УДК 004.413 https://doi.org/10.35546/kntu2078-4481.2023.3.12

П. В. БУРЧАК
аспірант кафедри програмного забезпечення комп'ютерних систем
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
ORCID: 0000-0002-5939-4484

Л. М. ОЛЕЩЕНКО
кандидат технічних наук,
доцент кафедри програмного забезпечення комп'ютерних систем
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
ORCID: 0000-0001-9908-7422

МЕТОД ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ВЕБДОДАТКУ В ЕКОСИСТЕМІ ФРЕЙМВОРКУ REACT

У статті розглянуто наявні методи підвищення продуктивності вебдодатку в екосистемі фреймворку React, їх переваги та недоліки, зокрема, використання React.memo, PureComponent, shouldComponentUpdate, Lazy Loading, Code Splitting та мемоізації селекторів. Запропоновано оптимізований метод, який дозволяє скоротити час виконання програми в середньому на 17%. Основна ідея запропонованого методу полягає в розділенні стану даних вебдодатку на атомарні фрагменти. Кожна сутність має свій власний фрагмент стану, що ізолює його від інших. Ці фрагменти використовуються React Context API для передачі конфігурації сутності, включаючи дані та функції для її зміни. Context API використовується для забезпечення обробки усієї програми. Стан розділяється на атомарні фрагменти, що дозволяє реалізувати їх ізоляцію. За допомогою функції hook отримується доступ до цих фрагментів та їх зміни. Компоненти, які використовують цей метод, автоматично реагують на зміни стану і оновлюються, якщо стан змінився. Головною перевагою такого підходу є можливість використання технології замикання та передавання функції обробників стану. Запропонований метод базується на підході, який було запозичено з бібліотеки Recoil. Цей підхід є схожим на стандартні функції-хуки у фреймворку React, для якого було розроблено запропонований метод. Було використано переваги та недоліки використання популярних бібліотек для керування станом у вебдодатках. Метою модифікації було уникнення надмірних операцій, коли поточний та новий стани не відрізняються, покращення продуктивності при додаванні великої кількості елементів та забезпечення можливості використання схожих частин коду багаторазово. Особливу увагу приділено проблемі надмірних операцій, які виникають, коли всі компоненти, що підписані на зміну стану, автоматично оновлюються, навіть якщо самі значення стану не змінились. Додано можливість перевірки рівності станів перед їх оновленням, що дозволяє заощадити ресурси та зберегти продуктивність. Також реалізовано спрощено роботу з функціями зміни стану, щоб їх можна було знову використовувати без повторення коду.

Ключові слова: *методи підвищення продуктивності вебдодатків, екосистема фреймворку React, Context API, PureComponent, React.memo, shouldComponentUpdate, Redux, Recoil, Lazy Loading, Code Splitting, мемоізація селекторів, концепція керування станом.*

P. V. BURCHAK
Postgraduate Student at the Computer Systems Software Department
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
ORCID: 0000-0002-5939-4484

L. M. OLESHCHENKO
Candidate of Technical Sciences,
Associate Professor at the Computer Systems Software Department
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
ORCID: 0000-0001-9908-7422

METHOD FOR INCREASING THE PERFORMANCE OF A WEB APPLICATION IN THE REACT FRAMEWORK ECOSYSTEM

The article describes the available methods for improving the performance of a web application in the React framework ecosystem, their advantages and disadvantages, in particular, the use of React.memo, PureComponent, shouldComponentUpdate, Lazy Loading, Code Splitting and memoization of selectors. An optimized method is proposed,

which allows to reduce the program execution time by an average of 17%. The main idea of the proposed method is to divide the state into atomic fragments. Each entity has its own state fragment that isolates it from others. These fragments are used by the React Context API to pass entity configuration, including data and functions to modify it. The Context API is used to provide stateful application-wide handling. The state is divided into atomic fragments, which allows their isolation. These fragments are accessed and modified using the hook function. Components that use this method automatically respond to state changes and are updated if the state has changed. The main advantage of this approach is the possibility of using the closing technology and transferring the function of state handlers. The proposed method is based on an approach that was borrowed from the Recoil library. This approach is similar to the standard hook functions in the React framework, for which the proposed method was developed. The advantages and disadvantages of using popular state management libraries in web applications were discussed. The goal of the modification was to avoid redundant operations when the current and new states are indistinguishable, to improve performance when adding large numbers of items, and to allow similar parts of the code to be used multiple times. Special attention is paid to the problem of redundant operations, which occurs when all components subscribed to a state change are automatically updated, even if the state values themselves have not changed. Added the ability to check the equality of states before updating them, which allows to save resources and preserve performance. Also implemented simplified handling of state change functions so that they can be used again without repeating code.

Key words: *methods to improve performance of web applications, React framework ecosystem, React, Context API, PureComponent, React.memo, shouldComponentUpdate, Redux, Recoil, Lazy Loading, Code Splitting, memoization of selectors, concept of state management.*

Постановка проблеми

Підвищення продуктивності вебдодатків в екосистемі фреймворку React супроводжується різними проблемами і викликами, особливо в комплексних або великих додатках. Наприклад, React може виконувати рендеринг компонентів, навіть якщо їхні властивості не змінилися. Це може призвести до зайвих витрат часу на обчислення і малопродуктивної роботи вебдодатку. У великих додатках кількість компонентів та елементів DOM може бути дуже великою, що призводить до повільного рендерингу та завантаження сторінок. Велика кількість обчислень або маніпуляцій з DOM може блокувати основний потік браузера, що призводить до відчутного збою продуктивності. Також неоптимізований стан компонентів або надмірне використання властивостей (props) можуть призвести до зайвого рендерингу та обчислювальних операцій. Не оптимізовані мережеві запити або надмірна мережева взаємодія можуть впливати на продуктивність додатка та збільшувати час завантаження. Некоректно налаштовані компоненти, які зациклюються у безкінечних рендерах, можуть спричинити збої в роботі додатка та сповільнити його роботу. Використання неоптимізованих запитів до локального сховища даних, такого як Redux, може призвести до надмірного оновлення стану. Завантаження великих обсягів даних на клієнтську частину вебдодатку також може збільшити час завантаження сторінки і затримати відгук додатка. Зайве використання пам'яті може також спричинити зміну використання дискового простору та вплинути на продуктивність. Не оптимізовані стилі і макети також можуть призвести до повільного рендерингу і збільшити час завантаження.

Метою статті є дослідження наявних методів програмної оптимізації продуктивності вебдодатків в екосистемі фреймворку React та пропозиція модифікованого методу, який дозволить скоротити час виконання програми у порівнянні наявними аналогами.

Програмні методи підвищення продуктивності вебдодатків в екосистемі фреймворку React

Загальною метою оптимізації продуктивності вебзастосунків в React є забезпечення плавної та ефективної роботи додатка для користувачів та зменшення витрат ресурсів. Оптимізація продуктивності вебдодатків в екосистемі фреймворку React включає в себе різні аспекти розробки, щоб забезпечити швидку та ефективну роботу додатка.

Розглянемо відомі програмні методи оптимізації продуктивності вебдодатків в екосистемі фреймворку React.

Метод *React.memo* та його аналог для класових компонентів – клас *PureComponent* використовуються для уникнення зайвого перерендерингу компонентів, якщо їх властивості не змінилися. Це зменшує витрати на рендеринг та покращує продуктивність. Складні обчислення, які не змінюються під час рендерингу, можна винести за межі рендер-методу і кешувати їх результати для зменшення навантаження. Використання ключів при рендерингу списків допомагає React ідентифікувати компоненти і ефективно оновлювати їх при зміні порядку чи кількості елементів.

Приклад коду використання React.memo:

```
import React from 'react';
const MyComponent = React.memo(function MyComponent(props) {
  /* код компонента */
});
```

Оптимізація ререндерингу компонентів здійснюється завдяки використанню методів shouldComponentUpdate, memo, або Hooks useMemo і useCallback для керування повторного рендерингу компонентів.

Метод *shouldComponentUpdate* є одним із методів життєвого циклу компонента в React, що дозволяє розробнику контролювати здійснення оновлення компонента після зміни стану або пропсів. Розробник може визначити власні умови, за якими компонент повинен оновлюватися, і повернути true, якщо оновлення потрібно, або false, якщо оновлення не потрібно. Сигнатура методу *shouldComponentUpdate* виглядає так:

```
shouldComponentUpdate(nextProps, nextState)
```

nextProps – це нові властивості, які компонент отримає після оновлення;

nextState – це новий стан компонента, який буде встановлений після оновлення.

Метод *shouldComponentUpdate* повинен повертати булеве значення (true або false). Якщо він повертає true, то компонент буде оновлено. Якщо він повертає false, то компонент залишиться без змін. Основними випадками використання shouldComponentUpdate є оптимізація оновлень компонента, зменшення навантаження та заборона оновлень певних компонентів.Розробник вебдодатку може перевіряти, чи дійсно відбулися зміни в стані або пропсах, які вплинуть на рендер компонента, і виключити зайві оновлення. Це особливо корисно для великих компонентів або в разі, коли розробник знає, що зміни не вплинуть на відображення. Розробник може зменшити кількість зайвих рендерів і оновлень, що поліпшить продуктивність додатка. Наприклад, можна перевіряти, чи справді змінилися лише певні пропси перед тим, як рендерити компонент. Також можуть бути певні компоненти, які не повинні оновлюватися після змін стану або пропсів. За допомогою *shouldComponentUpdate* можна гарантувати, що ці компоненти залишаються сталими. Приклад використання *shouldComponentUpdate* для оптимізації оновлень компонента:

```
import React, { Component } from 'react';
class MyComponent extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    // Порівнюємо попередні і нові пропси або стан
    if (this.props.someProp === nextProps.someProp) {
      return false; // Не потрібно оновлювати, якщо `someProp` залишилось незмінним
    }
    return true; // Оновлювати, якщо є інші зміни
  }

  render() {
    // Рендер компонента
  }
}
```

Зауважимо, що *shouldComponentUpdate* – це метод оптимізації, який необхідно обережно використовувати, оскільки невірне визначення умови оновлення може призвести до неправильної роботи компонента. Також в сучасних версіях React можна використовувати *Hook React.memo* для досягнення подібного ефекту для функціональних компонентів.

Використання Lazy Loading і Code Splitting: розділення програмного коду на менші фрагменти та їх «ліниве завантаження» (lazy loading) допомагає зменшити час завантаження додатка і підвищити продуктивність. Використання кешування, HTTP/2, та багатопоточних запитів також дозволяє покращити швидкість завантаження даних з сервера. Використання SSR (Server-Side Rendering) або SSG (Static Site Generation) може покращити швидкість завантаження сторінок, особливо на стартових сторінках.

Використання реактивних стилів і оптимізованих CSS забезпечує мінімізацію витрат на маніпулювання DOM і використання оптимізованих CSS підходів, таких як CSS-in-JS або PostCSS.

Мемоізація селекторів (або мемоізовані селектори) – це техніка оптимізації в управлінні станом і витягуванні даних зі стану в екосистемі фреймворку React та бібліотеках керування станом, таких як Redux або Reselect. Використання бібліотеки Reselect допомагає мемоізувати результати селекторів, які використовуються для вибору частин стану. Селектори – це функції, які використовуються для вибору і обробки даних зі стану, які потім використовуються в компонентах React для рендерингу. Мемоізація селекторів полягає в збереженні результатів викликів селектора для певних вхідних даних та повторного використання цих результатів при подальших викликах селектора з тими самими вхідними даними. Це дозволяє уникнути повторних і обчислювально дорогих обчислень та зменшити зайві оновлення компонентів. Reselect надає функції *createSelector*, яка автоматично мемоізує результати селекторів і використовує їх для оптимізації вибірки даних зі стану.

Приклад використання мемоізованого селектора за допомогою Reselect:

```
import { createSelector } from 'reselect';
const selectData = (state) => state.data;
const memoizedSelector = createSelector(
  [selectData],
  (data) => {
    // Обчислення результату на основі вибраних даних
    return /* результат */;
  }
);
```

Хоча мемоізація селекторів має свої недоліки і обмеження, вона залишається корисною технікою для оптимізації продуктивності додатків React зі складним станом і великою кількістю даних.

Оптимізація стану і Redux: якщо для розроблення програмного забезпечення використовується Redux, потрібно використовувати такі бібліотеки, як Reselect і Redux Toolkit, які суттєво підвищують продуктивність роботи вебдодатку.

Використання Web Workers забезпечує виконання обчислень в окремому потоці, що не блокує основний потік браузера і покращує реактивність додатка.

Таблиця 1

Порівняння поширених програмних методів підвищення продуктивності вебдодатків в екосистемі фреймворку React

Програмний метод	*Переваги методу*	*Недоліки методу*
Використання React. memo і PureComponent	1. Не потрібно додаткового коду для визначення умови оновлення компонентів. 2. Зменшення навантаження, уникнення зайвих оновлень компонентів під час їхнього рендерингу, що може покращити продуктивність.	1.Оптимізація не завжди потрібна: не всі компоненти мають навантаження, яке варто оптимізувати. 2. Іноді можуть залишитися сценарії, де відбуваються зайві оновлення.
Використання shouldComponentUpdate	1. Можливість точного керування оновленнями компонентів. 2. Можливість оптимізувати дрібні оновлення.	1. Вимагає більше коду та явної реалізації методу. shouldComponentUpdate. 2. Може стати складним при роботі з великими компонентами.
Використання більш дрібних компонентів	1. Зберігає код більш структурованим і підтримуваним. 2. Може сприяти реюзабельності компонентів.	Додаткова складність управління багатьма компонентами.
Використання Redux або MobX	1. Централізований стан, який забезпечує одноразове джерело true для стану додатка. 2. Прозорість і відстежуваність змін стану. 3. Зручне керування великими додатками.	1. Додатковий шар складності, який вимагає додаткового коду для визначення дій та редюсерів. 2. Може бути зайвим для невеликих додатків.
Мемоізація селекторів	1. Покращення продуктивності завдяки уникненню повторних обчислень селекторів при тих самих вхідних даних. 2. Зменшення навантаження: завдяки використанню мемоізованих селекторів зменшується кількість зайвих рендерів компонентів, оскільки результати селекторів залишаються сталими, якщо вхідні дані не змінилися. 3. Збереження стабільності: завдяки мемоізації можна бути впевненим, що результати селекторів завжди залишаються незмінними для одних і тих самих даних. 4. Зменшення витрат ресурсів: заощадження ресурсів обчислення, оскільки мемоізовані селектори не потребують зайвих обчислень при однакових вхідних даних.	1.Складність реалізації. 2. Додаткова пам'ять. 3. Можливість появи помилок: неправильне використання мемоізації селекторів може призвести до неправильної роботи додатка. Невірно визначена умова мемоізації може призвести до неправильної кешування результатів. 4. Додатковий код: впровадження мемоізації вимагає додаткового коду для визначення і кешування результатів селекторів, що може робити код складнішим і менш зрозумілим. 5. Обмеженість використання: мемоізація селекторів корисна в основному для оптимізації виборки даних зі стану. Не допомагає уникати інших видів оновлень компонентів, таких як оновлення через зміну пропсів.

Керування локальним станом даних вебдодатку в екосистемі фреймворку React є важливим аспектом розробки для досягнення збереження стану вебдодатка та оновлення інтерфейсу користувача. Керування станом дозволяє зберігати та оновлювати дані, які використовуються в вебдодатку. Це може бути інформація, введена користувачем, стан компонентів тощо. Зміна стану вебдодатка спричиняє оновлення інтерфейсу користувача. Вебдодаток може реагувати на дії користувача, події в системі або зміни в мережі шляхом оновлення відображення на вебсторінці.

React дозволяє створювати реактивні додатки, де зміна стану автоматично спричиняє оновлення відображення. Це полегшує розробку додатків, які реагують на зміни в реальному часі. Керування станом допомагає уникнути розповсюдження одних і тих же даних по всьому додатку. Розробник може створити одну централізовану точку керування станом і забезпечити її єдиним джерелом truth для даних. Керування станом дозволяє розробнику зберігати стан додатка між сеансами роботи користувача. Це корисно для збереження налаштувань, даних авторизації та іншої інформації.

Централізоване керування станом робить код більш підтримуваним і зручним у тестуванні. Розробник може окремо тестувати редюсери, дії та компоненти, які використовують стан. Керування станом в React також допомагає розробнику впроваджувати інші архітектурні патерни, такі як Flux, Redux, MobX, для більшої організації і керування станом. У React та екосистемі його фреймворків, таких як Redux або MobX, керування локальним станом даних зазвичай відбувається через використання станових змінних (state), редюсерів (reducers), дій (actions) та властивостей (props).

Розглянемо загальну концепцію керування станом в React.

Станові змінні (State) визначаються у компонентах за допомогою методу useState (для функціональних компонентів) або через властивість state у класових компонентах. Наприклад:

```
// Функціональний компонент з використанням useState
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);
  };
  return (
    <div>
      <p>Лічильник: {count}</p>
      <button onClick={increment}>Збільшити</button>
    </div>
  );
}
```

У класових компонентах:

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
  }
  increment() {
    this.setState({ count: this.state.count + 1 });
  }
  render() {
    return (
      <div>
        <p>Лічильник: {this.state.count}</p>
        <button onClick={() => this.increment()}>Збільшити</button>
      </div>
    );
  }
}
```

Дії (Actions) – це об'єкти, які вказують, яким чином потрібно змінити стан. У Redux, прикладі фреймворку для керування станом, це може виглядати так:

```
// Дія (action) для збільшення значення лічильника
const incrementAction = {
  type: 'INCREMENT',
};
```

Редюсери (Reducers) визначають, яким чином стан додатка має змінитися на основі дій. У Redux це може виглядати так:

```
// Редюсер для збільшення значення лічильника
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
};
```

Store (Сховище) – об'єкт, у якому зберігається увесь стан вебдодатка. Цей об'єкт містить стан, редюсери і може виглядати так:

```
import { createStore } from 'redux';
const store = createStore(counterReducer);
```

Для передачі даних між батьківськими і дочірніми компонентами використовуються *властивості (props)*. Наприклад:

```
// Передача стану через властивість
<ChildComponent count={count} />
```

Це загальна концепція керування станом в React. Конкретний метод та підхід може варіюватися в залежності від фреймворку або бібліотеки для керування станом (наприклад, Redux, MobX, тощо), які використовуються, а також від потреб та функціональності конкретного вебдодатка.

Аналіз останніх досліджень та публікацій

Метою роботи [1] є аналіз популярної бібліотеки керування станом Redux. У цій роботі представлено метод React Hooks, який продемонстрував деякі покращення продуктивності вебдодатків. Дані результати дослідження були виміряні в інструменті для вимірювання продуктивності Google Chrome Dev Tools. У роботі [2] продуктивність бібліотеки Redux вимірювалася та порівнювалася з методом React Context API. Незважаючи на те, що бібліотека Redux проста у використанні, вона демонструє значні проблеми з продуктивністю під час роботи з об’єктами великих даних. Також було детально проаналізовано підхід хуків, що підтверджує його масштабованість. Дослідження [3-8] містять огляд бібліотек та фреймворків для розробки сучасних вебдодатків. Для порівняння розглянутих бібліотек JavaScript було обрано декілька базових метрик – час виконання методу управління станом для одиниці роботи та придатність для використання розробником. Для порівняння результатів продуктивності вебдодатків було використано інструменти Google Chrome Lighthouse та Sonarqube. Загалом у розглянутих публікаціях не описано методи оптимізації часу доступу до даних вебдодатку та їх змін, що впливає на продуктивність вебдодатку, тому дослідження оптимізації управління даними локального стану вебдодатку є актуальним завданням.

Запропонований метод підвищення продуктивності вебдодатку в екосистемі фреймворку React

Основна ідея запропонованого методу полягає в розділенні стану на атомарні фрагменти. Кожна сутність має свій власний фрагмент стану, що ізолює його від інших. Ці фрагменти використовуються Context API для передачі даних сутності, включаючи функції для їх зміни (рис. 1).

Функція *hook* надає доступ до цього фрагмента, забезпечує взаємодію з даними та внесення змін. Програмний інтерфейс може бути різним, оскільки конфігурація фрагмента стану задається під час його створення.

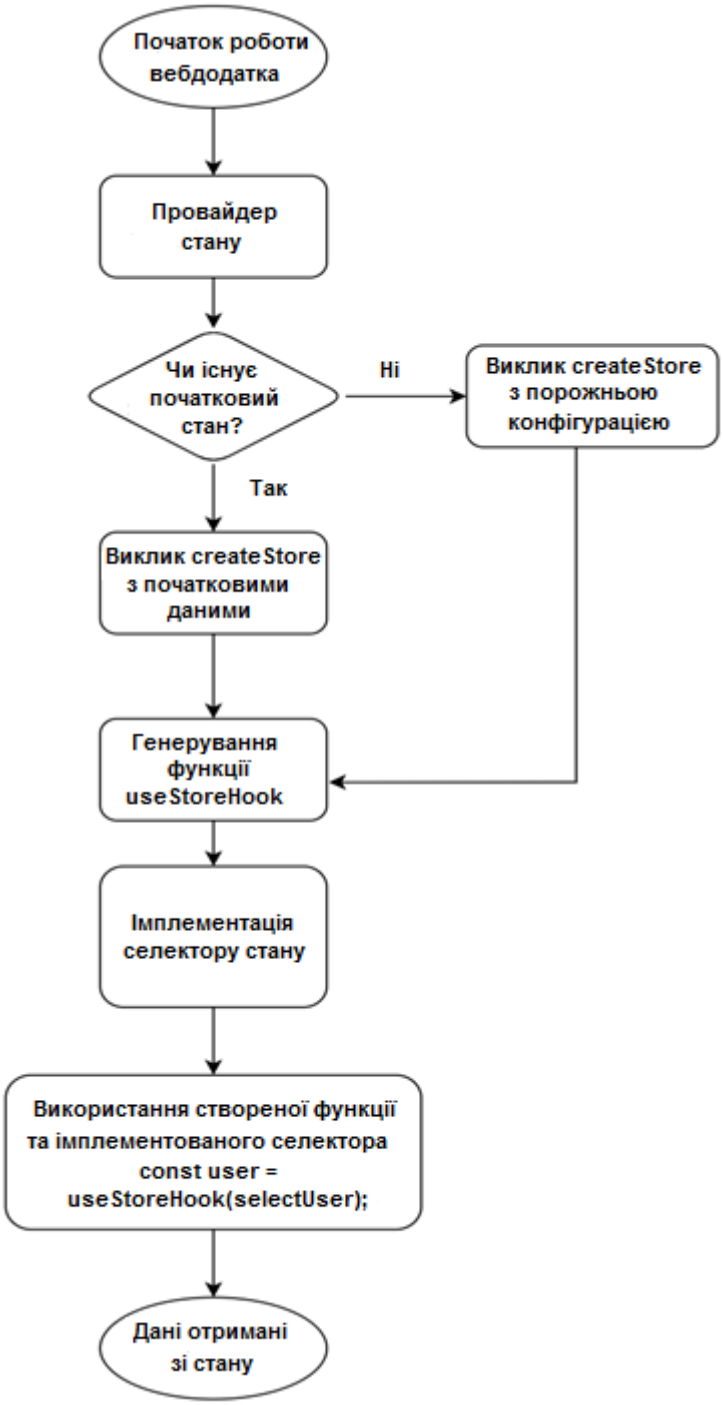

Рис. 1. Схема алгоритму запропонованого методу

Context API використовується для забезпечення роботи зі станом у всьому вебдодатку. Стан розділяється на атомарні фрагменти, що дозволяє здійснювати їх ізоляцію. За допомогою функції hook ми отримуємо доступ до цих фрагментів та можемо змінювати їх. Компоненти, які використовують цей метод, автоматично реагують на зміни стану і оновлюються, якщо стан дійсно змінився. Ми також додали оптимізації, щоб уникнути зайвих операцій, якщо стан не змінився, і спростили роботу з функціями зміни стану, що дозволяє їх багаторазове використання без дублювання коду.

Для створення фрагмента стану використовується функція create:

```
interface create = (storeConfig) => useStoreHook;
```

Синтаксис (storeConfig) => useStoreHook означає, що функція використовує параметр типу storeConfig та повертає значення useStoreHook.

Інтерфейс функцій створення стану:

```
interface create = (storeConfig) => useStoreHook;
interface storeConfig = (
set: (any) => void,
get?: (void) => any)
=> (stateConfig :Record<string, any>);
interface useStoreHook =
(selector: (any) => object) => (stateConfig: Record<string, any>);
```

Тип any – будь-який тип, Record<string, any> – об’єкт з парами ключ-значення, рядок та будь-який тип відповідно. Методи *set* та *get* можуть бути викликані при конфігурації для наступного використання значення полів та для підрахунку певних значень у функціях. Метод create виконує дві функції: задає початкове значення стану та описує функції, що виконують операції над даними стану. Згенеровану методом функцію-хук *useStore* використано як спосіб отримання даних стану безпосередньо для застосування у користувацькому інтерфейсі. Головною перевагою такого підходу є можливість використання технології замикання та передавання функції обробників стану. Запропонований метод загалом базується на підході, який було запозичено з бібліотеки Recoil. Цей підхід є схожим на стандартні функції-хуки у фреймворку React, для якого було розроблено запропонований метод. Ми вдосконалили його, порівнюючи з іншими популярними бібліотеками для керування станом у вебзастосунках. Метою було уникнення надмірних операцій, коли поточний та новий стани не відрізняються, покращити продуктивність при додаванні великої кількості елементів та забезпечити можливість використання схожих частин коду багаторазово. Під час розробки оптимізованого методу було також враховано недоліки інших підходів. Особливу увагу приділено проблемі надмірних операцій, які виникають, коли всі компоненти, що підписані на зміну стану, автоматично оновлюються, навіть якщо самі значення стану не змінились. Було додано можливість перевірки рівності станів перед їх оновленням, що дозволяє заощадити ресурси та зберегти продуктивність. Також було спрощено роботу з функціями зміни стану, щоб їх можна було використовувати знову без повторення коду.

Результати дослідження

Для дослідження та порівняння програмних методів були обрані бібліотеки Redux, MobXState-Tree, Recoil. Аналіз методів виконано за допомогою утиліти SonarQube. Для оцінки результів дослідження використано браузер Google Chrome та утиліта DevTools. Для розробки клієнтської частини програмного забезпечення та проведення дослідження була обрана мова JavaScript для створення скриптів вебсторінок, яка надає можливість на стороні клієнта взаємодіяти з користувачем, керувати браузером, а також асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд вебсторінки. Для спрощення процесу налаштування та розробки вебдодатку була обрана бібліотека React через її сумісність з обраними бібліотеками та наявність React Dev Tools для вимірювання часу програмних методів та оцінки якості їх роботи. Для оцінки швидкодії програмних методів використовувався веббраузер Google Chrome для вимірювання часу роботи вебдодатку за певний період часу та перегляду діаграм розподілу часу роботи додатка для конкретних дій, таких як рендеринг, системні операції та програмний код виконання. Враховуючи описані вимоги, у програмному забезпеченні було створено умови для того, щоб кожен із обраних для дослідження методів використовувався в бібліотеках і демонструвалась їх робота на основі певних даних. Основною вимогою до програмного забезпечення, що надає виведення в графічний інтерфейс користувача результатів роботи різних методів даного дослідження була підтримка Google Chrome 95+, React 17+ та Node 16+. Для порівняння точності методи використовувались в однакових умовах, були оброблені однакові дані та відповідно до їх кількості, над даними також були виконані однакові дії: додавання сегментів даних, редагування, видалення.

Щоб оцінити час роботи методів, які працюють над локальним станом даних вебдодатку, було використано великий обсяг даних (до 100 000 елементів), дані відповідають структурі масиву з великою кількістю елементів. Апаратне забезпечення, яке використовувалось для проведенні дослідження: MacBook Pro, процесор 2 GHz Quad-Core Intel Core i5, Intel Iris Plus Graphics 1536 MB, 16 GB пам'яті 3733 MHz LPDDR4X. Операційна система – macOS Ventura 13.0.1. Програмне забезпечення: Google Chrome 108, React v18.0.1, Node v16.13.1. На прикладі використання бібліотеки Redux збільшення часу виконання непропорційно зростає зі збільшенням кількості елементів.

При збільшенні кількості елементів до 100 000 результат бібліотеки MobX був отриманий майже за 250 секунд, що в 3 рази повільніше аналогів. Бібліотеки Redux і Recoil показують приблизно однакові результати за ~80 секунд, що потребує покращення. Аналізуючи дані, отримані під час тестування швидкості додавання елементів, можна зробити висновок, що бібліотеки Redux і Recoil показують кращі результати при додаванні

більшої кількості елементів. Було порівняно продуктивність різних програмних методів та продуктивність запропонованого оптимізованого методу. Запропонована оптимізація, яка мінімізує залежність від сторонніх бібліотек, дає кращі результати під час додавання елементів до масиву порівняно з іншими методами. Однак при великій кількості елементів покращення є незначним. При скиданні до порожнього масиву запропонований метод займає 3844 мс для простих даних і 3979 мс для складних даних. У всіх сценаріях тестування швидкості запропонований метод перевершує альтернативи, особливо під час додавання або редагування елементів масиву. Незалежно від розміру масиву, запропонований метод незмінно перевершує конкурентів, маючи перевагу на 12% для 1000 елементів, 16% для 5000 і 18% для 10 000 елементів. Це демонструє стабільність і масштабованість запропонованого методу. Порівнюючи популярні бібліотеки, було виявлено, що MobX чудово справляється з невеликими наборами даних, але має труднощі під час більшого навантаження. Навпаки, Redux зберігає стабільну продуктивність у різних тестах.

Висновки та подальша робота

У статті розглянуто наявні методи підвищення продуктивності вебдодатку в екосистемі фреймворку React, їх переваги та недоліки, зокрема, використання React.memo, shouldComponentUpdate, Lazy Loading, Code Splitting та мемоізації селекторів. Запропоновано модифікований метод керування локальним станом даних вебдодатків для створення додатків в екосистемі фреймворку React, що дозволяє скоротити час виконання програми в середньому на 17%, порівнюючи кожен зі сценаріїв тестування методу. Оскільки запропоний метод показав кращі результати, ніж популярні існуючі рішення відносно часу виконання програми, можна сказати, що мета дослідження була досягнута. Аналіз програмних методів та оцінку якості відсканованого коду виконувався за допомогою утиліти SonarQube. Для аналізу та оцінки результатів розглянутих методів використовувалася утиліта браузера Google Chrome DevTools. Дослідження робить деякі висновки щодо варіантів використання кількох популярних рішень. Наприклад, використання бібліотеки Recoil у більшості випадків показує продуктивність навіть кращу, ніж за використання інших бібліотеки. Крім того, подальший розвиток і вдосконалення оптимізованого методу може стати достатньо зрілим для комерційного використання та перевершити поточні рішення.

Подальші напрями дослідження – це аналіз складності програмних методів з використанням метрик цикломатичної та когнітивної складності, аналіз кожного файлу, пов'язаного з використанням бібліотечних інструментів, тестування даних метрик за допомогою запропонованого методу. Крім того, подальше дослідження в цій галузі також включає в себе більш глибокий аналіз інших популярних програмних рішень і тестових випадків для них. У майбутньому на основі отриманих результатів, якщо буде доведено ефективність запропонованого методу для більшості випадків, метод можна буде опублікувати як бібліотеку для загального користування.

Список використаної літератури

1. Pronina D., Kyrychenko I. (2022) Comparison of Redux and React Hooks Methods in Terms of Performance. *COLINS-2022: 6th International Conference on Computational Linguistics and Intelligent Systems, May 12–13*, Gliwice, Poland, pp. 3-7.

2. Shailesh Shivakumar, P.V. Suresh (2018) A Survey and Analysis of Techniques and Tools for Web Performance Optimization, *Journal of Information Organization*, Vol. 8, No. 2, pp. 31-57.

3. Chi, Xiaoni, Liu, Bichuan, Niu, Qi, Wu, Qiuxuan (2012) Web Load Balance and Cache Optimization Design Based Nginx under High-Concurrency Environment, *Third International Conference on Digital Manufacturing & Automation*, pp. 1029-1032.

4. Yanan Wang, Huarui Wu and Feng Huang (2014) High-performance concurrent Web application system analysis and research, *Computer Engineering and Design Optimization*, vol. 08, pp. 2976-2981.

5. Y. Yao and J. Xia (2016) Analysis and research on the performance optimization of Web application system in high concurrency environment, *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, pp. 321-326, doi: 10.1109/ITNEC.2016.7560374.

6. Mohammed Zagane, Mustapha Kamel Abdi (2019) Evaluating and Comparing Size, Complexity and Coupling Metrics as Web Applications Vulnerabilities Predictors. *International Journal of Information Technology and Computer Science (IJITCS)*, Vol.11, No.7, pp.35-42, DOI: 10.5815/ijitcs.2019.07.05

7. Ruqia Bibi, Munazza Jannisar, Mamoona Inayet (2016) Quality Implication for Prognoses Success in Web Applications. *IJMECS*, Vol.8, No.3, pp.37-44, DOI: 10.5815/ijmecs.2016.03.05

8. Oleshchenko, L., Burchak, P. (2023) Web Application State Management Performance Optimization Methods. In: Hu, Z., Dychka, I., He, M. (eds) *Advances in Computer Science for Engineering and Education VI. ICCSEEA 2023. Lecture Notes on Data Engineering and Communications Technologies*, vol. 181. Springer, Cham. https://doi.org/10.1007/978-3-031-36118-0_6