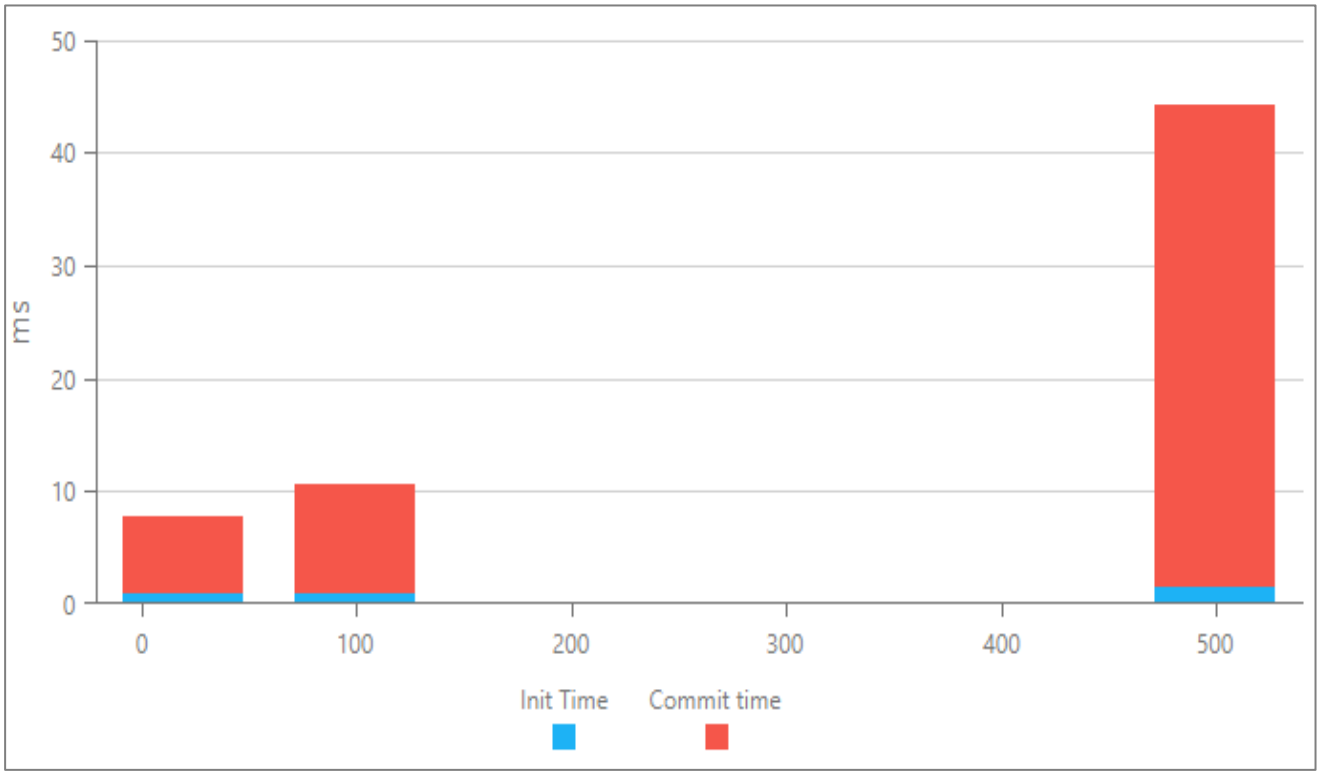

Рисунок 24 – Результати реалізації MobX №2. Завантаження даних

Як видно з рисунку (див. рис. 24), у порівнянні зі звичайною реалізацією, ми маємо приблизно 15% прискорення, починаючи зі 100 елементів. Дане прискорення пов’язано з тим, що при використанні директиви @computed, ми маємо дві порівняно невеликі операції, замість однієї операції, при якій усі необхідні дані оновлюються одразу.

І хоча прискорення у 15% не дозволяє суттєво наблизитись до швидкості обробки великої кількості даних у Redux, таке прискорення є гарним результатом, зважаючи на те, що це рішення не потребує ні додаткових інструментів, ні великих зусиль зі сторони розробника.

Оновлення даних з директивою, яке повністю покладається на використання директиви @computed, наведено на рисунку 25.

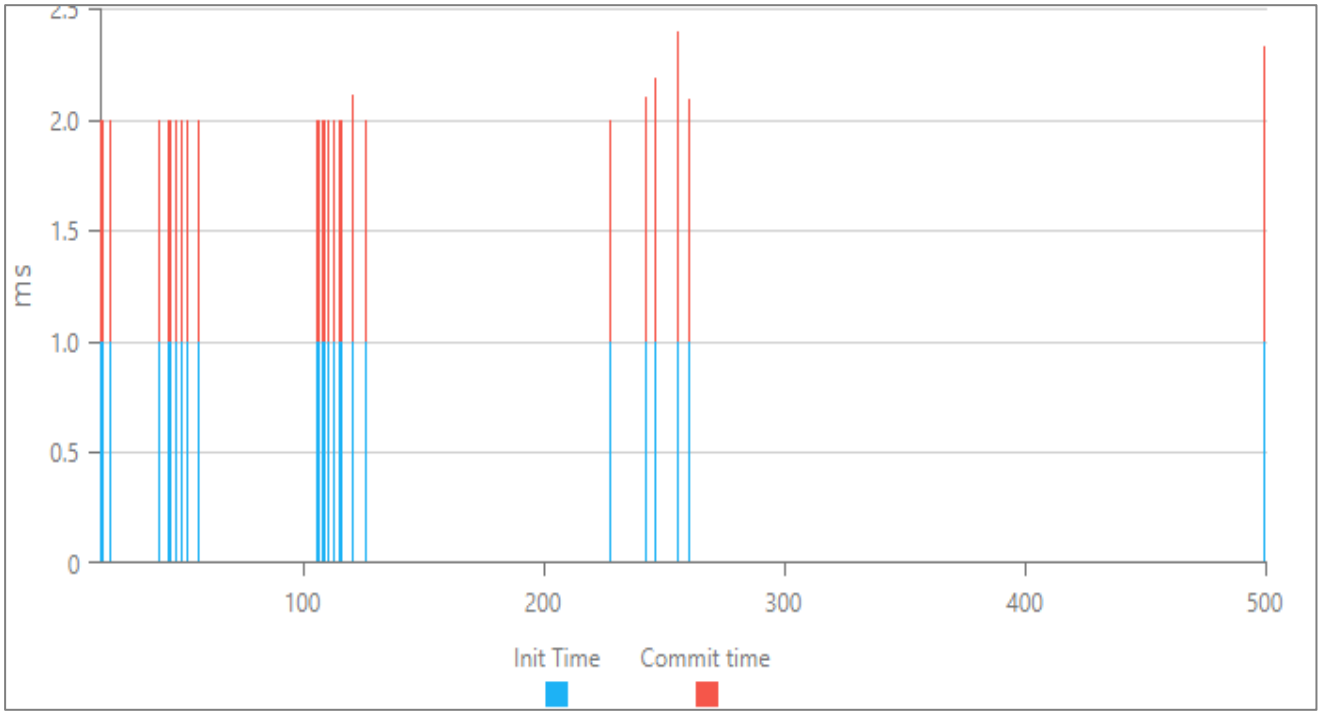

Рисунок 25 – Результати реалізації MobX №2. Оновлення даних

З результатів, наведених на рисунку (див. рис. 25), можна зробити висновок, що перехід від директиви @observable до директиви @computed для оновлень елементів покращив продуктивність, у найкращому випадку, вдвічі. Різниця при оновленні є більшою, ніж при завантаженні, бо у разі оновлення, всі операції виконуються через директиву @computed.

У більшості випадків оновлення займає менш, ніж секунду, навіть при приблизній кількості у 250 елементів, що є дуже гарним результатом, який є близьким до кешування у Redux.

Слід зазначити, що при використанні директиви @computed особливу увагу потрібно приділити взаємодії глобального сховища з компонентом, який його використовує. Наступним є моделювання вбудованого у React рішення Context API та React Hooks.

Результати моделювання початкового завантаження даних у глобальне сховище Context API й подальша їх передача до провайдеру контексту та до усіх компонентів у ієрархії наведені на рисунку 26.

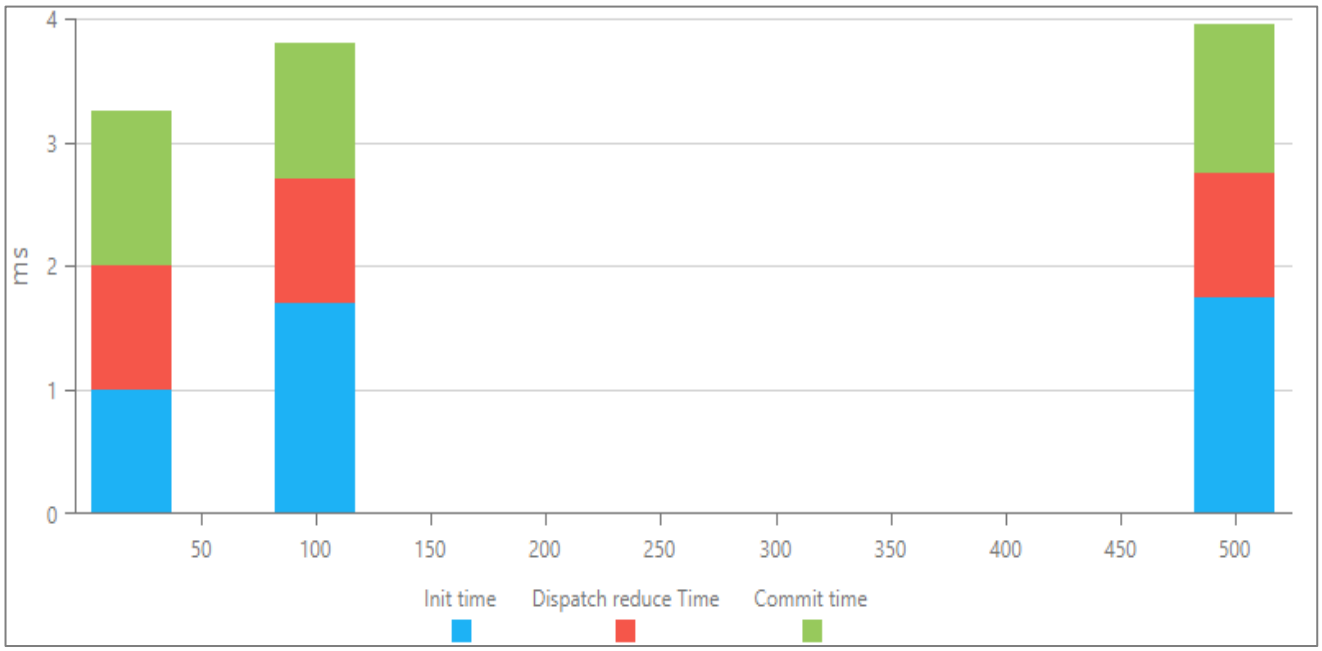

Рисунок 26 – Результати реалізації Context API. Завантаження даних

Як видно з вищенаведеного рисунку (див. рис. 26), швидкість завантаження за допомогою вбудованого у React Context API близька до Redux реалізації і також не погіршується при збільшенні розміру даних.

Це можна пояснити тим, що Context API є частиною самого React та не має жодних додаткових інструментів та розширень, які можуть сповільнювати роботу.

Додатковою перевагою Context API є можливість створювати декілька сховищ з глобальними даними та унікальними для них функціями-редьюсерами. Кожне сховище може використовувати як свої власні дії, так і дії, які були імпортовані, наприклад, з іншого сховища.

Таке рішення дозволяє уникнути проблеми бібліотеки Redux, яка має одне сховище та безліч редьюсерів у ньому, що приводить до постійної необхідності їх пошуку для обробки дії від компоненту.

Результат моделювання оновлення даних у глобальному сховищі Context API наведено на рисунку 27.

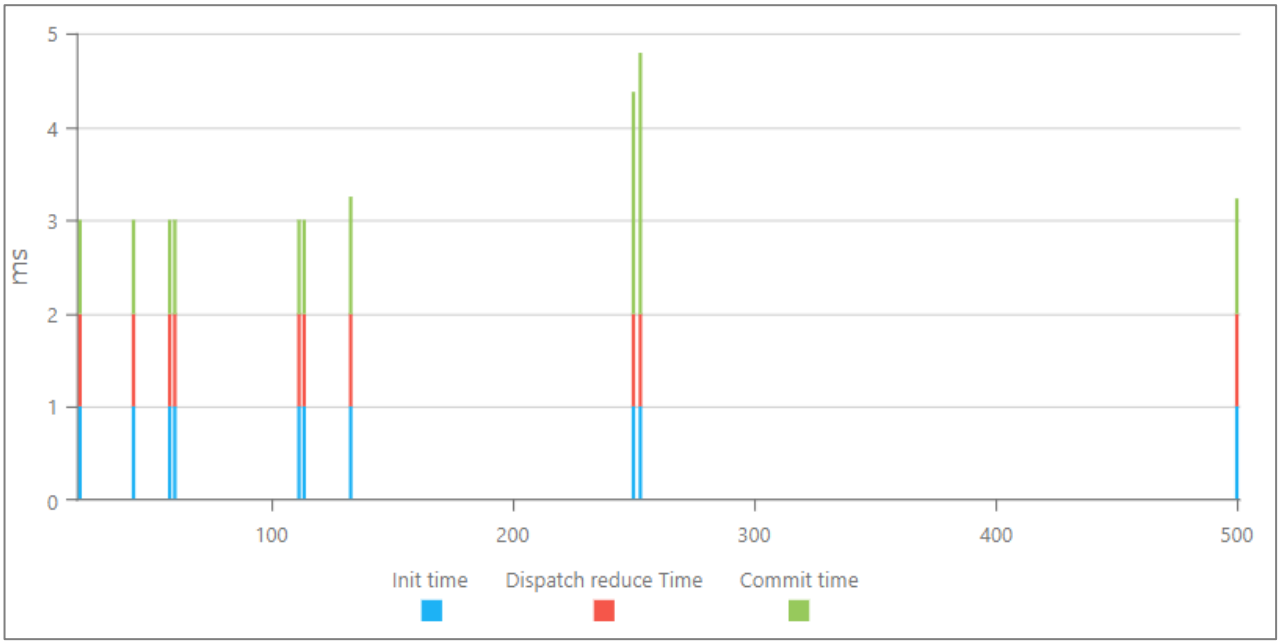

Рисунок 27 – Результати реалізації Context API. Оновлення даних

Базуючись на рисунку (див. рис. 27), можна зробити висновок, що швидкість оновлення за допомогою вбудованого у React Context API близька до MobX реалізації. Також дана реалізація позбавлена проблеми Redux, яка приводить до збільшення часу ініціалізації дії, котра вже була видалена з кешу. Залежність від розміру даних при цьому є мінімальною та може бути пояснена пропорційним збільшенням навантаження на браузер при збільшенні розміру даних.

За результатами проведених моделювань, можна зробити висновок, що за продуктивністю роботи компонентів внутрішнього устрою найкращим засобом є Context API. Бібліотека MobX у свою чергу, хоча й має хорошу продуктивність при роботі з невеликими об’ємами даних, особливо з @computed реалізацією, має суттєві проблеми при оновленні великих об’ємів інформації. Бібліотека Redux, напроти, може добре працювати з будь-якими об’ємами інформації й чудово підходить до високочастотних оновлень. Проте вона може мати проблеми при виклику різних функцій-редьюсерів, які ще не були закешовані.

4.2 Дослідження продуктивності взаємодії React компонентів додатку з глобальним сховищем

Виходячи з методики оцінювання ефективності, а саме процедури моделювання продуктивності взаємодії глобального сховища з React компонентами, наведеної в підрозділі 2.5.3, базою для проведення дослідження були взяті компоненти та реалізації глобальних сховищ, які були створенні під час дослідження продуктивності внутрішнього устрою методів та засобів контролю глобального сховища.

Таким чином, основними компонентами, які використовуються для перевірки взаємодії є компонент списку та окремий елемент списку, розроблені у попередньому дослідженні.

Зазначені компоненти були промодельовані на реалізаціях глобального сховища, які були розроблені під час попереднього дослідження (наведені в підрозділі 4.1). Була використана лише базова реалізація Redux, так як у подальших реалізаціях не була змінена взаємодія з компонентами.

До оброблюємих даних входять:

– кількість компонентів списку, які були оновлені;

– максимальний час, який був витрачений на ререндеринг компоненту списку;

– мінімальний час, який був витрачений на ререндеринг компоненту списку;

– середній час, який був витрачений на ререндеринг компоненту списку;

– середнє значення часу (далі – «повний час»), який знадобився на оновлення усієї ієрархії, починаючи від провайдеру контексту (якщо такий наявний у реалізації) та закінчуючи усіма компонентами списку.

Усі дані окрім повного часу отримуються за допомогою вбудованого у React компонента-обгортки для проведення профілювання та компілюються на сервері.

Дані про повний час, у свою чергу будуть отримуватися за допомогою браузерного розширення для React під назвою React developer tools та компілюватися вручну.

Оновлення й завантаження даних у глобальні сховища виконується так само, як й у попередньому дослідженні, опис якого наведено у розділі 4.1.

Першим є моделювання початкового завантаження даних, кількість яких була встановлена у максимальну – 500 елементів.

Результати наведені у таблиці 2.

Таблиця 2 – Тривалість рендерингу окремих компонентів та усього додатку при 500 елементах вибірки

Засіб / Характеристика	Redux	MobX №1	MobX №2	Context
Мінімальний час	1.03	1.12	1.13	1.06
Максимальний час	17.68	27.35	19.2	25.96
Середній час	1.95	2.21	2.27	2.02
Мінімальний повний час	1160.4	1243.5	1078.2	1238.7
Середній повний час	1022.71	1062.16	934.06	1015.64
Мінімальний повний час	747.5	845.9	753.3	843

Підсумовуючи результати оцінювання (див. табл 2), можна сказати, що різні реалізації глобального сховища ніяк не впливають на рендеринг окремих компонентів. Такий результат є очікуваним, бо ці компоненти не працюють зі сховищем напряму, а лише отримують дані від батьківського компоненту.

Час повного рендерингу ієрархії, у свою чергу, залежить від обраної реалізації. У той час, коли різниця у результатах між Redux, MobX №1 та Context невелика, результати реалізації MobX №2 є досить значущими.

Цей результат пов'язаний з використанням директиви @computed, яка зменшує час на завантаження даних до сховища і їх передачу до компоненту й, виходячи з репозиторію бібліотеки [20], є більш досконалим рішенням для оновлення компонентів, аніж директива @observable.

Якщо при початковому завантаженні елементів компоненти будуть рендеритися у будь-якому разі, то при їх оновленні основною задачею для покращення продуктивності додатку на React є уникнення зайвих перемалювань інтерфейсу.

При будь-якій поточній реалізації, при фільтрації елементів будуть перемальовані усі компоненти, навіть якщо їх дані не були змінені.

Ця проблема вирішується у будь-якій реалізації за допомогою вбудованої в React «мемоїзації» компонентів, яка, у свою чергу, порівнює попередні та поточні дані за допомогою неглибокого порівняння, що означає порівняння лише верхніх властивостей. Таким чином, ця реалізаціє не є цілком безпечною, бо при зміні вкладених властивостей компонент не буде оновлений.

Вищезазначена проблема була виправлена ручною реалізацією мемоїзації, що включає в себе порівняння кожної властивості, зміна якої повинна викликати ререндеринг компоненту.

Таким чином, у випадку використання компонентів, які працюють з колекціями даних, мемоїзація компонентів не залежить від обраного методу контролю глобального сховища, а цілком полягається на архітектуру організації компонентів та їх окрему реалізацію.

Наступне моделювання направлене на оцінювання продуктивності роботи додатку при частих оновленнях даних у глобальному сховищі. Для моделювання були розроблені два компоненти, які зв’язані між собою глобальним сховищем. Перший компонент є головним і відповідає за ініціювання даних та їх оновлення у глобальному сховищі. Другий компонент, у свою чергу, відстежує зміни у глобальному сховищі та відображує їх у інтерфейсі користувача. Сутність моделювання полягає у перевірці можливостей методів та засобів контролю глобального сховища з обробки високочастотних викликів по оновленню даних та їх передачі до компонентів.

Для проведення моделювання було використано компоненти, які працюють з медіа, а також саме відео. Основний компонент програє відео, інформація з якого повинна бути передана до іншого компоненту. Виходячи з того, що передача

потокового відео не має сенсу у межах поточного моделювання, дані будуть передаватися у форматі «data: URL», що дозволить передавати закодовані у виді рядка дані. Таким чином, перший компонент буде програвати відео, котре у свою чергу буде розбиватися на кадри, відмальовуватися у html канвасі та передаватися до глобального сховища. Другий компонент відстежує зміни у глобальному сховищі та відмальовує їх за допомогою html канвасу до інтерфейсу користувача.

Результати моделювання Redux сховища наведені на рисунку 28.

Рисунок 28 – Оригінальне відео та скріншот (кадр) цього відео, який передається через глобальне сховище Redux кожні 4 мілісекунди

Для відстеження продуктивності роботи глобального сховища використовується відео з секундоміром (включаючи мілісекунди), кадри якого передаються до глобального сховища кожні 4 мілісекунди. Такий інтервал був обраний, бо він є мінімально можливим для браузеру. Незважаючи на передачу даних з інтервалом 4мс, рендеринг браузера обмежений до 60 кадрів на секунду,

тобто відбувається з інтервалом у 16.66 мілісекунд. Втім, це не впливає на проведення моделювання.

Як видно з рисунку (див. рис. 28), при оновленні даних через сховище Redux, може виникати затримка. Рисунок показує максимальну затримку, яка була помічена під час моделювання, яка дорівнює 33 мілісекундам. Така затримка не є постійною, виникає час від часу й не є зростаючою. Таким чином, можна сказати, що Redux справляється зі своєю задачею навіть при необхідності великої кількості оновлень у секунду.

На рисунку 29 зображені результати моделювання для бібліотеки MobX.

Рисунок 29 – Оригінальне відео та скріншот (кадр) цього відео, який передається через глобальне сховище MobX кожні 4 мілісекунди

При моделюванні сховища MobX не було помічено жодних проблем з затримкою між оригінальним відео та кадрами, які були отримані на виході.

Цей результат є показовим та ще раз підтверджує, результати моделювання отримані у розділі 4.1, які показали, що MobX чудово працює при оновленнях з порівняно невеликим об’ємом даних.

Результати моделювання для реалізації Context API + React Hooks наведені на рисунку 30.

Рисунок 30 – Оригінальне відео та скріншот (кадр) цього відео, який передається через глобальне сховище Context API кожні 4 мілісекунди

На рисунку (див. рис. 30) видно, що Context API має проблему схожу з проблемою Redux реалізації, при якій можуть виникати невеликі короткочасні затримки. Утім, цей результат все ще є задовільним і говорить про те, що даний засіб реалізації глобального сховища не повинен викликати жодних проблем у реальних додатках, у яких така частота оновлення у більшості випадків не потрібна.

Слід зазначити, що додаток, який використовується для моделювання, є далеким від реальних додатків з точки зору розміру, логіки та додаткових бібліотек, які можуть негативно вплинути на швидкодію.

Підсумовуючи проведене моделювання високочастотного оновлення, можна зробити висновок, що будь-яка реалізація є достатньо продуктивною для виконання великої кількості оновлень щосекунди й може стабільно використовуватися. Однак, реалізація MobX показала себе найкраще через відсутність будь-якої затримки протягом усього моделювання.

4.3 Дослідження ефективності методів та засобів контролю глобального стану React додатків за критеріями ефективності розробки програмного забезпечення та критеріями його роботи

Метою проведення аналізу отриманого програмного коду розробленої системи та окремих реалізацій глобального сховища є визначення того, на скільки обрані методи задовольняють критеріям ефективності розробки програмного забезпечення, визначеним у розділі 2.2, а також критеріям ефективності роботи програмного забезпечення, визначеним у розділі 2.3, а саме – розміру бібліотеки при збірці проекту та його розміщенню на сервері.

Так як React додатки у більшості випадків виконуються на сервері у збудованому, наприклад, за допомогою Webpack, вигляді, конфігурація та оптимізація збірки проекту є важливою. Основною оптимізацією є зменшення розміру вихідних файлів.

При розробці програмного забезпечення на React та JavaScript взагалі основний розмір проекту займають бібліотеки, які імпортуються, наприклад, за допомогою менеджеру пакетів. Так як досліджуємі методи та засоби контролю глобального стану також є бібліотеками, їх розмір при збірці та можливості оптимізації потрібно дослідити.

Була проведена збірка кожного проекту, які були реалізовані під час виконання дослідження у розділах 4.1 та 4.2.

Слід зазначити, що при збірці проекту були максимально можливо відділені вихідні файли js, які підпадають під тестування, від усіх інших файлів, які не є специфічними для реалізації, але використовуються у проекті. Це дозволило отримати розмір саме виконуємого js коду проекту, а не усіх його частин. Окрім розділення файлів при збірці жодних оптимізації коду React та його компонентів проведено не було, що призвело до файлів, розмір яких набагато більший за рекомендований.

Результати були зібрані та представлені у таблиці 3.

Таблиця 3 – Розмір проектів з використанням різних реалізацій контролю глобального стану

Реалізація	Розмір
Redux №1	2.30 Кб
Redux №2	2.35 Кб
Redux моделювання високочастотного оновлення	214 Кб
MobX №1	2.32 Мб
MobX №2	2.32 Мб
MobX моделювання високочастотного оновлення	266 Кб
Context API	2.26 Мб
Context API моделювання високочастотного оновлення	267 Кб

Як видно з таблиці (див. табл. 3), використання різних реалізацій контролю глобального стану викликає незначну різницю у розмірі зібраного проекту.

Найбільш показовими є проекти для високочастотного оновлення, так як вони мають мінімум коду компонентів. Більшу частину цих проектів складають бібліотеки для глобальних сховищ та логіка для їх реалізації. Як видно з таблиці, найбільш «легкою» бібліотекою виявився Redux, при якій розмір проекту майже на 20% менший за MobX та Context API.

Найбільш цікавий результат виявився при порівнянні Context API та Redux. Незважаючи на те, що Context API не використовує жодних бібліотек, а лише

вбудовані у React елементи, це не допомогло йому зменшити розмір вихідного проекту. Таким чином, можна сказати, що реалізація Redux є більш досконалою, надаючи при цьому більше можливостей з точки зору розширення, контролю та модифікації сховищ.

Наступним етапом у аналізі отриманих реалізацій є визначення кількості необхідного коду для досягнення однакового функціоналу. Цей критерій напряму впливає на швидкість розробки програмного продукту та його підтримку (чим більший та складніший код, тим складніше його підтримувати).

Усі додатки були реалізовані за допомогою мови програмування TypeScript, яка включає в себе максимальну типізацію усіх структур даних. Для повноти експерименту, було вирішено включити типізацію, яка використовується у реалізації того чи іншого глобального сховища, до кінцевого результату.

Отримані результати було зведено у таблицю 4.

Таблиця 4 – Кількість коду, необхідного для реалізації функціоналу за допомогою обраного засобу та його реалізації

Характеристика / Засіб та його реалізація	Кількість файлів	Кількість рядків коду		
		Вцілому	З них типізація	
			Вцілому	«inline» типізація
Redux №1	8	240	82	26
Redux №2	-	-	-	-
Redux моделювання високочастотного оновлення	1	52	17	4
MobX №1	1	77	9	9
MobX №2	1	74	9	9
MobX моделювання високочастотного оновлення	1	13	2	2
Context API	4	164	53	12
Context API моделювання високочастотного оновлення	1	64	23	5

Проаналізувавши дані, наведені у таблиці (див. табл. 4), можна зробити висновок, що є велика різниця у реалізації MobX порівняно з Redux та Context API.

Говорячи про моделювання високочастотного оновлення, MobX знадобилось приблизно у 4 рази менше коду, ніж необхідно у Redux, та приблизно у 5 разів менше, ніж необхідно при Context API реалізації. Це є приголомшливою різницею, приймаючи до уваги той факт, що був розроблений однаковий, порівняно невеликий, фрагмент логіки.

При дослідженні продуктивності різних реалізацій засобів з розділу 4.1 ми маємо хоча й меншу різницю, але вона все ще є значною. Зменшення різниці можна пояснити тим, що для реалізації сховища на Redux та Context API ми повинні написати деякий шаблонний код. У подальшому, цього коду стає менше, таким чином трохи зменшується різниця у розмірі програми.

Слід зазначити, що хоча Context API та Redux використовують схожі структури і методи для реалізації сховищ, у Redux додатково задіяна бібліотека Redux-Saga для виконання асинхронних запитів до серверу. Дана бібліотека при оптимальній файловій структурі потребує створення додаткового файлу для кожної групи редьюсерів, яка наявна у сховищі.

Також з таблиці 4 видно, що кількість коду для типізації відрізняється навіть більше, ніж загальна кількість коду. Таким чином MobX, окрім меншого загального об’єму програми, має більш просту типізацію досягаючи повного покриття, як і у випадку з застосуванням Redux та Context. Також, виходячи з проведеного аналізу коду, однією з найбільших переваг MobX є можливість простого інтегрування сховища у вже існуючий додаток, особливо додаток написаний за принципами ООП. Це досягається невеликою кількістю коду MobX, сутність якого зазвичай полягає у додаванні декораторів до властивостей класу. Тобто, додаючи декоратор @observable до будь-якої властивості, компоненти додатку можуть відстежувати оновлення цієї властивості без жодних суттєвих змін.

Реалізація сховища Redux №2 не перевірялась, так як її відмінність від першої версії полягає в додаткових пустих функціях-редьюсерах, котрі не мають прямого відношення до реалізованої функціональності й служать для перевірки масштабованості системи.

Останнім етапом аналізу реалізованих глобальних сховищ є аналіз задоволення критеріям ефективності розробки програмного забезпечення, які були викладені у розділі 2.2, та базуються на проведеному у попередніх розділах моделюванні.

Як було визначено раніше, даний аналіз буде виконаний шляхом порівняння реалізованих під час моделювання засобів між собою.

Так як порівняння проводиться між трьома засобами, можливі значення знаходяться у діапазоні між 1 та 3 балами. Чим більше значення, тим краще даний засіб відповідає критерію ефективності у порівнянні з іншими. У разі, коли неможливо визначити, який засіб задовольняє критерію краще, значення для цього критерію може бути однаковим.

Таким чином, для наявного представлення даних, була сформована таблиця 5 з критеріями, які відповідають розглянутим засобам.

Таблиця 5 – Порівняння обраних засобів за критеріями ефективності розробки програмного забезпечення

Засіб / Критерій (бали)	Redux	MobX	Context API
Масштабованість (scalability)	3	2	1
Можливості налагодження (debugging)	3	2	1
Сумісність (compatibility)	2	1	3
Крива навчання (learning curve)	1	3	2
Ширина інструментарію	3	2	1
Сума	12	10	8

Обґрунтуємо значень критеріїв наведених у таблиці 5.

Щодо масштабованості, обрані у дослідженні засоби мають досить відмінні підходи. Так у Redux передбачається використання задокументованих архітектурних підходів, що, хоча й має свої недоліки, веде до одного з найкращих

архітектурних рішень, яке дуже просто розширювати й адаптувати та воно є основою для багатьох інших засобів. MobX, у свою чергу, дозволяє розробнику самому вирішувати як використовувати бібліотеку та як саме будувати архітектуру. Такий спосіб є більш складним, так як потребує об’ємних знань від розробника, буде відрізнятися у різних проектах та не буде підтверджений часом як Redux архітектура. Найгіршою є ситуація з Context API, яке водночас не передбачає підготовлених архітектурних рішень маючи при цьому більше обмежень у порівняння з MobX бібліотекою.

З точки зору можливості дебагінгу, Redux безсумнівно випереджає своїми можливостями інші засоби. Окрім архітектури, яка спрощує відладку й знижує можливість допущення помилки, Redux надає розробникам розширення для Chrome під назвою Redux-dev-tools, яке має широкі можливості, основним з яких є:

– відстеження кожної Redux-дії та даних, які були до, після та під час цієї дії;

– покрокове відстеження стану додатку з можливістю навігації по виконаним Redux-діям;

– представлення даних у трьох видах, а саме: графічний вид, вид логування, та вид об’єктів.

У порівнянні з Redux, MobX хоча й надає можливості відладки за допомогою розширення браузеру Chrome, його функціонал є досить обмеженим та включає лише представлення даних у виді логування. Context API, у свою чергу, не має жодної повноцінної утиліти для налагодження додатку, що є суттєвим недоліком.

У плані сумісності обраних засобів абсолютним лідером є Context API. Причиною цього є те, що він є вбудованою функціональністю React та його розробкою й тестуванням займаються та ж команда розробників, котра відповідає за бібліотеку React. Таким чином, кількість можливих помилок при взаємодії з React компонентами є мінімальною. Redux у своїй реалізації частково полягається на Context API, а також підтримується одним з основних розробників бібліотеки React, тому чудово працює з будь-якою версією React та більшістю інших інструментів. MobX, у свою чергу, може викликати деякі проблеми при інтеграції у React. Так, наприклад, він потребує додаткових налаштувань та конфігурацій, а

також може мати суттєві зміни у нових версіях, які потрібно додаткового адаптувати при оновленні.

Крива навчання є досить суб'єктивним критерієм, утім, при вивчення Redux у розробника може виникнути більше проблем. Основна складність полягає у необхідності розуміння усіх патернів та парадигм даного засобу. Велика кількість шаблонного коду також не йде на користь та призводить до того, що код на Redux більш складно зрозуміти. MobX є протилежним до Redux та є досить простим у розумінні, особливо для тих, хто розуміється у об'єктно-орієнтованому програмуванні. Більш за те, велика кількість роботи у MobX виконується без відому та контролю розробника, що також зменшує кількість інформації, яку йому необхідно знати. Context API є чимось середнім між Redux та MobX і хоча й має досить схожі на Redux підходи, не нав'язує їх використання. Таким чином, реалізація на Context API може бути більш простою.

Як і у випадку з дебагінгом, Redux має широкий спектр можливостей майже для кожного випадку. Даний інструментарій включає в себе різноманітні бібліотеки для керуванням формами, асинхронними запитами, логуванням та іншими частинами додатків. MobX також надає розробнику велику кількість бібліотек для різноманітних задач, але у більшості випадків вони не дозволяють створити таку ж безшовну інфраструктуру, яка є у Redux. У Context API, у свою чергу, майже повністю відсутня можливість розширення.

Підсумовуючи вищевикладені аргументи, а також беручи до уваги дані, наведені у таблиці 5, можна зробити висновок, що з точки зору задоволення обраних засобів критеріям ефективності розробки програмного забезпечення найкращим виявилася бібліотека Redux.

MobX є досить гарним рішенням, але можливі проблеми з сумісністю, а також слабкі можливості відладки коду є його недоліками.

Context API, хоча й зайняв останню позицію, є вбудованою функціональністю й не має жодних залежностей, тому може успішно використовуватися у певних випадках.

4.4 Рекомендації по використанню методів та засобів контролю глобального стану React додатків за результатами оцінювання їх ефективності

Результати моделювання та подальший аналіз використання обраних методів та засобів контролю глобального стану React додатків не дозволяє однозначно визначитися з найбільш ефективним засобом або його реалізацією. Кожний з опрацьованих методів має свої переваги й недоліки та успішно пройшов перевірку на усіх етапах моделювання. Таким чином, є сенс визначати найбільш ефективний метод у певному контексті з наданням рекомендацій по його використанню.

За результатами проведених досліджень можна сказати, що метод оснований на Redux має, безсумнівно, найбільш супроводжуєму та масштабовану, хоча й досить громіздку архітектуру, яка за даними досліджень, наведених у розділах 4.1 та 4.2, надає більш ніж достатню продуктивність. Цей факт та можливість Redux гарно працювати з будь-яким розміром даних та частотою їх оновлення робить даний метод найкращим вибором при розробці великих додатків, вимоги до яких можуть додаватися навіть після року розробки й ефективно реалізовуватися за допомогою широкого інструментарії Redux. Втім, як було зазначено, дана бібліотека вимагає від розробника дотримання деяких правил та написання великої кількості коду. Тому при розробці невеликих додатків; додатків, архітектура яких вже була визначена й не співпадає з парадигмами Redux; або додатків, швидкодія яких є першочерговою задачею, застосування бібліотеки MobX може стати кращим вибором.

Бібліотека MobX показала себе як засіб, котрий при невеликих зусиллях зі сторони розробника дає один з найкращих результатів з продуктивності, особливо при високочастотних оновленнях, а також має достатній інструментарій для побудови гарної архітектури й функціоналу для підтримки малих, середніх та, іноді, навіть досить великих додатків. Перевагою бібліотеки є також можливість легкого інтегрування глобального сховища у вже існуючі проекти, особливо проекти, розроблені за принципами ООП. Втім, при розробці проектів з

застосуванням даної бібліотеки слід уважно стежити за даними, які використовуються та оновлюються у глобальному сховищі, бо значне збільшення обсягу даних може значно знизити продуктивність. Гнучка архітектура у такому разі може як допомогти реалізувати швидкий та масштабований додаток, який може працювати з будь-якими даними, так і принести низку проблем. Таким чином, хоча MobX є легшим для вивчення, правильне його використання потребує широких знань та розуміння внутрішньої роботи даної бібліотеки [21], що у випадку використання Redux полегшується гарною шаблонною архітектурою.

Context API у свою чергу хоча й має високу продуктивність, яка перебільшує продуктивність MobX та Redux при правильному використанні, програє останнім у спектрі можливостей по розширенню, тестуванню та відладці програмного забезпечення. Як і MobX, Context API дозволяє будувати свою архітектуру, а не притримуватися вже визначеної. Незважаючи на це, розробники зазвичай обирають підхід схожий на Redux, бо навідміну від MobX, Context API не має достатньої гнучкості при використанні, а також гарно працює з архітектурою наближеною до Redux або Flux. Утім, навіть при своїх недоліках, даний засіб є вбудований у React та може бути використаним у будь-який момент, навіть у парі з MobX або Redux як допоміжний засіб у вирішенні невеликих задач з менеджменту глобальних даних.

Підсумовуючи вищенаведене, Redux є гарним вибором у випадках:

– якщо метою є створення супроводжуємої та масштабованої архітектури, яка може показати достатню продуктивність;

– якщо передбачається робота з даними великого розміру, які при цьому можуть часто оновлюватися;

– якщо метою є створення великого додатку, який потребує широкого інструментарію по відлагодженню та розширенню для задоволення вимог навіть після року розробки.

Redux є поганим вибором у випадках:

– якщо розробляється невеликий проект, при роботі над яким витрата часу на написання великої кількість шаблонного коду та дотримання строгих правил архітектури Redux може привести до непотрібних затримок у релізі;

– якщо проект вже має певну архітектуру, яка не зовсім співпадає або навіть конфліктує з досить громіздкими архітектурними рішеннями Redux;

– якщо швидкодія додатку є першочерговою задачею та при цьому не передбачається роботи з великими об'ємами даних.

Що стосується MobX, він є гарним вибором у випадках:

– якщо розроблюється невеликий додаток;

– якщо швидкість роботи додатку є першочерговою задачею, особливо при необхідності частого оновлення невеликого об'єму даних;

– якщо необхідна свобода при розробці архітектури додатку;

– якщо необхідно швидко інтегрувати глобальне сховище в існуючий проект.

MobX є поганим вибором у випадках:

– якщо розробники не мають достатньо досвіду для проектування власної архітектури, що може привести до проблем з масштабованістю та стабільністю роботи додатку;

– якщо передбачається необхідність оновлення великих об'ємів даних, що може привести до проблем з продуктивністю у MobX;

– якщо проект розраховано на розробку великою командою протягом тривалого часу, що може викликати проблеми через гнучкий та неоднозначно визначений архітектурний підхід у MobX.

Щодо використання Context API, то він є гарним вибором у випадках:

– якщо у проекті потрібен лише базовий функціонал глобальних сховищ, який не потребує встановлення додаткових бібліотек;

– якщо необхідно вирішити невеликі задачі, які не мають жорстких вимог до швидкодії або архітектури, наприклад, задачі по керуванню мовами або темами додатку;

– якщо наявного у проекті MobX або Redux недостатньо, Context API може використовуватися як допоміжний засіб.

Context API є поганим вибором у випадках:

– якщо потрібна стійка й масштабована архітектура;

– якщо потрібна гнучка архітектура;

– якщо додаток є досить складним, що може потребувати широких можливостей у його налагодженні та розширенні, інструментів для чого не міститься у Context API.

Аналізуючи отримані рекомендації, можемо зробити висновок, що неможливо однозначно визначити певний метод або засіб контролю глобального стану не враховуючи специфіки розробки проекту, кваліфікації розробників, а саме їх готовності до побудови архітектури, а також, вимог до додатку (його швидкодії та гнучкості).

ВИСНОВКИ

Атестаційна робота присвячена актуальній темі – дослідженню впливу методів та засобів контролю глобального стану компонентів на ефективність розробки, виконання та супроводу React додатків. Аналіз предметної області дозволив виявити існуючі проблеми при розробці React додатків з використанням різних методів та засобів контролю глобального стану, а також їх переваги і недоліки.

Досліджені й обґрунтовані критерії оцінювання ефективності розробки React додатків та їх роботи, а також визначені метрики для оцінювання критеріїв ефективності.

Запропонована методика оцінювання ефективності методів та засобів контролю глобального стану React додатків, розроблені процедури моделювання продуктивності роботи компонентів внутрішнього устрою методів та засобів контролю глобального стану React додатку і продуктивності взаємодії React компонентів з глобальним сховищем.

Розроблена програмна система для дослідження ефективності обраних методів та засобів контролю глобального стану.

За допомогою розробленої програмної системи проведено дослідження:

– продуктивності роботи компонентів внутрішнього устрою методів та засобів контролю глобального стану додатку;

– продуктивності взаємодії React компонентів додатку з глобальним сховищем;

– ефективності методів та засобів контролю глобального стану React додатків за критеріями ефективності розробки програмного забезпечення.

На основі аналізу результатів оцінювання ефективності розроблені рекомендації по використанню методів та засобів контролю глобального стану React додатків.

Результати оцінювання ефективності та розроблені рекомендації щодо вибору методів та засобів контролю глобального стану React додатків дозволять пришвидшити розробку React додатків, зменшивши її складність та збільшити продуктивність і стабільність роботи додатків, а також зменшити затрати на їх супровід.

Результати дослідження були оприлюднені на XXIV міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті» (тези доповіді «Критерії вибору стратегії керування глобальним станом React додатків»), а також опублікована стаття «Дослідження продуктивності методів та засобів контролю глобального стану компонентів React додатків» в міжнародному науковому електронного журналу «Наука онлайн».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Facebook. Introducing Hooks. URL: https://reactjs.org/docs/hooks-intro.html (дата звернення: 23.09.2020).

2. Adam Boduc. Flux Architecture – Packt Publishing, 2016. – p. 352.

3. Dan Abramov. React-Redux экосистема. Документація. URL: https://redux.js.org/introduction/ecosystem (дата звернення: 03.10.2020).

4. Karl Wiegers. Software Requirements (Developer Best Practices) 3rd Edition - Microsoft Press, 2013. – p. 670.

5. A. Fedosejev. React.js Essentials: A fast-paced guide to designing and building scalable and maintainable web apps with React.js – Packt Publishing, 2015. – p. 208.

6. Marc Garreau. Redux in Action – Manning Publications, 2018. – p. 312.

7. Ковалев Є., Лєсна Н. Побудова швидкодіючих клієнтських веб-застосувань за допомогою сучасних програмних засобів і технологій // Наука онлайн: міжнародний електронний науковий журнал. 2018. №12.

8. Фоменко О., Лєсна Н. Дослідження методів і моделей асинхронного оновлення даних для підвищення продуктивності web-систем // Наука онлайн: міжнародний електронний науковий журнал. 2019. №1.

9. Dan Abramov. A Predictable State Container for JS Apps. URL: https://react-redux.js.org/ (дата звернення: 03.10.2020).

10. About MobX. URL: https://mobx.js.org/README.html (дата звернення: 04.10.2020).

11. Dan Abramov. Context API. URL: https://reactjs.org/docs/context.html (дата звернення 04.10.2020).

12. Carl Vitullo. Performance Profiling a Redux App. URL: https://dev.to/vcarl/performance-profiling-a-reduxapp-1k3b (дата звернення: 05.10.2020).

13. Steve Armstrong. React + Redux Performance and the Benchmarks to Prove It. URL: https://tech.smartling.com/react-redux-performance-and-the-benchmarks-to-prove-it-79b0bc9f25a4 (дата звернення: 06.10.2020).

14. Vytautas Pranskunas. Performance Comparison of State Managenent Solutions in React. URL: https://medium.com/@vpranskunas/performance-comparison-of-state-management-solutions-in-react-8aee5ae15b3c (дата звернення: 20.10.2020).

15. Kuzochkina A., Z. Dudar, Shirokopetleva M. Analyzing and Comparison of NoSQL DBMS // International Scientific and Practical Conference «Problems of Infocommunications. Science and Technology» (PIC S&T`2018). 2018. Proceedings 8632133. pp. 560-565.

16. Redux-Saga. URL: https://redux-saga.js.org/ (дата звернення 11.10.2020).

17. Dan Abramov. Reducers. URL: https://redux.js.org/basics/reducers (дата звернення 06.10.2020).

18. Creating observable state. URL: https://mobx.js.org/observable-state.html (дата звернення 06.10.2020).

19. Deriving information with computeds. URL: https://mobx.js.org/observable-state.html (дата звернення 06.10.2020).

20. Michel Weststrate. What's difference between @computed decorator and getter function. URL: https://github.com/mobxjs/mobx/issues/161 (дата звернення 14.11.2020).

21. Georgy Glezer. MobX Tips and Pitfals. URL: https://levelup.gitconnected.com/mobx-tips-and-pitfalls-92e635108653 (дата звернення 17.11.2020).