Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Розробка онлайн магазину для продажу техніки

Текстова частина до курсової роботи
за спеціальністю «Комп’ютерні науки»

Керівник курсової роботи
ас. Калітовський Б.В.
(прізвище та ініціали)

_____
(підпис)
“__” _____ 2021 р.
Виконав студент Кенийз В.Л.
“__” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

доцент, к.ф.-м.н.

_____ О. П. Жежерун

(підпис)

“___” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту 3 року навчання БП «Комп'ютерні науки»
Кенийзу Віталію Людвиковичу
на тему:
Розробка онлайн магазину для продажу техніки

Зміст ТЧ до курсової роботи:

1. Вступ
2. Опис предметної області
3. Теоретичне підгрунтя
4. Реалізація застосунку
5. Висновки по роботі та рекомендації для подальших
6. Список використаних джерел
7. Додаток А
8. Додаток Б

Дата видачі “___” _____ 2021 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Розробка онлайн магазину для продажу техніки

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітки
1.	Отримання завдання на курсову роботу.	11.10.2020	
2.	Огляд технічної літератури за темою роботи.	12.12.2020	
3.	Аналіз актуальності теми	12.01.2021	
4.	Дослідження теоретичної частини	31.01.2021	
5.	Програмування практичної частини	01.02.2021	
6.	Демонстрація демо-проєкту науковому керівнику	10.03.2021	
7.	Кінцеві правки практичної частини	06.04.2021	
8.	Корегування роботи згідно зауважень наукового керівника	20.04.2021	
9.	Написання теоретичної частини	25.04.2011	
10.	Створення презентації проєкту	05.05.2011	
11.	Здача роботи для перевірки на плагіат	17.05.2021	
12.	Захист курсової роботи(проєкту)	18.05.2021	

Студент _____

Керівник _____

“_____”_____

Зміст

Анотація

У даній роботі розглянуто та описано вивчення концепції та функціональних можливостей стеку технологій MERN. Було реалізовано додаток інтернет-магазину на основі цього стеку. Також детально пояснена кожна технологій даного стеку та допоміжних пакетів і бібліотек для побудови програми.

ВСТУП

Останніми роками веб-розробка швидко розвивається. Багато інструментів та технологій були введені для покращення досвіду розробників, досвіду користувачів та продуктивності веб програм. Кожен веб-додаток може бути побудований безліччю різних технологій. Термін "стек", який вказує на поєднання різних технологій, що використовуються для створення додатків, приділяли увагу з моменту створення LAMP стеку (Linux, Apache, MySQL та PHP). У наш час існує безліч варіантів для розробників вибрати стек для розробки нових додатків.

З огляду на останні розробки веб-технологій, концепція єдиної сторінки програми (SPA) стала популярною. Ідея, на яку покладаються SPA - уникати перезавантаження програми та повторне отримання вмісту всієї веб-сторінки з сервера для оновлення користувацького інтерфейсу. У порівнянні з традиційним способом перевантаження веб-сторінки для отримання нових даних, використання SPA покращує взаємодію з користувачами, уникаючи постійних перезавантажень та покращує продуктивність програми, лише отримуючи відповідні дані.

Зростання популярності SPA збільшило використання інтерфейсних фреймворків та бібліотек, оскільки значна частина роботи проводиться на клієнтській частині. Один із найдавніших стеків що уособлювало прийняття SPA - це стек MEAN, який складається з MongoDB як база даних, AngularJS як фреймворк клієнтської частини, Express як веб-сервер і Node як середовище виконання, був одним із найбільш звичних стеків для створення веб-додатків. React, представлений як альтернатива AngularJS швидко набув популярності у фронтенд-спільноті, тим самим замінивши "A" в MEAN для формування MERN стеку.

РОЗДІЛ 1. ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

Мільйони підприємств використовують інтернет як ефективний канал зв'язку. Це дозволяє їм обмінюватися інформацією зі своїм цільовим ринком та здійснювати швидкі та безпечні операції. Однак для цього потрібно збирати та зберігати всі необхідні дані та мати засоби обробки цієї інформації для представлення її користувачеві.

Веб-програми використовують комбінацію технологій для обробки інформації та пошуку інформації (Express), та (JavaScript та HTML) для представлення інформації користувачам. Це дозволяє користувачам взаємодіяти з компанією за допомогою онлайн-форм, систем управління вмістом, кошиків для покупок, відгуків тощо. Крім того, веб-застосунки дозволяють працівникам створювати документи, ділитися інформацією, співпрацювати над проєктами незалежно від місця розташування чи пристрою.

Тоді виникає запитання, як огранізувати цю роботу, і які інструменти використати для найбільш ефективного робочого процесу. Існує достатньо багато технологій розробки веб-сайтів, які зарекомендували себе високою функціональністю, надійністю та підтримкою в разі труднощей.

Тому досить важливим етапом розробки є вибір стеку технологій. Адже в майбутньому від цього залежатимуть багато факторів, серед яких: зменшення часу на тестування, можливість розгортання різних частин програми самостійно або встановлення меж між підсистемами.

Загалом, термін "стек" відноситься до поєднання різних технологій для створення веб-додатків. Існує три стеки з сімейства JS: MERN (MongoDB, Express, React та Node), який є одним з найпопулярніших та MEAN (MongoDB, Express, Angular та Node), який відрізняється від попереднього тільки клієнтською частиною, яку тут відіграє Angular, що робить цей стек більш важким при входженні та стеку MEVN (MongoDB, Express, VueJS та Node) – найменш популярний з них. Також є декілька алтернативних: LAMP (Linux, Apache, MySQL, PHP), який є чудовим вибором для корпоративних програм,

які використовуються високого рівня налаштувань, що робить цей стек вимогливим. Ruby on Rails (Ruby, SQLite, PHP) теж цікавий та потужний стек, основним мінусом якого є досить повільна швидкість виконання в порівнянні з NodeJS та важкість в пошуках гарно описаної документації. І заклюючним з відомих стеків є стек Django (Python, Django, MySQL), який дуже популярний та затребуваний стек, та він гарно показує себе тільки для великих та складних проєктів і може викликати труднощі якщо ви захочете відійти від шаблону складання проєкту на цьому стеці.

Тому, проаналізувавши аналоги стеків можна дійти висновку, що кожен з них має свої переваги та недоліки в певних напрямках. Але серед них стек MERN виглядає найбільш підходящим для початківців та при цьому може легко використовуватись для маштабних застосунку.

Отже, основними цілями було опис та вивчення концепцій та функціональних можливостей стеку MERN та реалізація веб-сайту на основі стеку MERN. Кожна технологія зі стеку MERN, а також допоміжні пакети та бібліотеки для побудови програми, такі як Redux та JWT були детально пояснені на початку курсової роботи, разом із актуальністю та характеристиками цього стеку. Крім того, було розроблено повнофункціональний додаток за допомогою стеку MERN для подальшого вивчення концепцій та сучасної практики щодо цього стеку технологій. Застосунок досить гарно показав себе у вирішенні поставлених цілей та труднощів, які виникли упродовж розробки. Також, працюючи з цим стеком з'явилося більш глубоке розуміння всіх переваг, серед яких простота побудови структури проєкту, висока продуктивність розробленої системи та інше.

РОЗДІЛ 2. ТЕОРЕТИЧНЕ ПІДРГУНТЯ

2.1 Поняття стеку

У цьому розділі будуть детально пояснені різні технології, що поєднуються для формування стеку MERN. що вони демонструють, як вони функціонують та з'єднуються. Крім того, інші інструменти, які допомагають керувати даними додатків, обробкою помилок та аутентифікацією користувача. Буде обговорена та вивчена реалізація кінцевої програми.

2.2 MERN

MERN - це стек, який базується на основі стеку MEAN, який вперше було представлене командою інженерів, що працює в MongoDB, у 2013 році. Стек MEAN - це абревіатура комбінацій мов та фреймворків: "M" для MongoDB, "E" для Express, "A" для AngularJS та "N" для Node. Замінивши популярний фреймворк AngularJS на бібліотеку React для покриття зовнішнього інтерфейсу та об'єднали її як стек MERN, React може супроводжувати інші технології для створення програм, орієнтованих на Javascript та JSON. [1]

У стеці MERN, MongoDB виступає як документо-орієнтована база даних, Express - це веб Framework та веб-сервер, React працює як клієнтська бібліотека, а Node - це середовище виконання. Малюнок нижче пояснює архітектуру стеку MERN.

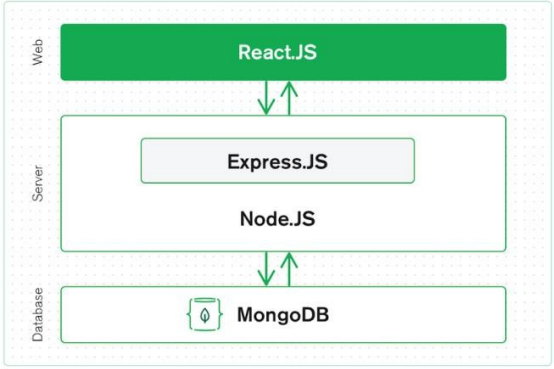

Рисунок 2.1. Трирівнева архітектура (клієнт, сервер, база даних) [2].

На основі *рисунку 2.1* видно, що стек MERN є повнорозмірним рішенням для розробки додатків, що використовує ідею давно встановленої 3-рівневої архітектури. Стек MERN складається з рівня відображення клієнта з React, рівня програми з Express і Node, а також рівня бази даних з MongoDB. [2]

2.2.1 MongoDB

Спочатку публічно представлений у 2007 році, MongoDB поступово виділився, як зручна технологія баз даних для розробників [3]. На відміну від SQL, який відомий як мова структурованих запитів, MongoDB є представником сімейства NoSQL загалом та мовних дерев на основі документів. [4] Термінологія NoSQL іноді згадується як не SQL або не тільки SQL. Тим не менше, більшість дійшла висновку, що бази даних NoSQL гнучко зберігають дані у форматі JSON-подібних документів на відміну від тих, що збираються реляційними базами даних [4].

Хоча база даних SQL буде збирати дані у поєднанні рядків і стовпців, база даних NoSQL буде впорядковувати інформацію в термінах документа, що містить масиви та об'єкти. Малі проєкти не можуть добре відображати різницю у використанні цих двох баз даних. Проте програма середнього розміру могла б реально продемонструвати членам команди, які безпосередньо керують та розробляють програму, переваги та труднощі кожного типу баз даних. [5]

2.2.2 Express

Як зазначено в документації на офіційному веб-сайті Express, Express є мінімальною структурою Node, яка може забезпечити надійний набір функціональних можливостей як для веб, так і для мобільних додатків [6]. Express сам по собі справді мінімальний, оскільки робить мало, але

покладається на проміжне програмне забезпечення і реалізує елементарний рівень на особливостях програми [4].

Проміжне програмне забезпечення відверто демонструє програмне забезпечення, що працює посередині життєвого циклу відповіді на запит. Один або кілька фрагментів проміжного програмного забезпечення виконуються для виконання точних завдань, таких як аутентифікація запитів або синтаксичний аналіз тіла запиту. Конвеєр завдань можна часто запускати з першим проміжним програмним забезпеченням, що викликається для обробки запиту. Це перше проміжне програмне забезпечення може закінчити запит і надіслати відповідь користувачам або викликати наступне проміжне програмне забезпечення, щоб продовжити запит. Той самий процес буде продовжуватися, як і кожен наступний. Проміжне програмне забезпечення бере результат попереднього як аргумент до останнього проміжного програмного забезпечення конвеєра. [7]

На малюнку нижче описаний процес від запуску запиту до остаточної функції, що надсилає відповідь.

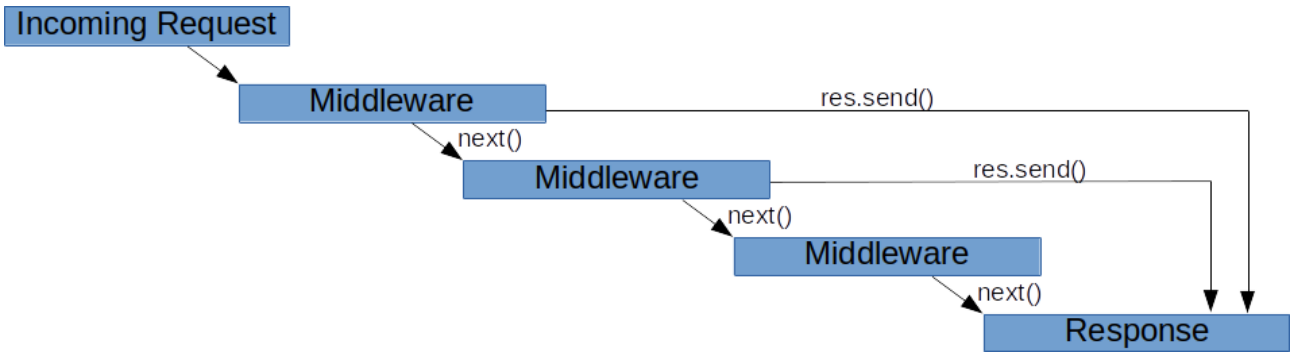

Рисунок 2.2 Запит-відповідь через проміжне програмне забезпечення [7]

Посилаючись на проміжне програмне забезпечення, точніше, воно стосується не лише функцій, які здатні доступатися не тільки до запитів HTTP та повернених об'єктів, але й до послідовних дій у циклі програми запит-відповідь [4]. Вище точно визначено, маршрутизація і фреймворк проміжного програмного забезпечення, такий як Express, може виконувати будь-які рядки коду, змінювати запит і результуючі об'єкти, відпишіться від циклу запит-

відповідь і викличте наступний метод проміжного програмного забезпечення, що стоїть у стеці. Якщо є такий випадок, що поточниа функція проміжного програмного забезпечення не може завершити цикл, користувачі можуть слідувати інструкціям на веб-сайті Express, який має викликати функцію, записану як next(), щоб пройти контроль до наступного входження проміжного програмного забезпечення, щоб уникнути незавершеного циклу [8].

Існують різні типи проміжного програмного забезпечення. Одним із найважливіших типів є проміжне програмне забезпечення рівня маршрутизатора, яке може використовуватися для обробки маршрутизації в Express. Воно схоже на будь яке інше проміжне програмне забезпечення, але воно обмежується лише екземпляром експрес-маршрутизатора. [8]

2.2.3 React

React - це бібліотека Javascript, випущена в 2013 році Facebook. Спочатку створена для вирішення складних великомасштабних користувацьких інтерфейсів зі зміною даних та прив'язкою їх у режимі реального часу, React продовжує зростати у розробці SPA програм та вдосконаленні інтерфейсних службових програм для всіх рівнів програмістів. Однією з функцій багатого функціоналу React, яка задовольняє великі виробничі додатки, які повинні бути швидкими та продуктивними, є віртуальний DOM або VDOM. Ця концепція представляє віртуальне представлення інтерфейсу користувача, яким React може швидко маніпулювати, не торкаючись реального інтерфейсу, використовуючи цей віртуальний об’єкт, щоб визначити, що потрібно зробити з реальним деревом DOM, та синхронізувати ці віртуальні та реальні дерева відповідно [9].

Поєднуючись з основними ідеями React, інші надійні функції, запроваджені командою інженерів Facebook для вирішення інших проблем будь яких веб або мобільних додатків, що вимагають адаптивного розміщення та

масштабованості зростаючих даних користувачів. також отримали схвалення від розробників у всьому світі. У межах даної курсової роботи лише деякі основні атрибути React згадані нижче у списку React Компоненти і React Хуки.

2.2.3.1 React Компоненти

Компоненти - основна концепція React, де розробникам пропонується розбити інтерфейс користувача на незалежні та багаторазові розділи. Компонент React може бути написаний двома способами: функціональний компонент або компонент класу, і найбільш легкий підхід до складання компонента - це написання функціонального компоненту, як функції Javascript, або використання класу ES6 для ілюстрації компонента. На думку React, ці дві методики, які продемонстровані нижче - ідентичні. [10]

```
function Welcome(props) {
  return (
    <div>
      <p>Hello {props.name} </p>
    </div>
  );
}
```

(а)

```
class Welcome extends React.Component {
  render() {
    return (
      <div>
        <p>Hello {this.props.name}</p>
    </div>
    )
  }
}
```

(б)

Рисунок 2.3 Функціональний компонент *(а)* проти компонента класу *(б)* в React

Як показано на *рисунку 2.3*, функціональні компоненти та компоненти класу мають однакові результати. Відображається компонент "p" зі словом "Hello...", а також пропс "name", передані в компонент.

Компоненти можуть посилатися один на одного, оскільки один компонент може бути батьківським компонентом, який містить безліч інших дочірніх компонентів, без обмеження на будь який рівень будь яких деталей. Незалежно від того, чи це клас, чи функціональний компонент, вони обидва дотримуються одного строгого правила, встановленого React: усі компоненти React - це чисті функції, які не змінюють свої властивості. Пропс - це набір входів, переданих як параметри компоненту, тоді як чиста функція ілюструє випадок, коли функція виконує логіку без зміни аргументів. Отже, компонент React працює як чиста функція, яка поважає свої вхідні дані і завжди повертає однакові результати для тих самих пропсів. [10]

2.2.3.1 React Хуки

Передача глобального стану за допомогою Redux виглядає продуктивно, незважаючи на це, управління локальним станом у складі компонента Redux вважається надмірним та непотрібним. Класи з React з самого початку добре виконують свою роботу з підтримання локальних станів з чіткою синтаксичною структурою. Це продовжує працювати і сьогодні, коли застосовується незалежно від масштабу проекту. Також, додатковим варіантом, який робить те саме, що і класи React є React Хуки, які були представлені Софі Алперт і Даном Абрамовим на React Conf 2018. [11]

Ця зміна є плавним переходом від класичних класів React, оскільки хуки не замінюють і не включають будь-які нові концепції React, наприклад, що стосуються властивостей, стану та життєвих циклів компонентів. Вступ Хуків також не означає відмову від класів React. Розробники та менеджери проєктів

можуть вільно вирішувати, чи хочуть вони спробувати щось нове, і рухатися вперед, або залишатись із тим самим синтаксисом, з яким працювали до цього. Спочатку хуки можуть викликати заплутане враження, але врешті-решт, логіка та цілі не далекі від основних ідей класів. На практиці можна сказати, що React Hooks зменшили кількість рядків коду, а також тимчасово усунули використання ключового слова "this". Насправді є більше відмінностей, ніж просто зменшення загальної кількості рядків коду та зміна вигляду програми.

React Hooks представляє хуки стану, які також відомі як useState хуки, які обробляють управління станом рівня компонентів. Якщо бути точнішим, useState - це хук, який переходить у стан React шляхом ініціалізації змінної стану, яка зберігається React-ом. Цей хук отримує та надсилає два значення як результати: поточний стан та функцію для його зміни. За допомогою хука useState стан компонента можна легко ініціалізувати, використовувати та оновлювати. [12]

Ще одним ключовим хуком, на який потрібно звернути увагу, є хук ефектів, який більш відомий як useEffect. Поки useState має справу зі станом, useEffect допомагає програмістам обробляти життєві цикли компонентів. Проблема розбиття логіки та даних на кілька життєвих циклів класу, наприклад, componentDidMount, componentDidUpdate, componentWillUnmount, була добре висвітлена в Effect Hook. Компонент React може охоплювати кілька ефектів, щоб відокремити проблеми маніпулювання даними. [12]

Одне, чого не вистачає класам React, в той час як Hooks мають відповідь - це спільна логіка функціональності. Щоб поділитися логікою стану, візуальною або невізуальною логікою раніше ніж Хуки, програмісти вибирають компоненти вищого порядку (HOC), або відтворюють шаблони пропсів. Що веде за собою відповідне налаштування ієрархії компонентів та робить програму більш складною для підтримання. З іншого боку, хуки дозволяють розробникам повторно використовувати логіку, не змінюючи структуру компонента. [11]

2.2.4 Node

Незважаючи на те, що технологія "народилася" 12 років тому, Node. (або Node.js) зарекомендував себе як життєво важливе середовище виконання JavaScript, що використовує популярність серверного JavaScript [13]. Середовище виконання не є ні мовою, ні структурою. Тим не менш, це потужний інструмент, побудований на двигуні V8 Chrome, який також працює з Javascript [4]. Використовуючи керовану подіями асинхронну та неблокувальну модель I/O, Node підтримує розробників ще одним варіантом створення легких додатків у реальному часі, крім стандартного шляху очікування та обслуговування запитів [4].

Для розробників, які в основному зосереджуються на Javascript, NodeJS надає переваги у створенні програм, а саме сервера та клієнта. Щоб докладніше розповісти про те, які ще переваги надає Node, програмісти доцільно виділяють менеджер пакетів Node або npm, який надає доступ до сотень тисяч пакетів, зареєстрованих у системі Node [4]. Реєстр npm вважався одним із найбільших у світі реєстрів пакетів, у 2017 році було зафіксовано понад триста п'ятдесят тисяч пакетів, багато з яких є відкритими кодами та розроблені розробниками по всьому світу [14].

2.2.5 Актуальність стеку MERN

Стек MERN - не єдина назва у списку комбінованих технологій для розробки веб-додатків. Можна назвати декілька, це MEAN (MongoDB, Express, Angular, NodeJS), LAMP (Linux, Apache, MySQL або MongoDB, PHP), Django (Python, Django, Apache, MySQL).

Існують різні причини, чому стек MERN є популярним та добре схваленим стеком технологій для створення веб-додатків. Стек MERN фокусується на одній базі коду з використанням Javascript та JSON, що

дозволяє розробникам глибше копати в певній мові програмування та покращити командну роботу та робочий процес у різних частинах цілого веб-додатку. Послідовність стеку також вказує на менший час для створення програми, менші зусилля для подальшого прогресу, легке розширення та підтримання програми. Не менш важливим є те, що кожен фрагмент стеку MERN пов’язаний із великою громадою, яка робить внесок у розвиток технологій та підтримку програмістів на будь-якому рівні. Доступно багато матеріалів з відкритим кодом, таких як документація, спеціальні пакети, додаткові бібліотеки. [4.]

2.3 Mongoose

Mongoose не є частиною MongoDB, а скоріше бібліотекою відображення об'єктних документів (ODM) для поліпшення роботи з Node та MongoDB [15]. Він не тільки забезпечує перевірку схем та управління взаємозв’язком даних, але також пов’язує об’єкти в коді та об’єкти в MongoDB [16]. *Рисунок 2.4,* нижче, описує зв’язок між Mongoose та іншими програмами:

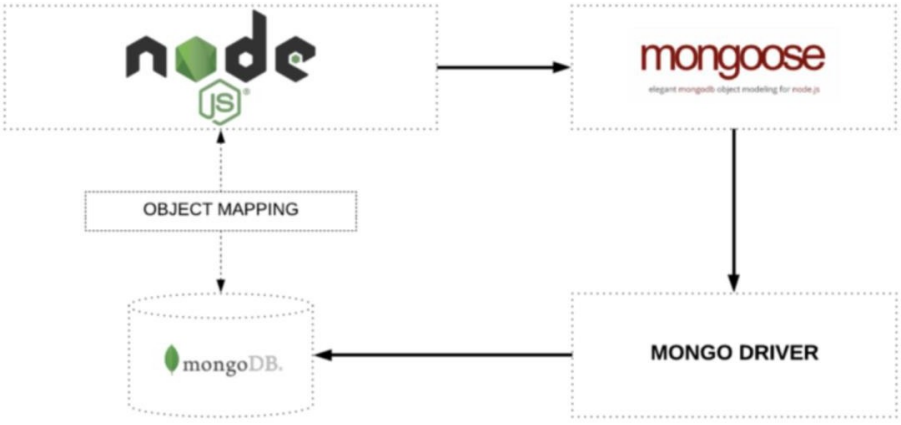

Рисунок 2.4 Mongoose відображає об'єкти між Node та MongoDB [16].

Як показано на *рисунку 2.4*, Mongoose використовується для Node для підключення до MongoDB за допомогою зіставлення об'єктів. Потім Mongoose підключається до MongoDB через драйвер Mongo. Через зв’язок між Mongoose, Node та MongoDB забезпечується можливість потоку даних. [16.]

Подібно до інших бібліотек ODM, першим кроком, з якого починається Mongoose, є схема [16]. Схема роз'яснює структуру даних та інші практики, перелічені на сторінці документації Mongoose: методи екземпляру, складений індекс, методи статичної моделі та проміжні засоби. Після завершення першого кроку, створені схеми відображатимуться у колекціях MongoDB та формуватимуть документи даних кожної колекції [16]. Другий крок, який програмістам потрібно виконати - це створення моделі Mongoose. Моделі - це компільовані конструктори схем, основними обов'язками яких є генерація та сканування документів бази даних Mongo. Інші можливості моделей, про які варто згадати - це запити, видалення та оновлення документів у базі даних [17].

2.4 Redux

Управління станами, мабуть, не є найбільш пріоритетною проблемою для дрібних проєктів, коли дані користувачів, відповіді серверів та кешовані дані не уповільнюють або порушують обмеження, які може обробляти веб-браузер. Незважаючи на це, коли додаток зростає, управління станом стає справжнім випробуванням для подальшого розвитку та налагодження програми, особливо коли змішуються два поняття асинхронності та мутації. Асинхронність визначає кілька змін в асинхронній послідовності, хоча мутація уточнює зміни стану програми. Ці дві концепції, як правило, складаються разом, коли в гру вступають непередбачувані події або час очікування відповіді, які, можливо, можуть видати несподівану поведінку. [18]

Сконструктований і розроблений Даном Абрамовим у 2015 році Redux, пропонував вирішити вищезазначену проблему шляхом перетворення передбачуваної мутації, не впливаючи на переваги асинхронності [18]. Якщо детальніше, то Redux зберігає стан в одному джерелі і вимагає суворої структури того, як може відбуватися модифікація стану, де відіграють багаторазові та чисті функції. Замість того, щоб змінювати значення даного об'єкта, чисті функції повертають ті самі нові значення на основі наданих аргументів, де б вони не були викликані. Таким чином, Redux дозволяє більш простий процес налагодження, тестування, підтримування коду та більш плавний розвиток досвіду розробки особисто чи в команді [19].

Ще одним визначним фактором, який було обрано Redux-ом, є його простота у використанні об'єктів і функцій Javascript. Він може добре поєднуватися з React та іншими клієнтськими бібліотеками або фреймворками, наприклад, AngularJS, Angular, VueJS, Polymer, Ember. Щоб додати більше можливостей до своєї гнучкості, Redux працює в різних середовищах, таких як браузер та сервер . [19]

Основна концепція Redux обертається навколо двох слів: редюсер та дія. Для спрощення термінів кожна мутація стану називається дією, а редюсер - це чиста функція, яка пов’язує стан та пов’язані з нею дії. Щоб запустити процес Redux, розробникам потрібно надіслати дію з мутацією стану, який записується як об'єкт Javascript, включаючи ім'я дії та іншу інформацію (якщо потрібно). для більш детального опису дії. Друге, що потрібно обробити - це написати редюсер, який приймає стан і дію як параметри і повертає новий стан. В результаті чистої функції повернутий результат редюсерів завжди має однакове очікуване значення, що робить стан незмінним і дотримується суворих вказівок, запропонованих Redux. [18]

2.5 Аутентифікація та авторизація з JWT

Аутентифікація - це процедура з’ясування того, ким є користувачем, тоді як авторизація - це процедура визначення того, до чого користувач може отримати доступ. Зазвичай аутентифікація проводиться перед авторизацією. Після встановлення того, ким є користувач, доступ до якогось ресурсу дозволений або відхилений. [20.]

Існують різні способи реалізації аутентифікації та авторизації для веб-додатків, де найбільш перевіреною в житті, і стандартною технікою є використання сеансів для обробки статусу користувача як на клієнті, так і на сервері [4]. Однак опублікована пізніше технологія JSON Web Token (JWT) реалізувала нову концепцію механізму без збереження стану, в якому зберігається статус користувача [4]. *Рисунок 2.5,* нижче, пояснює загальний процес механізму:

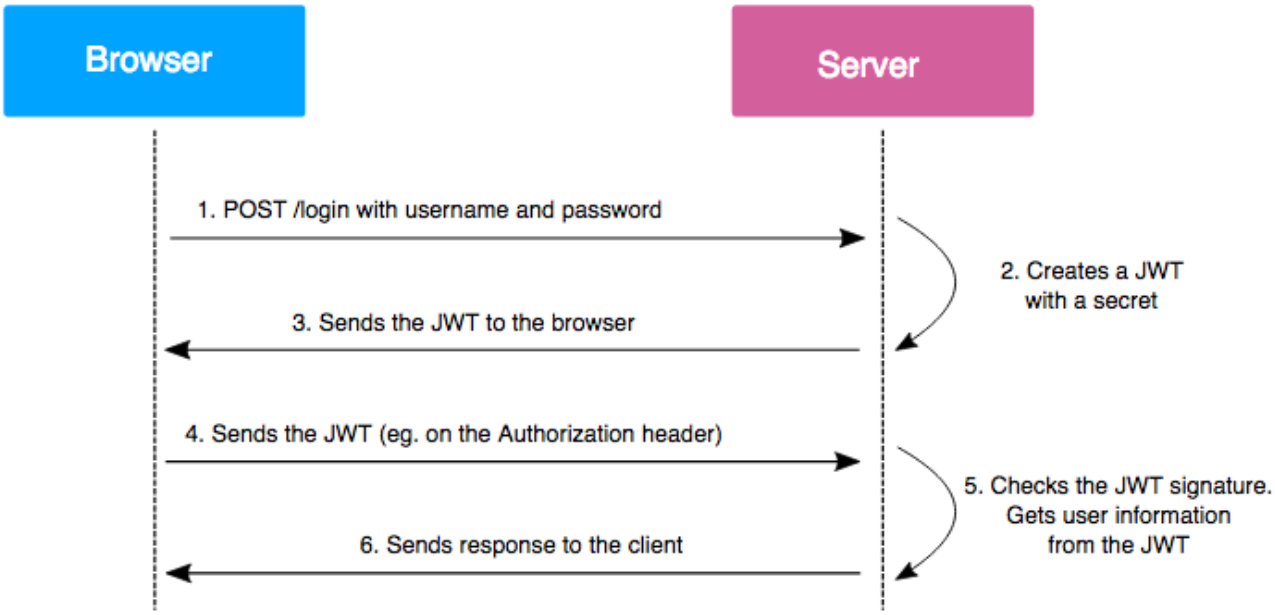

Рисунок 2.5 Процес авторизаціїї JWT [21]

Як показано на *рисунку 2.5*, по-перше, коли авторизується вхідний сервер, сервер створює токен JWT, підписаний за допомогою введених даних користувача та секретного ключа. Потім токе буде відправлений клієнту для зберігання у файлі cookie або локальному сховищі, таким чином передаючи клієнту підтримку стану користувача. Після успішного входу, в будь-який запит на захищені кінцеві точки, токен повинен бути приєднаний до заголовка запиту авторизації у форматі "Авторизація: Bearer <JSON веб токен>". Коли сервер обробляє запити до захищених кінцевих точок, токен пройде перевірку його дійсності. У цьому випадку доступ надається користувачеві, а в іншому випадку виникає помилка. [4.]

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

3.1 Вимоги до програми та налаштування проєкту

3.1.1 Вимоги до програми

Виходячи з основних проблем, що стосуються функціональних можливостей веб-сайту електронних продаж, програма ділиться на три категорії користувачів залежно від їх потреб та рівня доступу: відвідувач, зареєстрований користувач та адміністратор.

Перш за все, будь-які відвідувачі веб-сайту, повинні мати можливість переглянути список товарів разом із деталями кожного товару, включаючи зображення, опис та ціну, а також мати можливість шукати товари за назвою товару. Тим часом користувачі, які увійшли в систему, не тільки можуть робити те, на що здатні відвідувачі, але й мають доступ до більшої кількості переглядів, таких як екрани входу та виходу та сторінки для зміни контактних даних. Вони могли додавати товари в кошик для покупок, оплачувати чеки на покупки через Paypal, переглядати свої замовлення та перевіряти процес доставки замовлення. Нарешті, адміністратор - це роль користувача, що має повний доступ до програми. Крім усіх переглядів та функціональних

можливостей, до яких можуть наблизитись відвідувачі та зареєстровані користувачі, адміністратор керує додатком, додаючи, видаляючи, редагуючи базу даних продуктів або базу даних користувачів.

3.1.2 Налаштування середовища розробки

Перш ніж використовувати стек MERN для побудови програми, потрібно налаштувати середовище розробки, щоб технології та інструменти добре працювали разом та покращити розробку.

3.1.2.1 Git

Процес розробки повинен містити систему контролю версій, яка полегшує співпрацю, обмін кодом та перегляд змін коду [22]. Система контролю версій (VCS) відстежує історію змін і дозволяє розробникам вносити код та виправляти помилки, щоб гарантувати, що завжди можна повернутися до будь-якої попередньої версії [22]. На основі спостережень та опитування розробників Git є найпопулярнішим VCS у світі [22]. В результаті Git було встановлено та використано для контролю версій під час розробки цього додатка.

3.1.2.2 WebStorm

Середовище розробки WebStorm заснована на платформі IntelliJ, яку JetBrains вдосконалює вже більше 15 років. IDE тісно інтегрована з системами контролю версій, має багату екосистему плагінів і легко налаштовується під індивідуальні потреби користувача [23]. WebStorm є багатофункціональним серед яких: автодоповнення коду, перевірка помилок в реальному часі, навігація по коду, підсвічування синтаксису та вбудований термінал, який

полегшує процес розробки. WebStorm був обраний текстовим редактором для розробки застосунку.

3.1.3 Налаштування програми

Після налаштування середовища розробки було побудовано базове налаштування програми. Клієнтська частина програми був ініціалізований, після чого було налаштовано серверну частину. Потім зв’язок між клієнтом та серверною частину програми формувався відповідно.

3.1.3.1 Ініціалізація клієнтської частини

Налаштування типового робочого додатку React з нуля може бути непростим завданням, яке часто складається з налаштування конвеєру збірки Webpack та Babel, для транпіляції коду. Create React App - це інструмент, створений для швидкого встановлення зразка програми React, щоб полегшити розробку інтерфейсу за допомогою React. Для програми було створено типовий шаблон Create React App за допомогою команди на *рисунку 3.1*

```
npx create-react-app frontend
```

Рисунок 3.1 Команда для створення зразка React App

Після запуску вищевказаної команди шаблонний фроненд був налаштований без конфігурації та сформований у папці з назвою "frontend".

У папці React frontend, структура включає папку "node_modules", яка має встановлені залежності, і "public" папку, яка містить усі статичні файли.

Структура містить папку "src", яка охоплює компоненти програми. Крім того, створюється екземпляр файлу package.json, який перелічує залежності та відповідні версії, зберігає метадані клієнту та запускає визначені сценарії.

Файл package.json містить сценарії, які можна виконати для запуску, побудови або розробки програми.

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

Рисунок 3.2 Вбудовані команди React App в файлі package.json

На *рисунку 3.2* проілюстровані команди для запуску, побудови, тестування або вилучення клієнта. Щоб запустити програму на http://localhost:3000 як порт за замовчуванням, розробники можуть просто змінити каталог, щоб увійти до папки "frontend" і запустити "npm start" в терміналі, який буде запускати команда "start". Всередині папки src файл index.js є основною точкою входу Javascript в клієнтську частину програми. З іншого сторони, файл index.html всередині папки "public" є шаблоном сторінки для React програми.

3.1.3.2 Ініціалізація серверної частини

Для початку роботи над серверною базою було створено файл package.json для відстеження залежностей, документування проекту, запуску команд та ініціалізації серверної частини. Команда на *рисунку 3.3* була запущена з кореневої папки, щоб запропонувати низку запитань, щоб зібрати інформацію про проект. Потім пакет.json був сформований автоматично.

```
npm init
```

Рисунок 3.3 Команда для ініціалізації серверної частини

Файл server.js, який згодом був доданий, слугує входом до серверної частини програми, тоді як пакети "express" та "dotenv" були встановлені як залежності. Зміст початкового файлу server.js, зібраного на початку налаштування серверної бази, продемонстровано на *рисунку 3.4*

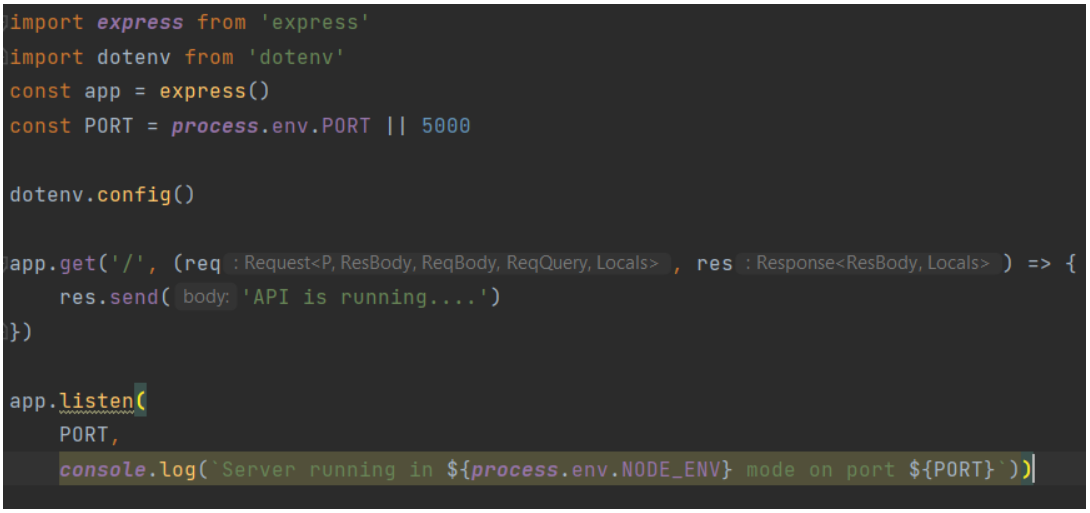

Рисунок 3.4 Вміст файлу server.js

З *рисунку 3.4* видно, що модуль "express" імпортовано за допомогою синтаксису ESModule, за ним слідує функція express (), що ініціює серверну частину. У рядку 2 пакет "dotenv" також імпортується та ініціалізується таким чином, що дані додатків, такі як токени або ключі API у файлі ".env", зберігаються поза історією git з міркувань безпеки. Програма запускає сервер, який підключається до порту 5000, а потім реєструє "Сервер працює на порту 5000" з консолі. Згодом при запитах до серверу він відповідатиме "API працює"

3.1.3.3 Комунікація між клієнтом та сервером

Проксі від Create React App був використаний на клієнті для зв’язку з серверною частиною на іншому порту. На *рисунку 3.5* показано конфігурацію, встановлену у папці "frontend" - package.json.

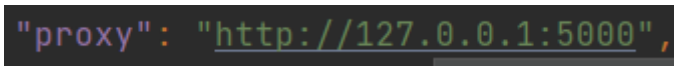

Рисунок 3.5 Проксі для комунікації клієнта з сервуром

На *рисунку 3.5* проксі-сервер із програми React App підключений до серверної частини, яка працює на порту 5000. В результаті формується зв'язок сервера з клієнтом. Крім того, потрібно було створити сценарії для безперебійного запуску програми. Модуль "nodemon" був доданий для постійного спостереження за змінами коду на сервері, тому немає необхідності в перезавантаженні сервера після кожної зміни коду.

Рисунок 3.6 Команди для запуску програми

На *рисунку 3.6* команда для запуску серверної частини - "npm run server", а команда для запуску клієнта - "npm run client". Для того, щоб програма працювала, критично важливо одночасно запускати як клієнта, так і сервер. Тому використовується пакет "concurrently", для одночасного запуску команд.

3.1.3.4 Mongoose з’єднання

Для підключення бази даних MongDB до програми використовується інструмент під назвою Mongoose. Як обговорювалось у розділі 2.3, Mongoose - це пакет об'єктного моделювання даних для Node, який дозволяє розробникам створювати Модель та схему для різних ресурсів бази даних.

```
import mongoose from 'mongoose'

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, options: {
      useUnifiedTopology: true,
      useNewUrlParser: true,
      useCreateIndex: true,
    })

    console.log(`MongoDB Connected: ${conn.connection.host}`.cyan.underline)
  } catch (error) {
    console.error(`Error: ${error.message}`.red.underline.bold)
    process.exit( code: 1)
  }
}
export default connectDB
```

Рисунок 3.7 Формування Mongoose з’єднання

Згідно з *рисунком 3.7* формується з'єднання Mongoose за допомогою змінної середовища MONGO_URI. Після цього функція імпортується у файл server.js і запускається для підключення програми до MongoDB та Mongoose. Згодом запуститвши сервер в терміналі відображається повідомлення про успіх якщо з'єднання вдалося. В іншому випадку повідомлення про помилку буде показано разом із помилкою, поверненою з Mongoose.

3.2 Розробка додатку

3.2.1 Розробка серверної частини

3.2.1.1 Модель Mongoose

Схема Mongoose визначає структуру документа, що зберігається в MongoDB, тоді як модель Mongoose формується шляхом обгортання схеми Mongoose для створення інтерфейсу до бази даних MongoDB для запитів, створення, видалення або оновлення записів. Всього в програмі три моделі: моделі користувачів, продуктів та замовлення.

Модель Користувача

(рисунок А.1), показує деталі моделі користувача. Екземпляр схеми Mongoose створюється за допомогою екземпляра і називається userSchema, що включає визначення схеми та бізнес-логіку для створення моделі користувача. Схема користувача має чотири обов'язкові поля: ім'я, електронну адресу, пароль та isAdmin. Електронну пошту встановлено як унікальне поле, оскільки не може бути делкілька людей з однаковими електронними адресами. Другий аргумент "timestamps" для схеми Mongoose встановлюється рівним true, так що Mongoose автоматично генерує поля createdAt та updatedAt, для зручності перевірки. Поле "isAdmin" має логічне значення і вказує, є користувач адміністратором чи ні, і за замовчуванням воно має значення false.

Згодом змодельована схема *(рисунок А.1),* для формулювання моделі користувача, яка експортується для повторного використання решти коду сервера. В рядку 18 *(рисунок А.1),* До схеми користувача приєднується хук "pre-save", який запускатиметься перед збереженням документа у базі даних. З міркувань безпеки паролі не повинні зберігатися в базі даних як звичайний текст і завжди повинні зашифруватись. Бібліотеку bcryptjs було встановлено для обробки шифрування та порівняння паролів у програмі. В рядку 23 *(рисунок А.1),* Salt (випадкові символи) для додавання до хешованого пароля

булb додані за допомогою методу genSalt(), який створений кількома генераціями. Згодом salt передавалась функції hash(), і зашифрованому паролю було присвоєно пароль, який буде збережено в базі даних. Незважаючи на це, шифрування поля пароля обробляється, лише якщо пароль було змінено або надіслано лише вперше. Тому було введено метод, щоб перевірити, чи не змінено пароль. Наприклад, якщо користувач оновлює лише поле імені або поле електронної пошти, але не пароль, буде викликана функція next(), і програма не оновлюватиме пароль. В іншому випадку додаток створить новий хеш, і користувач не зможе увійти в систему.

В рядках 14 по 16 *(рисунок А.1)*, до схеми користувача приєднується метод екземпляра, який називається "matchPassword" . Метод забезпечує порівняння між паролем, який користувач вводить під час входу, та зашифрованим паролем у базі даних за допомогою методу compare(). Оскільки шифрування паролів - це односторонній потік з міркувань безпеки, тобто неможливо розшифрувати зашифрований пароль як звичайний текст, метод compare() дозволяє порівняти паролі, щоб оцінити, чи ввів користувач правильний пароль.

Модель товару

На *(рисунок А.2)*, пояснюються деталі моделі товару. Екземпляр схеми Mongoose, яка називається productSchema, був ініціалізований, що інкапсулювало бізнес-логіку моделі товару. Схема товару містить безліч полів, пов’язаних з товаром. Вона містить назву, зображення, торгову марку, категорію, опис товару, ціну та кількість товару на складі, рейтинг та відгук користувача – це масив. Крім того, він містить користувача, який створив цей продукт, у формі ObjectId схеми Mongoose із прямим посиланням на модель користувача. Усі поля в схемі товару є обов’язковими.

Модель замовлення

У порівнянні зі схемами користувача та товару, схема замовлення має більше полів і виглядає складнішою. На *(рисунок А.3),* показано структуру схеми замовлення поряд з експортом моделі замовлення. Схема замовлення має ключі shippingAddress, PaymentMethod, PaymentResult, taxPrice, totalPrice і shippingPrice. Крім того, схема замовлення має ключ isPaid, який вказує, чи оплачується замовлення разом з ключем isDelivered, який показує, чи замовлення доставлено. Схема замовлення також має paidAt і DeliveryAt, яка відображає час оплати та доставки. Подібно до схеми товару, схема замовлення включає поле користувача, що представляє користувача, який створив цей продукт, із прямим посиланням на модель користувача. Поле orderItems - це масив, що включає впорядковані елементи.

3.2.1.2 Серверна авторизація та аутентифікація користувача

Коли запит подається на сервер Express, можна мати проміжне програмне забезпечення, яке може отримати доступ до будь-чого в об'єкті запиту та відповіді.

```
app.use(express.json())
```

Рисунок 3.8 Додавання проміжного програмного забезпечення

У програмі було використано проміжне програмне забезпечення express.json(), яке аналізує запити, що надходять із JSON. Як обговорювалося в розділі 2.2.2, також можна використовувати проміжне програмне забезпечення на рівні маршрутизатора, але воно повинно бути екземпляром функції

express.Router(). Використання проміжного програмного забезпечення на рівні маршрутизатора показано на *(рисунок А.4)*.

AuthRouter *(рисунок А.4)*, ініціалізується як екземпляр express.Router(). Потім, для запитів, що надходять до "/ ", authRouter обробляє маршрутизацію дочірніх маршрутів усередині нього. Наприклад, *(рисунок А.4)*, коли користувач переходить до "/login", єдиним дочірнім маршрутом authRouter, контролер authUser буде використовуватися для обробки запитів.

Єдиним маршрутом для входу користувачів у цій програмі є маршрут "/ login" із методом POST. authUser() *(рисунок А.4)*, виконує функцію контролера для цього маршруту. Коли користувач входить через цей маршрут, електронна пошта та пароль користувача перевірятимуться в базі даних. Після цього JWT буде використовуватися для надання доступу до певних частин та захищених маршрутів в API. Додаток створює екземпляр та підписує JWT із секретним ключем, коли користувач входить в систему, а потім відправляє його назад клієнту для використання та зберігання в клієнті. Якщо користувачеві потрібно отримати доступ до будь-якого захищеного маршруту, токен можна надіслати в заголовки, і сервер декодує токен тим самим секретним ключем, щоб перевірити, що токен дійсний. *(рисунок А.5),* відображає логіку контролера authUser().

Як показано на *(рисунок А.5)*, у рядку 13 запис користувача перевіряється в базі даних за допомогою електронної пошти. Якщо користувач із такою адресою електронної пошти існує, перевіряється пароль, який вводиться користувачем, чи відповідає він запису пароля, що зберігається в базі даних. У рядку 21, якщо користувач успішно підтверджений, програма надішле назад токен JWT підписаний за допомогою унікального _id користувача, який автоматично генерується MongoDB та секретним ключем. Метод createToken() створює підписаний JWT із секретним ключем, термін дії якого закінчується через 1 день. Контролер authUser() відправляє токен JWT назад, так, що якщо користувач отримує доступ до захищеного маршруту на сервері, користувач

може відправити токен JWT до заголовків на основі домовленості "Аутентифікація: Носій <TOKEN>".

Для авторизації користувачів було введено в дію проміжне програмне забезпечення з назвою "protect", яке обмежило доступ до кінцевих точок лише для користувачів, що увійшли в систему.

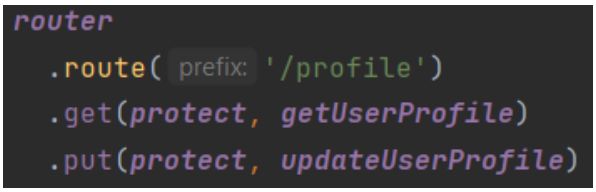

Рисунок 3.9 Використання проміжного програмного забезпечення "protect"

Маршрут під назвою "/profile" на *рисунку 3.9* використовується для відображення профілю користувача для зареєстрованого користувача. Маршрут охороняється за допомогою "protect" проміжного програмного забезпечення, щоб лише користувачі, що увійшли в систему, мали доступ до маршруту. Повний код "protect" проміжного програмного забезпечення наведено на *(рисунок А.6)*.

Як показано на *(рисунок А.6)*, перші заголовки запиту перевірялись, чи є в запиті заголовок авторизації. Після цього токен JWT витягується із заголовка авторизації. Після цього токен JWT декодується за допомогою функції JWT's verify() для отримання ідентифікатора користувача з токена JWT. Якщо JWT недійсний, функція verify() виведе помилку, і помилка буде надіслана назад клієнту. Потім проміжне програмне забезпечення перевіряє, чи є в базі даних користувач, який має той самий ідентифікатор користувача. Якщо буде знайдено користувача з однаковим ідентифікатором користувача в базі даних, проміжне програмне забезпечення дозволить користувачеві отримати доступ до маршруту. З іншого боку, поле "user" об'єкта запиту буде заповнене записом знайденого користувача в базі даних MongoDB для використання проміжними програмними забезпеченнями, що надходять після цього і називається "admin".

```
const admin = (req, res, next) => {
  if (req.user && req.user.isAdmin) {
    next()
  } else {
    res.status(401)
    throw new Error('Not authorized as an admin')
  }
}
```

Рисунок 3.10 Проміжне програмне забезпечення "admin"

Як показано на *рисунку 3.10*, у рядку 2, проміжне програмне забезпечення "admin" спочатку перевіряє, чи є поле "user" в об'єкті запиту, і чи поле "isAdmin" істинне. Якщо провірку пройдено, користувач є адміністратором і може отримати доступ до маршруту.

```
router.route('/').post(registerUser).get(protect, admin, getUsers)
```

Рисунок 3.11 Використання проміжного програмного забезпечення "admin"

З *рисунку 3.11* видно, що проміжне програмне забезпечення "admin" розміщується після "protect" проміжного програмного забезпечення, щоб "protect" могло пропустити об'єкт "user", що представляє користувача, який має доступ до маршруту. Потім проміжне програмне забезпечення "admin" бере користувацький об'єкт, доданий до об'єкта запиту "protect" проміжним програмним забезпеченням, і перевіряє, чи є поле "isAdmin" в об'єкті користувача істинним, що вказує на те, що користувач є адміністратором. Якщо це так, проміжне програмне забезпечення "admin" дозволить користувачеві отримати доступ до захищеного маршруту.

3.2.2 Розробка клієнтської частини

3.2.2.1 Інтеграція Redux

Redux використовується для зберігання глобального стану в цьому проекті. Наприклад, деякі частини станів, такі як, магазинні товари, з'являються в багатьох місцях, тому має сенс зробити їх доступними для всіх компонентів. Незважаючи на те, що можна перенести весь стан на компонент верхнього рівня і пропустити шматки стану через пропси, для програми такого розміру це буде погано. Для інтеграції Redux у програму спочатку було створено сховище Redux. На *(рисунок А.7)*, показано створення сховища Redux для програми.

Як пояснено на *(рисунок А.7)*, у рядку 6 основний редюсер було ініціалізовано комбінуванням менших редюсерів, кожен з яких керував частиною стану в додатку. Згодом сховище було реалізоване за допомогою функції createStore() від Redux, яка взяла основний редюсер, початковий стан і підсилювач, що складається з інструменту Redux dev та проміжного програмного забезпечення redux-thunk. Засіб розробника Redux допомагає візуалізувати сховище Redux у браузері під час розробки програми, тоді як проміжне програмне забезпечення redux-thunk надає можливість використовувати асинхронні дії в Redux, що дуже корисно при роботі з отриманням даних.

Далі було експортовано сховище Redux, яке використовується для створення екземпляра Redux у застосунку.

Компонент Provider було імпортовано з пакета "react-reduxProvider передав сховище Redux до всіх вкладених компонентів всередині нього. Потім компонент програми було імпортовано та використано всередині Provider-а, так що всі дочірні компоненти програми мають доступ до глобального сховища та можуть отримувати інформацію з нього.

3.2.2.2 Основна сторінка

Домашня сторінка - це сторінка, яка відображається за замовчуванням, коли користувач звертається до програми. Під час побудови домашньої сторінки потік Redux ретельно вибирає дані, оскільки цей потік повторно використовується на багатьох інших сторінках. Головна сторінка відображає товари магазину. Спочатку, коли користувач переходить на цю сторінку, товари будуть завантажені з сервера на клієнт, а потім будуть зіставлені для створення компонентів ProductCard. Потім компоненти ProductCard відображатимуться на сторінці.

Вибір товарів для домашньої сторінки здійснювався за допомогою Redux. Спочатку був створений редюсер для стану "products". На *(рисунок А.8)*, представлений код редюсера та типи дій для продуктів.

Як показано на *(рисунок А.8)*, PRODUCT_LIST_REQUEST, PRODUCT_LIST_SUCCESS і PRODUCT_LIST_FAIL - це лише три необхідні типи дій для продуктів, що представляють три статуси отримання даних: отримання даних відбувається, завантаження даних – успішне та завантаження даних - не успішне. Товар магазину має форму об’єкта, включаючий товар та помилки. Якщо тип дії - PRODUCTS_REQUEST, статус у товарній частині магазину буде REQUEST. Якщо тип дії - PRODUCTS_SUCCESS, статус буде SUCCESS, а продукти з корисного навантаження дії будуть розміщені в сховищі Redux. Нарешті, якщо виклик API не вдається, статус буде ERROR, а помилка буде включена до сховища Redux.

Редюсер та дії застосовуються для отримання даних товарів для головного сторінки після цього. *(рисунок А.9)*, відображає весь потік даних отримання продуктів.

Як видно на *(рисунок А.9)*, сховище Redux підключене до програми, а товар видобувається зі сховища Redux за допомогою хука useSelector, який був доступний. Помилка та властивості продукту в подальшому деструктуруються

з товару сховища Redux і використовуються для відображення інформації всередині компонента HomePage. У рядку 18 змінна диспетчеризації створюється за допомогою хука useDispatch з "react-redux" і може відправляти дії до сховища Redux. Коли компонент HomePage відображено, хук useEffect запускається і відправляє дію, викликаючи імпортовану функцію listProducts() *(рисунок А.10)*, яка виконується в рядку 21 *(рисунок А.9)* .

Після виклику функція listProducts() розсилає дію, що має тип PRODUCTS_LIST_REQUEST, що вказує на те, що розпочато отримання даних. Дані товарів згодом отримуються з маршруту "/api/products" на серверній панелі за допомогою пакету "axios". Якщо отримання даних буде успішним, функція listProducts() надішле тип дії PRODUCTS_LIST_SUCCESS, що містить корисне навантаження даних сервера. Якщо отримання даних не вдається, функція listProducts() надішле тип дії PRODUCTS_LIST_ERROR і дія буде містити корисне навантаження помилки, яка виникає під час отримання даних. Змінні стану, помилки та товар отримані зі стану продуктів Redux, відповідно відображатимуть основну сторінку. Логіка відображення основної сторінки описана на *(рисунок А.11)*.

Згідно *(рисунок А.11)*, на основі статусу отримання даних, компонент HomePage відображається по-різному для кожного типу стану. Якщо триває отримання даних, статус отримання даних - це ключ loading, на сторінці з’явиться завантажувач. Завантажувач покращує взаємодію з користувачем і вказує на те, що відбувається отримання даних. Якщо статус error, тобто отримання даних не вдається, відображатиметься повідомлення про помилку із вмістом помилки. В іншому випадку, якщо отримання даних буде успішним, товари будуть заповнені та зіставлені для створення компонентів ProductCard, які можна побачити на домашній сторінці.

3.2.3 Автентифікація користувача

Автентифікація користувача на клієнті складається зі сторінки Login, функціоналу входу та виходу.

3.2.3.1 Сторінка Login та функціонал входу

Сторінка входу - це місце, де користувач може увійти в програму для використання функціоналу, які може виконувати лише зареєстрований користувач, наприклад, додавання товарів у кошик, замовлення продуктів або оплата. Сторінка входу також має загальний заголовок для кожної сторінки програми. Вид сторінки входу вказаний на *(рисунок А.12)*.

Як показано на *(рисунок А.12)*, сторінка входу складається з форми з двома вхідними полями: електронна пошта та паролем, які передає користувач, та кнопки подання для відправки запиту на вхід. Коли користувач вводить електронну пошту та пароль, вони зберігаються у стані за допомогою хуків useState.

Як видно на *(рисунок А.13)*, для змінних електронної пошти та пароля спочатку встановлюються порожні рядки за допомогою хука useState. Згодом компонент підключається до Redux за допомогою хука useSelector, а після цього фрагмент стану UserLogin Redux стан був перенесений зі сховища Redux. Введення даних в поля форми електронної пошти та пароля занесуться до стану. Нарешті, коли користувач надішле запит на вхід, натиснувши кнопку «SIGN IN», сторінка віддасть дії, які формуються шляхом виконання функції login() з даними, які ввів користувач. Функція login() використовується для входу користувача та інформацію про користувача в локальне сховище. Стан userLogin вказує чи ввійшов користувач в систему. Якщо увійшов, то сторінка входу перенаправить користувача на домашню сторінку, де представлені товари магазину. Відтепер інформація про користувача буде доступною для використання у всій програмі.

3.2.3.2 Функціонал виходу

Коли користувач увійшов у систему, у шапці сайту буде випадаючий список, що відображає ім’я користувача. Коли користувач натискає випадаючий список, з'явиться опція виходу. Після натискання кнопки "Logout" програма надішле дію, щоб очистити інформацію про користувача в сховищі Redux та очистити інформацію про локальне сховище користувача.

ВИСНОВКИ ПО РОБОТІ ТА РЕКОМЕНДАЦІЇ ДЛЯ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Метою курсової роботи було проаналізувати та описати функціональність та основні характеристики стеку MERN та розробити додаток для інтернет-магазину. Кожна технологія, що включається в стек MERN, була детально вивчена разом із особливостями та релевантністю цілого стеку MERN. В докладі були ретельно досліджені сучасні практики, основні концепції стеку MERN та бібліотеки, що доповнюють цей стек.

Врешті-решт, додаток, готовий до розробки, було успішно створено. Будь-який відвідувач може зареєструватися протягом декількох кроків, проглядати товари та їх детальний опис, додавати товари в кошик та оплатити замовлення за допомогою Paypal. Завдяки вищим привілеям адміністратор може оновлювати асортимент товарів, позначати замовлення як оплачені, оновлювати базу даних користувачів та підвищувати користувачів до рівня адміністратора. Загалом, програма задовольнила всі заздалегідь визначені вимоги проєкту.

Курсову роботу можна розглядати як посібник розробки до стеку MERN і він може допомогти тим, хто хоче дізнатись більше про розробку за допомогою цього стеку. Врешті-решт стек MERN виявився здатним створювати досить складні full-stack програми. Однак кінцеву програму все одно можна вдосконалити, додавши новий функціонал, такий як: нові способи оплати, вхід за допомогою соціальних мереж, систему рекомендацій для конкретного користувача і тд. Також можуть бути додані додаткові розширені концепції, такі як візуалізація на стороні сервера та тестування, для вдосконалення різних аспектів програми.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 Karpov V. The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js [Електронний ресурс] / Valeri Karpov // MongoDB. – 2017. – Режим доступу до ресурсу: https://www.mongodb.com/blog/post/the-mean-stack-mongodb-expressjs-angularjs-and

2 MERN Stack [Електронний ресурс] // MongoDB. – 2021. – Режим доступу до ресурсу: https://www.mongodb.com/mern-stack.

3 Horowitz E. A Founder's Reflections on 10 Years of MongoDB [Електронний ресурс] / Eliot Horowitz // MongoDB. – 2017. – Режим доступу до ресурсу: https://www.mongodb.com/mern-stack.

4 Hoque S. Full-stack React Projects Second Edition. Birmingham, UK [Електронний ресурс] / Shama Hoque // Packt Publishing. – 2020. – Режим доступу до ресурсу: https://books.google.com.ua/books?id=097dDwAAQBAJ&pg=PP3&lpg=PP3&dq=4+Shama+Hoque.+Full-stack+React+Projects+Second+Edition.+Birmingham,+UK:+Packt+Publishing;+2020.&source=bl&ots=CMF47nj6s2&sig=ACfU3U0_A4Cs-goszkE0o6joVbzwBKw3YQ&hl=uk&sa=X&ved=2ahUKEwilk6_x8cvwAhVps4sKHQpyB40Q6AEwB3oECAgQAw#v=onepage&q=4%20Shama%20Hoque.%20Full-stack%20React%20Projects%20Second%20Edition.%20Birmingham%2C%20UK%3A%20Packt%20Publishing%3B%202020.&f=false.

5 Wodehouse C. 5 SQL vs. NoSQL Databases: What's the Difference? [Електронний ресурс] / Carey Wodehouse // upwork. – 2019. – Режим доступу до ресурсу: https://www.upwork.com/resources/sql-vs-nosql-databases-whats-the-difference?utm_source=google&utm_campaign=SEM_GGL_INTL_NonBrand_Marketplace_DSA&utm_medium=cpc&utm_content=113089129402&utm_term=&campaignid=11384804789&gclid=Cj0KCQjw4v2EBhCtARIsACan3nyHQ6pgqfFyVBTw37nF7UCEW1Cm8dukqIwKc4vEnIKUC8AICuE_ajEaAnbOEALw_wcB.

6 Express [Електронний ресурс] // Express. – 2021. – Режим доступу до ресурсу: https://expressjs.com.

7 What is middleware? How it works in nodeJS? A Simple implementation [Електронний ресурс] // medium. – 2019. – Режим доступу до ресурсу: https://medium.com/dataseries/what-is-middleware-how-it-works-in-nodejs-a-simple-implementation-485bcb9c3d53.

8 Using middleware [Електронний ресурс] // Express. – 2021. – Режим доступу до ресурсу: https://expressjs.com/en/guide/using-middleware.html.

9 Virtual DOM and Internals [Електронний ресурс] // React. – 2021. – Режим доступу до ресурсу: https://reactjs.org/docs/faq-internals.html.

10 Components and Props [Електронний ресурс] // React. – 2021. – Режим доступу до ресурсу: https://reactjs.org/docs/components-and-props.html.

11 Introducing Hooks [Електронний ресурс] // React. – 2021. – Режим доступу до ресурсу: https://reactjs.org/docs/hooks-intro.html.

12 Using the State Hook [Електронний ресурс] // React. – 2021. – Режим доступу до ресурсу: https://reactjs.org/docs/hooks-state.html.

13 A brief history of Node.js [Електронний ресурс] // NodeJS. – 2021. – Режим доступу до ресурсу: https://nodejs.dev/learn/a-brief-history-of-nodejs.

14 Goldwater M. An abbreviated history of JavaScript package managers [Електронний ресурс] / Matt Goldwater // JavaScript. – 2019. – Режим доступу до ресурсу: https://javascript.plainenglish.io/an-abbreviated-history-of-javascript-package-managers-f9797be7cf0e.

15 Учебник Express часть 3: Использование базы данных (с помощью Mongoose) [Електронний ресурс] // MDN Web Docs. – 2021. – Режим доступу до ресурсу: https://developer.mozilla.org/ru/docs/Learn/Server-side/Express_Nodejs/mongoose.

16 Karnik N. Introduction to Mongoose for MongoDB [Електронний ресурс] / Nick Karnik // freeCodeCamp. – 2018. – Режим доступу до ресурсу: https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/.

17 Models [Електронний ресурс] // mongoose. – 2021. – Режим доступу до ресурсу: https://mongoosejs.com/docs/models.html.

18 Motivation [Електронний ресурс] // Redux. – 2021. – Режим доступу до ресурсу: https://redux.js.org/understanding/thinking-in-redux/motivation.

19 Bachuk A. Redux · An Introduction [Електронний ресурс] / Alex Bachuk // Articles. – 2016. – Режим доступу до ресурсу: https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/.

20 Zuraida A. Authorization and Authentication [Електронний ресурс] / Audira Zuraida // Meduim. – 2018. – Режим доступу до ресурсу: https://medium.com/@audira98/authorization-and-authentication-cd0a7c994278.

21 Garcia R. JWT tokens and security – working principles and use cases [Електронний ресурс] / Romain Garcia // VAADATA. – 2016. – Режим доступу до ресурсу: https://www.vaadata.com/blog/jwt-tokens-and-security-

working-principles-and-use-cases/.

22 Git Handbook [Електронний ресурс] // GitHub. – 2021. – Режим доступу до ресурсу: https://guides.github.com/introduction/git-handbook/.

23 IDE features [Електронний ресурс] // JetBrains. – 2021. – Режим доступу до ресурсу: https://www.jetbrains.com/webstorm/features/ide-features.html.

Додаток А
(обов’язковий)
Рисунки вихідних кодів

```
import mongoose from 'mongoose'
import bcrypt from 'bcryptjs'

const userSchema = mongoose.Schema(
  {
    name: { type: String, required: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    isAdmin: { type: Boolean, required: true, default: false },
    },
  { timestamps: true}
)

userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password)
}

userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) {
    next()
  }

  const salt = await bcrypt.genSalt(10)
  this.password = await bcrypt.hash(this.password, salt)
})

const User = mongoose.model('User', userSchema)

export default User
```

Рисунок 1 Модель користувача

```
const productSchema = mongoose.Schema(
  {
    user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User', },
    name: { type: String, required: true, },
    image: { type: String, required: true, },
    brand: { type: String, required: true,},
    category: { type: String, required: true,},
    description: { type: String, required: true,},
    reviews: [reviewSchema],
    rating: { type: Number, required: true, default: 0,},
    numReviews: { type: Number, required: true, default: 0,},
    price: {  type: Number, required: true, default: 0,},
    countInStock: { type: Number, required: true, default: 0,},
  },
  {
    timestamps: true,
  }
)
```

Рисунок 2 Модель товару

```
const orderSchema = mongoose.Schema(
  {
    user: {  type: mongoose.Schema.Types.ObjectId,  required: true,  ref: 'User',},
    orderItems: [{
      name: { type: String, required: true },
      qty: { type: Number, required: true },
      image: { type: String, required: true },
      price: { type: Number, required: true },
      product: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'Product',},
    },],
    shippingAddress: {
      address: { type: String, required: true },
      city: { type: String, required: true },
      postalCode: { type: String, required: true },
      country: { type: String, required: true },
    },
    paymentMethod: {  type: String,  required: true,},
    paymentResult: {
      id: { type: String },
      status: { type: String },
      update_time: { type: String },
      email_address: { type: String },
    },
    taxPrice: { type: Number, required: true, default: 0.0,},
    shippingPrice: { type: Number, required: true, default: 0.0,},
    totalPrice: { type: Number, required: true, default: 0.0,},
    isPaid: { type: Boolean, required: true, default: false,},
    paidAt: { type: Date,},
    isDelivered: { type: Boolean, required: true, default: false,},
    deliveredAt: {  type: Date,},
  },
  { timestamps: true,}
)
```

Рисунок 3 Модель замовлення

```
import express from 'express'
const router = express.Router()
import {
  authUser,
  registerUser,
  getUserProfile,
  updateUserProfile,
  getUsers,
  deleteUser,
  getUserById,
  updateUser,
} from '../controllers/userController.js'
import { protect, admin } from '../middleware/authMiddleware.js'

router.route('/').post(registerUser).get(protect, admin, getUsers)
router.post('/login', authUser)
router
  .route('/profile')
  .get(protect, getUserProfile)
  .put(protect, updateUserProfile)
router
  .route('/:id')
  .delete(protect, admin, deleteUser)
  .get(protect, admin, getUserById)
  .put(protect, admin, updateUser)

export default router
```

Рисунок 4 Використання проміжного програмного забезпечення на рівні маршрутизатора

```
import asyncHandler from 'express-async-handler'
import generateToken from '../utils/generateToken.js'
import User from '../models/userModel.js'

import sendEmail from "../notifications/mail.js";

// @desc    Auth user & get token
// @route   POST /api/users/login
// @access  Public
const authUser = asyncHandler(async (req, res) => {
  const { email, password } = req.body

  const user = await User.findOne({ email })

  if (user && (await user.matchPassword(password))) {
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
      token: generateToken(user._id),
    })

  } else {
    res.status(401)
    throw new Error('Invalid email or password')
  }
})
```

Рисунок 5 Логіка контролера authUser()

```
import jwt from 'jsonwebtoken'
import asyncHandler from 'express-async-handler'
import User from '../models/userModel.js'

const protect = asyncHandler(async (req, res, next) => {
  let token

  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    try {
      token = req.headers.authorization.split(' ')[1]

      const decoded = jwt.verify(token, process.env.JWT_SECRET)

      req.user = await User.findById(decoded.id).select('-password')

      next()
    } catch (error) {
      console.error(error)
      res.status(401)
      throw new Error('Not authorized, token failed')
    }
  }

  if (!token) {
    res.status(401)
    throw new Error('Not authorized, no token')
  }
})
```

Рисунок 6 Проміжне програмне забезпечення "protect"

```
import { createStore, combineReducers, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import { composeWithDevTools } from 'redux-devtools-extension'
import { productListReducer } from './reducers/productReducers'

const reducer = combineReducers({
  productList: productListReducer,
})

const middleware = [thunk]

const store = createStore(
  reducer,
  initialState,
  composeWithDevTools(applyMiddleware(...middleware))
)

export default store
```

Рисунок 7 Сховище Redux в застосунку

```
export const productListReducer = (state = { products: [] }, action) => {
  switch (action.type) {
    case PRODUCT_LIST_REQUEST:
      return { loading: true, products: [] }
    case PRODUCT_LIST_SUCCESS:
      return {
        loading: false,
        products: action.payload.products,
        pages: action.payload.pages,
        page: action.payload.page,
      }
    case PRODUCT_LIST_FAIL:
      return { loading: false, error: action.payload }
    default:
      return state
  }
}
```

Рисунок 8 Редюсер та типи дій для товарів

```
13  const HomePage = ({ match }) => {
14    const keyword = match.params.keyword
15
16    const pageNumber = match.params.pageNumber || 1
17
18    const dispatch = useDispatch()
19
20    const productList = useSelector((state) => state.productList)
21    const { loading, error, products, page, pages } = productList
22
23    useEffect(() => {
24      dispatch(listProducts(keyword, pageNumber))
25    }, [dispatch, keyword, pageNumber])
26
```

Рисунок 9 Потік даних отримання товару

```
27  export const listProducts = (keyword = '', pageNumber = '') => async (
28    dispatch
29  ) => {
30    try {
31      dispatch({ type: PRODUCT_LIST_REQUEST })
32
33      const { data } = await axios.get(
34        `/api/products?keyword=${keyword}&pageNumber=${pageNumber}`
35      )
36
37      dispatch({
38        type: PRODUCT_LIST_SUCCESS,
39        payload: data,
40      })
41    } catch (error) {
42      dispatch({
43        type: PRODUCT_LIST_FAIL,
44        payload:
45          error.response && error.response.data.message
46            ? error.response.data.message
47            : error.message,
48      })
49    }
50  }
```

Рисунок 10 Завантаження даних

```jsx
<h1>Latest Products</h1>
{loading ? (
  <Loader />
) : error ? (
  <Message variant='danger'>{error}</Message>
) : (
  <>
    <Row>
      {products.map((product) => (
        <Col key={product._id} sm={12} md={6} lg={4} xl={3}>
          <Product product={product} />
        </Col>
      ))}
    </Row>
    <Paginate
      pages={pages}
      page={page}
      keyword={keyword ? keyword : ''}
    />
  </>
)}
```

Рисунок 11 Логіка відображення основної сторінки

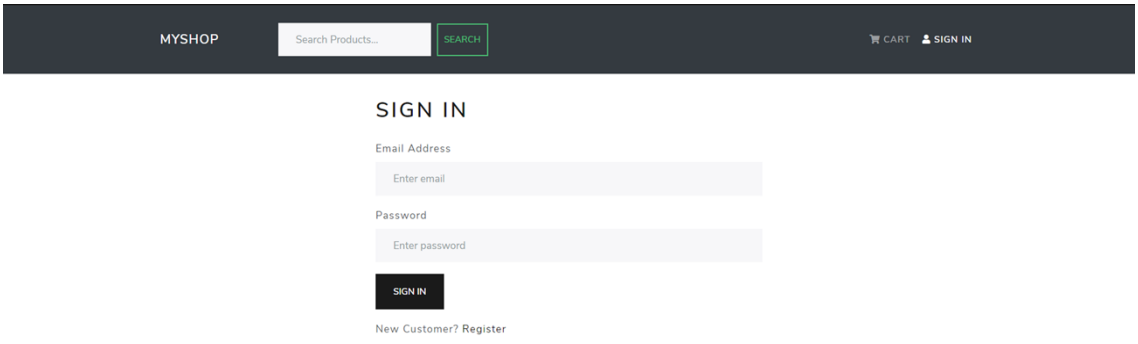

Рисунок 12 Вигляд сторінки Login

```
10  const LoginPage = ({ location, history }) => {
11    const [email, setEmail] = useState('')
12    const [password, setPassword] = useState('')
13
14    const dispatch = useDispatch()
15
16    const userLogin = useSelector((state) => state.userLogin)
17    const { loading, error, userInfo } = userLogin
18
19    const redirect = location.search ? location.search.split('=')[1] : '/'
20
21    useEffect(() => {
22      if (userInfo) {
23        history.push(redirect)
24      }
25    }, [history, userInfo, redirect])
26
27    const submitHandler = async (e) => {
28      e.preventDefault()
29      dispatch(login(email, password))
30    }
```

Рисунок 13 Логіка сторінки Login

Додаток Б
(обо'вязковий)

Перелік прийнятих скорочень

API	Інтерфейс прикладної програми
CSS	Каскадні таблиці стилів
DOM	Об'єктна модель документа
ES6	Стандарт мови програмування
HTML	Мова розмітки гіпертексту
HTTP	Протокол передачі даних
HOC	Компоненти вищого порядку
I/O	Введення / Виведення
JSON	Формат обміну даними
MEAN	MongoDB, Express, Angular, Node
MERN	MongoDB, Express, React, Node
NPM	Менеджер пакетів для JavaScript
ODM	Зіставлення документа об'єкта
ORM	Об'єктно-реляційне відображення
SEO	Оптимізація пошукової системи
SPA	Односторінковий застосунок
SQL	Мова структурованих запитів
VCS	Системи контролю версій
VDOM	Віртуальний DOM