ПРИКЛАДНІ ПРОБЛЕМИ КОМП'ЮТЕРНИХ НАУК,
БЕЗПЕКИ ТА МАТЕМАТИКИ, 2025, № 5

УДК 004.41

МЕТОДИ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ ОДНОСТОРІНКОВИХ ЗАСТОСУНКІВ НА ПРИКЛАДІ ФРЕЙМВОРКУ REACT

Сюх О.М. (ORCID: 0009-0005-3294-6241)
Волинський національний університет імені Лесі Українки

PERFORMANCE OPTIMIZATION METHODS FOR SINGLE-PAGE APPLICATIONS USING THE REACT FRAMEWORK

Siukh O.M.
Lesya Ukrainka Volyn National University

Abstract. The paper explores methods for optimizing the performance of single-page applications (SPAs) built with React. It focuses on key techniques such as lazy loading, code splitting, memoization, and dynamic imports. These approaches aim to reduce the initial bundle size, minimize unnecessary re-rendering, and improve load times. The impact of each method on client-side speed and efficiency is analyzed using practical examples and basic performance metrics.

Keywords: React, single-page application, lazy loading, code splitting, memoization.

Вступ. У сучасній веброзробці дедалі ширшого поширення набувають односторінкові вебзастосунки (SPA — Single Page Applications), які забезпечують динамічну взаємодію з користувачем без необхідності повного перезавантаження сторінки. Завдяки цьому значно покращується користувацький досвід, оскільки інтерфейс стає швидшим, гнучкішим та більш чутливим до дій користувача. Водночас із розширенням функціональності таких застосунків зростає складність їхньої структури та обсяг даних, що опрацьовуються на клієнтській стороні. У цих умовах особливої актуальності набуває питання оптимізації продуктивності SPA.

Метою цього дослідження є аналіз ключових підходів до оптимізації продуктивності односторінкових React-застосунків, таких як мемоізація, відкладене завантаження (lazy loading), динамічний імпорт модулів та мінімізація коду. Застосування цих методів дозволяє суттєво зменшити час завантаження, обсяг переданих ресурсів, а також кількість повторних обчислень під час рендерингу компонентів.

До основних завдань дослідження належать:

- виявлення технік, які безпосередньо впливають на швидкодію клієнтської частини SPA;
- аналіз принципу роботи кожного з підходів
- оцінка впливу оптимізацій на завантаження та рендеринг

Ключовою особливістю SPA є можливість мінімізувати кількість запитів до сервера та використовувати єдину HTML-сторінку з динамічним оновленням за допомогою JavaScript. У цьому контексті React забезпечує ефективну архітектуру компонування, повторного використання елементів інтерфейсу та оптимізації за рахунок віртуального DOM. Дослідження технік оптимізації є актуальним для розробки масштабованих, стабільних і ресурсоефективних вебзастосунків, що відповідають вимогам сучасного користувача.

Методологія дослідження. Дослідження базується на поєднанні теоретичного аналізу та практичної експериментальної перевірки ефективності методів оптимізації продуктивності вебзастосунків. Методологія побудована таким чином, щоб забезпечити всебічне вивчення, як впровадження різних оптимізацій впливає на роботу інтерфейсу

користувача, швидкість рендерингу та загальну продуктивність односторінкового вебдодатку.

На першому етапі дослідження було проведено систематичний аналіз літературних джерел, технічної документації та практичних кейсів, присвячених оптимізації React-застосунків. Основна увага приділялася методам, що впливають на ефективність рендерингу, управління пам'яттю, структурування коду та завантаження компонентів.

Другий етап передбачав створення експериментального середовища у вигляді прототипу вебзастосунку, який включав ключові функціональні елементи, притаманні реальним проєктам. Серед них були динамічний список товарів, інтерактивні фільтри, сторінки з додатковою інформацією та навігацією. Застосунок було реалізовано у двох основних версіях — базовій (без оптимізацій) і покроково оптимізованій. У другій версії поступово впроваджувались окремі техніки, зокрема мемоізація, віртуалізація елементів, асинхронне завантаження модулів та інші.

Порівняння результатів між версіями дозволило зробити кількісну оцінку впливу кожного з підходів. При цьому особлива увага приділялася умовам, максимально наближеним до реального користувацького середовища — з обмеженням пропускної здатності каналу, затримками мережі та емуляцією роботи на різних типах пристроїв.

Таким чином, метод дослідження включає:

- аналітичне вивчення сучасних підходів до оптимізації продуктивності;
- практичну реалізацію прототипу застосунку з фіксованою логікою;
- поетапне впровадження оптимізацій із контролем результатів;
- кількісне вимірювання ефекту кожного підходу;
- інтерпретацію отриманих даних для формулювання висновків щодо доцільності застосування певних методів.

Такий підхід дозволяє не лише перевірити гіпотези щодо ефективності окремих технік оптимізації, але й сформувати узагальнену методику, яку можна використовувати при розробці продуктивних React-застосунків у різних умовах.

У центрі дослідження: мемоізація, відкладене завантаження та мінімізація коду — методи, що вважаються базовими для підвищення ефективності SPA-застосунків.

Мемоізація. В основі цього підходу лежить ідея кешування результатів обчислень. У React, зокрема, використовуються функції React.memo або useMemo, які дозволяють зберігати результат рендерингу або обчислення і не виконувати його повторно, якщо вхідні дані не змінилися. Цей метод особливо корисний у складних компонентах, що часто оновлюються внаслідок змін стану.

Відкладене завантаження (lazy loading). Сутність цього методу полягає у завантаженні лише тих частин застосунку, які необхідні на даному етапі. Замість того, щоб передавати браузеру весь код одразу, застосунок динамічно завантажує модулі лише тоді, коли користувач звертається до відповідного функціоналу. Це дозволяє суттєво зменшити час першого завантаження сторінки.

Мінімізація та tree-shaking. Під час збірки застосунку важливо прибрати зайвий код, що не використовується. Це досягається за допомогою спеціалізованих збирачів, таких як Vite або Webpack, які дозволяють автоматично видалити мертві гілки імпортів (tree-shaking) та стискати фінальний код до мінімального розміру (мінімізація). У результаті зменшується обсяг трафіку, необхідного для завантаження сторінки.

Для перевірки ефективності зазначених підходів було створено React-застосунок, який імітує роботу типового онлайн-магазину. У початковій версії застосунку жодна з оптимізацій не була реалізована. Цей стан був зафіксований як контрольний, із замірами часу завантаження, часу рендерингу компонентів, використання пам'яті тощо.

Далі, на кожному етапі дослідження поступово впроваджувалася одна з технік оптимізації. Після кожної зміни здійснювалося повторне вимірювання ключових

показників за допомогою інструментів Chrome DevTools, Lighthouse та React Profiler. Завдяки цьому стало можливим простежити, який саме внесок у загальну продуктивність вносить кожен із підходів окремо, а також у сукупності.

Таким чином, методологія дослідження поєднує описовий аналіз теоретичних підходів до оптимізації, реалізацію цих підходів на практиці, а також вимірювання результатів із подальшим їх аналізом. Отримані результати дозволяють зробити висновки про доцільність впровадження окремих технік оптимізації в конкретних випадках, а також надають розробникам орієнтири щодо вибору найбільш ефективних стратегій.

Результати дослідження та їхнє обговорення. Після формування теоретичної основи та вибору релевантних методів оптимізації було проведено низку експериментів з метою перевірки їхньої ефективності на практиці. У процесі аналізу особлива увага приділялась зміні часу першого рендеру, взаємодії користувача з інтерфейсом, навантаження на DOM, а також обсягу використаних ресурсів.

Мемоізація компонентів (React.memo та useMemo) дозволяє уникати непотрібних повторних рендерів компонентів, якщо їхні пропси не змінились. Це особливо важливо у компонентах, що відображаються багаторазово, наприклад у списках або таблицях.

У межах дослідження було створено компонент товарної картки (<ProductCard />), який використовувався для візуалізації великої кількості товарів у колекції. На початковому етапі кожна зміна у стані списку (наприклад, під час фільтрації або сортування) спричиняла повторний рендер усіх елементів, незалежно від того, чи змінились їхні властивості. Це призводило до зайвих витрат ресурсів і уповільнення роботи інтерфейсу, особливо на слабших пристроях або при великому обсязі даних.

Після впровадження useMemo, вдалося досягти суттєвого зменшення кількості оновлень. Інтерфейс став реагувати плавніше, а час до взаємодії зменшився, що в реальних умовах забезпечує помітно комфортніше користування застосунком.

Щоб об'єктивно оцінити вплив застосування useMemo, було проведено серію вимірювань продуктивності компонента списку товарів до та після оптимізації. У таблиці 1 наведено результати дослідження.

Таблиця 1
Результати оптимізації за допомогою useMemo

Метрика	До мемоізації	Після
Середній час рендеру списку	160 мс	53 мс
Повторний рендер одного елемента	5 разів	1 раз
Загальна кількість рендерів	950	270

Ці інструменти дозволяють зменшити кількість непотрібних повторних рендерів компонентів, однак їх надмірне або необґрунтоване використання може викликати зворотний ефект — перевантаження пам'яті або ускладнення логіки коду. Ефективність мемоізації спостерігається лише в ситуаціях, коли вхідні дані змінюються рідко, а обчислення є дійсно ресурсоємним.

Отже, мемоізація продемонструвала значне покращення продуктивності при взаємодії зі списком товарів. Особливо ефективною вона виявилась у ситуаціях, коли змінюється лише частина даних, а структура залишається незмінною. Однак варто

зауважити, що надмірне використання useMemo може ускладнити підтримку коду та не дати приросту продуктивності при простих обчисленнях.

Ліниве завантаження (lazy loading) є ще одним дієвим підходом до оптимізації продуктивності односторінкових вебзастосунків. Цей метод дозволяє уникнути завантаження всіх компонентів або ресурсів одразу при старті застосунку, натомість підвантажуючи лише ті частини, які необхідні користувачеві в конкретний момент. У React такий механізм реалізується за допомогою React.lazy та Suspense [1].

У рамках дослідження цей підхід було реалізовано для сторінок, що не є критично важливими при першому завантаженні — наприклад, сторінки профілю користувача, окремих товарів або адміністративної панелі. Раніше ці компоненти були включені у загальну збірку, що збільшувало час першого завантаження застосунку та затримувало відображення головного інтерфейсу.

Після впровадження React.lazy та Suspense у сторінки почали підвантажуватись лише в момент навігації до них. Це дозволило зменшити початковий обсяг JavaScript-коду, що завантажується браузером, а відповідно — і час до першого відображення інтерфейсу (Time to Interactive, TTI). У користувацькому досвіді це проявляється як швидший старт застосунку та менше очікування на відображення основного контенту.

Щоб зафіксувати зміни, було використано Lighthouse для вимірювання ключових метрик до та після застосування лінивого завантаження. Загальні результати представлено в таблиці 2.

Таблиця 2
Результати оптимізації за допомогою lazy loading

Метрика	До lazy loading	Після
Початковий розмір JavaScript-бандлу	~780 KB	~420 KB
Time to Interactive (TTI)	3.8 с	2.1 с
Індекс швидкодії (Performance Score, Lighthouse)	72	91

Водночас, при першому виклику такого компонента користувач може помітити затримку, особливо якщо не передбачено адекватного запасного вмісту. Тому цей метод не підходить для ключових компонентів інтерфейсу, які мають бути доступними миттєво.

Таким чином, застосування lazy loading суттєво зменшило навантаження при старті застосунку та покращило сприйняття швидкості роботи користувачем. Натомість важливо грамотно визначати, які саме компоненти можуть бути завантажені пізніше, і не застосовувати цей підхід до критично важливих елементів, які мають з'являтися одразу після завантаження сторінки [3].

Мінімізація коду (minification) — це один із базових і водночас ефективних підходів до оптимізації продуктивності вебзастосунків. Суть цього методу полягає у видаленні всіх непотрібних символів із вихідного коду (пробілів, коментарів, переведень рядків тощо), а також у стисканні імен змінних та функцій. У результаті зменшується розмір JavaScript-, CSS- та HTML-файлів, що дозволяє пришвидшити завантаження сторінки та зменшити обсяг передаваних даних.

У рамках дослідження мінімізацію було впроваджено як одну з фінальних стадій оптимізації під час збирання застосунку у production-режимі за допомогою Vite. Цей інструмент автоматично застосовує Terser для JavaScript-файлів і CSSnano для стилів, що забезпечує якісне стиснення без впливу на функціональність.

До впровадження мінімізації загальний розмір бандлів був доволі значним через наявність незмінного коду, неефективних конструкцій та відсутність стиснення. Після її застосування обсяг файлів зменшився майже вдвічі, що позитивно позначилось на швидкості завантаження та відображенні інтерфейсу. У таблиці 3 наведено результати замірів до і після застосування мінімізації.

Таблиця 3
Результати мінімізації коду

Метрика	До мінімізації	Після
Загальний розмір JavaScript-файлів	~670 KB	~340 KB
Розмір основного CSS-файлу	~110 KB	~58 KB
Час завантаження головної сторінки	1.9 с	1.2 с
Індекс швидкодії (Performance Score)	79	93

Водночас цей метод не розв'язує проблеми, пов'язані з неефективною архітектурою або погано організованою логікою компонентів. Production-збірка лише підсилює результат якісно реалізованого коду, але не виправляє його помилки.

Таким чином, мінімізація є простим у реалізації, але дуже ефективним кроком для покращення продуктивності застосунку, особливо у поєднанні з іншими підходами, як-от lazy loading чи code splitting. Важливо впевнитися, що мінімізація не впливає на функціональність (наприклад, через неправильну роботу з глобальними змінними), однак при правильному налаштуванні цей метод є безпечним і значно підвищує ефективність вебзастосунку.

Ефективність компонентної та модульної архітектури в React і її вплив на продуктивність. Однією з ключових концепцій React є компонентний підхід, згідно з яким інтерфейс користувача розбивається на незалежні, повторно використовувані частини — компоненти. Кожен компонент інкапсулює власну логіку, стан та шаблон відображення (JSX-розмітку), що дозволяє створювати великі й масштабовані інтерфейси у зручний та структурований спосіб. Разом з цим React заохочує модульність, тобто організацію коду у вигляді окремих файлів або директорій за принципом "один файл — одна відповідальність". Такий підхід дає низку переваг як під час розробки, так і в аспектах продуктивності виконання застосунку.

Чітка структурованість та масштабованість. Компонентна архітектура дозволяє розділити складний інтерфейс на малі частини, які легше розробляти, тестувати та обслуговувати. Наприклад, замість одного великого файлу з розміткою та логікою можна створити окремі компоненти для картки товару, фільтра, кнопки додавання до кошика тощо. Кожен компонент відповідає лише за свій невеликий фрагмент інтерфейсу, що зменшує кількість побічних ефектів та помилок.

Це спрощує підтримку коду, тому що зміни у внутрішній логіці одного компонента, як правило, не впливають на інші, якщо правильно дотримуватись принципів передавання props і стану.

Повторне використання компонентів. React-компоненти можна багаторазово використовувати у різних частинах застосунку без дублювання коду. Наприклад, одна й та ж кнопка або картка товару може бути відображена у списках, обраних елементах, історії покупок тощо. Це не лише знижує час розробки, а й забезпечує консистентність інтерфейсу. Повторне використання також дозволяє централізовано вносити зміни. Для цього достатньо оновити логіку або стилі компонента один раз, і ці зміни автоматично застосуються скрізь, де він використовується.

Оптимізація продуктивності через ізоляцію оновлень. React ефективно працює завдяки віртуальному DOM і алгоритму порівняння змін (diffing), що дозволяє оновлювати тільки ті частини інтерфейсу, які змінилися. Компонентна модель дозволяє React точніше відслідковувати, які саме частини потрібно оновити, і мінімізувати обсяг роботи браузера.

Покращення часу завантаження (з використанням модульності). Модульний підхід дозволяє організовувати код так, щоб при збірці застосовувались техніки розділення коду (code splitting). Інструменти на кшталт Vite або Webpack можуть автоматично розділити JavaScript-код на окремі частини (чанки), які завантажуються лише за потребою. Таким чином, компоненти, які не потрібні одразу (наприклад, сторінки профілю чи адміністративна панель), не потрапляють у початковий бандл і не збільшують час першого завантаження сторінки.

Можливість тестування та перевикористання логіки. Окрім візуального поділу, React дозволяє організовувати бізнес-логіку у вигляді кастомних хуків (custom hooks), які також підтримують модульний підхід. Такі хуки інкапсулюють повторювану логіку (наприклад, обробку запитів, пагінацію, керування формами) і забезпечують ще більшу ізольованість та повторне використання. Це особливо зручно в командній розробці або в проектах, що швидко розвиваються та доповнюються новими функціональностями.

Обмеження компонентної та модульної архітектури. Попри всі переваги, компонентний підхід має свої виклики. Надмірна фрагментація інтерфейсу може ускладнити навігацію по кодовій базі та призвести до збільшення кількості неузгоджених компонентів. Якщо не впроваджено чітких правил неймінгу, структурування папок та повторного використання, код може швидко стати нечитабельним. Також при неправильному управлінні станом додатка (наприклад, у великих ієрархіях компонентів) можливе дублювання логіки, що знижує ефективність архітектури. Щодо модульності — зловживання дрібними модулями без реальної потреби у повторному використанні може ускладнити збірку і зрештою знизити продуктивність застосунку через надмірну кількість імпортів.

Висновки. У ході дослідження було реалізовано та проаналізовано сучасні підходи до створення продуктивних, масштабованих односторінкових вебзастосунків за допомогою бібліотеки React. Застосування компонентної та модульної архітектури дозволило досягти високої структурованості коду, спростити підтримку та розширення функціоналу, а також забезпечити повторне використання елементів інтерфейсу.

Особливу увагу було приділено оптимізації продуктивності, зокрема впровадженню **лінивого завантаження (lazy loading)** для другорядних сторінок застосунку, **мемоізації компонентів** з використанням useMemo та React.memo для зменшення кількості непотрібних рендерів, а також **мінімізації JavaScript-коду** на етапі збірки для скорочення розміру файлів і пришвидшення завантаження.

Компонентний підхід у React, у поєднанні з ефективним використанням віртуального DOM та можливістю точкового оновлення інтерфейсу, суттєво знижує навантаження на браузер. Ізольованість компонентів дає змогу мінімізувати рендери та уникати небажаних побічних ефектів. Крім того, модульна організація проєкту

спростила розділення логіки та представлення, що стало основою для розділення коду (code splitting) та гнучкого керування залежностями.

Ці заходи позитивно вплинули на загальний користувацький досвід. Застосунок став швидше завантажуватись, реагувати на дії користувача, а також легше адаптується до змін і масштабування в майбутньому. Попри це, ефективність методів оптимізації залежить не лише від їх технічного впровадження, але й від контексту використання. Розмір застосунку, частота змін даних, поведінка користувача та архітектура проєкту відіграють ключову роль у тому, наскільки виправданими будуть ті чи інші оптимізації. Наприклад, мемоізація компонентів за допомогою React.memo або useMemo може бути надлишковою у простих або швидкорендерних елементах і лише ускладнювати підтримку коду. Аналогічно, застосування розділення коду (code splitting) до надто дрібних компонентів може призвести до надмірної кількості HTTP-запитів, що ускладнює завантаження. Мініфікація та зменшення бандлу завдяки сучасним збирачам — корисний підхід, однак її ефект помітний лише при достатньому обсязі коду. Тому розробнику важливо не лише знати можливості оптимізацій, але й вміти критично оцінити доцільність їх застосування у кожному конкретному випадку, балансуючи між складністю реалізації та реальним приростом продуктивності.

Таким чином, дослідження підтвердило, що поєднання компонентного та модульного підходу з техніками оптимізації продуктивності є ефективною стратегією розробки сучасних вебзастосунків. Використання цих практик дозволяє створювати не лише якісний код, але й забезпечувати високу швидкодію, зручність у підтримці та задоволення користувачів від роботи з інтерфейсом.

Бібліографія

1. *React*. *lazy*. URL: https://react.dev/reference/react/lazy#lazy
2. Оптимізація продуктивності. *React Dev*. URL: https://uk.legacy.reactjs.org/docs/optimizing-performance.html
3. *MDN Web Docs – Lazy loading*. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading
4. *Google Developers – Chrome DevTools Performance Analysis*. URL: https://developer.chrome.com/docs/devtools/evaluate-performance/
5. Побережний О. Що таке SPA додаток?. *Asabix*. 06.03.2024. URL: https://asabix.com.ua/what-is-a-single-page-application/
6. Подобєд О. Оптимізація продуктивності в React-застосунку. *DOU*. 26.09.2023. URL: https://dou.ua/forums/topic/45406/

References

1. *React*. *lazy*. URL: https://react.dev/reference/react/lazy#lazy
2. Optimizing Performance – *React. Reac*t – A JavaScript library for building user interfaces. URL: https://legacy.reactjs.org/docs/optimizing-performance.html.
3. *MDN Web Docs – Lazy loading*. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading
4. *Google Developers – Chrome DevTools Performance Analysis*. URL: https://developer.chrome.com/docs/devtools/evaluate-performance/
5. Poberezhnuy O. What is a Single Page Application?. *Asabix*. 06.03.2024. URL: https://asabix.com.ua/what-is-a-single-page-application/
6. Podobed O. Optimizing performance in a React application. *DOU*. 26.09.2023. URL: https://dou.ua/forums/topic/45406/