ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота

на тему: «**СИСТЕМА ДЛЯ МЕДИЧНОЇ ЛАБОРАТОРІЇ НА ОСНОВІ REACT.JS»**

на здобуття освітнього ступеня бакалавра
зі спеціальності __122 Комп'ютерні науки_____
(код, найменування спеціальності)
освітньо-професійної програми __Комп'ютерні науки _____
(назва)

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

_____ ___Ілля ЄРШОВ_____
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. __КНД-43_____

_____Ілля ЄРШОВ_____
(Ім'я, ПРІЗВИЩЕ)

Керівник: __доктор філософії, Юлія БЕРЕЗОВСЬКА
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Київ – 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально–науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук
Ступінь вищої освіти − «Бакалавр»
Спеціальність підготовки − «122 Комп'ютерні науки»
Освітньо-професійна програма − «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____Віктор Вишнівський
"____" _____2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
Єршов Ілля Володимирович
(прізвище, ім'я, по батькові)

1.Тема кваліфікаційної роботи: «Система для медичної лабораторії на основі React.js»
Керівник кваліфікаційної роботи Юлія БЕРЕЗОВСЬКА, доктор філософії. _,
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу №56 від «24» лютого 2025 р.

2. Строк подання студентом роботи 30.05.2025 р.
3. Вихідні дані до роботи: Науково-технічна література з питань, пов'язаних з темою роботи, Дані про технологій розробки лабораторних систем, Веб-додатки типу Single Page Application.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Вибір доцільних технологій для розробки медичної системи
2. Аналіз технологій використаних для розробки системи;
3. Розробка системи на основі обраних сучасних технологій.

5. Перелік ілюстративного матеріалу: презентація
1. Мета роботи, об'єкт та предмет дослідження
2. Структурно-логічна схема дослідження
3. Аналіз існуючих фреймворків

4. Технології Графічного Оформлення UI
5. Технології управління глобальним станом у React.js
6. Порівняння доречних мов програмування
7. Сценарій використання системи
8. Зовнішній вигляд розробленої системи (головна сторінка)
9. Сторінки списків (звіти та запити)
10. Сторінка запиту
11. Сторінка звіту
12. Налаштування користувача
13. Висновки

6. Дата видачі завдання 25.02.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Порівняльний аналіз існуючих технологій. Вибір основної технології розробки.	02.03.2025	Виконано
2	Аналіз та підбір другорядних технологій.	07.03.2025	Виконано
3	Поглиблене вивчення основної технології проєкту	15.03.2025	Виконано
4	Поглиблене вивчення другорядних технологій проєкту	23.03.2025	Виконано
5	Аналіз наукових джерел, підготовка матеріалів, ілюстрацій та посилань для використання у роботі	30.03.2025	Виконано
6	Розробка програмного забезпечення на основі обраних технологій	15.04.2025	Виконано
7	Тестування розробленої системи, виправлення в ній помилок	18.04.2025	Виконано
8	Написання кваліфікаційної роботи бакалавра	29.05.2025	Виконано

Здобувач вищої освіти _____ ____Ілля_ЄРШОВ_____
(підпис) (Ім'я, ПРІЗВИЩЕ)

Керівник
кваліфікаційної роботи _____ _Юлія БЕРЕЗОВСЬКА___
(підпис) (Ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 63 стор., 20 рис., 0 табл., 18 джерел.

Мета роботи – розробити веб-орієнтовану інформаційну систему для обробки ключових процесів медичної лабораторії на основі бібліотеки **React.js**; підвищити оперативність обслуговування пацієнтів, зменшити паперовий документообіг і закласти потенціал на інтеграцію з діагностичним обладнанням та зовнішніми медичними сервісами.

Об’єкт дослідження – інформаційно-технологічна інфраструктура медичної лабораторії, що охоплює процеси реєстрації зразків, маршрутизації заявок, обробки результатів аналізів, зберігання та візуалізації медичних даних.

Предмет дослідження – програмні засоби розробки веб-застосунків з використанням React.js та супутнього стеку, що забезпечують надійну, масштабовану та безпечну роботу інформаційної системи.

Короткий зміст роботи:

У вступі обґрунтовано актуальність теми, сформульовано мету, завдання, об’єкт і предмет дослідження.

У першому розділі розглянуті популярні технології для вирішення необхідних задач для розробки лабораторної системи, методом поверхневого порівняння Їх переваг та недоліків обрано ті, що найбільше задовольняють вимогам проєкту. Також проведено первинне ознайомлення з важливими поняттями, використаними, та детальніше описаними у другому розділі.

У другому розділі детально розглянуто ключові для розробленої системи можливості використаних технологій, їхній інструментарій, безпосередньо застосований при розробці системи, глибше описано чому реалізація проєкту засобами обраних технологій була оптимальною.

У третьому розділі розглянуто функціонал розробленої системи, реалістичні сценарії її використання, описано як саме розроблена програма може принести користь лабораторії, що, потенційно, її впровадить.

У висновках підсумовано досягнення поставленої мети, розглянуто теоретичні можливості розвитку проєкту, інтеграції зі сторонніми сервісами для розширення його функціоналу.

КЛЮЧОВІ СЛОВА: Front-End розробка, веб-додаток, React, JavaScript, TypeScript, HTML, CSS, Redux, інформаційна система, медична лабораторія

ЗМІСТ

Стор.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

1. ЛІС – лабораторна інформаційна система.
2. Фронт-енд (англ. «Front-End») / інтерфейс / клієнт / клієнтська частина – це все те, що бачить і з чим взаємодіє користувач на веб-сайті або в додатку, також відомий як User Interface (UI).
3. Бібліотека – це набір готових інструментів (функцій, класів та методів), для вирішення певних поширених задач у програмуванні, використання яких економить час на розробку.
4. Патерн (англ. «Pattern») – узагальнений шаблон вирішення повторюваної задачі.
5. Фреймворк (англ. «Framework») – технологія, що надає структурну основу, чіткі правила і патерни для всього проекту, а також набір попередньо реалізованих основних інструментів, для вирішення поширених архітектурних задач на мові програмування, до якої обраний фреймворк відноситься.
6. API (Application Programming Interface) – це інтерфейс програмування додатків, який визначає набір правил, методів і структур, за якими програмісти можуть взаємодіяти з певною технологією, бібліотекою або сервісом.
7. IDE (Integrated Development Environment) – це програма, яка допомагає розробникам писати код зручно й швидко. Вона поєднує редактор коду, підсвічування синтаксису, автодоповнення, дебагер, термінал та інші інструменти в одному місці. Приклад: Visual Studio Code, WebStorm.
8. User Experience (UX) – це досвід користувача при взаємодії з продуктом. У контексті веб-додатків UX включає в себе зручність навігації, швидкість інтерфейсу, інтуїтивність використання і загальне задоволення від роботи з системою.
9. Developer Experience (DX) – це загальні враження розробника від роботи з певними інструментами, технологіями кодовою базою та середовищем розробки.

10. DOM (Document Object Model) – це структура веб-сторінки, яку браузер створює з HTML-коду. Через звернення до DOM JavaScript може змінювати вміст сторінки.
11. URL (Uniform Resource Locator) – це адреса, за якою знаходиться ресурс в інтернеті. Наприклад, https://google.com – це URL сайту Google. Складається з протоколу (https), домену (google.com) і, можливо, додаткового шляху (/search).
12. SPA (Single Page Application) – веб-додаток, який працює без перезавантаження сторінки. Весь вміст сторінки оновлюється динамічно через JavaScript, навіть при навігації між різними URL.
13. JS / TS (JavaScript / TypeScript) – мови програмування. JS – стандартна мова для вебу. TS – це вдосконалений JS з типами для зручнішої та безпечнішої розробки.
14. Dependency injection (впровадження залежностей) – спосіб передавати об'єкти, які потрібні класу або функції, ззовні. Допомагає уникати дублювання коду й робить його гнучкішим.
15. Boilerplate code – типовий, повторюваний код, який часто потрібен для старту проєкту, навіть якщо він не виконує нічого "цікавого" з точки зору бізнес-логіки.
16. Рендеринг – процес виведення, відмалювання, контенту на екран (наприклад, відображення веб-сторінки у браузері).
17. Рефакторинг – покращення коду без зміни його поведінки (робиш код чистішим і зрозумілішим, не ламаючи те, що вже працює).
18. Дебагінг (англ. «Debugging») / дебаг – пошук і виправлення помилок у програмі.
19. Props (англ. «Properties») / пропси – це властивості/дані, які передаються компоненту для конкретизації та персоналізації його поведінки, як аргументи функції. Через props можна зробити конкретний компонент гнучким, під дещо змінні потреби, замість створення нового.
20. MVP (Minimum Viable Product) – Прототип, мінімальний робочий продукт із базовими функціями, щоб протестувати ідею.
21. Unit tests (модульні тести) – тести, які перевіряють окремі частини програми (наприклад, одну функцію, один компонент) на коректність роботи.

22. Стек (англ. «Stack», тобто technology stack) – набір технологій та інструментів які використовуються для створення певного програмного продукту.
23. Хук (англ. «Hook») – це спеціальна функція в бібліотеці «React», яка дозволяє «підключити» певну функціональність, до функціональних компонентів. Вони роблять код більш компактним та читабельним, надають змогу повторно використовувати функціонал у різних компонентах проекту в лаконічному вигляді.
24. Middleware – це проміжна функція, яка стоїть між запитом і відповіддю. Вона перехоплює дію або запит, може змінити, зупинити або доповнити його, і потім передати далі.
25. ГК – гарячі клавіші.

ВСТУП

Актуальність теми. Медичні лабораторії – ключова ланка системи охорони здоров'я: саме від швидкості та точності лабораторних досліджень залежать ≈ 70 % клінічних рішень. Станом на кінець 2020 року в Україні функціонувало близько 2800 медичних лабораторій, які сумарно виконали майже 500 мільйонів досліджень. Із них 76% належали до системи Міністерства охорони здоров'я, 16% – приватним компаніям (що забезпечили 14% усіх досліджень), а 7% – лабораторіям відомчого підпорядкування [1]. Такий обсяг роботи вимагає наявності ефективних цифрових рішень для обробки, збереження та управління результатами досліджень, а також для спрощення роботи з документообігом між персоналом. Більшість українських лабораторій усе ще використовує застарілі, фрагментовані програмні продукти або взагалі паперові журнали, що спричиняє дублювання даних, втрату часу та підвищує ризик помилок. На світовому ринку домінують дорогі закриті системи, орієнтовані на великі мережі. Розробка веб-орієнтованої лабораторної інформаційної системи (ЛІС) на основі відкритої технології React.js дає змогу створити доступне, масштабоване рішення.

Аналіз останніх досліджень і публікацій. Концепції побудови ЛІС докладно розглянуті у працях P. E. Hickman [2] та інших – висвітлено інтеграцію ЛІС з допоміжними медичними інформаційними системами. Разом із тим, питання проєктування веб-застосунків повного циклу (від приймання біоматеріалу до видачі результатів), побудованих на сучасних, відкритих до використання технологіях, – залишається недостатньо опрацьованим, що і зумовило вибір теми.

Мета і завдання дослідження. Мета роботи – розробити лабораторну інформаційну систему на базі React.js, з метою автоматизації основних процесів роботи медичної лабораторії та скорочення часу обслуговування пацієнтів. Для досягнення мети вирішувалися такі завдання:

1. здійснити аналіз поширених вимог до ЛІС і сформулювати функціональні вимоги;

2. визначити вид розробляємого ПЗ (Desktop додаток, веб-додаток, тощо.) доцільний для задач проєкту;
3. провести аналіз існуючих технологій придатних для розробки обраного виду ПЗ, обрати основну технологію, та супутні інструменти;
4. реалізувати прототип системи на базі обраних методом аналізу і порівняння технологій та інструментів;
5. провести ручне тестування розробленої системи, усунути виявлені під час тестування недоліки;

Об’єкт дослідження – інформаційно-технологічні процеси медичної лабораторії, пов’язані з реєстрацією зразків, виконанням аналізів і видачею результатів.

Предмет дослідження – архітектурні, алгоритмічні та програмні рішення розробки веб-додатків на React.js, що забезпечують безперервне, швидке та безпечне функціонування ЛІС.

Методи дослідження. Для виконання роботи використано: структурно-функціональний аналіз, прототипування, патерни архітектури SPA, сучасні методи та інструменти ручного тестування.

Наукова новизна та практична значущість отриманих результатів. Уперше для українських лабораторій запропоновано комплексне, проте гнучке, SPA-рішення для ПЗ типу ЛІС на основі React.js. До розробки системи було застосовано найсучасніші підходи та інструментарій. Розроблена система може бути впроваджена в медичних закладах різного масштабу, зменшуючи час обробки замовлень та підвищуючи прозорість зберігання та руху даних.

Теоретична, методична та практична значущість. Розроблена ЛІС може бути адаптована для різних медичних установ, що потребують сучасної веб-платформи обробки лабораторних даних. Система має потенціал до інтеграції із зовнішніми сервісами для розширення можливостей автоматизації та обміну даними.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел (18 найменувань) та 14-ти

додатків. Загальний обсяг – 63 аркуші друкованого тексту, ілюстрованих 24-ма рисунками.

1 АНАЛІЗ ТЕХНОЛОГІЙ

1.1 Вибір основної технології

У процесі розробки будь-якої системи постає питання вибору технологічного стеку, тобто які технології використати для її розробки. В теорії, можна обмежитись стандартними, простими, інструментами, без застосування додаткових, складніших технологій. На прикладі фронт-енд розробки, розглянемо наступний стек: стандартну мову розмітки веб-сторінок HTML (HyperText Markup Language), для надання браузеру базової інформації про присутність на сторінці тих чи інших елементів до показу, CSS (Cascading Style Sheets), для надання елементам бажаного зовнішнього вигляду та розташування, і мову програмування JavaScript, що дає можливість зробити елементи веб-сторінки функціональними, інтерактивними. Використання цього стеку однозначно потребуватиме менше попередньої підготовки, оскільки навчитись застосовувати потрібно буде лише «базу», без додаткових ускладнень на вивчення фреймворку та сторонніх бібліотек, проте, з власного досвіду роботи фронт-енд розробником можу зазначити, що це значно уповільнить процес розробки та зменшить потенційні функціональні можливості розробленого додатку.

Оскільки час на розробку завжди обмежений- очікуваннями замовника; строками, встановленими на розробку проєкту в компанії; датою здачі кваліфікаційної роботи, а втратити частину потенційного функціоналу – значить програти конкуренцію на ринку, тому розробницька компетенція дозволяє мені обрати більш комплексні технології для реалізації поставленої задачі, що надасть мені більше можливостей.

Розглянемо декілька фреймворків, порівняємо їх основні можливості з трійцею HTML, CSS, JavaScript та оберемо найкращий варіант для розробки клієнтської частини системи для медичної лабораторії, веб-додатку StudioLAB

WEB. Серед сучасних рішень найпоширенішими є Angular та React. У цьому розділі розглянуто особливості кожного, їхні переваги та недоліки.

1.1.1 HTML, CSS, JavaScript

Використання трійки HTML, CSS і JavaScript є базовим підходом до створення веб-застосунків. HTML відповідає за структуру сторінки, CSS – за її зовнішній вигляд, а JavaScript забезпечує взаємодію користувача з елементами інтерфейсу. Такий підхід дозволяє реалізувати повноцінні функціональні веб-сторінки без використання додаткових фреймворків.

Переваги:

1. Простота у вивченні та використанні. Ці технології є базовими для фронт-енд розробки, мають багаторічну історію та велику кількість доступних ресурсів для навчання. Вони добре документовані, мають зрозумілу логіку застосування, що робить їх відносно простими для початківців.
2. Не потребує зовнішніх бібліотек чи фреймворків. Можна реалізувати повноцінну веб-сторінку, не встановлюючи жодних додаткових залежностей. Це зменшує розмір проєкту, та вихідного коду, і знижує поріг входу – не потрібно розбиратися з налаштуванням сучасних інструментів на початковому етапі.
3. Швидке завантаження сторінки. Оскільки браузеру не потрібно підгружати вихідний код додаткових інструментів, використання лише нативних можливостей браузера означає мінімальну кількість стороннього JavaScript-коду, що позитивно впливає на продуктивність і швидкість завантаження. Це особливо корисно для лендингів, односторінкових сайтів або сайтів із низькою складністю логіки.

Недоліки:

1. Відсутність стандартизації структури коду. Немає чіткої, рекомендованої структури проєкту, що потребуватиме додаткових зусиль на розробку архітектури проєкту на початку.

2. Обмежені можливості повторного використання компонентів. Коли виникатиме потреба повторно використати вже написаний код якогось елементу сайту – наприклад, віджета чи кнопки – такий підхід вимагатиме дублювання коду, що вплине на подальшу легкість підтримки та масштабування проєкту, оскільки для зміни UI компонента в усіх місцях його використання, зміни в коді доведеться вносити в декількох (а інколи і в багатьох) місцях кодової бази. Така роздробленість, через людський фактор, часто провокуватиме дефекти в кінцевому продукті. Цю проблему вирішують фреймворки, які ми далі розглянемо.
3. Відсутність вбудованої реактивності.

Що таке «реактивність»? Реактивність – це оновлення UI у відповідь на оновлення даних. Усі сучасні фреймворки пропонують вбудовані декларативні (declarative) механізми синхронізації між фактичними даними та тими, що відображені в інтерфейсі. Без сторонніх технологій цієї поведінки також можна досягти, але лише імперативним (imperative) шляхом.

Imperative vs declarative підхід до написання коду. Імперативний стиль (imperative) означає, що розробник має детально описувати, як саме щось зробити. Розглянемо приземлений приклад – задача: увійти у двері. У випадку з JavaScript без бібліотек у коді потрібно власноруч, явно, прописати:

1. Підійти до дверей
2. Взятися за ручку
3. Натиснути
4. Відчинити
5. Увійти
6. Зачинити

Тобто розробник сам має керувати кожним кроком. Йому потрібно знати механіку дверей і як правильно з ними взаємодіяти.

У декларативному стилі (declarative) програміст просто каже: «я хочу увійти», і автоматичні двері самі розпізнають його намір і відкриваються. Не потрібно думати, як саме вони працюють, важливо лише ціль, яку потрібно досягти.

Уміння користуватись декларативними інструментами – це як знати формули для вирішення математичних задач. В сучасній розробці не варто винаходити велосипед, його просто потрібно зібрати з вже реалізованих компонентів, вже відточеними за нас інструментами.

«Будь-яка досить розвинена технологія не відрізняється від магії. » – Артур Кларк.

Практичний приклад порівняння реалізації лічильника кліків по кнопці за двома вищеописаними підходами наведено у додатку 1. Приклад простий, але навіть в дрібних проєктах імперативний стиль швидко починає створювати проблеми, закриває простір для масштабування проекту, ускладнює відладку програми. Тому варто завжди схилятися до декларативного підходу в розробці програмних продуктів, а звідси – до вибору технологій, що його підтримують, таких як ті, що ми нижче розглянемо.

1.1.2 Angular

Angular – це повноцінний фреймворк розроблений компанією Google, побудований на TypeScript – обгортці JavaScript'у, що додає в JS типізацію змінних. Він містить усе необхідне для побудови масштабованих SPA. Angular має суворо стандартизовану структуру коду та патерни розробки, тому часто використовується у великих корпоративних проєктах з глибокою інтеграцією сервісів та великою кодовою базою.

Переваги:

1. Інтегрована система модулів, сервісів, маршрутизації.
2. Потужна CLI (інструменти командного рядка). Потребує певного часу на освоєння, проте значно покращує DX, якщо вміти нею користуватись.
3. TypeScript за замовчуванням. Чому сама це є перевагою я розписав у наступному підрозділі, 1.2 «Вибір мови програмування».

Примітки [3]:

1. Модулі – це папки або контейнери для організації коду. В Angular

модуль об'єднує пов’язані частини застосунку – компоненти, сервіси, директиви, тощо. Основним модулем зазвичай є AppModule, до якого можна створювати модулі функціоналу (наприклад, UserModule, ProductModule) для кращої організації функціоналу та коду. Це допомагає розділити застосунок на логічні частини, які можуть завантажуватись лише за потреби, коли певний модуль фактично потрібен, для відкритої зараз сторінки.
2. Сервіси – це багаторазово використовувані класи, які містять бізнес-логіку або код для доступу до даних. Наприклад, UserService може містити функції getUser(), updateUser(), тощо. Вони часто впроваджуються (через механізм dependency injection) у компоненти або інші сервіси. Це дозволяє зберігати код компонентів чистим, організованим і відокремлювати бізнес-логіку від інтерфейсу.
3. Маршрутизація – вбудований в Angular «роутер» дозволяє «безшовно» переходити між різними сторінками веб-застосунку. Наприклад, перехід з /home на /profile, без перезавантаження сторінки.

Недоліки:

1. Великий обсяг коду для реалізації простих речей (так званий «boilerplate code»).

   Для виконання базових задач Angular вимагає створення значної кількості структурних елементів – інтерфейсів, сервісів, модулів, шаблонів.
2. Для цілей проєкту StudioLAB WEB Angular надто «бюрократизований», хоча проект, у перспективі, може значно вирости і набрати більш комплексного функціоналу, великим масштабним «enterprise» проєктом він не стане, потреби додатку можна задовільнити фреймворками-конкурентами, що також пропонують ключові для проєкту переваги Angular, але зі швидшим developer experience, через меншу суворість.

1.1.3 React

React – бібліотека для побудови інтерфейсів від компанії Meta (Facebook). Вона дає змогу створювати компоненти, прості для повторного використання, та динамічно оновлювати контент інтерфейсу ефективніше за Angular, через механізм Virtual DOM (детально про нього пізніше, у 2-му розділі). React застосовується як у стартапах, так і у великих корпораціях (Netflix, Instagram, Airbnb). На сьогодні, React – найпопулярніша бібліотека для розробки SPA.

Переваги:

1. Компонентна структура, яка спрощує повторне використання та підтримку коду. React базується на ідеї розбиття інтерфейсу на незалежні компоненти, кожен з яких відповідає за свою частину функціональності та інтерфейсу. Це дозволяє повторно використовувати код, покращує модульність та значно полегшує підтримку й тестування, особливо в масштабних проєктах, а також зберігає принцип поділу відповідальності, за яким кожен модуль має відповідати за свій окремий функціонал.
2. Висока продуктивність завдяки Virtual DOM. React використовує концепцію віртуального DOM, який дозволяє ефективно оновлювати лише ті частини інтерфейсу, що справді змінилися. Завдяки цьому зменшується кількість операцій над реальним DOM – найдорожчою частиною з точки зору продуктивності – і забезпечується швидкий відгук UI, без підвисань, навіть у складних додатках [4]. Схема різниці оновлення браузерного DOM напряму та через Virtual DOM наведена на рисунку 1.1.

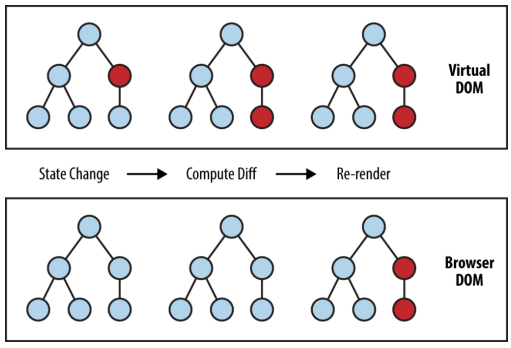


Рисунок 1.1 – Virtual DOM vs DOM [5]

3. Велика спільнота, активна підтримка. React – один із найпопулярніших інструментів у фронт-енд розробці, за яким стоїть компанія Meta (Facebook). Це гарантує регулярні покращення бібліотеки та довгострокову підтримку. Величезна спільнота означає велику кількість навчальних матеріалів, готових рішень, плагінів та бібліотек, що пришвидшують і спрощують розробку.
4. Сумісність із TypeScript.
5. Власне синтаксичне розширення JSX, що дозволяє писати HTML-розмітку безпосередньо в JavaScript-коді. Поверхнева схема написання JSX наведена на рисунку 1.2. JSX дозволяє поєднати розмітку та логіку в межах одного файлу, що значно спрощує роботу з інтерфейсом та функціональним кодом. У порівнянні з традиційним підходом, де HTML, CSS і JS є розділеними, JSX сприяє кращому, наявному, розумінню структури компонента, його роботи та стилізації, і дозволяє зручно управляти елементами у декларативному стилі.

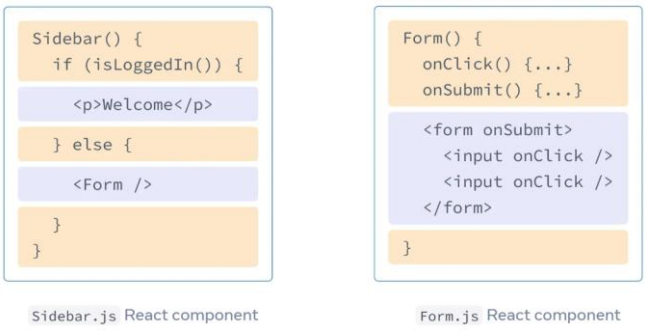


Рисунок 1.2 – Схема написання JSX коду [6]

Недоліки:

1. React – це бібліотека, а не повноцінний фреймворк. Він відповідає лише за рендеринг інтерфейсу і потребує підключення додаткових бібліотек для, наприклад, маршрутизації (React Router) та багатьох інших фундаментальних задач, вирішених в, наприклад, Angular, «з коробки».
2. Відсутність стандартизації продиктованих самим інструментом та його документацією підходів до розробки, структури коду, патернів вирішення поширених проблем.

1.1.4 Критерії вибору технології

Під час вибору технології для побудови клієнтської частини системи StudioLAB WEB вирішальними були визначені такі фактори:

1. Підтримка компонентної архітектури, що забезпечує можливість повторного використання коду, полегшує супровід і тестування окремих частин інтерфейсу.
2. Загальна швидкість розробки на обраній технології.
3. Гнучкість у розробці функціоналу та можливість побудови великого складного додатку.

Усі ці умови найкраще задовольняє саме React, що поєднує простоту, гнучкість та потужність у реалізації складних інтерфейсів. Справедливо буде зазначити, що «з коробки» Angular має більше вбудованого функціоналу, проте React має достатньо розвинену інфраструктуру додаткових, легковагих, проте потужних бібліотек, що легко перекривають брак певної вбудованої функціональності, а також швидко інтегруються в проєкт, хоча це й потребує прийняття додаткових рішень щодо архітектури інтеграцій зі сторонніми бібліотеками на ранніх етапах розробки.

Після детального аналізу зазначених технологій було обрано React як основну бібліотеку для реалізації програми. На відміну від «чистого» JS, він забезпечує структурований підхід до розробки, дозволяє працювати з компонентною архітектурою та реактивністю. У порівнянні з Angular, React є більш доцільним для швидкої реалізації функціоналу, не перевантажує проєкт складною архітектурою та «boilerplate» кодом, а брак чіткої, встановленої «з коробки», структури не є вагомим для проєкту.

1.2 Вибір мови програмування

Обрана для розробки системи бібліотека React підтримує як JavaScript, так і TypeScript. А тому наступним кроком варто обрати мову програмування. У процесі

створення складних веб-додатків, таких як StudioLAB WEB, важливу роль відіграє мова програмування, яка використовується для написання бізнес-логіки, управління станом інтерфейсу та взаємодії з API, для отримання даних до показу та їх модифікації. У фронт-енд розробці де-факто стандартом є JavaScript, однак з кожним роком дедалі більшої популярності набуває його надбудова – TypeScript, що додає систему типів і суттєво покращує стабільність та масштабованість проєктів. У цьому підрозділі проведено порівняння обох мов за ключовими критеріями: безпека, зручність у роботі, передчасне виявлення помилок, продуктивність розробки та підтримка інструментів.

1.2.1 JavaScript

JavaScript – це інтерпретована, слабо типізована мова програмування, яка за замовчуванням підтримується у браузерах. Інтерпретована – означає, що код JavaScript виконується без попередньої компіляції у машинний код. Замість цього, інтерпретатор, наприклад V8 у браузері Chrome, аналізує, компілює та виконує код «на льоту» під час роботи програми.

JavaScript виконується построково: кожна команда читається, аналізується і одразу ж виконується. Однак у сучасних двигунах реалізовано Just-In-Time (JIT) компіляцію, яка комбінує інтерпретацію з оптимізованою компіляцією частин коду для пришвидшення виконання [8].

Такий підхід дозволяє динамічно завантажувати нові скрипти, наприклад код потрібних для окремо взятої сторінки, на яку зараз перейшов користувач, що, з метою оптимізації швидкості первинного завантаження сторінки, не був завантажений одразу, при переході на сайт. Це важко реалізувати у компільованих мовах (наприклад Java), проте це також ускладнює відладку і знижує продуктивність у порівнянні з мовами, код програм яких повністю компілюється перед виконанням.

Схема підготовки коду компільованих та інтерпретованих мов до виконання програми наведена на рисунку 1.3.

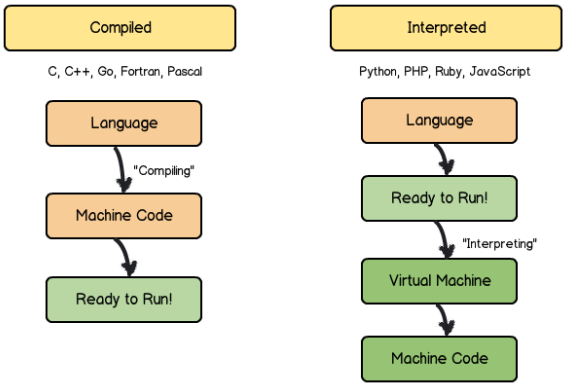


Рисунок 1.3 – Схема-порівняння компіляції та інтерпретації [7]

Переваги:

1. Низький поріг входження для початківців.
2. Вища швидкість розробки MVP через відсутність необхідності написання типів.

Недолік, глобально виділити можна лише один – відсутність системи типів. Хоча, з точки зору швидкої розробки прототипу– це перевага, при довгостроковій роботі з проєктом, швидкість написання коду компенсується проблемами відсутності типів: регулярні помилки типу, наприклад, передача у функцію числа (number) замість рядка (string) виявляються лише під час виконання коду у браузері, а не підсвічуються червоним кольором одразу в IDE, як для типізованих мов. Без типізації або написання unit-тестів часто такі баги важко відловити в процесі розробки, і вони спливають потім, під час ручного тестування, або гірше, на клієнті кінцевих користувачів. Крім того, без типізації ускладнюється підтримка кодової бази. В цілому, поведінка функцій та потік даних у програмі стають менш передбачуваними.

1.2.2 TypeScript

TypeScript – це обгортка JavaScript, що додає статичну типізацію, підтримку інтерфейсів (абстрактних класів з типізованими полями, але без методів) та розширену роботу з об’єктними структурами (Object, Array, Map, тощо). На етапі

збірки додатку код TypeScript компілюється в звичайний JavaScript для запуску в браузері.

Переваги:

1. Автодоповнення та підсвічування помилок в IDE, завдяки статичній типізації. Через відсутність у JS типів, треба більше тривіальних речей прописувати руками, TypeScript економить на цьому час, хоча і додає часу на написання типів. З типізацією можна, наприклад, писати половину назви змінної, або функції, при зверненні до неї, а середовище розробки автоматично запропонує доповнення назви.
2. Чітке визначення контрактів між модулями через інтерфейси. Наприклад, ми можемо точно знати, які дані очікує отримати React-компонент на вхід, для правильного вимальовування інтерфейсу, або об’єкт якої структури ми очікуємо отримати від бек-енду, через написання типів або інтерфейсів.
3. Краще масштабування проєктів – типи забезпечують читабельність, що дозволяє легше розібратися в існуючому коді та краще зрозуміти вірний шлях для додання нового функціоналу в програмний продукт, не ламаючи старого, а також передбачуваність поведінки додатку при внесені змін та доповнень в існуючий функціонал.
4. Легкий рефакторинг – зміна назви змінної або функції автоматично перевіряється у всіх місцях використання. А потенційно зламані, внесеними змінами, імпорти дочірніх залежностей в більш великі модулі будуть підсвічені або, також, автоматично виправлені.

Недоліки:

1. Додатковий етап транспіляції (TS → JS). Оскільки браузери не розуміють TypeScript напряму, код потрібно попередньо транспілювати у звичайний JavaScript. Це додає проміжний етап до процесу генерування вихідного коду, що потребує додаткового налаштування проєкту – потрібна конфігурація, так званого «збирача» (наприклад, Vite, використаний для цієї роботи), і правильна інтеграція з іншими інструментами. Також під

час транспіляції можуть виникати помилки, які не мають впливу на виконання, але все одно блокують збірку. У деяких випадках це може сповільнити ітеративну розробку, особливо в великих проєктах або при некоректній конфігурації [9].

2. Іноді потребує надмірного описування типів, особливо для реалізації простих речей, що може значно уповільнити первинну розробку функціоналу, проте забезпечить надійність його підтримки.

У контексті створення масштабованої системи, яка працює з чутливими медичними даними та повинна гарантувати стабільність і передбачуваність, TypeScript є беззаперечно кращим вибором. Його система типів:

1. Значно знижує ризик помилок у середовищі реального використання.
2. Прискорює дебагінг та рефакторинг.
3. Забезпечує високу якість і підтримуваність коду в довгостроковій перспективі і покращує developer experience, взаємодію з кодом в інтегрованому середовищі розробки.

Тому для проєкту StudioLAB WEB мовою програмування було обрано TypeScript.

Примітивні приклади для порівняння JS та TS коду наведено у додатку 2.

1.3 Вибір технології стилізації та графічного оформлення UI

У процесі розробки веб-додатків важливим аспектом є не лише функціональність, а й зовнішній вигляд інтерфейсу, його зручність та узгодженість. Вибір технології для стилізації інтерфейсу має стратегічне значення, оскільки у комерційній розробці не прийнято повноцінно поєднувати декілька підходів до вирішення настільки глобальних задач. У цьому підрозділі розглянуто три поширені підходи до стилізації: класичне поєднання HTML та CSS, утилітарна бібліотека Tailwind CSS та бібліотека UI компонентів Chakra UI. Основна увага приділяється оцінці зручності розробки, можливості масштабування, доступності (accessibility) та гнучкості кастомізації.

1.3.1 CSS

Базовим підходом до стилізації інтерфейсу є використання HTML у поєднанні з CSS. Цей метод забезпечує повний контроль над структурою і виглядом сторінки. Однак, при масштабуванні проєкту виникає низка труднощів:

1. Складність повторного використання стилів. Звичайний CSS не забезпечує зручних механізмів повторного використання стилів між компонентами або блоками. Якщо потрібно застосувати той самий стиль до кількох елементів, доводиться копіювати правила або створювати узагальнені класи, що ускладнює підтримку. З часом це призводить до надлишкового коду і труднощів із масштабуванням.
2. Відсутність єдиних стандартів компонування. CSS сам по собі не диктує структуру чи підхід до організації стилів. Без чітких правил (наприклад, BEM, SMACSS або CSS Modules) різні розробники можуть писати стилі по-різному, що спричиняє хаос у проєкті. Це ускладнює командну роботу, рев'ю коду й підтримку проєкту в довгостроковій перспективі.
3. Необхідність ручного управління класами та медіа-запитами. У звичайному CSS розробник самостійно має стежити за відповідністю класів HTML-елементам, правильно організовувати їх і підтримувати адаптивність для різних пристроїв через механізм «медіа-запитів». У великих проєктах це стає трудомістким і може спричиняти додаткові помилки, особливо при частих змінах дизайну чи необхідності підтримки різних розмірів екранів.
4. Висока ймовірність дублювання стилів. У CSS легко випадково дублювати стилі – наприклад, написати схожі правила для різних класів або не помітити, що вже існує відповідний клас, що задовільнив би потреби стилізації нового доданого на сайт елементу. Це веде до нераціонального розростання коду, зниження продуктивності та важчої підтримки. Без автоматизації чи стандартизації такі дублікати майже неминучі.

Такий підхід є доцільним для невеликих статичних сторінок, але менш ефективним для динамічних SPA-додатків, яким і являється розроблена в рамках цієї роботи програма.

Проте, знання чистого CSS все ж буде завжди доречним, оскільки деякі бібліотеки готових UI компонентів, наприклад, Chakra UI, – що ми розглянемо пізніше, – під капотом мають велику кількість своїх, базових стилів, що можуть конфліктувати з компонентами з інших бібліотек, чи просто нас не влаштовувати за вимогами дизайну. Такі розбіжності, з очікуваною графікою можна завжди перевизначити на вищому рівні за допомогою стандартного CSS.

1.3.2 Tailwind CSS

Tailwind CSS – утилітарна CSS-бібліотека, яка надає утилітарні класи для швидкого створення стилів без написання детального CSS. Підхід "utility-first" дозволяє описувати зовнішній вигляд безпосередньо у JSX/HTML-коді, а не в окремо виділених файлах, як звичайний CSS.

Утилітарні класи – це маленькі, монофункціональні класи, які відповідають за одну конкретну стилізацію: відступ, колір, розмір шрифту, розміщення тощо.

Переваги:

1. Висока швидкість верстки. Tailwind дозволяє розробникам швидко створювати інтерфейси без потреби в окремих CSS-файлах чи написанні стилів вручну. Усе оформлення виконується безпосередньо в розмітці за допомогою класів, що особливо ефективно для створення прототипів або швидкої реалізації дизайну з фіксованими вимогами.
2. Добра документація та велика спільнота. Офіційна документація Tailwind добре структурована, зрозуміла та наповнена прикладами. Вона значно полегшує вивчення бібліотеки, а також пришвидшує роботу, завдяки зручному пошуку класів і готовим фрагментам коду. Крім того, велика спільнота людей, знайомих з цією бібліотекою, гарантує швидке

вирішення проблем і легкий пошук інформації щодо рішення популярних задач.

3. Можливість створення дизайн-систем. Tailwind дозволяє централізовано налаштовувати теми, кольори, шрифти, відступи, тощо, через файл конфігурації. Це дає змогу створити стабільну та контрольовану дизайн-систему, яка легко масштабується та адаптується під бренд компанії, але початкові зусилля, на створення такої теми, вимагаються більші, ніж на аналогічну роботу в бібліотеках компонентів, як-от Chakra UI.

Недоліки:

1. Зниження читабельності JSX через надмірну кількість класів типу «className="bg-white p-4 rounded-lg shadow-md text-sm text-gray-700"». Велика кількість класів безпосередньо в розмітці може зробити JSX/HTML важчим для сприйняття. Це ускладнює читання, рев'ю коду та підтримку, особливо в складних компонентах, з великою кількістю стилів. У таких випадках, розробники часто змушені вигадувати обхідні рішення, наприклад, винос частини стилів у змінні або використання умовних класів.
2. Відсутність готових UI-компонентів – усі, навіть базові, компоненти (напр. кнопку) потрібно створювати з нуля. Tailwind надає лише утилітарні класи, а не повноцінні компоненти. Тому всі кнопки, форми, модальні вікна та інші елементи інтерфейсу потрібно збирати вручну. Це займає додатковий час на початкових етапах розробки проєкту.
3. Додаткові зусилля для забезпечення доступності (a11y, accessibility). Оскільки Tailwind не містить вбудованих механізмів для забезпечення доступності (на відміну від деяких UI-систем), розробник має самостійно піклуватися про фокус-менеджмент, ARIA атрибути, коректну розмітку тощо. Це потребує додаткових знань і часу.
4. Важче підтримувати консистентність стилів у командних проєктах. У великих командах, якщо не дотримуватися внутрішніх стандартів або не використовувати шаблони, можуть виникати розбіжності в підходах до

верстки і стилізації між розробниками. Наприклад, два розробники можуть створити схожі компоненти з різними наборами класів, що веде до несистемності стилів і ускладнює підтримку.

Tailwind CSS – чудовий вибір для швидкого створення MVP, проте я бажав закласти в проєкт, розроблений в рамках цієї кваліфікаційної роботи, більше простору для масштабування, і, в цілому, з власного досвіду роботи, я не дуже люблю Tailwind, через те, що ним легко перегрузити файли проєкту, вони часто стають нечитабельними, через велику кількість вказаних в них утилітарних стилів, а в серйозній розробці, у суворій відповідності з наданим дизайном, таких стилів багато буде завжди.

Навіщо потрібні атрибути ARIA? Атрибути ARIA (Accessible Rich Internet Applications) в HTML «навішуються» на зацікавлені елементи і використовуються для покращення доступності додатку, особливо для людей, які користуються екранними читачами або іншими допоміжними технологіями, людей з інвалідністю. Вони допомагають передати додаткову інформацію про елементи інтерфейсу, яку звичайний HTML не може повноцінно донести до таких допоміжних інструментів [10].

HTML сам по собі іноді не здатен описати складні елементи інтерфейсу (наприклад, кастомні випадаючі списки, модальні вікна, вкладки тощо). ARIA заповнює ці прогалини, дозволяючи зробити прозору взаємодію з такими компонентами можливою для всіх користувачів.

1.3.3 Chakra UI

Chakra UI – сучасна бібліотека UI компонентів для React, UI-система, що поєднує гнучкість, простоту та високі стандарти доступності. Вона базується на дизайн-принципах styled-system і дозволяє створювати UI за допомогою стилізаційних пропсів безпосередньо в JSX.

Styled System – це підхід до стилізації компонентів у React, заснований на utility-пропсах. Він дозволяє передавати стилі прямо в пропси компонента, не

пишучи окремий CSS. Підхід дещо схожий на Tailwind, але він дає на виході більш читабельний код та набагато вищу гнучкість.

Переваги:

1. Гнучка темізація з можливістю зміни компонентів бібліотеки на рівні всього застосунку, розвинена стандартна тема. Chakra «з коробки» надає нам доступ не лише до готових компонентів, наприклад, вже стилізованих кнопок, а й окремих токенів у спрощеному вигляді, наприклад світло-сірий колір: gray.100 замість #f4f4f5. Крім того, система темізація у Chakra UI дозволяє легко задавати глобальні кольори, розміри, шрифти, для широкого використання по всьому проєкту, а також перемикатися між світлою та темною темами додатку. Це особливо корисно для підтримки корпоративного стилю або створення продуктів із динамічним виглядом. Такий продукт можна постачати різним клієнтами, з однаковим, або дещо кастомізуємим UI, в корпоративному стилі замовника, покриваючи, по суті, потреби декількох клієнтів одним програмним продуктом.
2. Зручний синтаксис. Стилі пишуться у вигляді пропсів – px={4} (padding: 4), bg="gray.100" (background: #f4f4f5), що підвищує читабельність і не роздуває надмірно кількість коду. Такий підхід значно спрощує візуальне сприйняття коду компонента. Стилі інтегровані безпосередньо в JSX, завдяки чому структура компонента стає більш лаконічною, а також не створюються окремі стилізовані файли.
3. Повна підтримка доступності за замовчуванням. Chakra UI забезпечує доступність більшості елементів інтерфейсу – фокус-менеджмент, підтримка клавіатури, правильне оголошення ролей елементів програмами читачами екрану і атрибутів ARIA. Це дозволяє розробникам створювати застосунки, доступні людям з інвалідністю, без додаткових зусиль і спеціалізованих знань.
4. Широкий набір готових адаптивних компонентів (модальні вікна, кнопки, форми, вкладки тощо). Chakra UI забезпечує високий рівень готовності – компоненти вже мають графічне оформлення, стилі та адаптивність. Це

дозволяє швидко будувати інтерфейси без необхідності писати типові елементи з нуля, що особливо важливо в MVP та стартап-проєктах.

5. Нативна інтеграція з TypeScript. Усі компоненти мають прописані типи, що дозволяє використовувати підказки, автозаповнення та перевірку типів в код-редакторі. Це покращує DX, підвищує стабільність коду, зменшує кількість помилок і пришвидшує розробку.

Недоліки:

1. Менша популярність порівняно зі схожими бібліотеками компонентів, наприклад Material UI, що потенційно означає підвищену складність вирішення проблем, у разі їх виникнення, через меншу спільноту, та, як наслідок, меншу кількість інформації по Chakra в інтернеті. Незважаючи на високу якість, Chakra UI поступається за популярністю лідерам ринку. Це означає, що при нестандартних задачах знайти приклади, відповіді або готові рішення може бути складніше, а вирішення деяких проблем вимагатиме більше часу та дослідницької роботи.
2. Обмежена кількість повноцінних «темплейтів», комплексних компонентів, що вирішують задачі більш широкого спектру, наприклад повноцінний каркас для форми оплати на сайті, у порівнянні з деякими, більш «дорослими», бібліотеками. Хоча Chakra надає багато базових елементів, для більш складних сценаріїв, таких як системи управління контентом, багаторівневі таблиці, інтегровані системи форм – доводиться створювати власні рішення або доповнювати сторонніми бібліотеками. Це може уповільнити розробку і збільшити кількість коду написаного вручну.

Усі згадані підходи мають право на існування, і вибір конкретного залежить від специфіки проєкту. Проте для реалізації клієнтської частини StudioLAB WEB було обрано саме Chakra UI з наступних причин:

1. Забезпечує високу швидкість розробки без втрати якості коду, поєднує в собі переваги утилітарних стилів Tailwind та готових до використання компонентів.

2. Гарантує доступність інтерфейсу для користувачів з інвалідністю, що користуються допоміжними програмами для сприйняття контенту, «з коробки». Це не обхідна вимога для програми, що оперуватиме в медичному домені.
3. Легко масштабується та налаштовується під потреби проєкту завдяки системі темізації і стильових пропсів.
4. Спрощує підтримку та розвиток графіки додатку.

Завдяки цим перевагам Chakra UI виявився оптимальним вибором для побудови сучасного, доступного та масштабованого інтерфейсу для медичної системи.

Приклад коду імплементації однакового стильового оформлення компоненту з використанням усіх розглянутих вище технологій наведено в додатку З.

1.4 Вибір технології управління глобальним станом

Ефективне управління станом інтерфейсу є критичним аспектом при розробці сучасних веб-додатків. Нагадаю, що стан потрібен для динамічного оновлення інтерфейсу у відповідь на дії користувача, показу актуальних даних без оновлення сторінки. Існує умовне розділення стану на локальний і глобальний.

Локальний стан стосується окремого компонента або обмеженої частини інтерфейсу, наприклад, стану модального вікна чи форми, або прикладу компонента лічильника, наведеного в додатку 1. Для обробки локального стану достатньо вбудованого React-хука useState, який повністю задовольняє потреби обробки локального стану компонента.

Глобальний стан охоплює ширший контекст, дані, спільні для багатьох компонентів, або й взагалі для всіх компонентів додатку. Наприклад, нинішня обрана тема, де абсолютно усім компонентам інтерфейсу потрібно знати, чи тема зараз світла чи темна, аби динамічно, спираючись на цю дану, змінювати відповідно свій колір. На рисунку 1.4, на прикладі бібліотеки Redux схематично показано, як виглядала б передача таких даних ланцюжком, від компонента до

компонента багато разів, у порівнянні з передачею їх у «сховище» глобального стану, яке само оновить UI усіх зацікавлених компонентів.

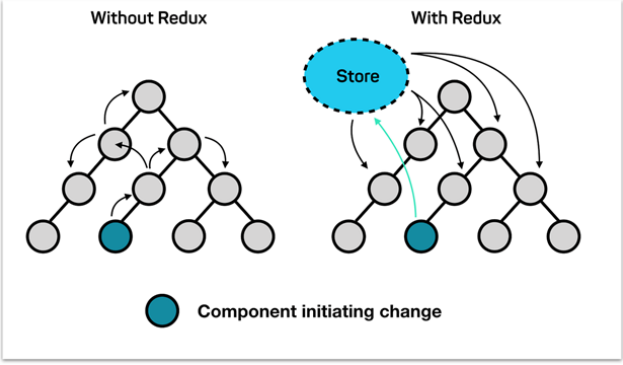


Рисунок 1.4 – Оновлення стану з та без Redux [11]

У контексті побудови програми StudioLAB WEB, яка обробляє медичні дані, результати аналізів, документообіг, автентифікацію прав користувача, згідно якої контент додатку може змінюватись у різних частинах додатку, та, власне, світлої/темної теми, виникає потреба в потужному, гнучкому та масштабованому рішенні для роботи із глобальним станом додатку. У цьому розділі порівнюються 3 популярні підходи до управління станом у React-додатках: вбудований хук useContext, легковага бібліотека Zustand та потужна й найпоширеніша бібліотека Redux Toolkit.

1.4.1 useContext

useContext – це вбудований хук React, який дозволяє передавати дані між компонентами без явної передачі пропсів на кожному рівні. Він часто використовується для зберігання глобальних значень – наприклад теми, мови, авторизації користувача – у невеликих або середніх за масштабом застосунках.

Переваги:

1. Не потребує зовнішніх залежностей. useContext входить до складу React, тож не вимагає встановлення сторонніх бібліотек. Це знижує загальну складність проєкту.
2. Проста реалізація для невеликих застосунків. Для передачі глобального стану достатньо створити контекст і обгорнути в нього дерево

компонентів. Такий підхід легко реалізується, не потребує складної архітектури і добре підходить для невеликих SPA-додатків.

3. Зрозумілий API. Інтерфейс useContext інтуїтивно зрозумілий: розробник створює контекст, огортає ним компоненти і отримує доступ до даних за допомогою хука всередині цих компонентів – це легка концепція для сприйняття та безпосереднього використання.

Недоліки:

1. Неоптимізовані повторні рендери при оновленні контексту. Коли змінюється значення контексту, перерендерюються всі компоненти, що в нього обгорнуті, незалежно від того, чи конкретний компонент зацікавлений у зміненій змінній контексту. Це може призводити до зниження загальної продуктивності додатку, оскільки, нагадаю, рендеринг – найдорожча, з точки зору продуктивності, операція на фронт-енді, особливо якщо структура дерева компонентів велика або контекст часто оновлюється.
2. Відсутність вбудованої підтримки асинхронних дій. useContext не надає механізмів для асинхронної логіки або обробки побічних ефектів. Щоб реалізувати складніші сценарії, потрібно комбінувати його з іншими хуками або сторонніми бібліотеками.
3. Обмежена масштабованість для великих додатків. У великих застосунках виникає потреба у багатьох контекстах, які важко координувати між собою. Це ускладнює архітектуру та збільшує ризик помилок, пов’язаних із повторними рендерами, надмірною глибиною вкладеності контекстів або непередбачуваними залежностями.

Дерево компонентів (англ. «Component tree») – це структура вкладених React-компонентів, яка показує, які компоненти містить в собі певний компонент.

Подібно до дерева DOM: є батьківські (parent) компоненти і дочірні (child). Вони створюють ієрархію, яка відображає логіку інтерфейсу, наприклад: <App> → містить <Header>, <Main>, <Footer> → <Main> містить <Sidebar> і <Content> – це і є дерево компонентів.

1.4.2 Zustand

Zustand – це мінімалістична бібліотека для керування станом у React-застосунках. Вона побудована на базі простого API, яке дозволяє створювати глобальні або локальні сховища стану поза межами окремих React-компонентів, із мінімальним впливом на продуктивність.

Переваги:

1. Простота та компактність синтаксису. Zustand дозволяє створити зовнішнє, для дерева компонентів, сховище стану всього кількома рядками коду, без потреби у редукторах, діях чи шаблонному коді. Це значно пришвидшує початок роботи, знижує вхідний поріг і робить код лаконічним та легким для підтримки.
2. Ефективне оновлення стану без зайвих ре-рендерів. Бібліотека оптимізована для уникнення непотрібних повторних рендерів: компоненти підписуються лише на ті дані зі стану, які їм справді потрібні. Це дозволяє зберігати високу продуктивність навіть у складних і насичених інтерфейсах.
3. Вбудована підтримка асинхронних функцій. На відміну від класичних рішень, Zustand дозволяє без зайвих обгорток використовувати async/await та інші асинхронні патерни безпосередньо всередині сховища. Це полегшує реалізацію, наприклад, запитів до API, та прибирає потребу використання окремих бібліотеках для обробки асинхронних подій [12].

Недоліки:

1. Обмежена інтеграція з інструментами Chrome DevTools. Хоча існують деякі інструменти для дебагу стану, Zustand не має такого ж рівня підтримки DevTools, як, наприклад, Redux. Це ускладнює відстеження змін стану в реальному часі, особливо в процесі розробки великих застосунків, що погіршує DX.
2. Менш розвинена екосистема. Попри зростаючу популярність, екосистема Zustand поки що менш розвинена, ніж у «класичних» рішень типу Redux

або MobX. Це означає менше готових прикладів, обгорток і рішень для типових сценаріїв, що іноді вимагає додаткового часу на дослідження або написання власної логіки з нуля.

1.4.3 Redux Toolkit

Redux Toolkit (RTK) – це рекомендований спосіб роботи з Redux, розроблений авторами самої бібліотеки Redux. Він надає сучасні абстракції для ефективного управління глобальним станом, зменшуючи кількість шаблонного коду та помилок, а також пропонуючи стандартизований та масштабований підхід до побудови бізнес-логіки застосунку.

Переваги:

1. Централізоване управління станом. RTK забезпечує чітку структуру, де увесь глобальний стан керується з єдиного місця – це значно спрощує контроль, дебаг і масштабування застосунку. Такий підхід особливо корисний у великих проєктах із багатьма модулями, де важливо зберігати послідовність і контроль над даними.
2. Підтримка DevTools, middleware, масштабованих проєктів. Redux Toolkit інтегрується з Redux DevTools, що дає змогу відстежувати історію змін стану, переглядати стан на кожному етапі відмальовування інтерфейсу у браузері та відкатувати його назад, для ручного тестування проблем. Крім того, RTK легко підтримує middleware (наприклад логування подій, реєстрацію дій в аналітику, обробку помилок), що робить його потужним інструментом у проєктах будь-якого розміру.
3. Стандартизовані патерни. Стандартизація RTK зменшує різноманітність підходів до розробки. Це підвищує передбачуваність архітектури, полегшує вирішення поширених проблем та допомагає підтримувати чистоту коду.

Недоліки:

1. Дещо вища складність початкової конфігурації, що потребує певної кількості «boilerplate» коду.
2. Може бути надлишковим для дуже малих додатків. У простих застосунках із мінімальним обсягом стану використання Redux Toolkit може виявитися надмірним. Базові задачі, які можна реалізувати кількома хуками, тут потребуватимуть значної конфігурації, що ускладнить малий проект без достатніх підстав.

Усі розглянуті підходи мають свої переваги й можуть бути ефективними залежно від масштабу і структури проєкту. Проте для реалізації системи StudioLAB WEB було обрано Redux Toolkit, оскільки він:

1. Забезпечує строгий і зрозумілий підхід до керування глобальним станом.
2. Добре масштабується.
3. Інтегрується з DevTools, що дуже допомагає в розробці.
4. Має велику спільноту і активну підтримку з боку розробників.

Redux Toolkit демонструє ідеальний баланс між гнучкістю, структурованістю та функціональністю для складних інтерфейсів, де важливі передбачуваність у реалізації функціоналу, контроль та розширюваність.

Отже, фінальний обраний стек ключових технологій, без врахування відносно дрібних бібліотек, що закривають менш важливі потреби, такий:

1. Фронт-енд бібліотека, основна технологія, каркас проєкту: React.
2. Мова програмування: TypeScript.
3. Технологія графічного оформлення UI: Chakra UI.
4. Технологія управління глобальним станом: Redux Toolkit.

2 АНАЛІЗ ОБРАНОГО СТЕКУ ТЕХНОЛОГІЙ

2.1 React

У центрі React лежить ідея, що інтерфейс можна розбити на малі, незалежні компоненти, замість написання цілого сайту всього у трьох файлах – index.html, index.css, index.js, – по одному файлу на зону відповідальності, як це пропагують уроки для новачків на різних інтернет ресурсах.

Прості компоненти нагадують атоми, наприклад кнопка та поле вводу (англ. «Input»), вони не мають власного контексту, проте є невід'ємними будівельними блоками інтерфейсу. Атоми можуть збиратись в більші компоненти – молекули, що вже виконують якусь маленьку, ізольовану задачу, наприклад поле пошуку (SearchField), складене з атомів наведених вище – Input для вводу запиту та кнопки «Пошук». З молекул складаються ще більш повноцінні та функціональні компоненти – організми, наприклад секція пошуку (SearchSection), поєднання молекули пошукової строки та секції для показу результатів під нею. Таке уявлення про компоненту структуру React називається «Atomic Design». Розширена схема принципу Atomic Design наведена на рисунку 2.1.

Слідування даному патерну значно спрощує сприйняття коду, дозволяє легше уявити згадане в попередньому розділі дерево компонентів, що в свою чергу дає легше зрозуміти взаємозв'язок між компонентами, їх вкладеність, спільний контекст, і т.д [13].

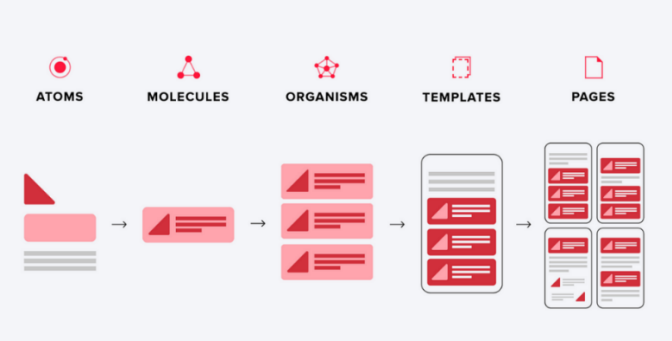


Рисунок 2.1 – Схема втілення Atomic Design [13]

Поглиблюючись в технологію, React-компоненти – це функції, що можуть, як і звичайні функції, приймати на вхід параметри (пропси), та відповідно до переданих пропсів «підлаштовуватись», або просто відображати відповідний контент. Наприклад, компонент кнопки може мати різний напис для різних цілей в додатку, а також приймати на вхід і, відповідно, використовувати різні обробники події кліку, простіше кажучи, виконувати різні дії на клік користувача. Іноді замість пропсів можна передавати у компонент children, принципової різниці немає, лише дещо змінюється синтаксис, а також children є доречнішим для реалізації деяких патернів React розробки, які я упущу в рамках цієї роботи. В контексті даної роботи, можемо сприймати children як звичайні пропси.

Приклад реалізації в коді вище наведених компонентів за «Atomic Design» наведено у додатку 4, для верстки було використано JSX та UI компоненти Chakra, і далі так буде в усіх прикладах кодів цієї роботи, а також у Додатках до неї.

Кожен компонент описує, як має виглядати інтерфейс та його бізнес логіку, а React перетворює, збирає, ці «описи» в те, що внутрішній двигун браузеру може фактично відобразити (звичайні HTML/CSS/JS файли). Оскільки компонент можна повторно використовувати, просто передаючи йому різні дані на вхід (пропси), дизайнери отримують узгоджений спосіб уникати дублювання та дрібних розбіжностей в макетах, у виді відступів відмінних на пару пікселів, коли на увазі мався той самий віджет, а продуктові команди можуть мислити інтерфейсом як сукупністю взаємозамінних блоків, а не єдиною монолітною сторінкою.

Відомі фреймворки Vue та Angular також використовують компоненти, але Angular поєднує їх із більшою, обов’язковою до конфігурації та підтримки платформою, тоді як React свідомо обмежує свою зону відповідальності на роботі саме з інтерфейсом, щоб розробники могли вільно підбирати додаткові інструменти. Такий вузький фокус високо цінується за гнучкість.

Ще один ключовий принцип – декларативне програмування. Замість того щоб поетапно наказувати браузеру, що робити («знайди список, встав новий елемент, зміни його колір»), розробник пише функцію, яка по суті каже: «Якщо в кошику три товари, покажи біля значка число 3». Щоразу, коли дані змінюються,

React повторно виконує функцію та тихо з’ясовує, що потрібно змінити на екрані. Альтернативний імперативний стиль, типовий для старих jQuery-проєктів, дає максимальний контроль, але погано масштабується, бо кожна нова функція множить кількість інструкцій, які програміст має вручну синхронізувати. Натомість декларативний стиль React тримає опис «як це повинно виглядати» поруч із даними, що його визначають, тож ментальна модель залишається керованою навіть у великих застосунках.

Під капотом React створює віртуальне представлення DOM-дерева браузера. Реальний DOM потужний, але повільний: його модифікація напряму змушує браузер перераховувати розмітку й перемальовувати пікселі. React тому тримає власну полегшену копію сторінки в оперативній пам’яті – Virtual DOM. Коли стан змінюється, React формує нове Virtual DOM-дерево, порівнює його з попереднім, виявляє гілки, що відрізняються, і вносить мінімальний набір правок у реальний DOM необхідних для коректного відображення оновлених даних. Оскільки операції з пам’яттю на порядки швидші за взаємодію з DOM, більшість роботи виконується без участі браузера. Офіційний алгоритм зіставлення «Reconciliation» проходить обидва дерева від кореня, вважаючи елементи різних типів різними піддеревами й перевикористовуючи гілки, коли типи збігаються. Схему оновлення DOM дерева за алгоритмом віртуального DOM, – Reconciliation – наведено на рисунку 2.2. У лівій гілці червоним підсвічено компоненти, стан яких змінився, а у правій, синім, – ті самі компоненти, та їх дочірні компоненти, що були фактично перемальовані браузером. Перемальовувати дочірні компоненти, коли змінився стан батьківських – стандартна поведінка React, оскільки в такому сценарії, для дочірніх компонентів змінився контекст, а з ним, потенційно, і дані, що мають бути відображені на сторінці.

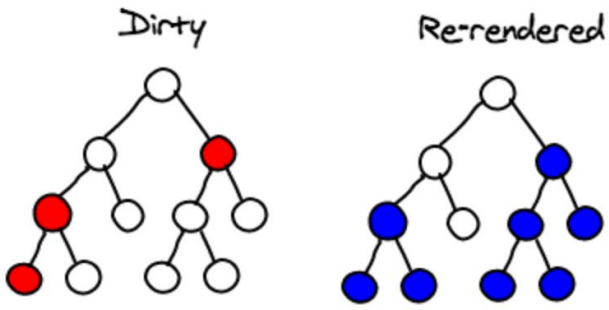


Рисунок 2.2 – Схема роботи алгоритму «Reconciliation» [14]

Розглянемо основні інструменти React, які були використані при розробці системи. У першому розділі вже згадувались React-хуки, що є основним набором інструментів бібліотеки, які вигідно виділяють її на фоні конкуруючих рішень.

Хуки викликаються в середині компонента, як звичайні функції, та повертають відповідні засоби для маніпуляції компонентом з середини. Назви хуків починаються зі слова «use».

useState – хук відповідальний за зберігання та оновлення внутрішнього стану компонента. Він приймає параметром на вхід початкове значення змінної стану, повертає змінну стану та set функцію, для модифікації значення змінної стану, приклад: const [data, setData] = useState([]). Виклик set функції – єдиний спосіб змінити значення state змінної, пряме переназначення (напр. data = [‘newData’]) не дозволяється. Точніше, воно не приведе до помилки виконання програми чи збірки проєкту, проте зміна значення не спровокує ре-рендер і, відповідно, не буде відображена у компоненті, тобто змінна data в логіці виконання коду матиме нове значення, а на екрані в браузері – інше. Така поведінка і пояснює необхідність використання useState, це єдиний спосіб спровокувати перемальовування компоненту, а звідси й відображення дійсних, актуальних даних у React.

Бувають випадки, коли для розробки певного функціоналу нам необхідна змінна, при зміні значення якою не повинен відбуватись ре-рендер. Чому не використати звичайну JS змінну? Оскільки ми все ж очікуємо отримувати актуальне значення цієї змінної між рендерами (як з useState), існує росте і зрозуміле обмеження: React компоненти, як вже згадувалось – це функції, а при ре-рендері функція (компонент) просто перевиконується зверху вниз, таким чином

обнуляючи будь які змінні. Виключення є змінні повернені хуками, вони «підчиняються своїм правилам».

Хук useRef повертає нам об’єкт, що має поле current всередині. При зміні поля current ре-рендер не викликається, проте, у разі ре-рендеру компоненту за інших причин (зміна власного стану або пропсів) – актуальне значення буде передано між ре-рендерами. Окрім того, useRef можна використовувати для прямих DOM-маніпуляцій над елементами, коли це необхідно.

В реальних веб-додатках рендером компонентів інтерфейсу функціонал програми не обмежується. Існує необхідність отримувати дані з бек-енду, «підписуватись» на real-time події (напр. отримання повідомлень), показ рекламних спливаючих вікон за часовим інтервалом, – все це від рендерингу графіки має бути відокремлено, бути не його частиною, а побічним ефектом.

Хук useEffect приймає два параметри: функцію до виконання та масив залежностей. У масив залежностей можна передавати будь що, що провокує ре-рендеринг, тобто state змінні та пропси, і при зміні однієї з залежностей передана функція буде перевикличена заново (за замовчуванням вона буде викликана при першому рендері). useEffect виконується після завершення вимальовування DOM у браузері.

useLayoutEffect працює майже так само, як useEffect, але виконується після розрахування (не вимальовування) DOM до того як браузер покаже оновлений кадр. Використовується коли нам потрібно динамічно порахувати розташування якогось елементу чи, наприклад, автоматично проскролити сторінку до бажаної початкової позиції. Окільки useEffect виконується після вимальовування, з ним буде помітне мерехтіння між початково вимальованим кадром, та зміненим в середині useEffect, з useLayoutEffect ми отримаємо одразу очікуваний рендер.

Часто багатьом компонентам може бути потрібно спиратись на той самий стан: поточна кольорова тема сайту (світла / темна), дані поточного користувача, обрана мова – це все приклади даних, до яких додатку часто потрібен доступ скрізь, з середини багатьох різних компонентів. Передавати їх пропсами через усі рівні незручно й крихко. Функція createContext створює глобальне значення, а useContext

дозволяє будь-якому нащадку компонента, в якому об’явлено контекст, прочитати його і підписатись на зміни стану контексту.

Як вже вище згадувалось, кожен ре-рендер виконує тіло функції компонента наново. Буває що в тілі присутня якась дорога операція, наприклад: фільтрація великого масиву, складні обчислення. useMemo мемоізує (або простіше – кешує) результат і перераховує його лише коли змінюється масив залежностей (синтаксис однаковий з useEffect). Це зменшує навантаження на процесор і оперативну пам’ять.

Приклад використання useMemo наведено на рисунку 2.3.

```
const [data, setData] = useState(0);
// Симуляція "дорогого" обчислення
const squared = useMemo(() => {
  return data.doSomeVeryExpensiveCalculations();
}, [data]);
```

Рисунок 2.3 – Приклад використання useMemo

useCallback має той самий синтаксис що useMemo та схоже призначення, – він кешує функції. У тілі компонента ми можемо створювати функції, наприклад handleClick для обробки події натискання користувачем на кнопку, проте на кожен перевиклик функції компонента при ре-рендерах усі створені в середині компонента функції були б перестворені, забиваючи оперативну пам’ять до того моменту, поки автоматичний збірник сміття («garbage collector») вбудований у JavaScript їх не прибере. З useCallback ми, як і з useMemo вказуємо залежності, і тільки у разі їх зміни, функція буде перестворена наново.

Приклад використання useCallback наведено на рисунку 2.4.

```
// Функція не створюється наново при кожному рендері
const logValue = useCallback(() => {
  console.log("Поточне значення:", value);
}, [value]);
```

Рисунок 2.4 – Приклад використання useCallback

У підсумку, хуки дозволяють функціональним компонентам керувати станом, побічними ефектами, даними, до яких потрібен спільний доступ багатьом компонентам, загальною продуктивністю програми та доступом до DOM. Вони скорочують код і заохочують будувати інтерфейс з дрібних, взаємо поєднуваних

частин, задовольняючи Atomic Design, а крім того – надають незамінні інструменти для фронт-енд розробників, в цілому, без яких важко уявити реалізацію сучасних веб додатків, а розробити їх самотужки – виглядає непосильною задачею. Життя без них означало б важчий синтаксис, дублювання логіки та повільніші застосунки – саме ті проблеми, які хуки й покликані вирішити.

Приклади використання хуків не супроводжених рисунками наведено в додатку 5.

2.2 TypeScript

TypeScript – мова програмування розроблена Microsoft, що вирішує головну проблему JavaScript – відсутність типізації. Якщо хтось мав справу з розробкою на Java, а потім на Python – ситуація подібна, з тією різницею, що синтаксис у JS та TS однаковий, в TS лише додається додатковий код для зазначення типів.

Приклад простої типізованої TS функції наведено на рисунку 2.5.

```
// Визначаємо простий тип для вхідних даних
type Product = {
  name: string;
  price: number;
};

// Функція, яка застосовує знижку і повертає нову ціну
function applyDiscount(product: Product, discountPercent: number): number {
  const discount = product.price * (discountPercent / 100);
  return product.price - discount;
}
```

Рисунок 2.5 – Приклад типізованої TS функції

TypeScript має під капотом потужний механізм аналізу коду на етапі компіляції, що також звітує IDE про помилки компіляції, особисто я обираю TypeScript замість JavaScript навіть для виконання дрібних задач, саме через цю, – дійсно невід’ємну – функцію. Найгірше навіть не те, що JS не дає про них знати завчасно, а й те, що звітує він про них з надто малою кількістю заплутаних між собою деталей, двигун TS поводить себе, очевидно, інакше, дозволяючи економити значущу кількість часу на відладці програми.

Ще одна перевага TypeScript – можливість поступової адаптації. Проєкт може початися на чистому JavaScript, а потім поступово мігрувати на TS, що

дозволяє відносно легкий та поступовий рефакторинг проєкту, що починався на JS, без блокування комерційної розробки.

Оскільки браузери розуміють тільки нативний для них JS, необхідний етап транспіляції – компілятор TS видаляє з коду типи й генерує звичайний JavaScript, який виконується в будь-якому браузері. JavaScript динамічно типізований: змінна може містити будь-яке значення, але не будь який шматок коду, де змінна згодом буде використана, може обробити будь яке з тих її можливих значень, а помилки типів стають помітні лише під час виконання. TypeScript розв’язує це, дозволяючи явно вказувати типи, компілятор перевіряє всі можливі гілки (шляхи виконання коду) та сигналізує про невідповідності, що переносить багато багів з етапу виконання коду у браузері замовника, ставлячи успішність розробленого продукту під питання, до вікна редактора коду розробника й дає змогу сміливо рефакторити, маючи «страховку» у вигляді контролера типів («Type checker»).

Типова система структурна, а не номінальна: сумісність визначається формою значення, а не назвою змінної. Якщо два об’єкти мають ті самі властивості зі сумісними типами, кожен може замінити інший. Розробник може описувати об’єднання типів (union), перетини (intersection), а також умовні типи (conditional), комбінуючи типи у нові, збірні типи [15].

Основні можливості обраної для розробки проєкту мови розглянули, а тепер поглибимось у принцип її роботи. Говорити будемо про цикл подій браузера («JavaScript Event Loop»). Схематичне зображення роботи Event Loop наведено на рисунку 2.6.

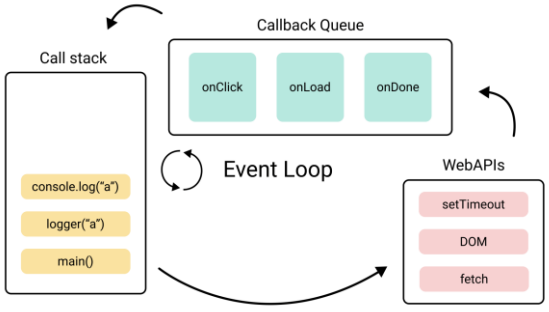


Рисунок 2.6 – Схема роботи Event Loop

Контекст того, як компілюється TS у JS нам більше не знадобиться. Кожна вкладка браузера має свій «стек» викликів («Call Stack») та «Callback Queue», поділену на 2 дві основні черги: «macrotasks» (макрозадача) та «microtasks». Коли браузер запускає скрипт, він наповнює стек синхронним кодом, кодом що виконується послідовно, а не, наприклад, як запити до бек-енду – викликається зараз, а відповідь отримаємо згодом. Задачі зі стеку – перші у черзі на виконання, щойно стек спорожніє, цикл подій перевіряє microtask-чергу; якщо там щось є, виконуються усі макрозадачі до кінця по черзі, а потім одну мікрозадачу (з microtasks queue). Тільки після проходження вище наведених ітерацій цикл може виконати рендер, вимальовування контенту у браузері, а потім починає наступну ітерацію, у тому самому порядку, за тими самими принципами.

Концепт складний для пояснення, але розглянемо його на прикладі React компонента, що робить запит в API. Наведеного на рисунку 2.7:

1. З точки зору циклу подій, компонент – звичайна функція, вона виконається у call stack та породить 1 macrotask викликом функції fetch.
2. Macrotask запуститься, проте не поверне нам відповідь одразу, її від сервера доведеться почекати, тому йдемо далі до рендеру.
3. Вимальовується контент компонента, оскільки users (див рисунок 2.7) з API запиту ще тільки очікуються, відображаємо стан завантаження.
4. Цикл подій повторює ці ітерації багато разів на секунду, якщо в черговій ітерації запит до API вже повернув результат – на етапі вимальовування покажемо на екран список користувачів, замість компоненту використаного для візуалізації процесу завантаження.

```jsx
function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => response.json())
      .then(data => {
        setUsers(data);
        setLoading(false);
      });
  }, []);
  if (loading) {
    return <p>Loading...</p>;
  }
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Рисунок 2.7 – React-компонент, що робить запит в API

TypeScript не впливає на роботу циклу подій; згенерований в момент його компіляції JS бере участь у тій самій схемі планування циклу подій. Уміння орієнтуватися в ній важливе, для розробки, проте без реальної практики до кінця зрозуміти її та навчитись оптимізувати код під її поведінку – неможливо. Тому я вважаю недоцільним поглиблюватись в Event Loop детальніше, навіть якщо він лежить у ядрі розробленої в рамках цієї класифікаційної роботи ЛІС.

Існують і потенційні альтернативи TypeScript, інші типізовані, дещо схожі за ідеєю мови. Dart від Google, також компілюється в JavaScript, проте позиціонується як окрема мова програмування, зі своєю екосистемою. Haxe, Elm і ReasonML теж генерують вихідний JavaScript код, але мають настільки віддалений синтаксис від JS, що сприймаються вже як зовсім інші мови програмування. А ще, вони не адаптовані популярними існуючими технологіями, такими як React, тому свою позицію на ринку мов програмування TS, в найближчому, майбутньому точно не втратить.

Підсумовуючи: TypeScript покращує JavaScript, додаючи йому надійну систему статичної типізації, не змінюючи семантики мови для середовища її виконання. Впровадження TS у проєкт дозволяє писати продуктивні веб-додатки з кращим DX, і особисто я, з власного досвіду, рекомендував би усім JavaScript розробникам якомога раніше додати його у список своїх компетенцій, оскільки, на

сьогоднішній день, він вже витісняє навіть звичайний JS, за популярністю використання.

2.3 Chakra UI

Chakra UI – це відкрита бібліотека UI компонентів для React, мета якої дати розробникам змогу швидко переходити від ідеї проєкту до його реалізації, з самого старту гарантуючи функціональність, однорідність дизайну, гнучкість персоналізації та просто красивий інтерфейс.

Дійсно, з власного досвіду роботи, інтерфейс на Charka «з коробки» дуже гарно виглядає, а стандартна палітра кольорів задовільнить, на мою думку, більшість сучасних проєктів, тоді як, наприклад, прямий конкурент – Material UI має дещо візуально перегружені стандартні компоненти, які підсвідомо хочеться зробити більш мінімалістичними, особливо це стосується стандартних анімацій. До Chakra компонентів, в свою чергу, внесень в стандартну тему можна не робити зовсім, якщо цього не вимагає дизайн застосунку.

Філософія бібліотеки полягає в тому, щоб розглядати кожен видимий елемент як дрібний або середній за своєю суттю React-компонент вузького призначення (атом або молекулу), що може приймати на вхід так звані «style props» (приклади в додатку 4). Ці пропси напряму дублюють стандартні CSS-токени, іноді із навмисно скороченим синтаксисом (напр. padding - p) для зручності. Також, як раніше вже згадувалось, Chakra постачає компоненти з наперед прописаними всередині, детальними, артибутами ARIA, добре обробленою навігацією між функціональними елементами (кнопки, поля вводу, посилання, тощо.) за домогою клавіатури (що є дуже важливим функціоналом для «продвинутих» користувачів) та виразним підсвічуванням актуального фокусу, тобто продукти стилізовані за допомогою Chakra автоматично задовольняють вимоги до accessibility і не змушують окремо виділяти під це розробницькі ресурси пізніше.

Розглянемо детально які інструменти, надані цією бібліотекою, були активно використані для розробки ЛІС, в рамках цієї кваліфікаційної роботи. Найменші

примітиви Chakra – Box, Text, Heading, Flex, Stack, HStack, VStack, Grid, Center і Container. Box – це заміна HTML div, що «розуміє» всі токени відступів, кольорів і розмітки, тому слугує базою для більшості інших компонентів Chakra. Text та Heading обгортають текстову CSS стилізацію для звичайного тексту та заголовків. Flex і Grid реалізують відповідно функціонал flexbox (display: flex) і grid (display: grid). Stack та його горизонтальна (HStack) й вертикальна варіації (VStack) автоматично вставляють інтервали між дочірніми елементами, звільняючи від ручного прописування відступів. Center і Container дають можливості гнучкого налаштування відцентрованої позиції елементів, як локально, всередині певних компонентів, так і на цілій сторінці.

Із цих примітивів складаються багатші компоненти, зокрема Button, Input, Select, Textarea, Modal, Tooltip, Popover, тощо. Кожен із цих компонентів усе ще приймає style props, тому, наприклад, Button можна змінити за розміром, кольором чи ефектом при наведені миші прямо в файлі компонента, де він використовується.

Приклад різної стилізації кнопок, з використанням всього трьох пропсів Chakra: colorScheme, variant та size, наведено на рисунку 2.8.

Рисунок 2.8 – кнопки Chakra UI різних стилів

Кнопок у розробленій ЛІС, – StudioLAB WEB – звісно, чимало, і я був радий працювати з ними, як і з іншими UI елементами, саме інструментами Chakra UI, це був, без перебільшень, проривний для мене досвід.

Input, Select і Textarea – це основні елементи для взаємодії користувача з інтерфейсом, коли виникає потреба введення даних. Input використовується для вводу короткого, однострокового тексту, наприклад, ім’я, email, пароль тощо. Select, на відміну від Input, не дає користувачеві можливості вільного вводу тексту, а пропонує вибір з обмеженого списку варіантів. Це особливо зручно у випадках, коли вибір дійсно обмежений: країни, міста, бажаний метод доставки, точні дані, що не підлягають вигадуванню. Використання Select спрощує заповнення форм і мінімізує ризик помилок, спровокованих некоректним вводом вільних значень там,

де їх просто не може бути. Textarea вирішує іншу задачу– ввід довгого тексту, наприклад, коментарів, скарг або доповнень до введених даних. На відміну від Input, в Textarea користувач має змогу розділити текст на рядки натисканням клавіші Enter, за бажанням розробника, максимальну висоту текстового поля вводу можна обмежити і додати скролл до елементу, таким чином економлячи в інтерфейсі місце, якого, в системах ЛІС, – через величезну кількість задач, що має виконувати система – так часто бракує.

Modal – це потужний інструмент для обмеження концентрації уваги користувача на використаній в момент часу одиниці функціоналу програми. Коли потрібно, щоб користувач не відволікався ні на що інше, і сконцентрувався на певній дії – наприклад, підтвердженні видалення, заповненні важливої форми або перегляді важливої інформації – часто доречним буде використання модального вікна. Воно перекриває фон і диктує чіткий сценарій – завершити дію або, якщо за вимогами проєкту дозволено, закрити модальне вікно. У Chakra UI Modal працює «з коробки» з усіма тонкощами: закриттям по натисканню клавіші Escape, автофокусом на першому інтерактивному елементі модального вікна при його відкритті, для зменшення потреби використання миші та пришвидшення роботи досвідчених користувачів, і плавною анімацією та адаптацією розміру під різні екрани.

Приклад вигляду модального вікна Modal наведено на рисунку 2.9.

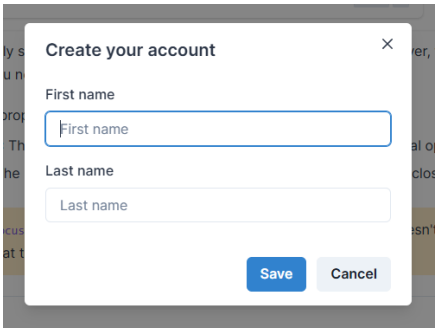


Рисунок 2.9 – компонент Chakra UI Modal

Tooltip – це маленький, але важливий компонент, який допомагає пояснити користувачеві мінімалістичний інтерфейс без зайвого статичного тексту на екрані, що перманентно займав би на ньому місце. Він з’являється лише при наведенні курсора на елемент, якому він назначений. Наприклад, якщо в інтерфейсі є кнопка

лише з іконкою, без напису, у tooltip можна розписати її пояснення її призначення, оскільки, – навіть з вдало обраною іконкою – не всім користувачам воно може бути зрозуміле одразу. Інший популярний варіант використання – відображення повної версії скороченого тексту там, де нам бракувало місця показати його повністю. Таким чином, ми запобігаємо постійним пояснюючим написам, що зробили б інтерфейс важким, проте все ж піклуємось про кінцевого користувача, і лишаємо йому підказки.

Приклад вигляду компонента Tooltip наведено на рисунку 2.10.

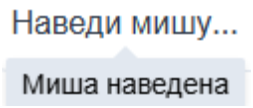


Рисунок 2.10 – Chakra UI Tooltip

Popover – це розширена версія Tooltip та, одночасно, полегшена версія Modal. Якщо tooltip – це лише коротке пояснення, то popover – це справжній міні-Modal: з заголовком, кнопками, текстом, полями вводу, коротше – довільним контентом, не лише текстом. Вимальовується він, за стандартною поведінкою бібліотеки, при натисканні на елемент якому він назначений і дозволяє виконати додаткові, приховані дії, не лишаючи поточної сторінки. Наприклад, показати додаткові опції, поля вводу чи профільну інформацію, недоречну для постійного відображення в інтерфейсі. Popover ідеально підходить для сценаріїв, де Modal – це занадто, а Tooltip – замало. Chakra UI подає його як готове рішення з керованим фокусом, розташуванням, адаптивністю та відмінною інтеграцією в загальний дизайн системи.

Приклад вигляду компонента Popover наведено на рисунку 2.11

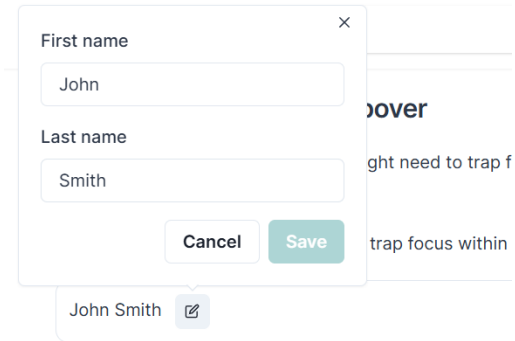


Рисунок 2.11 – Chakra UI Popover

Усі ці компоненти об’єднує одна мета – зробити взаємодію користувача з інтерфейсом простим, зручним та інтуїтивним. Вони не лише естетичні, а й продумані з точки зору функціональності та поведінки в реальних сценаріях використання.

Проте навіть такі, здавалося б ідеальні, компоненти, не завжди здатні задовільнити дизайн-потреби проєкту, на допомогу приходить вбудована темізація, що базується на корегуванні стандартного словника дизайн-токенів Chakra, що охоплює палітру кольорів, шрифти, розміри шрифту та елементів інтерфейсу, відступи, розміри, тіні, тощо. Розробник може розширити або повністю перевизначити будь-який компонент Chakra через використання функції extendTheme, повернений нею об’єкт потім предається у контекст ChakraProvider, що обгортає весь код програми. ChakraProvider – це обгортка над вже нам знайомим, вдубованим в React, Context, а оскільки він огортає весь додаток – кастомні, перевизначені в темі Chakra компоненти, будуть доступні до використання в усіх файлах коду програми. Під час збірки додатку персоналізована тема зливається з темою за замовчуванням, тож навіть сторонні компоненти можуть читати нові токени.

ChakraProvider – не єдина обгортка вже знайомого нам стандартного функціоналу React. Chakra також надає власні хуки, для спрощення певних операцій, зав’язаних на поточному стані динамічних характеристик інтерфейсу користувача, ось декілька прикладів [16]:

1. useBreakpointValue – дозволяє вказати, яке значення присвоїти змінній відповідно до ширини екрану в даний момент часу; динамічно реагує на зміну розміру вікна у браузері; повертає змінну, якій буде присвоєно відповідне значення. Типовий приклад використання- призначення різного розміру заголовків для мобільних пристроїв та стаціонарних комп’ютерів.
2. useColorModeValue – схожий за принципом на попередній хук, проте спирається на актуальну кольорову тему, а не на розмір екрану. Приймає два значення: для присвоєння змінній, що він повертає, коли тема світла,

та, відповідно, темна. Хоча в більшості випадків компоненти адаптуються самі, при використанні нестандартних кольорів, для потреб проєкту, необхідно вручну вказати відповідний колір для темної теми, якщо він має відрізнятись.

3. useClipboard – дає змогу програмно, по кліку на текст, кнопку, тощо, копіювати текст з екрану, або запрограмований в коді текст. Прибирає потребу вибирати текст, що користувач бажає скопіювати за допомогою миші чи комбінацій клавіш. В ЛІС копіювати дані з одного місця в інше – часта потреба, тому цей хук є дуже доречним не лише в теорії, а й на практиці.

Поговоримо про те, як Chakra працює під капотом, адже, згадуємо – браузер на розуміє бібліотек типу React та, власне, Chakra UI. Він знає лише HTML, CSS та JavaScript. Отже, як ці лаконічні, прості у використанні компоненти, стають зрозумілим для браузера CSS? Двигун Chakra перетворює style props на атомарні класи за допомогою алгоритму хешування Panda та вносить ці класи в єдиний стильовий файл, який підвантажується при запуску додатку – відкритті сайту в браузері. Коли компонент з певними style props використовується повторно, Chakra не створює нові класи щоразу, а повторно використовує вже згенеровані раніше. Це дозволяє мінімізувати обсяг стилів і покращити продуктивність. Крім того, завдяки атомарності, підхід дозволяє максимально ефективно комбінувати стилі між різними компонентами, уникаючи конфліктів і дублювання. Стилі створюються динамічно, але з урахуванням попередньо визначених темою значень, що гарантує узгоджений дизайн по всьому сайту. У підсумку, навіть складні інтерфейси з великою кількістю стилізованих компонентів залишаються легкими та швидкими для рендерингу у браузері, завдяки тому, як всередині реалізована Chakra.

У порівнянні з прямим конкурентом – Material UI Chakra обмінює величезний каталог компонентів на загальну меншу вагу вихідного коду та модель «token-first». Chakra прагне бути «достатньо нестилізованим», щоб брендинг був тривіальним, але «достатньо стилізованим», щоб сторінка виглядала пристойно без дотику до

теми. На відміну від Tailwind CSS, стильові токени розміщені в пропсах, а не в класах, тож доводиться писати менше тексту, досягаючи того ж результату, а типи TypeScript, які з коробки підтримуються в Chakra, гарантують наявність токена та валідність переданого в нього значення.

Chakra UI на 100% задовольняє дизайн-потреби розробляємої лабораторної системи – StudioLAB WEB, через своє відмінне графічне оформлення «з коробки», без обов'язкової потреби додаткових дотиків до теми, та широкі функціональні можливості її стандартної бібліотеки компонентів.

2.4 Redux Toolkit

Redux Toolkit (RTK) – це найпопулярніше рішення для оперування глобальним станом в React проєктах на сьогоднішній день. RTK надає набір верхньорівневих функцій, які дозволяють керувати розрізненим по усіх компонентах додатку станом на основі чітких, перевірених шаблонів.

Розберемось в основних сутностях бібліотеки, почнемо з так званого «slice» або зрізу. Зріз інкапсулює в собі пов'язану смисловим навантаженням частину глобального стану (напр. дані авторизованого кристувача) та логіку, що нею керує. Slice визначає початковий стан, набір «reducers» (редьюсерів) – функцій, що дають можливість змінювати дані зберігаємого зрізом стану та автоматично створює відповідні action-и. Redux Toolkit дозволяє створювати зрізи за допомогою функції createSlice, що спрощує структурування логіки за окремими сутностями або модулями додатка.

Приклад створення зрізу простого лічильника наведено на рисунку 2.12.

```
const counterSlice = createSlice({
  name: 'counter',
  initialState: { count: 0 },
  reducers: {
    // Reducer: збільшення значення
    increment: (state) => { state.count += 1 },
    // Reducer: зменшення значення
    decrement: (state) => { state.count -= 1 },
    // Reducer: приймає action з довільним числом та додає його до count
    addAmount: (state, action: PayloadAction<number>) => {
      state.count += action.payload
    },
  },
})
// Експортуємо action-и, які автоматично створюються
export const { increment, decrement, addAmount } = counterSlice.actions
// Експортуємо reducer (його треба додати в store)
export default counterSlice.reducer
```

Рисунок 2.12 – Код зрізу лічильника

action (дія) – це звичайний JS об'єкт, що передається для відповідної зміни стану в reducer. Вище, на рисунку 2.11, третій reducer приймає дію, «payload» якої є числом. Результатом відпрацювання редьюсера addAmount буде актуальне значення змінної стану count збільшене на число з action.payload. До речі, знання перекладу слова «payload» з англійської, – корисний вантаж– також допомагає зрозуміти суть об’єкту action, він потрібен редьюсеру аби спроектувати дані з «вантажу» переданого в дії на актуальний стан зрізу, за прописаною в ньому логікою.

Примітка: в RTK є певна заплутаність з найменуванням сутностей, по суті, об’єкт reducers описаний у функції createSlice мав би називатись actions, а замість параметру action вони мали б приймати, наприклад, напряму payload, лишаючи назву reducer для об’єкту createSlice.reducer, що потім експортується в сховище глобального стану (детальніше нижче по тексту), з метою «повідомити» усім файлам додатку про існування і доступність нового зрізу, та дати змогу користуватись як його даними, так і діями, але ми ці умовності упустимо, в рамках розробки проєкту цієї кваліфікаційної роботи проблема найменування не принципова. Далі, для спрощення сприйняття матеріалу, в контексті RTK я буду оперувати чіткими, знайомими та більш зрозумілими, проте ширшими, поняттями – такими як: функція, параметри, об’єкт, тощо.

store – центральне сховище глобального стану додатка. У ньому зберігається єдиний об’єкт, що відображає стан, до якого мають доступ усі React-компоненти. Store забезпечує методи для зчитування поточного стану, надсилання дій (actions)

та підписки на зміни стану. У Redux Toolkit store створюється за допомогою функції configureStore, в яку додаються усі зрізи проєкту.

Приклад створення сховища наведена на рисунку 2.13.

```
import counterReducer from '../features/counter/counterSlice'
export const store = configureStore({
  reducer: {
    // Додаємо наш редьюсер до кореневого редьюсера
    counter: counterReducer,
  },
})
```

Рисунок 2.13 – Створення RTK store

Store, з усіма своїми зрізами огортає весь код React-додатку, методом передавання об’єкту store в RTK Provider.

Розберемось, як взаємодіяти з наданими інструментами в компонентах. Redux Tookit, як і попередньо розглянута технологія – Chakra UI, надає свої хуки: useAppSelector та useAppDispatch. useAppSelector – хук, що дає доступ, повертає, дані зі сховища методом звернення «state.slice.value», де state – увесь глобальний стан додатку, slice – назва зрізу, до даних якого нам треба звернутись, а value – назва змінних у зрізі, що зберігає потрібні нам дані. useAppDispatch – це диспетчер. Він повертає нам функцію dispatch, яка в свою чергу приймає параметром будь яку функцію імпортовану з будь якого зрізу, з відповідними параметрами до неї. По суті диспетчер просто приймає запит на дію, і за кулісами відбувається відповідне оновлення стану [17].

Як і зі звичайним локальним станом, компонент «підписаний» на зміну стану через useAppSelector буде перемальовано у разі оновлення стану через useAppDispatch, але тільки якщо було оновлено одне з полів зрізу, на яке компонент фактично підписаний. Це вигідно відрізняє зрізи RTK від використання, наприклад, вбудованого у React Context, оскільки компоненти підписані на будь-яку дану з взятого контексту будуть перемальовані при зміні будь-якої (навіть іншої) даної контексту незалежно, чи змінились цікаві для них дані. Це, очевидно, покращує продуктивність додатку, оскільки зайві ре-рендери – найдорожча для комп’ютера марна операція.

На рисунку 2.14 наведено приклад компонента, що використовує розглянутий вище зріз.

```
import { increment, decrement, addAmount } from './counterSlice'
const CounterComponent = () => {
  const count = useAppSelector((state) => state.counter.count)
  const dispatch = useAppDispatch()
  return (
    <div>
      <h2>Лічильник: {count}</h2>
      <button onClick={() => dispatch(increment())}>Збільшити</button>
      <button onClick={() => dispatch(decrement())}>Зменшити</button>
      <button onClick={() => dispatch(addAmount(4))}>Збільшити на 4</button>
    </div>
  )
}
```

Рисунок 2.14 – Компонент, що використовує counterSlice

При будь-якому виклику dispatch, усі компоненти додатку, наскільки б вони не були далеко розкидані по інтерфейсу, будуть перемальовані, показуючи актуальні дані, повернені useAppSelector.

Ще одна дуже цінна здатність RTK – зручне управління асинхронними операціями, такими як, наприклад, отримання даних з API. Уявимо ситуацію: маємо сторінку, з великою кількістю компонентів (карток, віджетів, таблиць, графіків, тощо), зав’язаних на якійсь одній сутності (наприклад звіті по медичним аналізам), – питання – що показувати на екрані, при переході на сторінку? Адже даних, для відображення їх на екран, ми ще не маємо – вони будуть миттєво запитані з бекенду при відкритті сторінки, але відповідь від сервера доведеться почекати. Хорошою практикою в сучасному дизайні інтерфейсів є показувати індикацію завантаження роздільно для всіх компонентів інтерфейсу, замість завантаження цілої сторінки перед її відкриттям, а ще краще – так званий «skeleton loading», що схематично відображає, як будуть розташовані дані на екрані, коли вони все ж надійдуть.

Приклад вигляду skeleton loading наведено на рисунку 2.15.

Рисунок 2.15 – Skeleton loading

Окрім стану завантаження, у разі помилки відповіді з сервера, бажано так само показати якесь повідомлення про помилку, чи іконку помилки, на всіх

зацікавлених компонентах, аби точно дати користувачеві чітке розуміння, які саме дані або частина інтерфейсу наразі втрачено через помилкове відпрацювання сервісу, це і UI/UX практика добра, і спілкування користувача зі службою підтримки, якщо така є, значно полегшить. То як опрацювати всі ці сценарії, для великої кількості компонентів? На допомогу приходить функція createAsyncThunk та об'єкт додаткових редьюсерів зрізу – extraReducers.

У зрізі створюються 3 поля: status – для зберігання актуального стану запиту (за замовчуванням «idle» – бездіяльний), error – для збереження даних про помилку (null за замовчуванням) та data, куди ми помістимо дані у разі позитивної відповіді сервера. У файлі зрізу, але окремо від самого зрізу, додається функція createAsyncThunk, в середині якої відбувається власне сам запит до API, що може знаходитись в одному з трьох можливих станів у взятий момент часу: pending, rejected, resolved – у процесі, відхилено, задовільнено – відповідно. createAsyncThunk повертає action подібний до тих, що генерували звичайні редьюсери зрізу, розглянуті раніше – він аналогічно викликається через диспетчер, наприклад dispatch(fetchUserData()).

Для обробки станів запиту, виконаного за допомогою цього інструменту, використовуються додаткові редьюсери, логіка наступна: в додатковому редьюсері, до відповідного createAsyncThunk додаємо обробники на три випадки: завантаження, що має змінити status з idle на loading, а також помістити в error та data значення null (нічого), оскільки виклик цього обробника означає, що запит було зроблено заново, і якщо це не перший його виклик, тобто вже маємо якісь попередні дані, їх треба перевести наново в їх початковий стан; потім маємо обробник успішного запиту, що має помістити отриману відповідь в поле data та status = success і, нарешті, обробник помилки, що має покласти об'єкт, описуючий помилку, для відображення даних про неї на UI, в поле error, а статусу назначити відповідне дієслово в минулому часі, зазвичай «rejected».

Приклад обробки подій createAsyncThunk наведено на рисунку 2.16, без його об'яви, вона для розуміння теми не є необхідною.

```
extraReducers: (builder) => {
  builder
    .addCase(fetchUser.pending, (state) => {
      state.status = 'loading'
      state.error = null
    })
    .addCase(fetchUser.fulfilled, (state, action) => {
      state.status = 'success'
      state.user = action.payload
    })
    .addCase(fetchUser.rejected, (state, action) => {
      state.status = 'rejected'
      state.error = action.error.message
    })
},
```

Рисунок 2.16 –Обробник станів запиту createAsyncThunk

Підсумовуючи тему обробки асинхронних подій, RTK надає розробникам інструмент, для гнучкої адаптації інтерфейсу під будь-яку стадію отримання даних, будь-то очікування, успіх їх отримання, або помилка від серверу. І все це, як React і пропагує в декларативному стилі, без необхідності знати внутрішню складову.

Як інструмент для операцій над глобальним, поділеним між багатьма компонентами інтерфейсу інформативної та сучасної ЛІС, краще ніж Redux Toolkit – рішення, на сьогоднішній день, не існує, саме тому ця бібліотека – мій безумовний вибір для закриття відведеної їй зоні відповідальності.

3 РОЗРОБКА СИСТЕМИ

На основі обраних методом порівняльного аналізу та глибоко вивчених технологій, а також на базі досліджених вимог до систем типу ЛІС у науковій літературі, було розроблено веб-додаток StudioLAB WEB. Основні встановлені функціональні вимоги до проєкту були такі [18]:

1. Автентифікація користувача.
2. Наявність інформаційної панелі.
3. Пошук та створення особистих карток пацієнтів, редагування даних пацієнтів.
4. Створення запитів на аналізи, перегляд списку та окремих запитів, можливість швидко відкрити конкретний запит.
5. Додавання до запиту аналізів та додатків.
6. Генерація PDF звітів на основі закінчених запитів, можливість перегляду списку та окремих запитів.
7. Налаштування користувача.

Примітки:

1. Система передбачає зберігання та отримання даних, але це поза зоною відповідальності бібліотеки React та її оточення, тому, для цілей демонстрації розробленого ПЗ, робота з даними була симульована комбінованим методом звернення до самописного API на C# .NET та, місцями, показу імітаційних даних.
2. Для оброки навігації між сторінками використано стандартне рішення для проєктів на React – бібліотека React Router.

3.1 Автентифікація користувача

Безумовно критично важливий функціонал, адже до програми такого типу, що обробляє чутливі медичні дані пацієнтів медичної установи, очевидно, доступ

будь-хто мати не може. Потрібно було забезпечити безпечний доступ, наданий лише допущеним користувачам, тому додаток зустрічає користувача сторінкою входу, де потребується введення імені користувача та паролю. Форма вводу, до речі, як і інші форми в додатку обробляється бібліотекою Formik, а для валідації введених значень її підтримала бібліотека yup. Це поєднання забезпечило гнучку та дійсно зручну обробку форм вводу, яких у розробленій системі чимало. У регульованих середовищах автентифікація користувачів також є вимогою для відповідності законам про захист персональних даних, таким як HIPAA або GDPR. Наявність перемикача видимості пароля підвищує зручність вводу без шкоди для безпеки.

У відповідь на успішний запит на автентифікацію система отримує та зберігає JWT токен автентифікації та дату закінчення терміну його дії. Якщо термін дії токена сплив або його було підмінено на невалідний – користувача буде повернуто на сторінку входу для проведення повторної автентифікації. Для зберігання цих даних у проєкті було використано так зване «локальне сховище» (localStorage) – механізм, що є в усіх сучасних браузерах. Спосіб не найнебезпечніший, проте не потребував додаткових компетенцій, поза межами обраного технологічного стеку, маючи повноцінного Back-End розробника на підтримку – автентифікацію я реалізував би через безпечніший механізм – cookies, проте для потреб демонстрації функціоналу StudioLAB WEB – цілком достатньо й фактично реалізованого варіанту.

3.2 Інформаційна панель

Перша сторінка, що зустрічає користувача після успішної автентифікації – інформаційна панель. Що підсвічує основні кількісні показники роботи лабораторії за певний період часу, останні сповіщення про проблемні аналізи та дані про сам застосунок, такі як його версія, версія використаного браузера, тощо.

Відображені на панелі кількісні показники на віджетах в самому верху

сторінки наступні: сумарна кількість отриманих зразків, підписаних та непідписаних звітів, а також незавершених звітів, тобто запитів у роботі.

Нижче маємо наступні 3 віджети з графіками, реалізованими засобами бібліотеки recharts:

1. Запити – відображає динаміку замовлених, взятих у роботу та завершених графік за робочий тиждень.
2. Зразки – відображає загальну кількість взятих зразків та зразків в опрацюванні у певний момент часу.
3. Звіти – показує кількість підписаних та непідписаних запитів у момент часу.

Базово показуються лише вимальовані графіки, а при наведені на графік мишею – детальна інформація у тій точці.

Під графіком маємо наступні дві картки: сповіщення про проблеми та дані застосунку. Таблиця сповіщень показує наступну інформацію: до якого звіту відноситься сповіщення, ким і коли воно було залишене, а також – саме повідомлення, з опціональним відображенням критичного статусу сповіщення, і, нарешті, статус сповіщення, що дає уявлення про нинішній етап його оброки. Карта ж інформації застосунку, показує версію самого застосунку, версію використаного браузера, і потенційно може показувати будь-які інші дані, такі як версію бек-енду, бази даних, кількість підключених користувачів, тощо.

Ще один важливий елемент інтерфейсу – ліва бокова панель, з неї маємо доступ до ключових секцій системи:

1. Головна сторінка – інформаційна панель.
2. Секція запитів – у ній маємо наступні 3 віджети: новий запит, що переносить нас на сторінку ідентифікації пацієнта, список запитів, що відкриває список усіх створених запитів з гнучкою фільтрацією, та віджет швидкого доступу до запиту, що відкриває модальне вікно з полем вводу номера запиту. Останні два є можливість швидко відкривати через ГК f6 та shift + f6 відповідно.

3. Секція звітів – на даній стадії розвитку проєкту пропонує лише один віджет – список існуючих звітів.
4. Сторінка налаштувань користувача.

3.3 Ідентифікація пацієнта та створення запиту

Пошук вже існуючого в базі пацієнта або додавання нового, у разі його відсутності та створення для нього нового запиту на аналізи – об’єднано спільним сценарієм користувача. Перейдемо в секцію запитів та натиснемо «новий запит» – відкриється сторінка ідентифікації пацієнта з формою вводу імені та прізвища для пошуку, якщо пацієнта не знайдено, або за введеними даними знайдено один єдиний збіг – переходимо на сторінку картки пацієнта, нового або існуючого відповідно. Якщо ж збігів більше – відкривається модальне вікно з таблицею знайдених збігів, та кнопками «закрити» (ГК f4) та «створити нового (пацієнта)» (ГК Esc), натиснути на стрічку потрібного пацієнта аби перейти до його картки.

Примітки:

1. Поля вводу прізвища на імені на цій сторінці – кастомні компоненти, так звані SearchableSelect, вони допускають вільний ввід, а також пошук по вже знайомим системі даним, поєднуючи звичайний Input та Select. Таким чином, за вводом «YER» можна знайти, наприклад, прізвище «YERSHOV». Для розробки цього компоненту активно були використані хуки useEffect, useState та useRef, а також засоби бібліотеки formik та велика кількість пропсів для подальшої гнучкої кастомізації поведінки компонента.
2. Таблиця, у якій показуються знайдені збіги, створена засобами бібліотеки kendo react grid, як і таблиці списків запитів та звітів. Бібліотека «з коробки» надає пагінацію та інші важливі для складних таблиць інструменти, такі як гнучка зміна ширини колонок, чи сортування за зростанням/спаданням по даним у певній колонці.

3. У цій самій таблиці було оброблено швидку навігацію з клавіатури: клавішами-стрілками вверх/вниз змінюється обраний пацієнт, натиснути Enter аби перейти на сторінку його картки, плюс ГК згадані вище. Застосовані інструменти: хук React useEffect, що при першому рендері таблиці створює слухач події натискання клавіш та назначає різним клавішам відповідні обробники.

Картка пацієнта, до якої ми переходимо наступним кроком, дає можливість візуалізувати, та змінити дані пацієнта. Єдина важлива зміна, навмисне упущена в проєкті – ідентифікаційний код, оскільки він потребував би великої кількості додаткових контролів та обрахувань, перша версія додатку була розроблена без його урахування. Коли оператор системи переконався, що дані пацієнта коректні, можна натискати «Створити запит» (ГК alt + a), запит буде додано, і оператора буде переведено на сторінку новоствореного запиту.

3.4 Сторінка запиту

Сторінка запиту у StuidoLAB WEB надає усі найнеобхідніші функціональні можливості – верхня її частина поділена на дві картки – інформація про сам запит, така як, наприклад, його унікальний номер та стан обробки, та дані пацієнта, за яким цей запит закріплено, такі як повне ім’я та вік. Нижче розмістились функціональні компоненти сторінки: секція дій, секція аналізів та секція додаткової інформації.

У секції дій маємо наступні 3 кнопки:

1. «Додати аналіз» – при натисканні фокусує поле пошуку над таблицею аналізів, яка у новому запиті завжди буде пустою. Шукати потрібний аналіз можна за його унікальним номером та назвою, якщо збігів за введеним пошуковим запитом знайдено більше ніж один – відкриється модальне вікно, подібне до того, що було реалізовано для відображення знайдених пацієнтів на кроці

ідентифікації, вікно також підтримує ті самі ГК, за деяким виключенням.

2. «Додатки до запиту» – відкриває модальне вікно, що дає можливість додати файли, такі як: PDF, docx (word), xlsx (excel) та картинки до запиту, надати їм інформативний опис, а згодом – візуалізувати їх, завантажити або роздрукувати.
3. «Новий запит» та «Видалити запит» – додаткового пояснення не потребують, при створенні нового запиту увесь попередній сценарій користувача починається з першого кроку, а при видаленні – оператора буде перенесено на сторінку всіх запитів, про яку ми поговоримо пізніше.

Таблиця аналізів, як вже зрозуміло, дає можливість шукати та додавати відповідні аналізи, згодом – чітко розуміти які аналізи потрібні конкретному пацієнту за запитом. Додатково в таблиці було збережено гарну UX практику – можливість відмінити новододані аналізи відповідною кнопкою. Додані аналізи підсвічуються в таблиці іншим кольором, відносно тих, що вже були частиною запиту, і до того, як буде натиснута кнопка «зберегти», вони не будуть фактично додані до запиту у базі.

Додатково була реалізована можливість видалення аналізів з запиту, у крайньому лівому стовпчику таблиці прапорцями модна обрати аналізи, що мають бути видалені, та видалити їх відповідною кнопкою. Видалення потребує додаткового підтвердження наміру у блокуючому спливаючому вікні. До речі, усі спливаючі вікна, як і графічну складову інтерфейсу програми в цілому, було реалізовано засобами бібліотеки Chakra UI.

Секція додаткової інформації показує, чи запит є терміновим до виконання, а також дає змогу додати до запиту додаткові коментарі та конкретизувати потребу виконання запиту.

Зверніть увагу, як дані запиту розкидано між різними компонентами на екрані, але отримуються вони лише один раз на верхньому рівні – при завантаженні сторінки, по суті на рівні «обгортки» всіх цих віджетів. Для постачання їм усім

необхідної частини даних та певного іншого функціоналу додатку, було використано Redux Toolkit.

3.5 Сторінки списку запитів на звітів

Сторінки списків виконано за схожим патерном: зверху маємо фільтри та кнопку «пошук», у таблиці під ними – відфільтрований список звітів або запитів, відповідно. Клік по самій лівий клітинці таблиці переносить на сторінку звіту чи запиту, відповідно до сутності показаних даних. Оскільки структура сторінок повторюється, активно були перевикористані компоненти, слідуючи правильним підходам React та Atomic Design. Поля вводу фільтрів мають особливу стилізацію, що відрізняє їх від усіх інших полів вводу у програмі, і одразу дає користувачеві чітко зрозуміти, що перед ним на екрані саме фільтри. Окремий варіант для фільтрів було прописано на рівні теми Chakra UI, що позбавило мене необхідності стилізувати кожне поле окремо.

3.6 Генерація звітів та сторінка звіту

Функціонал генерації pdf файлу на основі даних звіту по проведеним аналізам мав би бути розроблений на бек-енді, тому був свідомо пропущений і імітований для демонстраційних цілей, як і функціонал додавання звітів. На сторінці деталей звіту є візуалізація pdf (наразі це статичний файл), а також основні дані звіту: код звіту, код відповідного запиту, нотатки, чи є критичні результати аналізів, а також зручна навігація між звітами прямо з цієї сторінка, відповідними кнопками навігації або слайдером. Додатково реєструється факт перегляду звіту для зменшення кількості випадків службової недбалості, як тільки звіт було відкрито вперше – реєструється ім’я користувача, дата, та час перегляду.

Ще один важливий для ЛІС функціонал, проте ще не доданий на даній стадії проєкту – електронний підпис звітів. Він потребував би вагомої логіки на бек-енді та сторонньої інтеграції з ПЗ для електронного підпису pdf. Наразі додаток

дозволив би лише фізичний підпис після роздрукування паперової версії, за умови додавання генерації справжніх pdf файлів.

3.7 Сторінка налаштувань

На цій сторінці користувач може змінити свої дані, колір теми (світла/темна) та вийти з системи. Також користувач може чітко побачити свої права, адже в ЛІС важливо підтримувати варіативні права доступу – не всім користувачам повинен бути доступний увесь функціонал системи.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи бакалавра було проведено дослідження теми розробки інформаційної системи для медичних лабораторій на основі технології React.js з метою розробки функціонального прототипу такої системи.

Результатом дослідження став практичний результат – розроблена програма StudioLAB WEB, з реалізованим основним функціоналом ЛІС: захист системи автентифікацією користувачів, повний цикл додавання та модифікації запитів, від додавання в систему пацієнта та специфікації необхідних йому аналізів до можливості повного видалення запиту, візуалізація звітів (поки демонстраційна, але все ж функціональний каркас майбутньої повноцінної системи) та базових налаштувань користувача, плюс приємне доповнення у вигляді підтримки темної теми, що є гарною практикою у сучасному UI/UX.

Для вибору доречних інструментів розробки системи для медичної лабораторії було проаналізовано сучасні технології фронт-енд розробки та популярні вимоги до систем такого типу, методом вивченням наукових джерел та інтернет-ресурсів технічного та медичного направлення. Встановлено, що найдоречнішими технологіями для реалізації поставлених цілей кваліфікаційної роботи є:

1. Бібліотека React.js – як сучасний, надійний і продуктивний каркас великого веб-застосунку.
2. TypeScript – мова програмування що значно покращила DX у процесі розробки і дозволила завершити проєкт у строк.
3. Chakra UI – бібліотека відповідальна за сучасний дизайн розробленого ПЗ, його однорідність та «state of art» user experience.
4. Redux Toolkit – за можливість зручно надавати спільний доступ до даних різним компонентам, розкиданих по різним секціям сторінки.

На основі обраного стеку та дрібних допоміжних технологій і було розроблено систему, що свідчить про можливість та, що більш важливо, доречність застосування React.js та суміжних із ним технологій для розробки лабораторних інформаційних систем.

Лишився в проєкті і простір для розширення – додаванням сторонніх інтеграцій та розробкою спеціалізованого бек-енд сервісу, можна розробити фунціонал генерації файлів звіту, їх електронного підпису та, наприклад, сигналізації аномалій в аналізах. Потенційно можна ще реалізувати повідомлення для лікарів та внутрішній чат між співробітниками лабораторії, аби зменшити потребу у використанні сторонніх програм у лабораторії, що позитивно впливатиме на робочі процеси та зручність роботи в цілому, коли все максимально інтегровано в одне ПЗ, а також підвищить безпечність, адже будь-яка додаткова програма – потенційна додаткова вразливість.

Отже, поставлену мету роботи досягнуто, основні завдання виконано, а шляхи подальшого розвитку системи окреслено, що підтверджує практичну значущість проведеного дослідження. Крім того, враховуючи, що StudioLAB WEB є першою лабораторно-інформаційною системою, розробленою з використанням технології React, наукова новизна роботи також підтверджується.

ПЕРЕЛІК ПОСИЛАНЬ

1. delo.ua, Досліджувати, щоб перемагати: ринок приватної лабораторної медицини стабілізується // [Електронний ресурс]: https://delo.ua/business/doslidzuvati-shhob-peremagati-rinok-privatnoyi-laboratornoyi-medicini-ukrayini-stabilizujetsya-414007/, Рік публікації: 2023
2. Hickman P.E., Interpretative commenting in clinical chemistry, Clinical Biochemist Reviews. // [Наукова стаття]: https://www.sciencedirect.com/science/article/pii/S2352551721000433, Рік видання: 2010
3. Jayaraman K.D., Kumar A., Optimizing Single-Page Applications Through Angular Framework Innovations, International Journal of Research in Modern Engineering and Emerging Technology. // [Наукове дослідження]: https://www.researchgate.net/publication/390200750_Optimizing_Single_Page_Applications_SPA_Through_Angular_Framework_Innovations, Рік видання: 2024
4. Rajala O., Impact of React Component Libraries on Developer Experience – An Empirical Study on Styling Approaches // [Наукове дослідження]: https://www.researchgate.net/publication/383941317_Impact_of_React_component_libraries_on_developer_experience_-_An_empirical_study_on_component_libraries%27_styling_approaches, Рік видання: 2024
5. Adhithi Ravichandran, React Virtual DOM Explained in Simple English // [Електронний ресурс]: https://adhithiravi.medium.com/react-virtual-dom-explained-in-simple-english-fc2d0b277bc5, Рік публікації: 2019
6. react.dev, Writing Markup with JSX // [Електронний ресурс]: https://react.dev/learn/writing-markup-with-jsx, Дата звернення: 02.04.2025
7. Manuel Grullon, Blurred Lines: Is Ruby an interpreted language and what does that even mean? // [Електронний ресурс]: https://medium.com/@astermanuelg/blurred-lines-is-ruby-an-interpreted-language-2d3d6bca3d37, Рік публікації: 2018

8. Gonçalves de Oliveira B., Endo A.T., Vergilio S.R., Characterizing Security-Related Commits of JavaScript Engines // [Наукова стаття]: https://www.scitepress.org/Papers/2023/119661/119661.pdf, Рік видання: 2023
9. Saptarshi Bhattacharyya, Application of TypeScript Language: A Brief Overview // [Наукова стаття]: https://www.researchgate.net/publication/304653125_Application_of_TypeScript_Language_A_Brief_Overview, Рік видання: 2007
10. developer.mozilla.org, ARIA // [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA, Дата звернення: 02.04.2025
11. Madeline Corman, Introduction to Redux // [Електронний ресурс]: https://medium.com/@madelinecorman/an-introduction-to-redux-359baccde63d, Рік публікації: 2020
12. Veeranjaneyulu Veeri, State Management in React: Redux vs Zustand - A Comprehensive Guide // [Наукова стаття]: https://www.researchgate.net/publication/385694701_State_Management_in_React_Redux_vs_Zustand_-_A_Comprehensive_Guide, Рік видання: 2024
13. Remon Kelada, Atomic Design: Revolutionizing Frontend Engineering // [Електронний ресурс]: https://www.linkedin.com/pulse/atomic-design-revolutionizing-frontend-engineering-remon-botros-rykfc/, Рік публікації: 2023
14. Ram Krishna, Digging deeper inside the Reconciliation algorithm of react // [Електронний ресурс]: https://medium.com/@coolram2104/digging-deeper-inside-the-reconciliation-algorithm-of-react-f0d428ba4ae9, Рік публікації: 2018
15. Phani Sekhar Emmanni, The Role of TypeScript in Enhancing Development with Modern JavaScript Frameworks // [Наукова стаття]: https://www.researchgate.net/publication/380361058_The_Role_of_TypeScript_in_Enhancing_Development_with_Modern_JavaScript_Frameworks, Рік видання: 2021
16. v2-chakra-ui.com, Create accessible React apps with speed // [Електронний ресурс]: https://v2.chakra-ui.com/, Дата звернення: 21.04.2025

17. Narender Reddy Karka, State Management in Large-Scale React Applications: A Comprehensive Analysis // [Наукова стаття]: https://www.researchgate.net/publication/389660738_State_Management_in_Large-Scale_React_Applications_A_Comprehensive_Analysis, Рік видання: 2025
18. Vera Lukić, Laboratory Information System – Where are we Today? // [Наукова стаття]: https://pmc.ncbi.nlm.nih.gov/articles/PMC6287214/, Рік видання: 2017

ДОДАТКИ

Додаток 1

Imperative (JavaScript)

```
<p id="count">0</p>
<button id="btn">Increase</button>

<script>
let count = 0; // ініціалізувати змінну лічильника
document.getElementById('btn') // знайти елемент кнопки в DOM по id
  .addEventListener('click', () => { // "навісити" на кнопку обробник кліків
    count++; // збільшити значення лічильника на 1
    // знайти елемент, що показує значення лічильника в DOM по його id,
    // та змінити в ньому відображаємий текст
    document.getElementById('count').innerText = count;
  });
</script>
```

Declarative (React)

```
function Counter() {
  // ініціалізація змінної стану, отримання її set функції для зміни значення
  const [count, setCount] = React.useState(0);

  return (
    <div>
      {/* count оновлюється автоматично при його зміні викликом функції setCount */}
      <p>{count}</p>
      { /* вказуємо кнопці збільшувати значення по кліку */ }
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

Додаток 2

JavaScript function

```
function printUserInfo(user) {
  console.log("Name:", user.name);
  console.log("Age:",  user.age);
}

const user = {
  name: "Olena",
  age: 25,
  isAdmin: true,
};

printUserInfo(user); // працює, навіть якщо є зайві або некоректні поля

// А ось помилка — але JavaScript в IDE нічого не скаже,
// помилка себе проявить вже на запущеному проекті
printUserInfo({ name: "Andrii" }); // undefined для age
```

TypeScript function

```typescript
interface User {
  name: string;
  age: number;
}

function printUserInfo(user: User): void {
  console.log("Name:", user.name);
  console.log("Age:",  user.age);
}

const user: User = {
  name: "Olena",
  age: 25,
  // isAdmin: true // ❌ Помилка: зайва властивість
};

printUserInfo(user); // ✅ все добре

// printUserInfo({ name: "Andrii" }); // ❌ Помилка: відсутнє поле age
```

JavaScript class

```javascript
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    return `Hi, I'm ${this.name}`;
  }
}

const person = new Person("Sofiia", 30);
console.log(person.greet());
```

TypeScript class з типами та модифікаторами доступу

```typescript
class Person {
  private name: string;
  private age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public greet(): string {
    return `Hi, I'm ${this.name}`;
  }
}

const person = new Person("Sofiia", 30);
console.log(person.greet());

// console.log(person.name); // ❌ Помилка: name — приватне
```

Додаток 3

HTML + CSS

```
<!-- окремий файл index.html -->
<div class="card">
  <h1>Welcome</h1>
  <button>Click Me</button>
</div>

<!-- окремий файл styles.css -->
.card {
  width: 300px;
  margin: 100px auto;
  padding: 20px;
  text-align: center;
  border: 1px solid #ccc;
  border-radius: 8px;
}

.card button {
  padding: 8px 16px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 4px;
}
```

Tailwind CSS

```
<div class="w-72 mx-auto mt-24 p-5 text-center border border-gray-300 rounded-lg">
  <h1 class="text-xl font-bold mb-4">Welcome</h1>
  <button class="px-4 py-2 bg-blue-500 text-white rounded">Click Me</button>
</div>
```

Chakra UI

```
import { Box, Button, Heading } from '@chakra-ui/react';

function Card() {
  return (
    <Box
      w="300px"
      m="100px auto"
      p="20px"
      textAlign="center"
      border="1px solid #ccc"
      borderRadius="8px"
    >
      <Heading size="md" mb={4}>Welcome</Heading>
      <Button colorScheme="blue">Click Me</Button>
    </Box>
  );
}
```

Додаток 4

Вже реалізований атом кнопки з бібліотеки Chakra UI

```
import { Button } from "@chakra-ui/react";
```

Молекула SearchField

```
// МОЛЕКУЛА SearchField
import { useState } from "react";
import {
  Input,
  InputGroup,
  InputLeftElement,
  IconButton,
} from "@chakra-ui/react";
import { SearchIcon } from "@chakra-ui/icons";
import Button from "../atoms/Button";

export default function SearchField({ onSearch }) {
  const [value, setValue] = useState("");

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        onSearch(value);
      }}
    >
      <InputGroup maxW="400px">
        <InputLeftElement pointerEvents="none">
          <SearchIcon color="gray.400" />
        </InputLeftElement>

        <Input
          value={value}
          onChange={(e) => setValue(e.target.value)}
          placeholder="Введіть запит…"
        />

        {/* можна замість окремої кнопки використати IconButton */}
        <IconButton
          aria-label="Пошук"
          icon={<SearchIcon />}
          type="submit"
          ml={2}
          colorScheme="blue"
        />
        {/* або:
        <Button type="submit" ml={2}>Пошук</Button>
        */}
      </InputGroup>
    </form>
  );
}
```

Молекула SearchResults

```
// МОЛЕКУЛА SearchResults
// ...imports
export default function SearchResults({ results }) {
  if (!results?.length) {
    return (
      <Box mt={4}>
        <Text color="gray.500">Нічого не знайдено.</Text>
      </Box>
    );
  }

  return (
    <SimpleGrid columns={[1, 2, 3]} spacing={6} mt={4}>
      {results.map((item) => (
        <Box
          key={item.id}
          borderWidth="1px"
          borderRadius="lg"
          p={4}
          textAlign="center"
        >
          {item.image && (
            <Image
              src={item.image}
              alt={item.title}
              borderRadius="md"
              mb={3}
            />
          )}
          <Text fontWeight="semibold">{item.title}</Text>
        </Box>
      ))}
    </SimpleGrid>
  );
}
```

Організм SearchSection

```
// ОРГАНІЗМ SearchSection
// ...imports
export default function SearchSection() {
  const [results, setResults] = useState([]);

  const handleSearch = async (query) => {
    const items = await searchSimulationFunction(query);
    setResults(items);
  };

  return (
    <Box py={8}>
      <Heading as="h2" size="lg" mb={6}>
        Пошук товарів
      </Heading>

      {/* молекула з інпутом + кнопкою */}
      <SearchField onSearch={handleSearch} />

      {/* молекула з результатами */}
      <SearchResults results={results} />
    </Box>
  );
}
```

Додаток 5

useRef

```
import { useRef, useState } from "react";

function InputFocus() {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleClick = () => {
    // Фокусуємо input при натисканні на кнопку
    inputRef.current?.focus();
  };

  return (
    <div>
      <input ref={inputRef} placeholder="Введіть щось..." />
      <button onClick={handleClick}>Фокус на input</button>
    </div>
  );
}

// Другий приклад: useRef для збереження попереднього значення без повторного рендеру
function PreviousValueTracker() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef<number | null>(null);

  const handleClick = () => {
    prevCountRef.current = count;
    setCount(count + 1);
  };

  return (
    <div>
      <p>Поточне: {count}</p>
      <p>Попереднє: {prevCountRef.current}</p>
      <button onClick={handleClick}>Збільшити</button>
    </div>
  );
}
```

useEffect

```jsx
import { useEffect, useState } from "react";

function Timer() {
  const [seconds, setSeconds] = useState(0);

  // Запускаємо інтервал при першому рендері
  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((s) => s + 1);
    }, 1000);

    // При демонтажі компонента очищаємо інтервал
    return () => clearInterval(interval);
  }, []);

  return <p>Минуло секунд: {seconds}</p>;
}
```

useContext

```jsx
// Приклад з useContext
// Створюємо контекст зі значенням за замовчуванням
const ThemeContext = createContext("light");

function ThemedComponent() {
  const theme = useContext(ThemeContext); // Отримуємо значення теми з контексту

  return <div>Поточна тема: {theme}</div>;
}

// Компонент, який огортає інші і надає контекст
function AppWithContext() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedComponent />
    </ThemeContext.Provider>
  );
}
```

ПРЕЗЕНТАЦІЯ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота на ступінь вищої освіти бакалавр
за темою:
«СИСТЕМА ДЛЯ МЕДИЧНОЇ ЛАБОРАТОРІЇ НА ОСНОВІ REACT.JS»

Виконав:
студент групи КНД-43
Єршов Ілля Володимирович
Керівник:
доцент кафедри комп'ютерних наук, доктор філософії
Березовська Юлія Володимирівна

Слайд 1

ЗАГАЛЬНА ХАРАКТЕРИСТИКА КВАЛІФІКАЦІЙНОЇ РОБОТИ

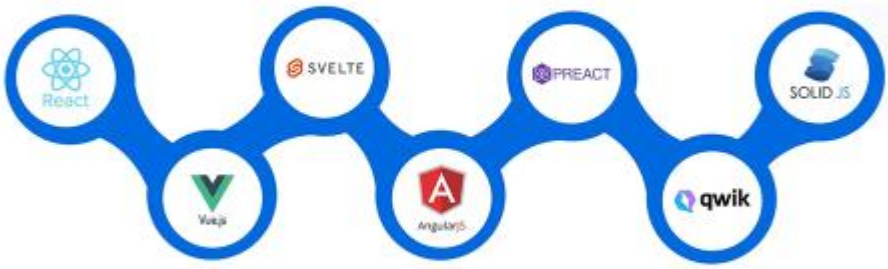
Мета дослідження:	оптимізувати роботу медичних лабораторій через впровадження інформаційної системи.
Об'єкт дослідження:	процес розробки ІС для лабораторних закладів на основі технології React.js.
Предмет дослідження:	методи розробки лабораторного ПЗ.

Слайд 2

СТРУКТУРНО-ЛОГІЧНА СХЕМА ДОСЛІДЖЕННЯ

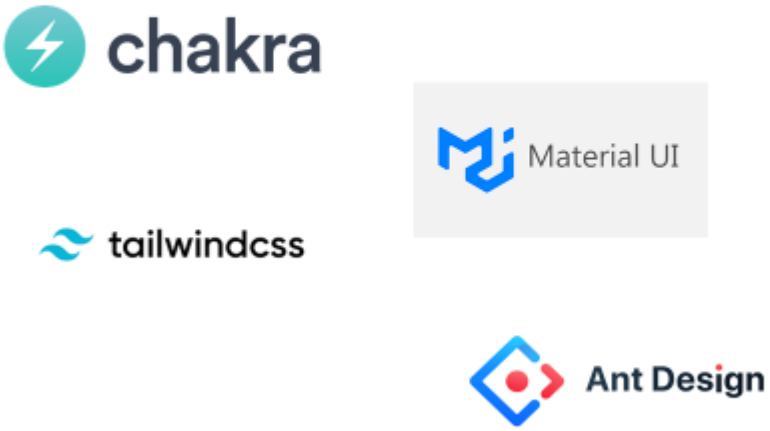
Слайд 3

АНАЛІЗ ІСНУЮЧИХ ФРЕЙМВОРКІВ



Слайд 4

ТЕХНОЛОГІЇ ГРАФІЧНОГО ОФОРМЛЕННЯ UI

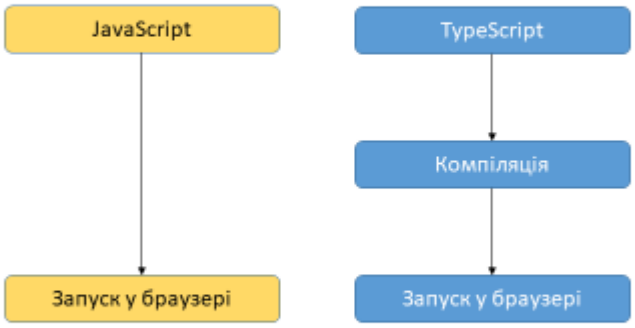


Слайд 5

ТЕХНОЛОГІЇ УПРАВЛІННЯ ГЛОБАЛЬНИМ СТАНОМ У REACT.JS

Слайд 6

ПОРІВНЯННЯ ДОРЕЧНИХ МОВ ПРОГРАМУВАННЯ



Слайд 7

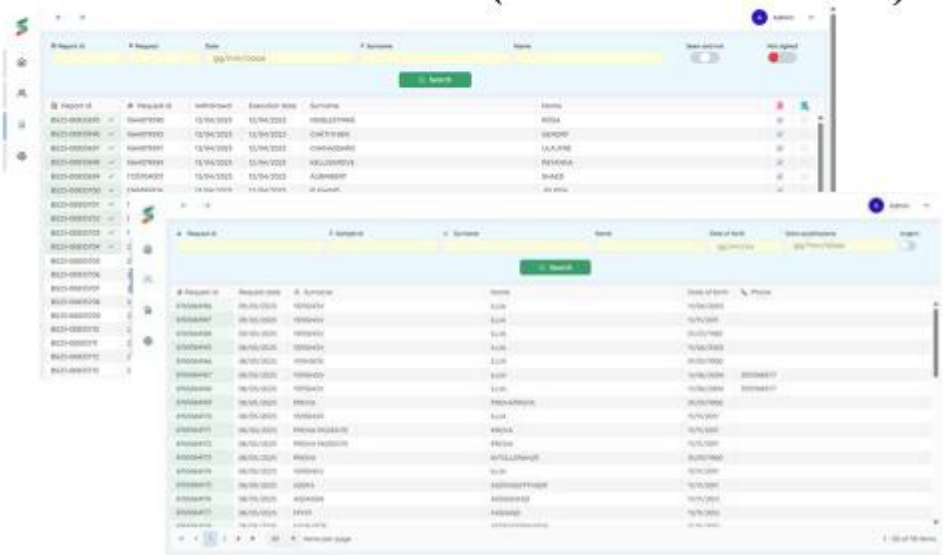
СЦЕНАРІЙ ВИКОРИСТАННЯ СИСТЕМИ

Слайд 8

ЗОВНІШНІЙ ВИГЛЯД РОЗРОБЛЕНОЇ СИСТЕМИ (ГОЛОВНА СТОРІНКА)

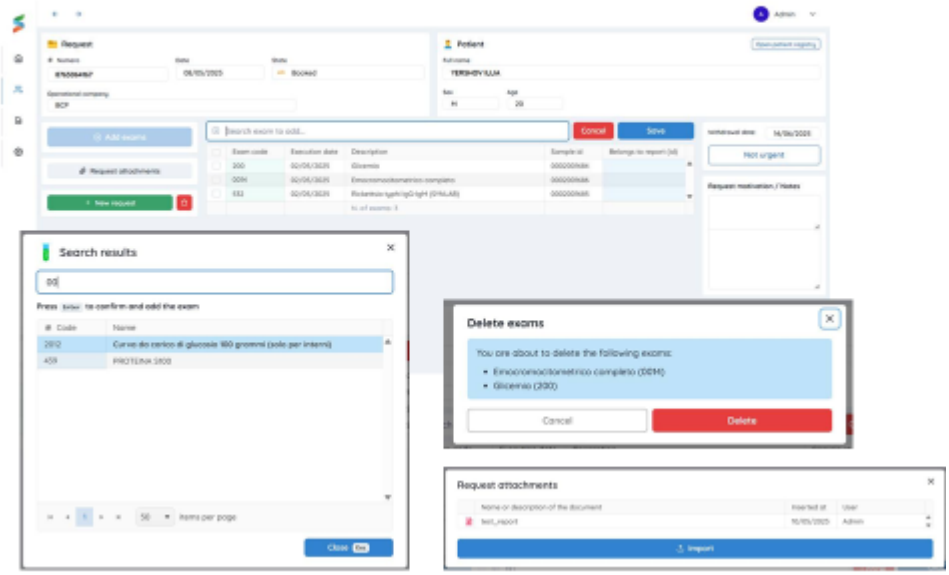
Слайд 9

СТОРІНКИ СПИСКІВ (ЗВІТИ ТА ЗАПИТИ)



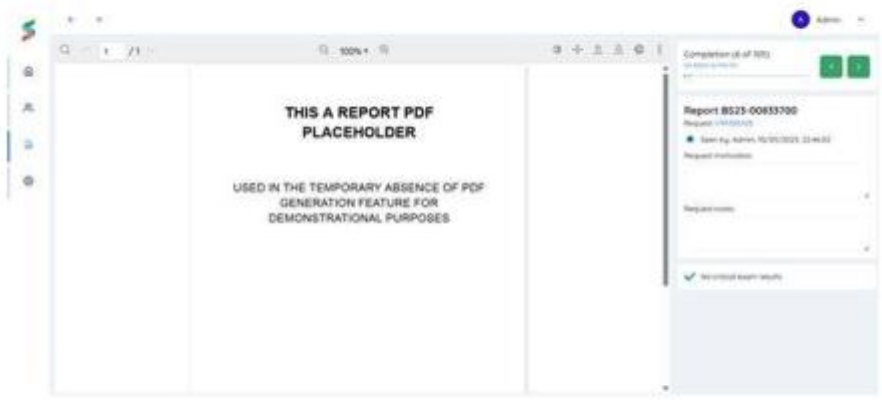
Слайд 10

СТОРІНКА ЗАПИТУ



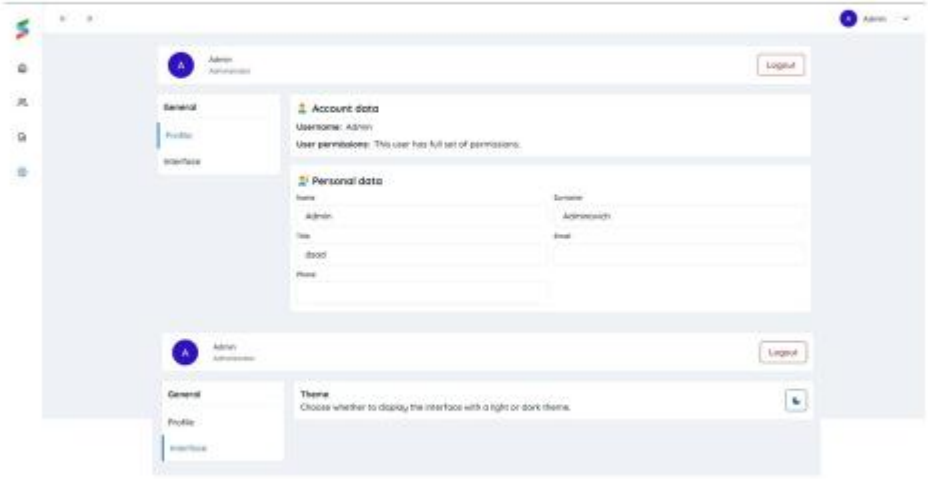
Слайд 11

СТОРІНКА ЗВІТУ



Слайд 12

НАЛАШТУВАННЯ КОРИСТУВАЧА



Слайд 13

ВИСНОВОК

1. Досліджено методи розробки лабораторних ІС.
2. Досліджено сучасні технології розробки веб-застосунків.
3. Розглянуті технології для вирішення популярних проблем у розробці SPA додатків.
4. Встановлено функціональні вимоги до медичної лабораторних систем.
5. Розроблено структуру програмного забезпечення.
6. Розроблено та протестовано програмне забезпечення.

Слайд 14