Інформаційні технології

УДК 004.02

Пироженко Сергій Станіславович
студент
Харківського національного університету радіоелектроніки

Лєсна Наталя Совєтівна
кандидат технічних наук,
професор кафедри програмної інженерії
Харківський національний університет радіоелектроніки

ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ МЕТОДІВ ТА ЗАСОБІВ КОНТРОЛЮ ГЛОБАЛЬНОГО СТАНУ КОМПОНЕНТІВ REACT ДОДАТКІВ

Анотація. *У дослідженні була розглянута продуктивність роботи внутрішніх функцій Redux, MobX та Context API, які, на поточний час, є основними засобами контролю глобального стану компонентів у React додатках.*

Ключові слова: *стан React додатку, глобальний стан, компонент, Redux, MobX, Context API, незмінність даних, редьюсер.*

Проблема і актуальність дослідження. Актуальність досліження методів та засобів контролю глобального стану React додатків зумововлена тим, що технології сучасної веб-розробки дуже стрімко розвиваються, постійно привносячи нові підходи, бібліотеки та фремворки. Втім, на сьогоднішній день, найпопулярнішим вибором серед фронт-енд розробників

усього світу є React.js. React дозволяє розробляти веб-сайти будь-якої складності й розміру завдяки створенню незалежних компонентів із можливістю повторного використання. При зростанні кількості останніх, React ставить питання щодо контролю передачі, оновлення та розподілення між ними доступу до даних. Зазвичай, це питання має бути вирішено на стадії проектування архітектури й помилкове рішення може привести до недостатньої продуктивності додатку, неправильної реалізації критичних моментів або зайвих витрат часу. У сучасному світі продуктивність веб-сайтів є критичним моментом, особливо при використанні їх на мобільних платформах, тому важливо використовувати підходи, які є найбільш ефективними.

Огляд поточного стану об'єкта дослідження. На сьогодні, основними підходами до контролю глобального стану, що конкурують, є бібліотека Redux, бібліотека MobX, а також вбудоване у React рішення – Context API.

Бібліотека Redux основана на архітектурі Flux та є найпопулярнішою серед конкурентів і за даними менеджеру пакетів npm використовується у більше ніж половині проектів. 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 достягнуто шляхом незмінності даних у глобальиих 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 ростом, що й послужило підставою для проведення даного дослідження.

Бібліотека MobX у свою чергу надає розробнику повний контроль за створенням архітектури додатку, а також можливість змінювати об'єкти та їх частини за своїм бажанням. Це приводить до зменшення розміру коду й, у теорії, підвищення продуктивності роботи React компонентів. Через такий підхід, у останні роки дана бібліотека постійно збільшує свою аудиторію.

Context API є вбудованим рішенням, яке не передбачає конкретного підходу до вирішення проблеми з глобальним станом компонентів, а лише надає інструменти для реалізації (деякі з яких використовуються в Redux та MobX).

Більшість проаналізованих досліджень, наприклад, [1] та [2], були націлені на взаємодію засобу контролю глобального стану з компонентами React, а не на продуктивність роботи самого засобу/методу.

Об'єктом дослідження є глобальне сховище React додатків.

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

Метою даного досліджнення є аналіз продуктивності методів та засобів контролю глобального стану React додатків.

Дослідження продуктивності внутрнішнього устрою Redux, MobX та Context API засобів. За критерій продуктивності даного дослідження був взятий час у мілісекундах.

Для тестування були розроблені:

- node.js сервер з базою даних для агрегації та зберігання проміжної інформації тестів, а також для отримання тестових даних для компонентів у React клієнті;

- п'ять клієнтських додатків React для проведення тестування, у двух з яких була реалізована бібліотека Redux, ще у двох – MobX і в останньому –

Context API. Хоча була можлива реалізація усіх засобів у одному клієнському додатку, це може вплинути на точність результатів;

- клієнстський додаток для відображення й порівняння результатів тестів.

Кожен з клієнтських додатків для тестування використовує однакові дані й компоненти, у які були вбудовані різні засоби з контролю глобального стану відповідно до документації, що наведена у [3 - 5]. У кожному з додатків для тестування наявні: компонент для відображення списку, компонент для відображення окремих елементів списку та компонент з фільтрами для списку.

Тестуванню підлягають дві наступні дії:

- початкове завантаження даних з серверу з подальшим збереженням їх у глобальному сховищі та передання до компонентів. Ініціація дії виконується з головного компонента додатку після його відмальовування на сторінці;

- фільтрація даних, які знаходяться у глобальному сховищі. Ініціація дії виконується з компоненту фільтрів для списку.

Для тестування внутрішнього устрою засобів 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 [6], яка використовується для асинхронних дій; час передачі дії від Redux-Saga до Redux редьюсера [7]; час від фіксації оновлених даних до отримання їх компонентом. Для бібліотеки Redux для синхронного оновлення даних, ми не використовуємо Redux-saga, тому вимірюються лише ініціація дії від компонента до Redux редьюсера та

час фіксації дії у зворотньому 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 час і розмір (кількість) заторкнутих даних. Час, який нижчий за 1мс округлюється до 1мс.

Для тестування початкового завантаження були протестовані вибірки з 20, 100 та 500 елементів. Кожен тестовий випадок було виконано 10 разів та отримано середнє значення. Тестування оновлення кількості елементів буде залежати від набору фільтрів. Кількість початково збережених елементів при оновленні буде дорівнювати 500 елементам.

На рисунку 1 наведено тестування базової реалізації Redux.

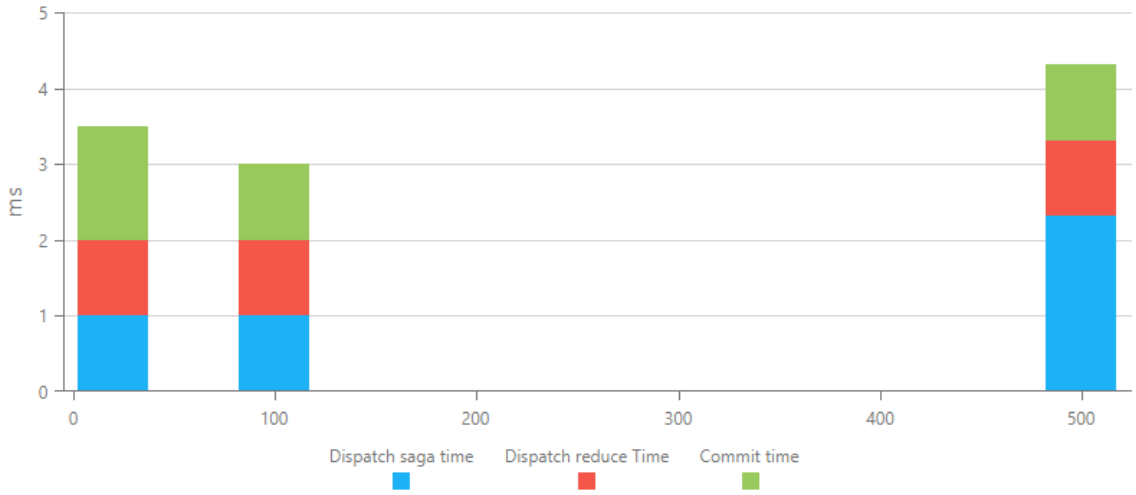

Рис. 1. Тестування реалізації Redux №1. Завантаження даних

Як видно з рисунку 1, при початковому завантаженні даних для Redux не має значення їх розмір. Після проведення 10 тестів для 20/100/500

елементів завантаження даних у Redux займає 3.5мс у середньому, що є несуттєвим значенням.

На рисунку 2 наведено результати тестування оновлення даних у базовій реалізації Redux.

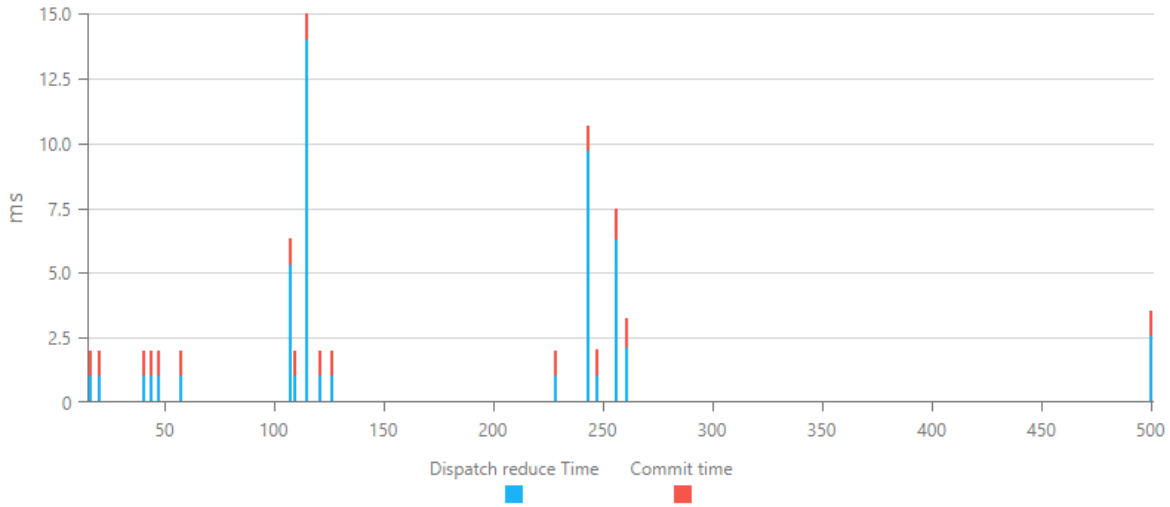

Рис. 2. Тестування реалізації Redux №1. Оновлення даних

Як видно з рисунку 2, у даному випадку залежності між кількістю елементів та часом виконання немає. Різні середні значення Dispatch обумовлені тим, що Redux вміє кешувати ініціювання однакових дій протягом деякого часу. Таким чином, якщо дія знаходиться у кешу, її виконання займає близько 1мс, у протилежному разі – від 10 до 18мс. Різниця у часі між різними вибірками пов'язана з різним інтервалом між їх виконанням.

Реалізація Redux №2 полягає у додаванні великої кількості редьюсерів, що ,за багатьма скаргами розробників, може знижувати швидкість роботи. Окрім редьюсерів для обробки завантаження та оновлення даних у реалізацію було додано 180 пустих, що, за словами авторів бібліотеки, не повинно вплинути на роботу.

Результати тестування при навантаженні та оновлені даних наведені на рисунках 3 и 4 відповідно.

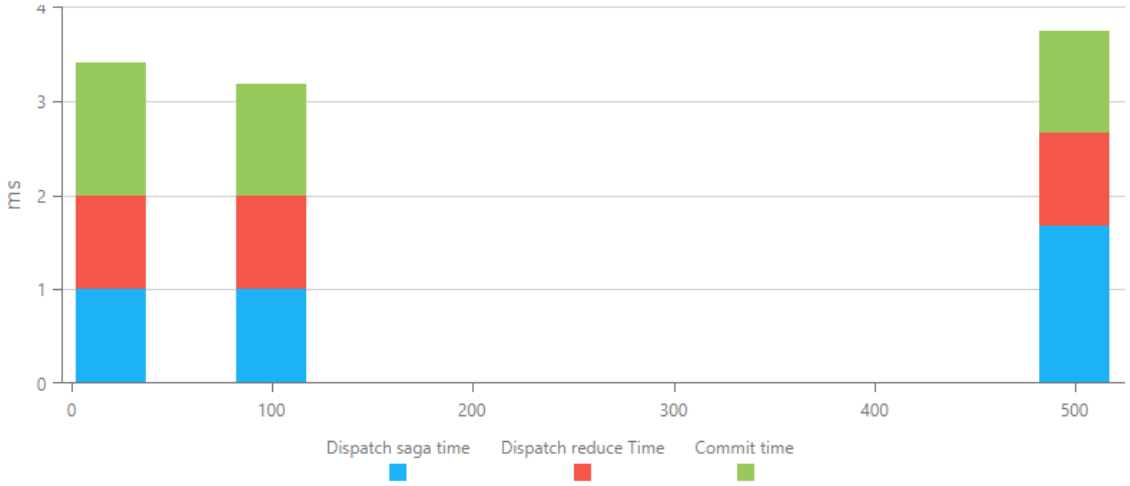

Рис. 3. Тестування реалізації Redux №2. Завантаження даних

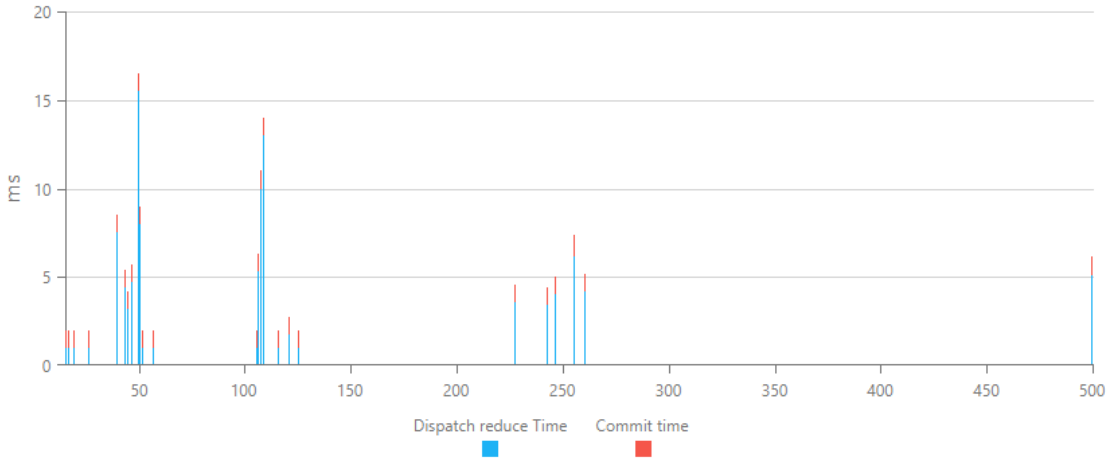

Рис. 4. Тестування реалізації Redux №2. Оновлення даних

Як видно з графіків, додавання великої кількості редьюсерів у проект не привело до суттєвих змін.

При реалізації сховища за допомогою бібліотеки MobX ініціювання дій є простим викликом функцій, тому час ініціації повинен залишатися постійним. Графіки тестування MobX при завантаженні та оновленні даних наведено на рисунках 5 та 6 відповідно.

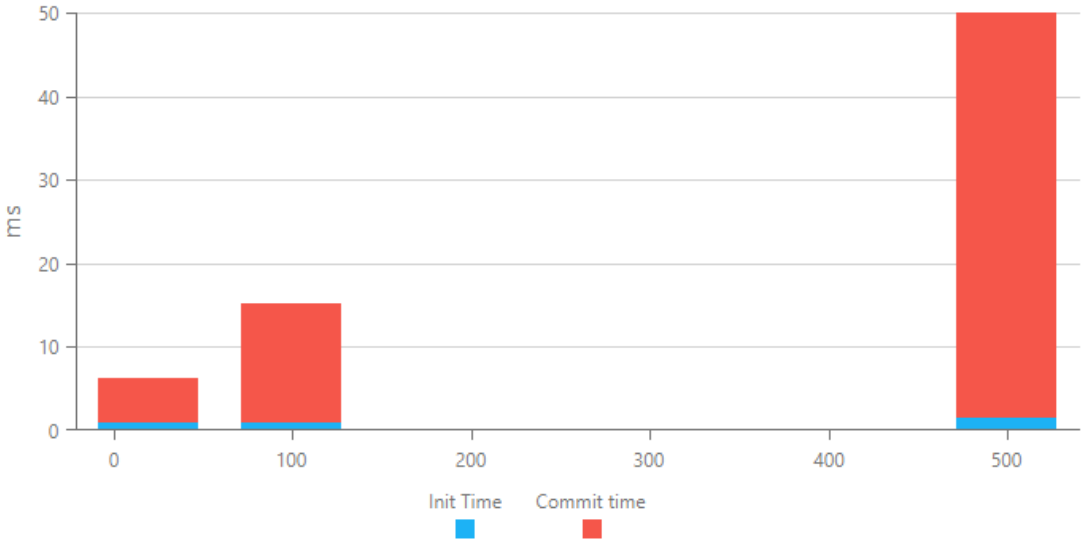

Рис. 5. Тестування реалізації MobX №1. Завантаження даних

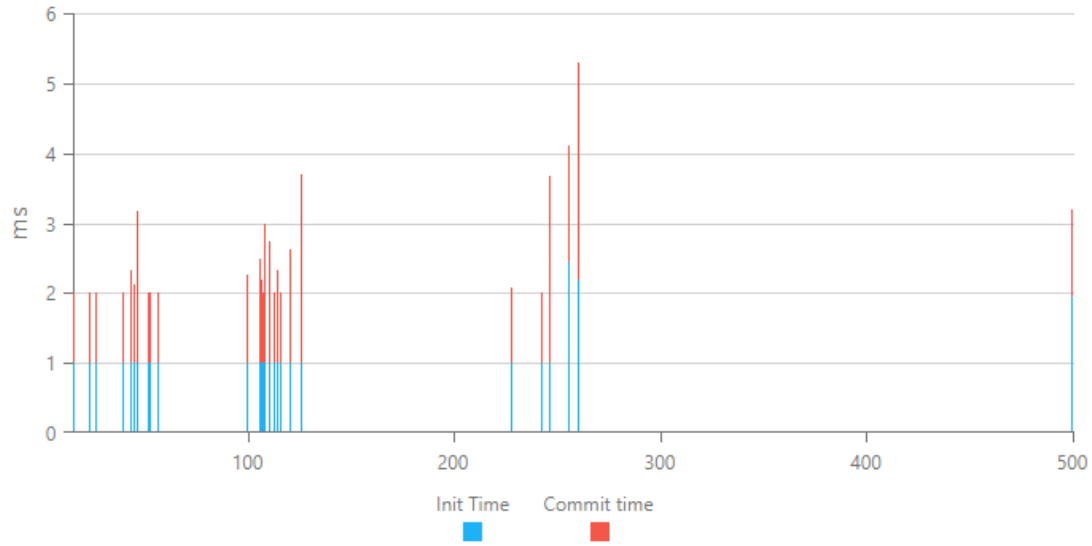

Рис. 6. Тестування реалізації MobX №1. Оновлення даних

Як видно з рисунку 5, при завантаженні даних час ініціалізації залишається незмінним, але 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.

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 (два для елементів списку і два для фільтрів). Утім, якщо будуть виконуватися послідовні оновлення з великою кількістю даних, ця різниця буде відчутною в порівнянні з Redux, який може кешувати дії і не залежить від розміру зберігаємих даних.

Реалізація MobX №2 полягає в переході від прямого оновлення списку за допомогою @observable [8] до підрахунку значення за допомогою @computed [9], який ініційований оновленням залежностей (фільтрів). Результати завантаження зображені на рисунку 7.

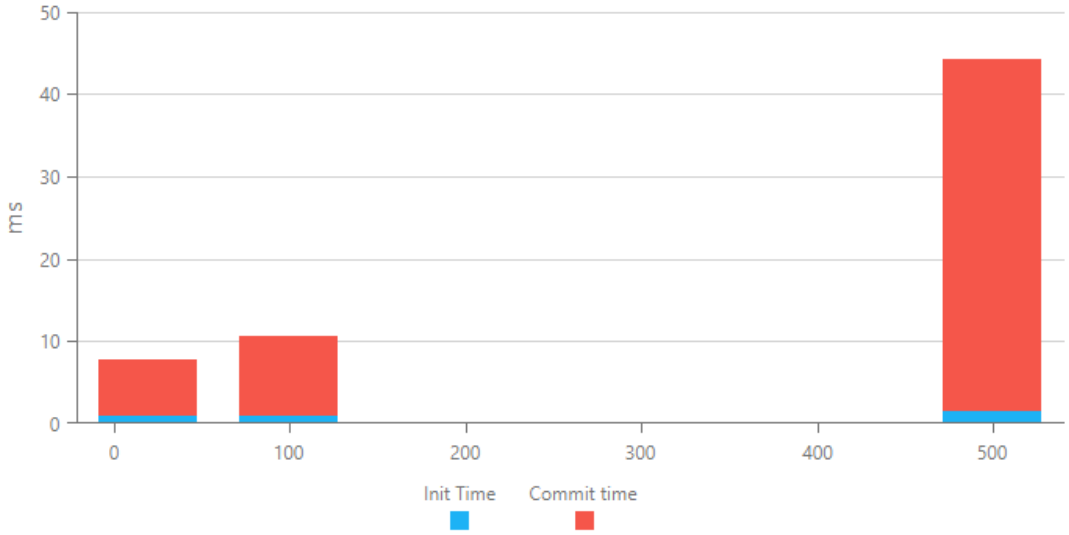

Рис. 7. Тестування реалізації MobX №2. Завантаження даних

Як видно з рисунку 7, 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.

Оновлення даних з директивою @computed зображено на рисунку 8.

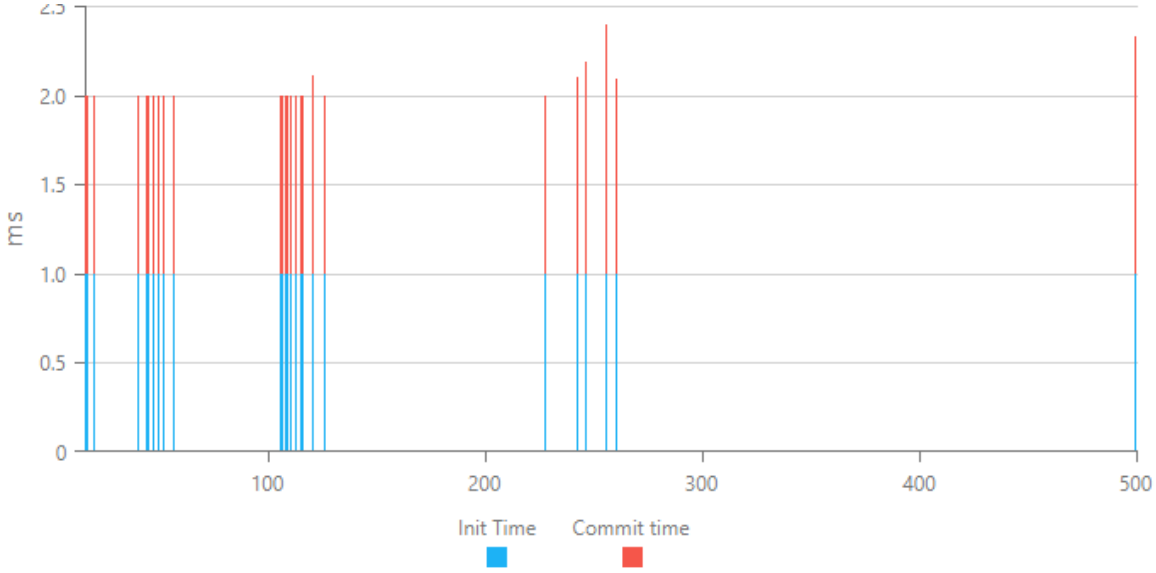

Рис. 8. Тестування реалізації MobX №2. Оновлення даних

З результатів, наведених на рисунку 8 можна зробити висновок, що перехід до @computed значень покращив продуктивність до двох разів. У більшості випадків, оновлення займає мінімальний можливий у тестуванні час – 1мс.

Слід зазначити, що при використанні директиви @computed особливу увагу потрібно приділити взаємодії з компонентом, який використовує дані.

Результати проведеного тестування при завантаженні даних в Context API наведено на рисунку 9.

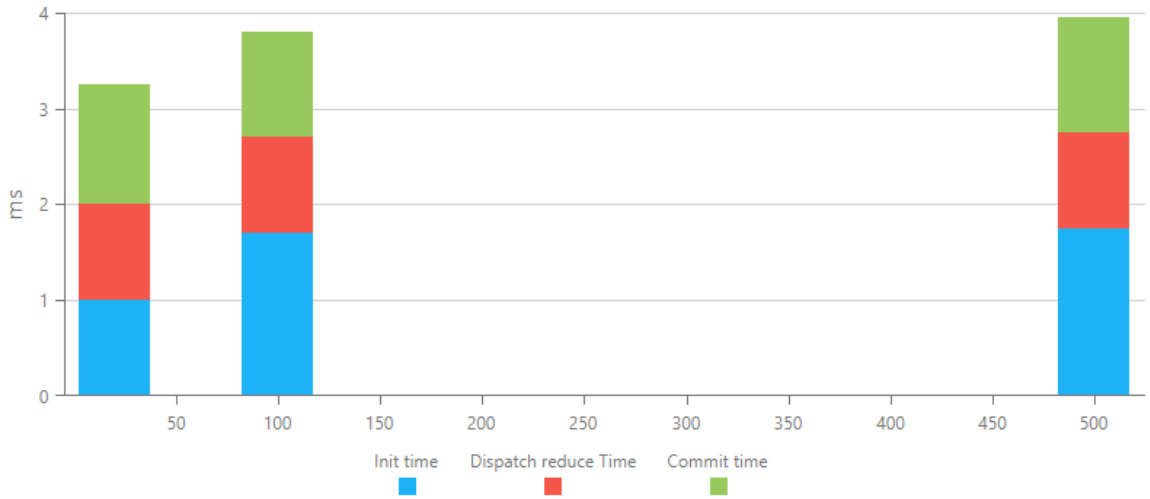

Рис. 9. Тестування реалізації Context API. Завантаження даних

Як видно з рисунку 9, поведінка Context API при завантаженні даних схожа на Redux й майже не залежить від розміру цих даних.

Результати тестування оновлення даних на Context зображено на рисунку 10.

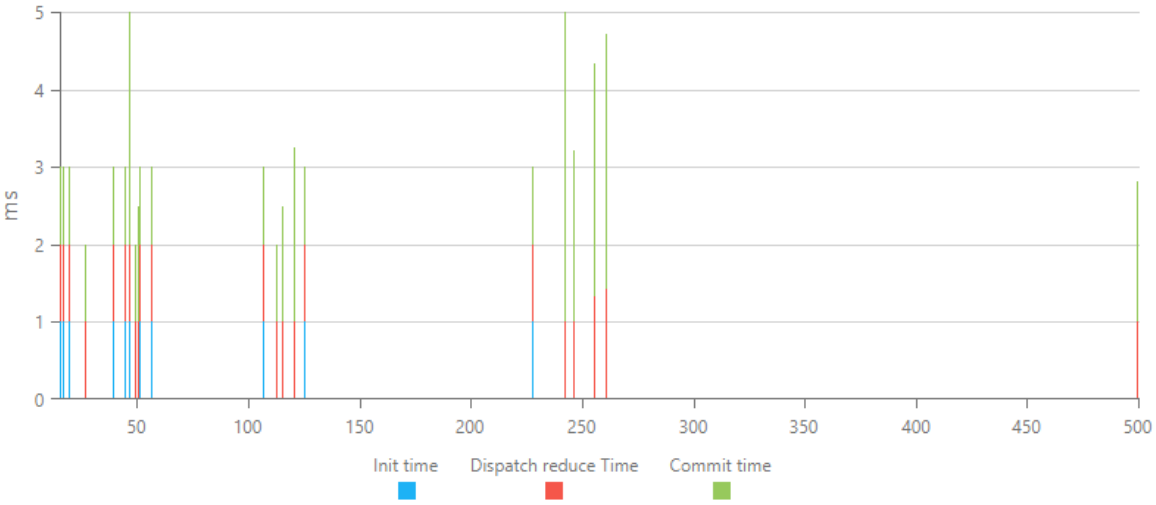

Рис. 10. Тестування реалізації Context API. Оновлення даних

Швидкість оновлення за допомогою вбудованого у React Context API близька до MobX реалізації й не погіршується при збільшенні розміру даних. Додатковою перевагою Context API є можливість створювати декілька сховищ з глобальними даними та унікальними для них функціями-

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

Висновки. За результатами проведеного тестування, можна зробити висновок, що за продуктивністю внутрішнього устрою найкращим засобом є Context API. Це можна пояснити тим, що він є частиною самого React та не має жодних додаткових інструментів та розширень, які можуть сповільнювати роботу.

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, може мати проблеми при виклику різних функцій-редьюсерів, які ще не були закешовані. Але продуктивність, у такому разі, навряд чи буде критичною.

Слід зазначити, що проведені тестування показують лише роботу внутрішнього устрою засобів і не торкаються проблем взаємодії компонентів та глобальних сховищ, що потребує додаткових досліджень, тому Context API, який за своєю продуктивністю має значну перевагу, на практиці може бути недоречним рішенням для конкретного додатку або вимагатиме додаткових налаштувань компонентів.

Література

1. Carl Vitullo. Performance Profiling a Redux App. URL: https://dev.to/vcarl/performance-profiling-a-reduxapp-1k3b (дата звернення: 05.10.2020).
2. 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).
3. Dan Abramov. React-Redux. Документація. URL: https://react-redux.js.org/ (дата звернення: 03.10.2020).
4. MobX. Документація. URL: https://mobx.js.org/README.html (дата звернення: 04.10.2020).
5. Context. Документація. URL: https://reactjs.org/docs/context.html (дата звернення 04.10.2020).
6. Redux-Saga. Документація. URL: https://redux-saga.js.org/ (дата звернення 11.10.2020)
7. Dan Abramov. Reducers. URL: https://redux.js.org/basics/reducers (дата звернення 06.10.2020).
8. MobX. Creating observable state. URL: https://mobx.js.org/observable-state.html (дата звернення 06.10.2020).
9. MobX. Deriving information with computeds. URL: https://mobx.js.org/observable-state.html (дата звернення 06.10.2020).