

Міністерство освіти і науки України
Державний університет «Житомирська політехніка»

Кваліфікаційна наукова
праця на правах рукопису

ВОЙТЮК ОЛЕГ ВАЛЕРІЙОВИЧ

УДК 004.4:004.514+004.021+004.422.8

ДИСЕРТАЦІЯ

**МОДЕЛІ, МЕТОДИ ТА ПРОГРАМНІ ЗАСОБИ ОНОВЛЕННЯ
ІНТЕРФЕЙСУ ВЕБЗАСТОСУНКІВ НА ОСНОВІ ОБ'ЄКТІВ З
ГЛИБОКОЮ ВКЛАДЕНІСТЮ**

121 Інженерія програмного забезпечення

12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело
_____ /Войтюк О.В.

Науковий керівник: Лобанчикова Надія Миколаївна, кандидат технічних наук,
доцент, доцент кафедри інженерії програмного забезпечення

Житомир – 2026

АНОТАЦІЯ

Войтюк О.В. Моделі, методи та програмні засоби оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії з галузі знань 12 Інформаційні технології за спеціальністю 121 Інженерія програмного забезпечення. – Державний університет «Житомирська політехніка» Міністерства освіти і науки України, Житомир, 2026.

Актуальність теми дослідження обумовлена розвитком інформаційних технологій та широким застосуванням вебзастосунків у різних сферах діяльності, зокрема в бізнесі, електронній комерції, державних і освітніх сервісах. За умов зростання кількості користувачів і вимог до якості обслуговування особливого значення набувають продуктивність і швидкодія клієнтської частини систем. Ключову роль у забезпеченні ефективної взаємодії користувача із вебзастосунком відіграють процеси оновлення інтерфейсу, від яких залежить швидкість і коректність відображення даних. Недостатня оптимізація цих процесів може призводити до затримок, зниження продуктивності додатку та погіршення роботи з вебзастосунком на рівні користувача.

Сучасні вебзастосунки характеризуються високою динамічністю та працюють із великими обсягами даних, що мають складну ієрархічну структуру та численні залежності. Підтримка інтерактивного редагування додатково ускладнює процес оновлення інтерфейсу та збільшує навантаження на клієнтську частину. Традиційні підходи до рендерингу, зокрема використання DOM, Virtual DOM не завжди забезпечують необхідну ефективність в умовах високої частоти змін і складної структури даних. У зв'язку з цим актуальним є розробка нових моделей, методів та програмних засобів оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю. Їх впровадження сприятиме підвищенню продуктивності застосунків і покращенню якості користувацької взаємодії.

Метою роботи є розробка моделей, методів та програмних засобів оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю.

Зміст дисертаційного дослідження викладено у трьох розділах, у яких представлено та обґрунтовано основні результати проведеного дослідження.

У **вступі** обґрунтовано актуальність теми дисертації, сформульовано мету, об'єкт та предмет дослідження, визначені основні завдання дослідження, окреслено використані методи, подано відомості про наукову новизну та практичну значимість отриманих результатів, а також наведено дані щодо їх апробації та публікацій.

Перший розділ дисертаційної роботи присвячено аналізу сучасних моделей, методів та технологій оновлення інтерфейсу вебзастосунків та їх оптимізації. Представлено порівняльний аналіз внутрішньої архітектури оновлення вебзастосунків, підходи та механізми оновлення, зокрема DOM, Virtual DOM, Shadow DOM, Signals та Reactive Rendering. Визначено, що керування DOM значно ускладнюється під час роботи з динамічними списками та таблицями, оскільки додавання, видалення чи переміщення елементів часто призводить до втрати стану інтерактивних компонентів і потребує додаткової логіки. Зазначено, що повне «перестворення» елементів також нівелює внутрішні оптимізації браузера, що негативно впливає на продуктивність.

Проведений аналіз підтвердив, що глибока вкладеність Virtual DOM ускладнює роботу з великими формами та залежними полями, оскільки зміна одного значення може спричинити каскадні оновлення й суттєво знижувати продуктивність. Визначено, що додаткова проблема «батьківських» оновлень призводить до непотрібних оновлень дочірніх компонентів, що змушує розробників застосовувати складні підходи до управління станом і ускладнює архітектуру застосунку. Виявлено, що ізоляція Shadow DOM ускладнює передачу даних та керування залежностями між глибоко вкладеними компонентами, що змушує розробників створювати власні механізми на кшталт контексту, оскільки стандарт Web Components цього не забезпечує.

З'ясовано, що продуктивність і стилізація таких компонентів значною мірою залежать від обраної архітектури оновлень та механізмів роботи зі стилями, що часто потребує поєднання Shadow DOM з додатковими бібліотеками й техніками. Підтверджено, що Signals ускладнюють розробку тим, що вимагають правильного керування підписками та уникнення циклічних залежностей, які важко діагностувати у великих застосунках, особливо коли залежності формуються динамічно. Визначено, що необхідність явного створення сигналів забезпечує контроль і продуктивність, але потребує дисципліни розробника, оскільки пропущене огортання значення робить його нереактивним. Виявлено, що у великих додатках Reactive Rendering стикається зі складними графами залежностей, де зміна одного значення може спричинити каскад оновлень, тому системи реактивності застосовують дедуплікацію та топологічне сортування для контролю послідовності обчислень. Визначено, що локальність оновлень дозволяє оновлювати лише ті частини інтерфейсу, що справді залежать від змінених даних, забезпечуючи високу продуктивність навіть у глибоко вкладених структурах.

Проведено аналіз методів та технологій оновлення інтерфейсу вебзастосунків, зокрема розглянуті стратегії клієнтського рендерингу (CSR), серверного рендерингу (SSR), статичну генерацію сайтів (SSG), інкрементальну статичну регенерацію (ISR), прогресивну гідратацію, стрімінговий SSR та edge-side рендеринг (ESR). Визначено, що ефективність вебплатформ визначається вибором стратегії оновлення інтерфейсу вебзастосунків та управління даними, причому найбільш перспективними виявляються гібридні підходи, що поєднують статичну генерацію та інкрементальну регенерацію. Виявлено, що глибоко вкладені структури даних створюють значні виклики для продуктивності вебзастосунків, оскільки ускладнюють рендеринг, реактивність і прогнозованість оновлень. Зазначено, що традиційні підходи – DOM, Virtual DOM і Shadow DOM – виявляються недостатньо ефективними в умовах великих деревоподібних структур, що призводить до каскадних оновлень, дорогих операцій та нестабільної роботи інтерфейсу. Обґрунтовано необхідність знайдення нових моделей і алгоритмів

оптимізації, здатних ефективно працювати з глибоко вкладеними даними та мінімізувати навантаження на оновлення інтерфейсів вебзастосунків.

У другому розділі запропонована формальна модель, яка описує глибоко вкладені структури даних та процес точкового оновлення інтерфейсу вебзастосунків як задачу мінімізації DOM-операцій. Використання графа залежностей і fine-grained реактивності дозволяє локалізувати оновлення та уникати надмірних обчислень, властивих традиційним diffing-підходам. Формально доведено, що такий метод забезпечує кращу або «не гіршу» асимптотичну ефективність, створюючи основу для оптимізованих алгоритмів оновлення інтерфейсу вебзастосунків. Запропонований метод точкового оновлення інтерфейсу вебзастосунків забезпечує локальність оновлень завдяки використанню графа залежностей та fine-grained реактивності. На відміну від традиційних підходів, оновлюються лише ті елементи, що справді залежать від змінених даних, що суттєво зменшує кількість DOM-операцій. Метод демонструє ефективність у роботі з глибоко вкладеними структурами та складними зв'язками, забезпечуючи стабільну продуктивність інтерфейсу.

Проведено моделювання прототипу програмного додатку для динамічного оновлення інтерфейсу вебзастосунків, який реалізує запропоновані принципи локальності оновлень та fine-grained реактивності. Представлено UML-діаграми послідовності та активності, які демонструють детермінований цикл обробки змін, у якому рішення про повторне оновлення інтерфейсу вебзастосунків приймається на основі перевірки релевантності змін через механізм dependencyList. Запропонована архітектура використовує НОС-обгортки як проміжний рівень логіки, що фільтрує події та дозволяє оновлювати лише ті компоненти, які безпосередньо залежать від змінених даних. Такий підхід усуває глобальні перерахунки дерева, характерні для традиційних моделей, і забезпечує часову складність оновлення, пропорційну кількості залежних елементів. Модель також формалізує подієво-орієнтований цикл оновлень, що підвищує передбачуваність та стабільність роботи інтерфейсу.

У третьому розділі представлено прототипи програмного засобу оновлення інтерфейсу вебзастосунків, які реалізованого як засобами сучасного фронтенд-стеку (TypeScript, React, Redux Toolkit, Higher-Order Components (HOC), React.memo, lodash.isEqual, Vite), так і нативного JavaScript. Представлено особливості реалізації та архітектурні та програмні аспекти впровадження. Проведено оцінку продуктивності та швидкодії прототипу програмного засобу оновлення інтерфейсу реалізованого з використанням стеку фронтенд технологій та засобами нативного JavaScript.

Експериментальні дослідження підтвердили ефективність запропонованого методу точкового оновлення інтерфейсу, реалізованого як засобами сучасного фронтенд-стеку (TypeScript, React, Redux Toolkit, HOC, React.memo, lodash.isEqual, Vite), так і нативного JavaScript. У React-прототипі час Scripting при 1000 рядках зменшився з 1137,67 мс до 488,33 мс (приріст 132,97 %, прискорення у 2,3 раза), тоді як у нативній реалізації у сценарії «зміна одного поля» при 10 000 рядках змінилась з 40312,67 мс до 86,67 мс (приріст 46414,62 %, прискорення у 465 разів), що підтверджує універсальність ефекту локалізації оновлень.

Встановлено, що ключовим чинником підвищення продуктивності є зменшення кількості повторних операцій над DOM/Virtual DOM. У React-реалізації використання Redux Toolkit, HOC-обгортки і мемоізації знизило навантаження на Rendering до 50 % у сценарії «зміна папки» при 1000 рядках. У нативному JavaScript кількість DOM-оновлень при 10 000 рядках скоротилася з 70044 до 3 у сценарії «зміна папки» (понад 23 тис. разів), а у сценарії «зміна колонки» – з 70044 до 10000 (у 7 разів), що демонструє високу масштабованість підходу.

Показано, що зі зростанням складності інтерфейсу ефективність методу лише посилюється. Для нативного JavaScript у сценарії «зміна папки» при 10 000 рядках час Scripting зменшився з 42226,33 мс до 334,33 мс (приріст 12530,01 %, прискорення у 126 разів), а Rendering з 3247,00 мс до 1713,00 мс (у 1,9 раза). У сценарії «зміна колонки» час Scripting скоротився – з 41554,33 мс до 684,00 мс (приріст 5975,19 %, прискорення у 61 раз), а Rendering – з

3164,67 мс до 863,33 мс (у 3,7 рази). Це підтверджує практичну придатність методу для високопродуктивних вебзастосунків із великими табличними структурами та глибокою вкладеністю.

Таким чином, прототипи програмного засобу підтвердили ефективність запропонованої моделі та методу як у кількісних, так і в якісних характеристиках, забезпечуючи значне зменшення навантаження на браузерне середовище. Отримані результати не є випадковими, оскільки кожен експериментальний сценарій виконувався багаторазово за однакових умов, а усереднення значень забезпечило низьку варіативність показників. Систематичний вигреш продуктивності, що повторюється у всіх масштабах (10, 100, 1000 рядків для прототипу з використанням стеку фронтенд технологій та 100, 1000 та 10000 для прототипу, реалізованого засобами нативного JavaScript), свідчить про детермінований характер ефекту, а не про випадкові флуктуації. Додатково стабільні тренди зменшення часу Scripting, Rendering і Painting підтверджують, що оптимізація працює закономірно й відтворювано, незалежно від складності інтерфейсу.

Проведені дослідження в межах дисертаційної роботи дозволили отримати наступні нові **наукові результати**, зокрема:

- *вперше* розроблено формальну модель точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, у якій процес оновлення формалізовано як задачу мінімізації кількості DOM-операцій за обмежень коректності відображення; у межах моделі введено поняття області впливу змін та мінімальної множини DOM-вузлів, що підлягають оновленню та забезпечують теоретичне підґрунтя для побудови ефективних алгоритмів точкового оновлення інтерфейсу; формально обґрунтовано застосування графа залежностей на рівні окремих значень даних як базового алгоритму поширення змін, що забезпечує зменшення області оновлення до мінімально необхідної підмножини елементів; запропоновано метод точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, що, на відміну від наявних підходів, орієнтованих на порівнянні

змін, базується на принципах детального відстеження зв'язків, графу залежностей та використанні НОС-обгортки як проміжного рівня логіки;

- *удосконалено* алгоритм оновлення компонентів вебінтерфейсів, який, на відміну від наявних підходів, дозволяє виконувати точкове оновлення компонентів залежно від релевантності змін та забезпечує зменшення часу виконання браузерних сценаріїв обробки даних і зниження навантаження на процес відображення інтерфейсу;

- *набули подальшого розвитку* теоретичні та методичні засади оптимізації оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, що дозволило системно виявити фундаментальні обмеження традиційних підходів, орієнтованих на порівнянні змін, та обґрунтувати перехід до детального відстеження залежностей як якісно нового напрямку розвитку продуктивних вебінтерфейсів.

Практична значимість одержаних результатів полягає у тому, що:

- реалізовано прототипи програмного засобу точкового оновлення інтерфейсу вебзастосунків на базі React/Redux та нативного JavaScript, що підтвердили практичну ефективність запропонованих моделі, методу та алгоритму: для React-прототипу при 1000 рядках час виконання сценаріїв логіки скоротився у 2,3 рази; для нативного JavaScript у сценарії зміни одного поля при 10 000 рядках приблизно у 465 разів, а кількість оновлених DOM-елементів скоротилася з 70 044 до 3;

- розроблено додатковий модуль у вигляді НОС-обгортки, що забезпечує відстеження змін у глобальному стані та прийняття рішень щодо необхідності точкового оновлення в режимі реального часу;

- проведено серію експериментів, що підтвердили масштабованість та ефективність запропонованих моделі, методу та алгоритму для високонавантажених вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками; встановлено, що ефективність методу посилюється зі зростанням складності структури інтерфейсу.

Матеріали дослідження можуть бути використані командами розробників вебзастосунків, що стикаються з проблемами продуктивності рендерингу при роботі з великими ієрархічними структурами даних, а також у навчальному процесі при підготовці фахівців з інженерії програмного забезпечення у відповідних дисциплінах.

Отримані теоретичні та практичні результати дослідження пройшли апробації в ІТ-компаніях та в освітньому процесі Житомирщини, що підтверджується відповідними довідками про впровадження, які додаються, зокрема в ТОВ «Науковий парк Державного університету «Житомирська політехніка», ТОВ «АЙТІ ЛАБ СТУДІО», ТОВ «ВМК ГРУП ЮА», та в освітньому процесі Державного університету «Житомирська політехніка» (довідка №44-22.00/417 від 01.04.2026 р.).

Основні результати дисертаційної роботи опубліковано у 8 наукових працях, зокрема, 3 статті опубліковано у фахових виданнях, включених до переліку наукових фахових видань України категорії «Б», та 5 у матеріалах наукових конференцій.

Ключові слова: веб, вебдодаток, граф, дані, інтерфейс, контент, методи, моделі, обробка даних, оновлення інтерфейсу вебзастосунків, оптимізація швидкодії, програмне забезпечення, програмний додаток, рендеринг, система, структура.

ABSTRACT

Voitiuk O.V. Models, methods and software tools for web application interface updates based on deeply nested objects. – Qualifying scientific work as a manuscript.

Dissertation for the degree of Doctor of Philosophy in the field of knowledge 12 information technologies, specialty 121 software engineering. – Zhytomyr Polytechnic State University, Ministry of Education and Science of Ukraine, Zhytomyr, 2026.

The relevance of the research topic is determined by the development of information technologies and the widespread use of web applications across various domains, including business, e-commerce, government, and educational services. With growing user loads and increasing quality-of-service requirements, the performance and responsiveness of the client-side components of such systems become critically important. A key role in enabling effective user–application interaction is played by interface update processes, which directly affect the speed and accuracy of data rendering. Insufficient optimization of these processes can lead to delays, degraded application performance, and a poor user experience.

Modern web applications are characterized by high dynamism and operate with large volumes of data that possess complex hierarchical structures and numerous interdependencies. Support for interactive editing further complicates the interface update process and increases the load on the client side. Conventional rendering approaches, in particular the use of DOM and Virtual DOM, do not always provide the required efficiency under conditions of high change frequency and complex data structures. Accordingly, there is a pressing need to develop new models, methods, and software tools for updating the interfaces of web applications based on deeply nested objects. Their adoption is expected to improve application performance and enhance the quality of user interaction.

The aim of the work is to develop models, methods, and software tools for updating the interfaces of web applications based on deeply nested objects.

The content of the dissertation research is presented in three chapters, in which the main results of the study are presented and substantiated.

The Introduction justifies the relevance of the dissertation topic, formulates the aim, object, and subject of the research, defines the main research tasks, outlines the methods employed, and provides information on the scientific novelty and practical significance of the results obtained, as well as data on their approbation and publications.

Chapter One of the dissertation is devoted to the analysis of contemporary models, methods, and technologies for updating web application interfaces and their optimization. A comparative analysis of the internal architectures of web application update mechanisms is presented, covering approaches such as DOM, Virtual DOM, Shadow DOM, Signals, and Reactive Rendering. It is established that DOM management is significantly complicated when working with dynamic lists and tables, since adding, removing, or relocating elements frequently leads to the loss of state in interactive components and requires additional logic. It is also noted that the complete “recreation” of elements negates browser-level optimizations, negatively impacting performance.

The analysis confirmed that deep Virtual DOM nesting complicates the handling of large forms and dependent fields, as modifying a single value can trigger cascading updates and substantially degrade performance. The problem of “parent” updates is identified as causing unnecessary re-renders of child components, compelling developers to adopt complex state management strategies and increasing architectural complexity. It is found that Shadow DOM isolation complicates data transfer and dependency management between deeply nested components, forcing developers to implement custom mechanisms analogous to context, since the Web Components standard does not provide this natively. The performance and styling of such components are shown to depend heavily on the chosen update architecture and style-handling mechanisms, often requiring the combination of Shadow DOM with additional libraries and techniques. It is confirmed that Signals impose a development overhead by requiring careful subscription management and avoidance of cyclic dependencies, which are difficult to diagnose in large applications,

especially when dependencies are formed dynamically. The requirement to explicitly create signals provides control and performance but demands developer discipline, since omitting the wrapping of a value renders it non-reactive. It is found that in large applications Reactive Rendering encounters complex dependency graphs, where changing a single value can trigger a cascade of updates; accordingly, reactivity systems apply deduplication and topological sorting to control the sequence of computations. The locality of updates is identified as enabling the refresh of only those parts of the interface that genuinely depend on changed data, ensuring high performance even in deeply nested structures.

An analysis of interface update strategies and technologies is carried out, including Client-Side Rendering (CSR), Server-Side Rendering (SSR), Static Site Generation (SSG), Incremental Static Regeneration (ISR), progressive hydration, streaming SSR, and Edge-Side Rendering (ESR). It is established that the efficiency of web platforms is determined by the choice of interface update strategy and data management approach, with hybrid strategies combining static generation and incremental regeneration proving most promising. It is revealed that deeply nested data structures present significant challenges for web application performance, complicating rendering, reactivity, and the predictability of updates. It is noted that conventional approaches – DOM, Virtual DOM, and Shadow DOM – prove insufficiently effective under large tree-structured data conditions, leading to cascading updates, costly operations, and unstable interface behavior. The necessity of developing new optimization models and algorithms capable of efficiently handling deeply nested data and minimizing the update load on web application interfaces is substantiated.

Chapter Two proposes a formal model that describes deeply nested data structures and the targeted interface update process for web applications as a problem of minimizing DOM operations. The use of a dependency graph and fine-grained reactivity enables the localization of updates and the avoidance of excessive computations inherent in traditional diffing approaches. It is formally proven that this method provides equal or superior asymptotic efficiency, establishing a theoretical foundation for optimized interface update algorithms. The proposed

method for targeted web application interface updates ensures update locality through the use of a dependency graph and fine-grained reactivity. Unlike traditional approaches, only those elements that genuinely depend on the changed data are updated, substantially reducing the number of DOM operations. The method demonstrates effectiveness when working with deeply nested structures and complex interdependencies, ensuring stable interface performance.

A simulation of a prototype software application for dynamic web application interface updates is conducted, implementing the proposed principles of update locality and fine-grained reactivity. UML sequence and activity diagrams are presented, demonstrating a deterministic change-processing cycle in which the decision to re-render the interface is made on the basis of relevance verification through a dependencyList mechanism. The proposed architecture employs HOC wrappers as an intermediate logic layer that filters events and allows updating only those components that directly depend on changed data. This approach eliminates the global tree recomputation characteristic of traditional models and provides an update time complexity proportional to the number of dependent elements. The model also formalizes an event-driven update cycle, enhancing the predictability and stability of interface operation.

Chapter Three presents prototypes of software tools for updating web application interfaces, implemented both with a modern front-end stack (TypeScript, React, Redux Toolkit, Higher-Order Components (HOC), React.memo, lodash.isEqual, Vite) and in native JavaScript. The implementation features and the architectural and programmatic aspects of the deployment are described. Performance and responsiveness evaluations are conducted for both the front-end stack prototype and the native JavaScript implementation.

The experimental studies confirmed the effectiveness of the proposed targeted interface update method, implemented both with a modern front-end stack (TypeScript, React, Redux Toolkit, HOC, React.memo, lodash.isEqual, Vite) and in native JavaScript. In the React prototype, the Scripting time for 1,000 rows decreased from 1,137.67 ms to 488.33 ms (an improvement of 132.97 %, a 2.3× speedup), while in the native implementation for the “single field change” scenario

with 10,000 rows the Scripting time changed from 40,312.67 ms to 86.67 ms (an improvement of 46,414.62 %, a $465\times$ speedup), confirming the universality of the update-localization effect.

It is established that the key factor behind the performance gains is a reduction in the number of repeated DOM/Virtual DOM operations. In the React implementation, the use of Redux Toolkit, HOC wrappers, and memoization reduced the Rendering load by up to 50 % in the “folder change” scenario with 1,000 rows. In native JavaScript, the number of DOM updates for 10,000 rows decreased from 70,044 to 3 in the “folder change” scenario (a reduction of more than 23,000 times), and from 70,044 to 10,000 in the “column change” scenario (a $7\times$ reduction), demonstrating the high scalability of the approach.

It is shown that the effectiveness of the method only increases as interface complexity grows. For native JavaScript in the “folder change” scenario with 10,000 rows, the Scripting time decreased from 42,226.33 ms to 334.33 ms (an improvement of 12,530.01 %, a $126\times$ speedup), and Rendering decreased from 3,247.00 ms to 1,713.00 ms (a $1.9\times$ speedup). In the “column change” scenario, the Scripting time was reduced from 41,554.33 ms to 684.00 ms (an improvement of 5,975.19 %, a $61\times$ speedup), and Rendering from 3,164.67 ms to 863.33 ms (a $3.7\times$ speedup). These results confirm the practical suitability of the method for high-performance web applications with large tabular structures and deep nesting.

Thus, the software prototypes confirmed the effectiveness of the proposed model and method in both quantitative and qualitative terms, providing a significant reduction in the load on the browser environment. The results obtained are not incidental, as each experimental scenario was executed multiple times under identical conditions and the averaging of values ensured low variance in the metrics. The systematic performance gains, consistently reproduced across all scales (10, 100, and 1,000 rows for the front-end stack prototype, and 100, 1,000, and 10,000 rows for the native JavaScript prototype), attest to the deterministic nature of the effect rather than random fluctuations. The consistently stable trends of decreasing Scripting, Rendering, and Painting times further confirm that the optimization operates consistently and reproducibly, regardless of interface complexity.

The research conducted within the dissertation has yielded the following new scientific results, in particular:

- for the first time, a formal model of targeted web application interface updates based on deeply nested objects with multiple interdependencies is developed, in which the update process is formalized as a problem of minimizing the number of DOM operations subject to rendering correctness constraints; within the model, the concepts of the scope of change impact and the minimal set of DOM nodes subject to update are introduced, providing a theoretical foundation for the construction of efficient targeted interface update algorithms; the application of a value-level dependency graph as the basic change propagation algorithm is formally substantiated, ensuring the reduction of the update scope to the minimally required subset of elements; a method for targeted web application interface updates based on deeply nested objects with multiple interdependencies is proposed, which, unlike existing comparison-oriented approaches, is grounded in the principles of fine-grained dependency tracking, a dependency graph, and the use of HOC wrappers as an intermediate logic layer;
- the algorithm for updating web interface components is improved; unlike existing approaches, the improved algorithm enables targeted component updates based on change relevance and reduces browser script execution time and interface rendering load;
- the theoretical and methodological foundations of web application interface update optimization based on deeply nested objects with multiple interdependencies receive further development, enabling a systematic identification of the fundamental limitations of traditional comparison-oriented approaches and substantiating the transition to fine-grained dependency tracking as a qualitatively new direction for the development of high-performance web interfaces.

The practical significance of the results obtained lies in the following:

- prototypes of software tools for targeted web application interface updates are implemented on the basis of React/Redux and native JavaScript, confirming the practical effectiveness of the proposed model, method, and algorithm: for the React prototype with 1,000 rows, the execution time of scripting scenarios was reduced by

2.3 times; for the native JavaScript prototype in the single-field-change scenario with 10,000 rows, approximately 465 times, and the number of updated DOM elements was reduced from 70,044 to 3;

- an additional module in the form of an HOC wrapper is developed, which tracks changes in the global state and makes real-time decisions regarding the necessity of targeted updates;

- a series of experiments is conducted confirming the scalability and effectiveness of the proposed model, method, and algorithm for high-load web applications based on deeply nested objects with multiple interdependencies; it is established that the effectiveness of the method increases as the structural complexity of the interface grows.

The research materials may be used by web application development teams facing rendering performance issues when working with large hierarchical data structures, as well as in the educational process for training specialists in software engineering.

The theoretical and practical results of the research have been approbated in IT companies and in the educational process of Zhytomyr region, as evidenced by the corresponding implementation certificates attached, including those from LLC “Science Park of Zhytomyr Polytechnic State University”, LLC “IT LAB STUDIO”, LLC “VMK GROUP UA”, and the educational process of Zhytomyr Polytechnic State University (Certificate No. ф44-22.00/417 dated 01.04.2026).

The main results of the dissertation are published in 8 scientific works, including 3 articles published in professional journals included in the list of professional scientific publications of Ukraine, category B, and 5 papers in scientific conference proceedings.

Keywords: web, web application, graph, data, interface, content, methods, models, data processing, web application interface update, performance optimization, software, application, rendering, system, structure.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові статті, в яких опубліковано основні наукові результати
дисертації:

**Статті у фахових виданнях, включених до переліку наукових
фахових видань України категорії «Б»:**

1. Войтюк О.В., Плечистий Д.Д. Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. *Технічна інженерія*. 2023. № 1 (91). С. 140-145. DOI: [https://doi.org/10.26642/ten-2023-1\(91\)-140-145](https://doi.org/10.26642/ten-2023-1(91)-140-145).

2. Войтюк О. В. Аналіз методів та технологій рендерингу вебзастосунків. *Вчені записки Таврійського національного університету імені В.І. Вернадського*. Серія: Технічні науки. 2025. Том 36 (75). № 6. С.74-80. DOI: <https://doi.org/10.32782/2663-5941/2025.6.2/11>.

3. Войтюк О.В., Лобанчикова Н.М. Методи оптимізації швидкодії оновлення вебзастосунків на основі об'єктів з глибокою вкладеністю. *Computer Science and Applied Mathematics*. 2026. № 1. С.56-68. DOI: <https://doi.org/10.26661/2786-6254-2026-1-06>.

Публікації у матеріалах наукових конференцій:

4. Войтюк О. В. Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. Тези V Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення». м. Житомир, 01-02 грудня 2022 р. м. Житомир: «Житомирська політехніка». 2022 р. С. 12-13. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2023/02/12.pdf>.

5. Войтюк О. В., Плечистий Д. Д. Багатозалежні зв'язки у рекурсивній структурі фронтенд додатку. Тези VI Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення». м. Житомир, 29-30 листопада 2023 р., м. Житомир: «Житомирська політехніка». 2023. С.56-57. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2024/01/56-1.pdf>.

6. Войтюк О. В. Плечистий Д. Д. Методи оптимізації рекурсивних циклів в JavaScript. Міжнародна конференція «Передові технології в інформаційно-комунікаційній інженерії» (ATICE'2024): Матеріали конф. м. Одеса, Україна, 17-20 лип., 2024 р. / гол. ред. кол. С. В. Ківалов. Одеса, Міжнар. гуманітар. ун-т. 2024. С.12-14. URL: <https://hdl.handle.net/11300/28155>; DOI:10.32837/11300.28155.

7. Фещенко С.О., Мацієвський В.А., Нікітчук Т.М., Коренівська О.Л., Войтюк О.В. Розробка вебзастосунку для дослідження реологічних властивостей біологічних рідин. Тези XII Міжнародної науково-практичної конференції «Topical aspects of modern scientific research», 8-10 серпня 2024 р. Токіо, Японія. С. 98-101. URL: <https://surl.li/ixykb>.

8. Войтюк О.В. Глибоко вкладені структури даних із багатозалежними зв'язками: продуктивність та оновлення стану при рендерингу. Тези VIII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення, 02-03 грудня 2025 р. м. Житомир: «Житомирська політехніка». С.293-294. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2026/01/293.pdf>.

ЗМІСТ

Перелік прийнятих умовних скорочень	21
Вступ.....	22
РОЗДІЛ 1. Аналіз сучасних моделей, методів та технологій оновлення інтерфейсу вебзастосунків та їх оптимізація.....	31
1.1. Порівняльний аналіз внутрішньої архітектури оновлення вебзастосунків	31
1.2. Аналіз методів та технологій оновлення інтерфейсу вебзастосунків ...	43
1.3. Аналіз проблем продуктивності при оновленні інтерфейсу вебзастосунків з глибокою вкладеністю структур	54
Висновки до першого розділу	64
РОЗДІЛ 2. Формалізація моделей представлення та оптимізації оновлення інтерфейсу вебзастосунків з глибокою вкладеністю	66
2.1. Формальна модель точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю.....	66
2.2. Метод точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками.....	75
2.3. Моделювання прототипу програмного додатку для динамічного оновлення інтерфейсу вебзастосунків	96
Висновки до другого розділу	105
РОЗДІЛ 3. Реалізація та оцінка якості прототипу програмного забезпечення.....	107
3.1. Реалізація прототипу програмного засобу оновлення інтерфейсу вебзастосунку з використанням стеку фронтенд технологій	107
3.2. Архітектурні та програмні аспекти впровадження прототипу	117

3.3. Оцінка продуктивності та швидкодії прототипу програмного засобу оновлення інтерфейсу, реалізованого з використанням стеку фронтенд технологій.....	125
3.4. Оцінка продуктивності та швидкодії прототипу програмного засобу оновлення інтерфейсу реалізованого засобами нативного JavaScript.....	143
Висновки до третього розділу	166
Висновки.....	170
Література	175
Додатки	190

Перелік прийнятих умовних скорочень

Скорочення, термін, позначення	Пояснення
CLS	Cumulative Layout Shift – Метрика Core Web Vitals, що вимірює візуальну стабільність вебсторінки
CSR	Client-Side Rendering – Клієнтський рендеринг
FCP	First Contentful Paint – метрика вебпродуктивності
INP	Interaction to Next Paint – Ключова метрика Core Web Vitals від Google, яка вимірює швидкість реакції вебсторінки на будь-які взаємодії користувача
ISR	Incremental Static Regeneration – Інкрементальна регенерація статичних сторінок
LCP	Largest Contentful Paint – Ключова метрика Google, що вимірює час завантаження найбільшого видимого елемента на сторінці (зображення, відео або великого блоку тексту) від початку завантаження до його відображення користувачем
SEO	Search Engine Optimization – Пошукова оптимізація сайтів
SPA	Single Page Application – Односторінкові сайти
SSG	Static Site Generation – Статична генерація сайтів
SSR	Server-Side Rendering – Серверний рендеринг
TTFB	Time To First Byte – Метрика, що показує час від відправлення запиту браузером до моменту отримання першого байта даних від вебсервера
TTI	Time To Interactive – Метрика вебпродуктивності, яка вимірює час, за який вебсторінка стає повністю інтерактивною та готовою до взаємодії з користувачем

Вступ

Актуальність теми зумовлена стрімким розвитком цифрових технологій, глобальною діджиталізацією суспільства та зростанням вимог до якості, швидкодії й масштабованості вебзастосунків. Сучасні інформаційні системи обслуговують мільйони користувачів, забезпечують важливі бізнес-процеси, підтримують електронну комерцію, державні сервіси, медичні та освітні платформи. У таких умовах методи, технології та механізми оновлення інтерфейсу вебзастосунків стають ключовим інструментом забезпечення ефективної взаємодії людини з цифровим середовищем, оскільки саме швидкість і коректність відображення інтерфейсу визначають якість взаємодії користувачів із вебзастосунком.

Кожен розробник прагне створити програмний продукт із високими показниками продуктивності, що включають швидкість роботи, доступність, якість обслуговування та зручність дизайну. Дотримання цих аспектів безпосередньо впливає на утримання користувача в межах будь-якого вебзастосунку. Одним із ключових факторів швидкодії є швидкість промальовування вебелементів, яка визначає, наскільки оперативно інтерфейс реагує на дії користувача. Ігнорування цього аспекту, особливо на ранніх етапах проєктування архітектури, призводить до деградації продуктивності та втрати конкурентоспроможності продукту.

У простих вебдодатках, де кількість ітерацій мінімальна, потреба у складних архітектурних рішеннях є незначною. Проте сучасні системи дедалі частіше виходять за межі статичних шаблонів і передбачають динамічне формування інтерфейсу, конфігурацію елементів користувачем або автоматичну зміну структури даних системою. Прикладом таких рішень є конфігураційні інтерфейси, зокрема системи налаштування CNC-обладнання, де кожна одиниця має унікальні властивості та алгоритми обчислення. У подібних застосунках фронтенд-частина працює з великими обсягами даних,

складними формами, численними залежностями між полями, обрахунками, трансформаціями та інтеграцією зі сховищами даних.

Надання користувачу можливості редагувати всі властивості сутностей призводить до проблеми «динамічної зміни наповнення додатку», яка суттєво збільшує навантаження на механізми оновлення інтерфейсу вебзастосунку. У більшості випадків такі інтерфейси мають «структурний скелет» на стороні серверної частини, що характеризується глибокою вкладеністю та численними залежними зв'язками. Це створює значні виклики для клієнтської частини, оскільки саме вона відповідає за промальовування та оновлення візуальних компонентів у реальному часі.

Швидкість роботи вебзастосунку є важливою характеристикою. Низька продуктивність інтерфейсу спонукає користувачів залишати ресурс, що негативно позначається на бізнес-показниках. Хоча швидкість роботи залежить від багатьох факторів – інтернет-з'єднання, браузера, серверної частини – саме клієнтська логіка та ефективність оновлення інтерфейсу визначають, наскільки швидко користувач побачить результат взаємодії.

Ускладнення клієнтської частини великою кількістю даних, вкладеними структурами, складними залежностями та динамічними обчисленнями вимагає застосування спеціалізованих моделей та алгоритмів оптимізації оновлення інтерфейсу вебзастосунків. Традиційні підходи (DOM, Virtual DOM, компонентний diffing) демонструють обмежену ефективність у випадках глибокої вкладеності та високої частоти змін. Це обумовлює потребу у комплексному дослідженні моделей, методів та алгоритмів оптимізації швидкодії оновлень, здатних забезпечити баланс між продуктивністю, інтерактивністю та доступністю вебзастосунків.

Таким чином, актуальною є наукова задача розробки моделей, методів та алгоритмів оптимізації швидкодії оновлення вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. Її вирішення сприятиме підвищенню продуктивності сучасних вебсистем, покращенню

взаємодії та забезпеченню конкурентоспроможності цифрових продуктів у динамічному технологічному середовищі.

Зв'язок роботи з науковими програмами, планами та темами. Тема дисертації затверджена Вченою радою Державного університету «Житомирська політехніка» (протокол № 1 від 26.01.2026 р.)

Дисертаційне дослідження виконувалось відповідно до НДР «Розробка, проєктування і тестування сайту підприємства» (12.01НДР-37-2023) та НДР «Модернізація веб-сайту підприємства» (12.01НДР-39-2023) кафедри інженерії програмного забезпечення Державного університету «Житомирська політехніка» (2023-2025 рр.).

Мета і завдання дослідження. Метою роботи є розробка моделей, методів та програмних засобів оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю.

Відповідно до поставленої мети виділено наступні **задачі дослідження:**

- провести аналіз сучасних моделей, методів та технологій оновлення інтерфейсу вебзастосунків та його оптимізації, зокрема провести порівняльний аналіз внутрішньої архітектури оновлення вебзастосунків, аналіз методів та технологій оновлення інтерфейсу вебзастосунків, аналіз проблем продуктивності при оновленні інтерфейсу вебзастосунків з глибокою вкладеністю структур;

- формалізувати модель точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками із застосуванням графу залежностей на рівні окремих значень даних як базового алгоритму поширення змін.

- розробити метод точкового оновлення інтерфейсу вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язкам;

- провести моделювання прототипу програмного додатку для точкового оновлення інтерфейсу вебзастосунків та здійснити його програмну реалізацію;

– провести експериментальну перевірку запропонованих рішень на основі прототипу вебзастосунку та оцінити його ефективність.

Об’єкт дослідження – процеси оновлення інтерфейсу вебзастосунків на основі об’єктів з глибокою вкладеністю.

Предметом дослідження – моделі, методи та програмні засоби оновлення інтерфейсу вебзастосунків на основі об’єктів з глибокою вкладеністю.

Методи дослідження. Для виконання всіх поставлених задач дисертаційного дослідження та отриманих наукових результатів застосовано комплекс методів. Використані методи охоплюють теоретичний, аналітичний, формальний, інженерний та експериментальний рівні, що забезпечило цілісність і наукову обґрунтованість отриманих результатів. Системний підхід використано для аналізу предметної області; теоретичний аналіз і синтез застосовувався для дослідження сучасних підходів до оновлення інтерфейсів вебзастосунків, моделей реактивності, алгоритмів diffing, механізмів керування станом та оптимізації DOM-операцій; аналіз наукових джерел і практичних реалізацій дозволив виявити обмеження традиційних архітектур та сформулювати вимоги до нової моделі точкового оновлення інтерфейсу вебзастосунків на основі об’єктів з глибокою вкладеністю та багатозалежними зв’язками; методи формального моделювання використано для опису структури даних, процесу та механізму оновлення інтерфейсу вебзастосунку; методи математичного моделювання застосовано для побудови моделі дерева даних, графа залежностей, функцій оновлення DOM та критеріїв оптимальності, що забезпечило можливість строгого доведення асимптотичних переваг fine-grained реактивності над diffing-підходами; методи структурного та поведінкового моделювання (UML-діаграми послідовності, активності, компонентів) використовувалися для опису логіки роботи прототипу, взаємодії між компонентами, механізму фільтрації подій та локалізації оновлень, що дозволило формально перевірити коректність алгоритмів до їх програмної реалізації; методи програмної реалізації та

інженерний експеримент застосовувався для створення прототипу вебзастосунку з використанням React, Redux та HOC-обгортки, що дало змогу перевірити працездатність теоретичної моделі в реальних умовах браузерного середовища, враховуючи обмеження головного потоку, асинхронність та поведінку JavaScript-двигуна; експериментальне профілювання продуктивності, а саме інструменти профілювання браузера використано для вимірювання часу Scripting, Rendering, Painting, кількості повторних оновлень, завантаження CPU та споживання пам'яті, що дозволило кількісно оцінити ефективність запропонованого методу та визначити його вплив на продуктивність у різних масштабах інтерфейсу (10, 100, 1000 рядків); метод порівняльного аналізу використовувався для зіставлення базової реалізації (примітивного підходу) та оптимізованої конфігурації за однакових умов, що дозволило об'єктивно оцінити вигреш у продуктивності та масштабованості; методи статистичного аналізу результатів, зокрема усереднення результатів, оцінка варіативності, аналіз стабільності показників та перевірка повторюваності експериментів застосовано для підтвердження достовірності експериментальних даних, що дозволило підтвердити не випадковість отриманих експериментальних даних та їх детермінованість.

Використання технологій штучного інтелекту. Формування наукових положень, постановка мети і завдань дослідження, проведення експериментів, обробка результатів та формулювання висновків виконані автором самостійно. Технології штучного інтелекту не використовувались для генерування тексту дисертації чи формулювання наукових результатів.

Наукова новизна та теоретичне значення отриманих результатів полягає у наступному:

- *вперше* розроблено формальну модель точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, у якій процес оновлення формалізовано як задачу мінімізації кількості DOM-операцій за обмежень коректності відображення; у межах моделі введено поняття області впливу змін та мінімальної множини

DOM-вузлів, що підлягають оновленню та забезпечують теоретичне підґрунтя для побудови ефективних алгоритмів точкового оновлення інтерфейсу; формально обґрунтовано застосування графа залежностей на рівні окремих значень даних як базового алгоритму поширення змін, що забезпечує зменшення області оновлення до мінімально необхідної підмножини елементів; запропоновано метод точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, що, на відміну від наявних підходів, орієнтованих на порівнянні змін, базується на принципах детального відстеження зв'язків, графу залежностей та використанні НОС-обгортки як проміжного рівня логіки;

– *удосконалено* алгоритм оновлення компонентів вебінтерфейсів, який, на відміну від наявних підходів, дозволяє виконувати точкове оновлення компонентів залежно від релевантності змін та забезпечує зменшення часу виконання браузерних сценаріїв обробки даних і зниження навантаження на процес відображення інтерфейсу;

– *набули подальшого розвитку* теоретичні та методичні засади оптимізації оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, що дозволило системно виявити фундаментальні обмеження традиційних підходів, орієнтованих на порівнянні змін, та обґрунтувати перехід до детального відстеження залежностей як якісно нового напрямку розвитку продуктивних вебінтерфейсів.

Практична значимість одержаних результатів полягає у тому, що:

– реалізовано прототипи програмного засобу точкового оновлення інтерфейсу вебзастосунків на базі React/Redux та нативного JavaScript, що підтвердили практичну ефективність запропонованих моделі, методу та алгоритму: для React-прототипу при 1000 рядках час виконання сценаріїв логіки скоротився у 2,3 рази; для нативного JavaScript у сценарії змін одного поля при 10 000 рядках приблизно у 465 разів, а кількість оновлених DOM-елементів скоротилася з 70 044 до 3;

– розроблено додатковий модуль у вигляді НОС-обгортки, що забезпечує відстеження змін у глобальному стані та прийняття рішень щодо необхідності точкового оновлення в режимі реального часу;

– проведено серію експериментів, що підтвердили масштабованість та ефективність запропонованих моделі, методу та алгоритму для високонавантажених вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками; встановлено, що ефективність методу посилюється зі зростанням складності структури інтерфейсу.

Матеріали дослідження можуть бути використані командами розробників вебзастосунків, що стикаються з проблемами продуктивності рендерингу при роботі з великими ієрархічними структурами даних, а також у навчальному процесі при підготовці фахівців з інженерії програмного забезпечення у відповідних дисциплінах.

Впровадження результатів дослідження в практику підтверджується відповідними довідками про впровадження, які додаються, зокрема в ТОВ «Науковий парк Державного університету «Житомирська політехніка», ТОВ «АЙТІ ЛАБ СТУДІО», ТОВ «ВМК ГРУП ЮА», та в освітньому процесі Державного університету «Житомирська політехніка» (довідка №44-22.00/417 від 01.04.2026 р.).

Особистий внесок здобувача. Всі основні результати дисертаційного дослідження, які представлені до захисту, одержані автором особисто (звіт подібності сформований за допомогою програми StrikePlagiarism додається). У публікаціях, написаних у співавторстві, здобувачеві належать наступні результати. У роботі [1] здобувачем визначено актуальність тематики дослідження, проведено аналіз останніх досліджень та публікацій, здійснено аналіз продуктивності додатка за допомогою інструментів Performance insight, визначено шляхи оптимізації оновлення інтерфейсів вебзастосунків. У роботі [3] здобувачем здійснено огляд літератури та порівняльний аналіз сучасних моделей, методів, архітектур та алгоритмів оновлення інтерфейсу вебзастосунків; сформовано концепцію точкового оновлень, яка базується на

принципах локальності оновлень, fine-grained реактивності, прямого оновлення DOM, графу залежності та прив'язки вузлів до даних DOM-вузлів; побудовано схеми процесів генерації та оновлення елементів користувацького інтерфейсу із застосуванням запропонованого оптимізованого методу; проведено експериментальне дослідження. У роботі [5] здобувачем визначено проблему побудови рекурсивного дерева елементів, запропоновано і обґрунтовано шляхи оптимізації та механізми реалізації. У роботі [6] здобувачем визначено проблеми методу рекурсії, що відіграє ключову роль у побудові складних користувацьких інтерфейсів, проведено аналіз відомих механізмів та принципів покращення продуктивності вебзастосунків. У роботі [7] здобувачем запропоновано структуру вебзастосунку на основі об'єктів з глибокою вкладеністю та визначено основні функціональні можливості вебзастосунку.

Апробація результатів дисертації. Основні результати дисертаційного дослідження доповідалися та обговорювалися на наступних наукових та науково-практичних конференціях міжнародного та національного рівня:

Основні результати наукових досліджень неодноразово доповідалися на ряді міжнародних та Всеукраїнських наукових конференціях та семінарах, зокрема: V Всеукраїнській науково-технічній конференції «Комп'ютерні технології: інновації, проблеми, рішення» (Україна, 2022), VI Всеукраїнській науково-технічній конференції «Комп'ютерні технології: інновації, проблеми, рішення» (Україна, 2023), Міжнародній конференції «Передові технології в інформаційно-комунікаційній інженерії» (ATICE'2024, Україна, 2024), XII Міжнародній науково-практичній конференції «Topical aspects of modern scientific research» (Японія, 2024), VIII Всеукраїнській науково-технічній конференції «Комп'ютерні технології: інновації, проблеми, рішення» (Україна, 2025).

Публікації. За результатами проведених досліджень в межах дисертаційної роботи опубліковано 8 наукових праць, з них 3 статті у

фахових наукових виданнях України категорії «Б» та 5 у матеріалах наукових конференцій.

Структура роботи. Дисертаційна робота складається зі вступу, трьох розділів, висновків, вісьми додатків та списку використаних джерел, що включає 124 найменування. Загальний обсяг роботи становить 258 сторінок, у тому числі 153 сторінки основного тексту, містить 49 рисунків, 22 таблиці.

РОЗДІЛ 1. Аналіз сучасних моделей, методів та технологій оновлення інтерфейсу вебзастосунків та їх оптимізація

1.1. Порівняльний аналіз внутрішньої архітектури оновлення вебзастосунків

Сучасні вебдодатки характеризуються високою динамічністю, складністю інтерфейсів та вимогами до продуктивності. Зростання обсягів даних, глибоко вкладених структур та інтерактивності призвело до появи різних архітектур рендерингу вебзастосунків, що оптимізують оновлення інтерфейсу [14;35]. В межах дисертації під рендерингом будемо розуміти процес перетворення вихідного коду та даних на інтерактивну, візуально відображену сторінку, яку бачить користувач, тобто «відмальовування» інтерфейсу, що забезпечує взаємодію людину з програмою. Під об'єктами з глибокою вкладеністю розуміється патерн даних, де об'єкт містить масиви чи інші об'єкти, що посиляються один на одного.

Проведемо більш детальний аналіз п'яти ключових підходів, зокрема DOM, Virtual DOM, Shadow DOM, Signals та Reactive Rendering [45;63]. Кожен із них має власну модель оновлення інтерфейсу, механізми оптимізації та вплив на архітектуру застосунку. Document Object Model (DOM) є стандартною моделлю представлення HTML-документа у вигляді дерева об'єктів, рис. 1.1. Кожен елемент, атрибут або текстовий вузол є частиною дерева, яким можна керувати через JavaScript [15]. DOM є основою для всіх інших архітектур рендерингу, оскільки саме він визначає, як браузер відображає сторінку [25;65]. Оновлення DOM є доволі складним, оскільки кожна зміна може спричинити перерахунок стилів (recalculate styles), переформатування макета (layout), перемальовування (repaint), рендеринг (composite). Ці операції є блокуючими та можуть спричиняти затримки, особливо у випадку великої кількості дрібних оновлень. Легко створити баги при складному інтерфейсі взаємодії користувача з додатком (User Interface (UI)). Масштабні зміни в дереві призводять до значної деградації

продуктивності. DOM не оптимізований для частих оновлень у динамічних SPA (Single-Page Applications) [88].

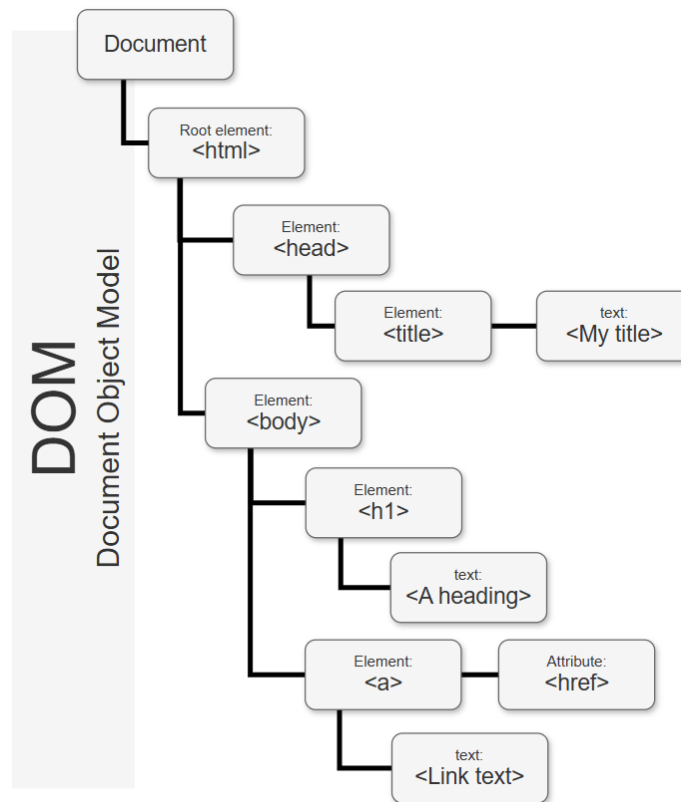


Рис.1.1. Вигляд об'єктної моделі документа (DOM) [43]

Серед переваг слід відмітити просту та інтуїтивну модель, прямий доступ до елементів сторінки та те, що DOM підтримується всіма браузерами. Підсумовуючи можна зробити висновок про те, що DOM є фундаментом вебрендерингу, але його пряме використання неефективне для складних інтерфейсів із глибокою вкладеністю.

Механізм роботи DOM базується на принципі синхронного виконання операцій. При будь-якій зміні властивостей елемента браузер ініціює послідовність етапів оновлення інтерфейсу вебзастосунків, що включає:

- перерахунок стилів (style recalculation) для всіх елементів, які можуть бути ураженими зміною;
- розрахунок геометрії елементів (layout/reflow), що визначає позиціонування та розміри блоків;

- побудову списку відображення (paint), де формуються растрові шари для візуалізації;
- композитування (composite), під час якого шари об'єднуються у фінальне зображення на екрані.

Кожен із цих етапів має свою обчислювальну складність, і саме їх послідовне виконання є причиною деградації продуктивності у випадку інтенсивних маніпуляцій із DOM [61;93]. Для ілюстрації впливу DOM-операцій на продуктивність розглянемо типовий сценарій оновлення списку із 1000 елементами. При зміні текстового вмісту кожного елемента через `innerHTML` або `textContent` браузер виконує 1000 окремих `reflow`-операцій, якщо зміни не згруповані. Навіть за умови використання методу `updateBatch`, що дозволяє об'єднати зміни у єдину транзакцію, час виконання залишається лінійно залежним від кількості елементів. У випадку глибоко вкладених структур, де один елемент списку містить вкладені компоненти (наприклад, карточки товарів із зображеннями, описами, кнопками та випадаючими меню), складність зростає експоненційно. Це пов'язано з тим, що зміна батьківського елемента може спричинити каскадний `reflow` усіх дочірніх вузлів, навіть якщо їх вміст не змінився.

Додатковою проблемою прямих DOM-маніпуляцій є відсутність автоматичного відстеження залежностей між даними та інтерфейсом. Розробник має вручну синхронізувати зміни стану додатку з відповідними DOM-елементами, що призводить до помилок у складних інтерфейсах. Наприклад, при оновленні моделі даних, яка впливає на кілька незалежних компонентів інтерфейсу, розробник повинен явно викликати оновлення кожного з цих компонентів. У разі пропуску одного з оновлень виникає розбіжність між станом даних та візуальним представленням, що призводить до некоректної роботи додатку.

Складність управління DOM зростає ще більше при роботі з динамічними списками та таблицями, де елементи можуть додаватися, видалятися або змінювати позицію. Використання традиційних підходів,

таких як повне очищення контейнера та повторне створення елементів, призводить до втрати стану інтерактивних елементів (фокусу, позиції скролу, значень у формах) [24;39]. Збереження цих станів вимагає додаткової логіки, що значно ускладнює код та збільшує ймовірність помилок. Крім того, повне перестворення DOM-елементів скидає внутрішні оптимізації браузера, такі як кешування геометрії та стилів, що додатково погіршує продуктивність.

Virtual DOM являє собою абстрактне (віртуальне) дерево, що відображає стан інтерфейсу в пам'яті, рис. 1.2 [24;25]. Зміни застосовуються спочатку до віртуального дерева, після чого алгоритм diffing визначає мінімальний набір оновлень для реального DOM.

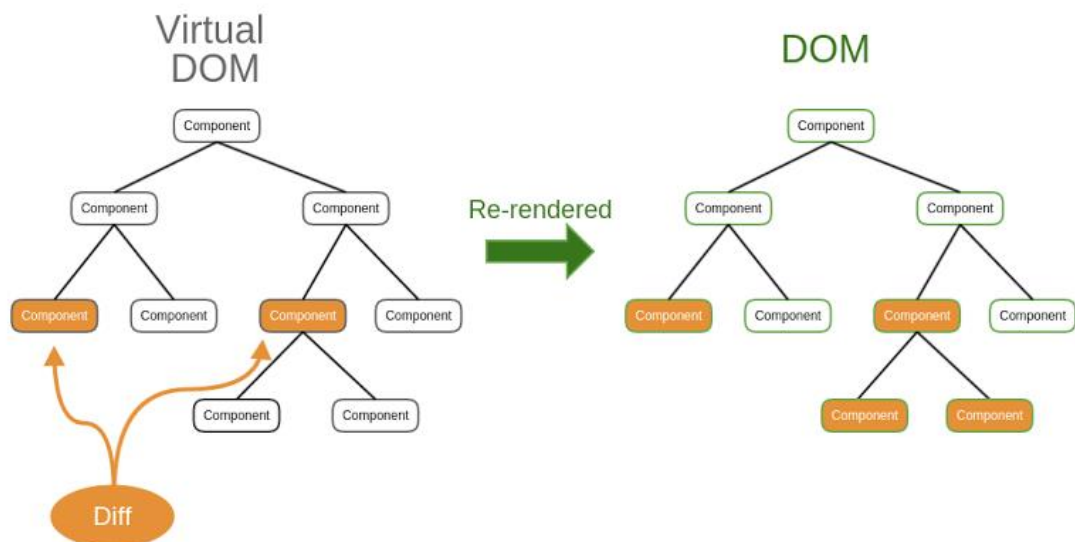


Рис.1.2. Схематичне зображення концепції Virtual DOM [73]

В результаті такого підходу зменшується кількість операцій над реальним DOM, з'являється можливість оптимізації через батчинг, мемоізацію, ключі, підтримується декларативний стиль програмування, що використовується у React, Vue 2 та Preact [122;123]. Алгоритм diffing має лінійну складність $O(n)$, де n – кількість елементів, що гірше для дерев з глибокою вкладеністю, де даний алгоритм стає «вузьким місцем». VDOM значно зменшує кількість операцій над DOM, але має власні накладні витрати на побудову нового дерева, порівняння дерев та створення патчів. Як висновок можна зазначити, що Virtual DOM значно покращує продуктивність у

порівнянні з прямим DOM-маніпулюванням, але не вирішує проблеми глибокої вкладеності, де diffing стає надто дорогим.

Алгоритм diffing у Virtual DOM базується на евристичному підході, який припускає, що елементи різних типів генерують різні піддерева, а розробник може вказати, які елементи залишаються стабільними між оновленнями за допомогою атрибуту `key`. React використовує алгоритм узгодження (reconciliation), що працює за часову складність $O(n)$, де n – загальна кількість елементів у дереві [101,102]. Порівняння відбувається пошарово: спочатку порівнюються кореневі елементи, потім їх діти, і так далі до листів дерева.

Одним із ключових обмежень Virtual DOM є те, що порівняння дерев необхідне навіть у випадках, коли реальних змін немає або вони локалізовані у невеликій частині інтерфейсу. Наприклад, при зміні одного елемента у списку з 10 000 елементів алгоритм diffing повинен пройти всі елементи, щоб визначити, який саме змінився. Навіть якщо використовується оптимізація через `shouldComponentUpdate` або `React.memo`, перевірка умов оновлення також має свою «вартість», особливо у випадку великої кількості компонентів [7;55]. Крім того, щоб diffing працював ефективно, розробник повинен дотримуватися певних правил, таких як використання стабільних ключів для елементів списків та уникнення створення нових об'єктів або функцій під час рендеру, інакше мемоізація не спрацює, і компоненти перерендеруються навіть без реальних змін даних.

Проблема глибокої вкладеності у Virtual DOM особливо гостро проявляється у випадку великих форм із залежними полями. Припустимо, форма містить 50 полів, з яких 20 мають складні залежності між собою (наприклад, вибір значення у одному полі впливає на доступність і валідацію інших полів). При зміні одного поля React перерендерує всю форму, оскільки стан форми зазвичай зберігається на верхньому рівні компонента. Навіть якщо розробник використовує методи оптимізації, такі як розбиття форми на підкомпоненти з власним станом або використання бібліотек управління формами (React Hook Form, Formik), складність управління залежностями між

полями залишається високою. У деяких випадках зміна одного поля може призвести до каскадних перерендерів усіх залежних полів, що погіршує продуктивність та реактивність інтерфейсу.

Додатковим недоліком Virtual DOM є те, що він не усуває проблему «батьківських» оновлень. Якщо стан змінюється у кореновому компоненті, всі дочірні компоненти потенційно можуть бути перерендерені, навіть якщо їхні пропси не змінилися. Для уникнення цього React пропонує інструменти, такі як Context API з Provider/Consumer, useContext, або використання бібліотек управління станом (Redux, MobX, Zustand), але кожен із цих підходів має свої обмеження [20;43]. Наприклад, Context API призводить до перерендеру всіх споживачів контексту при будь-якій зміні значення контексту, навіть якщо конкретний споживач не залежить від зміненої частини даних. Це змушує розробників створювати численні контексти для різних частин стану, що ускладнює архітектуру додатку [37;47].

Shadow DOM – це механізм інкапсуляції DOM-піддерев, що дозволяє створювати компоненти з власними стилями та структурою [45;60]. Він є частиною стандарту Web Components, рис.1.3. Кожен компонент має власне ізольоване дерево, стилі та логіку. До переваг слід віднести повну інкапсуляцію стилів і структури, зменшення конфліктів між компонентами, можливість створення високорівневих UI-бібліотек [42;81] Слід зазначити, що Shadow DOM не оптимізує рендеринг сам по собі, а тільки вирішує проблему інкапсуляції, глибока вкладеність компонентів може ускладнювати оновлення та існує обмежена інтеграція з деякими фреймворками. Тому він корисний тільки у поєднанні з іншими архітектурами рендерингу.

Механізм інкапсуляції Shadow DOM забезпечує ізоляцію стилів та DOM-структури, створюючи логічний бар'єр між внутрішнім вмістом компонента та зовнішнім документом. Це досягається через створення окремого DOM-піддерева, яке прикріплюється до елемента-хоста (host element) через метод attachShadow(). Існує два режими Shadow DOM: відкритий (open) та закритий (closed). У відкритому режимі зовнішній код

може отримати доступ до Shadow DOM через властивість shadowRoot елемента-хоста, тоді як у закритому режимі доступ заборонено, що забезпечує повну інкапсуляцію. Вибір режиму впливає на можливість зовнішнього тестування, налагодження та розширення компонентів.

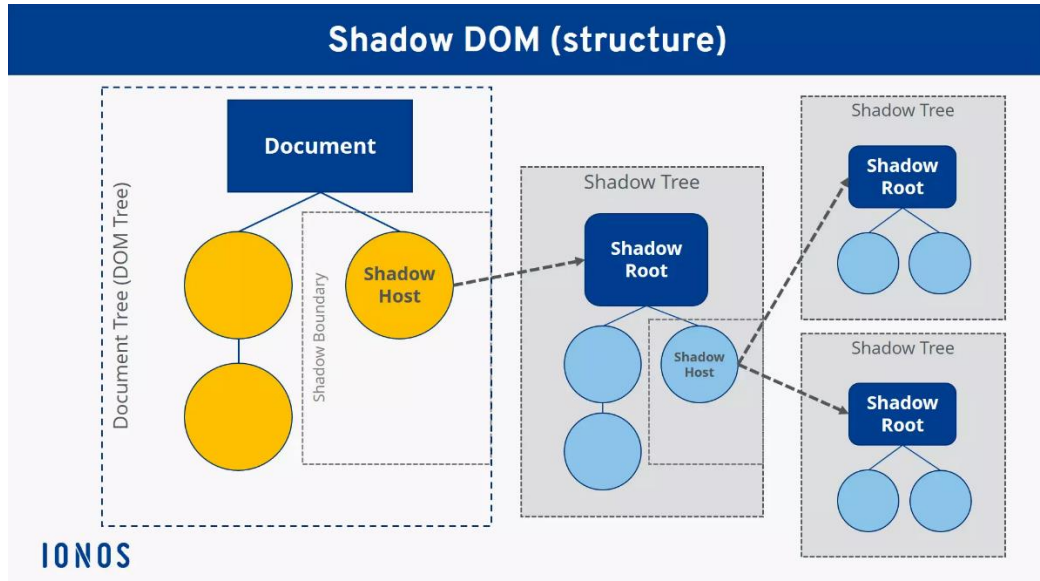


Рис.1.3. Структура Shadow DOM [73]

Одним із головних викликів при роботі з Shadow DOM є управління передачею даних між зовнішнім документом та інкапсульованим компонентом [87;90]. Оскільки Shadow DOM ізольований, стандартні методи передачі даних через DOM-події або глобальні змінні стають менш зручними. Web Components використовують атрибути та властивості елемента-хоста для передачі даних всередину компонента, а користувацькі події (Custom Events) для зворотного зв'язку. Проте, у випадку глибоко вкладених компонентів, де один Web Component містить інші Web Components, передача даних через кілька рівнів стає складною. Виникає необхідність реалізації механізмів, схожих на Context API у React, але без вбудованої підтримки у стандарті Web Components [77;80].

Продуктивність Shadow DOM у контексті глибоко вкладених структур визначається не стільки самим механізмом інкапсуляції, скільки способом оновлення внутрішнього вмісту компонентів. Якщо компонент використовує пряме DOM-маніпулювання для оновлення свого вмісту, продуктивність

залишається низькою. Тому на практиці Shadow DOM часто поєднується з іншими архітектурами рендерингу. Наприклад, бібліотека Lit (раніше LitElement) використовує Shadow DOM для інкапсуляції разом із системою шаблонів та реактивними властивостями, що забезпечують ефективне оновлення інтерфейсу. Інша бібліотека, Stencil, компілює Web Components із використанням Virtual DOM, поєднуючи переваги інкапсуляції та точкового оновлення інтерфейсу.

Ще однією важливою особливістю Shadow DOM є робота зі стилями. Стили, визначені всередині Shadow DOM, не впливають на зовнішній документ, і навпаки. Це вирішує проблему конфліктів імен класів та глобального забруднення CSS, але водночас створює складнощі при спробі стилізувати компоненти ззовні. Для управління стилізацією Shadow DOM підтримує механізм slotted content через елемент `<slot>`, що дозволяє вставляти зовнішній контент у компонент, а також використовує CSS-змінні (CSS Custom Properties), які можуть проникати через межі Shadow DOM. Це дозволяє зовнішнім стилям частково впливати на внутрішній вигляд компонента, зберігаючи при цьому інкапсуляцію основної структури.

Концепція Signals є моделлю fine-grained реактивності, де кожне значення є реактивним вузлом, що знає, хто від нього залежить [22;48]. При зміні сигналу оновлюються лише ті частини UI, які прямо залежать від цього значення. Signals формує граф залежності, де вузли є значеннями, в ребра – залежностями. Зміна вузла тригерить оновлення лише залежних обчислень, що дозволяє уникнути diffing та глобальних перерендерів, що забезпечує високу продуктивність при глибокій вкладеності. Використовуються у SolidJS, Angular 17+, Qwik. Ця модель потребує іншого мислення, де імперативна реактивність замінює декларативний diffing, що може бути складнішим. Існує ймовірність у складності з відстеженням залежностей у великих системах ну і не всі фреймворки підтримують signals нативно [47;48]. Тому підсумовуючи відзначаємо, що Signals є однією з найефективніших

сучасних архітектур для роботи з глибоко вкладеними структурами, оскільки оновлює лише мінімально необхідні вузли.

Архітектура Signals відрізняється від традиційних підходів тим, що кожне значення є окремим реактивним вузлом у графі залежностей [22;63]. На відміну від Virtual DOM, де порівнюються цілі компоненти або піддерева, Signals працює на рівні окремих значень та обчислень. Це досягається через механізм підписок, де кожне derived value (похідне значення) автоматично підписується на сигнали, від яких воно залежить. При зміні сигналу всі залежні обчислення виконуються заново, але лише ті, які реально залежать від зміненого значення. Це забезпечує максимально точкові оновлення та усуває необхідність у diffing.

Граф залежностей у Signals будується динамічно під час виконання коду. Коли створюється ефект (effect) або обчислюване значення (computed), фреймворк відстежує, до яких сигналів здійснюється доступ під час виконання функції ефекту або обчислення. Цей процес називається dependency tracking. У SolidJS, наприклад, dependency tracking реалізовано через глобальний контекст виконання: коли починається виконання ефекту, його функція додається до стека активних обчислень, і всі доступи до сигналів у цьому контексті автоматично реєструються як залежності. Після завершення виконання ефект підписується на всі зареєстровані сигнали. Це відбувається прозоро для розробника та не вимагає явного зазначення залежностей.

Однією з ключових переваг Signals є те, що оновлення відбуваються поза циклом рендерингу компонентів. У традиційних фреймворках, таких як React, зміна стану призводить до виклику рендер-функції компонента, яка повертає нове віртуальне дерево, після чого виконується diffing. У Signals рендер-функція викликається лише один раз під час ініціалізації компонента для встановлення зв'язків між сигналами та DOM-вузлами. Після цього всі оновлення інтерфейсу відбуваються через прямі маніпуляції DOM-вузлами, які прив'язані до змінених сигналів. Це усуває проміжний шар у вигляді Virtual DOM та значно зменшує накладні витрати.

Проте Signals має і свої складнощі. Головна з них – необхідність коректного управління підписками та уникнення циклічних залежностей. Якщо сигнал А залежить від сигналу В, а сигнал В залежить від сигналу А (прямо чи опосередковано), виникає безкінечний цикл оновлень. Сучасні реалізації Signals (SolidJS, Angular Signals) включають механізми виявлення таких циклів, але їх складно діагностувати, особливо у великих додатках з численними взаємозалежностями. Додатковою проблемою є те, що *dependency tracking* працює лише під час виконання коду, тому динамічні залежності (наприклад, коли доступ до сигналу відбувається за умовою) можуть не бути відстежені коректно, якщо умова не виконується під час початкового виконання ефекту.

Ще однією особливістю Signals є необхідність явного створення сигналів для реактивних значень [62;84]. На відміну від систем на основі Proху (як у Vue 3), де будь-який об'єкт автоматично стає реактивним, у Signals розробник повинен явно обгортати значення у `createSignal()` або аналогічну функцію. Це дає більше контролю, але вимагає дисципліни: забувши створити сигнал, розробник отримує нереактивне значення, зміни якого не викликають оновлень інтерфейсу. З іншого боку, явне створення сигналів дозволяє уникнути неявних підписок та зменшити кількість спостережуваних об'єктів, що покращує продуктивність у випадку роботи з великими структурами даних. Результати порівняльного аналізу внутрішньої архітектури рендерингу представлено у таблиці 1.1.

Reactive Rendering – це підхід, у якому інтерфейс автоматично оновлюється при зміні стану, а залежності між даними та UI відстежуються на рівні окремих виразів або компонентів [45;91]. Reactive Rendering реалізовано у SolidJS, Svelte, MobX, Vue 3 (Composition API). До переваг відносимо мінімальні оновлення DOM, відсутність глобального diffing, високу продуктивність у складних ієрархіях та природну підтримку глибокої реактивності. Слід відзначити, що Reactive Rendering має складнішу модель мислення для розробників, є ймовірність виникнення циклічних залежностей

та потребує ретельного проектування структури стану. Reactive Rendering це еволюція архітектур рендерингу, яка забезпечує найкращу продуктивність у сценаріях з глибокою вкладеністю.

Таблиця 1.1

Порівняльний аналіз моделей рендерингу та оновлення інтерфейсу

Архітектура	Продуктивність	Причина	Чи підходить для глибокої вкладеності?	Складність реалізації
DOM	Низька	Прямі операції над DOM	Ні	Проста модель
Virtual DOM	Середня	Diffing + батчинг	Частково	Середня складність
Signals	Висока	Точкові оновлення	Так	Складна реактивна модель
Reactive Rendering	Найвища	Fine-grained реактивність	Так	Залежить від фреймворку

Традиційні архітектури (DOM, Virtual DOM) не забезпечують достатньої продуктивності при роботі з глибоко вкладеними структурами. Сучасні реактивні підходи (Signals, Reactive Rendering) демонструють значно кращі результати завдяки точковому оновленню та відсутності глобального diffing. Це обґрунтовує необхідність розробки нових моделей і алгоритмів оптимізації рендерингу, що враховують структуру вкладених об'єктів.

Reactive Rendering як парадигма еволюціонував від простих observable-патернів до складних систем з автоматичним відстеженням залежностей. Історично, реактивне програмування у вебi почалося з бібліотек, таких як Knockout.js та Backbone.js, які впроваджували observable-властивості та двостороннє зв'язування даних (two-way data binding). Проте ці ранні реалізації мали значні обмеження: залежності потрібно було визначати вручну, а оновлення часто були неефективними через відсутність точкового контролю. Сучасні системи реактивного рендерингу, такі як MobX та Vue 3

Reactivity, усувають ці недоліки, використовуючи Proxy API для автоматичного перехоплення доступу до властивостей об'єктів та відстеження залежностей на рівні окремих полів [56;58].

Ключовою відмінністю сучасного Reactive Rendering від Virtual DOM є момент, коли визначається, які частини інтерфейсу потрібно оновити. У Virtual DOM це відбувається під час порівняння дерев (diffing phase), тобто після рендерингу компонента. У реактивних системах залежності визначаються під час виконання коду, що читає дані (tracking phase), і оновлення тригеруються одразу при зміні цих даних (reactive phase). Це усуває необхідність у порівнянні дерев, оскільки система точно знає, які частини UI залежать від змінених даних. Наприклад, у MobX при зміні observable-властивості автоматично викликаються reaction-функції, які підписані на цю властивість, а вони, у свою чергу, оновлюють відповідні DOM-вузли або компоненти.

Vue 3 Composition API демонструє елегантну реалізацію Reactive Rendering через функції `ref()` та `reactive()` [55;78]. Функція `ref()` створює реактивну обгортку для примітивного значення, тоді як `reactive()` перетворює об'єкт у реактивний проху. Під капотом Vue використовує Proxy для перехоплення операцій `get` та `set`, що дозволяє відстежувати доступ до властивостей та реєструвати залежності. Коли компонент рендериться, всі доступи до реактивних даних у шаблоні автоматично реєструються як залежності рендер-функції. При зміні даних Vue визначає, які компоненти залежать від змінених властивостей, та ставить їх у чергу на оновлення. Оновлення виконуються асинхронно у наступному мікротаску, що дозволяє об'єднати кілька змін в одне оновлення інтерфейсу.

Одним із викликів Reactive Rendering є управління складними залежностями у великих додатках [92;94]. У разі, коли одне реактивне значення залежить від багатьох інших, а ті, у свою чергу, залежать від ще більшої кількості значень, створюється глибокий граф залежностей. При зміні одного базового значення може тригеруватися каскад оновлень, які

поширюються по всьому графу. Для контролю таких ситуацій системи реактивності використовують механізми дедуплікації (dedupe), де кожне обчислення у графі виконується максимум один раз за один цикл оновлення, навіть якщо кілька його залежностей змінилися. Також важливим є порядок виконання оновлень: системи реактивності використовують топологічне сортування графа залежностей, щоб забезпечити коректну послідовність обчислень.

Reactive Rendering особливо ефективний у сценаріях з глибокою вкладеністю завдяки локальності оновлень. Якщо змінюється лише одне поле у глибоко вкладеному об'єкті, оновлюються лише ті частини UI, які безпосередньо залежать від цього поля, а не весь батьківський компонент чи піддерево. Це забезпечує майже константну складність оновлення $O(1)$ відносно розміру всього додатку, оскільки вартість оновлення пропорційна лише кількості залежних обчислень та DOM-операцій, а не загальній кількості компонентів чи елементів. Такий підхід робить реактивні системи особливо привабливими для додатків з великими формами, складними таблицями або динамічними дашбордами, де часті оновлення невеликих частин інтерфейсу є важливими для продуктивності та користувацького досвіду.

1.2. Аналіз методів та технологій оновлення інтерфейсу вебзастосунків

Питанням порівняльного аналізу стратегій оновлення інтерфейсу вебзастосунків присвячено роботи закордонних науковців з цієї тематики – В.Чалла [12], Т. Іскандар, М. Лубіс, Т. Кусумасарі та А. Лубіс [26], а також Ф. Павіч і Л. Бркіч [52]. Вони систематизують переваги й обмеження клієнтського та серверного підходів. Архітектурні аспекти серверного оновлення інтерфейсу вебзастосунків та його вплив на продуктивність корпоративних систем досліджують С. Женне [18], К. Валламсетла [75] та Х. Екпобімі [23]. Автори підкреслюють значущість SSR для SEO-оптимізації та масштабованості. Проблеми клієнтського оновлення інтерфейсу вебзастосунків у сучасних мережевих застосунках висвітлюють К. Насенок і

М. Войцеховська [46]. Гібридним підходам та інноваційним технологіям оновлення інтерфейсу вебзастосунків приділяють увагу Ф. Карвалью та П. Фіалю [10, 11], які досліджують прогресивний серверний рендеринг з асинхронними API, а також К. Чен [13], який аналізує модульний рендеринг та адаптивну гідратацію у React-застосунках. Порівняльні дослідження продуктивності сучасних вебфреймворків представлено у працях Л. Курапаті [38], Р. Олліла, Н. Мякітало та Т. Мікконен [49]. Однак досліджувана проблематика недостатньо висвітлено в сучасних українських наукових дослідженнях. Потребують подальшого системного дослідження питання комплексної оптимізації гібридних моделей оновлення інтерфейсу вебзастосунків [5;50], які поєднують переваги різних стратегій для забезпечення балансу між продуктивністю, масштабованістю та якістю користувацького досвіду у високонавантажених вебзастосунках [71;94].

Упродовж останніх років спостерігається стрімкий розвиток методів оновлення інтерфейсу вебзастосунків, які спрямовані на підвищення продуктивності, масштабованості та якості користувацького досвіду [89;104]. Сучасні вебзастосунки використовують різноманітні стратегії рендерингу: клієнтський рендеринг (CSR - Client-Side Rendering), серверний рендеринг (SSR - Server-Side Rendering), статичну генерацію сайтів (SSG - Static Site Generation), інкрементальну статичну регенерацію (ISR - Incremental Static Regeneration), прогресивну гідратацію, стрімінговий SSR та edge-side рендеринг (ESR) [8]. Кожен підхід має свої переваги та недоліки стосовно продуктивності, оптимізації для пошукових систем (SEO - Search Engine Optimization), масштабованості та складності впровадження.

Клієнтський рендеринг (CSR) – це підхід, який передбачає, що основна логіка рендерингу вебзастосунку виконується у браузері користувача, а не на сервері [105;106]. CSR забезпечує модульність, гнучкість і швидку розробку, використовуючи компонентну архітектуру та сучасні фреймворки (React, Vue, Angular). Він сприяє створенню застосунків на основі компонентів [46]. Важливою перевагою є висока інтерактивність. Динамічне оновлення

інтерфейсу без повного перезавантаження сторінки забезпечує плавний користувацький досвід. До того ж, ефективне управління станом досягається за допомогою сучасних підходів, таких як: Flux, Redux та реактивне програмування, що дозволяють централізовано керувати станом застосунку та підвищує його надійність і масштабованість [12;107].

Обмеження CSR полягають у складнощах з SEO та індексацією, тому що контент генерується в браузері користувача (на клієнті), рис.1.4. Це може призводити до некоректного відображення сторінок у пошукових системах. Додатковою проблемою є великі розміри файлів [110]. Для повноцінної роботи застосунку потрібно завантажувати значний обсяг JavaScript, що негативно впливає на продуктивність у повільних мережах або слабких пристроях та піднімає питання безпеки [108;109]. Також виникають ризики безпеки через експонування логіки застосунку та API в браузері, що підвищує вразливість до атак [114].

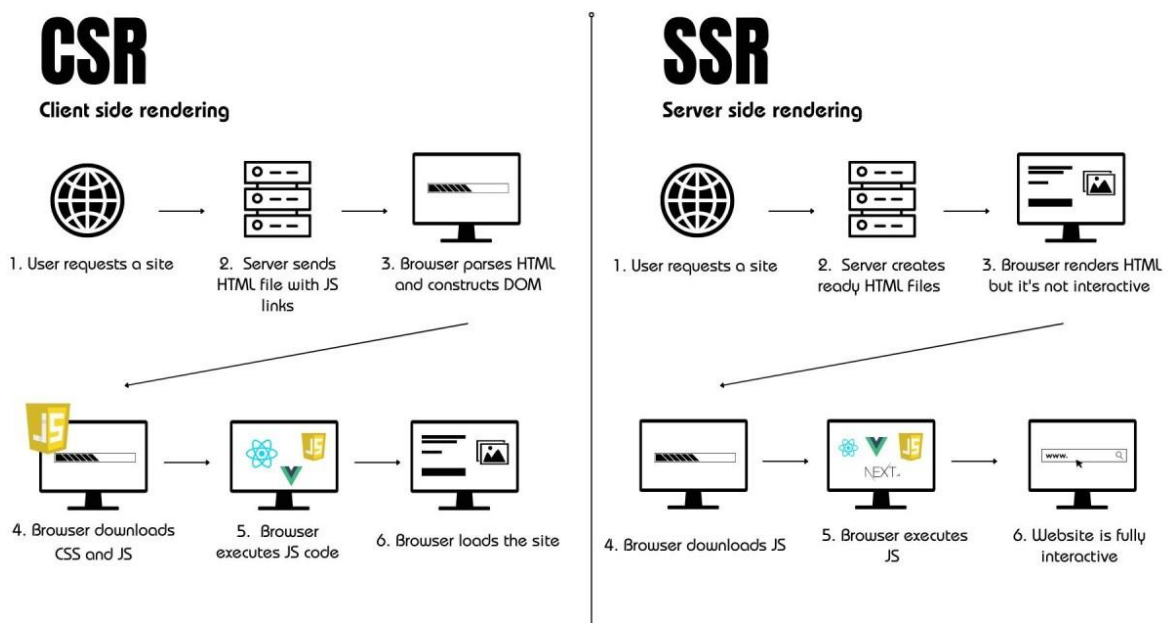


Рис.1.4. Концепція клієнтського (CSR) та серверного (SSR) рендерингу [79]

Серверний рендеринг (SSR) як ключова технологія для підвищення продуктивності, SEO та масштабованості сучасних вебзастосунків, насамперед для великих платформ і корпоративних рішень – це підхід, за якого HTML-код сторінки генерується на сервері, а не в браузері користувача [23;

38], рис.1.4. Його ключові переваги полягають у значному покращенні продуктивності, внаслідок того, що цей підхід зменшує час до першого відображення контенту (Time-to-First-Byte, First Contentful Paint) [111;112]. Також перевагою є SEO та індексація. Сервер одразу повертає повноцінний HTML, завдяки чому пошукові системи коректно індексують сторінки, водночас підвищують видимість сайту [74;76]. До того ж, SSR забезпечує масштабованість, у такий спосіб дозволяючи обслуговувати тисячі спільних користувачів, насамперед за умови використання сучасних фреймворків, таких як Next.js, Nuxt.js чи Angular. Сучасні технології SSR включають гібридні моделі, які поєднують цей підхід з інкрементальною статичною генерацією (ISR), статичною генерацією (SSG) та клієнтською інтерактивністю, рис.1.5 [113;116].

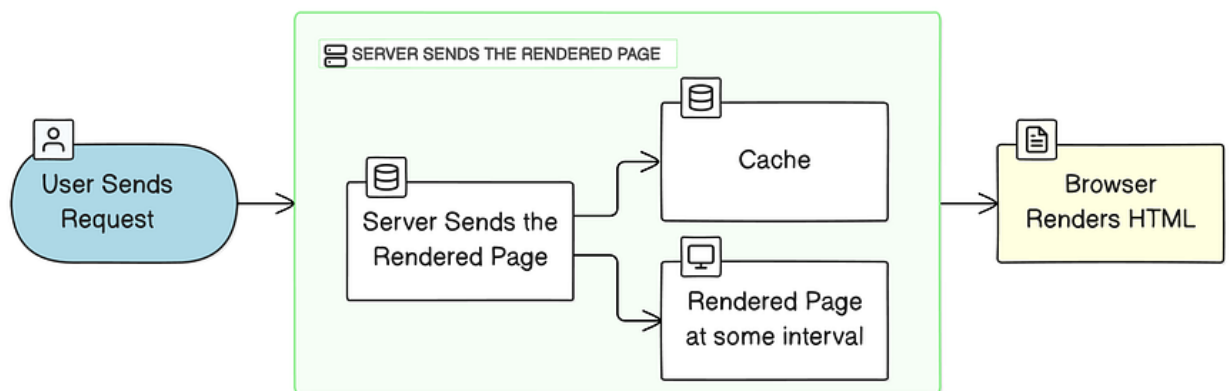


Рис. 1.5. Схема інкрементальної статичної регенерації (ISR) [36]

Це дозволяє збалансувати швидкість, динамічність і SEO [18; 23]. Також важливим напрямом досліджуваного питання є прогресивний SSR, який, завдяки асинхронним шаблонам і поступовому рендерингу, дозволяє відображати контент відповідно до надходження даних. У такий спосіб це покращує досвід користувачів, навіть у випадку складних запитів [10; 11]. Додатково використовується модульна гідратація, що забезпечує гнучке керування відновленням роботи компонентів залежно від пристрою, мережі та їхньої важливості, а також зменшуючи навантаження на клієнтів.

Водночас SSR має свої обмеження. Однією з основних проблем є складність синхронізації стану між сервером і клієнтом, яка потребує

ретельної реалізації, щоб уникнути розбіжностей під час гідратації [13; 75]. Крім того, спостерігаються високі витрати на серверні ресурси за великої кількості синхронних запитів, що потребує застосування кешування та CDN [23; 41]. Також існують потенційні ризики безпеки, якщо не дотримуватися сучасних практик захисту [53]. Впровадження SSR як ефективного підходу для створення швидких SEO-оптимізованих і масштабованих вебзастосунків, вимагає врахування складності синхронізації стану та оптимізації серверних ресурсів і безпеки, але сучасні фреймворки та архітектурні патерни значно спрощують ці завдання [3; 75].

Гібридні стратегії (ISR, SSG) поєднують переваги SSR і CSR – швидке завантаження статичних сторінок, можливість динамічного оновлення контенту та масштабованість. ISR дозволяє оновлювати сторінки на сервері без повної регенерації, а SSG – попередньо генерувати сторінки під час збірки. Це знижує навантаження на сервер і забезпечує високу продуктивність для великої кількості користувачів [12; 18], водночас поєднує швидкість статичної генерації (сторінки доставляються з CDN) із динамічністю серверного рендерингу (оновлення контенту на сервері), що гарантує високу продуктивність навіть для великих застосунків. Важливою перевагою також є оптимізація ресурсів, тому, що оновлюються лише змінені сторінки, що зменшує навантаження на сервер і дозволяє ефективно обробляти високий трафік [86; 40].

Однак впровадження ISR супроводжується низкою певних проблем. Зокрема, виникає складність налаштування механізмів оновлення, що необхідно для уникнення розбіжностей у даних [119]. Крім того, існує потреба у додаткових заходах безпеки через динамічне оновлення контенту [114]. Також слід враховувати витрати на інфраструктуру, які для великих проєктів вимагають оптимізації кешування та балансування. Таким чином, ISR у веброзробці постає як гнучкий та високопродуктивний метод рендерингу, який балансує між статичністю та динамічністю, а аналогічні гібридні підходи у суміжних сферах, зокрема в Image Super-Resolution, підтверджують

ефективність комбінування стратегій для досягнення оптимального співвідношення якості, швидкості й використання ресурсів [40; 86]. Гібридний метод рендерингу ISR – є сучасною стратегією, що забезпечує високу продуктивність, масштабованість і динамічність контенту.

Гібридний підхід до рендерингу SSG поєднує переваги попередньої генерації сторінок із динамічними або інтерактивними можливостями, що дозволяє створювати швидкі, масштабовані та сучасні вебзастосунки. Основні його принципи полягають у попередній генерації контенту. HTML-сторінки створюються під час збірки, що забезпечує миттєве завантаження та високу продуктивність. За такої умови гібридний підхід дозволяє додавати динамічні елементи за допомогою клієнтського JavaScript, API-запитів або інкрементального оновлення сторінок, зокрема через ISR [12]. Унаслідок використання компонентної архітектури та можливості поєднання SSG із SSR чи CSR досягається модульність і масштабованість, що створює гнучкі сценарії використання. Практична реалізація цього підходу забезпечує покращений користувацький досвід (основний контент відображається швидко, а динамічні частини підвантажуються асинхронно), завдяки цьому підвищується інтерактивність. Водночас попередньо згенерований HTML оптимізує SEO, а гібридні механізми дозволяють оновлювати контент без потреби у повній регенерації сайту. Це створює гнучкість для складних застосунків, де можна комбінувати SSG із SSR, ISR чи клієнтським рендерингом, адаптуючи методи до статичних, динамічних або персоналізованих сторінок [12]. Подібні гібридні підходи застосовуються і в суміжних галузях, зокрема у 3D-рендерингу [82].

Усі розглянуті підходи базуються на компонентній моделі, де дерево компонентів визначає структуру UI. У CSR дерево формується на стороні клієнта, в SSR – на сервері, в гібридних підходах – комбіновано. Управління станом (state management) доволі важливе для синхронізації між сервером і клієнтом, насамперед під час гідратації. З цією метою використовуються Redux, Context API, MobX, а також оптимізації для мінімізації повторного

оновлення інтерфейсу вебзастосунків і ефективної роботи з великими деревами компонентів [67; 72]. Відповідно, сучасні вебфреймворки базуються на компонентній моделі, де дерево компонентів визначає структуру користувацького інтерфейсу незалежно від конкретної технології реалізації. Загальною концепцією є проміжний шар абстракції між компонентами застосунку та браузерним DOM, що дозволяє оптимізувати оновлення інтерфейсу.

Алгоритм узгодження є важливим компонентом для фреймворків з Virtual DOM. Класичний алгоритм порівняння дерев має складність $O(n^3)$, що є неприйнятним для реальних застосунків. Сучасні фреймворки реалізують евристичні $O(n)$ алгоритми. Вони базуються на двох припущеннях – елементи різних типів генерують різні дерева, а розробники можуть надавати підказки через ключі (key prop) для ефективного відстеження елементів. Процес узгодження включає порівняння попереднього та нового дерева, ідентифікацію мінімального набору змін та застосування лише необхідних оновлень до реального DOM. React Fiber представляє еволюцію цього підходу. Він впроваджує інкрементальний рендеринг через розбиття роботи на малі одиниці, що дозволяє призупиняти, відновлювати та пріоритизувати оновлення, водночас підтримуючи дві версії дерева – current tree та work-in-progress tree [34]. Vue використовує подібну стратегію з реактивною системою на основі Proху, що забезпечує автоматичне відстеження залежностей [121].

У клієнтському рендерингу (CSR) дерево компонентів формується повністю в браузері після завантаження JavaScript [119;122]. Сервер надсилає мінімальний HTML-документ, браузер завантажує ресурси, фреймворк ініціалізується та будує дерево компонентів, динамічно проводячи рендер інтерфейсу. Користувацькі взаємодії викликають зміни стану компонентів, що призводить до часткового повторного рендерингу без перезавантаження сторінки. Управління станом у CSR досягається через спеціалізовані бібліотеки. Серверний рендеринг (SSR) вимагає складнішої архітектури через необхідність синхронізації між сервером та клієнтом. Код фреймворку

виконується на сервері, дерево компонентів рендериться у HTML з початковим станом, після чого сервер надсилає повністю сформований HTML браузеру. Коли клієнт отримує HTML разом з JavaScript, відбувається процес гідратації (hydration) – універсальна концепція для SSR-фреймворків (Next.js для React, Nuxt для Vue, SvelteKit для Svelte, Angular Universal) [121;124]. В межах процесу фреймворк реконструює своє внутрішнє представлення компонентів на стороні клієнта, прикріплює обробники подій та робить сторінку інтерактивною [17]. Гідратація вимагає узгодження між серверно-рендереним DOM та клієнтським представленням для виявлення розбіжностей. Великою проблемою є те, що вся сторінка залишається невідгукуною до завершення гідратації, внаслідок того, що інтерактивні елементи не можуть реагувати на взаємодії до прикріплення обробників подій. Vue автоматично намагається відновити та скоригувати pre-rendered DOM відповідно до клієнтського стану під час виявлення невідповідностей, хоча це призводить до втрат продуктивності [59].

Синхронізація стану між сервером та клієнтом є фундаментальною проблемою SSR незалежно від фреймворку. Дані, які отримано на сервері для початкового рендерингу, мають бути доступні на стороні клієнта для гідратації без повторного завантаження. Загальноприйнятий підхід передбачає серіалізацію стану у глобальну змінну (`window.INITIAL_STATE`, `window.NUXT`, тощо) в HTML-рядку, яку клієнт використовує для ініціалізації сховища управління станом. Сучасні фреймворки впроваджують революційні можливості для SSR. Зокрема, Streaming HTML дозволяє починати відправку HTML до завершення завантаження всіх даних, розбиваючи HTML на фрагменти для прогресивного рендерингу браузером, що забезпечує швидший First Contentful Paint [66]. Selective Hydration надає пріоритет гідратації частинам, з якими взаємодіє користувач, за такої умови створюючи ілюзію миттєвої гідратації. React 18 реалізує це через Suspense, Next.js 15 використовує Server Components з автоматичним streaming, а Vue 3 впроваджує async components з подібною функціональністю [54].

Гібридні підходи (ISR, SSG) поєднують переваги різних стратегій рендерингу та підтримуються усіма основними meta-фреймворками. Прогресивна та селективна гідратація представляють новий етап еволюції вебрендерингу [1]. Так, Islands Architecture є framework-agnostic підходом, який розбиває сторінку на незалежні «острівці» інтерактивності, що оточені статичним HTML. Astro підтримує React, Vue, Svelte та Solid синхронно з директивами `client:visible`, `client:idle`, `client:load` для контролю гідратації. Fresh використовує Preact з islands-first підходом, Marko впроваджує власну реалізацію, а Eleventy інтегрує islands через web components [27; 28]. На відміну від монолітних JavaScript-бандлів, «острівці» завантажуються паралельно та гідруються ізольовано, при цьому компоненти з низьким пріоритетом не блокують критичні. Прогресивна гідратація фокусується на важливих компонентах, рендерує статичний контент на сервері та гідрує лише інтерактивні елементи post-load [16].

React Server Components (RSC) представляють гібридний підхід, який поєднує серверо-центричну модель SSR з клієнто-центричною моделлю CSR. Він дозволяє створювати змішане дерево компонентів з серверними та клієнтськими компонентами. Цей підхід забезпечує виконання частини логіки та рендерингу безпосередньо на сервері, що суттєво зменшує навантаження на клієнтське середовище та прискорює завантаження застосунку. Практичне застосування RSC демонструє його ефективність у системах із високими вимогами до динамічності та персоналізації інтерфейсу. Водночас RSC вимагає ретельного розподілу логіки між серверними та клієнтськими компонентами, який вимагає від розробників глибокого розуміння архітектурних особливостей гібридного рендерингу та механізмів синхронізації стану [51]. Узагальнену схему процесу гідратації вебзастосунків (Hydration) зображено на рисунку 1.6.

SSR, насамперед з використанням сучасних фреймворків (наприклад, Next.js), демонструє значні переваги для високонавантажених застосунків. Спостерігається покращення Time-to-First-Byte, зниження bounce rate та

підвищення SEO [26; 38]. CSR забезпечує гнучкість, модульність і швидкість розробки, але може призводити до збільшення розміру файлів, проблем із SEO та продуктивністю на слабких пристроях [26; 52]. Гібридні та інкрементальні підходи дозволяють поєднувати переваги SSR і CSR.

У питанні, яке пов'язано із конкретним вибором фреймворків та архітектур, слід зазначити, що популярні фреймворки (React, Angular, Vue, Svelte, Blazor) мають різні стратегії рендерингу, що впливають на масштабованість і продуктивність. Найкращі результати досягаються за рахунок компіляційних оптимізацій та реактивних моделей [49; 52]. Для великих корпоративних систем актуальні архітектури з динамічною інклюзією віддалених компонентів (composable renderer services) [57].

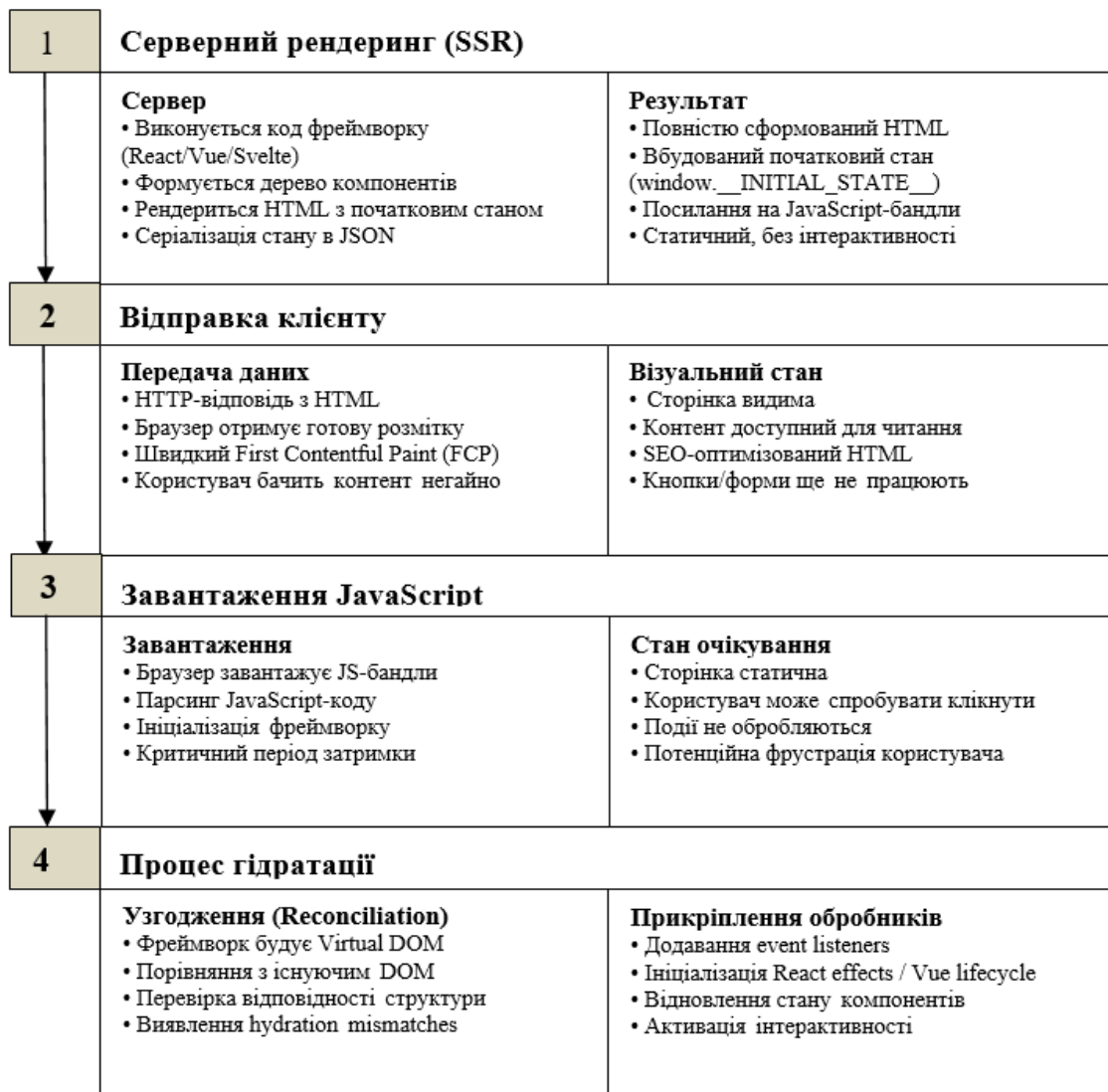


Рис. 1.6. Узагальнена схема процесу гідратації (Hydration) вебзастосунків

Вплив кожного підходу оновлення інтерфейсу вебзастосунків на роботу з деревоподібними структурами даних та процес їхнього оновлення, який дозволяє визначити ключові відмінності між стратегіями та обґрунтувати вибір оптимального методу залежно від специфіки застосування узагальнено в порівняльній таблиці 1.2.

Таблиця 1.2.

Порівняння сучасних підходів оновлення інтерфейсу вебзастосунків

Підхід	Характер формування дерева	Швидкість початкового відображення	Локальне оновлення вузлів	Вплив на SEO	Навантаження на сервер	Складність оновлення даних
CSR	Створення дерева повністю на клієнті після завантаження JS	Низька при великому дереві	Висока, можливе точкове оновлення	Низький без пререндеру	Низьке	Легке оновлення локальних вузлів
SSR	Створення дерева на сервері та гідратація на клієнті	Висока	Середня, часто потребує додаткового CSR	Високий	Високе при великій кількості запитів	Оновлення через повторний SSR або CSR
SSG	Створення дерева під час збірки	Дуже висока	Низька без CSR-довантаження	Високий	Низьке	Потрібна збірка або CSR-довантаження
ISR	Попереднє дерево та регенерація за розкладом/подією	Дуже висока	Середня, можливі часткові оновлення	Високий	Низьке/середнє	Автоматична або запитувана інвалідація
RSC / Islands	Часткове дерево на сервері та клієнтські інтерактивні фрагменти	Висока	Висока в інтерактивних частинах	Високий	Середнє	Точкове оновлення окремих сегментів

Проведене дослідження дозволяє зробити висновки, що ефективність вебплатформ безпосередньо залежить від обраної стратегії відображення

контенту та управління даними. Жоден із відомих підходів не може бути визнаний універсальним рішенням для всіх типів застосунків.

Найбільший потенціал для створення високопродуктивних вебзастосунків демонструють гібридні стратегії, зокрема статична генерація сайтів та інкрементальна статична регенерація. Особливу увагу приділено ролі компонентної архітектури та механізмів управління станом у забезпеченні узгодженості між різними рівнями рендерингу. Правильна організація дерева компонентів і вибір відповідних інструментів управління станом визначають загальну ефективність застосунку та якість користувацького досвіду. Подальший розвиток досліджень пов'язаний з поглибленим вивченням гібридних моделей рендерингу та розробкою методик автоматизованого вибору оптимальної стратегії залежно від специфіки конкретного застосунку. Важливим завданням залишається розробка комплексних методик оцінювання та прогнозування продуктивності різних підходів рендерингу на етапі проєктування інформаційних систем, що дозволить обґрунтовано приймати архітектурні рішення для корпоративних платформ і високонавантажених вебсервісів.

1.3. Аналіз проблем продуктивності при оновленні інтерфейсу вебзастосунків з глибокою вкладеністю структур

Сучасні вебзастосунки дедалі частіше працюють із великими та складними структурами даних, що мають значну глибину вкладеності. Такі структури виникають у системах управління контентом, аналітичних панелях, конфігураторах, редакторах, інструментах візуалізації, а також у застосунках, що оперують деревами, графами або вкладеними JSON-об'єктами. Глибока вкладеність створює низку специфічних проблем продуктивності, які суттєво впливають на швидкодію інтерфейсу, стабільність роботи та загальний користувацький досвід.

Однією з ключових проблем є збільшення обчислювальної складності операцій оновлення інтерфейсу вебзастосунків. У традиційних архітектурах,

що використовують Virtual DOM, оновлення інтерфейсу передбачає порівняння двох дерев станів. При глибокій вкладеності кількість вузлів, що беруть участь у diffing-процесі, зростає експоненційно. Це призводить до збільшення часу обробки змін, навіть якщо фактичні оновлення стосуються лише невеликої частини дерева.

Глибока вкладеність часто супроводжується складними залежностями між вузлами. У фреймворках із декларативною моделлю компонентів (React, Vue) зміна стану на верхньому рівні може спричинити повторне оновлення інтерфейсу вебзастосунків усіх дочірніх компонентів, навіть якщо їх стан не змінився. Це явище відоме як каскадні перерендери. У великих деревоподібних структурах каскадні оновлення можуть охоплювати сотні компонентів, що призводить до збільшення часу рендерингу, зростання навантаження на процесор, зниження плавності анімацій, погіршення реакції інтерфейсу на дії користувача.

У реактивних архітектурах (Vue 3, MobX, Angular Signals) продуктивність залежить від точності визначення залежностей між даними та елементами інтерфейсу [64;77]. При глибокій вкладеності зростає кількість реактивних вузлів, збільшується кількість зв'язків між ними, ускладнюється визначення мінімального набору елементів, які потребують оновлення [95;99]. Неправильне або надмірне відстеження залежностей може спричинити надлишкові реакції, коли оновлення одного вузла запускає ланцюг непотрібних оновлень у всьому дереві.

Навіть за наявності оптимізацій, будь-які зміни у глибоко вкладених структурах неминуче призводять до операцій над DOM. Оскільки DOM є важкою структурою, операції над ним мають високу вартість, що обумовлене необхідністю перерахунку стилів, переформатуванням макета, перерисовкою елементів. При глибокій вкладеності навіть локальні зміни можуть спричинити глобальні перерахунки, що значно збільшує час оновлення інтерфейсу вебзастосунків [100].

Глибоко вкладені структури часто мають нерівномірний розподіл змін: одні частини дерева оновлюються рідко, інші – дуже часто. Це створює труднощі для прогнозування продуктивності, вибору оптимальних стратегій оновлення, забезпечення стабільної частоти кадрів [97;103]. У складних інтерфейсах це може призвести до «стрибків» продуктивності, коли система працює плавно у більшості випадків, але різко деградує при певних сценаріях.

DOM, Virtual DOM та навіть Shadow DOM мають фундаментальні обмеження при роботі з глибокими структурами [45]. Так DOM надто повільний для частих оновлень, Virtual DOM потребує порівняння великих дерев, Shadow DOM вирішує інкапсуляцію, але не продуктивність. Ці обмеження створюють потребу у нових підходах, які мінімізують кількість операцій над DOM, забезпечують точкове оновлення, враховують структуру вкладених об'єктів, адаптуються до характеру змін.

Метод рекурсії посідає ключову роль у вирішенні питань побудови складного користувацького інтерфейсу [95;98]. При правильному налагодженні рекурсивної функції можливо досягнути бажаного результату без допомоги циклів та купи дублюючого коду. Рекурсії допомагають вирішити не тільки питання обчислень та пошуку значень, але й побудови макету додатку. Гарним прикладом слугують вкладені списки, які мають однакову структуру, стилі та правила поведінки, але різні значення та розміщення [6;83]. У прикладі зі списками метод рекурсії вирішить поставлену задачу, з іншої сторони, якщо ціллю відмалювання є не прості елементи, а складні компоненти взаємодії користувача, то метод може нашкодити продуктивності додатку.

Проблема рекурсії полягає у виділенні великої частин ресурсів на обчислення, що може призвести до візуальних дефектів, таких як: заморожування процесу рендерингу, перенавантаження стеку операцій, довга відповідь додатку на користувацькі дії [98]. Для вирішення проблеми методу було проаналізовано низку рішень, кожен з яких пропонує механізми покращення продуктивності за рахунок виділення пам'яті на додаткові операції [89;104].

Принцип мемоізації полягає в збереженні результатів обчислень функції для уникнення повторних обчислень з тими ж вхідними даними [19]. Це особливо корисно для рекурсивних функцій, які обчислюють однакові підзадачі багаторазово:

```
const memo = {};
function fibonacci(n) {
  if (n in memo) {
    return memo[n];
  }
  if (n <= 1) {
    return n;
  }
  memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
  return memo[n];
}
```

В даному прикладі результат обчислення однієї ітерації числа Фібоначі не втрачається після обчислень, а зберігається у пам'яті, щоб при повторному виклику цієї функції спочатку перевірити наявність готового результату. Цей метод значно зменшує кількість рекурсивних викликів і покращує продуктивність за рахунок уникнення повторних обчислень.

Даний метод також має свої недоліки, адже система має виділяти додаткову пам'ять, а також підтримка таких частин пам'яті може бути ускладнена великою кількістю параметрів які треба зберегти для перевірки. Мемоізація є одним із ключових механізмів оптимізації рендерингу [19;96]. Проте при глибокій вкладеності складно визначити, які частини дерева можна кешувати, зміни на верхніх рівнях часто інвалідовують кеш нижніх рівнів, глибокі структури потребують складних стратегій порівняння (shallow vs deep compare). У результаті мемоізація або не дає очікуваного ефекту, або створює додаткові накладні витрати.

Багато рекурсивних алгоритмів можна переписати в ітеративній формі за допомогою циклів і стеків:

```
function factorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

```
function factorialIterative(n) {
  let acc = 1;
  while (n > 0) {
    acc *= n;
    n--;
  }
  return acc;
}
```

Це дозволяє уникнути переповнення стеку викликів і покращує використання пам'яті. В прикладі, що представлено вище, рекурсія факторіала замінюється ітеративним підходом в якому обрахунки відбуваються в середині циклу. Такий підхід дозволяє уникнути проблеми з переповненням стеку, а також є більш швидким варіантом обчислення, оскільки не потребує викликів функцій.

«Розділяй і володарюй» – це техніка, яка дозволяє паралелізувати рекурсивні обчислення, використовуючи Web Workers для розподілу навантаження між потоками:

```
// Основний потік
const worker = new Worker('fibonacciWorker.js');
worker.postMessage(n);
worker.onmessage = function(event) {
  console.log(event.data); // Отримуємо результат
};
// Веб-працівник (fibonacciWorker.js)
self.onmessage = function(event) {
  const n = event.data;
  function fibonacci(n) {
    if (n <= 1) {
      return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
  self.postMessage(fibonacci(n));
};
```

Це особливо корисно для великих обчислень, що можуть блокувати основний потік. Основними перевагами цього методу є зниження навантаження на основний потік та можливість обробляти великі обсяги даних паралельно. Даний метод складний у реалізації та налагодженні, адже треба додаткові витрати на передачу даних між потоками.

Для довгих рекурсивних обчислень, які можуть блокувати основний потік, можна використовувати `setTimeout` для розбивки задачі на частини [95].

```
function factorialAsync(n, callback) {
  function helper(n, acc) {
    if (n === 0) {
      callback(acc);
    } else {
      setTimeout(() => helper(n - 1, n * acc), 0);
    }
  }
  helper(n, 1);
}
```

Це дозволяє іншим завданням виконуватися між частинами обчислень, зберігаючи інтерфейс користувача чутливим. Розуміючи відмінність синхронних та асинхронних операцій можна перенаправити обчислення в стек асинхронних операцій. Так як `setTimeout` змусить функцію перейти в стек асинхронних операцій, факторіал числа почне виконуватись на фоні, даючи можливість іншим синхронним операціям отримати пріоритет виконання.

Проблеми продуктивності при оновленні інтерфейсу вебзастосунків глибоко вкладених структур є комплексними та багатовимірними. Вони охоплюють як обчислювальні аспекти (складність `diffing`, каскадні оновлення, реактивність), так і архітектурні (обмеження DOM, Virtual DOM, Shadow DOM). Аналіз показує, що традиційні підходи не забезпечують достатньої ефективності для сучасних високонавантажених вебзастосунків. Це обґрунтовує необхідність розробки нових моделей, методів і алгоритмів оптимізації оновлення інтерфейсу вебзастосунків, здатних ефективно працювати з глибоко вкладеними структурами даних.

Для детального дослідження можливості оптимізації промалювання вебзастосунку було проаналізовано сайт замовлення виготовлення тривимірних моделей, а саме, приватний кабінет користувача у якому є можливість налаштовувати параметри виготовлення моделі та налаштування для обладнання (рис. 1.7) [4;21].

Конфігурація металу

Назва	Пластик
<Слайдер> Щільність	0,250

Тестова група залежностей видимості

Верхній шар

<Залежність видимості> Верхній шар

<Число> Кількість верхніх шарів	2
☐ <Перемикач> Увімкнення верхнього шару	
<Число> верхній відступ шару	0,4 mm

<Вибір> Лазер

☒ <Вибірка елементу> Одиночний лазер

<Число> Одиночний лазер T0

2

☐ <Вибірка> Подвійний лазер

<Число> Подвійний лазер T0

3

<Число> Подвійний лазер T1

3

Рис. 1.7. Сторінка налаштування обладнання

У даному прикладі, рис. 1.8, можна спостерігати деревовидну структуру фронтенд архітектури, яка підтверджується відповідним запитам від серверної частини.

```
▼ {id: 'BbQxos9TXpeBTP0NNKQp7', type: 'folder', Label: 'Налаштування', dependency: undefined, childs: Array(6)} ⓘ
  ▼ childs: Array(6)
    ▼ 0:
      defaultValue: ""
      dependency: undefined
      id: "name"
      label: "Ім'я адміна"
      type: "text"
      ▶ [[Prototype]]: Object
    ▼ 1:
      defaultValue: ""
      dependency: Array(1)
      ▼ 0:
        rule: Array(1)
        ▼ 0:
          operator: "=="
          result: "hide"
          value: "1"
          ▶ [[Prototype]]: Object
          length: 1
          ▶ [[Prototype]]: Array(0)
          watchedId: "name"
          ▶ [[Prototype]]: Object
          length: 1
          ▶ [[Prototype]]: Array(0)
          id: "second-name"
          label: "Фамілія адміна"
          type: "text"
          ▶ [[Prototype]]: Object
        ▶ 2: {id: 'age', type: 'number', label: 'Вік адміна', dependency: undefined, defaultValue: undefined}
        ▶ 3: {id: 'additional-toggle', type: 'toggle', label: 'Активувати додаткові налаштування', dependency: undefined, defaultValue: true}
        ▶ 4: {id: 'additional-folder', type: 'folder', label: 'Додаткові налаштування', dependency: Array(1), childs: Array(3)}
        ▶ 5: {id: 'users', type: 'table', label: 'Користувачі', dependency: undefined, columns: Array(7)}
          length: 6
          ▶ [[Prototype]]: Array(0)
          dependency: undefined
          id: "BbQxos9TXpeBTP0NNKQp7"
          label: "Налаштування"
          type: "folder"
      ▶ [[Prototype]]: Object
```

Рис. 1.8. Вигляд структури даних отримана від серверної частини вебзастосунку

Маючи готовий приклад, можна переходити до етапу аналізу продуктивності вебзастосунку. Основним інструментом аналізу буде слугувати вбудований в браузер Google Chrome – Performance insight. Після проведення аналізу можна помітити велику кількість рендерів компонентів та об’ємні процеси по калькуляції та промальовуванню цих компонентів, що так само свідчить про неоптимізованість системи (рис. 1.9) [95;96].

Додатковим і найпростішим способом перевірки кількості оновлень інтерфейсу вебзастосунків є виставлення точок повідомлення, однак для цього необхідно знайти в ресурсі сайту код, який відповідає за оновлення інтерфейсу кінцевих гілок віртуального DOM-дерева.

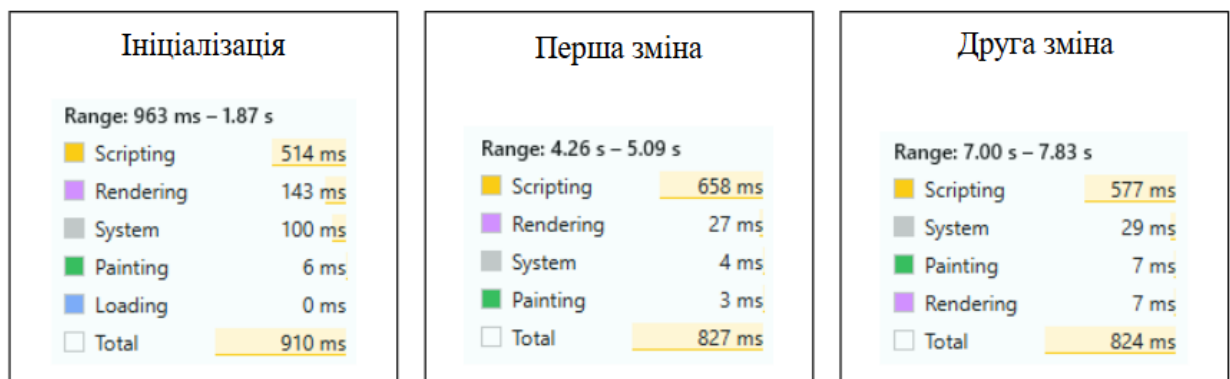


Рис. 1.9. Загальний аналіз продуктивності сторінки

Додатковим і найпростішим способом перевірки кількості оновлень інтерфейсу вебзастосунків є виставлення точок повідомлення, однак для цього необхідно знайти в ресурсі сайту код, який відповідає за оновлення інтерфейсу кінцевих гілок віртуального DOM-дерева. У підсумку (рис. 1.10) можна зробити висновок, що кількість оновлень інтерфейсу вебзастосунків для рекурсивного відмалювання усіх компонентів є досить високою, водночас збільшення кількості даних на серверній частині може призвести до значних втрат продуктивності додатку.

Головним елементом аналізу є перегляд рекурсивного методу промальовування всередині коду та усіх дочірніх елементів, які використовуються в середині функції.

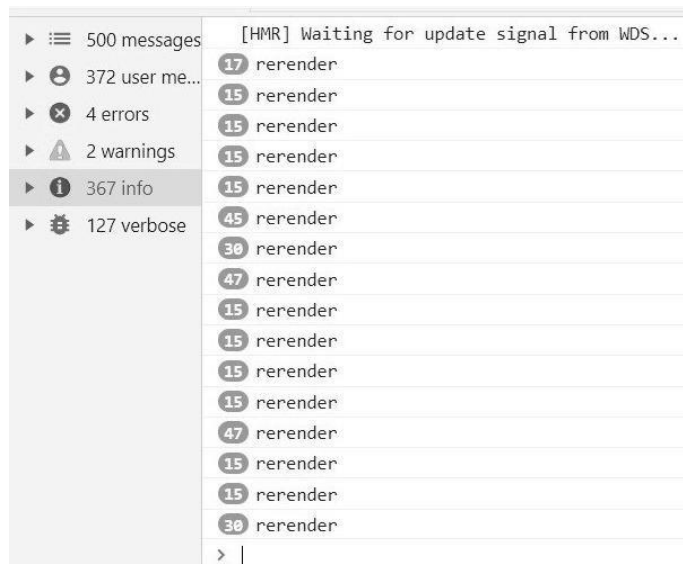


Рис. 1.10. Аналіз продуктивності за допомогою консолі браузера

На рисунку 1.11 авторами репрезентовано фрагмент програмного коду з критичними помилками в розробці, які сприяють втраті продуктивності. Сюди слід віднести відсутність умов перевірки стану вхідних значень (React.memo), відсутність кешування калькуляції і визначення функцій (useMemo, useCallback), асинхронність зв'язаних полів, велику кількість вхідних параметрів, неправильне використання ключового атрибута key, навантаження компонентів бізнес логікою (відсутність розбиття на розумні і прості компоненти), наявність «тяжких» змінних, які використовують тільки частину своїх значень та недоречна деструктуризація вхідних параметрів.

Наявність низки таких аргументів є приводом для створення методу оптимізації з покращення продуктивності додатку, шляхом змінення коду та застосування загальних практик і патернів розробки програмного продукту. Основними критеріями оцінки ефективності методу оптимізації є час промалювання веб елементів та швидкість реакції на інтеракцію користувача, що досягається зменшенням кількості оновлень інтерфейсу вебзастосунків на сторінці. Вона також мінімізує кількість та розміри стовпців під час перегляду продуктивності сторінки за допомогою інструменту Performance insight. Метод оптимізації промалювання дозволяє уникнути проблем стосовно

продуктивності додатка на етапі відмалювання багаторівневих структур даних та під час їх використання користувачем.

```
const Folder: React.FC<FolderProps> = ({
  folder, values,
  touched, errorsList,
  creation, duplication,
  level, urgentFormValidation,
  importantValidation,
}) => {
  return (
    {folder.entries.map((nestedFolder: any) => {
      if (nestedFolder.hasOwnProperty('folder')) {
        return (
          <Folder
            key={JSON.stringify(nestedFolder)}
            folder={nestedFolder}
            values={values}
            touched={touched}
            errorsList={errorsList}
            parentFolder={{folder as any}.entries}
            creation={creation}
            duplication={duplication}
            level={level + 1}
            urgentFormValidation={urgentFormValidation}
            importantValidation={importantValidation}
          />
        );
      }
    })}
  );
};

export const RenderNode = ({folder}) => {
  switch (folder.type) {
    case 'table':
      return (...);
    case 'radio':
      return (...);
    case 'data':
      return (...);
    case 'select':
      return (...);
    case 'changer':
      return (...);
    case 'checkbox':
      return (...);
    case 'number_node':
      return (...);
    case 'integer_node':
      return (...);
    case 'multi':
      return (...);
    case 'text_node':
    default:
      return (...);
  }
};
```

Рис. 1.11. Фрагмент коду з рекурсивним відмальовуванням кінцевих гілок

Детальний аналіз дозволив нам виявити низку проблем стосовно продуктивності системи. Шляхом використання інструментів розробника та перегляду кодової частини було знайдено вразливості системи, які без додаткового рефакторінгу зможуть ускладнити процес налагодження для розробника та унеможливлювати роботу з додатком для користувача. Підтримка такого коду може збільшити час на пошук та виправлення помилок, а клієнтська частина з легкістю відштовхне вибагливих користувачів.

Висновки до першого розділу

Здійснено системний аналіз сучасних архітектур оновлень інтерфейсу вебзастосунків, досліджено проблеми продуктивності, що виникають при роботі з глибоко вкладеними структурами даних, а також розглянуто наявні підходи до оптимізації процесів оновлення інтерфейсу.

Аналіз архітектур DOM, Virtual DOM, Shadow DOM, Signals та Reactive Rendering показав, що традиційні моделі оновлення інтерфейсу вебзастосунків не забезпечують достатньої ефективності при роботі з великими ієрархічними структурами. Virtual DOM, попри свою популярність, має фундаментальні обмеження, пов'язані з необхідністю порівняння великих дерев станів, що призводить до зростання обчислювальних витрат. Shadow DOM вирішує проблему інкапсуляції, але не оптимізує процес оновлення інтерфейсу. Натомість сучасні реактивні підходи, зокрема signals та fine-grained реактивність, демонструють значно вищу продуктивність завдяки точковому оновленню окремих вузлів.

Дослідження проблем продуктивності показало, що глибока вкладеність структур даних створює специфічні виклики, які не можуть бути ефективно вирішені традиційними методами. Серед ключових проблем – експоненційне зростання складності diffing, каскадні перерендери, труднощі відстеження залежностей, висока вартість DOM-операцій та обмежена ефективність мемоізації. Ці фактори призводять до нестабільності продуктивності, затримок у відображенні інтерфейсу та зниження якості користувацького досвіду.

Огляд наявних підходів до оптимізації оновлення інтерфейсу вебзастосунків показав, що хоча сучасні фреймворки пропонують широкий спектр технік – від батчингу оновлень і мемоізації до compile-time оптимізації та віртуалізації – жоден із них не забезпечує універсального рішення для роботи з глибоко вкладеними структурами. Більшість методів або потребують

значних ручних налаштувань, або не масштабуються для великих дерев, або не враховують динамічний характер змін у складних структурах даних.

Таким чином, результати дослідження дозволяють сформулювати наукову проблему, що полягає у відсутності моделей, методів та алгоритмів, спеціально орієнтованих на оптимізацію оновлення інтерфейсу вебзастосунків, які працюють з глибоко вкладеними структурами даних та багатозалежними зв'язками. Виявлені обмеження наявних підходів обґрунтовують необхідність розробки нових рішень, здатних забезпечити точкове оновлення інтерфейсу, мінімізацію обчислювальних витрат та стабільну продуктивність у високонавантажених системах.

Результати проведеного аналізу та його результатів відображено у роботах [95;96;100] здобувача.

РОЗДІЛ 2. Формалізація моделей представлення та оптимізації оновлення інтерфейсу вебзастосунків з глибокою вкладеністю

2.1. Формальна модель точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю

Проведемо розробку формальної моделі точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю. Дана модель формалізує структуру даних, процес оновлення DOM та механізм мінімізації кількості DOM-операцій та буде теоретичним підґрунтям для розробки методу точкового оновлення інтерфейсу вебзастосунків.

Глибоко вкладені структури даних, що використовуються у сучасних вебзастосунках, можуть бути представлені у вигляді дерев або орієнтованих ациклічних графів. Для побудови моделей оптимізації оновлення інтерфейсу вебзастосунків необхідно формалізувати ці структури таким чином, щоб вони відображали як статичні властивості даних, так і динаміку їх змін.

Нехай структура даних S є впорядкованим деревом, модель якого можна представити у вигляді (2.1):

$$S = (V, E, r), \quad (2.1)$$

де V – множина вузлів; E – множина ребер, $E \subseteq V \times V$; r – кореневий вузол, $r \in V$.

Кожен вузол $v \in V$ має атрибути, вигляд (2.2):

$$v = (id, type, value, children), \quad (2.2)$$

де id – унікальний ідентифікатор; $type$ – тип вузла (об'єкт, масив, примітив, компонент тощо); $value$ – значення або набір властивостей; $children$ – множина дочірніх вузлів, $children \subseteq V$.

Глибина дерева визначається за формулою (2.3):

$$depth(S) = \max_{v \in V} dist(r, v), \quad (2.3)$$

де $dist(r, v)$ – довжина шляху від кореня до вузла.

Модель оновлення інтерфейсу вебзастосунків визначимо як відображення дерева у DOM вигляду (2.4):

$$R: V \rightarrow D, \quad (2.4)$$

де D – множина DOM-вузлів, $R(v)$ – DOM-представлення вузла v .

Таким чином, оновлення інтерфейсу вебзастосунків структури даних зводиться до побудови дерева DOM-вузлів, ізоморфного або частково ізоморфного дерева даних. Зміни у структурі даних визначимо як набір операцій, «вставити», «видалити», «оновити», «змінити порядок», що матиме вигляд (2.5):

$$\Delta = \{insert, delete, update, reorder\} \quad (2.5)$$

Формальна зміна – це відображення вигляду (2.6):

$$\delta: S_t \rightarrow S_{t+1}, \quad (2.6)$$

де S_t – стан структури у момент часу t ; δ – зміна, що застосовується до структури.

Кожна зміна має локальну область впливу змін, (2.7):

$$scope(\delta) \subseteq V \quad (2.7)$$

У випадку глибокої вкладеності, (2.8):

$$|scope(\delta)| \ll |V| \quad (2.8)$$

Але традиційні алгоритми оновлення інтерфейсу вебзастосунків часто обробляють значно більшу частину дерева, що призводить до надмірних обчислень. Для ефективного оновлення інтерфейсу вебзастосунків необхідно визначити залежності між вузлами:

$$Dep = \{(v_i, v_j) | v_j \rightarrow v_i\} \quad (2.9)$$

У традиційних фреймворках залежності визначаються на рівні компонентів, тоді як у реактивних – на рівні окремих значень. У випадку глибокої вкладеності, (2.10) що створює значні витрати при оновленні:

$$|Dep| \rightarrow O(|V|^2) \quad (2.10)$$

Оновлення інтерфейсу визначимо як функцію (2.11):

$$U: (S_t, \delta) \rightarrow D_{t+1}, \quad (2.11)$$

де U – функція оновлення DOM, D_t – стан DOM у момент часу t .

Мета оптимізації заключається в тому, щоб мінімізувати DOM-операцій, необхідні для відображення змін (2.12):

$$\min |U(S_t, \delta)|, \quad (2.12)$$

де $|U(S_t, \delta)|$ – кількість модифікацій DOM, необхідних для відображення змін.

Оскільки операції над DOM мають високу потребу у ресурсах, мінімізація їх кількості є важливою для продуктивності. Коректність відображення визначаємо за (2.13), тобто DOM після оновлення повинен точно відповідати новому стану даних:

$$R(S_{t+1}) = D_{t+1} \quad (2.13)$$

Локальність змін визначимо за (2.7), але оптимізація повинна гарантувати, що кількість модифікацій DOM буде відповідати оновленню тільки тих вузлів, що змінилися (2.14)

$$|U(S_t, \delta)| \approx |\text{scope}(\delta)|. \quad (2.14)$$

Стабільність продуктивності визначимо за допомогою (2.15):

$$\forall t: \text{time}(U(S_t, \delta)) < \tau, \quad (2.15)$$

де τ – допустимий поріг часу оновлення.

Задача оптимізації оновлення інтерфейсу вебзастосунків заключається в тому, щоб знайти модель M та алгоритм A такі, щоб мінімізувати кількість модифікацій DOM, необхідних для відображення змін при дотриманні обмежень коректності та стабільності, тобто (2.16):

$$A(M, S_t, \delta) \rightarrow \min |U(S_t, \delta)|. \quad (2.16)$$

Ефективність оновлення інтерфейсу вебзастосунків значною мірою залежить від того, наскільки точно система може визначити, які частини інтерфейсу залежать від змінених даних. Для цього вводимо формальну модель залежностей у вигляді графу (2.17):

$$G_{dep} = (V, Dep), \quad (2.17)$$

де V – множина вузлів структури даних, Dep – множина залежностей, $Dep \subseteq V \times V$. Пара $(v_i, v_j) \in Dep$ означає, що зміна вузла v_i впливає на зміну вузла v_j .

До властивостей графа залежностей слід віднести орієнтованість (залежності мають напрям), ациклічність (цикли призводять до нескінченних оновлень) та локальність (у добре оптимізованих системах), (2.18)

$$|Dep(v)| \ll |V|. \quad (2.18)$$

У традиційних фреймворках (React, Vue) залежності визначаються на рівні компонентів, що призводить до надмірних оновлень. У реактивних системах (SolidJS, Svelte) залежності визначаються на рівні окремих значень, що значно зменшує розмір графа. Зміна δ у вузлі v породжує множину вузлів, які необхідно оновити, (2.19):

$$Affected(v) = \{u | (v, u) \in Dep\}. \quad (2.19)$$

У традиційних системах модель поширення матиме вигляд, (2.20):

$$|Affected(v)| \approx O(|V|). \quad (2.20)$$

У реактивних, відповідно (2.21):

$$|Affected(v)| \approx O(1). \quad (2.21)$$

Це фундаментальна різниця між diffing-архітектурами та fine-grained реактивністю. Оптимальне оновлення визначаємо як, (2.22):

$$Affected^*(v) = \min Affected(v), \quad (2.22)$$

тобто мінімальна множина вузлів, які необхідно оновити для коректного відображення змін.

Задача оптимізації полягає у побудові такої моделі залежностей, яка мінімізує $|Affected(v)|$. Для оцінки ефективності моделей та алгоритмів оптимізації оновлення інтерфейсу вебзастосунків вводимо формальні метрики. Кількість DOM-операцій, які необхідно мінімізувати, (2.23):

$$C_{dom} = |U(S_t, \delta)|. \quad (2.23)$$

Глибину оновлень визначатимемо за (2.24):

$$depth_{update} = \max_{v \in Affected(\delta)} dist(r, v). \quad (2.24)$$

Чим менша глибина оновлення, тим стабільніша продуктивність. Тоді час оновлення інтерфейсу вебзастосунків (реальна швидкодія інтерфейсу) визначаємо за (2.25):

$$T_{render} = time(U(S_t, \delta)). \quad (2.25)$$

Стабільність продуктивності можна визначити за формулою (2.26):

$$Var(T_{render}) \rightarrow 0. \quad (2.26)$$

Стабільність важлива для UX: навіть рідкісні пікові затримки погіршують взаємодію. Складність алгоритму визначаємо за формулою (2.27):

$$C_{alg} = O(f(|V|, depth(S))). \quad (2.27)$$

Оптимальні алгоритми повинні мати $C_{alg} \approx O(1)$, або $O(\log n)$. Оновлення інтерфейсу вебзастосунків буде оптимальним відносно зміни δ , якщо множина DOM-операцій $U(S_t, \delta)$ мінімальна за потужністю серед усіх коректних оновлень, тобто не існує іншого оновлення U' , такого що:

$$R(S_{t+1}) = D'_{t+1} \wedge |U'(S_t, \delta)| < |U(S_t, \delta)|. \quad (2.28)$$

Нехай зміна δ зачіпає лише вузол v_0 та підмножину його нащадків **Desc**(v_0). Тоді будь-яке коректне оновлення DOM, яке модифікує вузли, що не належать *Affected*(v_0), є надлишковим, тобто не є оптимальним у сенсі означення (2.27). За означенням *Affected*(v_0) містить усі вузли, відображення яких у DOM прямо або опосередковано залежить від v_0 . Якщо DOM змінюється поза образом *Affected*(v_0), тоді змінюються елементи, для яких дані не змінювалися. Такі операції не змінюють коректність, але збільшують $|U|$, отже, оновлення не мінімальне, тобто не є оптимальним.

Нехай існують дві стратегії оновлення а саме глобальна (diffing дерева компонентів) та fine-grained (оновлення на рівні значень за графом G_{dep}). Якщо для будь-якої зміни δ виконується (2.29).

$$|Affected_{fine}(\delta)| \leq |Affected_{global}(\delta)|, \quad (2.29)$$

то стратегія fine-grained забезпечує не гіршу, а в середньому кращу асимптотичну продуктивність за кількістю DOM-операцій. При цьому кожна стратегія породжує свій набір уражених вузлів, DOM-операції пропорційні розміру множини уражених вузлів. Якщо для кожної δ множина уражених вузлів у fine-grained не більша, ніж у глобальній, то (2.30):

$$C_{dom}^{fine}(\delta) \leq C_{dom}^{global}(\delta). \quad (2.30)$$

Сумарно за всі зміни – менша або рівна кількість операцій, тобто краща або не гірша складність. На основі аналізу наявних підходів (див. Розділ 1) та формальних моделей, що представлено вище, сформуємо концепцію

точкового оновлення інтерфейсу вебзастосунків, яка ґрунтується на принципах локальності оновлень, *fine-grained* реактивності, прямого оновлення DOM, графу залежності та прив'язки вузлів до даних DOM-вузлів.

Принцип локальності оновлень базується на оновленні лише тих DOM-вузлів, які безпосередньо залежать від змінених даних. *Fine-grained* реактивність передбачає те, що залежності визначаються на рівні окремих значень, а не компонентів. Принцип прямого оновлення DOM базується на тому, що замість глобального *diffing* застосовується точкове оновлення. Граф залежностей використовується для визначення мінімальної множини уражених вузлів. Прив'язка вузлів даних до DOM-вузлів забезпечує $O(1)$ доступ до елементів інтерфейсу.

Ці принципи дозволяють мінімізувати кількість DOM-операцій та забезпечити стабільну продуктивність навіть при значній глибині вкладеності. Граф залежностей G_{dep} є ключовим елементом системи, оскільки визначає, які вузли інтерфейсу необхідно оновлювати при зміні даних. Граф залежностей будується під час ініціалізації або компіляції шаблону. Для кожного вузла даних $v \in V$ визначається множина DOM-вузлів, які залежать від його значення. Для кожного DOM-вузла визначаються всі вузли даних, що впливають на його відображення. На основі цих відповідностей формується множина ребер Dep .

Для мінімізації обчислювальних витрат потрібна не просто структура даних, а структура, що містить інформацію для оновлення інтерфейсу вебзастосунків. Введемо для кожного вузла $v \in V$, (2.2) розширений кортеж (2.31):

$$v = (id, type, value, children, ctx, dep, cost), \quad (2.31)$$

де $ctx(v)$ – контекст оновлення інтерфейсу вузла вебзастосунку (налаштування форматування, шаблон, стиль, регіон інтерфейсу тощо); $dep(v)$ – множина залежностей вузла (дані або інші вузли), від яких залежить його відображення, яку можна представити у вигляді (2.32), $cost(v)$ – оцінка

вартості оновлення вузла (кількість DOM-операцій, орієнтовний час оновлення, або абстрактна «вага» вузла).

$$dep(v) \subseteq V \cup P, \quad (2.32)$$

де P – множина зовнішніх параметрів (стани, сигнали, глобальні змінні).

Щоб реалізувати локальне оновлення замість глобального, вводимо поняття регіонів. Визначимо розбиття множини вузлів на підмножини $Z = \{Z_1, Z_2, \dots, Z_k\}$, $Z_i \subseteq V$, такі що $\bigcup_{i=1}^k Z_i = V$, при чому $Z_i \cap Z_j = \emptyset$ для $i \neq j$.

Кожен регіон Z_i – це логічно цілісна частина інтерфейсу, яка може оновлюватися незалежно (форма, панель, блок конфігурації тощо). Для кожного регіону вводиться агрегована вартість, (2.33):

$$cost(Z_i) = \sum_{v \in Z_i} cost(v). \quad (2.33)$$

Нехай δ – зміна структури даних, тоді її область впливу $scope(\delta) \subseteq V$. Ідея оптимізації заключається в тому, що при обробці δ оновлювати мінімальну кількість регіонів та вузлів.

Враховуючи (2.17), (2.18) та (2.31), до кожної зміни δ , яка торкнеться вузла v_0 , множина вузлів для оновлення (множина уражених вузлів) визначається на формулою (2.34):

$$Affected(v_0) = \{v \in V \mid \exists \text{ шлях } (v_0 \rightsquigarrow v) \text{ у } G_{dep}\}. \quad (2.34)$$

Це саме та множина, яку має сенс розглядати при оновлення інтерфейсу вебзастосунків, а все інше то зайві обчислення. Визначаємо відображення $Bind: V \rightarrow D$, яке встановлює відповідність між вузлом даних та DOM-вузлом. Це дозволяє не будувати нові піддерева і не виконувати глобальний diffing та оновлювати лише конкретний DOM-вузол для конкретного вузла даних v . Для зміни δ задаємо множину уражених вузлів, (2.35):

$$Affected(\delta) = \bigcup_{v_0 \in scope(\delta)} Affected(v_0). \quad (2.35)$$

Тоді цільову функцію моделі представимо у вигляді (2.36):

$$\min C_{dom}(\delta) = \min \sum_{v \in Affected(\delta)} cost(v), \quad (2.36)$$

При умові, що всі залежні вузли оновлені коректно, $Affected(\delta)$ не містить надлишкових вузлів, які не змінювалися і не залежать від змін. Інакше кажучи, модель представлення вкладених структур даних вважається оптимізованою для оновлення інтерфейсу вебзастосунків, якщо залежності Dep описані наскільки можливо локально (fine-grained), регіони Z_i спроектовані так, щоб мінімізувати сумарний $cost(Z)$, що оновлюється при типовій зміні δ , прив'язки $Bind$ забезпечують прямий доступ до DOM без додаткових пошуків або перебудови дерев.

Нехай зміна δ зачіпає вузол v_0 . Тоді для моделі, що використовує граф залежностей G_{dep} та пряму прив'язку $Bind$, верхня межа кількості DOM-операцій дорівнює, (2.37):

$$C_{dom}^{opt}(\delta) = \sum_{v \in Affected(v_0)} cost(v). \quad (2.37)$$

І ця межа є строго меншою або рівною верхній межі для моделей, що використовують глобальний diffing $C_{dom}^{opt}(\delta) \leq C_{dom}^{diff}(\delta)$. У diffing-моделях оновлення виконується для всього піддерева, що містить v_0 , (2.38):

$$C_{dom}^{diff}(\delta) = \sum_{v \in Subtree(v_0)} cost(v). \quad (2.38)$$

У запропонованій моделі (2.33) оновлюються лише вузли, які мають залежність від v_0 , (2.39):

$$Affected(v_0) \subseteq Subtree(v_0). \quad (2.39)$$

Оскільки множина уражених вузлів є мінімальною множиною, необхідною для коректного оновлення, то справедлива наступна нерівність (2.40):

$$\sum_{v \in Affected(v_0)} cost(v) \leq \sum_{v \in Subtree(v_0)} cost(v). \quad (2.40)$$

Отже, справедлива наступна нерівність, (2.41):

$$C_{dom}^{opt}(\delta) \leq C_{dom}^{diff}(\delta). \quad (2.41)$$

Таким чином, формально доведено правильність запропонованих рішень. В подальшому проведемо практичне підтвердження отриманих теоретичних результатів.

2.2. Метод точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками

Традиційні підходи до оновлення інтерфейсу (DOM-маніпуляції, Virtual DOM diffing, компонентний рендеринг) вирішують задачу глобально, тобто, при будь-якій зміні стану перераховується або порівнюється значна частина дерева елементів. Такий підхід не є оптимальним для глибоко вкладених структур, де зміна одного вузла торкається лише незначної підмножини елементів. Запропонований метод точкового оновлення вирішує цю проблему, замінюючи глобальний перерахунок локальним, коли оновлюються виключно ті елементи інтерфейсу, які мають пряму залежність від зміненого значення даних. Метод точкового оновлення інтерфейсу вебзастосунків складається з таких кроків:

Крок 1. Ініціалізація структури та графа залежностей. На етапі завантаження вебзастосунку виконується однократна побудова дерева даних $S = (V, E, r)$, (2.1) та графа залежностей $G_{dep} = (V, Dep)$, (2.17). Кожен вузол v_j отримує список v_i – ідентифікаторів вузлів, від значень яких він залежить у вигляді множини залежностей Dep , (2.9). Одночасно, для кожного вузла встановлюється пряма прив'язка до відповідного DOM-вузла відповідно до відображення $R: V \rightarrow D$, (2.4). Ця операція виконується один раз і не повторюється при наступних оновленнях.

Крок 2. Інкапсуляція вузлів у НОС-обгортки. Кожен елемент інтерфейсу загортається у компонент вищого порядку. НОС-обгортка є програмним втіленням принципу *fine-grained* реактивності і виконує роль проміжного рівня логіки між сховищем стану та елементом інтерфейсу. Вона зберігає список залежностей вузла ($\text{dependencyList} = \text{dep}(v)$) та підписується на зміни у сховищі. При отриманні сигналу про зміну НОС-обгортка не оновлює елемент автоматично, а спочатку перевіряє, чи входить ідентифікатор зміненого вузла, v_i , до її dependencyList .

Крок 3. Поширення зміни через граф залежностей G_{dep} . Коли користувач ініціює зміну δ у вузлі v_0 , сховище фіксує ідентифікатор зміненого вузла ($\text{changedId} = v_0$) та оновлене значення. Далі метод визначає множину уражених вузлів $\text{Affected}(\delta)$, яка не містить надлишкових вузлів. Таким чином, замість глобального обходу дерева виконується цільове сповіщення лише тих НОС-обгортки, чий dependencyList містить changedId . Таким чином реалізується область впливу зміни $\text{scope}(\delta) \subseteq V$.

Крок 4. Перевірка релевантності та точкове оновлення DOM. Кожна НОС-обгортка при отриманні сигналу перевіряє умову: $\text{dependencyList.includes}(\text{changedId})$. Якщо умова виконується, компонент отримує оновлені дані зі сховища та ініціює повторне відмалювання лише свого DOM-вузла. Якщо умова не виконується – компонент залишається без змін. Таким чином, кількість DOM-операцій пропорційна кількості залежних вузлів $\text{Affected}(\delta)$, а не загальній кількості вузлів $|V|$, що забезпечує виконання умови оптимальності (2.12) та (2.16).

Крок 5. Стабілізація стану. Після завершення точкового оновлення ідентифікатор зміненого вузла скидається ($\text{changedId} = \text{undefined}$), і система переходить у стабільний стан до наступної користувацької взаємодії. Відсутність каскадних перерахунків забезпечує стабільність часу відгуку: $\text{Var}(T_{\text{render}}) \rightarrow 0$.

Таким чином, Метод точкового оновлення відповідає за оновлення інтерфейсу вебзастосунків завдяки чіткій взаємодії трьох компонентів:

формальної моделі, що визначає структуру даних і залежності; графа залежностей G_{dep} , що є ядром методу і забезпечує точне визначення множини уражених вузлів НОС-обгортки, що є програмним механізмом реалізації точкового оновлення. Саме ця комбінація забезпечує асимптотичну складність оновлення $O(k)$, де k – кількість залежних елементів, що суттєво перевищує ефективність традиційних підходів з $O(n)$ та $O(n \cdot \log n)$.

Для належного розуміння запропонованих рішень, введемо пояснення елементів діаграм та схем, що представлені на рис.2.1. Так, кожен інтерактивний елемент, яким взаємодіє користувач має номер (№), який включає рівень вкладеності (X) та порядковий номер (Y). Прикладом таких елементів може бути поле для введення даних, випадний список або перемикач, рис.2.1.

Під сховищем ми будемо розуміти місце зберігання інформації про те, який елемент змінився та яке його поточне значення. НОС-обгортка – надбудова над кожним елементом (вузлом) дерева структури програмного продукту, яка містить програмну логіку, що відповідає за зміною елементів у сховищі і приймає рішення про потребу у перемальовуванні елементу вебзастосунку. Під даними ми будемо розуміти данні у вигляді деревовидної структури (в тому числі з глибокою вкладеністю), які містять усю інформацію для рекурсивного промальовування вебдодатку.

На рис.2.2 відображено процес генерації елементів користувацького інтерфейсу у спрощеному вигляді, який не використовує оптимізованих методів оновлення. Діаграма ілюструє традиційну схему формування елементів на основі вхідних даних, яка не враховує локальний характер змін та залежності між компонентами в структурі даних. Процес починається з передачі даних із серверної частини до клієнтської.

Отримані дані використовуються для ініціалізації форми, під час якої відбувається задання значень для всіх елементів інтерфейсу незалежно від того, чи потребують вони оновлення чи ні. Після цього здійснюється

створення перших елементів масиву даних, що виступає точкою входу в подальшу обробку структури.

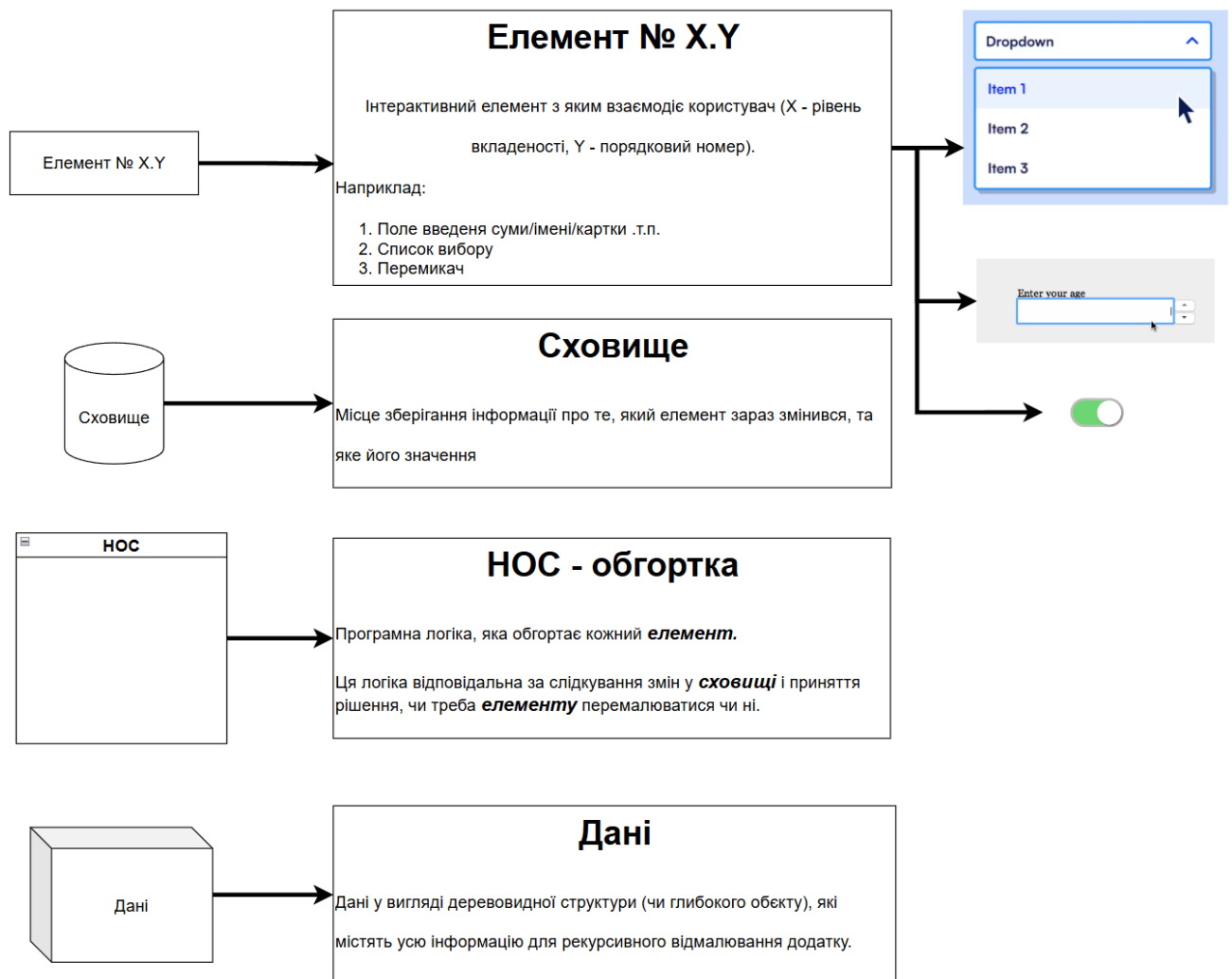


Рис.2.1. Пояснення елементів діаграм та схем

Далі алгоритм переходить до послідовного обходу вкладених елементів. На кожному кроці перевіряється наявність дочірніх елементів у поточного вузла. У разі їх наявності виконується створення всіх дочірніх елементів, після чого процес повторюється рекурсивно для кожного з них. Якщо ж дочірні елементи відсутні, то відбувається зупинка роботи з поточним вузлом. Окремо на діаграмі зазначено, що на кожному етапі виконуються рекурсивні перевірки та виклики, які супроводжують створення елементів. Це призводить до додаткових обчислювальних витрат, оскільки обробка здійснюється для всієї підструктури незалежно від фактичних змін у даних.

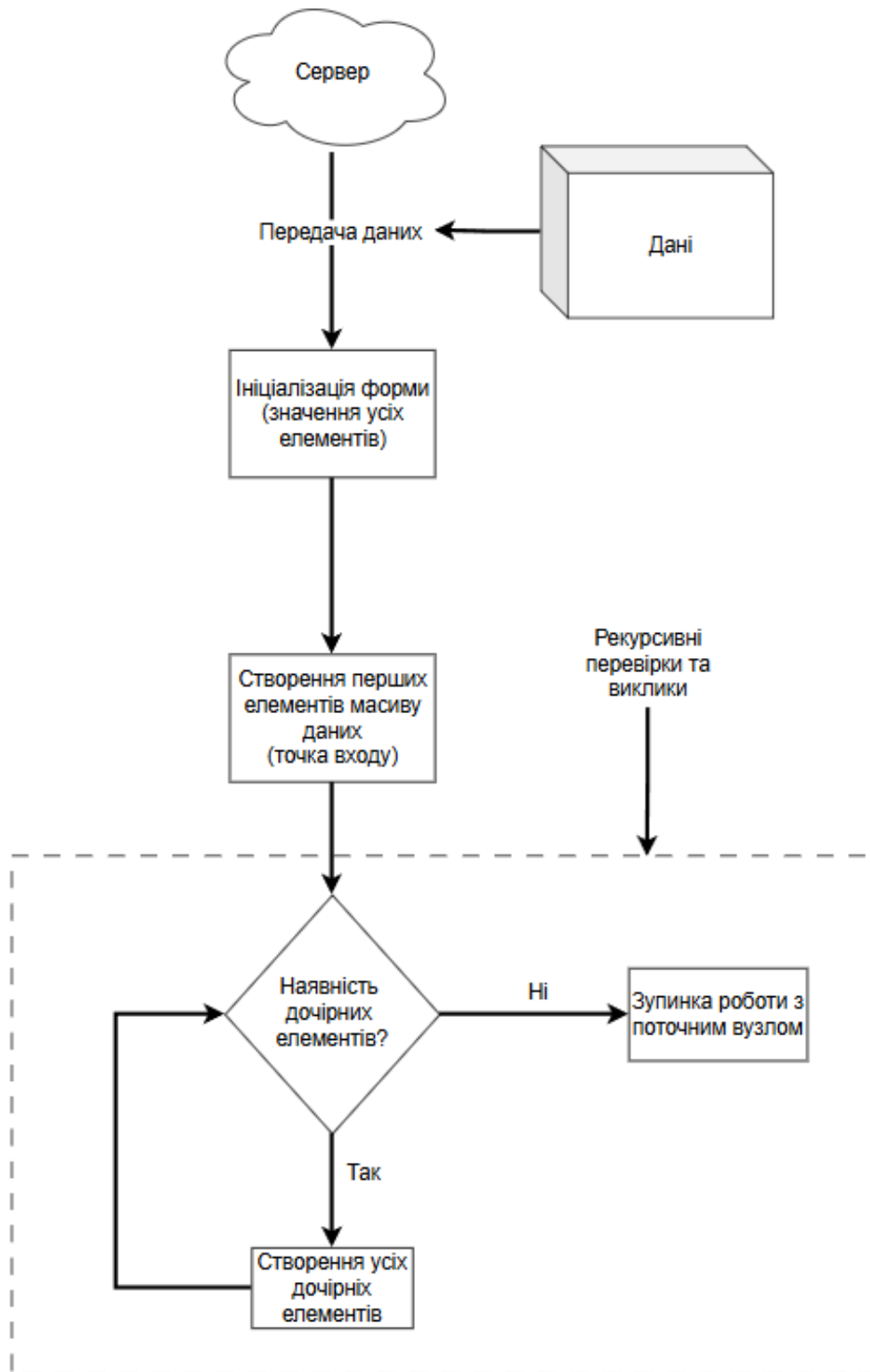


Рис.2.2. Схема процесу генерації елементів користувацького інтерфейсу у спрощеному вигляді (без використання методу)

Діаграма ілюструє традиційну схему формування елементів на основі вхідних даних, яка не враховує локальний характер змін та залежності між компонентами в структурі даних. Процес починається з передачі даних із

серверної частини до клієнтської. Отримані дані використовуються для ініціалізації форми, під час якої відбувається задання значень для всіх елементів інтерфейсу незалежно від того, чи потребують вони оновлення чи ні. Після цього здійснюється створення перших елементів масиву даних, що виступає точкою входу в подальшу обробку структури.

Далі алгоритм переходить до послідовного обходу вкладених елементів. На кожному кроці перевіряється наявність дочірніх елементів у поточного вузла. У разі їх наявності виконується створення всіх дочірніх елементів, після чого процес повторюється рекурсивно для кожного з них. Якщо ж дочірні елементи відсутні, то відбувається зупинка роботи з поточним вузлом. Окремо на діаграмі зазначено, що на кожному етапі виконуються рекурсивні перевірки та виклики, які супроводжують створення елементів. Це призводить до додаткових обчислювальних витрат, оскільки обробка здійснюється для всієї підструктури незалежно від фактичних змін у даних.

Таким чином, наведений на рисунку підхід демонструє глобальний характер обробки структури, за якого оновлення інтерфейсу виконується шляхом повного або часткового повторного створення вкладених елементів. Такий підхід не забезпечує локальності оновлень і може призводити до надлишкових операцій, особливо у випадку глибоко вкладених структур даних, що обґрунтовує необхідність застосування оптимізованих методів оновлення інтерфейсу вебзастосунків.

На рис.2.3 зображено просту форму представлення елементів інтерфейсу без використання точкового методу оновлення. Схема ілюструє ієрархічну структуру даних та відповідних елементів інтерфейсу, яка формується на основі вхідних даних у традиційному підході. У верхній частині схеми розташовано блок «Дані», який представляє вихідний набір інформації, отриманої для відображення. Дані передаються до форми, що виконує роль централізованого контейнера та місця збереження всіх користувацьких змін. Форма є єдиною точкою керування станом і використовується для ініціалізації та подальшого оновлення всіх вкладених елементів. Далі від форми

формується кореневий елемент інтерфейсу (Елемент №1.1), від якого рекурсивно будуються дочірні елементи.

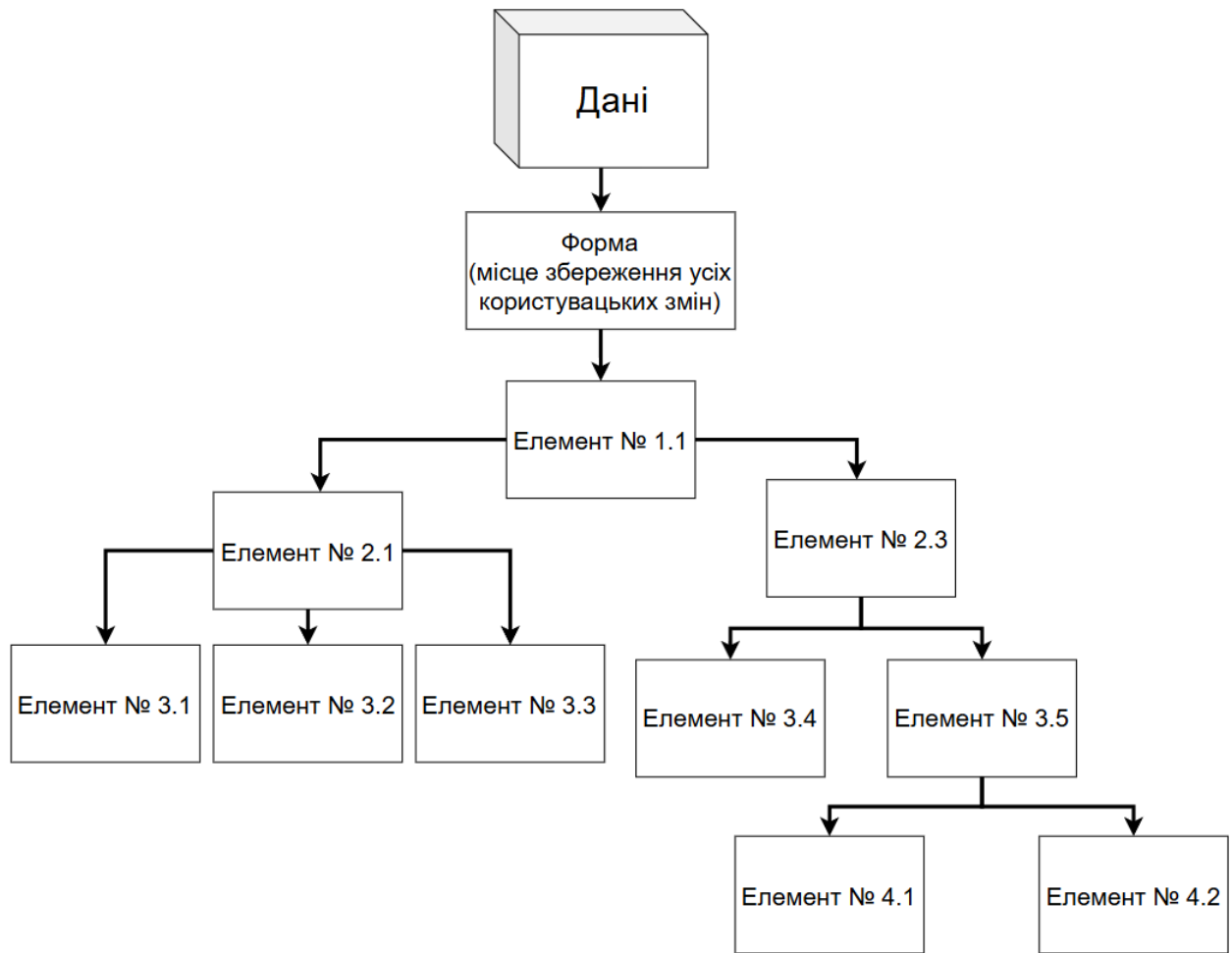


Рис.2.3. Проста форма представлення елементів інтерфейсу без використання точкового методу оновлення

Схема, що представлена на рис. 2.3 демонструє деревоподібну структуру представлення, де кожен елемент може мати декілька дочірніх елементів наступного рівня (Елементи №2.1, №2.3), які, у свою чергу, породжують елементи глибших рівнів вкладеності (Елементи №3.1–№3.5, №4.1, №4.2).

Такий підхід передбачає, що всі елементи інтерфейсу жорстко пов'язані зі структурою форми та залежать від її загального стану. У разі зміни будь-якого значення у формі оновлення може поширюватися на значну частину дерева елементів, незалежно від того, які саме вузли фактично залежать від змінених даних. Таким чином, наведений на рисунку підхід ілюструє глобальний характер представлення та оновлення елементів, властивий

традиційним реалізаціям без використання локальних залежностей і fine-grained реактивності. Це призводить до надлишкових оновлень при роботі з глибоко вкладеними структурами та обґрунтовує доцільність застосування точкового методу оновлення інтерфейсу вебзастосунків.

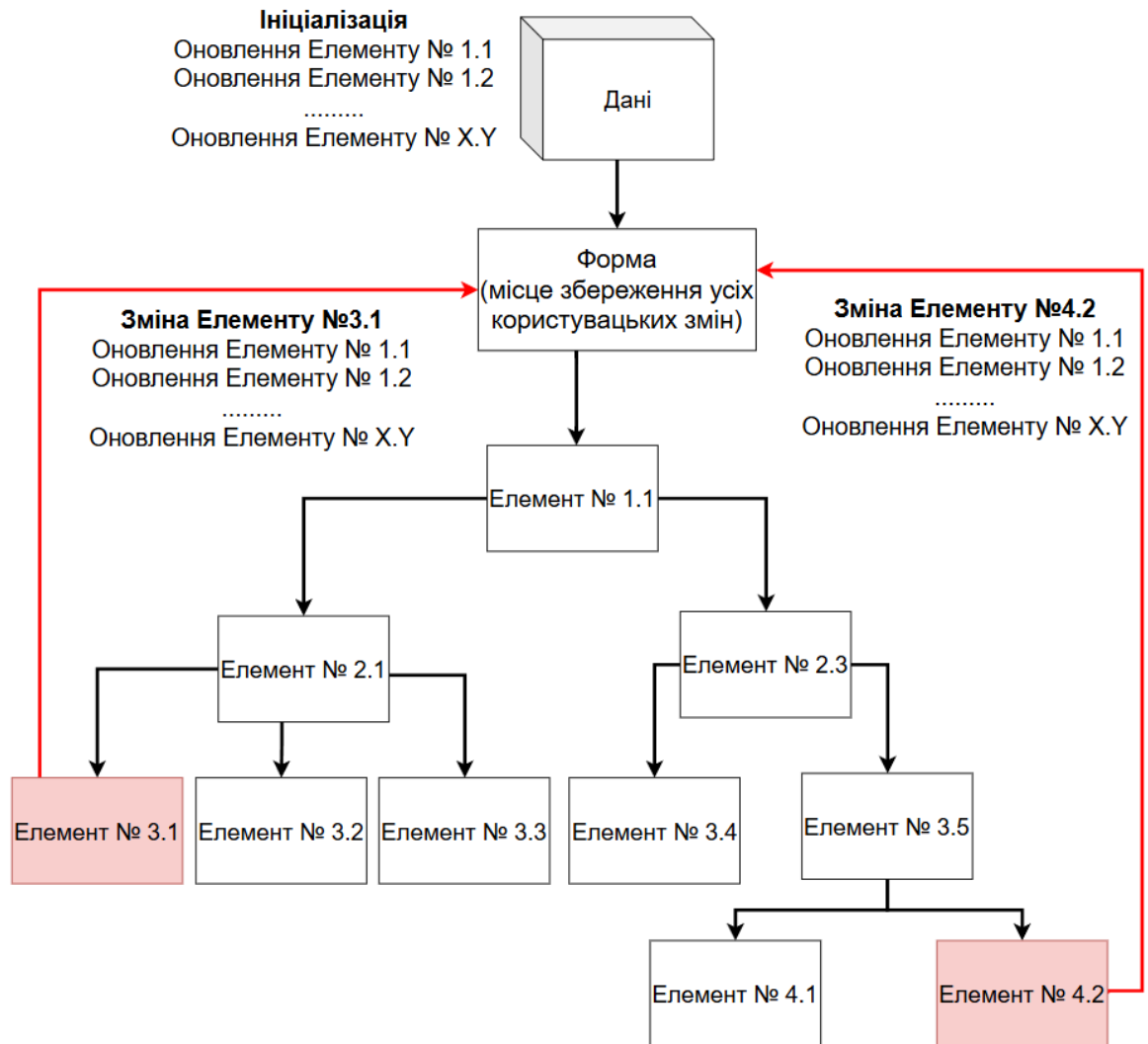


Рис.2.4. Ієрархічна структура елементів форми та процес їх оновлення під час ініціалізації та при зміні даних

В архітектурі, що представлена на рис.2.4 зміна даних на будь-якому вузлі призведе до перемалювання усіх вузлів (адже дані каскадно спускаються від одного вузла до іншого). Кожна інтеракція користувача є тригером зміни форми, а зміна форми є тригером повторного оновлення елементів.

У верхній частині схеми відображено етап ініціалізації, під час якого на основі блоку «Дані» виконується послідовне оновлення усіх елементів інтерфейсу (Елементи №1.1, №1.2, ..., №X.Y). Отриманий стан зберігається у

формі, яка виступає централізованим сховищем усіх користувацьких змін та є єдиною точкою керування станом інтерфейсу. Нижче наведено ієрархічну деревоподібну структуру елементів інтерфейсу, де кореневий елемент (Елемент №1.1) породжує дочірні елементи наступних рівнів вкладеності (Елементи №2.*, №3.*, №4.*). Така структура відповідає класичній моделі представлення інтерфейсу у вигляді дерева компонентів.

Червоним контуром на схемі показано область повторного оновлення інтерфейсу вебзастосунків, яка активується при зміні будь-якого окремого елемента. Зокрема, при зміні Елемента №3.1 або Елемента №4.2 ініціюється оновлення стану форми, що призводить до повторного оновлення всіх елементів інтерфейсу, включно з тими, які не залежать від змінених даних. Це відображено у вигляді повторного оновлення елементів верхніх рівнів (№1.*, №2.*) та інших піддерев.

Таким чином, рисунок 2.4 наочно демонструє відсутність локальності оновлень у традиційному підході: будь-яка зміна окремого вузла призводить до глобального оновлення значної частини дерева елементів. Такий механізм обумовлює надлишкові обчислення та збільшення кількості DOM-операцій, особливо у випадку глибоко вкладених структур даних, що підтверджує необхідність застосування оптимізованих методів оновлення інтерфейсу вебзастосунків, заснованих на локальних залежностях і *fine-grained* реактивності. Перед наведенням удосконаленого підходу до оновлення інтерфейсу доцільно окремо показати етап формування масиву залежностей, оскільки саме він є базою для подальшого локального визначення компонентів, що потребують повторної обробки. Якщо у примітивному підході зміна значення часто призводить до надмірного проходження по великій частині структури, то в запропонованому варіанті попередньо формується спеціалізоване подання зв'язків між елементами. Таке подання дає змогу зафіксувати, які саме вузли залежать від конкретного параметра, і надалі використовувати цю інформацію під час оновлення стану. Блок-схема, рис.

2.5, відображає послідовність побудови масиву залежностей на основі вхідної ієрархічної структури даних.

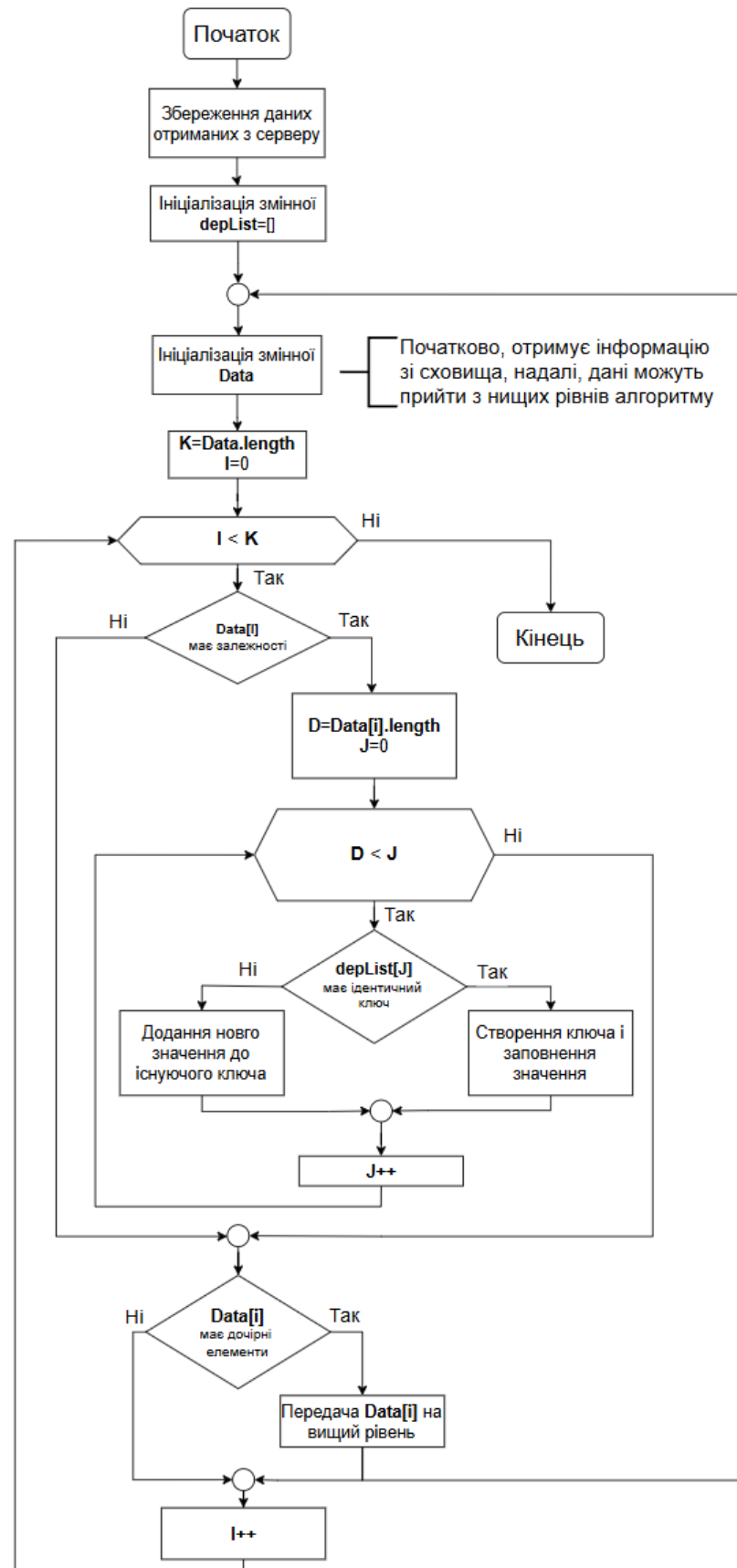


Рис.2.5. Блок-схема алгоритму формування масиву залежностей

На початковому етапі виконується ініціалізація службових змінних, після чого алгоритм послідовно аналізує кожен елемент масиву, перевіряє наявність залежностей, додає їх до структури та, за потреби, рекурсивно переходить до дочірніх вузлів. У результаті формується узагальнений масив залежностей, який акумулює інформацію про зв'язки між елементами незалежно від рівня їх вкладеності.

Після завершення роботи алгоритму формується масив залежностей, у якому для кожного керуючого параметра зберігається перелік елементів, чия видимість, доступність або стан залежать від його значення.

Це дозволяє перейти від суцільного перегляду всієї структури до адресної обробки лише тих вузлів, які дійсно пов'язані зі змінним параметром. Таким чином, створений масив виступає проміжною, але важливою структурою для оптимізації подальших етапів оновлення вебзастосунку.

На рисунку 2.6 представлена блок-схема процесу генерації елементів користувацького інтерфейсу із застосуванням запропонованого точкового методу який базується на формальній моделі представлення глибоко вкладених структур даних, розроблений у пункті 2.1.

Процес починається з передачі даних із сервера, які формують початковий стан структури даних. Отримані дані використовуються для ініціалізації сховища, що відповідає стану структури S_t у момент часу t (2.6).

Наведена на рис.2.7 блок-схема ілюструє логіку використання масиву залежностей під час обробки зміни значення елемента. Після введення нового значення система зберігає ідентифікатор зміненого елемента та його нове значення у сховищі, а далі виконує послідовну перевірку: чи має поточний елемент список залежностей, чи належить змінений ідентифікатор до цього списку, а також чи відрізняється нове значення від попереднього. Лише за виконання всіх необхідних умов елемент отримує дозвіл на оновлення.

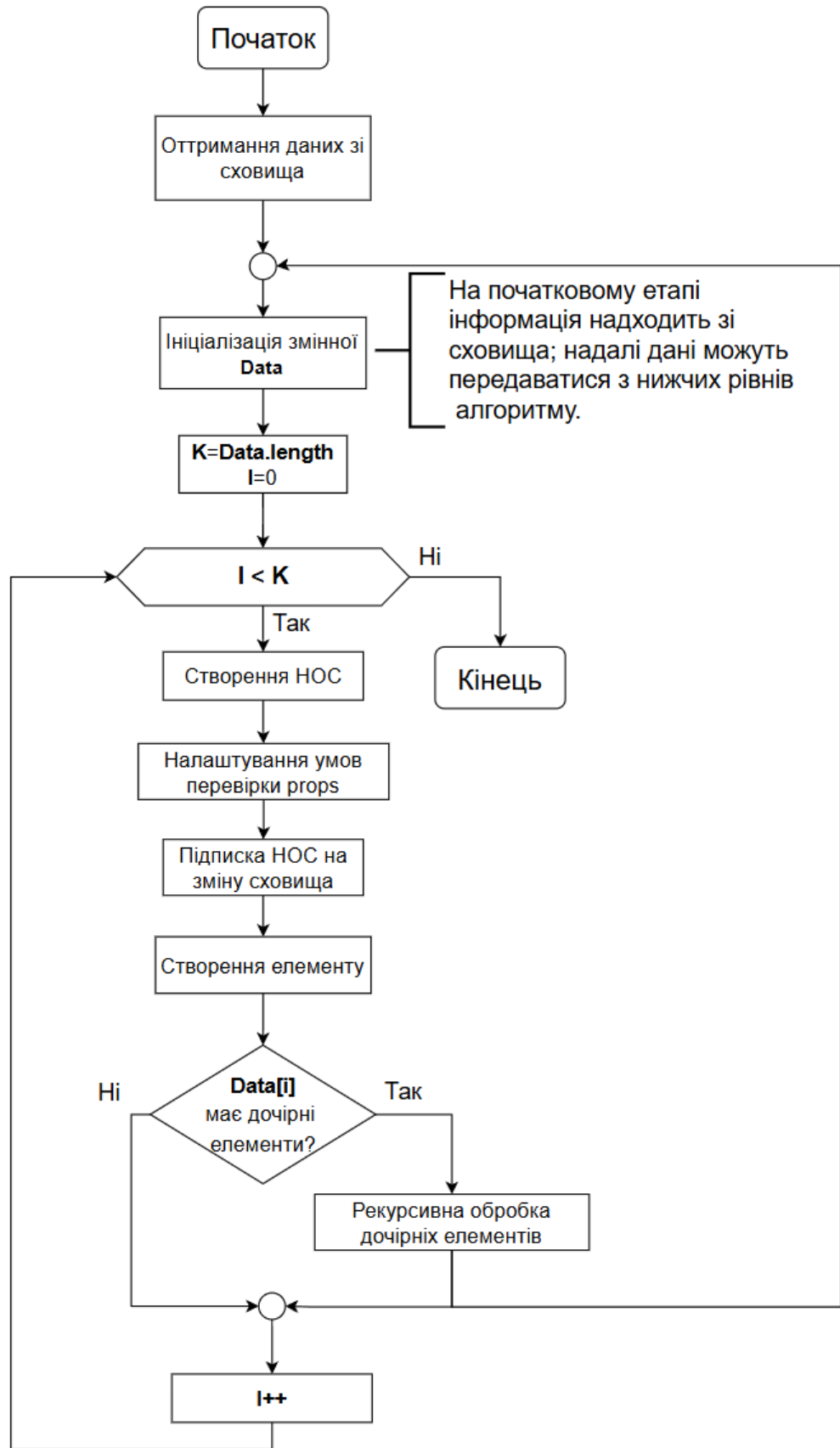


Рис.2.6. Блок-схема процесу генерації елементів користувацького інтерфейсу із застосуванням запропонованого точкового методу

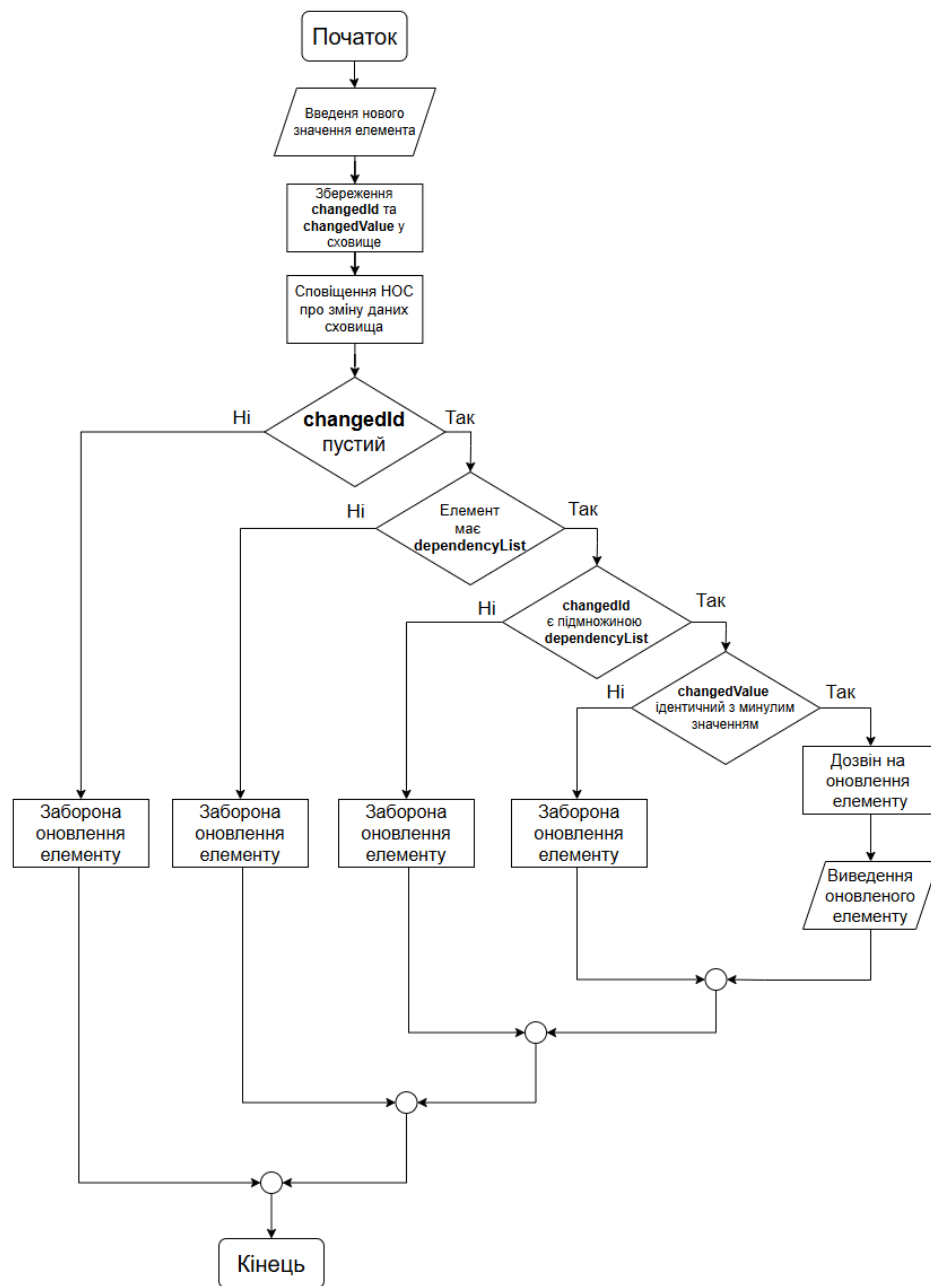


Рис.2.7. Блок-схема алгоритму точкового оновлення елемента на основі масиву залежностей

На цьому етапі додатково виконується створення масиву (графа) залежностей, який відповідає графу залежностей, визначеному у формулі (2.17). Побудова графа залежностей здійснюється одноразово та забезпечує фіксацію залежностей між окремими значеннями даних. Далі виконується ініціалізація форми, під час якої задаються початкові значення елементів інтерфейсу та встановлюються прямі прив'язки між вузлами структури даних і відповідними DOM-вузлами, що реалізує відображення (2.4).

На відміну від традиційного підходу, форма не є джерелом глобального повторного оновлення, а використовується як контейнер стану. Основна частина алгоритму, виділена пунктирною рамкою, відповідає локальному створенню елементів структури. Для кожного вузла даних створюється НОС (Higher-Order Component), який: підписується на зміни відповідного фрагмента сховища, зберігає список залежностей вузла відповідно до множини $dep(v)$, визначеної у (2.31) та (2.32) відповідно, визначає умови повторного оновлення елемента.

Такий підхід безпосередньо реалізує *fine-grained* реактивність, описану в розділі 2.1, де залежності визначаються на рівні окремих значень, а не компонентів. Після створення НОС виконується створення елемента інтерфейсу та його інкапсуляція у відповідний НОС, що забезпечує контрольоване та локальне оновлення DOM. Далі перевіряється наявність дочірніх елементів, і у разі їх існування алгоритм рекурсивно застосовується до кожного з них. Якщо дочірні елементи відсутні, робота з поточним вузлом завершується.

При виникненні зміни даних δ метод використовує побудований граф залежностей для визначення локальної області впливу $score(\delta)$ відповідно до (2.7) та формує мінімальну множину уражених вузлів, що відповідає формулі (2.33). Оновлення DOM виконується лише для елементів, що належать цій множині, що забезпечує мінімальну кількість DOM-операцій у сенсі означення оптимального оновлення (2.21).

Таким чином, наведений на рисунку процес є алгоритмічним втіленням формальної моделі точкового оновлення інтерфейсу вебзастосунків, у якій поєднано граф залежностей, локальність оновлень, *fine-grained* реактивність та пряму прив'язку вузлів даних до DOM-вузлів. Це дозволяє усунути надлишкові обчислення, характерні для глобального *diffing*-підходу, і забезпечити стабільну продуктивність навіть за значної глибини вкладеності структури даних, що узгоджується з теоретичними положеннями пункту 2.1.

На рисунку 2.8 представлено складну форму представлення елементів інтерфейсу з використанням точкового методу оновлення. Показано, що кожен елемент інтерфейсу має створений НОС. На рисунку 2.6 показано зв'язки між елементами, що відображені у вигляді стрілок. Слід зазначити, що після першого рендеру всього дерева усі зв'язки зникають, тобто кожен елемент та його «обгортка» стають незалежними.

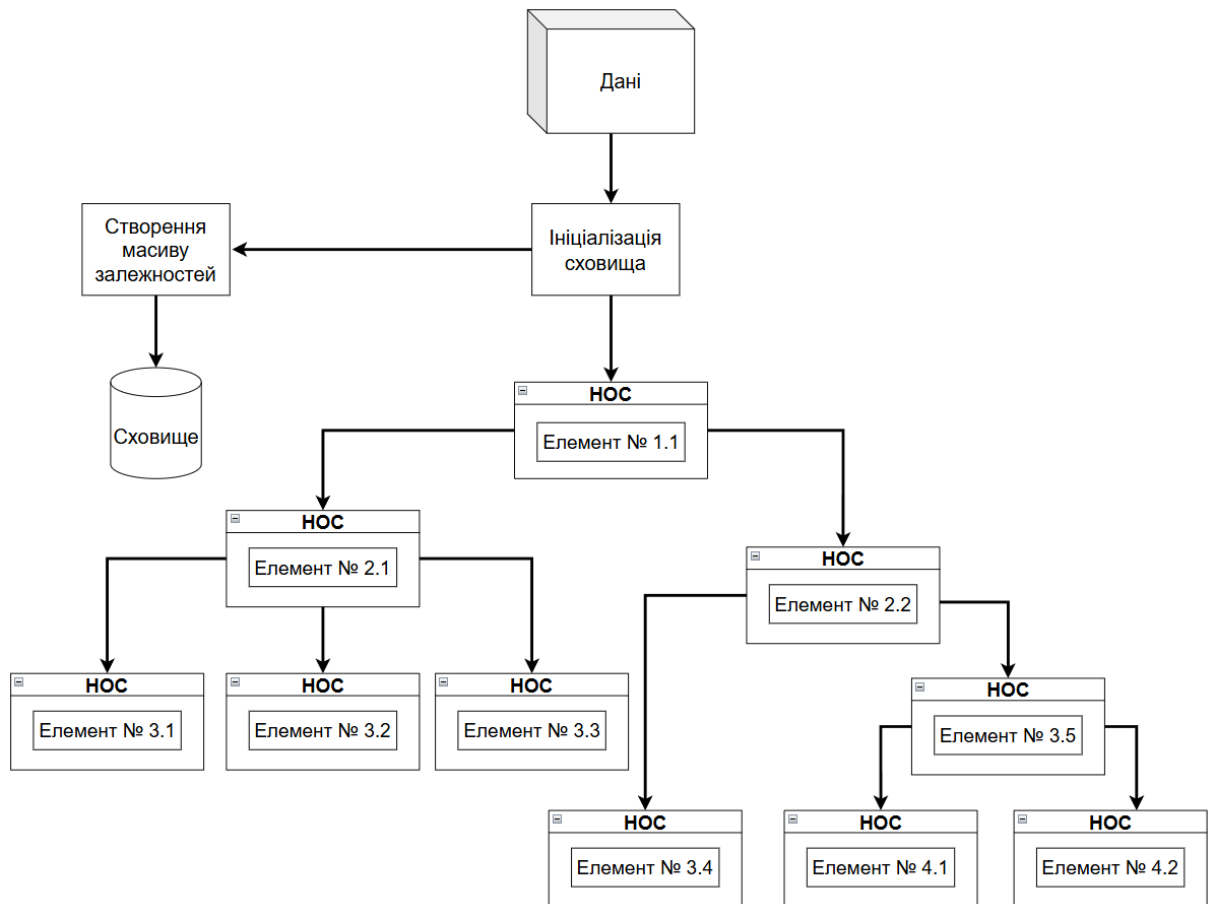


Рис.2.8. Складна форма представлення елементів інтерфейсу з використанням точкового методу оновлення

На рисунку 2.8 показано зв'язки між елементами, що відображені у вигляді стрілок. Слід зазначити, що після першого рендеру всього дерева усі зв'язки зникають, тобто кожен елемент та його «обгортка» стають незалежними. Змоделюємо випадок, де Елемент №3.1 та Елемент №2.1 зв'язані між собою, тобто мають залежність, Елемент 4.2 також має залежність від стану/зміни Елемента 3.4, рис.2.9. На рисунку 2.9 показано сценарій, за якого користувач вносить зміну до Елемента №3.1.

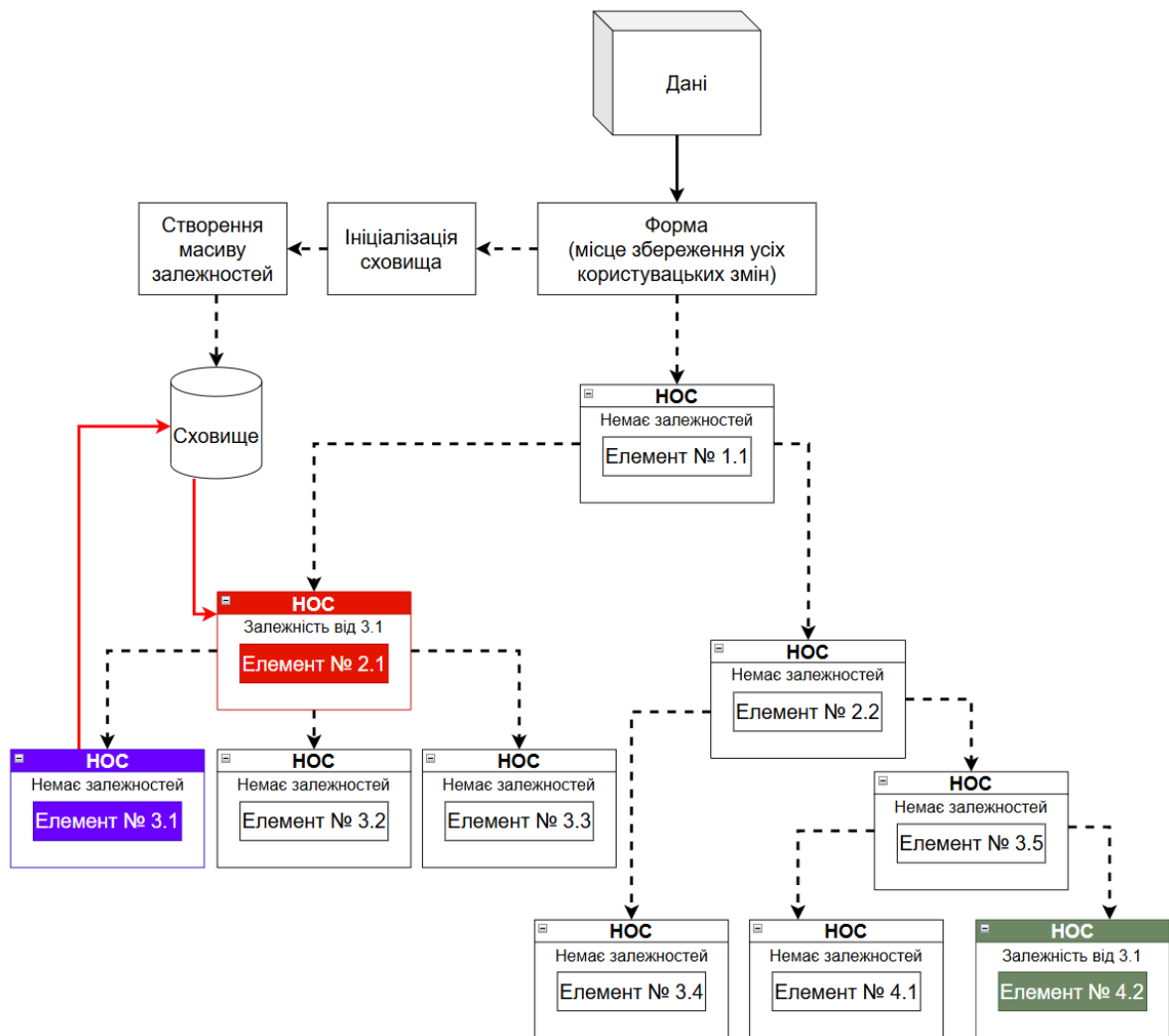


Рис.2.9. Приклад локального оновлення інтерфейсу при наявності залежностей між окремими елементами

Ця зміна формально описується як операція δ над станом S_t відповідно до (2.5). На основі побудованого графа залежностей метод визначає локальну область впливу $score(\delta)$ згідно з (2.7). До цієї області потрапляють лише ті елементи, які прямо або опосередковано залежать від зміненого значення, зокрема Елемент №2.1 та Елемент №4.2. Кольорове виділення на схемі відображає мінімальну множину уражених вузлів, яка підлягає оновленню. Усі інші елементи інтерфейсу, що не належать цій множині, не піддаються повторному оновленню, незважаючи на їхню наявність у загальній ієрархії. Це безпосередньо відповідає означенню оптимального оновлення (2.21) та формулі мінімальності множини уражених вузлів (2.33).

Таким чином, наведений на рисунку приклад наочно демонструє ключове положення формальної моделі, що поширення змін визначається не структурою дерева елементів, а графом залежностей між значеннями даних. Запропонований підхід забезпечує локальність оновлень, зменшення кількості DOM-операцій та усунення надлишкових оновлень (перерендерингів), характерних для традиційних моделей із глобальним керуванням станом.

На рисунку 2.10 показано процес сповіщення про зміну значення Елементу №3.1 та оновлення тільки відповідних (залежних) елементів. Проведемо аналіз даної схеми. Користувач вносить зміну у значення Елементу №3.1. Елемент передає повідомлення у сховище про те, що відбулася взаємодія з компонентом. Це початкова точка тригера системи.

Сховище є центральним вузлом обробки даних, воно виконує роль єдиного джерела істини. Сховище не лише зберігає нове значення, а й реєструє метадані. У прикладі, рис.2.8, сховище отримує нові дані і зберігає ідентифікатор елемента, який був змінений, у нашому випадку, ідентифікатор 3.1, що дозволяє системі зрозуміти контекст зміни. Сховище сповіщає форму про прибуття нового значення, відповідно відбувається зберігання нових даних у формі. Також сховище розсилає широкомовне повідомлення всім зареєстрованим компонентам (НОС) про прибуття нового значення до сховища. НОСи, відповідно, перевіряють чи залежні вони від ідентифікатора елемента, якщо залежність існує (Елемент 2.1 та Елемент 4.2), то елемент оновлює відповідне значення, тобто оновлює свій стан (відбувається рендеринг тільки залежного елемента).

У випадку, якщо змінений ідентифікатор відсутній у `dependencyList`, НОС все одно обробляє отримане повідомлення (наприклад, для підтримки цілісності підписки чи внутрішньої логіки), однак не ініціює оновлення компонента. Відповідно, рендеринг не відбувається.

Такий підхід дозволяє реалізувати ефективний алгоритм селективного оновлення інтерфейсу: система уникає зайвих перерендерів, зменшує навантаження на віртуальний DOM і підвищує загальну продуктивність

застосунку, особливо в умовах великої кількості взаємопов'язаних компонентів і частих змін стану.

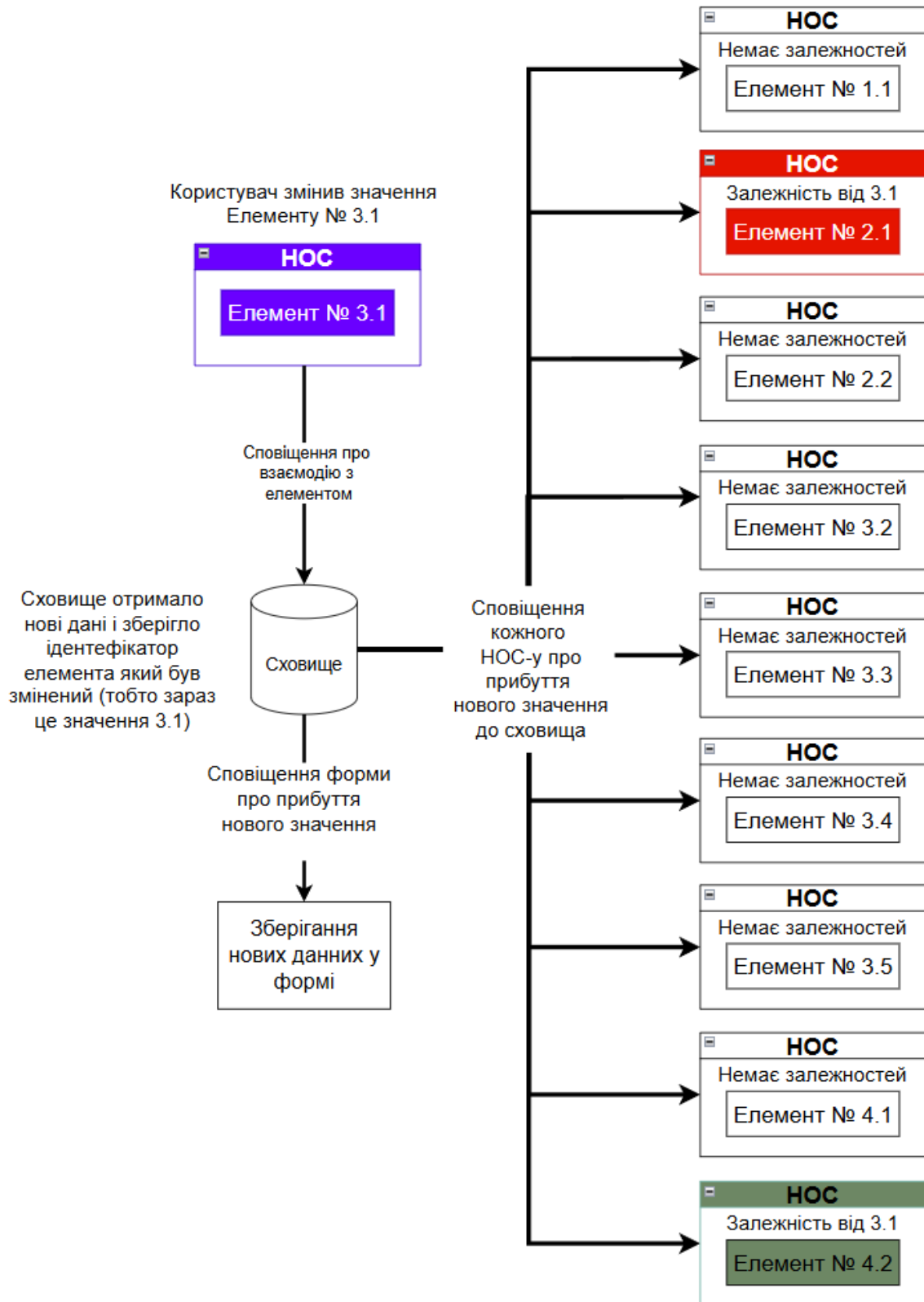


Рис.2.10. Схема процесу сповіщення про зміну значення елемента та оновлення тільки відповідних (залежних) елементів

Проведемо опис процесу прийняття рішення НОС щодо оновлення елемента. На рисунках 2.11-2.13 зображено завершальний етап роботи запропонованого методу, який є ключовим для розуміння алгоритму прийняття рішення щодо оновлення окремого елемента інтерфейсу. Схема демонструє, яким чином НОС (Higher-Order Component), спираючись на граф залежностей, визначає необхідність або відсутність необхідності оновлення DOM-вузла. Відповідно до формальної моделі, розробленої у пункті 2.1, кожен НОС пов'язаний із конкретним вузлом структури даних та містить множину залежностей $dep(v)$. При надходженні сигналу про зміну стану δ , що відповідає операції (2.5), НОС ініціює процедуру аналізу впливу цієї зміни. На схемі послідовно показано три типові випадки, які можуть виникати під час прийняття рішення.

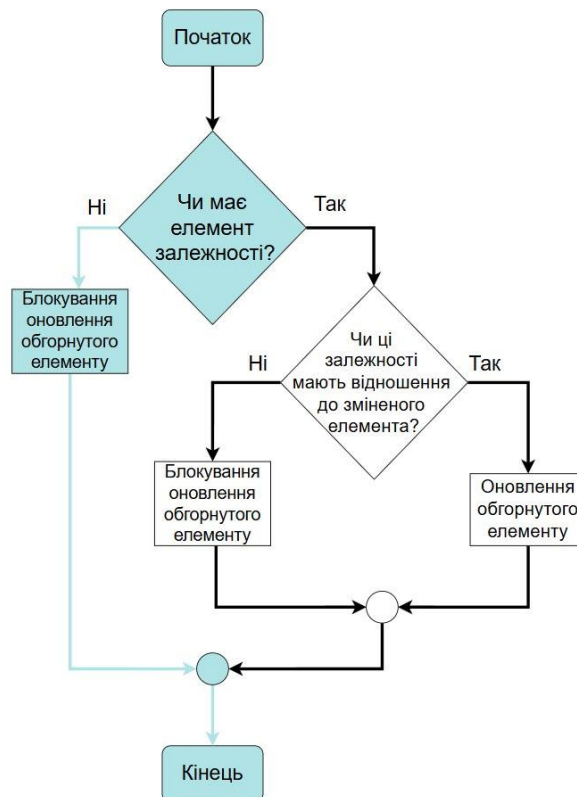


Рис.2.11. Узагальнена схема алгоритму прийняття рішення НОС про перемальовування елемента: випадок 1

Перший випадок – НОС отримує сповіщення про зміну стану, однак відповідний елемент не має залежностей або множина $dep(v)$ є порожньою, рис.2.11. У цьому випадку, відповідно до означення локальної області впливу $score(\delta)$ (2.7), зміна не може вплинути на відображення елемента. НОС блокує оновлення, і DOM-оновлення не виконується.

Другий випадок (рис.2.12) показує, що НОС має залежності (Елемент №2.1 є залежним від Елемента №3.1), і змінене значення безпосередньо входить до множини $dep(v)$ поточного елемента. У цьому випадку елемент належить до мінімальної множини уражених вузлів, визначеної формулою (2.33). НОС дозволяє оновлення, після чого виконується оновлення відповідного DOM-вузла. Показаний на рисунку алгоритм прийняття рішення реалізує означення оптимального оновлення (2.21), оскільки перерендеринг виконується виключно для тих елементів, для яких це є необхідним з точки зору формальної моделі. Блокування оновлення в інших випадках забезпечує локальність оновлень і мінімізацію кількості DOM-операцій.



Рис.2.12. Узагальнена схема алгоритму прийняття рішення НОС про перемальовування елемента: випадок 2

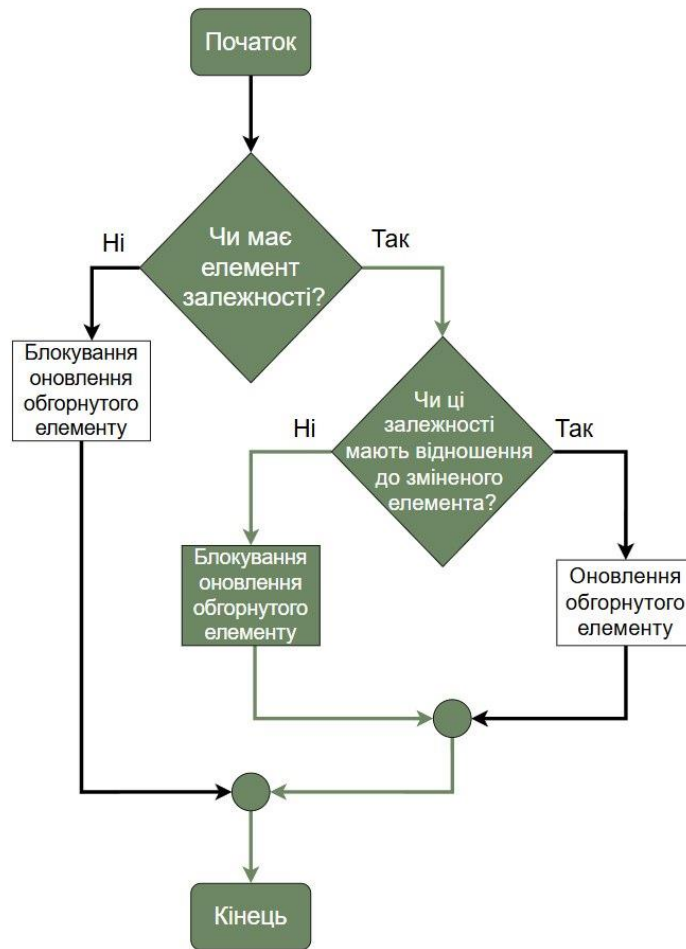


Рис.2.13. Узагальнена схема алгоритму прийняття рішення НОС про перемальовування елемента: випадок 3

У третьому випадку, рис.2.13, НОС містить залежності (Елемент №4.2), однак змінене значення не належить множині залежностей поточного елемента (залежність від Елементу №3.4). Формально це означає, що змінений вузол не входить до $score(\delta)$ для даного елемента. У такому разі НОС також блокує оновлення, запобігаючи надлишковим DOM-операціям.

Таким чином, рисунок 2.11 наочно ілюструє, що рішення про оновлення інтерфейсу приймається не на основі ієрархічної структури елементів, а на основі аналізу графа залежностей між значеннями даних. Саме цей алгоритм забезпечує ефективність запропонованого методу та відрізняє його від традиційних diffing-підходів.

Наявні підходи до оновлення користувацьких інтерфейсів, зокрема Virtual DOM-архітектури (React) та fine-grained реактивні системи (SolidJS, Svelte, сигнальні моделі), спрямовані на зменшення кількості надлишкових оновлень інтерфейсу шляхом оптимізації процесу оновлення. Водночас у більшості з них рішення щодо необхідності оновлення елемента приймається на рівні реалізації та не має чіткої формальної інтерпретації. Запропонований у роботі підхід відрізняється тим, що процес оновлення інтерфейсу формалізовано у вигляді задачі оптимізації на основі графа залежностей між значеннями даних. На відміну від diffing-підходів, де оновлення визначається порівнянням станів ієрархічних структур, та наявних fine-grained реалізацій, у яких залежності обробляються імпліцитно, у даній роботі введено область впливу зміни та мінімальну множину уражених вузлів, для яких оновлення є необхідним. Це дозволяє приймати рішення про оновлення елементів на основі формально визначених умов, незалежно від конкретної архітектури або інструментальної реалізації, та забезпечує алгоритмічно обґрунтовану локальність оновлень.

2.3. Моделювання прототипу програмного додатку для динамічного оновлення інтерфейсу вебзастосунків

Для побудови моделей прототипу програмного додатку та реалізації запропонованих методів та алгоритмів використаємо мову моделювання UML [118]. На рис.2.14 представлено UML-діаграму послідовності дій (Sequence Diagram), яка деталізує часову логіку та протокол взаємодії між компонентами системи управління станом під час оновлення даних. Вона описує алгоритм детермінованого оновлення інтерфейсу за допомогою алгоритму фільтрації подій.

Процес починається з асинхронної події від користувача (onChange), яка обробляється вхідним інтерфейсним компонентом (Текстове поле). Далі надсилається команда `dispatch(clearChangedId)` до Сховища, що через Ред'юсер встановлює значення `changedId = undefined`. Це необхідно для

очищення контексту попередніх транзакцій. Текстове поле ініціює *dispatch* (*updateValues(id, newValue)*), що призводить до мутації стану в Ред'юсері. На цьому етапі фіксуються нові дані та ідентифікатор зміненого елемента (*changedId = id*). Після оновлення Сховище розсилає сповіщення про зміну стану всім підписаним високорівневим компонентам (НОС-обгортка). Повідомлення містить як нові значення (*values*), так і маркер ініціатора зміни (*changedId*). НОС-обгортка виконує операцію перевірки належності ідентифікатора до множини залежностей компонента *changedId* $\in$ *dependencyList*.

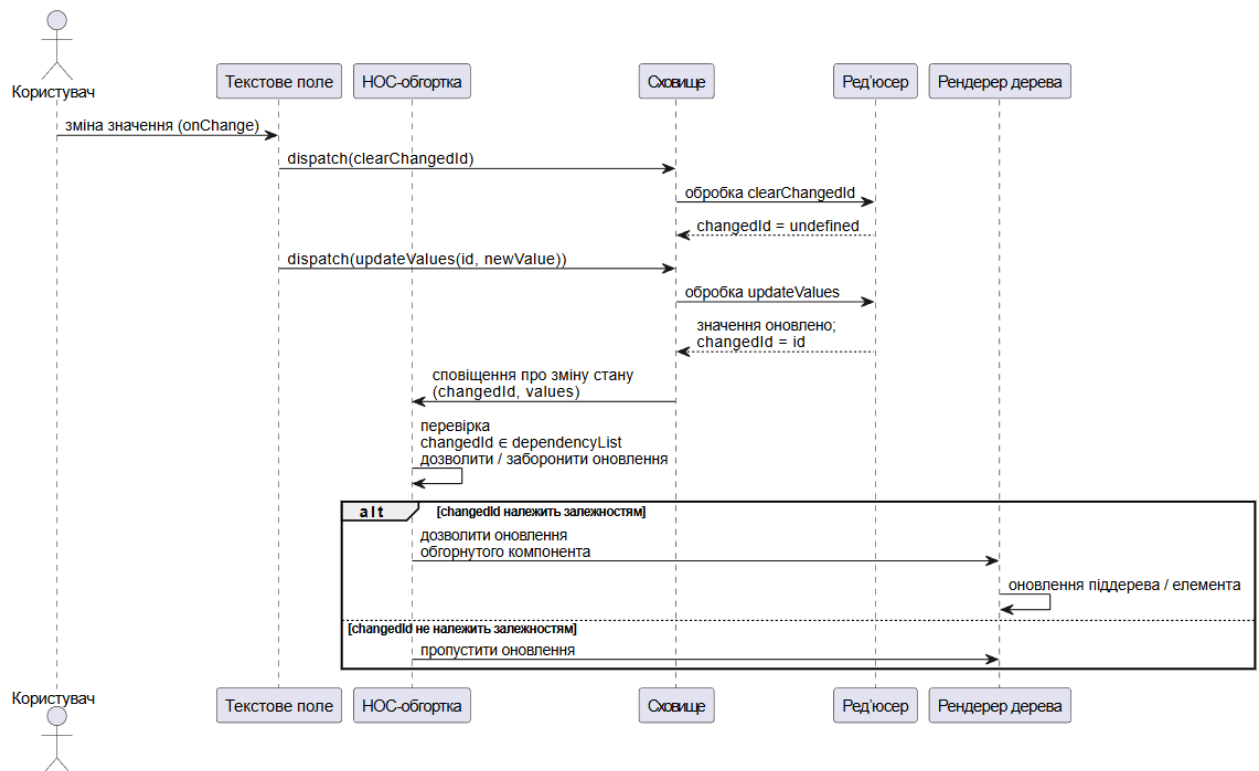


Рис.2.14. Діаграма послідовності дій

Умовне розгалуження реалізується блоком *alt*. Діаграма ілюструє механізм оптимізації швидкодії оновлення вебзастосунку через умовний оператор. Якщо *changedId* належить списку залежностей, НОС дозволяє повторне оновлення обгорнутого компонента, що активує Рендерер дерева для оновлення відповідного піддерева. Якщо ідентифікатор не знайдено у списку

залежностей, операція оновлення ігнорується («пропустити оновлення»), що заощаджує обчислювальні ресурси.

Дана послідовність реалізує патерн Granular Updates (гранульовані оновлення). На відміну від класичного підходу, де будь-яка зміна в Сховищі призводить до повного перерахунку дерева (Reconciliation), тут застосовано ідентифікаційну фільтрацію на рівні НОС. Це дозволяє досягти часової складності оновлення UI, що пропорційна лише кількості безпосередньо залежних елементів, а не загальному розміру дерева.

Наведена діаграма активності, рис.2.15, відображає формалізований алгоритм керування процесом повторного рендерингу вебзастосунку на основі механізму відстеження змін стану та списку залежностей. Вона моделює подієво-орієнтований цикл обробки оновлень, у межах якого рішення про необхідність оновлення візуального представлення приймається на основі перевірки релевантності змін.

Діаграма починається зі стартової події, яка ініціює життєвий цикл обробки компонента. Початковий етап має характер ініціалізації, тобто підготовки всіх необхідних структур даних, параметрів та підписок на джерела змін. На цьому етапі система перебуває у стані конфігураційного налаштування, коли формується контекст, у межах якого надалі буде прийматися рішення про доцільність оновлення інтерфейсу.

Першою операцією є отримання вхідних параметрів компонента – `props`. У контексті компонентно-орієнтованої архітектури `props` розглядаються як зовнішні параметри, що передаються компоненту з батьківського середовища або контейнера керування. Вони визначають початковий стан, конфігурацію відображення, логіку умовного відтворення та інші аспекти поведінки. Формально `props` можна розглядати як множину параметризованих значень, що утворюють вектор початкових умов функціонування компонента.

Наступним кроком є отримання списку залежностей – *dependencyList*. Цей список відіграє ключову роль у представленій моделі, оскільки він задає підмножину змінних стану або ідентифікаторів ресурсів, зміни яких мають

бути інтерпретовані як значущі для даного компонента. Інакше кажучи, *dependencyList* виконує роль фільтра чутливості

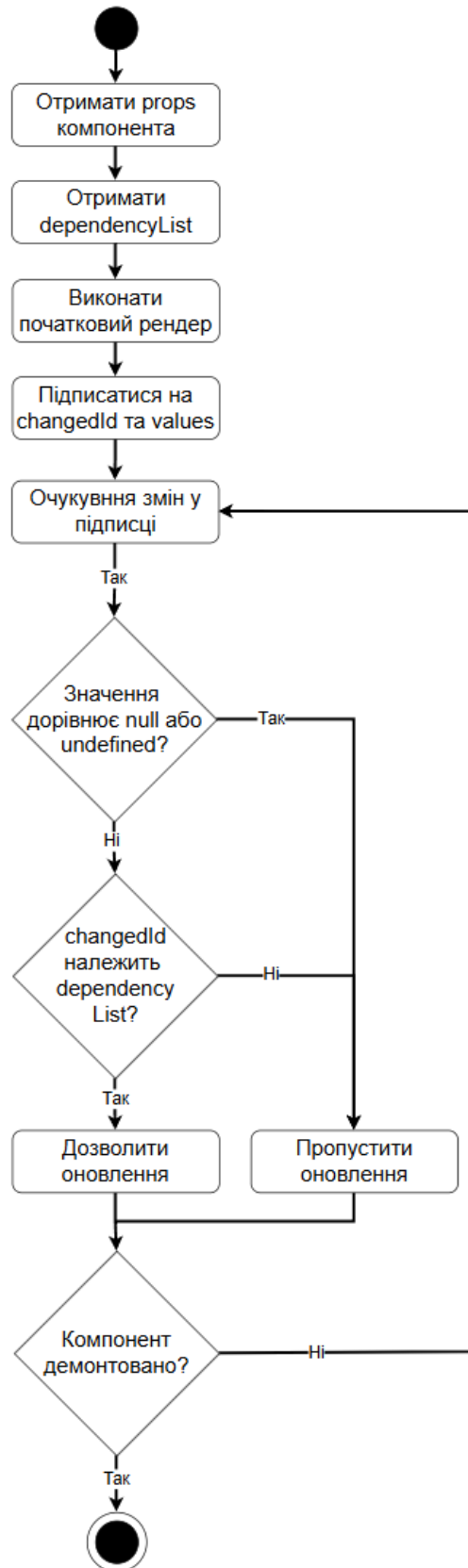


Рис.2.15. Діаграма активності для алгоритму «прийняття рішення про оновлення»

У формальному сенсі це множина ідентифікаторів, що визначає область спостереження компонента в просторі станів системи. Якщо позначити повний простір можливих змін як універсальну множину U , тоді *dependencyList* є підмножиною $D \subset U$, і лише події, що належать D , можуть потенційно спричинити оновлення інтерфейсу вебдодатку. Така селективність є основою оптимізації, оскільки дозволяє уникати зайвих обчислень.

Після формування списку залежностей виконується операція підписки на події змін. На діаграмі це представлено як підписка на *changedId* та *values*. Це означає, що компонент реєструється у механізмі сповіщень або шині подій і починає отримувати повідомлення про зміни стану. *Changedid* є ідентифікатором сутності або параметра, який було змінено, тоді як *values* містить нові значення або структуру даних після модифікації. Такий підхід відповідає патерну *observer* (спостерігач) або *publish–subscribe* (видавець-підписник), де компонент виступає як реактивний споживач подій. На цьому етапі формується канал реактивної взаємодії між джерелом стану та механізмом відображення.

Після завершення підписки система переходить у фазу очікування події зміни стану. Це концептуально важливий момент, оскільки він переводить алгоритм з імперативного послідовного режиму у подієво-орієнтований. Замість безперервного опитування стану використовується модель пасивного очікування, за якої виконання активується лише при надходженні сигналу. Такий механізм зменшує обчислювальне навантаження і відповідає принципам реактивного програмування. Очікування може бути реалізоване через асинхронні обробники, стріми подій, *callback*-функції або реактивні потоки.

Після надходження події виконується операція зчитування *changedId*. Це виявлення та аналіз ідентифікатора зміненої сутності з повідомлення про подію. Даний ідентифікатор є ключовим параметром для подальшого прийняття рішення. У формальній моделі він виступає як маркер, що відображає координату зміни у просторі станів. Саме *changedId*

використовується у наступних умовних перевірках, які визначають логіку переходів у діаграмі.

Перший умовний вузол перевіряє валідність ідентифікатора зміни. Формулювання умови – *changedId* дорівнює null або undefined – означає, що система повинна відфільтрувати події, які не мають коректного або визначеного джерела. Такі ситуації можуть виникати внаслідок службових сигналів, помилок передачі, ініціалізаційних імпульсів або неконсистентних оновлень. З точки зору формальної логіки це перевірка належності значення до множини допустимих ідентифікаторів. Якщо *changedId* не визначений, подія вважається непридатною для подальшої обробки в контексті оновлення інтерфейсу. Відповідна гілка алгоритму веде до дії «заборонити оновлення». Це означає, що візуальне оновлення не виконується, а система повертається до циклу очікування наступних подій. Така перевірка виконує роль захисного бар'єра, що забезпечує коректність виконання.

Якщо ж ідентифікатор зміни визначений, алгоритм переходить до другого умовного вузла, який перевіряє належність *changedId* до *dependencyList*. Цей етап є центральним у всій логічній конструкції. Він реалізує механізм семантичної релевантності змін. Перевірка *changedId* ∈ *dependencyList* є операцією належності елемента до множини. Якщо результат позитивний, зміна вважається значущою для компонента, оскільки вона торкається однієї з його залежностей. Якщо ж *changedId* не входить до списку залежностей, зміна визнається нерелевантною, і оновлення не повинне відбуватися. Таким чином, *dependencyList* виконує роль декларативного контракту чутливості компонента до змін стану.

У випадку, коли *changedId* належить *dependencyList*, активується гілка дозволеного оновлення. Спочатку виконується операція дозволу оновлення. Це логічне розблокування механізму оновлення представлення. У програмній реалізації це може відповідати встановленню прапорця, виклику методу оновлення або активації пайплайну перерахунку. Далі відбувається передача оновлених values. Це означає, що нові дані стану передаються у внутрішній

контекст компонента або безпосередньо в механізм шаблонізації. Передача значень може включати копіювання структур, нормалізацію даних, обчислення похідних параметрів або трансформації формату. Після цього виконується операція оновлення візуального відображення. Вона може включати повторне обчислення віртуального дерева представлення, диференційне порівняння з попереднім станом та мінімальне оновлення реального DOM або іншої цільової структури відображення. Цей етап є обчислювально найдорожчим, тому вся попередня логіка фільтрації спрямована на мінімізацію кількості його викликів.

Якщо ж *changedId* не входить до *dependencyList*, виконується альтернативна гілка – заборона оновлення. Вона симетрична до гілки невалідного ідентифікатора, однак має іншу семантику: подія є валідною, але не релевантною для даного компонента. Це означає, що зміна стосується іншої частини системи. Такий механізм особливо важливий у складних інтерфейсах із великою кількістю компонентів, де глобальний стан може змінюватися часто, але кожен окремий компонент залежить лише від обмеженого підмножини параметрів.

Після виконання будь-якої з гілок, дозволеного або забороненого оновлення, потоки керування сходяться у вузлі злиття. Це забезпечує структурну цілісність алгоритму та повернення до єдиної траєкторії виконання. Такий вузол є стандартним елементом діаграм діяльності та означає завершення умовного розгалуження. Після злиття виконується перевірка наявності наступних змін. Цей вузол реалізує ітеративну природу алгоритму. Система аналізує, чи існують нові події у черзі або потоці. Якщо такі події є, відбувається повернення до етапу очікування зміни стану, тобто цикл продовжується. Якщо ж подальші зміни відсутні, процес завершується фінальним вузлом.

З точки зору теорії обчислювальних процесів дана модель є циклічним подієвим автоматом із фільтрацією за предикатами. Стан автомата визначається набором підписок, списком залежностей і поточним контекстом

значень. Перехідні функції визначаються результатами логічних перевірок. Алгоритм має властивості детермінованості за фіксованого набору вхідних подій і *dependencyList*. Він також демонструє властивість локальності впливу: лише події з обмеженої множини можуть викликати дорогі операції оновлення.

Архітектурно така схема підтримує принципи ефективності, масштабованості та модульності. Ефективність досягається шляхом уникнення надлишкових оновлень інтерфейсу. Масштабованість забезпечується тим, що кожен компонент має власний список залежностей і власний контур реактивності. Модульність реалізується через ізоляцію логіки прийняття рішення про оновлення в межах компонента. Крім того, алгоритм добре узгоджується з концепцією чистих функцій від стану до представлення, де оновлення інтерфейсу вебзастосунків розглядається як детермінована функція від релевантних даних.

У підсумку, діаграма формалізує оптимізований цикл реактивного оновлення інтерфейсу вебзастосунків, що базується на трьох ключових механізмах: декларативному визначенні залежностей, подієвій моделі сповіщення та умовній фільтрації релевантності змін. Така модель дозволяє мінімізувати обчислювальні витрати, підвищити передбачуваність поведінки інтерфейсу та забезпечити контрольований життєвий цикл оновлення візуального представлення компонента.

Діаграма класів для контрактів властивостей елементів представлено на рис. А.1 додатку А. Ця діаграма класів відображає структуру контрактів властивостей елементів інтерфейсу та задає єдину модель опису полів, їх типів і взаємозв'язків. У центрі схеми знаходиться базовий елемент *BasicField*, який містить спільні для всіх елементів атрибути: ідентифікатор, тип, назву та необов'язковий список залежностей. Від нього наслідуються конкретні класи елементів, зокрема *TextInput*, *NumberInput*, *Toggle*, *Dropdown*, *Folder* і *Table*. Для них визначено спеціалізовані властивості, наприклад значення за

замовчуванням, набір опцій для випадяючого списку, дочірні елементи для контейнера *Folder* та колонки для *Table*.

Клас *Dependency* задає, від якого елемента залежить поле, а клас *Rule* формалізує умову перевірки через оператор, значення та результат, яким може бути *hide* або *visible*. Переліки *FieldType*, *Operator* і *Result* стандартизують допустимі типи полів, операції порівняння та результати виконання правил. Для представлення поточного стану структури використано *StructureSliceState*, що містить дерево елементів, поточні та початкові значення, а також ідентифікатор зміненого елемента, тоді як *ValueData* і *TableValue* задають формат зберігання значень у звичайних і табличних елементах.

На рис. А.2, додатку А представлено діаграму класів для контрактів властивостей компонентів (props). Ця діаграма класів відображає контракти властивостей компонентів, що використовуються для передавання даних у відповідні елементи інтерфейсу. Для кожного типу компонента визначено окремий набір props, зокрема *ToggleProps*, *TextElementProps*, *NumberElementProps*, *DropdownElementProps*, *TableElementProps* і *FolderElementProps*. У їх складі задаються структурні дані елемента (*treeData*), поточні значення (*valueData*), а для вкладених у таблицю елементів додатково можуть використовуватися *rowData* та ознака *inTable*. Такий підхід забезпечує типізоване й уніфіковане передавання параметрів до компонентів залежно від їх функціонального призначення.

Окремо на діаграмі подано узагальнені контракти для оптимізаційних механізмів, зокрема *OptimizationMethodProps* і *MemoizedComponentProps*, які орієнтовані на роботу з дочірніми елементами, табличними значеннями та загальною структурою даних. Наявність інтерфейсу *UpdateValueFunction* вказує на використання єдиного механізму оновлення значень для різних типів елементів. У цілому діаграма демонструє, що архітектура компонентів побудована на чітко типізованих контрактах, які забезпечують узгодженість між структурою даних, станом інтерфейсу та процедурами оновлення.

Висновки до другого розділу

Сформовано формальну модель представлення глибоко вкладених структур даних та процесу їх оновлення, що дозволяє описати оновлення інтерфейсу як задачу оптимізації кількості DOM-операцій.

Запропоновано граф залежностей на рівні окремих значень даних як основний інструмент визначення мінімальної множини вузлів, які необхідно оновлювати при зміні стану. Запровадження графа залежностей дозволяє точно визначити область впливу змін і мінімізувати кількість вузлів, що потребують оновлення.

Формально доведено, що *fine-grained* реактивний підхід забезпечує не гіршу, а в середньому кращу асимптотичну ефективність порівняно з традиційними *diffing*-архітектурами.

Розроблено метод точкового оновлення інтерфейсу вебзастосунку, який алгоритмічно реалізує запропоновану теоретичну модель та створює основу для подальшої експериментальної перевірки ефективності, демонструє суттєві переваги над традиційними підходами за рахунок використання локальності оновлень та *fine-grained* реактивності.

На відміну від глобального оновлення, запропонований метод дозволяє оновлювати лише ті елементи, які дійсно залежать від змінених даних. Використання НОС-компонентів і централізованого сховища забезпечує контрольований механізм відстеження змін і прийняття рішень щодо оновлення. Це дозволяє значно зменшити навантаження на систему та підвищити продуктивність вебзастосунків. У підсумку, метод забезпечує ефективну обробку складних структур із великою кількістю залежностей.

Моделювання прототипу програмного додатку підтвердило ефективність запропонованого підходу на практиці та продемонструвало його відповідність теоретичним положенням. Реалізований алгоритм подієво-орієнтованої обробки змін із використанням списків залежностей дозволяє здійснювати точкове оновлення інтерфейсу. Введення перевірок

релевантності змін (через `dependencyList` та `changedId`) забезпечує фільтрацію подій і запобігає зайвим оновленням. Така архітектура сприяє підвищенню масштабованості, стабільності та передбачуваності роботи системи, також є доцільним для використання у сучасних високонавантажених вебзастосунках.

Результати дослідження представлені у роботах [95;97;98;100;101] здобувача.

РОЗДІЛ 3. Реалізація та оцінка якості прототипу програмного забезпечення

3.1. Реалізація прототипу програмного засобу оновлення інтерфейсу вебзастосунку з використанням стеку фронтенд технологій

У межах дисертаційного дослідження розроблено прототип програмного забезпечення призначений для експериментальної перевірки моделей, методів і алгоритмів оновлення інтерфейсу вебзастосунків, що працюють з об'єктами з глибокою вкладеністю та множинними міжелементними залежностями. Реалізацію здійснено із використанням сучасного фронтенд-стеку TypeScript, JavaScript, React, Redux Toolkit, HOC, мемоізації, Vite. Основною метою створення прототипу є практичне підтвердження можливості зменшення кількості повторних оновлень компонентів інтерфейсу без порушення коректності візуального відображення та логіки взаємодії з користувачем.

Особливістю досліджуваного класу вебзастосунків є використання деревовидних структур даних, у яких кожен вузол може представляти як окремий елемент інтерфейсу (текстове поле, перемикач, випадаючий список), так і контейнер (папка, таблиця), що включає вкладені елементи. При цьому між елементами можуть існувати логічні залежності, які визначають умови їх відображення або доступності. У таких умовах стандартні механізми керування станом у клієнтських фреймворках часто призводять до надлишкових повторних оновлень інтерфейсів, що негативно впливає на продуктивність вебзастосунку.

Прототип реалізовано як клієнтський вебзастосунок, у якому структура інтерфейсу задається декларативно у вигляді JSON-об'єкта, а стан усіх елементів централізовано зберігається у глобальному сховищі. Повторне оновлення компонентів відбувається лише у разі, якщо зміна стану безпосередньо впливає на відповідний елемент. Таким чином, прототип виступає експериментальним середовищем для дослідження впливу

локалізації оновлень на загальну продуктивність системи оновлення інтерфейсу вебзастосунку.

Декларативний опис структури у форматі JSON дозволяє відокремити логіку опису інтерфейсу від логіки його візуалізації, що є принципово важливим для дослідження алгоритмів оновлення інтерфейсів. Контейнерні елементи (папки, таблиці) додатково містять вкладені елементи, що формує ієрархічне дерево довільної глибини. Залежності між елементами описуються через посилання на ідентифікатори інших вузлів та набір правил, які визначають поведінку елемента при зміні стану залежного вузла (наприклад, приховування або відображення). Архітектура клієнтської частини прототипу побудована за принципами компонентно-орієнтованого програмування з централізованим керуванням станом. Ключовими архітектурними компонентами є модель структури інтерфейсу, централізоване сховище стану, набір універсальних візуальних компонентів та механізм контролю повторного оновлення інтерфейсу. Оптимізована структурна схема оновлення деревовидного інтерфейсу з використанням НОС-обгортки представлена на рис.2.6. Інтерфейс прототипу програмного засобу на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками представлено на рис.3.1, а лістинг програми розміщено в додатку Б.

Централізоване сховище стану використовується як єдине джерело істини, що забезпечує консистентність даних у межах усього інтерфейсу. Будь-яка взаємодія користувача з елементами призводить до зміни стану у сховищі, після чого ініціюється механізм перевірки залежностей і приймається рішення щодо повторного оновлення. Для керування станом у прототипі використовується глобальне сховище, у якому зберігаються структура інтерфейсу, значення всіх елементів та ідентифікатор останнього зміненого елемента. Ключовим елементом цієї моделі є маркер зміни, який дозволяє однозначно визначити, який саме елемент ініціював оновлення стану. Такий підхід дозволяє відмовитися від повного аналізу дерева при кожній зміні та зосередитися лише на релевантних вузлах.

Ключовою особливістю реалізації прототипу є використання універсальної обгортки компонентів, яка виконує функцію посередника між глобальним станом і візуальним компонентом. Така обгортка отримує інформацію про зміну стану, володіє списком залежностей відповідного елемента і приймає рішення щодо дозволу або заборони повторного оновлення інтерфейсу.

Папка: Налаштування

Ім'я адміна

Фамілія адміна

Вік адміна

Активувати додаткові налаштування ☒

Папка: Додаткові налаштування

Області

Активна область	Назва	Населення (млн.)	Локація
☒	Вінницька	1,53	Центр
☐	Волинська	1,02	Захід
☒	Дніпропетровська	3,15	Схід
☐	Донецька	4,1	Схід
☒	Житомирська	1,21	Північ

Папка: Налаштування доступу до вибірки планет

Венера ☒

Земля ☒

Нептун ☒

Сатурн ☐

Уран ☐

Плутон ☐

Папка: Налаштування доступу до вибірки статі

Чоловік ☒

Жінка ☒

Інше ☐

Користувачі

Активний користувач	Ім'я	Вік	Стать	Місце знаходження	Машина	Діти
☒	Іван	31	Чоловік	Венера	BMW	3
☒	Ірина	24	Жінка	Виберіть значення	Audi	5
☒	Сергій	60	Жінка	Нептун	Aston Martin	4
☒	Павло	65	Чоловік	Нептун	Ferrari	4
☐	Євген	43	Жінка	Виберіть значення	Ferrari	3

Рис.3.1. Інтерфейс прототипу програмного засобу на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками

Обгортка інкапсулює логіку аналізу залежностей, що дозволяє зберегти візуальні компоненти простими та не перевантаженими умовною логікою. Кожен компонент інтерфейсу розглядається як автономний спостерігач, який реагує лише на релевантні для нього зміни. Такий підхід забезпечує локалізацію повторних оновлень, що є принципово важливим для інтерфейсів з великою кількістю елементів і складними залежностями. З формальної точки зору задача прийняття рішення про повторне оновлення інтерфейсу вебзастосунку зводиться до перевірки належності ідентифікатора зміненого елемента множині залежностей поточного компонента.

Наведений фрагмент коду, лістинг 3.1, реалізує спеціалізований підхід до керування оновлення компонентів у React-застосунку з використанням Redux, орієнтований на роботу зі складними деревоподібними структурами даних та логічними залежностями між елементами інтерфейсу. Основна ідея полягає в тому, щоб ізолювати повторні оновлення лише тими компонентами, для яких зміна даних є семантично значущою, а не примушувати до оновлення весь піддеревний фрагмент інтерфейсу. Такий підхід особливо актуальний у випадках, коли форма або UI-дерево містить велику кількість елементів, взаємопов'язаних умовами відображення, валідації або обчислюваних значень.

Лістинг 3.1. – Реалізація НОС-обгортки для точкового контролю повторного оновлення компонентів

```
const withDependencyWatcher = (component: any) => {
  const MemorizedComponent = memo(component, lodash.isEqual);
  return (props: MemorizedComponentProps): any => {
    const [localProps, setLocalProps] = useState(props);
    const [dependencyList] = useState<string[]>(getDependencyList(props.treeData));
    const changedId = useAppSelector(state => state.structure.changedId);
    const newValueData = useAppSelector(state => state.structure.values);
    const dispatch = useAppDispatch();
    const updateValue = useCallback((id: string | ComplexId, newValue: string | number |
boolean) => {
      dispatch(clearChangedId());
      setTimeout(() => {
        let changedId = id;
        if (props.inTable && props.rowData) {
          const tableindex = (props.valueData[props.treeData.id] as
TableValue[]).findIndex(data => data.id === props.rowData.id);
```

```

        const complexId = { tableId: props.treeData.id, elementId: id as string, rowIndex:
tableindex };
        changedId = complexId;
    }
    dispatch(updateValues({ id: changedId, newValue }));
  }, 0);
}, [dispatch]);
useEffect(() => {
  if (newValueData && changedId && dependencyList.includes(changedId)) {
    setLocalProps({ ...props, valueData: newValueData });
  }
}, [changedId]);
return <MemorizedComponent {...localProps} updateValue={updateValue}/> ;
}
};
export default withDependencyWatcher;

```

Функція *withDependencyWatcher* є прикладом Higher-Order Component, тобто функції вищого порядку, яка приймає React-компонент і повертає новий компонент із розширеною поведінкою. Вхідний компонент не змінюється безпосередньо, натомість він «обгортається» додатковим рівнем логіки, який відповідає за відстеження змін у глобальному стані та за вирішення питання, чи повинні ці зміни призвести до оновлення конкретного екземпляра компонента. Такий підхід добре узгоджується з принципами композиції в React і дозволяє відокремити інфраструктурну логіку оптимізації від безпосередньої логіки відображення.

На початковому етапі всередині *withDependencyWatcher* створюється мемоізована версія переданого компонента за допомогою *React.memo* та кастомного компаратора *lodash.isEqual*. Використання глибокого порівняння пропсів тут має принципове значення, оскільки в контексті складних форм пропси часто містять вкладені об'єкти, масиви або структури зі змінною глибиною. Поверхнєве порівняння у таких випадках не дозволило б коректно визначити еквівалентність станів, тоді як *isEqual* забезпечує більш точну семантичну перевірку. Хоча такий підхід може бути обчислювально дорожчим, він виправданий у сценаріях, де вартість зайвого оновлення значно перевищує вартість порівняння.

Компонент, який повертається з *withDependencyWatcher*, є функціональним компонентом, що використовує хуки React для керування локальним станом і побічними ефектами. Локальний стан *localProps* ініціалізується вхідними пропсами і надалі використовується як єдине джерело даних, що передається у мемоізований компонент. Це рішення дозволяє відокремити пропси, які приходять ззовні, від пропсів, що фактично використовуються для оновлення, і оновлювати останні лише за певних умов.

Окрему роль відіграє формування списку залежностей *dependencyList*. Він обчислюється один раз під час ініціалізації компонента на основі *treeData*, що описує вузол або піддерево UI-структури. Логіка побудови цього списку винесена у функцію *getDependencyList*, яка концептуально формує перелік ідентифікаторів полів, зміна яких може вплинути на поточний компонент. Фіксація цього списку в *useState* без подальшого оновлення підкреслює припущення, що структура залежностей для конкретного вузла є статичною протягом життєвого циклу компонента.

Компонент підписується на два значення зі сховища *Redux*: *changedId* і *values*. Перше використовується як маркер останньої зміни у структурі даних, друге містить актуальні значення всіх полів. Такий підхід дозволяє мінімізувати обсяг інформації, що аналізується при кожному оновленні стану: замість повного порівняння дерева значень достатньо перевірити, який саме ідентифікатор був змінений.

Функція *updateValue*, оголошена з використанням *useCallback*, інкапсулює логіку оновлення значень у *Redux*. Вона спочатку очищає попередній *changedId*, а потім, із мінімальною асинхронною затримкою, формує новий ідентифікатор зміни та диспатчить екшен *updateValues*. Наявність *setTimeout* з нульовою затримкою тут використовується як спосіб розвести у часі два стани: скидання ідентифікатора зміни та встановлення нового. Це дозволяє коректно зафіксувати факт зміни навіть у випадках, коли оновлення відбуваються дуже швидко або для одного й того самого поля.

Особливу увагу приділено сценарію роботи з табличними даними. Якщо компонент знаходиться всередині таблиці, ідентифікатор зміни перетворюється на складену структуру *ComplexId*, яка містить ідентифікатор таблиці, ідентифікатор елемента та індекс рядка. Це рішення дозволяє точно локалізувати зміну в межах багатовимірної структури даних і уникнути хибних спрацьовувань залежностей для інших рядків або колонок.

Ключова логіка оптимізації реалізується в *useEffect*, який реагує на зміну *changedId*. Усередині перевіряється, чи входить змінений ідентифікатор до списку залежностей поточного компонента. Лише за цієї умови локальні пропси оновлюються новими значеннями зі сховища. Якщо ж зміна не має відношення до поточного вузла, оновлення не відбувається, навіть попри те, що глобальний стан Redux був оновлений. Таким чином, реалізується вибіркове реагування на зміни, що суттєво знижує кількість непотрібних оновлень інтерфейсу.

Фінальним кроком є оновлення мемоізованого компонента з використанням *localProps* та передачею йому функції *updateValue*. З точки зору дочірнього компонента, він працює у звичайному режимі, не маючи інформації про складну інфраструктуру відстеження залежностей. Це відповідає принципу інкапсуляції та дозволяє використовувати *withDependencyWatcher* повторно для різних типів елементів без зміни їхньої внутрішньої реалізації.

У підсумку, лістинг 3.1 та код додатку Б, демонструє інженерно виважений підхід до оптимізації оновлення інтерфейсу в React-застосунках із глибоко вкладеними структурами та складною логікою залежностей. Поєднання мемоізації, явного маркера змін, локального кешування пропсів і контекстної інтерпретації ідентифікаторів дозволяє досягти значного виграшу в продуктивності без порушення декларативної моделі React і без ускладнення бізнес-логіки базових компонентів.

У наведеній реалізації поєднано два рівні оптимізації. Перший рівень (локалізація оновлень props): `setLocalProps({...props, valueData:`

`newValueData`)) викликається лише коли `changedId ∈ dependencyList`. Тобто більшість компонентів отримує новий глобальний `state`, але не оновлює власні `props`, отже, не провокує повторне оновлення. Другий рівень (фільтрація ререндеру на рівні `React.memo`): навіть якщо локальні `props` незначно змінені, `lodash.isEqual` може не пропустити оновлення, якщо структура `props` не відрізняється. Це дає ефект «децентралізованого керування оновленням (перерендерингом)»: рішення приймається на рівні кожного вузла без глобального проходження дерева.

Алгоритм прийняття рішення про повторне оновлення представлено у вигляді рисунку 2.9. У першому випадку НОС отримав сповіщення, про зміну сховища, але на етапі першої перевірки не дозволив обгорнутому елементу оновитись, адже список залежностей відсутній. У другому випадку НОС отримав сповіщення, про зміну сховища, перший етап перевірки виявив наявність залежностей і другий етап підтвердив, що залежності зв'язані. Пройшовши дві перевірки НОС дозволяє обгорнутому елементу зробити оновлення. В останньому випадку НОС отримав сповіщення, про зміну сховища, перший етап перевірки виявив наявність залежностей, але другий етап перевірки виявив, що залежності не зв'язані. В цьому випадку НОС блокує рендер обгорнутого елемента.

Папка є рекурсивним контейнером і демонструє ключову властивість системи: дерево інтерфейсу обходиться зверху вниз, але оновлення конкретного вузла визначається правилами видимості та НОС-обгорткою. Реалізація контейнерного компонента з рекурсивним оновленням вкладених елементів представлено у лістингу 3.2.

Лістинг 3.2 – Реалізація контейнерного компонента з рекурсивним оновленням вкладених елементів

```
const FolderElement = ({treeData, valueData}: FolderElementProps) => {
  const folderVisible = isElementVisible(treeData, valueData);
  return (
    <>
      {folderVisible &&
        <div className="folder">
          <div className="folder-label">{`Папка: ${treeData.label}`}</div>
```

```

        {treeData.chilids.map(child => renderChildElements(child, valueData))}
      </div>
    }
  </>
);
};
export default withDependencyWatcher(FolderElement);

```

Тут видно важливий принцип, а саме, компонент не підписується на глобальне *values* напряму, а отримує *valueData* через обгортку тільки у релевантних випадках. Окремим важливим рішенням є винесення логіки вибору компонента за типом вузла в єдину функцію. Це зменшує зв'язність і робить систему розширюваною, а саме, додавання нового типу поля зводиться до додавання нового *case*.

TextElement демонструє розмежування локального UI-стану (*localValue*) і глобального стану (Redux *values*) (лістинг 3.4). Локальне значення забезпечує швидку реакцію інпуту, а глобальна синхронізація виконується через *updateValue*, який передається через НОС.

Лістинг 3.4 – Реалізація атомарного текстового елемента інтерфейсу з локальним керуванням станом та інтеграцією з алгоритмом точкового оновлення

```

const TextElement = ({treeData, inTable, valueData, updateValue, rowData}:
TextElementProps) => {

  const valueFromValueData = getValueFromValueData(treeData, rowData || valueData) as
string | undefined;
  const [localValue, setLocalValue] = useState(valueFromValueData);
  const handleChange = (e: React.ChangeEvent<HTMLInputElement >) => {
    const newValue = e.target.value;
    setLocalValue(e.target.value);
    updateValue(treeData.id, newValue);
  };
  return (
    <div className={`text-wrapper ${inTable ? 'text-wrapper-for-table' : ''}`>
      {!inTable && <span>{treeData.label}</span>}
      <input type="text" value={localValue} onChange={handleChange} className="text-input"
    />
    </div>
  );
};
export default withDependencyWatcher(TextElement);

```

Здійснено детальний розгляд практичної реалізації прототипу програмного забезпечення, що втілює запропоновані у дисертаційному дослідженні моделі, методи та алгоритми точкового оновлення інтерфейсів вебзастосунків із деревовидною структурою даних і глибокою вкладеністю елементів. Реалізація прототипу підтвердила можливість формування інтерфейсу на основі декларативного опису структури та залежностей, що дозволяє відокремити логіку представлення від логіки обробки стану та створює передумови для масштабування системи без ускладнення програмного коду візуальних компонентів.

Запропонований механізм централізованого керування станом у поєднанні з децентралізованим контролем повторного оновлення забезпечує локалізацію обчислювальних витрат і зменшення кількості надлишкових оновлень інтерфейсу. Формалізація залежностей у вигляді множин для кожного елемента та їх інтерпретація як орієнтованого графа дозволили звести задачу прийняття рішення про повторне оновлення до перевірки релевантності конкретної зміни, що принципово відрізняється від традиційних підходів із глобальним оновленням дерева компонентів. Особливу увагу приділено підтримці складених структур, зокрема табличних елементів, для яких розроблено уніфікований механізм ідентифікації, що забезпечує коректну інтеграцію таких елементів у загальну модель залежностей.

Таким чином, реалізований прототип не лише відтворює теоретичні положення, сформульовані у попередніх розділах, але й демонструє їх практичну придатність для використання у реальних веб-застосунках зі складною логікою взаємодії елементів. Отримані результати створюють надійну основу для подальшої експериментальної оцінки ефективності запропонованого підходу та кількісного аналізу його впливу на продуктивність оновлення інтерфейсу, що є предметом розгляду наступних підрозділів дисертаційної роботи.

3.2. Архітектурні та програмні аспекти впровадження прототипу

Упровадження розробленого прототипу програмного забезпечення здійснювалося з урахуванням специфіки клієнтського браузерного середовища, яке характеризується обмеженими обчислювальними ресурсами, асинхронною моделлю виконання коду та необхідністю забезпечення високої реактивності інтерфейсу. На відміну від серверних застосунків, клієнтські веб-системи виконуються безпосередньо на пристроях користувачів, що зумовлює залежність продуктивності від характеристик апаратного забезпечення, реалізації браузера та версії JavaScript-движка. У таких умовах особливого значення набуває мінімізація кількості операцій, пов'язаних з оновленням інтерфейсу, оскільки надмірні оновлення безпосередньо впливають на час відгуку системи та суб'єктивне сприйняття її швидкодії.

Браузерне середовище накладає обмеження на тривалість безперервного виконання JavaScript-коду, оскільки тривалі обчислення можуть блокувати головний потік і призводити до «заморожування» інтерфейсу. З цієї причини архітектура прототипу була спроектована таким чином, щоб уникати глобальних операцій над усією структурою інтерфейсу при кожній зміні стану. Запропонований підхід локалізації оновлення дозволяє зменшити навантаження на головний потік виконання за рахунок обмеження кількості компонентів, для яких виконується оновлення.

Важливою вимогою до середовища виконання є також коректна підтримка сучасних можливостей мови JavaScript, зокрема замикань, функціональних компонентів та механізмів мемоізації. Використання сучасних JavaScript-движків забезпечує достатній рівень оптимізації виконання коду, що є ключовим для ефективної роботи механізмів глибокого порівняння пропсів і обробки складних структур стану. У контексті роботи з великими обсягами DOM-вузлів важливим є також ефективне використання віртуального DOM, яке дозволяє мінімізувати кількість прямих маніпуляцій із реальною DOM-структурою браузера.

Особливістю досліджуваних інтерфейсів є значна кількість однотипних елементів, зокрема у складі табличних структур і рекурсивних контейнерів. У таких умовах будь-яке неконтрольоване оновлення призводить до пропорційного зростання кількості операцій над DOM-вузлами. Саме тому вимоги до середовища виконання безпосередньо вплинули на архітектурні рішення прототипу, орієнтовані на мінімізацію кількості реальних змін у DOM і збереження стабільної продуктивності інтерфейсу.

Розроблений прототип орієнтований на використання як інтегровний модуль у складі реальних веб-застосунків, що формують інтерфейс динамічно на основі опису структури та залежностей. Декларативний характер JSON-опису інтерфейсу дозволяє розглядати прототип як універсальний механізм побудови користувацького інтерфейсу, який може бути підключений до наявних систем без необхідності глибокої модифікації їхньої внутрішньої логіки. У такому сценарії серверна частина застосунку відповідає за формування або зберігання структури інтерфейсу, тоді як клієнтська частина виконує його інтерпретацію та оптимізоване оновлення. Проведемо опис процесу перетворення структури даних у DOM-дерево та організацію оновлення інтерфейсу. Представимо поточне віртуальне DOM-дерево, яке є абстрактним представленням інтерфейсу:

```
const OLD_DOM = {
  data: {
    id: 1,
    name: 'A',
    type: 'div',
    value: undefined,
    childs: [
      {
        id: 2,
        name: 'B',
        type: 'toggle',
        value: false,
        disabled: false,
        childs: [
          {
            id: 3,
            name: 'C',
            type: 'button',
            value: false,
```

```

        disabled: false
      },
      {
        id: 4,
        name: 'D',
        type: 'input',
        value: false,
        disabled: false
      }
    ]
  }
]
}
};

```

Воно формується на основі виклику компонента (позначено як $A \rightarrow B \rightarrow C, D$) і містить тип елемента (div, button, toggle, input), набір властивостей (props), вкладені дочірні елементи, статичні та динамічні значення. Це дерево ще не пов'язане з реальним DOM – воно існує лише в пам'яті. Після створення віртуального дерева система переходить до етапу commit, де створюються реальні DOM-елементи, встановлюються атрибути та властивості, додаються обробники подій, елементи вставляються у документ. На цьому етапі браузер виконує layout, paint і відображає інтерфейс.

Коли відбувається зміна стану, наприклад, виклик `setState(true)`, рис.3.2 компонент В викликається повторно для формування нового віртуального DOM, який відображає актуальний стан інтерфейсу. Компонент генерує нове дерево, яке може відрізнитися від попереднього лише окремими властивостями.

Знизу на рис.3.2, представлено блок «Порівняння віртуальних DOM», де зображено два стани: попередній та новий. На цьому етапі виконується алгоритм diffing – порівняння двох дерев віртуального DOM з метою визначення відмінностей між ними. Червоним виділено змінені елементи, що підлягають оновленню. Це дозволяє системі визначити мінімальний набір операцій, необхідних для синхронізації реального DOM зі станом даних. Після виявлення змін відбувається етап «Виклик diff для формування нового знімка», який формує список необхідних оновлень. Ці зміни передаються далі у вигляді набору інструкцій для оновлення DOM, так зокрема при виявленій

зміні в результаті порівняння сформовано `document.getElementById(2).value = true`. Після чого формується виклик `C` для формування нового зліпка, враховуючи ієрархічну структуру.

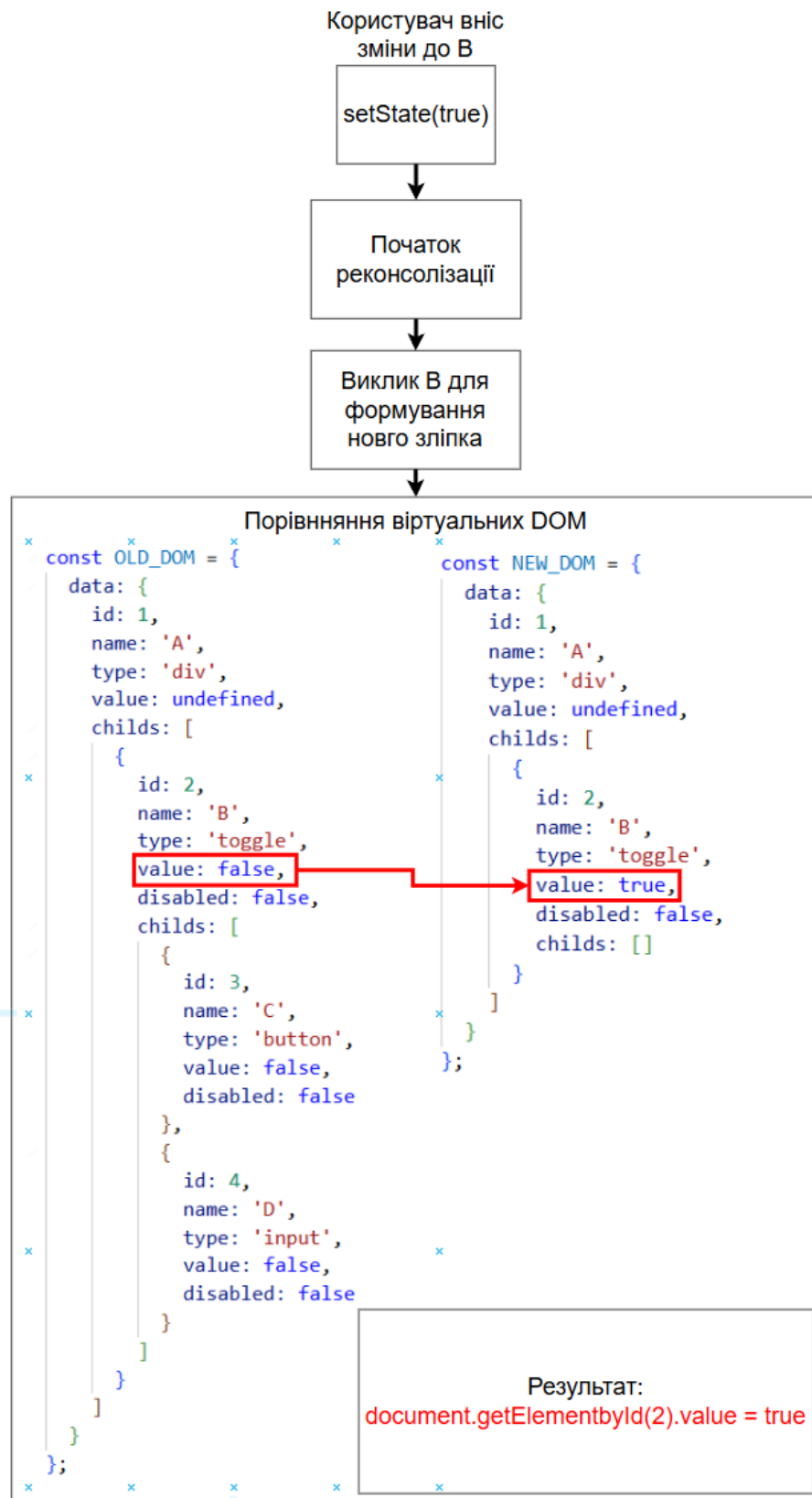


Рис.3.2 Схема ініціалізації змін та порівняння віртуальних DOM

На рис.3.3 показано порівняння двох дерев віртуального DOM та визначення відмінностей між ними, що виділені червоним. Враховуючи наявність змін формується інструкція `document.getElementById(3).disabled = true`.

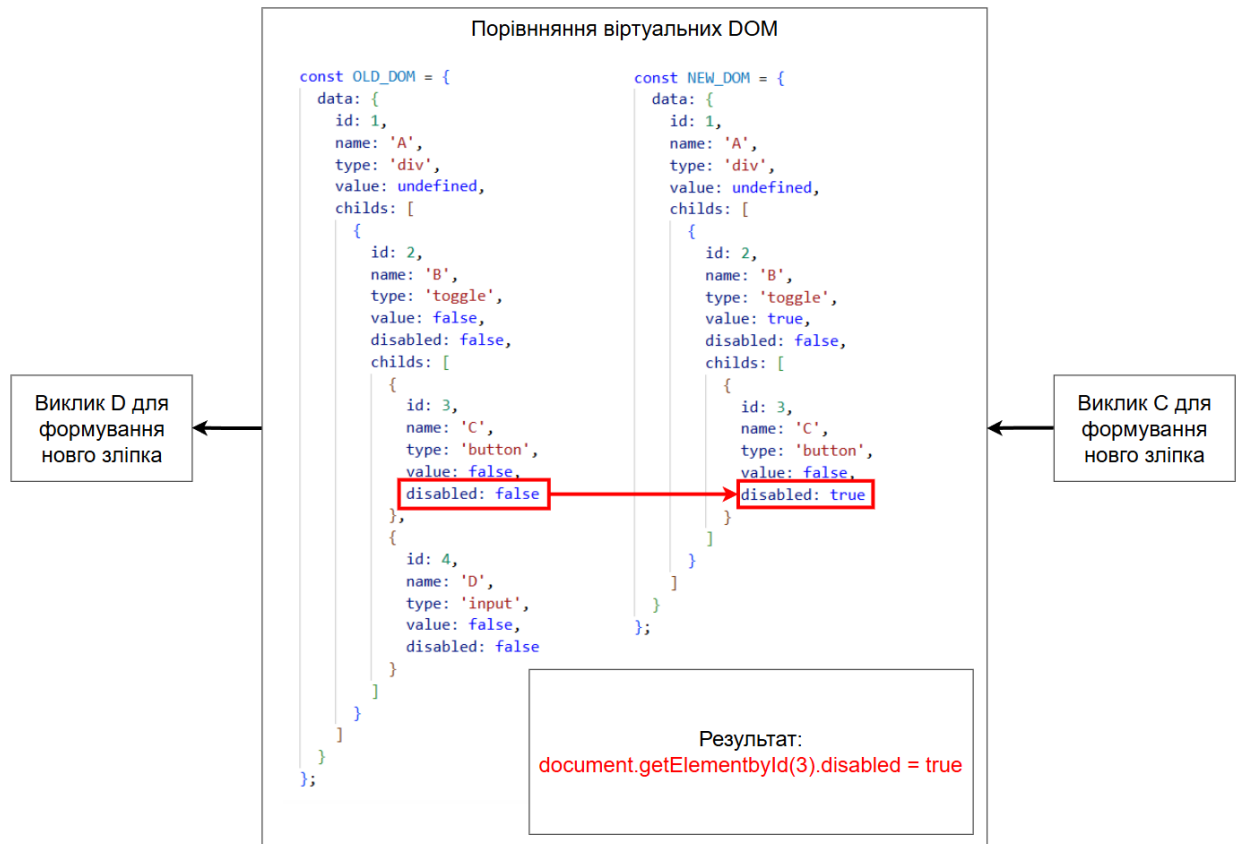


Рис. 3.3. Порівняння віртуального DOM після виклику C для формування нового зліпка

Враховуючи структуру та наявність зв'язків, відбувається виклик D для формування нового зліпка, рис.3.4. Проводиться порівняння віртуальних DOM, в результаті якого видно, що змін елементу D не відбулося. Відповідно файл з інструкціями не сформовано.

Інтеграція прототипу передбачає адаптацію формату вхідних даних до очікуваної моделі, зокрема уніфікацію ідентифікаторів елементів і правил залежностей. Такий підхід дозволяє використовувати різні джерела даних для формування інтерфейсу, включаючи конфігураційні файли, відповіді REST- або GraphQL-сервісів, а також локальні сховища. При цьому зміна структури

інтерфейсу не потребує перекомпіляції клієнтського коду, що є суттєвою перевагою для систем, які активно еволюціонують.

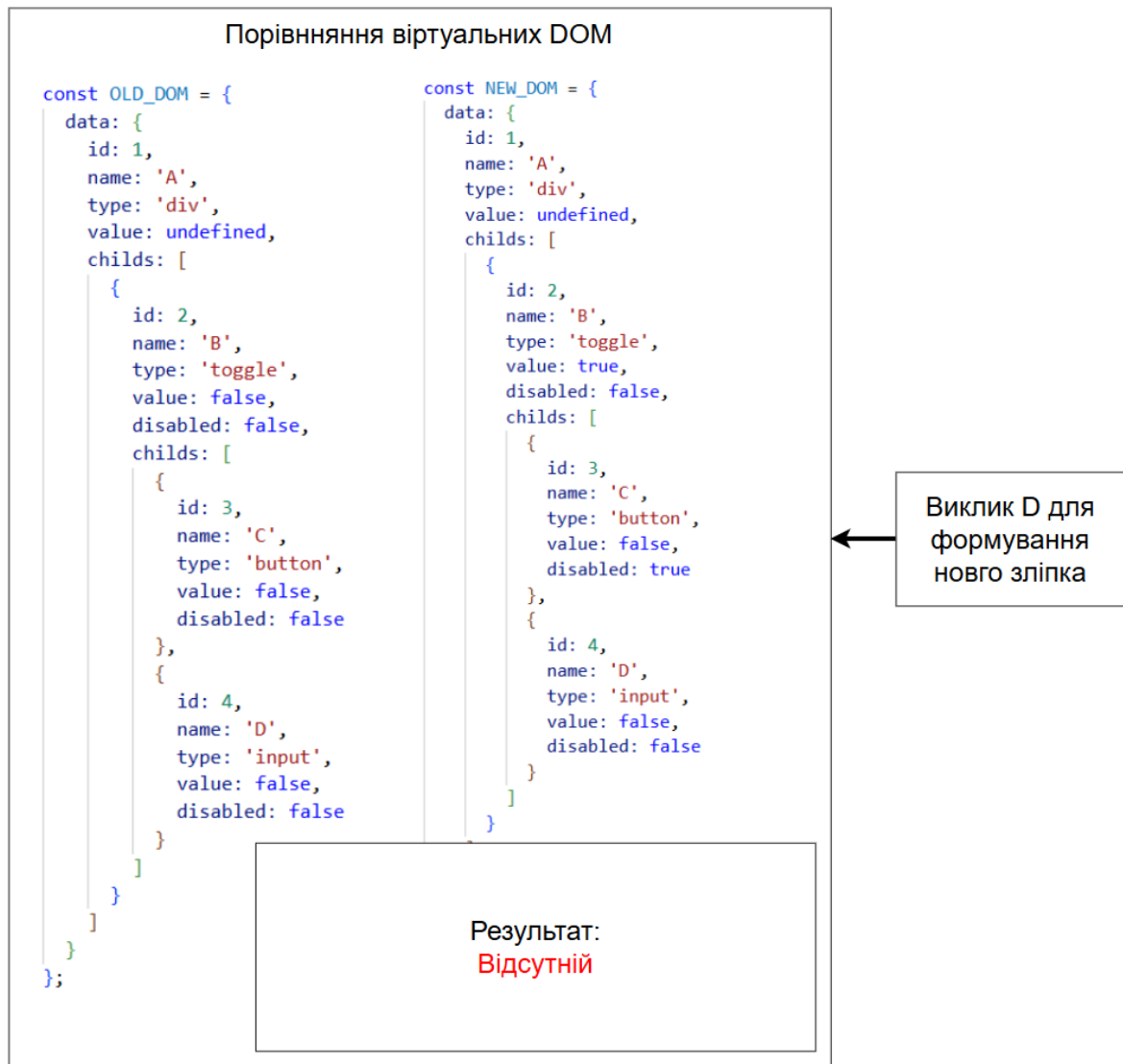


Рис. 3.4. Порівняння віртуального DOM після виклику D для формування нового зліпка

Після порівняльного аналізу віртуальних DOM, що підкреслює ітеративний характер процесу відбувається завершення процесу реконсолізації, рис.3.5. Наступним кроком є застосування змін до DOM, після чого процес оновлення завершується, а система переходить у стабільний стан до наступної зміни. Таким чином, реальний DOM оновлюється мінімально необхідним чином, що значно підвищує продуктивність оновлення інтерфейсу.

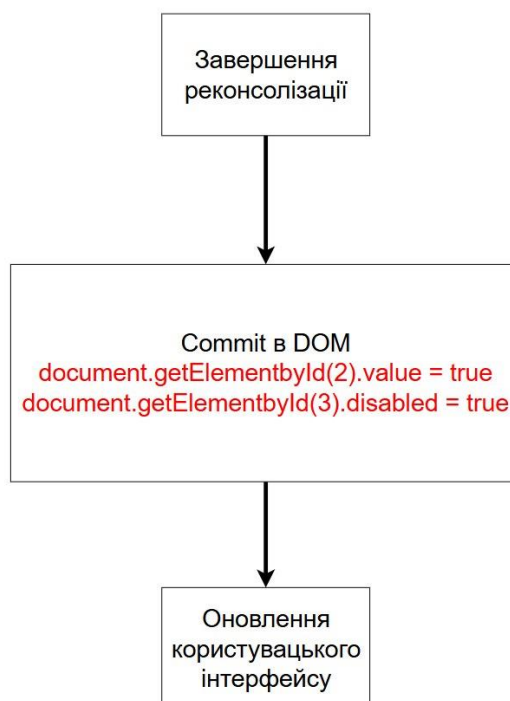


Рис.3.5. Візуалізація завершення процесу оновлення інтерфейсу

Розширюваність прототипу забезпечується модульною організацією компонентів та фабричним підходом до їх вибору на основі типу вузла структури. Додавання нового типу елемента або нового правила залежності не потребує зміни базового механізму оптимізації оновлення, оскільки відповідна логіка інкапсульована в універсальній обгортці компонентів. Це дозволяє адаптувати систему до специфічних вимог конкретних проєктів, не порушуючи цілісності архітектурного рішення.

Одним із ключових аспектів впровадження прототипу є його поведінка при зростанні складності інтерфейсу, що може проявлятися у збільшенні глибини дерева елементів, кількості логічних залежностей між ними та обсягу табличних даних. У традиційних підходах до керування станом такі фактори призводять до нелінійного зростання кількості повторних оновлень, що негативно впливає на продуктивність. Запропонований у прототипі механізм локалізації оновлень дозволяє уникнути цього ефекту за рахунок того, що кожен компонент реагує лише на зміни, релевантні для його множини залежностей.

При збільшенні глибини дерева інтерфейсу система зберігає передбачувану поведінку, оскільки рішення про повторне оновлення приймається незалежно для кожного вузла. Це означає, що навіть у випадку складних вкладених структур оновлення окремого елемента не призводить до каскадного оновлення всього піддерева. Аналогічно, зростання кількості залежностей між елементами не спричиняє глобального ускладнення алгоритму обробки змін, оскільки перевірка релевантності зміни зводиться до локальної операції перевірки належності ідентифікатора зміненого елемента множині залежностей компонента.

Особливої уваги потребує масштабування табличних структур, у яких кількість рядків може динамічно змінюватися під час роботи користувача. У прототипі ці структури інтегруються в загальну модель стану за допомогою складених ідентифікаторів, що дозволяє зберегти однорідність механізму точкового оновлення. Аналіз вузьких місць показує, що основні витрати при масштабуванні пов'язані не з обробкою залежностей, а з обсягом даних, які зберігаються у стані та передаються між компонентами, що визначає напрями подальшої оптимізації.

Незважаючи на ефективність запропонованого підходу, у процесі впровадження прототипу було виявлено низку обмежень, які визначають межі його застосування. Одним із таких обмежень є використання глибокого порівняння пропсів у механізмі мемоізації компонентів. Для елементів, що оперують великими вкладеними структурами даних, такі операції можуть становити додаткове навантаження на процесор. Проте у більшості практичних сценаріїв вартість глибокого порівняння є меншою, ніж вартість надмірних повторних оновлень, що робить цей компроміс виправданим.

Перспективним напрямом розвитку запропонованого підходу є перехід від імпліцитного представлення залежностей у вигляді списків до явної моделі графа залежностей, що дозволило б виконувати додаткові оптимізації на рівні структури інтерфейсу. Такий підхід може забезпечити більш точний аналіз впливу змін стану на різні частини інтерфейсу та відкрити можливості для

попередньої оптимізації порядку оновлень. Окрім цього, перспективним є використання серверної підготовки залежностей, за якої частина обчислень виконується на сервері, а клієнт отримує вже оптимізований опис структури інтерфейсу.

3.3. Оцінка продуктивності та швидкодії прототипу програмного засобу оновлення інтерфейсу, реалізованого з використанням стеку фронтенд технологій

Оцінка якості розробленого прототипу програмного забезпечення є завершальним етапом дослідження, спрямованим на кількісне та якісне підтвердження ефективності запропонованих моделей, методів і алгоритмів оновлення інтерфейсу вебзастосунків [29;30]. На відміну від попередніх підрозділів, у яких розглядалися питання реалізації та архітектурного впровадження, у цьому пункті основна увага зосереджується на експериментальній перевірці того, наскільки запропонований підхід дозволяє зменшити обчислювальні витрати та покращити продуктивні характеристики інтерфейсу в умовах зростання складності структури даних.

Метою експериментального дослідження є встановлення залежності між використанням механізму точкового оновлення на основі залежностей та ключовими показниками продуктивності клієнтського вебзастосунку. До таких показників належать час оновлення інтерфейсу при зміні стану, кількість повторних оновлень компонентів, стабільність часу відгуку при зростанні кількості елементів та загальна масштабованість системи. Оцінка якості прототипу здійснюється шляхом порівняльного аналізу двох конфігурацій системи, які відрізняються наявністю або відсутністю запропонованого механізму оптимізації.

Методика оцінки побудована таким чином, щоб максимально ізолювати вплив досліджуваного фактору, а саме механізму локалізації точкового оновлення, від інших аспектів реалізації. Для цього обидві конфігурації використовують ідентичну структуру інтерфейсу, однаковий набір вхідних

даних та виконуються в одному і тому ж середовищі. Єдиною відмінністю між ними є спосіб реагування компонентів на зміну стану, що дозволяє коректно інтерпретувати отримані результати з точки зору ефективності запропонованого підходу.

У якості контрольної конфігурації використовується реалізація прототипу без механізму точкового контролю оновлень, у якій будь-яка зміна стану призводить до повторного оновлення всіх компонентів, що отримують дані зі сховища. Експериментальна конфігурація, у свою чергу, використовує обгортку компонентів із механізмом відстеження залежностей, унаслідок чого повторне оновлення виконується лише для тих елементів інтерфейсу, для яких зміна стану є релевантною. Таке порівняння дозволяє безпосередньо оцінити внесок запропонованого алгоритмічного рішення у покращення продуктивності.

Важливим аспектом методики є повторюваність експериментів і відтворюваність результатів. Усі вимірювання виконуються за фіксованих умов із багаторазовим повторенням однакових сценаріїв взаємодії з інтерфейсом. Отримані значення усереднюються, що дозволяє зменшити вплив випадкових факторів, таких як фонові процеси операційної системи або коливання продуктивності браузерного середовища. Такий підхід забезпечує достатню надійність експериментальних даних і дозволяє використовувати їх для подальшого аналітичного узагальнення.

Оцінка якості прототипу не обмежується лише вимірюванням абсолютних значень показників продуктивності, а включає також аналіз їхньої динаміки при зростанні складності інтерфейсу. Це дозволяє зробити висновки не лише щодо поточного виграшу у продуктивності, але й щодо масштабованості запропонованого підходу, що є доволі важливим для вебзастосунків, орієнтованих на роботу з великими обсягами даних і складними структурами інтерфейсу.

Для об'єктивної оцінки якості розробленого прототипу програмного забезпечення було використано набір кількісних показників, що

безпосередньо відображають ефективність процесу оновлення клієнтського інтерфейсу. Вибір метрик здійснювався з урахуванням специфіки вебзастосунків, які працюють із динамічними деревовидними структурами даних, а також з огляду на можливість їх практичного вимірювання у браузерному середовищі. Усі показники, використані в експериментальному дослідженні, відповідають даним, зафіксованим у таблицях вимірювань, і застосовуються як для примітивної, так і для експериментальної конфігурацій системи.

Одним із базових показників продуктивності є час оновлення інтерфейсу у відповідь на зміну стану, який відображає тривалість процесу повторного оновлення компонентів після взаємодії користувача. Цей показник характеризує безпосередній вплив алгоритмів оптимізації на швидкодію системи та має вирішальне значення для суб'єктивного сприйняття інтерфейсу. Зменшення часу оновлення свідчить про ефективну локалізацію обчислень і зниження навантаження на головний потік виконання браузера.

Для більш глибокого аналізу продуктивності було також використано показники завантаження процесора, що відображають обчислювальні витрати, пов'язані з виконанням JavaScript-коду та обробкою оновлень інтерфейсу. Зростання цього показника зазвичай корелює з надмірними оновленням та складними операціями порівняння стану, тоді як його зниження свідчить про більш ефективне використання ресурсів клієнтського середовища. Аналіз завантаження процесора дозволяє оцінити не лише швидкодію, але й енергоефективність застосунку, що є важливим аспектом для мобільних пристроїв.

Додатково в межах експериментального дослідження аналізувалося споживання пам'яті, яке відображає обсяг ресурсів, необхідних для зберігання структури інтерфейсу, стану елементів і допоміжних даних, пов'язаних із механізмом оптимізації. Хоча споживання пам'яті не є безпосереднім індикатором швидкодії, його стабільність при зростанні складності інтерфейсу є важливою характеристикою якості реалізації. Значні коливання цього

показника можуть свідчити про неефективне управління даними або надмірне дублювання структур стану.

Окрему увагу в дослідженні приділено аналізу динаміки показників продуктивності при зміні параметрів тестових сценаріїв, зокрема кількості елементів інтерфейсу, глибини їх вкладеності та обсягу табличних даних. Такі вимірювання дозволяють оцінити масштабованість прототипу та виявити характер залежності між складністю структури інтерфейсу і витратами на його оновлення. У цьому контексті використані метрики розглядаються не ізольовано, а в сукупності, що дозволяє сформувати цілісне уявлення про поведінку системи в умовах зростання навантаження.

З метою узагальнення результатів експериментів у роботі також використовується порівняльний аналіз відносних змін показників між контрольними та експериментальними вимірюваннями. Такий підхід дозволяє оцінити не лише абсолютний виграш у продуктивності, але й відносну ефективність запропонованого механізму оптимізації в різних сценаріях використання. Отримані значення слугують основою для подальшої інтерпретації результатів і формулювання висновків щодо доцільності застосування запропонованого підходу у практичних вебзастосунках.

Експериментальне дослідження якості розробленого прототипу програмного забезпечення ґрунтується на серії контрольованих сценаріїв, що відтворюють типові та граничні умови використання веб-застосунків із динамічно сформованими інтерфейсами. Основною метою формування таких сценаріїв було забезпечення репрезентативності вимірювань та можливості коректного порівняння результатів між контрольними й експериментальними конфігураціями системи. Усі експерименти проводилися з використанням однакової структури інтерфейсу та ідентичних вхідних даних, що дозволяє розглядати отримані результати як наслідок впровадження механізму оптимізації оновлення, а не як ефект відмінностей у початкових умовах.

Вихідні дані для експериментів сформовані на основі декларативного опису інтерфейсу, який містить набір елементів різних типів, об'єднаних у

деревовидну структуру з визначеними логічними залежностями, рис.3.6-рис.3.10. Структура інтерфейсу включає як атомарні елементи введення, так і складені контейнерні елементи, зокрема вкладені папки та табличні структури з динамічною кількістю рядків. Такий склад дозволяє оцінити поведінку системи в умовах різної глибини вкладеності та різного ступеня зв'язаності між елементами.

Папка: Налаштування

1 → Ім'я адміна Олег

1 → Фамілія адміна Алмазов

Вік адміна 18

Активувати додаткові налаштування ☒

Папка: Додаткові налаштування

Країна

1 → Код країни

Населення (млн.) 1

Папка: Налаштування доступу до вибірки планет

Венера ☒

Земля ☒

Нептун ☒

Сатурн ☐

Уран ☐

Плутон ☐

Папка: Налаштування доступу до вибірки статі

Чоловік ☒

Жінка ☒

Інше ☐

Текстовий елемент 103

Користувачі

Активний користувач	Ім'я	Вік	Місце прописки
☒	Оксана	99	Сатурн
☒	Аліна	65	Плутон
☒	Оксана	34	Земля
☐	Олег	71	Сатурн
☐	Анна	100	Нептун
☐	Ірина	75	Нептун
☐	Владислав	30	Сатурн
☐	Ярослав	99	Земля
☒	Андрій	29	Сатурн

100 →

Рис. 3.6. Інтерфейс програмного засобу, що містить 103 текстові елементи

Для дослідження впливу масштабу інтерфейсу на продуктивність було сформовано декілька конфігурацій структури, які відрізняються кількістю елементів, глибиною дерева та обсягом табличних даних, рис.3.6-рис.3.10. Зміна цих параметрів дозволяє простежити, як поводить система при

поступовому ускладненні інтерфейсу та чи зберігається ефективність запропонованого механізму оптимізації в умовах зростання навантаження. Особливу увагу приділено сценаріям, у яких зміна одного елемента потенційно може впливати на значну кількість інших вузлів, що є характерним для складних форм із умовною логікою відображення.

Кожен експериментальний сценарій передбачає виконання серії однакових користувацьких дій, зокрема зміну значень окремих полів введення, перемикання логічних прапорців та оновлення значень у табличних рядках.

Папка: Налаштування

Ім'я адміна

Фамілія адміна

Вік адміна

Активувати додаткові налаштування ☒

Папка: Додаткові налаштування

Країна

Код країни

Населення (млн.)

Папка: Налаштування доступу до вибірки планет

Венера ☒

Земля ☒

Нептун ☒

Сатурн ☐

Уран ☐

Плутон ☐

Папка: Налаштування доступу до вибірки статі

Чоловік ☒

Жінка ☒

Інше ☐

Користувачі

Активний користувач	Ім'я	Вік	Місце прописки
☒	Оксана	99	Сатурн
☒	Аліна	65	Плутон
☒	Оксана	34	Земля
☐	Олег	71	Сатурн
☐	Анна	100	Нептун
☐	Ірина	75	Нептун
☐	Владислав	30	Сатурн
☐	Ярослав	99	Земля
☒	Андрій	29	Сатурн

Елемент перемикавання 110

1 →

6 →

3 →

100 →

Рис.3.7. Інтерфейс програмного засобу, що містить 110 елементів перемикавання

Такі дії імітують типову взаємодію користувача з інтерфейсом і дозволяють зафіксувати реакцію системи у вигляді часових і ресурсних показників. Для підвищення достовірності результатів кожен сценарій виконувався багаторазово, а отримані значення усереднювалися, що зменшує вплив випадкових коливань продуктивності клієнтського середовища.

Папка: Налаштування

Ім'я адміна

Фамілія адміна

Вік адміна

Активувати додаткові налаштування ☒

Папка: Додаткові налаштування

Країна

Код країни

Населення (млн.)

Папка: Налаштування доступу до вибірки планет

Венера ☒

Земля ☒

Нептун ☒

Сатурн ☐

Уран ☐

Плутон ☐

Папка: Налаштування доступу до вибірки статі

Чоловік ☒

Жінка ☒

Інше ☐

Користувачі

Активний користувач	Ім'я	Вік	Місце прописки
☒	Оксана	99	Сатурн
☒	Аліна	65	Плутон
☒	Оксана	34	Земля
☐	Олег	71	Сатурн
☐	Анна	100	Нептун
☐	Ірина	75	Нептун
☐	Владислав	30	Сатурн
☐	Ярослав	99	Земля
☒	Андрій	29	Сатурн

Елемент числа 102

1

1

100

Рис.3.8. Інтерфейс програмного засобу, що містить 102 елементи числа

Вимірювання показників продуктивності здійснювалися з використанням інструментів профілювання браузера, які дозволяють зафіксувати час виконання операцій оновлення, кількість повторних оновлень

компонентів, завантаження процесора та споживання пам'яті. Отримані дані структуровано у вигляді таблиць, що відображають результати для кожної конфігурації інтерфейсу окремо для контрольної та експериментальної реалізацій. Такий формат представлення даних забезпечує зручність подальшого аналізу та візуалізації результатів.

Папка: Налаштування

Ім'я адміна

Фамілія адміна

Вік адміна

Активувати додаткові налаштування ☒

Папка: Додаткові налаштування

Країна

Код країни

Населення (млн.)

Папка: Налаштування доступу до вибірки планет

Венера ☒

Земля ☒

Нептун ☒

Сатурн ☐

Уран ☐

Плутон ☐

Папка: Налаштування доступу до вибірки статі

Чоловік ☒

Жінка ☒

Інше ☐

Користувачі

Активний користувач	Ім'я	Вік	Місце прописки
☒	Оксана	99	Сатурн
☒	Аліна	65	Плутон
☒	Оксана	34	Земля
☐	Олег	71	Сатурн
☐	Анна	100	Нептун
☐	Ірина	75	Нептун
☐	Владислав	30	Сатурн
☐	Ярослав	99	Земля
☒	Андрій	29	Сатурн

Рис.3.9. Інтерфейс програмного засобу, що містить 101 випадковий елемент

Важливою особливістю підготовки вихідних даних є дотримання принципу порівнянності експериментів. Усі вимірювання проводилися в

однаковому середовищі виконання, із фіксованими параметрами браузера та апаратної платформи, що дозволяє мінімізувати вплив зовнішніх чинників. Це забезпечує коректність порівняльного аналізу та дозволяє інтерпретувати відмінності між результатами як прямий наслідок використання або невикористання механізму оптимізації оновлення інтерфейсу.

Папка: Налаштування

Ім'я адміна

Фамілія адміна

Вік адміна

Активувати додаткові налаштування ☒

Папка: Додаткові налаштування

Країна

Код країни

Населення (млн.)

Папка: Налаштування доступу до вибірки планет

Венера ☒

Земля ☒

Нептун ☒

Сатурн ☐

Уран ☐

Плутон ☐

Папка: Налаштування доступу до вибірки статі

Чоловік ☒

Жінка ☒

Інше ☐

Користувачі

Активний користувач	Ім'я	Вік	Місце прописки
☒	Оксана	99	Сатурн
☒	Аліна	65	Плутон
☒	Оксана	34	Земля
☐	Олег	71	Сатурн
☐	Анна	100	Нептун
☐	Ірина	75	Нептун
☐	Владислав	30	Сатурн
☐	Ярослав	95	Земля
☒	Андрій	29	Сатурн

Рис.3.10. Інтерфейс програмного засобу, що містить 602 Option

На рис.3.11 представлено схему взаємодії елементів користувацького інтерфейсу та відображення подій, що виникають у результаті однієї зміни стану. У верхній частині рисунку показано основні типи елементів інтерфейсу із зазначенням кількості таких елементів, зокрема текстовий елемент (103),

елемент перемикання (110), числовий елемент (102), випадаючий список (101) та елемент списку опцій (602). Кожен із цих елементів є складовою частиною єдиної системи та може реагувати на зміну стану інших компонентів. Знизу показано, як одна зміна ініційована користувачем в інтерфейсі вебзастосунку впливає на оновлення інтерфейсу, що продемонстровано у журналі виконання, де відображено послідовність викликів або оновлень елементів інтерфейсу. Виділені записи демонструють, що внаслідок однієї зміни відбувається каскадне оновлення кількох компонентів, зокрема: елемента перемикання, текстового елемента, числового елемента, елемента опції та випадаючого списку. Таким чином, рис.3.11 ілюструє механізм поширення змін у системі інтерфейсу користувача, де одна подія спричиняє узгоджене оновлення взаємопов'язаних елементів, що забезпечує цілісність і актуальність відображуваних даних.

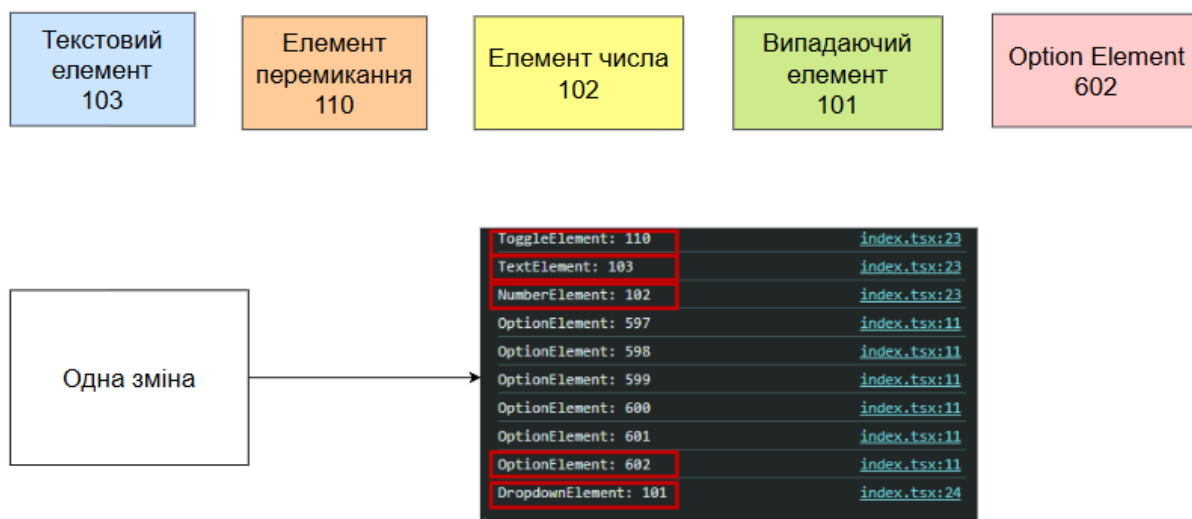


Рис. 3.11. Схему взаємодії елементів користувацького інтерфейсу та відображення подій, що виникають у результаті однієї зміни стану

Сформовані таким чином експериментальні сценарії та відповідні набори вихідних даних створюють основу для подальшого кількісного аналізу ефективності розробленого прототипу. На їх основі буде здійснено

порівняльний аналіз отриманих результатів, що дозволить оцінити вплив запропонованого підходу на продуктивність і масштабованість вебзастосунків із деревовидними структурами інтерфейсу та багатозалежними зв'язками.

Порівняльний аналіз виконано для двох конфігурацій прототипу: базової реалізації (далі – «примітивний підхід»), у якій зміна стану провокує надлишкові оновлення компонентів, та реалізації із запропонованим механізмом точкового оновлення на основі відстеження залежностей (далі – «метод»). Для забезпечення коректності зіставлення в усіх серіях використано однакові сценарії взаємодії та однакові масштаби табличного фрагмента інтерфейсу (10, 100, 1000 рядків). Як основні складові часу відгуку використано метрики профілювання браузера: Scripting (час виконання JavaScript-логіки), Rendering (час, пов'язаний з побудовою/оновленням дерева відображення) та Painting (час фактичного промальовування).

10 рядків таблиці. Зміна провокує зміну одного поля				100 рядків таблиці. Зміна провокує зміну одного поля				1000 рядків таблиці. Зміна провокує зміну одного поля			
Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)
1	59	6	11	1	240	23	40	1	1120	86	37
2	63	8	12	2	223	25	40	2	1190	80	32
3	64	9	13	3	241	25	45	3	1103	61	31
	62,00	7,67	12,00		234,67	24,33	41,67		1137,67	75,67	33,33
System: 112 ms, Scripting: 63 ms, Rendering: 64 ms, Painting: 59 ms	System: 69 ms, Scripting: 116 ms, Rendering: 62 ms, Painting: 40 ms	System: 132 ms, Scripting: 110 ms, Rendering: 208 ms, Painting: 191 ms	System: 55 ms, Scripting: 12 ms, Rendering: 13 ms, Painting: 6 ms	System: 69 ms, Scripting: 116 ms, Rendering: 62 ms, Painting: 40 ms	System: 132 ms, Scripting: 110 ms, Rendering: 208 ms, Painting: 191 ms	System: 55 ms, Scripting: 12 ms, Rendering: 13 ms, Painting: 6 ms	System: 23 ms, Scripting: 25 ms, Rendering: 25 ms, Painting: 23 ms	System: 132 ms, Scripting: 110 ms, Rendering: 208 ms, Painting: 191 ms	System: 132 ms, Scripting: 110 ms, Rendering: 208 ms, Painting: 191 ms	System: 132 ms, Scripting: 110 ms, Rendering: 208 ms, Painting: 191 ms	System: 132 ms, Scripting: 110 ms, Rendering: 208 ms, Painting: 191 ms
10 рядків таблиці. Зміна провокує зміну папки				100 рядків таблиці. Зміна провокує зміну папки				1000 рядків таблиці. Зміна провокує зміну папки			
Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)
1	38	4	5	1	200	9	15	1	1134	37	15
2	35	5	6	2	209	9	10	2	1162	118	28
3	37	7	11	3	197	9	18	3	1212	82	31
	36,67	5,33	7,33		202,00	9,00	14,33		1169,33	79,00	24,67
System: 58 ms, Scripting: 38 ms, Rendering: 4 ms, Painting: 5 ms	System: 49 ms, Scripting: 35 ms, Rendering: 6 ms, Painting: 5 ms	System: 45 ms, Scripting: 37 ms, Rendering: 11 ms, Painting: 7 ms	System: 37 ms, Scripting: 35 ms, Rendering: 6 ms, Painting: 5 ms	System: 200 ms, Scripting: 209 ms, Rendering: 9 ms, Painting: 15 ms	System: 58 ms, Scripting: 10 ms, Rendering: 9 ms, Painting: 9 ms	System: 47 ms, Scripting: 18 ms, Rendering: 9 ms, Painting: 9 ms	System: 197 ms, Scripting: 209 ms, Rendering: 9 ms, Painting: 15 ms	System: 1134 ms, Scripting: 1162 ms, Rendering: 208 ms, Painting: 191 ms	System: 1134 ms, Scripting: 1162 ms, Rendering: 208 ms, Painting: 191 ms	System: 1134 ms, Scripting: 1162 ms, Rendering: 208 ms, Painting: 191 ms	System: 1134 ms, Scripting: 1162 ms, Rendering: 208 ms, Painting: 191 ms
10 рядків таблиці. Зміна провокує зміну колонки				100 рядків таблиці. Зміна провокує зміну колонки				1000 рядків таблиці. Зміна провокує зміну колонки			
Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)
1	57	6	8	1	240	21	37	1	1184	168	49
2	63	9	13	2	229	26	50	2	1291	73	23
3	60	5	8	3	253	28	49	3	1191	93	15
	60,00	6,67	9,67		240,67	25,00	45,33		1222,00	111,33	29,00
System: 61 ms, Scripting: 63 ms, Rendering: 60 ms, Painting: 8 ms	System: 65 ms, Scripting: 63 ms, Rendering: 60 ms, Painting: 8 ms	System: 62 ms, Scripting: 63 ms, Rendering: 60 ms, Painting: 8 ms	System: 60 ms, Scripting: 63 ms, Rendering: 60 ms, Painting: 8 ms	System: 240 ms, Scripting: 229 ms, Rendering: 25 ms, Painting: 37 ms	System: 125 ms, Scripting: 50 ms, Rendering: 49 ms, Painting: 21 ms	System: 114 ms, Scripting: 114 ms, Rendering: 208 ms, Painting: 191 ms	System: 253 ms, Scripting: 229 ms, Rendering: 25 ms, Painting: 37 ms	System: 1184 ms, Scripting: 1291 ms, Rendering: 208 ms, Painting: 191 ms	System: 1184 ms, Scripting: 1291 ms, Rendering: 208 ms, Painting: 191 ms	System: 1184 ms, Scripting: 1291 ms, Rendering: 208 ms, Painting: 191 ms	System: 1184 ms, Scripting: 1291 ms, Rendering: 208 ms, Painting: 191 ms

Рис. 3.12. Первинні результати профілювання часу виконання «Примітивного підходу» для різних сценаріїв зміни стану та розмірів табличних структур

Первинні результати профілювання часу виконання «Примітивного підходу» для різних сценаріїв зміни стану та розмірів табличних структур представлено на рис.3.12. Первинні результати профілювання часу виконання «Методу» для різних сценаріїв зміни стану та розмірів табличних структур представлено на рис.3.13.

10 рядків таблиці. Зміна проковує зміну одного поля				100 рядків таблиці. Зміна проковує зміну одного поля				1000 рядків таблиці. Зміна проковує зміну одного поля				Підсумки (Зміна проковує зміну одного поля)			
Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)		Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
1	41	8	13	1	84	16	29	1	480	56	38	Тест №1	37,00	7,67	12,67
2	33	6	10	2	86	20	43	2	518	29	8	Тест №2	83,33	17,67	35,33
3	37	9	15	3	80	17	34	3	467	62	32	Тест №3	488,33	49,00	26,00
	37,00	7,67	12,67		83,33	17,67	35,33		488,33	49,00	26,00				
															
10 рядків таблиці. Зміна проковує зміну папки				100 рядків таблиці. Зміна проковує зміну папки				1000 рядків таблиці. Зміна проковує зміну папки				Підсумки (Зміна проковує зміну папки)			
Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)		Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
1	33	4	6	1	78	15	25	1	518	71	18	Тест №1	34,67	5,00	7,33
2	31	5	7	2	111	10	18	2	459	47	17	Тест №2	90,33	12,00	20,33
3	40	6	9	3	82	11	18	3	457	40	20	Тест №3	478,00	52,67	18,33
	34,67	5,00	7,33		90,33	12,00	20,33		478,00	52,67	18,33				
															
10 рядків таблиці. Зміна проковує зміну колонки				100 рядків таблиці. Зміна проковує зміну колонки				1000 рядків таблиці. Зміна проковує зміну колонки				Підсумки (Зміна проковує зміну колонки)			
Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)	Номер спроби	Scripting (ms)	Rendering (ms)	Painting (ms)		Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
1	44	7	9	1	151	23	40	1	750	63	16	Тест №1	44,00	6,33	8,67
2	42	8	12	2	111	32	48	2	763	70	21	Тест №2	135,00	27,33	43,67
3	46	4	5	3	143	27	43	3	798	87	29	Тест №3	770,33	73,33	22,00
	44,00	6,33	8,67		135,00	27,33	43,67		770,33	73,33	22,00				
															

Рис. 3.13. Первинні результати профілювання часу виконання «Методу» для різних сценаріїв зміни стану та розмірів табличних структур

У таблиці 3.1 наведено різницю середніх значень та відносну зміну «продуктивності» середніх значень, розраховану як відношення часу базової реалізації до часу оптимізованої, з відніманням одиниці у відсотковому вираженні; позитивне значення означає виграш від застосування методу, а від'ємне – погіршення.

Таблиця 3.1

Середні значення, для сценарію «зміна проковує зміну одного поля»

Примітив				Метод		
Розміри таблиці	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
10	62,00	7,67	12,00	37,00	7,67	12,67
100	234,67	24,33	41,67	83,33	17,67	35,33
1000	1137,67	75,67	33,33	488,33	49,00	26,00

Для сценарію «зміна провокує зміну одного поля» спостерігається найбільш виражений ефект від запропонованого підходу саме у складовій Scripting, що узгоджується з логікою «методу»: локалізація оновлення обмежує обсяг виконуваних обчислень у JavaScript і кількість обробників/перевірок, які виконуються каскадно при глобальному оновленні. Різниця середніх значень, для сценарію «зміна провокує зміну одного поля» представлено у таблиці 3.2.

Таблиця 3.2

Різниця середніх значень для сценарію «зміна провокує зміну одного поля»

Розміри таблиці	Scripting (ms)	Rendering (ms)	Painting (ms)
10	25,00	0,00	-0,67
100	151,33	6,67	6,33
1000	649,33	26,67	7,33

Для 10 рядків Scripting зменшується з 62,00 мс до 37,00 мс, тобто на 25,00 мс, що відповідає зростанню продуктивності на 67,57 %, таблиця 3.3. Для 100 рядків Scripting зменшується з 234,67 мс до 83,33 мс (на 151,33 мс), що відповідає зростанню продуктивності на 181,60 %. Для 1000 рядків Scripting зменшується з 1137,67 мс до 488,33 мс (на 649,33 мс), що відповідає зростанню продуктивності на 132,97 %. Проведемо аналіз продуктивності середніх значень для сценарію «зміна провокує зміну одного поля», табл.3.3

Таблиця 3.3

Значення продуктивності середніх значень для сценарію «зміна провокує зміну одного поля»

Розміри таблиці	Scripting	Rendering	Painting
10	67,57%	0,00%	-5,26%
100	181,60%	37,74%	17,92%
1000	132,97%	54,42%	28,21%

На рівні Rendering ефект у малому масштабі відсутній (7,67 мс проти 7,67 мс для 10 рядків), а для більших масштабів спостерігається зменшення: для 100 рядків 24,33 мс проти 17,67 мс (виграш 37,74 %), для 1000 рядків 75,67 мс проти 49,00 мс (виграш 54,42 %). На рівні Painting для 10 рядків фіксується незначне погіршення (12,00 мс проти 12,67 мс; –5,26 %), яке інтерпретується як допустимий накладний ефект оптимізаційного шару при малому навантаженні; однак при 100 і 1000 рядках Painting зменшується відповідно з 41,67 мс до 35,33 мс (виграш 17,92 %) та з 33,33 мс до 26,00 мс (виграш 28,21%). Сукупно результати цього сценарію підтверджують, що «метод» найбільш результативний у ситуації, коли зміна впливає на обмежену множину залежних компонентів, а базова реалізація провокує надлишкове виконання логіки оновлення. Діаграмне представлення для цього сценарію, рис.3.14, дозволяє візуально зафіксувати домінування складової Scripting у загальній структурі витрат часу та демонструє, що саме вона визначає основний виграш при застосуванні «методу». Для 1000 рядків різниця між стовпчиками Scripting є найбільшою, що підтверджує висновок про масштабний характер ефекту оптимізації за зростання кількості елементів.



Рис.3.14. Діаграми результатів сценарію «зміна провокує зміну одного поля»

Для сценарію «зміна провокує зміну папки» характер змін відрізняється, табл. 3.4, оскільки подія впливає на контейнерний компонент, що у ряді випадків неминує залучає додаткові операції на рівнях Rendering і Painting.

Таблиця 3.4

Середні значення, для сценарію «зміна провокує зміну папки»

Примітив				Метод		
Розмір и таблиці	Середній Scripting (ms)	Середній Renderin g (ms)	Середній Painting (ms)	Середній Scripting (ms)	Середній Renderin g (ms)	Середній Painting (ms)
10	36,67	5,33	7,33	34,67	5,00	7,33
100	202,00	9,00	14,33	90,33	12,00	20,33
1000	1169,33	79,00	24,67	478,00	52,67	18,33

Різниця середніх значень, для сценарію «зміна провокує зміну папки» представлено у таблиці 3.5.

Таблиця 3.5

Різниця середніх значень для сценарію «зміна провокує зміну папки»

Розміри таблиці	Scripting (ms)	Rendering (ms)	Painting (ms)
10	2,00	0,33	0,00
100	111,67	-3,00	-6,00
1000	691,33	26,33	6,33

Для 10 рядків ефект «методу» є невеликим, що відповідає очікуванню, табл.3.6: Scripting зменшується з 36,67 мс до 34,67 мс (перевага 5,77 %), Rendering зменшується з 5,33 мс до 5,00 мс (перевага 6,67 %), Painting не змінюється (7,33 мс). Для 100 рядків «метод» демонструє суттєве скорочення Scripting з 202,00 мс до 90,33 мс (перевага 123,62 %), однак на рівні Rendering і Painting фіксується збільшення. Так Rendering зростає з 9,00 мс до 12,00 мс (-25,00%), Painting зростає з 14,33 мс до 20,33 мс (-29,51 %). Цей результат доцільно інтерпретувати як прояв «перерозподілу витрат» між стадіями, а саме «метод» зменшує обсяг JavaScript-логіки, але при зміні контейнера може збільшувати частку робіт, які виконуються на етапі побудови/оновлення

відображення та промальовування, особливо у випадку, коли візуальні зміни зачіпають більшу область інтерфейсу.

Таблиця 3.6

Значення продуктивності середніх значень для сценарію «зміна провокує зміну папки»

Розміри таблиці	Scripting	Rendering	Painting
10	5,77%	6,67%	0,00%
100	123,62%	-25,00%	-29,51%
1000	144,63%	50,00%	34,55%

При 1000 рядках ефект «методу» стає однозначно позитивним для всіх трьох складових: Scripting зменшується з 1169,33 мс до 478,00 мс (перевага 144,63 %), Rendering зменшується з 79,00 мс до 52,67 мс (перевага 50,00 %), Painting зменшується з 24,67 мс до 18,33 мс (перевага 34,55 %).



Рис.3.15. Діаграми результатів сценарію ««зміна провокує зміну папки»»

Таким чином, у «зміна провокує зміну папки» «метод» демонструє найбільшу практичну цінність саме на великих обсягах даних, рис.3.15, тоді

як на середньому масштабі можливий компроміс у вигляді зростання Rendering/Painting за одночасного істотного скорочення Scripting.

Для сценарію «зміна провокує зміну колонки» результати демонструють стабільну перевагу «методу» у складовій Scripting і переважно позитивний ефект у Rendering/Painting при великому масштабі, табл.3.7- табл. 3.9.

Таблиця 3.7

Середні значення, для сценарію «зміна провокує зміну одного колонки»

Примітив				Метод		
Розмір и таблиці	Середні й Scripting (ms)	Середній Renderin g (ms)	Середні й Painting (ms)	Середні й Scripting (ms)	Середній Renderin g (ms)	Середні й Painting (ms)
10	60,00	6,67	9,67	44,00	6,33	8,67
100	240,67	25,00	45,33	135,00	27,33	43,67
1000	1222,00	111,33	29,00	770,33	73,33	22,00

Для 10 рядків Scripting зменшується з 60,00 мс до 44,00 мс (виграш 36,36 %), Rendering зменшується з 6,67 мс до 6,33 мс (виграш 5,26 %), Painting зменшується з 9,67 мс до 8,67 мс (виграш 11,54 %).

Таблиця 3.8

**Різниця середніх значень для сценарію
«зміна провокує зміну одного колонки»**

Розміри таблиці	Scripting (ms)	Rendering (ms)	Painting (ms)
10	16,00	0,33	1,00
100	105,67	-2,33	1,67
1000	451,67	38,00	7,00

Для 100 рядків Scripting зменшується з 240,67 мс до 135,00 мс (виграш 78,27 %), Painting дещо зменшується з 45,33 мс до 43,67 мс (виграш 3,82 %), однак Rendering зростає з 25,00 мс до 27,33 мс (–8,54 %). Для 1000 рядків ефект стає однозначно позитивним за всіма складовими: Scripting зменшується з

1222,00 мс до 770,33 мс (виграш 58,63 %), Rendering зменшується з 111,33 мс до 73,33 мс (виграш 51,82 %), Painting зменшується з 29,00 мс до 22,00 мс (виграш 31,82 %).

Таблиця 3.9

Значення продуктивності середніх значень для сценарію «зміна провокує зміну одного колонки»

Розміри таблиці	Scripting	Rendering	Painting
10	36,36%	5,26%	11,54%
100	78,27%	-8,54%	3,82%
1000	58,63%	51,82%	31,82%

Загальна інтерпретація цього сценарію полягає в тому, що метод найкраще проявляє себе при зростанні кількості рядків, рис. 3.16., оскільки зміна, що в базовій конфігурації призводить до широкого каскаду оновлень, у оптимізованій конфігурації локалізується на рівні залежних компонентів та пов'язаних сегментів таблиці.

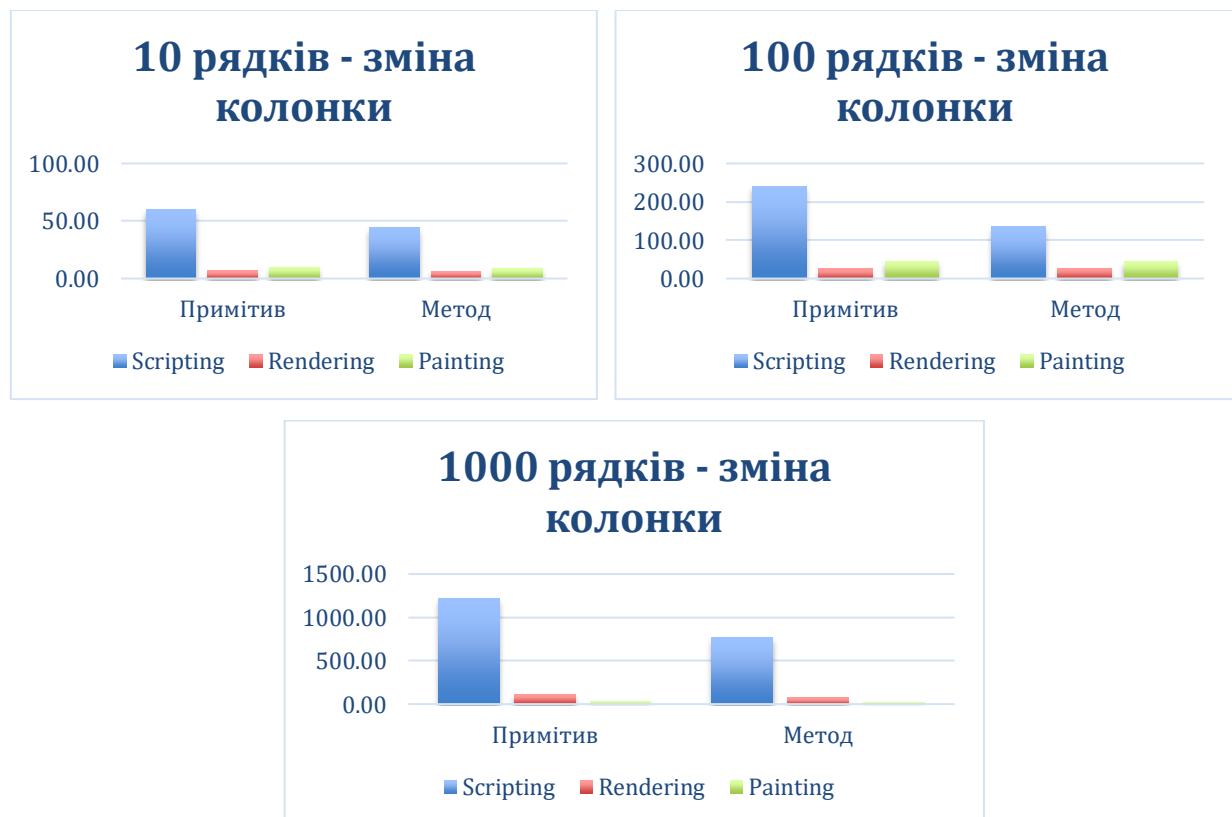


Рис.3.16. Діаграми результатів сценарію ««зміна провокує зміну колонки»

Візуальне порівняння, рис.3.16, підтверджує, що при 1000 рядків різниця між конфігураціями є суттєвою не лише у Scripting, але й у Rendering та Painting, що інтерпретується як прямий наслідок зменшення кількості компонентів, які підлягають повторному оновленню, і, як результат, зменшення обсягу робіт з перерахунку віртуального DOM та промальовування.

Узагальнюючи результати трьох сценаріїв, можна зробити висновок, що запропонований метод демонструє систематичний позитивний ефект у складовій Scripting у всіх розглянутих випадках та на всіх масштабах, причому найбільший абсолютний виграш досягається при 1000 рядках, де зменшення обчислювального навантаження на JavaScript-логіку становить сотні мілісекунд. Водночас для окремих сценаріїв середнього масштабу (зокрема «зміна папки» та «зміна колонки» при 100 рядках) можуть фіксуватися локальні погіршення у Rendering або Painting, що пояснюється характером події та її впливом на область візуальних змін, а також перерозподілом витрат між стадіями конвеєра відображення. При великих масштабах (1000 рядків) метод проявляє найбільш стійку перевагу, забезпечуючи одночасне скорочення Scripting, Rendering і Painting, що підтверджує його масштабованість і практичну доцільність для веб-застосунків із великими деревовидними структурами та багатозалежними зв'язками між елементами.

3.4. Оцінка продуктивності та швидкодії прототипу програмного засобу оновлення інтерфейсу реалізованого засобами нативного JavaScript

Для підтвердження отриманих результатів проводимо реалізацію двох програмних продуктів засобами нативного JavaScript без застосування зовнішніх фреймворків. Лістинг програми представлено у Додатку В та Додатку Д.

Перший варіант реалізації не включає запропонований метод та представляє собою JavaScript-додаток на чистому DOM API без використання фреймворків, побудований за принципом повного перерендерингу дерева

компонентів при кожній зміні стану. Структура додатку складається з модуля ініціалізації, модуля керування станом та набору функцій для кожного типу елемента інтерфейсу. `Index.html` містить мінімальну HTML-структуру з контейнером `#root`, у який динамічно монтується інтерфейс. `Styles.css` визначає стилі для папок, таблиць, текстових полів, перемикачів та інших елементів. Дані форми описуються декларативно (`data.js`) у вигляді дерева вузлів (`treeData`), що дозволяє динамічно генерувати довільно вкладені структури папок та полів різних типів.

Механізм реактивності у цьому підході реалізований через єдину глобальну функцію перерендерингу, зареєстровану в модулі стану (`state.js`) як зворотний виклик. При будь-якій зміні значення незалежно від того, яке конкретне поле було відредаговано відбувається повне очищення кореневого DOM-елемента та побудова всього дерева компонентів з нуля. Видимість кожного елемента при цьому обчислюється заново шляхом перевірки його залежностей відносно поточного стану з використанням набору операторів порівняння. Усі модулі (`buildFolder`, `buildToggle`, `buildText`, `buildNumber`, `buildDropdown`, `buildTable`) працюють за однаковим принципом – перевіряють видимість елемента через `isElementVisible()`, створюють DOM-елементи, прив'язують обробники подій, які викликають `updateValue()` та повертають готовий DOM-вузол. Файл `init.js` містить ключову логіку примітивного підходу.

Головним недоліком такої архітектури є відсутність будь-якої вибіркової оновленні. Кожна взаємодія користувача з будь-яким полем форми призводить до перебудови всього дерева DOM, попри те що переважна більшість елементів не змінила ні своїх даних, ні свого стану видимості. При великій кількості вузлів у дереві та значному обсязі табличних даних це спричиняє суттєве навантаження на браузер, оскільки кожен елемент форми перебудовується при кожній зміні стану. Такий підхід є концептуально простим і зручним для розуміння, однак не масштабується при зростанні розміру форми.

Другий варіант реалізації вирішує проблему продуктивності шляхом впровадження вибіркового механізму оновлення, заснованого на шаблоні проєктування «Спостерігач» (Observer). На відміну від примітивного підходу, дерево компонентів (treeData) будується один раз при завантаженні сторінки та надалі не перебудовується повністю. Ключовим нововведенням є попередньо обчислена множина залежностей, яка формується під час ініціалізації та зберігає для кожного елемента перелік ідентифікаторів стану, від яких він залежить. Це дозволяє кожному компоненту при побудові зареєструвати підписку лише на ті зміни стану, які безпосередньо впливають на його видимість або значення.

Файл index.html містить мінімальну HTML-структуру з контейнером root, у який динамічно монтується інтерфейс. Файл styles.css визначає стилі для всіх типів елементів: папок, інпутів, таблиць, перемикачів, що забезпечує уніфікований зовнішній вигляд та адаптивність інтерфейсу. Для декларативного опису інтерфейсів сформовано модуль data.js, який містить набір функцій (createFolder, createText, createToggle, createDropdown, createTable), що формують дерево конфігурації treeData. У цьому дереві описано вкладені структури папок (до 30 рівнів вкладеності), елементи введення різних типів, таблицю користувачів із гетерогенними колонками, правила залежностей між елементами, початкові значення стану (initialValueData) та карту залежностей (dependencyMap), що використовується реактивним ядром. Таким чином, data.js виконує роль доменно-специфічної мови (DSL) для опису інтерфейсу. Модуль state.js реалізує реактивне ядро системи, яке реалізовано в ручну. Кожен тип елемента має окремий модуль, так buildFolder.js створює контейнер-папку та реалізує рендеринг дочірніх елементів, buildToggle.js реалізує побудову перемикача (checkbox) з двостороннім зв'язком зі станом, buildText.js відповідає за текстове поле з реактивним оновленням, buildNumber.js формує числове поле з автоматичним приведенням типів, buildDropdown.js генерує випадний список з підтримкою приховування опцій та залежностей, buildTable.js відповідає за побудову

таблиці з великою кількістю рядків та різними типами клітинок, а `buildCell.js` відповідає за формування клітинок таблиці. Усі модулі підтримують умовну видимість та автоматичний перерендеринг при зміні залежних значень. Таким чином система реалізує НОС-підхід, реактивне оновлення DOM, умовний рендеринг, дерево компонентів, централізований стан, автоматичне очищення неактуальних спостерігачів. Фактично це власний міні-фреймворк, який повторює ключові концепції React/Vue, але реалізований вручну. Обидва підходи використовують однакову декларативну модель інтерфейсу (`treeData`), однак суттєво відрізняються архітектурою, продуктивністю та масштабованістю, табл.3.10. В програмі реалізовано глибоку вкладеність, рис.3.17.

Таблиця 3.10

Порівняльна характеристика прототипу програмного продукту

Характеристика	Примітивний підхід	Метод
Модель оновлення	Повний перерендер усього інтерфейсу	Часткове оновлення лише залежних елементів
Архітектура	Лінійна, без оптимізації	Реактивна, компонентна
Продуктивність	Низька при збільшенні даних	Висока, стабільна
Масштабованість	Обмежена	Висока
Призначення	Демонстраційний, базовий	Реальний, промисловий

Експериментальні дослідження проводимо аналогічно до методики, що описана у п.3.3. Обидві конфігурації використовують ідентичну структуру інтерфейсу, однаковий набір вхідних даних та виконуються в одному і тому ж середовищі. Умови проведення експерименту фіксовані, багаторазово повторюються за однакових сценаріїв взаємодії з інтерфейсом, а для зменшення впливу випадкових факторів отримані значення усереднюються. Оцінювання якості створеного прототипу не зводиться лише до фіксації абсолютних значень показників продуктивності, а охоплює також аналіз їхньої зміни за умов ускладнення інтерфейсу.

Папка: Папка 17

Папка: Папка 18

Папка: Папка 19

Нептун ☒

Папка: Папка 20

Папка: Папка 21

Папка: Папка 22

Сатурн ☐

Папка: Папка 23

Папка: Папка 24

Папка: Папка 25

Папка: Папка 26

Уран ☐

Папка: Папка 27

Папка: Папка 28

Активувати вибір планет ☒

Плутон ☐

Папка: Папка 29

Папка: Папка 30

Користувачі

Активний користувач	Ім'я	Вік	Стать	Місце знаходження	Машина	Діти
☐	Жінка	90	Чоловік			5
☐	Інше	16	Чоловік		Audi	2
☐	Інше	4	Чоловік			2
☐	Жінка	99	Жінка			
☐	Чоловік	77	Чоловік	Венера	BMW	5
☐	Жінка	67	Чоловік			
☐	Інше	91	Чоловік	Венера		
☒	Чоловік	96	Чоловік		Audi	
☒	Інше	62	Чоловік	Нептун		1
☒	Чоловік	22				4
☐	Жінка	41				

Рис.3.17. Інтерфейс прототипу програмного продукту на основі об'єктів з глибокою вкладеністю на нативному JS

Для забезпечення об'єктивності оцінювання було застосовано комплекс кількісних метрик, які безпосередньо характеризують ефективність процесу оновлення клієнтського інтерфейсу. Для забезпечення коректності порівняння в усіх серіях експериментів застосовувалися однакові сценарії взаємодії та ідентичні масштаби табличного фрагмента інтерфейсу (100, 1000 і 10000 рядків). Як ключові компоненти часу відгуку використовувалися метрики браузерного профілювання: Scripting (тривалість виконання JavaScript-логіки), Rendering (час, пов'язаний із формуванням або оновленням дерева

відображення) та Painting (тривалість безпосереднього промальовування елементів інтерфейсу). На рисунку 3.18 представлені первинні результати профілювання часу виконання «Примітивного підходу» для різних сценаріїв змін стану та розмірів табличних структур.

100 рядків. Зміна поля				1000 рядків. Зміна поля				10000 рядків. Зміна поля				Зміна поля			
Спроба	Scripting	Rendering	Painting	Спроба	Scripting	Rendering	Painting	Спроба	Scripting	Rendering	Painting		Scripting	Rendering	Painting
1	410	37	10	1	3795	312	16	1	40967	3471	90	100	393,00	34,00	9,00
2	385	33	7	2	3808	283	15	2	38299	3405	85	1000	3800,00	303,67	15,33
3	384	32	10	3	3797	316	15	3	41672	3301	88	10000	40312,67	3392,33	87,67
	393,00	34,00	9,00		3800,00	303,67	15,33		40312,67	3392,33	87,67				
100 рядків. Зміна набору				1000 рядків. Зміна набору				10000 рядків. Зміна набору полів				Зміна набору полів			
Спроба	Scripting	Rendering	Painting	Спроба	Scripting	Rendering	Painting	Спроба	Scripting	Rendering	Painting		Scripting	Rendering	Painting
1	412	35	10	1	3813	315	15	1	39159	3232	91	100	394,00	34,00	10,67
2	383	33	11	2	3804	279	15	2	40285	3145	85	1000	3847,00	293,00	15,33
3	387	34	11	3	3924	285	16	3	47235	3364	90	10000	42226,33	3247,00	88,67
	394,00	34,00	10,67		3847,00	293,00	15,33		42226,33	3247,00	88,67				
100 рядків. Зміна колонки				1000 рядків. Зміна колонки				10000 рядків. Зміна колонки				Зміна колонки			
Спроба	Scripting	Rendering	Painting	Спроба	Scripting	Rendering	Painting	Спроба	Scripting	Rendering	Painting		Scripting	Rendering	Painting
1	346	31	10	1	3280	224	14	1	41245	3103	98	100	344,67	28,67	9,33
2	358	29	9	2	3092	208	11	2	40912	3292	89	1000	3227,00	218,00	13,00
3	330	26	9	3	3309	222	14	3	42506	3099	90	10000	41554,33	3164,67	92,33
	344,67	28,67	9,33		3227,00	218,00	13,00		41554,33	3164,67	92,33				

У таблиці 3.11 наведено різницю середніх значень та відносну зміну «продуктивності» середніх значень, розраховану як відношення часу базової реалізації («примітивний підхід») до часу оптимізованої («метод»), з відніманням одиниці у відсотковому вираженні; позитивне значення означає перевагу від застосування методу, а від’ємне – погіршення.

Таблиця 3.11

**Середні значення, для сценарію «зміна провокує зміну одного поля»
прототипу програмного продукту на нативному JS**

Примітив				Метод		
Розміри таблиці	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
100	393,00	34,00	9,00	1,00	6,33	10,00
1000	3800,00	303,67	15,33	27,67	50,67	23,00
10000	40312,67	3392,33	87,67	86,67	743,00	63,33

Показники Scripting демонструють найбільш виражену різницю між двома підходами. У «примітивному підході» час виконання JavaScript-логіки зростає різко та непропорційно збільшенню розміру таблиці – від 393 мс для 100 рядків до 3800 мс для 1000 рядків і до 40312,67 мс для 10 000 рядків. Така динаміка свідчить про те, що «примітивний підхід» виконує повне перерахування логіки, незалежно від того, що зміна стосується лише одного поля. На противагу цьому, «метод» демонструє значно нижчі значення – 1 мс, 27,67 мс та 86,67 мс відповідно. Зростання є плавним і контрольованим, що вказує на локальність обробки змін та відсутність зайвих обчислень. Навіть на найбільшому масштабі «метод» зберігає час Scripting у межах, прийнятних для інтерактивних вебінтерфейсів.

Метрика Rendering відображає витрати часу на оновлення дерева відображення. У «примітивному підході» спостерігається суттєве зростання –

34 мс, 303,67 мс та 3392,33 мс для трьох масштабів відповідно. Це свідчить про те, що при кожній зміні система виконує масштабне оновлення DOM, що стає критичним при великих обсягах даних. «Метод» демонструє значно кращу поведінку – 6,33 мс, 50,67 мс та 743 мс. Хоча Rendering також зростає зі збільшенням кількості рядків, абсолютні значення залишаються суттєво нижчими. Це пояснюється тим, що «метод» виконує часткове оновлення DOM, обмежуючись лише тими елементами, які дійсно зазнали змін.

Показники Painting характеризують витрати часу на промальовування елементів інтерфейсу. У «примітивному підході» значення становлять 9 мс, 15,33 мс та 87,67 мс, демонструючи поступове зростання разом зі збільшенням розміру таблиці. «Метод» має значення 10 мс, 23 мс та 63,33 мс відповідно. На малих масштабах «примітивний підхід» інколи демонструє трохи кращі результати, що може бути пов'язано з тим, що повне переформування інтерфейсу інколи дає більш передбачуваний шлях промальовування. Проте на великих таблицях «метод» стає ефективнішим, оскільки зменшує кількість repaint-операцій завдяки мінімізації області оновлення.

Отримані результати демонструють чітку перевагу «методу» над «примітивним підходом» у всіх ключових метриках. Найбільший вигреш спостерігається у Scripting, де «метод» забезпечує прискорення від сотень до понад чотирьохсот разів. Rendering також значно ефективніший у «методу», що підтверджує зменшення обсягу DOM-операцій. Painting демонструє змішану динаміку, але на великих масштабах оптимізований підхід («метод») стабільно перевершує «примітивний підхід». У сукупності це свідчить про те, що «метод» забезпечує масштабованість, стабільність часу відгуку та ефективне використання ресурсів браузера, тоді як «примітивний підхід» стає непридатним при збільшенні обсягів даних.

Різниця середніх значень, для сценарію «зміна провокує зміну одного поля» представлено у таблиці 3.12. Різниця часу виконання для метрики Scripting демонструє найбільш виражений і стабільний приріст на користь «методу». Для таблиці з 100 рядків різниця становить 392 мс, що вже свідчить

про суттєве зменшення обчислювальних витрат. При збільшенні розміру до 1000 рядків різниця зростає до 3772,33 мс, а для 10 000 рядків досягає 40226 мс. Така динаміка підтверджує, що «примітивний підхід» виконує значний обсяг зайвих обчислень, тоді як «метод» локалізує оновлення і мінімізує кількість операцій.

Таблиця 3.12

Різниця середніх значень для сценарію «зміна провокує зміну одного поля» прототипу програмного продукту на нативному JS

Розміри таблиці	Scripting (ms)	Rendering (ms)	Painting (ms)
100	392,00	27,67	-1,00
1000	3772,33	253,00	-7,67
10000	40226,00	2649,33	24,33

Зростання різниці є майже пропорційним збільшенню розміру таблиці, що підкреслює масштабованість «методу» та неефективність «примітивного підходу». Різниця часу Rendering також демонструє суттєве зростання зі збільшенням розміру таблиці. Для 100 рядків вона становить 27,67 мс, що вказує на помітне зменшення обсягу DOM-операцій у «методу». Для 1000 рядків різниця збільшується до 253 мс, а для 10 000 рядків – до 2649,33 мс. Така динаміка свідчить про те, що «примітивний підхід» виконує значно більший обсяг роботи з оновлення дерева відображення, тоді як «метод» обмежується частковими оновленнями. Зростання різниці підтверджує, що «метод» значно знижує навантаження на механізми рендерингу браузера, особливо при великих обсягах даних. Показники різниці часу Painting мають змішану динаміку. Для таблиці з 100 рядків різниця становить -1 мс, що означає незначну перевагу «примітивного підходу» на малих масштабах. Для 1000 рядків різниця збільшується до -7,67 мс, що також свідчить про трохи швидше промальовування у «примітивному підході». Однак при 10 000 рядках ситуація змінюється: різниця стає позитивною (24,33 мс), тобто «метод»

починає працювати швидше. Це пояснюється тим, що при великих обсягах даних «примітивний підхід» виконує повне промальовування значної частини інтерфейсу, тоді як «метод» обмежує repaint лише тими елементами, які дійсно змінилися. Таким чином, «метод» демонструє перевагу у Painting лише на великих масштабах, де часткове оновлення дає відчутний ефект.

Аналіз різниці середніх значень часу виконання підтверджує, що «метод» забезпечує суттєве зменшення витрат часу на всіх ключових етапах обробки інтерфейсу. Найбільший виграш спостерігається у Scripting, де різниця зростає від сотень до понад сорока тисяч мілісекунд при збільшенні розміру таблиці. Rendering також демонструє значне зменшення часу, що свідчить про ефективність часткових оновлень DOM. Метрика Painting показує перевагу «методу» лише на великих масштабах, що є природним наслідком оптимізації області промальовування.

Проведемо аналіз продуктивності середніх значень для сценарію «зміна провокує зміну одного поля», табл.3.13. У межах аналізу було розглянуто відсоткове збільшення продуктивності «методу» порівняно з «примітивним підходом» для трьох ключових метрик (Scripting, Rendering та Painting) на масштабах таблиць 100, 1000 та 10 000 рядків. Ці показники відображають, наскільки ефективнішим є «метод» у відсотковому вимірі, що дозволяє оцінити не лише абсолютний виграш, а й відносну ефективність оптимізації.

Таблиця 3.13

Значення продуктивності середніх значень для сценарію «зміна провокує зміну одного поля» для прототипу на нативному JS

Розміри таблиці	Scripting	Rendering	Painting
100	39200,00%	436,84%	-10,00%
1000	13634,94%	499,34%	-33,33%
10000	46414,62%	356,57%	38,42%

Показники відсоткового збільшення продуктивності для Scripting є найбільш вираженими серед усіх метрик. Для таблиці з 100 рядків приріст становить 39200%, що свідчить про колосальне зменшення часу виконання логіки JavaScript у «методу». При збільшенні розміру таблиці до 1000 рядків приріст становить 13634,94%, а для 10 000 рядків – 46414,62%. Така динаміка демонструє, що «метод» забезпечує надзвичайно високий відносний виграш, особливо на великих обсягах даних. Це підтверджує, що «примітивний підхід» виконує значний обсяг зайвих обчислень, тоді як «метод» обмежується локальними оновленнями. Відсоткові значення у десятках тисяч відсотків підкреслюють радикальну різницю в ефективності між двома підходами. Відсоткове збільшення продуктивності для Rendering також демонструє стабільно високі значення. Для таблиці з 100 рядків приріст становить 436,84%, що вказує на суттєве зменшення обсягу DOM-операцій у «методу». Для 1000 рядків приріст зростає до 499,34%, а для 10 000 рядків становить 356,57%. Хоча відсоткові значення нижчі, ніж у Scripting, вони все одно свідчать про значне зниження навантаження на механізми рендерингу браузера. Зменшення приросту на найбільшому масштабі (356,57%) пояснюється тим, що Rendering залишається операцією, чутливою до кількості елементів DOM, навіть при оптимізації. Проте «метод» стабільно забезпечує у 3–5 разів кращу продуктивність. Показники відсоткового збільшення продуктивності для Painting мають змішану динаміку. Для таблиці з 100 рядків приріст становить -10%, що означає, що «примітивний підхід» на малих масштабах інколи промальовує інтерфейс трохи швидше. Для 1000 рядків різниця збільшується до -33,33%, що також свідчить про перевагу «примітивного підходу» на середніх масштабах. Однак, при 10 000 рядків ситуація змінюється: приріст становить 38,42%, тобто «метод» починає працювати швидше. Це пояснюється тим, що при великих обсягах даних «примітивний підхід» виконує повне промальовування значної частини інтерфейсу, тоді як «метод» обмежує repaint лише тими елементами, які

зазнали змін. Таким чином, Метод демонструє перевагу у Painting лише на великих масштабах, де часткове оновлення дає відчутний ефект.

Аналіз відсоткового збільшення продуктивності підтверджує, що «метод» забезпечує значний вигрaш у всіх ключових аспектах обробки інтерфейсу. Найбільший приріст спостерігається у Scripting, де значення сягають десятків тисяч відсотків, що свідчить про радикальне зменшення обчислювальних витрат. Rendering також демонструє стабільно високий приріст продуктивності, що підтверджує ефективність часткових оновлень DOM. Метрика Painting показує перевагу «методу» лише на великих масштабах, що є природним наслідком оптимізації області промальовування.

Діаграмне представлення для цього сценарію, рис.3.20, дозволяє візуально зафіксувати домінування складової Scripting у загальній структурі витрат часу та демонструє, що саме вона визначає основний вигрaш при застосуванні методу.

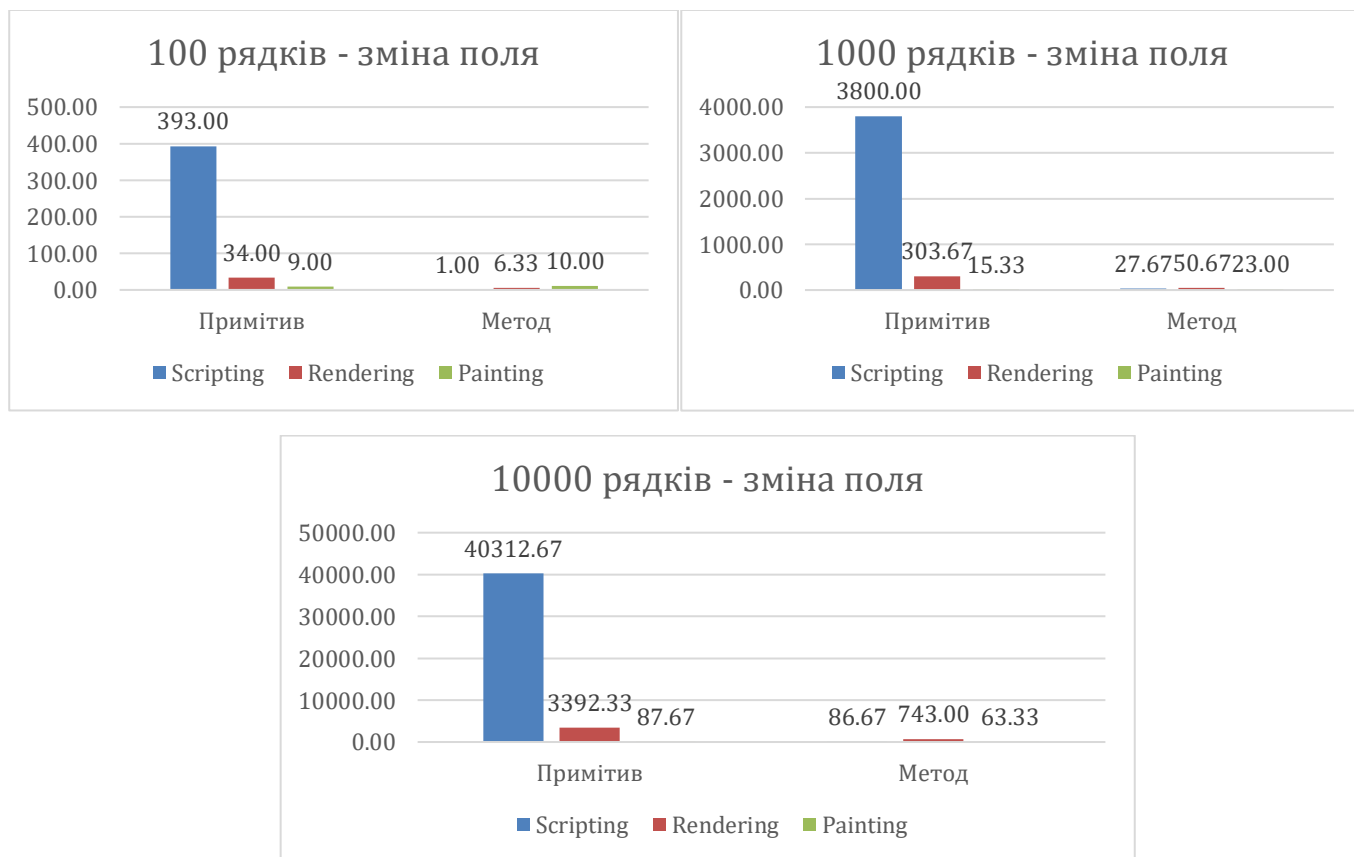


Рис.3.20. Діаграми результатів сценарію «зміна провокує зміну одного поля» для прототипу програмного продукту на нативному JS

Для 1000 рядків різниця між стовпчиками Scripting є найбільшою, що підтверджує висновок про масштабний характер ефекту оптимізації за зростання кількості елементів.

Для сценарію «зміна провокує зміну папки» характер змін відрізняється, табл. 3.14. У межах цього сценарію було проаналізовано продуктивність «примітивного підходу та Методу при зміні, що впливає на цілу групу елементів (папку). На відміну від сценарію, де змінюється лише одне поле, тут очікується більший обсяг оновлень, що дозволяє оцінити стійкість обох підходів до більш складних операцій. Аналіз охоплює три ключові метрики – Scripting, Rendering та Painting – на масштабах таблиць 100, 1000 та 10 000 рядків.

Таблиця 3.14

Середні значення, для сценарію «зміна провокує зміну папки» для прототипу на нативному JS

Розміри таблиці	Примітив			Метод		
	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
100	394,00	34,00	10,67	3,67	9,33	14,00
1000	3847,00	293,00	15,33	35,67	61,33	20,00
10000	42226,33	3247,00	88,67	334,33	1713,00	125,67

Показники Scripting демонструють значну різницю між двома підходами, що стає особливо помітним при збільшенні розміру таблиці. У «примітивному підході» час виконання логіки JavaScript становить 394 мс для 100 рядків, 3847 мс для 1000 рядків та 42226,33 мс для 10 000 рядків. Така динаміка свідчить про те, що «примітивний підхід» виконує повне перерахування логіки для всіх елементів, навіть якщо зміна стосується лише певної групи. «Метод» демонструє значно нижчі значення: 3,67 мс, 35,67 мс та 334,33 мс відповідно. Зростання є плавним і пропорційним, що підтверджує локальність обробки змін та ефективне уникнення зайвих обчислень. Навіть

при 10 000 рядках «метод» зберігає час Scripting у межах, прийнятних для інтерактивних інтерфейсів. У «примітивному підході» Rendering становить 34 мс, 293 мс та 3247 мс відповідно. Це свідчить про те, що при зміні папки система виконує масштабне оновлення DOM, що стає критичним при великих обсягах даних. «Метод» демонструє значно кращі результати – 9,33 мс, 61,33 мс та 1713 мс. Хоча Rendering також зростає зі збільшенням кількості рядків, абсолютні значення залишаються суттєво нижчими. Показники Painting демонструють змішану динаміку, але загалом підтверджують перевагу «методу» на великих масштабах. У «примітивному підході» значення становлять 10,67 мс, 15,33 мс та 88,67 мс. Метод має значення 14 мс, 20 мс та 125,67 мс відповідно. Отримані результати демонструють, що «метод» забезпечує суттєве зменшення часу виконання Scripting та Rendering у порівнянні з «примітивним підходом», навіть у складнішому сценарії, де зміна стосується цілої групи елементів. Найбільший виграш спостерігається у Scripting, де Метод зменшує час виконання у десятки разів. Rendering також демонструє значне зниження навантаження, що підтверджує ефективність часткових оновлень DOM. Метрика Painting показує змішану динаміку

У табл.3.15 розглядається різниця між середніми значеннями часу виконання «примітивного підходу» та «методу» для трьох ключових метрик (Scripting, Rendering та Painting) на масштабах таблиць 100, 1000 та 10 000 рядків.

Таблиця 3.15

**Різниця середніх значень для сценарію «зміна провокує зміну папки»
для прототипу на нативному JS**

Розміри таблиці	Scripting (ms)	Rendering (ms)	Painting (ms)
100	390,33	24,67	-3,33
1000	3811,33	231,67	-4,67
10000	41892,00	1534,00	-37,00

Ці показники дозволяють оцінити, наскільки «метод» зменшує обчислювальні витрати у порівнянні з «примітивним» підходом у ситуації, коли зміна зачіпає цілу групу елементів (папку). Різниця часу виконання для метрики Scripting демонструє значний і стабільний вигаш на користь «методу». Для таблиці з 100 рядків різниця становить 390,33 мс, що вже свідчить про суттєве зменшення обсягу виконуваної логіки. При збільшенні розміру таблиці до 1000 рядків різниця зростає до 3811,33 мс, а для 10 000 рядків досягає 41892 мс. Така динаміка підтверджує, що «примітивний підхід» виконує повне перерахування логіки для всієї групи елементів, тоді як «метод» обмежується локальними оновленнями. Зростання різниці є майже пропорційним збільшенню обсягу даних, що свідчить про високу масштабованість «методу» та недостатню ефективність «примітивного підходу» при великих структурах. Різниця часу Rendering також демонструє суттєве зростання зі збільшенням розміру таблиці. Для 100 рядків вона становить 24,67 мс, що вказує на помітне зменшення обсягу DOM-операцій у Методу. Для 1000 рядків різниця збільшується до 231,67 мс, а для 10 000 рядків – до 1534 мс. Показники різниці часу Painting демонструють змішану динаміку. Для таблиці з 100 рядків різниця становить -3,33 мс, що означає, що «примітивний підхід» на малих масштабах інколи промальовує інтерфейс трохи швидше. Для 1000 рядків різниця становить -4,67 мс, що також свідчить про невелику перевагу «примітивного підходу» у промальовуванні. При збільшенні розміру таблиці до 10 000 рядків різниця стає значно більшою (-37 мс), що означає, що «примітивний підхід» виконує Painting швидше, ніж «метод». Це пояснюється тим, що у сценарії зміни папки оновлюється велика група елементів, і часткове промальовування у «методу» може вимагати складнішої роботи з областями repaint. У таких випадках повне промальовування, характерне для «примітивного підходу» виявляється швидшим.

Відсоткове збільшення/зменшення продуктивності «методу» порівняно з «примітивним підходом» для трьох ключових метрик на масштабах таблиць 100, 1000 та 10 000 рядків представлено у табл.3.16. Ці показники дозволяють

оцінити не лише абсолютний виграш, а й відносну ефективність оптимізації, що є критично важливим для аналізу масштабованості.

Таблиця 3.16

Значення продуктивності середніх значень для сценарію «зміна провокує зміну папки» для прототипу на нативному JS

Розміри таблиці	Scripting	Rendering	Painting
100	10645,45%	264,29%	-23,81%
1000	10685,98%	377,72%	-23,33%
10000	12530,01%	89,55%	-29,44%

Показники відсоткового збільшення продуктивності для Scripting демонструють надзвичайно високі значення, що свідчить про радикальне зменшення часу виконання логіки JavaScript у «методу». Для таблиці з 100 рядків приріст становить 10645,45%, що вже вказує на суттєву перевагу «методу». При збільшенні розміру таблиці до 1000 рядків приріст становить 10685,98%, а для 10 000 рядків – 12530,01%. Така динаміка свідчить про те, що «примітивний підхід» виконує значний обсяг зайвих обчислень, тоді як «метод» локалізує оновлення та мінімізує кількість операцій. Відсоткові значення у діапазоні 10–12 тисяч відсотків підкреслюють високу масштабованість «методу» та його здатність ефективно працювати навіть при великих обсягах даних. Відсоткове збільшення продуктивності для Rendering також демонструє стабільно високі значення. Для таблиці з 100 рядків приріст становить 264,29%, що вказує на суттєве зменшення обсягу DOM-операцій у «методу». Для 1000 рядків приріст зростає до 377,72%, що свідчить про ще більшу ефективність часткових оновлень. Для 10 000 рядків приріст становить 89,55%, що є нижчим, але все одно підтверджує перевагу «методу». Зменшення приросту на найбільшому масштабі пояснюється тим, що Rendering залишається операцією, чутливою до кількості елементів DOM, навіть при оптимізації. Проте «метод» стабільно забезпечує у 1,9–4,7 рази кращу продуктивність, що є значним показником для реальних

вебінтерфейсів. Показники відсоткового збільшення продуктивності для Painting демонструють зворотну динаміку порівняно з іншими метриками. Для таблиці з 100 рядків приріст становить -23,81%, що означає, що «примітивний підхід» на малих масштабах промальовує інтерфейс швидше. Для 1000 рядків різниця становить -23,33%, що також свідчить про перевагу «примітивного підходу». При збільшенні розміру таблиці до 10 000 рядків приріст становить -29,44%, що означає, що «примітивний підхід» продовжує демонструвати швидше промальовування. Це пояснюється тим, що у сценарії зміни папки оновлюється велика група елементів, і часткове промальовування у «методу» може вимагати складнішої роботи з областями repaint.

Найбільший приріст спостерігається у Scripting, де значення сягають понад 12 тисяч відсотків, що свідчить про радикальне зниження обчислювальних витрат. Rendering також демонструє стабільно високий приріст продуктивності, що підтверджує ефективність часткових оновлень DOM. Візуалізація результатів представлена на рис.3.21.

Таким чином, у «зміна провокує зміну папки» метод демонструє найбільшу практичну цінність саме на великих обсягах даних, рис.3.21. Аналіз відсоткового збільшення продуктивності підтверджує, що «метод» забезпечує значний виграш у Scripting та Rendering, навіть у складному сценарії, де зміна зачіпає цілу групу елементів. Метрика Painting показує зворотну динаміку у цьому сценарії «примітивний підхід» може бути швидшим, що пов'язано зі специфікою промальовування великих груп елементів.

Таким чином, у сценарії «зміна провокує зміну папки» метод демонструє найбільшу практичну цінність саме на великих обсягах даних, рис.3.21. Аналіз відсоткового збільшення продуктивності підтверджує, що «метод» забезпечує значний виграш у Scripting та Rendering, навіть у складному сценарії, де зміна зачіпає цілу групу елементів. Метрика Painting показує зворотну динаміку у цьому сценарії «примітивний підхід» може бути швидшим, що пов'язано зі специфікою промальовування великих груп елементів.

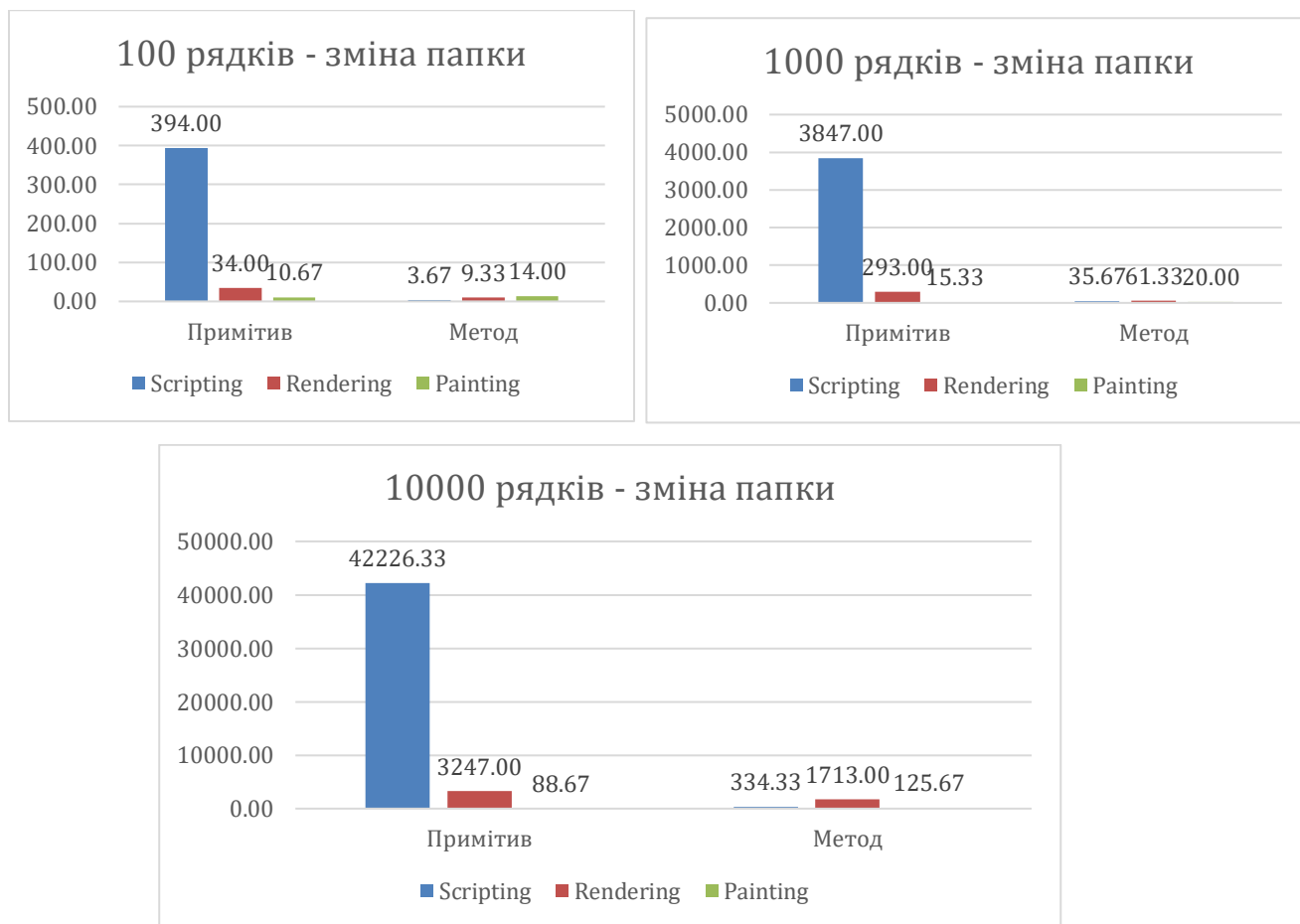


Рис.3.21. Діаграми результатів сценарію «зміна проковує зміну папки» для прототипу програмного продукту на нативному JS

У сценарії «зміна проковує зміну колонки» оцінюється продуктивність «примітивного підходу» та «методу» у випадку, коли зміна зачіпає всю колонку таблиці, тобто множину елементів, розташованих вертикально. Такий тип оновлення є складнішим, ніж зміна одного поля, але менш масштабним, ніж зміна цілої папки. Аналіз охоплює так само три ключові метрики на масштабах таблиць 100, 1000 та 10 000 рядків, табл.3.17- табл. 3.19. Для 100 рядків Scripting зменшується з 344,67 мс до 7,3 мс (перевага 4600 %), Rendering зменшується з 28,67 мс до 8,33 мс (перевага 244 %), Painting зростає з 9,33 мс до 11,67 мс (програш 20 %). Для 1000 рядків Scripting зменшується 3227 мс до 63,33 мс (перевага становить 4995,26%), Rendering зменшується з 218 мс до 64,67 мс (перевага 237,11 %), Painting зростає з 13 мс до 17,67 мс (програш 26,42 %).

Таблиця 3.17

**Середні значення, для сценарію «зміна провокує зміну одного колонки»
для прототипу на нативному JS**

Примітив				Метод		
Розміри таблиці	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)	Середній Scripting (ms)	Середній Rendering (ms)	Середній Painting (ms)
100	344,67	28,67	9,33	7,33	8,33	11,67
1000	3227,00	218,00	13,00	63,33	64,67	17,67
10000	41554,33	3164,67	92,33	684,00	863,33	90,67

Для 10000 рядків Scripting зменшується 41554,33 мс до 684 мс (перевага становить 5975,19%), Rendering зменшується з 3164,67 мс до 863,33 мс (перевага 266,56 %), Painting зменшується з 92,33 мс до 90,67 мс (перевага 1,84 %). У сукупності ці результати підтверджують, що «метод» забезпечує масштабованість, ефективність та стабільність продуктивності, тоді як «примітивний підхід» демонструє різке погіршення продуктивності при збільшенні обсягів даних.

Таблиця 3.18

**Різниця середніх значень для сценарію
«зміна провокує зміну одного колонки» для прототипу на нативному JS**

Розміри таблиці	Scripting (ms)	Rendering (ms)	Painting (ms)
100	337,33	20,33	-2,33
1000	3163,67	153,33	-4,67
10000	40870,33	2301,33	1,67

Різниця часу виконання для метрики Scripting демонструє значний і стабільний виграш на користь «методу». Для таблиці з 100 рядків різниця становить 337,33 мс, що вже свідчить про суттєве зменшення обсягу виконуваної логіки. При збільшенні розміру таблиці до 1000 рядків різниця

зростає до 3163,67 мс, а для 10 000 рядків досягає 40870,33 мс. Різниця часу Rendering також демонструє суттєве зростання зі збільшенням розміру таблиці. Для 100 рядків вона становить 20,33 мс, що вказує на помітне зменшення обсягу DOM-операцій у «методу». Для 1000 рядків різниця збільшується до 153,33 мс, а для 10 000 рядків до 2301,33 мс. Показники різниці часу Painting демонструють іншу динаміку, ніж у попередніх метриках. Для таблиці з 100 рядків різниця становить -2,33 мс, що означає, що «примітивний підхід» на малих масштабах інколи промальовує інтерфейс трохи швидше. Для 1000 рядків різниця становить -4,67 мс, що також свідчить про невелику перевагу «примітивного підходу» у промальовуванні. При збільшенні розміру таблиці до 10 000 рядків різниця становить 1,67 мс, тобто «метод» починає працювати трохи швидше.

Таблиця 3.19

Значення продуктивності середніх значень для сценарію «зміна провокує зміну одного колонки» для прототипу на нативному JS

Розміри таблиці	Scripting	Rendering	Painting
100	4600,00%	244,00%	-20,00%
1000	4995,26%	237,11%	-26,42%
10000	5975,19%	266,56%	1,84%

Показники відсоткового збільшення продуктивності для Scripting демонструють дуже високі значення, що свідчить про суттєве зменшення часу виконання логіки JavaScript у «методу». Для таблиці з 100 рядків приріст становить 4600%, що вже вказує на значну перевагу «методу». При збільшенні розміру таблиці до 1000 рядків приріст становить 4995,26%, а для 10 000 рядків – 5975,19%. Тобто «метод» зменшує час Scripting у 46-60 разів. Відсоткове збільшення продуктивності для Rendering також демонструє стабільно високі значення. Для таблиці з 100 рядків приріст становить 244%, що вказує на суттєве зменшення обсягу DOM-операцій у «методу». Для 1000 рядків приріст становить 237,11%, а для 10 000 рядків – 266,56%. Ці значення

свідчать про те, що «метод» забезпечує у 2,3–3,6 рази кращу продуктивність у порівнянні з Примітивним підходом. Показники відсоткового збільшення продуктивності для Painting демонструють зворотну динаміку порівняно з іншими метриками. Для таблиці з 100 рядків приріст становить -20%, що означає, що «примітивний підхід» на малих масштабах промальовує інтерфейс швидше. Для 1000 рядків різниця становить -26,42%, що також свідчить про перевагу «примітивного підходу». При збільшенні розміру таблиці до 10 000 рядків приріст становить 1,84%, тобто «метод» починає працювати трохи швидше.

Загальна інтерпретація цього сценарію полягає в тому, що метод найкраще проявляє себе при зростанні кількості рядків, рис. 3.22., оскільки зміна, що в базовій конфігурації призводить до широкого каскаду оновлень, у оптимізованій конфігурації локалізується на рівні залежних компонентів та пов'язаних сегментів таблиці..

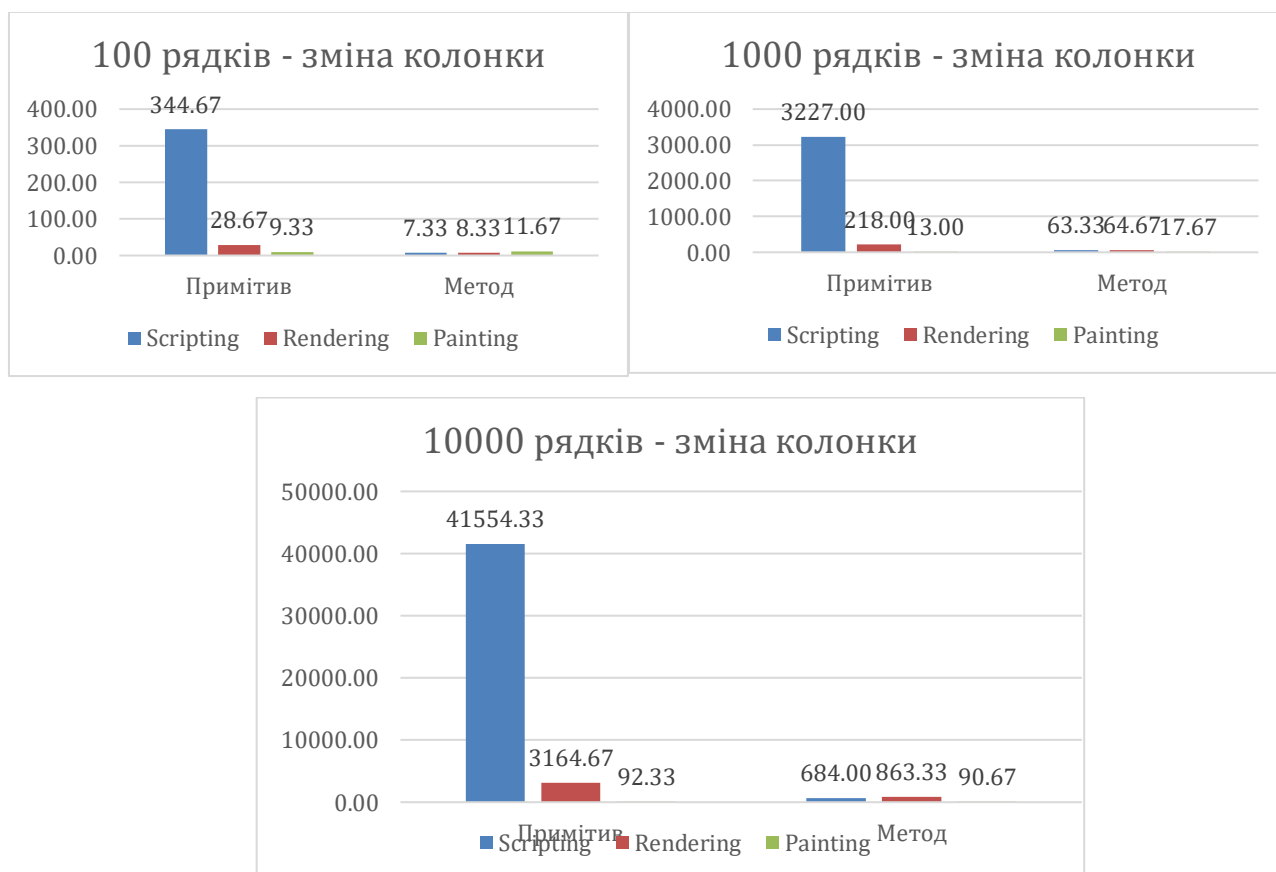


Рис.3.22. Діаграми результатів сценарію «зміна провокує зміну колонки» для прототипу програмного продукту на нативному JS

Візуальне порівняння, рис.3.22, підтверджує, що при 1000 рядків різниця між конфігураціями є суттєвою не лише у Scripting, але й у Rendering та Painting, що інтерпретується як прямий наслідок зменшення кількості компонентів, які підлягають повторному оновленню, і, як результат, зменшення обсягу робіт з перерахунку віртуального DOM та промальовування.

В межах дисертації також проведено аналіз динаміки кількості оновлень DOM-елементів для всіх трьох сценаріїв, результати якого представлено у таблиці 3.20, рис.3.23. У межах експериментального дослідження було проведено порівняння кількості елементів, що підлягають оновленню у трьох типових сценаріях зміни даних: «зміна одного поля», «зміна папки» та «зміна колонки». Для кожного сценарію оцінювалися два підходи – «примітивний підхід» та «метод» на трьох масштабах таблиць, а саме 100, 1000 та 10 000 рядків. Отримані результати демонструють принципово різні моделі поведінки цих підходів та дозволяють оцінити їхню масштабованість.

Таблиця 3.20

Кількість оновлень DOM-елементів для прототипу на нативному JS

Розмір и таблиці	«Зміна провокує зміну одного поля»		«Зміна провокує зміну папки»		«Зміна провокує зміну колонки»	
	«примітивний підхід»	«метод»	«примітивний підхід»	«метод»	«примітивний підхід»	«метод»
100	744	0	744	3	744	100
1000	7044	0	7044	3	7044	1000
1000	700400	0	700400	3	700400	10000

У сценарії «зміна провокує зміну одного поля» «примітивний підхід» оновлює всю таблицю, незалежно від того, що зміна стосується лише одного елемента. Кількість оновлень становить 744 для таблиці з 100 рядків, 7044 для 1000 рядків та 70044 для 10 000 рядків. «Метод», натомість, виконує локальне оновлення, і кількість змінених елементів дорівнює 0, оскільки оновлення відбувається на рівні окремого вузла без перерахування структури. Це

свідчить про те, що «метод» повністю усуває зайві операції (100%), тоді як «примітивний підхід» демонструє лінійну залежність від розміру таблиці та не масштабується. У сценарії «зміна провокує зміну папки» «примітивний підхід» знову оновлює всю таблицю, що призводить до однакових значень відповідно.

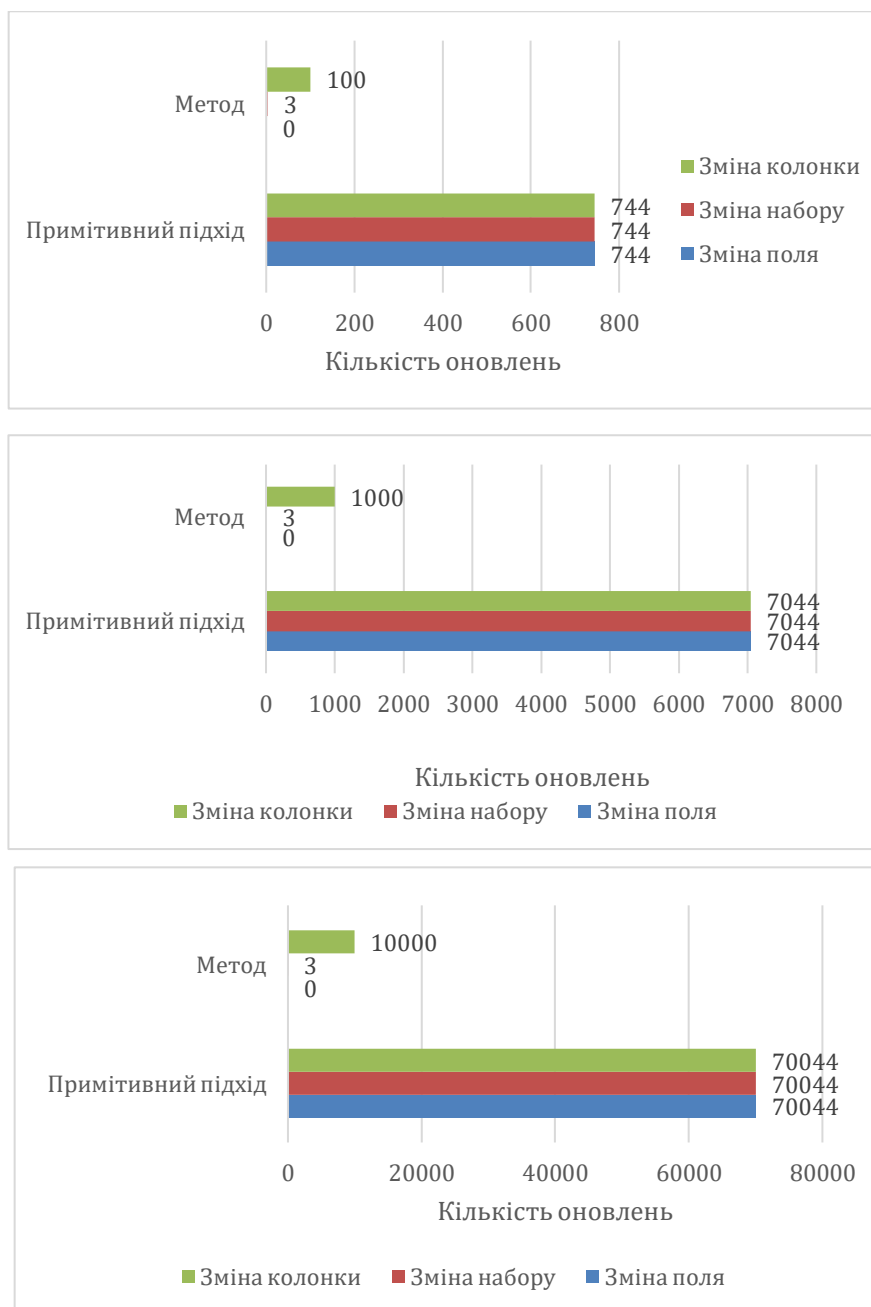


Рис.3.23. Кількість оновлень DOM-елементів при різних сценаріях для прототипу програмного продукту на нативному JS

У сценарії «зміна провокує зміну колонки» «примітивний підхід» знову оновлює всю таблицю, тоді як «метод» оновлює рівно стільки елементів,

скільки містить колонка. Це означає, що «метод» працює оптимально, виконуючи рівно той обсяг роботи, який відповідає структурі змін. «Примітивний підхід» виконує зайві операції, оновлюючи елементи, що не зазнали змін.

Висновки до третього розділу

Реалізований прототип програмного засобу підтвердив практичну ефективність запропонованих моделей і методів точкового оновлення інтерфейсу вебзастосунків із глибоко вкладеними структурами даних та багатозалежними зв'язками. Використання стеку TypeScript, React, Redux Toolkit, НОС-обгортки, мемоізації та Vite забезпечило коректне відтворення складних деревовидних структур та контрольоване оновлення їх стану. Архітектура з централізованим керуванням станом і децентралізованим контролем повторних оновлень, зокрема механізм `dependencyList`, дозволила суттєво зменшити кількість непотрібних перерендерів. Підтримка складених структур через `ComplexId` підтвердила універсальність підходу, а результати роботи прототипу довели, що метод забезпечує оновлення лише змінених вузлів інтерфейсу, уникаючи глобальних перерахунків дерева та зберігаючи стабільну продуктивність у складних інтерфейсах.

Архітектурні та програмні рішення підтвердили здатність системи забезпечувати децентралізований контроль оновлень у глибоко вкладених інтерфейсах. Двоетапний алгоритм перевірки залежностей, наявності `dependencyList` та відповідності `changedId`, дозволив уникнути глобального обходу дерева компонентів і суттєво зменшити кількість DOM-операцій. Реалізація рекурсивних контейнерів (папок) та універсальних рендер-функцій підтвердила масштабованість підходу, а інкапсуляція логіки оновлень у НОС забезпечила розширюваність системи без ускладнення візуальних компонентів. Це підтвердило практичну придатність запропонованого механізму для вебзастосунків зі складною логікою взаємодії та багаторівневими структурами даних.

Архітектуру побудовано на принципах компонентно-орієнтованого програмування React із використанням централізованого керування станом Redux Toolkit, що дозволило реалізувати механізм децентралізованого прийняття рішень про повторне оновлення компонентів без необхідності глобального обходу дерева інтерфейсу. Проведений аналіз механізму роботи Virtual DOM, процесів diffing, rendering та painting підтвердив, що локалізація змін забезпечує зменшення кількості операцій над реальним DOM навіть за умов зростання складності інтерфейсу. Прототип продемонстрував коректну роботу на структурах, що включають 103 текстові елементи, 110 елементів перемикачів, 102 числові елементи, 101 випадючий список та 602 option-елементи, а також підтримку табличних структур із 10, 100 та 1000 рядками. Встановлено, що використання складених ідентифікаторів (ComplexId) для табличних даних забезпечує точну локалізацію змін та уникнення хибних оновлень. Водночас визначено перспективні напрями подальшого розвитку, зокрема перехід до явної графової моделі залежностей та часткове перенесення обчислень на серверну сторону.

Виконано експериментальну оцінку роботи розробленого прототипу шляхом порівняння базової реалізації («примітивний підхід») та реалізації із запропонованим механізмом точкового оновлення компонентів («метод»). Оцінювання проводилося за показниками Scripting, Rendering та Painting із використанням інструментів профілювання браузера в однакових умовах виконання та на ідентичних структурах інтерфейсу. Отримані результати підтвердили суттєву перевагу запропонованого «методу» над «примітивним підходом» у всіх сценаріях. У сценарії «зміна провокує зміну одного поля» час Scripting зменшився з 62 до 37 мс (+67,57%) для 10 рядків, з 234,67 до 83,33 мс (+181,60%) для 100 рядків та з 1137,67 до 488,33 мс (+132,97%) для 1000 рядків. У сценарії «зміна провокує зміну папки» при 1000 рядках Scripting скоротився з 1169,33 до 478 мс (+144,63%), Rendering – з 79 до 52,67 мс (+50%), Painting – з 24,67 до 18,33 мс (+34,55%). Отримані результати підтверджують, що зі зростанням кількості елементів і глибини вкладеності ефективність

запропонованого механізму зростає, що свідчить про його високу масштабованість та практичну доцільність для складних вебзастосунків.

Виконано експериментальну оцінку роботи прототипу програмного засобу оновлення інтерфейсу, реалізованого засобами нативного JavaScript, із метою визначення ефективності запропонованого механізму локалізації оновлень без використання високорівневих фронтедн-фреймворків. Оцінювання проводилося шляхом порівняння «примітивного підходу», що передбачає повне оновлення інтерфейсу, та «методу», заснованого на локалізації змін і вибіркового оновленні DOM-елементів. Дослідження охоплювало три типові сценарії: «зміна одного поля», «зміна папки» та «зміна колонки» на масштабах 100, 1000 та 10 000 рядків, а саме оцінювання здійснювалося за метриками Scripting, Rendering та Painting.

Результати експериментального дослідження підтвердили суттєву перевагу запропонованого методу насамперед у складовій Scripting, де зафіксовано радикальне скорочення часу виконання JavaScript-логіки. У сценарії «зміна одного поля» для 10 000 рядків час Scripting зменшився з 40312,67 мс до 86,67 мс, що відповідає приросту продуктивності 46414,62 %, тоді як Rendering скоротився з 3392,33 мс до 743,00 мс (+356,57 %). У сценарії «зміна папки» при 10 000 рядків час Scripting зменшився з 42226,33 мс до 334,33 мс (+12530,01 %), а Rendering – з 3247,00 мс до 1713,00 мс (+89,55 %). Для сценарію «зміна колонки» при тому самому масштабі Scripting скоротився з 41554,33 мс до 684,00 мс (+5975,19 %), а Rendering – з 3164,67 мс до 863,33 мс (+266,56 %). Отримані результати свідчать про те, що зі зростанням складності структури інтерфейсу ефективність запропонованого методу не лише зберігається, а й посилюється.

Окремо встановлено, що ключовою причиною приросту продуктивності є істотне скорочення кількості повторних оновлень DOM-елементів. Якщо у «примітивному підході» при 10 000 рядках оновлювалося до 70044 DOM-елементів, то «метод» у сценарії «зміна папки» оновлював лише 3 елементи, а у сценарії «зміна колонки» лише елементи відповідної колонки (10 000

елементів), що підтверджує локальний характер обробки змін і значно кращу масштабованість запропонованого підходу. Разом із тим встановлено, що метрика Painting демонструє змішану динаміку. У частині сценаріїв повне промальовування виявилось швидшим, однак при великих масштабах часткове оновлення починає забезпечувати перевагу і в цій складовій. Отримані результати підтверджують універсальність запропонованого методу та його практичну придатність для побудови високопродуктивних вебінтерфейсів засобами нативного JavaScript.

Результати проведеного дослідження відображено у роботах [98;99;100;101;121] здобувача.

Висновки

Відповідно до поставленої мети та завдань дослідження із задач оптимізації оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю отримано такі основні результати: проведено порівняльний аналіз внутрішньої архітектури оновлення вебзастосунків, аналіз методів та технологій оновлення інтерфейсу вебзастосунків, аналіз проблем продуктивності при оновленні інтерфейсу глибоко вкладених структур; формалізовано модель представлення глибоко вкладених структур даних при оновленні інтерфейсу вебзастосунків з метою оптимізації; розроблено метод точкового оновлення вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язкам; проведено моделювання прототипу програмного додатку для точкового оновлення інтерфейсу вебзастосунків та здійснено його програмну реалізацію та експериментальну перевірку запропонованих рішень на основі двох прототипів вебзастосунку, проведено оцінку ефективності.

Результати дослідження дали підстави для таких висновків:

1. Проведений порівняльний аналіз сучасних архітектур оновлення інтерфейсу, а саме DOM, Virtual DOM, Shadow DOM, Signals, Reactive Rendering, дозволив виявити їх фундаментальні обмеження. Встановлено, що традиційні diffing-орієнтовані підходи характеризуються значними обчислювальними витратами, що зростають зі збільшенням розміру та глибини структури даних, а також призводять до каскадних перерендерингів і втрати локальності оновлень. Якісно визначено, що такі підходи не забезпечують стабільної продуктивності, а кількісно призводять до зростання часу обробки змін у великих конфігураціях. Визначено, що основними причинами зниження ефективності вебінтерфейсів є висока складність diffing-процесів, надлишкові DOM-операції, каскадні оновлення компонентів, складність відстеження залежностей та обмеженість існуючих механізмів мемоізації. Встановлено, що глибоко вкладені структури даних створюють

експоненційне зростання кількості взаємозв'язків, що ускладнює прогнозованість і стабільність процесу оновлення.

2. Вперше розроблено модель точкового оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, у якій процес оновлення формалізовано як задачу мінімізації кількості DOM-операцій за обмежень коректності відображення. У межах моделі введено поняття області впливу змін та мінімальної множини DOM-вузлів, що підлягають оновленню, що дозволило формалізувати принцип локальності та забезпечити теоретичне підґрунтя для побудови ефективних алгоритмів. Вперше формально обґрунтовано застосування графа залежностей на рівні окремих значень даних як базового механізму поширення змін. Доведено, що такий підхід забезпечує зменшення області оновлення до мінімально необхідної підмножини елементів та дозволяє досягти асимптотичної складності $O(k)$, де k – кількість залежних вузлів, що є суттєвим покращенням порівняно з традиційними підходами ($O(n)$, $O(n \cdot \log n)$ і вище).

3. Запропоновано метод точкового оновлення інтерфейсу вебзастосунків, що базується на fine-grained реактивності, графах залежностей та використанні НОС-обгортки як проміжного рівня логіки. Метод забезпечує детермінований цикл обробки змін із використанням механізмів `dependencyList` та перевірки релевантності змін, що дозволяє уникнути надлишкових оновлень і забезпечити оновлення лише тих компонентів, які безпосередньо залежать від змінених даних.

4. Проведено моделювання прототипу програмного додатку для точкового оновлення інтерфейсу вебзастосунків та його програмну реалізацію, як клієнтського вебзастосунку на базі React та Redux, у якому структура інтерфейсу задається декларативно у вигляді JSON-об'єкта, а стан усіх елементів централізовано зберігається у глобальному сховищі. Повторне оновлення компонентів відбувається лише у разі, якщо зміна стану безпосередньо впливає на відповідний елемент. Ключовою особливістю реалізації є функція `withDependencyWatcher` – НОС, що приймає React

компонент і повертає новий компонент із розширеною поведінкою. Вхідний компонент не змінюється безпосередньо, натомість він «обгортається» додатковим рівнем логіки, який відповідає за відстеження змін у глобальному стані та приймає рішення щодо необхідності оновлення конкретного екземпляра компонента. Для кожного компонента формується список залежностей `dependencyList`, який обчислюється один раз під час ініціалізації. Проведено реалізацію прототипу програмного продукту засобами нативного JavaScript з метою визначення ефективності запропонованого механізму локалізації оновлень без використання високорівневих фреймворків.

5. Виконане експериментальне дослідження, яке підтвердило ефективність запропонованої моделі та методу точкового оновлення інтерфейсу вебзастосунків, реалізованого як засобами сучасного фреймворку (TypeScript, React, Redux Toolkit, Higher-Order Components (HOC), React.memo, lodash.isEqual, Vite), так і нативного JavaScript. В обох реалізаціях встановлено суттєве скорочення часу виконання JavaScript-логіки (Scripting) порівняно з «примітивним підходом». Для React-прототипу при 1000 рядках час зменшився з 1137,67 мс до 488,33 мс, що відповідає приросту продуктивності 132,97 % або приблизно у 2,3 рази. Для реалізації на нативному JavaScript у сценарії «зміна одного поля» при 10 000 рядках час Scripting скоротився з 40312,67 мс до 86,67 мс, що відповідає приросту 46414,62 % або прискоренню приблизно у 465 разів. Це підтверджує, що локалізація оновлень забезпечує значний ефект незалежно від технологічної платформи реалізації.

Встановлено, що основним фактором підвищення продуктивності є локалізація оновлень та зменшення кількості повторних операцій над DOM/Virtual DOM. У React-реалізації використання Redux Toolkit, HOC-обгортки і механізмів мемоізації дозволило знизити навантаження на Rendering до 50,00 % (або у 1,5 рази) у сценарії «зміна папки» при 1000 рядках. Для нативного JavaScript встановлено, що при 10 000 рядках кількість оновлень DOM-елементів скоротилася з 70044 до 3 елементів у сценарії «зміна

папки», тобто більш ніж у 23 тис. разів, а у сценарії «зміна колонки» оновлювалися лише елементи відповідної колонки (10 000 елементів замість 70044, або приблизно у 7 разів менше), що підтверджує високу масштабованість запропонованого підходу.

Отримані результати засвідчили, що зі зростанням складності структури інтерфейсу ефективність запропонованого методу посилюється, а не зменшується. Для нативного JavaScript у сценарії «зміна папки» при 10 000 рядках час Scripting скоротився з 42226,33 мс до 334,33 мс, що відповідає приросту 12530,01% або прискоренню приблизно у 126 разів, тоді як Rendering зменшився з 3247,00 мс до 1713,00 мс (89,55 %, або майже у 1,9 раза). У сценарії «зміна колонки» час Scripting скоротився з 41554,33 мс до 684,00 мс, що означає приріст 5975,19 % або приблизно у 61 раз, а Rendering – з 3164,67 мс до 863,33 мс (266,56 %, або у 3,7 раза). Це підтверджує практичну придатність запропонованого підходу для створення високопродуктивних вебзастосунків зі складними деревовидними структурами даних та великими табличними інтерфейсами.

Проведене дослідження не вичерпує всіх аспектів поставленої проблеми, тому доцільним є продовження наукового пошуку у таких напрямках: розширення формальної моделі для роботи з гетерогенними структурами даних, що передбачає адаптацію моделі до сценаріїв, де дані мають змішану природу (масиви, графи, вкладені списки, динамічні типи), що дозволить застосовувати підхід у більш широкому спектрі вебзастосунків; дослідження можливостей поєднання запропонованого методу з архітектурами React, Vue, Angular, Solid, Svelte та Web Components для створення гібридних рішень; оптимізації графа залежностей для великих динамічних систем, що передбачає вивчення алгоритмів автоматичного спрощення, агрегації та компресії графа залежностей, що дозволить зменшити накладні витрати при роботі з інтерфейсами, які змінюють структуру в реальному часі; дослідження паралельного та асинхронного оновлення інтерфейсу, що включатиме використання Web Workers, OffscreenCanvas,

потоків та мікротасків для розподілу обчислень і зменшення блокувань основного потоку браузера; розробки адаптивних стратегій оновлення на основі поведінкових патернів користувача, які б автоматично змінювали частоту, глибину та пріоритетність оновлень залежно від активності користувача, типу взаємодії та контексту виконання; інтеграції машинного навчання для прогнозування областей оновлення; розширення прототипу до повноцінного фреймворку або бібліотеки; дослідження впливу оптимізованого рендерингу на енергоспоживання та мобільні платформи; вивчення можливостей застосування моделі поза вебсередовищем.

Література

1. Abdallah A., Anadani S., Ismail H. Internet-Based 3D Volumes with Signed Distance Fields: Establishing a WebGL Rendering Infrastructure. *Proceedings of the International Conference on Computer and Applications (ICCA)*. 2024. Pp. 1–6. DOI: <https://doi.org/10.1109/ICCA62237.2024.10927852>.
2. Badiy M., Amounas F., Azrour M., Hammoudeh M. Advanced image super-resolution using deep learning approaches. *Radioelectronic and Computer Systems*. 2025. Is. 1. Pp. 187–198. DOI: <https://doi.org/10.32620/reks.2025.1.13>.
3. Bekmanova G., Yergesh B., Omarbekova A. et. al. Requirements for the Development of a Website Builder with Adaptive Design. *Proceedings of the 9th International Conference on Computer Science and Engineering (UBMK)*. 2024. Pp. 265–270. DOI: <https://doi.org/10.1109/UBMK63289.2024.10773412>.
4. Bernatska N., Dzhumelia E., Kochan O., Salamon I. Digital economy and smart financial management: using ai tools for financial literacy. *Advances in Transdisciplinary Engineering: Artificial Intelligence, Medical Engineering and Education. 8th International Conference (AIMEE 2024)*. Huangshi, China, 26-27 October. 2024. 2025. Volume 65. Pp. 158-167 DOI: [10.3233/ATDE250123](https://doi.org/10.3233/ATDE250123).
5. Bernatska N., Dzhumelia E., Kochan O., Salamon I., Typilo I. Web-based platform for real-time data monitoring visualization: analysis and perspectives. In: Hu, Z., Yanovsky, F., Dychka, I., He, M. (eds) *Advances in Computer Science for Engineering and Education VII (ICCSEEA 2024). Lecture Notes on Data Engineering and Communications Technologies*, 2025. Vol. 242. Springer, Cham. Pp. 271-283. DOI: https://doi.org/10.1007/978-3-031-84228-3_23.
6. Bernatska N., Dzhumelia E., Typilo I., Kochan O., Salamon I. Neural networks for environmental monitoring: a web application for image recognition. *Advances in Transdisciplinary Engineering: Artificial Intelligence, Medical Engineering and Education. 7th International Conference (AIMEE 2023)*. Guangzhou, China, 9–10 November 2023. 2024. Volume 48. Pp. 3-15 DOI: [10.3233/ATDE231310](https://doi.org/10.3233/ATDE231310).

7. Beyond the Virtual DOM: The Rise of Fine-Grained Reactivity URL: <https://bryancode.dev/en/blog/beyond-the-virtual-dom-the-rise-of-fine-grained-reactivity>. (дата звернення 26.03.2026).
8. Boutsis A., Ioannidis C., Verykokou S. Multi-Resolution 3D Rendering for High-Performance Web AR. *Sensors*. Basel, Switzerland. 2023. №23(15). URL: <https://doi.org/10.3390/s23156885>.
9. Braik, M., Ryalat, M. H., Al-Zoubi, H. A novel meta-heuristic algorithm for solving numerical optimization problems: Ali Baba and the forty thieves. *Neural Computing and Applications*/ 2022 34. Pp. 409–455. DOI: <https://doi.org/10.1007/s00521-021-06392-x>.
10. Carvalho F., Fialho P. Enhancing SSR in Low-Thread Web Servers: A Comprehensive Approach for Progressive Server-Side Rendering with any Asynchronous API and Multiple Data Models: *Proceedings of the 19th International Conference on Web Information Systems and Technologies*. 2023. Pp. 37–48. DOI: <https://doi.org/10.5220/0012165300003584>.
11. Carvalho F.M. Progressive Server-Side Rendering with Suspendable Web Templates. *Web Information Systems Engineering – WISE 2024*. 2024. Vol. 15440. DOI: https://doi.org/10.1007/978-981-96-0576-7_33.
12. Challa V. Comprehensive Analysis of Modern Application Rendering Strategies: Enhancing Web and Mobile User Experiences. *Journal of Engineering and Applied Sciences Technology*. 2022. Vol. 4. Is. 4. Pp. 1–6. DOI: [https://doi.org/10.47363/JEAST/2022\(4\)248](https://doi.org/10.47363/JEAST/2022(4)248)
13. Chen K. Improving Front-end Performance through Modular Rendering and Adaptive Hydration (MRAH) in React Applications. *Department of Electrical and Computer Engineering, Carnegie Mellon University Pittsburgh*. 2025. DOI: <https://doi.org/10.48550/ARXIV.2504.03884>.
14. Cherukuri B. R. Building scalable web applications: Best practices for backend architecture. *International Journal of Science and Research (IJSR)*. 2024. Vol. 13, No 10. P. 126–139. DOI: <https://dx.doi.org/10.21275/ES24928085711>.

15. Elisabeth Robson, Eric Freeman. Head First JavaScript Programming: A Brain-Friendly Guide. Фабула. 2022. 672 с.
16. Exploring Web Rendering: Progressive Hydration. Babel Magazine. URL: <https://surl.li/xdxlso>.
17. Frontend SSR frameworks benchmarked: Angular, Nuxt, NextJs and SvelteKit. URL: <https://surl.li/egjxr>.
18. Genne S. Optimizing Enterprise Web Performance Through Server-Side Rendering: A Next.js Framework Implementation. *European Modern Studies Journal*. 2025. Vol. 9. Is. 3. Pp. 255–264. DOI: [https://doi.org/10.59573/emsj.9\(3\).2025.21](https://doi.org/10.59573/emsj.9(3).2025.21).
19. Germán Cocca «What is Memoization? How and When to Memoize in JavaScript» URL: <https://www.freecodecamp.org/news/memoization-in-javascript-and-react/>
20. Gorai S. K. Web Development Barriers and Challenges . *International journal of scientific research in engineering and management*. 2025. Vol. 09, iss. 01. P. 1–9. DOI: <https://doi.org/10.55041/ijrem40734>.
21. Hamzaturrazak M., Jonemaro E., Pinandito A. Performance Analysis of 3D Rendering Method on Web-Based Augmented Reality Application Using WebGL and OpenGL Shading Language. *Proceedings of the 8th International Conference on Sustainable Information Engineering and Technology*. 2023. Pp. 637–643. DOI: <https://doi.org/10.1145/3626641.3626949>.
22. Hanol M. I Benchmarked. Signals vs Virtual DOM - Here's What I Found. DEV. 2025. URL: https://dev.to/mike_hanol_e21eef42461b5e/i-benchmarked-signals-vs-virtual-dom-heres-what-i-found-3do7.
23. Harrison Oke Ekpobimi. Building high-performance web applications with NextJS. *Computer Science & IT Research Journal*. 2024. Is. 5(8). Pp. 1963–1977. DOI: <https://doi.org/10.51594/csitrj.v5i8.1459>.
24. Honey Sharma «Introduction to Web Workers» URL: <https://medium.com/naukri-engineering/introduction-to-web-workers-3dc8c7dce0be>

25. Hubert O. How Virtual DOM works? Myths, facts and practice. Uniquedevs 2025. URL: <https://uniquedevs.com/en/blog/how-virtual-dom-works-myths-facts-and-practice/>.(дата звернення 27.03.2026)
26. Iskandar T., Lubis M., Fabrianti K. T., Ridho Lubis A. Comparison between client-side and server-side rendering in the web development. *IOP Conference Series: Materials Science and Engineering*. 2020. Vol. 801. . DOI: <https://doi.org/10.1088/1757-899X/801/1/012136>.
27. Islands Architecture Definition. Sanity. URL: <https://surl.li/dkbgjr>.
28. Islands Architecture. JuniorDev4Life. URL: <https://surl.li/ventzt>.
29. ISO 9001:2015 Quality management systems. Requirements. 2015. URL: <https://www.iso.org/standard/62085.html>.
30. ISO 9004:2018 Quality management. Quality of an organization. Guidance to achieve sustained success. 2018. URL: <https://www.iso.org/standard/70397.html>.
31. ISO/IEC 25010:2023 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuARE) - Product quality model. 2023. URL: <https://surl.li/mrvhap>.
32. ISO/IEC/IEEE 90003:2018 Software engineering. Guidelines for the application of ISO 9001:2015 to computer software. 2018. URL: <https://www.iso.org/standard/74348.html>.
33. Javad. Virtual DOM vs Reactive DOM. 2025. URL: <https://dev.to/javadinteger/virtual-dom-vs-reactive-dom-5g41>.
34. Kalyanaraman K. A deep dive into React Fiber. *LogRocket*. URL: <https://blog.logrocket.com/deep-dive-react-fiber/>.
35. Khan, S. M. Onion Architecture Used in Software Development. 2023. URL: https://www.researchgate.net/publication/371006360_Onion_Architecture_Used_in_Software_Developmen.
36. Khushbu Choksi. What are CSR, SSR, SSG, and ISR? 04.07.2024 URL <https://medium.com/@khushbuchoksi/what-are-csr-ssr-ssg-and-isr-d59201e3b060>

37. Kuchuk N., Shyshatskyi A., Zhuravskyi Y., Stasiuk T., Nalapko O., Sliusar P., Protas N., Shaposhnikova O., Pronin S., Havryliuk O. The development of a method for visualizing the states of the national security system. *Technology Audit and Production Reserves*. 2023. № 5/2(73). Pp. 18-21. DOI: <https://doi.org/10.15587/2706-5448.2023.285986>.
38. Kurapati L. Different Types of Architectures in Frontend Design and Development. *International Journal For Multidisciplinary Research*. Is. 6(5). . DOI: <https://doi.org/10.36948/ijfmr.2024.v06i05.28707>.
39. Li L., Huang, Y., Qiao X. et. al. Toward Distributed Collaborative Rendering Service for Immersive Mobile Web. *IEEE Network*. 2024. Vol. 38. Is. 3. Pp. 137–145. . DOI: <https://doi.org/10.1109/MNET.133.2200524>.
40. Li S. The Application of Deep Learning in Image Super-resolution Reconstruction and Enhancement Technology. *Proceedings of the 2024 9th International Conference on Cyber Security and Information Engineering*. 2024. Pp. 350–355. . DOI: <https://doi.org/10.1145/3689236.3689246>.
41. Lytvynov O., Hruzyn D. Methods for optimizing the loading and updating of web pages using cloud technologies. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2023. Is. 4, Pp. 40–50. DOI: <https://doi.org/10.32782/IT/2023-4-5>.
42. Majumder, A. S. The Influence of UX Design on User Retention and Conversion Rates in Mobile Apps. *arXiv*. 2025. DOI: <https://doi.org/10.48550/arXiv.2501.13407>.
43. MEAN Stack Tutorials: Vskills. Cubezoid Solutions Pvt. Ltd., New Delhi. URL: <https://www.vskills.in/certification/tutorial/document-object-model-dom-3/>
44. Michal Aibin, Milos Simic. From «Recursive to Iterative» URL: <https://www.baeldung.com/cs/convert-recursion-to-iteration>
45. Monali Bedre. Decoding React JS: The Distinction Between Virtual DOM and Shadow DOM and Real DOM. 2024. URL:

<https://www.linkedin.com/pulse/decoding-react-js-distinction-between-virtual-dom-shadow-monali-bedre-mf6re/>. (дата звернення 30.03.2026).

46. Nasenok K., Voitsekhovska M. Client-side rendering issues in the modern worldwide networ. *Technical sciences and technologies*. 2024. Is. 4(38), Pp. 197–207. . DOI: [https://doi.org/10.25140/2411-5363-2024-4\(38\)-197-207](https://doi.org/10.25140/2411-5363-2024-4(38)-197-207).

47. Nikulina O. and Khatsko K. Method of converting the monolithic architecture of a front-end application to microfrontends. *Bulletin of National Technical University "KhPI". Series: System Analysis, Control and Information Technologies*. 2 (10). (Dec. 2023). C.79–84. DOI: <https://doi.org/10.20998/2079-0023.2023.02.12>.

48. Ollila R., Mäkitalo N., Mikkonen T. Modern Web Frameworks: A Comparison of Rendering Performance. *Journal of Web Engineering*. 2022. №21(03), Pp. 789–814. DOI: <https://doi.org/10.13052/jwe1540-9589.21311>.

49. Ollila, R., Mäkitalo, N., & Mikkonen, T. Modern Web Frameworks: A Comparison of Rendering Performance. *Journal of Web Engineering*. 21(03), p.789-814. DOI: <https://doi.org/10.13052/jwe1540-9589.21311>.

50. Owaid S. R., Zhuravskyi Y., Lytvynenko O., Veretnov A., Sokolovskyi D., Plekhova G., Hrinkov V., Pluhina T., Neronov S., & Dovbenko O. Development of a method of increasing the efficiency of decision-making in organizational and technical systems. *Eastern-European Journal of Enterprise Technologies*. 2024. 1(4 (127)), 14–22. DOI: <https://doi.org/10.15587/1729-4061.2024.298568>.

51. Patil O. Real Time Inventory Management System powered by Generative User Interface. *Interantional journal of scientific research in engineering and management*. 2024. Is. 8(4). Pp. 1–5. DOI: <https://doi.org/10.55041/IJSREM32623>.

52. Pavić F., Brkić L. Methods of Improving and Optimizing React Web-applications. *Proceedings of the 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. 2021. Pp. 1753–1758. DOI: <https://doi.org/10.23919/MIPRO52101.2021.9596762>.

53. Pellegrino G., Catakoglu O., Balzarotti D., Rossow C. Uses and Abuses

of Server-Side Requests. *Research in Attacks, Intrusions, and Defenses*. 2016. Vol. 9854, Pp. 393–414. DOI:https://doi.org/10.1007/978-3-319-45719-2_18.

54. React 18: Partial Hydration. *Medium*. URL: <https://surl.li/ivdlzl>.

55. React. (n.d.). Render and commit: Optimizing performance. URL: <https://react.dev/learn/render-and-commit#optimizing-performance>.

56. Saiapina I. Relational Database Work Optimization Of The Transport Information System. *Transport Systems and Technologies*. 2022. (40). Pp. 261–266. DOI: <https://doi.org/10.32703/2617-9040-2022-40-23>.

57. Sathyakumar D. Composable Renderer Services: A Study of Architectures for Dynamic View Transclusion in Distributed Web Applications. *2024 IEEE Cloud Summit*. 2024. Pp. 178–187. DOI: <https://doi.org/10.1109/Cloud-Summit61220.2024.00036>.

58. Sereda, D. Creating a Multi-tier Architecture for Web Applications: Design and Implementation. *American Scientific Research Journal for Engineering, Technology, and Sciences*. Vol. 102 № 1. 2025. P. 134-139. https://asrjetsjournal.org/American_Scientific_Journal/article/view/11686.

59. Server-Side Rendering (SSR). Vue.org URL: <https://surl.lt/vuyrkw>.

60. Shadow DOM/ IONOS editorial team. IONOS Digital Guide. 2023 URL: <https://www.ionos.ca/digitalguide/websites/web-development/shadow-dom/>

61. Shah H. Advancing Web Development – Enhancing Component-Based Software Engineering and Design Systems Through HTML5 Customized Built-in Elements // *International journal of Web & Semantic Technology*. 2024. Vol. 15, iss. 1. P. 15–26. DOI: [10.5121/ijwest.2024.15102](https://doi.org/10.5121/ijwest.2024.15102).

62. Sharma, N., Charan, S., , S., & , S. Performance and Developer Experience Comparison of Redux, Zustand, and Context API in React Applications. *International Journal on Science and Technology*. 2025. DOI: <https://doi.org/10.71097/ijst.v16.i2.5026>.

63. Shrinivass A. Signal-First Architectures: Rethinking Front-End Reactivity. *ArXiv:2506.13815*. 2025. DOI: <https://doi.org/10.48550/arXiv.2506.13815>.

64. Shyshatskyi A., Zhuravskyi Y., Plekhova G., Shostak I., Feoktystova O., Dmytriieva O., Starynskyi I., Strepetov A., Rudoi S., Zakharov A. Development of a polymodel resource management complex for intelligent decision support systems. *Eastern-European Journal of Enterprise Technologies*. 2025. 5(4 (137)). C. 41–63. DOI: <https://doi.org/10.15587/1729-4061.2025.340387>.
65. Sidorov D. V. Analyzing Virtual DOM Performance in Front-End Frameworks. *Наука і техніка*. 2024. №8(36). C.841–858. URL: <https://perspectives.pp.ua/index.php/nts/article/view/14379/14449>.
66. Sumit S. The Next.js 15 Streaming Handbook — SSR, React Suspense, and Loading Skeleton. URL: <https://surl.li/ozqdcc>.
67. Sun Y. Server-Side Rendering. *Practical Application Development with AppRun*. 2019. Pp. 191–217. DOI: https://doi.org/10.1007/978-1-4842-4069-4_9.
68. Sundaramoorthy S. Uml Diagramming: A Case Study Approach. New York: Auerbach Publications, 2022. 430 p. DOI: 10.1201/9781003287124. DOI: <https://doi.org/10.1201/9781003287124>.
69. Tapas Adhikary. Synchronous vs Asynchronous JavaScript – Call Stack, Promises, and More. URL: <https://www.freecodecamp.org/news/synchronous-vs-asynchronous-in-javascript/>
70. Tarkhan A. B., Zhuravskyi Y., Shyshatskyi A., Pluhina T., Dudnyk V., Kiris I., Nalapko O., Protas N., Neronov S., & Nechyporuk V. Development of a solution search method using an improved fish school algorithm. *Eastern-European Journal of Enterprise Technologies*. 2023. 4 (124). C. 27–33. DOI: [10.15587/1729-4061.2023.284315](https://doi.org/10.15587/1729-4061.2023.284315).
71. Tetiana A. Vakaliuk, Dariia G. Skripchenko, Mariia O. Medvedieva and Mykhailo G. Medvediev. Development of an intelligent chatbot for a hospital website. *Proceedings of The Seventh International Workshop on Computer Modeling and Intelligent Systems (CMIS-2024)*, Zaporizhzhia, Ukraine, May 3, 2024. Edited by Sergey Subbotin. CEUR Workshop Proceedings. 2024. Vol. 3702. Pp. 317-328. DOI: <https://doi.org/10.5281/zenodo.12636164>.

72. Thakkar M. Building React Apps with Server-Side Rendering: Use React, Redux, and Next to Build Full Server-Side Rendering Applications. Apress. 2020. DOI: <https://doi.org/10.1007/978-1-4842-5869-9>.

73. TsungTingDu. Virtual DOM. 2020/ URL: <https://ithelp.ithome.com.tw/articles/10248434>

74. Vakaliuk Tetiana A., et al. Development of a web service for creating tests based on text analysis using natural language processing technologies. *International Journal of Research in E-Learning*. 2023. Vol. 9 (2). Pp. 1-22. DOI: <https://doi.org/10.31261/IJREL.2023.9.2.04>.

75. Vallamsetla K. The Impact of Server-Side Rendering on UI Performance and SEO. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*. 2024. Is. 10(5). Pp. 795–804. URL: <https://doi.org/10.32628/CSEIT241051067>.

76. Vesper. Measuring time-to-interactivity for modern web pages. In: *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation, NSDI*. USENIX Association. [Online; accessed 14. Nov. 2023] 2018. URL: https://www.cs.princeton.edu/~ravian/publications/vesper_nsd18.pdf.

77. Vyas R. Comparative analysis on front-end frameworks for web applications. *International Journal for Research in Applied Science and Engineering Technology*. 2022. Vol. 10. No7.P. 298-307. URL: <https://www.ijraset.com/research-paper/comparative-analysis-on-front-end-frameworks-for-web-applications>.

78. Wang Z., Yan B., Xing S. et. al. Research on large-scale 3D scene rendering method for web application. *Eighth Symposium on Novel Photoelectronic Detection Technology and Applications*. SPIE. 2022. P.731. DOI: <https://doi.org/10.1117/12.2627369>.

79. Which one is the best for SEO: Client-side Rendering (CSR) vs. Server-side Rendering (SSR) Virtual Oplossing Pvt Ltd.28.03.2025. URL: <https://www.linkedin.com/pulse/which-one-best-seo-client-side-rendering-csr-vs-server-side-kobec/>

80. World Wide Web Consortium. (n.d.). Resource Description Framework (RDF). URL: <https://www.w3.org/RDF>.
81. Xinghai, W. Conceptualizing Synergies Between SEO Practices and UX Design for Enhanced Website Usability and Engagement. *Journal of Digitainability, Realism & Mastery (DREAM)*. 2023. 2(05). C.18–22. DOI: <https://doi.org/10.56982/dream.v2i05>.
82. Xuan J., Huang Y., Guo Y., Jia B. Image-Based 3D Gaussian Rendering: A Hybrid Approach for Scene Visualization. *2024 12th International Conference on Information Systems and Computing Technology (ISCTech)*. 2024. Pp. 1–5. URL: <https://doi.org/10.1109/ISCTech63666.2024.10845536>.
83. Ye L., Zhou C., Peng H. et. al. Multi-level feature interaction image super-resolution network based on convolutional nonlinear spiking neural model. *Neural Networks*. 2024. URL: <https://doi.org/10.1016/j.neunet.2024.106366>.
84. Yerokhin, A., & Kameniev, D. Optimizing re-rendering in web applications: problem analysis and a React-based solution. *Management Information System and Devises*. 2025. 184. Pp. 90-99 DOI: <https://doi.org/10.30837/0135-1710.2025.184.090>.
85. Yu G., Liu C., Fang T. et. al. A survey of real-time rendering on Web3D application. *Virtual Reality & Intelligent Hardware*, 2023. Vol. 5. Is. 5, Pp. 379–394. DOI: <https://doi.org/10.1016/j.vrih.2022.04.002>.
86. Yu M., Shi J., Xue C. et. al. A review of single image super-resolution reconstruction based on deep learning. *Multimedia Tools and Applications*. 2023. Vol. 83. Is. 18. Pp. 55921–55962. DOI: <https://doi.org/10.1007/s11042-023-17660-4>.
87. Zhang, Y., Oliveira, M., & Bennett, C. Improving Front-end Performance through Modular Rendering and Adaptive Hydration (MRAH). 2025. URL: <https://arxiv.org/abs/2504.03884>. (дата звернення: 20.01.2026).
88. Zhebka V., Gertsyuk M., Sokolov V., Malinov V., Sablina M. Optimization of Machine Learning Method to Improve the Management Efficiency

of Heterogeneous Telecommunication Network CEUR Workshop Proceedings, 2022, 3288, Pp. 149–155. URL: <https://elibrary.kubg.edu.ua/id/eprint/43202>.

89. Архітектурні підходи до розробки масштабованих вебзастосунків / О.В. Склярєнко та ін. Кібербезпека: освіта, наука, техніка. 2024. Вип. 24. Т. 4. С. 341–350.

90. Бернацька Н., Джумеля Е., Кочан О. Розробка веб-інтерфейсу інформаційно-аналітичної системи екологічного моніторингу з використанням бібліотеки React. *Інформаційні технології та суспільство*. 2023. Вип.1 (7). С. 6-12. DOI: <https://doi.org/10.32689/maup.it.2023.1.1>.

91. Бондарчук А. П., Хлиста І. А. Вирішення проблеми часткового оновлення вмісту веб-сторінки шляхом оптимізації параметрів SPA. *Телекомунікаційні та інформаційні технології*. 2023. № 2. С. 45-52. DOI: [10.31673/2412-4338.2023.041320](https://doi.org/10.31673/2412-4338.2023.041320).

92. Бондарчук, А., Жебка, В., Корецька, В., Шилкіна, А. Порівняльна характеристика web-орієнтованих інструментів автоматизації освітнього процесу в умовах цифрової трансформації. *Публічно-управлінські та цифрові практики*. 2024. (1). С.13-21. DOI: <https://doi.org/10.31673/2786-7412.2023.011327>.

93. Брухтій С. С. Дослідження методів рендерингу web-сайтів для пошукової оптимізації сайтів (SEO): пояснювальна записка до кваліфікаційної роботи здобувача вищої освіти на другому (магістерському) рівні, спеціальність 121 – Інженерія програмного забезпечення. Харків. нац. ун-т радіоелектроніки. Харків. 2024. 66 с. URL: <https://openarchive.nure.ua/handle/document/28936>.

94. Вакалюк Т.А., Антонюк Д. С., Мінтій І. С., Новіцька І. В., Довгалюк І. С. Методи побудови рекомендаційних систем та інтеграція штучного інтелекту у фінансових додатках. *Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки*. 2025. Т. 36(75), № 3(2). С. 120–124. DOI: [10.32782/2663-5941/2025.3.2/16](https://doi.org/10.32782/2663-5941/2025.3.2/16).

95. Войтюк О. В. Плечистий Д. Д. Методи оптимізації рекурсивних циклів в JavaScript. Міжнародна конференція «Передові технології в інформаційно-комунікаційній інженерії» (ATICE'2024): Матеріали конф. м. Одеса, Україна, 17-20 лип., 2024 р. / гол. ред. кол. С. В. Ківалов. Одеса, Міжнар. гуманітар. ун-т. 2024. С.12-14. URL: <https://hdl.handle.net/11300/28155>; DOI:10.32837/11300.28155.

96. Войтюк О. В. Аналіз методів та технологій рендерингу вебзастосунків. *Вчені записки Таврійського національного університету імені В.І. Вернадського*. Серія: Технічні науки. 2025. Том 36 (75). № 6. С.74-80. DOI: <https://doi.org/10.32782/2663-5941/2025.6.2/11>.

97. Войтюк О. В. Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. Тези V Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення». м. Житомир, 01-02 грудня 2022 р. м. Житомир: «Житомирська політехніка». 2022 р. С. 12-13. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2023/02/12.pdf>.

98. Войтюк О. В., Плечистий Д. Д. Багатозалежні зв'язки у рекурсивній структурі фронтенд додатку. Тези VI Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення». м. Житомир, 29-30 листопада 2023 р., м. Житомир: «Житомирська політехніка». 2023. С.56-57. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2024/01/56-1.pdf>.

99. Войтюк О.В. Глибоко вкладені структури даних із багатозалежними зв'язками: продуктивність та оновлення стану при рендерингу. Тези VIII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення, 02-03 грудня 2025 р. м. Житомир: «Житомирська політехніка». С.293-294. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2026/01/293.pdf>.

100. Войтюк О.В., Лобанчикова Н.М. Методи оптимізації швидкодії оновлення вебзастосунків на основі об'єктів з глибокою вкладеністю.

Computer Science and Applied Mathematics. 2026. № 1. С.56-68. DOI: <https://doi.org/10.26661/2786-6254-2026-1-06>.

101. Войтюк О.В., Плечистий Д.Д. Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. *Технічна інженерія*. 2023. № 1 (91). С. 140-145. DOI: [https://doi.org/10.26642/ten-2023-1\(91\)-140-145](https://doi.org/10.26642/ten-2023-1(91)-140-145).

102. Гаватюк М. В., Саяпіна І. О. Удосконалений метод цільового оновлення інтерфейсу користувача для підвищення ефективності вебзастосунків на основі реактивних потоків та віртуального DOM. *Технічна інженерія*.(1(95). 2025. С. 259–265. DOI: [https://doi.org/10.26642/ten-2025-1\(95\)-259-265](https://doi.org/10.26642/ten-2025-1(95)-259-265).

103. Гончар В.М. Порівняння різних видів рендеру інформації на вебсторінку. *Прикладні аспекти сучасних міждисциплінарних досліджень*. 2022. С. 103-105. URL: <https://jpasmd.donnu.edu.ua/article/view/12929>.

104. Громський О.О., Вакалюк Т.А., Нерода С.І. Аналіз моделей аналітичних веб-систем прогнозу міжбіржової вартості цифрових криптоактивів. *Вісник Херсонського національного технічного університету*. 2025. Том 2. №1(92). С.51-59. DOI: <https://doi.org/10.35546/kntu2078-4481.2025.1.2.7>

105. Двірничук К.В., Вацек Д.О. Веб-програмування та веб-дизайн: навчальний посібник: ел.вид. Чернівці: Чернівецький національний університет ім. Ю.Федьковича, 2022. 472 с. Режим доступу: <https://shorturl.at/dfAkF>.

106. Євдокимов В. В. Морозов А. В. Вакалюк Т. А. Формалізовані методи та алгоритми для створення рекомендаційних систем. *Актуальні проблеми автоматизації та інформаційних технологій*. 2024. №28. С.120–133. . DOI: <http://dx.doi.org/10.15421/432411>.

107. Козуб Г. О., Сурма Ю. Ю., Артеменко О. І. Роль вебкомпонентів у побудові сучасних інтерфейсів: переваги та обмеження. *Вісник Херсонського*

національного технічного університету. 2025. №2(93). Том 2. С. 175-180. DOI: <https://doi.org/10.35546/kntu2078-4481.2025.2.2.21>.

108. Любчак В. О., Савостян В. В., Козолуп П. Д. Безпека Java-додатків: навчальний посібник. Суми: Сумський державний університет. 2023. – 72 с. URL: <https://essuir.sumdu.edu.ua/server/api/core/bitstreams/4819a931-45ea-44ce-96b6-62620b440de3/content>.

109. Мельниченко Д.В., Вакалюк Т.А., Форрахов О.В., Гордієнко І.В. Проектування вебзастосунку для керування робочим часом. *Вісник Херсонського національного технічного університету*. 2024. №3(90). С.238-250. DOI: [10.35546/kntu2078-4481.2024.3.30](https://doi.org/10.35546/kntu2078-4481.2024.3.30).

110. Морозов А. В., Шапіренко С. Л. Автоматизований аналіз клієнтських відгуків для прийняття управлінських рішень у сервісних системах. *Технічна інженерія*. 2025. 2(96). С.108-116. DOI: [https://doi.org/10.26642/ten-2025-2\(96\)-108-116](https://doi.org/10.26642/ten-2025-2(96)-108-116).

111. Морозов А., Вакалюк Т., Кубрак Ю., Зосімович Д. Аналіз факторів впливу на архітектури програмних систем. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2022. №1. С. 44–52. DOI: <https://doi.org/10.32782/IT/2022-1-7>.

112. Непийвода М., Рекало О., Мичуда Л., Коробейнікова Т. Web applications technological stack. *Modern Engineering and Innovative Technologies*, 1(22-01). С. 104–112. DOI: <https://doi.org/10.30890/2567-5273.2022-22-01-038>.

113. Павлюшин М.Ю., Саяпіна І.О. Метод та програмне забезпечення для оптимізації продуктивності баз даних. *Системні технології*. 3(164). 2026. С.227-234. DOI: <https://doi.org/10.34185/1562-9945-3-164-2026-18>.

114. Пірог О.В. Безпека вебдодатків: навч.посібн.. Електронні дані. Житомир: Житомирська політехніка.2025. 290с.

115. Посібник: знайомство з React. URL: <https://uk.legacy.reactjs.org/tutorial/tutorial.html>. (дата звернення 30.03.2026).

116. Рендеринг на стороні серверу (SSR). URL: <https://ua.vuejs.org/sponsor/>.

117. Самулінас С., Житеньова Н. Інновації в UI/UX-дизайні як ключ до успішного просування вебресурсів. *Professional Art Education*. 2025. 6(1). С.93–106. DOI: <https://doi.org/10.34142/27091805.2025.6.01.09>.
118. Свінцицька О. М., Граф М. С., Нікітчук Т.М. «Метод Use Case в плануванні проєктів з інформаційних технологій». *Технічна інженерія*, 2022. Вип. 1(89). С. 77-84. DOI:[10.26642/ten-2022-1\(89\)-77-84](https://doi.org/10.26642/ten-2022-1(89)-77-84).
119. Сипягін О. І., Южаков М. А., Ібрагімов Р. І. Архітектурні рішення для масштабованих вебзастосунків з урахуванням продуктивності та затримок. *Вісник Херсонського національного технічного університету*. 2025. №3(94). Том 2. С.417-422. DOI: <https://doi.org/10.35546/kntu2078-4481.2025.3.2>.
120. Тулашвілі Юрій. Web-програмування. MERN fullstack development: навчальний посібник. Луцьк:РВВ ЛНТУ, 2023.242 с.
121. Фещенко С.О., Мацієвський В.А., Нікітчук Т.М., Коренівська О.Л., Войтюк О.В. Розробка вебзастосунку для дослідження реологічних властивостей біологічних рідин. Тези XII Міжнародної науково-практичної конференції «Topical aspects of modern scientific research», 8-10 серпня 2024 р. Токіо, Японія. С. 98-101. URL: <https://surl.li/ixykb>.
122. Філімончук Т. В., Золотайко Є. І., Севостьянова О. М. Модель архітектури вебзастосунку на основі сучасних багатошарових архітектур. *Вісник Херсонського національного технічного університету*. 2026. №2(97). С.411-424. DOI: <https://doi.org/10.35546/kntu2078-4481.2026.2.50>.
123. Харов'юк В. React та Віртуальний DOM: Чому оновлення сторінки стало блискавично швидким? Digital WebCraft. 2026. URL: <https://webcraft.org/blog/react-ta-virtualniy-dom-chomu-onovlennya-storinki-stalo-bliskavichno-shvidkim>.
124. Чуб І.М., Демченко К.В. Оптимізація продуктивності рішень побудованих за допомогою бібліотек REACTJS та D3. *Вісник Харківського національного автомобільно-дорожнього університету*. 2024. Вип. 104. С. 15-21. DOI: <https://doi.org/10.30977/BUL.2219-5548.2024.104.1.15>.

Додатки

Діаграми

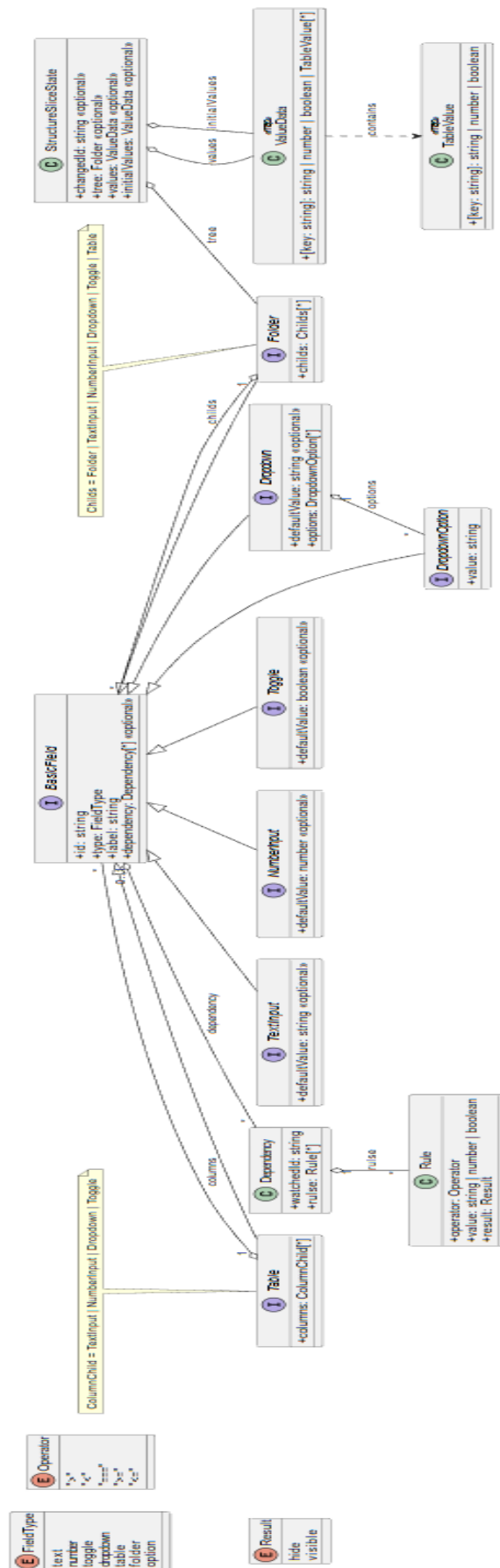


Рис. А.1. Діаграма класів для контрактів властивостей елементів

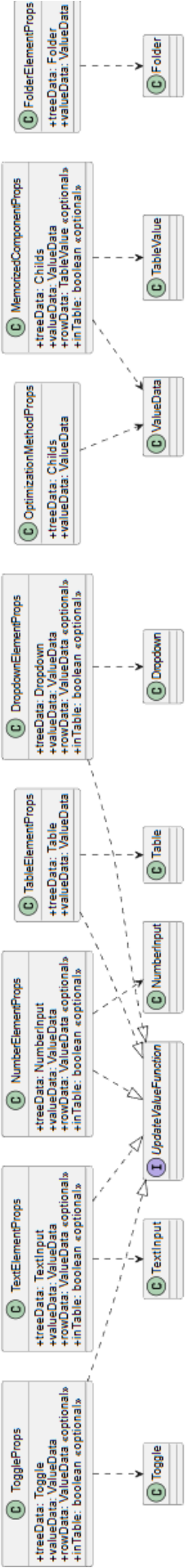


Рис. А.2. Діаграма класів для контрактів властивостей компонентів (props)

Лістинг програмного коду з використанням стеку фронтенд технологій

index.html

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<link rel="icon"
type="image/svg+xml"
href="/vite.svg" />
<meta name="viewport"
content="width=device-width, initial-
scale=1.0" />
<title>React Redux App</title>
</head>
<body>
<noscript>You need to enable
JavaScript to run this app.</noscript>
<div id="root"></div>
<script type="module"
src="/src/main.tsx"></script>
</body>
</html>
```

src/App.css

```
.App {
width: 100%;
height: 100%;
};
```

src/App.tsx

```
import "./App.css"
import { SimpleTree } from "../simple-
tree";
import DataWrapperForComplexTree
from "../complex-tree";
export const App = () => {
```

```
return (
<div className="App">
<SimpleTree />
<DataWrapperForComplexTree />
</div>
);
};
src/HOC/types.tsx
import { Childs, TableValue,
ValueData } from "../data/types";
export interface
MemorizedComponentProps {
treeData: Childs;
valueData: ValueData;
rowData?: TableValue;
inTable?: boolean;
};
src/HOC/utils.tsx
import { Childs } from "../data/types";
export const getDependencyList =
(treeData: Childs): string[] => {
return treeData.dependency
?
treeData.dependency.map(dependenc
y => dependency.watchedId)
: [];
};
src/HOC/withDependencyWatcher.tsx
import { memo, useCallback,
useEffect, useState } from "react";
import lodash from 'lodash';
import { useAppDispatch,
useAppSelector } from
"../app/hooks";
```

```

import { clearChangedId,
updateValues } from
"../app/structureSlice";
import { getDependencyList } from
"./utils";
import { MemorizedComponentProps
} from "./types";
import { ComplexId } from
"../utils/types";
import { TableValue } from
"../data/types";
const withDependencyWatcher =
(component: any) => {
const MemorizedComponent =
memo(component, lodash.isEqual);
return (props:
MemorizedComponentProps): any =>
{
const [localProps, setLocalProps] =
useState(props);
const [dependencyList] =
useState<string[]>(getDependencyList(
props.treeData));
const changedId =
useAppSelector(state =>
state.structure.changedId);
const newValueData =
useAppSelector(state =>
state.structure.values);
const dispatch = useAppDispatch();
const updateValue = useCallback((id:
string | ComplexId, newValue: string |
number | boolean) => {
dispatch(clearChangedId());
setTimeout(() => {
let changedId = id;
if (props.inTable && props.rowData)
{

```

```

const tableindex =
(props.valueData[props.treeData.id]
as TableValue[]).findIndex(data =>
data.id === props.rowData.id);
const complexId = { tableId:
props.treeData.id, elementId: id as
string, rowIndex: tableindex };
changedId = complexId;
}
dispatch(updateValues({ id:
changedId, newValue }));
}, 0);
}, [dispatch]);
useEffect(() => {
if (newValueData && changedId &&
dependencyList.includes(changedId))
{
setLocalProps({...props, valueData:
newValueData});
}
}, [changedId]);
return <MemorizedComponent
{...localProps}
updateValue={updateValue}/> ;
}
};
export default
withDependencyWatcher;
src/app/hooks.ts
import { TypedUseSelectorHook,
useDispatch, useSelector } from
'react-redux';
import type { RootState, AppDispatch
} from './store';
export const useAppDispatch = () =>
useDispatch<AppDispatch>();
export const useSelector:
TypedUseSelectorHook<RootState>
= useSelector;

```

```

src/app/store.ts
import { configureStore } from
 '@reduxjs/toolkit';
import structureReducer from
 './structureSlice';
export const store = configureStore({
  reducer: {
    structure: structureReducer,
  },
});
export type RootState =
  ReturnType<typeof store.getState>;
export type AppDispatch = typeof
  store.dispatch;
src/app/structureSlice.ts
import { createSlice, PayloadAction }
  from '@reduxjs/toolkit';
import { Folder, ValueData } from
  '../data/types';
import { ComplexId } from
  '../utils/types';
export type StructureSliceState = {
  changedId?: string;
  tree?: Folder;
  values?: ValueData;
  initialValues?: ValueData;
};
const initialState: StructureSliceState
  = {
  changedId: undefined,
  tree: undefined,
  values: undefined,
  initialValues: undefined
};
const structureSlice = createSlice({
  name: 'structure',
  initialState,
  reducers: {

```

```

  setInitialStructure(state, action:
    PayloadAction<Pick<StructureSliceSt
      ate, 'tree' | 'values'>>) {
    state.tree = action.payload.tree;
    state.values = action.payload.values;
    state.initialValues =
      action.payload.values;
  },
  clearChangedId(state) {
    state.changedId = undefined;
  },
  updateValues(state, action:
    PayloadAction<{ id: string |
      ComplexId, newValue: string |
      number | boolean}>) {
    const { id, newValue } =
      action.payload;
    const newValues = Object.assign({},
      state.values) as ValueData;
    if (typeof id === 'string') {
      newValues[id] = newValue;
    } else {
      (newValues[id.tableId] as
        ValueData[])[id.rowIndex][id.element
          Id] = newValue
    }
    state.changedId = typeof id ===
      'string' ? id :
      `${id.tableId}[${id.rowIndex}][${id.el
        ementId}`;
    state.values = newValues;
  },
});
export const {
  setInitialStructure,
  updateValues,
  clearChangedId,
} = structureSlice.actions;

```

```

export default structureSlice.reducer;
src/complex-
fields/dropdown/index.css
.dropdown-wrapper {
width: 300px;
display: flex;
align-items: center;
margin: 5px 0;
}
.dropdown-wrapper-for-table {
width: 150px;
}
.dropdown-wrapper span {
white-space: nowrap;
}
.dropdown-input {
width: 100%;
margin-left: 10px;
}
src/complex-
fields/dropdown/index.tsx
import { useState } from "react";
import withDependencyWatcher from
"../../HOC/withDependencyWatcher";
import { getValueFromValueData }
from "../../utils";
import { DropdownElementProps }
from "./types";
import './index.css'
import { isElementVisible } from
"../../utils/dependencyChecker";
const DropdownElement =
({treeData, inTable, valueData,
updateValue, rowData}:
DropdownElementProps) => {
const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as string |
undefined;

```

```

const [localValue, setLocalValue] =
useState(valueFromValueData);
const handleChange = (e:
React.ChangeEvent<HTMLSelectEle
ment>) => {
const newValue = e.target.value;
setLocalValue(e.target.value);
updateValue(treeData.id, newValue);
};
return (
<div className={`dropdown-
wrapper ${inTable ? 'dropdown-
wrapper-for-table' : ''}`>
{!inTable &&
<span>{treeData.label}</span>
<select className="dropdown-input"
value={localValue}
onChange={handleChange}>
<option value="">Виберіть
значення</option>
{treeData.options.map(option => {
const optionisVisible =
isElementVisible(option, valueData);
return optionisVisible &&(
<option value={option.value}
key={option.id}>
{option.label}
</option>
)}
)}
</select>
</div>
);
};
export default
withDependencyWatcher(DropdownE
lement);
src/complex-
fields/dropdown/types.tsx

```

```

import { Dropdown, ValueData }
from "../../data/types";
import { UpdateValueFunction } from
"./utils/types";
export interface
DropdownElementProps extends
UpdateValueFunction {
treeData: Dropdown;
valueData: ValueData;
rowData?: ValueData;
inTable?: boolean;
};
src/complex-fields/folder/index.css
.folder {
display: flex;
flex-direction: column;
border: 1px solid black;
padding: 10px;
box-sizing: border-box;
margin-bottom: 10px;
}
.folder-label {
margin: 10px 0;
font-size: larger;
display: flex;
flex-direction: column;
}
src/complex-fields/folder/index.tsx
import { isElementVisible } from
"./utils/dependencyChecker";
import { FolderElementProps } from
"./types";
import withDependencyWatcher from
"./HOC/withDependencyWatcher";
import { renderChildElements } from
"./utils";
import './index.css'

```

```

const FolderElement = ({treeData,
valueData}: FolderElementProps) =>
{
const folderVisible =
isElementVisible(treeData,
valueData);
return (
<>
{folderVisible &&
<div className="folder">
<div className="folder-
label">{ Папка:
${treeData.label}}`</div>
{treeData.chilids.map(child =>
renderChildElements(child,
valueData)))}
</div>
}
</>
);
};
export default
withDependencyWatcher(FolderElem
ent);
src/complex-fields/folder/types.tsx
import { Folder, ValueData } from
"./../data/types";
export interface FolderElementProps
{
treeData: Folder;
valueData: ValueData;
}
src/complex-fields/folder/utils.tsx
import { JSX } from "react";
import DropdownElement from
"./dropdown";
import NumberElement from
"./number";

```

```

import FolderElement from
"../folder";
import ToggleElement from
"../toggle";
import TableElement from "../table";
import TextElement from "../text";
import { Childs, Dropdown, Folder,
NumberInput, Table, TextInput,
Toggle, ValueData } from
"../../data/types";
export const renderChildElements =
(child: Childs, valueData:
ValueData): JSX.Element => {
switch(child.type) {
case "folder":
return <FolderElement
treeData={child as Folder}
key={child.id}
valueData={valueData}/>;
case "text":
return <TextElement treeData={child
as TextInput} key={child.id}
valueData={valueData}/>;
case "number":
return <NumberElement
treeData={child as NumberInput}
key={child.id}
valueData={valueData}/>;
case "toggle":
return <ToggleElement
treeData={child as Toggle}
key={child.id}
valueData={valueData}/>;
case "dropdown":
return <DropdownElement
treeData={child as Dropdown}
key={child.id}
valueData={valueData}/>;
case "table":
return <TableElement
treeData={child as Table}
key={child.id}
valueData={valueData}/>;
default:
return <div key={child.id}></div>;
}
};
src/complex-fields/number/index.css
.number-wrapper {
width: 300px;
display: flex;
align-items: center;
margin: 5px 0;
}
.number-wrapper-for-table {
width: 150px;
}
.number-wrapper span {
white-space: nowrap;
}
.number-input {
width: 100%;
margin-left: 10px;
}
src/complex-fields/number/index.tsx
import { useState } from "react";
import withDependencyWatcher from
"../../HOC/withDependencyWatcher";
import { getValueFromValueData }
from "../../utils";
import { NumberElementProps }
from "../types";
import './index.css'
const NumberElement = ({treeData,
inTable, valueData, updateValue,
rowData}: NumberElementProps) =>
{

```

```

const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as number |
undefined;
const [localValue, setLocalValue] =
useState(valueFromValueData);
const handleChange = (e:
React.ChangeEvent<HTMLInputEle
ment >) => {
const newValue =
Number(e.target.value);
setLocalValue(newValue);
updateValue(treeData.id, newValue);
};
return (
<div className={`number-wrapper
${inTable ? 'number-wrapper-for-
table' : ''}`>
{!inTable &&
<span>{treeData.label}</span>
<input type="number"
value={localValue}
onChange={handleChange}
className="number-input"/>
</div>
);
};
export default
withDependencyWatcher(NumberEle
ment);
src/complex-fields/number/types.tsx
import { NumberInput, ValueData }
from "../../data/types";
import { UpdateValueFunction } from
 "../../utils/types";
export interface
NumberElementProps extends
UpdateValueFunction {
treeData: NumberInput;

```

```

inTable?: boolean;
valueData: ValueData;
rowData?: ValueData;
};
src/complex-fields/table/index.css
.table-wrapper {
height: 300px;
overflow: auto;
margin: 10px 0;
}
.table-wrapper span {
white-space: nowrap;
}
.table {
margin-top: 10px;
}
.table-wrapper table {
border-collapse: collapse;
border: 2px solid rgb(140 140 140);
font-family: sans-serif;
font-size: 0.8rem;
letter-spacing: 1px;
}
.table-wrapper thead {
background-color: rgb(228 240 245);
}
.table-wrapper th,td {
border: 1px solid rgb(160 160 160);
padding: 8px 10px;
}
.table-wrapper td:last-of-type {
text-align: center;
}
.table-wrapper tbody > tr:nth-of-
type(even) {
background-color: rgb(237 238 242);
}
src/complex-fields/table/index.tsx
import { JSX } from "react";

```

```

import withDependencyWatcher from
"../../HOC/withDependencyWatcher";
import { getValueFromValueData }
from "../../utils";
import DropdownElement from
"../../dropdown";
import NumberElement from
"../../number";
import ToggleElement from
"../../toggle";
import TextElement from "../../text";

import { ColumnChild, Dropdown,
NumberInput, TableValue, TextInput,
Toggle } from "../../data/types";
import { TableElementProps } from
"./types";
import './index.css'
const TableElement = ({treeData,
valueData}: TableElementProps) => {
const valueFromValueData =
getValueFromValueData(treeData,
valueData) as TableValue[];
const renderCell = (child:
ColumnChild, valueDataFromRow:
TableValue): JSX.Element => {
switch(child.type) {
case "text":
return <TextElement
rowData={valueDataFromRow}
valueData={valueData}
treeData={child as TextInput}
key={child.id} inTable={true}/>;
case "number":
return <NumberElement
rowData={valueDataFromRow}
valueData={valueData}
treeData={child as NumberInput}
key={child.id} inTable={true} />;

```

```

case "toggle":
return <ToggleElement
rowData={valueDataFromRow}
valueData={valueData}
treeData={child as Toggle}
key={child.id} inTable={true}/>;
case "dropdown":
return <DropdownElement
rowData={valueDataFromRow}
valueData={valueData}
treeData={child as Dropdown}
key={child.id} inTable={true}/>;
default:
return <div key={child.id}></div>;
}
};
return (
<div className="table-wrapper">
<span>{treeData.label}</span>
<table className="table">
<thead>
<tr>
{treeData.columns.map(column =>
<th style={{width: '150px'}}
key={column.id}>
{column.label}
</th>
)}
</tr>
</thead>
<tbody>
{valueFromValueData.map(row =>
<tr key={row.id as string}>
{treeData.columns.map(column =>
<td style={{width: '150px'}}
key={column.id}>
{renderCell(column, row)}
</td>
)}
)}

```

```

</tr>
)}
</tbody>
</table>
</div>
);
};
export default
withDependencyWatcher(TableElement);
src/complex-fields/table/types.tsx
import { Table, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";
export interface TableElementProps
extends UpdateValueFunction {
treeData: Table;
valueData: ValueData;
};
src/complex-fields/text/index.css
.text-wrapper {
width: 300px;
display: flex;
align-items: center;
margin: 5px 0;
}
.text-wrapper-for-table {
width: 150px;
}
.text-wrapper span {
white-space: nowrap;
}
.text-input {
width: 100%;
margin-left: 10px;
}
src/complex-fields/text/index.tsx
import { useState } from "react";

```

```

import withDependencyWatcher from
"../../HOC/withDependencyWatcher";
import { getValueFromValueData }
from "../../utils"
;
import { TextElementProps } from
"./types";
import './index.css'
const TextElement = ({treeData,
inTable, valueData, updateValue,
rowData}: TextElementProps) => {

const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as string |
undefined;
const [localValue, setLocalValue] =
useState(valueFromValueData);
const handleChange = (e:
React.ChangeEvent<HTMLInputEle
ment >) => {
const newValue = e.target.value;
setLocalValue(e.target.value);
updateValue(treeData.id, newValue);
};
return (
<div className={`text-wrapper
${inTable ? 'text-wrapper-for-table' :
''}`>
{!inTable &&
<span>{treeData.label}</span>
<input type="text"
value={localValue}
onChange={handleChange}
className="text-input" />
</div>
);
};

```

```

export default
withDependencyWatcher(TextElement);
src/complex-fields/text/types.tsx
import { TextInput, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";
export interface TextElementProps
extends UpdateValueFunction{
treeData: TextInput;
inTable?: boolean;
valueData: ValueData;
rowData?: ValueData;
};
src/complex-fields/toggle/index.css
.checkbox-element {
margin: 5px;
}
.checkbox-element span {
white-space: nowrap;
}
.toggle-wrapper {
display: flex;
align-items: center;
}
.toggle-wrapper-for-table {
justify-content: center;
}

src/complex-fields/toggle/index.tsx

import { useState } from "react";
import withDependencyWatcher from
"../../HOC/withDependencyWatcher";
import { getValueFromValueData }
from "../../utils";
import { ToggleProps } from
"./types";

```

```

import './index.css'
const ToggleElement = ({treeData,
inTable, valueData, updateValue,
rowData}: ToggleProps) => {
const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as boolean |
undefined;
const [localValue, setLocalValue] =
useState<boolean>(Boolean(valueFromValueData));
const handleChange = (e:
React.ChangeEvent<HTMLInputElement >) => {
const newValue = e.target.checked;
setLocalValue(e.target.checked);
updateValue(treeData.id, newValue);
};
return (
<div className={`toggle-wrapper
${inTable ? 'toggle-wrapper-for-table'
: ''}`>
{!inTable &&
<span>{treeData.label}</span>
<input type="checkbox"
checked={localValue}
onChange={handleChange}
className="checkbox-element"/>
</div>
);
};
export default
withDependencyWatcher(ToggleElement);
src/complex-fields/toggle/types.tsx
import { Toggle, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";

```

```

export interface ToggleProps extends
UpdateValueFunction {
treeData: Toggle;
inTable?: boolean;
valueData: ValueData;
rowData?: ValueData;
};
src/complex-tree/ComplexTree.tsx
import FolderElement from
"./complex-fields/folder";
import { ComplexTreeProps } from
"./types";
import "./index.css"

const ComplexTree = ({treeData,
valueData}: ComplexTreeProps) => {
return (
<div className="complex-tree">
<FolderElement treeData={treeData}
valueData={valueData}/>
</div>
);
};
export default ComplexTree;
src/complex-tree/index.css
.complex-tree {
width: 100%;
height: 100%;
};
src/complex-tree/index.tsx
import { useEffect } from "react";
import { treeData as
treeDataFromBE, valueData as
valueDataFromBE } from
"./data/data";
import { useAppDispatch,
useAppSelector } from
"./app/hooks";

```

```

import { setInitialStructure } from
"./app/structureSlice";
import ComplexTree from
"./ComplexTree";
const DataWrapperForComplexTree
= () => {
const tree = useAppSelector(state =>
state.structure.tree);
const initialValues =
useAppSelector(state =>
state.structure.initialValues);
const dispatch = useAppDispatch();
useEffect(() => {
if (valueDataFromBE &&
treeDataFromBE) {
dispatch(
setInitialStructure({
tree: treeDataFromBE,
values: valueDataFromBE,
}))
);
}, [valueDataFromBE,
treeDataFromBE]);
return ( tree && initialValues &&
<ComplexTree treeData={tree}
valueData={initialValues}/> );
};
export default
DataWrapperForComplexTree;
src/complex-tree/types.tsx
import { Childs, ValueData } from
"./data/types";
export interface ComplexTreeProps {
treeData: Childs;
valueData: ValueData
}

src/data/data.tsx

```

```

import {
  createDropdown,
  createFolder,
  createNumber,
  createTable,
  createText,
  createToggle,
  generateBooleanDependency,
  generateDropDownDependency,
} from "./generator";
import { ukraineRegionsTable } from
"./state-data";
import { Folder, ValueData } from
"./types";
import { getUsers } from "./user-
data";
export const treeData: Folder =
createFolder('Налаштування',[
  createText("Ім'я адміна", "", "name"),
  createText("Фамілія адміна ",
    "", "second-name",
    generateBooleanDependency('name',
      'hide', '1')),
  createNumber("Вік адміна",
    undefined, "age"),
  createToggle("Активувати додаткові
    налаштування", true, "additional-
    toggle"),
  createFolder('Додаткові
    налаштування', [
    createTable("Області", [
      createToggle("Активна область",
        true, "tableState"),
      createText("Назва", "", "tableName"),
      createNumber("Населення (млн.)",
        0, "tablePeopleNumber"),
      createDropdown("Локація", [
        {label: "Північ", value: 'Північ'},
        {label: "Південь", value: 'Південь'},

```

```

        {label: "Захід", value: 'Захід'},
        {label: "Схід", value: 'Схід'},
        {label: "Центр", value: 'Центр'}},
        ], "", "tableLocation"),
      ], 'regions'),
    createFolder('Налаштування
      доступу до вибірки планет', [
      createToggle("Венера", true, 'venera'),
      createToggle("Земля", true, 'zemlya'),
      createToggle("Нептун", true,
        'neptun'),
      createToggle("Сатурн", false,
        'saturn'),
      createToggle("Уран", false, 'uran'),
      createToggle("Плутон", false,
        'pluton'),
      ], "planet-folder"),
    createFolder('Налаштування
      доступу до вибірки статі', [
      createToggle("Чоловік", true,
        "male"),
      createToggle("Жінка", true, 'female'),
      createToggle("Інше", false, 'other'),
      ], 'human'),
    ], 'additional-folder',
    generateBooleanDependency('additional-
      toggle', 'hide', false)
  ),
  createTable("Користувачі", [
    createToggle("Активний
      користувач", true, "table-toggle"),
    createText("Ім'я", "", "table-text"),
    createNumber("Вік", 1, "table-
      number"),
    createDropdown("Стать", [
      {label: "Чоловік", value: 'Чоловік'},
      {label: "Жінка", value: 'Жінка'},
      {label: "Інше", value: 'Інше'},
      ], "", "table-dropdown-stat"),

```

```

createDropdown("Місце
знаходження", [
  {label: "Венера", value: 'Венера',
optionDependency:
generateBooleanDependency('venera',
'hide', false)},
  {label: "Земля", value: 'Земля',
optionDependency:
generateBooleanDependency('zemlya'
, 'hide', false)},
  {label: "Нептун", value: 'Нептун',
optionDependency:
generateBooleanDependency('neptun',
'hide', false)},
  {label: "Сатурн", value: 'Сатурн',
optionDependency:
generateBooleanDependency('saturn',
'hide', false)},
  {label: "Уран", value: 'Уран',
optionDependency:
generateBooleanDependency('uran',
'hide', false)},
  {label: "Плутон", value: 'Плутон',
optionDependency:
generateBooleanDependency('pluton',
'hide', false)},
], "Земля", "table-dropdown-star",
generateDropDownDependency(['ven
era', 'zemlya', 'neptun', 'saturn', 'uran',
'pluton'])),
createDropdown("Машина", [
  {label: "BMW", value: 'BMW'},
  {label: "Audi", value: 'Audi'},
  {label: "Ferrari", value: 'Ferrari'},
  {label: "Aston Martin", value: 'Aston
Martin'},
], "", "table-dropdown-car"),
createDropdown("Діти", [
  {label: "1", value: '1'},

```

```

{label: "2", value: '2'},
{label: "3", value: '3'},
{label: "4", value: '4'},
{label: "5", value: '5'},
{label: "6", value: '6'},
], "", "table-dropdown-star-childrens"),
], 'users'),
]);
export const valueData: ValueData =
{
  'name': 'Олег',
  'second-name': 'Алмазов',
  'age': 18,
  'additional-toggle': true,
  'planet-folder': "",
  'country': "",
  'country-code': "",
  'people': "",
  'venera': true,
  'zemlya': true,
  'neptun': true,
  'saturn': false,
  'uran': false,
  'pluton': false,
  'human': "",
  'male': true,
  'female': true,
  'other': false,
  'additional-folder': "",
  'users': getUsers(10),
  'regions': ukraineRegionsTable
};
console.log(valueData, treeData)
src/data/generator.tsx
import { nanoid } from "nanoid"
import {
  BasicField,
  Childs,
  Folder,

```

```

TextInput,
NumberInput,
Dropdown,
DropdownOption,
FieldType,
Toggle,
Table,
ColumnChild
} from "./types"
import { Dependency, Result } from
"./types-dependency";
export const createBasicField = (
label: string,
type: FieldType,
id?: string,
dependency?: Dependency[]
): BasicField => {
return {
id: id || nanoid(),
type,
label,
dependency
};
};
export const createFolder = (
label: string,
childs: Childs[],
id?: string,
dependency?: Dependency[]
): Folder => {
return {
...createBasicField(label, "folder", id,
dependency),
childs,
};
};
export const createText = (
label: string,
defaultValue: string,

```

```

id?: string,
dependency?: Dependency[]
): TextInput => {
return {
...createBasicField(label, "text", id,
dependency),
defaultValue,
};
};
export const createNumber = (
label: string,
defaultValue?: number,
id?: string,
dependency?: Dependency[]
): NumberInput => {
return {
...createBasicField(label, "number",
id, dependency),
defaultValue,
};
};
export const createToggle = (
label: string,
defaultValue: boolean,
id?: string,
dependency?: Dependency[]
): Toggle => {
return {
...createBasicField(label, "toggle", id,
dependency),
defaultValue,
};
};
export const createDropdownOption =
(
label: string,
value: string,
id?: string,
dependency?: Dependency[]

```

```

): DropdownOption => {
  return {
    ...createBasicField(label, "option", id,
      dependency),
    value,
  };
};

export const createDropdown = (
  label: string,
  options: {
    label: string,
    value: string,
    optionDependency?: Dependency[]
  }[],
  defaultValue?: string,
  id?: string,
  dependency?: Dependency[]
): Dropdown => {
  return {
    ...createBasicField(label,
      "dropdown", id, dependency),
    options: options.map(option =>
      createDropdownOption(
        option.label,
        option.value,
        undefined,
        option.optionDependency
      )),
    defaultValue
  };
};

export const createTable = (
  label: string,
  columns: ColumnChild[],
  id?: string,
  dependency?: Dependency[]
): Table => {
  return {

```

```

    ...createBasicField(label, "table", id,
      dependency),
    columns,
  };
};

export const generateMockData =
(count: number) => {
  const names = [
    "Олег", "Іван", "Анна", "Марія",
    "Олена", "Дмитро", "Андрій",
    "Ірина",
    "Максим", "Катерина", "Сергій",
    "Наталя", "Богдан", "Юлія",
    "Роман",
    "Вікторія", "Павло", "Оксана",
    "Михайло", "Софія", "Артур",
    "Дарина",
    "Ярослав", "Аліна", "Євген",
    "Поліна", "Владислав", "Тетяна"
  ];
  const genders = ["Чоловік",
    "Жінка"];
  const planets = ["Земля", "Нептун",
    "Венера", "Сатурн", "Уран",
    "Плутон"];
  const cars = ['BMW', 'Audi', 'Ferrari',
    'Aston Martin'];
  const childrens = ["1", "2", "3", "4",
    "5", "6"];
  const randomFrom = (arr: any) =>
    arr[Math.floor(Math.random() *
      arr.length)];
  const randomNumber = (min: any,
    max: any) =>
    Math.floor(Math.random() * (max -
      min + 1)) + min;
  return Array.from({ length: count },
    (_, index) => ({
      id: index + 1,

```

```

"table-toggle": Math.random() < 0.5,
"table-text": randomFrom(names),
"table-number": randomNumber(18,
100),
"table-dropdown-stat":
randomFrom(genders),
"table-dropdown-star":
randomFrom(planets),
"table-dropdown-car":
randomFrom(cars),
"table-dropdown-star-childrens":
randomFrom(childrens),
));
};
export const
generateBooleanDependency =
(watchedId: string, result: Result,
value: boolean): Dependency[] => {
return [
{
watchedId,
rulse: [{operator: '===', result, value}]
}
]
};
export const
generateDropDownDependency =
(watchedIds: string[]): Dependency[]
=> {
return watchedIds.map(id => ({
watchedId: id,
rulse: [{operator: '===', result:
"visible", value: true}]
})));
};
src/data/state-data.tsx
export const ukraineRegionsTable = [
{ id: 'region-1', tableState: true,
tableName: "Вінницька",

```

```

tablePeopleNumber: 1.53,
tableLocation: "Центр" },
{ id: 'region-2', tableState: false,
tableName: "Волинська",
tablePeopleNumber: 1.02,
tableLocation: "Захід" },
{ id: 'region-3', tableState: true,
tableName: "Дніпропетровська",
tablePeopleNumber: 3.15,
tableLocation: "Схід" },
{ id: 'region-4', tableState: false,
tableName: "Донецька",
tablePeopleNumber: 4.10,
tableLocation: "Схід" },
{ id: 'region-5', tableState: true,
tableName: "Житомирська",
tablePeopleNumber: 1.21,
tableLocation: "Північ" },
{ id: 'region-6', tableState: false,
tableName: "Закарпатська",
tablePeopleNumber: 1.25,
tableLocation: "Захід" },
{ id: 'region-7', tableState: true,
tableName: "Запорізька",
tablePeopleNumber: 1.64,
tableLocation: "Південь" },
{ id: 'region-8', tableState: false,
tableName: "Івано-
Франківська",tablePeopleNumber:
1.36, tableLocation: "Захід" },
{ id: 'region-9', tableState: true,
tableName: "Київська",
tablePeopleNumber: 1.79,
tableLocation: "Північ" },
{ id: 'region-10', tableState: false,
tableName: "Кіровоградська",
tablePeopleNumber: 0.90,
tableLocation: "Центр" },

```

```

{ id: 'region-11', tableState: true,
  tableName: "Луганська",
  tablePeopleNumber: 2.12,
  tableLocation: "Схід" },
{ id: 'region-12', tableState: false,
  tableName: "Львівська",
  tablePeopleNumber: 2.48,
  tableLocation: "Захід" },
{ id: 'region-13', tableState: true,
  tableName: "Миколаївська",
  tablePeopleNumber: 1.09,
  tableLocation: "Південь" },
{ id: 'region-14', tableState: false,
  tableName: "Одеська",
  tablePeopleNumber: 2.35,
  tableLocation: "Південь" },
{ id: 'region-15', tableState: true,
  tableName: "Полтавська",
  tablePeopleNumber: 1.38,
  tableLocation: "Центр" },
{ id: 'region-16', tableState: false,
  tableName: "Рівненська",
  tablePeopleNumber: 1.14,
  tableLocation: "Захід" },
{ id: 'region-17', tableState: true,
  tableName: "Сумська",
  tablePeopleNumber: 1.03,
  tableLocation: "Північ" },
{ id: 'region-18', tableState: false,
  tableName: "Тернопільська",
  tablePeopleNumber: 1.02,
  tableLocation: "Захід" },
{ id: 'region-19', tableState: true,
  tableName: "Харківська",
  tablePeopleNumber: 2.60,
  tableLocation: "Схід" },
{ id: 'region-20', tableState: false,
  tableName: "Херсонська",

```

```

  tablePeopleNumber: 1.00,
  tableLocation: "Південь" },
{ id: 'region-21', tableState: true,
  tableName: "Хмельницька",
  tablePeopleNumber: 1.24,
  tableLocation: "Захід" },
{ id: 'region-22', tableState: false,
  tableName: "Черкаська",
  tablePeopleNumber: 1.18,
  tableLocation: "Центр" },
{ id: 'region-23', tableState: true,
  tableName: "Чернівецька",
  tablePeopleNumber: 0.90,
  tableLocation: "Захід" },
{ id: 'region-24', tableState: false,
  tableName: "Чернігівська",
  tablePeopleNumber: 0.98,
  tableLocation: "Північ" },
{ id: 'region-25', tableState: true,
  tableName: "Автономна Республіка
Крим",tablePeopleNumber: 1.91,
  tableLocation: "Південь" }
];

```

```

src/data/types-dependency.ts
type Operator = '>' | '<' | '===' | '>=' |
'<=';
export type Result = 'hide' | 'visible';
export interface Rule {
  operator: Operator;
  value: string | number | boolean;
  result: Result;
};
export interface Dependency {
  watchedId: string;
  rule: Rule[];
};
src/data/types.ts

```

```

import { Dependency } from "../types-
dependency";
export type FieldType = 'text' |
'number' | 'toggle' | 'dropdown' | 'table'
| 'folder' | 'option';
export type Childs = Folder |
TextInput | NumberInput | Dropdown
| Toggle | Table;
export type ColumnChild = TextInput
| NumberInput | Dropdown | Toggle;
export interface Folder extends
BasicField {
  childs: Childs[];
};
export interface BasicField {
  id: string;
  dependency?: Dependency[];
  type: FieldType;
  label: string;
}
export interface TextInput extends
BasicField {
  defaultValue?: string;
};
export interface NumberInput extends
BasicField {
  defaultValue?: number;
};
export interface Toggle extends
BasicField {
  defaultValue?: boolean;
};
export interface DropdownOption
extends BasicField {
  value: string;
};
export interface Dropdown extends
BasicField {
  defaultValue?: string;

```

```

options: DropdownOption[];
};
export type TableValue = {
[key: string]: string | number | boolean
;
};
export interface Table extends
BasicField {
  columns: ColumnChild[];
};

export type ValueData = {
[key: string]: string | number | boolean
| TableValue[];
};
src/data/user-data.tsx
const users = [
{
  "id": 999,
  "table-toggle": false,
  "table-text": "Богдан",
  "table-number": 48,
  "table-dropdown-stat": "Жінка",
  "table-dropdown-star": "Плутон",
  "table-dropdown-car": "Audi",
  "table-dropdown-star-childrens": "5"
},
{
  "id": 1000,
  "table-toggle": true,
  "table-text": "Юлія",
  "table-number": 74,
  "table-dropdown-stat": "Жінка",
  "table-dropdown-star": "Сатурн",
  "table-dropdown-car": "Audi",
  "table-dropdown-star-childrens": "5"
}
];

```

```

export const getUsers = (n: number)
=> {
  return users.slice(0, n);
}
src/fields/dropdown/index.css
.dropdown-wrapper {
  width: 300px;
  display: flex;
  align-items: center;
  margin: 5px 0;
}
.dropdown-wrapper-for-table {
  width: 150px;
}
.dropdown-wrapper span {
  white-space: nowrap;
}
.dropdown-input {
  width: 100%;
  margin-left: 10px;
}
src/fields/dropdown/index.tsx
import { useState } from "react";
import { getValueFromValueData }
from "../utils";
import { OptionElement } from
"./option";

import { DropdownElementProps }
from "./types";
import './index.css'
export const DropdownElement =
({treeData, inTable, valueData,
updateValue, rowData}:
DropdownElementProps) => {
  const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as string |
undefined;

```

```

const [localValue, setLocalValue] =
useState(valueFromValueData);
const handleChange = (e:
React.ChangeEvent<HTMLSelectEle
ment>) => {
  const newValue = e.target.value;
  setLocalValue(e.target.value);
  updateValue(treeData.id, newValue);
};
return (
<div className={`dropdown-
wrapper ${inTable ? 'dropdown-
wrapper-for-table' : ''}`>
  {!inTable &&
    <span>{treeData.label}</span>
    <select className="dropdown-input"
value={localValue}
onChange={handleChange}>
      <option value="">Виберіть
значення</option>
      {treeData.options.map(option =>
        <OptionElement treeData={option}
valueData={valueData}
key={option.id}/>
      )}
    </select>
  </div>
);
};
src/fields/dropdown/option/index.css
.dropdown-wrapper {
  width: 300px;
  display: flex;
  align-items: center;
  margin: 5px 0;
}
.dropdown-wrapper-for-table {
  width: 150px;
}

```

```

.dropdown-wrapper span {
  white-space: nowrap;
}
.dropdown-input {
  width: 100%;
  margin-left: 10px;
}
src/fields/dropdown/option/index.tsx
import { isElementVisible } from
"../../utils/dependencyChecker";
import { OptionElementProps } from
"./types";
export const OptionElement =
({treeData, valueData}:
OptionElementProps) => {
  const optionVisible =
isElementVisible(treeData,
valueData);
  return (
    optionVisible &&
    <option value={treeData.value}
    key={treeData.id}>
    {treeData.label}
    </option>
  );
};
src/fields/dropdown/option/types.tsx
import { DropdownOption,
ValueData } from "../../data/types";
export interface OptionElementProps {
  valueData: ValueData;
  treeData: DropdownOption;
};
src/fields/dropdown/types.tsx
import { Dropdown, ValueData }
from "../../data/types";
export interface
DropdownElementProps {

```

```

treeData: Dropdown;
valueData: ValueData;
rowData?: ValueData;
inTable?: boolean;
updateValue: (id: string, newValue:
string | number | boolean) => void;
};
src/fields/folder/index.css
.folder {
  display: flex;
  flex-direction: column;
  border: 1px solid black;
  padding: 10px;
  box-sizing: border-box;
  margin-bottom: 10px;
}
.folder-label {
  margin: 10px 0;
  font-size: larger;
  display: flex;
  flex-direction: column;
}
src/fields/folder/index.tsx
import { JSX } from "react";
import { isElementVisible } from
"../../utils/dependencyChecker";
import { DropdownElement } from
"./dropdown";
import { ToggleElement } from
"./toggle";
import { NumberElement } from
"./number";
import { TableElement } from
"./table";
import { TextElement } from "../text";
import { Childs, Folder, TextInput,
NumberInput, Toggle, Dropdown,
Table } from "../../data/types";

```

```

import { FolderElementProps } from
"./types";
import './index.css'
export const FolderElement =
({treeData, valueData, updateValue}:
FolderElementProps) => {
  const folderVisible =
isElementVisible(treeData,
valueData);
  const renderChildElements = (child:
Childs): JSX.Element => {
    switch(child.type) {
      case "folder":
        return <FolderElement
treeData={child as Folder}
key={child.id}
valueData={valueData}
updateValue={updateValue}/>;
      case "text":
        return <TextElement treeData={child
as TextInput} key={child.id}
valueData={valueData}
updateValue={updateValue}/>;
      case "number":
        return <NumberElement
treeData={child as NumberInput}
key={child.id}
valueData={valueData}
updateValue={updateValue}/>;
      case "toggle":
        return <ToggleElement
treeData={child as Toggle}
key={child.id}
valueData={valueData}
updateValue={updateValue}/>;
      case "dropdown":
        return <DropdownElement
treeData={child as Dropdown}
key={child.id}
valueData={valueData}
updateValue={updateValue}/>;
      case "table":
        return <TableElement
treeData={child as Table}
key={child.id}
valueData={valueData}
updateValue={updateValue}/>;
      default:
        return <div key={child.id}></div>;
    }
  };
  return (
    <>
      {folderVisible &&
        <div className="folder">
          <div className="folder-
label">{`Папка:
${treeData.label}`}</div>
          {treeData.chlds.map(child =>
renderChildElements(child))}
        </div>
      }
    </>
  );
};
src/fields/folder/types.tsx
import { Folder, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";
export interface FolderElementProps
extends UpdateValueFunction {
  treeData: Folder;
  valueData: ValueData;
}
src/fields/number/index.css
.number-wrapper {
  width: 300px;

```

```

display: flex;
align-items: center;
margin: 5px 0;
}
.number-wrapper-for-table {
width: 150px;
}
.number-wrapper span {
white-space: nowrap;
}
.number-input {
width: 100%;
margin-left: 10px;
}
src/fields/number/index.tsx
import { useState } from "react";
import { getValueFromValueData }
from "../utils";
import { NumberElementProps }
from "./types";
import './index.css'
export const NumberElement =
({treeData, inTable, valueData,
updateValue, rowData}:
NumberElementProps) => {
const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as number |
undefined;
const [localValue, setLocalValue] =
useState(valueFromValueData);
const handleChange = (e:
React.ChangeEvent<HTMLInputEle
ment >) => {
const newValue =
Number(e.target.value);
setLocalValue(newValue);
updateValue(treeData.id, newValue);
};

```

```

return (
<div className={`number-wrapper
${inTable ? 'number-wrapper-for-
table' : ''}`>
{!inTable &&
<span>{treeData.label}</span>}
<input type="number"
value={localValue}
onChange={handleChange}
className="number-input"/>
</div>
);
};
src/fields/number/types.tsx
import { NumberInput, ValueData }
from "../data/types";
import { UpdateValueFunction } from
"../utils/types";
export interface
NumberElementProps extends
UpdateValueFunction {
treeData: NumberInput;
inTable?: boolean;
valueData: ValueData;
rowData?: ValueData;
};
src/fields/table/index.css
.table-wrapper {
height: 300px;
overflow: auto;
margin: 10px 0;
}
.table-wrapper span {
white-space: nowrap;
}
.table {
margin-top: 10px;
}
.table-wrapper table {

```

```

border-collapse: collapse;
border: 2px solid rgb(140 140 140);
font-family: sans-serif;
font-size: 0.8rem;
letter-spacing: 1px;
}
.table-wrapper thead {
background-color: rgb(228 240 245);
}
.table-wrapper th,td {
border: 1px solid rgb(160 160 160);
padding: 8px 10px;
}
.table-wrapper td:last-of-type {
text-align: center;
}
.table-wrapper tbody > tr:nth-of-
type(even) {
background-color: rgb(237 238 242);
}
src/fields/table/index.tsx
import { JSX } from "react";
import { ColumnChild, Dropdown,
NumberInput, TableValue, TextInput,
Toggle } from "../data/types";
import { TableElementProps } from
"./types";
import './index.css'
import { TextElement } from "../text";
import { NumberElement } from
"./number";
import { ToggleElement } from
"./toggle";
import { DropdownElement } from
"./dropdown";
import { getValueFromValueData }
from "../utils";
import { ComplexId } from
"../utils/types";

```

```

export const TableElement =
({treeData, valueData, updateValue}:
TableElementProps) => {
const valueFromValueData =
getValueFromValueData(treeData,
valueData) as TableValue[];
const renderCell = (child:
ColumnChild, valueDataFromRow:
TableValue): JSX.Element => {
const updateValueForCell = (id:
string | ComplexId, newValue: string |
number | boolean) => {
const tableindex =
(valueData[treeData.id] as
TableValue[]).findIndex(data =>
data.id === valueDataFromRow.id);
updateValue({tableId: treeData.id,
elementId: id as string, rowIndex:
tableindex }, newValue);
};
switch(child.type) {
case "text":
return <TextElement
rowData={valueDataFromRow}
valueData={valueData}
treeData={child as TextInput}
key={child.id} inTable={true}
updateValue={updateValueForCell}/>
>;
case "number":
return <NumberElement
rowData={valueDataFromRow}
valueData={valueData}
treeData={child as NumberInput}
key={child.id} inTable={true}
updateValue={updateValueForCell}/>
>;
case "toggle":

```

```

return <ToggleElement
  rowData={valueDataFromRow}
  valueData={valueData}
  treeData={child as Toggle}
  key={child.id} inTable={true}
  updateValue={updateValueForCell}/
>;
case "dropdown":
return <DropdownElement
  rowData={valueDataFromRow}
  valueData={valueData}
  treeData={child as Dropdown}
  key={child.id} inTable={true}
  updateValue={updateValueForCell}/
>;
default:
return <div key={child.id}></div>;
}
};
return (
  <div className="table-wrapper">
    <span>{treeData.label}</span>
    <table className="table">
      <thead>
        <tr>
          {treeData.columns.map(column =>
            <th style={{ width: '150px' }}
              key={column.id}>
                {column.label}
            </th>
          )}
        </tr>
      </thead>
      <tbody>
        {valueFromValueData.map(row =>
          <tr key={row.id as string}>
            {treeData.columns.map(column =>
              <td style={{ width: '150px' }}
                key={column.id}>
                  {renderCell(column, row)}
              </td>
            )}
          </tr>
        )}
      </tbody>
    </table>
  </div>
);
};
src/fields/table/types.tsx
import { Table, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";
export interface TableElementProps
extends UpdateValueFunction {
  treeData: Table;
  valueData: ValueData;
};
src/fields/text/index.css
.text-wrapper {
  width: 300px;
  display: flex;
  align-items: center;
  margin: 5px 0;
}
.text-wrapper-for-table {
  width: 150px;
}
.text-wrapper span {
  white-space: nowrap;
}
.text-input {
  width: 100%;
  margin-left: 10px;
}
src/fields/text/index.tsx
import { useState } from "react";

```

```

import { getValueFromValueData }
from "../../utils"
;
import { TextElementProps } from
"./types";
import './index.css'
import { isElementVisible } from
"../../utils/dependencyChecker";
export const TextElement =
({treeData, inTable, valueData,
updateValue, rowData}:
TextElementProps) => {
const valueFromValueData =
getValueFromValueData(treeData,
rowData || valueData) as string |
undefined;
const [localValue, setLocalValue] =
useState(valueFromValueData);
const handleChange = (e:
React.ChangeEvent<HTMLInputEle
ment >) => {
const newValue = e.target.value;
setLocalValue(e.target.value);
updateValue(treeData.id, newValue);
};
const disabled =
isElementVisible(treeData,
valueData);
return (
<div className={`text-wrapper
${inTable ? 'text-wrapper-for-table' :
''}`>
{!inTable &&
<span>{treeData.label}</span>
<input type="text"
value={localValue}
onChange={handleChange}
className="text-input"
disabled={!disabled}/>

```

```

</div>
);
};
src/fields/text/types.tsx
import { TextInput, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";
export interface TextElementProps
extends UpdateValueFunction{
treeData: TextInput;
inTable?: boolean;
valueData: ValueData;
rowData?: ValueData;
};
src/fields/toggle/index.css
.checkbox-element {
margin: 5px;
}
.checkbox-element span {
white-space: nowrap;
}
.toggle-wrapper {
display: flex;
align-items: center;
}
.toggle-wrapper-for-table {
justify-content: center;
}
src/fields/toggle/index.tsx
import { useState } from "react";
import { getValueFromValueData }
from "../../utils";
import { ToggleProps } from
"./types";
import './index.css'
export const ToggleElement =
({treeData, inTable, valueData,

```

```

updateValue, rowData}:
ToggleProps) => {
  const valueFromValueData =
    getValueFromValueData(treeData,
    rowData || valueData) as boolean |
    undefined;
  const [localValue, setLocalValue] =
    useState<boolean>(Boolean(valueFromValueData));
  const handleChange = (e:
    React.ChangeEvent<HTMLInputElement>) => {
    const newValue = e.target.checked;
    setLocalValue(e.target.checked);
    updateValue(treeData.id, newValue);
  };
  return (
    <div className={`toggle-wrapper
    ${inTable ? 'toggle-wrapper-for-table'
    : ''}`}>
    {!inTable &&
    <span>{treeData.label}</span>
    <input type="checkbox"
    checked={localValue}
    onChange={handleChange}
    className="checkbox-element"/>
    </div>
    );
  };
}
src/fields/toggle/types.tsx
import { Toggle, ValueData } from
"../../data/types";
import { UpdateValueFunction } from
"../../utils/types";
export interface ToggleProps extends
UpdateValueFunction {
  treeData: Toggle;
  inTable?: boolean;
  valueData: ValueData;

```

```

  rowData?: ValueData;
};
src/index.css
body {
  margin: 0;
  font-family:
    -apple-system, BlinkMacSystemFont,
    "Segoe UI", "Roboto", "Oxygen",
    "Ubuntu",
    "Cantarell", "Fira Sans", "Droid
    Sans", "Helvetica Neue", sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
code {
  font-family:
    source-code-pro, Menlo, Monaco,
    Consolas, "Courier New",
    monospace;
}
html, body, #root {
  width: 100%;
  height: 100%;
  padding: 0;
  margin: 0;
}
src/main.tsx
import { createRoot } from "react-
dom/client";
import { Provider } from "react-
redux";
import { store } from "../app/store";
import { App } from "../App";
import "../index.css";
const container =
  document.getElementById("root")
if (container) {
  const root = createRoot(container)
  root.render(

```

```

<Provider store={store}>
<App />
</Provider>
)
}
src/simple-tree/index.css
.simple-tree {
width: 100%;
height: 100%;
};
src/simple-tree/index.tsx
import { useState } from "react";
import { treeData as
treeDataFromBE, valueData as
valueDataFromBE } from
"./data/data";
import { FolderElement } from
"./fields/folder";

import { ComplexId } from
"./utils/types";
import "./index.css"
import { ValueData } from
"./data/types";
export const SimpleTree = () => {
const [valueData, setValueData] =
useState(valueDataFromBE);
const updateValue = (id: string |
ComplexId, newValue: string |
number | boolean) => {
const newValueData =
Object.assign({}, valueData);
if (typeof id === 'string') {
newValueData[id] = newValue;
} else {
(newValueData[id.tableId] as
ValueData[])[id.rowIndex][id.element
Id] = newValue
}
}

```

```

setValueData(newValueData);
};
return (
<div className="simple-tree">
<FolderElement
treeData={treeDataFromBE}
valueData={valueData}
updateValue={updateValue}/>
</div>
);
};
src/utils/dependencyChecker.tsx
import { Childs, DropdownOption,
ValueData } from "../data/types";
import { Result, Rule } from
"./data/types-dependency";
type AcceptedValues = string |
number | boolean | undefined;
export const isElementVisible =
(element: Omit<Childs |
DropdownOption, 'Table'>,
valueData: ValueData): boolean => {
if (element.dependency &&
element.dependency[0]) {
const { watchedId, rule } =
element.dependency[0];
const watchedValue =
valueData[watchedId] as
AcceptedValues;
const arrayOfresult = rule.map(rule
=> checkOneRules(rule,
watchedValue));
return arrayOfresult.every(res => res
=== 'visible');
}
return true;
};

```

```

const checkOneRules = (rule: Rule,
watchedValue: AcceptedValues):
Result => {
  switch(rule.operator) {
    case "<":
      if (typeof watchedValue === 'number'
        && typeof rule.value === 'number')
        {
          return
            generateVisibility(watchedValue <
              rule.value, rule.result);
        }
      return 'visible'
    case ">":
      if (typeof watchedValue === 'number'
        && typeof rule.value === 'number')
        {
          return
            generateVisibility(watchedValue >
              rule.value, rule.result);
        }
      return 'visible'
    case "<=":
      if (typeof watchedValue === 'number'
        && typeof rule.value === 'number')
        {
          return
            generateVisibility(watchedValue <=
              rule.value, rule.result);
        }
      return 'visible';
    case "===":
      return
        generateVisibility(watchedValue ===
          rule.value, rule.result);
    case ">=":
      if (typeof watchedValue === 'number'
        && typeof rule.value === 'number')
        {

```

```

      return
        generateVisibility(watchedValue >=
          rule.value, rule.result);
        }
      return 'visible';
    default:
      return 'visible';
        }
      };
    const generateVisibility =
      (conditionMet: boolean, action:
        Result): Result => {
      const expectedResult = action;
      const unexpectedResult = action ===
        'hide' ? 'visible' : 'hide';
      return conditionMet ? expectedResult
        : unexpectedResult;
      };
    src/utils/index.tsx
    import { Childs,
      TableValue, ValueData } from
      "../data/types";
    import { isTextInput, isDropdown,
      isNumberInput, isTable, isToggle }
      from "../type-checker";
    export const getValueFromValueData
      = (child: Childs, valueData:
        ValueData): string | number | boolean
        | TableValue[] | undefined => {
      const id = child.id;
      if (isTextInput(child)) return
        valueData[id] || child.defaultValue;
      if (isDropdown(child)) return
        valueData[id] || child.defaultValue;
      if (isNumberInput(child)) return
        valueData[id] || child.defaultValue;
      if (isTable(child)) return
        valueData[id];

```

```

if (isToggle(child)) return
valueData[id] !== undefined ?
valueData[id] : child.defaultValue;
};
src/utils/type-checker.tsx
import { Childs, Dropdown, Folder,
NumberInput, Table, TextInput,
Toggle } from "../data/types";
export const isTextInput = (f:
Childs): f is TextInput => f.type ===
'text';
export const isNumberInput = (f:
Childs): f is NumberInput => f.type
=== 'number';
export const isToggle = (f: Childs):
f is Toggle => f.type === 'toggle';
export const isDropdown = (f:
Childs): f is Dropdown => f.type
=== 'dropdown';

```

```

export const isTable = (f: Childs):
f is Table => f.type === 'table';
export const isFolder = (f: Childs):
f is Folder => f.type === 'folder';
src/utils/types.tsx
export interface ComplexId {
tableId: string;
rowIndex: number;
elementId: string;
};
export interface UpdateValueFunction
{
updateValue: (
id: string | ComplexId,
newValue: string | number | boolean
) => void;
};

```

Лістинг програмного коду

засобами нативного JavaScript для примітивного методу

index.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport"
content="width=device-width, initial-
scale=1.0" />
  <title>Only JS (HOC)</title>
  <link rel="stylesheet"
href="styles.css" />
</head>
<body>
  <div id="root"></div>
  <script type="module"
src="init.js"></script>
</body>
</html>
```

styles.css

```
*, *::before, *::after { box-sizing:
border-box; }
body {
  margin: 0;
  padding: 0;
  font-family: -apple-system,
BlinkMacSystemFont, "Segoe UI",
"Roboto",
  "Oxygen", "Ubuntu", "Cantarell",
"Fira Sans", "Droid Sans",
  "Helvetica Neue", sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
html, body, #root {
  width: 100%;
```

```
height: 100%;
padding: 0;
margin: 0;
}
.folder {
  display: flex;
  flex-direction: column;
  border: 1px solid black;
  border-width: 1px 0 0 1px;
  padding: 10px;
  margin-bottom: 10px;
  width: 1250px;
}
.folder-label {
  font-size: larger;
  display: flex;
  flex-direction: column;
  margin-bottom: 6px;
}
.checkbox-element { margin: 5px; }
.toggle-wrapper {
  display: flex;
  align-items: center;
}
.toggle-wrapper-for-table { justify-
content: center; }
.toggle-wrapper span { white-space:
nowrap; }
.text-wrapper {
  width: 300px;
  display: flex;
  align-items: center;
  margin: 5px 0;
}
```

```

.text-wrapper-for-table { width:
150px; }
.text-wrapper span { white-space:
nowrap; }
.text-input { width: 100%; margin-
left: 10px; }
.number-wrapper {
width: 300px;
display: flex;
align-items: center;
margin: 5px 0;
}
.number-wrapper-for-table { width:
150px; }
.number-wrapper span { white-space:
nowrap; }
.number-input { width: 100%;
margin-left: 10px; }
.dropdown-wrapper {
width: 300px;
display: flex;
align-items: center;
margin: 5px 0;
}
.dropdown-wrapper-for-table { width:
150px; }
.dropdown-wrapper span { white-
space: nowrap; }
.dropdown-input { width: 100%;
margin-left: 10px; }
.table-wrapper {
height: 600px;
overflow: auto;
margin: 10px 0;
}
.table-wrapper > .table-label {
white-space: nowrap;
display: block;
}

```

```

.data-table {
margin-top: 10px;
border-collapse: collapse;
border: 2px solid rgb(140 140 140);
font-family: sans-serif;
font-size: 0.8rem;
letter-spacing: 1px;
}
.data-table thead { background-color:
rgb(228 240 245); }
.data-table th,
.data-table td {
border: 1px solid rgb(160 160 160);
padding: 8px 10px;
width: 150px;
}
.data-table td:last-of-type { text-align:
center; }
.data-table tbody > tr:nth-of-
type(even) {
background-color: rgb(237 238 242);
}

```

types.js

```

export const TYPES = {
  FOLDER: 'folder',
  TOGGLE: 'toggle',
  TEXT: 'text',
  NUMBER: 'number',
  DROPDOWN: 'dropdown',
  TABLE: 'table',
  OPTION: 'option',
};

```

init.js

```

import { buildFolder } from
'./buildFolder.js';
import { treeData } from './data.js';
import { setRerenderCallback } from
'./state.js';

```

```

const root =
document.getElementById('root');
function rerender() {
  root.replaceChildren();
  const el = buildFolder(treeData);
  if (el) root.appendChild(el);
}
setRerenderCallback(rerender);
rerender();
state.js
import { initialValueData } from
'./data.js';
import { TYPES } from './types.js';
export let state =
JSON.parse(JSON.stringify(initialVal
ueData));
let rerenderFn = null;
export function
setRerenderCallback(fn) {
  rerenderFn = fn;
}
export function updateValue(id,
newValue) {
  if (typeof id === 'string') {
    state[id] = newValue;
  } else {
    state[id.tableId][id.rowIndex][id.elem
entId] = newValue;
  }
  rerenderFn?.();
}
export function
isElementVisible(element, valueData)
{
  if (!element.dependency ||
!element.dependency[0]) return true;
  const { watchedId, rule } =
element.dependency[0];

```

```

const watchedValue =
valueData[watchedId];
return rule.map(r => checkRule(r,
watchedValue)).every(r => r ===
'visible');
}
function checkRule(rule,
watchedValue) {
  switch (rule.operator) {
    case '<': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue <
rule.value, rule.result) : 'visible';
    case '>': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue >
rule.value, rule.result) : 'visible';
    case '<=': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue <=
rule.value, rule.result) : 'visible';
    case '>=': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue >=
rule.value, rule.result) : 'visible';
    case '===': return
resolveVisibility(watchedValue ===
rule.value, rule.result);
    default: return 'visible';
  }
}
function
resolveVisibility(conditionMet,
action) {

```

```

    return conditionMet ? action :
    (action === 'hide' ? 'visible' : 'hide');
  }
  export function getFieldValue(field,
    valueData) {
    const stored = valueData[field.id];
    switch (field.type) {
      case TYPES.TEXT: return stored
      !== undefined ? stored :
      (field.defaultValue ?? "");
      case TYPES.NUMBER: return
      stored !== undefined ? stored :
      (field.defaultValue ?? "");
      case TYPES.DROPDOWN: return
      stored !== undefined ? stored :
      (field.defaultValue ?? "");
      case TYPES.TOGGLE: return
      stored !== undefined ? stored :
      (field.defaultValue ?? false);
      case TYPES.TABLE: return stored
      ?? [];
      default: return stored;
    }
  }

```

data.js

```

let idCounter = 0;
function nanoid() { return 'fld_' +
  (++idCounter); }
function createBasicField(label, type,
  id, dependency) {
  return { id: id || nanoid(), type, label,
    dependency };
}
function createFolder(label, childs,
  id, dependency) {
  return { ...createBasicField(label,
    'folder', id, dependency), childs };
}

```

```

function createText(label,
  defaultValue, id, dependency) {
  return { ...createBasicField(label,
    'text', id, dependency), defaultValue };
}
function createNumber(label,
  defaultValue, id, dependency) {
  return { ...createBasicField(label,
    'number', id, dependency),
    defaultValue };
}
function createToggle(label,
  defaultValue, id, dependency) {
  return { ...createBasicField(label,
    'toggle', id, dependency), defaultValue
  };
}
function
createDropdownOption(label, value,
  id, dependency) {
  return { ...createBasicField(label,
    'option', id, dependency), value };
}
function createDropdown(label,
  options, defaultValue, id,
  dependency) {
  return {
    ...createBasicField(label,
    'dropdown', id, dependency),
    options: options.map(o =>
      createDropdownOption(o.label,
        o.value, undefined,
        o.optionDependency)
    ),
    defaultValue,
  };
}
function createTable(label, columns,
  id, dependency) {

```

```

    return { ...createBasicField(label,
'table', id, dependency), columns };
  }
  function
generateBooleanDependency(watched
Id, result, value) {
  return [{ watchedId, rule: [{
operator: '===', result, value }] }];
}
  function
generateDropDownDependency(watc
hedIds) {
  return watchedIds.map(id => ({
    watchedId: id,
    rule: [{ operator: '===', result:
'visible', value: true }],
  }));
}
  const genders = ['Чоловік', 'Жінка',
'Інше'];
  const planets = ['Венера', 'Земля',
'Нептун', 'Сатурн', 'Уран', 'Плутон'];
  const cars = ['Audi', 'BMW',
'Mercedes', 'Toyota', 'Honda', 'Ford',
'Tesla', 'Mazda', 'Kia', 'Nissan'];
  const randomItem = (array) =>
array[Math.floor(Math.random() *
array.length)];
  const randomInt = (min, max) =>
Math.floor(Math.random() * (max -
min + 1)) + min;
  const randomBool = () =>
Math.random() < 0.5;
  const allUsers = Array.from({ length:
10000 }, (_, i) => ({
    id: i + 1,
    'table-toggle': randomBool(),
    'table-text': randomItem(genders),
    'table-number': randomInt(1, 100),

```

```

    'table-dropdown-stat':
randomItem(genders),
    'table-dropdown-star':
randomItem(planets),
    'table-dropdown-car':
randomItem(cars),
    'table-dropdown-star-childrens':
String(randomInt(0, 10)),
  }));
  function getUsers(count) { return
allUsers.slice(0, count); }
  const treeData = createFolder('Папка
1', [
    createFolder('Папка 2', [
      createFolder('Папка 3', [
        createToggle('Активувати вибір
статі', true, 'state-toggle'),
        createFolder('Папка 4', [
          createFolder('Папка 5', [
            createFolder('Папка 6', [
              createToggle('Венера', true,
'venera',
generateBooleanDependency('planet-
toggle', 'hide', false)),
              createToggle('Простий
перемикач', true, 'prosto'),
              createFolder('Папка 7', [
                createFolder('Папка 8', [
                  createFolder('Папка 9', [
                    createFolder('Папка 10', [
                      createFolder('Папка 11', [
                        createFolder('Налаштування
доступу до вибірки статі', [
                          createToggle('Чоловік', true, 'male',
generateBooleanDependency('state-
toggle', 'hide', false)),

```

```

createToggle('Жінка', true, 'female',
generateBooleanDependency('state-
toggle', 'hide', false)),
  createToggle('Інше', false, 'other',
generateBooleanDependency('state-
toggle', 'hide', false)),
    ], 'human'),
      createFolder('Папка 12',
[
        createFolder('Папка
13', [
          createFolder('Папка
14', [
            createToggle('Земля', true, 'zemlya',
generateBooleanDependency('planet-
toggle', 'hide', false)),
              createFolder('Папка
15', [
                createFolder('Папка 16', [
                  createFolder('Папка 17', [
                    createFolder('Папка 18', [
                      createFolder('Папка 19', [
                        createToggle('Нептун', true, 'neptun',
generateBooleanDependency('planet-
toggle', 'hide', false)),
                          createFolder('Папка 20', [
                            createFolder('Папка 21', [
                              createFolder('Папка 22', [
                                createToggle('Сатурн', false, 'saturn',

```

```

generateBooleanDependency('planet-
toggle', 'hide', false)),
  createFolder('Папка 23', [
    createFolder('Папка 24', [
      createFolder('Папка 25', [
        createFolder('Папка 26', [
          createToggle('Уран', false, 'uran',
generateBooleanDependency('planet-
toggle', 'hide', false)),
            createFolder('Папка 27', [
              createFolder('Папка 28', [
                createToggle('Активувати вибір
планет', true, 'planet-toggle'),
                createToggle('Плутон', false, 'pluton',
generateBooleanDependency('planet-
toggle', 'hide', false)),
                  createFolder('Папка 29', [
                    createFolder('Папка 30', [
                      createTable('Користувачі', [
                        createToggle('Активний
користувач', true, 'table-toggle'),
                        createText("Ім'я", "", 'table-text'),
                        createNumber('Вік', 1, 'table-
number'),
                        createDropdown('Стать', [
                          { label: 'Чоловік', value: 'Чоловік',
optionDependency:
generateBooleanDependency('male',
'hide', false) },
                          { label: 'Жінка', value: 'Жінка',
optionDependency:
generateBooleanDependency('female',
'hide', false) },
                          { label: 'Інше', value: 'Інше',
optionDependency:
generateBooleanDependency('other',
'hide', false) },

```

```

], ", 'table-dropdown-stat',
generateDropDownDependency(['male', 'female', 'other'])),
  createDropdown('Місце
знаходження', [
{ label: 'Венера', value: 'Венера',
optionDependency:
generateBooleanDependency('venera',
'hide', false) },
{ label: 'Земля', value: 'Земля',
optionDependency:
generateBooleanDependency('zemlya'
, 'hide', false) },
{ label: 'Нептун', value: 'Нептун',
optionDependency:
generateBooleanDependency('neptun',
'hide', false) },
{ label: 'Сатурн', value: 'Сатурн',
optionDependency:
generateBooleanDependency('saturn',
'hide', false) },
{ label: 'Уран', value: 'Уран',
optionDependency:
generateBooleanDependency('uran',
'hide', false) },
{ label: 'Плутон', value: 'Плутон',
optionDependency:
generateBooleanDependency('pluton',
'hide', false) },
], 'Земля', 'table-dropdown-star',

generateDropDownDependency(['venera', 'zemlya', 'neptun', 'saturn', 'uran',
'pluton'])),
  createDropdown('Машина', [
{ label: 'BMW', value: 'BMW' },
{ label: 'Audi', value: 'Audi' },
{ label: 'Ferrari', value: 'Ferrari' },
{ label: 'Aston Martin', value: 'Aston
Martin' },
], ", 'table-dropdown-car'),
  createDropdown('Діти', [
{ label: '1', value: '1' },
{ label: '2', value: '2' },
{ label: '3', value: '3' },
{ label: '4', value: '4' },
{ label: '5', value: '5' },
{ label: '6', value: '6' },
], ", 'table-dropdown-star-childrens'),

], 'users'),
],
'folder_30'),
],
'folder_29'),
],
'folder_28'),
],
'folder_27'),
],
'folder_26'),
],
'folder_25'),
],
'folder_24'),
],
'folder_23'),
],
'folder_22'),
],
'folder_21'),
], 'folder_20'),
], 'folder_19'),
], 'folder_18'),
], 'folder_17'),
], 'folder_16'),
], 'folder_15'),

```

```

        ], 'folder_14'),
        ], 'folder_13'),
        ], 'folder_12'),
        ], 'folder_11'),
        ], 'folder_10'),
        ], 'folder_9'),
        ], 'folder_8'),
        ], 'folder_7'),
        ], 'folder_6'),
        ], 'folder_5'),
        ], 'folder_4'),
        ], 'folder_3'),
        ], 'folder_2'),
    ], 'folder_1');
const initialValueData = {
  'additional-folder': '',
  'planet-toggle': true,
  'state-toggle': true,
  'venera': true,
  'zemlya': true,
  'neptun': true,
  'saturn': false,
  'uran': false,
  'pluton': false,
  'human': '',
  'male': true,
  'female': true,
  'other': false,
  'prosto': true,
  'users': getUsers(100),
};
export { treeData, initialValueData };
buildCell.js
import { buildToggle } from
'./buildToggle.js';
import { buildText } from
'./buildText.js';
import { buildNumber } from
'./buildNumber.js';

```

```

import { buildDropdown } from
'./buildDropdown.js';
import { TYPES } from './types.js';
export function buildCell(column,
rowIndex, tableId) {
  console.log(2)
  switch (column.type) {
    case TYPES.TOGGLE: return
buildToggle(column, true, rowIndex,
tableId);
    case TYPES.TEXT: return
buildText(column, true, rowIndex,
tableId);
    case TYPES.NUMBER: return
buildNumber(column, true, rowIndex,
tableId);
    case TYPES.DROPDOWN: return
buildDropdown(column, true,
rowIndex, tableId);
    default: return
document.createElement('div');
  }
}
buildChild.js
import { buildFolder } from
'./buildFolder.js';
import { buildToggle } from
'./buildToggle.js';
import { buildText } from
'./buildText.js';
import { buildNumber } from
'./buildNumber.js';
import { buildDropdown } from
'./buildDropdown.js';
import { buildTable } from
'./buildTable.js';
import { TYPES } from './types.js';
export function buildChild(child) {
  console.log(1)

```

```

switch (child.type) {
  case TYPES.FOLDER: return
    buildFolder(child);
  case TYPES.TOGGLE: return
    buildToggle(child, false, null, null);
  case TYPES.TEXT: return
    buildText(child, false, null, null);
  case TYPES.NUMBER: return
    buildNumber(child, false, null, null);
  case TYPES.DROPDOWN: return
    buildDropdown(child, false, null,
    null);
  case TYPES.TABLE: return
    buildTable(child);
  default: return
    document.createElement('div');
}

```

buildDropdown.js

```

import { state, isElementVisible,
updateValue, getFieldValue } from
'./state.js';
export function
buildDropdown(treeData, inTable,
rowIndex, tableId) {
  if (!isElementVisible(treeData,
state)) return null;
  const sourceData = (inTable &&
tableId != null)
    ? state[tableId][rowIndex]
    : state;
  const wrapper =
document.createElement('div');
  wrapper.className = 'dropdown-
wrapper' + (inTable ? ' dropdown-
wrapper-for-table' : '');
  if (!inTable) {
    const span =
document.createElement('span');

```

```

    span.textContent = treeData.label;
    wrapper.appendChild(span);
  }
  const select =
document.createElement('select');
  select.className = 'dropdown-
input';
  const defaultOpt =
document.createElement('option');
  defaultOpt.value = "";
  defaultOpt.textContent = 'Виберіть
значення';
  select.appendChild(defaultOpt);
  for (const opt of treeData.options) {
    if (isElementVisible(opt, state)) {
      const o =
document.createElement('option');
      o.value = opt.value;
      o.textContent = opt.label;
      select.appendChild(o);
    }
  }
  select.value =
String(getFieldValue(treeData,
sourceData) ?? "");
  select.addEventListener('change', ()
=> {
    if (inTable && tableId != null) {
      updateValue({ tableId, elementId:
treeData.id, rowIndex }, select.value);
    } else {
      updateValue(treeData.id,
select.value);
    }
  });
  wrapper.appendChild(select);
  return wrapper;
}

```

buildFolder.js

```

import { state, isElementVisible }
from './state.js';
import { buildChild } from
'./buildChild.js';
export function buildFolder(treeData)
{
  if (!isElementVisible(treeData,
state)) return null;
  const el =
document.createElement('div');
  el.className = 'folder';
  const label =
document.createElement('div');
  label.className = 'folder-label';
  label.textContent = 'Панка: ' +
treeData.label;
  el.appendChild(label);
  for (const child of treeData.children) {
    const childEl = buildChild(child);
    if (childEl)
el.appendChild(childEl);
  }
  return el;
}

```

buildNumber.js

```

import { state, isElementVisible,
updateValue, getFieldValue } from
'./state.js';
export function
buildNumber(treeData, inTable,
rowIndex, tableId) {
  if (!isElementVisible(treeData,
state)) return null;
  const sourceData = (inTable &&
tableId != null)
    ? state[tableId][rowIndex]
    : state;
  const wrapper =
document.createElement('div');

```

```

  wrapper.className = 'number-
wrapper' + (inTable ? ' number-
wrapper-for-table' : '');
  if (!inTable) {
    const span =
document.createElement('span');
    span.textContent = treeData.label;
    wrapper.appendChild(span);
  }
  const input =
document.createElement('input');
  input.type = 'number';
  input.className = 'number-input';
  input.value =
String(getFieldValue(treeData,
sourceData) ?? '');
  input.addEventListener('input', () =>
{
    const val = Number(input.value);
    if (inTable && tableId != null) {
      updateValue({ tableId, elementId:
treeData.id, rowIndex }, val);
    } else {
      updateValue(treeData.id, val);
    }
  });
  wrapper.appendChild(input);
  return wrapper;
}

```

buildTable.js

```

import { state, isElementVisible,
getFieldValue } from './state.js';
import { buildCell } from
'./buildCell.js';
export function buildTable(treeData)
{
  if (!isElementVisible(treeData,
state)) return null;

```

```

const rows = getFieldValue(treeData,
state) || [];
const wrapper =
document.createElement('div');
  wrapper.className = 'table-
wrapper';
  const label =
document.createElement('span');
  label.className = 'table-label';
  label.textContent = treeData.label;
  wrapper.appendChild(label);
  const table =
document.createElement('table');
  table.className = 'data-table';
  const thead =
document.createElement('thead');
  const headerRow =
document.createElement('tr');
  for (const col of treeData.columns) {
    const th =
document.createElement('th');
    th.style.width = '150px';
    th.textContent = col.label;
    headerRow.appendChild(th);
  }
  thead.appendChild(headerRow);
  table.appendChild(thead);
  const tbody =
document.createElement('tbody');
  const frag =
document.createDocumentFragment()
;
  rows.forEach((row, rowIndex) => {
    const tr =
document.createElement('tr');
    for (const col of treeData.columns)
    {
      const td =
document.createElement('td');

```

```

    td.style.width = '150px';
    const cellEl = buildCell(col,
rowIndex, treeData.id);
    if (cellEl) td.appendChild(cellEl);
    tr.appendChild(td);
  }
  frag.appendChild(tr);
});
tbody.appendChild(frag);
table.appendChild(tbody);
wrapper.appendChild(table);
return wrapper;
}

```

buildText.js

```

import { state, isElementVisible,
updateValue, getFieldValue } from
'./state.js';
export function buildText(treeData,
inTable, rowIndex, tableId) {
  if (!isElementVisible(treeData,
state)) return null;
  const sourceData = (inTable &&
tableId != null)
  ? state[tableId][rowIndex]
  : state;
  const wrapper =
document.createElement('div');
  wrapper.className = 'text-wrapper'
+ (inTable ? ' text-wrapper-for-table' :
'');
  if (!inTable) {
    const span =
document.createElement('span');
    span.textContent = treeData.label;
    wrapper.appendChild(span);
  }
  const input =
document.createElement('input');
  input.type = 'text';

```

```

    input.className = 'text-input';
    input.value =
String(getFieldValue(treeData,
sourceData) ?? '');
    input.addEventListener('input', () =>
{
    if (inTable && tableId !== null) {
        updateValue({ tableId, elementId:
treeData.id, rowIndex }, input.value);
    } else {
        updateValue(treeData.id,
input.value);
    }
});
    wrapper.appendChild(input);
    return wrapper;
}

```

buildToggle.js

```

import { state, isElementVisible,
updateValue, getFieldValue } from
'./state.js';
export function
buildToggle(treeData, inTable,
rowIndex, tableId) {
    if (!isElementVisible(treeData,
state)) return null;
    const sourceData = (inTable &&
tableId !== null)
        ? state[tableId][rowIndex]
        : state;
    const wrapper =
document.createElement('div');

```

```

    wrapper.className = 'toggle-
wrapper' + (inTable ? ' toggle-
wrapper-for-table' : '');
    if (!inTable) {
        const span =
document.createElement('span');
        span.textContent = treeData.label;
        wrapper.appendChild(span);
    }
    const checkbox =
document.createElement('input');
    checkbox.type = 'checkbox';
    checkbox.className = 'checkbox-
element';
    checkbox.checked =
Boolean(getFieldValue(treeData,
sourceData));

```

```

checkboxbox.addEventListener('change',
() => {
    if (inTable && tableId !== null) {
        updateValue({ tableId, elementId:
treeData.id, rowIndex },
checkboxbox.checked);
    } else {
        updateValue(treeData.id,
checkboxbox.checked);
    }
});
    wrapper.appendChild(checkboxbox);
    return wrapper;
}

```

Лістинг програмного коду засобами нативного JavaScript

index.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport"
content="width=device-width, initial-
scale=1.0" />
  <title>Only JS (HOC)</title>
  <link rel="stylesheet"
href="styles.css" />
</head>
<body>
  <div id="root"></div>
  <script type="module"
src="init.js"></script>
</body>
</html>
```

styles.css

```
*, *::before, *::after { box-sizing:
border-box; }
body {
  margin: 0;
  padding: 0;
  font-family: -apple-system,
BlinkMacSystemFont, "Segoe UI",
"Roboto",
  "Oxygen", "Ubuntu", "Cantarell",
"Fira Sans", "Droid Sans",
  "Helvetica Neue", sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
html, body, #root {
```

```
width: 100%;
height: 100%;
padding: 0;
margin: 0;
}
.folder {
  display: flex;
  flex-direction: column;
  border: 1px solid black;
  border-width: 1px 0 0 1px;
  padding: 10px;
  margin-bottom: 10px;
  width: 1250px;
}
.folder-label {
  font-size: larger;
  display: flex;
  flex-direction: column;
  margin-bottom: 6px;
}
.checkbox-element { margin: 5px; }
.toggle-wrapper {
  display: flex;
  align-items: center;
}
.toggle-wrapper-for-table { justify-
content: center; }
.toggle-wrapper span { white-space:
nowrap; }
.text-wrapper {
  width: 300px;
  display: flex;
  align-items: center;
```

```

    margin: 5px 0;
  }
  .text-wrapper-for-table { width:
150px; }
  .text-wrapper span { white-space:
nowrap; }
  .text-input { width: 100%; margin-
left: 10px; }
  .number-wrapper {
    width: 300px;
    display: flex;
    align-items: center;
    margin: 5px 0;
  }
  .number-wrapper-for-table { width:
150px; }
  .number-wrapper span { white-space:
nowrap; }
  .number-input { width: 100%;
margin-left: 10px; }
  .dropdown-wrapper {
    width: 300px;
    display: flex;
    align-items: center;
    margin: 5px 0;
  }
  .dropdown-wrapper-for-table { width:
150px; }
  .dropdown-wrapper span { white-
space: nowrap; }
  .dropdown-input { width: 100%;
margin-left: 10px; }
  .table-wrapper {
    height: 600px;
    overflow: auto;
    margin: 10px 0;
  }
  .table-wrapper > .table-label {
    white-space: nowrap;

```

```

    display: block;
  }
  .data-table {
    margin-top: 10px;
    border-collapse: collapse;
    border: 2px solid rgb(140 140 140);
    font-family: sans-serif;
    font-size: 0.8rem;
    letter-spacing: 1px;
  }
  .data-table thead { background-color:
rgb(228 240 245); }
  .data-table th,
  .data-table td {
    border: 1px solid rgb(160 160 160);
    padding: 8px 10px;
    width: 150px;
  }
  .data-table td:last-of-type { text-align:
center; }
  .data-table tbody > tr:nth-of-
type(even) {
    background-color: rgb(237 238 242);
  }

```

types.js

```

export const TYPES = {
  FOLDER: 'folder',
  TOGGLE: 'toggle',
  TEXT: 'text',
  NUMBER: 'number',
  DROPDOWN: 'dropdown',
  TABLE: 'table',
  OPTION: 'option',
};
init.js
import { buildFolder } from
'./buildFolder.js';
import { treeData } from './data.js';

```

```

document.getElementById('root').app
endChild(buildFolder(treeData));
state.js
import { initialValueData } from
'./data.js';
import { TYPES } from './types.js';
export let state =
JSON.parse(JSON.stringify(initialVal
ueData));
const registry = [];
const updateCounts = {};
export function getUpdateCounts() {
  return { ...updateCounts };
}
export function
registerWatcher(depList, updateFn,
container, id) {
  registry.push({ depList, updateFn,
container, id });
}
function notifyWatchers(changedId) {
  for (let i = registry.length - 1; i >= 0;
i--) {
    if (registry[i].container &&
!registry[i].container.isConnected) {
      registry.splice(i, 1);
    }
  }
  for (const { depList, updateFn, id }
of registry) {
    if (depList.includes(changedId)) {
      if (id !== undefined)
updateCounts[id] = (updateCounts[id]
?? 0) + 1;
      updateFn(state);
    }
  }
}

```

```

export function updateValue(id,
newValue) {
  if (typeof id === 'string') {
    state[id] = newValue;
    notifyWatchers(id);
  } else {
    state[id.tableId][id.rowIndex][id.elem
entId] = newValue;
  }
}
export function
isElementVisible(element, valueData)
{
  if (!element.dependency ||
!element.dependency[0]) return true;
  const { watchedId, rule } =
element.dependency[0];
  const watchedValue =
valueData[watchedId];
  return rule.map(r => checkRule(r,
watchedValue)).every(r => r ===
'visible');
}
function checkRule(rule,
watchedValue) {
  switch (rule.operator) {
    case '<': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue <
rule.value, rule.result) : 'visible';
    case '>': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue >
rule.value, rule.result) : 'visible';
    case '<=': return (typeof
watchedValue === 'number' &&

```

```

typeof rule.value === 'number') ?
resolveVisibility(watchedValue <=
rule.value, rule.result) : 'visible';
  case '>=': return (typeof
watchedValue === 'number' &&
typeof rule.value === 'number') ?
resolveVisibility(watchedValue >=
rule.value, rule.result) : 'visible';
  case '===': return
resolveVisibility(watchedValue ===
rule.value, rule.result);
  default: return 'visible';
}
}
function
resolveVisibility(conditionMet,
action) {
  return conditionMet ? action :
(action === 'hide' ? 'visible' : 'hide');
}
export function getFieldValue(field,
valueData) {
  const stored = valueData[field.id];
  switch (field.type) {
    case TYPES.TEXT: return stored
    !== undefined ? stored :
(field.defaultValue ?? "");
    case TYPES.NUMBER: return
stored !== undefined ? stored :
(field.defaultValue ?? "");
    case TYPES.DROPDOWN: return
stored !== undefined ? stored :
(field.defaultValue ?? "");
    case TYPES.TOGGLE: return
stored !== undefined ? stored :
(field.defaultValue ?? false);
    case TYPES.TABLE: return stored
?? [];
  default: return stored;

```

```

}
}
data.js
let idCounter = 0;
function nanoid() { return 'fld_' +
(++idCounter); }
function createBasicField(label, type,
id, dependency) {
  return { id: id || nanoid(), type, label,
dependency };
}
function createFolder(label, childs,
id, dependency) {
  return { ...createBasicField(label,
'folder', id, dependency), childs };
}
function createText(label,
defaultValue, id, dependency) {
  return { ...createBasicField(label,
'text', id, dependency), defaultValue };
}
function createNumber(label,
defaultValue, id, dependency) {
  return { ...createBasicField(label,
'number', id, dependency),
defaultValue };
}
function createToggle(label,
defaultValue, id, dependency) {
  return { ...createBasicField(label,
'toggle', id, dependency), defaultValue
};
}
function
createDropdownOption(label, value,
id, dependency) {
  return { ...createBasicField(label,
'option', id, dependency), value };
}

```

```

function createDropdown(label,
options, defaultValue, id,
dependency) {
  return {
    ...createBasicField(label,
'dropdown', id, dependency),
    options: options.map(o =>
      createDropdownOption(o.label,
o.value, undefined,
o.optionDependency)
    ),
    defaultValue,
  };
}

function createTable(label, columns,
id, dependency) {
  return { ...createBasicField(label,
'table', id, dependency), columns };
}

function
generateBooleanDependency(watched
Id, result, value) {
  return [{ watchedId, rule: [{
operator: '===', result, value }] }];
}

function
generateDropDownDependency(watc
hedIds) {
  return watchedIds.map(id => ({
    watchedId: id,
    rule: [{ operator: '===', result:
'visible', value: true }],
  }));
}

const genders = ['Чоловік', 'Жінка',
'Інше'];
const planets = ['Венера', 'Земля',
'Нептун', 'Сатурн', 'Уран', 'Плутон'];

```

```

const cars = ['Audi', 'BMW',
'Mercedes', 'Toyota', 'Honda', 'Ford',
'Tesla', 'Mazda', 'Kia', 'Nissan'];
const randomItem = (array) =>
array[Math.floor(Math.random() *
array.length)];
const randomInt = (min, max) =>
Math.floor(Math.random() * (max -
min + 1)) + min;
const randomBool = () =>
Math.random() < 0.5;
const allUsers = Array.from({ length:
10000 }, (_, i) => ({
  id: i + 1,
  'table-toggle': randomBool(),
  'table-text': randomItem(genders),
  'table-number': randomInt(1, 100),
  'table-dropdown-stat':
randomItem(genders),
  'table-dropdown-star':
randomItem(planets),
  'table-dropdown-car':
randomItem(cars),
  'table-dropdown-star-childrens':
String(randomInt(0, 10)),
}));
function getUsers(count) { return
allUsers.slice(0, count); }
const treeData = createFolder('Папка
1', [
  createFolder('Папка 2', [
    createFolder('Папка 3', [
      createToggle('Активувати вибір
статі', true, 'state-toggle'),
      createFolder('Папка 4', [
        createFolder('Папка 5', [
          createFolder('Папка 6', [
            createToggle('Венера', true,
'venera',

```

```

generateBooleanDependency('planet-
toggle', 'hide', false)),
    createToggle('Простий
перемикач', true, 'prosto'),
    createFolder('Папка 7', [
        createFolder('Папка 8', [
            createFolder('Папка 9', [
                createFolder('Папка 10', [
                    createFolder('Папка 11', [

createFolder('Налаштування
доступу до вибірки статі', [

createToggle('Чоловік', true, 'male',
generateBooleanDependency('state-
toggle', 'hide', false)),
        createToggle('Жінка',
true, 'female',
generateBooleanDependency('state-
toggle', 'hide', false)),
        createToggle('Інше',
false, 'other',
generateBooleanDependency('state-
toggle', 'hide', false)),
        ], 'human'),
        createFolder('Папка 12',
[
            createFolder('Папка
13', [
                createFolder('Папка
14', [

createToggle('Земля', true, 'zemlya',
generateBooleanDependency('planet-
toggle', 'hide', false)),
            createFolder('Папка
15', [

createFolder('Папка 16', [

createFolder('Папка 17', [

createFolder('Папка 18', [

createFolder('Папка 19', [

createToggle('Нептун', true, 'neptun',
generateBooleanDependency('planet-
toggle', 'hide', false)),

createFolder('Папка 20', [

createFolder('Папка 21', [

createFolder('Папка 22', [

createToggle('Сатурн', false, 'saturn',
generateBooleanDependency('planet-
toggle', 'hide', false)),

createFolder('Папка 23', [

createFolder('Папка 24', [

createFolder('Папка 25', [

createFolder('Папка 26', [

createToggle('Уран', false, 'uran',
generateBooleanDependency('planet-
toggle', 'hide', false)),

createFolder('Папка 27', [
.....createFold
er('Папка 28', [
createToggle('Активувати вибір
планет', true, 'planet-toggle'),

```

```

createToggle('Плутон', false, 'pluton',
generateBooleanDependency('planet-
toggle', 'hide', false)),
createFolder('Папка 29', [
createFolder('Папка 30', [
createTable('Користувачі', [
createToggle('Активний
користувач', true, 'table-toggle'),
createText("Ім'я", "", 'table-text'),
createNumber('Вік', 1, 'table-
number'),
createDropdown('Стать', [
{ label: 'Чоловік', value: 'Чоловік',
optionDependency:
generateBooleanDependency('male',
'hide', false) },
{ label: 'Жінка', value: 'Жінка',
optionDependency:
generateBooleanDependency('female',
'hide', false) },
{ label: 'Інше', value: 'Інше',
optionDependency:
generateBooleanDependency('other',
'hide', false) },
], "", 'table-dropdown-stat',
generateDropDownDependency(['mal
e', 'female', 'other'])),
createDropdown('Місце
знаходження', [
{ label: 'Венера', value: 'Венера',
optionDependency:
generateBooleanDependency('venera',
'hide', false) },
{ label: 'Земля', value: 'Земля',
optionDependency:
generateBooleanDependency('zemlya'
, 'hide', false) },
{ label: 'Нептун', value: 'Нептун',
optionDependency:

```

```

generateBooleanDependency('neptun',
'hide', false) },
{ label: 'Сатурн', value: 'Сатурн',
optionDependency:
generateBooleanDependency('saturn',
'hide', false) },
{ label: 'Уран', value: 'Уран',
optionDependency:
generateBooleanDependency('uran',
'hide', false) },
{ label: 'Плутон', value: 'Плутон',
optionDependency:
generateBooleanDependency('pluton',
'hide', false) },
], 'Земля', 'table-dropdown-star',
generateDropDownDependency(['ven
era', 'zemlya', 'neptun', 'saturn', 'uran',
'pluton'])),
createDropdown('Машина', [
{ label: 'BMW', value: 'BMW' },
{ label: 'Audi', value: 'Audi' },
{ label: 'Ferrari', value: 'Ferrari' },
{ label: 'Aston Martin', value: 'Aston
Martin' },
], "", 'table-dropdown-car'),
createDropdown('Діти', [
{ label:
'1', value: '1' },
{ label:
'2', value: '2' },
{ label: '3',
value: '3' },
{ label: '4',
value: '4' },
{ label: '5', value:
'5' },
{ label: '6', value: '6'
},
], "", 'table-dropdown-star-childrens'),

```

```

    ], 'users'),
    ], 'folder_2'),
    ], 'folder_1');

'folder_30'),
    ],
'folder_29'),
    ],
'folder_28'),
    ],
'folder_27'),
    ],
'folder_26'),
    ],
'folder_25'),
    ],
'folder_24'),
    ],
'folder_23'),
    ],
'folder_22'),
    ],
'folder_21'),
    ], 'folder_20'),
    ], 'folder_19'),
    ], 'folder_18'),
    ], 'folder_17'),
    ], 'folder_16'),
    ], 'folder_15'),
    ], 'folder_14'),
    ], 'folder_13'),
    ], 'folder_12'),
    ], 'folder_11'),
    ], 'folder_10'),
    ], 'folder_9'),
    ], 'folder_8'),
    ], 'folder_7'),
    ], 'folder_6'),
    ], 'folder_5'),
    ], 'folder_4'),

    ], 'folder_3'),
    ], 'folder_2'),
    ], 'folder_1');

const initialValueData = {
  'additional-folder': '',
  'planet-toggle': true,
  'state-toggle': true,
  'venera': true,
  'zemlya': true,
  'neptun': true,
  'saturn': false,
  'uran': false,
  'pluton': false,
  'human': '',
  'male': true,
  'female': true,
  'other': false,
  'prosto': true,
  'users': getUsers(100),
};

function
collectDependencies(rootNode) {
  const dependencyMap = {};
  function traverse(currentNode) {
    if (currentNode.dependency &&
currentNode.dependency.length > 0) {
      dependencyMap[currentNode.id]
=
currentNode.dependency.map(depend
ency => dependency.watchedId);
    }
    if (currentNode.chilts) {
      for (const child of
currentNode.chilts) traverse(child);
    }
    if (currentNode.columns) {

```

```

    for (const column of
currentNode.columns)
traverse(column);
  }
  if (currentNode.options) {
    for (const option of
currentNode.options)
traverse(option);
  }
}
traverse(rootNode);
return dependencyMap;
}
const dependencyMap =
collectDependencies(treeData);
export { treeData, initialValueData,
collectDependencies, dependencyMap
};
buildCell.js
import { buildToggle } from
'./buildToggle.js';
import { buildText } from
'./buildText.js';
import { buildNumber } from
'./buildNumber.js';
import { buildDropdown } from
'./buildDropdown.js';
import { TYPES } from './types.js';
export function buildCell(column,
rowIndex, tableId) {
  switch (column.type) {
    case TYPES.TOGGLE: return
buildToggle(column, true, rowIndex,
tableId);
    case TYPES.TEXT: return
buildText(column, true, rowIndex,
tableId);

```

```

    case TYPES.NUMBER: return
buildNumber(column, true, rowIndex,
tableId);
    case TYPES.DROPDOWN: return
buildDropdown(column, true,
rowIndex, tableId);
    default: return
document.createElement('div');
  }
}

```

buildChild.js

```

import { buildFolder } from
'./buildFolder.js';
import { buildToggle } from
'./buildToggle.js';
import { buildText } from
'./buildText.js';
import { buildNumber } from
'./buildNumber.js';
import { buildDropdown } from
'./buildDropdown.js';
import { buildTable } from
'./buildTable.js';
import { TYPES } from './types.js';
export function buildChild(child) {
  switch (child.type) {
    case TYPES.FOLDER: return
buildFolder(child);
    case TYPES.TOGGLE: return
buildToggle(child, false, null, null);
    case TYPES.TEXT: return
buildText(child, false, null, null);
    case TYPES.NUMBER: return
buildNumber(child, false, null, null);
    case TYPES.DROPDOWN: return
buildDropdown(child, false, null,
null);
    case TYPES.TABLE: return
buildTable(child);

```

```

    default: return
document.createElement('div');
}
}
buildDropdown.js
import { state, isElementVisible,
registerWatcher, updateValue,
getFieldValue } from './state.js';
import { dependencyMap } from
'./data.js';
export function
buildDropdown(treeData, inTable,
rowIndex, tableId) {
    const depList =
dependencyMap[treeData.id] ?? [];
    function buildContent(currentState)
    {
        if (!isElementVisible(treeData,
currentState)) return null;
        const sourceData = (inTable &&
tableId != null)
        ? currentState[tableId][rowIndex]
        : currentState;
        const wrapper =
document.createElement('div');
        wrapper.className = 'dropdown-
wrapper' + (inTable ? ' dropdown-
wrapper-for-table' : '');
        if (!inTable) {
            const span =
document.createElement('span');
            span.textContent = treeData.label;
            wrapper.appendChild(span);
        }
        const select =
document.createElement('select');
        select.className = 'dropdown-
input';

```

```

const defaultOpt =
document.createElement('option');
    defaultOpt.value = "";
    defaultOpt.textContent = 'Виберіть
значення';
    select.appendChild(defaultOpt);
    for (const opt of treeData.options)
    {
        if (isElementVisible(opt,
currentState)) {
            const o =
document.createElement('option');
            o.value = opt.value;
            o.textContent = opt.label;
            select.appendChild(o);
        }
    }
    select.value =
String(getFieldValue(treeData,
sourceData) ?? "");
    select.addEventListener('change', ()
=> {
        if (inTable && tableId != null) {
            updateValue({ tableId,
elementId: treeData.id, rowIndex },
select.value);
        } else {
            updateValue(treeData.id,
select.value);
        }
    });
    wrapper.appendChild(select);
    return wrapper;
}
if (depList.length === 0) return
buildContent(state);
const container =
document.createElement('div');
    container.style.display = 'contents';

```

```

function render(currentState) {
  console.log('render:
buildDropdown —', treeData.id);
  container.replaceChildren();
  const el =
buildContent(currentState);
  if (el) container.appendChild(el);
}
render(state);
registerWatcher(depList, render,
container, treeData.id);
return container;
}
buildFolder.js
import { state, isElementVisible,
registerWatcher } from './state.js';
import { buildChild } from
'./buildChild.js';
import { dependencyMap } from
'./data.js';
export function buildFolder(treeData)
{
  const depList =
dependencyMap[treeData.id] ?? [];
  function buildContent(currentState)
  {
    if (!isElementVisible(treeData,
currentState)) return null;
    const el =
document.createElement('div');
    el.className = 'folder';
    const label =
document.createElement('div');
    label.className = 'folder-label';
    label.textContent = 'Панка: ' +
treeData.label;
    el.appendChild(label);
    for (const child of treeData.chlds)
  {

```

```

    el.appendChild(buildChild(child));
  }
  return el;
}
if (depList.length === 0) return
buildContent(state);
const container =
document.createElement('div');
container.style.display = 'contents';
function render(currentState) {
  console.log('render: buildFolder —
', treeData.id);
  container.replaceChildren();
  const el =
buildContent(currentState);
  if (el) container.appendChild(el);
}
render(state);
registerWatcher(depList, render,
container);
return container;
}
buildNumber.js
import { state, isElementVisible,
registerWatcher, updateValue,
getFieldValue } from './state.js';
import { dependencyMap } from
'./data.js';
export function
buildNumber(treeData, inTable,
rowIndex, tableId) {
  const depList =
dependencyMap[treeData.id] ?? [];
  function buildContent(currentState)
  {
    if (!isElementVisible(treeData,
currentState)) return null;

```

```

    const sourceData = (inTable &&
tableId !== null)
    ? currentState[tableId][rowIndex]
    : currentState;
    const wrapper =
document.createElement('div');
    wrapper.className = 'number-
wrapper' + (inTable ? ' number-
wrapper-for-table' : '');
    if (!inTable) {
        const span =
document.createElement('span');
        span.textContent = treeData.label;
        wrapper.appendChild(span);
    }
    const input =
document.createElement('input');
    input.type = 'number';
    input.className = 'number-input';
    input.value =
String(getFieldValue(treeData,
sourceData) ?? '');
    input.addEventListener('input', ()
=> {
        const val = Number(input.value);
        if (inTable && tableId !== null) {
            updateValue({ tableId,
elementId: treeData.id, rowIndex },
val);
        } else {
            updateValue(treeData.id, val);
        }
    });
    wrapper.appendChild(input);
    return wrapper;
}
if (depList.length === 0) return
buildContent(state);

```

```

    const container =
document.createElement('div');
    container.style.display = 'contents';
    function render(currentState) {
        console.log('render: buildNumber
—', treeData.id);
        container.replaceChildren();
        const el =
buildContent(currentState);
        if (el) container.appendChild(el);
    }
    render(state);
    registerWatcher(depList, render,
container, treeData.id);
    return container;
}

```

buildTable.js

```

import { state, isElementVisible,
registerWatcher, getFieldValue }
from './state.js';
import { buildCell } from
'./buildCell.js';
import { dependencyMap } from
'./data.js';
export function buildTable(treeData)
{
    const depList =
dependencyMap[treeData.id] ?? [];
    function buildContent(currentState)
    {
        if (!isElementVisible(treeData,
currentState)) return null;
        const rows =
getFieldValue(treeData, currentState)
|| [];
        const wrapper =
document.createElement('div');
        wrapper.className = 'table-
wrapper';
    }
}

```

```

    const label =
document.createElement('span');
    label.className = 'table-label';
    label.textContent = treeData.label;
    wrapper.appendChild(label);
    const table =
document.createElement('table');
    . table.className = 'data-table';
    const thead =
document.createElement('thead');
    const headerRow =
document.createElement('tr');
    for (const col of treeData.columns)
{
    const th =
document.createElement('th');
    th.style.width = '150px';
    th.textContent = col.label;
    headerRow.appendChild(th);
}
    thead.appendChild(headerRow);
    table.appendChild(thead);
    const tbody =
document.createElement('tbody');
    const frag =
document.createDocumentFragment()
;
    rows.forEach((row, rowIndex) => {
    const tr =
document.createElement('tr');
    for (const col of
treeData.columns) {
    const td =
document.createElement('td');
    td.style.width = '150px';
    td.appendChild(buildCell(col,
rowIndex, treeData.id));
    tr.appendChild(td);
}

```

```

    frag.appendChild(tr);
});
tbody.appendChild(frag);
table.appendChild(tbody);
wrapper.appendChild(table);
return wrapper;
}
if (depList.length === 0) return
buildContent(state);
const container =
document.createElement('div');
container.style.display = 'contents';
function render(currentState) {
    console.log('render: buildTable —',
treeData.id);
    container.replaceChildren();
    const el =
buildContent(currentState);
    if (el) container.appendChild(el);
}
render(state);
registerWatcher(depList, render,
container, treeData.id);
return container;
}

```

buildText.js

```

import { state, isElementVisible,
registerWatcher, updateValue,
getFieldValue } from './state.js';
import { dependencyMap } from
'./data.js';
export function buildText(treeData,
inTable, rowIndex, tableId) {
    const depList =
dependencyMap[treeData.id] ?? [];
    function buildContent(currentState)
{
    if (!isElementVisible(treeData,
currentState)) return null;

```

```

    const sourceData = (inTable &&
tableId !== null)
    ? currentState[tableId][rowIndex]
    : currentState;
    const wrapper =
document.createElement('div');
    wrapper.className = 'text-wrapper'
+ (inTable ? ' text-wrapper-for-table' :
");
    if (!inTable) {
        const span =
document.createElement('span');
        span.textContent = treeData.label;
        wrapper.appendChild(span);
    }
    const input =
document.createElement('input');
    input.type = 'text';
    input.className = 'text-input';
    input.value =
String(getFieldValue(treeData,
sourceData) ?? "");
    input.addEventListener('input', ()
=> {
        if (inTable && tableId !== null) {
            updateValue({ tableId,
elementId: treeData.id, rowIndex },
input.value);
        } else {
            updateValue(treeData.id,
input.value);
        }
    });

    wrapper.appendChild(input);
    return wrapper;
}
if (depList.length === 0) return
buildContent(state);

```

```

    const container =
document.createElement('div');
    container.style.display = 'contents';
    function render(currentState) {
        console.log('render: buildText —',
treeData.id);
        container.replaceChildren();
        const el =
buildContent(currentState);
        if (el) container.appendChild(el);
    }
    render(state);
    registerWatcher(depList, render,
container, treeData.id);
    return container;
}

```

buildToggle.js

```

import { state, isElementVisible,
registerWatcher, updateValue,
getFieldValue } from './state.js';
import { dependencyMap } from
'./data.js';
export function buildToggle(treeData,
inTable, rowIndex, tableId) {
    const depList =
dependencyMap[treeData.id] ?? [];
    function buildContent(currentState)
{
        if (!isElementVisible(treeData,
currentState)) return null;
        const sourceData = (inTable &&
tableId !== null)
        ? currentState[tableId][rowIndex]
        : currentState;
        const wrapper =
document.createElement('div');
        wrapper.className = 'toggle-
wrapper' + (inTable ? ' toggle-
wrapper-for-table' : "");

```

```

    if (!inTable) {
      const span =
document.createElement('span');
      span.textContent = treeData.label;
      wrapper.appendChild(span);
    }
    const checkbox =
document.createElement('input');
    checkbox.type = 'checkbox';
    checkbox.className = 'checkbox-
element';
    checkbox.checked =
Boolean(getFieldValue(treeData,
sourceData));

checkbox.addEventListner('change',
() => {
  if (inTable && tableId != null) {
    updateValue({ tableId,
elementId: treeData.id, rowIndex },
checkbox.checked);
  } else {
    updateValue(treeData.id,
checkbox.checked);
  }
}

```

```

});
wrapper.appendChild(checkbox);
return wrapper;
}
if (depList.length === 0) return
buildContent(state);
const container =
document.createElement('div');
container.style.display = 'contents';
function render(currentState) {
  console.log('render: buildToggle —
', treeData.id);
  container.replaceChildren();
  const el =
buildContent(currentState);
  if (el) container.appendChild(el);
}
render(state);
registerWatcher(depList, render,
container, treeData.id);
return container;
}

```

**Список публікацій здобувача за темою дисертації та відомості про
апробацію результатів дисертації**

**Статті у виданнях, включених до переліку наукових фахових
видань України категорії «Б»:**

1. Войтюк О. В., Плечистий Д. Д. Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. *Технічна інженерія*. 2023. № 1 (91). С. 140-145. DOI: [https://doi.org/10.26642/ten-2023-1\(91\)-140-145](https://doi.org/10.26642/ten-2023-1(91)-140-145)
2. Войтюк О. В. Аналіз методів та технологій рендерингу вебзастосунків. *Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки*. 2025. Том 36 (75). № 6. С.74-80. DOI: <https://doi.org/10.32782/2663-5941/2025.6.2/11>.
3. Войтюк О. В., Лобанчикова Н. М. Методи оптимізації швидкодії оновлення вебзастосунків на основі об'єктів з глибокою вкладеністю. *Computer Science and Applied Mathematics*. 2026. № 1. С.56-68. DOI: <https://doi.org/10.26661/2786-6254-2026-1-06>.

Публікації у матеріалах наукових конференцій:

4. Войтюк О. В., Плечистий Д. Д. Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками. Тези V Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення». м. Житомир, 01-02 грудня 2022 р. м. Житомир: «Житомирська політехніка». 2022. С. 12-13. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2023/02/12.pdf>.
5. Войтюк О. В., Плечистий Д. Д. Багатозалежні зв'язки у рекурсивній структурі фронтенд додатку. Тези VI Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення». м.

Житомир, 29-30 листопада 2023 р. м. Житомир: «Житомирська політехніка». 2023. С. 56-57. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2024/01/56-1.pdf>.

6. Войтюк О. В. Плечистий Д. Д. Методи оптимізації рекурсивних циклів в JavaScript. Міжнародна конференція «Передові технології в інформаційно-комунікаційній інженерії» (ATICE'2024): Матеріали конф. м. Одеса, Україна, 17-20 лип., 2024 р. / гол. ред. кол. С. В. Ківалов. Одеса, Міжнар. гуманітар. ун-т. 2024. С.12-14. URL: <https://hdl.handle.net/11300/28155>; DOI: 10.32837/11300.28155.

7. Фещенко С.О., Мацієвський В.А., Нікітчук Т.М., Коренівська О.Л., Войтюк О.В. Розробка вебзастосунку для дослідження реологічних властивостей біологічних рідин. Тези XII Міжнародної науково-практичної конференції «Topical aspects of modern scientific research», 8-10 серпня 2024 р. Токіо, Японія. С. 98-101. URL: <https://surl.li/ixykb>.

8. Войтюк О. В. Глибоко вкладені структури даних із багатозалежними зв'язками: продуктивність та оновлення стану при рендерингу. Тези VIII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення, 02-03 грудня 2025 р. м. Житомир. Житомир: «Житомирська політехніка». С. 293-294. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2026/01/293.pdf>.

Відомості про апробацію результатів дисертаційного дослідження

Войтюка Олега Валерійовича

«Моделі, методи та програмні засоби оновлення інтерфейсу вебзастосунків
на основі об'єктів з глибокою вкладеністю»

зі спеціальності 121 «Інженерія програмного забезпечення»

Масові науково-практичні заходи міжнародного рівня

1. Міжнародна конференція «Передові технології в інформаційно-комунікаційній інженерії» (ATICE'2024) (Україна, м. Одеса: Міжнародний гуманітарний університет, 17-20 липня 2024 р.). Форма участі – заочна, публікація тез доповіді на тему *«Методи оптимізації рекурсивних циклів в JavaScript»* у збірнику матеріалів конференції.

2. XII Міжнародної науково-практичної конференції «Topical aspects of modern scientific research» (Японія, м. Токіо: CPN Publishing Group, 8-10 серпня 2024 р.). Форма участі – заочна, публікація тез доповіді на тему «Розробка вебзастосунку для дослідження реологічних властивостей біологічних рідин» у збірнику матеріалів конференції.

Масові науково-практичні заходи всеукраїнського рівня

3. V Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення» (Україна, м. Житомир: «Житомирська політехніка», 01-02 грудня 2022 р.). Форма участі – очна, виступ на секційному засіданні, публікація тез доповіді на тему *«Оптимізація промальовування вебзастосунку на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками»* у збірнику матеріалів конференції.

4. VI Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення» (Україна, м. Житомир: «Житомирська політехніка», 29-30 листопада 2023 р.) Форма участі – очна, виступ на секційному засіданні, публікація тез доповіді на тему *«Багатозалежні зв'язки у рекурсивній структурі фронтед додатку»* у збірнику матеріалів конференції.

5. VIII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення» (Україна, м. Житомир: «Житомирська політехніка», 02-03 грудня 2025 р.). Форма участі – очна, виступ на секційному засіданні, публікація тез доповіді на тему *«Глибоко вкладені структури даних із багатозалежними зв'язками: продуктивність та оновлення стану при рендерингу»* у збірнику матеріалів конференції.

Впровадження результатів наукового дослідження

VMKGROUP

ТОВАРИСТВО З ОБМЕЖЕНОЮ ВІДПОВІДАЛЬНІСТЮ «ВМК ГРУП ІОА»

Житомир, вул. Пушкінська, будинок 54, квартира 2, 10002

ЄДРПОУ 44862102, тел. +38 (097) 284 -05 -84

IBAN: UA42 3204780000026006924926910 в АБ «УКРГАЗБАНК»

ДОВІДКА

про впровадження результатів наукового дослідження

ВОЙТЮКА Олега Валерійовича

на тему «Моделі, методи та програмні засоби оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю»
на здобуття наукового ступеня доктора філософії
за спеціальністю 121 Інженерія програмного забезпечення
у галузі знань 12 Інформаційні технології

Дана довідка підтверджує, що теоретичні та практичні результати дисертаційного дослідження ВОЙТЮКА Олега Валерійовича на тему «Моделі, методи та програмні засоби оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю» проходили експериментальну перевірку в VMKGroup при оптимізації вебплатформи електронної комерції. Запропоновані алгоритми було інтегровано в систему відображення товарних каталогів з багаторівневою категоризацією: категорії, підкатегорії, бренди, характеристики, фільтри, відгуки та рекомендації. Тестування показало значне підвищення швидкості відображення каталогу та зменшення навантаження на клієнтську частину при роботі з деревами категорій глибиною до 12 рівнів. Покращення продуктивності інтерфейсу позитивно вплинуло на конверсію та отримало позитивні відгуки користувачів.

Ефективність роботи інтерфейсів e-commerce платформи безпосередньо впливає на користувацький досвід та комерційні показники. Розроблені ВОЙТЮКОМ О.В. методи оптимізації рендерингу складних структур даних демонструють високу практичну цінність для індустрії електронної комерції, що підтверджує актуальність і своєчасність проведеного дослідження.

Директор
ТОВ «ВМК ГРУП ІОА»



Максим ГОРОБЧУК

ДОВІДКА

про впровадження результатів дисертаційного дослідження
на тему «Моделі, методи та програмні засоби оновлення інтерфейсу
вебзастосунків на основі об'єктів з глибокою вкладеністю»

ВОЙТЮКА Олега Валерійовича

на здобуття наукового ступеня доктора філософії
за спеціальністю 121 Інженерія програмного забезпечення

Дана довідка підтверджує, що науково-практичні результати дисертаційного дослідження на здобуття наукового ступеня доктора філософії за спеціальністю 121 Інженерія програмного забезпечення ВОЙТЮКА О.В. були апробовані в ТОВ «АЙТІ ЛАБ СТУДІО» при розробці інтернет-продуктів для клієнтів. Розроблені програмні засоби оптимізації рендерингу застосовувалися в модулі відображення інформації для клієнтів, де необхідно обробляти глибоко вкладені структури: рахунки, категорії операцій, підкатегорії, теги, фільтри та аналітичні звіти. Впровадження запропонованих методів забезпечило суттєве скорочення часу завантаження та оновлення даних, що позитивно вплинуло на користувацький досвід та отримало схвальні відгуки від клієнтів компанії.

Швидкість та надійність роботи вебінтерфейсів є ключовими факторами задоволеності клієнтів та конкурентоспроможності установи. Результати дослідження ВОЙТЮКА О.В. демонструють високу ефективність у вирішенні проблем продуктивності при роботі з багаторівневими даними, що підтверджує їх практичну значущість та відповідність потребам сучасного бізнесу.

Директор
ТОВ «АЙТІ ЛАБ СТУДІО»



Ігор ТОЛСТОЙ

ДОВІДКА

про впровадження результатів дисертаційного дослідження
 аспіранта кафедри інженерії програмного забезпечення
 Державного університету «Житомирська політехніка»

ВОЙТЮКА Олега Валерійовича

на тему «Моделі, методи та програмні засоби оновлення інтерфейсу
 вебзастосунків на основі об'єктів з глибокою вкладеністю»
 на здобуття наукового ступеня доктора філософії
 за спеціальністю 121 Інженерія програмного забезпечення
 у галузі знань 12 Інформаційні технології

Дана довідка підтверджує, що отримані результати дисертаційного дослідження ВОЙТЮКА Олега Валерійовича впроваджено в ТОВ «Науковий парк Державного університету «Житомирська політехніка» при створенні інформаційних систем цифрового університету. Розроблені автором модель оптимізованого оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками, граф залежностей на рівні окремих значень даних як основного механізму поширення змін, що дозволяє визначати мінімальну множину уражених вузлів за рахунок використання проміжного рівня логіки у вигляді НОС-обгортки, метод вибіркового оновлення інтерфейсу вебзастосунків було адаптовано до інтегрованої веб-платформи управління освітніми процесами, яка оперує складними багаторівневими структурами даних (організаційна структура університету, освітні програми, навчальні плани, дисципліни, розклади, контингент студентів, академічні групи, індивідуальні навчальні траєкторії, електронне портфоліо, наукова діяльність та адміністративні процеси). Використання розроблених моделей, методів та алгоритмів забезпечило суттєве покращення продуктивності оновлення вебінтерфейсів при навігації по ієрархічних даних глибиною понад 9 рівнів та підвищило ефективності роботи адміністративних і науково-педагогічних працівників Державного університету «Житомирська політехніка» загалом.

Цифрова трансформація вищої освіти вимагає створення комплексних інформаційних систем, здатних ефективно обробляти великі обсяги структурованих взаємопов'язаних даних. Запропоновані ВОЙТЮКОМ О.В. моделі, методи та алгоритми демонструють високу практичну цінність для побудови продуктивних вебзастосунків цифрового університету, що свідчить про актуальність, практичну цінність та затребуваність отриманих наукових результатів у сфері освітніх технологій та цифровізації.

Директор
 ТОВ «Науковий парк
 Державного університету
 «Житомирська політехніка»



М.О. ПСЮК



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
 Ministry of Education and Science of Ukraine, Zhytomyr Polytechnic State University

вул. Чуднівська, 103, м. Житомир, 10005
 103, Chudnivska Str., Zhytomyr, Ukraine, 10005
 Phone./fax: (0412) 24-14-22, 24-14-23, e-mail: rector@ztu.edu.ua, https://ztu.edu.ua, код ЄДРПОУ 05407870
СИСТЕМА УПРАВЛІННЯ ЯКІСТЮ ВІДПОВІДАЄ ДСТУ ISO 9001:2015 та ДСТУ ISO 21001:2019
QUALITY MANAGEMENT SYSTEM ISO 9001:2015 and ISO 21001:2019

Від 01.04.2026 № 44-22.00/414
 На № _____ від _____

ДОВІДКА

про впровадження результатів дисертаційного дослідження
на тему «Моделі, методи та програмні засоби оновлення інтерфейсу
вебзастосунків на основі об'єктів з глибокою вкладеністю»

ВОЙТЮКА Олега Валерійовича

на здобуття наукового ступеня доктора філософії
 за спеціальністю 121 Інженерія програмного забезпечення
 у галузі знань 12 Інформаційні технології

Дана довідка підтверджує, що науково-теоретичні та практичні результати дисертаційного дослідження ВОЙТЮКА О. В., які полягають у розробці формальної моделі оптимізованого оновлення інтерфейсу вебзастосунків на основі об'єктів з глибокою вкладеністю та багатозалежними зв'язками; обґрунтуванні використання графа залежностей на рівні окремих значень даних як основного механізму поширення змін; удосконаленні поняття локальності оновлень у вебінтерфейсах, яке формалізовано через область впливу зміни та мінімальну множину DOM-вузлів, необхідних для коректного оновлення інтерфейсу; розробці методу вибіркового оновлення інтерфейсу вебзастосунків; побудові моделей та алгоритмів прототипу програмного додатку для динамічного оновлення інтерфейсу вебзастосунків та розробленого прототипу програмного забезпечення використовуються в освітньому процесі факультету інформаційно-комп'ютерних технологій Державного університету «Житомирська політехніка» при підготовці фахівців за спеціальністю F2 Інженерія програмного забезпечення, а саме викладанні освітньої компоненти «Вебтехнології», «Людино-машинний інтерфейс», «Backend-розробка», «Frontend-розробка» а також під час модернізації системи електронного навчання (LMS). Впровадження запропонованих програмних засобів забезпечило відчутне прискорення роботи інтерфейсу при навігації по складних курсах. Проведені експериментальні дослідження показали працездатність та ефективність запропонованих рішень.

Проректор з
 науково-педагогічної роботи



Андрій МОРОЗОВ

Звіт подібності

Звіт не був оцінений

Метадані

ДОКУМЕНТ

Заголовок

Войтюк дисертація 07_04_2026

Автор

Олег Войтюк

Науковий керівник / Експерт

Zhytomyr Polytechnic

ІД документа

333547177

ОРГАНІЗАЦІЯ

Назва організації

State University "Zhytomyr Polytechnic"

підрозділ

State University "Zhytomyr Polytechnic"

ЗВІТ

Дата звіту

4/7/2026

Дата редагування

4/7/2026

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.73%

9.86%

КП 1

0.00%

8.16%

КП 2

25

23746

189313

Довжина фрази для коефіцієнта подібності 2

Кількість слів

Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		16
Інтервали		0
Мікропробіли		55
Білі знаки		0
Парафрази (SmartMarks)		156

Джерела

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз

Колір тексту

#	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА ГАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://www.tech-vernadskyjournals.in.ua/journals/2025/5_2025/part_2/13.pdf	561 (2.36 %)

2	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	403 (1.70 %)
3	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	217 (0.91 %)
4	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	150 (0.63 %)
5	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	124 (0.52 %)
6	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	87 (0.37 %)
7	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	87 (0.37 %)
8	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	79 (0.33 %)
9	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	63 (0.27 %)
10	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	52 (0.22 %)
Програма обміну базами даних (0.00 %)		
#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИФІКАЦІЙНИХ СЛІВ (ФРАГМЕНТІВ)
Інтернет (9.06 %)		
#	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИФІКАЦІЙНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://www.tech.vernadskyjournals.in.ua/journals/2025/6_2025/part_2/13.pdf	1977 (16) (8.33 %)
2	https://pluginhighway.ca/blog/recursion-without-accumulator-a-powerful-technique-for-simplifying-code-and-improving-efficiency	21 (2) (0.09 %)
3	https://wp.kpi.ua/files/arf/2017/%D0%9F%D1%80%D0%B8%D1%82%D0%B8%D1%81%D0%BA%D0%B0%D1%87_%20%D0%90%D0%B2%D1%82%D0%BE%D1%80%D0%B5%D1%84%D0%B5%D1%80%D0%B0%D1%82.pdf	20 (1) (0.08 %)
4	http://academy.gov.ua/IMKD/library_nadru/Vydromferat/de7ee2db_94e4eca8aad_e40e23b641ee.pdf	18 (3) (0.08 %)
5	https://linfo.library.com/711	16 (2) (0.07 %)
6	https://www.stud24.eu/marketing/rozvidenie-sistemoj-marketingovyh-kommunikacij-turisticheskij/443853-1671825-page1.html	14 (1) (0.06 %)
7	https://medium.com/@wekly-wa/https-recurshy-finding-the-nth-fibonacci-number-55ebb1c8bb5	14 (1) (0.06 %)
8	https://kpi.ua/bitstreams/9fd4692-9244-4b86-92e4-c2b23979e25/download	13 (1) (0.05 %)
9	https://zlu.edu.ua/site/files/1/docs/%D0%9D%D0%B0%D1%83%D0%BA%D0%B0%D0%90%D1%81%D0%BE%D1%86%D1%80%D0%B0%D0%BD%D1%82%D0%B0%D0%B4%D0%BE%D0%BA%D1%82%D0%BE%D1%80%D0%B0%D0%BD%D1%82%D1%83%D1%80%D0%B0%D0%A0%D0%B0%D0%B7%D0%BE%D0%B2%D1%86%D0%B0%D0%B4%D0%B8/2025/%D0%9B%D0%B8%D1%81%D0%B5%D0%BD%D0%BA%D0%BE_%D0%92%D0%86%D0%94%D0%93%D0%A3%D0%9A_%D0%9F%D0%B0%D0%B2%D0%BB%D0%BE%D0%B2%D0%A1%D0%B5%D1%80%D0%B3%D1%86%D0%B8%CC%86.pdf	12 (1) (0.05 %)
10	https://old.zlu.edu.ua/site/files/0nslife/vysnovok_Sydornov.pdf	11 (1) (0.05 %)
11	https://www.konf1.zzz.com.ua/1-mizhnarodna-naukovo-praktichna-konferentsiya-topical-aspects-of-modern-scientific-research-28-30-06-2023-tokio-yaponiya/index-1.htm	10 (1) (0.04 %)
12	http://ena.jp.edu.ua:8080/bitstream/nbf/13375/1/arf_%D0%9C%D1%83%D1%85%D0%B0.doc	7 (1) (0.03 %)
13	https://lib.iltta.gov.ua/d/eprint/710849/1/arf_Antonuk_Printed.pdf	7 (1) (0.03 %)

14	https://cyberleninka.ru/article/n/ustchena-otsinka-parametrv-aditivnyy-sumishi-radikalny-ta-ekstremnyy-negativnyy-zavyad	6 (1) (0.03 %)
15	https://ism.knu.ua/sites/default/files/awards/serafim_markiv_o_o.pdf	5 (1) (0.02 %)

Список прийнятих фрагментів

#	ЗМІСТ	КІЛЬКІСТЬ ОДНАРОДНИХ СРІВ (ФРАГМЕНТІВ)
	https://www.tech.vernadskyjournals.in.ua/journal... 	1977 (8.33%)