ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методика оптимізації web-запитів і баз даних для підвищення продуктивності web-застосунку для лабораторних досліджень»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми Інженерія програмного забезпечення

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Богдан ВИГОВСЬКИЙ
(підпис)

Виконав: здобувач вищої освіти група ПДМ-63

Богдан ВИГОВСЬКИЙ

Керівник: Наталя ТРІНТІНА
к.т.н., доцент

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Виговському Богдану Сегійовичу

1. Тема кваліфікаційної роботи: «Методика оптимізація web-запитів і баз даних для підвищення продуктивності web-застосунку для лабораторних досліджень»

керівник кваліфікаційної роботи Наталя ТРІНТІНА, к.т.н., доцент каф. Інтернет Технологій,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, методи оптимізації веб-застосунку, оптимізація бази даних.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методів оптимізації веб-застосунку.
2. Принципи і методи комплексної оптимізації веб-застосунку.
3. Розробка методу оптимізації веб-запитів і бази даних.

5. Перелік графічного матеріалу: *презентація*

1. Мета, об’єкта та предмет дослідження.
2. Актуальність роботи
3. Порівняльний аналіз.
4. Практичний результат бібліотеки reacr redux.
5. Практичний результат бібліотеки Axios.
6. Практичний результат бібліотеки React-Router-DOM та бібліотеки use-Form.
7. Результат моделювання.
8. Математична модель для дослідження кількості запитів на секунду(RPS – Requests Per Second).
9. Модель із використанням часових вікон(Sliding Window).
10. Середній час обробки одного запиту.
11. Висновки.
12. Публікації та апробація роботи.

6. Дата видачі завдання«16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.24	
2	Вивчення матеріалів для оптимізації веб-застосунку та оптимізації бази даних	24.10-29.10.24	
3	Дослідження методів оптимізації веб-застосунку	30.10-01.11.24	
4	Аналіз оптимізації за допомогою бібліотеки Million.js	02.11-05.11.24	
5	Дослідження шляхів оптимізації бази даних	06.11-16.11.24	
6	Застосування оптимізації бази даних	17.11-20.11.24	
7	Оформлення роботи: вступ, висновки, реферат	21.11-26.11.24	
8	Розробка демонстраційних матеріалів	26.11-27.11.24	
9	Попередній захист роботи	28.11-11.12.2024	

Здобувач вищої освіти _____ Богдан ВИГОВСЬКИЙ
(підпис)

Керівник кваліфікаційної роботи _____ Наталя ТРІНТІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра 85 с., 2 табл., 61 рис., 5 дод., 55 джерела.

Об`єкт дослідження – веб-додаток, що використовується для обробки та зберігання лабораторних досліджень.

Предмет дослідження – інтерфейс користувача і база даних веб-додатку, що використовується для обробки та зберігання лабораторних досліджень та їх взаємодія.Мета роботи – оптимізувати веб-додаток для зберігання та обробки результатів лабораторного дослідження.

Методи дослідження — оптимізація інтерфейсу користувача за допомогою використання сторонніх бібліотек, зменшення кількості написання коду та оптимізація взаємодії серверної частини з інтерфейсом користувача. Також оптимізовано роботу серверної частини, за допомогою структурування бази даних та використання блокчейн технологій у процесі розробки.

На основі отриманих даних ми можемо зробити висновок, що розглянуті підходи до оптимізації значно покращують стабільність роботи веб-додатку, покращують безпеку даних, що зберігаються в хмарному сховищі та полегшують взаємодію користувача з веб-додатком, що значно збільшує довіру користувача.

В результаті проведених досліджень, ми можемо вибрати оптимальний технологічний стек для розробки веб-додатку для зберігання та обробки лабораторних досліджень та отримуємо стабільну модель поведінки додатку з мінімальною вірогідністю виникнення критичних помилок та збоїв у роботі веб-додатку.

ABSTRACT

Text part of the qualification work for obtaining a master's degree 80 p., 2 tables, 61 figures, 5 appendix, 55 sources.

The object of the study is a web application used for processing and storing laboratory research.

The subject of the study is the user interface and database of the web application used for processing and storing laboratory research and their interaction. The purpose of the work is to optimize the web application for storing and processing laboratory research results.

Research methods - optimization of the user interface using third-party libraries, reducing the amount of code writing and optimizing the interaction of the server part with the user interface. The operation of the server part was also optimized by structuring the database and using blockchain technologies in the development process.

Based on the data obtained, we can conclude that the considered approaches to optimization significantly improve the stability of the web application, improve the security of data stored in the cloud storage and facilitate user interaction with the web application, which significantly increases user trust.

As a result of the research, we can choose the optimal technology stack for developing a web application for storing and processing laboratory research and obtain a stable application behavior model with a minimal probability of critical errors and failures in the web application.

ЗМІСТ

ВСТУП

З розвитком Інтернету веб-системи знаходять все більше використання в повсякденному житті компаній і споживачів. Із збільшенням використання веб-додатків їх надійність є важливою проблемою. Веб-програми популярні для ділових операцій, оскільки використання веб-програми не потребує встановлення великого та складного програмного забезпечення на комп'ютері користувача. Користувач може отримати доступ до більшості веб-додатків за допомогою програмного забезпечення браузера. Крім того, більша частина обробки заявок відбувається на централізованому сервері. Таким чином, бізнес може легко оновлювати та підтримувати свою програму, вносячи зміни в програму на сервері. Після кожного оновлення нову версію програми не потрібно встановлювати на багатьох комп'ютерах користувачів. Веб-додатки широко використовуються як системи електронної пошти, системи онлайн-покупок та аукціонів, вікі та блоги. Крім того, відомі компанії-виробники програмного забезпечення розробляють веб-додатки для створення та редагування документів в Інтернеті (наприклад, офісне програмне забезпечення).

Тому питання надійності та оптимізації роботи веб-додатку на даний момент є одним з найважливіших під час планування та розробки. Оптимізації підлягає не лише інтерфейс користувача, але і база даних має бути захищеною та структурованою. В умовах сучасного розвитку ІТ-технологій та велику вірогідність втрати особистих даних, дуже важливо захистити усі вразливі вузли та зробити структуру в якій будуть окремо зберігатись дані кожного користувача.

Мета роботи – оптимізувати веб-додаток для зберігання та обробки результатів лабораторного дослідження.

Завдання роботи – оптимізація роботи інтерфейсу користувача, бази даних та їх взаємодії під час роботи з додатком.

Об'єкт дослідження – веб-додаток, що використовується для обробки та зберігання лабораторних досліджень.

Предмет роботи – методи оптимізації веб-додатку, що використовується для обробки та зберігання лабораторних досліджень.

У цій роботі розглянуто та наведено сучасні методи оптимізації та вплив тих та інших методів на роботу веб додатку. На прикладі веб-додатку для зберігання та обробки результатів лабораторного дослідження планується провести оптимізацію роботи інтерфейсу користувача методами, що прискорюють роботу веб-додатку та спрощують обробку серверних запитів. Базу даних планується оптимізувати по структурі, та створити власні API для зменшення навантаження на стороні користувацького інтерфейсу.

В результаті проведеної оптимізації зроблений висновок та оцінку результатів оптимізації, їх актуальність та необхідність при сучасних методах та підходах при розробці веб-додатків для зберігання та обробки результатів лабораторного дослідження.

Наукова новизна роботи полягає у використанні блокчейн технологій у підході розробки веб-додатку для зберігання та обробки лабораторних досліджень. Вперше для розробки такого типу додатків запропоновано та реалізовано до використання принцип розробки MERN (акронім «MongoDB», «Express», «React» і «Node JS»), в результаті чого ми отримуємо стабільний та безпечний веб додаток, дані якого захищені від стороннього впливу та втрати особистих даних.

Результати дослідження були викладені в тезах та опубліковані у Всеукраїській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», яка відбулась 24 квітня 2024 року.

Також стаття «Розробка методики оптимізації веб-запитів і баз даних для підвищення продуктивності веб-застосунку для лабораторних робіт» знаходиться на етапі публікування у журналі «Зв’язок».

1 ЛІТЕРАТУРНИЙ ОГЛЯД НАУКОВО-ТЕХНІЧНОЇ ЛІТЕРАТУРИ

1.1. Основні відомості про веб-додаток

Веб-додаток складається з набору сторінок, які доступні користувачам через браузер і передаються кінцевому користувачеві через мережу. Веб-сторінка може бути статичною, де вміст є постійним для всіх користувачів, або динамічною, де вміст змінюється відповідно до введення користувача. У веб-додатках введення користувача (навігація та введення даних) напряму впливає на стан системи. Клієнтські сторінки програми зазвичай написані в HTML із вбудованим Javascript або VBscript і відображаються веб-браузером на стороні клієнта. Створені сервером сторінки зазвичай складаються зі сценаріїв CGI, сторінок активного сервера, сторінок сервера Java або сервлетів, які виконуються веб-сервером на стороні сервера та надають інформацію клієнтам. Компонент може бути шаблоном HTML, Java applet, елементом керування ActiveX, Java Bean або будь-яким програмним модулем, який взаємодіє зі сторінками клієнта, сторінками сервера чи іншими компонентами. Навіть просту веб-програму можна написати з використанням кількох мов програмування, наприклад, HTML і Javascript для інтерфейсу, Java або JSP для середнього рівня та MySQL як серверної мови [1].

Веб-програми демонструють характеристики традиційних програм, графічних інтерфейсів користувача (GUI), баз даних і розподілених програм. Графічні інтерфейси користувача та додатки баз даних подібні до веб-додатків щодо їхнього характеру відтворення подій. Введення користувача може призвести до неочікуваних змін у керуванні/потоці даних у програмі. Веб-програми подібні до розподілених програм, оскільки вони, по суті, працюють і спілкуються через мережу. Крім того, веб-додатки мають численні технології та компоненти. Їх розподілений і неоднорідний характер робить тестування веб-додатків складним завданням [2].

Веб-програми не потрібно завантажувати, оскільки доступ до них здійснюється через мережу. Користувачі можуть отримати доступ до веб-програми через веб-браузер, наприклад Google Chrome, Mozilla Firefox або Safari.

Щоб веб-програма працювала, їй потрібен веб-сервер, сервер додатків і база даних. Веб-сервери керують запитами, які надходять від клієнта, тоді як сервер додатків виконує поставлене завдання. База даних зберігає будь-яку необхідну інформацію. Такий тип програм зазвичай мають короткі цикли розробки та невеликі групи розробників [3].

Веб-програми мають багато переваг. Серед основних переваг можна виділити:

- Кілька користувачів можуть отримати доступ до однієї версії програми.
- Користувачам не потрібно встановлювати додаток.
- Користувачі можуть отримати доступ до програми через різні платформи, такі як настільний комп'ютер, ноутбук або мобільний пристрій.
- Користувачі можуть отримати доступ до програми через кілька браузерів.

У секторі мобільних версій веб-програми іноді протиставляються традиційним програмам, які розробники створюють спеціально для певної платформи чи пристрою та встановлюють на ньому. Традиційні програми зазвичай можуть використовувати спеціальне апаратне забезпечення пристрою, наприклад GPS або камеру в мобільній версії програми.

Програми, які поєднують два підходи, іноді називають гібридними програмами. Гібридні програми працюють подібно до веб-програм, але встановлюються на пристрій так само, як традиційна програма. Гібридні програми також можуть використовувати ресурси, що стосуються пристрою, за допомогою внутрішніх API. Завантажені традиційні програми іноді можуть працювати в автономному режимі; однак гібридні програми не мають цієї функції. Гібридна програма, як правило, матиме подібні елементи навігації до веб-програми, оскільки вони в основному базуються на принципах розробки веб-додатків [4].

1.2. Принципи та методи розробки інтерфейсу користувача

Для розробки інтерфейсу потрібно набагато більше, ніж написання коду. Як і інші аспекти технології, цілі розробки інтерфейсу змінилися за останнє десятиліття. У цьому розділі описуються методи розробки зовнішнього інтерфейсу та їх переваги, проблеми, з якими стикаються розробники, і нові тенденції.

Розробка інтерфейсу, також відома як розробка на стороні клієнта, — це реалізація бачення та концепції дизайну веб-сторінок або програм за допомогою коду. Цікавим аспектом розробки інтерфейсу є поява нових архітектур на початку 2020-х років:

- Web3 (блокчейн у поєднанні з веб-архітектурою);
- Metaverse (тривимірна версія Інтернету, яка ще не створена, підтримується Meta, Microsoft, Nvidia та іншими).

У розробників інтерфейсу з’являється все більше можливостей впливати як на Web3, так і на метавсесвіт [5].

Стан Web3 у 2024 році відзначений значними досягненнями та проблемами, що вказує як на зрілість, так і на постійну еволюцію простору Web3. Web3 пропонує набір можливих технологій, які можуть повністю змінити форму багатьох різних галузей, включаючи дослідження та освіту. Сьогодні ми стикаємося з проблемами, пов’язаними з регулюванням, безпекою даних і конфіденційністю, а також з управлінням і бюрократією – це лише деякі з них. Оскільки Web3 набуває все більшої популярності в усьому світі, він активує нові парадигми для залучення, управління, створення, ітерації та впровадження навколо результатів досліджень способами, які раніше були складними та трудомісткими. Децентралізована автономна організація (DAO) є однією з таких утиліт Web3, що розширює можливості, і в цій структурі дослідники та експерти мають можливість негайно впроваджувати та тестувати результати своїх досліджень у малому чи великому масштабі, залежно від потреби [6].

Децентралізований характер технології блокчейн забезпечує основу для глибоких системних змін у суспільстві. Загалом централізована платформа

покладається на контрольовану базу даних як основу для надання цінності своїм користувачам, що вимагає надійності стороннього постачальника послуг. Багато Інтернет-додатків (наприклад, електронна пошта та система доменних імен) залишаються значною мірою централізованими з точки зору управління та розробки ядра. Ця централізація несе з собою проблеми з прозорістю, цілісністю даних, конфіденційністю та безпекою даних, з чіткими кореляціями з поточною багатогранною вбудованою центральністю Інтернету від структури зв'язку клієнт-сервер до публічних хмар і систем на основі хмар [7]. Подібним чином питання довіри до хмарного сховища даних є ще одним викликом централізованого характеру Інтернету, оскільки необхідно перевірити, чи хмара не пошкоджує дані, що зберігаються клієнтами [8].

1.2.1. Використання фреймворків для розробки веб-додатків

Ландшафт веб-розробки все більше визначається потужними інтерфейсними фреймворками, які дозволяють розробникам створювати швидкі, ефективні та динамічні інтерфейси користувача. Серед найпопулярніших фреймворків сьогодні React, Angular і Vue.js. Кожен із цих фреймворків має окрему філософію проектування, архітектуру та характеристики продуктивності, що робить їх придатними для різних типів проектів [9].

React.js

React — це бібліотека JavaScript, розроблена Facebook, яка в основному використовується для створення інтерфейсів користувача. Його основна філософія обертається навколо декларативної компонентної архітектури, де кожен компонент представляє частину інтерфейсу користувача. React використовує віртуальну DOM (VDOM) для оптимізації візуалізації, що зводить до мінімуму прямі маніпуляції з реальним DOM, що призводить до покращення продуктивності динамічних інтерфейсів користувача [10].

Angular.js

Angular, розроблений компанією Google, — це комплексний фреймворк, який надає комплексне рішення для створення складних веб-додатків. Angular використовує більш впевнений підхід, ніж React, пропонуючи вбудовані рішення

для маршрутизації, перевірки форм, управління станом і HTTP-запитів. Двостороннє прив'язування даних і впровадження залежностей Angular є ключовими функціями, які дозволяють легше інтегрувати різні компоненти програми [11].

Vue.js

Vue.js — це прогресивний фреймворк JavaScript, який поєднує найкращі функції React і Angular, пропонуючи більш легкий і гнучкий підхід. Vue розроблений таким чином, щоб його можна було поступово адаптувати, тобто його можна інтегрувати в існуючі проекти без повного переписування. Vue має реактивну систему зв'язування даних, подібну до Angular, але реалізує більш спрощену архітектуру. Vue також використовує віртуальний DOM, такий як React, забезпечуючи ефективне відтворення [12].

Щоб забезпечити об'єктивний вибір,оцінюють продуктивність React, Angular і Vue.js за кількома ключовими показниками. До них належать:

1. Початковий час завантаження
2. Швидкість візуалізації
3. Використання пам'яті
4. Масштабованість

Для кожного тесту потрібно реалізувати базову програму зі списком справ із однаковою функціональністю для всіх трьох фреймворків. Це забезпечує стійку базу для порівняння. Тести проводяться в ідентичних умовах, з використанням сучасних версій браузера та контрольованого мережевого середовища для мінімізації зовнішніх факторів [13].

Початковий час завантаження

Початковий час завантаження веб-програми є критично важливим показником продуктивності, особливо для програм, орієнтованих на мобільні або повільніші мережі. Швидший час завантаження забезпечує кращий досвід роботи з користувачем, оскільки користувачі з меншою ймовірністю залишать програму, якщо вона швидко завантажується. React продемонстрував найшвидший початковий час завантаження серед трьох фреймворків. Його мінімалістична архітектура, зосереджена на шарі перегляду, дозволила йому

завантажуватися відносно швидко. Віртуальний механізм DOM у React гарантує, що відтворюються лише ті частини програми, які потребують оновлення, що призводить до меншого розміру пакета. З іншого боку, у Angular був найповільніший початковий час завантаження. Його всеохоплюючий характер, який включає такі інструменти, як ін'єкція залежностей і складні рішення маршрутизації, сприяв більшому розміру пакета та довшому часу завантаження. Велика залежність Angular від TypeScript також ускладнює збірку, що може додатково збільшити час завантаження. Vue.js працює краще, ніж Angular, але трохи гірше, ніж React, з точки зору початкового часу завантаження. Мінімалістичний підхід Vue до комплектування та його ефективна реактивна система сприяли відносно швидкому завантаженню, хоча й не такому швидкому, як React [14].

Швидкість візуалізації

Швидкість відтворення вимірює, наскільки ефективно фреймворк може оновлювати інтерфейс користувача у відповідь на зміни в стані програми. У проведених тестах React знову перевершив два інші фреймворки завдяки своєму ефективному віртуальному алгоритму DOM-маніпуляцій. React оновлює лише необхідні частини DOM, зменшуючи непотрібні повторні візуалізації. Vue.js уважно слідував за React із схожою продуктивністю з точки зору рендерингу. Його система реагування, яка базується на механізмі відстеження залежностей, забезпечує оновлення лише зачеплених компонентів у разі зміни стану. Швидкість візуалізації Vue достатньо висока для більшості програм, хоча вона відстає від React у екстремальних сценаріях, пов'язаних зі складними оновленнями інтерфейсу користувача. Швидкість візуалізації Angular була найнижчою з трьох. Його двостороннє зв'язування даних і складний цикл дайджесту часто призводили до частішого повторного рендерингу, ніж це було необхідно, особливо у великомасштабних програмах. Незважаючи на те, що механізм виявлення змін Angular оптимізований, він усе ще має проблеми зі швидкістю рендерингу порівняно з іншими фреймворками [15].

Використання пам’яті

Використання пам’яті є вирішальним фактором під час створення програм, які потребують масштабування. Надмірне споживання пам’яті може призвести до уповільнення роботи, особливо коли в програмі активні кілька компонентів. Загалом React продемонстрував найефективніше використання пам’яті завдяки своїй віртуальній DOM і підходу до вибіркового повторного рендерингу. Оскільки React оновлює лише ті частини DOM, які змінилися, це мінімізує використання пам’яті, особливо в динамічних програмах.

Vue.js продемонстрував порівнянну ефективність пам’яті з React. Його тонка система реагування гарантує, що непотрібні дані не зберігаються в пам’яті, зберігаючи легкість програми. Однак Vue не оптимізує так агресивно, як React, щодо візуалізації віртуального DOM, що може призвести до дещо більшого використання пам’яті у великих програмах. Використання пам’яті Angular було помітно вищим, ніж React і Vue. Його двостороннє прив’язування даних і широкі функції інфраструктури призводять до більшого споживання пам’яті, особливо коли складність програми зростає. Велика залежність Angular від служб і впровадження залежностей також збільшує обсяг пам’яті, роблячи його менш придатним для додатків, яким потрібно економити пам’ять [16].

Масштабованість

Масштабованість є ключовим фактором для великих програм із високим трафіком. Масштабована структура повинна ефективно обробляти великі обсяги даних, взаємодії користувачів і одночасні запити. React і Vue.js показали хороші результати з точки зору масштабованості. Віртуальний DOM і компонентна архітектура React забезпечують ефективне оновлення навіть у великих програмах з багатьма компонентами. Фреймворк також підтримує відтворення на стороні сервера (SSR), що допомагає покращити масштабованість, розвантажуючи завдання відтворення на сервер. Модульний дизайн і система реактивності Vue також сприяють його масштабованості. Здатність Vue до поступового масштабування робить його хорошим варіантом для проектів, які можуть починатися з невеликого обсягу, але потребують масштабування з часом.

Однак екосистема інструментів Vue все ще розвивається порівняно з React, що може обмежити деякі розширені функції масштабованості. Angular зіткнувся з більшими проблемами з масштабованістю, особливо у великих програмах. Незважаючи на те, що потужні інструменти Angular для впровадження залежностей і маршрутизації забезпечують міцну основу для створення складних додатків, його більший обсяг пам'яті та нижча швидкість візуалізації можуть спричинити вузькі місця продуктивності в міру зростання програми. Крім того, механізм виявлення змін Angular, хоч і ефективний, може стати вузьким місцем у великомасштабних програмах із багатьма динамічними компонентами [17].

Отже, згідно з порівняльною характеристикою React.js стала однією з найпопулярніших бібліотек JavaScript, яка привернула стільки уваги на IT-ринку. Завдяки своїм досконалим характеристикам і швидкості експерти віддали перевагу цьому варіанту, особливо для розробки веб-сайтів для лабораторних досліджень.

Ця інтерфейсна бібліотека з відкритим кодом заснована на композиції, що означає, що вона збирає повторно використовувані компоненти, щоб забезпечити бажаний інтерфейс користувача. Його створено з урахуванням архітектурної парадигми Model View Controller (MVC) і підтримується великою кількістю надійних і доступних модулів npm JavaScript. Ось чому React такий популярний. Він дозволяє виконати будь-яку роботу швидко і легко.

Крім того, оскільки це структура, орієнтована на користувальницький інтерфейс, вона дуже підходить для розробки додатків лабораторних досліджень, які прагнуть надати найкращий досвід користувача. Завдяки фреймворкам, які були побудовані навколо нього, розробка React.js стала популярною серед розробників [18].

React js має вбудовані блоки коду та компоненти дизайну, які можна використовувати для популярних програм React js. Redux і Material UI є двома добре відомими бібліотеками. Ці фрагменти коду можна використовувати на кількох веб-сайтах, заощаджуючи час на розробку та встановлення.

Основною метою React JS є створення інтерфейсів користувача (UI), які покращують продуктивність програми. Він використовує віртуальний DOM

(об'єкт JavaScript), що прискорює роботу програми. У JavaScript віртуальний DOM швидший за традиційний DOM. ReactJS можна використовувати як на стороні клієнта, так і на сервері, а також з іншими фреймворками чи бібліотеками. Він використовує шаблони компонентів і даних для покращення читабельності та допомоги в обслуговуванні великих програм [19].

React, по суті, спрощує створення інтерактивних інтерфейсів користувача. Це дозволяє створювати автономні компоненти інтерфейсу користувача, а також цілі інтерфейси користувача, що містять усі візуальні аспекти, а також логіку, яка лежить в основі та регулює ці частини.

Перш за все, React призначений для розробки потужних компонентно-орієнтованих програм. Однак, хоча React і є зовнішньою технологією, React не є самовпевненим і може використовуватися з низкою альтернативних внутрішніх технологій. У цьому сенсі Reacts функціонує як процвітаюча екосистема інструментів і технік, що пояснює, чому він має так багато різних варіантів використання – створення веб-додатків (з чого все почалося), розробка мобільних додатків, розробка додатків для настільних комп’ютерів, створення статичних сайтів і навіть Розробка додатків VR [20].

Потреба в розробці та наданні інтерактивних веб-додатків зросла швидко. Таким чином, бібліотеки інтерфейсу користувача (UI) JavaScript були представлені на зниження продуктивності та розміру комплекту. Для вирішення цієї проблеми був створений Million.js створено для створення доповнених компілятором бібліотек інтерфейсу JavaScript які є розширюваними, продуктивними та легкими. Це було досягнуто шляхом розробки обчислювально ефективного алгоритму розрізнення, який спирається на компілятор, а потім вимірювання ефективності за допомогою серії релевантних і вичерпних контрольні показники. Крім того, реалізовані вбудовані механізми, що дозволяють імперативна оптимізація, що дозволяє компілятору безпосередньо пропускати розбіжності під час виконання [21].

У порівнянні з іншими методами візуалізації віртуального DOM, ці висновки показали, що Million.js має високу продуктивність, від 133% до 300% більше операцій за секунду, ніж інші віртуальні бібліотеки DOM [22].

Million.js — це віртуальний DOM. Віртуальний DOM існує завдяки заміні властивість innerHTML для маніпулювання DOM зазвичай не рекомендується, оскільки вона використовує синтаксичний аналізатор HTML у браузері для модифікації DOM дерева. Віртуальний DOM намагається виправити це шляхом створення віртуального представлення DOM всередині об'єктів JavaScript, які є незначними з точки зору обчислень. Кожен об'єкт JavaScript або віртуальний вузол можна розділяти, щоб визначити зміни і виправляти в DOM. Таким чином, це дозволяє точно модифікувати DOM зменшуючи надмірні витрати на взаємодію з DOM.

Однак, на відміну від інших віртуальних бібліотек DOM, Million.js інтегрує способи для компілятора оптимізує гілки умов і зменшує сторонні відмінності. Це дозволяє значно покращити час обробки і навантаженість на структуру сайту. Насправді Million.js може досягти найкращої обробки по часу для зміни дерева віртуальних вузлів за умови правильної оптимізації порівняно з іншими віртуальними бібліотеками DOM [23].

Ключові вузли. Найефективніші віртуальні механізми DOM використовують механізм ключа для ідентифікації віртуальних вузлів під час розрізнення. Це дозволяє точніше маніпулювати DOM, зменшуючи витрати. Однак у більшості випадків дочірні елементи з ключем потребують суттєвих відмінностей, що призводить до надзвичайно неефективного використання часу обробки. Натомість Million.js використовує ключі, щоб визначити, чи є віртуальний вузол статичним, що зменшує час на обох сторонах.

Прапори компілятора. Оскільки механізми віртуального DOM зазвичай заохочує використання препроцесора для компіляції шаблону DSL, наприклад JSX або SFC, можна використовувати спеціальні підказки компіляції. Прапори компілятора дозволяють детерміноване виправлення поточного вузла та дочірніх вузлів, оптимізуючи гілки умов. Таким чином, позначки компілятора скорочують час, необхідний для розрізнення. Рівень сумісності JSX для Million.js виявляє дочірній вміст і застосовує такі прапори під час збірки.

Дельти. У деяких випадках, особливо з тестами, маніпуляції з DOM є простими та передбачуваними. Розбіжності для таких випадків є непотрібною

роботою, тому Million.js підтримує концепцію, яка називається «дельтами», надаючи програмний API для застосування детермінованих імперативних модифікацій до DOM.

Планування. Хоча оптимізація необробленої продуктивності може дати кращі числові порівняльні показники, ця оптимізація не забезпечує сталої роботи зі швидкістю 60 кадрів на секунду під час навігації на веб-сторінці. Завдяки можливості оновити процес виправлення за допомогою планування, виправлення в DOM можна відкласти на пізніший час, якщо основний потік заблоковано, пом'якшуючи сценарії «зависання сторінки»[24].

1.3. Принципи та методи розробки серверної частини

В умовах розробки веб додатків, що швидко розвиваються, вибір технології бази даних відіграє вирішальну роль у вирішенні проблем, пов'язаних із транзакціями великого обсягу. Технологія блокчейн пропонує децентралізований підхід до управління базами даних, підвищуючи цілісність і безпеку даних за допомогою незмінних записів і систем розподіленої книги. Його властиві характеристики роблять його особливо придатним для додатків, які вимагають прозорих історій транзакцій і стійкості до втручання.

Однак блокчейн стикається з обмеженнями масштабованості, які можуть вплинути на його продуктивність у середовищах із високим вмістом транзакцій. Конфіденційність даних є ще одним важливим моментом для баз даних лабораторних досліджень згідно міжнародних вимог. Традиційні реляційні бази даних забезпечують надійні механізми безпеки та конфіденційності даних завдяки контролю доступу, шифруванню та дотриманню таких стандартів, як GDPR. Навпаки, нові технології, такі як розподілені бази даних і методи збереження конфіденційності, такі як гомоморфне шифрування, з'являються для вирішення зростаючих проблем щодо витоку даних і несанкціонованого доступу [25].

Масштабовані рішення для зберігання є важливими для керування величезними обсягами даних, створених за результатами досліджень. Хмарні бази даних, бази даних SQL і розподілені файлові системи пропонують гнучку

масштабованість, що дозволяє організаціям ефективно справлятися з піковими навантаженнями та швидким зростанням даних. Ці рішення підтримують різні моделі даних і можуть бути оптимізовані для продуктивності та економічності, задовольняючи різноманітні потреби платформ, що оперують збереженням даних лабораторних досліджень[26].

Технологія блокчейн, спочатку розроблена для підтримки криптовалют, привернула значну увагу своїми потенційними можливостями застосування за межами цифрових валют, зокрема в розробці додатків для лабораторних досліджень.

Його основні властивості — децентралізація, незмінність і прозорість — пропонують ряд переваг, які стосуються кількох важливих аспектів керування базами даних. В основі привабливості блокчейна лежить його децентралізований характер. На відміну від традиційних баз даних, які покладаються на центральний орган, блокчейн працює в системі розподіленої книги, де дані копіюються на кількох вузлах мережі. Така децентралізація підвищує безпеку, зменшуючи ризик виникнення єдиної точки збою. Якщо один вузол скомпрометовано, цілісність даних залишається незмінною через розподілену природу книги. Ця функція особливо цінна для платформ, що зберігають результати лабораторних досліджень, які обробляють конфіденційні транзакції та особисту інформацію, оскільки вона зменшує ризик підробки даних і несанкціонованого доступу [27].

Ще однією істотною перевагою технології блокчейн є її незмінність. Після того, як дані записані в блокчейні, їх практично неможливо змінити або видалити, не змінюючи всі наступні блоки, для чого потрібен консенсус у мережі. Ця незмінність забезпечує цілісність записів транзакцій, що робить його чудовим інструментом для підтримки прозорих і надійних історій транзакцій. Для підприємств, у сфері лабораторних досліджень це означає підвищену довіру та підзвітність у транзакціях, оскільки клієнти та зацікавлені сторони можуть перевірити автентичність та історію своїх транзакцій, не побоюючись маніпулювання даними [28].

1.3.1. Проблеми конфіденційності бази даних

У сфері лабораторних досліджень захист даних користувачів має першорядне значення через збільшення частоти порушень даних і проблем конфіденційності. Оскільки такі платформи обробляють величезні обсяги особистої інформації, надійні заходи конфіденційності є важливими для захисту цих даних від несанкціонованого доступу та зловживання.

Для вирішення проблем конфіденційності з'явилися різні технології та практики, кожна з яких має свої переваги та обмеження. Традиційні реляційні бази даних вже давно є основою систем лабораторних досліджень, пропонуючи надійні механізми конфіденційності даних через встановлені протоколи безпеки. Ці бази даних використовують засоби контролю доступу, щоб гарантувати, що лише авторизовані користувачі можуть отримати доступ до конфіденційних даних [29].

Крім того, шифрування відіграє вирішальну роль у захисті даних як у стані спокою, так і під час передачі. Завдяки кодуванню даних у формат, який неможливо прочитати, шифрування гарантує, що навіть у разі перехоплення даних або доступу неавторизованих осіб вони залишаються захищеними від зловживання. Дотримання нормативних актів щодо захисту даних, таких як Загальний регламент захисту даних (GDPR) і Каліфорнійський закон про конфіденційність споживачів (CCPA), є ще одним важливим аспектом заходів із забезпечення конфіденційності . Реляційні бази даних підтримують відповідність нормативним вимогам, дозволяючи компаніям впроваджувати такі функції, як анонімізація та псевдонімізація даних. Ці методи допомагають захистити особистість користувачів, одночасно дозволяючи аналізувати дані без розкриття особистої інформації [30].

Оскільки занепокоєння конфіденційністю продовжує зростати, з'явилися нові технології та методи для подальшого покращення захисту даних. Методи збереження конфіденційності, такі як гомоморфне шифрування, пропонують передові рішення для захисту конфіденційних даних. Гомоморфне шифрування дозволяє виконувати обчислення із зашифрованими даними без необхідності їх попереднього розшифровування. Цей підхід дозволяє платформам аналізувати

та обробляти дані, зберігаючи їхню конфіденційність, усуваючи занепокоєння щодо витоку даних під час обробки [31].

Ще одна нова технологія — це розподілення бази даних, яке пропонує переваги конфіденційності завдяки розділенню даних і реплікації на кількох вузлах. Цей розподіл зменшує ризик витоку даних завдяки децентралізації зберігання конфіденційної інформації. Крім того, розподілені бази даних часто включають розширені функції безпеки, такі як точне керування доступом і моніторинг у реальному часі, для подальшого покращення захисту даних. Незважаючи на ці досягнення, заходи щодо конфіденційності повинні постійно розвиватися, щоб усунути нові загрози та вразливості. Платформи, що зберігають результати лабораторних досліджень повинні прийняти багаторівневий підхід до конфіденційності даних, поєднуючи традиційне шифрування та контроль доступу з передовими технологіями збереження конфіденційності. Регулярні перевірки безпеки, тестування на проникнення та дотримання нормативних актів, що розвиваються, є важливими для того, щоб заходи конфіденційності залишалися ефективними для захисту даних користувачів [32].

1.3.2 Основні положення масштабованості бази даних

Масштабоване сховище є критично важливим компонентом для платформ, що зберігають результати лабораторних досліджеь, яке повинно обробляти значні та часто непередбачувані обсяги даних. Оскільки онлайн-транзакції, взаємодія користувачів і генерація даних продовжують зростати, можливість ефективного масштабування інфраструктури зберігання стає важливою для підтримки продуктивності та забезпечення бездоганної взаємодії з користувачем.

Хмарні бази даних. Стали популярним рішенням для масштабованого зберігання. Ці бази даних використовують хмарну інфраструктуру для забезпечення розподілу ресурсів за запитом, дозволяючи компаніям збільшувати чи зменшувати обсяги сховища залежно від поточних потреб. Ця еластичність має вирішальне значення для платформ медичних структур, які відчувають

коливання моделей трафіку, наприклад під час сезону епідемій або спалаху хвороби. Хмарні бази даних також пропонують високу доступність і надійність, оскільки дані копіюються на кількох серверах і географічних місцях, що знижує ризик втрати даних і простоїв [33].

Бази даних SQL. Інший варіант зберігання, призначений для обробки великих обсягів неструктурованих або напівструктурованих даних. На відміну від традиційних реляційних баз даних, які покладаються на фіксовані схеми та складні запити, бази даних SQL забезпечують гнучкі моделі даних, які можна легко налаштувати для розміщення різноманітних типів даних і структур. Ця гнучкість особливо корисна для медичних структур, які керують різними даними, зокрема каталогами продуктів, відгуками клієнтів та історією транзакцій. Бази даних SQL, такі як сховища ключ-значення, сховища документів і сховища сімейства стовпців, пропонують високу продуктивність і масштабованість, що робить їх добре підходящими для програм, які вимагають швидкого доступу до даних і обробки [34].

Розподілені файлові системи. Також відіграють важливу роль у масштабованих рішеннях для зберігання. Ці системи розподіляють дані між кількома вузлами, забезпечуючи паралельну обробку та розширення сховища. Розподілені файлові системи, такі як Hadoop Distributed File System (HDFS) і Apache Cassandra, забезпечують відмовостійкість і високу пропускну здатність, що робить їх ідеальними для виконання великомасштабної обробки даних і вимог до зберігання [35].

Розповсюджуючи дані між кількома серверами, розподілені файлові системи можуть ефективно керувати великими обсягами даних, зберігаючи продуктивність і надійність.

Незважаючи на їхні переваги, для зберігання потрібен ретельний контроль, щоб уникнути потенційних проблем. Наприклад, хмарні бази даних можуть нести змінні витрати залежно від використання, що вимагає ретельного моніторингу та бюджетування. Бази даних SQL, хоч і гнучкі, можуть вимагати додаткових зусиль для реалізації узгодженості та керування транзакціями. Розподілені файлові системи можуть бути складними в налаштуванні та

обслуговуванні, вимагаючи спеціальних знань і ресурсів. Масштабовані рішення для зберігання даних є важливими для задоволення зростаючих вимог до обсягу даних і продуктивності. Хмарні бази даних пропонують гнучкі та еластичні можливості зберігання, бази даних SQL забезпечують високу продуктивність для різних типів даних, а розподілені файлові системи забезпечують ефективну обробку великих даних. Вибравши та впровадивши відповідне масштабоване рішення для зберігання, компанії можуть покращити свою здатність обробляти великі обсяги транзакцій, оптимізувати продуктивність і забезпечити надійну та надійну роботу користувача [36].

1.3.3. Цілісність бази даних

Цілісність даних є фундаментальним аспектом керування базами даних, особливо в медичної структури, де точність і надійність є найважливішими. Забезпечення цілісності даних передбачає підтримку послідовності, точності та достовірності даних протягом усього життєвого циклу. В медичній установі, де щодня обробляються величезні обсяги досліджень і даних користувачів, підтримка цілісності даних має вирішальне значення для ефективності роботи, довіри клієнтів і дотримання нормативних стандартів.

Одним із основних механізмів забезпечення цілісності даних є обмеження бази даних і правила перевірки. Наприклад, реляційні бази даних використовують такі обмеження, як первинні ключі, зовнішні ключі та унікальні обмеження для забезпечення точності та узгодженості даних. Первинні ключі гарантують, що кожен запис у таблиці є унікальним і ідентифікованим, а зовнішні ключі підтримують посилальну цілісність між пов’язаними таблицями. Ці обмеження запобігають вставці недійсних або повторюваних даних і забезпечують коректну підтримку зв’язків між елементами даних.

Перевірка даних є ще одним важливим компонентом цілісності даних. Правила перевірки застосовуються до вхідних даних, щоб переконатися, що вони відповідають заздалегідь визначеним критеріям перед введенням у базу даних. Цей процес допомагає запобігти помилкам і невідповідностям, відфільтровуючи недійсні або неповні дані. Наприклад, платформа може використовувати правила

перевірки, щоб переконатися, що адреси користувачів відповідають певному формату або що результати правильно відформатовані та дійсні [36].

На додаток до цих традиційних методів, сучасні технології баз даних використовують передові методи для підвищення цілісності даних. Наприклад, розподілені бази даних використовують алгоритми консенсусу та реплікацію, щоб забезпечити узгодженість даних на кількох вузлах. Алгоритми консенсусу, такі як Paxos або Raft, допомагають підтримувати узгодженість, вимагаючи згоди більшості вузлів перед прийняттям будь-яких змін даних. Реплікація, з іншого боку, передбачає копіювання даних між кількома вузлами, щоб забезпечити надлишковість і гарантувати, що дані залишаються доступними, навіть якщо деякі вузли виходять з ладу.

Цілісність даних також тісно пов’язана з безпекою даних. Шифрування відіграє вирішальну роль у захисті даних від несанкціонованого доступу та підробки. Шифруючи дані як у стані спокою, так і під час передачі, організації можуть захистити конфіденційну інформацію від зміни або доступу неавторизованих сторін. Крім того, засоби контролю доступу та механізми автентифікації гарантують, що лише авторизовані користувачі можуть змінювати або переглядати дані, що ще більше сприяє цілісності даних. Аудит і моніторинг є життєво важливими методами підтримки цілісності даних. Регулярні перевірки допомагають виявляти й усувати потенційні проблеми чи аномалії, а постійний моніторинг дозволяє виявляти проблеми з цілісністю даних у режимі реального часу. Впроваджуючи системи реєстрації та моніторингу, платформи, що оброблюють результати лабораторних досліджень можуть відстежувати зміни в даних, виявляти неавторизований доступ і швидко реагувати на потенційні порушення цілісності [37].

1.3.1. Обробка великих обсягів даних

Обробка великого обсягу транзакцій є критичним викликом для платформ лабораторних досліджень, які часто відчувають значні та коливальні транзакції. Ефективне керування цими транзакціями має важливе значення для

забезпечення безперебійної роботи, підтримки задоволеності користувачів і запобігання збою системи або погіршенню продуктивності.

Кілька стратегій і технологій використовуються для вирішення проблем, пов’язаних із обробкою великих обсягів у середовищах постійного потоку пацієнтів. Масштабованість є основним критерієм для керування великими обсягами медичних заключень. Балансування навантаження є однією з важливих стратегій для обробки великих обсягів транзакцій. Балансувальники навантаження розподіляють вхідний трафік між кількома серверами або екземплярами, гарантуючи, що жоден сервер не буде перевантажений. Цей розподіл допомагає підтримувати продуктивність і надійність, запобігаючи вузьким місцям і знижуючи ризик збоїв сервера. Балансування навантаження може бути реалізоване на різних рівнях, включаючи рівні додатків, мережі та бази даних, щоб оптимізувати використання ресурсів і забезпечити постійний час відповіді [38].

Кешування є ефективним методом для підвищення продуктивності та керування великими обсягами транзакцій. Кешування передбачає зберігання даних, до яких часто звертаються, у місці тимчасового зберігання, наприклад у кеш-пам’яті, щоб зменшити потребу в повторюваних запитах до бази даних. Швидко обслуговуючи кешовані дані, платформи можуть значно зменшити затримку та покращити взаємодію з користувачем.

Популярні рішення для кешування включають мережі доставки контенту (CDN) і системи кешування пам’яті, такі як Redis і Memcached. Оптимізація бази даних також відіграє важливу роль в управлінні великими обсягами медичних заключень. Такі методи, як індексування, розділення та сегментування, допомагають підвищити продуктивність і масштабованість бази даних. Індексування покращує швидкість запитів, створюючи структури даних, які сприяють швидшому пошуку, тоді як розділення та шардинг розподіляють дані між кількома базами даних або серверами, щоб збалансувати навантаження та скоротити час доступу. Правильний дизайн і оптимізація бази даних забезпечують ефективну обробку медичних заключень навіть за умов високого навантаження.

Стратегії реплікації та резервного копіювання даних є важливими для забезпечення доступності та надійності даних під час великих обсягів транзакцій. Реплікація передбачає створення копій бази даних на кількох серверах або місцях, забезпечуючи надлишковість і відмово-стійкість. Цей підхід гарантує, що дані залишаються доступними навіть у разі збою сервера. Регулярне резервне копіювання додатково захищає від втрати даних, уможливлюючи відновлення в разі системних збоїв або інших проблем. Впроваджуючи ці стратегії та технології, платформи медичних структур можуть ефективно обробляти великі обсяги транзакцій, підтримувати продуктивність і надійність, а також забезпечувати безперебійну роботу користувачів. Вирішення проблем, пов'язаних із обробкою великих обсягів, має вирішальне значення для успіху та стійкості операцій з обробки та зберігання в динамічному та конкурентному середовищі [39].

1.3.1. Мова для програмування серверної частини веб-додатків

Node.js — це швидкий і масштабований інструмент веб-розробки, який спрощує розробку, дозволяючи розробникам використовувати ту саму мову та бібліотеки в усьому стеку веб-утиліт, його неблокуюча керована подіями архітектура дозволяє легко створювати веб-пакети, які можуть обробляти широкий спектр одночасних користувачів або з'єднань node js заохочує модульну розробку покращує підтримку та швидкість розробки ідеально підходить для архітектур мікросервісів розкладає складні програми на менші можливі пропозиції, які можна розробляти незалежно розгортає та масштабує екосистему node js включаючи бібліотеки та фреймворки прискорює розробку і забезпечує захист безперервна оптимізація інструментів і стратегій node js оптимізує код і продуктивність додатків, дозволяючи розробникам проектувати, створювати та підтримувати веб-додатки, ефективно забезпечуючи їх актуальність і довгострокову функціональність [40].

Сьогодні в умовах стрімкого розвитку Інтернету багато сайтів стикаються з новими проблемами, такими як проблема багато користувальницьких запитів і високого паралелізму. Мова динамічних сценаріїв JavaScript стала надзвичайно

популярною для клієнтів і широко використовується у веб-розробці. Node.js — це платформа, побудована на середовищі виконання JavaScript Chrome для легкого створення швидких, масштабованих мережевих програм [41]. Node.js використовує керовану подіями неблокуючу модель введення-виведення, що робить його легким і ефективним, ідеальним для додатків реального часу з інтенсивним об'ємом даних, які працюють на розподілених пристроях [41]. Опитування популярності Node.js, проведені офіційним веб-сайтом, показують, що середня кількість завантажень становить понад 35 000 з моменту випуску версії 0.10 у березні 2013 року. Корпорації швидко усвідомлюють важливість Node.js, і п'ять основних постачальників PAAS підтримують Node.js [42]. На сьогоднішній день JavaScript є першою популярною мовою GitHub із 133 137 репозиторіями [43]. Якщо говорити про оцінку продуктивності веб-технологій, багато дослідників виконували відповідну роботу. Але наша робота відрізняється від інших цими двома аспектами. По-перше, ми розглядаємо як об'єктивні систематичні тести (бенчмарк), так і реалістичні тести поведінки користувача (сценарій) з двох сторін і використовуємо найновіший комерційний інструмент тестування LoadRunner. По-друге, ми головним чином дбаємо про ефективність підтримки одночасних користувачів, щоб задовольнити попит на веб-сайти реального часу з інтенсивним введенням-виведенням. Ця стаття присвячена впливу трьох різних веб-технологій: Node.js, PHP і Python-Web на продуктивність Інтернету. Питання безпеки та масштабованості виходять за рамки статті. В основному ми використовуємо бенчмарк тести та сценарні тести. Крім того, в статті запропоновано один універсальний метод оцінки техніки веб-розробки на основі порівняння продуктивності, який можна використовувати для оцінки будь-якої нової веб-технології [43].

У дослідженнях в основному порівняння технологій веб-розробки PHP, Python-Web і Node.js. Тому можна обрати кілька різних модулів для створення трьох середовищ. База даних, що використовувалась Mysql5.5 [44, 45].

1.3.1.2 Тест «Hello World» запитів

Із зростанням кількості користувачів продуктивність трьох технологій демонструє тенденцію до зростання перед тим, як зменшитися, якщо кількість запитів не перевищує 10000. Як показано на Рис. 1.1 та Рис. 1.2, «середня кількість запитів за секунду» найбільша і складає 3703,5, коли кількість користувачів Node.js становить 100. Тим часом «середній час на запит» є найкоротшим і становить 0,27 мс. Потім «середня кількість запитів за секунду» сповільнюється та підтримує стабільний стан близько 2700. Що стосується Python-Web, «середня кількість запитів за секунду» залишається стабільною на рівні близько 500, а найвища — 559,42. На даний момент «середній час на запит» є найкоротшим і становить 1,788 мс. Для PHP він також досягає піку, коли кількість користувачів становить 100. «Середнє число запитів за секунду» на той час становить 2977,54. «Середній час на запит» становить найкоротше 0,336 мс. Зі збільшенням кількості користувачів «середня кількість запитів за секунду» зменшується до 200 і залишається стабільною [44].

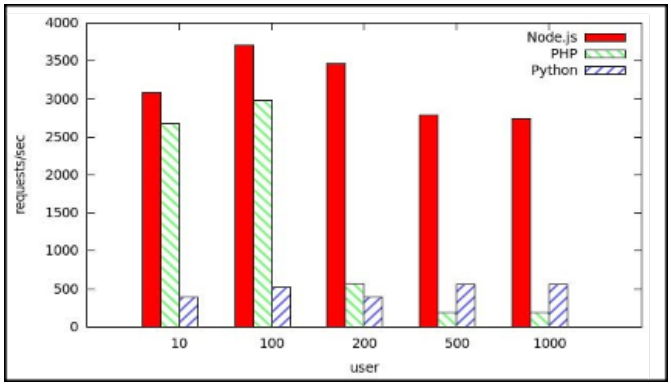


Рис. 1.1 Результати «Hello World» кількість запитів на секунду

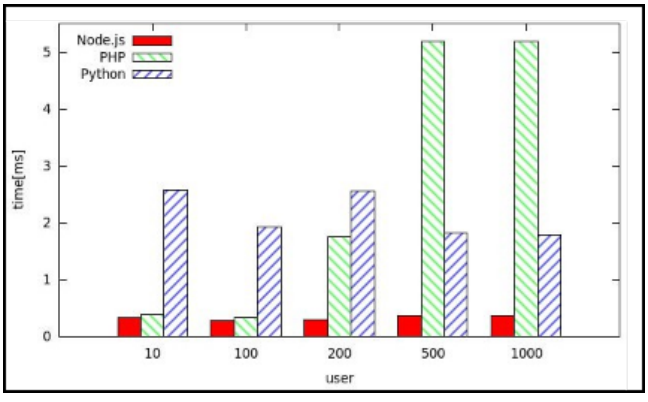


Рис. 1.2 Результати «Hello World» кількість часу на запит

Отже, згідно досліджень, продуктивність Node.js краща, ніж у двох інших при однаковій кількості користувачів. Продуктивність Node.js у два рази вища, ніж у PHP при базовому порівнянні продуктивності, і в шість-сім разів вища, ніж у Python-Web. Крім того, поточні користувачі, яких може обслуговувати Node.js, набагато більші, ніж PHP, не кажучи вже про Python-Web. Тому його продуктивність набагато краща, ніж PHP і Python-Web, коли присутня велика навантаженість на додаток [44].

1.3.1.1. Тестування обчислення послідовності Фібоначчі

Рис 1.3 та Рис 1.4. показують продуктивність обчислення десятого значення послідовності Фібоначчі, коли кількість запитів становить 10000, а кількість користувачів постійно зростає. З точки зору Node.js, «середня кількість запитів на секунду» — це найвище значення, яке досягає 2777,72 разів на секунду, а «середній час на запит» становить 0,36 мс, коли користувачів 100. Крім того, «середня кількість запитів на секунду» ” Node.js зберігається від 2000 до 2800. Пікові запити в секунду зменшуються в 1,5 рази в порівнянні з тим же станом порівнюючи тести «Hello World». Але для Python-Web цей тест має схожі результати з попереднім тестом, навіть кращі результати за тих же умов. PHP також збільшується до максимального значення, коли кількість користувачів досягає 100, «середнє число запитів за секунду» становить 3127,98. Між ним і модулем «Hello World» немає великої різниці [44].

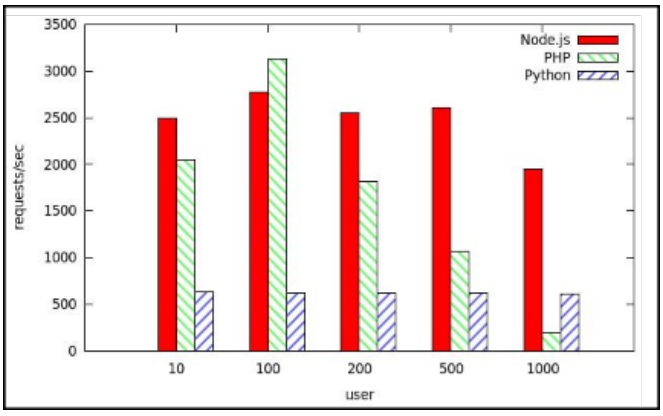


Рис. 1.3 Результати обчислень послідовності Фібоначчі запитів на секунду

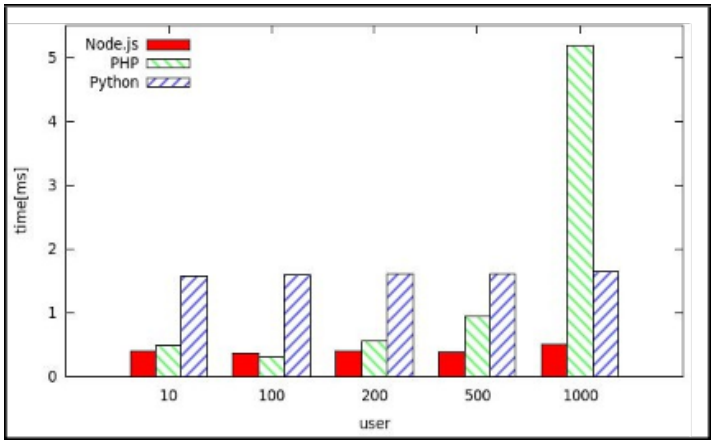


Рис. 1.4 Результати обчислень послідовності Фібоначчі кількість часу на запит

Як ми бачимо, продуктивність Node.js краща, ніж у двох інших, у даному тесті. Але результати, дуже подібні до тесту «Hello World» в однакових умовах, тому для подальшої перевірки обрано інші числа послідовності. Обчислюючи двадцяте та тридцяте значення Фібоначчі, коли користувачів 10 Табл.1.1 Продуктивність трьох веб-технологій падає різною мірою зі збільшенням числа Фібоначчі. Усі три мови значно знижують продуктивність, особливо PHP, «середнє число запитів за секунду» падає з 2000 до 2. Крім PHP, Python-Web знижує з 600 до 3. Node.js також значно зменшується з 2500 до 60. Це явище означає, що всі три веб-технології не адаптуються до програм, що потребують обчислень. Однак Node.js працює краще серед інших у цьому тесті [44].

Таблиця 1.1.

Результати обчислення послідовності Фібоначчі (10/20/30)

Мова програмування	Обчислення послідовності	Середнє число запитів за секунду	Середній час запиту
Node.js	Фібоначчі(10/20/30)	2491.77/1529.4/58.85	0.401/0.654/16.993
Python Web	Фібоначчі(10/20/30)	633.68/209.89/2.9	1.578/4.764/345.307
PHP	Фібоначчі(10/20/30)	2051.22/168.8/1.78	0.488/5.942/560.533

Відповідно до результатів, наведених вище, проведено кілька тестів з Node.js, щоб знайти різницю в продуктивності для різних одночасних користувачів у міру збільшення обчислення.

Рис.1.5 та Рис. 1.6 показують результати обчислення послідовності Фібоначчі (10/20/30), кількість користувачів зростає з 10 до 1000.

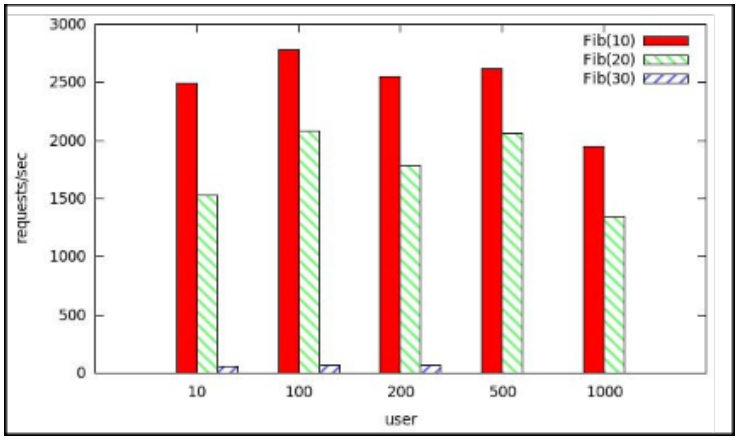


Рис. 1.5 Результати обчислення послідовності Фібоначчі (10/20/30) кількість запитів за секунду для Node.js

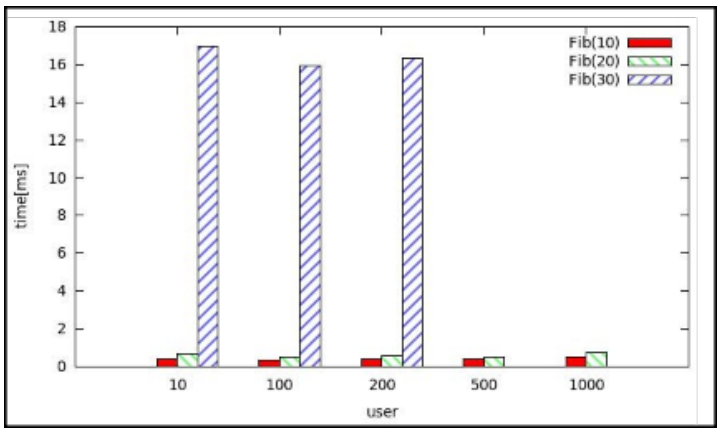


Рис. 1.6 Результати обчислення послідовності Фібоначчі (10/20/30) середній час запиту для Node.js

Хоча є невелика різниця між результатами при одному обчисленні, загальна тенденція залишається стабільною. «Середнє значення запитів за секунду» Фібоначчі (10) становить від 2000 до 2800. Фібоначчі (20) становить від 1300 до 2000. Фібоначчі (30) становить близько 60. Тим часом перевірка Фібоначчі (30) переривається, коли користувачі до 500. Отже, приріст

розрахунку приносить набагато більший ефект, ніж більші користувачі. Ми робимо простий висновок, що Node.js більше адаптований до додатків із інтенсивним введенням-виведенням, а не до додатків із інтенсивним обчисленням, оскільки додатки з інтенсивним обчисленням не використовують переваги Node.js [45].

1.3.1.2. Робота з базою даних

Щоб підтвердити висновок та довести, що Node.js адаптований до програми, яка оброблює та зберігає лабораторні дослідження, проведено тест роботи з базою даних Рис. 1.7. та Рис. 1.8. показують результати цього тесту.

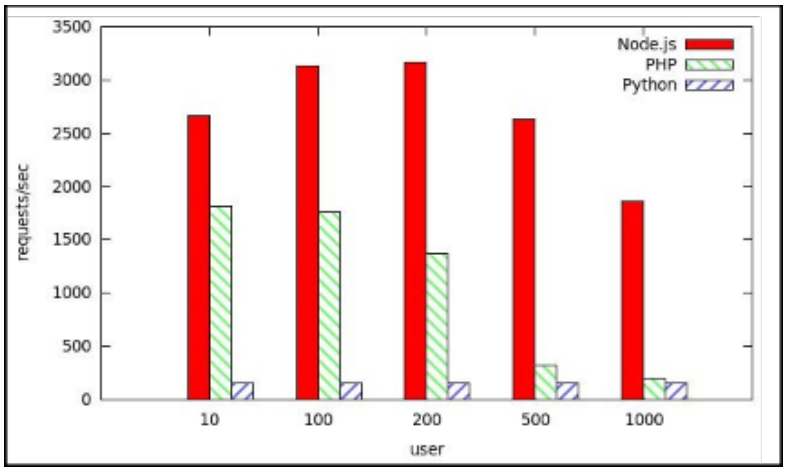


Рис. 1.7 Результати тесту роботи з базою даних кількість запитів за секунду

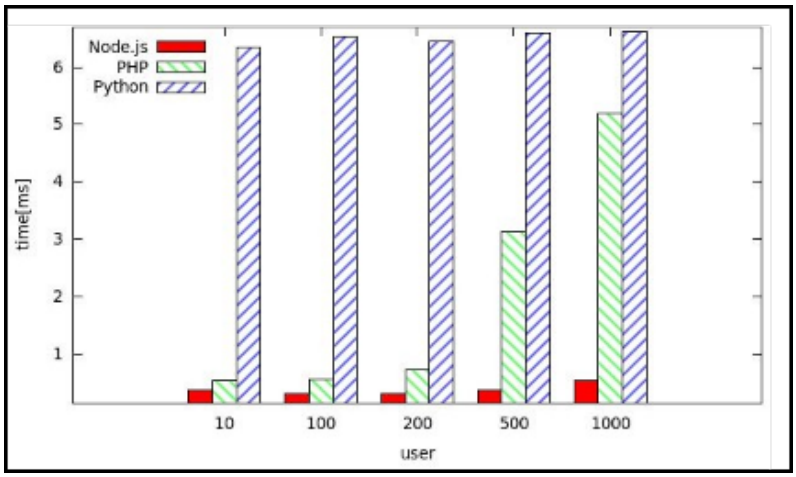


Рис 1.8 Результати тесту роботи з базою даних середній час запиту

Максимальне «середнє число запитів за секунду» Node.js становить 3164,46, що в 20 разів більше, ніж у Python-Web, і в 2 рази більше, ніж у PHP. Крім того, пікове значення тесту з Node.js не має великої різниці з тестом «Hello World». «Середнє число запитів за секунду» PythonWeb становить близько 150 і залишається стабільним із збільшенням кількості користувачів. Але для PHP «середня кількість запитів за секунду» сповільнюється, а «середній час на запит» довший. Це доводить, що Node.js більше підходить для додатків із інтенсивним введенням-виведенням серед трьох, тоді як PHP застосовний до невеликих веб-додатків. Node.js працює набагато краще, ніж PHP, у ситуації високого паралелізму, незалежно від порівняльних тестів. PHP добре обробляє невеликі запити, але має проблеми з великими запитами. Крім того, Node.js надає перевагу використанню в ситуаціях із інтенсивним введенням-виведенням, а не на сайтах із інтенсивним обчисленням [46].

Загалом Python-Web має багато зрілих фреймів для розробки великомасштабних веб-сайтів, таких як YouTube і Source Forge. Node.js — це нова технологія, яка має багато переваг у ситуації інтенсивної обробки даних, але це трохи важко для розробників, які не знайомі з асинхронним програмуванням. Що стосується PHP, то це стара техніка, популярна для використання на сайтах малого та середнього масштабу [47].

1.4. Взаємодія інтерфейсу користувача та серверної частини веб-додатку

У міру розвитку галузі веб-розробки MERN — акронім «MongoDB», «Express», «React» і «Node JS» — став домінуючим стеком у 2023 році. Завдяки відносній простоті технологій, що входять до складу цього стеку , один розробник може ефективно керувати як зовнішнім, так і внутрішнім кінцем програми. MongoDB, яка є базою даних без SQL; Express, який є фреймворком Node JS, що використовується у внутрішній розробці; React, яка є бібліотекою JavaScript, яка використовується у розробці інтерфейсу; і NodeJS, яке є середовищем для JavaScript; це компоненти, які складають стек MERN [48].

Стек MERN — це набір міцних і надійних технологій, які можна використовувати для створення масштабованих головних веб-додатків разом із компонентами сервера, інтерфейсу та бази даних. За допомогою JavaScript можна створювати повноцінні комп'ютерні програми набагато швидше та легше. MERN Stack — це технологія з відкритим вихідним кодом, яка забезпечує зручну, комплексну структуру JavaScript для створення динамічних, динамічно реагуючих програм і веб-сторінок.

Трьох-рівнева архітектурна система в MERN здебільшого складається з таких рівнів:

- інтерфейсний рівень, веб;
- сервер знаходиться посередині;
- база даних є серверною частиною.

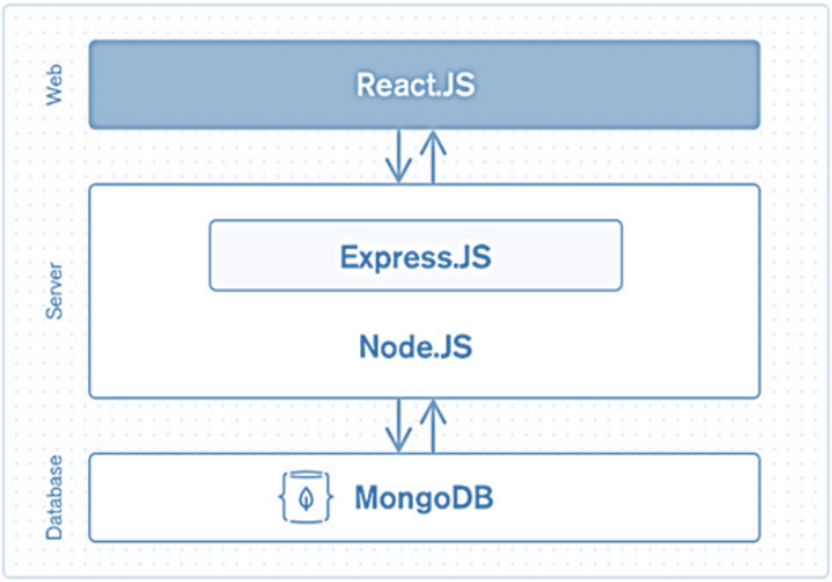


Рис. 1.9 MERN архітектура

Оскільки це код з відкритим вихідним кодом, на основі якого працюють спеціалісти з технологій у всьому світі, MERN віддають перевагу стартапам. Стек MERN — це структура з відкритим кодом для розробки високопродуктивних веб-сайтів і онлайн-додатків. Оскільки програмне забезпечення з відкритим вихідним кодом не залежить від постачальника, не виникне додаткових перешкод, через які ви коли-небудь вирішите змінити або оновити програмне забезпечення [49].

Замість того, щоб чекати завершення однієї операції перед початком іншої, Node.js дозволяє програмам продовжувати працювати, незважаючи на затримки введення/виведення. Техніка відома як асинхронний ввід-вивід.

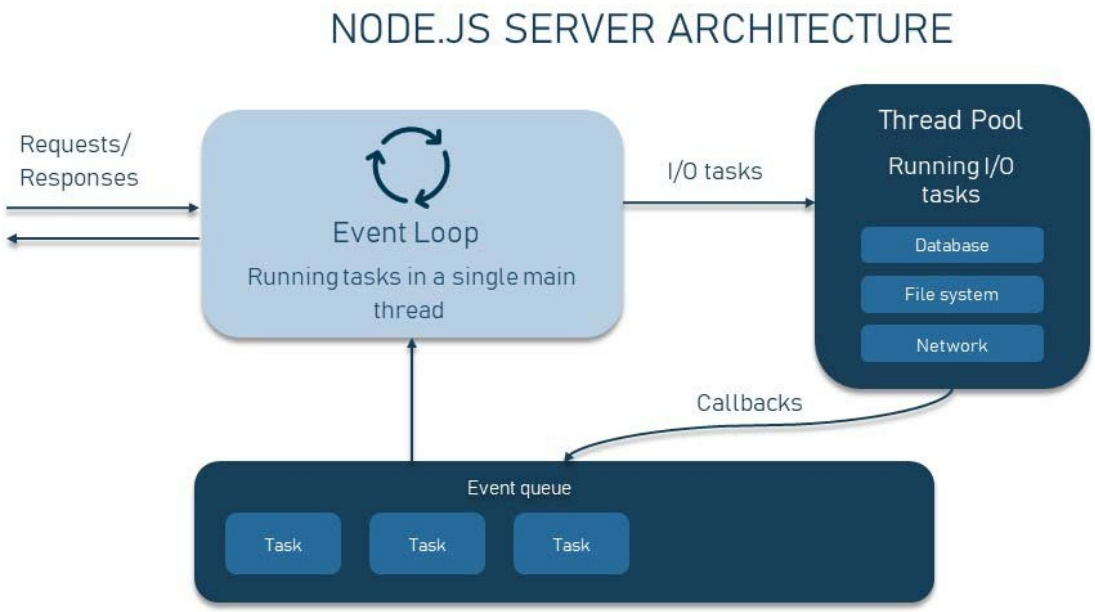


Рис. 1.9 Внутрішня робота Node.js

Функції потрібно отримати дані з мережі, обробити їх, а потім повернути результат. У світі синхронних операцій це означає, що програмі доведеться чекати, поки функція отримає дані та виконає свою роботу, що блокуватиме інші дії. «Callbacks» та «Promises». «Callbacks» — це функції, які викликаються після завершення операцій введення/виведення. Вони головна нитка, як тільки це ясно. «Callbacks» можуть бути вкладені в інші, що ускладнює код і може призвести до «пекла зворотних викликів». Цикл подій – отримує нові «Callbacks» та перевіряє чергу подій на наявність запитів, що очікують на розгляд, після завершення попередніх завдань. Після цього цикл починається знову Веб-програми традиційно покладаються на веб-сервер, щоб пасивно очікувати HTTP-запитів від клієнтів. У відповідь на HTTP-запит сервер передасть керування обробнику маршруту, який він визначить найбільш підходящим. [50].

Орієнтований на документ дизайн NoSQL MongoDB робить його добре придатним для зберігання масивних наборів даних. MongoDB зберігає інформацію не в таблицях, а в колекціях і документах. Документи, які

складаються з пар ключ-значення, є основною одиницею зберігання даних, яку можна використовувати під час роботи з MongoDB. Колекції схожі на таблиці в реляційній базі даних, оскільки містять колекції документів і дій. База даних під назвою MongoDB вперше з'явилася десь у середині 2000-х років [51].

Особливості MongoDB:

- Кожна база даних має власні колекції, а ці колекції мають власні документи. Кількість полів у кожному документі може відрізнятися. Кожен файл може мати унікальний розмір і містити різну кількість інформації;
- Розробники знайдуть структуру документа більш звичною, оскільки вона відображає те, як вони створюють класи та об'єкти на своїх улюблених мовах програмування. На відміну від таблиць, розробники зазвичай стверджують, що їхні класи мають ієрархічну структуру на основі пар ключ-значення;
- У MongoDB попередньо визначена схема не потрібна для рядків (або документів). Натомість поля можна створювати динамічно.
- Формат даних MongoDB дозволяє легко зберігати масиви та представляти складні структури, такі як ієрархії.
- Середовища, створені на основі масштабу MongoDB, досить добре. Кластери були визначені підприємствами по всьому світу; деякі з них використовують 100 або більше вузлів і зберігають мільйони записів у своїх базах даних [52].

2 ПРИНЦИПИ І МЕТОДИ КОМПЛЕКСНОЇ ОПТИМІЗАЦІЇ ВЕБ-ЗАСТОСУНКУ

2.1. Оптимізація маніпуляції DOM-елементів за допомогою Million.js

Бібліотека Million.js оптимізує роботу веб-додатку на рівні інтерфейсу користувача, прискорюючи маніпуляцію DOM-елементів при цьому не змінюючи структуру коду.

Для встановлення та налаштування бібліотеки вся необхідна інформація знаходиться на офіційному сайті «https://million.dev/docs». Так як для розробки ми використовуємо Node.js інсталяція пакету відбувається наступним шляхом:

1. Відкриваємо термінал в програмі Virtual Studio Code (Рис. 2.1.);

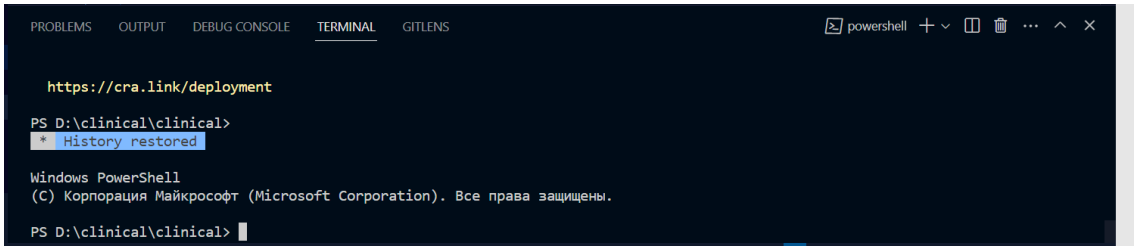


Рис. 2.1 Термінал в програмі Virtual Studio Code

2. Вписуємо в команду строку [npx million@latest] для запуску процесу інсталяції та підтверджуємо початок процесу(Рис. 2.2.)

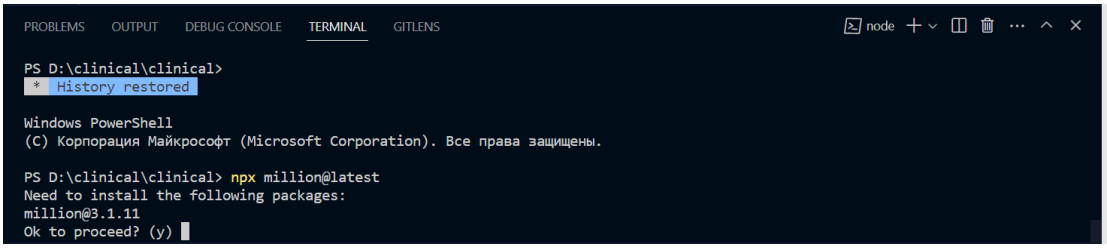


Рис. 2.2 Початок інсталяції бібліотеки Million.js

3. Підтверджуємо версію пакету, яку плануємо інсталювати (Рис. 2.3.)

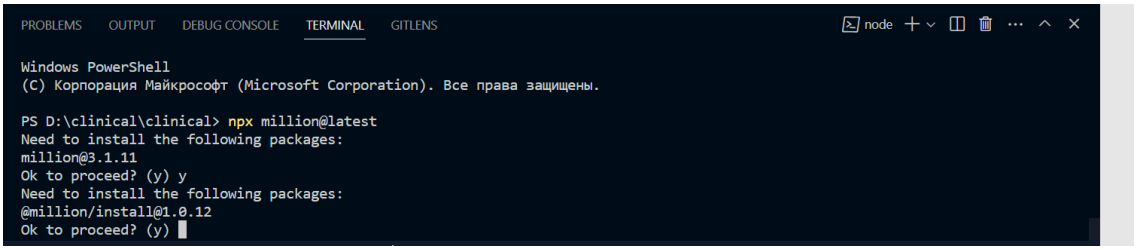


Рис. 2.3 Підтвердження версії пакету бібліотеки Million.js.

4. Після появи повідомлення про успішне встановлення пакету (Рис. 2.4.) бібліотека готова до використання

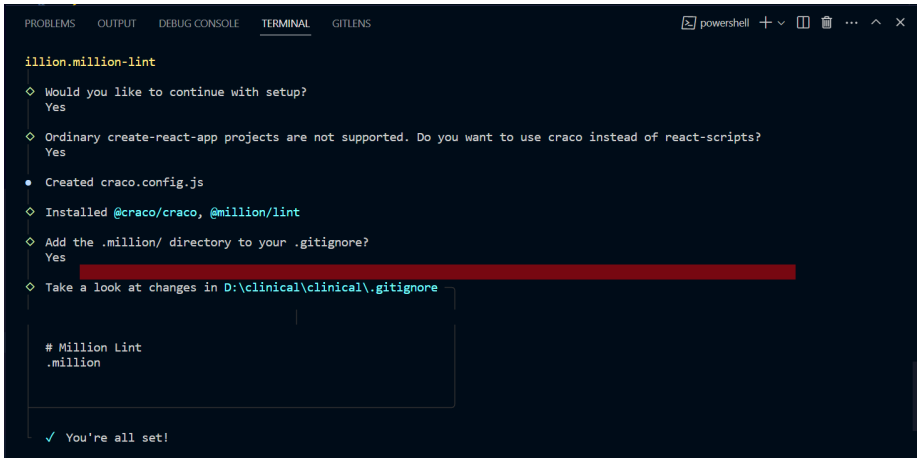


Рис. 2.4 Повідомлення про успішне встановлення бібліотеки Million.js

2.2 Оптимізація взаємодії компонентів React.js за допомогою бібліотеки Redux Toolkit

Бібліотек Redux допомагає розробнику передавати значення між компонентами без зв'язування між дочірніми та батьківськими елементами. Також за допомогою цієї бібліотеки зручно зберігати дані для створення форм та їх обробки.

Для інсталяції бібліотеки ми звертаємось до документації на офіційному сайті «https://react-redux.js.org/». Згідно документації необхідно мати версію React 16.8.3 або вище. Так як для розробки ми використовуємо Node.js інсталяція пакету відбувається наступним шляхом:

1. Відкриваємо термінал в програмі Virtual Studio Code (Рис. 2.1.);

2. Вписуємо в команду строку [npm install react-redux] для запуску процесу інсталяції (Рис. 2.5.)

Рис. 2.5 Початок інсталяції бібліотеки React Redux Toolkit

3. Після появи повідомлення про успішне встановлення пакету (Рис. 2.6.) бібліотека готова до використання

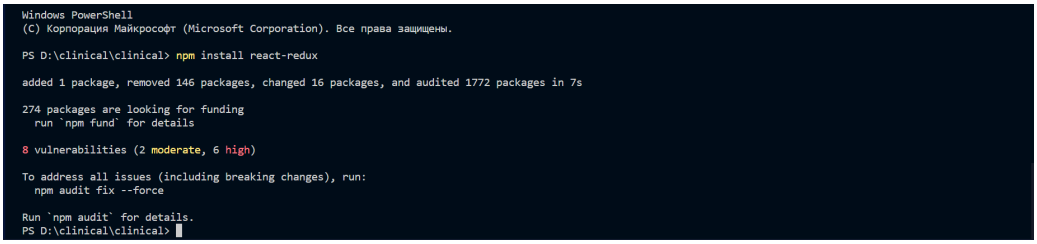


Рис. 2.6 Повідомлення про успішне встановлення бібліотеки React Redux Toolkit

2.3. Оптимізація HTTP-запитів за допомогою бібліотеки Axios

Axios – це HTTP-клієнт, заснований на Promise для node.js та браузера. Він ізоморфний може працювати в браузері і node.js з тією ж базою кодів. На стороні сервера він використовує нативний node.js http-модуль, тоді як стороні клієнта (браузер) він використовує XMLHttpRequests.

Для підключення Axios до проекту потрібно виконати дії прописані на офіційному сайті бібліотеки «https://axios-http.com/». Так як для розробки ми використовуємо Node.js інсталяція пакету відбувається наступним шляхом:

1. Відкриваємо термінал в програмі Virtual Studio Code (Рис. 2.1.);
2. Вписуємо в команду строку [npm install axios] для запуску процесу інсталяції (Рис. 2.7.)

Рис. 2.7 Початок інсталяції бібліотеки Axios.

3.Після появи повідомлення про успішне встановлення пакету (Рис. 2.8.) бібліотека готова до використання

Рис. 2.8 Повідомлення про успішне встановлення бібліотеки Axios

2.4. Оптимізація навігації веб-додатку за допомогою бібліотеки React Router

React Router полегшує «маршрутизацію на стороні клієнта». На традиційних веб-сайтах браузер запитує документ із веб-сервера, завантажує та оцінює ресурси CSS і JavaScript і відображає HTML, надісланий із сервера. Коли користувач натискає посилання, процес починається заново для нової сторінки. Маршрутизація на стороні клієнта дозволяє програмі оновлювати URL-адресу після натискання посилання без повторного запиту на інший документ із сервера.

Для підключення React Router до проекту потрібно виконати дії прописані на офіційному сайті бібліотеки «https://reactrouter.com/». Так як для розробки ми використовуємо Node.js інсталяція пакету відбувається наступним шляхом:

1. Відкриваємо термінал в програмі Virtual Studio Code (Рис. 2.1.);

2. Вписуємо в команду строку [npm install react-router-dom] для запуску процесу інсталяції (Рис. 2.9.)

Рис. 2.9 Початок інсталяції бібліотеки React Router.

3. Після появи повідомлення про успішне встановлення пакету (Рис. 2.10.) бібліотека готова до використання

Рис. 2.10 Повідомлення про успішне встановлення бібліотеки React Router.

2.5. Оптимізація роботи з формами за допомогою бібліотеки UseForm

React Hook Form зменшує кількість коду, який потрібно написати, одночасно видаляючи непотрібне повторне рендеринг. У вас є можливість ізолювати повторне відтворення компонентів, що покращує продуктивність вашої сторінки чи програми.

Для підключення React Router до проекту потрібно виконати дії прописані на офіційному сайті бібліотеки «https://react-hook-form.com/». Так як для розробки ми використовуємо Node.js інсталяція пакету відбувається наступним шляхом:

1. Відкриваємо термінал в програмі Virtual Studio Code (Рис. 2.1.);

2. Вписуємо в команду строку [npm install react-hook-form] для запуску процесу інсталяції (Рис. 2.11.)

Рис. 2.11 Початок інсталяції бібліотеки React Hook Form.

3. Після появи повідомлення про успішне встановлення пакету (Рис. 2.12.) бібліотека готова до використання.

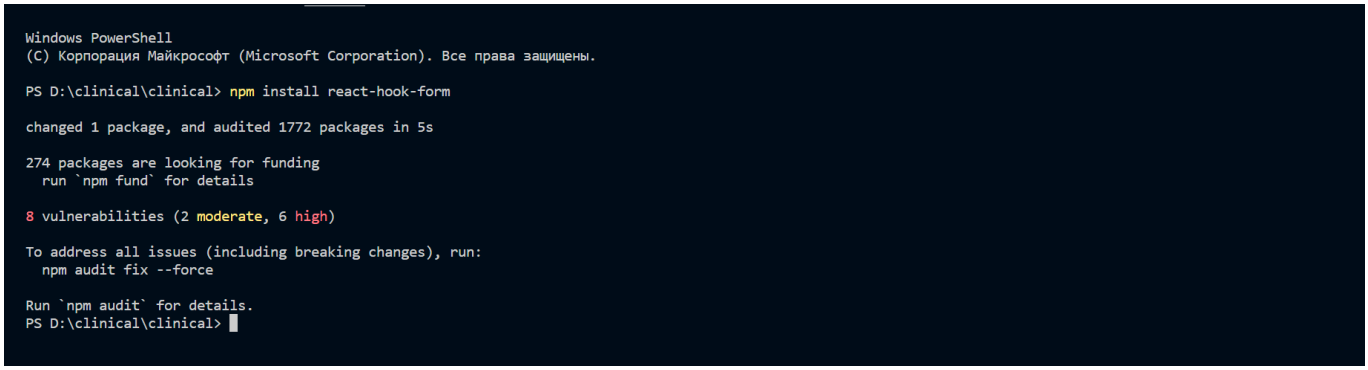


Рис. 2.12 Повідомлення про успішне встановлення бібліотеки React Hook Form.

2.6. Оптимізація бази даних за допомогою структуризації даних

Для максимальної ефективності керування даними сховища, розділи та індекси мають знаходиться в оптимальному стані для запланованого та фактичного використання у виробничому середовищі. Оптимізація використання даних покращує продуктивність запитів, оптимізує ресурси та покращує загальну ефективність системи.

Існує два типи оптимізації за допомогою яких можна значно покращити роботу веб-додатку Табл.2.1.. Згідно наведеного опису для нашого веб-додатку ми застосуємо два підходи в залежності від розділу в базі даних та покращення ефективності обробки даних [53].

Таблиця 2.1.

Типи оптимізації бази даних

Тип оптимізації	Визначення	Приклад	Випадки використання
Вертикальне розділення	Розділення таблиці на менші таблиці, вибравши певні стовпці або поля для кожного розділу. Кожен розділ представляє підмножину повних даних.	Якщо є таблиця зі стовпцями A, B, C і D, тоді можна створити одну таблицю зі стовпцями A і B, а іншу — зі стовпцями C і D	1. Таблиця містить багато стовпців, але запити не отримують доступ до всіх стовпців разом. 2. Деякі стовпці більші за інші, і їх розділення може покращити продуктивність введення-виведення. 3. Різні фрагменти даних мають різні моделі доступу.

Продовження таблиці 2.1.

Тип оптимізації	Визначення	Приклад	Випадки використання
Горизонтальне розділення	Розподіл даних на рядки або діапазони значень (також називається сегментуванням). Кожен розділ містить підмножину рядків зі схожими характеристиками.	Якщо є таблиця з рядками з 1 по 1000, можна створити одну секцію з рядками з 1 по 500, а іншу – з рядками з 501 по 1000.	1. Набір даних завеликий для одного місця або сервера. 2. Доступ до даних здійснюється на основі певних діапазонів або фільтрів. 3. Щоб підвищити продуктивність, необхідно розподілити навантаження між фізичними вузлами або серверами.

2.7.Оптимізація обробки запитів за допомогою створення API

Розробка API відіграє вирішальну роль у розробці програмного забезпечення, забезпечуючи зв’язок і обмін даними між різними програмними системами, програмами або компонентами. API (інтерфейс прикладного програмування) визначає набір правил і протоколів, які регулюють взаємодію компонентів програмного забезпечення один з одним [54].

Формати даних: API зазвичай використовують такі формати даних, як JSON (об’єктна нотація JavaScript) або XML (розширювана мова розмітки), щоб структурувати дані, якими обмінюються. JSON став стандартом де-факто завдяки своїй простоті, читабельності та широкій підтримці сучасних мов програмування.

Методи HTTP: API зазвичай використовують протокол HTTP (Hypertext Transfer Protocol) для зв’язку. Найпоширенішими методами HTTP, які використовуються в розробці API, є GET, POST, PUT і DELETE, які відповідають відповідно отриманню, створенню, оновленню та видаленню ресурсів.

Автентифікація та безпека: API часто вимагають автентифікації, щоб гарантувати, що лише авторизовані користувачі або системи можуть отримати доступ до захищених ресурсів. Загальні механізми автентифікації включають ключі API, OAuth і веб-токени JSON (JWT) [55].

2.8. Оптимізація навантаження інтерфейсу користувача

На прикладі веб-додатку для обробки та збереження результатів лабораторного дослідження ми маємо необхідність створювати елементи які можуть навантажити інтерфейс користувача і зменшити стабільність та продуктивність веб-додатку.

Саме для цього можна частково перенести функціонал з інтерфейсу користувача на сторону сервера.

Одним з таких елементів є результати у PDF-форматі, де вказано результати дослідження та заключення лікаря. Замість того, щоб створювати форму на стороні інтерфейсу користувача це можна перенести на сторону сервера зменшивши завантаженість та зменшити кількість запитів для відправки форми назад на сервер. Додатково це забезпечить цілісність даних та допоможе реалізувати збереження версій при редагуванні направлень після завершення дослідження.

Другий елемент – це QR-код, за допомогою якого ми можемо налаштувати навігацію по направленням та пацієнтам, і також відслідковувати статус направлення, для забезпечення цілісності даних та блокування несанкціонованих змін у направленні та помилкових дублікатів результатів.

Перенесення цих двох функцій має зменшити навантаженість та знімає необхідність у створенні додаткових компонентів та функцій на стороні інтерфейсу користувача.

3 АНАЛІЗ І УЗАГАЛЬНЕННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

3.1. Результати оптимізації за допомогою бібліотеки Million.js

Після інсталяції бібліотеки, вона автоматично запускається та оптимізує усі функції, що знаходяться на стороні інтерфейсу користувача. Бібліотека оцінює час роботи кожної функції, їх важливість та значимість відносно усього проекту. За допомогою Million lint (вбудований плагін у застосунку Visual Studio Code) при компіляції веб додатку кожна функція і елемент оцінюється на продуктивність по часу обробки інформації. Таким чином після написання функції ми бачимо час обробки і можемо визначити слабкі місця у нашому веб-додатку.

Надалі визначивши повільні елементи, ми можемо оптимізувати, змінивши частину коду, або провести дефрагментацію на більш дрібні функції, які будуть обробляти лише ту частину, яка необхідна.

Тим самим ми можемо під час розробки контролювати кожен елемент, виявляти слабкі місця та оптимізувати роботу усього інтерфейсу користувача. Також, своєчасне виявлення не оптимізованих функцій допомагає зекономити час розробки, адже після публікації веб-додатку та відкриття доступу до клієнта, внести будь які зміни стає вкрай складним і може призвести то зупинки роботи веб-додатка.

Отже, використання даного засобу оптимізації є вкрай ефективним через його простоту виявлення неоптимізованих частин та надання доступу до інформації, яку неможливо отримати без сторонніх сервісів або додатково створених юніт тестів на серверній стороні. Тому, що сторонні сервісі роблять веб-додаток більш вразливим та залежним від зовнішніх факторів.

3.2. Результати оптимізації за допомогою бібліотеки React Redux

Після інсталяції бібліотеки ми можемо створювати сховища та передавати дані між елементами без зв'язування їх між собою. Це значно зменшує кількість написаного коду та дозволяє уникати зайвих зв'язувань між дочірними та батьківськими елементами. Для послідовного синтексису створюється файл index.js як основне сховище усіх функцій react redux (Рис. 3.1).

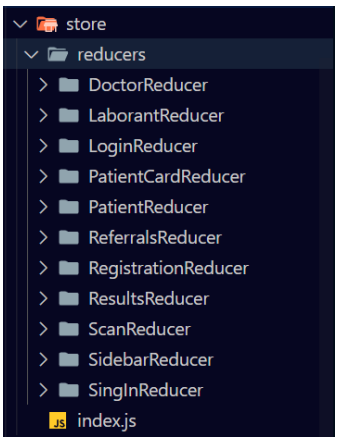


Рис. 3.1 Структура файлів для сховища у веб-додатку

Функція у бібліотеці react redux яка записує та передає данні між елементами називається reducer. Кожна окрема функція працює автономно та дозволяє зберігати дані передані до неї локально. Також у цій функції дані можна змінювати та передавати зміни назад. Для виклику цих функцій в середині компоненту їх дані та стан потрібно зберігати в сховище (на Рис.3.2 «configureStore»). За допомогою вбудованої функції в бібліотеці ми його викликаємо, зберігаємо в середину усі функції які записані для нашого веб-додатку в папці reducers (Рис.3.1.). Для доступу у всіх елементах проекту функцій зі сховища необхідно його експортувати «export default store» (Рис.3.2).

```
import mainSlice from "./reducers/SidebarReducer/SidebarReducer";
import { configureStore } from "@reduxjs/toolkit";
import loginSlice from "./reducers/LoginReducer/LoginReducer";
import RegFormSlice from "./reducers/RegistrationReducer/Form"
import labFormSlice from "./reducers/LaborantReducer/Form";
import LaborantSlice from "./reducers/LaborantReducer/LaborantReducer";
import DocFormSlice from "./reducers/DoctorReducer/Form";
import DoctorSlice from "./reducers/DoctorReducer/DoctorReducer";
import WorkersSlice from "./reducers/PatientReducer/PatientReducer";
import RegistrationSlice from "./reducers/RegistrationReducer/RegistrationReducer";
import typeOfEnterSlice from "./reducers/LoginReducer/Form";
import RegistrationScanSlice from "./reducers/ScanReducer/RegistrationScanReducer";
import ResultsSlice from "./reducers/ResultsReducer/ResultsReducer";
import ReferralsSlice from "./reducers/ReferralsReducer/ReferralsReducer";
import PatientCardSlice from "./reducers/PatientCardReducer/PatientCardReducer";

const store = configureStore({
    reducer: {
        main: mainSlice.reducer,
        login: loginSlice.reducer,
        workers: WorkersSlice.reducer,
        referrals: ReferralsSlice.reducer,
        regForm: RegFormSlice.reducer,
        regFormOutput: RegistrationSlice.reducer,
        typeOfEnterSlice: typeOfEnterSlice.reducer,
        labForm: labFormSlice.reducer,
        labFormOutput: LaborantSlice.reducer,
        docForm: DocFormSlice.reducer,
        docFormOutput: DoctorSlice.reducer,
        RegScan: RegistrationScanSlice.reducer,
        ResultsSlice: ResultsSlice.reducer,
        patientCard: PatientCardSlice.reducer,
    },
});

export default store;
```

Рис. 3.2 Основне сховище функцій створене за допомогою React Redux

В середині функції ми створюємо елемент, який пізніше передається у сховище для відстежування його стану, та доступності в окремих елементах. Для прикладу на Рис.3.3 зображено функцію, яка зберігає всередині стан форми заключення лікаря («formCompleted: false»). За замовчуванням форма завжди не заповнена і лише після заповнення та підпису лікарем стан змінюється на «true», та дозволяє відправити запит до бази даних на формування протоколу. Це допомагає запобігти відправки пустих значень, адже валідація цього процесу перевіряє стан форми на декількох етапах.

```
import { createSlice } from '@reduxjs/toolkit';

const DoctorSlice = createSlice({
    name: 'doctorOutputSlice',
    initialState:{
        formCompleted: false,
        docotorCard: {
            doc: '',
            data:{
                archive: '',
                sign:{},
                table: {},
                color: '',
                description: '',
                conclusion: '',
            }
        }
    },
    reducers: {
        CISHCreation: (state,action) =>{
            state.formCompleted = true
            state.docotorCard = action.payload.credential
        },
        ExpressCreation: (state,action) =>{
            state.formCompleted = true
            state.docotorCard = action.payload.credential
        },
        GistologyCreation: (state,action) =>{
            state.formCompleted = true
            state.docotorCard = action.payload.credential
        },
        IGHCreation: (state,action) =>{
            state.formCompleted = true
            state.docotorCard = action.payload.credential
        },
        CardClean: (state, action) =>{
            state.formCompleted = false
            state.docotorCard = {
```

Рис. 3.3 Функція обробки та валідації форми заключення лікаря

```
                doc: '',
                data:{
                    archive: '',
                    sign:{},
                    table: {},
                    color: '',
                    description: '',
                    conclusion: '',
                }
            }
        }
    }
})

export const {CISHCreation, IGHCreation, GistologyCreation, ExpressCreation, CardClean} = DoctorSlice.actions;
export default DoctorSlice;
```

Продовження Рис. 3.3 Функція обробки та валідації форми заключення лікаря

Також як видно з коду (Рис.3.3.) ми можемо валідувати саму структуру даних, визначаючи їх тип (object, string, number, Boolean etc.) щоб уникнути помилки при відправці до бази даних. Для створення функції ми викликаємо вбудовану в біліотеці react redux функцію «createSlice», яка зберігає початковий стан функції та надає можливість змінювати стан за допомогою «reducers». «Slice» має в собі три основні компоненти:

1. name – назва функції яку ми зберігаємо в сховище та можемо через це ім’я передавати дані до елементів проекту;

2. initialState – об’єкт в якому зберігається початковий стан функції;

3. reducers – це методи які дозволяють змінювати початковий стан функції, який пізніше можна передавати між елементами;

Вкінці ми окремо експортуємо методи, які викликаються всередині елементів, та стан функції, який зберігається всередині сховища. Також кожен метод приймає два параметри «state» та «action»:

- state – це стан функції, початковий або змінений, який ми можемо змінити;
- action – параметри які приймає метод з елементу в якому він був викликаний;

Отже, на прикладі елементу, який відповідає за створення заключення лікаря ми можемо побачити, що після створення заключення та підпису лікарем ми змінюємо параметр «formCompleted» на true, що дозволяє потім відправити

дані до бази даних. Також, дані з форми ми завантажуємо в середину за передаючи параметр «payload.credetial» за допомогою «action», оновлюючи початковий стан «doctorCard» з пустого на заповнений і вже після цього його відправляти на серверну частину. В кінці після відправки даних ми викликаємо метод «CardClean» та змінюємо стан функції на початковий.

Ще один приклад, ефективності використання бібліотеки react redux можна побачити на прикладі створення форми для заповнення (Рис.3.4.).

```
import { createSlice } from '@reduxjs/toolkit';
import {values} from "./FormData";

const RegFormSlice = createSlice({
    name: 'regFormInput',
    initialState: values,
    reducers: {
        formCreation: (state, action) => {
            state.push(action.payload);
        },
        pushElements: (state, action) => {
            state.fields.map(el => {
                if(el.name === action.payload.credential.id) {
                    el.props.push({
                        "value": action.payload.credential.obj,
                        "label": action.payload.credential.obj,
                        "id": action.payload.credential.obj,
                        "type": 'option'
                    })
                }
            })
        }, pushToRegistrationForm: (state, action) => {
            state.fields.map(el => {
                if(el.name === action.payload.formName) {
                    el.props.push({
                        "value": action.payload.name,
                        "label": action.payload.name,
                        "id": action.payload.id,
                        "type": 'option'
                    })
                }
            })
```

Рис. 3.4 Функція передачі значень для створення форми реєстрації пацієнта.

```
        },
    }
})

export const {formCreation, pushElements, pushToRegistrationForm} = RegFormSlice.actions;
export default RegFormSlice;
```

Продовження Рис. 3.4 Функція передачі значень для створення форми реєстрації пацієнта.

Як ми бачимо на Рис.3.4. структура функції така сама як і на Рис.3.3.. Ми зберігаємо початкові дані для форми у форматі json окремим файлом та беремо їх як початковий стан функції, після чого ми викликаємо ці значення в елементі який відповідає за формування цієї форми. Також ми можемо створювати динамічні поля у формі, які користувач може змінювати за потреби. На Рис.3.5. ми можемо побачити два елементи («Заклад який скеровує» та «Відділення закладу») куди користувач може додавати елементи за необхідності. Для того, щоб не викликати постійний ре-рендер сторінки з оновленням форми, ми за допомогою функції додаємо локально елемент і додатково відправляємо дані до бази даних, щоб зберегти це для наступних заповнень. Тим самим ми запобігаємо зайвих витрат в часі та навантажень і полегшуємо сприйняття додатку користувачем.

Рис. 3.5 Приклад динамічного елементу на сторінці форми для реєстрації пацієнту

Для передачі даних з елементу до функції ми викликаємо вбудований хук «useDispatch()» (Рис.3.6). Для зменшення кількості символів в середині елементу зазвичай створюють констату «dispatch», якій в якості значення присвоюють хук, це також дозволяє викликати цей хук в середині функцій та інших хуках, що в

свою чергу дає можливість додатково валідувати процес та запобігати запису пустих значень в середину сховища.

```
import React, { useEffect, useState } from 'react'
import { useForm } from 'react-hook-form'
import classes from './doctorScan.module.scss'
import { doctorService } from '../../api/doctorService'
import { useDispatch } from 'react-redux'
import { ScanCompleted } from '../../store/reducers/ScanReducer/RegistrationScanReducer'

Tabnine | Edit | Explain
const DoctorScan = ({onChange}) => {
    const {
        register,
        handleSubmit
    } = useForm()

    const [scan, setScan] = useState({
      "status": '',
      "quantityMaterial": '',
      "research": '',
      "id": '',
    })

    const dispatch = useDispatch()

    const onSubmit = async(data) =>{
        let id = (eval('({obj:' + data.resultsScan + '})')).obj.id
        await doctorService.getDirection(id).then(res=>{
          if(res?.data){
            onChange({
              "id": id,
              "status": res?.data.status,
              "research": res?.data.research
            })
            setScan({
              "status": res?.data.status,
              "quantityMaterial": res?.data.quantityMaterial,
              "research": res?.data.research,
              "id": id
            })
          }
        })
    }
```

```
    useEffect(()=>{
        if (scan.research !== ''){
          console.log(scan)
          dispatch(ScanCompleted({credential: scan}))
        }
      },[scan])
```

Рис. 3.6 Приклад передачі даних до сховища React Redax

Отже, як ми бачимо на Рис.3.6. отримуючи значення з поля де сканується QR-code ми створюємо стан за допомогою «useState()» з даними що були зашифровані в коді, після чого ми перевіряємо стан і якщо він не містить пусті значення передаємо їх до нашого сховища. Пізніше інші елементи використовуватимуть ці значення для вибору форми, що з'являється на сторінці та валідації стану дослідження, щоб не порушити порядок дій виконання дослідження. Для виклику цих значень в будь-якому іншому елементі ми використовуємо вбудований хук «useSelector()», який допомагає отримувати записані значення до сховища в елементах не залежно від їх розташування (Рис.3.7.).

```
import React, { useEffect, useState } from 'react'
import classes from './scanFinished.module.scss'
import { useForm } from 'react-hook-form'
import { useDispatch, useSelector } from 'react-redux'
import { resultsFormating } from '../../store/reducers/ResultsReducer/ResultsReducer'
import {ClearInputs} from '../../store/reducers/ScanReducer/RegistrationScanReducer'
import { laborantService } from '../../api/laborantService'

Tabnine | Edit | Explain
const ScanFinished = ({numbers}) => {

    const dispatch = useDispatch()
    const regForm = useSelector(state=>state.RegScan.ScanInputs)
```

Рис. 3.7 Приклад передачі значень зі сховища до елементів

Як ми бачимо на Рис.3.7. викликаючи вбудований хук ми визначаємо, що викликаємо «state», після чого зазначаємо, що нам потрібен стан функції(яку ми визначаємо по параметру «name», та за необхідності вибираємо яке саме значення стану нам необхідно відслідковувати. І тепер в середині елементу ми можемо отримувати значення та визначати зміну цього стану, що дозволяє створювати валідаційні перевірки стану та маніпулювати елементами відповідно від стану який ми отримали.

Отже, можна стверджувати, що дана бібліотека значно оптимізує роботу веб-додатку, за рахунок зменшення кількості написаного коду та запобігання

утворення «контрольованих» та «не контрольованих» зв’язувань між елементами. Адже, одною з особливостей React.js, це неможливість просто передачі даних від дочірнього то батьківського елементу, а лише тільки через «контрольоване» зв’язування, що може призвести до виникнення помилок та збою роботи додатку.

3.3 Результати оптимізації HTTP-запитів за допомогою бібліотеки Axios

Після встановлення бібліотеки ми можемо створити структуру, яка значно полегшить роботу розробнику та зменшить кількість написаного коду. Це дозволить створити типічні запити, які можна використовувати в будь-якому елементі. Для послідовного синтексису створюється файл instance.js як основний файл в якому зберігається вся інформація по роботі з сервером (Рис. 3.8).

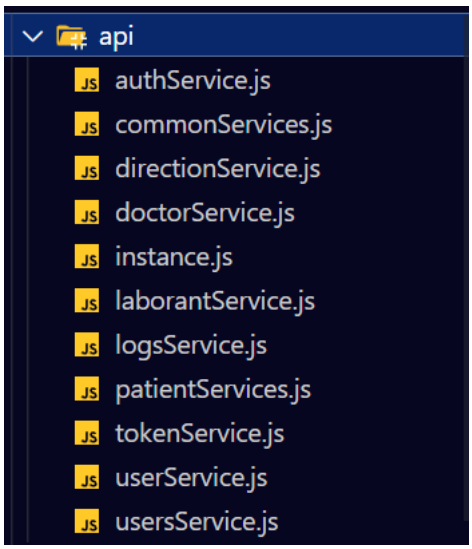


Рис. 3.8 Структура файлів для веб-запитів бібліотеки Axios

На Рис.3.9. зображена структура основного файлу. В ньому ми визначаємо основну адресу для запиту та визначаємо для неї змінну (наприклад API_URL). Також в цьому файлі ми оброблюємо усі типи відповідей від серверу (успішні та помилкові) і керуємо «TokenService» як додатковий елемент захисту додатку

```
import axios from "axios";
import {TokenService} from "./tokenService";
import errorMessage from "../const/Tost";

export const API_URL = 'http://91.234.32.114:3001';

const $api = axios.create({
    withCredentials: true,
    baseURL: API_URL
})

Tabnine | Edit | Test | Explain | Document | Ask
$api.interceptors.request.use((config) => {
    config.headers.Authorization = `Bearer ${localStorage.getItem('token')}`
    return config;
})

$api.interceptors.response.use(
    Tabnine | Edit | Test | Explain | Document | Ask
    (response) => {
        return response; // Return the response if it's successful
    },
    Tabnine | Edit | Test | Explain | Document | Ask
    async (error) => {
        const originalRequest = error.config;

        // Avoid retrying the token refresh request itself
        if (error.response.status === 401 && originalRequest && !originalRequest._isRetry && originalRequest.url !== `${API_URL}/refresh`) {
            originalRequest._isRetry = true; // Mark the request as retried to avoid looping

            try {
                // Send a request to refresh the access token
                const response = await $api.get(`${API_URL}/refresh`, { withCredentials: true });

                // If we receive a new access token, save it and retry the original request
                if (response?.data?.accessToken) {
                    TokenService.saveToken(response.data.accessToken);

                    // Add the new token to the original request's headers
                    originalRequest.headers.Authorization = `Bearer ${response.data.accessToken}`;

                    // Retry the original request with the updated token
                    return $api.request(originalRequest);
                } else {
                    // If there's no access token in the response, handle it as a failed refresh
                    throw new Error('Failed to refresh token');
                }
            } catch (refreshError) {
                // Handle errors during the token refresh
                errorMessage(refreshError, refreshError?.response?.data?.message);

                // Optionally log out the user or redirect to login if the refresh fails
            }
        }

        // If it's a different error or the refresh failed, rethrow the error
        throw error;
    }
);

export default $api;
```

Рис. 3.9 Основний компонент бібліотеки Axios для валідації HTTP запитів

У випадку виникнення помилки, ми повідомляємо користувачу про це за допомогою бібліотеки «Tost» (приклад помилки на Рис.3.10). Виводячи суть помилки, що покращує роботу з інтерфейсом і допомагає зрозуміти в чому може бути причина.

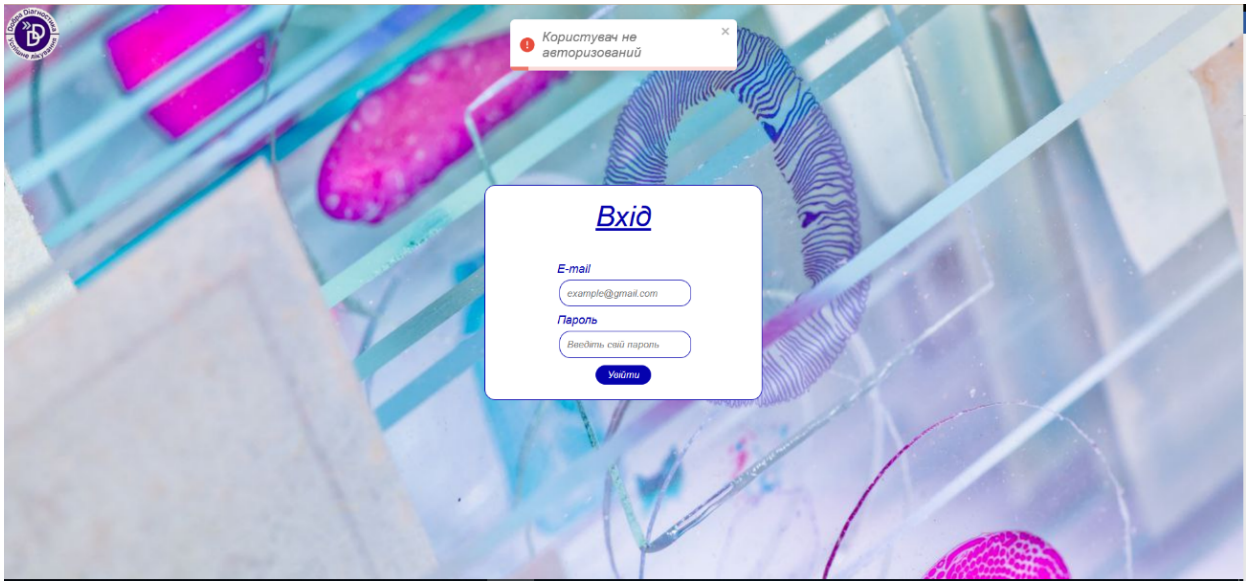


Рис. 3.10. Приклад повідомлення про помилку

Після цього ми можемо створювати окремі запити без повторних валідацій помилки, що допомагає зменшити кількість написаного коду.

«TokenService» - або сервіс для отримання та оновлення коду доступу. Цей сервіс допомагає убезпечити користувача від несанкціонованого доступу до даних. Цей ключ отримується початково при вході користувача до системи та має час «валідності» - час за який цей ключ дає доступ до редагування, запиту та запису нових даних. Після проходження часу «валідності» до серверу відправляється запит на оновлення і після успішного запиту відправляється оновлений ключ (Рис.3.11.)

```
import $api, {API_URL} from "./instance";…

const TOKEN_KEY = 'auth_token';

export const TokenService = {
    async refreshToken(){
        try{
            const response = await $api.get(`${API_URL}/refresh`);

            if (response?.data){
                localStorage.setItem(TOKEN_KEY, response.data.accessToken);
                import $api, {API_URL} from "./instance";
                import errorMessage from "../const/Tost";
                import {jwtDecode} from "jwt-decode";

                const TOKEN_KEY = 'auth_token';

                export const TokenService = {
                    async refreshToken(){
                        try{
                            const response = await $api.get(`${API_URL}/refresh`);

                            if (response?.data){
                                localStorage.setItem(TOKEN_KEY, response.data.accessToken);
                                return `Bearer ${response.data.accessToken}`;
                            }
```

Рис. 3.11 Компонент проекту веб-додатку «TokenService»

```
            } catch(error){
                errorMessage(error, error?.response?.data?.message);
            }
        },
        saveToken(token){
            localStorage.setItem('token', token);
        },
        getToken(){
            return localStorage.getItem('token');
        },
        getInfoFromToken(){
            const tokenData = localStorage.getItem('auth_token');
            return jwtDecode(tokenData);
        },
        removeToken(){
            localStorage.removeItem(TOKEN_KEY);
```

Продовження Рис. 3.11 Компонент проекту веб-додатку «TokenService»

Далі ми розглянемо один із компонентів в яких створюється шаблон HTTP-запиту на Рис.3.12., де зображено набір функцій для створення та редагування форми заключення лаборанта та відслідковування стадії дослідження.

```
import $api, {API_URL} from "./instance";

export const laborantService = {
    async getDirection(id){
        try{
            return await $api.get(`${API_URL}/direction/manage/${id}?withAssociates=true`,)
        } catch(error){
            errorMessage(error, error?.response?.data?.message);
        }
    },
    async updateLabForm(id, data){
        try{
            return await $api.put(`${API_URL}/direction/manage/${id}/laboratory-form`, data)
        } catch(error){
            errorMessage(error, error?.response?.data?.message);
        }
    },
    async updateStatusDirection(id, data){
        try{
            return await $api.put(`${API_URL}/direction/manage/${id}`, data)
        } catch(error){
            errorMessage(error, error?.response?.data?.message);
        }
    }
}
```

Рис. 3.12 Приклад компоненту для HTTP-запитів

Структура для усіх цих функцій однакова, спочатку ми імпортуємо з головного компоненту «instance» два основних компоненти «$api» та «API_URL»:

- $api – об’єкт в якому знаходиться основні «headers» для HTTP-запиту;
- API_URL – перемінна, в яку занесена інформація про адресу запиту, та була задекларована в основному файлі бібліотеки;

Після імпорту основних компонентів ми додатково імпортуємо бібліотеку «Tost» для відображення помилки користувачу. Далі компонент складається з основної функції, що експортується до компонентів проекту. Основна функція має всередині під функції, які відповідають за той чи інший запит для нашого направлення. Основна функція допомагає в компонентному розділенні структури HTTP-запитів та кращої читабельності коду. Під функції всередині основної усі асинхронні, адже це типовий вид функцій для запитів, від яких ми очикуємо отримати щось у відповідь. Кожна під функція має окрему назву, яка викликається в середині елементу та параметри які вона приймає. В залежності від того, що необхідно для запиту це може бути наприклад частина адреси для запиту (наприклад на Рис.3.12 кожен компонент очікує параметр «id», який виступає у вигляді ідентифікатора пацієнта та дозволяє отримати необхідне направлення) та значення, які будуть записані до бази даних (наприклад на Рис.3.12. для функції «updateLabForm()» ми отримуємо параметр «data», в якому зберігається інформація яку вводить лаборант та записує в історію дослідження). Для отримання помилкових статусів використовуємо стандартний конструктор «try…catch» який відслідковує стан запиту і у випадку помилки висвічує її користувачу (приклад Рис.3.10.).

На Рис.3.13 зображено приклад імпорту функцій для HTTP-запитів. Ми імпортуємо тільки основну функцію у вигляді об’єкта і вже від основної функції ми будемо викликати необхідну під функцію та передавати до неї значення.

```
import React, { useEffect, useState, useRef } from 'react'
import Form from '../../../Forms/EnterForm'
import classes from './docotorForm.module.scss'
import { useDispatch, useSelector } from 'react-redux'
import { CISHCreation, IGHCreation, GistologyCreation, ExpressCreation, CardClean } from '../../../../store/reducers/DoctorReducer/DoctorReducer'
import { doctorService } from '../../../../api/doctorService'
```

Рис. 3.13 Приклад імпорту об’єкту основної функції

Після імпорту основної функції ми отримуємо доступ до усіх внутрішніх функцій та викликаємо їх у необхідний час, передаючи усі необхідні параметри (наприклад Рис.3.14.. ми створюємо асинхронну функцію по формуванню протоколу заключення лікаря та змінюємо статус дослідження на завершений).

```
const DoctorResult = async(id,data) => {
    await doctorService.updateDocForm(id,data).then(res=>{
        if(res) console.log('completed')
    })
    await doctorService.updateStatusDirection(id, {"status":'finished'}).then(res=>{
        if(res) console.log('updated')
    })
    if(id){
        printResult(id)
    }
}

Tabnine | Edit | Test | Explain | Document | Ask
useEffect(()=>{
    if(doctorCard.data.description !== ''){
        if (type.research == 'CISH'){
            dispatch(CISHCreation({credential: doctorCard}))
        } else if (type.research == 'ІГХ'){
            dispatch(IGHCreation({credential: doctorCard}))
        } else if (type.research == 'Гістологія'){
            dispatch(GistologyCreation({credential: doctorCard}))
        } else if (type.research == 'Експрес'){
            dispatch(ExpressCreation({credential:doctorCard}))
        }
        DoctorResult(type.id,doctorCard)
        if(print){
            dispatch(CardClean())
            formFinished(false)
            setRenderForm(false)
        }
    }
},[doctorCard,print])
```

Рис. 3.14 Приклад використання під функції у бібліотеці Axios

Отже ми бачимо, що функція «DoctorResult()» відповідає за запит. В неї ми передаємо параметр «id» та «data». В ній ми одразу використовуємо два запити:

1. updateDocForm – передаємо усі значення які отримали з форми заключення лікаря та зберігаємо їх у базі даних;

2. updateStatusDirectiob – де ми змінюємо статус дослідження на завершений та сповіщаємо пацієнта про отримання результатів дослідження;

І вже після відправки заключення ми передаємо параметр «id» у функцію «printResult()» для формування pdf-файлу заключення. «UseEffect()» хук допомагає відслідкувати зміни стану елементів які відповідають за значення у формі заключення і у випадку якщо дані не пусті (ми перевіряємо параметр «description», який є обов'язковий у формі заключення), і якщо все відвалідовано ми передаємо значення у сховище відповідно до типу дослідження та передаємо параметри до запитів. Після чого ми керуючи елементами перекидаємо користувача на початкову сторінку сканування направлення.

Отже, розглянувши основні принципи та методи застосування бібліотеки Axios ми можемо зробити висновок, що дана технологія дуже оптимізує написаний код, полегшує його сприйняття розробниками. Значно спрощує валідацію помилок отриманих від серверної частини і можемо їх легко виводити користувачу, щоб він вчасно отримував інформацію і розумів причини виникнення їх. Також багаторазовість використання функцій запиту у різних елементах з меншою кількістю написаного коду значно прискорює роботу веб-додатку та зменшує навантаженість серверної частини.

3.4 Результати оптимізації за допомогою бібліотеки React Router

Після інсталяції бібліотеки ми отримуємо повний доступ до її функцій та створюємо структуру маршрутизації (Рис.3.15.), що допомагає переходити між сторінками веб-додатку без оновлення сторінки, що зменшує навантаження та у разі повільного підключення до інтернету значно допоможе користувачу у роботі.

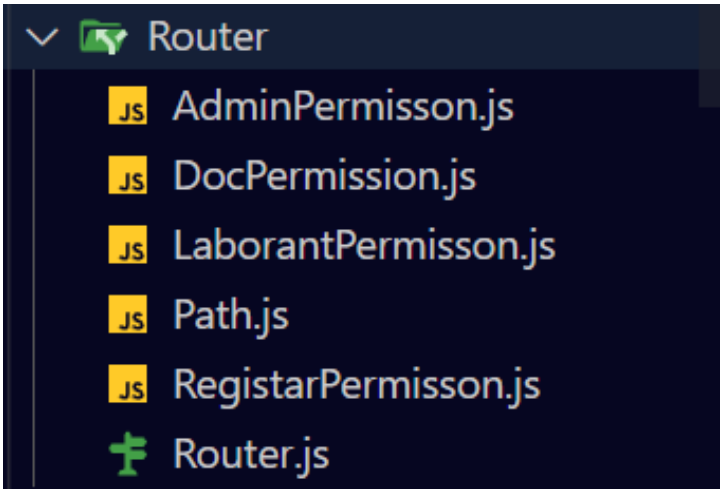


Рис. 3.15 Структура фалів для маршрутизації сторінок у веб-додатку

У структурі проекту ми створюємо окрему директорію «Router», яка містить два основних файли «Path.js» та «Router.js»:

• Path.js – файл, в якому знаходяться усі посилання на сторінки, на які повинен переходити користувач;

• Router.js – файл, який містить основну логіку маршрутизації та валідації доступу користувача до сторінки;

«Path.js» містить в собі об’єкт «PATH» (Рис.3.16.) в якому ми визначаємо шлях до сторінки відповідно до її назви. Після декларування об’єкту ми його експортуємо до файлу «Router.js».

```
export const PATH = {
    ErrorPage: '/ErrorPage',
    FormPage: '/FormPage',
    Home: '/Home',
    WorkersList: '/WorkersList',
    PatientList: '/PatientList',
    ScanPage: '/ScanPage',
    RegistrationForm: '/FindPatientForm',
    LaborantForm: '/LaborantForm',
    DoctorForm: '/DoctorForm',
    Referrals: '/Referrals',
    Anamnez: '/PatientCard/:id',
    NewUser: '/CreateNewUser',
    EditUser: '/EditUser/:id',
    EditPatient: '/EditPatient/:id',
    ReferralCard: '/ReferralCard/:id',
    LogsPage: '/LogsPage'
```

Рис. 3.16 Структура файлу «Path.js» та його основні елементи

«Router.js» - це основний файл маршрутизації, в якому можна виділити такі компоненти:

1. Імпортовані елементи веб-додатку які відповідають та валідують процес навігації;

2. router - це маршрутна карта, яка визначає батьківські, дочірні елементи та валідацію доступу до елементу.

Як ми бачим на Рис.3.17 перша частина коду складається з імпортованих елементів після яких ми декларуємо об’єкт «router» в середині якого ми викликаємо функцію «createBrowserRouter()». Ця функція створює масив об’єктів, де ми створюємо основний батькивський елемент та надаємо йому значення «path» у вигляд ‘/’, що дасть змогу відобрадати його коли ми вводимо адресу без будь яких компонентів він буде відображатись за замовчуванням. В нашому проекті це основний елемент «App» в якому компілюється увесь додаток. В разі помилкового переходу ми декларуємо «errorElement», який буде відображатись у разі помилкового з’єднання або вписування значень в адресу сайту, які не існують. Після чого ми декларуємо дочірні елементи «children», з яких і складається проект. Кожен дочірній елемент має такі параметри:

- path – шлях який веде до компоненту та був задекларований в елементі «Path.js»;
- element – тут декларується елемент який планується до відображення та за необхідності додається валідація доступу користувача;

```
import * as React from "react";
import {
    createBrowserRouter,
} from "react-router-dom";
import ErrorPage from "../Pages/ErrorPage";
import App from "../../App";
import {PATH} from "./Path";
import Home from "../Pages/Home";
import FormPage from "../Pages/Forms/FormPage";
import WorkersPage from "../Pages/WorkersFolder/WorkersPage";
import ScanPage from "../Pages/Forms/ScanPage";
import LaborantForm from "../Pages/Forms/LaborantForm";
import DoctorForm from "../Pages/Forms/DoctorForm";
import Referrals from "../Pages/ReferralFolder/PatientRefferals/ReferralsList";
import PatientCard from "../Pages/PatientFolder/PatientCard";
import AddUser from "../Pages/WorkersFolder/AddUser";
import EditUser from "../Pages/ReferralFolder/PatientRefferals/EditReferral";
import EditPatient from "../Pages/PatientFolder/EditPatient";
import PatientList from "../Pages/PatientFolder/PatientList";
import FindPatientForm from "../Pages/Forms/FindPatientForm";
import DocPermission from "./DocPermission";
import RefferalCard from "../Pages/ReferralFolder/PatientRefferals/RefferalCard";
import LogsPage from '../Pages/LogsPage/index'
import AdminPermission from "./AdminPermisson";
import LaborantPermission from "./LaborantPermisson";
import RegistarPermission from "./RegistarPermisson";

export const router = createBrowserRouter([
    {
        path: "/",
        element: <App />,
        errorElement: <ErrorPage />,
        children: [
            {
                path: PATH.FormPage,
                element:
                <DocPermission>
                    <FormPage />
                </DocPermission>,
            },{
                path: PATH.Home,
                element: <Home />,
            },{
                path: PATH.WorkersList,
                element:
                <AdminPermission>
                    <WorkersPage />
                </AdminPermission>,
            },{
```

Рис. 3.17 Структура маршрутної карти проекту

Як ми бачимо на Рис.3.17 деякі файли містять валідацію доступу. Наприклад елемент «<LogsPage/>» який відображає усі дії, що виконував користувач доступні лише для адміністратора, щоб він міг відслідковувати виконані дії та міг запобігти помилковим діям. На Рис.3.18 зображено основний

принцип валідації доступу, в якому відбувається фільтрування відповідно до ролі користувача.

```
import React from 'react';
import { Navigate } from 'react-router-dom';
import { useSelector } from 'react-redux';
import { toast } from 'react-toastify';

Tabnine | Edit | Explain
const AdminPermission = ({children}) => {
    const role = useSelector(state=>state.login.userInfo.role)

    if(role === 'admin'){
        return children
    }
    toast.error('Право доступу обмежене системним адміністратором!')

    return <Navigate to="/home" />;
};

export default AdminPermission;
```

Рис. 3.18 Структура валідації доступу відповідно до ролі користувача

Як ми бачимо викликаючи зі сховища дані про користувача за допомогою хука «useSelector()» ми на основі отриманих даних даємо доступ або, в разі невідповідності ролі ми відправляємо користувача на початкову сторінку та за допомогою бібліотеки «Toast» сповіщаємо його про обмеження прав у доступі Рис.3.19.

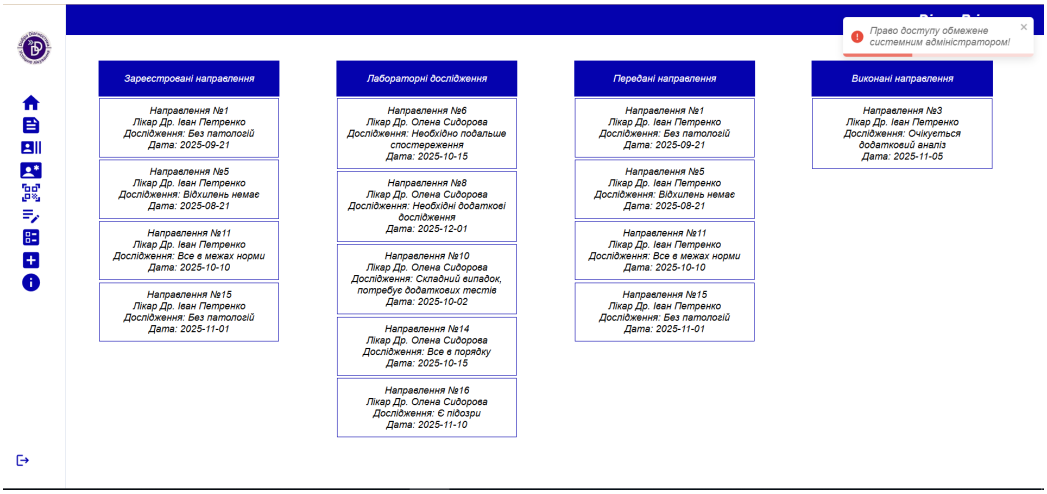


Рис. 3.19 Сповіщення про невідповідність прав доступу

Для повного застосування маршрутизації ми повинні у компоненті «index.js», в якому відбувається формування усієї структури веб-додатку, його компіляція та відображення користувачу, обгорнути компонент «App», який приймає в себе дочірні елементи в елемент бібліотеки «RouterProvider», який містить в собі параметр «router» з маршрутною картою усього веб додатку Рис.3.20

```
import React from 'react';
import ReactDOM from 'react-dom/client'; // Correct import
import App from './App';
import store from "./store";
import {router} from "./components/Router/Router";
import {RouterProvider} from "react-router-dom";
import {Provider} from "react-redux";

const root = ReactDOM.createRoot(document.getElementById('root'));  // No change here

root.render(
    <Provider store={store}>
        <RouterProvider router={router}>
            <App/>
        </RouterProvider>
    </Provider>
);
```

Рис. 3.20 Застосування маршрутної карти проекту до усіх елементів.

І після застосування маршрутної карти ми можемо валідувати доступ відповідно до ролі користувача та переходити між сторінками відповідно до сценарію дії або дії користувача.

Отже, бібліотека React-Router-Dom значно оптимізує веб-додаток, зменшуючи навантаження та додає додаткову валідацію доступу, запобігаючи несанкціонованого доступу. Навантаження зменшується через відсутність зайвих оновлень сторінки при переході між елементами, утворюючи стійку модель SPA(з англ. «Single Page Application» - односторінковий веб-додаток). Більшість розрбників притримуються правил та методів розробки SPA, через його стабільність та оптимізованість, і застосовують перехід із завантаженням лише у крайніх випадках, адже при нестабільному підключенні до інтернету,

користувачу доведеться знову очікувати поки браузер створить та скомпілює увесь написаний код та відобразить його на сторінку.

3.5 Результати оптимізації за допомогою бібліотеки UseForm

Після інсталяції бібліотеки ми маємо доступ до формування елементу «form» за допомогою додаткових інструментів. Для початку потрібно імпортувати хук «useForm()» Рис.3.21.

```
import React, { useEffect, useState } from 'react'
import classes from './smallLabForm.module.scss'
import { Controller, useForm } from 'react-hook-form'
import classNames from 'classnames';
import Select from 'react-select'
import { doctorService } from '../../../api/doctorService'
import { useSelector } from 'react-redux'
import { laborantService } from '../../../api/laborantService';
```

Рис. 3.21 Імпорт основних елементів для оптимізації компонента «form»

Також для оптимізації та валідації нам необхідно імпортувати додатково компонент «Controller» та «Select». Ці компоненти допоможуть створити динамічні «dropdown-select» елементи які зображені на Рис.3.22..

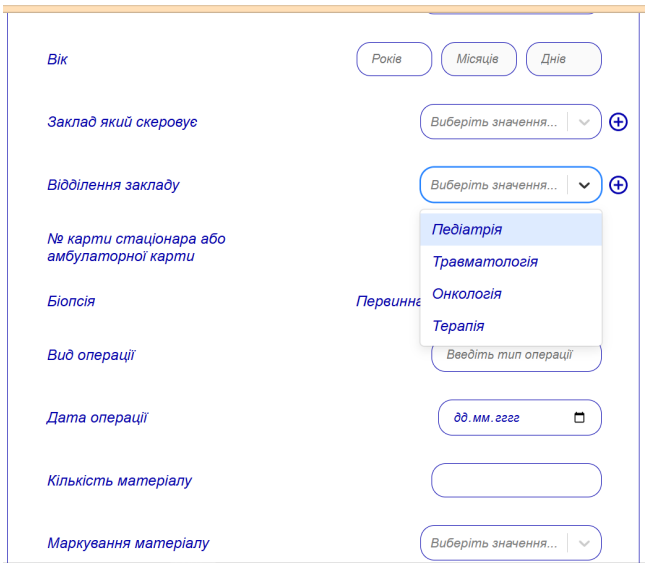


Рис. 3.22 Загальний список елементів у динамічну полі форми

В цих елементах можна гортати вниз до необхідного елементу або вводити ключові символи для сортування елементів в середині «селекту» та зменшення часу на пошук необхідного елементу Рис.3.23.

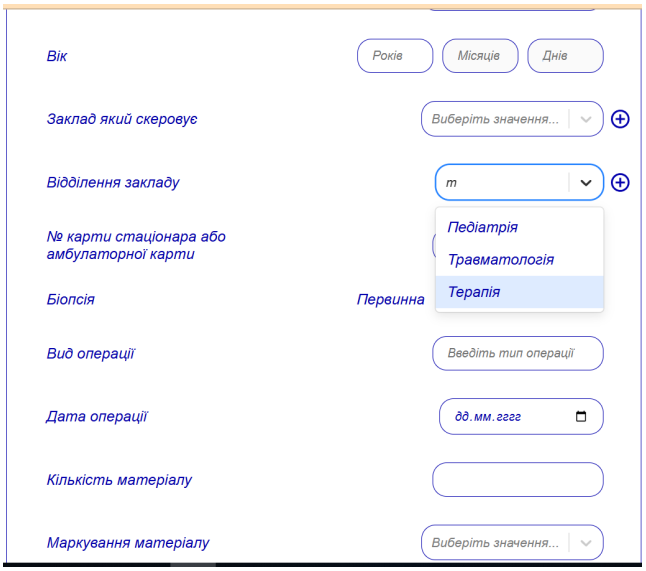


Рис. 3.23 Відфільтрований список по ключовому символу.

Все це значно полегшує роботу користувача, зменшуючи час на пошук необхідного елементу. Для роботи з бібліотекою ми декларуємо змінні та в якості значення даємо параметр «useForm()».Це означає що під час внесення цих змінних в якості параметрів та елементів компоненту їх функції будуть братись з бібліотеки Рис.3.24.

```
const {register, handleSubmit, setValue,trigger,watch,formState:
{errors},control=useForm()
```

Рис. 3.24 Основні елементи які ми використовуємо з бібліотеки «useForm»

Як показано на Рис.3.24 це лише частина компонентів, які можуть застосовуватись у даній бібліотеці, але це усі необхідні для нашого веб-додатку. Кожен компонент відповідає за свої функції:

- «register» - це параметр, який визначає ім’я елементу;

- «handleSubmit» - це основна функція яка відслідковує стандартну дію «submit» та отримує усі внесені дані із зареєстрованих полей форми;
- «setValue» - це параметр, який дозволяє задавати значення для елементу форми;
- «watch» - це функція яка відслідковує введення даних та роботу з елементом;
- «formState» - це параметр який дозволяє відслідковувати помилки при внесенні даних до форми та виводити їх користувачу;
- «control» - це параметр необхідний для створення складних елементів форми та контролювати роботу з ними;
- «trigger» -це параметр який може викликати функції в середині елементів форми;

Для отримання значень форми, які повинні відображатись користувачу створюємо у сховищі окремий масив об’єктів, який містить усю необхідну інформацію (Додаток А) та передаємо його за допомогою бібліотеки react redux у файл де компілюється форма.

В масиві є ключові об’єкти які несуть свої власні функції:

- Title – заголовок до самої форми який відображається над усіма елементами;
- dataType – валідаційний параметр який визначає рівень доступу до форми;
- formType – валідаційний параметр для сортування в елементах та виборі форми відповідно до типу дослідження;
- identifier – валідаційний параметр для сортування отриманих з форми значень;
- fields – масив об’єктів, який відповідає за створення компонентів форми:
    - legend – заголовок для поля яке відображається в середині форми;
    - id – ключ який необхідний при відображенні списку значень;
    - type – параметр який визначає тип елементу форми, в який буде внесено значення;

- required – Boolean значення, яке визначає необхідність внесення даних в елемент;
- name – назва поля, яке призначається елементу форми;
- options – Boolean значення, яке визначає приналежність елементу до динамічного типу;
- props – масив значень поля для вводу даних:
  - label – назва елементу для введеня даних, яке призначається, якщо елементів більше ніж один у блоці форми;
  - id – ключ який призначається, якщо елементів більше ніж один у блоці форми;
  - type – визначає тип введених даних (текст, число дата та ін.);
  - placeholder – параметр який допомагає користувачу зрозуміти, що потрібно вносити в елемент;

Для послідовного да коректного відображення форми, в елементі створюється валідація, яка відповідає за тип форми який буде відображено(відповідно до типу дослідження яке обирається. Та отримує необхідні значення з бази даних для динамічних полей, якщо вони присутні у формі (Рис.3.25).

```
values.map(el=>{
    if (el.formType === type.research){
        value = el
        value.fields.map(async(element)=>{
            if(element.legend == 'Лікар'){
                const res = await doctorService.getAllworkers()
                    if(res?.data){
                        res?.data.doctor.map(docs=>{
                            console.log(docs.name)
                            let obj = {
                                "value": docs.name,
                                "label": docs.name,
                                "id":docs.id,
                                "type": 'option'
                            }
                            props.push(obj)
                        })
                    }
            }
        })
    }
```

Рис. 3.25 Валідація параметрів форми, які будуть відображені на сторінці.

Після валідації ми можемо відобразити форму на сторінці,за допомогою цієї бібліотеки великі за обсягом елементів форми, займають значно менше місця в середині елементів та повністю валідовані за значеннями та подіями в середині (Додаток Б)

Як ми бачимо в Додатку Б. принцип написання дуже схожий зі звичним, без використання бібліотеки, але для кожного «inpput» ми маємо в середині валідаційні параметри, які визначаються за об'єктом Рис.3.24.. Також у елементі форм, є одразу функція «handleSubmit», яка попереджує оновлення сторінки під час завтердження введених даних, одразу валідує їх по значенням і вертає у функції назад структуровані дані форматом JSON, які ми одразу після перевірки можемо відправляти до бази даних.

Також одна з форм при реєстрації має складну валідацію. Вік який вносять має три ключових параметри: Рік, Місяць та День. Рік має значення від 0 до 102, вносити значення місяць можна лише, якщо поле Рік дорівнює 0, і відповідно день потребує значень полів Рік та Місяць 0. Ітреба валідувати цю логіку, щоб користувач не міг вносити параметри, які не відповідають логікі, яка описана вище. Це також легко реалізувати за допомогою Рис.3.26.

```
useEffect(() => {
    if (watch("year") === 0){
        document.querySelector('[name="mounth"]').disabled = false
    } else if (watch("year") > 0 || watch("year") === ''){
        document.querySelector('[name="mounth"]').value = ''
        document.querySelector('[name="day"]').value = ''
        document.querySelector('[name="mounth"]').disabled = true
        document.querySelector('[name="day"]').disabled = true
    }
}, [watch("year")])

Tabnine | Edit | Test | Explain | Document | Ask
useEffect(() => {
    if (watch("mounth") === 0){
        document.querySelector('[name="day"]').disabled = false
    } else if (watch("mounth") > 0 || watch("mounth") === ''){
        document.querySelector('[name="day"]').value = ''
        document.querySelector('[name="day"]').disabled = true
    }
}, [watch("mounth")]);
```

Рис. 3.26 Приклад використання параметру «watch»

Отже, як ми бачимо на Рис.3.26 параметром «watch» ми відслідковуємо будь-які дії з елементом і потім в разі відповідно внесених даних ми можемо валідувати поведінку користувача.

В результаті роботи з цією бібліотекою ми можемо зробити висновок, що вона значно оптимізує роботу с формами, які є ключовими в нашому веб-додатку, також допомагає валідувати внесення даних та на виході отримує відсортовані дані форматом, який повністю готовий до відправки на сторону серверу, що підвищує надійність додатку.

3.6 Результати оптимізації бази даних за допомогою структуризації даних

Згідно огляду літератури, ми визначили що існує два типи структури бази даних: вертикальна та горизонтальна. В нашому проекті ми використовуємо обидва типи структурування, в залежності від типу даних. На Рис.3.27 показана загальна структура серверної частини проекту.

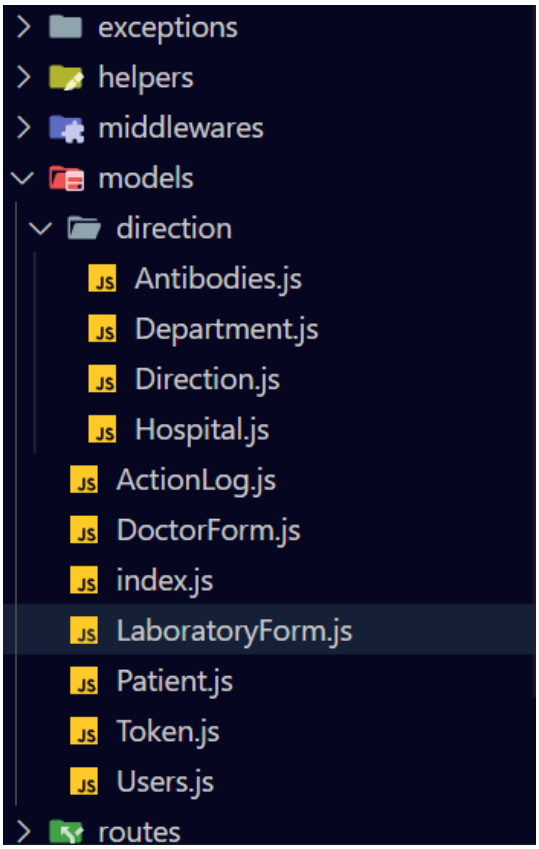


Рис. 3.27 Структура файлів у проекті, які передають масив даних

Кожен тип даних має свій «роут» (оброблювач подій, який передає, видаляє та редагує дані на стороні серверу) Рис.3.28.

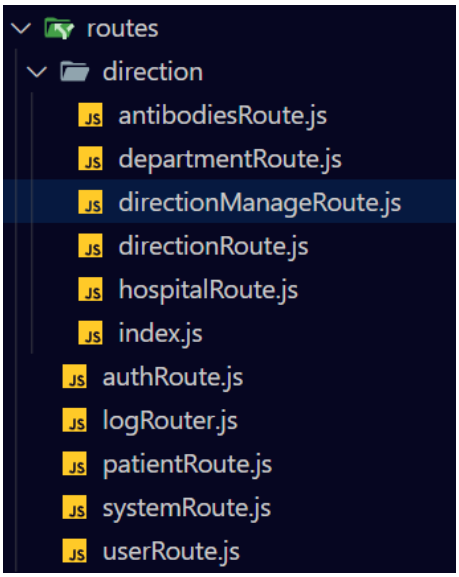


Рис. 3.28 Структура «роутів» у проекті веб-додатку.

У папці «models» усі дані які надходять на серверну частину, структуруються та валідуються, згідно визначених типів даних, які повинні зберігатись у базі даних отримуючи вертикальну структуру у самій базі даних. Приклад цієї логіки зображено в Додатку В. В якому отримується інформація про пацієнта, структурується і далі по маршруту передається до бази даних, де вона зберігається у форматі JSON.

Усі типи даних, що проходять структурювання компілюються в одному файлі Рис.3.29

```
const { Hospital } = require('./direction/Hospital');
const { Department } = require('./direction/Department');
const { Direction } = require('./direction/Direction');
const { Antibodies } = require('./direction/Antibodies');
const { Patient } = require('./Patient');
const { Token } = require('./Token');
const { User } = require('./Users');
const { LaboratoryForm } = require('./LaboratoryForm');
const { DoctorForm } = require('./DoctorForm');
const { ActionLog } = require('./ActionLog');

const { Aliases } = require('../constants')

User.hasMany(LaboratoryForm, { foreignKey: 'laboratoryAssistantId', as: Aliases.User.LAB_FORMS });
User.hasMany(LaboratoryForm, { foreignKey: 'doctorId', as: Aliases.User.LAB_FORMS_DOC });
User.hasMany(DoctorForm, { foreignKey: 'doctorId', as: Aliases.User.DOC_FORMS });
// User.hasMany(Direction, { foreignKey: 'doctorId', as: Aliases.User.DIRECTIONS }); // changed for static info about doctor

// Direction.belongsTo(User, { foreignKey: 'doctorId', as: Aliases.Direction.DOCTOR }); // changed for static info about doctor
Direction.belongsTo(Patient, { foreignKey: 'patientId', as: Aliases.Direction.PATIENT });
Direction.belongsTo(Hospital, { foreignKey: 'hospitalId', as: Aliases.Direction.HOSPITAL });
Direction.belongsTo(Department, { foreignKey: 'departmentId', as: Aliases.Direction.DEPARTMENT });
Direction.belongsTo(LaboratoryForm, { foreignKey: 'labFormId', as: Aliases.Direction.LAB_FORM });
Direction.belongsTo(DoctorForm, { foreignKey: 'docFormId', as: Aliases.Direction.DOC_FORM });

Direction.belongsToMany(Antibodies, { through: 'DirectionAntibodies', as: Aliases.Direction.ANTIBODIES });
Antibodies.belongsToMany(Direction, { through: 'DirectionAntibodies', as: Aliases.Direction.ANTIBODIES });

LaboratoryForm.belongsTo(User, { foreignKey: 'laboratoryAssistantId', as: Aliases.LaboratoryFrom.LABORATORY_ASSISTANT });
LaboratoryForm.belongsTo(User, { foreignKey: 'doctorId', as: Aliases.LaboratoryFrom.DOCTOR });

ActionLog.belongsTo(User, { foreignKey: 'userId', as: Aliases.ActionLog.USER });
ActionLog.belongsTo(User, { foreignKey: 'targetUserId', as: Aliases.ActionLog.TARGET_USER });
ActionLog.belongsTo(Patient, { foreignKey: 'patientId', as: Aliases.ActionLog.PATIENT });
ActionLog.belongsTo(Direction, { foreignKey: 'directionId', as: Aliases.ActionLog.DIRECTION });
```

Рис. 3.29 Компіляція усіх фалів директорії «models».

Таким чином ми утворюємо вертикальну структуру, яка допомагає зменшити час обробки даних при запиті та убезпечує втрату усього масиву даних, якщо один блок структури пошкоджений.

Після обробки ми використовуємо «роут» для передачі або видобутку даних, використовуючи для цього один з «контролерів» (приклад контролера Рис.3.30).

```
'use strict';

const { APIError } = require('../exceptions/APIError');
const actionLogService = require('../services/actionLogService');

Tabnine | Edit | Test | Explain | Document | Ask
async function getActionLogs(req, res) {
  const query = req.query;
  const withAssociates = query.withAssociates === 'true';

  try {
    const actionLogs = await actionLogService.getLogsByCriteria(query, withAssociates);

    res.json({
      count: actionLogs.count,
      data: actionLogs.rows,
    });
  } catch (error) {
    throw APIError.ServerError(error.message);
  }
}

module.exports = {
  getActionLogs,
};
```

Рис. 3.30 Приклад «контролера» для спостереження дій користувача у додатку.

На Рис.3.31 зображено приклад «роуту» для зв’язку бази даних та серверу з різними методами маніпуляції даних.

```
'use strict';

const { APIError } = require('../exceptions/APIError');
const actionLogService = require('../services/actionLogService');

Tabnine | Edit | Test | Explain | Document | Ask
async function getActionLogs(req, res) {
  const query = req.query;
  const withAssociates = query.withAssociates === 'true';

  try {
    const actionLogs = await actionLogService.getLogsByCriteria(query, withAssociates);

    res.json({
      count: actionLogs.count,
      data: actionLogs.rows,
    });
  } catch (error) {
    throw APIError.ServerError(error.message);
  }
}

module.exports = {
  getActionLogs,
};
```

Рис. 3.31 Приклад «роуту» для роботи з користувачем.

Вертикальна структура формується налаштуваннями бази даних, щоб при досяганні критичного значення стовпчика (~500 об’єктів), наступні дані записуються в інший стовпчик і так далі по мірі заповнення бази. Тим самим при появі великих об’ємів збережених значень ми прискоримо обробку запитів, по ключовим значенням, одразу переходячи в стовпчик, в діапазоні якого знаходиться об’єкт, який запитується.

Отже, оптимізація структури даних відіграє одну з ключових ролей при роботі з серверною частино веб додатку, значно прискорюючи обробку запитів, захищаючи цілісність та безпечність даних, та покращуючи процес обробки отриманих значень шляхом стандартизації та типування даних.

3.7 Оптимізація обробки запитів шляхом створення вбудованих API

Для якісної обробки та розділення запитів по типам та функція створюються API. Вони необхідні для полегшення розробки інтерфейсу користувача. У структурі проекту створюється окрема директорія «services», в якій кожна окрема API має свій файл (приклад на Рис.3.32).

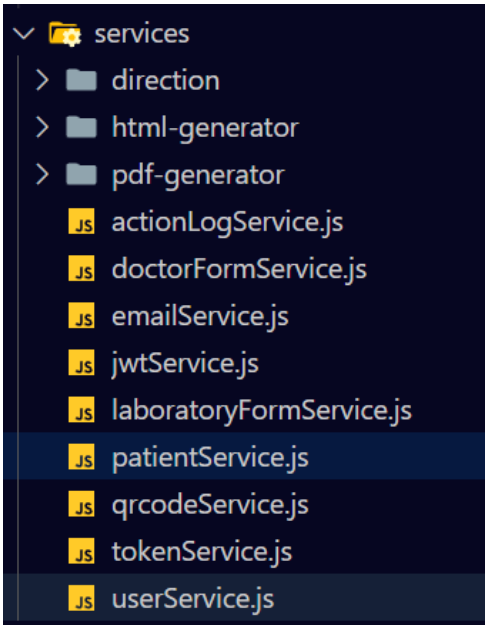


Рис. 3.32 Структура файлів API у проекті веб-додатку.

Приклад API створеного у проекті представлено на Рис.3.33. Як ми бачимо API не лише створює унікально логіку для кожної дії, яка запускається з інтерфейсу користувача, але й відслідковує помилки які можуть з’явитись на стороні сервісу. Кожна API підв’язана з обробниками подій, структуруванням даних та передачею їх до бази даних.

```
'use strict';
const logger = require('../../utils/logger').DirectionLogger;
const { APIError } = require('../../exceptions/APIError');
const { Department } = require('../../models');

Tabnine | Edit | Test | Explain | Document | Ask
async function create({ name }) {
  logger.debug('Attempting to create department', { name });

  const departments = await getAll();

  const isExistName = departments.find(department => department.name === name);

  if (isExistName) {
    logger.warn(`Department with name: ${ name } already exists`);
    throw APIError.BadRequest(`Departments with name: ${ name } already exist`);
  }

  try {
    const department = await Department.create({ name });
    logger.debug(`Department created successfully: ${ department.id }`);
    return department;
  } catch (error) {
    logger.error('Error creating department', { error: error.message });
    throw APIError.ServerError(error.message);
  }
}

Tabnine | Edit | Test | Explain | Document | Ask
async function getAll() {
  try {
    logger.debug('Fetching all departments');
    return await Department.findAll();
  } catch (error) {
    logger.error('Error fetching all departments', { error: error.message });
    throw APIError.ServerError(error.message);
  }
}

module.exports = {
  create,
  getAll,
};
```

Рис. 3.33 Приклад структури API для обробки динамічних полів форми.

Для коректної перевірки який тип даних приймає API та що віддає у відповідь на сторону користувацького інтерфейсу ми використовуємо сервіс POSTMAN. На якому підв'язуємо наш проект, та по мірі додавання нових API оновлюємо та зберігаємо зміни. На Рис.3.34. демонструється приклад перевірки API по входу користувача до веб-додатку.

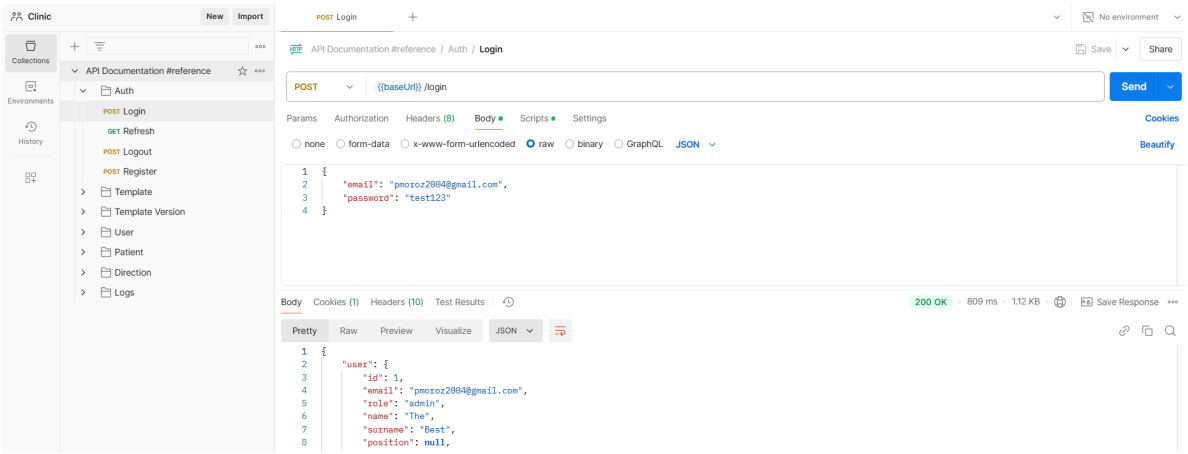


Рис. 3.34 Приклад перевірки роботи створених API.

На Рис.3.34 зліва у стовпчику розташована список усіх створених API та структурована відповідно до того як і в проекті веб-додатку. По центру ми бачимо два діалогових вікна:

1. У верхньому тип даних, які приймаються під час запиту (назва та тип);
2. У нижньому ми бачимо як виглядають дані отримані у відповідь з серверної частини веб-додатку;

У верхній частині ми можемо вибрати метод, який буде використовуватись під час створення запиту (POST, GET, DELETE та ін). Після чого ми вибираємо базову адресну строку, яка зав'язана з серверною частиною та друга частина, це «роут» який відповідає за передачу, отримання та модифікацію даних, які очікуються під час запиту. Після внесення даних, які очікуються на відправку до серверної частини, ми натискаємо кнопку «Send» та переглядаємо статус запиту та дані які отримуємо.

Це полегшує процес розробки, адже не обов'язково запускати проект та зв'язувати його з інтерфейсом користувача, а достатньо перевірити роботу API

прописати усі функції на стороні інтерфейсу користувача, а потім зв’язувати базу даних, серверну частину та інтерфейс користувача.

Для перехвату помилок, їх обробку та повернення до інтерфейсу користувача, створюється окремий файл Рис.3.35, в якому в залежності від коду помилки, що отримується від API, в якості відповіді запиту отримуємо статус код та повідомлення про перебіг події, яка відбувалась. Це покращую комунікативний зв’язок між серверною частиною та інтерфейсом користувача.

```
'use strict';

const { ErrorMessages } = require('../constants');

class APIError extends Error {
  constructor(status, message) {
    super(message);

    this.status = status;
    this.message = message;
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  static NotFound(message = ErrorMessages.NOT_FOUND) {
    return new APIError(404, message);
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  static BadRequest(message = ErrorMessages.BAD_REQUEST) {
    return new APIError(400, message);
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  static Unauthorized(message = ErrorMessages.UNAUTHORIZED) {
    return new APIError(401, message);
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  static Forbidden(message = ErrorMessages.FORBIDDEN) {
    return new APIError(403, message);
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  static Conflict(message = ErrorMessages.CONFLICT) {
    return new APIError(409, message);
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  static ServerError(message = ErrorMessages.SERVER_ERROR) {
    return new APIError(500, message);
  }
  module.exports = { APIError };
}
```

Рис. 3.35 Структура обробки помилок отриманих від функції API

Також, для відображення зрозумілого тексту помилки створюється файл з сталими значеннями, в яких кодові слова на англійській мові перетворює на українську та відображає користувачу Додаток Г. Також файл містить в собі усі перемінні, що отримуються від форм користувача та перетворює на звичні формати для бази даних.

Отже, створення API значно оптимізує роботу під час розробки веб-додатку. А також допомагає налаштувати комунікацію між сервером та користувачем, створюючи унікальне середовище взаємодії без залучення сторонніх веб-сервісів.

3.8 Результати оптимізації за допомогою оптимізації навантаження інтерфейсу користувача

Для створення сучасних веб-додатків, частина функцій, які очікує отримати користувач важко реалізувати за допомогою стандартних функцій вбудованих у звичайній JavaScript або фреймворк (React.js, Vue.js та ін). Це вимагає додаткового ресурсу зі сторони користувацького інтерфейсу, що створює додаткове навантаження та може знизити надійність веб-додатку.

Щоб зменшити вплив сторонніх сервісів, та запобігти появленню неочікуваної поведінки або закриття сервісу і втрату частини веб-додатку вже після стадії презентації, потрібно розробляти власні сервіси, валідувати їх, тим самим покращуючи безпеку.

В нашому веб-додатку, присутні дві функції які потрібно розробити сами:

1. Генерація pdf-файлу з висновком лікаря або направленням на дослідження;

2. Генерація QR-code для контролю перебігу дослідження та навігації між сторінками;

Для генерації pdf-файлу створена окрема директорія Рис.3.40., де знаходяться шаблони які генеруються. В кожному шаблоні зібрано формат даних, прості стилі та динамічні компоненти.

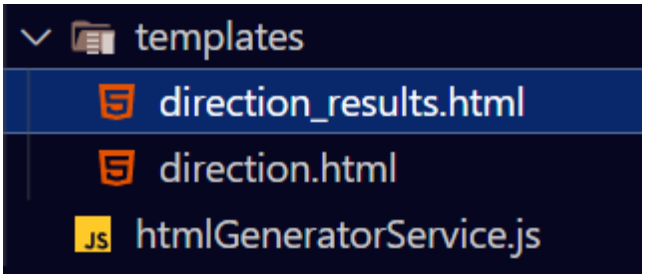


Рис. 3.36 Шаблони pdf-файлів.

За отримання та відображення даних відповідає мікросервіс htmlGeneratorService.js структура якого зображена на Рис.3.37.

```
'use strict';

const fs = require('fs').promises;
const path = require('path');
const Handlebars = require('../../exceptions/handlebars');
const { APIError } = require('../../exceptions/APIError');
const logger = require('../../utils/logger').HTMLLogger;
const qrcodeService = require('../qrcodeService');
const { prepareDirectionDataToPDF } = require('../../utils/pdf');
const directionService = require('../direction/directionService');
const { prepareResult } = require('../../utils/pdf');

class HtmlGeneratorService {
  Tabnine | Edit | Test | Explain | Document | Ask
  async generateDirectionHtml(direction) {
    try {
      logger.debug(`Starting HTML generation for direction: ${ direction.id }`);

      const templatePath = path.join(__dirname, 'templates/direction.html');
      logger.debug(`Reading template from path: ${ templatePath }`);

      const templateContent = await fs.readFile(templatePath, 'utf-8');
      logger.info(`Template content successfully read for direction ID: ${ direction.id }`);

      logger.debug(`Compiling Handlebars template for direction ID: ${ direction.id }`);
      const template = Handlebars.compile(templateContent);

      if (!template) {
        throw APIError.NotFound(`Template Content not found`);
      }

      const preparedDirection = prepareDirectionDataToPDF(direction);

      preparedDirection.qrCodeUrl = await qrcodeService.generateQRCodeDirectionWithPatient(direction.id);
      logger.info(`Generated QR code URL for direction ID: ${ direction.id }`);

      const html = template(preparedDirection);
      logger.debug(`HTML generated for direction ID: ${ direction.id }`);

      return html;
    } catch (error) {
      logger.error(`Error generating HTML for direction ID: ${ direction.id }, error: ${ error.message }`);
      throw error;
    }
  }

  Tabnine | Edit | Test | Explain | Document | Ask
  async generateResultHTML(directionId) {
    try {
      logger.debug(`Starting result HTML generation for direction ID: ${ directionId }`);
      const direction = await directionService.getById(directionId, true);

      if (!direction) {
        throw APIError.BadRequest(`Direction with id: ${ directionId } not found`);
      }

      const templatePath = path.join(__dirname, 'templates/direction_results.html');
      logger.debug(`Template path for research type '${ direction.research }': ${ templatePath }`);

      if (!templatePath) {
        throw APIError.ServerError(`Template for ${ direction.research } not found`);
      }

      const preparedResultData = prepareResult(direction);

      preparedResultData.qrCodeUrl = await qrcodeService.generateQRCodeDirectionWithPatient(direction.id);
      logger.info(`Generated QR code URL for direction ID: ${ direction.id }`);

      const templateContent = await fs.readFile(templatePath, 'utf-8');
      logger.debug(`Template content successfully read for research type: ${ direction.research }`);

      const template = Handlebars.compile(templateContent);
      const html = template(preparedResultData);
      logger.debug(`HTML generated for result of direction ID: ${ directionId }`);

      return html;
    } catch (error) {
      logger.error(`Error generating result HTML for direction ID: ${ directionId }, error: ${ error.message }`);
      throw error;
    }
  }
}

module.exports = new HtmlGeneratorService();
```

Рис. 3.37 Стуктура формування pdf-файлу на серверній стороні веб-додатку

Після формування та запиту ми повинні відправити файл на друк та зберегти версію у базі даних. Тим самим ми робимо лише один запит на отримання, а з серверної частини автоматично відправляєм на збереження у базі даних. Для цього потрібно спочатку сформувати елемент, який буде сховано від користувача і розмітки Рис.3.38.

```
<iframe ref={iframeRef} style={{display: 'none'}}/>
```

Рис. 3.38 Елемент для генерації pdf-файлу

Після чого ми створюємо функцію яка поміщає в середину файл отриманий зі сторони серверу і одразу відправляє його на друк Рис.3.39.

```
const prinResult = async(id) => {
    await doctorService.getDoctorPdf(id).then(res=>{
        if (res?.data){
            const iframe = iframeRef.current;
            console.log(iframe)
            const iframeDoc = iframe.contentDocument || iframe.contentWindow.document;

            iframeDoc.open();
            iframeDoc.write(res?.data);
            iframeDoc.close();
            console.log('printed')

            iframe.onload = () => {
                iframe.contentWindow.print();
                setPrint(true)
            };
        }
    })
```

Рис. 3.39. Функція друку pdf-файлу.

Другий елемент проекту який можна перенести на серверну частину це генерація QR-code. Цей код розташований на направленні яке формується при реєстрації пацієнта та додається на сторінку pdf-файлу. Для формування коду було створено мікросервіс, який створює та передає до шаблону QR-code,в якому знаходиться інформація для відслідковування перебігу дослідження. Після

генерації код не зберігається у базі даних і стирається з серверної частини рисунок.3.40.

```
'use strict';

const QRCode = require('qrcode');
const { APIError } = require('../exceptions/APIError');
const directionService = require('./direction/directionService')
const logger = require('../utils/logger').QRCodeLogger;

Tabnine | Edit | Test | Explain | Document | Ask
async function generateQRCode(data, options = {}) {
  logger.debug(`Generating QR code for data: ${ JSON.stringify(data) }`);

  try {
    const jsonData = JSON.stringify(data);

    const qrCodeDataURL = await QRCode.toDataURL(jsonData, options);

    logger.info('QR code generated successfully');
    return qrCodeDataURL;
  } catch (err) {
    logger.error(`Error generating QR code: ${ err.message }`);
    throw APIError.ServerError(`Error generating QR code: ${ err.message }`);
  }
}

Tabnine | Edit | Test | Explain | Document | Ask
async function generateQRCodeFile(data, filePath, options = {}) {
  logger.debug(`Generating QR code file for data: ${ JSON.stringify(data) }`);

  try {
    const jsonData = JSON.stringify(data);

    await QRCode.toFile(filePath, jsonData, options);

    logger.info(`QR code file generated successfully at ${ filePath }`);
  } catch (err) {
    logger.error(`Error generating QR code file: ${ err.message }`);
    throw APIError.ServerError(`Error generating QR code file: ${ err.message }`);
  }
}

Tabnine | Edit | Test | Explain | Document | Ask
async function generateQRCodeDirectionWithPatient(directionId, options = {}) {
  if (!directionId) {
    throw APIError.BadRequest(`DirectionId must be provided`)
  }

  logger.debug(`Generating QR code for directionId: ${ directionId }`);

  const direction = await directionService.getById(directionId, true)

  if (!direction) {
    throw APIError.BadRequest(`Not found direction with id: ${ directionId }`)
  }

  const { patient } = direction

  const data = {
    directionId,
    patient,
  };

  return generateQRCode(data, options);
}

module.exports = {
  generateQRCodeDirectionWithPatient,
  generateQRCode,
  generateQRCodeFile
};
```

Рис.3.40. Мікросервіс створення QR-коду.

Отже, після створення власних мікросервісів ми убезпечили веб-додаток від непердбачуваних збоїв, розвантажили користувацький інтерфейс та зберегли стабільну продуктивну роботу веб-додатку.

ВИСНОВКИ

Отже, в результаті проведення оптимізації веб-додатку з збереження та обробки лабораторних досліджень можна зробити наступні висновки.

Використання бібліотеки Million.js допомагає оптимізувати будь-який веб-додаток, ще на стадії розробки інтерфейсу користувача, адже він одразу оцінює час обробки кожної функції та елементу інтерфейсу користувача, надає висновок щодо оцінки та розробник може вчасно виявити слабкі місця та усунити їх або зробити дефрагментацію коду, на більш дрібні функції які будуть оброблювати конкретно поставлені типи даних, тим самим зменшуюючи навантаження на систему.

Також, бібліотека React Redux Toolkit, оптимізує взаємодію елементів інтерфейсу користувача між собою. Використовуючи дану бібліотеки ми змогли уникнути контрольованих та неконтрольованих зв’язувань між дочірніми та батьківськими елементами. Це допомагає уникнути виникнення фатальних помилок при передачі параметрів та полегшує утворення розгалуженої системи зв’язків. Також використовуючи React Redux ми можемо не використовувати local storage на сторінці, що також підвищує продуктивність. Ще одною перевагою даного методу оптимізації являється можливість зберігання даних для рендерингу форм, що робить генерацію будь яких динамічних елементів універсальною та зменшує кількість написаного коду.

Інший метод який ми використали це бібліотека React Router DOM. За допомогою якої ми створюємо розгалужену систему маршрутизації, при цьому при переході між сторінками нам не потрібно оновлювати та завантажувати сторінки заново. Це оптимізує роботу додатку, особливо при появі повільного підключення до інтернету. Також ми можемо оптимізувати доступ та валідувати його відповідно до ролей користувачів, що убезпечує наш веб-додаток від несанкціонованих доступів та витоку інформації, що є важливим у випадку роботи з результатами лабораторних досліджень та таємницею пацієнта-лікаря.

Ще одним методом оптимізації було обрано використання бібліотеки Axios, що створює уніфіковані функції HTTP запитів, валідує отриманні дані та

дозволяє використовувати однакові типи запитів у різних елементах проекту. Це дозволяє зменшити кількість коду у проекті, відслідковувати помилки, які можуть з’явитись під час обробки, відправки та передачі даних, що значно підвищує надійність веб-додатку. Таким чином можна зробити висновок, що ця бібліотека доцільна до використання під час розробки методів оптимізації веб-застосунків.

Для оптимізації відтворення та компілювання елементів форми, що є ключовими у нашому веб-додатку, було обрано бібліотеку useForm, яка не лише робить елемент форми універсальним, незалежно від місця компіляції, але й допомагає зменшити кількість функцій для обробки отриманих даних введених користувачем та відслідковувати дії користувача з будь-яким полем форми, що допомагає також зменшити кількість написаного коду, що допомагає оптимізувати час відтворення веб-додатку на сторінці браузеру.

Для оптимізації серверної частини створеного проекту ми обрали метод створення структурованої бази даних (горизонтальна та вертикальна структури), що підвищує надійність збережених даних та пришвидшує час обробки даних під час маніпуляції в серверній частині. Другий обраний підхід заключався у створенні кастомних API, які допоможуть оптимізувати розробку під час налаштування запитів зі сторони інтерфейсу користувача та розділити усі запити за типом даних, які плануються до зміни. Використання кастомних API більш ефективне ніж використання сторонніх сервісів, адже ми виключаємо фактори ризику такі як: блокування ресурсу, втрата ключа доступу, закриття сервісу та ін.. Це все підвищує надійність веб-додатку та оптимізує час запиту, адже сервера сторонніх ресурсів можуть бути перевантаженні у пікові моменти та сповільнювати час обробки даних.

Для оптимізації інтерфейсу користувача ми намагались зменшити до мінімуму використання сторонніх сервісів для створення елементів, які неможливо створити за допомогою звичайних інструментів фреймворку. Таким чином ми створили власні сервіси на серверній частині веб-додатку, що підвищує надійність та зменшує час обробки та створення елементів, зменшує кількість запитів між інтерфейсом користувача та серверної частини. Адже

власноруч створенні елементи можна одразу зберігати за необхідності без повторних відправок з інтерфейсу користувача, що зменшує кількість написаного коду, що в свою чергу зменшує час компіляції проекту у браузері.

Отже, головним чином основні фактори для оптимізації це:

1. Час обробки даних серверною частиною;
2. Час компіляції веб застосунку;
3. Час обробки даних функціями фреймворків;
4. Кількість символів написаних розробниками, що впливають на розмір кінцевого файлу;
5. Час взаємодії між інтерфейсом користувача та серверної частини;

Загалом, використовуючи сучасні методи та підходи до розробки ми можемо зробити стабільний та конкуретно-спроможний проект. Який зможе працювати однаково продуктивно при різних швидкостях підключення до інтернету, що є дуже важливим аспектом при роботі з веб-додатками. Також під час оптимізації ми значно підвищили цілісність та безпечність даних що зберігаються в хмароному сховищі, зробивши максимально ізольовану систему з мінімальною кількістю сторонніх сервісів, що створюють елементи проекту.

У наступних тезах доповіді на конференціях:

Стаття:

1. Виговський Б.С., Трінтіна Н.А. Методика оптимізації web-запитів і бази даних для підвищення продуктивності web-застосунку для лабораторних досліджень // Зв’язок. 2024 (Подана до друку)

Тези публікації:

1. Виговський Б.С., Трінтіна Н.А. Фрагментація бази даних для рівномірного навантаження структури сайту // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» - Київ: ДУІКТ, 2023. - С. 140-142
2. Виговський Б.С., Трінтіна Н.А. Прискорення взаємодії React.JS з DOM-елементами за допомогою бібліотеки Million.js // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» - Київ: ДУІКТ, 2023. - С. 142-144

ПЕРЕЛІК ПОСИЛАНЬ

1. Yaghini M, Akhavan R. DIMMA: A Design and Implementation Methodology for Metaheuristic Algorithms - A Perspective from Software Development. International Journal of Applied Metaheuristic Computing (IJAMC). 2010;1:57-74
2. Adali EA, Işik AT, Kundakci N. Todim Method For The Selection Of The Elective Courses. European Scientific Journa. 2016;12(10):314-324.
3. . Ben-Gal I, Kagan E. Information Theory: Deep Ideas, Wide Perspectives, and Various Applications. Entropy. 2021;23(2):232.
4. Zhou F, Chen TY. A hybrid approach combining AHP with TODIM for blockchain technology provider selection under the Pythagorean fuzzy scenario. Artificial Intelligence Review. 2022;55(7):5411-43.
5. L. Liu, S. Zhou, H. Huang, and Z. Zheng, “From technology to society: an overview of blockchain based DAO,” IEEE Open Journal of the Computer Society (Volume: 2) 2021; 204-215
6. H. L. Gururaj, A. A. Manoj, A. A. Kumar, A. M. Holla, S. M. Nagarajath, and V. Ravi Kumar, “Blockchain: A new era of technology,” Cryptocurrencies and Blockchain Technologies and Applications; 2020; (pp.1-24)
7. J. Zarrin, H. W. Phang, L. B. Saheer, and B. Zarrin, “Blockchain for decentralization of internet: prospects, trends and challenges,” Cluster Computing 24(1) 2021
8. K. Nabben, “Decentralised Autonomous Organisations (DAOs) as data trusts: a general-purpose data governance framework for decentralised data ownership, storage and utilisation" SSRN Electronic Journal, 2021
9. Singh, A. P, Ghosh, S.Sahay, S. K., Kumar, C., Modi, A., & Mondal, S. (2021). Application of synchrophasor angular difference as a grid monitoring tool and for assessment of Real time voltage Stability-Case Study. 2021 9th IEEE International Conference on Power Systems (ICPS)
10. Gackenheimer, C. (2015). Introducing Flux: An Application Architecture for React. Introduction to React. Apress

11. Ramos, M., Valente, M.T. and Terra, R., 2018. AngularJS: A survey study. IEEE Software, 35(2)pp.72-79

12. Crawford, T.; Hussain, T. A comparison of server side scripting technologies. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2017), Las Vegas, NV, USA, 17–20 July 2017; pp. 69–76.

13. Piastou, M. (2024). Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue. js. World Journal of Advanced Engineering Technology and Sciences, 9(2), 366-376.

14. Nordén, M. To What Extent Does Ruby on Rails Effect Performance in Applications. Degree project in Computer Science. Master's Thesis, Linköping University, Linköping, Sweden, 2015.

15. Persson, K. Optimizing Ruby on Rails for Performance and Scalability. Degree Project in Computer Science. Master's Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2016.

16. Yang, J.; Yan, C.; Wan, C.; Lu, S.; Cheung, A. View-Centric Performance Optimization for Database-Backed Web Applications. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 994–1004.

17. Piastou, M. (2024). Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue. js. World Journal of Advanced Engineering Technology and Sciences, 9(2), 366-376.

18. Guha, A., Saftoiu, C., & Krishnamurthi, S. (2015, October 3). The Essence of JavaScript. Brown University. Retrieved June 17, 2021, from https://arxiv.org/pdf/1510.00925.pdf

19. Holland, B. (2016, January 11). What To Expect From JavaScript In 2016-Frameworks [What To Expect From JavaScript In 2016 - Frameworks]. Telerik. Retrieved October 18, 2020, from https://www.telerik.com/blogs/what-to-expect-from-javascript-in-2016-frameworks

20. Muyldermans, D. (2019, May 10). How does the virtual DOM compare to other DOM update mechanisms in JavaScript frameworks? University of Dublin. http://www.daisyms.com/THESIS.pdf

21. Eisenman, B. (2015). Learning React Native: Building Native Mobile Apps with JavaScript. Reilly Media, Inc.

22. Sen, R. (n.d.). A Strategic Analysis of Competition Between Open Source and Proprietary Software. Texas A&M University. https://econwpa.ub.uni-muenchen.de/econ-wp/io/removed/0510004.pdf

23. George, B., & Williams, L.(2004). A structured experiment of test-driven development. Penn State University. Retrieved June 17,2021, from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.147&rep=rep1&type=pdf

24. Peterson, M.(2020, May 28).JavaScript DOM Manipulation Performance. Blekinge Institute of Technology. Retrieved June 17 ,2021, from https://www.diva-portal.org/smash/get/diva2:1436661/FULLTEXT01.pdf

25. Reddy, V. M. (2021). Blockchain Technology in E-commerce: A New Paradigm for Data Integrity and Security. Revista Espanola de Documentacion Cientifica, 15(4), 88-107. Maddireddy, Bharat Reddy, and Bhargava Reddy Maddireddy. "Proactive Cyber Defense: Utilizing AI for Early Threat Detection and Risk Assessment." International Journal of Advanced Engineering Technologies and Innovations 1.2 (2020): 64-83.

26. Javeed, Danish, Muhammad Shahid Saeed, Prabhat Kumar, Alireza Jolfaei, Shareeful Islam, and AKM Najmul Islam. "Federated learning-based personalized recommendation systems: An overview on security and privacy challenges." IEEE Transactions on Consumer Electronics (2023).

27. Kashyap, Varun, and Narayanan Subramanian. "Secured Authentication System for Product Verification: Integrating Blockchain and Secure Data Management." In 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT), pp. 1-6. IEEE, 2023.

28. Hawashin, Diana, Mohamed Nemer, Senay A. Gebreab, Khaled Salah, Raja Jayaraman, Muhammad Khurram Khan, and Ernesto Damiani. "Blockchain applications in UAV industry: Review, opportunities, and challenges." Journal of Network and Computer Applications 230 (2024): 103932.

29. Ferlito, Sergio, Salvatore Ippolito, Celestino Santagata, Paolo Schiattarella, and Girolamo Di Francia. "A Study on an IoT-Based SCADA System for Photovoltaic Utility Plants." Electronics 13, no. 11 (2024): 2065.

30. Reddy, V. M. (2024). The Role of NoSQL Databases in Scaling E-commerce Platforms. International Journal of Advanced Engineering Technologies and Innovations, 1(3), 262-296

31. Maddireddy, Bhargava Reddy, and Bharat Reddy Maddireddy. "Evolutionary Algorithms in AI-Driven Cybersecurity Solutions for Adaptive Threat Mitigation." International Journal of Advanced Engineering Technologies and Innovations 1.2 (2021): 17-43.

32. Yanamala, Anil Kumar Yadav. "Secure and Private AI: Implementing Advanced Data Protection Techniques in Machine Learning Models." International Journal of Machine Learning Research in Cybersecurity and Artificial Intelligence 14.1 (2023): 105-132.

33. Maddireddy, Bharat Reddy, and Bhargava Reddy Maddireddy. "Cybersecurity Threat Landscape: Predictive Modelling Using Advanced AI Algorithms." International Journal of Advanced Engineering Technologies and Innovations 1.2 (2022): 270-285.

34. Maddireddy, Bhargava Reddy, and Bharat Reddy Maddireddy. "Real-Time Data Analytics with AI: Improving Security Event Monitoring and Management." Unique Endeavor in Business & Social Sciences 1.2 (2022): 47-62.

35. Shen, Meng, Liehuang Zhu, and Ke Xu. Blockchain: empowering secure data sharing. Berlin/Heidelberg, Germany: Springer, 2020.

36. Rabbi, ASM Fazle, TM Ragib Shahrier, Md Mushfiqur Rahman Miraz, Sazia Rahman, and Dip Nandi. "Beyond the Hype: A Proposed Model Based on Critical Analysis of Blockchain Technology's Potential to Address Supply Chain Issues."

37. Iftikhar, Abeer, Kashif Naseer Qureshi, Muhammad Shiraz, and Saleh Albahli. "Security, trust and privacy risks, responses, and solutions for high-speed smart cities networks: A systematic literature review." Journal of King Saud University-Computer and Information Sciences (2023): 101788.

38. Sarwatt, Doreen Sebastian, Yujia Lin, Jianguo Ding, Yunchuan Sun, and Huansheng Ning. "Metaverse for Intelligent Transportation Systems (ITS): A Comprehensive Review of Technologies, Applications, Implications, Challenges and Future Directions." IEEE Transactions on Intelligent Transportation Systems (2024).
39. Yanamala, Anil Kumar Yadav. "Cost-Sensitive Deep Learning for Predicting Hospital Readmission: Enhancing Patient Care and Resource Allocation." International Journal of Advanced Engineering Technologies and Innovations 1.3 (2022): 56-81.
40. S.Tilkov, S.Vinoski, “Node.js: Using JavaScript to Build High-Performance Network Programs”, IEEE Internet Computing, 2010.
41. http://Node.js.org/
42. http://strongloop.com/developers/Node-js-infographic/?utm_source=ourjs.com#3.
43. R.Jain, “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling”, John Wiley & Sons, Inc., New York, NY, 1991.
44. E.Cecchet, A.Chanda, S.Elnikety, J.Marguerite, and W.Zwaenepoel, “Performance Comparison of Middleware Architectures for Generating Dynamic Web Content", Proceedings of 4th Middleware Conference, Rio de Janeiro, Brazil, June 2003.
45. U.Ramana, T.Prabhakar, “Some Experiments with the Performance of LAMP Architecture”, Proceedings of the 2005 Fifth International Conference on Computer and Information Technology, 2005.
46. A.Ranjan, R.Kumar, J.Dhar, “A Comparative Study between Dynamic Web Scripting Languages”, Data Engineering and Management, 2012.
47. T.Scott, T.Michiaki, S.Toyotaro, T.Akihiko, and O.Tamiya,“Performance Comparison of PHP and JSP as Server-Side Scripting Languages”, Middleware, 2008.
48. “MERN Stack - Javatpoint,”www.javatpoint.com. https://www.javatpoint.com/mern-stack (accessed May 18, 2023).

49. A. Kapoor, “Benefits of Using MERN Stack,” Medium, Nov. 19, 2021. https://enlear.academy/ benefits-of-using-mern-stack-7e0c732b5214 (acess- ed May 18, 2023)

50. R. Sheldon and J. Denman, “Node.js (Node),” WhatIs.com, Nov. 2022. https://www.techtarget. com/whatis/definition/Nodejs (accessed May 18, 2023).

51. “What is MongoDB Atlas? — MongoDB Atlas,” www.mongodb.com. https://www.mongodb. com/docs/atlas/ (accessed May 19, 2023).

52. Webandcrafts, “Top Advantages and Dis-advantages of MongoDB NoSQL Database,” Webandcrafts Blog, Oct. 15, 2021. https:// webandcrafts.com/blog/mongodb-advantages-and-disadvantages/ (accessed May 19, 2023)

53. Performance optimization in Microsoft SQL server DBMS. [Electronic resource] Access mode: https://novainfo.ru/article/11469 .

54. How to optimize work with databases: basic methods and recommendations. [Electronic resource] Access mode: https://eurobyte.ru/articles/kak-optimizirovat-rabotu-s-bazami-dannykhosnovnye-metody-i-rekomendacii/

55. Best Practices for Optimizing Database Performance: Tips and Techniques for Developers. [Electronic resource] Access mode: https://www.code-sample.com/2023/06/optimizing-databaseperformance-tips.html .

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

МЕТОДИКА ОПТИМІЗАЦІЇ WEB-ЗАПИТІВ І БАЗ ДАНИХ ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ WEB-ЗАСТОСУНКУ ДЛЯ ЛАБОРАТОРНИХ ДОСЛІДЖЕНЬ

Виконав: студент групи ПДМ-63 Виговський Богдан Сергійович

Керівник: к.т.н., доц., доцент кафедри ІТ Трінтіна Наталя Альбертівна

Київ - 2025

МЕТА, ОБ’ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація веб-додатка для зберігання та обробки результатів лабораторного дослідження.

Об’єкт дослідження: веб-додаток, що використовується для обробки та зберігання лабораторних досліджень.

Предмет дослідження: методи оптимізації веб-додатку, що використовується для обробки та зберігання лабораторних досліджень.

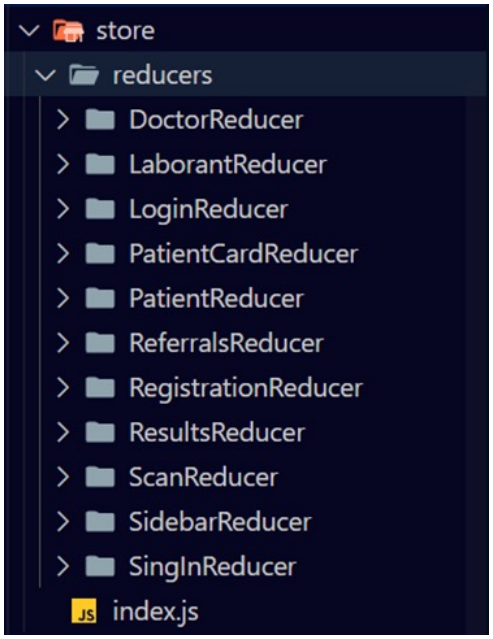
АКТУАЛЬНІСТЬ РОБОТИ

Наукова новизна роботи полягає у використанні блокчейн технологій у підході розробки веб-додатку для зберігання та обробки лабораторних досліджень. Вперше для розробки такого типу додатків запропоновано та реалізовано до використання принцип розробки MERN (акронім «MongoDB», «Express», «React» і «Node JS»), в результаті чого ми отримуємо стабільний та безпечний веб додаток, дані якого захищені від стороннього впливу та втрати особистих даних.

Порівняльний аналіз

	Мій проект	Сінево	Діла
Час завантаження	0,27мс	3сек	2,4сек
Зручність (інтерфейс) для клієнта	+	+	-
Функціональність	+	-	-

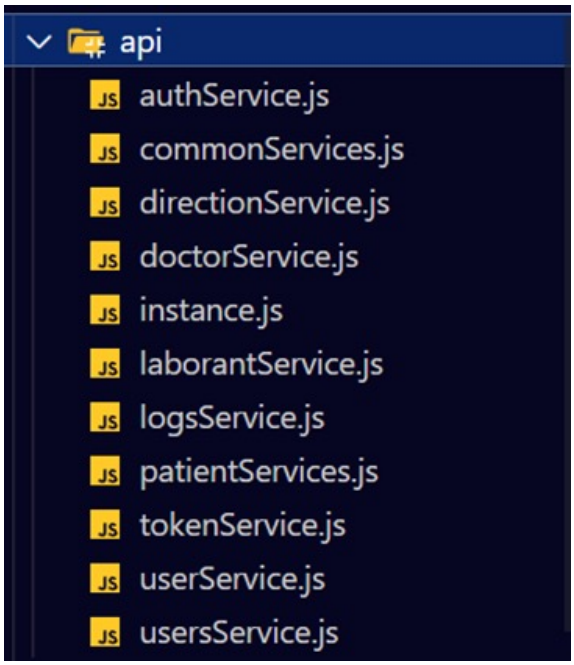
ПРАКТИЧНИЙ РЕЗУЛЬТАТ



Функція у бібліотеці react redux яка записує та передає данні між елементами називається reducer. Кожна окрема функція працює автономно та дозволяє зберігати дані передані до неї локально. Також у цій функції дані можна змінювати та передавати зміни назад.

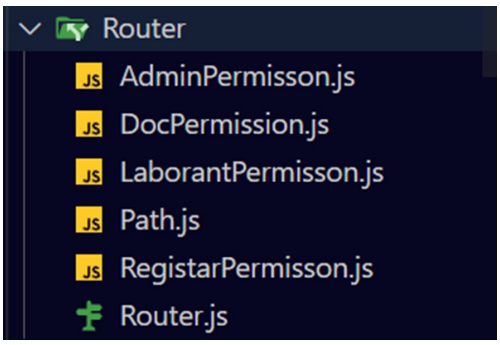
Дана бібліотека значно оптимізує роботу веб-додатку, за рахунок зменшення кількості написаного коду та запобігання утворення «контрольованих» та «не контрольованих» зв’язувань між елементами.

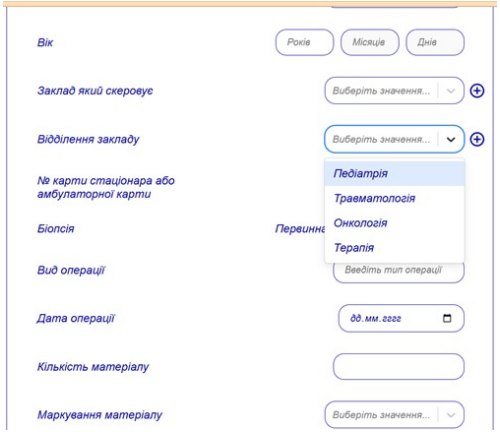
ПРАКТИЧНИЙ РЕЗУЛЬТАТ



Розглянувши основні принципи та методи застосування бібліотеки Axios ми можемо зробити висновок, що дана технологія дуже оптимізує написаний код, полегшує його сприйняття розробниками. Значно спрощує валідацію помилок отриманих від серверної частини і можемо їх легко виводити користувачу, щоб він вчасно отримував інформацію і розумів причини виникнення їх. Також багаторазовість використання функцій запиту у різних елементах з меншою кількістю написаного коду значно прискорює роботу веб-додатку та зменшує навантаженість серверної частини.

ПРАКТИЧНИЙ РЕЗУЛЬТАТ

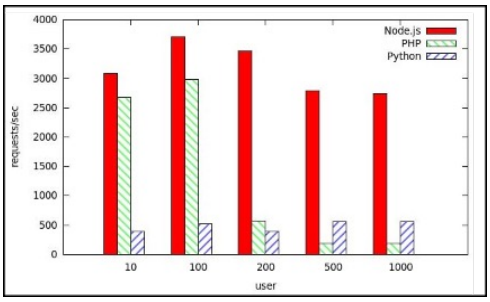




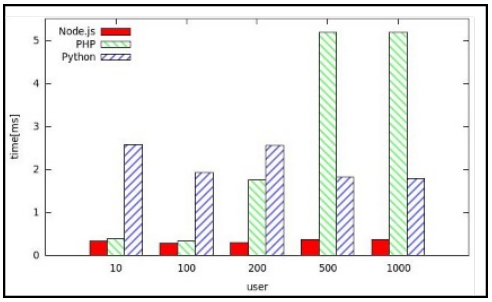
Бібліотека React-Router-Dom значно оптимізує веб-додаток, зменшуючи навантаження та додає додаткову валідацію доступу, запобігаючи несанкціонованого доступу. Навантаження зменшується через відсутність зайвих оновлень сторінки при переході між елементами, утворюючи стійку модель SPA(з англ. «Single Page Application» - односторінковий веб-додаток).

В результаті роботи з бібліотекою useForm ми можемо зробити висновок, що вона значно оптимізує роботу с формами, які є ключовими в нашому веб-додатку, також допомагає валідувати внесення даних та на виході отримує відсортовані дані форматом, який повністю готовий до відправки на сторону серверу, що підвищує надійність додатку.

РЕЗУЛЬТАТИ МОДЕЛЮВАННЯ



Результати «Hello World» кількість запитів на секунду



Результати «Hello World» кількість часу на запит

Математична модель для дослідження **кількості запитів на секунду (RPS - Requests Per Second)**

Основна формула кількості запитів на секунду

$$R = \frac{N}{T}$$

Де:

- R – кількість запитів на секунду
- N – загальна кількість запитів за період
- T – час у секундах, протягом якого виконували запит

Модель із використанням часових вікон (Sliding Window)

Для дослідження навантаження у змінних умовах (пікових періодах) використовується середня кількість запитів за часовий проміжок

$$R_w = \frac{\sum_{i=1}^{w} N_i}{T_w}$$

Де:

- N_i - кількість запитів у і-тому вікні
- W – це кількість вікон
- T_w - тривалість одного вікна

Середній час обробки одного запиту

Де:

- T_i - час обробки і-го запиту
- N- загальна кількість запитів

$$T = \frac{\sum_{i=1}^{N} T_i}{N}$$

Висновки

- **Розроблено:** функціонал веб-додатку із використанням Million.js, що допомогло оптимізувати його продуктивність на стадії розробки та виявити слабкі й повільні місця у користувацькому інтерфейсі.
- **Проаналізовано:** використання React Redux Toolkit, що дозволило уникнути контрольованих і неконтрольованих зв’язків між дочірніми та батьківськими елементами.
- **Створено:** розгалужену систему маршрутизації за допомогою React Router DOM. Це забезпечило безперервний перехід між сторінками без необхідності їх повторного завантаження.
- **Створено:** кастомні API для оптимізації розробки під час налаштування запитів зі сторони інтерфейсу користувача, що дозволило розділити усі запити за типом даних, які плануються до зміни.
- **Впроваджено:** бібліотеку Axios, яка дозволяє створювати уніфіковані функції HTTP-запитів, валідувати отримані дані та використовувати однакові типи запитів у різних елементах проєкту.
- **Реалізовано:** структуровану базу даних з горизонтальною та вертикальною структурами для серверної частини проєкту. Це підвищило надійність збереження даних та пришвидшило їх обробку під час маніпуляцій.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Виговський Б.С., Трінтіна Н.А. Методика оптимізації web-запитів і бази даних для підвищення продуктивності web-застосунку для лабораторних досліджень // Зв'язок. 2024 (Подана до друку)

Тези доповідей:

1. Виговський Б.С., Трінтіна Н.А. Фрагментація бази даних для рівномірного навантаження структури сайту // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» - Київ: ДУІКТ, 2023. - С. 140-142
2. Виговський Б.С., Трінтіна Н.А. Прискорення взаємодії React.JS з DOM-елементами за допомогою бібліотеки Million.js // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» - Київ: ДУІКТ, 2023. - С. 142-144

ДОДАТОК Б. ПРИКЛАД ЗНАЧЕНЬ ДЛЯ СТВОРЕННЯ ФОРМИ ЛАБОРАНТА

```
const values = [
    {
        "formType": 'Gistology',
        "idetifier":[
            "description",
            "cassets",
            "archive",
            "calcium",
            "doc",
            "laborant"
        ],
        "title": 'Гістологія',
        "dataType": 'laborant',
        "fields": [
            {
                "legend": 'Макроскоповий
опис',
                "id":1,
                "type": 'textarea',
                "required": true,
                "name": 'description',
                "options": false,
                "props": [
                    {
                    "label": '',
                    "id":881,
                    "type": 'text',
                    "placeholder":
'Введіть дані...'
                    },
                ]
            },
            {
                "legend": 'Кількість
касет',
                "id":2,
                "type": 'text',
                "required": true,
                "name": 'cassetsAfter',
                "options": false,
                "props": []
            },
            {
                "legend": 'Наявність
архіву',
                "id":3,
                "type": 'radio',
                "required": true,
                "name": 'archive',
                "options": false,
                "props": [
                    {
                    "label": 'Так',
                    "id":661,
                    "type": 'radio'
                    },
                    {
                    "label": 'Ні',
                    "id":662,
                    "type": 'radio'
                    },
                ]
            },
            {
                "legend":
'Декальцінація',
                "id":4,
                "type": 'radio',
                "required": true,
                "name": 'calcium',
                "options": false,
                "props": [
                    {
                    "label": 'Так',
                    "id":661,
                    "type": 'radio'
                    },
                    {
                    "label": 'Ні',
                    "id":662,
                    "type": 'radio'
                    },
                ]
            },
            {
                "legend": 'Лікар',
                "id":5,
                "type": 'text',
                "required": true,
                "name": 'doc',
                "options": true,
                "props": []
            },
            {
                "legend": 'Лаборант',
                "id":5,
                "type": 'text',
                "required": true,
                "name": 'laborant',
                "options": true,
                "props": []
            },
        ],
        "button": "Сформувати",
```

```
        "class": 'form__box__register'
    },{
        "formType": 'Express',
        "idetifier":[
            "description",
            "doc",
            "laborant"
        ],
        "title": 'Експрес',
        "dataType": 'laborant',
        "fields": [
            {
                "legend": 'Макроскоповий
опис',
                "id":1,
                "type": 'textarea',
                "required": true,
                "name": 'description',
                "options": false,
                "props": [
                    {
                    "label": '',
                    "id":881,
                    "type": 'text',
                    "placeholder":
'Введіть дані...'
                    },
                ]
            },
            {
                "legend": 'Лікар',
                "id":5,
                "type": 'text',
                "required": true,
                "name": 'doc',
                "options": true,
                "props": []
            },
            {
                "legend": 'Лаборант',
                "id":5,
                "type": 'text',
                "required": true,
                "name": 'laborant',
                "options": true,
                "props": []
            },
        ],
        "button": "Сформувати",
        "class": 'form__box__register'
    }
]
```

ДОДАТОК В. ПРИКЛАД НАПИСАННЯ ФОРМИ ЗА ДОПОМОГОЮ БІБЛІОТЕКИ USEFORM

```
<form className={classes.form}
onSubmit={handleSubmit(onSubmit)}>
            {values.formType === 'registration' && <div
className={classes.form__registration__iconsBox}>
                <ContentCopyIcon onClick={() => {
                    trigger(copyHandler())
                }} className={classes.icon}/>
                <PrintIcon onClick={() => {
                    trigger(printHandler())
                }} className={classes.icon}/>
            </div>
            }
            {values?.fields?.map((el, index) => (
                <div className={classes[values.class]}
key={index}>
                    <legend className={classes.form__label}
htmlFor={el.type}>{el.legend}</legend>
                    <div
className={classNames(classes.form__label__container,
{[classes.cassets__container]: el.name == 'cassets'})}>
                        {el.options
                            ?
                            <>
                                <Controller
                                    control={control}
                                    name={el.name}
                                    render={({field: {onChange,
value}}) => (
                                        <Select
                                            className={classes.select}
                                            options={el.props}
                                            value={value ? value : []}
                                            onChange={onChange}
                                            disableSearch
                                            hasSelectAll={false}
                                            placeholder='Виберіть
значення...'
                                        />
                                    )}
                                />
                                {el.name !== 'research' && el.name
!== 'markedMaterial' && <AddCircleOutlineIcon
onClick={() => {
                                    trigger(addFieldHandler(el.legend,
el.name))
                                }}
className={classes.icon}></AddCircleOutlineIcon>}
                            </>
                            : el?.props?.map((props, i) => (
                                <div
className={classNames(classes.form__label__container__b
ox)} key={'a' + i}>
                                    <div><label>{props.label}</label></div>
                                    <div>
                                        {el.name === 'clinicalConclusion'
|| el.name === 'clinicalInfo'
                                            ? <textarea

className={classNames(classes.form__input,
{[classes.errorInput]: errors[el.name]})}
                                                type={props.type}

placeholder={props.placeholder}
                                                pattern={props.pattern}
                                                disabled={props.show}
                                                value={el.type === 'radio' ?
props.label : props.value}
                                                {...register(el.name == 'age' ||
el.name == 'cassetsAfter' ? props.register : el.name,
{required: el.required})}
                                            />
                                            : <input

className={classNames(classes.form__input,
{[classes.errorInput]: errors[el.name]})}
                                                type={props.type}
                                                min={props.min}
                                                max={props.max}

placeholder={props.placeholder}
                                                pattern={props.pattern}
                                                disabled={props.show}
                                                value={el.type === 'radio' ?
props.label : props.value}
                                                {...register(el.name == 'age' ||
el.name == 'cassetsAfter' ? props.register : el.name,
{required: el.required})}
                                            />}
                                    </div>
                                </div>
                            ))
                        }
                    </div>
                </div>
            ))}
            <button className={classes.form__button}
type='submit'>
                {values.button}
            </button>
        </form>
```

ДОДАТОК Г. ПРИКЛАД ОБРОБКИ ДАНИХ ОТРИМАНИХ ВІД КОРИСТУВАЧА

```
const { DataTypes } = require('sequelize');
const sequelize = require('../config/db');
const { Genders } = require('../constants');
const Patient = sequelize.define(
  'Patient',
  {
    name  : {
      type     : DataTypes.STRING,
      allowNull: false,
      validate : {
        notEmpty: { msg: 'Patient name is required' },
      },
    },
    lastName  : {
      type     : DataTypes.STRING,
      allowNull: false,
      validate : {
        notEmpty: { msg: 'Patient lastName is required' },
      },
    },
    middleName  : {
      type     : DataTypes.STRING,
      allowNull: true,
    },
    gender      : {
      type     : DataTypes.ENUM(Genders.MALE,
Genders.FEMALE),
      allowNull: false,
      validate: {
        notEmpty: { msg: 'Gender is required' },
      },
    },
    cardNumber   : {
      type     : DataTypes.STRING,
      allowNull: false,
      validate : {
        notEmpty: { msg: 'Card number is required' },
      },
    },
    patientNumber: {
      type     : DataTypes.STRING,
      allowNull: false,
      validate : {
        notEmpty: { msg: 'Patient number is required' },
      },
    },
    phoneNumber  : {
      type     : DataTypes.STRING,
      allowNull: true,
      validate : {
        is: /^\+380\d{9}$/,
      },
    },
    outside: {
      type: DataTypes.BOOLEAN,
      allowNull: false,
      defaultValue: true,
    }
  },
  {}
);
```

ДОДАТОК Д. ФАЙЛ ЗІ СТАЛИМИ КОДОВИМИ СЛОВАМИ

```
const actionLogs = require('./actionsLog');

const Roles = {
  ADMIN             : 'admin',
  DOCTOR            : 'doctor',
  REGISTRAR         : 'registrar',
  LABORATORY_ASSISTANT: 'laboratoryAssistant',
}

const DirectionStatuses = {
  REGISTERED     : 'registered',
  SCAN_AFTER_FORM : 'scanAfterForm',
  SCAN_BEFORE_FORM: 'scanBeforePaint',
  SCAN_AFTER_PAINT: 'scanAfterPaint',
  TO_DOCTOR      : 'toDoctor',
  FINISHED       : 'finished'
}

const DirectionStatusesList =
Object.values(DirectionStatuses)

const BiopsyTypes = {
  PRIMARY  : 'Первинна',
  SECONDARY: 'Вторинна'
}

const Genders = {
  MALE  : 'male',
  FEMALE: 'female'
}

const MarkedMaterials = {
  OPERATIONAL: 'Операційний матеріал',
  BIOPSY     : 'Біопсія',
};

const ResearchOptions = {
  HISTOLOGY: 'Гістологія',
  IGCH     : 'ІГХ',
  CISH     : 'CISH',
  EXPRESS  : 'Експрес',
};

const ResearchOptionsList = Object.values(ResearchOptions)

const Aliases = {
  Direction    : {
    DEPARTMENT: 'department',
    HOSPITAL  : 'hospital',
    PATIENT   : 'patient',
    LAB_FORM  : 'laboratoryForm',
    DOC_FORM  : 'doctorForm',
    DOCTOR    : 'doctor',
    ANTIBODIES: 'antibodies'
  },
  User         : {
    LAB_FORMS    : 'laboratoryForms',
    LAB_FORMS_DOC: 'laboratoryDoctorForms',
    DOC_FORMS    : 'doctorForms',
    DIRECTIONS   : 'directions'
  },
  Patient      : {
    DIRECTIONS: 'directions',
  },
  LaboratoryFrom: {
    DOCTOR             : 'doctor',
    LABORATORY_ASSISTANT: 'laboratoryAssistant'
  },
  DoctorFrom   : {
    DOCTOR: 'doctor',
  },
  ActionLog    : {
    USER       : 'user',
    TARGET_USER: 'targetUser',
    PATIENT    : 'patient',
    DIRECTION  : 'direction',
  }
};
```