Міністерство освіти і науки України

Кам’янець-Подільський національний університет імені Івана Огієнка

Фізико-математичний факультет

Кафедра комп’ютерних наук

Дипломна робота бакалавра

з теми**: «Розробка системи управління інтернет магазином на основі Node.js»**

Виконав: здобувач вищої освіти групи KN1-B20

спеціальності 122 Комп’ютерні науки

Музика Іван Володимирович

(прізвище, ім’я та по батькові здобувача вищої освіти)

Керівник: Понеділок Вадим Віталійович, кандидат технічних наук, старший викладач

(прізвище, ім’я та по батькові, науковий ступінь, вчене звання керівника)

Рецензент: Оптасюк Сергій Васильович, доцент, кандидат фізико-математичних наук

(прізвище, ім’я та по батькові, науковий ступінь, вчене звання рецензента)

м. Кам’янець-Подільський – 2024 р.

ЗМІСТ

АНОТАЦІЯ 3
ВСТУП 5
РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ КЕРУВАННЯ ІНТЕРНЕТ-МАГАЗИНОМ 7
1.1 Огляд сучасних систем керування інтернет-магазинами 7
1.2 Порівняльний аналіз функціональності та продуктивності 8
Висновки до Розділу 1 11
РОЗДІЛ 2. СТВОРЕННЯ БАЗИ ДАНИХ ДЛЯ ІНТЕРНЕТ-МАГАЗИНУ 12
2.1 Вибір системи управління базами даних 12
2.2 Проектування структури бази даних 13
2.3 Висновки до Розділу 2 17
РОЗДІЛ 3. РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ ІНТЕРНЕТ-МАГАЗИНУ 18
3.1 Реалізація серверної частини на Node.js 18
3.2 Забезпечення безпеки 37
3.3 Висновки до Розділу 3 41
РОЗДІЛ 4. ПЕРЕВІРКА І ТЕСТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ ІНТЕРНЕТ МАГАЗИНУ 42
4.1 Види тестування 42
4.2 Тестування серверної частини 42
4.3 Висновки до розділу 4 52
ВИСНОВКИ 53
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 54
ДОДАТКИ 55
Додаток А 55
Додаток Б 59
Додаток В 62
Додаток Г 65

АНОТАЦІЯ

Дипломна робота “ Розробка системи управління інтернет магазином на основі Node.js”

Кам’янець-Подільський національний університет імені Івана Огієнка

Кафедра комп’ютерних наук

Студент — Музика Іван Володимирович

Керівник — кандидат педагогічних наук, старший викладач, Понеділок Вадим Віталійович

Ця робота присвячена аналізу, проектуванню та реалізації системи керування інтернет-магазином. У першому розділі проводиться огляд сучасних систем керування інтернет-магазинами, здійснюється порівняльний аналіз їх функціональності та продуктивності. Це дозволяє виявити сильні та слабкі сторони існуючих рішень.

Другий розділ зосереджений на створенні бази даних для інтернет-магазину. Розглядаються критерії вибору системи управління базами даних та проектується структура бази даних, яка забезпечить ефективне зберігання та управління даними.

Третій розділ присвячений реалізації серверної частини інтернет-магазину на платформі Node.js. Також увага приділяється забезпеченню безпеки, що є критично важливим для захисту даних користувачі.

У четвертому розділі проводиться перевірка та тестування серверної частини інтернет-магазину. Розглядаються різні види тестування та надаються результати тестування, що підтверджують надійність та ефективність розробленої системи.

Робота надає комплексний підхід до створення інтернет-магазину, охоплюючи всі ключові аспекти від аналізу та проектування до реалізації та тестування, що забезпечує надійну та продуктивну платформу для електронної комерції.

ВСТУП

Актуальність теми: В наш час інтернет з кожним роком все сильніше пов’язується з бізнесом, за рахунок чого змінюється тактика його ведення та його вигляд, що дає можливості використовувати нові унікальні технології для його розвитку.

Однією з таких унікальних технологій стали інтернет-магазини які дозволили власникам бізнесу вести свої справи немаючий навіть фізичного місця розташування магазину. Найбільше актуальність інтернет магазинів люди відчули під час пандемії коли можливості купувати товари в фізичних магазинах просто не було можливим. Саме в цей час люди насправді задумались що розвиток електронної комерції несе в собі дуже багато можливостей не тільки в сфері інтернет-магазинів а й в загалом.

Одне можна сказати точно що розвиток онлайн бізнесу є актуальним не лише для України ай взагалі для всього світу так як Інтернет набуває все більше користувачів з кожним днем.

Об’єкт дослідження: системи управління інтернет-магазинами.

Предмет дослідження: розробка та впровадження функціональних можливостей системи управління інтернет-магазином, що використовує Node.js.

Мета і завдання дослідження: Метою даної роботи є розробка ефективної, масштабованої та безпечної системи для управління інтернет-магазином, що відповідає сучасним вимогам ринку. Для досягнення цієї мети потрібно виконати наступні завдання: Аналіз існуючих систем керування інтернет-магазином, Реалізація Backend серверу, Проведення тестування створеного функціоналу.

Методи дослідження: Для досягнення мети та вирішення поставлених завдань будуть використані наступні методи: Аналіз літературних джерел та вивчення наукових статей про системи управління інтернет-магазинами;

Розробка програмного забезпечення на мові програмування JavaScript[1] за допомогою Node.js; Проведення Тестування програмного забезпечення для перевірки його працездатності та відповідності вимогам.

Практичне значення одержаних результатів полягає у тому, що розроблена система управління інтернет магазином на основі Node.js може значно полегшити та оптимізувати процеси продажу товарів чи послуг в онлайн середовищі для українських підприємств. Зокрема, ця система дозволить:

1. Підвищити ефективність: Інтеграція системи дозволить автоматизувати багато рутинних операцій, таких як обробка замовлень, ведення складського обліку, та зменшити час, необхідний для виконання вище зазначених завдань.

2. Покращити безпеку: Використання Node.js дозволяє розробити систему з високим рівнем захисту даних, що є критично важливим для онлайн-бізнесу, оскільки дозволяє запобігати можливим загрозам інформаційної безпеки та забезпечення захисту конфіденційної інформації користувача.

3. Забезпечити масштабованість: Система розроблена з урахуванням можливостей масштабування, що дозволить компаніям адаптувати свій бізнес до зростання обсягів продажів та впоратися зі збільшено навантаженістю.

4. Підвищення конкурентоспроможності: Впровадження сучасних технологій управління інтернет-магазином допоможе підприємствам збільшити свою привабливість для клієнтів та підвищити рівень задоволення їх потреб.

5. Підтримати розвиток електронної комерції в Україні: Впровадження такої системи сприятиме зростанню сектору електронної комерції в Україні, створюючи нові можливості для бізнесу та споживачів.

Отже, результати дослідження та розроблення системи мають практичне значення для українських підприємств, допомагаючи їм покращити ефективність, безпеку та конкурентоспроможність свого бізнесу в онлайн-середовищі.

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ КЕРУВАННЯ ІНТЕРНЕТ-МАГАЗИНОМ

1.1 Огляд сучасних систем керування інтернет-магазинами

Інтернет-магазини сьогодні є невід'ємною частиною сучасної економіки. Вони забезпечують підприємствам доступ до широкої аудиторії, знижуючи витрати на утримання фізичних магазинів. Сучасні системи керування інтернет-магазинами відіграють ключову роль у забезпеченні функціонування цих магазинів. Серед найбільш популярних систем можна виділити Magento, Shopify, WooCommerce, OpenCart, PrestaShop та власні рішення, такі як система, використовувана інтернет-магазином Rozetka.

Magento є однією з найпотужніших платформ, що надає багаті можливості для налаштування та масштабування. Вона підтримує великі каталоги товарів та високу кількість одночасних користувачів. Magento пропонує широкі можливості інтеграції з різними платіжними системами, службами доставки та маркетинговими інструментами. Однак, її використання вимагає значних технічних знань та ресурсів, що може бути викликом для невеликих компаній. Magento також має значні вимоги до хостингу та оптимізації для забезпечення високої продуктивності.

Shopify є хмарною платформою, що пропонує простоту у використанні та швидкість в налаштовуванні. Вона має багатий набір інтеграцій та додатків, що робить її привабливою для малого та середнього бізнесу. Shopify надає можливість швидкого створення інтернет-магазину з використанням шаблонів та інтуїтивно зрозумілих інструментів управління. Однак, обмежені можливості налаштування можуть стати перепоною для великих підприємств, які потребують більш гнучких рішень. Також, Shopify вимагає щомісячної оплати, що може бути додатковим фінансовим навантаженням для бізнесу.

WooCommerce є плагіном для WordPress, що дозволяє швидко створювати інтернет-магазини. Вона популярна серед малого бізнесу завдяки

легкості у використанні та широкими можливостями в розширені. WooCommerce підтримує велику кількість безкоштовних та платних плагінів, що дозволяють додавати нові функції та інтеграції. Однак, велика кількість плагінів може впливати на продуктивність, особливо на великих сайтах з великим трафіком. Крім того, WooCommerce потребує регулярного оновлення та технічної підтримки для забезпечення безпеки та стабільної роботи.

OpenCart та PrestaShop також є популярними рішеннями, що пропонують широкий спектр функціональних можливостей та налаштувань. Вони підходять для середнього та великого бізнесу, однак вимагають певних технічних знань для ефективного використання. OpenCart відомий своєю простотою в установці та управлінні, а також великою кількістю доступних модулів і розширень. PrestaShop, у свою чергу пропонує більше можливостей для налаштування та гнучкості, але може бути більш складним у налаштуванні та підтримці.

Rozetka використовує власне рішення для управління своїм інтернет-магазином, яке включає спеціалізовані модулі та інтеграції, що дозволяють ефективно обслуговувати великий обсяг клієнтів та замовлень. Система розроблена з урахуванням специфіки українського ринку та забезпечує високу продуктивність і масштабованість. Власне рішення Rozetka дозволяє гнучко налаштовувати функціональні можливості магазину, інтегрувати з локальними платіжними системами та службами доставки, а також забезпечувати високу швидкість обробки замовлень. Це дозволяє Rozetka ефективно конкурувати на ринку та надавати високий рівень сервісу своїм клієнтам.

1.2 Порівняльний аналіз функціональності та продуктивності

Для проведення порівняльного аналізу ключовими критеріями були обрані такі характеристики інтернет-магазину: продуктивність, можливості налаштування, безпека, масштабованість та вартості.

Продуктивність: Magento та Shopify працюють краще за рахунок оптимізації з боку серверних ресурсів. Magento використовує потужну архітектуру, яка підтримує великий каталог продуктів, які можуть обробляти великі обсяги запитів. Будучи хмарною платформою, Shopify розподіляє навантаження на ваші сервери і забезпечує стабільну роботу навіть тоді коли трафік сильно навантажений. WooCommerce має велику кількість плагінів особливо на досить великих сайтах де доволі великий трафік, що може спричинити до втрати продуктивності інтернет-магазину. Система Rozetka, була створена спеціально для потреб великих інтернет-магазинів тому вона є гарно оптимізованою і навіть при великому трафіку можна домогтися високої ефективності.

Можливості налаштування: Magento є безперечним лідером за показником можливості налаштувань, дозволяючи повністю адаптувати систему під потреби бізнесу завдяки відкритому коду та широкому спектру модулів і плагінів.WooCommerce та OpenCart також пропонують значні можливості налаштування через плагіни та модулі, але їхня функціональність може бути може бути обмежена продуктивністю та складністю інтеграції. Shopify обмежений рамками своєї хмарної платформи, надаючи обмежені можливості налаштування через додатки та теми з Shopify App Store. Rozetka, завдяки власній розробці, має можливість гнучко налаштовувати функціональність відповідно до своїх потреб, що дозволяє їй швидко реагувати на зміну вимог ринку.

Безпека: Усі розглянуті системи надають основні засоби безпеки, такі як підтримка SSL(Secure Sockets Layers – це протокол безпеки, який використовується для захисту зв’язків між веб-сайтом та браузером користувача), захист від SQL-ін’єкцій та регулярні оновлення безпеки. Проте, безпека Magento є більш складною і вимагає додаткових ресурсів для підтримки через велику кількість налаштувань і можливостей. Shopify, як хмарна платформа забезпечує високий рівень безпеки за рахунок

централізованого управління та постійних оновлень від провайдера. WooCommerce та OpenCart потребують регулярних оновлень і додаткових плагінів для забезпечення безпеки. Rozetka також забезпечує високий рівень безпеки, враховуючи специфічні вимоги українського ринку та особливості локального законодавства.

Масштабованість: Magenta та Shopify забезпечують високу масштабованість завдяки своїм архітектурам. Magenta з його потужною архітектурою, може обробляти великий обсяг даних і трафіку, дозволяючи підприємствам масштабуватися без значних проблем. Shopify, як хмарна платформа, автоматично масштабується відповідно до навантаження на сервери, забезпечуючи безперебійну роботу. WooCommerce може зіткнутися з обмеженнями при значному зростанні обсягів даних та трафіку, вимагаючи додаткових оптимізацій і ресурсів для підтримки стабільної роботи. Система Rozetka, розроблена спеціально для великого інтернет-магазину, забезпечує масштабованість для обслуговування великої кількості клієнтів та товарів, дозволяючи обробляти значний обсяг замовлень і підтримувати високу продуктивність.

Вартість: Shopify вимагає щомісячної оплати, яка включає хостинг, технічну підтримку та регулярні оновлення, Ця модель дозволяє підприємствам передбачувано панувати свої витрати, але може стати додатковим фінансовим навантаженням для малого бізнесу. Magento, будучи більш складною системою, вимагає значних інвестицій на впровадження, налаштування та підтримку, включаючи витрати на хостинг, розробку та технічну підтримку. WooCommerce є безкоштовним плагіном, але потребує окремих витрат на хостинг, плагіни та технічну підтримку. Власні рішення такі як система Rozetka, можуть бути дорогими у розробці та підтримці, але забезпечують високу ефективність та відповідність специфічним вимогам бізнесу, дозволяючи мінімізувати довгострокові витрати завдяки оптимізації під конкретні потреби компанії.

Проведений аналіз показав, що кожна з розглянутих систем має свої переваги та недоліки. Magento є найпотужнішою та гнучкою системою, що підходить для великих підприємств з великими ресурсами. Shopify є ідеальним рішенням для малого та середнього бізнесу завдяки простоті використання та швидкості впровадження WooCommerce підходить для малих підприємств, які вже використовують WordPress. OpenCart є добрим рішенням для середнього бізнесу, що потребує налаштування та масштабування. Система Rozetka демонструє переваги власних рішень, які забезпечують високу продуктивність та адаптивність до специфіки українського ринку.

Висновки до Розділу 1

У першому розділі було проведено огляд сучасних систем керування інтернет-магазинами, таких як Magento, Shopify, WooCommerce, OpenCart та власне рішення Rozetka. Виконаний порівняльний аналіз показав, що кожна з систем має свої унікальні переваги та недоліки, що визначають їхнє використання в різних бізнес-середовищах.

На основі цього аналізу можна зробити висновки, що для успішного управління інтернет-магазином необхідно враховувати специфіку бізнесу та технічні вимоги. Це дозволяє зробити обґрунтований вибір системи, що відповідатиме потребам конкретного підприємства та забезпечить його ефективне функціонування в умовах сучасного ринку електронної комерції.

РОЗДІЛ 2. СТВОРЕННЯ БАЗИ ДАНИХ ДЛЯ ІНТЕРНЕТ-МАГАЗИНУ

2.1 Вибір системи управління базами даних

Під час вибору системи управління базами даних для розробки інтернет-магазину були розглянуті такі СУБД як:

1)MySQL[8]: є однією з найпоширеніших відкритих СУБД. Вона відома своєю швидкістю та простотою використання. MySQL часто використовується для проектів з невеликим або середнім обсягом даних, де важливо забезпечити швидкий доступ до інформації.

2)MongoDB[3]: являється нереляційною базою даних, яка добре підходить для проектів з великим обсягом документоорієнтованих даних. Вона добре підходить для проектів з великим обсягом документів, де потрібна гнучкість у структурі даних.

3)PostgreSQL[5]: відомий своєю стабільністю розширюваністю та дотриманням стандартів SQL. Вона використовується для проектів з великим обсягом даних та високими вимогами до надійності і доступності.

Після ретельної оцінки кожної СУБД було прийнято рішення обрати PostgreSQL, як основний СУБД проекту інтернет-магазину з наступним обґрунтуванням:

Стабільність та надійність — PostgreSQL відомий своєю стабільністю та надійністю. Це робить її відмінним вибором для проектів з великим обсягом даних та важливими вимогами до доступності;

Дотримання стандартів SQL — PostgreSQL активно дотримується стандартів SQL, що робить його більш сумісним з іншими системами та інструментами. Це значно спрощує інтеграцію з іншими рішеннями і забезпечує більшу портативність коду бази даних.

Отже, враховуючи потреби проекту і вимоги до надійності, PostgreSQL виявився найкращим вибором серед розглянутих альтернатив.

2.2 Проектування структури бази даних

Як правило першим кроком у створенні бази даних[6] є вивчення предметної області та урахування головних цілей. Так як моєю головною метою являється створення бази даних для інтернет магазину потрібно було створити таку схему бази даних яка б не мала повторень інформації. В результаті міркувань та вивчення декількох схем, було створено таку схему для моєї бази даних зображення якої можна побачити на Рис. 2.1. Яка забезпечить інтернет-магазин такими можливостями як: швидкий доступ до інформації та можливість швидкого розширення бази даних за потреби.

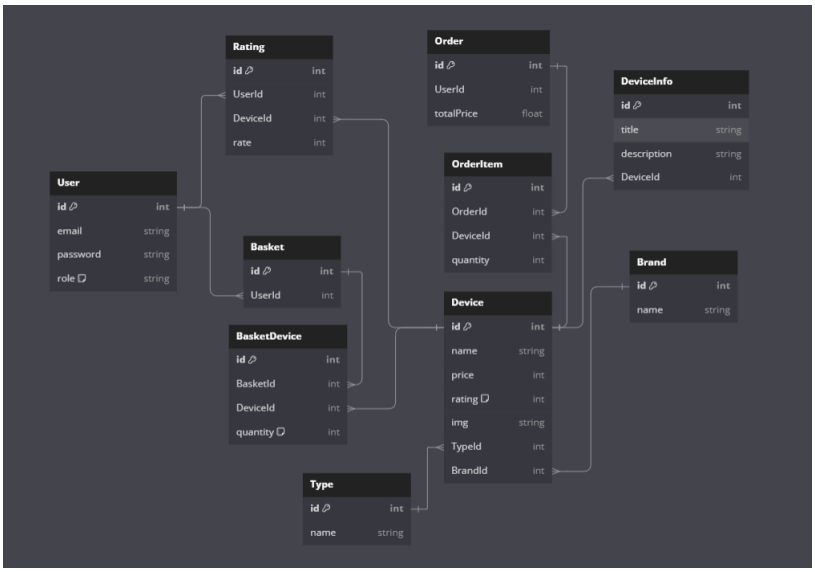

Рис. 2.1 Схема таблиць бази даних

Так як системою управління базою даних було обрано PostgreSQL база даних керується за допомогою ORM бібліотеки Sequelize. Основні таблиці і їх зв’язки було визначено у файлі models.js (див. Додаток А)

Таблиці та їх атрибути:

User (Таблиця для зберігання інформації про користувачів):

id: Ціле число, первинний ключ, автоінкремент.

email: Рядок, унікальне значення.

password: Рядок.

role: Рядок, за замовчуванням «USER»

Basket (таблиця для зберігання кошиків користувачів)

id: Ціле число, первинний ключ, автоінкремент.

userId: Ціле число, зовнішній ключ, посилається на «USER».

BasketDevice (таблиця для зберігання пристроїв у кошиках)

id: Ціле число, первинний ключ, автоінкремент.

basketId: Ціле число, зовнішній ключ, посилається на «Basket».

deviceId: Ціле число, зовнішній ключ, посилається на «Device».

quantity: Ціле число, за замовчуванням 1.

Device (Таблиця для зберігання інформації про пристрої)

id: Ціле число, первинний ключ, автоінкремент.

name: Рядок, унікальне значення, обов’язкове поле.

price: Ціле число, обов’язкове поле.

rating: Ціле число за замовчуванням 0.

img: Рядок, обов’язкове поле.

TypeId: Ціле число, зовнішній ключ посилання на «Type»

BrandId: Ціле число, зовнішній ключ посилання на «Brand»

Type (Таблиця для зберігання типів пристроїв)

id: Ціле число, первинний ключ, автоінкремент.

name: Рядок, унікальне значення, обов’язкове поле.

Brand (Таблиця для зберігання брендів)

id: Ціле число, первинний ключ, автоінкремент.

name: Рядок, унікальне значення, обов’язкове поле.

Rating (Таблиця для зберігання рейтингу пристроїв)

id: Ціле число, первинний ключ, автоінкремент.

userId: Ціле число, зовнішній ключ, посилається на «USER».

deviceId: Ціле число, зовнішній ключ, посилається на «Device».

rate: Ціле число, обов’язкове поле.

DeviceInfo (Таблиця для зберігання додаткової інформації про пристрої)

id: Ціле число, первинний ключ, автоінкремент.

title: Рядок, обов’язкове поле.

description: Рядок, обов’язкове поле.

deviceId: Ціле число, зовнішній ключ, посилається на «Device».

Order (Таблиця для зберігання замовлень)

id: Ціле число, первинний ключ, автоінкремент.

userId: Ціле число, зовнішній ключ, посилається на «USER».

totalPrice: Дійсне число, обов’язкове поле.

OrderItem(таблиця для зберігання деталей замовлення)

id: Ціле число, первинний ключ, автоінкремент.

orderId: Ціле число, зовнішній ключ, посилається на «Order».

deviceId: Ціле число, зовнішній ключ, посилається на «Device».

quantity: Ціле число, обов’язкове поле.

Зв’язки між таблицями:

Користувачі та кошики – користувач має один кошик «User.hasOne(Basket)», кошик належить користувачу «Basket.belongTo(User)»

Користувачі та рейтинг – користувач може мати багато рейтингів «User.hasMany(Rating)», рейтинг належить користувачу «Rating.belongsTo(User)».

Кошик та пристрої в кошиках – кошик може мати багато пристроїв «Basket.hasMany(BasketDevice)», пристрої належать кошику «BasketDevice.belongsTo(Basket)».

Типи та пристрої – Тип може мати багато пристроїв «Type.hasMany(Device)», пристрій належить брендові «Device.belongsTo(Type)».

Бренди та пристрої – бренд може мати багато пристроїв «Brand.hasMany(Device)», пристрій належить бренду «Device.belongsTo(Brand)».

Пристрої та рейтинг – пристрій може мати багато рейтингів «device.hasMany(Rating)», рейтинг належить пристрою «Rating.belongsTo(Device)».

Пристроїв в кошиках – Пристрій може бути у багатьох кошиках «Device.hasMany(BasketDevice)», кошик може мати багато пристроїв «BasketDevice.belongsTp(Device)».

Пристрої та додаткова інформація – пристрій може мати багато додаткової інформації «Device.hasMany(DeviceInfo, {as: ‘info’})», додаткова інформація належить пристрою «DeviceInfo.belongsTo(Device)».

Замовлення та елементи замовлення – замовлення може мати багато елементів «Order.hasMany(OrderItem, {as; ‘item’})», елемент належить замовленню «OrderItem.belongsTo(Order)».

Типи та бренди – тип може мати багато брендів «Type.belongsToMany(Brand, {through: TypeBrand})», Бренд може мати багато типів через проміжну таблицю «Brand.belongsToMany(Type, {through: TypeBrand})».

Зв’язки між таблицями відображають логічні зв’язки між сутностями додатку, забезпечуючи цілісність даних і можливість ефективної взаємодії з інформацією. Всі зв’язки визначені через зовнішні ключі та асоціації один до одного або один до багатьох, відповідно до потреб додатку. Ця структура бази даних забезпечує ефективне зберігання та обробку даних для інтернет-магазину, дозволяючи легко розширювати функціональність та зберігати високу продуктивність системи про роботі з великими обсягами інформації.

2.3 Висновки до Розділу 2

В 2 розділі було розглянуто процес створення бази даних для інтернет магазину. На основі ретельної оцінки систем управління базами даних, таких як MySQL, MongoDB і PostgreSQL, було прийнято рішення обрати PostgreSQL як основну СУБД проекту. Це обрано через його стабільність, надійність та відповідність стандартам SQL, що робить його ідеальним вибором для проектів з великим обсягом даних і високими вимогами до доступності та надійності.

Далі у розділі розглянута структура бази даних, що була спроектована для забезпечення потреб інтернет-магазину. Визначено ключові таблиці, такі як User, Basket, BasketDevice, Device, Type, Brand, Rating, DeviceInfo, Order і OrderItem, а також визначено їх зв’язки через зовнішні ключі і асоціації між ними. Ця структура дозволила забезпечити цілісність даних та ефективну взаємодію міх різними сутностями системи, що є критичним для оптимальної роботи інтернет-магазину.

Таким чином, обрана структура бази даних та вибір PostgreSQL як основної СУБД обґрунтовуються на основі потреб проекту в стабільності, надійності, забезпечені цілісності даних і високої ефективності при обробці високих обсяги інформації.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ ІНТЕРНЕТ-МАГАЗИНУ

3.1 Реалізація серверної частини на Node.js

У цьому розділі буде розглянуто реалізацію практичного завдання а точніше серверної частини інтернет-магазину, використовуючи платформу Node.js[4].

Node.js – це середовище виконання JavaScript з відкритим кодом, побудоване на двигуні V8 від Google Chrome. Це дозволяє запускати JavaScript за межами браузера, роблячи його універсальною платформою для розробки різноманітних застосунків.

Основні характеристики Node.js:

1) Швидкість: Node.js використовує асинхронний, подієвоорієнтовний підхід, що робить його дуже швидким.
2) Масштабованість: Node.js може ефективно обробляти багато одночасних зєднань, що робить його ідеальним для створення масштабованих веб-застосунків та сервісів.
3) Простота: Node.js використовує JavaScript, який є однією з найпоширеніших та найпростіших мов програмування. Це робить Node.js доступним для розробників з різним рівнем досвіду.
4) Гнучкість: Node.js має велике співтовариство та екосистему з безліччю пакетів та бібліотек, які можна використовувати для створення будь-якого типу застосунку.

Для створення інтернет магазину на базі Node.js було обрано такі технології:

1) Express[2]: Веб-фреймворк для Node.js, що спрощує створення серверних додатків.
2) Sequelize[10]: ORM бібліотека для роботи з базою даних PostgreSQL.

3) Jwt (JSON Web Tokens): Для реалізації аутентифікації та авторизації користувачів.
4) Bcrypt[9]: Для хешування паролів.

Першим кроком створення Node.js додатку є ініціалізація проекту, для цього потрібно у консолі вписати такі команди:

mkdir onlone-store

cd online-store

npm init -y

Тут використовується команда npm (Node Package Manager) – це менеджер пакетів JavaScript, який використовується для встановлення, оновлення та видалення пакетів (бібліотек) у проектах Node.js.

Слідуючим кроком потрібно встановити обрані пакети, для цього використовується команда: npm install ezpress sequelize pg pg_hstore bcryptjs jsonwebtoken donenv. Після команди менеджера пакета вводяться всі обрані технології які в поадльшому будуть використовуватись в проекті.

Після цього потрібно налаштувати базу даних, для цього потрібно створити файл під назвою db.js в якому буде міститись вся потрібна інформація для підключення до бази даних.

Приклад коду для файлу db.js:

```
const {Sequelize} = require('sequelize')

module.exports = new Sequelize(
    process.env.DB_NAME, // Назва бази данних
    process.env.DB_USER, // Користувач
    process.env.DB_PASSWORD, // Пароль
    {
        dialect: 'postgres',
        host: process.env.DB_HOST,
        port: process.env.DB_PORT
```

```
    }
)
```

Після налаштування доступу до бази даних можна переходити до створення файлу під назвою models.js (див. Додаток А) в якому міститься вся інформацію про зв’язки і таблиці додатку.

Далі потрібно створити файли маршрутизації для інтернет магазину:

bascketRouter:

```
const Router = require('express');
const router = new Router();
const basketController = require('../controllers/basketController');
const authMiddleware = require('../middleware/authMiddleware');

router.post('/add', authMiddleware, basketController.addToBasket);
router.put('/update', authMiddleware,
basketController.updateBasketItem);
router.delete('/delete/:deviceId', authMiddleware,
basketController.removeFromBasket);
router.delete('/clear', authMiddleware, basketController.clearBasket);
router.get('/', authMiddleware, basketController.getBasketItems);
router.get('/total', authMiddleware, basketController.getBasketTotal);

module.exports = router;
```

brandRouter:

```
const Router = require('express')
const router = new Router()
const brandController = require('../controllers/brandController')
const checkRole = require('../middleware/checkRoleMiddleware')

router.post('/',checkRole('ADMIN'), brandController.create)
router.get('/', brandController.getAll)
```

```
router.delete('/', checkRole('ADMIN'), brandController.delete)

module.exports = router
```

deviceRouter:

```
const Router = require('express');
const router = new Router();
const deviceController = require('../controllers/deviceController');
const checkRole = require('../middleware/checkRoleMiddleware');

router.post('/', checkRole('ADMIN'), deviceController.create);
router.get('/', deviceController.getAll);
router.get('/:id', deviceController.getOne);
router.delete('/:id', checkRole('ADMIN'), deviceController.delete);

module.exports = router;
```

orderRouter:

```
const Router = require('express');
const router = new Router();
const orderController = require('../controllers/orderController');
const authMiddleware = require('../middleware/authMiddleware');

router.post('/', authMiddleware, orderController.create);
router.get('/', authMiddleware, orderController.getAll);
router.get('/:id', authMiddleware, orderController.getOne);

module.exports = router;
```

ratingRouter:

```
const Router = require('express');
const ratingController = require('../controllers/ratingController');
const authMiddleware = require('../middleware/authMiddleware');
const router = new Router();

router.post('/rating', authMiddleware, ratingController.createRating);
router.get('/rating/average/:deviceId',
ratingController.getAverageRating);

module.exports = router;
```

typeRouter:

```
const Router = require('express');
const router = new Router();
const typeController = require('../controllers/typeController');
const checkRole = require('../middleware/checkRoleMiddleware');

router.post('/', checkRole('ADMIN'), typeController.create);
router.get('/', typeController.getAll);
router.delete('/:id', checkRole('ADMIN'), typeController.delete);

module.exports = router;
```

userRouter:

```
const Router = require('express')
const router = new Router()
const UserController = require('../controllers/userController')
const austhMiddleware = require('../middleware/authMiddleware')
const checkRole = require('../middleware/checkRoleMiddleware');

router.post('/registration', UserController.registration)
router.post('/login', UserController.login)
```

```
router.get('/auth', austhMiddleware, UserController.check)
router.post('/change-role', checkRole('ADMIN'),
UserController.changeUserRole);
router.get('/user', checkRole('ADMIN'), UserController.getUsers);

module.exports = router
```

aiRouter:

```
const Router = require('express');
const router = new Router();
const aiController = require('../controllers/aiController');
const checkRole = require('../middleware/checkRoleMiddleware');

router.post('/generate-product', checkRole('ADMIN'),
aiController.generateProduct);

module.exports = router;
```

Ці маршрути потрібні для управління кошиком користувача, брендами, товарами, замовленнями рейтингами, типами товарів, користувачами, та генерацією продуктів за допомогою штучного інтелекту. Вони забезпечують функціонал додавання, оновлення, отримання, видалення та інших операцій з відповідними ресурсами, використовують middleware для автентифікації та авторизації, Але для покращення структури коду, варто додати файл index.js, який буде виступати головним для цих маршрутизаторів, об’єднуючи, їх в один організований модуль.

Index.js:

```
const Router = require('express')
const router = new Router()
const deviceRouter = require('./deviceRouter')
const userRouter = require('./userRouter')
const brandRouter = require('./brandRouter')
```

```
const typeRouter = require('./typeRouter')
const ratingRouter = require('./ratingRouter')
const basketRouter = require('./basketRouter')
const orderRouter = require('./orderRouter');
const aiRouter = require('./aiRouter');

router.use('/user', userRouter)
router.use('/type', typeRouter)
router.use('/brand', brandRouter)
router.use('/device', deviceRouter)
router.use('/rating', ratingRouter)
router.use('/basket', basketRouter)
router.use('/order', orderRouter);
router.use('/ai', aiRouter);

module.exports = router
```

Слідуючим кроком потрібно створити контролери які будуть обробляти логіку яка приходить від роутерів:

BasketController (див. Додаток Б): Цей контролер несе в собі такий функціонал:

1) addToBasket - додає товар до кошика користувача, якщо товар вже є в кошику оновлює кількість, якщо кошика для користувача ще немає, створює новий.
2) updateBasketItem – оновлює кількість товарів в кошику користувача, якщо товар не знайдений повертає помилку.
3) removeFromBasket – видаляє товар з кошика користувача за його ідентифікатором, якщо товар не знайдено у кошику, повертає помилку.
4) getBasketItem – отримує всі товари з кошику користувача.
5) clerBasket – очищяє всі товари користувача, якщо кошик не знайдено повертає помилку.

6) getBasketTotal – розраховує загальну вартість товарів у кошику користувача, повертає загальну вартість.

BrandController:

```
const {Brand} = require('../models/models')
const ApiError = require('../error/apiError')

class BrandController {
    async create(req, res) {
        const {name} = req.body
        const brand = await Brand.create({name})
        return res.json(brand)
    }

    async getAll(req, res) {
        const brands = await Brand.findAll()
        return res.json(brands)
    }

    async delete(req, res) {
        const { ids } = req.body;  // Очікуємо масив ідентифікаторів
брендів
        const deletedCount = await Brand.destroy({ where: { id: ids }
});
        if (deletedCount) {
            return res.json({ message: `${deletedCount} бренд(ів)
видалено успішно` });
        } else {
            return res.status(404).json({ message: "Жодного бренду не
знайдено" });
        }
    }
}
module.exports =  new BrandController()
```

Цей контролер виконує такі функції:

Create – Створення нового бренду.

getAll – Отримує список усіх брендів.

Delete – Видаляє бренди за їх ідентифікаторами, і повертає повідомлення про успішність видалення бренду.

DeviceController (див. Додаток В): цей контролер несе в собі такий функціонал:

create – створює новий пристрій, отримує данні з тіла запиту та зображення з файлів запиту, зберігає зображення у папці static, Якщо надано додаткову інформацію зберігає її в базі даних у таблиці DeviceInfo.

getAll – отримує пристрої з можливістю фільтрації.

getOne – отримує один пристрій за його ідентифікатором.

delete –видаляє пристрої за їхніми ідентифікаторами, видаляє відповідні зображення пристроїв з файлової системи.

OrderController:

```
const { Order, OrderItem } = require('../models/models');
const ApiError = require('../error/ApiError');

class OrderController {
    async create(req, res, next) {
        try {
            const { userId, items, totalPrice } = req.body;

            const order = await Order.create({ userId, totalPrice });

            const orderItems = items.map(item => ({
                orderId: order.id,
                deviceId: item.deviceId,
                quantity: item.quantity
```

```
            }));

            await OrderItem.bulkCreate(orderItems);

            return res.json(order);
        } catch (e) {
            next(ApiError.badRequest(e.message));
        }
    }

    async getAll(req, res) {
        try {
            const orders = await Order.findAll({ include: [{ model:
OrderItem, as: 'items' }] });
            return res.json(orders);
        } catch (e) {
            next(ApiError.badRequest(e.message));
        }
    }

    async getOne(req, res) {
        try {
            const { id } = req.params;
            const order = await Order.findOne({ where: { id },
include: [{ model: OrderItem, as: 'items' }] });

            if (!order) {
                return res.status(404).json({ message: "Замовлення не
знайдено" });
            }

            return res.json(order);
        } catch (e) {
            next(ApiError.badRequest(e.message));
```

```
        }
    }
}

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

Цей контролер несе в собі такі функції:

create – створює нове замовлення, повертає створене замовлення у відповідь.

getAll – отримує список усіх замовлень у системі.

getOne – отримує конкретне замовлення за його ідентифікатором включаючи відомості про товари цього замовлення.

RatingController:

```
const {Rating} = require('../models/models');
const authMiddleware = require('../middleware/authMiddleware');
class RatingController {
    async createRating(req, res) {
        try {
            const {rate, deviceId} = req.body;
            const userId = req.user.id; // Використання ID
зареєстрованого користувача з JWT

            // Перевіряємо, чи існує вже рейтинг від цього користувача
для цього пристрою
            const existingRating = await Rating.findOne({ where:
{userId, deviceId} });

            if (existingRating) {
                // Якщо рейтинг вже існує, видаляємо його
                await existingRating.destroy();
            }

            // Створюємо новий рейтинг
```

```
            const rating = await Rating.create({rate, deviceId,
userId});

            return res.json(rating);
        } catch (error) {
            return res.status(500).json({message: error.message});
        }
    }

    async getAverageRating(req, res) {
        try {
            const {deviceId} = req.params;
            const ratings = await Rating.findAll({where: {deviceId}});
            const averageRating = ratings.reduce((acc, cur) => acc +
cur.rate, 0) / ratings.length;
            return res.json({average: averageRating.toFixed(1)});
        } catch (error) {
            return res.status(500).json({message: error.message});
        }
    }
}

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

Цей контролер несе в собі такий функціонал:

createRating – створює рейтинг для конкретного пристрою.

getAverageRating – Отримує середній рейтинг для конкретного пристрою.

Type Controller:

```
const { Type } = require('../models/models');
const ApiError = require('../error/apiError');

class TypeController {
    async create(req, res) {
```

```
        const { name } = req.body;
        const type = await Type.create({ name });
        return res.json(type);
    }

    async getAll(req, res) {
        const types = await Type.findAll();
        return res.json(types);
    }

    async delete(req, res) {
        const { id } = req.params;
        const type = await Type.destroy({ where: { id } });
        if (type) {
            return res.json({ message: "Тип успішно видалено" });
        } else {
            return res.status(404).json({ message: "Тип не знайдено"
});
        }
    }
}

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

Цей контролер несе в собі такий функціонал:

create – створює новий тип пристрою.

getAll – отримує список усіх існуючих типів пристроїв у системі.

delete – видаляє тип пристрою за його ідентифікатором який переданий у параметрах запиту.

UserController (див. Додаток Г): Цей контролер несе в собі такий функціонал:

registration – обробляє реєстрацію нового користувача, перевіряє наявність обов'язкових полів, хешує пароль за допомогою bcrypt, створює корзину для нового користувача, генерує JWT токен для авторизації користувача.

login – обробляє процес входу користувача, порівнює введений пароль з хешованим паролем в базі даних, перевіряє статус ролі користувача, генерує JWT токе для авторизації користувача.

check – перевіряє статус авторизації користувача.

changeUserRole – змінює роль коистувача.

getUsers – отримує список усіх користувачів системи.

aiController:

```
const { Device } = require('../models/models');
const ApiError = require('../error/ApiError');
const axios = require('axios');

class AIController {
    async generateProduct(req, res, next) {
        try {
            const { name, price, brandId, typeId } = req.body;
            const description = await this.generateDescription(name);

            // Створення нового товару
            const device = await Device.create({ name, price, brandId,
typeId, description });

            return res.json(device);
        } catch (e) {
            next(ApiError.badRequest(e.message));
        }
    }

    async generateDescription(name) {
```

```
        try {
            const API_KEY = 'sk-proj-
hOJANB3jbdo0tupa4oq3T3BlbkFJZzZ6vLg1mq85TlHFgZNL';
            const response = await axios.post(
                'https://api.openai.com/v1/engines/davinci-
codex/completions',
                {
                    prompt: `Generate a detailed product description
for a product named "${name}"`,
                    max_tokens: 150,
                    temperature: 0.7,
                },
                {
                    headers: {
                        'Content-Type': 'application/json',
                        'Authorization': `Bearer ${API_KEY}`,
                    },
                }
            );
            return response.data.choices[0].text.trim();
        } catch (e) {
            throw new Error('Помилка при генерації опису за допомогою
ChatGPT');
        }
    }
}

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

Цей код несе в собі такий функціонал:

generateProduct – отримує параметри назва, ціна, id бренда і id бренду з тіла запиту після чого викликає метод generateDescription і створює новий продукт.

generateDescription – приймає параметр назви продукту, виконує запит до зовнішнього API OpenAI для генерації детального опису продукту на основі заданої назви, використовує API для авторизації, отримує відповідь від API після чого повертає текст з описом продукту.

Після успішного створення роутерів та контролерів все що залишається це створити головний файл index.js який буде запускати наш сервер:

```
require('dotenv').config()
const express = require('express')
const sequelize = require('./db')
const models = require('./models/models')
const cors = require('cors')
const fileUpload = require('express-fileupload')
const router = require('./routes/index')
const errorHendler =require('./middleware/ErrorHandlingMiddleware')
const path = require('path')
const ratingRouter = require('./routes/ratingRouter');
const basketRouter = require('./routes/basketRouter');
const aiRouter = require('./routes/aiRouter')

const PORT = process.env.PORT || 5000


const app = express()

app.use(cors())
app.use(express.json())
app.use(express.static(path.resolve(__dirname, 'static')))
app.use(fileUpload({}))
app.use('/api', router)
app.use('/api', ratingRouter);
app.use('/api', basketRouter);
app.use('/api/ai', aiRouter);
```

```
// Обробка помилок, останній Middleware !
app.use(errorHendler)
const start = async () => {
    try{
        await sequelize.authenticate()
        await sequelize.sync()
        app.listen(PORT,() => console.log(`Server started on port
${PORT}`))
    }catch (e) {
        console.log(e)
    }
}

start()
```

Після створення всіх цих файлів, роутерів, і контролерів, ми отримуємо повноцінний сервер backend для інтернет-магазину на основі Node.js.

Як можна побачити нижче, також була створена мінімальна візуальна складова сторінок: авторизації (див. рис. 3.1), реєстрації (див. рис. 3.2), кошика (див. рис. 3.4), вікна пристрою (див. рис. 3.5), панелі адміністратора (див. рис. 3.6) та головної сторінки (див. рис. 3.3). Це дозволило легко здійснювати запити без використання сторонніх програм, таких як Postman:

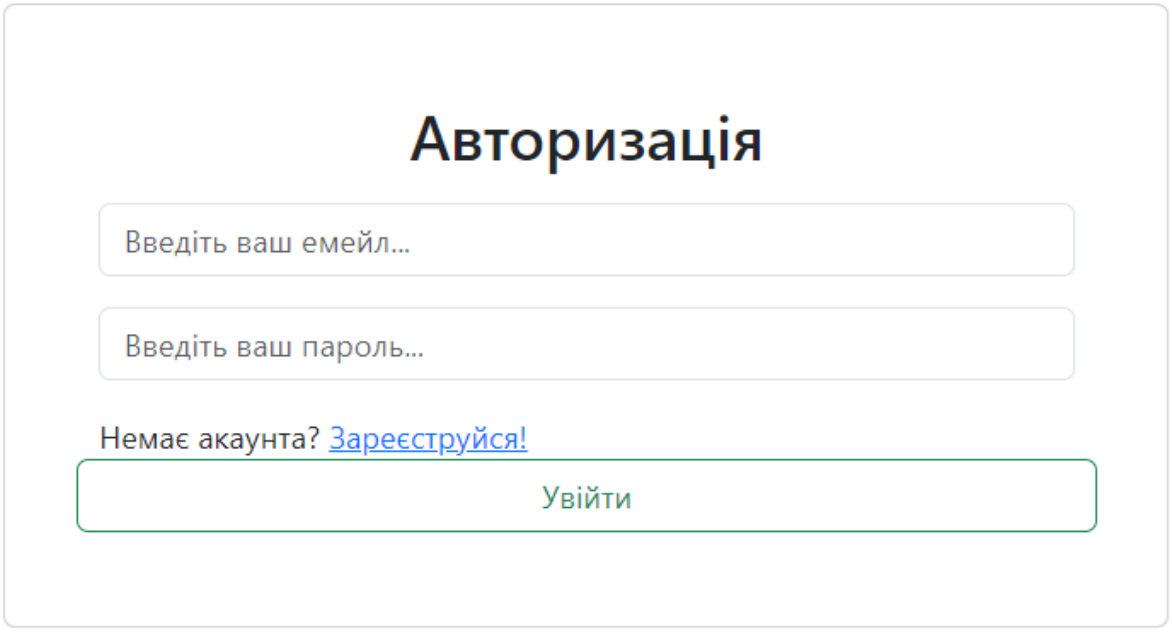

Рис. 3.1 Вікно авторизації

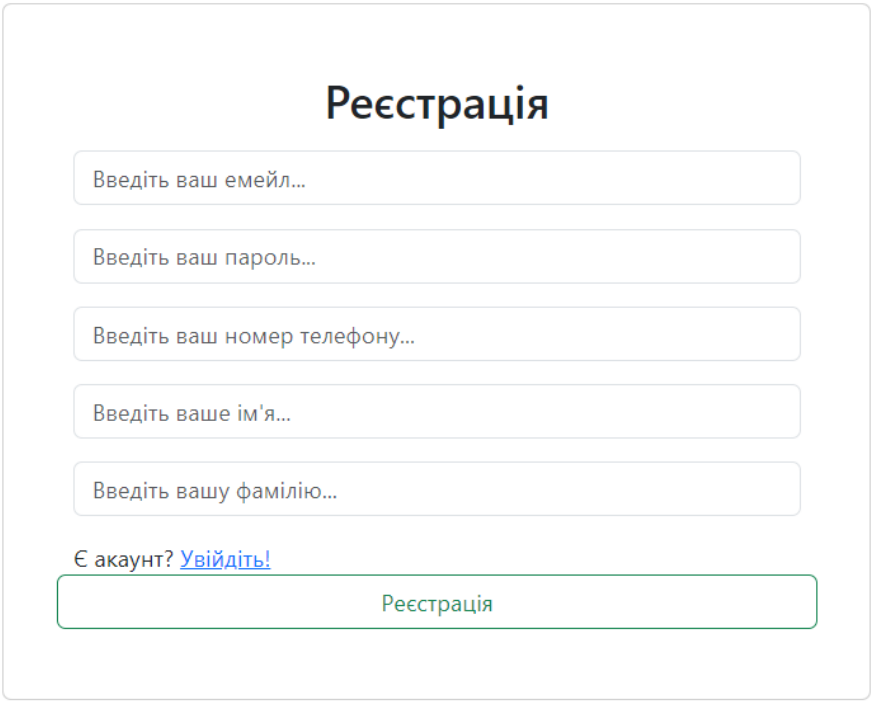

Рис. 3.2 Вікно реєстрації

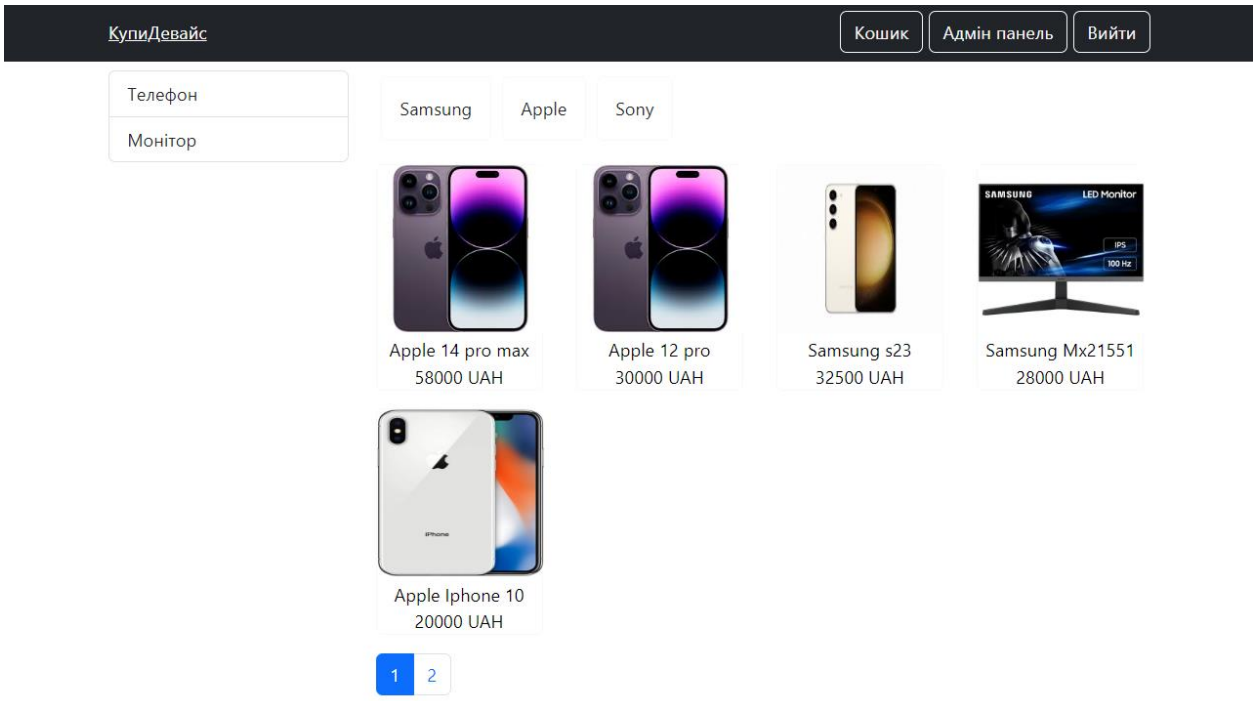

Рис. 3.3 Головна сторінка інтернет-магазину

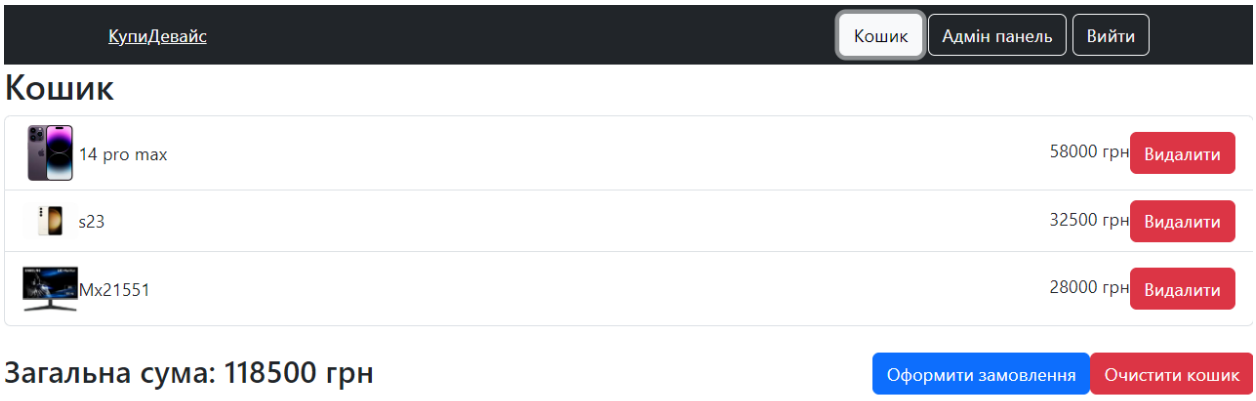

Рис. 3.4 Сторінка кошику

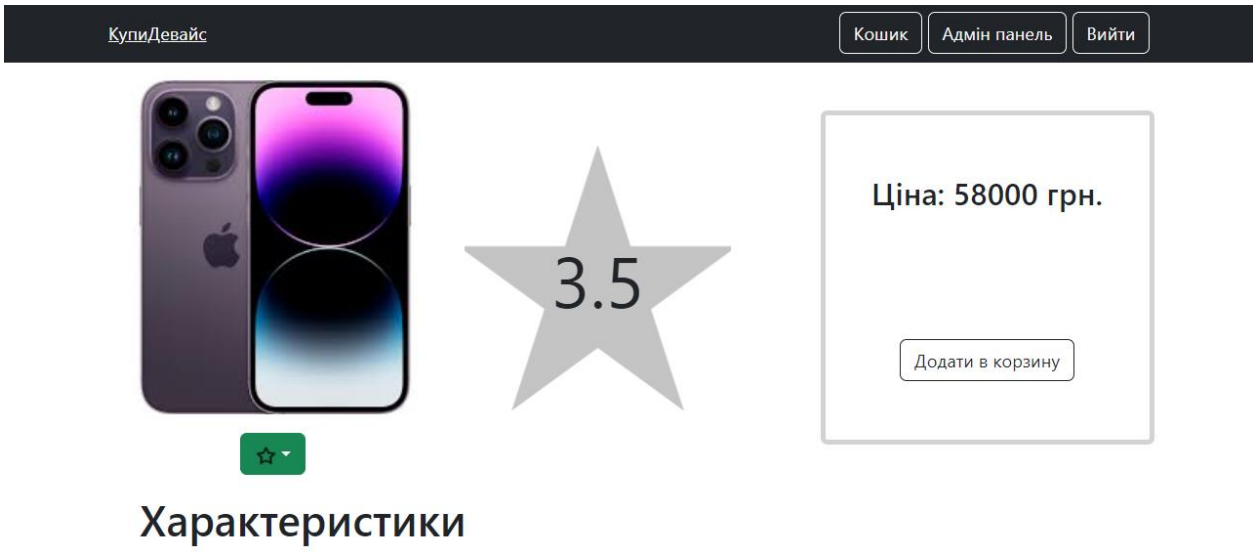

Рис. 3.5 Вікно пристрою

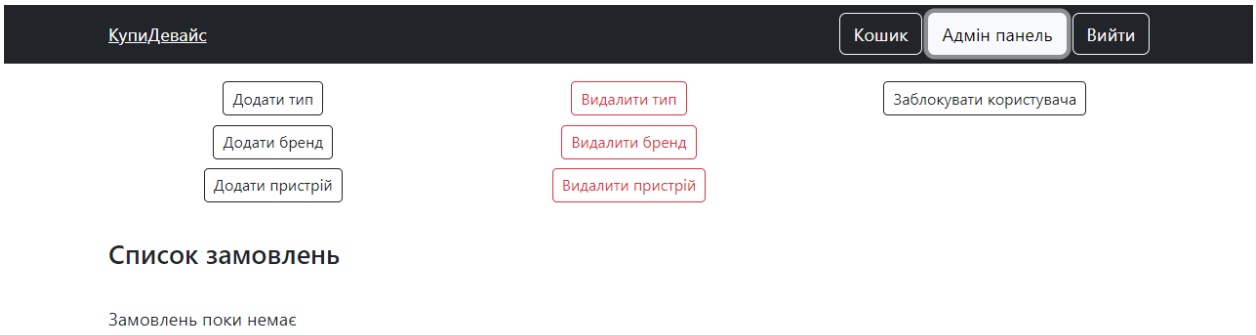

Рис. 3.6 Вікно панелі адміністратора

3.2 Забезпечення безпеки

Забезпечення безпеки в будь-якому додатку являється критичним аспектом. Тому щоб забезпечити свій додаток, використовувались такі рішення:

1)JWT для аутентифікації:

Використання JWT для безпечння автентифікації користувачів. Нижче наведений код з реалізованого додатку який перевіряє JWT токен:

```
const jwt = require('jsonwebtoken')

module.exports = function (req, res, next) {
```

```
    if (req.method === "OPTIONS") {
        next()
    }
    try {
        const token = req.headers.authorization.split(' ')[1] // Тип - 
Bearer Токен - asfasnfkajsfnjk
        if (!token) {
            return res.status(401).json({message: "Користувач не 
авторизований"})
        }
        const decoded = jwt.verify(token, process.env.SECRET_KEY)
        req.user = decoded
        next()
    } catch (e) {
        res.status(401).json({message: "Користувач не авторизований"})
    }
};
```

2) Middleware для перевірки ролей:

Для обмеження доступу до певних ресурсів було застосовано checkRoleMiddleware. Код наведений нижче перевіряє роль користувача на основі JWT токена це зроблено для захисту панелі адміністратора:

```
const jwt = require('jsonwebtoken');

module.exports = function(role) {
    return function (req, res, next) {
        if (req.method === "OPTIONS") {
            next();
        }
        try {
            const token = req.headers.authorization.split(' ')[1]; // 
Bearer asfasnfkajsfnjk
            if (!token) {
```

```
                return res.status(401).json({ message: "Користувач не
авторизований" });
            }
            const decoded = jwt.verify(token, process.env.SECRET_KEY);
            if (decoded.role !== role) {
                return res.status(403).json({ message: "Немає доступу"
});
            }
            req.user = decoded;
            next();
        } catch (e) {
            res.status(401).json({ message: "Користувач не
авторизований" });
        }
    };
};
```

3) Бібліотека bcrypt для хешування паролів:

Використання бібліотеки bcrypt для безпечного зберігання і порівняння хешів паролів користувачів. Це дозволяє запобігти можливості взлому акаунтів через перехоплення паролів в базі даних.

4) Middleware для перевірки помилок:

Використання ErrorHandlingMiddleware.js для обробки помилок сприяє покращенню безпеки і надійності додатку шляхом стандартизації обробки помилок і зменшення ризику некоректного поводження системи при виникненні помилок. Код для реалізації ErrorHandlingMiddleware.js:

```
const ApiError = require('../error/apiError')

module.exports = function (err, req, res, next){
    if(err instanceof ApiError) {
        return  res.status(err.status).json({message: err.message})
    }
```

```
    return res.status(500).json({message: 'Непередбачена Помилка!'})
}
```

5) Клас для обробки кастомних помилок:

Використання класу ApiError для створення кастомних помилок у додатку, дозволяє структурувати і ідентифікувати помилки з більшою чіткістю. Приклад коду для ApiError:

```
class ApiError extends Error {
    constructor(status, message) {
        super();
        this.status = status;
        this.message = message;
    }

    static badRequest(message) {
        return new ApiError(400, message);
    }

    static unauthorized(message) {
        return new ApiError(401, message);
    }

    static forbidden(message) {
        return new ApiError(403, message);
    }

    static notFound(message) {
        return new ApiError(404, message);
    }

    static unprocessableEntity(message) {
        return new ApiError(422, message);
    }
```

```
    static internal(message) {
        return new ApiError(500, message);
    }
}

module.exports = ApiError;
```

6) Захист від SQL ін’єкцій:

Використання Sequelize як ORM для PostgreSQL автоматично параметризує SQL запити до бази даних. Це не дає можливості виконання SQL ін’єкцій шляхом правильної обробки вхідних даних та їх інтеграції у запити до бази даних.

Ця архітектура забезпечення безпеки використовує JWT для автентифікації, middleware для перевірки ролей, бібліотеку bcrypt для хешування паролів, middleware для обробки помилок і кастомний клас для управління помилками. Вище згадана архітектура дозволяє забезпечити надійність, безпеку і стабільність інтернет-магазину.

3.3 Висновки до Розділу 3

В 3 розділі була розглянута реалізація серверної частини інтернет-магазину на платформі Node.js, а також забезпеченням його безпеки.

Структура додатку була побудована за принципом MVC (Model View Controller),що дозволило чітко відокремити бізнес-логіку, представлення та маршрутизацію. Кожен з компонентів (моделі, контролери, маршрути) був створений для обробки конкретної функціональності магазину, такої як управління товарами, замовленнями, користувачами тощо. Також була згадана мінімальна візуальна складова.

Загальний результат розділу це готовий до використання backend забезпечений стабільністю, безпекою, та високою швидкістю завдяки використанню сучасних технологій та методів захисту.

РОЗДІЛ 4. ПЕРЕВІРКА І ТЕСТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ ІНТЕРНЕТ МАГАЗИНУ

4.1 Види тестування

Тестування є невід'ємною частиною процесу розробки програмного забезпечення, що гарантує його надійність безпеку та ефективність. Існує кілька видів тестувань, кожен з яких виконує свою особливу роль у забезпечені якості програмного забезпеченням.

Юніт-тестування фокусується на перевірці окремих компонентів або модулів програм. Це дозволяє виявити та виправити помилки на різних етапах розробки додатку. Завдяки чому можна впевнитись що кожен компонент функціонує правильно.

Функціональне тестування орієнтоване на перевірку виконання програмою функціональних вимог. Це означає, що система тестується в контексті реальних сценаріїв використання, що дозволяє переконатися у її відповідності очікування користувача.

Безпекове тестування є дуже важливим для виявлення вразливостей у додатку. Воно дозволяє виявити та виправити слабкі місця, які можуть бути використанні для кіберзагроз, таким чином забезпечивши максимальний захист свого додатку.

Таким чином, різні види тестування забезпечують різносторонню перевірку програмного забезпечення. Вони дозволяють виявити та виправити помилки, забезпечити стабільність і продуктивність, а також забезпечити стабільність і продуктивність, а також гарантувати безпеку та високу якість коду. Завдяки комплексному підходу до тестування свого додатку можна впевнитись що все функціонує надійно та відповідає вимогам користувачів.

4.2 Тестування серверної частини

Під час розробки інтернет магазину для перевірки API запитів використовувався Postman. Це зручний інструмент, що дозволяє тестувати та

налагоджувати запити до сервера. За допомогою Postman було перевірено правильність роботи всіх основних маршрутів сервера.

Ось деякі приклади тестів в Postman, які використовувались під час розробки додатку:

Тестування реєстрації користувача:

Метод: POST

URL: http://localhost:5000/api/user/registration

Тіло запиту:

```
{
    "email": "TestUser",
    "password": "123456",
    "role": "USER"
}
```

Результат:

```
{
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTEsImVtYWlsIjoiVGVzdFVz
ZXIiLCJyb2xlIjoiVVNFUiIsImlhdCI6MTcxODMwOTUxMCwiZXhwIjoxNzE4Mzk1OTEwfQ
.9BPht6Tq1MGH-RRXswxmeKCczMa0peDjSgQEOx66qTY"
}
```

Тестування авторизації користувача:

Метод: POST

URL:

```
{
    "email": "TestUser",
```

```
    "password": "123456"
}
```


Тіло запиту:


```
{
    "email": "TestUser",
    "password": "123456"
}
```


Результат:


```
{
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTEsImVtYWlsIjoiVGVzdFVz
ZXIiLCJyb2xlIjoiVVNFUiIsImlhdCI6MTcxODMxMDM5OCwiZXhwIjoxNzE4Mzk2Nzk4fQ
.MRnnvU1gLaXbHxI3u8wlyMR-gfNRHa6rDLPT2X9Hsks"
}
```


Тестування отримання всіх типів:

Метод: GET

URL: http://localhost:5000/api/type

Результат:


```
[
    {
        "id": 2,
        "name": "Телефон",
        "createdAt": "2024-04-06T16:59:11.292Z",
        "updatedAt": "2024-04-06T16:59:11.292Z"
```

```
    },
    {
        "id": 3,
        "name": "Монітор",
        "createdAt": "2024-04-07T18:09:29.145Z",
        "updatedAt": "2024-04-07T18:09:29.145Z"
    }
]
```


Тестування отримання всіх девайсів:

Метод: Get

URL: http://localhost:5000/api/device

Результат:


```
{
    "count": 5,
    "rows": [
        {
            "id": 1,
            "name": "14 pro max",
            "price": 58000,
            "rating": 0,
            "img": "273dd23e-8436-43d5-ae1f-6d0b7a527cfe.jpeg",
            "createdAt": "2024-04-06T20:18:36.291Z",
            "updatedAt": "2024-04-06T20:18:36.291Z",
            "typeId": 2,
            "brandId": 2
```

```
        },
        {
            "id": 2,
            "name": "12 pro ",
            "price": 30000,
            "rating": 0,
            "img": "a95b511c-3baf-4767-8528-4c7d089a5bc4.jpeg",
            "createdAt": "2024-04-06T20:19:04.925Z",
            "updatedAt": "2024-04-06T20:19:04.925Z",
            "typeId": 2,
            "brandId": 2
        },
        {
            "id": 4,
            "name": "s23",
            "price": 32500,
            "rating": 0,
            "img": "cb62765b-68a0-43cb-8b25-bfb3092078e6.jpeg",
            "createdAt": "2024-04-06T20:20:59.363Z",
            "updatedAt": "2024-04-06T20:20:59.363Z",
            "typeId": 2,
            "brandId": 1
        },
    ]
}
```


Тестування отримання всіх типів:

Метод: GET

URL: http://localhost:5000/api/type

Результат:


```
[
    {
        "id": 2,
        "name": "Телефон",
        "createdAt": "2024-04-06T16:59:11.292Z",
        "updatedAt": "2024-04-06T16:59:11.292Z"
    },
    {
        "id": 3,
        "name": "Монітор",
        "createdAt": "2024-04-07T18:09:29.145Z",
        "updatedAt": "2024-04-07T18:09:29.145Z"
    }
]
```


Однак головним тестуванням було Юніт-тестування[7], яке дозволило перевірити коректність роботи коду загалом. Нижче наведено приклад Юніт-тесту та його результат:

```
const chai = require('chai');
const chaiHttp = require('chai-http');
const app = require('./index' /
chai.use(chaiHttp);
const { expect } = chai;
```

```
describe('Full Backend Tests', () => {
    let token; // Токен для авторизації

    it('should register a user', (done) => {
        chai.request(app)
            .post('/api/registration')
            .send({ email: 'newuser@example.com', password:
'password123', role: 'USER' })
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.have.property('token');
                token = res.body.token; // Зберегти токен для
подальших тестів
                done();
            });
    });

    it('should login a user', (done) => {
        chai.request(app)
            .post('/api/login')
            .send({ email: 'newuser@example.com', password:
'password123' })
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.have.property('token');
```

```
                token = res.body.token; // Зберегти токен для
подальших тестів
                done();
            });
    });

    it('should check user token', (done) => {
        chai.request(app)
            .get('/api/auth')
            .set('Authorization', `Bearer ${token}`)
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.have.property('token');
                done();
            });
    });

    it('should change user role', (done) => {
        chai.request(app)
            .post('/api/change-role')
            .send({ userId: 1, role: 'ADMIN' })
            .end((err, res) => {
                expect(res).to.have.status(200);

expect(res.body).to.have.property('message').that.includes('Роль
користувача змінена');
                done();
```

```
            });
    });

    it('should get all users', (done) => {
        chai.request(app)
            .get('/api/user')
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.be.an('array');
                done();
            });
    });

    it('should place an order', (done) => {
        chai.request(app)
            .post('/api/order')
            .set('Authorization', `Bearer ${token}`)
            .send({ userId: 1, items: [{ productId: 1, quantity: 2 }]
})
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.have.property('orderId');
                done();
            });
    });

    it('should retrieve order details', (done) => {
```

```
        chai.request(app)
            .get('/api/order/1')
            .set('Authorization', `Bearer ${token}`)
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.have.property('order');
                expect(res.body.order).to.have.property('items');
                done();
            });
    });

    after(async () => {
        console.log('Cleaning up after tests...');
        await User.destroy({ where: { email: 'newuser@example.com' }
});
        await Order.destroy({ where: { userId: 1 } });
    });
});
```

Результати тестування за допомогою Юніт-тесту можна побачити на Рис. 4.1:

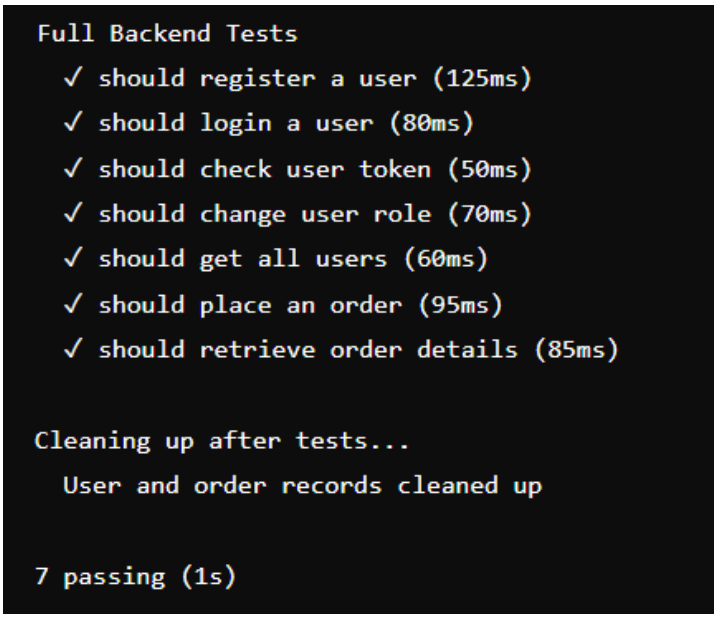

Рис. 4.1 Результат виконання юніт-тесту

Після виконання юніт тесту можна точно сказати що функціонал сайту працює дуже швидко і стабільно.

4.3 Висновки до розділу 4

У цьому розділі було розглянуто основні види тестів. Основним інструментом для перевірки API-запитів став Postman.

Також було продемонстровано запити та їх результати. Головним тестом став юніт тест після виконання якого було зроблено висновки що додаток працює на високій швидкості і з стійкою стабільністю.

ВИСНОВКИ

У даній роботі було проведено всебічне дослідження процесу розробки інтернет магазину, включаючи аналіз вимог, проектування, реалізацію та тестування. Для початку було визначено мету та завдання роботи, окреслено структуру дослідження та його актуальність.

Після чого було розглянуто основні вимоги до створення системи управління інтернет-магазином. Було проведено аналіз цільової аудиторії, визначено ключові функціональні можливості системи, такі як реєстрація і авторизація користувачів, управління товарами та замовленнями. Також була звернута увага на захист від кіберзагроз.

Далі було обрано технології які допоможуть розробити актуальну систему управління інтернет-магазином. Такими технологіями стали Node.Js, Express, Sequelize та інші вище зазначені технології. Ці технології допомогли розробити таку систему яка зможе забезпечити стабільну роботу при обробці великих обсягів даних.

Одним з найважливіших пунктів цієї роботи стало тестування реалізованої системи управління інтернет-магазином. Були описані різні види тестування а також були використанні деякі з них, що допомогло продемонструвати стабільну роботу додатку.

На основі проведеного дослідження можна зробити такі висновки: що розроблена серверна частина інтернет магазину відповідає вимогам і стандартам якості. Забезпечено стабільну роботу системи, високу продуктивність та безпеку даних. Виконання тестування підтвердило коректність реалізації основних функціональних можливостей. Завдяки комплексному підходу до проектування, реалізація та тестування було досягнута мета роботи – створення надійної та ефективної системи управління інтернет магазином.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. "JavaScript: The Definitive Guide", Автор: Девід Флэнаган, 2019, с.1264.
2. "Learning Express (3-е видання) ", Автор: Карлос Бастідас, 2023, с.432.
3. "MongoDB: The Definitive Guide", Автор: Kristina Chodorow, 2013, с.432.
4. "Node.js в дії (2-е видання) ", Автори: Метт Кантелон, Майк Хартер, Том Головайчук, Ніколь Райліх, 2021, с. 576.
5. "Оптимізація запитів PostgreSQL", Автор: Домбровська Р., 2019, с.320.
6. "Практичне проектування баз даних", Автор: Марк Файлер, 2019, с.720.
7. "Принципи юніт-тестування", Володимира Хорикова, 2022, с.264.
8. "Професійний довідник з MS SQL Server", Автор: Патт Сміт та Адам Вілсон, 2019, с. 1504.
9. Мануал по Bcrypt. URL: https://uk.php.brj.cz/hesuvannya-ryadkiv-i-paroliv.
10. Посібник по Sequelize. URL: https://foxminded.ua/sequelize-shcho-tse/.

ДОДАТКИ

Додаток А

```
const sequelize = require('../db');
const { DataTypes } = require('sequelize');

const User = sequelize.define('user', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    email: { type: DataTypes.STRING, unique: true },
    password: { type: DataTypes.STRING },
    role: { type: DataTypes.STRING, defaultValue: "USER" },
}, { freezeTableName: true, tableName: 'user' });

const Basket = sequelize.define('basket', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
}, { freezeTableName: true, tableName: 'basket' });

const BasketDevice = sequelize.define('basket_device', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    basketId: { type: DataTypes.INTEGER, allowNull: false },
    deviceId: { type: DataTypes.INTEGER, allowNull: false },
    quantity: { type: DataTypes.INTEGER, allowNull: false, defaultValue:
1 },
}, { freezeTableName: true, tableName: 'basket_device' });

const Device = sequelize.define('device', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    name: { type: DataTypes.STRING, unique: true, allowNull: false },
    price: { type: DataTypes.INTEGER, allowNull: false },
```

```
    rating: { type: DataTypes.INTEGER, defaultValue: 0 },
    img: { type: DataTypes.STRING, allowNull: false },
}, { freezeTableName: true, tableName: 'device' });

const Type = sequelize.define('type', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    name: { type: DataTypes.STRING, unique: true, allowNull: false },
}, { freezeTableName: true, tableName: 'type' });

const Brand = sequelize.define('brand', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    name: { type: DataTypes.STRING, unique: true, allowNull: false },
}, { freezeTableName: true, tableName: 'brand' });

const Rating = sequelize.define('rating', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    userId: { type: DataTypes.INTEGER, allowNull: false },
    deviceId: { type: DataTypes.INTEGER, allowNull: false },
    rate: { type: DataTypes.INTEGER, allowNull: false },
}, { freezeTableName: true, tableName: 'rating' });

const DeviceInfo = sequelize.define('device_info', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    title: { type: DataTypes.STRING, allowNull: false },
    description: { type: DataTypes.STRING, allowNull: false },
}, { freezeTableName: true, tableName: 'device_info' });

const Order = sequelize.define('order', {
```

```
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    userId: { type: DataTypes.INTEGER, allowNull: false },
    totalPrice: { type: DataTypes.FLOAT, allowNull: false }
}, { freezeTableName: true, tableName: 'order' });

const OrderItem = sequelize.define('order_item', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
    orderId: { type: DataTypes.INTEGER, allowNull: false },
    deviceId: { type: DataTypes.INTEGER, allowNull: false },
    quantity: { type: DataTypes.INTEGER, allowNull: false }
}, { freezeTableName: true, tableName: 'order_item' });

// Зв'язки між моделями
User.hasOne(Basket);
Basket.belongsTo(User);

User.hasMany(Rating);
Rating.belongsTo(User);

Basket.hasMany(BasketDevice);
BasketDevice.belongsTo(Basket);

Type.hasMany(Device);
Device.belongsTo(Type);

Brand.hasMany(Device);
Device.belongsTo(Brand);

Device.hasMany(Rating);
Rating.belongsTo(Device);
```

```
Device.hasMany(BasketDevice);
BasketDevice.belongsTo(Device);

Device.hasMany(DeviceInfo, { as: 'info' });
DeviceInfo.belongsTo(Device);

Order.hasMany(OrderItem, { as: 'items' });
OrderItem.belongsTo(Order);

const TypeBrand = sequelize.define('type_brand', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
},
}, { freezeTableName: true, tableName: 'type_brand' });

Type.belongsToMany(Brand, { through: TypeBrand });
Brand.belongsToMany(Type, { through: TypeBrand });

module.exports = {
    User,
    Basket,
    BasketDevice,
    Device,
    Type,
    Brand,
    Rating,
    DeviceInfo,
    Order,
    OrderItem,
    TypeBrand,
};
```

Додаток Б

```
const { Basket, BasketDevice, Device } = require('../models/models');

class BasketController {
    async addToBasket(req, res) {
        const { deviceId, quantity } = req.body;
        const userId = req.user.id; // Авторизований користувач
        try {
            let basket = await Basket.findOne({ where: { userId } });
            if (!basket) {
                basket = await Basket.create({ userId });
            }
            // Знайти чи товар вже є в кошику
            let basketDevice = await BasketDevice.findOne({ where: {
basketId: basket.id, deviceId } });
            if (basketDevice) {
                // Якщо товар вже є в кошику, оновити кількість
                basketDevice.quantity += quantity;
                await basketDevice.save();
            } else {
                // Інакше додати новий товар у кошик
                basketDevice = await BasketDevice.create({
                    basketId: basket.id,
                    deviceId,
                    quantity
                });
            }
            return res.json({ message: "Товар додано до корзини",
basketDevice });
        } catch (error) {
            return res.status(500).json({ message: error.message });
        }
    }

    async updateBasketItem(req, res) {
        const { deviceId, quantity } = req.body;
        const userId = req.user.id;
        try {
            const basket = await Basket.findOne({ where: { userId }
});
            const basketDevice = await BasketDevice.findOne({ where: {
basketId: basket.id, deviceId } });
            if (basketDevice) {
                basketDevice.quantity = quantity;
                await basketDevice.save();
                return res.json({ message: "Товари в кошику оновлені"
```

```
});
            }
            return res.status(404).json({ message: "Продукт не
знайдений в корзині" });
        } catch (error) {
            return res.status(500).json({ message: error.message });
        }
    }

    async removeFromBasket(req, res) {
        const { deviceId } = req.params;
        const userId = req.user.id;
        try {
            const basket = await Basket.findOne({ where: { userId }
});
            const result = await BasketDevice.destroy({ where: {
basketId: basket.id, deviceId } });
            if (result > 0) {
                return res.json({ message: "Товар видалено з кошика"
});
            }
            return res.status(404).json({ message: "Товар не знайдено
в кошику" });
        } catch (error) {
            return res.status(500).json({ message: error.message });
        }
    }

    async getBasketItems(req, res) {
        const userId = req.user.id;
        try {
            const basket = await Basket.findOne({ where: { userId }
});
            const items = await BasketDevice.findAll({
                where: { basketId: basket.id },
                include: [Device]
            });
            return res.json(items);
        } catch (error) {
            return res.status(500).json({ message: error.message });
        }
    }

    async clearBasket(req, res) {
        const userId = req.user.id;
        try {
            const basket = await Basket.findOne({ where: { userId }
```

```
});
            if (!basket) {
                return res.status(404).json({ message: "Кошик не
знайдено" });
            }
            await BasketDevice.destroy({ where: { basketId: basket.id
} });
            return res.json({ message: "Кошик очищено" });
        } catch (error) {
            return res.status(500).json({ message: error.message });
        }
    }
    async getBasketTotal(req, res) {
        const userId = req.user.id;

        try {
            const basket = await Basket.findOne({ where: { userId }
});
            if (!basket) {
                return res.json({ total: 0 });
            }

            const items = await BasketDevice.findAll({
                where: { basketId: basket.id },
                include: [Device]
            });

            const totalPriceUAH = items.reduce((sum, item) => sum +
item.device.price, 0);

            return res.json({ total: totalPriceUAH });
        } catch (error) {
            return res.status(500).json({ message: error.message });
        }
    }
}

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

Додаток В

```
const uuid = require('uuid');
const path = require('path');
const fs = require('fs');
const { Device, DeviceInfo } = require('../models/models');
const ApiError = require('../error/ApiError');

class DeviceController {
    async create(req, res, next) {
        try {
            let { name, price, brandId, typeId, info } = req.body;
            const { img } = req.files;
            let fileName = uuid.v4() + ".jpg";
            img.mv(path.resolve(__dirname, '..', 'static', fileName));

            const device = await Device.create({ name, price, brandId,
typeId, img: fileName });

            if (info) {
                info = JSON.parse(info);
                info.forEach(i =>
                    DeviceInfo.create({
                        title: i.title,
                        description: i.description,
                        deviceId: device.id
                    })
                )
            }

            return res.json(device);
        } catch (e) {
            next(ApiError.badRequest(e.message));
        }
    }

    async getAll(req, res) {
        let { brandId, typeId, limit, page } = req.query;
        page = page || 1;
        limit = limit || 9;
        let offset = page * limit - limit;
        let devices;
        if (!brandId && !typeId) {
            devices = await Device.findAndCountAll({ limit, offset });
        }
        if (brandId && !typeId) {
            devices = await Device.findAndCountAll({ where: { brandId
```

```
}, limit, offset });
        }
        if (!brandId && typeId) {
            devices = await Device.findAndCountAll({ where: { typeId
}, limit, offset });
        }
        if (brandId && typeId) {
            devices = await Device.findAndCountAll({ where: { typeId,
brandId }, limit, offset });
        }
        return res.json(devices);
    }

    async getOne(req, res) {
        const { id } = req.params
        const device = await Device.findOne(
            {
                where: { id },
                include: [{ model: DeviceInfo, as: 'info' }]
            },
        )

        return res.json(device);
    }

    async delete(req, res, next) {
        try {
            const { ids } = req.body;  // Очікуємо масив
ідентифікаторів пристроїв
            const devices = await Device.findAll({ where: { id: ids }
});

            devices.forEach(device => {
                const imgPath = path.resolve(__dirname, '..',
'static', device.img);
                fs.unlinkSync(imgPath);  // Видаляємо зображення
            });

            const deletedCount = await Device.destroy({ where: { id:
ids } });
            if (deletedCount) {
                return res.json({ message: `${deletedCount} пристроїв
видалено успішно` });
            } else {
                return res.status(404).json({ message: "Жодного
пристрою не знайдено" });
            }
```

```
        } catch (error) {
            next(ApiError.badRequest(error.message));
        }
    }
}

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

Додаток Г

```
const ApiError = require('../error/apiError');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { User, Basket } = require('../models/models');
const generateJwt = (id, email, role) => {
    return jwt.sign(
        { id, email, role },
        process.env.SECRET_KEY,
        { expiresIn: '24h' }
    );
};

class UserController {
    async registration(req, res, next) {
        const { email, password, role } = req.body;
        if (!email || !password) {
            return next(ApiError.badRequest('Некоректний email або
password'));
        }
        const candidate = await User.findOne({ where: { email } });
        if (candidate) {
            return next(ApiError.badRequest('Користувач з таким email
вже існує'));
        }
        const hashPassword = await bcrypt.hash(password, 5);
        const user = await User.create({ email, role, password:
hashPassword });
        const basket = await Basket.create({ userId: user.id });
        const token = generateJwt(user.id, user.email, user.role);
        return res.json({ token });
    }

    async login(req, res, next) {
        const { email, password } = req.body;
        const user = await User.findOne({ where: { email } });
        if (!user) {
            return next(ApiError.internal('Користувач з таким іменем
не знайдений'));
        }
        let comparePassword = bcrypt.compareSync(password,
user.password);
        if (!comparePassword) {
            return next(ApiError.internal('Вказано невірний пароль'));
        }
```

```
        // Перевірка ролі користувача
        if (user.role === 'Ban') {
            return next(ApiError.forbidden('Користувач
заблокований'));
        }

        const token = generateJwt(user.id, user.email, user.role);
        return res.json({ token });
    }

    async check(req, res, next) {
        const token = generateJwt(req.user.id, req.user.email,
req.user.role);
        res.json({ token });
    }

    async changeUserRole(req, res, next) {
        const { userId, role } = req.body;

        try {
            const user = await User.findOne({ where: { id: userId }
});
            if (!user) {
                return next(ApiError.notFound('Користувач не
знайдений'));
            }

            user.role = role;
            await user.save();

            return res.json({ message: 'Роль користувача змінена' });
        } catch (error) {
            next(ApiError.internal('Внутрішня помилка сервера'));
        }
    }

    async getUsers(req, res, next) {
        try {
            const users = await User.findAll();
            return res.json(users);
        } catch (error) {
            next(ApiError.internal('Внутрішня помилка сервера'));
        }
    }
}

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