Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження ефективності різних підходів до розробки форм у React
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-2

Олександр ДУДНИК
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Віктор КАУК
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Дуднику Олександру Олеговичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження ефективності різних підходів до розробки форм у React»

Затверджена наказом по університету від 15.04.2025р. №290Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20.06.2025

3. Вихідні дані до роботи методичні вказівки до виконання кваліфікаційної роботи магістра, офіційна документація React, документація бібліотек для створення форм, методології та принципи розробки користувацьких інтерфейсів, методики оцінки продуктивності веб-застосунків.

4. Перелік питань, що потрібно опрацювати в роботі
аналіз предметної галузі, огляд й аналіз літературних та наукових джерел, постановка задачі, опис прийнятих проектних рішень, планування експерименту, опис програмної реалізації, опис експериментальних досліджень, аналіз отриманих результатів.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	21.04.2025	*виконано*
3	Аналіз літературних джерел	26.04.2025	*виконано*
4	Вибір підходів для дослідження	01.05.2025	*виконано*
5	Планування експерименту	07.05.2025	*виконано*
6	Програмна реалізація тестових застосунків	16.05.2025	*виконано*
7	Проведення експерименту	23.05.2025	*виконано*
8	Аналіз результатів експерименту	29.05.2025	*виконано*
9	Підготовка до апробації результатів дослідження. Публікація матеріалів	03.06.2025	*виконано*
10	Підготовка пояснювальної записки	09.06.2025	*виконано*
11	Підготовка презентації та доповіді	11.06.2025	*виконано*
12	Перевірка на плагіат	12.06.2025	*виконано*
13	Нормоконтроль	12.06.2025	*виконано*
14	Рецензування	13.06.2025	*виконано*
15	Попередній захист	14.06.2025	*виконано*
16	Занесення диплома в електронний архів	19.06.2025	*виконано*
17	Допуск до захисту у зав. кафедри	20.06.2025	*виконано*

Дата видачі завдання 16 квітня 2025р.

Студент (ка) _____ Олександр ДУДНИК
(підпис)

Керівник роботи _____ доц. Віктор КАУК
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 145 с., 10 рис., 3 табл., 22 джерела.

ВЕБ-ФОРМИ, КОРИСТУВАЦЬКИЙ ІНТЕРФЕЙС, ФРОНТ-ЕНД, FINAL FORM, FORMIK, REACT, REACT HOOK FORM, REDUX FORM.

Об’єктом дослідження є розробка форм у React, інструменти та підходи до їх створення.

Предметом дослідження є ефективність різних підходів до створення форм у React.

Метою роботи є виявлення найбільш ефективного підходу до створення форм у React.

Методи дослідження включають: теоретичний аналіз (аналіз предметної галузі, аналіз наукових публікацій за темою), проведення експериментів (створення тестових проєктів для оцінки ефективності різних підходів), порівняльний аналіз (узагальнення результатів експериментів, порівняння підходів за визначеними критеріями шляхом вирішення багатокритеріальної задачі прийняття рішень).

У результаті роботи було розроблено п’ять програм з використанням різних підходів до розробки форм: з використанням «чистого» React, бібліотеки Formik, бібліотеки React Hook Form, бібліотеки Final Form та бібліотеки Redux Form.

FINAL FORM, FORMIK, FRONT-END, REACT, REACT HOOK FORM, REDUX FORM, USER INTERFACE, WEB FORMS.

The object of research is the development of forms in React, tools and approaches to their creation.

The subject of research is the effectiveness of different approaches to creating forms in React.

The aim of the work is to identify the most effective approach to creating forms in React.

Research methods include: theoretical analysis (analysis of the subject area, analysis of scientific publications on the topic), conducting experiments (creating test projects to evaluate the effectiveness of different approaches), comparative analysis (summarizing experimental results, comparing approaches according to defined criteria by solving a multi-criteria decision-making problem).

As a result of the work, five programs were developed using different approaches to form development: using "pure" React, Formik library, React Hook Form library, Final Form library, and Redux Form library.

Завідувачу кафедри
ПІ_____
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, сласне ім’я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElArKhNURE

Я, Дудник Олександр Олегович, здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-2, кафедра програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження ефективності різних підходів до розробки форм у React», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії «ElArKhNURE». Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ «ElArKhNURE». Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

20.06.2025

ЗМІСТ

ВСТУП

У сучасному світі веб-розробка є однією з найдинамічніших галузей інформаційних технологій. Серед численних інструментів для створення веб-застосунків бібліотека React займає провідне місце завдяки своїй гнучкості, продуктивності та активній спільноті розробників.

Одним із важливих аспектів створення сучасних веб-застосунків є розробка ефективних форм для збору, перевірки та обробки даних користувачів. Форми є основою взаємодії між користувачами та системами: від реєстрації та входу до системи до складних бізнес-процесів, таких як оформлення замовлень чи заповнення анкет. Правильний вибір підходу та інструментів для створення форм може суттєво вплинути на якість кінцевого продукту, швидкість розробки та підтримки коду. У цьому контексті постає питання вибору оптимального підходу до розробки форм, який би відповідав потребам конкретного проєкту.

Існує широкий спектр бібліотек і підходів для створення форм у React, таких як Formik, React Hook Form, Final Form та інші. Кожна з них має свої переваги й недоліки, які впливають на продуктивність, зручність використання та підтримку коду. Дослідження ефективності цих бібліотек є важливим для розробників, оскільки дозволяє обрати найкраще рішення для конкретних завдань, зменшити витрати часу на розробку та підвищити якість кінцевого продукту.

Отже, актуальність дослідження ефективності різних підходів до розробки форм у React зумовлена необхідністю надання чітких рекомендацій для вибору бібліотеки або підходу до розробки форм у React, що сприятиме підвищенню ефективності веб-розробки та поліпшенню взаємодії з користувачами.

Головною метою цього дослідження є виявлення найбільш ефективного підходу до створення форм у React залежно від специфіки завдань і потреб проєкту. Головні задачі які необхідно виконати для досягнення цієї мети включають: аналіз існуючих підходів до розробки та виявлення найпопулярніших з них, проведення експериментальних досліджень, проведення на основі отриманих результатів порівняльного аналізу, визначення найбільш ефективного підходу.

Об’єктом дослідження є розробка форм у React як частина процесу веб-розробки.

Предметом дослідження є ефективність різних підходів до створення форм у React.

Методи дослідження включають: теоретичний аналіз (дослідження предметної галузі, аналіз наукових публікацій), експериментальний метод (створення тестових проєктів для оцінки ефективності різних підходів), порівняльний аналіз (узагальнення результатів експериментів, порівняння підходів за визначеними критеріями).

Отримані результати можуть бути корисні front-end розробникам та командам розробки, та можуть слугувати орієнтиром при виборі оптимальних рішень для реалізації форм у своїх React-проєктах, що в кінцевому підсумку сприяє підвищенню ефективності розробки та якості веб-додатків.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі дослідження

Із кожним роком кількість веб-сайтів та веб-додатків продовжує стрімко зростати. За даними опитування веб-серверів Netcraft у вересні 2024 року, в інтернеті налічувалося понад 193 мільйонів активних веб-сайтів, причому щодня з’являється близько 252 тисяч нових [1]. Ця статистика демонструє масштаби та динаміку розвитку веб-технологій.

Спектр цих веб-сайтів надзвичайно різноманітний, та варіюється від простих сайтів-візитівок до складних корпоративних систем та інтерактивних веб-сервісів. Але, незалежно від їх типу, призначення та складності, майже кожен з них містить форми як ключовий елемент взаємодії з користувачем. Форми є фундаментальним компонентом сучасних веб-сайтів, що забезпечує двосторонню комунікацію між користувачем та системою. Вони використовуються для реєстрації та авторизації користувачів, оформлення замовлень в інтернет-магазинах, відправлення заявок у різні служби, проведення опитувань та анкетувань, пошуку інформації тощо. Без форм інтернет так і залишився би лише одностороннім каналом передачі інформації, без можливості повноцінної інтерактивної взаємодії.

Враховуючи центральну роль форм у взаємодії з користувачем, їх якість є одним з факторів, який безпосередньо впливає на загальне враження від веб-ресурсу [2]. Вони повинні бути гарними, зручними та швидкими. Погано реалізовані форми можуть суттєво погіршити користувацький досвід: повільна валідація полів дратує користувачів та змушує їх чекати, втрата введених даних через технічні помилки викликає розчарування, проблеми з відображенням полів заважають вносити дані. Особливо критичними ці проблеми стають у довгих формах, таких як оформлення замовлення чи заповнення анкети, де користувач витрачає значний час на введення інформації.

При цьому, масштабність та динаміка розвитку веб-технологій створює певні виклики для розробників. В умовах, коли кількість проектів та їх складність постійно зростає, а терміни їх реалізації скорочуються, критично важливим стає наявність ефективних інструментів та підходів до розробки. Ці інструменти

повинні забезпечувати швидкість розробки та бути зручними для розробника, але при цьому, вони повинні також підтримувати високу якість кінцевого продукту.

Одним з важливих аспектів у цьому контексті є розробка форм, що потребує особливої уваги через їх складність та роль у користувацькій взаємодії. Сучасні форми часто містять десятки полів різних типів, складну логіку валідації, динамічну поведінку, залежності між полями та асинхронні взаємодії з сервером. При цьому вони повинні залишатися продуктивними, надійними та зручними для користувачів.

На початку розвитку веб-технологій розробка здійснювалася за допомогою базових засобів HTML та JavaScript, що вимагало написання значної кількості шаблонного коду та ускладнювало створення складних інтерактивних інтерфейсів. З розвитком веб-технологій з'явилися різноманітні фреймворки та бібліотеки, які пропонують більш ефективні підходи до розробки користувацьких інтерфейсів загалом та форм зокрема. Ці інструменти надають розробникам готові компоненти, зручні засоби управління станом додатку та механізми для створення складної інтерактивної логіки.

Серед різноманітних фреймворків та бібліотек для розробки веб-застосунків особливе місце займає React [3] - відкрита JavaScript-бібліотека, представлена компанією Facebook (зараз Meta) у 2013 році. React швидко став одним із найпопулярніших інструментів для створення користувацьких інтерфейсів і наразі використовується мільйонами розробників та компаній по всьому світу.

Передумовою створення React стала потреба Facebook у більш ефективному інструменті для розробки складних інтерфейсів. Традиційні підходи до розробки веб-додатків не справлялися з викликами, які ставила зростаюча складність користувацьких інтерфейсів соціальної мережі [4].

Поява React.js різко розширила можливості front-end розробників у створенні зручних інтерфейсів.

Ключовою інновацією React стало впровадження концепції віртуального DOM (Document Object Model) та декларативного підходу до створення інтерфейсів. Замість прямої маніпуляції елементами веб-сторінки, React

використовує віртуальне представлення DOM, що дозволяє ефективно визначати необхідні зміни та застосовувати їх пакетно. Це суттєво підвищує продуктивність додатків, особливо у випадках з частими оновленнями інтерфейсу.

Інша важлива особливість React - компонентний підхід до розробки. Весь інтерфейс розбивається на незалежні, повторно використовувані компоненти, кожен з яких має власну логіку та стан. Це дозволяє створювати складні інтерфейси з простих будівельних блоків, спрощує підтримку коду та забезпечує можливість повторного використання компонентів у різних частинах додатку або навіть у різних проектах.

Ще одним важливим фактором популярності React є його екосистема - велика спільнота розробників, численні бібліотеки та інструменти, що розширюють базову функціональність. Ця екосистема надає готові рішення для типових завдань розробки, включаючи управління станом додатку, маршрутизацію, роботу з формами та інші аспекти створення веб-додатків [5].

Базовий функціонал React надає всі необхідні інструменти для створення форм: можливість управління станом компонентів через useState, обробку подій введення даних, валідацію полів тощо. Такий підхід, який часто називають «чистим React» (Pure React), дозволяє створювати форми будь-якої складності, забезпечуючи повний контроль над їх поведінкою та максимальну гнучкість.

Однак при розробці форм з використанням лише базового функціоналу React розробники часто стикаються з необхідністю написання значної кількості шаблонного коду. Для кожного поля форми потрібно створювати окремий стан, писати обробники подій, реалізовувати валідацію, обробляти помилки та забезпечувати коректну роботу з сервером. У простих формах з декількома полями це не становить проблеми, але при розробці складних форм з десятками полів, залежностями між ними та комплексною валідацією, такий підхід може призводити до збільшення об'єму коду, його складності та, як наслідок, ймовірності помилок.

Ці обмеження «чистого» підходу спонукали спільноту розробників до створення спеціалізованих бібліотек та рішень для роботи з формами в React. Такі інструменти пропонують різні підходи до спрощення процесу розробки форм,

автоматизації рутинних операцій та забезпечення надійності роботи з даними форм. При цьому кожен з цих підходів має свої особливості, переваги та потенційні обмеження, що впливає на їх ефективність у різних сценаріях використання.

Аналіз різних джерел (наприклад: спільнота «DEV Community» [6], сайт «WeAreDevelopers» [7], сайт «GeekFlare» [8]) дозволяє виділити кілька бібліотек, які найчастіше згадуються як одні з найефективніших рішень для роботи з формами в React. Це бібліотеки «Formik», «React Hook Form», «Final Form» та «Redux Form». Ці бібліотеки мають велику кількість завантажень на npm, та отримують високі оцінки розробників за їх функціональність, продуктивність та зручність використання.

Formik [9] є однією з найпопулярніших бібліотек для роботи з формами в React. Її основна перевага полягає у наданні простого та декларативного API для роботи з формами. Бібліотека бере на себе всі рутинні завдання: управління станом форми, обробку подій, валідацію та обробку помилок. Formik надає набір готових компонентів, які можна використовувати для швидкої розробки форм, але при цьому зберігає достатню гнучкість для кастомізації поведінки форми. Особливістю Formik є її тісна інтеграція з бібліотекою Yup для валідації даних, що спрощує процес визначення правил валідації полів форми.

React Hook Form [10] представляє більш сучасний підхід до розробки форм, базуючись на концепції React Hooks. Ця бібліотека відрізняється високою продуктивністю завдяки мінімізації повторних рендерингів компонентів. React Hook Form використовує неконтрольовані компоненти та нативну валідацію браузера, що позитивно впливає на швидкодію. Бібліотека надає потужний API для валідації, інтеграції з TypeScript та роботи з різними UI-бібліотеками. Особливу увагу в React Hook Form приділено оптимізації продуктивності та зменшенню розміру бандла.

Final Form [11] є незалежною від фреймворків бібліотекою для роботи з формами, яка має офіційну інтеграцію з React. Ключовою особливістю цієї бібліотеки є її модульна архітектура та висока продуктивність. Final Form використовує підписку на зміни для оновлення тільки тих компонентів, які дійсно

потребують оновлення. Бібліотека надає широкі можливості для валідації, включаючи асинхронну валідацію та валідацію на рівні форми. Final Form також підтримує складні сценарії використання, такі як масиви полів та вкладені форми.

Redux Form [12] була однією з перших популярних бібліотек для роботи з формами в React і тісно інтегрується з Redux - популярним рішенням для управління станом додатку. Бібліотека зберігає стан форми в Redux store, що може бути корисним у великих додатках, де потрібна централізована обробка даних форм. Redux Form надає широкі можливості для валідації, форматування введення та обробки подій форми. Однак, через зберігання стану форми в Redux store, ця бібліотека може призводити до надмірних оновлень компонентів та зниження продуктивності у великих формах.

Кожна з цих бібліотек має свої переваги та обмеження, що впливає на їх вибір для конкретних проектів.

Окремо варто відзначити Yup [13] - бібліотеку валідації, яка часто використовується разом з вищезгаданими рішеннями. Yup надає декларативний спосіб опису схем валідації, що робить процес валідації форм більш структурованим та передбачуваним. Інтеграція Yup з бібліотеками форм дозволяє створювати складні правила валідації з мінімальними затратами коду.

В останні роки також набирає популярності бібліотека Zod [14], яка пропонує потужну систему валідації з першокласною підтримкою TypeScript. На відміну від Yup, Zod робить більший акцент на типобезпеці та може використовуватися не лише для валідації форм, але й для валідації даних на рівні всього додатку.

1.2 Огляд й аналіз літературних, наукових джерел

Перед початком роботи за темою було проведено ретельний пошук та відбір джерел.

Основною проблемою, з якою довелось зіткнутися при пошуку літератури, є відсутність матеріалів, що стосуються безпосередньо аналізу підходів до створення форм у React та порівняння бібліотек для створення форм у React. Тому до розгляду були включені також роботи, які охоплюють суміжні теми:

- аналіз та порівняння бібліотек для React;
- аналіз та порівняння бібліотек загалом;
- загальні підходи до порівняння ефективності програмних рішень;
- методи вимірювання продуктивності та ефективності;
- методи профілювання продуктивності у React.

Вибір джерел ґрунтувався на чотирьох ключових критеріях: авторитетність, актуальність, об’єктивність та достовірність.

Авторитетність: у якості джерел обирались статті опубліковані у авторитетних наукових виданнях та конференційних збірниках присвячених інформаційним технологіям, та наукові праці людей, що мають академічні досягнення у сфері інформаційних технологій.

Актуальність: обирались джерела інформації опубліковані упродовж останніх 10 років, однак перевага надавалась найбільш новим матеріалам, більшості з яких не більше 3 років. Це забезпечує врахування сучасного стану технологій.

Об’єктивність: для вивчення обиралися матеріали, які базуються на конкретних даних та результатах досліджень, а не суб'єктивних уподобаннях авторів. Перевага віддавалася матеріалам з демонстрацією практичних прикладів, графіків або інших вимірюваних показників.

Достовірність: обирались джерела, що містять перевірені дані та аргументовані висновки, що ґрунтуються на дослідженнях та детальному аналізі.

В результаті було сформовано добірку джерел, для подальшого аналізу.

У роботі Ііда Каіну «Optimization in React.js: Methods, Tools, and Techniques to Improve Performance of Modern Web Applications» [15] розглядаються методи оптимізації веб-застосунків, створених з використанням React.js. У цій роботі нас зацікавили розглянуті питання профілювання продуктивності застосунків на React.

Робота Санни Салонен «Evaluation of UI Component Libraries in React Development» [16] присвячена оцінці UI-бібліотек компонентів для React, зокрема визначенню критеріїв їх порівняння та вибору оптимального варіанту для різних проєктів. У роботі проаналізовано три популярні бібліотеки за п'ятьма ключовими

категоріями. Для нашої роботи це джерело корисне тим, що пропонує структурований підхід до оцінки інструментів розробки.

У тезах Фернандо Лопеса де ла Мора та Сари Наді «Which library should I use? A metric-based comparison of software libraries» [17] детально розглядається методика метрико-орієнтованого порівняння програмних бібліотек. Для нашого дослідження особливо цінним є запропонований підхід до систематичного порівняння бібліотек за різними метриками, що, безумовно, можна застосувати для оцінки бібліотек React форм.

У роботі Хаті Чхетрі «Comparative Study of Front-end Frameworks : React and Angular» [18] представлено комплексний порівняльний аналіз React та Angular. Для нашої роботи особливо цінним є методологія порівняння фреймворків та детально розроблена система критеріїв оцінювання, яку можна адаптувати для порівняння бібліотек форм у React.

У статті Ґуршина Каура та Раджі Ґауранґа Тіварі «Comparison and Analysis of Popular Frontend Frameworks and Libraries: An Evaluation of Parameters for Frontend Web Development» [19] розглядається порівняльний аналіз популярних фронтенд-фреймворків та бібліотек на основі ключових параметрів оцінки. Для нашого дослідження цінними є визначені критерії оцінювання технологій, які можна застосувати для порівняння бібліотек форм у React.

У статті Фані Секхара «Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development» [20] проводиться порівняльний аналіз трьох провідних фронтенд-фреймворків. Для нашого дослідження цінною є запропонована методологія оцінки технологій, яку можна адаптувати для порівняння бібліотек форм у React.

У роботі Дарії Проніної та Ірини Кириченко «Comparison of Redux and React Hooks Methods in Terms of Performance» [21] ретельно проведено порівняльний аналіз двох підходів до управління станом у React-застосунках: Redux та React Hooks. У цій роботі нас цікавлять розроблені метрики та методологія оцінки ефективності різних підходів до розробки, що може бути застосовано для порівняння бібліотек форм.

Розглянемо зміст обраної літератури більш детально та проаналізуємо основні положення кожної з представлених робіт.

У роботі «Optimization in React.js: Methods, Tools, and Techniques to Improve Performance of Modern Web Applications» детально розглядається тема профілювання продуктивності застосунків на React та описуються ключові інструменти для її вимірювання. Ці інструменти включають:

- Chrome DevTools Lighthouse - для комплексного аналізу продуктивності веб-додатку;
- React Developer Tools Profiler - для відстеження рендерингу компонентів;
- React.js Profiler Component - для програмного профілювання продуктивності компонентів.

Ці інструменти дозволяють виміряти та проаналізувати ефективність різних підходів до розробки, що є критичним для порівняльного аналізу бібліотек форм.

У роботі «Evaluation of UI Component Libraries in React Development» наводяться корисні критерії для оцінки UI-бібліотек компонентів, які можна адаптувати для оцінки бібліотек для розробки форм. Автор пропонує наступні критерії для порівняння бібліотек:

- компоненти та ресурси - розглядається різноманітність та гнучкість функціональних можливостей бібліотеки. У контексті форм це може включати наявність готових компонентів для різних типів полів введення, можливості валідації та обробки даних.
- документація - розглядається наявність якісної документації з детальними прикладами використання, що буде особливо актуально при роботі з формами, де часто потрібна інтеграція складних функцій валідації та обробки даних.
- сумісність - розглядаєються можливості використання бібліотеки в різних контекстах. Для форм це особливо важливо з точки зору адаптивності до різних пристроїв, доступності та можливості інтеграції з іншими інструментами React-екосистеми.

- обмеження - аналізуються технічні та загальні обмеження бібліотек. У контексті форм це може включати обмеження щодо кастомізації, продуктивності при роботі з великими формами та можливостей розширення функціоналу.
- надійність - фокусується на активності розробки та підтримці спільноти. Цей аспект особливо важливий для вибору бібліотеки форм, оскільки стабільність та регулярні оновлення гарантують довгострокову підтримку та вирішення потенційних проблем.

У тезах «Which library should I use? A metric-based comparison of software libraries» представлено комплексний підхід до порівняння програмних бібліотек, який може бути адаптований для нашого аналізу бібліотек React форм. Зокрема, корисною для нашого дослідження є методологія оцінки бібліотек за наступними метриками:

- популярність (кількість проєктів, що використовують бібліотеку);
- частота оновлень (як часто виходять нові версії);
- час відповіді на проблеми та їх закриття (показник активності підтримки);
- зворотна сумісність (наявність критичних змін між версіями);
- схильність до помилок (кількість виправлених багів);
- продуктивність та безпека (на основі повідомлених проблем).

Ця методологія може бути безпосередньо застосована для порівняльного аналізу бібліотек форм у React, що дозволить провести їх об'єктивну оцінку за визначеними критеріями.

У роботі «Comparative Study of Front-end Frameworks : React and Angular» пропонується комплексна система критеріїв оцінювання, яку ми можемо адаптувати для порівняння бібліотек форм:

- крива навчання: Цей критерій охоплює комплексну оцінку того, наскільки легко новому розробнику почати роботу з бібліотекою. Він враховує не лише якість початкової документації, але й загальну інтуїтивність архітектури бібліотеки.

- продуктивність: У рамках цього критерію проводиться детальний аналіз технічних характеристик бібліотеки. Оцінюється швидкість початкового завантаження компонентів, ефективність їх роботи при взаємодії з користувачем, оптимізація розміру фінального бандлу.
- екосистема та інструменти: Цей критерій досліджує загальну розвиненість інфраструктури навколо бібліотеки. Розглядається наявність додаткових інструментів та бібліотек, які спрощують розробку, можливості для тестування компонентів, а також інтеграція з іншими популярними інструментами розробки.
- підтримка спільноти та документація: В межах цього критерію аналізується якість та доступність інформаційних ресурсів. Оцінюється не лише повнота офіційної документації та регулярність її оновлення, але й наявність додаткових навчальних матеріалів від спільноти.
- популярність: Цей критерій відображає загальне прийняття бібліотеки в професійній спільноті. Він включає аналіз розміру активної спільноти розробників, попит на фахівців на ринку праці та доступність готових рішень.

У статті «Comparison and Analysis of Popular Frontend Frameworks and Libraries: An Evaluation of Parameters for Frontend Web Development» наведено систему параметрів для оцінки фронтенд-технологій, яку ми можемо адаптувати для порівняння бібліотек форм:

- популярність та підтримка спільноти - цей критерій дозволяє оцінити активність розробки бібліотеки, швидкість виправлення помилок та наявність готових рішень типових проблем;
- документація - якість та повнота документації є критичним фактором для оцінки зручності використання бібліотеки форм;
- продуктивність - цей параметр важливий для оцінки швидкодії різних підходів до валідації та обробки даних форм;
- масштабованість - можливість ефективно працювати зі складними формами та великою кількістю полів.

Ця система критеріїв надає структуровану основу для порівняльного аналізу різних бібліотек форм у React та дозволяє об'єктивно оцінити їх переваги та недоліки.

У статті «Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development» автор використовує систематизований підхід до оцінювання Angular, React і Vue.js, зосереджуючись на таких критеріях:

- продуктивність - вимірюється через час завантаження, ефективність виконання та швидкість реакції.
- продуктивність розробників - оцінюється на основі опитувань і відгуків розробників про зручність використання, криву навчання та швидкість розробки.
- екосистема та підтримка спільноти - враховується доступність сторонніх інструментів і бібліотек, а також активність розробників у спільноті.
- крива навчання та документація - аналізується якість документації та зручність для новачків.

Ці критерії можуть бути адаптовані для оцінювання бібліотек форм у React. Наприклад, продуктивність можна виміряти через швидкість обробки форм і роботи з валідацією, а підтримку екосистеми – через наявність сторонніх розширень або інтеграцій.

У роботі «Comparison of Redux and React Hooks Methods in Terms of Performance» детально описано комплексний підхід до оцінки ефективності різних методів розробки React-застосунків. Автори пропонують набір ключових метрик для порівняння:

- відсоткова різниця в обсязі коду при імплементації базових методів для першої сутності проекту;
- відсоткова різниця в обсязі коду при імплементації методів для наступних сутностей;
- відсоток повторно використовуваного коду;
- прогнозований коефіцієнт зростання обсягу коду при додаванні нових сутностей;

- різниця у розмірі коду управління станом у production-збірці;
- різниця у використанні пам'яті при зберіганні однакового обсягу даних;
- різниця у часі операцій оновлення стану.

Ці метрики дозволяють всебічно оцінити ефективність різних підходів до розробки як з точки зору продуктивності, так і з точки зору підтримки коду в довгостроковій перспективі. Особливо важливим для нашого дослідження є вимірювання використання пам'яті та часу виконання операцій, оскільки ці показники безпосередньо впливають на продуктивність форм у React-застосунках у реальних умовах експлуатації. Додатково, метрики пов'язані з обсягом коду допоможуть оцінити складність підтримки різних бібліотек форм у довгостроковій перспективі.

Проаналізовані джерела демонструють високу актуальність для дослідження ефективності різних підходів до розробки форм у React, оскільки охоплюють широкий спектр аспектів розробки сучасних веб-застосунків та методології їх оцінювання.

Актуальність розглянутих джерел підтверджується такими ключовими факторами, як часова релевантність, технологічна відповідність та методологічна цінність.

Часова релевантність: Більшість проаналізованих робіт опубліковано протягом останніх трьох років, що забезпечує актуальність інформації в контексті швидкого розвитку фронтенд-технологій. Особливо важливим є те, що роботи охоплюють період активного розвитку React та його екосистеми, включаючи впровадження та становлення React Hooks, що суттєво вплинуло на підходи до розробки форм.

Технологічна відповідність: Розглянуті джерела охоплюють сучасні аспекти розробки React-застосунків, включаючи оптимізацію продуктивності, управління станом та роботу з компонентами. Це особливо актуально для дослідження форм, оскільки вони часто є критичними елементами веб-застосунків з точки зору продуктивності та користувацького досвіду.

Методологічна цінність: Представлені в джерелах методології оцінки та порівняння технологій є актуальними для сучасного стану розробки веб-застосунків. Зокрема, роботи пропонують систематичні підходи до оцінки продуктивності, масштабованості та зручності використання, що безпосередньо застосовно до аналізу бібліотек форм.

Проведений аналіз літературних джерел дозволяє зробити ряд важливих висновків щодо поточного стану досліджень у сфері розробки форм у React та методології їх оцінювання.

Існує загальна методологічна база для оцінки ефективності програмних рішень та бібліотек, яка може бути адаптована для порівняння підходів до розробки форм. Наявні дослідження в галузі фронтенд-розробки пропонують всебічні та добре структуровані комплексні системи критеріїв для оцінювання фронтенд-технологій з урахуванням специфіки різних проектних завдань. Існують апробовані методи вимірювання продуктивності React-застосунків, які можуть бути використані для об'єктивної оцінки ефективності різних підходів до розробки форм.

Водночас аналіз виявив суттєві прогалини у існуючих дослідженнях.

Відсутні комплексні дослідження, присвячені безпосередньо порівнянню різних підходів та бібліотек для розробки форм у React. Наявні роботи зосереджені на загальному порівнянні фреймворків або окремих аспектах розробки.

Виявлені прогалини обґрунтовують необхідність проведення комплексного дослідження ефективності різних підходів до розробки форм у React. Таке дослідження має включати:

- розробку специфічних метрик для оцінки ефективності форм;
- створення тестових сценаріїв для порівняння різних підходів;
- аналіз продуктивності при роботі з формами різної складності;
- вивчення впливу різних підходів на загальну ефективність застосунку.

Результати такого дослідження дозволять розробникам приймати більш обґрунтовані рішення при виборі підходів та бібліотек для розробки форм у React-застосунках.

1.3 Постановка задачі

Головна мета цього дослідження полягає у виявленні найбільш ефективного підходу до розробки форм у React.

Для досягнення цієї мети необхідно виконати такі задачі:

- проаналізувати існуючі підходи до розробки форм у React та виявити найбільш популярні з них для подальшого порівняння;
- визначити критерії для оцінки та порівняння різних підходів;
- розробити тестові приклади форм, використовуючи обрані підходи;
- провести експериментальне дослідження та зібрати кількісні та якісні показники ефективності кожного з підходів;
- визначити найбільш ефективний підхід шляхом вирішення багатокритеріальної задачі прийняття рішень.

Методи дослідження включають:

а) теоретичний аналіз:
   - аналіз предметної галузі;
   - аналіз наукових публікацій та технічних статей за темою;
   - вивчення технічної документації React та досліджуваних бібліотек для розробки форм.

б) експериментальне дослідження:
   - створення тестових форм з використанням різних підходів;
   - збір метрик продуктивності, таких як час рендерингу, обсяг пам'яті, що використовується, та швидкість валідації даних.

в) багатокритеріальний аналіз:
   - формування критеріїв для оцінки підходів;
   - порівняння підходів за визначеними критеріями шляхом вирішення багатокритеріальної задачі прийняття рішень.

Слід зазначити, що дослідження має певні обмеження, аналіз яких представлено нижче.

Часові обмеження для проведення дослідження можуть впливати на глибину аналізу та можливість всебічного аналізу досліджуваних підходів до розробки форм.

Дослідження відображає стан розвитку технологій та підходів до розробки форм у React станом на 2024 рік. Враховуючи динамічний характер розвитку веб-технологій, отримані результати та висновки можуть втратити актуальність з появою нових версій бібліотек та зміною загальноприйнятих практик розробки.

Дослідження фокусується виключно на екосистемі React і не розглядає рішення для інших фреймворків.

Дослідження обмежене аналізом найбільш поширених підходів до розробки форм у React, оскільки повне охоплення усіх існуючих бібліотек та методологій не є можливим через їх значну кількість та постійний розвиток екосистеми React.

Дослідження може бути обмежене певним набором метрик ефективності, не враховуючи всі можливі аспекти.

Дослідження носить аналітичний характер та фокусується на аналізі існуючих підходів до розробки форм у React, не передбачаючи розробку нових методологій чи технічних рішень.

Результати дослідження надають узагальнену оцінку ефективності підходів до розробки форм, що не враховує специфічні вимоги та особливості окремих проєктів. Отримані висновки можуть потребувати додаткової адаптації при застосуванні в конкретних технічних умовах та бізнес-контекстах.

Дослідження базується на матеріалах наявних у відкритому доступі, що може впливати на повноту теоретичної бази через неможливість опрацювання закритих наукових публікацій та досліджень.

Для виконання дослідження необхідні наступні ресурси:

а) технічні ресурси:
   - персональний комп’ютер з достатньою потужністю для розробки та тестування;
   - програмне забезпечення для розробки;
   - інструменти для тестування продуктивності.

б) інформаційні ресурси:
   - офіційна документація React;
   - документація досліджуваних бібліотек для форм;
   - наукові публікації та статті у відкритому доступі;
   - онлайн-ресурси для розробників.

в) часові ресурси:
   - час на вивчення та аналіз теоретичного матеріалу;
   - час на практичну реалізацію тестових прикладів;
   - час на аналіз та порівняння результатів;
   - час на оформлення результатів дослідження;

Результатом дослідження стане визначення найбільш ефективного підходу до розробки форм у React на основі багатокритеріального аналізу. Комплексна оцінка дозволить відобразити переваги та недоліки кожного з розглянутих підходів.

Результати дослідження матимуть безпосереднє практичне застосування при розробці нових React-проєктів, де важливим аспектом є ефективна реалізація форм. Отримані дані стануть основою для прийняття зважених рішень при виборі інструментів розробки, та допоможуть командам розробників уникнути потенційних проблем з продуктивністю та скоротити час на прийняття архітектурних рішень.

2 ОПИС ПРИЙНЯТИХ ПРОЄКТНИХ РІШЕНЬ

2.1 Базовий підхід до створення форм у React

React як бібліотека для розробки користувацьких інтерфейсів пропонує власний підхід до створення та управління формами, що базується на керованому стані компонентів. Цей «нативний» або «чистий» підхід до розробки форм не вимагає додаткових залежностей, проте має свої особливості, переваги та обмеження.

У React існує два основних підходи до створення компонентів: класові та функціональні компоненти. Ці підходи мають різну синтаксичну структуру, життєвий цикл та особливості роботи, що безпосередньо впливає на способи розробки форм.

Класові компоненти були першим способом створення інтерактивних елементів інтерфейсу в React. Вони представляють собою JavaScript-класи, що розширюють базовий клас React.Component. При роботі з формами в класових компонентах зазвичай використовується локальний стан (this.state) для зберігання значень полів. Кожне поле форми зв'язується зі своїм значенням у стані через атрибути value та onChange. Для оновлення значень полів форми використовується метод setState(), який забезпечує реактивну природу компонента.

У класових компонентах також важливим аспектом є правильне зв'язування (binding) обробників подій з контекстом компонента, що часто робиться у конструкторі. Класові компоненти мають доступ до методів життєвого циклу, таких як componentDidMount (викликається після першого рендеру компонента), componentDidUpdate (викликається після оновлення компонента) та інші, що дозволяє контролювати поведінку форми на різних стадіях її існування.

З випуском React 16.8 у 2019 році були представлені хуки (Hooks), які принципово змінили підхід до розробки компонентів. Функціональні компоненти, які раніше вважалися «безстановими» і використовувалися переважно для простого відображення даних, отримали можливість мати власний стан та використовувати інші можливості, раніше доступні тільки класовим компонентам.

Функціональні компоненти представляють собою звичайні JavaScript-функції, що повертають React-елементи. При роботі з формами у функціональних компонентах використовується хук useState для управління станом полів форми. Цей підхід дозволяє створювати більш лаконічні та читабельні компоненти без необхідності роботи з контекстом this та явного зв'язування обробників подій.

Крім useState, для роботи з формами часто використовуються й інші хуки. Наприклад, useEffect дозволяє виконувати побічні ефекти, такі як валідація полів або завантаження початкових даних форми. Хук useRef може використовуватися для прямого доступу до DOM-елементів форми, коли це необхідно.

Важливою перевагою функціональних компонентів є можливість створення власних хуків для інкапсуляції та повторного використання логіки форм. Це дозволяє виділити всю складну логіку роботи з формою в окремий модуль, що значно спрощує тестування та робить код більш підтримуваним.

На сьогодні функціональні компоненти з хуками стали стандартом у спільноті React-розробників. Офіційна документація React рекомендує використовувати саме цей підхід для нових проектів. Класові компоненти залишаються підтримуваними для забезпечення зворотної сумісності, але вважаються застарілим підходом, і їх використання в нових розробках не рекомендується.

При реалізації форм у сучасних React-застосунках функціональні компоненти надають більше гнучкості та призводять до написання більш декларативного коду, що краще відповідає ідеології React як бібліотеки для побудови користувацьких інтерфейсів.

Процес розробки форм у React з використанням функціональних компонентів базується на принципі керованих компонентів (controlled components). Цей підхід передбачає, що React контролює дані форми через стан компонента, забезпечуючи «єдине джерело правди» для всіх полів форми.

Основою для створення форм у функціональних компонентах є хук useState, який дозволяє керувати станом форми. Зазвичай для форми створюється єдиний

об'єкт стану, що містить значення всіх полів. При кожній зміні будь-якого поля форми відбувається оновлення відповідного значення у стані.

Процес управління формою включає кілька ключових етапів. Спочатку ініціалізується початковий стан форми, який може бути порожнім або містити значення за замовчуванням. Для кожного поля форми встановлюється двосторонній зв’язок: значення поля бере своє значення зі стану через атрибут value, а при зміні значення поля викликається обробник події onChange, який оновлює стан.

Обробник події onChange зазвичай реалізується як функція, що отримує об'єкт події, з якого витягується значення поля та його ім'я. Завдяки використанню атрибута name для ідентифікації полів форми, можна створити універсальний обробник, який буде працювати з будь-яким полем форми. Такий підхід дозволяє уникнути створення окремого обробника для кожного поля.

Важливою частиною роботи з формами є валідація введених даних. У функціональних компонентах валідація може бути реалізована через окремий стан помилок та відповідні функції перевірки. Валідація може виконуватися як під час введення даних (через обробник onChange), так і під час подання форми (через обробник onSubmit).

Обробник події onSubmit відповідає за обробку даних форми при її поданні. Він запобігає стандартній поведінці браузера (перезавантаженню сторінки) за допомогою методу preventDefault() та виконує необхідні дії з даними форми, такі як відправка на сервер або обробка в клієнтському застосунку.

Особливу увагу варто приділити обробці різних типів полів форми, таких як чекбокси, радіокнопки, списки вибору та файлові поля. Кожен з цих типів має свої особливості в контексті керованих компонентів. Наприклад, для чекбоксів використовується атрибут checked замість value, а для файлових полів зазвичай потрібно використовувати неконтрольований підхід через обмеження браузера.

2.2 Валідація форми за допомогою Yup

Валідація введених користувачем даних є критичним аспектом розробки форм. Для реалізації надійної та декларативної валідації в React-застосунках широко використовується бібліотека Yup. Ця бібліотека дозволяє створювати схеми валідації, які описують структуру даних та правила їх перевірки, що значно спрощує процес управління валідацією форм.

Yup пропонує потужний та виразний API для визначення схем валідації. На відміну від підходів з ручним написанням функцій валідації для кожного поля, Yup дозволяє декларативно описувати правила в зрозумілому форматі. Схема валідації створюється за допомогою набору методів, що визначають тип даних та правила перевірки.

Основний процес використання Yup з React-формами включає кілька кроків. Спочатку визначається схема валідації, яка описує всі поля форми та правила їх перевірки. Ця схема може включати різноманітні правила: перевірку типу даних, обов'язковість заповнення, мінімальну та максимальну довжину, відповідність регулярному виразу, унікальність значення та інші.

Для типових полів форми Yup надає відповідні типи валідації: string() для текстових полів, number() для числових, boolean() для перемикачів, date() для дат, array() для масивів та object() для вкладених об'єктів. Кожен тип має свій набір методів валідації, які можна об'єднувати в ланцюжки для створення складних правил.

Після визначення схеми її можна використовувати для валідації даних форми. Це може відбуватися як під час подання форми, так і під час введення даних. Для валідації під час подання форми схема застосовується до всіх даних форми одразу за допомогою методу validate() або validateSync(). У разі виявлення помилок, Yup генерує об'єкт помилок, який можна використовувати для відображення повідомлень користувачу.

Для валідації під час введення даних можна використовувати метод validateAt(), який перевіряє конкретне поле. Це дозволяє створювати інтерактивні

форми, де користувачі отримують миттєвий зворотний зв'язок про коректність введених даних.

Одна з ключових переваг Yup полягає в можливості створення складних умовних правил валідації. За допомогою методів when() та test() можна визначати правила, які залежать від значень інших полів форми. Наприклад, можна валідувати поле "підтвердження пароля" на відповідність полю "пароль" або змінювати правила валідації залежно від типу користувача.

Інтеграція Yup з функціональними компонентами React зазвичай реалізується через стан помилок та функції валідації. При зміні поля форми виконується валідація нового значення за допомогою Yup, і результат зберігається у стані помилок. Цей стан потім використовується для відображення повідомлень про помилки поруч із відповідними полями.

Для спрощення інтеграції Yup часто використовується спільно з бібліотеками управління формами, такими як Formik або React Hook Form. Ці бібліотеки надають готові механізми для інтеграції схем Yup та автоматичного управління станом помилок.

Важливою особливістю Yup є локалізація повідомлень про помилки. Бібліотека дозволяє налаштовувати текст повідомлень про помилки, що особливо важливо для багатомовних застосунків. Можна визначити глобальні повідомлення для всіх схем або задати унікальні повідомлення для конкретних правил валідації.

Для складних форм з багатьма полями Yup дозволяє створювати модульні схеми валідації. Можна визначити окремі схеми для різних частин форми, а потім об'єднати їх за допомогою методу object().shape() або concat(). Це покращує структуру коду та полегшує його підтримку при зміні вимог до валідації.

Використання Yup для валідації форм у React має кілька переваг порівняно з підходами на основі власного коду. По-перше, це зменшення обсягу коду, оскільки не потрібно писати окремі функції валідації для кожного правила. По-друге, це покращення читабельності коду завдяки декларативному стилю опису правил. По-третє, це спрощення підтримки та тестування, оскільки вся логіка валідації зосереджена в схемах Yup, які легко модифікувати та тестувати.

Завдяки своїй гнучкості та потужності, Yup став стандартом де-факто для валідації форм у React-екосистемі, забезпечуючи надійний та декларативний спосіб перевірки введених користувачем даних.

2.3 Створення форм за допомогою Formik

Formik є однією з найпопулярніших бібліотек для управління формами в React-застосунках. Вона дозволяє значно спростити роботу з формами, надаючи розробникам готове рішення для обробки стану форми, валідації даних та відправки форми на сервер. Formik істотно знижує кількість шаблонного коду, необхідного для реалізації функціональних форм, що робить її привабливим вибором для проєктів будь-якого масштабу.

Основною концепцією Formik є централізоване управління станом форми. Замість того, щоб вручну створювати стан для кожного поля форми та писати обробники подій, Formik бере на себе все управління станом і надає розробнику простий API для взаємодії з формою. Це дозволяє зосередитися на бізнес-логіці та дизайні форми, а не на технічних аспектах управління станом.

Для створення форми за допомогою Formik використовується компонент Formik або хук useFormik для функціональних компонентів. При ініціалізації Formik необхідно визначити кілька ключових параметрів: початкові значення полів форми (initialValues), функцію для обробки подання форми (onSubmit) та опціонально схему валідації (validationSchema), яка часто реалізується за допомогою Yup.

Процес валідації в Formik повністю інтегрований з бібліотекою Yup, що дозволяє декларативно описувати правила валідації для кожного поля форми. Formik автоматично відстежує помилки валідації та надає доступ до них через об'єкт errors. Крім того, Formik підтримує валідацію як при поданні форми, так і під час введення даних, забезпечуючи миттєвий зворотний зв'язок користувачу.

Для відображення полів форми Formik надає кілька підходів. Перший підхід використовує компоненти Field та Form, які автоматично підключаються до контексту Formik та спрощують роботу з полями форми. Другий підхід заснований

на рендер-пропсах або хуках, що дає більше гнучкості при створенні користувацьких полів або при інтеграції з UI-бібліотеками.

Важливою особливістю Formik є обробка подання форми. При виклику onSubmit Formik автоматично запобігає стандартній поведінці браузера та передає поточні значення форми у функцію обробник. Крім того, Formik надає стан isSubmitting, який можна використовувати для відображення індикатора завантаження під час асинхронної обробки форми, наприклад, при відправці даних на сервер.

Formik також підтримує обробку складних форм з вкладеними полями або масивами. Для роботи з вкладеними об'єктами використовується точкова нотація в іменах полів, а для роботи з масивами — компонент FieldArray, який надає методи для додавання, видалення та перестановки елементів масиву.

Для розробки великих форм Formik пропонує можливість розділення форми на менші компоненти. Для цього можна використовувати контекст Formik через хуки useField, useFormikContext та інші, які надають доступ до стану форми та методів Formik в будь-якому компоненті, вкладеному в Formik.

Одним із значних переваг Formik є управління станом форми при явних і неявних змінах значень. Formik надає методи setFieldValue, setFieldTouched та інші, які дозволяють програмно змінювати значення полів форми та їх стани. Це особливо корисно для полів, які не мають прямого відповідника в HTML, таких як слайдери або спеціалізовані селектори.

Окрім основних функцій, Formik також пропонує розширені можливості для обробки специфічних сценаріїв. Наприклад, можна використовувати функцію validate для власної валідації, що не покривається Yup, або встановлювати різні схеми валідації залежно від стану форми.

Для оптимізації продуктивності Formik використовує внутрішню мемоізацію та ефективно оновлює тільки ті частини форми, які дійсно змінилися. Це особливо важливо для великих форм з багатьма полями, де надмірний перерендеринг може призвести до проблем з продуктивністю.

Для багатьох проєктів Formik пропонує оптимальний баланс між простотою використання та функціональністю. Вона абстрагує складності роботи з формами, не обмежуючи розробника в можливостях кастомізації та розширення. Це робить Formik потужним інструментом для розробки форм різної складності — від простих форм входу до складних багатосторінкових форм з динамічними полями та складною логікою валідації.

2.4 Створення форм за допомогою React Hook Form

React Hook Form — це бібліотека для управління формами, яка оптимізована для високої продуктивності та мінімального перерендеру компонентів. На відміну від нативного підходу React, ця бібліотека дотримується концепції некерованих компонентів, використовуючи для доступу до даних форми API рефів, що забезпечує суттєвий виграш у продуктивності.

Концептуальною основою React Hook Form є використання некерованих компонентів (uncontrolled components) у поєднанні з хуком useForm, який надає всі необхідні методи та стани для роботи з формою. Цей підхід дозволяє зменшити кількість перерендерів, оскільки React не потрібно відстежувати кожну зміну в полях форми, як це відбувається при керованому підході.

Для початку роботи з React Hook Form необхідно імпортувати та ініціалізувати хук useForm. Цей хук повертає об'єкт, що містить методи для реєстрації полів, обробки подання форми, доступу до значень полів та помилок валідації. Основні функції, які надає useForm, включають register для реєстрації полів, handleSubmit для обробки подання форми, formState для доступу до стану форми та setValue для програмної зміни значень полів.

Реєстрація полів форми здійснюється за допомогою методу register, який повертає всі необхідні пропси для елемента форми: name, onChange, onBlur, ref. Цей метод також дозволяє задати правила валідації для кожного поля, такі як required, minLength, pattern та інші. React Hook Form підтримує як вбудовану валідацію, так і інтеграцію з популярними бібліотеками валідації, такими як Yup, Zod або Joi.

Важливою перевагою React Hook Form є потужна система валідації, яка підтримує як синхронну, так і асинхронну валідацію. Помилки валідації доступні через об'єкт formState.errors та можуть бути відображені поруч з відповідними полями форми. Валідація може виконуватися при різних подіях: під час зміни значення поля, при втраті фокусу або при поданні форми, що дозволяє гнучко налаштовувати користувацький досвід.

React Hook Form також пропонує зручні механізми для роботи з вкладеними об'єктами та масивами даних. Використовуючи спеціальний синтаксис для імен полів (наприклад, "user.address.street" або "items[0].name"), можна реєструвати поля, що відповідають складним структурам даних. Для роботи з динамічними масивами полів бібліотека надає спеціальний хук useFieldArray, який дозволяє додавати, видаляти та переміщувати елементи масиву.

Для відстеження стану форми, такого як dirty (наявність змін порівняно з початковими значеннями), isSubmitting (форма подається), isSubmitted (форма була подана) та інших, використовується об'єкт formState. Цей об'єкт дозволяє динамічно реагувати на зміни стану форми та відображати відповідні елементи інтерфейсу, наприклад, індикатор завантаження під час відправки даних.

React Hook Form також надає можливість спостереження за значеннями полів за допомогою хука useWatch. Цей хук дозволяє підписатися на зміни окремих полів або всієї форми, не викликаючи перерендер всього компонента форми. Це особливо корисно для реалізації залежних полів, де значення одного поля залежить від значення іншого.

Важливою особливістю React Hook Form є можливість інтеграції з існуючими бібліотеками компонентів, такими як Material-UI, Ant Design, Chakra UI та інші. Для цього бібліотека надає компонент Controller, який виступає посередником між компонентами інтерфейсу та системою керування формою. Controller приймає компонент керованого типу та забезпечує його взаємодію з React Hook Form, не порушуючи парадигму некерованих компонентів.

Для повторного використання логіки форм React Hook Form пропонує ряд механізмів: можливість створення власних хуків на основі useForm, використання

FormProvider для передачі контексту форми в дочірні компоненти, а також механізм схем значень за замовчуванням та перетворення значень. Ці інструменти дозволяють ефективно організувати код для складних форм з багатьма полями та складною логікою валідації.

Загалом, React Hook Form забезпечує оптимальний баланс між продуктивністю, гнучкістю та зручністю розробки. Завдяки використанню некерованих компонентів та рефів, ця бібліотека мінімізує кількість перерендерів, що особливо важливо для форм з великою кількістю полів або в застосунках, де критичною є продуктивність. При цьому розробнику надається простий та декларативний API, який значно спрощує типові задачі з управління формами: валідацію, обробку помилок, динамічні поля та інші сценарії.

2.5 Створення форм за допомогою Final Form

Final Form - це незалежна від фреймворку бібліотека для управління формами, з офіційною обгорткою React Final Form для екосистеми React. Ця бібліотека розроблена з акцентом на мінімалізм, продуктивність та композицію, що робить її потужним інструментом для створення як простих, так і надзвичайно складних форм.

На відміну від нативного підходу React або React Hook Form, Final Form повністю відокремлює логіку управління формою від компонентів інтерфейсу. Бібліотека працює за принципом підписки (subscription-based), що дозволяє компонентам отримувати оновлення лише коли змінюються конкретні частини стану форми, до яких вони підписані. Це значно знижує кількість перерендерів та оптимізує продуктивність, особливо у випадку складних форм з багатьма полями.

Для реалізації форми з використанням React Final Form необхідно використовувати два основні компоненти: Form та Field. Компонент Form створює екземпляр форми та надає контекст для всіх полів. Він приймає функцію onSubmit для обробки подання форми та опціональні параметри, такі як initialValues (початкові значення), validate (функція валідації всієї форми) та інші налаштування.

Компонент Field відповідає за окреме поле форми та використовує render props патерн. Він приймає функцію, яка отримує стан поля та допоміжні методи і повертає компонент для відображення. Завдяки системі підписок, кожен компонент Field перерендериться лише коли змінюються ті аспекти стану, на які він підписаний, наприклад, значення поля, помилки валідації або статус "торкання" (touched).

Валідація у Final Form може здійснюватися на кількох рівнях: на рівні всієї форми, на рівні окремого поля, або на рівні масиву полів. Для кожного рівня створюється відповідна функція валідації, яка приймає значення та повертає об'єкт помилок або undefined, якщо помилок немає. Final Form також підтримує асинхронну валідацію та валідацію при різних подіях, таких як зміна значення, втрата фокусу або подання форми.

Для роботи зі складними структурами даних Final Form пропонує гнучкий механізм іменування полів. З його допомогою можна легко працювати з вкладеними об'єктами та масивами. Наприклад, для поля в масиві можна використовувати ім'я виду "members[0].name", а для поля у вкладеному об'єкті — "user.address.city". Бібліотека також надає компонент FieldArray для роботи з динамічними масивами полів, що дозволяє додавати, видаляти та переміщувати елементи.

Важливою особливістю Final Form є підтримка ліній незалежних потоків валідації (record-level и field-level validation), що дозволяє організувати складні правила валідації, такі як перехресна валідація між полями або умовна валідація. Це робить бібліотеку особливо корисною для форм зі складною бізнес-логікою.

Final Form також надає засоби для локалізації (переклад текстів помилок), форматування та парсингу значень полів. Ці функції дозволяють автоматично перетворювати введені користувачем дані у формат, необхідний для бізнес-логіки, і навпаки, відображати дані в зручному для користувача форматі. Наприклад, можна автоматично форматувати введені цифри як валюту, дати або телефонні номери.

Бібліотека також підтримує потужну систему подій життєвого циклу форми, таких як initializing (форма ініціалізується), submitting (форма подається), submitFailed (виникла помилка під час подання) та інші. Це дозволяє реагувати на різні стани форми та створювати динамічні інтерфейси, які адаптуються до дій користувача.

Однією з сильних сторін Final Form є її мінімалістичний дизайн та модульність. Бібліотека надає основні функції для управління формами, а додаткова функціональність може бути реалізована за допомогою плагінів або власного коду. Це дозволяє зберігати розмір бандла мінімальним, включаючи лише необхідні функції.

Особливої уваги заслуговує декларативний підхід Final Form до управління формами. Розробник описує, як форма повинна виглядати та поводитися, а бібліотека бере на себе всю складність керування станом, валідацією та взаємодією з користувачем. Це робить код більш читабельним і підтримуваним, особливо у великих проєктах.

У контексті інтеграції з іншими бібліотеками компонентів, Final Form, як і React Hook Form, пропонує гнучкі механізми, які дозволяють використовувати компоненти із сторонніх бібліотек, таких як Material-UI або Ant Design. Це досягається завдяки патерну render props, який надає повний контроль над тим, як рендериться кожне поле.

Загалом, Final Form представляє собою потужний і гнучкий інструмент для управління формами в React-застосунках. Її архітектура, базована на підписках та декларативному підході, забезпечує високу продуктивність та чистий, підтримуваний код. Ця бібліотека особливо корисна для форм зі складною логікою, великою кількістю полів або високими вимогами до продуктивності.

2.6 Створення форм за допомогою Redux Form

Redux Form представляє собою бібліотеку для інтеграції керування станом форм з Redux, популярним інструментом для керування станом у React-додатках. Створена для вирішення проблеми складного керування станом форм, Redux Form

дозволяє зберігати всі дані форм централізовано в Redux-сховищі, що спрощує розробку складних форм та їх валідацію.

Концептуальною основою Redux Form є принцип "єдиного джерела істини", що означає зберігання всіх даних форми в одному місці – Redux-сховищі. Це дозволяє краще відстежувати зміни стану форми, спрощує налагодження та тестування, а також забезпечує більш передбачувану поведінку форм у великих додатках.

При використанні Redux Form, форма з'єднується з Redux-сховищем за допомогою спеціального ред’юсера. Цей ред’юсер автоматично обробляє події змін полів форми та зберігає їх значення у сховищі. Він також відповідає за відстеження стану форми, включаючи інформацію про валідність, чи була форма змінена, чи була вона надіслана тощо.

Основним механізмом роботи Redux Form є декоратор або функція вищого порядку, яка обгортає компонент форми та підключає його до Redux-сховища. Ця функція надає компоненту додаткові пропси, які дозволяють йому взаємодіяти зі сховищем Redux, включаючи методи для керування станом форми, такі як ініціалізація форми, скидання форми, відправка форми та інші.

Важливим аспектом Redux Form є його підхід до керування полями форми. Кожне поле форми реєструється в Redux-сховищі та отримує власний набір методів та властивостей, які дозволяють відстежувати його стан, включаючи такі аспекти як поточне значення, чи було поле змінено, чи було воно відвідано користувачем, чи містить воно помилки валідації тощо.

Модель валідації у Redux Form дозволяє визначати правила валідації на рівні форми або на рівні окремих полів. Валідація може виконуватися синхронно при зміні значення поля або асинхронно, наприклад, при перевірці унікальності значення через API. Помилки валідації зберігаються у Redux-сховищі разом із значеннями полів, що забезпечує централізований доступ до всієї інформації про стан форми.

Redux Form також надає механізми для обробки відправки форми. При відправці форми бібліотека автоматично виконує валідацію всіх полів і, якщо

форма валідна, викликає обробник відправки форми з поточними значеннями полів. Якщо форма не валідна, відправка блокується, і користувачу показуються повідомлення про помилки.

Архітектурно, Redux Form добре інтегрується з іншими частинами Redux-екосистеми, такими як Redux Thunk або Redux Saga, що дозволяє ефективно обробляти асинхронні операції при роботі з формами, наприклад, завантаження початкових значень форми з сервера або відправку даних форми на сервер.

Незважаючи на потужність і гнучкість Redux Form, важливо врахувати певні обмеження та недоліки цього підходу. Зокрема, Redux Form може призвести до підвищеного споживання ресурсів через постійне оновлення Redux-сховища при кожній зміні поля форми, що може негативно вплинути на продуктивність додатка з великою кількістю форм або полів.

У контексті сучасної розробки React-додатків, Redux Form поступово поступається місцем новішим бібліотекам, таким як React Hook Form або Formik, які використовують більш локальний підхід до керування станом форм і краще інтегруються з функціональними компонентами та хуками React. Вони пропонують більш інтуїтивні API та зменшують складність конфігурації для простих випадків. Тим не менш, Redux Form залишається потужним інструментом для керування формами в додатках, що використовують Redux для керування глобальним станом.

2.7 Проєктування архітектури тестових програм

Для проведення порівняльного аналізу ефективності різних підходів до розробки форм у React буде розроблено п’ять окремих тестових програм, кожна з яких буде реалізувати ідентичний функціонал, але з використанням різних підходів: «чистого» React, бібліотеки Formik, бібліотеки React Hook Form, бібліотеки Final Form та бібліотеки Redux Form.

Усі тестові програми мають однакову архітектуру, що забезпечує об’єктивність порівняння, та дозволяє зосередитися на особливостях кожного підходу. Архітектура базується на стандартній структурі React-додатку з наступними ключовими компонентами:

– index.html - головна HTML-сторінка, що служить точкою входу в додаток;
– index.js - точка входу JavaScript-коду, відповідає за рендеринг кореневого компонента;
– App.js - основний компонент додатку, що інтегрує всі інші модулі;
– FormExample.js - компонент форми, реалізований з використанням відповідного підходу;
– main.css - файл стилів для забезпечення єдиного візуального оформлення;
– Profiler - модуль для збору метрик продуктивності.

На верхньому рівні знаходиться HTML-файл index.html, у якому підключається кореневий JavaScript-файл index.js. У ньому рендериться кореневий React-компонент App.js.

Компонент App.js, в свою чергу, підключає наступні елементи:

– компонент форми FormExample.js, який реалізує логіку введення та валідації даних;
– компонент профайлера (Profiler), який обгортає форму і використовується для збору метрик продуктивності;
– файл стилів main.css, який відповідає за зовнішній вигляд інтерфейсу.

Загальна архітектура проекту наведена у діаграмі компонентів (див. рис. 2.1).

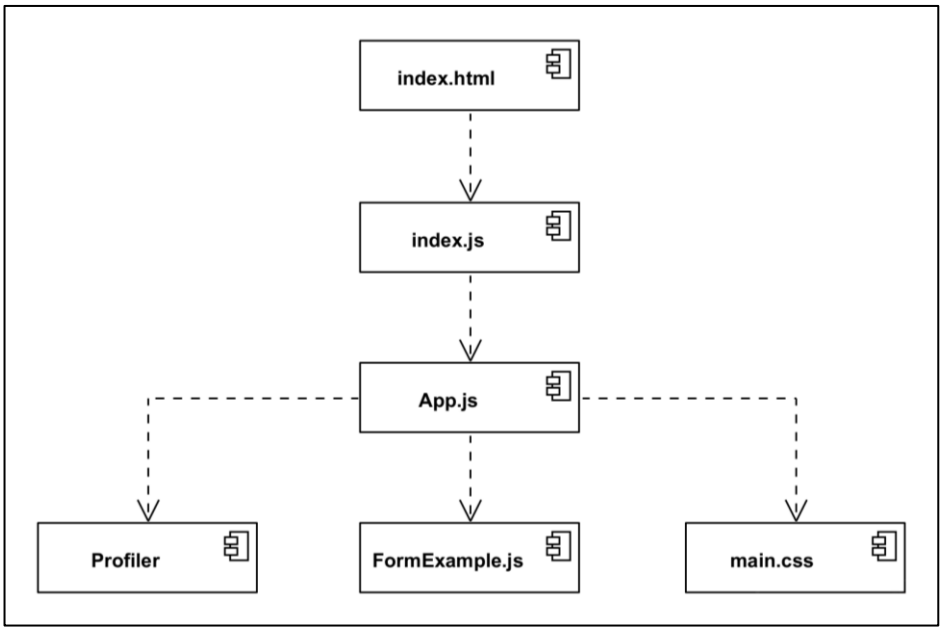

Рисунок 2.1 - Діаграма компонентів (виконано самостійно)

Форма складатиметься з низки полів різного типу, які охоплюють як прості текстові введення, так і більш складні елементи. Зокрема, планується включити наступні елементи:

- Name – текстове поле, яке приймає виключно великі та малі літери латинського алфавіту (A–Z, a–z);
- Email – текстове поле з валідацією формату електронної адреси;
- Phone number – текстове поле з валідацією телефонного номеру (починається з «+380» і містить загалом 12 цифр);
- Password – поле типу «password», яке повинно містити щонайменше 8 символів, включно з принаймні однією великою літерою, однією малою, однією цифрою та одним спеціальним символом;
- Gender – група радіокнопок (Male, Female, Other);
- Choose your hobbies – чекбокси (Reading, Drawing, Sport);
- Birthday – поле типу дата (calendar picker), що допускає лише дати, що відповідають віку користувача від 13 до 100 років;
- When can we call you? – поле вибору часу (time picker);
- Pick a number from 0 to 100 – числове поле з обмеженням значень в діапазоні від 0 до 100;
- Your favorite color – поле вибору кольору (color picker);
- Rate this form on a scale from 0 to 10 – повзунок (slider);
- Your favorite season – випадаючий список (dropdown) зі значеннями Spring, Summer, Autumn, Winter;
- Comment – багаторядкове текстове поле (textarea);
- Submit – кнопка надсилання форми.

Такий набір полів дозволить перевірити, як різні бібліотеки працюють з усіма основними типами елементів введення в React. Крім того, він створює достатнє навантаження на систему валідації, що дозволяє оцінити продуктивність з різними підходами. Для кожного поля передбачено валідацію як під час введення, так і під час спроби надіслати форму.

Інтерфейс форми є спільним для всіх тестових реалізацій. Wireframe-макет (каркасна модель) форми, який слугує основою для подальшої реалізації кожної з тестових версій, представлений на рисунку 2.2.

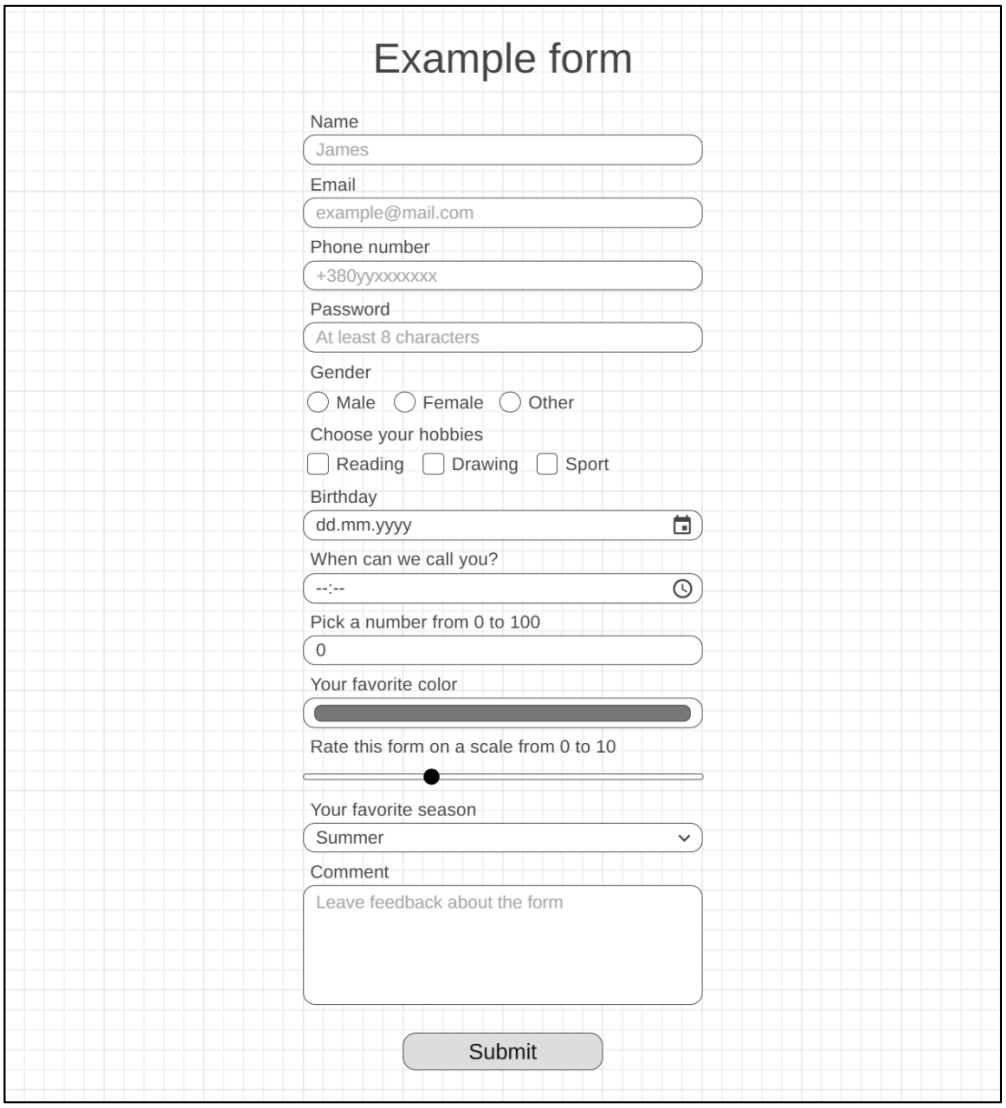

Рисунок 2.2 - Wireframe-макет форми (виконано самостійно)

Всі тестові програми мають реалізувати ідентичну логіку роботи з формою, яка включає кілька етапів обробки користувацьких дій. Алгоритм роботи форми представлено у вигляді діаграми активностей (див. рис. 2.3).

Процес взаємодії користувача з формою починається з ініціалізації компонента, після чого здійснюється рендеринг форми з порожніми полями або значеннями за замовчуванням. Далі система очікує дій користувача, які можуть бути двох типів: введення даних у поля форми або спроба відправлення форми.

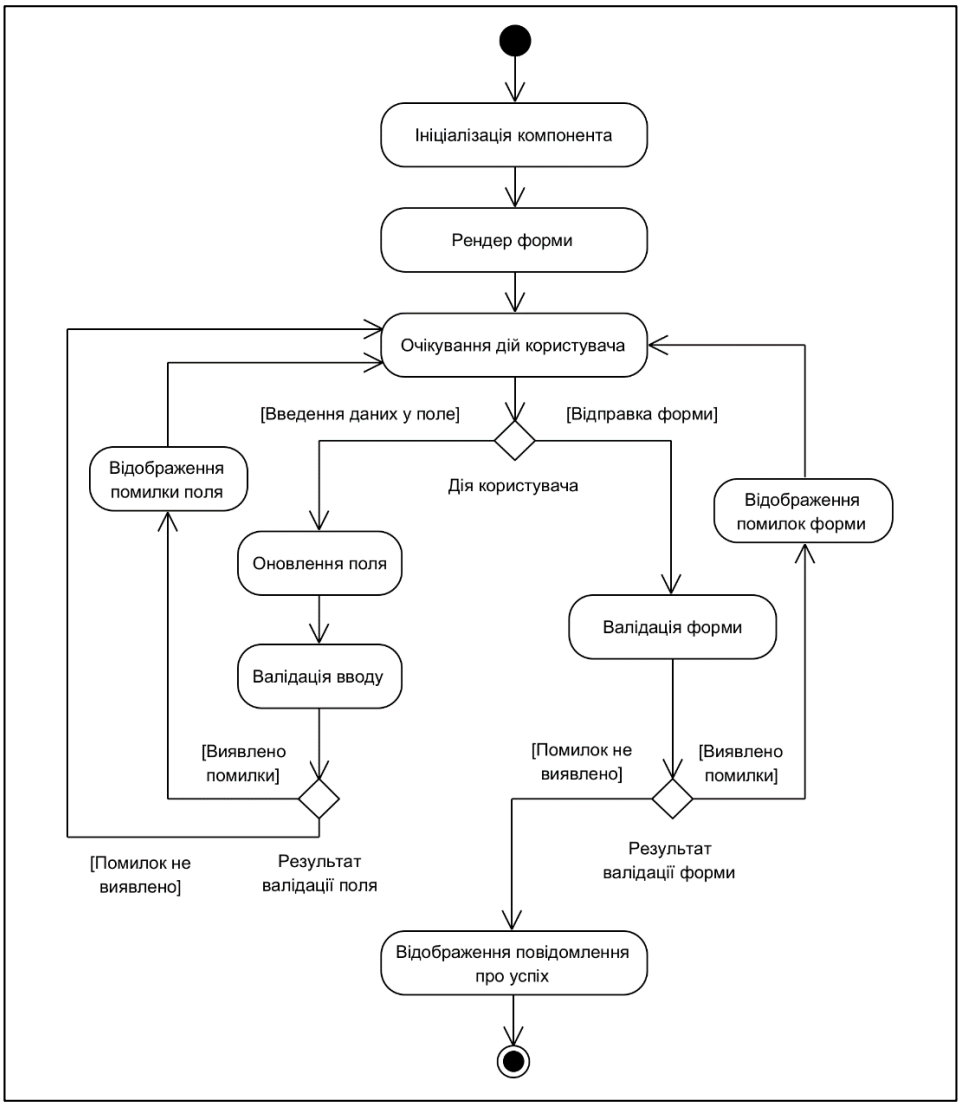

Рисунок 2.3 - Діаграма активностей (виконано самостійно)

При введенні даних у будь-яке поле форми запускається процес валідації введених даних. Валідація відбувається в реальному часі для забезпечення миттєвого зворотного зв'язку з користувачем. У разі виявлення помилок валідації, система відображає відповідні повідомлення про помилки під відповідними полями. Якщо введені дані коректні, поле вважається валідним, і помилки не відображаються.

Коли користувач натискає кнопку «Submit», система здійснює комплексну валідацію всіх полів форми. На цьому етапі перевіряються всі поля незалежно від того, чи вводились у них дані раніше. Якщо будь-яке з полів містить некоректні дані або є обов'язковим, але порожнім, система відображає відповідні повідомлення про помилки та перешкоджає відправленню форми.

У випадку успішної валідації всіх полів форми, система відображає повідомлення про успішне надіслання форми. Для спрощення тестування та зосередження уваги на продуктивності різних підходів, фактичне надсилання даних на сервер не реалізується.

2.8 Планування експериментальної частини дослідження

Для об’єктивного порівняння різних підходів до розробки форм у React та вибору найбільш ефективного варіанту доцільно використати математичний апарат теорії прийняття рішень. Оскільки кожен з доступних підходів має бути оцінений за декількома критеріями одночасно, такими як швидкодія, споживання пам'яті та час відгуку, класичні методи порівняння не дають повної картини. У такій ситуації оптимальним є застосування методів багатокритеріальної оптимізації, які дозволяють врахувати всі важливі характеристики та їх відносну важливість при виборі рішення.

У даному випадку ми розв'яжемо багатокритеріальну задачу прийняття рішень, яка дозволить формалізувати процес вибору та отримати обґрунтовані результати на основі експериментальних даних. Такий підхід забезпечить можливість не лише порівняти наявні альтернативи за кількісними показниками, але й врахувати різну важливість окремих характеристик з точки зору розробки реальних веб-застосунків.

Для вирішення нашої задачі будемо використовувати метод лінійної адитивної згортки з нормуючими та ваговими коефіцієнтами. Цей метод є одним з найбільш поширених та ефективних підходів до розв'язання багатокритеріальних задач, особливо коли необхідно врахувати різну важливість критеріїв та привести їх до спільної шкали оцінювання.

Згорткова модель з лінійною адитивною згорткою використовує нормовані значення критеріїв і ваги для їх комбінування. В основі методу лежить принцип адитивності, згідно з яким загальна корисність альтернативи може бути представлена як сума окремих корисностей за кожним критерієм. Кожен критерій нормується до спільної шкали, що дозволяє коректно порівнювати показники,

виміряні в різних одиницях. Після нормування, значення критерію множиться на ваговий коефіцієнт відповідно до його важливості в контексті вирішуваної задачі.

Такий підхід має ряд переваг для нашого дослідження:

- дозволяє враховувати відносну важливість різних характеристик форм;
- забезпечує прозорість процесу оцінювання та легкість інтерпретації результатів;
- надає можливість легко модифікувати вагові коефіцієнти при зміні пріоритетів;
- дозволяє отримати чітку кількісну оцінку для кожної альтернативи.

В результаті застосування методу ми отримаємо загальну оцінку для кожної альтернативи, яка буде враховувати всі важливі аспекти роботи форм з урахуванням їх відносної значущості для розробки реальних веб-застосунків.

Для визначення найбільш ефективного підходу до розробки форм у React ми будемо використовувати такі експериментально вимірювані критерії:

- кількість рядків коду;
- час першого рендерингу (мс);
- кількість ререндерів;
- мінімальний об’єм споживання пам’яті (Мб);
- максимальний об’єм споживання пам’яті (Мб);
- час відгуку INP (interaction to next paint) (мс);
- розмір бібліотеки (та залежностей) у бандлі (Кб).

Для проведення аналізу було відібрано п’ять найбільш поширених та перспективних підходів до розробки форм у React, кожен з яких має свої особливості реалізації та специфіку використання.

Ми маємо множину альтернатив $A = \{A_1, A_2, A_3, A_4, A_5\}$, де:

A_1 – використання «чистого» React;

A_2 – використання бібліотеки Formik;

A_3 – використання бібліотеки React Hook Form;

A_4 – використання бібліотеки Final Form;

A_5 – використання бібліотеки Redux Form.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Базова конфігурація та спільні компоненти системи

Після того, як було спроєктовано архітектуру програмного забезпечення та сплановано проведення експерименту, ми приступили до розробки тестових застосунків.

Для кожного варіанту реалізації було створено окремий React-застосунок за допомогою інструменту Create React App, що дозволяє швидко згенерувати типову структуру проєкту з мінімальними початковими налаштуваннями. Усі п’ять проєктів створювались однаково, шляхом виконання наступної команди в терміналі:

```
npx create-react-app <approach-name>-solution
```

Для проєктів назва обиралась відповідно до підходу: final-form-solution, formik-solution, pure-react-solution, react-hook-form-solution, redux-form-solution.

Після створення базового проєкту виконувалось встановлення необхідних залежностей для кожного конкретного підходу, за допомогою наступної команди:

```
npm install <package-name>
```

Спільними залежностями для всіх проектів є бібліотека Yup для схеми валідації.

Після створення базового проекту та встановлення необхідних залежностей було проведено реструктуризацію файлової організації: було створено окремі теки «components» для React-компонентів та «styles» для CSS-файлів. Компонент FormExample.js, що реалізує логіку форми, знаходиться в теці «components» і є єдиною змінною частиною кожного застосунку.

З-поміж усіх реалізацій винятком є проєкт з використанням Redux Form – у ньому, на відміну від інших, додатково створено окрему теку для зберігання Redux-логіки, зокрема конфігурації стору.

Фінальна структура файлів та директорій, яка використовується в усіх п’яти реалізаціях, представлена на рисунку 3.1.

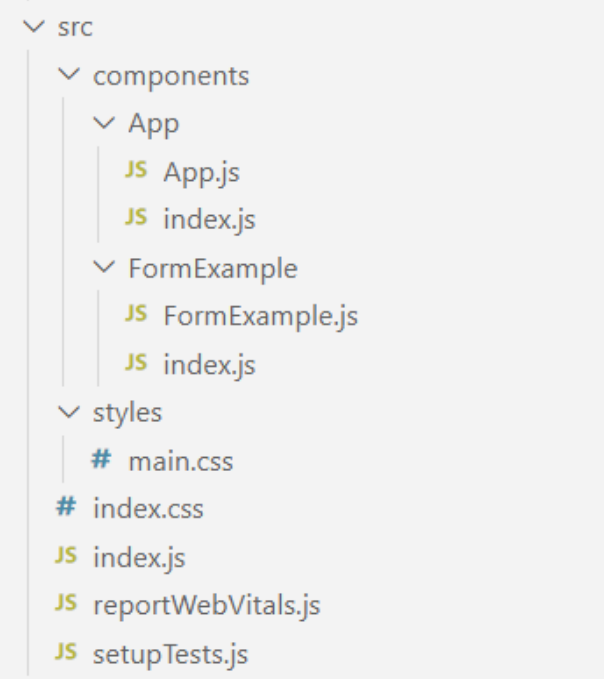

Рисунок 3.1 - Структура файлів та директорій (виконано самостійно)

Компонент App.js виконує роль кореневого компонента застосунку, який об’єднує всі його елементи та відповідає за ініціалізацію та відображення основного компонента форми. Його архітектура спроектована з урахуванням необхідності проведення моніторингу продуктивності.

У стандартному режимі компонент App.js має мінімалістичну структуру:

```
import "../../styles/main.css";
import FormExample from "../FormExample";

function App() {
  return (
    <FormExample />
  );
}

export default App;
```

Для проведення вимірювань продуктивності компонент App.js може бути модифікований для включення React Profiler API.

Завдяки React Profiler, під час тестування можна точно вимірювати кількість перерендерів та тривалість кожного рендерингу без додаткових сторонніх інструментів, просто спостерігаючи за виводом у консоль браузера або інтегруючи заміри в аналітичні скрипти.

Ця модифікація виконується залежно від конкретних метрик, які необхідно виміряти. Наведемо приклад коду, який виводить у консоль час першого рендерингу:

```
import { Profiler } from "react";
import "../../styles/main.css";
import FormExample from "../FormExample";

function App() {
  const onRenderCallback = (id, phase, actualDuration) => {
    if (phase === "mount") {
      console.log(`First render time for ${id}:
${actualDuration.toFixed(2)}ms`);
    }
  };

  return (
    <Profiler id="FormExample" onRender={onRenderCallback}>
      <FormExample />
    </Profiler>
  );
}

export default App;
```

Для забезпечення візуальної консистентності між усіма реалізаціями форм було створено єдиний файл стилів main.css, який використовується у всіх п’яти застосунках без змін. Стилі були розроблені таким чином, щоб усі форми мали однаковий зовнішній вигляд незалежно від використаної бібліотеки.

Файл стилів організований за модульним принципом з чіткою структурою:

- GLOBAL – глобальні налаштування, включаючи підключення шрифту Montserrat та визначення CSS-змінних для кольорової палітри;
- RESET STYLES – скидання стандартних стилів браузера для забезпечення кроссбраузерної сумісності;
- LAYOUT – макет та загальна структура форми;
- FIELDS – стилізація полів вводу та їх підписів;

– BUTTONS – оформлення кнопок з ефектами при взаємодії;
– SELECTORS – стилі для checkbox, radio button та інших елементів вибору;
– ERROR MESSAGE – оформлення повідомлень про помилки валідації;
– BROWSER TWEAKS – специфічні налаштування для різних браузерів.

Код файлу main.css наведено у додатку Д.

Усі експериментальні форми використовують одну й ту саму схему валідації, реалізовану за допомогою бібліотеки yup. Це дозволяє гарантувати, що перевірка даних у кожному варіанті реалізації відбувається за однаковими правилами, незалежно від того, яка бібліотека для форм була використана.

Обрана бібліотека yup є популярним інструментом для опису схеми даних і реалізації валідації у React-застосунках. Вона підтримує декларативний підхід, дозволяє створювати читаємий опис правил і легко інтегрується з усіма форм-бібліотеками, які були використані в експерименті, а також з ручною реалізацією.

Нижче наведено поетапний опис створення схеми валідації для форми.

Спершу створюється об’єкт validationSchema за допомогою методу yup.object().shape({...}), який визначає правила валідації для кожного поля форми. Ось фрагмент коду, що демонструє це:

```
const validationSchema = yup.object().shape({
    // поля форми та їх валідація
});
```

Поле name перевіряється на наявність лише англійських літер. Також воно є обов’язковим:

```
name: yup
    .string()
    .matches(/^[A-Za-z]+$/, "Name must contain only English letters")
    .required("Name is required"),
```

Для поля email використовується стандартна перевірка формату email-адреси, також поле є обов’язковим:

```
    email: yup
        .string()
        .email("Invalid email format")
        .required("Email is required"),
```

Поле phone має відповідати українському формату номера телефону, що починається з «+380»:

```
    phone: yup
        .string()
        .matches(/^\+380\d{9}$/, "Phone number must start with '+380' and
contain 12 digits")
        .required("Phone number is required"),
```

Для password задаються кілька правил: мінімальна довжина, наявність різних типів символів (великих та малих латинських літер, цифр, спец-символів):

```
    password: yup
        .string()
        .min(8, "Password must be at least 8 characters long")
        .matches(/[a-z]/, "Password must contain at least one lowercase
letter")
        .matches(/[A-Z]/, "Password must contain at least one uppercase
letter")
        .matches(/\d/, "Password must contain at least one number")
        .matches(/[@$!%*?&#]/, "Password must contain at least one
special character")
        .required("Password is required"),
```

Поле gender є обов’язковим, але не має додаткових обмежень щодо значення:

```
    gender: yup
        .string()
        .required("Gender is required"),
```

Поле time є обов’язковим, однак не має додаткових умов:

```
    time: yup
        .string()
        .required("Time is required"),
```

Для поля birthday додається кастомна перевірка віку – він має бути від 13 до 100 років:

```
    birthday: yup
        .date()
        .required("Birthday is required")
        .test(
            "age",
            "You must be at least 13 years old and not older than 100
years (because it is impossible)",
            (value) => {
                if (!value) return false;
                const today = new Date();
                const birthDate = new Date(value);
                const age = today.getFullYear() -
birthDate.getFullYear();
                const monthDiff = today.getMonth() -
birthDate.getMonth();
                if (monthDiff < 0 || (monthDiff === 0 && today.getDate()
< birthDate.getDate())) {
                    return age - 1 >= 13 && age - 1 <= 100;
                }
                return age >= 13 && age <= 100;
            }
        ),
```

Для поля number встановлено межі значення від 0 до 100 включно:

```
    number: yup
        .number()
        .required("Number is required")
        .min(0, "Number must be at least 0")
        .max(100, "Number must be at most 100"),
```

Таким чином, використовуючи Yup, створюється гнучка та легко підтримувана система валідації.

3.2 Розробка форм з використанням «чистого» React

Наведемо опис програмної реалізації застосунку з використанням «чистого» React підходу для розробки форм.

Основою реалізації є використання хука useState для управління станом форми. Всі дані форми зберігаються в одному об'єкті стану:

```
    const [formData, setFormData] = useState({
        name: "",
        email: "",
        //...
        comment: "",
    });
```

Для обробки змін у текстових полях реалізовано універсальну функцію handleChange:

```
const handleChange = (event) => {
    const { name, value } = event.target;
    setFormData(prev => ({
        ...prev,
        [name]: value
    }));
    validateField(name, value);
};
```

Ця функція використовує деструктуризацію для отримання імені поля та його значення, після чого оновлює стан за допомогою spread оператора. Одразу після оновлення викликається валідація конкретного поля.

Для чекбоксів створено окрему функцію обробки:

```
const handleCheckboxChange = (event) => {
    const { name, checked } = event.target;
    setFormData(prev => ({
        ...prev,
        [name]: checked
    }));
};
```

Валідація реалізована через інтеграцію з бібліотекою Yup. Для зберігання помилок валідації використовується окремий стан, валідація окремого поля виконується асинхронно:

```
const [fieldErrors, setFieldErrors] = useState({});

const validateField = async (name, value) => {
    try {
        await yup.reach(validationSchema, name).validate(value);
        setFieldErrors(prev => ({
            ...prev,
            [name]: undefined
        }));
    } catch (err) {
        setFieldErrors(prev => ({
            ...prev,
            [name]: err.message
        }));
    }
};
```

Кожне поле форми має стандартну структуру з міткою, інпутом та відображенням помилки. Ось приклад текстового поля:

```
<label className="form-label" htmlFor="name">Name</label>
<input
    className="form-field"
    type="text"
    id="name"
    name="name"
    value={formData.name}
    onChange={handleChange}
    placeholder="James"
/>
{fieldErrors.name && <div className="form-error-
message">{fieldErrors.name}</div>}
```

Для радіо-кнопок використовується прив'язка через атрибут checked:

```
<input
    className="form-radio"
    type="radio"
    id="male"
    name="gender"
    value="male"
    checked={formData.gender === "male"}
    onChange={handleChange}
/>
```

Процес відправки включає повну валідацію всіх полів:

```
const handleSubmit = async () => {
    try {
        await validationSchema.validate(formData, { abortEarly: false
});
        alert("Form submitted successfully!");
        setFieldErrors({});
    } catch (err) {
        if (err instanceof yup.ValidationError) {
            const errors = {};
            err.inner.forEach((error) => {
                errors[error.path] = error.message;
            });
            setFieldErrors(errors);
        }
    }
};
```

Чистий React підхід характеризується прямолінійністю та повним контролем над логікою форми. Всі зміни стану відбуваються через явні виклики setState, що забезпечує передбачуваність поведінки. Однак це призводить до написання значної кількості шаблонного коду, особливо для великих форм з багатьма полями.

Валідація виконується як в реальному часі (при зміні поля), так і при відправці форми, що забезпечує хороший користувацький досвід з негайним зворотним зв'язком про помилки.

Повний код файлу FormExample.js для програми що використовує «чистий» React наведено в додатку Е.

3.3 Розробка форм з використанням Formik

Formik кардинально змінює підхід до управління формами, використовуючи компонент-обгортку та систему рендер-пропсів. Основою є компонент Formik, який приймає конфігурацію форми:

```
<Formik
    initialValues={initialValues}
    validationSchema={validationSchema}
    onSubmit={handleSubmit}
>
    {({ isSubmitting }) => (
        <Form>
            {/* поля форми */}
        </Form>
    )}
</Formik>
```

Замість використання useState, початкові значення визначаються в окремому об’єкті:

```
const initialValues = {
    name: "",
    email: "",
    phone: "",
    password: "",
    gender: "",
    hobbyReading: false,
    hobbyDrawing: false,
    // ... інші поля
};
```

Цей об’єкт передається безпосередньо в Formik компонент, який автоматично створює та управляє внутрішнім станом форми.

Formik надає спеціалізовані компоненти для створення полів форми. Основним є компонент Field, який автоматично підключається до стану форми:

```
<label className="form-label" htmlFor="name">Name</label>
<Field className="form-field" type="text" id="name" name="name"
placeholder="James" />
<ErrorMessage className="form-error-message" name="name"
component="div" />
```

Компонент Field автоматично обробляє події onChange, onBlur та інші, не потребуючи написання власних обробників подій.

Для відображення помилок використовується компонент ErrorMessage:

```
<ErrorMessage className="form-error-message" name="email"
component="div" />
```

Цей компонент автоматично відображає помилку для відповідного поля, якщо така існує, та приховує себе, коли помилка відсутня.

Для складніших елементів форми, таких як select або textarea, використовується пропс as:

```
<Field className="form-field" as="select" id="season" name="season">
    <option value="summer">Summer</option>
    <option value="autumn">Autumn</option>
    <option value="winter">Winter</option>
    <option value="spring">Spring</option>
</Field>
```

Функція обробки відправки приймає не лише значення форми, а й об’єкт з корисними методами:

```
const handleSubmit = (values, { setSubmitting }) => {
    alert("Form submitted successfully!");
    setSubmitting(false);
};
```

Параметр setSubmitting дозволяє керувати станом завантаження форми, що автоматично відключає кнопку відправки під час процесу обробки.

Через рендер-функцію Formik надає доступ до різних аспектів стану форми:

```
{({ isSubmitting }) => (
    <Form>
        {/* поля форми */}
        <button className="form-button" type="submit"
disabled={isSubmitting}>
            Submit
        </button>
    </Form>
)}
```

Formik значно спрощує код форми, усуваючи необхідність в ручному управлінні станом та написанні обробників подій. Бібліотека автоматически інтегрується з Yup для валідації, забезпечуючи декларативний підхід до опису логіки перевірки даних. Основною перевагою є зменшення кількості шаблонного коду - замість створення окремих обробників для кожного поля, достатньо лише вказати атрибут name для компонента Field.

Повний код файлу FormExample.js для програми що використовує Formik наведено в додатку Ж.

3.4 Розробка форм з використанням React Hook Form

React Hook Form використовує хук useForm для створення та управління формою. Цей хук повертає набір методів та об’єктів для роботи з даними:

```
const {
    register,
    handleSubmit,
    control,
    formState: { errors },
} = useForm({
    resolver: yupResolver(validationSchema),
    defaultValues: {
        name: "",
        email: "",
        phone: "",
        // ... інші поля
    },
});
```

Основною особливістю є використання yupResolver для інтеграції з Yup валідацією та визначення початкових значень через параметр defaultValues.

Центральним механізмом є функція register, яка підключає HTML-елементи до системи управління формою. Ось приклад використання для текстового поля:

```
<input
    className="form-field"
    type="text"
    id="name"
    {...register("name")}
    placeholder="James"
/>
```

Функція register повертає об’єкт з необхідними пропсами (ref, name, onChange, onBlur), які розпаковуються через spread оператор безпосередньо в елемент.

Помилки доступні через об’єкт formState.errors та відображаються умовно:

```
    {errors.email && <div className="form-error-
message">{errors.email.message}</div>}
```

Кожна помилка містить властивість message з текстом помилки, визначеним у схемі валідації Yup.

Для радіо-кнопок всі елементи групи реєструються з однаковим ім'ям:

```
<input
    className="form-radio"
    type="radio"
    id="male"
    value="male"
    {...register("gender")}
/>
<input
    className="form-radio"
    type="radio"
    id="female"
    value="female"
    {...register("gender")}
/>
```

React Hook Form автоматично управляє станом групи радіо-кнопок, забезпечуючи вибір лише однієї опції.

Для деяких елементів, які потребують додаткового контролю, використовується компонент Controller:

```
<Controller
    name="color"
    control={control}
    render={({ field }) => (
        <input
            className="form-field"
            type="color"
            id="color"
            {...field}
        />
    )}
/>
```

Компонент Controller надає більш детальний контроль над рендерингом та поведінкою поля через рендер-пропс патерн.

Функція handleSubmit обгортає користувацький обробник та автоматично керує валідацією:

```
const onSubmit = (data) => {
    alert("Form submitted successfully!");
    console.log(data);
};

// JSX:
<form onSubmit={handleSubmit(onSubmit)}>
```

Якщо валідація пройшла успішно, викликається функція onSubmit з очищеними та перевіреними даними.

На відміну від Formik, React Hook Form використовує стандартний HTML елемент form:

```
<form onSubmit={handleSubmit(onSubmit)}>
    {/* поля форми */}
    <button className="form-button" type="submit">Submit</button>
</form>
```

React Hook Form відрізняється мінімальною кількістю перерендерингів, оскільки використовує неконтрольовані компоненти за замовчуванням. Форма зберігає дані в DOM, а не в React стані, що значно покращує продуктивність для великих форм.

Бібліотека надає інтуїтивний API через функцію register, яка дозволяє легко підключати будь-які HTML елементи форми до системи управління.

Повний код файлу FormExample.js для програми що використовує React Hook Form наведено в додатку И.

3.5 Розробка форм з використанням Final Form

React Final Form є високопродуктивною бібліотекою для управління формами, яка базується на принципі підписки та мінімізації повторних рендерів. Основою архітектури є компонент Form, який надає контекст для всіх полів форми та управляє станом.

Головний компонент форми побудований навколо обгортки Form з React Final Form. Ось фрагмент коду який демонструє основну структуру:

```
<Form
    onSubmit={onSubmit}
    validate={validate}
    initialValues={{
        name: "",
        email: "",
        // інші початкові значення
    }}
    render={({ handleSubmit, submitError }) => (
        <form onSubmit={handleSubmit}>
            {/* поля форми */}
        </form>
    )}
/>
```

Компонент Form приймає три ключові пропи: onSubmit для обробки відправки, validate для валідації та initialValues для початкових значень полів.

Кожне поле форми створюється за допомогою компонента Field, який забезпечує зв’язок між HTML-елементами та станом форми. Ось фрагмент коду який показує типову структуру поля:

```
<Field name="name">
    {({ input, meta }) => (
        <>
            <label className="form-label" htmlFor="name">Name</label>
            <input {...input} className="form-field" id="name"
placeholder="James" />
            {meta.touched && meta.error && <span className="form-
error-message">{meta.error}</span>}
        </>
    )}
</Field>
```

Об'єкт input містить всі необхідні пропи для HTML-елемента (value, onChange, onBlur), а meta надає інформацію про стан поля (touched, error, valid).

Для радіо-кнопок використовується кілька компонентів Field з однаковим атрибутом name але різними значеннями value. Ось фрагмент коду який демонструє реалізацію групи радіо-кнопок:

```
<Field name="gender" type="radio" value="male">
    {({ input }) => (
        <div className="form-selection-wrapper">
            <input {...input} className="form-radio" id="male" />
            <label className="form-selection-label"
htmlFor="male">Male</label>
        </div>
    )}
</Field>
```

Для чекбоксів кожен елемент має окреме ім’я поля, що дозволяє незалежно керувати їх станом.

Функція валідації інтегрується з формою через проп validate. Ось фрагмент коду який показує асинхронну валідацію з використанням Yup:

```
const validate = async (values) => {
    try {
        await validationSchema.validate(values, { abortEarly: false
});
    } catch (err) {
        if (err instanceof yup.ValidationError) {
            return err.inner.reduce((errors, error) => {
                errors[error.path] = error.message;
                return errors;
            }, {});
        }
    }
};
```

Функція повертає об'єкт з помилками, де ключі відповідають іменам полів, а значення - текстам помилок.

Обробник відправки отримує очищені значення форми та API для управління формою. Ось фрагмент коду який демонструє базову обробку:

```
const onSubmit = async (values, form) => {
    alert("Form submitted successfully!");
    form.reset();
};
```

Об’єкт form надає методи для скидання форми, програмної зміни значень полів та інших операцій.

Основною перевагою React Final Form є мінімальна кількість повторних рендерів завдяки системі підписок. Кожне поле підписується лише на зміни свого стану, що забезпечує високу продуктивність навіть для великих форм. Бібліотека також надає гнучкий API для складних сценаріїв валідації та управління станом форми.

Повний код файлу FormExample.js для програми що використовує Final Form наведено в додатку К.

3.6 Розробка форм з використанням Redux Form

Redux Form являє собою бібліотеку, яка інтегрує управління формами безпосередньо в Redux store. Основною особливістю є те, що весь стан форми зберігається в глобальному сховищі додатка, що забезпечує централізоване управління даними форм.

Redux Form використовує патерн Higher-Order Component для підключення форми до Redux store. Ось фрагмент коду який демонструє основну конфігурацію:

```
export default reduxForm({
  form: 'exampleForm',
  validate,
  destroyOnUnmount: false,
  enableReinitialize: false
})(FormExample);
```

HOC reduxForm приймає об’єкт конфігурації, де form - унікальний ідентифікатор форми в store, validate - функція валідації, а додаткові опції керують життєвим циклом форми.

На відміну від інших підходів, Redux Form використовує систему render-функцій для створення полів. Кожен тип поля має свою окрему render-функцію. Ось фрагмент коду який показує базову структуру поля:

```
    const renderField = ({ input, label, type, meta: { touched, error },
placeholder, min, max }) => (
      <div>
        <label className="form-label">{label}</label>
        <input {...input} type={type} className="form-field"
placeholder={placeholder} min={min} max={max} />
        {touched && error && <span className="form-error-
message">{error}</span>}
      </div>
    );
```

Render-функція отримує об'єкти input та meta, що містять всі необхідні пропи та метадані поля. Поля створюються за допомогою компонента Field, який зв’язує render-функції з Redux store. Ось фрагмент коду який демонструє використання:

```
    <Field name="name" type="text" component={renderField} label="Name"
placeholder="James" />
    <Field name="email" type="text" component={renderField} label="Email"
placeholder="example@mail.com" />
```

Компонент Field передає всі додаткові пропи до render-функції, що забезпечує гнучкість конфігурації.

Для групи радіо-кнопок створюється спеціалізована render-функція з власною логікою. Ось фрагмент коду який показує реалізацію:

```
    const renderRadioGroup = ({ input, options, label, meta: { touched,
error } }) => (
      <div>
        <div className="form-label">{label}</div>
        {options.map((option) => (
          <div className="form-selection-wrapper" key={option.value}>
            <input
              {...input}
              type="radio"
```

```
                value={option.value}
                checked={input.value === option.value}
                className="form-radio"
              />
              <label className="form-selection-
label">{option.label}</label>
            </div>
          ))}
          {touched && error && <span className="form-error-
message">{error}</span>}
        </div>
      );
```

Для чекбоксів необхідна ручна обробка масиву значень через кастомний onChange обробник.

Redux Form надає метод initialize для встановлення початкових значень програмно. Ось фрагмент коду який демонструє використання:

```
React.useEffect(() => {
  initialize({
    number: 0,
    color: '#EDBB36',
    rating: 10,
    season: 'summer',
    hobbies: []
  });
}, [initialize]);
```

Цей підхід дозволяє динамічно встановлювати значення полів після монтування компонента.

Функція валідації працює синхронно та повертає об'єкт помилок. Обробник відправки отримує доступ до стану валідації через проп valid. Ось фрагмент коду який показує логіку відправки:

```
const onSubmit = (values) => {
  if (!valid) {
    return;
  }
  alert("Form submitted successfully!");
  console.log(values);
};
```

Основною перевагою Redux Form є повна інтеграція з Redux екосистемою. Всі зміни в формі автоматично відображаються в Redux DevTools, що спрощує

налагодження. Стан форми доступний для читання з будь-якого компонента додатка через селектори Redux Form, що забезпечує потужні можливості для створення складних інтерфейсів з взаємозалежними формами.

Повний код файлу FormExample.js для програми що використовує Redux Form наведено в додатку Л.

У додатку М наведено інтерфейс розробленого застосунку (ідентичний для усіх п’яти варіантів).

3.7 Опис програмного підходу до проведення експериментів

В ході проведення експеримента ми маємо отримати наступні показники: кількість рядків коду, час першого рендерингу, кількість ререндерів, мінімальний та максимальний об’єм споживання пам’яті, час відгуку INP, розмір бібліотеки та залежностей у бандлі.

Для визначення кількості рядків коду ми використаємо інструмент Prettier для автоматичного форматування коду з єдиними налаштуваннями форматування. Це дозволить уникнути впливу стилістичних відмінностей на загальну кількість рядків, що забезпечить об’єктивність вимірювань.

Для вимірювання показників продуктивності, а саме часу першого рендерингу та кількості ререндерів ми будемо використовувати вбудовані інструменти React-екосистеми, такі як React Profiler API для профілювання компонентів та хук useEffect для відстеження життєвого циклу компонентів.

Для збору даних про ефективність використання пам’яті та час відгуку додатку (INP) буде застосовуватися інструментарій Chrome DevTools. Показники мінімального та максимального споживання пам’яті можна зафіксувати через Memory Profiler, а час відгуку INP вимірювався засобами Performance панелі для оцінки якості взаємодії користувача з інтерфейсом.

Для аналізу розміру бібліотеки та залежностей у фінальному бандлі використовуватиметься сервіс BundlePhobia (bundlephobia.com), який надає детальну інформацію про вагу npm-пакетів та їх вплив на розмір збірки.

4 ОПИС ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Проведення експериментальних досліджень

Після збору необхідної інформації та проведення підготовчої роботи (планування експериментів, проєктування та реалізації тестових програм), ми можемо приступити до проведення експериментальних досліджень.

Розроблені програми реалізують однакову форму, застосовуючи усі п’ять досліджуваних підходів: використання «чистого» React, бібліотек Formik, React Hook Form, Final Form та Redux Form.

Для кожного варіанту програми було проведено заміри за кожним визначеним критерієм:

- кількістю рядків коду;
- часом першого рендерингу (мс);
- кількістю ререндерів;
- мінімальним об’ємом споживання пам’яті (Мб);
- максимальним об’ємом споживання пам’яті (Мб);
- часом відгуку INP (мс);
- розміром бібліотеки та її залежностей у бандлі (Кб).

Експериментальні дослідження проводилися в контрольованому середовищі з дотриманням однакових умов для всіх досліджуваних підходів. Це забезпечило об'єктивність порівняння та достовірність отриманих результатів.

Тестування проводилося на комп’ютері з наступними характеристиками та програмним забезпеченням:

- процесор: AMD Ryzen 7 5800H (8 ядер 16 потоків)
- оперативна пам'ять: 16 GB
- операційна система: Windows 10 Pro 22H2
- браузер: Google Chrome 131.0.6778.265
- Node.js: v16.13.0
- React: 18.2.0

Першим кроком стало вимірювання кількості рядків коду, необхідних для реалізації кожної версії форми. Цей параметр відображає складність реалізації форми в межах кожного з підходів. Менша кількість рядків зазвичай свідчить про вищу декларативність або кращу абстракцію, що спрощує підтримку коду.

Для забезпечення об’єктивності вимірювань весь код кожної реалізації було автоматично відформатовано за допомогою Prettier з єдиними налаштуваннями форматування. Це дозволило уникнути впливу стилістичних відмінностей на загальну кількість рядків.

Як видно з діаграми, що наведена нижче (див. рис. 4.1), найменшу кількість рядків коду має реалізація з використанням Redux Form. Натомість Final Form виявився найменш лаконічним.

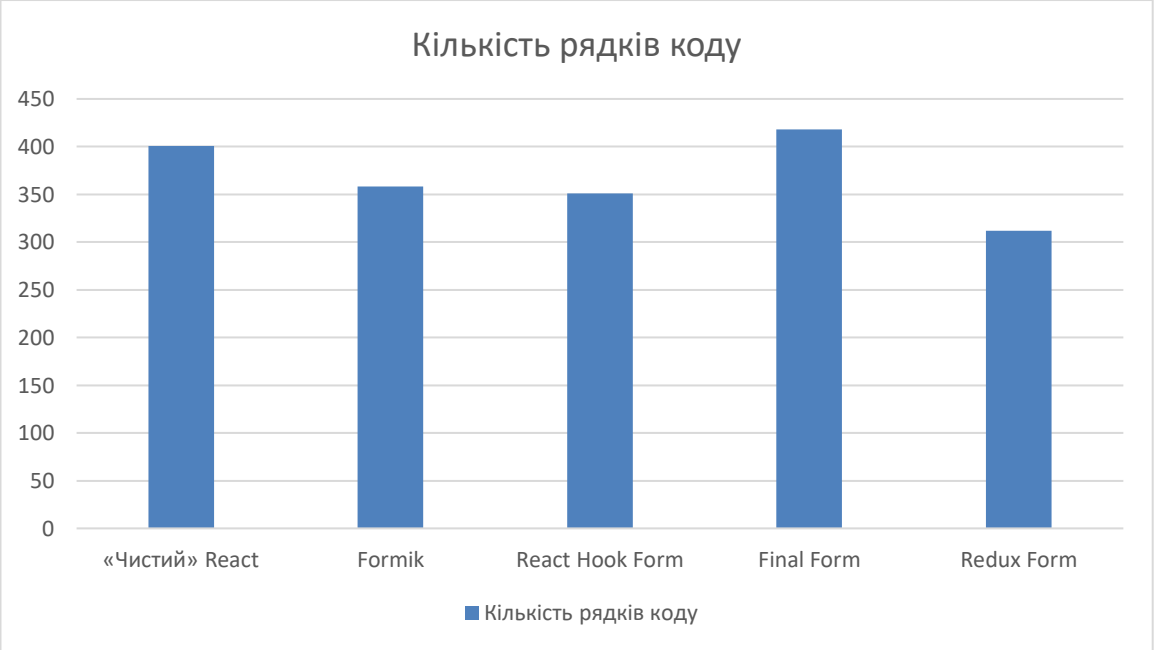

Рисунок 4.1 - Результати вимірювання кількості рядків коду (виконано самостійно)

Час першого рендерингу є критичним показником для продуктивності початкового завантаження форми. Він показує, скільки часу потрібно для відображення компонента після його ініціалізації. Для вимірювання цього показника використовувався React Profiler, що дозволив точно зафіксувати момент монтування компонента та тривалість його початкового рендерингу.

На діаграмі (див. рис. 4.2) видно, що найменший час першого рендеру показали «чистий» React та React Hook Form. Натомість, Redux Form мав найвищі показники, що пов’язано з додатковим рівнем абстракції та ініціалізацією внутрішніх механізмів, додатковими накладними витратами на ініціалізацію глобального стану, підключення до Redux store та створення HOC-обгорток для кожного поля форми.

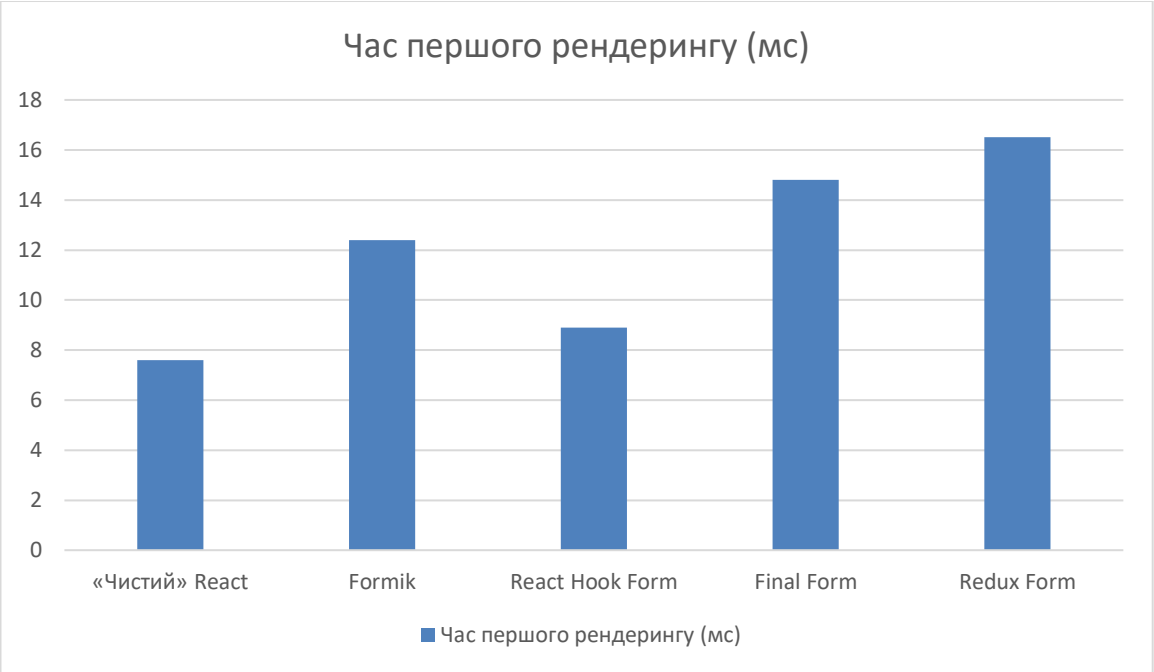

Рисунок 4.2 - Результати вимірювання часу першого рендерингу (виконано самостійно)

Кількість повторних рендерів напряму впливає на продуктивність додатку, особливо при великій кількості користувацьких взаємодій. Цей параметр дозволяє оцінити, наскільки ефективно бібліотека управляє оновленням стану. Для кожного підходу були застосовані адаптовані способи вимірювання, оскільки механізми керування станом у різних бібліотеках відрізняються.

Наступна діаграма (див. рис. 4.3) демонструє, що React Hook Form та Final Form показали найменшу кількість ререндерів завдяки оптимізованому підходу до оновлення окремих полів. Redux Form мав найбільшу кількість ререндерів, що може створювати додаткове навантаження на DOM.

Рисунок 4.3 - Результати вимірювання кількості ререндерів (виконано самостійно)

Наступна діаграма (див. рис. 4.4) ілюструє показники вимірювань мінімального та максимального об’ємів споживання пам’яті. Ці метрики характеризують стабільність та оптимальність використання ресурсів додатком у процесі виконання.

Для отримання цих даних кожна форма запускалась у браузері Google Chrome, після чого здійснювалася послідовність типової взаємодії з користувацьким інтерфейсом: заповнення полів, перемикання фокусів, валідація тощо. Після цього, за допомогою вкладки «Performance» у Chrome DevTools, фіксувалися мінімальні та максимальні значення споживання пам’яті протягом сеансу. Це дозволило оцінити не лише середній рівень використання ресурсів, а й пік навантаження під час активної роботи.

Як видно з діаграми, найбільший розрив між мінімальним та максимальним обсягом пам’яті спостерігається у реалізації з використанням Final Form – майже 22 Мб у піковому значенні, що може свідчити про менш ефективне управління ресурсами під час взаємодії з формою. Найменше споживання виявлено у варіанті на «чистому» React, однак і там фіксується помітне зростання при активному використанні. Formik та React Hook Form показали схожі результати зі стабільним

профілем пам’яті. Найвище мінімальне споживання серед усіх продемонстрував Redux Form, в той же час його максимальне значення залишається на середньому рівні, що вказує на відносно збалансоване використання пам’яті.

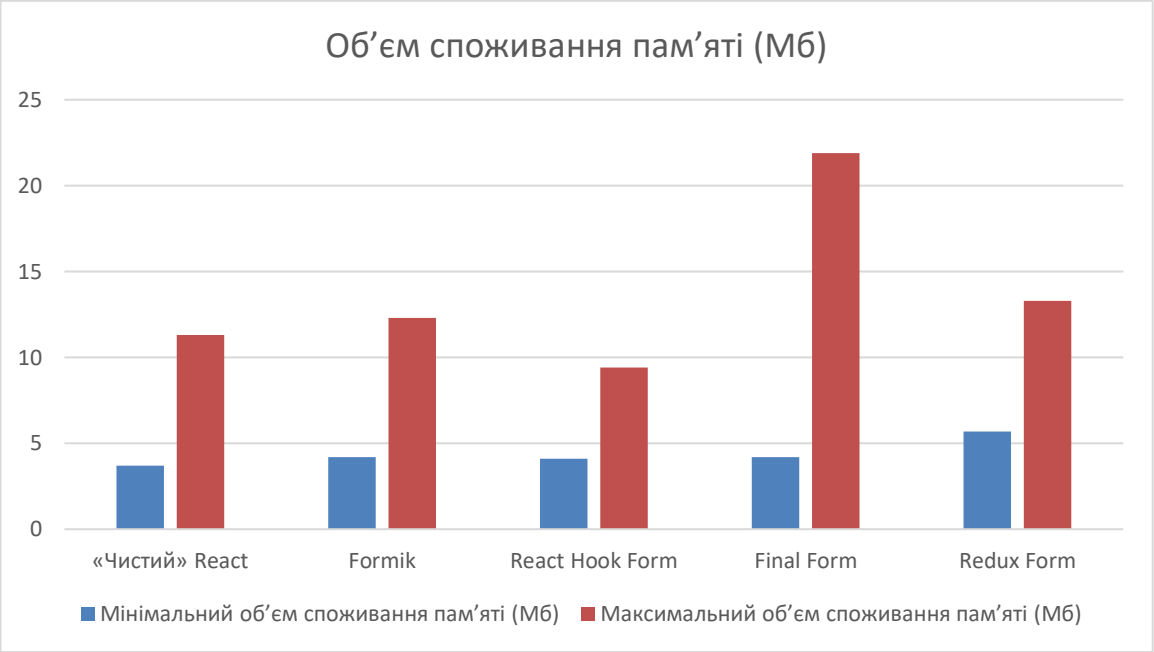

Рисунок 4.4 - Результати вимірювання об’єму споживання пам’яті (виконано самостійно)

INP (interaction to next paint) є важливою метрикою, яка характеризує затримку між дією користувача та відображенням змін в інтерфейсі. Чим менше значення INP, тим краща суб’єктивна «плавність» і швидкість роботи форми.

Для вимірювання INP використовувався Chrome DevTools, вкладка «Performance», де під час взаємодії з формою фіксувалися ключові події (наприклад, keydown, input, click) та час, що минув до наступного візуального оновлення. Кожен варіант форми тестувався за однаковою послідовністю дій, що дозволило отримати порівнювані результати.

Результати вимірювань продемонстровано на діаграмі «Час відгуку INP» (див. рис. 4.5). Найгірші результати, тобто найбільший час відгуку, показали бібліотеки Formik та Final Form. «Чистий» React, бібліотека React Hook Form і бібліотека Redux Form продемонстрували кращі результати з меншим часом відгуку.

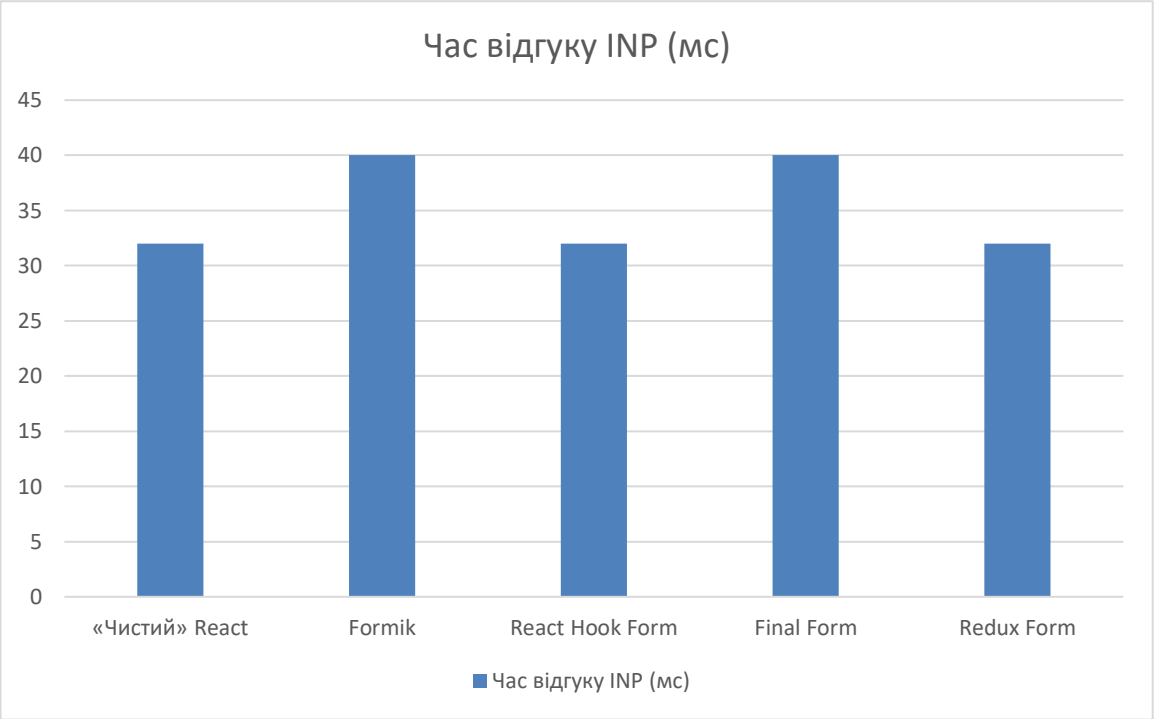

Рисунок 4.5 - Результати вимірювання часу відгуку INP (виконано самостійно)

Розмір бандлу має значення при доставці фронтенду кінцевому користувачу. Чим більше бібліотек та залежностей використовується, тим повільніше завантаження додатку, особливо за умов повільного інтернет-з'єднання або на мобільних пристроях.

На діаграмі, наведеній нижче (див. рис. 4.6), представлено вклад кожної бібліотеки та її залежностей у розмір фінального бандлу додатку, зібраного за допомогою Webpack у production-режимі.

Найменший вплив на розмір бандлу продемонструвала бібліотека Final Form. Найбільший розмір спостерігається у Redux Form, що зумовлено її залежністю від Redux та більш складною архітектурою. Formik і React Hook Form знаходяться в середньому діапазоні.

У випадку з «чистим» React додатковий розмір у бандлі відсутній, оскільки не підключаються жодні зовнішні бібліотеки для роботи з формами – використовується лише базовий функціонал React, який і так входить до основного бандлу застосунку. Це дозволяє досягти мінімального розміру бандлу, хоча може вимагати більше ручної роботи з обробкою стану та валідації.

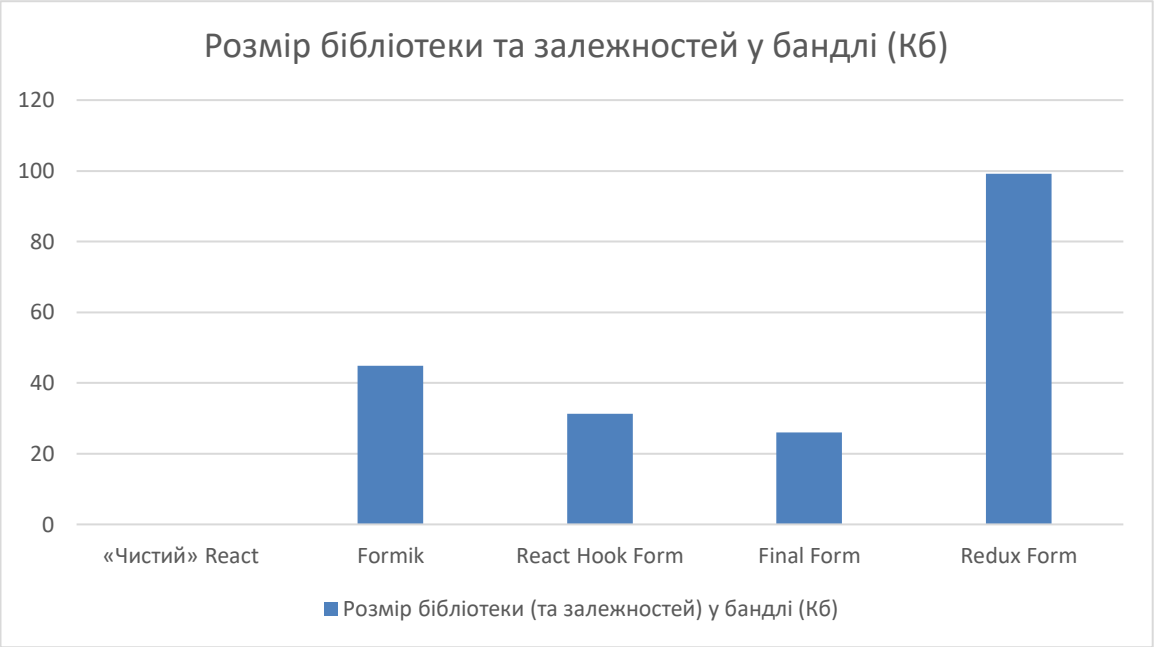

Рисунок 4.6 - Результати вимірювання розміру бібліотеки у бандлі (виконано самостійно)

Отже, тепер, маючи показники усіх вимірювань, ми можемо приступити до аналізу отриманих результатів.

4.2 Аналіз отриманих результатів

Після проведення експериментальних вимірювань за всіма визначеними метриками (кількість рядків коду, час першого рендерингу, кількість ререндерів, мінімальний та максимальний об’єм споживання пам’яті, час відгуку INP, розмір бібліотеки та її залежностей у бандлі), ми маємо змогу виконати аналіз отриманих даних з метою визначення найбільш ефективного підходу до реалізації форм у React. Для цього ми застосуємо метод лінійної адитивної згортки, який дозволить отримати інтегральну оцінку ефективності кожного підходу на основі декількох нормованих метрик із відповідними ваговими коефіцієнтами.

На основі проведених експериментальних вимірювань було сформовано векторний опис кожної альтернативи у просторі визначених критеріїв. Кожна альтернатива представлена як вектор з п’яти компонентів, що відповідають виміряним показникам ефективності.

У таблиці 4.1 наведено векторний опис альтернатив.

Таблиця 4.1 - Векторний опис альтернатив

Метрики	«Чистий» React	Formik	React Hook Form	Final Form	Redux Form
Кількість рядків коду	401	358	351	418	312
Час першого рендерингу (мс)	7,6	12,4	8,9	14,8	16,5
Кількість ререндерів	137	173	4	12	183
Мінімальний об’єм споживання пам’яті (Мб)	3,7	4,2	4,1	4,2	5,7
Максимальний об’єм споживання пам’яті (Мб)	11,3	12,3	9,4	21,9	13,3
Час відгуку INP (мс)	32	40	32	40	32
Розмір бібліотеки (та залежностей) у бандлі (Кб)	0	44,8	31,3	26	99,1

Нормалізуємо усі шкали за формулою:

$$x = \frac{x - max(x)}{\min(x) - \max(x)}$$

Для кожної метрики для кожної альтернативи вираховуються значення відносно мінімального та максимального значень у таблиці, що дозволяє порівнювати їх за однаковою шкалою.

Нормалізовані значення наведені у таблиці 4.2.

Таблиця 4.2 - Нормалізовані значення

Метрики	«Чистий» React	Formik	React Hook Form	Final Form	Redux Form
Кількість рядків коду	0,16	0,57	0,63	0,00	1,00
Час першого рендерингу	1,00	0,46	0,85	0,19	0,00
Кількість ререндерів	0,26	0,06	1,00	0,96	0,00
Мінімальний об’єм споживання пам’яті	1,00	0,75	0,80	0,75	0,00
Максимальний об’єм споживання пам’яті	0,85	0,77	1,00	0,00	0,69
Час відгуку INP	1,00	0,00	1,00	0,00	1,00
Розмір бібліотеки (та залежностей) у бандлі	1,00	0,55	0,68	0,74	0,00

Для визначення вагових коефіцієнтів критеріїв було використано метод експертного оцінювання з залученням кваліфікованого спеціаліста в галузі веб-розробки. Даний підхід дозволяє врахувати практичний досвід та експертну думку для забезпечення максимальної точності оцінки кожного критерію. Експерт був обраний за наступними критеріями:

- наявність практичного досвіду веб-розробки;
- спеціалізація в розробці React-додатків;
- досвід роботи з різними бібліотеками управління формами;
- участь у великих комерційних проектах;
- наявність технічного бекграунду в суміжних технологіях.

Ці критерії були обрані для забезпечення об’єктивності оцінки та зменшення ризику суб’єктивних похибок. Завдяки такому підходу вдалося залучити експерта, який має широкий практичний досвід і здатний оцінити всі аспекти ефективності компонентів веб-додатків.

Експерту було запропоновано оцінити відносну важливість кожного критерію в контексті розробки форм для реальних веб-додатків. Особлива увага приділялася обґрунтуванню кожної оцінки на основі практичного досвіду та розуміння потреб сучасних веб-застосунків.

В результаті аналізу експертних оцінок було визначено, що час першого рендерингу та кількість ререндерів є найкритичнішими показниками, оскільки вони безпосередньо впливають на користувацький досвід та сприйняття швидкодії додатку. Цим критеріям було присвоєно найвищі вагові коефіцієнти - 0,2.

Також доволі важливими критеріями є максимальний об’єм споживання пам’яті та час відгуку INP, оскільки ці показники суттєво впливають на плавність роботи інтерфейсу та загальну продуктивність додатку, особливо на пристроях з обмеженими ресурсами. Саме тому їм було присвоєно вагові коефіцієнти 0,15.

Кількість рядків коду, мінімальний об’єм споживання пам’яті та розмір бібліотеки та її залежностей у бандлі отримали коефіцієнт 0,1, оскільки ці показники впливають на довгострокову підтримку проекту, швидкість розробки та

початкове завантаження додатку, але мають менший безпосередній вплив на користувацький досвід порівняно з іншими показниками.

У таблиці 4.3 наведені значення, приведені за ваговими коефіцієнтами.

Таблиця 4.3 - Значення, приведені за ваговими коефіцієнтами

Метрики	«Чистий» React	Formik	React Hook Form	Final Form	Redux Form	Вага
Кількість рядків коду	0,02	0,06	0,06	0,00	0,10	0,1
Час першого рендерингу	0,20	0,09	0,17	0,04	0,00	0,2
Кількість ререндерів	0,05	0,01	0,20	0,19	0,00	0,2
Мінімальний об’єм споживання пам’яті	0,10	0,08	0,08	0,08	0,00	0,1
Максимальний об’єм споживання пам’яті	0,13	0,12	0,15	0,00	0,10	0,15
Час відгуку INP	0,15	0,00	0,15	0,00	0,15	0,15
Розмір бібліотеки (та залежностей) у бандлі	0,10	0,05	0,07	0,07	0,00	0,10
Оцінка	0,74	0,40	0,88	0,38	0,35	

На основі результатів експерименту з порівняння п’яти підходів до розробки форм у React, можна зробити певні висновки.

React Hook Form демонструє найкращу загальну ефективність з оцінкою 0,88, що робить цю бібліотеку оптимальним вибором для більшості проєктів. Ключовими перевагами є мінімальна кількість ререндерів (лише 4), що значно покращує продуктивність додатку, помірний розмір бібліотеки (31,3 Кб) та ефективне використання пам'яті з максимальним споживанням 9,4 Мб.

«Чистий» React посідає друге місце з оцінкою 0,74, що підтверджує його життєздатність для простих форм. Головною перевагою є відсутність додаткових залежностей, що зменшує розмір бандлу, та найшвидший час першого рендерингу (7,6 мс). Однак значна кількість ререндерів (137) може негативно впливати на продуктивність складних форм.

Formik займає третє місце з оцінкою 0,40, демонструючи середні показники серед досліджуваних бібліотек. Основними недоліками є підвищений час першого рендерингу (12,4 мс) та велика кількість ререндерів (173), що може значно погіршувати користувацький досвід у великих формах. Водночас бібліотека показує прийнятні результати за споживанням пам’яті і розміром у бандлі (44,8 Кб).

Final Form посідає четверте місце з оцінкою 0,38, маючи суттєві проблеми з ефективністю. Критичними недоліками є найгірше споживання пам'яті з максимальним показником 21,9 Мб та повільний час першого рендерингу (14,8 мс). Позитивним аспектом є відносно невелика кількість ререндерів (12) та компактний розмір бібліотеки (26 Кб).

Redux Form демонструє найнижчі показники з оцінкою 0,35, що робить його найменш ефективним рішенням серед досліджуваних. Основними проблемами є найбільший розмір бібліотеки та залежностей (99,1 Кб), найповільніший час першого рендерингу (16,5 мс) та велика кількість ререндерів (183). Ці показники свідчать про застарілість архітектури та необхідність переходу на більш сучасні альтернативи.

ВИСНОВКИ

В ході роботи була досліджена ефективність різних підходів до створення форм у React. Для цього було проведено аналіз предметної галузі, огляд й аналіз літературних та наукових джерел, сформульована постановка задачі, спроєктована архітектура програмного забезпечення, розроблено тестові застосунки та проведено експериментальне дослідження.

Аналіз предметної галузі продемонстрував важливість форм як фундаментального компонента сучасних веб-сайтів, що забезпечує механізм взаємодії між користувачем та системою. При цьому було виявлено ряд сучасних викликів, з якими стикаються при створенні веб-сайтів загалом, та форм зокрема.

Було проаналізовано React як одну з найпопулярніших бібліотек для розробки користувацьких інтерфейсів, виявлені його особливості та переваги.

В ході аналізу було виявлено основні підходи до розробки форм: використання базового функціоналу React («чистий React») та застосування спеціалізованих бібліотек. Серед найпопулярніших бібліотек для роботи з формами були виділені: Formik, React Hook Form, Final Form та Redux Form. Кожен з цих підходів має свої особливості та переваги. Також було розглянуто спеціалізовані бібліотеки для валідації даних форм - Yup та Zod.

В рамках огляду та аналізу літературних і наукових джерел було опрацьовано ряд наукових праць, що стосуються методології оцінки та порівняння програмних рішень. Зокрема, було адаптовано підходи до оцінки UI-бібліотек, фреймворків та програмних бібліотек для порівняння різних методів розробки форм. Приділено увагу методам вимірювання та профілювання продуктивності React-застосунків.

На етапі постановки задачі було сформульовано мету дослідження та визначено які задачі необхідно виконати для досягнення цієї мети. Було визначено методи дослідження, обмеження та необхідні ресурси.

Нами спланована експериментальна частина дослідження, в рамках якої було визначено сім ключових критеріїв оцінки: кількість рядків коду, час першого рендерингу, кількість ререндерів, мінімальний та максимальний об’єм споживання пам’яті, час відгуку INP, розмір бібліотеки та залежностей у бандлі. Для порівняння

було обрано п'ять альтернативних підходів: чистий React, Formik, React Hook Form, Final Form та рішення з використанням Redux.

Було спроектовано та реалізовано п’ять тестових застосунків з використанням кожного з досліджуваних підходів для подальшого проведення експериментів.

В рамках експериментального дослідження було проведено багатокритеріальний аналіз різних підходів до розробки форм у React. На основі експериментальних даних та з використанням методу лінійної адитивної згортки з нормуючими та ваговими коефіцієнтами було проведено комплексний аналіз ефективності кожного підходу.

Результати дослідження свідчать, що React Hook Form є оптимальним вибором для розробки форм у React, забезпечуючи найкращий баланс між продуктивністю, зручністю використання та розміром бандлу. Чистий React, хоча і посідає друге місце, може бути доцільним для простих форм, де важлива повна кастомізація. Formik та Final Form показали середні результати, тоді як Redux Form виявився найменш ефективним рішенням серед досліджених підходів.

Таким чином, проведене дослідження дозволило не лише визначити найефективніший підхід до розробки форм у React, але й виявити специфічні переваги кожного з розглянутих рішень.

Практична значимість отриманих результатів полягає в тому, що вони можуть бути використані розробниками та архітекторами програмного забезпечення при виборі оптимального підходу до реалізації форм у React-застосунках.

Подальші дослідження можуть бути спрямовані на розширення набору критеріїв оцінки, включення інших популярних бібліотек для роботи з формами, а також на дослідження впливу різних факторів на ефективність різних підходів.

За результатами проведеного дослідження підготовлено тези доповіді, які було представлено на молодіжній конференції MIT@AISm-2025 в рамках 1-ї Міжнародної науково-практичної конференції «СУЧАСНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ MIT@AIS-2025».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. How many websites are there in the world? *Siteefy.* URL: https://siteefy.com/how-many-websites-are-there/ (дата звернення: 07.01.2025).

2. Лозиченко А. В., Мельнікова Р. В. Дослідження методів виявлення помилок проектування сайтів. *Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління - Тези доповідей дванадцатої міжнародної науково-технічної конференції.* 2022. Т. 2.

3. React - the library for web and native user interfaces. *React.* URL: https://react.dev/ (дата звернення: 07.01.2025).

4. Hámori F. The history of React.js on a timeline. *RisingStack.* URL: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ (дата звернення: 07.01.2025).

5. Anderson L. The history of React.js: a story of innovation and community. *LinkedIn.* URL: https://www.linkedin.com/pulse/history-reactjs-story-innovation-community-l-anderson (дата звернення: 07.01.2025).

6. Akinyemi F. Top react form libraries for efficient form creation. *DEV Community.* URL: https://dev.to/femi_akinyemi/top-10-react-form-libraries-for-efficient-form-creation-hp2 (дата звернення: 07.01.2025).

7. Minvielle L. Top 5 react form libraries for developers. *WeAreDevelopers.* URL: https://www.wearedevelopers.com/magazine/react-form-libraries (дата звернення: 07.01.2025).

8. Kamunya T. 8 best react form libraries for developers. *Geekflare.* URL: https://geekflare.com/dev/best-react-form-libraries/ (дата звернення: 07.01.2025).

9. Formik: Build forms in React, without the tears. *Formik.* URL: https://formik.org/ (дата звернення: 07.01.2025).

10. React Hook Form - performant, flexible and extensible form library. *React Hook Form.* URL: https://react-hook-form.com/ (дата звернення: 07.01.2025).

11. React Final Form - High performance subscription-based form state management for React. *Final Form.* URL: https://final-form.org/react/ (дата звернення: 07.01.2025).

12. Getting started with redux-form. *Redux Form.* URL: https://redux-form.com/8.3.0/docs/gettingstarted.md/ (дата звернення: 07.01.2025).

13. Yup. *Npm.* URL: https://www.npmjs.com/package/yup (дата звернення: 07.01.2025).

14. Zod: TypeScript-first schema validation with static type inference. *Zod.* URL: https://zod.dev/?id=introduction (дата звернення: 07.01.2025).

15. Kainu I. Optimization in React.js: methods, tools, and techniques to improve performance of modern web applications : Бакалаврська робота. 2022. URL: https://trepo.tuni.fi/handle/10024/140258 (дата звернення: 07.01.2025).

16. Salonen S. Evaluation of UI Component Libraries in React Development : Магістерська робота. 2022. URL: https://trepo.tuni.fi/handle/10024/148025 (дата звернення: 07.01.2025).

17. de la Mora F. L., Nadi S. Which library should I use? *ICSE '18: 40th International Conference on Software Engineering*, м. Gothenburg Sweden. New York, NY, USA, 2018. URL: https://doi.org/10.1145/3183399.3183418 (дата звернення: 07.01.2025).

18. Chhetri K. Comparative Study of Front-end Frameworks : React and Angular : Бакалаврська робота. 2024. URL: https://www.theseus.fi/handle/10024/865488 (дата звернення: 07.01.2025).

19. Kaur G., Tiwari R. G. Comparison and Analysis of Popular Frontend Frameworks and Libraries: An Evaluation of Parameters for Frontend Web Development. *2023 4th International Conference on Electronics and Sustainable Communication Systems (ICESC)*, м. Coimbatore, India, 6–8 лип. 2023 р. 2023. URL: https://doi.org/10.1109/icesc57686.2023.10192987 (дата звернення: 07.01.2025).

20. Sekhar Emmanni P. Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development. *International Journal of Science and Research (IJSR)*. 2023. Т. 12, № 6. С. 2971–2974. URL: https://doi.org/10.21275/sr24401230015 (дата звернення: 07.01.2025).

21. Pronina D., Kyrychenko I. Comparison of Redux and React Hooks methods in terms of performance. *International Conference on Computational Linguistics and*

Intelligent Systems. 2022. URL: https://api.semanticscholar.org/CorpusID:250625540 (дата звернення: 07.01.2025).

22. Дудник О. О. GitHub - OleksandrDudnykNure/2025_M_PI_IPZm-23-2_Dudnyk_O_O. GitHub. URL: https://github.com/OleksandrDudnykNure/2025_M_PI_IPZm-23-2_Dudnyk_O_O (дата звернення: 12.06.2025).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

2. Лозиченко А. В., Мельнікова Р. В. Дослідження методів виявлення помилок проектування сайтів. *Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління - Тези доповідей дванадцатої міжнародної науково-технічної конференції.* 2022. Т. 2.

21. Pronina D., Kyrychenko I. Comparison of Redux and React Hooks methods in terms of performance. *International Conference on Computational Linguistics and Intelligent Systems.* 2022. URL: https://api.semanticscholar.org/CorpusID:250625540 (дата звернення: 07.01.2025).