МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)
травня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук,
освітньо-наукової програми «Інформатика»
на тему: «Інформаційна технологія проектування інформаційної системи електронної комерції»
здобувача групи ІН.м - 21н Ворошилова Данило Олександровича

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

_____ Данило ВОРОШИЛОВ
(підпис)

Керівник, старший викладач Олег БЕРЕСТ _____
(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 - Комп'ютерних наук, освітньо-наукової програми «Інформатика»
здобувача групи ІН.м - 21н Ворошилова Данило Олександровича

1. Тема роботи: «Інформаційна технологія проектування інформаційної системи електронної комерції»
затверджую наказом по СумДУ від *«8» березня 2024 року № 0234-VI*

2. Термін здачі здобувачем кваліфікаційної роботи *до 21 травня 2024 року*

3. Вхідні дані до кваліфікаційної роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження.
2) Огляд архітектури та методів реалізації 3) Практична реалізація 4) Висновок.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	Аналіз проблеми предметної області, постановка й формування завдань дослідження		
2	Огляд архітектури та методів реалізації		
3	Практична реалізація		

4	Аналіз отриманих результатів		
5	Оформлення пояснювальної записки до кваліфікаційної роботи		

Здобувач вищої освіти _____ Керівник _____

(підпис) (підпис)

АНОТАЦІЯ

Записка: 53 стор., 5 рис., 3 додатки, 15 літературних джерел.

Об’єкт дослідження — Інформаційна технологія проєктування інформаційних систем електронної комерції

Мета роботи — розробка інформаційної технології проєктування інформаційних систем електронної комерції, орієнтованої на високопродуктивний та масштабований веб-додаток.

Результати — проведено детальний аналіз сучасних методів та інструментів для розробки інформаційних систем у сфері eCommerce. На основі проведеного аналізу була розроблена інформаційна технологія, яка реалізована у вигляді веб-додатку. Основний функціонал додатку був протестований та готовий до загального використанням.

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ, ВЕБ-ДОДАТОК, БАЗА ДАНИХ, MONGODB, EXPRESS.JS, REACT, NODE.JS

ЗМІСТ

ВСТУП

У епоху стрімкого розвитку інформаційних технологій, які глибоко впливають на різні аспекти життя, неможливо перебільшити їхній вплив на розвиток різних економічних секторів. Однією з таких галузей, що активно інтегрує ці інновації, є сектор eCommerce. Використання передових інформаційних систем у сфері електронної комерції відкриває нові горизонти для зростання продажів, поліпшення управління клієнтськими базами та оптимізації процесів онлайн-торгівлі.

У 2024 році світ електронної комерції переживає значні зміни та інновації, визначаючи нові напрямки розвитку і виклики для бізнесів. Від традиційного онлайн-шопінгу до інтеграції штучного інтелекту, ця галузь постійно адаптується до змінюваних потреб та поведінки споживачів.

Онлайн-торгівля стала однією з найважливіших та найдинамічніших сфер у світовій економіці, зумовлюючи значні зміни у способах ведення бізнесу та споживчої поведінки. У цьому контексті, інтернет-комерція перетворилася на потужний двигун економічного розвитку, відкриваючи нові горизонти для підприємців та споживачів. Ця галузь невпинно еволюціонує, використовуючи інноваційні технології, що забезпечують зручність, швидкість та безпеку транзакцій. Онлайн-торгівля включає в себе широкий спектр активностей, від B2B (бізнес-до-бізнесу) та B2C (бізнес-до-споживача) моделей до найсучасніших форм електронного ринку. Ця сфера розвивається через постійне впровадження новітніх технологій, таких як штучний інтелект, машинне навчання, великі дані, та блокчейн, що не лише трансформують бізнес-процеси, але й створюють нові виклики та можливості для підприємств усіх розмірів. Враховуючи її стрімкий розвиток та все зростаюче значення у глобальній економіці, можна зробити висновок про актуальність даної дипломної роботи.

Метою цієї дипломної роботи є не лише аналіз поточного стану IT-розвитку в електронної комерції, а й виявлення основних викликів та

розробка передових практик, якими можна задовольнити специфічні потреби цього сектору.

Для досягнення визначеної мети буде проведено аналіз сучасних технологічних рішень, які використовуються в секторі eCommerce. Особлива увага буде зосереджена на дослідженні специфічних вимог, що ставляться до інформаційних систем у цій галузі. Це включатиме такі аспекти, як управління клієнтськими базами, аналіз поведінки споживачів, ефективне управління запасами та логістикою, оптимізація процесів доставки, а також інтеграція з передовими технологіями для автоматизації та збору даних для підвищення ефективності онлайн-продажів.

Результатом роботи буде розроблений веб-додаток, який було створено на основі аналізу сучасних методів та інструментів для розробки інформаційних систем у сфері електронної комерції.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Дослідження актуальності проблеми

Сектор eCommerce є ключовим та стратегічно важливим напрямком у економіці України. Він відіграє важливу роль не лише у задоволенні потреб внутрішнього ринку, але й у формуванні експортного потенціалу, зважаючи на зростаючу присутність українських онлайн-магазинів на міжнародній арені.

Аналізуючи сучасний стан розвитку електронної комерції, важливо відзначити, що через воєнні події галузь спочатку зазнала різкого спаду, за яким наступив збільшений попит на окремі категорії товарів, а після цього настала певна стабілізація. Ця стабілізація стала можливою завдяки поступовому, облаштуванню на нових місцях або повернення людей додому, що сприяє поступовому відновленню обсягів продажів в інтернеті.

Хоча більші eCommerce компанії в Україні вже активно впроваджують сучасні IT-рішення, такі як системи управління клієнтськими базами, аналітику даних та інші автоматизовані технології, багато середніх та малих онлайн-підприємств все ще стикаються з труднощами у впровадженні цих інновацій. Це обмежує їхні можливості зростання та ефективності.

Значний потенціал eCommerce сектору полягає у використанні інформаційних систем для оптимізації бізнес-процесів. Це може включати покращення управління запасами, автоматизацію логістики, підвищення точності маркетингових стратегій та використання аналітики для прийняття обґрунтованих рішень.

Існує важлива потреба в дослідженні та розробці інформаційних систем, спеціально адаптованих до потреб українських eCommerce підприємств. Це охоплює розробку доступних, масштабованих та ефективних рішень, що легко інтегруються в існуючі бізнес-процеси.

Таким чином, розробка та впровадження ефективних інформаційних систем для українського сектору eCommerce є не лише актуальною, але й

стратегічно важливою задачею, що може значно підвищити продуктивність, ефективність та конкурентоспроможність на світовому ринку.

1.2 Постановка задачі

Завданням дипломного проекту є розробка інформаційної системи, яка б оптимізувала управлінські процеси бізнесу в сфері eCommerce. А саме:

1. Огляд та аналіз сучасного стану eCommerce систем: Дослідити сучасні тренди та виклики у сфері електронної комерції. Аналізувати існуючі рішення та визначити ключові функціональні та технологічні вимоги до інформаційних систем у цій галузі.
2. Вивчення технологічного стеку MERN (MongoDB, Express.js, React, Node.js): Проаналізувати компоненти стеку MERN, їхні особливості, переваги та недоліки для розробки систем eCommerce. Визначити можливості інтеграції цих технологій у розробку сучасних eCommerce рішень.
3. Проектування архітектури системи: Розробити архітектуру інформаційної системи eCommerce, враховуючи потреби бізнесу, вимоги до користувачів, безпеку та масштабованість. Визначити структуру бази даних, серверну логіку, клієнтський інтерфейс та інтеграцію з зовнішніми сервісами.
4. Розробка прототипу системи eCommerce: На базі проектованої архітектури створити прототип системи eCommerce, використовуючи технології стеку MERN. Це має включати розробку фронтенду та бекенду, а також налаштування бази даних.
5. Тестування системи: Провести комплексне тестування розробленого прототипу, включаючи функціональне тестування, тестування безпеки, навантажувальне тестування, та виявлення можливих помилок і недоліків у системі.

Ця дипломна робота спрямована на розробку інформаційної системи, яка значно підвищить ефективність управління в секторі eCommerce України, а також стане фундаментом для подальшого розвитку та інтеграції новітніх технологій у цій сфері.

1.3 Огляд та аналіз сучасного стану eCommerce систем

Онлайн-купівля зараз вважається найбільш швидко зростаючим сегментом ритейлу. З кожним роком покупців в інтернеті стає більше. Ще вісім років тому їх було 1,5 млрд осіб, а за підсумками 2023 року – 2,64 млрд [1]. Разом із цим збільшується й обсяг ринку. В Україні ринок e-commerce у 2023 році сягнув майже 5 млрд доларів [2].

Цифрова трансформація для роздрібної торгівлі включає в себе автоматизацію процесів й оцифровку даних, а також використання хмарних технологій для їх зберігання [3]. Крім того, можна покращувати призначений для користувача досвід покупця за допомогою розширеної аналітики, доповненої й віртуальної реальності, штучного інтелекту. Також через біометричні POS-термінали можна сплачувати за товари за допомогою технології розпізнавання обличь [4].

Ключові переваги сфери електронної комерції, такі як простота, швидкість, зручність та безпека, залишаються актуальними. У порівнянні з минулим, коли покупки в інтернеті здійснювали майже виключно молоді люди віком від 18 до 23 років, наразі товари та послуги онлайн замовляють і користувачі в інших вікових категоріях, від 25 до 45 років. Ще однією тенденцією є зростання популярності покупок через смартфони через їх мобільність та зручність, натомість ноутбуки чи десктопи стають менш популярними.

Основним завданням платформ для електронної комерції залишається приваблення клієнтів та здійснення продажів [5]. Проте, платформи повинні відповідати ряду нових вимог, щоб бути гнучкими і стійкими:

Масштабованість: Інвестування коштів у продукт, який не має потенціалу для зростання, є неефективним. Сучасні платформи eCommerce повинні бути готовими витримувати стрімке збільшення трафіку і бути готовими до модифікацій і налаштувань.

Простота інтеграцій: Онлайн-магазини часто вимагають різних інструментів, від інтернет-еквайрингу до CRM та інших систем. Платформи повинні забезпечувати легку та швидку інтеграцію з різними інструментами.

Соціальність: Сегмент соціальної комерції швидко розвивається. Платформам слід максимально інтегрувати роботу з аудиторією соцмереж та інфлюєнсерами.

Сумісність з новими технологіями: Нові технології, такі як штучний інтелект (ШІ), машинне навчання, AR/VR, змінюють електронну комерцію. Платформи повинні бути готовими до впровадження таких рішень.

Кібербезпека: Захист особистих даних покупців стає все важливішим. Платформи повинні відповідати стандартам кібербезпеки.

Нова доба електронної комерції починається зі змін у звичках людей, а технології лише надають цій трансформації довершеного вигляду.

2 ОГЛЯД АРХІТЕКТУРИ ТА МЕТОДІВ РЕАЛІЗАЦІЇ

2.1 Вибір архітектури

З розвитком веб-розробки зародилася концепція односторінкових додатків (SPA – Single Page Applications), що характеризується наявністю однієї веб-сторінки на клієнтській стороні. При завантаженні нових модулів у SPA, контент на них оновлюється лише частково, оскільки нема потреби повторно завантажувати постійні елементи. Це збільшує швидкість відгуку та скорочує обсяг даних, що передається між браузером та сервером. Приклади зручних та корисних односторінкових додатків – Gmail та Google Translate [6].

Перевагами Single Page Applications можна вважати:

Доступність. Односторінкові додатки (SPA) надають користувачам можливість негайного доступу до повного спектру функціоналу з будь-якого типу пристрою, без зайвих турбот про сумісність, обсяг пам'яті, необхідні потужності або витрачений час на встановлення. Ця особливість робить SPA ідеальним вибором для сучасного, мобільного світу, де користувачі очікують миттєвого доступу до сервісів і додатків без будь-яких перешкод або затримок.

Універсальність. SPA можна використовувати з практично будь-якого пристрою, який має доступ до інтернету. Це стає можливим завдяки адаптивному дизайну, який враховує різні роздільні здатності екранів. Чи це ПК з великим монітором, чи мобільний телефон з невеликим екраном – SPA адаптується під усі ці умови, забезпечуючи зручність і легкість використання незалежно від обраного пристрою.

Можливість залучити великі обсяги даних. Завдяки тому, що SPA працює в браузері та інтегрується з хмарними сервісами, обсяг даних та функціонал додатку не обмежується локальною пам'яттю пристрою. Це означає, що користувачі можуть використовувати додатки з величезною кількістю даних і складними функціями, не турбуючись про вичерпання пам'яті своїх пристроїв.

Швидкість. Основна перевага SPA полягає в тому, що вся необхідна інформація завантажується один раз, після чого будь-які взаємодії з додатком відбуваються миттєво. Це не лише заощаджує час, який зазвичай витрачається на повторне завантаження сторінок, але й значно підвищує загальну продуктивність і зручність користування.

Можливості розробки. Розробникам доступні фреймворки, які спрощують створення архітектури проекту та пропонують чимало готових елементів для роботи.

Але як і будь-яка система, концепція SPA містить і недоліки, такі як:

Необхідність інтернет-з'єднання. Без доступу до мережі використовувати цей софт неможливо. Але навіть якщо десктопне ПЗ використовує у роботі зовнішні бази даних, доступ до Інтернету все одно необхідний.

Проблеми з SEO. Особливості SPA ускладнюють або унеможливлюють процес індексації пошуковими системами всіх модулів додатка. Це може спричинити труднощі з оптимізацією.

Не працює у користувачів з відключеною підтримкою JS. Багато хто відключає відображення JS-елементів у себе в браузерах, а Single Page Application використовує їх у роботі, тому може не працювати.

Ця нова парадигма привела до створення сучасних веб-фреймворків, таких як: Angular, React та Vue, однак для розробки повноцінного веб-додатку потрібні додаткові технології.

Одним з найвідоміших наборів технологій є MERN стек, який об'єднує MongoDB, Express, React та Node.js. Цей стек вирізняється використанням JavaScript у всіх частинах розробки, створюючи єдину екосистему, що сприяє швидкості, зручності та високій якості роботи. Включення npm як менеджера пакетів забезпечує доступ до великої кількості пакетів, які легко інтегрувати в додаток.

Архітектура MERN (рис. 2.1) дозволяє легко побудувати трирівневу архітектуру (інтерфейс, бек-енд, база даних), повністю використовуючи JavaScript і JSON [7].

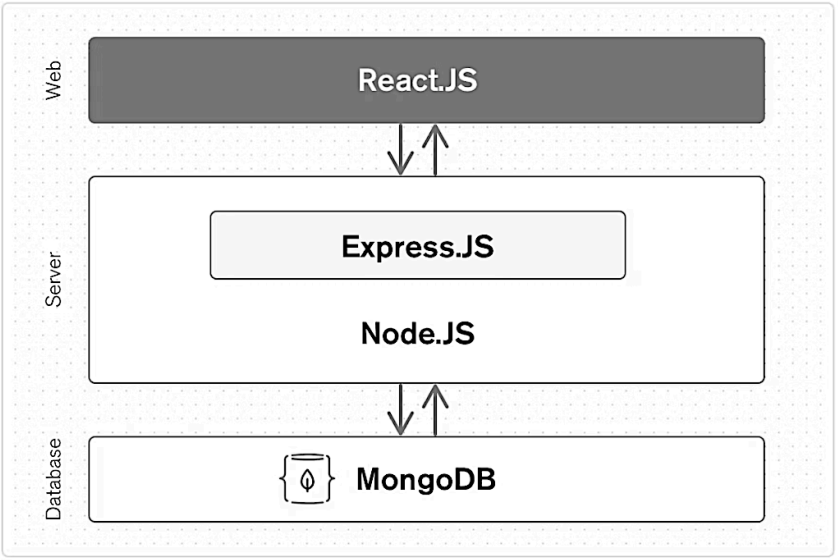

Рисунок 2.1 – Архітектура MERN

2.2 Аналітичний огляд мови програмування та бібліотек розробки

MERN стек є одним з популярних виборів для розробки повноцінних веб-додатків. Використання JavaScript на як на фронтенді, так і на бекенді, сприяє плавності розробки і зменшує контекстний перехід між мовами.

Як вже було згадано, застосування MERN стеку веде до уніфікації формату обміну даними (JSON) між базою даних, клієнтом і сервером, що в значній мірі спрощує обробку і передачу даних, зменшуючи час, необхідний для їх трансформації.

Найвищим рівнем стеку MERN є React.js. Це відкрита JavaScript бібліотека для створення інтерфейсів користувача, яка покликана вирішувати проблеми часткового оновлення вмісту вебсторінки, з якими стикаються в розробці односторінкових застосунків [8]. React дозволяє створювати складні інтерфейси за допомогою простих компонентів, підключати їх до даних на сервері та відтворювати їх як HTML.

Плюси React.js:

Компонентний підхід: React використовує компонентний підхід, що дозволяє розбивати інтерфейс на незалежні блоки, які легко перевикористовувати та управляти.

Декларативний: React робить код більш читабельним та легким для налагодження завдяки своєму декларативному стилю.

Велика екосистема і спільнота: Велика кількість доступних бібліотек, інструментів і фреймворків, а також велика і активна спільнота.

Підтримка одностороннього потоку даних: Це полегшує управління даними і допомагає запобігати помилкам в програмах.

Virtual DOM: Підвищує продуктивність, оскільки зміни в DOM виконуються в пам'яті, а не в реальному DOM.

Сумісність з SEO: SPA, створені за допомогою React, можуть бути оптимізовані для пошукових систем.

Мінуси React.js:

Висока швидкість розвитку: Часті оновлення можуть викликати проблеми з сумісністю та необхідність частого навчання.

Тільки бібліотека інтерфейсу: React є лише бібліотекою для інтерфейсів, тому для повноцінної розробки додатку потрібно було прийнято рішення використати додаткові бібліотеки.

Крива навчання: JSX і архітектура React можуть бути складними для новачків.

Інтеграція з іншими бібліотеками: Можуть виникнути труднощі при інтеграції з іншими бібліотеками та фреймворками.

Продуктивність для дуже великих додатків: Для додатків з дуже великим обсягом даних та складних інтерфейсів може виникнути потреба в додатковій оптимізації.

Наступним рівнем є серверна структура Express.js [14], яка працює на сервері Node.js. Express.js називає себе «швидким, непереборним, мінімалістичним веб-фреймворком для Node.js», і це справді саме те, що він є [9]. Express.js має потужні моделі для маршрутизації URL-адрес (зіставлення вхідної URL-адреси з функцією сервера) і обробки запитів і відповідей HTTP.

Плюси Express.js:

Легкість та гнучкість. Express.js має мінімалістичний підхід, пропонуючи основний набір функціональності і гнучкість у розширенні функціоналу за допомогою мідлварів та інших модулів.

Швидке розгортання. Завдяки своїй простоті та широкій підтримці спільноти, Express дозволяє швидко розробляти та розгортати веб-додатки.

Інтеграція з іншими технологіями. Express легко інтегрується з іншими технологіями стеку MEAN та MERN, що робить його ідеальним вибором для створення повноцінних веб-додатків.

Маршрутизація. Ефективна система маршрутизації дозволяє легко створювати складні URL-шляхи та обробляти запити.

Мінуси Express.js:

Потреба у додаткових модулях. Для багатьох стандартних функцій, як-от аутентифікація, потрібно встановлювати додаткові модулі.

Середній рівень абстракції. Express пропонує менший рівень абстракції порівняно з деякими іншими фреймворками, що може вимагати більш глибокого розуміння низькорівневих аспектів Node.js та веб-розробки.

Загалом, Express.js [15] є відмінним вибором для швидкої розробки легких веб-додатків і API.

І останнє що варто описати в цьому розділі, платформу з відкритим кодом для виконання високопродуктивних мережевих застосунків – Node.js [10].

Плюси Node.js:

Одномовність. Використання JavaScript для як фронтенду, так і бекенду, дозволяє розробникам використовувати єдину мову програмування по всьому стеку розробки, що спрощує навчання та розуміння проекту.

Асинхронність. Node.js використовує асинхронне програмування, що дозволяє обробляти велику кількість запитів ефективно і зменшує час відгуку сервера.

Широка екосистема. Менеджер пакунків(npm) Node.js, має одну з найбільших бібліотек модулів у світі, що забезпечує велику кількість інструментів та модулів для розширення функціональності.

Масштабованість. Node.js підходить для створення масштабованих та високопродуктивних додатків.

Мінуси Node.js:

Проблеми з продуктивністю при важких обчисленнях. Node.js не є оптимальним вибором для додатків, що виконують важкі CPU-обчислення, оскільки вони можуть блокувати обробку інших запитів.

Нові ревізії та їх невідповідність. Часті оновлення Node.js можуть призвести до проблем зі сумісністю, особливо в старих проектах.

Можна зробити висновок, що Node.js є потужним інструментом для розробки веб-додатків, особливо коли йдеться про розробку в реальному часі або додатків, що вимагають інтенсивного взаємодії з сервером.

2.3 Аналітичний огляд бази даних

Бази даних відіграють ключову роль у зберіганні, обробці та управлінні великими обсягами даних. Вибір відповідної бази даних може значно вплинути на продуктивність, масштабованість та ефективність веб-додатків та інформаційних систем. Проте, згідно концепції MERN стеку увагу буде зосереджено на базі даних MongoDB.

MongoDB високоефективна, не реляційна (NoSQL) база даних, що зберігає дані у вигляді гнучких JSON-подібних документів [11]. Її головна особливість – це здатність легко обробляти великі обсяги різноманітних даних і швидко адаптуватися до змінних вимог до структури даних [12]. MongoDB пропонує масштабованість і гнучкість з можливістю індексації і запитів, що робить її популярним вибором для великих додатків та додатків, які вимагають швидкої обробки великої кількості даних.

На рисунку 2.2 зображено статистику популярності баз даних станом на січень 2024 року [13]. Вона закриває п’ятірку кращих у загальному рейтингу та є першою з числа не реляційних.

417 systems in ranking, January 2024

Rank			DBMS	Database Model	Score		
Jan 2024	Dec 2023	Jan 2023			Jan 2024	Dec 2023	Jan 2023
1.	1.	1.	Oracle	Relational, Multi-model	1247.49	-9.92	+2.33
2.	2.	2.	MySQL	Relational, Multi-model	1123.46	-3.18	-88.50
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	876.60	-27.23	-42.79
4.	4.	4.	PostgreSQL	Relational, Multi-model	648.96	-1.94	+34.11
5.	5.	5.	MongoDB	Document, Multi-model	417.48	-1.67	-37.70
6.	6.	6.	Redis	Key-value, Multi-model	159.38	+1.03	-18.17
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	136.07	-1.68	-5.09
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	132.41	-2.19	-11.16
9.	↑ 10.	↑ 11.	Snowflake	Relational	125.92	+6.04	+8.66
10.	↓ 9.	↓ 9.	Microsoft Access	Relational	117.67	-4.08	-15.69

Рисунок 2.2 – Статистика популярності баз даних

Плюси MongoDB:

Гнучкість схем. MongoDB не вимагає визначення фіксованої схеми. Це означає, що документи в одній і тій же колекції можуть мати різні поля, що забезпечує велику гнучкість при розробці та масштабуванні додатків.

Масштабування. MongoDB підтримує горизонтальне масштабування за допомогою шардування (розподілу даних на різні сервери), що дозволяє легко обробляти великі обсяги даних.

Швидкість і продуктивність. Через свою структуру, орієнтовану на документи, MongoDB може забезпечувати високу швидкість читання та запису,

особливо для операцій, які вимагають роботи з великими обсягами неструктурованих даних.

JSON-подібний формат даних. MongoDB використовує BSON (Binary JSON) для зберігання даних, що робить її ідеальною для JavaScript додатків та легко інтегрується з багатьма сучасними технологіями.

Мова запитів. MongoDB пропонує потужну та гнучку мову запитів, яка підтримує складні запити та агрегацію даних.

Мінуси MongoDB:

Транзакції. Хоча MongoDB додала підтримку мультидокументних транзакцій, її підхід до транзакцій все ще менш зручний порівняно з реляційними базами даних, особливо для складних операцій.

Споживання ресурсів. MongoDB може вимагати більше ресурсів системи, особливо з огляду на використання оперативної пам'яті для ефективної роботи.

Управління даними. Управління великими обсягами даних та їх оптимізація може бути складнішою задачею у MongoDB порівняно з традиційними SQL-базами даних.

Забезпечення сумісності. Оновлення версій MongoDB можуть вимагати додаткових зусиль для забезпечення сумісності, особливо в системах, де використовуються старі версії.

Вартість. При великих масштабах використання, особливо при використанні хмарних версій, вартість може бути вищою, ніж при використанні традиційних SQL-систем.

MongoDB є відмінним вибором для проєктів, які потребують великої гнучкості, швидкості обробки неструктурованих даних та легкого масштабування. Вона ідеально підходить для додатків, які збирають, обробляють та аналізують великі обсяги даних в реальному часі.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Проєктування бази даних

База даних додатку складається з трьох моделей, що представлені у вигляді json-документів. Їх вміст можна переглянути у додатку А.

Модель **userSchema**, представляє модель користувача. Ось докладний опис цієї моделі:

name: Ім'я користувача, яке є обов'язковим полем. Довжина імені повинна бути від 4 до 30 символів.

email: Електронна пошта користувача, яка є унікальною та обов'язковою. Використовується валідатор validator.isEmail для перевірки чи введений рядок є дійсною електронною поштою.

password: Пароль, який є обов'язковим та має мінімальну довжину 8 символів. Він не включається при вибірці даних користувача з бази даних.

avatar: Об'єкт, що зберігає інформацію про аватар користувача, включаючи public_id та url. Обидва поля є обов'язковими.

role: Роль користувача у системі, за замовчуванням встановлена як "user".

createdAt: Дата створення облікового запису користувача, автоматично встановлюється на поточну дату та час.

resetPasswordToken та **resetPasswordExpire**: Використовуються для імплементації функціоналу скидання пароля.

Модель **ProductModel** представляє структуру даних для продукту, а саме:

name: Назва продукту, яка є обов'язковою та обрізається для видалення зайвих пробілів.

description: Опис продукту, обов'язкове поле.

price: Ціна продукту, обов'язкове поле з обмеженням максимальної довжини у 8 символів.

info: Додаткова інформація про продукт, обов'язкове поле.

ratings: Рейтинг продукту, зі значенням за замовчуванням 0.

images: Масив об'єктів зображень, кожен з яких містить product_id та url, обидва поля є обов'язковими.

category: Категорія продукту, обов'язкове поле.

stock: Кількість наявного товару на складі, обов'язкове поле з максимальною довжиною 4 символи та значенням за замовчуванням 1.

numOfReviews: Кількість відгуків про продукт, за замовчуванням 0.

reviews: Масив об'єктів, кожен з яких представляє відгук. Включає поля, такі як userId (посилання на модель користувача), name, ratings, title, comment, recommend (за замовчуванням true), createdAt та avatar. Усі поля, крім recommend, є обов'язковими.

user: містить ObjectId адміністратора, який додав продукт у базу даних. Це поле є обов'язковим і асоціюється з моделлю користувача (userModel).

createdAt: дата створення, що автоматично встановлюється на поточну дату та час при створенні запису продукту.

Модель **OrdersModel** відображає структуру замовлення в базі даних. Ця модель включає в себе різноманітні поля, які представляють деталі замовлення, а саме:

shippingInfo: Об'єкт, що зберігає інформацію про адресу доставки. Включає в себе поля для імені, прізвища, адреси, міста, штату, країни, поштового коду, номера телефону та електронної пошти. Усі поля є обов'язковими для заповнення.

orderItems: Масив об'єктів, де кожен об'єкт представляє окремий товар у замовленні. Кожен товар містить інформацію про назву, ціну, кількість, зображення та посилання на ID продукту. Усі поля обов'язкові.

user: Поле, що містить посилання на ID користувача, який зробив замовлення. Це поле є обов'язковим та асоціюється з моделлю користувача.

paymentInfo: Об'єкт, що зберігає інформацію про оплату, включаючи ID оплати та її статус. Усі поля обов'язкові.

paidAt: Дата, коли було здійснено оплату. Це необов'язкове поле.

itemsPrice, **taxPrice**, **shippingPrice**: Ці поля зберігають вартість товарів, податкові витрати та вартість доставки відповідно. Усі поля мають тип Number і є обов'язковими, зі значенням за замовчуванням 0.

totalPrice: Загальна вартість замовлення, яка є сумою вартості товарів, податків та доставки. Це поле є обов'язковим і має значення за замовчуванням 0.

orderStatus: Статус замовлення, який може бути "Processing", "Delivered", або інші статуси. Це поле є обов'язковим і має значення за замовчуванням "Processing".

deliveredAt: Дата доставки замовлення. Це необов'язкове поле.

createdAt: Дата створення запису замовлення. Поле автоматично заповнюється поточною датою і часом.

Так як MongoDB є нереляційною СУБД, то відобразити відношення між таблицями як у випадку з реляційною СУБД не є можливим.

3.2 Проєктування інтернет-магазину

Перед початком розробки програмного забезпечення для інтернет-магазину необхідно визначити його структуру та встановити зв'язки з сервером та базою даних для основних операцій.

Спершу слід створити архітектуру веб-сайту для користувачів. Вона має включати наступні сторінки:

- Головна сторінка;
- Сторінка з деталями продукту;
- Сторінка кошика для покупок;
- Сторінка для реєстрації та входу в систему;
- Сторінка для оформлення покупки;

- Платіжна сторінка, де користувач може сплатити за товари.

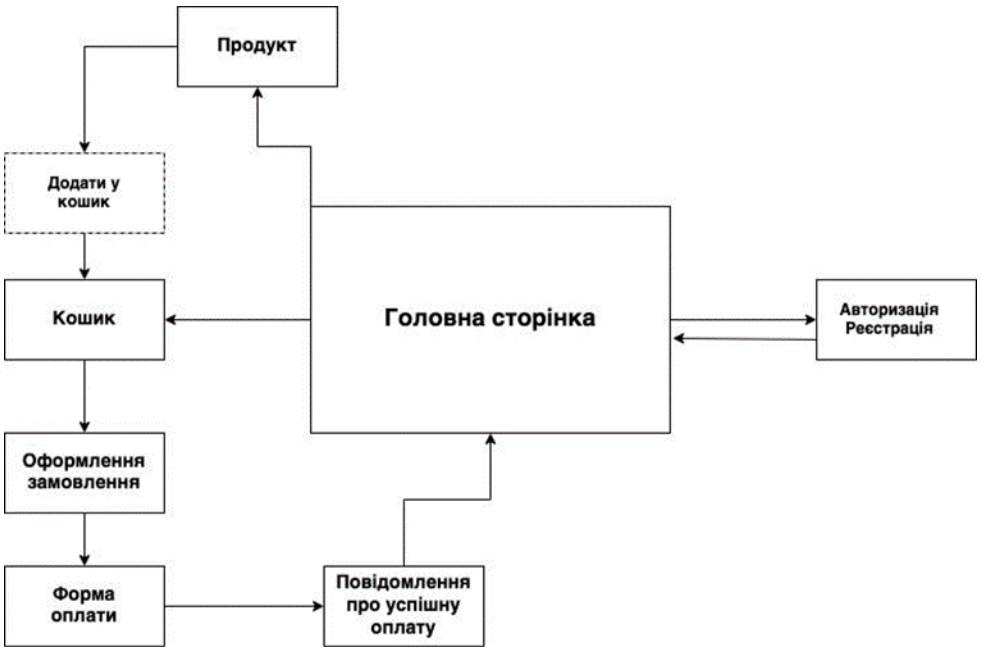

Рисунок 3.1 – Веб-застосунок з представленими раніше сторінками

Тепер розробимо механізм взаємодії нашого веб-сайту з сервером та базою даних. Цей процес включатиме наступні етапи:

1. Відображення товарів на сторінках сайту та їх опису
2. Можливість додавання товарів до кошика користувачем
3. Процес оформлення покупки
4. Реалізація оплати покупок

На рисунку 3.2 представлено візуалізацію того, як відбуватиметься процес з вищеперерахованими етапами.

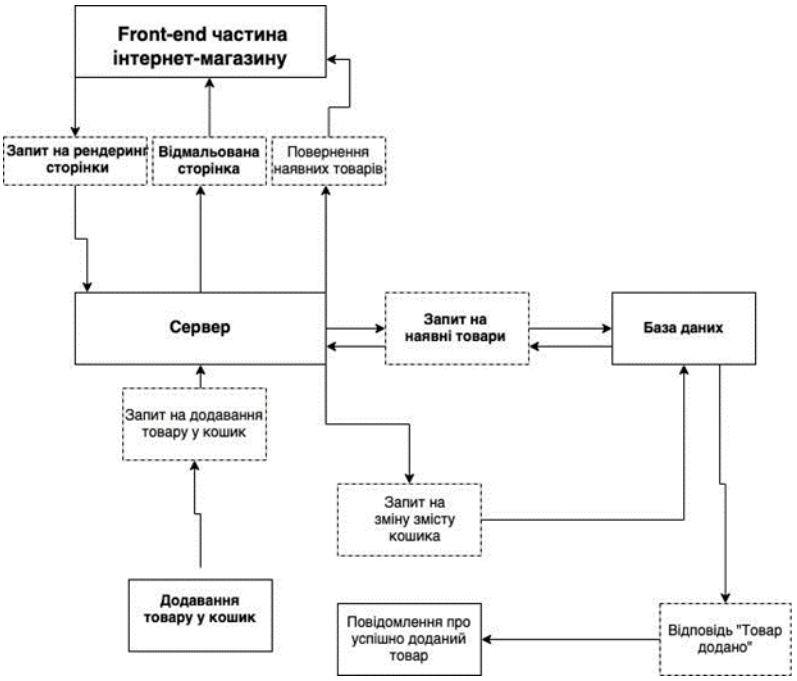

Рисунок 3.2 – візуалізація процесу «Відображення речей у застосунку», «Збереження речей в кошику»

Будь-який інтернет-магазин не може також обійтись без процесу «Оформлення замовлення». В цьому процесі варто передбачити такі етапи:

1. У більшості випадків, замовлення відразу оплачується при оформленні. Отже, нам потрібно інтегрувати платіжну систему. Бажано, декілька платіжних систем, адже комусь зручніше розрахуватися за допомогою Liqpay, хтось віддає перевагу WayForPay, а в наші часи також повинна бути опція оплати криптовалютою -–наприклад, з використанням платформи Binance.
2. Оплата може бути неуспішною, проте такий кейс переважно контролюється платіжною системою. Тим не менш, нашому веб-застосунку також треба контролювати відповіді від платіжної системи.
3. Якщо оплата є успішною – отже, нам потрібно зібрати усі дані про замовлення, передати їх на сервер інтернет-магазину та зберегти їх до бази даних. Після цього, надіслати користувачу повідомлення про те, що його замовлення успішно оформлене.

На рисунку 3.3 можна побачити візуалізацію вище описаного процесу та його етапів.

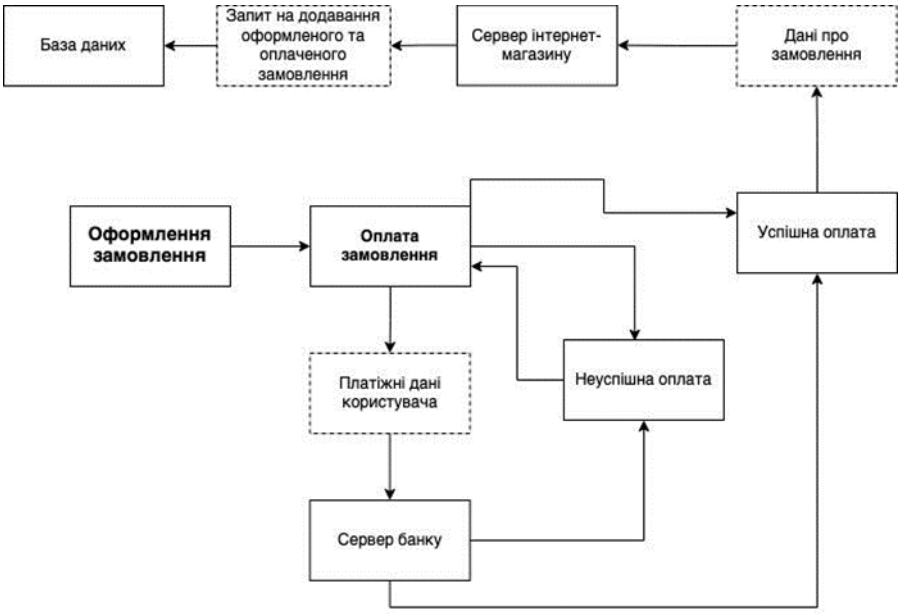

Рисунок 3.3 – візуалізація «Оформлення замовлення»

Візуалізація процесів показує, що архітектурний фундамент нашого веб-сайту ґрунтуватиметься на класичній клієнт-серверній моделі, яка використовує принципи RESTful. У цій моделі клієнтська сторона ініціює запити до сервера, який активно обробляє ці запити та надсилає необхідну інформацію назад клієнту для подальшого відображення.

Це дозволяє клієнтській стороні ефективно управляти візуалізацією даних, підвищуючи таким чином загальну реактивність та швидкість роботи веб-сайту. Ключовим аспектом такої архітектури є її асинхронний характер, де клієнт не змушений чекати завантаження всієї інформації одразу. Натомість, первинний рендеринг головної сторінки відбувається миттєво, а доповнення сторінки даними про товари - поступово, що забезпечує користувачу більш комфортну та приємну взаємодію з сайтом.

Ґрунтуючись на детально розробленій структурі веб-сайту, яка ретельно спланована на попередніх етапах. Далі перейдемо до наступної фази розробки.

Ця фаза передбачає безпосереднє втілення створеного проекту в життя, де кожен компонент інтернет-магазину буде реалізований з урахуванням зазначених вище принципів асинхронності та ефективності.

3.3 Програмна реалізація

Перейдемо безпосередньо до програмної реалізації нашого інтернет-магазину.

Почнемо з базового функціоналу, без якого не може обійтись жоден веб-застосунок, що потребує оформлення замовлення та їх оплату, а саме з аутентифікації.

Варто зазначити, що було прийнято рішення використати один з найпоширеніших способів аутентифікації – з використанням JWT-токену.

Аутентифікація за допомогою JWT (JSON Web Token) – це поширений спосіб забезпечення безпеки, який дозволяє передавати інформацію між клієнтом і сервером у безпечний, зручний і ефективний спосіб. JWT використовуються для ідентифікації користувачів та для перевірки їхніх прав доступу до ресурсів.

JWT складається з трьох основних компонентів:

1. **Header**: Це JSON-об'єкт, який зазвичай містить тип токена (typ) - JWT, та алгоритм хешування (alg), наприклад, HS256, RS256.
2. **Payload**: Це також JSON-об'єкт, який включає запитувані дані, такі як ідентифікатор користувача, ім'я, права доступу тощо. Ця частина може також містити інформацію про час випуску токена (iat), термін його дії (exp), тему (sub) та аудиторію (aud).
3. **Signature**: Щоб створити підпис, алгоритм, вказаний у заголовку, використовується для хешування заголовка, навантаження та секретного ключа сервера.

Ці три компоненти кодуються за допомогою Base64 та розділяються крапками, утворюючи строку у форматі header.payload.signature.

Процес аутентифікації з використанням JWT-токена можна розбити на такі кроки:

1. **Логін користувача**: Користувач вводить свої дані (наприклад, логін та пароль) на клієнтській стороні (наприклад, у веб-браузері).
2. **Генерація та видача JWT**: Сервер перевіряє введені користувачем дані. Якщо вони вірні, сервер створює JWT, в якому шифруються інформація про ідентифікатор користувача та інші релевантні дані, і надсилає цей токен користувачеві.
3. **Використання JWT для доступу до ресурсів**: Користувач включає цей токен як частину заголовків своїх запитів до сервера. Зазвичай токен передається у заголовку Authorization з префіксом Bearer.
4. **Перевірка та авторизація на сервері**: Сервер зчитує токен, декодує його та перевіряє підпис, щоб переконатися, що токен не було змінено. Якщо все в порядку, запит обробляється відповідно до прав користувача, закодованих у токені.

Нижче можна побачити розроблений нами код, який відповідає за JWT-аутентифікацію:

```
export const createAuthToken = (userInfo) => {
    return jwt.sign({
        _id: userInfo._id,
        name: userInfo.name,
        email: userInfo.email,
        isAdmin: userInfo.isAdmin
    }, process.env.JWT_SECRET_KEY || 'defaultsecret', {
        expiresIn: '30d',
    });
};

export const authenticateUser = (req, res, next) => {
    const authHeader = req.headers.authorization;
    if (authHeader) {
        const authToken = authHeader.slice(7); // Removes "Bearer " from the token
        jwt.verify(authToken, process.env.JWT_SECRET_KEY || 'defaultsecret',
(error, decoded) => {
            if (error) {
                res.status(401).send({message: 'Invalid token'});
            } else {
                req.user = decoded;
                next();
            }
```

```
        });
    } else {
        res.status(401).send({message: 'Token not provided'});
    }
};

export const authorizeAdmin = (req, res, next) => {
    if (req.user && req.user.isAdmin) {
        next();
    } else {
        res.status(401).send({message: 'Admin token required'});
    }
};
```

Розберемо по порядку кожну функцію, яка присутня в цьому шматку коду.

1. createAuthToken

Ця функція створює токен безпеки для ідентифікації користувачів на веб-сайті. Після успішного входу користувача, його особисті дані (ID, ім'я, email, інформація про адміністративні права) використовуються для генерації JWT-токена за допомогою бібліотеки jsonwebtoken. Токен шифрує ці дані та встановлює термін дії в 30 днів. Якщо спеціальний секретний ключ не заданий у змінних середовища, використовується стандартний ключ defaultsecret.

2. authenticateUser

Ця функція перевіряє токен, який користувач надсилає у заголовках запитів на сервер. Вона спочатку шукає токен у заголовку authorization, видаляє префікс "Bearer ", а потім намагається перевірити його справжність. Якщо токен дійсний, декодована інформація з нього використовується для ідентифікації користувача. Якщо з токеном щось не так (він змінений або застарів), сервер повертає помилку 401 про недійсний токен. Якщо токен відсутній, користувач отримує помилку, що токен не наданий.

3. authorizeAdmin

Ця функція перевіряє, чи має ідентифікований і автентифікований користувач (за допомогою токена) адміністративні права. Якщо в декодованих даних токена є позначка isAdmin, користувач може продовжувати виконання запиту. Якщо ні, сервер повертає помилку 401 про необхідність адміністративного токена.

Перейдемо тепер до розробки так званих «routers» або ж маршрутизаторів. В них будуть розроблені операції CRUD (Create, Read, Update, Delete) та ендпоїнти до них, за допомогою яких відбуватиметься взаємодія наших front-end та back-end частин.

У якості прикладу, буде представлено розроблені ендпоїнти для замовлень.

```
import express from 'express';
import handleAsync from 'express-async-handler';
import OrderModel from '../models/order.js';
import { isAuth, isAdmin } from '../utils.js';
import Stripe from 'stripe';
import dotenv from 'dotenv';

dotenv.config();

const stripeClient = new Stripe(process.env.STRIPE_SECRET_KEY);

const router = express.Router();

// Fetch all orders, only accessible by admin
router.get('/', isAuth, isAdmin, handleAsync(async (req, res) => {
    const allOrders = await OrderModel.find({}).populate('user', 'name');
    res.send(allOrders);
}));

// Fetch user's orders
router.get('/my', isAuth, handleAsync(async (req, res) => {
    const userOrders = await OrderModel.find({user: req.user._id});
    res.send(userOrders);
}));

// Create a new order
router.post('/', isAuth, handleAsync(async (req, res) => {
    if (req.body.orderItems.length === 0) {
        res.status(400).send({message: "Empty cart"});
    } else {
        const newOrder = new OrderModel({
            orderItems: req.body.orderItems,
            shippingAddress: req.body.shippingAddress,
            paymentMethod: req.body.paymentMethod,
            itemsPrice: req.body.itemsPrice,
            totalPrice: req.body.totalPrice,
            user: req.user._id
        });
        const createdOrder = await newOrder.save();
        res.status(201).send({message: 'Order created', order: createdOrder});
    }
}));

// Fetch a specific order
router.get('/:id', isAuth, handleAsync(async (req, res) => {
    const order = await OrderModel.findById(req.params.id);
    if (order) {
        res.send(order);
    } else {
        res.status(404).send({message: 'Order not found'});
    }
```

```
}));

// Create a Stripe checkout session
router.post('/create-checkout-session', async (req, res) => {
    const session = await stripeClient.checkout.sessions.create({
        payment_method_types: ['card'],
        mode: 'payment',
        success_url: `http://localhost:3000/order/${req.body.url}?success=true`,
        cancel_url: `http://localhost:3000/order/${req.body.url}`,
        line_items: [{
            amount: req.body.amount,
            currency: 'uah',
            name: 'Purchase',
            quantity: 1
        }]
    });
    res.json({id: session.id});
});

// Mark an order as paid
router.put('/:id/pay', isAuth, handleAsync(async (req, res) => {
    const order = await OrderModel.findById(req.params.id);
    if (order) {
        order.isPaid = true;
        order.paidAt = Date.now();
        order.paymentResult = {
            id: req.body.id,
            status: req.body.status,
            update_time: req.body.update_time,
            email_address: req.body.email_address
        };
        const updatedOrder = await order.save();
        res.send({message: 'Order paid', order: updatedOrder});
    } else {
        res.status(404).send({message: 'Order not found'});
    }
}));
// Delete an order
router.delete('/:id', isAuth, isAdmin, handleAsync(async (req, res) => {
    const order = await OrderModel.findById(req.params.id);
    if (order) {
        const deletedOrder = await order.remove();
        res.send({message: 'Order deleted', order: deletedOrder});
    } else {
        res.status(404).send({message: 'Order not found'});
    }
}));

export default router;
```

Проаналізуємо детальніше цей шматок коду. Перша його частина включає налаштування та імпорти. Тут завантажуються необхідні модулі, такі як Express, Mongoose для роботи з базою даних, допоміжні функції для аутентифікації та авторизації користувачів, а також бібліотека Stripe для інтеграції з платіжною

системою. Крім того, ініціалізується Stripe-клієнт за допомогою секретного ключа, отриманого з змінних оточення.

Наступні розділи описують різні функціональні можливості, реалізовані в цьому роутері:

1. **Отримання списку всіх замовлень**: Цей маршрут (GET /) обробляє запити на отримання повного списку всіх замовлень. Доступ до нього обмежений лише для адміністраторів, що перевіряється за допомогою функції isAuth та isAdmin. Якщо перевірка пройшла успішно, всі замовлення завантажуються з бази даних і повертаються, включаючи інформацію про користувачів, які зробили ці замовлення.
2. **Отримання замовлень поточного користувача**: Маршрут GET /my повертає список замовлень, зроблених поточним автентифікованим користувачем. Для цього використовується фільтр за ідентифікатором користувача.
3. **Створення нового замовлення**: Маршрут POST / обробляє запити на створення нового замовлення. Спочатку перевіряється, чи користувач авторизований. Потім дані замовлення, отримані в тілі запиту, використовуються для створення нового документа в базі даних за допомогою моделі OrderModel.
4. **Отримання деталей конкретного замовлення**: Маршрут GET /:id повертає детальну інформацію про замовлення з вказаним ідентифікатором, за умови, що користувач авторизований.
5. **Створення сесії Stripe для оплати**: Маршрут POST /create-checkout-session відповідає за створення сесії Stripe Checkout для обробки платежу за замовлення. Необхідні деталі, такі як сума замовлення, передаються в тілі запиту.
6. **Позначення замовлення як оплаченого**: Маршрут PUT /:id/pay оновлює статус замовлення, позначаючи його як оплачене, і зберігає деталі платежу, отримані з Stripe.

7. **Видалення замовлення**: Маршрут DELETE /:id дозволяє видалити замовлення за вказаним ідентифікатором, але лише для користувачів з адміністраторськими правами.

Цей код забезпечує повний цикл управління замовленнями, від створення та перегляду до оплати та видалення. Він взаємодіє з базою даних через модель OrderModel та інтегрується з Stripe для обробки платежів. Важливою особливістю є реалізація перевірки автентифікації та авторизації користувачів за допомогою функцій isAuth та isAdmin.

Тепер розглянемо деякі компоненти з front-end частини.

Наш проект фронтенду складається з таких основних частин:

1. package.json - Файл, що містить метадані про наш проект, що включає залежності модулів, скрипти для запуску проекту та іншу конфігурацію.
2. package-lock.json - Файл, який забезпечує консистенцію інсталяції залежностей між різними машинами.
3. public - Папка, що зазвичай містить статичні файли, як-от HTML-файли, зображення та іконки.
4. src - Головна папка з вихідним кодом проекту, включає JavaScript або TypeScript файли, компоненти React (якщо використовується React), стилі та інше.

Давайте подивимося більш детально на те, як виглядатиме структура майбутніх папок public і src для кращого розуміння структури нашого фронтенду.

Папка public міститиме наступний файл:

- index.html - основний HTML файл, який служить точкою входу для нашого веб-додатку. Він завантажується браузером як перша сторінка і зазвичай містить посилання на JavaScript файли, стилі та інші ресурси.

Також у нас буде присутня папка src. Структура папки src вказує на використання популярної архітектурної парадигми в React аплікаціях, із

використанням Redux для управління станом. Ось що міститься у кожному з компонентів:

1. App.js - Головний компонент React, який агрегує інші компоненти та є коренем нашого додатку.
2. index.js - Вхідний файл JavaScript, який рендерить компонент App у DOM за допомогою ReactDOM.
3. app.sass - Файл стилів SASS для загального стилю додатку.
4. store.js - Налаштування Redux store, яке забезпечує централізоване сховище для стану всього додатку.
5. **Папки:**
    - actions: Зберігає Redux action creators, які відповідають за відправлення даних до store.
    - components: Містить перевикористовувані UI компоненти React, які можуть бути використані у різних частинах додатку.
    - constants: Зберігає константи, наприклад, типи дій (action types), що використовуються у Redux.
    - reducers: Включає Redux reducers, які обробляють дії відправлені до store і оновлюють стан додатку.
    - screens: Ймовірно, містить компоненти на рівні сторінок, кожен з яких репрезентує певний екран або маршрут у додатку.

Ця структура є досить типовою для сучасних React-Redux проектів, де логіка розділена на чіткі зони відповідальності: управління станом, UI компоненти, стилізація, та бізнес-логіка. Це дозволяє легко масштабувати та підтримувати наш додаток.

Розробимо домашню сторінку нашого інтернет-магазину. Її код виглядатиме наступним чином:

```
import React, { useEffect } from 'react';
import ProductCard from '../components/Product';
import Spinner from '../components/LoadingBox';
import AlertBox from '../components/MessageBox';
import { useDispatch, useSelector } from 'react-redux';
import { fetchProducts } from '../actions/products';
```

```
import ProductCarousel from '../components/Carousel';

function HomeScreen(props) {
  const dispatch = useDispatch();
  const productData = useSelector(state => state.productList);
  const { loading, error, products } = productData;

  useEffect(() => {
    dispatch(fetchProducts());
  }, [dispatch]);

  return (
    <>
      <ProductCarousel></ProductCarousel>
      <div className="product__content container-sm">
        <div className="title">
          <h2>Всі товари</h2>
        </div>
        {loading ? (
          <Spinner></Spinner>
        ) : error ? (
          <AlertBox>{error}</AlertBox>
        ) : (
          <div className="row">
            {products.map(product => (
              <ProductCard key={product._id} product={product} />
            ))}
          </div>
        )}
        <div className="descr container-sm">
          <div className="descr__i">
            <i className="bi bi-shop"></i>
          </div>
          <div className="descr__title">
            <h6>Про магазин eCommHeaven</h6>
          </div>
          <div className="descr__text">
            Lorem ipsum dolor sit amet
          </div>
          <div className="descr__dop">
            <a href="/">Детальніше</a>
          </div>
        </div>
      </div>
    </>
  );
}

export default HomeScreen;
```

Перед нами React-компонент HomeScreen, який відповідає за відображення головної сторінки нашого інтернет-магазину. Цей компонент тісно інтегрований з Redux - системою управління станом, що дозволяє йому динамічно відображати продукти та керувати станом додатка.

Почнемо з імпортів:

- Ми підключаємо необхідні модулі React, включаючи хук useEffect, який дозволить нам виконувати побічні ефекти.
- Імпортуємо різноманітні UI-компоненти, такі як ProductCard, Spinner, AlertBox та ProductCarousel, які було прийнято рішення використовувати для формування візуального представлення.
- Також імпортуємо хуки Redux - useDispatch та useSelector, які допоможуть нам взаємодіяти з Redux-стором.
- І нарешті, імпортується дія Redux fetchProducts, яка буде відповідати за завантаження даних про продукти.

Тепер перейдемо до основної логіки компонента:

1. **Ініціалізація Redux Hooks**:
    - dispatch - хук, що дозволяє нам відправляти дії до Redux-стору.
    - productData - змінна, яка отримує частину стану стору (productList) через useSelector. Цей об'єкт містить інформацію про завантаження, помилки та список продуктів.
2. **useEffect для завантаження даних**:
    - Хук useEffect відправляє дію fetchProducts() при монтуванні компонента. Це запускає процес завантаження продуктів. Залежність [dispatch] гарантує, що ефект виконається лише одного разу.
3. **Рендеринг компонента**:
    - ProductCarousel - відображає карусель зображень або продуктів.
    - **Умовний рендеринг**:
        - loading - якщо дані завантажуються (loading true), показується компонент Spinner.
        - error - якщо сталася помилка (error містить значення), відображається AlertBox з текстом помилки.
        - products - якщо дані успішно завантажені без помилок, рендериться список продуктів за допомогою ProductCard, де

кожному продукту призначається унікальний ключ product._id.

- **Додатковий опис** - під списком продуктів розміщена додаткова інформаційна секція про магазин, з посиланням на більш детальну інформацію.

Цей компонент є ключовим елементом головної сторінки нашого інтернет-магазину. Він відповідає за завантаження даних про продукти, управління станом (наприклад, відображення індикатора завантаження чи повідомлень про помилки) та динамічне рендерення списку продуктів і додаткової інформації.

Більш детальний вихідний код буде представлено у додатках до даної дипломної роботи.

3.4 Тестування застосунку

Тепер потрібно провести тестування готового застосунку. Завдяки цьому буде перевірено, чи працює застосунок справно.

На рисунку 3.4 зображено головну сторінку нашого інтернет-магазину.

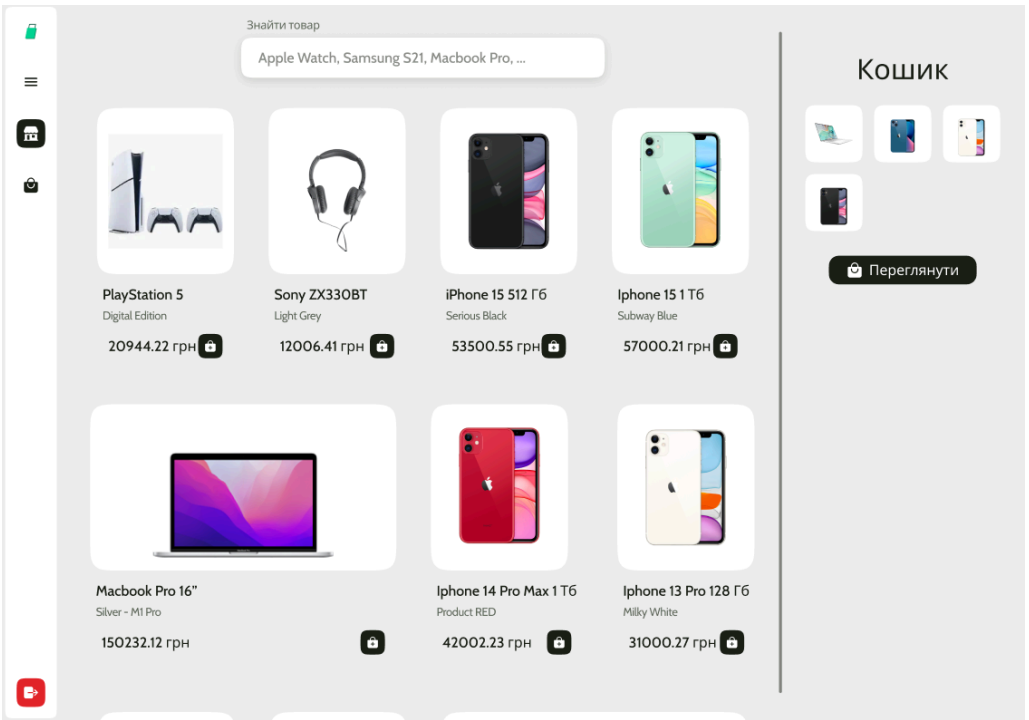

Рисунок 3.4 – Головна сторінка веб-застосунку

На ній можна побачити навігаційну панель, яка спрощує переміщення з однієї сторінки сайту на іншу, а також відображаються різноманітні товари.

Окрім цього, за допомогою навігаційної панелі можна швидко перейти до кошику, в якому зберігаються товари, що користувач обрав для придбання.

На рисунку 3.5 представлено вигляд сторінки «Кошик», а також товари у ньому.

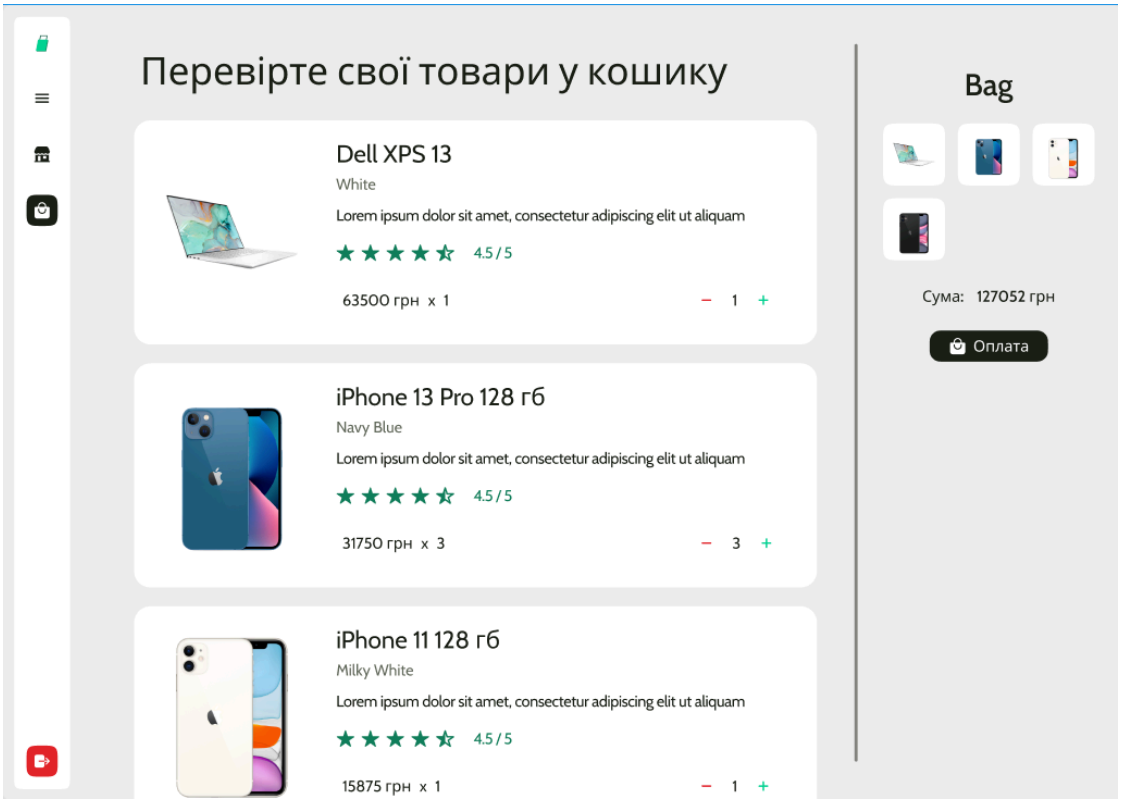

Рисунок 3.5 – сторінка «Кошик» з наявними у ній товарами

Також, як і заявлено, було інтегровано платіжну систему. На рисунку 3.6 та представлено вигляд оформлення замовлення та вибору способу оплати.

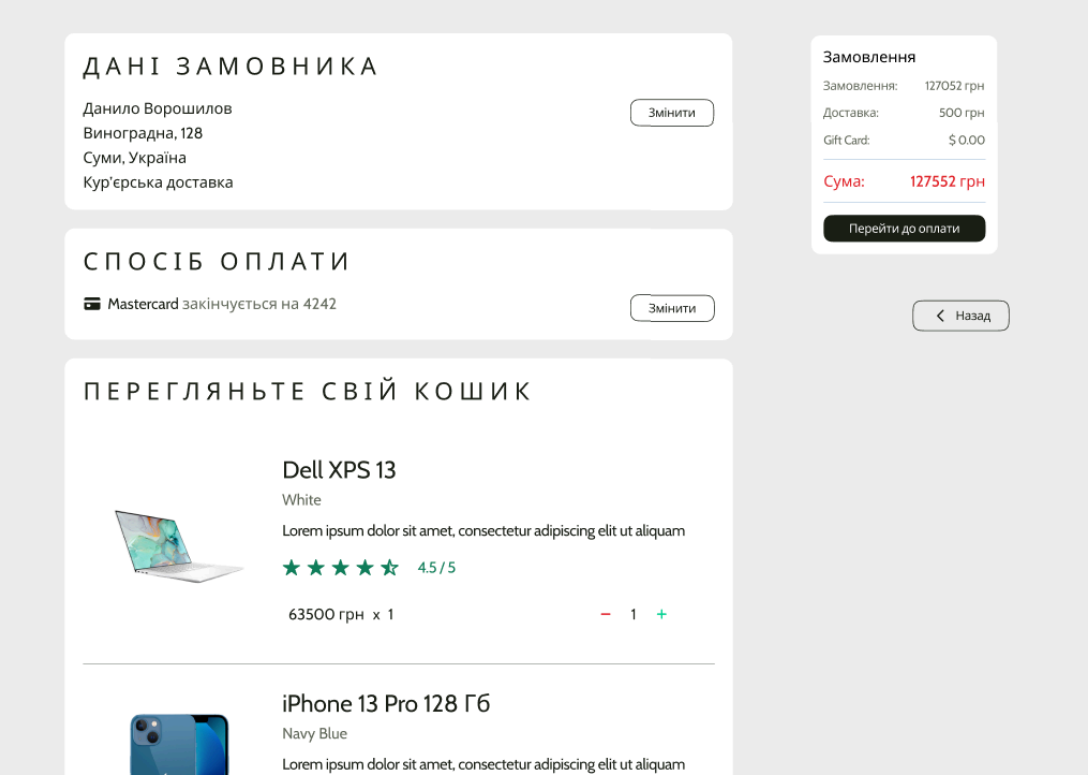

Рисунок 3.6 – форма оформлення замовлення та вибір способу оплати

Тепер переглянемо, як відображається інформація про товар. На рисунку 3.7 представлено приклад відображення товару в нашому інтернет-магазині.

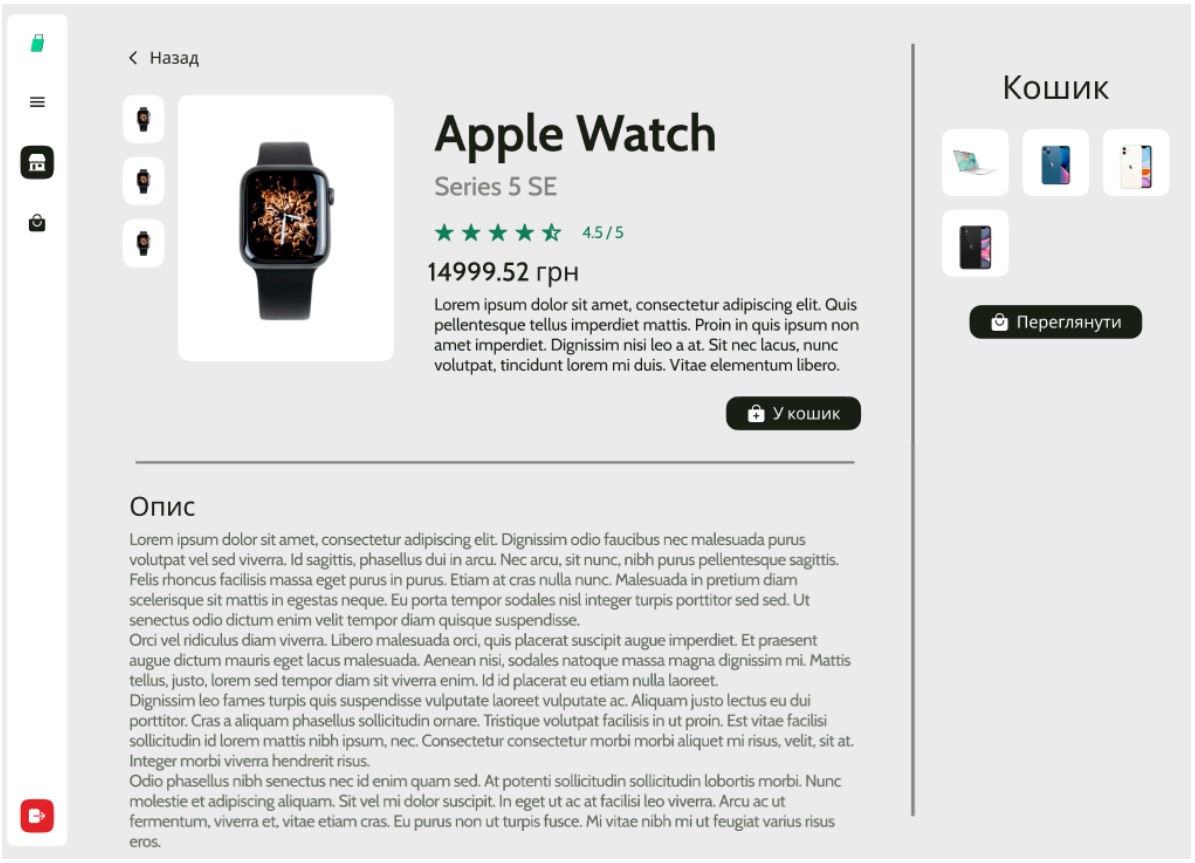

Рисунок 3.7 – приклад відображення товару

Таким чином, було проведенно тестування додатку. Інтернет-магазин працює справно, всі його елементи відображаються коректно, переходи між сторінками відбуваються правильно.

ВИСНОВКИ

У ході виконання дипломної роботи б була розроблена інформаційна технологія з використання стеку MERN (MongoDB, Express.js, React.js, Node.js), що довело ефективність використання стеку MERN для розробки інформаційних систем eCommerce. Цей стек технологій виявився гнучким, продуктивним і зручним у використанні для створення масштабованих веб-додатків.

Завдяки нереляційній базі даних MongoDB, система має високий рівень масштабованості. Express.js та Node.js забезпечують стабільну та швидку серверну частину, в той час як React.js дозволяє створювати інтуїтивно зрозумілі та динамічні клієнтські інтерфейси. Реалізована система відрізняється високою продуктивністю завдяки оптимізації запитів до бази даних та ефективному управлінню державою компонентів на стороні клієнта. Були впроваджені передові методи забезпечення безпеки, включаючи аутентифікацію та авторизацію користувачів, шифрування даних та захист від загроз мережі.

Інформаційна система успішно інтегрується з різними зовнішніми сервісами, такими як системи електронних платежів, сервіси доставки, соціальні мережі, що забезпечує широкі можливості для розвитку та просування бізнесу.

Результатом роботи є повноцінна інформаційна технологія проєктування, яка не лише задовольняє сучасні стандарти для систем подібного типу, але й містить широкі можливості для майбутнього розширення та вдосконалення. Розроблена система здатна стати ефективним інструментом для бізнесу, сприяючи зростанню продуктивності їхньої діяльності та удосконаленню бізнес-процесів.

Виконана дипломна робота відображає використання передових технологій у створенні інформаційних систем і може служити добрим зразком для подальших розробок у даному напрямку.

Для подальшого розвитку проєкту можливе впровадження функціоналу штучного інтелекту для аналізу поведінки користувачів та персоналізації пропозицій, а також розширення мобільної версії додатку.

СПИСОК ЛІТЕРАТУРИ

1. How many people shop online [Електронний ресурс] – Режим доступу: https://www.oberlo.com/statistics/how-many-people-shop-online
2. eCommerce market in Ukraine [Електронний ресурс] – Режим доступу: https://ecommercedb.com/markets/ua/all
3. Companies face an urgent choice: go digital, or go bust [Електронний ресурс] – Режим доступу: https://www.weforum.org/agenda/2020/10/digital-transformation-or-bust/
4. Зростання e-commerce на 77% та збільшення виробництва. Як технології змінюють світову економіку [Електронний ресурс] – Режим доступу: https://forbes.ua/company/zrostannya-e-commerce-na-77-ta-zbilshennya-virobnitstva-yak-tekhnolgii-zminyuyut-svitovu-ekonomiku-12052021-1562
5. Нова доба e-commerce настає просто зараз. Як до неї підготуватися? [Електронний ресурс] – Режим доступу: https://speka.media/nova-doba-ecommerce-nastupaje-prosto-zaraz-yak-do-neyi-pidgotuvatisya-p0y7y9
6. Що таке SPA-додатки [Електронний ресурс] – Режим доступу: https://wezom.com.ua/ua/blog/chto-takoe-spa-prilozheniya
7. Як працює стек MERN? [Електронний ресурс] – Режим доступу: https://www.mongodb.com/mern-stack
8. Стаття про React в енциклопедії «Вікіпедія» [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/React
9. Стаття про Express.js [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs
10. Стаття про Node.js в енциклопедії «Вікіпедія» [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Node.js
11. Стаття про MongoDB в енциклопедії «Вікіпедія» [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/MongoDB
12. Офіційний сайт MongoDB [Електронний ресурс] – Режим доступу: https://www.mongodb.com

13. Рейтинг баз даних від DB-Engines [Електронний ресурс] – Режим доступу: https://db-engines.com/en/ranking

14. Express.js – Official website [Електронний ресурс] – режим доступу: https://expressjs.com/

15. Початок роботи з Express – METANIT.COM [Електронний ресурс] – режим доступу: https://metanit.com/web/nodejs/4.1.php

ДОДАТКИ

Додаток А. Моделі бази даних

orderModel.js

```
const mongoose = require("mongoose");

const orderSchema = new mongoose.Schema({
  shippingInfo: {
    firstName : {
      type: String,
      required: true,
    },
    lastName : {
      type: String,
      required: true,
    },

    address: {
      type: String,
      required: true,
    },
    city: {
      type: String,
      required: true,
    },

    state: {
      type: String,
      required: true,
    },

    country: {
      type: String,
      required: true,
    },
    pinCode: {
      type: Number,
      required: true,
    },
    phoneNo: {
      type: Number,
      required: true,
    },
    email : {
      type: String,
      required: true,
    },

  },

  orderItems: [
    {
      name: {
        type: String,
        required: true,
      },
      price: {
        type: Number,
```

```
      required: true,
    },

    quantity: {
      type: Number,
      required: true,
    },

    image: {
      type: String,
      required: true,
    },

    productId: {
      type: mongoose.Schema.ObjectId,
      ref: "ProductModel",
      required: true,
    },
  },
],

user: {
  type: mongoose.Schema.ObjectId,
  ref: "userModel",
  required: true,
},
paymentInfo: {
  id: {
    type: String,
    required: true,
  },
  status: {
    type: String,
    required: true,
  },
},
// payment timing
paidAt: {
  type: Date,
},

itemsPrice: {
  type: Number,
  required: true,
  default: 0,
},

taxPrice: {
  type: Number,
  required: true,
  default: 0,
},

shippingPrice: {
  type: Number,
  required: true,
  default: 0,
},
totalPrice: {
  type: Number,
  required: true,
  default: 0,
```

```
  },
  orderStatus: {
    type: String,
    required: true,
    default: "Processing",
  },

  deliveredAt: Date,

  createdAt: {
    type: Date,
    default: Date.now,
  },
});

module.exports = mongoose.model("OrdersModel" , orderSchema);
```

ProductModel.js

```
const mongoose  = require("mongoose");
const productSchema = mongoose.Schema({
  name: {
    type: String,
    required: [true, "Please Enter prodcut name"],
    trim: true,
  },
  description: {
    type: String,
    required: [true, "Please Enter prodcut description"],
  },
  price: {
    type: Number,
    required: [true, "Please Enter product Price"],
    maxLength: [8, "Price cannot exceed 9 characters"],
  },
  info: {
    type: String,
    required: [true, "Please Enter product info"],
  },

  ratings: {
    type: Number,
    default: 0,
  },
  images: [
    {
      product_id: {
        type: String,
        required: true,
      },
      url: {
        type: String,
        required: true,
      },
    },
  ],
  category: {
    type: String,
    required: [true, "Please eneter Product Category"],
  },
  Stock: {
    type: Number,
    required: [true, "please Enter product stock"],
```

```
    maxLength: [4, "Stock cannot exceed 4 characters"],
    default: 1,
  },
  numOfReviews: {
    type: Number,
    default: 0,
  },
  reviews: [
    {
      userId: {
        type: mongoose.Schema.ObjectId,
        ref: "userModel",
        required: true,
      },
      name: {
        type: String,
        required: true,
      },
      ratings: {
        type: Number,
        required: true,
      },
      title: {
        type: String,
        required: true,
      },
      comment: {
        type: String,
        required: true,
      },
      recommend: {
        type: Boolean,
        default: true,
      },
      createdAt: {
        type: Date,
        default: Date.now,
      },
      avatar: {
        type: String,
        required: true,
      },
    },
  ],
  user: {
    type: mongoose.Schema.ObjectId, to the db
    ref: "userModel",
    required: true,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
});

const ProductModel  = mongoose.model("ProductModel" , productSchema);
module.exports =ProductModel

userModel.js
const mongoose = require("mongoose");
const validator = require("validator");
const bcrypt = require("bcryptjs");
```

```
const jwt = require("jsonwebtoken");
const crypto = require("crypto");

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Please Enter Your Name"],
    maxLength: [30, "Name cannot exceed 30 characters"],
    minLength: [4, "Name should have more than 4 characters"],
  },
  email: {
    type: String,
    required: [true, "Please Enter Your Email"],
    unique: true,

    validate: [validator.isEmail, "Please Enter a valid Email"],
  },

  password: {
    type: String,
    required: [true, "Please Enter Your Password"],
    minLength: [8, "Password should have more than 4 characters"],
    select: false,
  },
  avatar: {
    public_id: {
      type: String,
      required: true,
    },
    url: {
      type: String,
      required: true,
    },
  },
  role: {
    type: String,
    default: "user",
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
  resetPasswordToken: String,
  resetPasswordExpire: Date,
});

userSchema.pre("save", async function (next) {
  if (this.isModified("password") === false) {
    next();
  }
  this.password = await bcrypt.hash(this.password, 10);
});

userSchema.methods.getJWTToken = function () {
  return jwt.sign({ id: this._id }, process.env.JWT_SECRET, {
    expiresIn: process.env.JWT_EXPIRE,
  });
};

userSchema.methods.comparePassword = async function (password) {
  return await bcrypt.compare(password, this.password);
};
```

```
userSchema.methods.getResetPasswordToken = function () {
  const resetPassToken = crypto.randomBytes(20).toString("hex");
  this.resetPasswordToken = crypto
    .createHash("sha256")
    .update(resetPassToken)
    .toString("hex");
  this.resetPasswordExpire = Date.now() + 15 * 60 * 1000;

  return resetPassToken;
};

const userModel = mongoose.model("userModel", userSchema);
module.exports = userModel;
```

Додаток Б. Backend-частина застосунку

Product.js

```
import express from 'express';
import asyncHandler from 'express-async-handler';
import productsData from '../data.js';
import ProductModel from '../models/product.js';
import { isAdmin, isAuth } from '../utils.js';

const router = express.Router();

// Retrieve all products
router.get('/', asyncHandler(async (req, res) => {
    const products = await ProductModel.find({});
    res.send(products);
}));

// Seed the database with products
router.get('/seed', asyncHandler(async(req, res) => {
    // await ProductModel.deleteMany({});
    const seededProducts = await ProductModel.insertMany(productsData.products);
    res.send({ seededProducts });
}));

// Retrieve a single product by its ID
router.get('/:id', asyncHandler(async (req, res) => {
    const product = await ProductModel.findById(req.params.id);

    if (product) {
        res.send(product);
    } else {
        res.status(404).send({message: 'Product not found'});
    }
}));

// Create a new product
router.post('/', isAuth, isAdmin, asyncHandler(async (req, res) => {
    const product = new ProductModel({
      name: 'testName ' + Date.now(),
      image: '',
      price: 0,
      category: 'sample category',
      brand: 'sample brand',
      countInStock: 0,
```

```
      instock: true,
      raiting: 3,
      numReviews: 0,
      descr: 'sample description',
    });
    const newProduct = await product.save();
    res.send({message: 'Product created', product: newProduct});
}));

// Update an existing product
router.put('/:id', isAuth, isAdmin, asyncHandler(async (req, res) => {
    const productId = req.params.id;
    const product = await ProductModel.findById(productId);

    if (product) {
        product.name = req.body.name;
        product.price = req.body.price;
        product.image = req.body.image;
        product.category = req.body.category;
        product.brand = req.body.brand;
        product.countInStock = req.body.countInStock;
        product.instock = req.body.instock;
        product.descr = req.body.descr;

        const updatedProduct = await product.save();
        res.send({message: 'Product updated', product: updatedProduct});
    } else {
        res.status(404).send({message: "Product not found"});
    }
}));

// Delete a product
router.delete('/:id', isAuth, isAdmin, asyncHandler(async (req, res) => {
    const product = await ProductModel.findById(req.params.id);
    if (product) {
        const deletedProduct = await product.remove();
        res.send({message: 'Product deleted', product: deletedProduct});
    } else {
        res.status(404).send({message: 'Product not found'});
    }
}));

export default router;
```

Upload.js

```
import multer from 'multer';
import express from 'express';
import { isAuth } from '../utils.js';

const uploadRouter = express.Router();

// Налаштування зберігання файлів
const storage = multer.diskStorage({
    destination(req, file, callback) {
        callback(null, 'uploads/');  // Шлях зберігання файлів
    },
    filename(req, file, callback) {
        const fileExtension = file.originalname.split('.').pop();  // Розширення
файлу
```

```
        callback(null, `${Date.now()}.${fileExtension}`);  // Нова назва файлу
    },
});

// Ініціалізація multer з визначеним місцем зберігання
const uploader = multer({ storage });

// Маршрут для завантаження зображень
uploadRouter.post('/', isAuth, uploader.single('image'), (req, res) => {
    res.send(`/${req.file.path}`);  // Повертає шлях до завантаженого файлу
});

export default uploadRouter;
```

User.js

```
import express from 'express';
import data from '../data.js';
import User from '../models/user.js';
import asyncHandler from 'express-async-handler';
import bcrypt from 'bcryptjs';
import { generateToken, isAdmin, isAuth } from '../utils.js';

const userRouter = express.Router();

// Seed users
userRouter.get('/seed', asyncHandler(async (req, res) => {
    // await User.deleteMany({});
    const createdUsers = await User.insertMany(data.users);
    res.send({ createdUsers });
}));

// Sign in
userRouter.post('/signin', asyncHandler(async (req, res) => {
    const user = await User.findOne({ email: req.body.email });
    if (user && bcrypt.compareSync(req.body.password, user.password)) {
        res.send({
            _id: user._id,
            name: user.name,
            email: user.email,
            isAdmin: user.isAdmin,
            token: generateToken(user)
        });
    } else {
        res.status(401).send({ message: 'Invalid email or password' });
    }
}));

// Register
userRouter.post('/register', asyncHandler(async (req, res) => {
    const user = new User({
        name: req.body.name,
        email: req.body.email,
        phone: req.body.phone,
        password: bcrypt.hashSync(req.body.password, 8)
    });
    const createdUser = await user.save();
    res.send({
        _id: createdUser._id,
        name: createdUser.name,
```

```
            email: createdUser.email,
            phone: createdUser.phone,
            isAdmin: createdUser.isAdmin,
            token: generateToken(createdUser)
        });
}));

// Get user by ID
userRouter.get('/:id', asyncHandler(async (req, res) => {
    const user = await User.findById(req.params.id);
    if (user) {
        res.send(user);
    } else {
        res.status(404).send({ message: 'User not found' });
    }
}));

// Update user profile
userRouter.put('/profile', isAuth, asyncHandler(async (req, res) => {
    const user = await User.findById(req.user._id);
    if (user) {
        user.name = req.body.name || user.name;
        user.email = req.body.email || user.email;
        user.phone = req.body.phone || user.phone;
        if (req.body.password) {
            user.password = bcrypt.hashSync(req.body.password, 8);
        }
        const updatedUser = await user.save();
        res.send({
            _id: updatedUser._id,
            name: updatedUser.name,
            email: updatedUser.email,
            phone: updatedUser.phone,
            isAdmin: updatedUser.isAdmin,
            token: generateToken(updatedUser)
        });
    }
}));

// List all users
userRouter.get('/', isAuth, isAdmin, asyncHandler(async (req, res) => {
    const users = await User.find({});
    res.send(users);
}));

// Delete a user
userRouter.delete('/:id', isAuth, isAdmin, asyncHandler(async (req, res) => {
    const user = await User.findById(req.params.id);
    if (user && user.email !== 'admin@gmail.com') {
        const deletedUser = await user.remove();
        res.send({ message: 'User deleted', user: deletedUser });
    } else {
        res.status(404).send({ message: 'User not found or cannot delete admin'
});
    }
}));

// Update a user
userRouter.put('/:id', isAuth, isAdmin, asyncHandler(async (req, res) => {
    const user = await User.findById(req.params.id);
    if (user) {
        user.name = req.body.name || user.name;
```

```
        user.email = req.body.email || user.email;
        user.phone = req.body.phone || user.phone;
        user.isAdmin = req.body.isAdmin !== undefined ? req.body.isAdmin :
user.isAdmin;
        const updatedUser = await user.save();
        res.send({ message: 'User updated', user: updatedUser });
    } else {
        res.status(404).send({ message: 'User not found' });
    }
}));

export default userRouter;
```

Server.js

```
import express from 'express';
import mongoose from 'mongoose';
import dotenv from 'dotenv';
import path from 'path';
import userRouter from './routers/user.js';
import productRouter from './routers/product.js';
import orderRouter from './routers/order.js';
import uploadRouter from './routers/upload.js';

// Load environment variables
dotenv.config();

// Create Express app
const app = express();

// Middleware to parse JSON and urlencoded data
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Connect to MongoDB
mongoose.connect(process.env.MONGODB_URL || 'mongodb://localhost/avhuta-store', {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    useCreateIndex: true
});

// Set port from environment variables or default to 5000
const port = process.env.PORT || 5000;

// API routes
app.use('/api/uploads', uploadRouter);
app.use('/api/users', userRouter);
app.use('/api/products', productRouter);
app.use('/api/orders', orderRouter);

// Route to provide Stripe secret key
app.get('/api/config/stripe', (req, res) => {
    res.send({ stripeSecretKey: process.env.STRIPE_SECRET_KEY || 'sb' });
});

// Setup directory path for static files
const __dirname = path.resolve();
app.use('/uploads', express.static(path.join(__dirname, 'uploads')));

// Root endpoint
app.get('/', (req, res) => {
    res.send('Server is running smoothly');
```

```
});

// Error handling middleware
app.use((err, req, res, next) => {
    res.status(500).send({ message: err.message });
});

// Start the server
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}`);
});
```

Додаток В. Frontend-частина застосунку

Actions/cart.js

```
import axios from 'axios';
import {
  CART_ADD_ITEM,
  CART_REMOVE_ITEM,
  CART_SAVE_PAYMENT_METHOD,
  CART_SAVE_SHIPPING_ADDRESS
} from '../constants/cartConstants';

// Action to add a product to the cart
export const addToCart = (productId, qty) => async (dispatch, getState) => {
    const { data } = await axios.get(`/api/products/${productId}`);
    dispatch({
        type: CART_ADD_ITEM,
        payload: {
          product: data._id,
          name: data.name,
          image: data.image,
          price: data.price,
          countInStock: data.countInStock,
          qty
        },
    });
    // Store cart items in local storage
    localStorage.setItem('cartItems', JSON.stringify(getState().cart.cartItems));
};

// Action to remove a product from the cart
export const removeFromCart = (productId) => (dispatch, getState) => {
  dispatch({
    type: CART_REMOVE_ITEM,
    payload: productId
  });
  // Update local storage
  localStorage.setItem('cartItems', JSON.stringify(getState().cart.cartItems));
};

// Action to save the shipping address
export const saveShippingAddress = (addressData) => (dispatch) => {
  dispatch({
    type: CART_SAVE_SHIPPING_ADDRESS,
    payload: addressData
  });
  // Store shipping address in local storage
  localStorage.setItem('shippingAddress', JSON.stringify(addressData));
};
```

```
// Action to save the payment method
export const savePaymentMethod = (paymentData) => (dispatch) => {
  dispatch({
    type: CART_SAVE_PAYMENT_METHOD,
    payload: paymentData
  });
  // Payment method is currently not persisted to local storage
};
```

Screens/PlaceOrderScreen.js

```
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { createOrder } from '../actions/orderActions';
import CheckoutSteps from '../components/CheckoutSteps';
import { ORDER_CREATE_RESET } from '../constants/orderConstants';
import LoadingBox from '../components/LoadingBox';
import MessageBox from '../components/MessageBox';

const PlaceOrderScreen = ({ history }) => {
    const cart = useSelector(state => state.cart);
    const { shippingAddress, paymentMethod, cartItems } = cart;

    // Redirect user if payment method not set
    if (!paymentMethod) {
        history.push('/payment');
    }

    // Calculate prices
    const toPrice = (num) => Number(num.toFixed(2)); // Helper function to round
numbers
    const itemsPrice = toPrice(cartItems.reduce((a, c) => a + c.qty * c.price,
0));
    const shippingPrice = cart.shippingPrice || 0; // Assuming shippingPrice is
part of the cart object
    const taxPrice = toPrice(0.15 * itemsPrice); // Assuming tax rate is 15%
    const totalPrice = itemsPrice + shippingPrice + taxPrice;

    const dispatch = useDispatch();
    const { loading, success, error, order } = useSelector((state) =>
state.orderCreate);

    const placeOrderHandler = () => {
        dispatch(createOrder({ ...cart, orderItems: cartItems, itemsPrice,
shippingPrice, taxPrice, totalPrice }));
    };

    // Redirect upon success
    useEffect(() => {
        if (success) {
            history.push(`/order/${order._id}`);
            dispatch({ type: ORDER_CREATE_RESET });
        }
    }, [dispatch, order, history, success]);

    return (
        <div className="container">
            <CheckoutSteps step1 step2 step3 step4 />
            <h2>Order Confirmation</h2>
            <div className="row">
                <div className="col">
                    <h3>Shipping</h3>
                    <p>
```

```
                        <strong>Name:</strong> {shippingAddress.fullName}<br />
                        <strong>City:</strong> {shippingAddress.city}<br />
                        <strong>Address:</strong> {shippingAddress.address}
                    </p>
                    <button type="button" onClick={placeOrderHandler}
disabled={cartItems.length === 0} className="primary block">
                        Place Order
                    </button>
                    {loading && <LoadingBox />}
                    {error && <MessageBox variant="danger">{error}</MessageBox>}
                </div>
                <div className="col">
                    <h3>Order Items</h3>
                    {cartItems.map((item) => (
                        <div key={item.product} className="row">
                            <div>
                                <img src={item.image} alt={item.name} />
                            </div>
                            <div>
                                {item.name} {item.qty} x ${item.price} =
${item.qty * item.price}
                            </div>
                        </div>
                    ))}
                    <div className="row">
                        <h3>Order Summary</h3>
                        <div>Items: ${itemsPrice}</div>
                        <div>Shipping: ${shippingPrice}</div>
                        <div>Tax: ${taxPrice}</div>
                        <div>Total: ${totalPrice}</div>
                    </div>
                </div>
            </div>
        </div>
    );
};

export default PlaceOrderScreen;
```