БАКАЛАВРСЬКА РОБОТА

БР. ІП - 13.00.00.000 ПЗ

Група ІП-22-4

Сорока Олександр

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Сорока Олександр Іванович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Проєктування та оптимізація бази даних для високонавантаженого веб-застосунку інтернет-магазину
(назва роботи)

Інженерія програмного забезпечення
(назва освітньої програми)

121– Інженерія програмного забезпечення
(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Сорока О. І._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Дмитрик Тарас Богданович, асистент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В._____
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу
Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура

“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Сорока Олександр Іванович
(прізвище, ім’я, по-батькові)

1. Тема проєкту (роботи) "Проєктування та оптимізація бази даних для високонавантаженого веб-застосунку інтернет-магазину"

керівник проєкту (роботи) Дмитрик Тарас Богданович, асист.

затверджені наказом вищого навчального закладу від **“** _21_ ” _квітня 2026 р. № 179/7

2. Строк подання студентом проєкту (роботи) 10 червня 2026р.

3. Вихідні дані до проєкту (роботи) Розроблена база даних для високонавантаженого веб-застосунку інтернет-магазину та веб-застосунок інтернет-магазину

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)
1. Огляд існуючих рішень та теоретичних основ.
2. Аналіз вимог та постановка задачі.
3. Проєктування системи.
4. Реалізація проєкту.
5. Тестування та впровадження.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
1. Схема архітектури веб-застосунку (рис. 3.1, ст. 27),
2. Багатошарова архітектура веб-застосунку (рис. 3.2, ст. 28),
3. Схема компонентів серверної частини (рис. 3.3, ст. 27),
4. ER-діаграма бази даних (рис. 3.4, ст. 32),
5. Схема хмарного розгортання веб-застосунку (рис. 5.15, ст. 63),

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 10 січня 2026 р.

Керівник _____Дмитрик Т.Б._____
(підпис)

Завдання прийняв до виконання _____Сорока О.І._____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту	Примітка
1	Вибір та обґрунтування теми, аналіз предметної області	15.01.2026	виконано
2	Огляд існуючих концепцій проєктування БД, вибір СУБД та інструментів	28.01.2026	виконано
3	Проєктування та оптимізація схеми бази даних	21.02.2026	виконано
4	Реалізація БД, розробка веб-застосунку та тестування	10.04.2026	виконано
5	Оформлення бакалаврської роботи та презентаційних матеріалів	10.06.2026	виконано

Студент ____Сорока О. І.____
(підпис)

Керівник роботи ___Дмитрик Т.Б.___
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 75 сторінки, 23 рисунки, список використаних джерел із 35 найменувань.

Мета дипломної роботи є проєктування та оптимізація бази даних для інтернет-магазину з розробкою повнофункціонального веб-додатку.

Об’єкт дослідження: база даних інтернет-магазину.

Предмет дослідження: методи проєктування, нормалізації, індексації та оптимізації реляційних баз даних, а також ORM-засоби.

У першому розділі проаналізовано проблематику високонавантажених веб-застосунків та порівняно СУБД, ORM-системи й архітектурні підходи.

У другому розділі зібрані вимоги, сформульовано функціональні і нефункціональні вимоги та проаналізована постановка задачі.

У третьому розділі спроєктовано архітектуру, побудовано ER-модель і обґрунтовано вибір технологій.

У четвертому розділі описано: реалізацію фізичного проєктування БД, серверну частину на Express/Prisma, клієнтський SPA на React, а також оптимізацію продуктивності.

У п’ятому розділі проведено тестування, наведено результати й описано розгортання в хмарному середовищі (Render, TiDB Cloud.

Висновки роботи містять основні результати дослідження та їх аналіз, а також рекомендації щодо подальшого вдосконалення системи.

Отримані результати включають оптимізовану схему БД, що гарантує цілісність і швидкодію; веб-додаток із каталогом, кошиком, замовленнями, відгуками та адміністративною панеллю.

КЛЮЧОВІ СЛОВА: БАЗА ДАНИХ, ПРОЄКТУВАННЯ, ОПТИМІЗАЦІЯ, ORM, PRISMA, TYPESCRIPT, REACT, EXPRESS, MYSQL, TIDB.

ANNOTATION

The bachelor's thesis contains 75 pages, 23 figures, and a list of references with 35 sources.

The purpose of the thesis is to design and optimize a database for an online store with the development of a fully functional web application.

Object of research: the database of an online store.

Subject of research: methods of design, normalization, indexing, and optimization of relational databases, as well as ORM tools.

The first chapter analyzes the problems of high-load web applications and compares DBMS, ORM systems, and architectural approaches.

The second chapter collects requirements, formulates functional and non-functional requirements, and analyzes the task statement.

The third chapter designs the architecture, builds the ER model, and justifies the choice of technologies.

The fourth chapter describes: the implementation of the physical database design, the server side with Express/Prisma, the client SPA with React, and performance optimization.

The fifth chapter covers testing, presents results, and describes deployment in a cloud environment (Render, TiDB Cloud).

The conclusions contain the main results of the study, their analysis, and recommendations for further improvement of the system.

The results obtained include an optimized database schema that guarantees integrity and performance; a web application with a catalog, cart, orders, reviews, and an admin panel.

KEYWORDS: DATABASE, DESIGN, OPTIMIZATION, ORM, PRISMA, TYPESCRIPT, REACT, EXPRESS, MYSQL, TIDB.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних

СУБД – система управління базами даних

ORM – об’єктно-реляційне відображення (Object-Relational Mapping)

ER-модель – модель «сутність-зв’язок» (Entity-Relationship Model)

SQL – мова структурованих запитів (Structured Query Language)

REST – передача репрезентативного стану (Representational State Transfer)

API – прикладний програмний інтерфейс (Application Programming Interface)

JWT – JSON Web Token

SPA – односторінковий додаток (Single Page Application)

CSS – каскадні таблиці стилів (Cascading Style Sheets)

TS – TypeScript

JSON – JavaScript Object Notation

HTTP – протокол передачі гіпертексту (HyperText Transfer Protocol)

HTTPS – захищений протокол передачі гіпертексту (HTTP Secure)

SSL – рівень захищених сокетів (Secure Sockets Layer)

TLS – протокол захисту транспортного рівня (Transport Layer Security)

CLI – інтерфейс командного рядка (Command Line Interface)

NPM – менеджер пакетів Node (Node Package Manager)

ВСТУП

Сучасний інтернет-магазин неможливий без надійної та швидкої бази даних. Саме тому проєктування й оптимізація БД є актуальним завданням для будь-якого комерційного веб-застосунку.

Актуальність теми зумовлена зростанням електронної комерції, що потребує швидких та масштабованих баз даних, які неможливо створити без сучасних методів проєктування й оптимізації.

Мета роботи спроєктувати та оптимізувати реляційну базу даних для магазину й створити на її основі повнофункціональний веб-додаток.

Завданнями дослідження є проєктування та оптимізація БД, оцінка методів пагінації, реалізація веб-додатку та його хмарне розгортання.

Об'єкт дослідження база даних інтернет-магазину. Методи проєктування, нормалізації, індексації та оптимізації, а також засоби ORM для роботи з базою.

Методи дослідження включають структурне моделювання, нормалізацію, порівняльний аналіз продуктивності запитів, практичну реалізацію з використанням TypeScript, Express, React та Prisma Client.

Наукова новизна полягає в поєднанні класичних принципів проєктування з можливостями хмарної СУБД TiDB, що дало змогу отримати масштабоване та ефективне рішення.

Практичне значення готовий веб-додаток, який може бути впроваджений як основа комерційного інтернет-магазину; база даних забезпечує цілісність, швидкодію та зручне адміністрування. .

Бакалаврська робота містить 75 сторінки, 23 рисунки, список використаних джерел із 35 найменувань, 2 додатки.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ТЕОРЕТИЧНИХ ОСНОВ

1.1. Аналіз сучасного стану проблематики

Сфера електронної комерції перебуває у стані безперервного розвитку, що зумовлює постійне зростання вимог до швидкодії, масштабованості та надійності веб-застосунків. Фундаментом будь-якого сучасного інтернет-магазину є продуктивна база даних (БД), яка здатна ефективно обробляти великі масиви інформації та забезпечувати стабільну роботу в умовах стрімкого збільшення кількості користувачів.

Специфіка предметної області, зокрема реалізація інтернет-магазинів побутової техніки, створює додаткові виклики для проєктувальників архітектури та баз даних. До ключових проблемних аспектів, з якими стикаються сучасні високонавантажені системи, належать:

- Складність структурування даних: Товарні позиції характеризуються десятками різнорідних атрибутів (технічні характеристики, ціноутворення, габарити тощо), що ускладнює побудову гнучкої моделі даних без втрати продуктивності.
- Високе навантаження на операції пошуку: Користувачі масово застосовують комплексні фільтри за ціновими діапазонами, брендами та специфікаціями, що вимагає миттєвої відповіді від системи управління базами даних (СУБД) та грамотної стратегії індексування.
- Критична потреба у транзакційній цілісності: Сценарії оформлення замовлень, резервування товарів та оновлення складських залишків повинні виконуватися як атомарні операції, щоб унеможливити фінансові чи логістичні розбіжності.

Традиційні реляційні СУБД (наприклад, MySQL або MariaDB) чудово задовольняють вимоги щодо підтримки ACID-транзакцій та узгодженості інформації. Однак, при значному зростанні обсягів товарних каталогів та трафіку,

класичні монолітні реляційні СУБД стикаються з проблемою горизонтального масштабування. Для розподілу навантаження доводиться впроваджувати складні механізми шардування даних чи реплікації, що критично ускладнює адміністрування інфраструктури. Альтернативний підхід у вигляді мікросервісів вирішує проблему масштабованості, але створює нові труднощі, пов'язані з підтримкою розподілених транзакцій та збільшенням накладних витрат на розробку.

Ще однією вагомою проблемою є неоптимальна програмна взаємодія застосунків із базами даних. Застосування класичної посторінкової пагінації (з використанням параметрів LIMIT та OFFSET) на великих таблицях створює надлишкове навантаження на дискову підсистему та процесор СУБД. Окрім цього, розробники часто стикаються з недоліками існуючих інструментів об’єктно-реляційного відображення (ORM-систем), які можуть не гарантувати належної типобезпеки або генерувати неефективні SQL-запити.

Отже, сучасний етап проєктування високонавантажених веб-застосунків вимагає пошуку нових компромісних рішень. Актуальним напрямком подолання зазначених проблем є перехід до використання розподілених хмарних СУБД (наприклад, TiDB), які поєднують нативне горизонтальне масштабування з підтримкою звичного реляційного синтаксису. Паралельно необхідне впровадження сучасних підходів до написання серверної логіки: використання суворо типізованих інструментів екосистеми TypeScript (таких як Prisma ORM) та заміна класичної пагінації на оптимізовані алгоритми (cursor-based pagination). Комплексне застосування цих технологій дозволяє створити надійну основу для розвитку масштабованих систем електронної комерції.

1.2. Основні поняття та термінологія предметної області

Для систематизації дослідження, присвяченого проєктуванню та оптимізації баз даних для високонавантажених інформаційних систем у сфері електронної комерції, необхідно чітко визначити базовий термінологічний апарат.

Це дозволить сформувати єдине науково-технічне підґрунтя для подальшого аналізу архітектурних та інженерних рішень.

Високонавантажений веб-застосунок (High-Load Web Application) — це програмний комплекс, що функціонує в мережі Інтернет і характеризується здатністю одночасно обробляти значні обсяги вхідних запитів (метрика RPS - Requests Per Second), оперувати масивами даних великого розміру (Big Data) та підтримувати стабільну працездатність при експоненційному зростанні кількості активних користувачів. Основними критеріями ефективності таких систем є мінімальний час відгуку (Latency), висока пропускна здатність (Throughput) та відмовостійкість (Availability).

Масштабованість (Scalability) — властивість системи адаптуватися до зростаючих обсягів навантаження без втрати продуктивності. Розрізняють два типи масштабованості:

Вертикальне масштабування (Vertical Scaling / Scaling Up) — нарощування обчислювальних потужностей одного сервера (збільшення об'єму оперативної пам'яті, кількості ядер процесора, швидкості дискової підсистеми).

Горизонтальне масштабування (Horizontal Scaling / Scaling Out) — розподіл навантаження шляхом додавання нових вузлів (серверів) до загальної інфраструктури системи.

Реляційна система управління базами даних (РСУБД) — спеціалізоване програмне забезпечення для організації, збереження та маніпулювання даними, що представлені у вигляді взаємопов'язаних таблиць (відносин). Класичні РСУБД спираються на суворе дотримання вимог ACID (Atomicity — атомарність, Consistency — узгодженість, Isolation — ізольованість, Durability — довговічність), що є критично важливим для фінансових транзакцій та обліку залишків товарів в інтернет-магазинах [1].

Розподілена NewSQL СУБД — сучасний клас систем керування даними, який поєднує в собі нативну здатність NoSQL-систем до горизонтального масштабування та розподіленої обробки запитів із повним збереженням переваг реляційної моделі (підтримка SQL-синтаксису та транзакційної цілісності ACID).

Прикладом такої архітектури є технологія TiDB, яка відокремлює шар обчислень від шару безпосереднього збереження даних, забезпечуючи автоматичне шардування (секціонування) без втручання в код застосунку[2].

Об'єктно-реляційне відображення (ORM — Object-Relational Mapping) — технологія програмування, що зв'язує бази даних із концепціями об'єктно-орієнтованих мов програмування, створюючи віртуальну об'єктну базу даних. У контексті сучасних веб-технологій (зокрема екосистеми Node.js / TypeScript) ORM-системи нового покоління (наприклад, Prisma) виконують роль декларативного інструменту, що забезпечує повну типобезпеку (Type Safety) на етапі компіляції та автоматизує процес генерації складних SQL-запитів [3].

Курсорна пагінація (Cursor-based Pagination) — оптимізований алгоритм порційної вибірки даних із великих таблиць. На відміну від класичної пагінації зі зміщенням (Offset-based), яка змушує СУБД сканувати всі попередні рядки (що призводить до падіння швидкодії до O(N)), курсорна пагінація використовує унікальний ідентифікатор (курсор) останнього отриманого запису та фільтрацію через індекси, забезпечуючи стабільну швидкість виконання запиту, яка не залежить від номера сторінки (на відміну від OFFSET-пагінації, продуктивність якої деградує зі зростанням зсуву) [4, 5].

Індексування даних (Indexing) — створення додаткових службових структур даних (найчастіше на основі B-дерев або інвертованих списків), які дозволяють СУБД здійснювати високошвидкісний пошук записів за певними атрибутами, мінімізуючи потребу в повному скануванні таблиць (Full Table Scan) [6].

REST API (Representational State Transfer Application Programming Interface) — архітектурний стиль взаємодії між компонентами веб-застосунку (клієнтом та сервером), заснований на протоколі HTTP. Він передбачає використання стандартних методів (GET, POST, PUT, DELETE) для маніпуляції ресурсами та передачу даних у безстановому форматі (Stateless), що спрощує кешування та горизонтальне масштабування серверної частини [7].

Визначені поняття формують теоретичний базис для переходу до практичної реалізації архітектурних рішень високонавантаженого інтернет-магазину побутової техніки.

1.3. Аналіз сучасних СУБД, ORM-систем та архітектурних підходів

Ефективне функціонування високонавантажених платформ електронної комерції безпосередньо залежить від коректного вибору та комбінації технологічних рішень на трьох рівнях: збереження даних (СУБД), проміжного програмного забезпечення (ORM) та загальної організації компонентів системи (архітектурний стиль).

Аналіз та порівняння систем управління базами даних

У сучасній веб-розробці вибір СУБД для інтернет-магазинів традиційно коливається між класичними реляційними системами та новітніми розподіленими рішеннями. Для систем із високими вимогами до цілісності даних (транзакційність, облік фінансів, контроль складських залишків) NoSQL-рішення часто є неефективними через відсутність повноцінної підтримки ACID. Тому основними об'єктами аналізу залишаються реляційні та NewSQL СУБД [8].

Популярні класичні РСУБД, такі як PostgreSQL та MySQL/MariaDB, мають потужний інструментарій для індексування та оптимізації запитів. Проте за умов високого навантаження вертикальне масштабування цих систем швидко досягає фізичних меж серверного обладнання [10, 11, 12]. Альтернативою є архітектура NewSQL, яскравим представником якої є розподілена СУБД TiDB [9]. Вона поєднує транзакційні гарантії реляційних баз з горизонтальним масштабуванням, автоматично розподіляючи дані між вузлами без необхідності ручного шардування з боку розробника, порівняння СУБД зображено в таблиці 1.1.

Таблиця 1.1

Порівняльний аналіз сучасних СУБД для високонавантажених застосунків

Критерій порівняння	**MySQL / MariaDB**	**PostgreSQL**	**TiDB (NewSQL)**
Модель даних	Реляційна	Постреляційна / Реляційна	Розподілена реляційна
Масштабованість	Переважно вертикальна; горизонтальна через реплікацію / шардування	Переважно вертикальна; складні кластери (Patroni)	Нативна горизонтальна (відокремлений шар обчислень і збереження)
Підтримка ACID	Повна (для двигуна InnoDB)	Повна, високий рівень ізоляції	Повна (розподілені транзакції)
Швидкість читання великих обсягів	Середня, падає при великій вкладеності JOIN	Висока, завдяки оптимізатору запитів	Дуже висока (завдяки розподіленій архітектурі обробки)
Складність адміністрування	Низька / Середня	Середня	Висока (у локальних мережах), низька (у хмарі TiDB Cloud)

Аналіз інструментів об'єктно-реляційного відображення (ORM)

Взаємодія серверного коду із СУБД зазвичай реалізується за допомогою ORM-систем, які спрощують написання запитів та автоматизують міграції схем даних. В екосистемі Node.js / TypeScript існує кілька підходів до реалізації цього шару: класичні важкі ORM (TypeORM, Sequelize), декларативні системи нового покоління (Prisma) та легковагові конструктори запитів (Drizzle ORM), їх порівняння зображено в таблиці 1.2 [13, 14, 15, 16].

Таблиця 1.2

Порівняльна характеристика інструментів ORM для екосистеми Node.js

Назва ORM	**Підхід до моделювання**	**Рівень підтримки TypeScript**	**Ефективність генерації SQL**	**Безпека міграцій**
Sequelize	Data Mapper / Active Record (класичний)	Слабкий	Середня (схильність до проблеми N+1)	Середня (ручні або напівавтоматичні)
TypeORM	Data Mapper та Active Record	Добре (через класи та декоратори)	Середня (генерує складні для оптимізації запити)	Висока
Prisma	Декларативна схема (schema.prisma)	Відмінна (автоматична генерація типів клієнта)	Висока (вбудовані механізми оптимізації та батчингу)	Відмінна (автоматизований контроль версій схем)

Продовження таблиці 1.2

Drizzle ORM	SQL-like конструктор (нативний TypeScript)	Відмінна	Дуже висока (максимально наближена до сирого SQL)	Висока (потребує детального опису в коді)

Аналіз архітектурних підходів до побудови веб-застосунків

Для проєктування високонавантажених систем електронної комерції критично важливим є вибір загальної архітектурної моделі програмного забезпечення. Історично виділяють три основні підходи: монолітна архітектура, мікросервісна архітектура та триланкова клієнт-серверна архітектура з використанням REST API.

- Монолітна архітектура [17] об'єднує інтерфейс користувача, бізнес-логіку та доступ до даних в одному розгортаному модулі. Це спрощує початкову розробку, проте унеможливлює незалежне масштабування окремих модулів (наприклад, ізольоване масштабування каталогу товарів під час розпродажів).

- Мікросервісна архітектура [18, 19] розділяє систему на автономні сервіси, що взаємодіють через мережу. Вона забезпечує максимальну гнучкість, але суттєво підвищує складність розгортання, моніторингу та забезпечення цілісності даних між різними базами даних.

- Клієнт-серверна архітектура на основі REST API (з використанням SPA) [7, 20] є оптимальним компромісом. Вона передбачає повне відокремлення клієнтської частини (фронтенду) від серверної частини (бекенду).

Порівняння архітектурних підходів зображено в таблиці 1.3.

Таблиця 1.3

Порівняння архітектурних підходів для інтернет-магазину

Архітектурний шаблон	**Переваги**	**Недоліки**	**Доцільність для високих навантажень**
Моноліт	Швидка розробка, просте тестування, низькі накладні витрати на мережу	Складно масштабувати, довгий час збірки проєкту, зв'язаність коду	Низька (підходить лише для початкових етапів системи)

Продовження таблиці 1.3

Мікросервіси	Ізольоване масштабування сервісів, технологічна незалежність модулів	Складне налаштування CI/CD, проблема розподілених транзакцій	Висока (для масштабів великих корпорацій із багатьма командами)
Клієнт-сервер (SPA + REST API)	Чіткий розподіл зон відповідальності, легке кешування фронтенду на CDN, оптимізація трафіку	Потреба в окремому розгортанні клієнта і сервера, первинне завантаження SPA	Висока (ідеальний баланс швидкодії, масштабованості та швидкості розробки)

Розділення веб-застосунку на незалежний клієнтський додаток (наприклад, побудований на базі інструментів швидкої збірки Vite та React) та окремий REST API сервер (на базі експрес-фреймворків) дозволяє мінімізувати затримки передачі даних, оптимізувати обчислювальні ресурси серверів та гнучко керувати навантаженням на рівні хмарної інфраструктури розгортання [21, 22].

1.4. Висновки по розділу

У першому розділі було проведено комплексний огляд предметної області та досліджено ключові аспекти проєктування баз даних для високонавантажених веб-застосунків у сфері електронної комерції. За результатами аналізу виявлено, що сучасні інтернет-магазини стикаються з викликами експоненційного зростання обсягів даних та необхідністю миттєвої обробки складних пошукових запитів. Традиційні монолітні реляційні СУБД часто не витримують таких навантажень, що зумовлює потребу в пошуку нових підходів до архітектури бази даних без втрати транзакційної цілісності (ACID).

Для системного підходу до вирішення цих завдань було сформовано термінологічний базис, який включає поняття вертикального та горизонтального масштабування, NewSQL-систем, оптимізованої курсорної пагінації та сучасних інструментів ORM. На основі цього теоретичного підґрунтя проведено аналіз сучасних систем управління базами даних (детальне порівняння СУБД зображено в таблиці 1.1). Визначено, що для забезпечення відмовостійкості та автоматичного

горизонтального масштабування оптимальним рішенням є використання розподіленої хмарної NewSQL СУБД TiDB.

Разом із тим, розглянуто інструменти взаємодії серверної частини з базою даних (порівняльну характеристику ORM-систем зображено в таблиці 1.2). Встановлено, що декларативний підхід Prisma ORM забезпечує найкращу типобезпеку та високу продуктивність генерації SQL-запитів порівняно з класичними рішеннями. Також проаналізовано підходи до загальної побудови системи (порівняння архітектурних шаблонів зображено в таблиці 1.3), що дозволило довести доцільність використання клієнт-серверної архітектури на основі REST API для балансування навантаження та незалежного масштабування фронтенду і бекенду. Поєднання цих технологій утворює цілісну екосистему, яка дає змогу уникнути типових «вузьких місць»: неефективної пагінації, надлишкових запитів до бази даних та складнощів горизонтального масштабування.

Завдяки такому симбіозу вдається не лише розв'язати проблеми продуктивності на рівні бази даних, але й суттєво прискорити сам процес розробки. Автоматична генерація типів у Prisma усуває цілий клас помилок, пов'язаних із неузгодженістю даних між сервером і базою, а чітке розділення фронтенду та бекенду дозволяє командам працювати паралельно, незалежно масштабуючи окремі компоненти системи залежно від поточного навантаження.

Обґрунтований технологічний стек також вирізняється високою готовністю до інтеграції з хмарними платформами (зокрема, PaaS-рішеннями на кшталт Render). Це дозволяє не лише зменшити витрати ресурсів на адміністрування серверної інфраструктури, але й забезпечити безперебійне та гнучке розгортання нових версій застосунку. Таким чином, критичний огляд існуючих рішень та обґрунтування технологічного стеку повністю завершує етап теоретичного дослідження проблеми. Отримані результати є надійним фундаментом для переходу до другого розділу, де буде виконано детальний збір вимог зацікавлених сторін та сформульовано конкретну постановку задачі проєктування інтернет-магазину.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ ТА ПОСТАНОВКА ЗАДАЧІ

2.1. Збір та аналіз вимог користувачів

Процес проєктування будь-якого складного програмного забезпечення розпочинається з ідентифікації цільової аудиторії, ключових груп користувачів та детального аналізу їхніх потреб. Оскільки розробка ведеться у домені електронної комерції (інтернет-магазин побутової техніки), процес збору даних базувався на комплексному підході: аналізі існуючих рішень на ринку, моделюванні типових користувацьких сценаріїв (Use Cases) та вивченні бізнес-логіки сучасних маркетплейсів.

У результаті проведеного аналізу було виділено три ключові групи користувачів, кожна з яких взаємодіє з системою на різному рівні та висуває власні, специфічні запити до її функціональності.

1. Вимоги з боку покупців (клієнтів платформи)

Для кінцевих споживачів критичним фактором є зручність інтерфейсу (UX) та швидкість реакції системи. На основі аналізу сценаріїв поведінки визначено такі першочергові потреби:

- Швидкодія та пошук: Миттєвий пошук та гнучка фільтрація багатопараметричного каталогу товарів (за ціною, брендом, потужністю, енергокласом тощо). Це формує пряму вимогу до СУБД щодо оптимізації SELECT-запитів та ефективного індексування.
- Управління покупками: Зручна взаємодія з кошиком (Cart) та можливість відкладати товари у «список бажань» (Wishlist) для подальшого придбання.
- Прозорість історії: Наявність особистого кабінету з історією замовлень. При цьому клієнт повинен бачити ту назву та ціну товару, яка була актуальною саме на момент покупки (необхідність реалізації «знімків» даних, або snapshot-полів).

- Соціальна взаємодія: Можливість залишати відгуки, формувати рейтинг товарів та отримувати зворотний зв'язок від представників магазину.

2. Вимоги з боку адміністраторів

Ця група користувачів відповідає за щоденну операційну підтримку магазину. Їхня взаємодія із системою потребує надійності, автоматизації рутинних процесів та суворого розмежування прав доступу:

- Керування контентом: Наявність захищеної панелі керування (Admin Panel) для створення, редагування та деактивації товарів, ієрархічних категорій і брендів.
- Облік товарів: Відстеження складських запасів у реальному часі, можливість контролювати резерв товару під час активних замовлень.
- Обробка замовлень: Зручний інтерфейс для моніторингу нових замовлень, доступу до контактних даних та адрес доставки клієнтів, а також зміни статусів виконання (від «В очікуванні» до «Доставлено»).
- Модерація: Інструменти для перевірки та затвердження або відхилення відгуків покупців.

3. Вимоги з боку бізнесу (власників як користувачів системи)

Користувачі з боку бізнесу зацікавлені не стільки в інтерфейсних рішеннях, скільки в загальній архітектурній надійності, рентабельності та безпеці платформи:

- Транзакційна надійність (ACID): Гарантія цілісності фінансових даних. Процес оформлення замовлення повинен бути строго атомарним: система має одночасно списувати товар зі складу, очищати кошик і створювати чек. Збій на будь-якому етапі повинен скасовувати всю транзакцію.
- Масштабованість: Здатність архітектури витримувати раптові пікові навантаження (наприклад, у періоди сезонних розпродажів) без деградації швидкості роботи та без необхідності повної зупинки системи для апгрейду серверів.
- Кібербезпека: Надійне криптографічне хешування паролів та захист персональних даних клієнтів відповідно до сучасних стандартів веброзробки.

Таким чином, етап збору та аналізу вимог користувачів дозволив чітко структурувати бізнес-потреби. Розуміння очікувань покупців, адміністраторів та власників бізнесу стало надійним базисом для наступного кроку, перетворення цих абстрактних запитів у формалізований перелік функціональних і нефункціональних вимог до архітектури програмного забезпечення та схеми бази даних.

2.2. Формулювання функціональних та нефункціональних вимог

На основі зібраних потреб користувачів (підрозділ 2.1) та специфіки предметної області, наступним кроком є формалізація цих потреб у вигляді конкретних вимог до системи. Вони традиційно поділяються на функціональні (описують, що саме має робити веб-застосунок) та нефункціональні (визначають, як система повинна працювати: її продуктивність, безпеку та надійність).

Функціональні вимоги

Функціонал спроєктованого інтернет-магазину "TechnoBud" логічно розділяється на декілька підсистем, кожна з яких має власний набір бізнес-правил:

Підсистема каталогу та пошуку:

- Система повинна підтримувати багаторівневу ієрархію категорій (через механізм самопосилань) та прив'язку товарів до конкретних брендів.
- Кожен товар повинен мати набір базових характеристик (ціна, залишок, акційна ціна), а також можливість додавання динамічних технічних специфікацій та кількох зображень.
- Реалізація механізму складного пошуку та фільтрації товарів за різними параметрами (ціновий діапазон, бренд, рейтинг, наявність на складі, специфічні характеристики: колір, енергоклас).

Підсистема клієнтського досвіду (Кошик та Список бажань):

- Система повинна дозволяти додавати товари до списку бажань (Wishlist) та кошика (Cart).
- Кошик має бути доступним як для авторизованих клієнтів, так і для

незареєстрованих відвідувачів сайту (з ідентифікацією через унікальний sessionId).

- Під час зміни кількості товару в кошику система зобов'язана перевіряти доступний залишок на складі (поле stock).

Підсистема оформлення замовлень (Checkout):

- Користувач повинен мати можливість вибрати існуючу адресу доставки або додати нову під час оформлення.
- Вимога збереження історичної точності: При створенні позицій замовлення система повинна фіксувати актуальну на момент покупки назву товару, артикул та назву бренду (у вигляді snapshot-полів), щоб історія покупок залишалася коректною навіть у разі подальшого видалення або перейменування товару в каталозі.

Підсистема соціальної взаємодії:

- Клієнти повинні мати можливість залишати відгуки з рейтингом від 1 до 5 зірок.
- Адміністратори повинні мати інструментарій для модерації відгуків (isApproved) та надання відповідей на них (реалізація деревоподібної структури коментарів).

Підсистема адміністрування (Admin Panel):

- Наявність закритої панелі з розмежуванням прав доступу, де адміністратор може виконувати CRUD-операції (створення, читання, оновлення, видалення) над товарами, користувачами, брендами та категоріями.
- Автоматична генерація символьних посилань (slug) для нових товарів та категорій.

Нефункціональні вимоги

Оскільки розробка ведеться з прицілом на високі навантаження, нефункціональні вимоги є критичними для вибору СУБД та архітектури бекенду:

Продуктивність та оптимізація бази даних:

- Час відгуку пошукових запитів до каталогу має бути мінімізованим. Для цього система повинна використовувати індексацію зовнішніх ключів та

полів, що найчастіше беруть участь у фільтрації.

- Замість класичної посторінкової пагінації (з використанням LIMIT та OFFSET), яка деградує при великих обсягах даних, система повинна використовувати оптимізовану курсорну пагінацію (Cursor-based Pagination) у комбінації з нескінченним прокручуванням (Infinite Scroll) на клієнті.
- СУБД має забезпечувати уникнення проблеми "N+1 запиту" шляхом використання сучасних ORM (Prisma) для об'єднання запитів.

Транзакційність та цілісність даних (ACID):

- Процес створення замовлення повинен бути суворо транзакційним. Списання товару зі складу, резервування (reservedStock), створення чека та очищення кошика мають виконуватися як єдина атомарна операція з можливістю відкату (rollback) у разі помилки на будь-якому етапі.
- Для запобігання похибок округлення у фінансових розрахунках усі грошові поля мають використовувати тип даних із фіксованою точністю (Decimal замість Float).

Безпека та авторизація:

- Передача даних повинна відбуватися виключно захищеними каналами (SSL/TLS).
- Паролі користувачів суворо заборонено зберігати у відкритому вигляді; вони повинні бути зашифровані за допомогою надійних криптографічних алгоритмів (наприклад, bcryptjs).
- Автентифікація клієнтів та взаємодія між фронтендом і бекендом має здійснюватися через безстанові (stateless) токени доступу (JSON Web Tokens — JWT).

Масштабованість та розгортання:

- Архітектура серверної частини повинна підтримувати горизонтальне масштабування.
- База даних повинна мати можливість безшовної міграції від локальної реляційної СУБД (MariaDB) до розподіленої хмарної екосистеми (TiDB Cloud) без переписування прикладного програмного коду.

Сформульовані вимоги слугують безпосереднім технічним завданням для переходу до етапу проєктування структури реляційної бази даних та написання коду застосунку.

2.3. Постановка задачі проєктування системи

З урахуванням визначених потреб користувачів та сформульованих функціональних і нефункціональних вимог, фінальним кроком етапу аналізу є чітка постановка інженерної задачі. Вона визначає головну мету, межі відповідальності майбутньої системи, а також технічні припущення та обмеження, в рамках яких буде здійснюватися розробка.

Головна мета проєкту

Основною метою є проєктування, нормалізація та оптимізація реляційної бази даних для високонавантаженого інтернет-магазину побутової техніки «TechnoBud», а також розробка повнофункціонального клієнт-серверного веб-застосунку, який на практиці продемонструє ефективність і надійність закладених архітектурних рішень[27, 28].

Конкретні завдання розробки (Scope of Work)

Для досягнення поставленої мети необхідно виконати такий комплекс завдань:

- Створити концептуальну (ER-діаграма) та фізичну моделі бази даних, провести нормалізацію до третьої нормальної форми (3НФ) із контрольованою денормалізацією для історичних даних [29].
- Визначити оптимальні типи даних, спроєктувати систему первинних і зовнішніх ключів, а також розробити стратегію індексації для прискорення складних пошукових запитів.
- Реалізувати взаємодію серверної частини з базою даних через сучасну систему об'єктно-реляційного відображення (Prisma ORM), адаптовану для роботи з хмарною розподіленою СУБД (TiDB).

- Розробити захищений REST API (Express.js) для обробки бізнес-логіки, гарантуючи транзакційну цілісність ключових операцій (зокрема оформлення замовлення).
- Створити інтерактивний клієнтський інтерфейс (React SPA) із застосуванням механізмів зниження навантаження на базу даних (нескінченне прокручування та курсорна пагінація).

Межі системи (System Boundaries)

Проєкт охоплює повний внутрішній цикл роботи електронної комерції: від перегляду багатопараметричного каталогу та управління кошиком до модерації відгуків і зміни статусів замовлень в адміністративній панелі.

Водночас, з метою збереження фокусу на проєктуванні бази даних, поза межами даної роботи залишаються:

- Інтеграція з реальними платіжними шлюзами (наприклад, LiqPay, Stripe чи банківськими API). Оплата фіксується на рівні статусу замовлення.
- Синхронізація зі сторонніми логістичними сервісами (наприклад, API «Нової Пошти»). Адреса доставки зберігається в базі даних як текстовий формат або набір базових полів.
- Налаштування масових email- або SMS-розсилок для нотифікації клієнтів.

Припущення та технічні обмеження (Assumptions & Constraints)

Проєктування ведеться з урахуванням того, що кінцеве розгортання системи відбуватиметься у хмарному середовищі з використанням PaaS-рішень (Render) та DBaaS (TiDB Cloud Serverless). З огляду на це діють такі обмеження:

- Апаратні обмеження: Використання безкоштовних хмарних тарифів суворо лімітує доступні обчислювальні ресурси та кількість одночасних з'єднань із базою даних. Це диктує необхідність максимальної оптимізації SQL-запитів і обов'язкового використання пулу з'єднань (Connection Pool) на рівні серверної частини.
- Технологічні обмеження: База даних повинна зберігати сувору сумісність із протоколом MySQL, щоб код ORM-шару міг без змін функціонувати як у

локальному середовищі розробки (MariaDB), так і в хмарному кластері.

Кінцевий результат

Результатом виконання поставленої задачі має стати розгорнутий у хмарній інфраструктурі, повністю працездатний та протестований веб-застосунок. База даних цього застосунку повинна забезпечувати консистентність фінансових транзакцій, витримувати імітацію пікових навантажень (завдяки індексації та правильній пагінації) та мати достатній запас гнучкості для подальшого масштабування бізнесу без необхідності рефакторингу ядра схеми.

2.4. Висновки по розділу

У другому розділі було виконано ключовий етап підготовки до розробки проведено ґрунтовний збір вимог та сформовано детальне технічне завдання на проєктування системи. У ході дослідження ідентифіковано три основні групи користувачів майбутньої платформи: покупців, адміністраторів та власників бізнесу. Для кожної з цих груп було формалізовано специфічні очікування. Покупці потребують високої швидкодії інтерфейсу, зручного багатокритеріального пошуку та прозорого управління кошиком. Адміністраторам необхідні надійні інструменти для модерації контенту, управління статусами замовлень та контролю складських залишків. З боку бізнесу головним пріоритетом виступає забезпечення безперебійної роботи платформи, збереження фінансової цілісності та гарантування безпеки даних навіть в умовах стрімкого зростання клієнтського трафіку.

На основі зібраних даних сформовано ключові функціональні та нефункціональні вимоги. До функціональних належать: реалізація атомарного механізму оформлення замовлень, збереження «знімків» товарних характеристик для історичної точності покупок, підтримка ієрархічного каталогу та багаторівневих відгуків. З огляду на специфіку високонавантажених систем, визначено нефункціональні вимоги: впровадження курсорної пагінації з нескінченним прокручуванням для оптимізації роботи бази даних, суворі

стандарти безпеки (хешування паролів, JWT-авторизація, SSL-шифрування) та використання типів даних із фіксованою точністю для фінансових розрахунків.

З урахуванням обраного технологічного стеку та середовища розгортання було чітко окреслено межі відповідальності системи та технічні обмеження. Оскільки інфраструктура спирається на хмарні сервіси (PaaS для хостингу та DBaaS для бази даних), архітектура повинна враховувати суворі ліміти на обчислювальні ресурси та кількість одночасних з'єднань. Це вимагає впровадження пулу з'єднань на серверному рівні та максимальної оптимізації ORM-запитів для уникнення проблеми "N+1". Фокус роботи свідомо зосереджено на внутрішній бізнес-логіці інтернет-магазину та швидкодії бази даних, тоді як інтеграції із зовнішніми платіжними шлюзами чи логістичними API залишено поза межами поточного проєкту.

Таким чином, результати другого розділу формують вичерпну та однозначну постановку інженерної задачі. Трансформація абстрактних побажань користувачів у конкретні технічні специфікації дозволяє мінімізувати архітектурні ризики на етапі реалізації. Отримана документація є надійним орієнтиром для переходу до третього розділу, у якому буде безпосередньо розроблено багаторівневу архітектуру застосунку, побудовано UML-діаграми ключових процесів, а також виконано концептуальне та фізичне моделювання оптимальної схеми реляційної бази даних.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ

3.1. Розробка архітектури програмного забезпечення

Отримавши чітку постановку задачі та сформульовані функціональні й нефункціональні вимоги (Розділ 2), наступним кроком є безпосереднє технічне проєктування системи. Архітектура програмного забезпечення визначає загальну структурну організацію компонентів, способи їх взаємодії та правила інтеграції із зовнішніми сервісами. Для високонавантаженого інтернет-магазину «TechnoBud» обрано клієнт-серверну модель із чітким розділенням на фронтенд (SPA) та бекенд (REST API), що забезпечує баланс між продуктивністю, масштабованістю та швидкістю розробки.

Загальна схема взаємодії компонентів системи представлена на рисунку 3.1. Користувач через браузер взаємодіє з односторінковим додатком (React), який надсилає HTTP-запити до серверної частини на платформі Node.js із фреймворком Express. Серверна частина, у свою чергу, через ORM-прошарок (Prisma Client) виконує операції з базою даних, яка може бути розгорнута як локально (MariaDB), так і в хмарному середовищі (TiDB Cloud). Такий розподіл дозволяє незалежно масштабувати клієнтський інтерфейс і серверну логіку залежно від поточного навантаження.

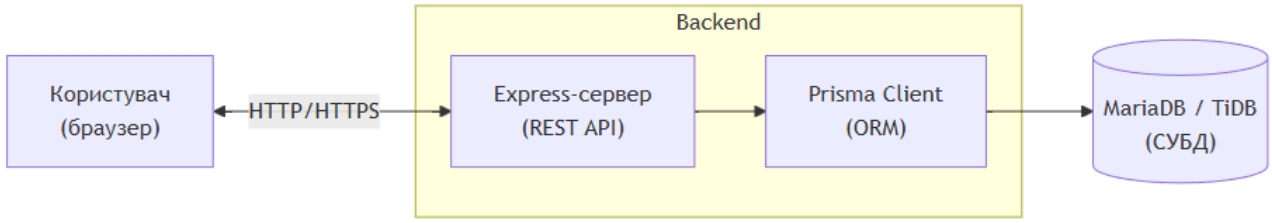

Рисунок 3.1 — Схема архітектури веб-застосунку

Серверна частина спроєктована за багатошаровим принципом, що забезпечує модульність, зручність тестування та легкість внесення змін. Її структура схематично зображена на рисунку 3.2. На верхньому рівні розташовані

маршрутизатори (Routes), які приймають вхідні HTTP-запити та спрямовують їх до відповідних контролерів. Контролери (Controllers) реалізують бізнес-логіку: виконують валідацію вхідних даних, звертаються до шару доступу до даних (Prisma Client) та формують JSON-відповіді. Такий розподіл обов'язків унеможливлює змішування логіки маршрутизації, обробки даних і взаємодії з базою даних.

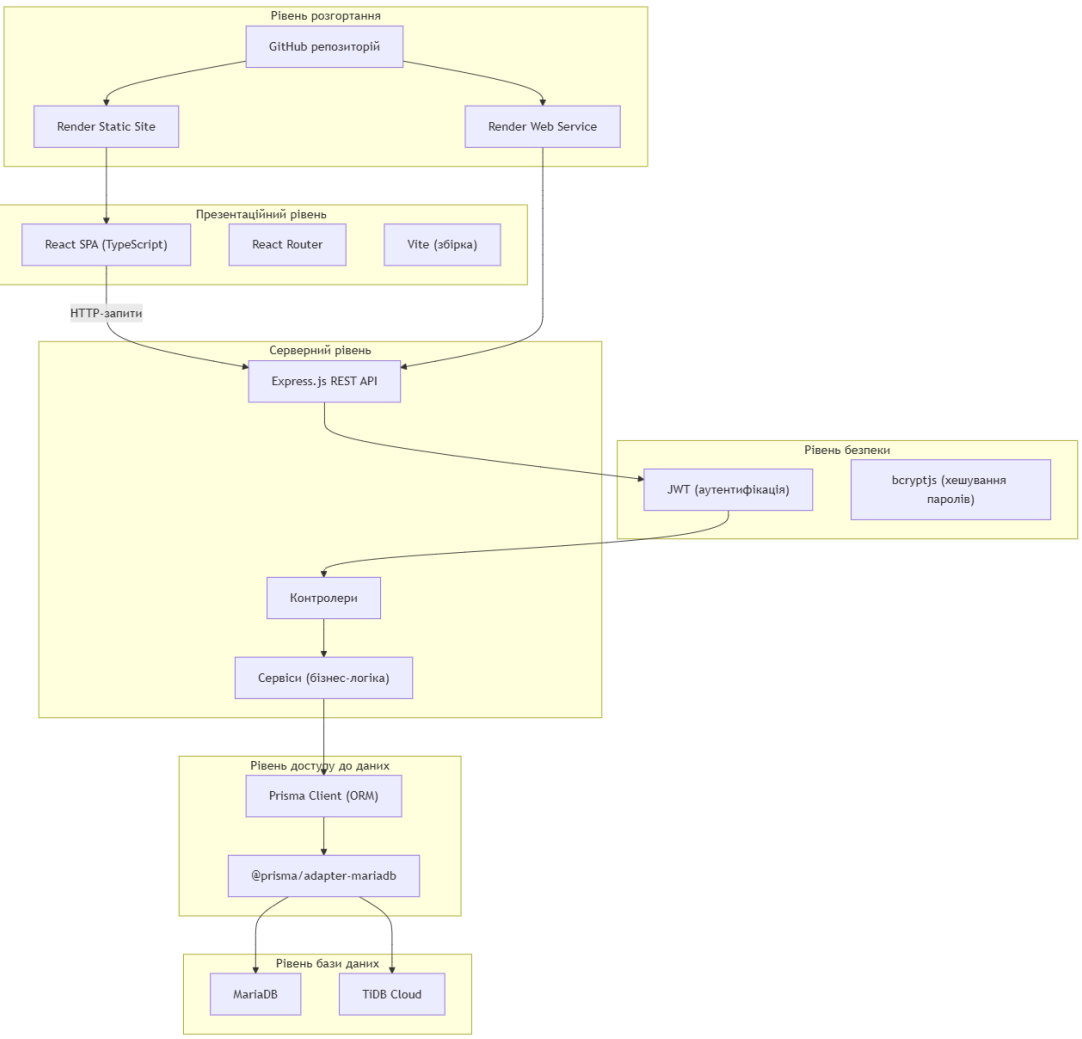

Рисунок 3.2 — Багатошарова архітектура веб-застосунку

Важливу роль в архітектурі відіграють проміжні обробники (middleware). Для обробки CORS-запитів, парсингу JSON-тіла, логування кожного запиту, а також для забезпечення безпеки (перевірка JWT-токенів, розмежування прав доступу ADMIN/CUSTOMER) застосовано стандартні та власні middleware-

функції. Завдяки цьому безпекова логіка винесена з контролерів, що зменшує дублювання коду та підвищує його читабельність.

Більш детально компоненти серверної частини показано на рисунку 3.3. Маршрутизатор спрямовує запит до відповідного контролера через ланцюжок middleware (авторизація, валідація). Контролер використовує Prisma Client для формування оптимізованих SQL-запитів, які через пул з'єднань надсилаються до СУБД [32].

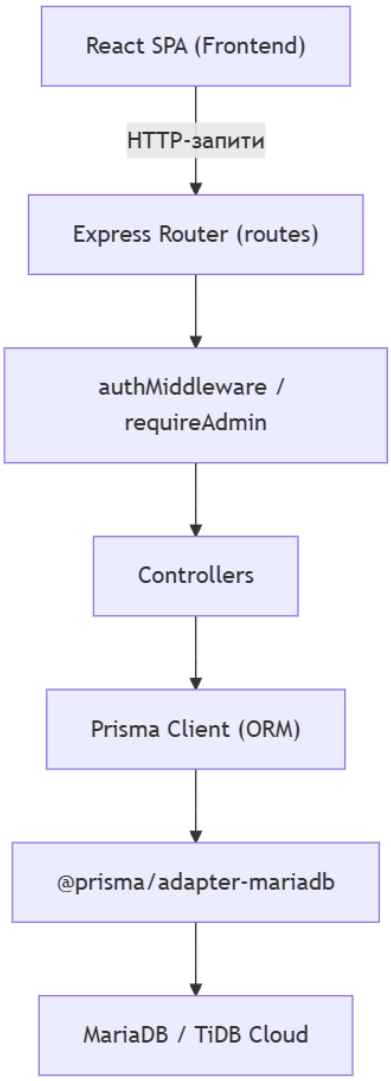

Рисунок 3.3 — Схема компонентів серверної частини

Клієнтська частина являє собою односторінковий додаток (Single Page Application), побудований на бібліотеці React із використанням TypeScript. Для маршрутизації на стороні клієнта застосовано React Router, що дає змогу

користувачеві переміщуватися між сторінками каталогу, кошика, особистого кабінету та адміністративної панелі без перезавантаження сторінки. Інструментом збірки обрано Vite, який забезпечує швидке гаряче оновлення під час розробки та оптимізовану виробничу збірку. Така конфігурація дозволяє досягти високої продуктивності інтерфейсу та мінімізувати обсяг переданих даних [35].

Взаємодія між клієнтською та серверною частинами здійснюється через REST API, спроєктоване відповідно до архітектурних принципів Representational State Transfer. API оперує основними сутностями системи (користувачі, товари, категорії, бренди, замовлення, відгуки, кошик, список бажань) через стандартні HTTP-методи: GET (отримання даних), POST (створення), PUT/PATCH (оновлення), DELETE (видалення). Усі відповіді сервера мають формат JSON і супроводжуються відповідними статус-кодами. Для захисту конфіденційних даних застосовано автентифікацію на основі JSON Web Tokens (JWT) та хешування паролів бібліотекою bcryptjs. Передача даних між клієнтом і сервером, а також між сервером і хмарною СУБД відбувається виключно захищеними каналами (SSL/TLS).

Таким чином, спроєктована архітектура поєднує переваги сучасних підходів до веб-розробки: швидкий та інтерактивний фронтенд на React із незалежним масштабуванням, модульний бекенд на Express із чітким розподілом відповідальності та продуктивний шар доступу до даних через Prisma Client. Це створює надійний фундамент для реалізації всіх функціональних вимог та забезпечення високої продуктивності системи за умов зростаючих навантажень.

3.2. Концептуальне моделювання та бізнес-процеси (UML, ER-модель)

Після розробки загальної архітектури системи наступним кроком є детальне моделювання її інформаційного ядра, бази даних, а також формалізація ключових бізнес-процесів за допомогою мови UML. Концептуальна модель даних, представлена у вигляді ER-діаграми (Entity-Relationship), дозволяє абстрагуватися від особливостей конкретної СУБД і зосередитися на логічній

структурі предметної області: визначити основні сутності, їхні атрибути та зв’язки між ними. Паралельно з цим UML-моделювання дає змогу візуалізувати динаміку системи, послідовність дій під час виконання критично важливих сценаріїв.

Концептуальна модель бази даних (ER-діаграма)

На основі аналізу вимог (підрозділи 2.1, 2.2) було виділено 15 ключових сутностей, які охоплюють усю предметну область інтернет-магазину побутової техніки. Їхнє призначення та взаємозв’язки відображено на ER-діаграмі (рисунок 3.4), а повний формальний опис структури у вигляді DBML-коду наведено в Додатку А.

До основних сутностей належать:

Category (Категорія) — ієрархічна структура каталогу, реалізована через самопосилання parentId, що дозволяє створювати дерево підкатегорій довільної глибини.

Brand (Бренд) — виробник товару з унікальним символьним кодом slug, логотипом та ознакою активності.

Product (Товар) — центральна сутність, яка об’єднує комерційні атрибути (ціна, стара ціна, залишок на складі), технічні характеристики (потужність, колір, матеріал, енергоклас, гарантія, вага), лічильники рейтингу та відгуків, а також прапорці активності й рекомендацій.

ProductImage (Зображення товару) та ProductSpecification (Специфікація) — слабкі сутності, пов’язані з товаром зв’язком «один-до-багатьох», що забезпечує гнучкість у зберіганні довільної кількості фотографій і технічних параметрів.

User (Користувач) — обліковий запис покупця або адміністратора; роль визначається переліком UserRole (CUSTOMER, ADMIN).

Address (Адреса доставки) — зберігає деталізацію адреси (країна, місто, вулиця тощо) та позначає одну з адрес як основну.

Cart (Кошик) та **CartItem** (Позиція кошика) — забезпечують тимчасове зберігання обраних товарів перед оформленням замовлення; кошик може бути прив’язаний до авторизованого користувача або до сесії неавторизованого

відвідувача.

Wishlist (Список бажань) та **WishlistItem** (Позиція списку бажань) — дозволяють користувачам відкладати товари для майбутніх покупок.

Order (Замовлення) та **OrderItem** (Позиція замовлення) — фіксують факт купівлі; позиції замовлення містять не лише посилання на товар, а й снапшоти (productNameSnapshot, brandNameSnapshot, skuSnapshot), завдяки яким історія покупок залишається незмінною навіть після редагування чи видалення товарів із каталогу.

Review (Відгук) — оцінка та коментар до товару; ієрархічна структура «відгук-відповідь» реалізована через самопосилання parentId.

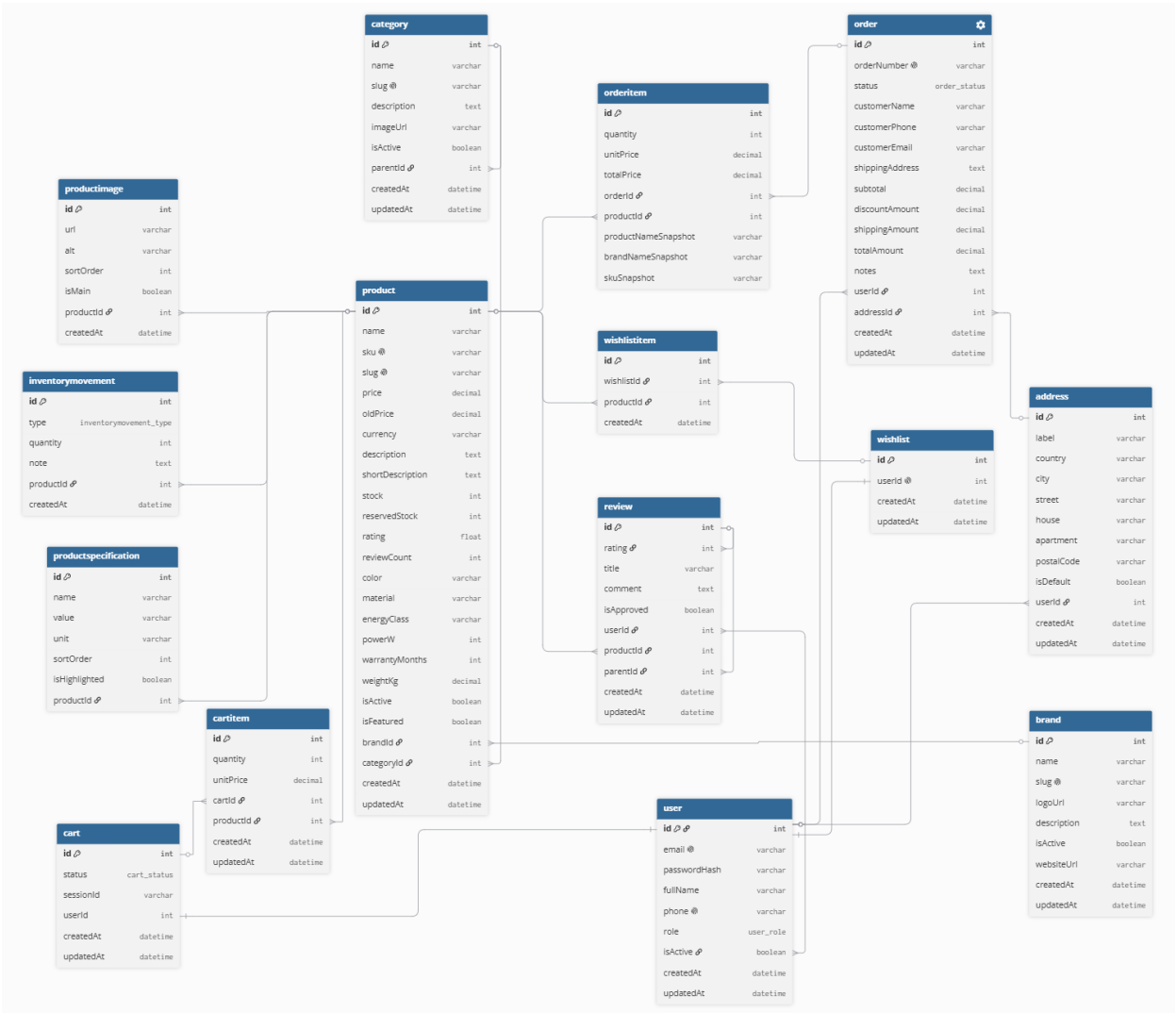

Рисунок 3.4 — ER-діаграма бази даних

Усі зв’язки між сутностями є зв’язками типу «один-до-багатьох» (1:N), за винятком зв’язку User - Cart, який є «один-до-одного» (кожен авторизований користувач має рівно один активний кошик). Зовнішні ключі налаштовано з каскадним видаленням для композиційних зв’язків, що гарантує цілісність даних на рівні СУБД.

Моделювання бізнес-процесів (UML)

Поряд зі статичною моделлю даних важливо відобразити динаміку системи, особливо для критичних операцій, де задіяна транзакційна логіка. Одним із таких процесів є оформлення замовлення, яке охоплює кілька атомарних кроків: перевірку наявності товарів, створення записів замовлення та його позицій, оновлення складських залишків і очищення кошика. Для наочної демонстрації цього сценарію побудовано UML-діаграму послідовності (Sequence Diagram), подану на рисунку 3.5.

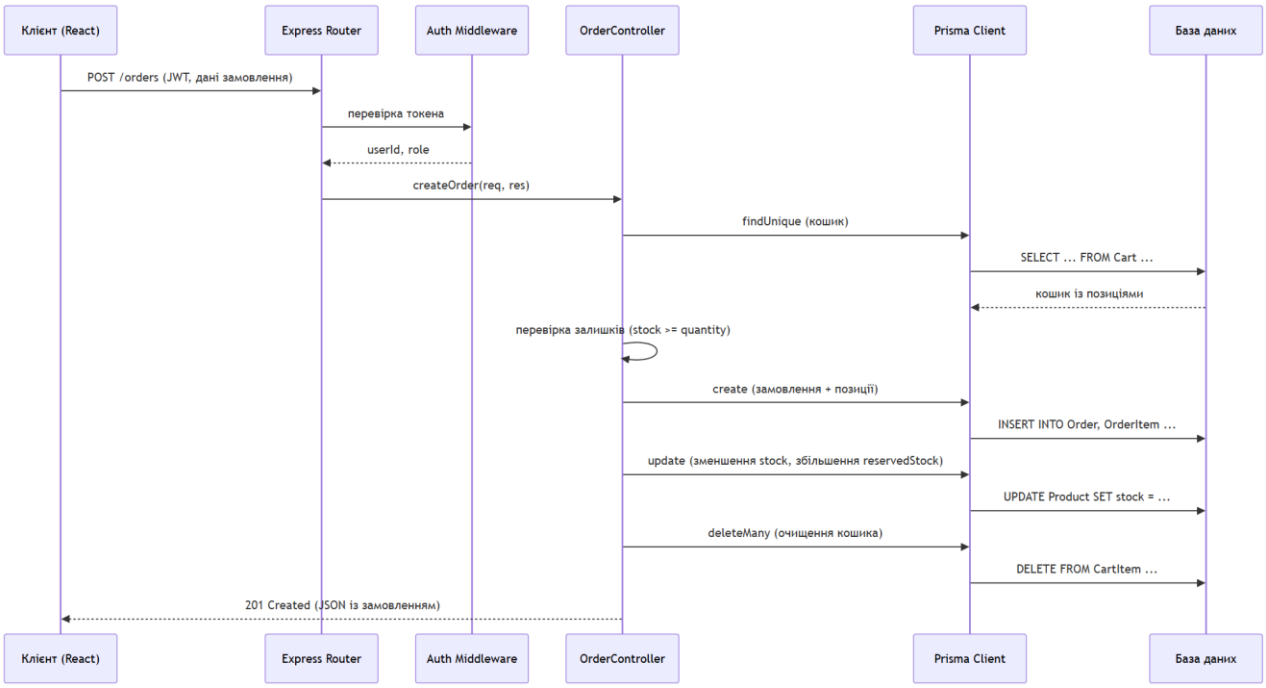

Рисунок 3.5 — UML-діаграма послідовності для оформлення замовлення

Діаграма демонструє такий алгоритм:

- Клієнт (браузер) надсилає POST-запит на створення замовлення до серверного API.

- Контролер замовлень перевіряє JWT-токен та отримує ідентифікатор користувача.
- Виконується послідовна взаємодія з базою даних: завантаження вмісту кошика з пов’язаними товарами, перевірка достатності залишків на складі, підрахунок сум.
- У межах єдиної транзакції Prisma одночасно створюється замовлення з позиціями, зменшується доступний залишок товарів та очищується кошик.
- У разі успіху клієнт отримує підтвердження з деталями замовлення; у випадку помилки (наприклад, нестача товару) відбувається відкат усіх змін, і клієнт отримує відповідне повідомлення.

Така візуалізація підтверджує, що спроєктована архітектура та структура бази даних здатні підтримувати сувору транзакційну цілісність, гарантуючи коректність фінансових і складських операцій.

Таким чином, у цьому підрозділі розроблено концептуальну модель бази даних, яка охоплює всі 15 сутностей із необхідними зв’язками та обмеженнями, а також виконано UML-моделювання ключового бізнес-процесу, оформлення замовлення. Отримані моделі є безпосередньою основою для фізичного проєктування БД та реалізації програмного коду, що розглядатимуться в наступних розділах.

3.3. Вибір та обґрунтування технологій розробки

Спираючись на проведений у розділі 1 порівняльний аналіз СУБД, ORM-систем та архітектурних підходів, а також на сформульовані в розділі 2 функціональні й нефункціональні вимоги, на цьому етапі проєктування приймається остаточне рішення щодо технологічного стеку веб-застосунку «TechnoBud». Ключовими критеріями вибору виступають: продуктивність та масштабованість бази даних, типобезпека кодової бази, зручність розробки й супроводження, а також можливість безперешкодного переходу від локального середовища до хмарної інфраструктури.

Мова програмування: TypeScript

І серверна, і клієнтська частини проєкту написані мовою TypeScript — надмножиною JavaScript, що додає статичну типізацію. Такий вибір дозволяє виявляти значну частину помилок на етапі компіляції, підвищує читабельність коду та спрощує рефакторинг. Крім того, TypeScript нативно підтримується Prisma, React та Express, що забезпечує наскрізну типобезпеку всієї системи [23].

Серверна частина: Express.js

Як основа для REST API обрано фреймворк Express.js — де-факто стандарт для створення веб-серверів на платформі Node.js. Він має мінімалістичний дизайн, велику кількість проміжних обробників (middleware) і стабільну продуктивність. Завдяки простоті маршрутизації та інтеграції з Prisma Client, Express дозволяє швидко реалізувати всю необхідну логіку: автентифікацію, CRUD-операції для товарів, категорій, брендів, обробку замовлень тощо [21, 32].

Клієнтська частина: React + Vite

Для побудови односторінкового додатку (SPA) використано бібліотеку React у поєднанні з інструментом збірки Vite. React забезпечує високу інтерактивність інтерфейсу, зручне керування станом і компонентний підхід, який спрощує повторне використання коду. Vite, своєю чергою, надає швидке гаряче оновлення під час розробки та оптимізовану виробничу збірку. Маршрутизація на клієнті реалізована за допомогою React Router [22, 35].

ORM та взаємодія з БД: Prisma

Як показав аналіз у підрозділі 1.3 (таблиця 1.2), Prisma поєднує переваги суворої типізації, декларативного опису схеми, автоматичної генерації клієнта та зручного інструменту міграцій. Для проєкту особливо важливою є підтримка адаптера @prisma/adapter-mariadb, який дозволяє працювати з MariaDB локально і з TiDB у хмарі через єдиний програмний інтерфейс. Крім того, Prisma Studio надає графічний інтерфейс для перегляду та редагування даних, що зручно на етапі наповнення магазину [13, 24].

Системи управління базами даних: MariaDB / TiDB

Для локальної розробки використовується MariaDB — безкоштовна,

стабільна та повністю сумісна з MySQL СУБД. Для виробничого середовища обрано хмарну TiDB Cloud, яка, як було встановлено в таблиці 1.1, забезпечує горизонтальне масштабування, високу доступність і при цьому зберігає сумісність із MySQL-протоколом. Такий підхід дозволяє вести розробку локально без додаткових витрат, а при розгортанні отримати всі переваги розподіленої хмарної СУБД [9, 11].

Інструменти автентифікації та безпеки

Для автентифікації користувачів застосовано JSON Web Tokens (JWT) — легковаговий, самодостатній механізм передачі тверджень між сторонами. Паролі користувачів зберігаються в базі даних у вигляді хешів, згенерованих бібліотекою bcryptjs. Така комбінація забезпечує надійний захист облікових записів і дозволяє реалізувати розмежування ролей (CUSTOMER / ADMIN) без надмірного ускладнення коду [25].

Хмарний хостинг: Render

Для розгортання веб-застосунку обрано платформу Render. Вона надає безкоштовний тариф для Web Service (серверна частина) і Static Site (клієнтська частина), підтримує автоматичний деплой із GitHub-репозиторію та має інтеграцію з Node.js «з коробки». Простота налаштування та наявність вбудованої підтримки змінних оточення (DATABASE_URL, JWT_SECRET тощо) зробили Render оптимальним вибором для цього проєкту [26].

Узагальнену структуру обраного технологічного стеку наведено в таблиці 3.1.

Таблиця 3.1

Стек технологій проєкту «TechnoBud»

Рівень	**Технологія**	**Призначення**
Мова програмування	TypeScript	Статична типізація, підвищення надійності коду
Клієнтська частина	React, React Router, Vite	SPA, маршрутизація, збірка
Серверна частина	Express.js	REST API, обробка запитів
ORM	Prisma, @prisma/adapter-mariadb	Об’єктно-реляційне відображення, міграції

Продовження таблиці 3.1

База даних (локальна)	MariaDB	Розробка та тестування
База даних (хмарна)	TiDB Cloud	Виробниче середовище, масштабування
Автентифікація	JSON Web Token, bcryptjs	Безпека, хешування паролів
Хмарний хостинг	Render	Деплой серверної та клієнтської частин

Обраний стек повністю задовольняє всі сформульовані вимоги: забезпечує цілісність даних, продуктивність, масштабованість, гнучкість, безпеку та підтримку транзакцій. Крім того, використання TypeScript на всіх рівнях і тісна інтеграція Prisma з обраними СУБД дозволяють зосередитися безпосередньо на реалізації бізнес-логіки, мінімізуючи технічні ризики. У наступному розділі буде детально описано практичну реалізацію проєкту на основі цього стеку.

3.4. Висновки по розділу

У третьому розділі було виконано ключовий етап проєктування системи, від загальної архітектурної концепції до деталізованих моделей даних і остаточного вибору інструментів розробки. Розроблена архітектура програмного забезпечення базується на клієнт-серверному підході з чітким розділенням на фронтенд (SPA на React) та бекенд (REST API на Express), що забезпечує незалежне масштабування компонентів та високу продуктивність інтерфейсу. Застосування багатошарової структури на серверному боці (маршрутизація, контролери, ORM-прошарок) у поєднанні з проміжними обробниками для автентифікації та авторизації створює гнучку, безпечну й зручну для тестування архітектуру.

У рамках концептуального моделювання побудовано ER-діаграму бази даних із 15 сутностями, яка повністю охоплює предметну область інтернет-магазину побутової техніки: ієрархічний каталог, товари з гнучкими специфікаціями, користувачів із розмежуванням ролей, кошик, список бажань, замовлення з історичними снапшотами даних, а також систему відгуків із деревоподібною структурою. Визначені зв’язки «один-до-багатьох» і каскадні

обмеження цілісності гарантують несуперечливість даних на рівні СУБД. Додатково за допомогою UML-діаграми послідовності змодельовано критичний бізнес-процес оформлення замовлення, що підтвердило здатність архітектури підтримувати атомарні транзакції з відкатом у разі помилок.

На основі порівняльного аналізу з першого розділу та сформульованих у другому розділі вимог було остаточно обґрунтовано технологічний стек проєкту: TypeScript як єдину мову програмування, React та Vite для клієнтської частини, Express і Prisma для серверної, MariaDB для локальної розробки та TiDB Cloud для виробничого середовища, а також JWT і bcryptjs для безпеки та Render для хмарного розгортання. Таке поєднання забезпечує наскрізну типобезпеку, високу продуктивність взаємодії з базою даних і можливість безшовного переходу між локальним і хмарним оточеннями. Крім того, декларативний підхід Prisma до опису схеми та автоматична генерація типізованого клієнта усувають цілий клас помилок, пов’язаних із неузгодженістю між структурою бази даних і прикладним кодом, що особливо важливо в умовах динамічного розвитку функціоналу. Архітектурна гнучкість, закладена на етапі проєктування, дозволяє в майбутньому безболісно розширювати систему новими модулями або мігрувати окремі компоненти на мікросервісну модель, якщо цього вимагатимуть масштаби бізнесу. Це гарантує довготривалу стабільність інвестицій у розробку та суттєво спрощує подальше супроводження системи.

Таким чином, результати третього розділу формують вичерпний проєктний базис, який повністю відповідає встановленим функціональним і нефункціональним вимогам. Отримані архітектурні рішення, концептуальна модель даних та обґрунтований стек технологій є безпосереднім фундаментом для переходу до практичної реалізації системи, опису програмного коду, фізичного проєктування бази даних і впровадження оптимізаційних заходів, чому буде присвячено наступний розділ роботи.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ПРОЄКТУ

4.1. Опис розробки програмного коду та структури репозиторію

Практична реалізація веб-застосунку «TechnoBud» базується на чіткому розділенні кодової бази на дві незалежні частини, серверну (backend) та клієнтську (frontend), що відповідає обраній клієнт-серверній архітектурі. Вихідний код організовано у вигляді монорепозиторію, який містить кореневу директорію проекту, директорію server для серверної логіки та директорію frontend для клієнтського SPA-додатку. Така структура дозволяє незалежно керувати залежностями, скриптами збірки та розгортанням кожної з частин системи.

Кореневий рівень містить файли загальної конфігурації: package.json зі скриптами для запуску всього проекту, файл змінних оточення .env, конфігурацію TypeScript (tsconfig.json), а також директорію prisma, у якій знаходиться файл схеми бази даних schema.prisma та файли міграцій. Така централізація ORM-конфігурації на верхньому рівні є вимогою Prisma і забезпечує єдине джерело істини щодо структури бази даних для всього проекту.

Серверна частина реалізована на платформі Node.js із використанням фреймворку Express.js та мови TypeScript. Вхідною точкою є файл server.ts, який виконує ініціалізацію сервера: завантажує змінні оточення, налаштовує підключення до бази даних через адаптер @prisma/adapter-mariadb, реєструє глобальні middleware (CORS, парсинг JSON, логування запитів) та монтує маршрутизатори для кожної групи API-ендпоінтів. Для підключення до СУБД використовується пул з'єднань (connectionLimit: 10) та SSL-шифрування, що гарантує безпеку передачі даних між сервером застосунку та хмарною TiDB.

Серверний код організовано за модульним принципом. У директорії routes/ зберігаються файли маршрутизаторів (наприклад, products.routes.ts, orders.routes.ts, auth.routes.ts), які визначають URL-шляхи та зв'язують їх із відповідними контролерами. Контролери, розташовані в директорії controllers/, містять функції-обробники для кожного HTTP-методу. Вони реалізують бізнес-

логіку: валідують вхідні дані, формують запити до бази даних через Prisma Client та повертають JSON-відповіді з відповідними статус-кодами. Окремо виділено middleware-функції (middleware/auth.ts, middleware/requireAdmin.ts), які забезпечують перевірку JWT-токенів та розмежування прав доступу для звичайних покупців і адміністраторів. Ініціалізація Prisma Client винесена в окремий модуль prisma.ts, що дозволяє імпортувати єдиний екземпляр клієнта в усі контролери без повторного налаштування. Код нижче демонструє, як реалізовано перевірку JWT-токену та розмежування прав доступу для адміністративних маршрутів:

```
export function requireAdmin(req: Request, res: Response, next: NextFunction) {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).json({ message: 'Token required' });
  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET!) as any;
    if (payload.role !== 'ADMIN') {
      return res.status(403).json({ message: 'Admin access required' });
    }
    (req as any).userId = payload.sub;
    next();
  } catch {
    return res.status(401).json({ message: 'Invalid token' });
  }
}
```

Клієнтська частина являє собою односторінковий додаток, написаний на React із використанням TypeScript. Збірку проекту виконує інструмент Vite, конфігурація якого знаходиться у файлі vite.config.ts. Вихідний код фронтенду згруповано в директоріях за функціональним призначенням: pages/ містить компоненти-сторінки (HomePage, ProductPage, CartPage, AdminPage, LoginPage тощо), components/ — перевикористовувані UI-компоненти (ProductCard, Header, Footer, модальні вікна), styles/ — файли CSS, api/ — базову константу API_BASE для звернення до серверного API. Маршрутизація між сторінками здійснюється за допомогою бібліотеки React Router, що дозволяє користувачеві переміщуватися між розділами магазину без перезавантаження сторінки.

Взаємодія клієнтської та серверної частин відбувається виключно через REST API. Усі HTTP-запити виконуються за допомогою стандартного fetch API. Для авторизованих запитів до заголовка додається Bearer-токен, отриманий під час логіну та збережений у localStorage браузера. Такий підхід забезпечує безстанову (stateless) архітектуру, що спрощує горизонтальне масштабування серверної частини.

Окремо варто відзначити реалізовані в коді механізми оптимізації продуктивності. На серверному боці у контролері товарів застосовано метод include Prisma для жадібного завантаження пов'язаних сутностей (зображення, специфікації, категорія, бренд), що усуває проблему N+1 запитів. Умови фільтрації передаються безпосередньо в об'єкт where, завдяки чому логіка відбору виконується на рівні СУБД, а не в коді застосунку. Для пагінації каталогу товарів реалізовано окремий ендпоінт /products/paginated, який приймає параметри cursor (ідентифікатор останнього отриманого запису) та take (кількість елементів на сторінку), що забезпечує стабільну швидкість вибірки незалежно від глибини прокручування.

На клієнтському боці в компоненті HomePage впроваджено механізм infinite scroll із використанням Intersection Observer. Під час досягнення користувачем кінця видимого списку автоматично надсилається запит за наступною порцією даних із поточним курсором. Це суттєво зменшує навантаження на базу даних, оскільки виключає необхідність завантаження всього каталогу одразу. Крім того, для зручності адміністрування реалізовано автоматичне генерування символьних кодів (slug) для категорій, брендів і товарів на основі їхніх назв, а також збереження поточного стану адміністративної панелі в sessionStorage, що дозволяє відновлювати вибрану вкладку після перезавантаження сторінки [31].

Контроль версій здійснюється за допомогою системи Git, а весь код розміщено в приватному репозиторії на GitHub. Це забезпечує відстеження змін, можливість повернення до попередніх версій та зручну інтеграцію з хмарною платформою Render для автоматичного деплою.

Таким чином, розроблена кодова база характеризується чіткою модульною структурою, суворим дотриманням типізації завдяки TypeScript, централізованим управлінням схемою даних через Prisma та реалізацією низки оптимізаційних прийомів, що в сукупності створюють надійний, продуктивний і легкий у супроводженні програмний продукт.

4.2. Фізичне проєктування бази даних та реалізовані алгоритми

Концептуальна модель, представлена ER-діаграмою (рисунок 3.4), на етапі фізичного проєктування трансформується в конкретну структуру бази даних, готову до розгортання в обраній СУБД. У проєкті «TechnoBud» цей процес реалізовано через декларативний опис сховища даних у файлі prisma/schema.prisma, який є єдиним джерелом істини про структуру бази даних. Prisma автоматично генерує необхідні SQL-міграції для створення таблиць, зв'язків, індексів та обмежень у цільовій СУБД (MariaDB або TiDB), що дозволяє уникнути ручного написання DDL-скриптів і зменшити вплив людського фактора [13, 30].

Вибір типів даних

Коректний вибір типів даних для кожного атрибуту сутності є критичним чинником, який впливає на точність обчислень, обсяг дискового простору та швидкодію запитів. У таблиці Product для грошових полів (price, oldPrice) застосовано тип Decimal(10,2), що гарантує абсолютну точність арифметичних операцій із копійками. Використання Float для фінансових розрахунків було відкинуто через ризик похибок округлення. Поле weightKg має тип Decimal(8,2), оскільки вага побутової техніки рідко перевищує сотні кілограмів. Цілочисельні лічильники (stock, reservedStock, reviewCount, warrantyMonths, powerW) та ідентифікатори (id, зовнішні ключі) використовують тип Int, який є природним для величин, що не можуть бути дробовими [8].

Текстові поля великої довжини (description, shortDescription, shippingAddress, comment) визначені як Text, що усуває обмеження на розмір

даних. Практика показала, що стандартний для MySQL/MariaDB тип Varchar(191) виявився недостатнім для зберігання деяких описів товарів та URL-адрес логотипів брендів, тому на виробничому сервері TiDB Cloud відповідні стовпці було модифіковано командою ALTER TABLE ... MODIFY ... TEXT. Для коротких текстових полів (назви, email, slug, phone) використовується Varchar із достатнім запасом довжини. Перелічувані значення (роль користувача, статус замовлення, тип руху запасів) реалізовано через enum-типи, що забезпечує цілісність даних на рівні схеми та надає автодоповнення в IDE завдяки генерації TypeScript-типів [13].

Ключі та обмеження цілісності

Для кожної таблиці визначено сурогатний первинний ключ, поле id з типом Int та атрибутом автоінкременту. Такий підхід гарантує стабільність ідентифікатора протягом усього життєвого циклу запису та забезпечує високу швидкість порівняння при операціях JOIN, порівняно з природними рядковими ключами.

Зовнішні ключі реалізують зв'язки, визначені на етапі концептуального моделювання. Для композиційних сутностей (ProductImage, ProductSpecification, CartItem, WishlistItem, OrderItem, Review) налаштовано каскадне видалення (onDelete: Cascade), завдяки чому при видаленні батьківського запису (товару, кошика, замовлення) автоматично видаляються всі залежні записи без додаткових запитів із прикладного коду. Для категорій, що містять товари, застосовано обмеження onDelete: Restrict, яке запобігає випадковому видаленню непустої категорії. Унікальні обмеження встановлено для полів, що використовуються для ідентифікації на рівні бізнес-логіки: User.email, User.phone, Product.sku, Product.slug, Category.slug, Brand.slug, Order.orderNumber. Складені унікальні обмеження на пари (cartId, productId) та (wishlistId, productId) запобігають дублюванню позицій у кошику та списку бажань [21].

Стратегія індексації

Індексація є основним механізмом підвищення продуктивності запитів у реляційних СУБД. У проєкті індекси створено для всіх зовнішніх ключів, оскільки вони беруть участь у запитах на з'єднання таблиць, а також для полів, за якими

найчастіше виконується фільтрація та сортування. Зведену інформацію про основні індекси наведено в таблиці 4.1.

Таблиця 4.1

Основні індекси бази даних

Таблиця	**Індексоване поле**	**Причина індексації**
Product	categoryId	JOIN із Category, фільтрація за категорією
Product	brandId	JOIN із Brand, фільтрація за брендом
Product	price	Фільтрація за ціновим діапазоном
Product	isActive	Відокремлення активних товарів від деактивованих
Product	createdAt	Сортування за датою додавання
Product	slug	Пошук товару для сторінки з детальною інформацією
Category	slug	Пошук категорії за символьним кодом
Category	parentId	Побудова дерева підкатегорій
Brand	slug	Пошук бренду за символьним кодом
Order	userId	Фільтрація замовлень користувача
Order	status	Фільтрація за статусом в адмін-панелі
Order	createdAt	Сортування за датою замовлення
Review	productId	Завантаження відгуків для конкретного товару
Review	userId	Перевірка, чи залишав користувач відгук
CartItem	cartId, productId	Складений унікальний індекс
WishlistItem	wishlistId, productId	Складений унікальний індекс

Така стратегія індексації дозволяє уникнути повного сканування таблиць для переважної більшості запитів, що надходять від клієнтської частини. Водночас свідомо не створювалися індекси для рідко використовуваних у фільтрах полів (color, material, energyClass), щоб не сповільнювати операції вставки та оновлення даних.

Реалізовані алгоритми

Окрім статичної структури таблиць та індексів, важливе значення мають алгоритми, закладені в серверну логіку для забезпечення цілісності та продуктивності.

Нормалізація та денормалізація. Схема бази даних приведена до третьої нормальної форми (3НФ), що мінімізує надлишковість і запобігає аномаліям оновлення. Усі неключові атрибути залежать виключно від первинного ключа своєї таблиці, а повторювані групи даних (зображення, специфікації) винесені в

окремі таблиці. Контрольованим винятком є свідома денормалізація таблиці OrderItem, де поля productNameSnapshot, brandNameSnapshot та skuSnapshot зберігають копію назви товару, бренду та артикулу на момент оформлення замовлення. Це гарантує історичну достовірність даних у разі подальшого редагування або видалення товару з каталогу [27, 28, 29].

Транзакційне оформлення замовлення. Процес створення замовлення є критичним щодо цілісності даних і реалізований як атомарна операція. Контролер createOrder послідовно виконує: завантаження кошика з товарами, перевірку достатності залишків на складі, створення запису замовлення з позиціями та снапшотами, зменшення доступного залишку (stock) і збільшення резерву (reservedStock) для кожного товару, очищення позицій кошика та зміну його статусу на CONVERTED. Усі ці дії виконуються в межах однієї транзакції Prisma, що забезпечує відкат усіх змін у разі помилки на будь-якому з етапів, код нижче наочно показує атомарність операції.

```
const order = await prisma.$transaction(async (tx) => {
  // Створення замовлення з позиціями
  const newOrder = await tx.order.create({
    data: {
      orderNumber,
      customerName, customerPhone, customerEmail,
      shippingAddress, subtotal, totalAmount, notes, userId,
      status: 'PENDING',
      items: {
        create: cart.items.map((item) => ({
          quantity: item.quantity,
          unitPrice: item.product.price,
          totalPrice: Number(item.product.price) * item.quantity,
          productId: item.productId,
          productNameSnapshot: item.product.name,
          brandNameSnapshot: item.product.brand?.name || null,
          skuSnapshot: item.product.sku,
        })),
      },
    },
    include: { items: true },
  });

  // Оновлення залишків
  for (const item of cart.items) {
    await tx.product.update({
      where: { id: item.productId },
```

```
          data: {
            stock: { decrement: item.quantity },
            reservedStock: { increment: item.quantity },
          },
        });
    }

    // Очищення кошика
    await tx.cartItem.deleteMany({ where: { cartId: cart.id } });
    await tx.cart.update({ where: { id: cart.id }, data: { status: 'CONVERTED' } });

    return newOrder;
});
```

Курсорна пагінація. Для оптимізації завантаження каталогу товарів реалізовано ендпоінт /products/paginated, який використовує курсорну пагінацію замість класичної зі зсувом (OFFSET). Клієнт передає параметр cursor ідентифікатор останнього отриманого товару, та take, бажану кількість елементів. Серверний код формує запит із умовою cursor: { id: cursor }, skip: 1, що дозволяє уникнути сканування попередніх сторінок. У поєднанні з механізмом Infinite Scroll на клієнті це суттєво знижує навантаження на базу даних і забезпечує стабільну швидкість вибірки незалежно від глибини прокручування [4, 5].

Таким чином, фізичне проєктування бази даних та реалізовані на його основі алгоритми створюють надійний фундамент для стабільної роботи веб-застосунку. У наступному підрозділі буде розглянуто заходи з оптимізації продуктивності та результати модульного тестування, які підтверджують ефективність прийнятих проєктних рішень.

4.3. Оптимізація продуктивності

Забезпечення високої продуктивності є однією з ключових нефункціональних вимог до системи, сформульованих у підрозділі 2.2. У проєкті «TechnoBud» оптимізація продуктивності реалізована на кількох рівнях: рівні бази даних (індексація, нормалізація, вибір типів даних), рівні ORM (ефективні запити, уникнення проблеми N+1) та рівні клієнт-серверної взаємодії (курсорна пагінація, Infinite Scroll). У цьому підрозділі розглянуто конкретні впроваджені

оптимізаційні заходи та проаналізовано їхню ефективність за допомогою метрик хмарної СУБД TiDB Cloud.

Оптимізація на рівні ORM та бази даних

Одним із найпоширеніших джерел падіння продуктивності в застосунках із використанням ORM є проблема N+1 запитів, коли для кожного отриманого запису основної таблиці виконується окремий запит за зв'язаними сутностями. У проєкті цю проблему усунено завдяки використанню методу include в Prisma, який формує єдиний агрегований SQL-запит із JOIN-ами для завантаження всіх пов'язаних даних (зображень, специфікацій, категорії, бренду) за одне звернення до бази даних [13]. Фрагмент коду контролера getProductsPaginated демонструє застосований підхід:

```
const products = await prisma.product.findMany({
    where,
    include: {
        images: true,
        specifications: true,
        category: true,
        brand: true,
    },
    orderBy: { id: 'asc' },
    ...(cursor && { cursor: { id: cursor }, skip: 1 }),
    take,
});
```

Умови фільтрації передаються безпосередньо в об'єкт where, завдяки чому вся логіка відбору виконується на рівні СУБД із використанням наявних індексів, а не в коді застосунку. Перевірка isActive: true для товару, категорії та бренду гарантує, що деактивовані сутності ніколи не потраплять у відповідь. Сортування (orderBy) також делеговано СУБД, що швидше за сортування в коді сервера чи клієнта [6].

Для обмеження кількості одночасних з'єднань із базою даних налаштовано пул з'єднань (connectionLimit: 10) в адаптері @prisma/adapter-mariadb. Це запобігає перевитраті ресурсів хмарної СУБД, особливо в умовах безкоштовного тарифу TiDB Cloud Serverless, який має суворі ліміти на кількість одночасних конектів [24]. Фрагмент нижче показує, як створюється підключення до СУБД із

використанням пулу з'єднань та SSL-шифрування, що є ключовим для безпечної роботи з хмарною TiDB:

```
// фрагмент ініціалізації адаптера
const adapter = new PrismaMariaDb({
   host: dbUrl.hostname,
   port: Number(dbUrl.port || 3306),
   user: dbUrl.username,
   password: dbUrl.password,
   database: dbUrl.pathname.slice(1),
   ssl: { ca: fs.readFileSync(path.join(__dirname, 'prisma', 'ca.pem')) },
   connectionLimit: 10,
});
const prisma = new PrismaClient({ adapter });
```

Оптимізація на рівні клієнт-серверної взаємодії

Класична посторінкова пагінація з параметрами LIMIT та OFFSET створює надлишкове навантаження на СУБД під час обробки великих таблиць. При виконанні запиту з OFFSET СУБД змушена сканувати всі попередні рядки, навіть якщо вони не потрапляють у фінальну вибірку. Для усунення цього недоліку в проєкті реалізовано курсорну пагінацію через ендпоінт /products/paginated [4, 5].

Алгоритм працює так:

- Клієнт надсилає запит із параметрами cursor (ідентифікатор останнього показаного товару) та take (бажана кількість елементів).
- Сервер формує SQL-запит із умовою WHERE id > cursor, пропускає перший запис (skip: 1) та обмежує вибірку значенням take.
- У відповідь повертається масив товарів; ідентифікатор останнього з них стає новим курсором для наступного запиту.

На клієнтському боці цей алгоритм поєднується з механізмом Infinite Scroll, реалізованим через Intersection Observer у компоненті HomePage. Коли користувач досягає кінця видимого списку, спрацьовує спостерігач, який автоматично надсилає запит за наступною порцією даних. На стороні клієнта механізм нескінченного прокручування реалізовано за допомогою Intersection Observer, що автоматично підвантажує наступну порцію товарів при досягненні кінця списку:

Аналіз метрик TiDB Cloud

Для оцінки ефективності впроваджених оптимізацій було проведено порівняльний аналіз ключових метрик хмарної СУБД TiDB Cloud до та після впровадження Infinite Scroll у поєднанні з курсорною пагінацією. Вимірювання виконувалися за однакових умов при послідовному перегляді каталогу товарів. Графіки, отримані з панелі моніторингу TiDB Cloud, представлено на рисунках 4.1-4.3 (до впровадження зверху, знизу після впровадження).

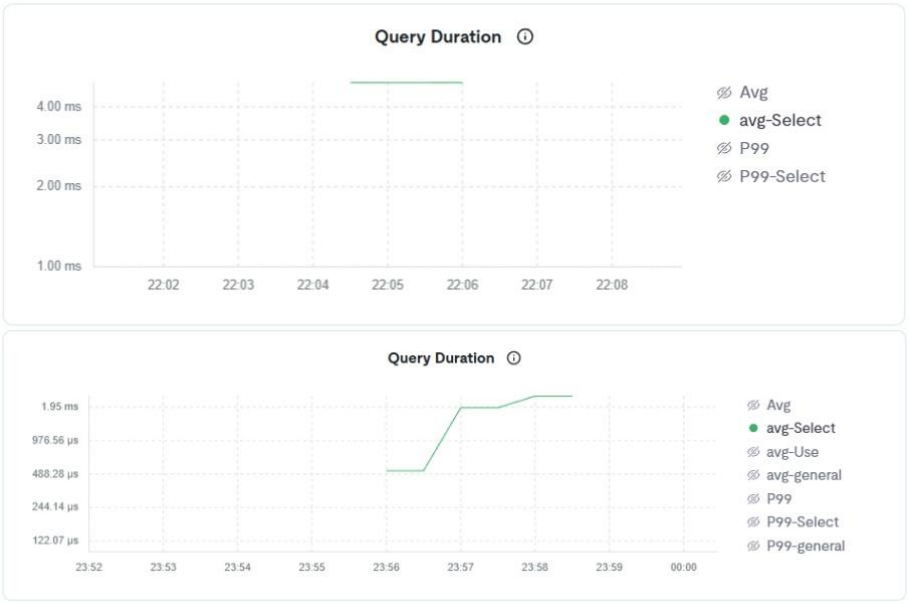

Рисунок 4.1 — Query Duration (Avg-Select) до та після впровадження

Рисунок 4.2 — Idle Connection Duration (P99) до та після впровадження

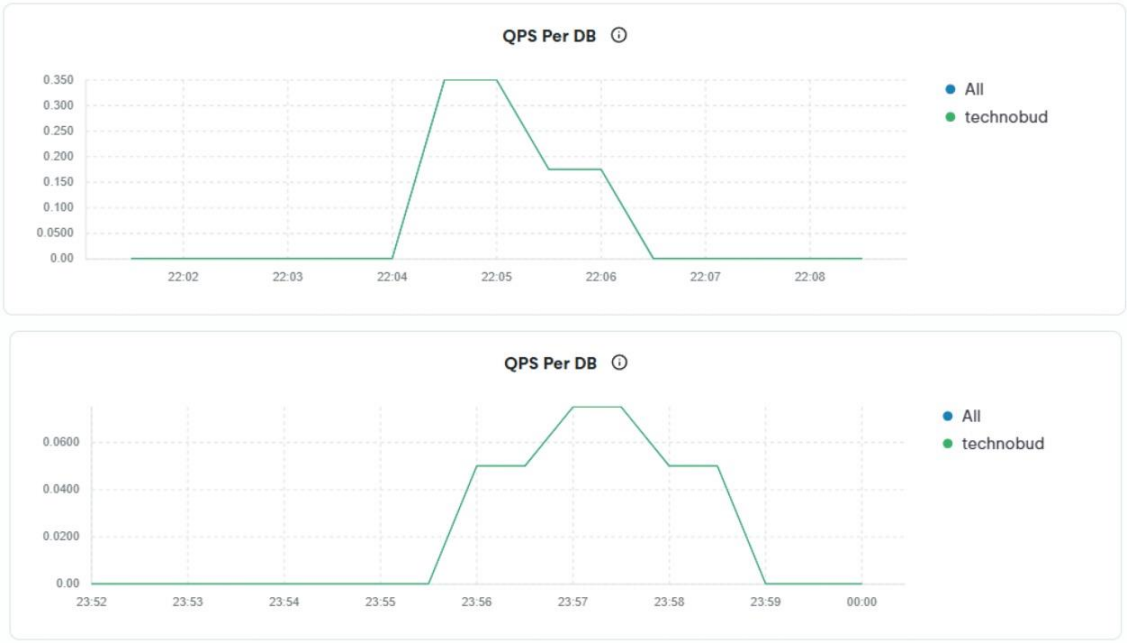

Рисунок 4.3 — QPS Per DB до та після впровадження

Наведені графіки наочно демонструють якісну зміну характеру навантаження: після впровадження Infinite Scroll та курсорної пагінації спостерігається кратне зниження пікових значень QPS, що свідчить про стабілізацію навантаження на базу даних. Кількісне зіставлення середніх показників до та після оптимізації наведено в таблиці 4.2.

Таблиця 4.2

Порівняння метрик TiDB Cloud до та після оптимізації

Метрика	**До впровадження**	**Після впровадження**
Query Duration (avg Select)	4,95 мс	2,45 мс
Idle Connection Duration (P99)	358,97 мс	166,60 мс
QPS Per DB	0,367	0,075

Аналіз отриманих метрик свідчить про суттєве зниження навантаження на базу даних. Частота запитів до бази даних (QPS Per DB) зменшилася майже в 5 разів, з 0,367 до 0,075. Це означає, що сервер СУБД виконує значно менше SQL-запитів, обробляючи лише ті дані, які дійсно потрібні клієнту. Середній час виконання SELECT-запитів скоротився з 4,95 мс до 2,45 мс, що пояснюється зменшенням обсягу одночасно оброблюваних даних та ефективнішим

використанням індексів при курсорній вибірці. Показник Idle Connection Duration (P99) знизився з 358,97 мс до 166,60 мс, що позитивно впливає на стабільність роботи пулу з'єднань.

4.4. Висновки до розділу

У четвертому розділі було детально описано практичну реалізацію веб-застосунку «TechnoBud» на основі спроєктованої архітектури та концептуальної моделі бази даних. Програмний код організовано у вигляді монорепозиторію з чітким розділенням на серверну (Express, TypeScript, Prisma) та клієнтську (React, Vite, TypeScript) частини, що забезпечує модульність, зручність супроводження та незалежне розгортання компонентів. Серверна логіка реалізована за багатошаровим принципом із розподілом на маршрутизатори, контролери та middleware, а клієнтський інтерфейс використовує механізм маршрутизації React Router для побудови SPA без перезавантаження сторінок.

У рамках фізичного проєктування бази даних визначено оптимальні типи даних (Decimal для грошових сум, Text для довгих описів, enum для статусів), реалізовано систему первинних і зовнішніх ключів із каскадним видаленням для композиційних зв'язків, а також створено понад 30 індексів для прискорення пошукових запитів (таблиця 4.1). Схему приведено до третьої нормальної форми з єдиним контрольованим винятком, денормалізацією позицій замовлення через snapshot-поля, що гарантує історичну достовірність даних про покупки. Ключові алгоритми, зокрема транзакційне оформлення замовлення та курсорна пагінація, реалізовані з використанням можливостей Prisma Client для забезпечення цілісності даних і стабільної продуктивності.

Особливу увагу приділено оптимізації продуктивності. Завдяки методу include в Prisma усунено проблему N+1 запитів, а передача умов фільтрації через об'єкт where дозволила виконувати всю логіку відбору на рівні СУБД. Впровадження курсорної пагінації у поєднанні з Infinite Scroll на клієнтському боці дало змогу знизити пікову частоту запитів до бази даних майже в 5 разів,

скоротити середній час виконання SELECT-запитів удвічі та зменшити показник Idle Connection Duration (P99) більш ніж на 50% (таблиця 4.2, рисунки 4.1-4.3). Отримані метрики TiDB Cloud підтверджують ефективність обраних методів оптимізації в умовах обмежених ресурсів безкоштовного хмарного середовища.

З точки зору безпеки реалізовано всі заплановані механізми: паролі зберігаються у вигляді bcrypt-хешів, автентифікація виконується через stateless JWT-токени, а з'єднання з хмарною базою даних захищене SSL/TLS. Система успішно розгорнута на платформі Render та взаємодіє з TiDB Cloud, демонструючи стабільну роботу навіть в умовах обмежених ресурсів безкоштовного тарифу. Комплексне тестування системи, якому присвячено наступний розділ, підтвердить відсутність критичних помилок.

Таким чином, отримані результати свідчать про успішну практичну реалізацію всіх проєктних рішень, закладених на етапі проєктування. Система демонструє стабільну роботу, високу продуктивність та готовність до розширення функціоналу. Наступний розділ роботи присвячено всебічному тестуванню реалізованого веб-застосунку та його розгортанню в хмарному середовищі, що є завершальним етапом життєвого циклу розробки.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

5.1. Стратегії та методи тестування системи

Тестування веб-застосунку «TechnoBud» проводилося на кількох рівнях із метою перевірки відповідності системи функціональним та нефункціональним вимогам, сформульованим у розділі 2, та оцінки ефективності оптимізаційних рішень. Загальна стратегія поєднувала ручне тестування, інструментальну перевірку окремих API-запитів, автоматизоване модульне тестування та аналіз метрик продуктивності. Основним засобом верифікації серверної частини став фреймворк Jest у поєднанні з бібліотекою Supertest, тоді як Postman застосовувався вибірково для візуалізації ключових сценаріїв.

У проєкті застосовано шість основних видів тестування:

1. **Функціональне тестування** – перевірка відповідності реалізованого функціоналу вимогам підрозділу 2.2. Перевірялися: робота ієрархічного каталогу з багатокритеріальною фільтрацією (ціна, бренд, категорія, рейтинг, колір, матеріал, енергоклас), функціонування кошика (додавання, зміна кількості, видалення позицій), додавання товарів до списку бажань, оформлення замовлення з автоматичним оновленням складських залишків, модерація відгуків із деревоподібною структурою відповідей, CRUD-операції в адміністративній панелі (товари, категорії, бренди, користувачі) та автоматичне генерування slug.

2. **Тестування REST API** – виконано у два етапи. На першому етапі за допомогою Postman перевірено успішну реєстрацію нового користувача (POST /auth/register, статус 201) та спробу доступу до захищеного маршруту без токена (GET /cart, помилка 401). На другому етапі повноту перевірки всіх критичних ендпоінтів забезпечив автоматизований набір тестів на основі Jest та Supertest.

3. **Автоматизоване модульне тестування** – реалізовано з використанням фреймворку Jest та бібліотеки Supertest, яка дозволяє надсилати HTTP-запити до Express-сервера без його окремого запуску. Для компіляції TypeScript-коду застосовано пресет ts-jest, що дало змогу писати тести рідною

мовою проєкту без додаткової трансформації. Конфігурацію Jest винесено в окремий файл jest.config.ts у корені проєкту, де вказано середовище виконання node, кореневу директорію з тестами та пресет ts-jest. Для коректної роботи глобальних функцій Jest (describe, it, expect тощо) у конфігурацію TypeScript додано відповідні типи (@types/jest) [33, 34].

Тестовий набір, розташований у файлі tests/api.test.ts, налічує 10 сценаріїв, які охоплюють основні групи ендпоінтів: публічні (GET /products, GET /products/paginated), автентифікаційні (POST /auth/register, POST /auth/login), захищені користувацькі (GET /cart, GET /auth/me) та адміністративні (POST /products, DELETE /products/:id). У тестах перевіряються не лише статус-коди відповідей, але й структура JSON (наявність обов'язкових полів, відповідність значень фільтрам). Для забезпечення валідності JWT-токенів у тестовому середовищі використовується той самий секретний ключ, що й у серверному коді. Тестовий користувач-покупець створюється через API реєстрації безпосередньо перед запуском тестів у хуці beforeAll, а адміністратор — через прямий запит до Prisma з наступною генерацією токена бібліотекою jsonwebtoken. Після завершення всіх тестів хук afterAll видаляє створені записи з бази даних, що запобігає засміченню тестового оточення, код автоматичного тестування наведений в Додатку Б.

4. Тестування інтерфейсу користувача – проводилося шляхом безпосередньої взаємодії з розгорнутим у хмарі веб-застосунком. Імітувалися типові сценарії поведінки покупця (перегляд каталогу з нескінченним прокручуванням, пошук товарів, додавання до кошика, оформлення замовлення) та адміністратора (керування контентом через адміністративну панель, зміна статусів замовлень, модерація відгуків). Окремо перевірялася валідація повідомлень про помилки (спроба замовити більше товару, ніж є в наявності) та коректність роботи Infinite Scroll при швидкому прокручуванні каталогу.

5. Тестування продуктивності – оцінювалося на основі метрик хмарної СУБД TiDB Cloud (Query Duration, Idle Connection Duration, QPS Per DB). Порівняння показників до та після впровадження курсорної пагінації (таблиця 4.2,

рисунки 4.1-4.3) підтвердило суттєве зниження навантаження на базу даних: частота запитів зменшилася майже в 5 разів, середній час виконання SELECT-запитів скоротився вдвічі.

6. Тестування безпеки – підтверджено неможливість доступу до захищених маршрутів без JWT-токену (очікувана помилка 401) та виконання адміністративних дій із роллю CUSTOMER (очікувана помилка 403). Перевірено, що паролі користувачів зберігаються в базі даних виключно у вигляді bcrypt-хешів, а з'єднання між сервером застосунку та TiDB Cloud здійснюється через SSL/TLS із використанням CA-сертифіката.

Завдяки поєднанню ручного та автоматизованого тестування вдалося не лише підтвердити відповідність системи встановленим вимогам, але й створити надійний механізм швидкої перевірки працездатності після внесення змін до кодової бази. Виявлені на ранніх етапах недоліки (переповнення VARCHAR-полів, необхідність коригування SQL-режиму для TiDB) були успішно усунені. Детальний опис отриманих результатів за кожним із зазначених напрямків із відповідними ілюстраціями наведено в наступному підрозділі.

5.2. Результати тестування

Застосування описаних у підрозділі 5.1 методів тестування дозволило всебічно перевірити працездатність веб-застосунку «TechnoBud» та оцінити його відповідність функціональним і нефункціональним вимогам. Нижче наведено результати за кожним із шести напрямків тестування.

Функціональне тестування підтвердило, що всі основні підсистеми інтернет-магазину працюють коректно. Ієрархічний каталог із багаторівневими категоріями відображається відповідно до структури бази даних; фільтрація за ціною, брендом, категорією, рейтингом, кольором, матеріалом та енергокласом виконується без помилок. Пошук товарів за назвою повертає релевантні результати. Кошик і список бажань коректно реагують на додавання, зміну кількості та видалення позицій. Оформлення замовлення супроводжується

автоматичним оновленням складських залишків (зменшення stock, збільшення reservedStock), а при спробі замовити більше товару, ніж є в наявності, система виводить інформативне повідомлення про помилку без створення замовлення. В адміністративній панелі успішно виконуються CRUD-операції над товарами, категоріями, брендами та користувачами; автоматичне генерування slug на основі назви працює для всіх типів сутностей.

Тестування REST API за допомогою Postman дало змогу візуально підтвердити коректність роботи ключових ендпоінтів. На рисунку 5.1 наведено результат успішної реєстрації нового користувача: запит POST /auth/register із валідними даними (fullName, email, password) повертає статус 201 Created та містить JWT-токен у тілі відповіді.

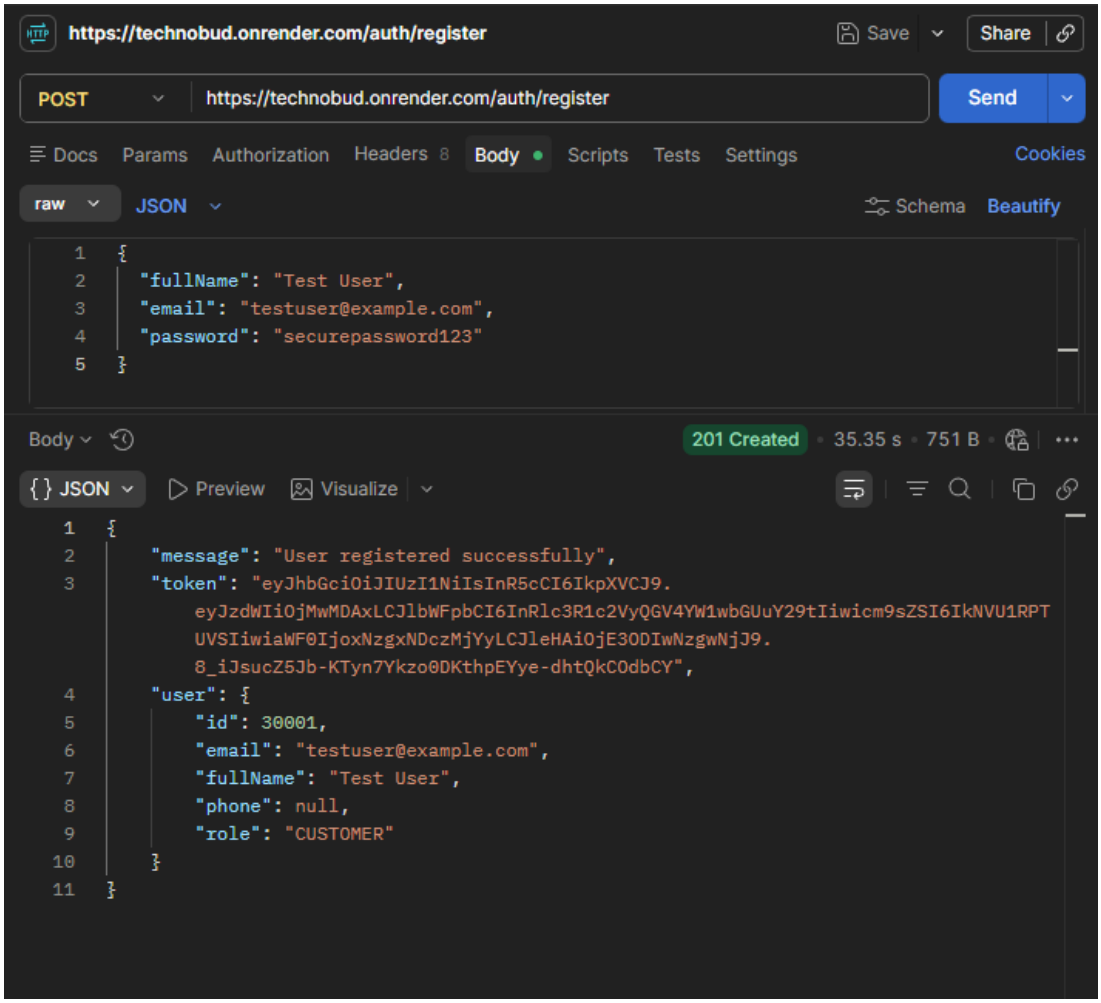

Рисунок 5.1 — Скріншот Postman з успішною реєстрацією (201) та токеном

Спроба доступу до захищеного маршруту GET /cart без передачі токена закономірно завершується помилкою 401 Unauthorized (рисунок 5.2), що підтверджує коректну роботу middleware автентифікації.

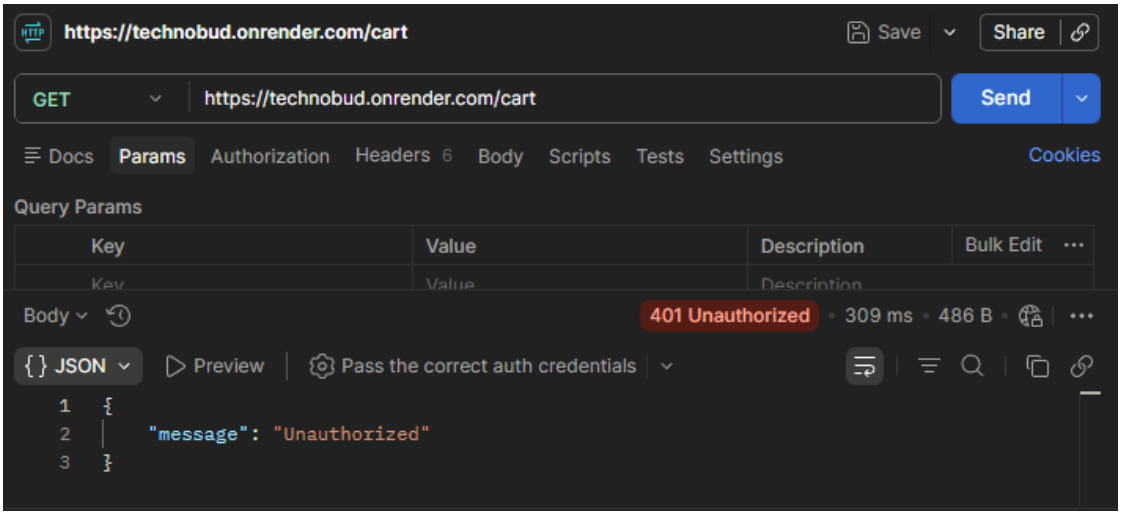

Рисунок 5.2 — Скріншот Postman з помилкою 401 при доступі без токена

Автоматизоване модульне тестування виконано за допомогою фреймворку Jest із бібліотекою Supertest. Тестовий набір із 10 сценаріїв охопив публічні, автентифікаційні, користувацькі та адміністративні ендпоінти. Усі тести пройшли успішно (рисунок 5.3), що підтверджує коректність налаштування middleware, правильність обробки фільтрів у контролерах, валідацію вхідних даних та розмежування прав доступу. Зелений список результатів у терміналі демонструє стабільну роботу серверної частини без жодної помилки.

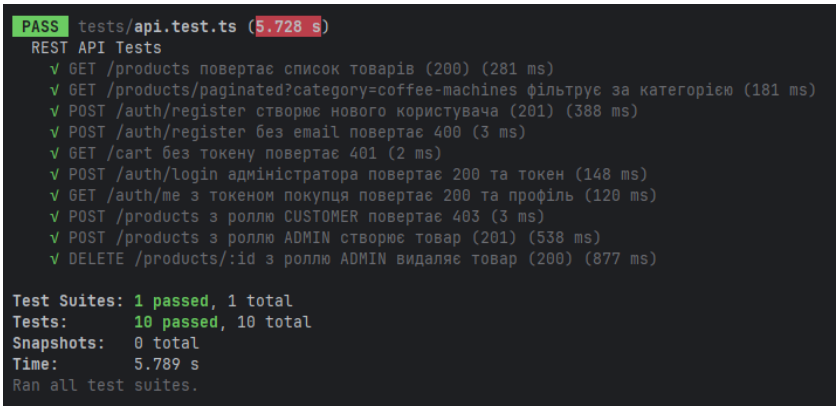

Рисунок 5.3 — Скріншот терміналу з результатами Jest

Тестування інтерфейсу користувача проводилося шляхом безпосередньої взаємодії з розгорнутим у хмарі сайтом. Перевірялися всі основні сторінки, доступні як звичайним покупцям, так і адміністраторам.

Головна сторінка (рисунок 5.4) відображає каталог товарів із фільтрами за категорією, брендом, ціновим діапазоном, рейтингом, кольором, матеріалом та енергокласом, а також підтримує нескінченне прокручування через механізм Infinite Scroll. Сторінка кошика (рисунок 5.5) показує список обраних товарів із можливістю зміни кількості, видалення позицій та відображає підсумкову суму, а при спробі перевищити доступний залишок виводить відповідне попередження.

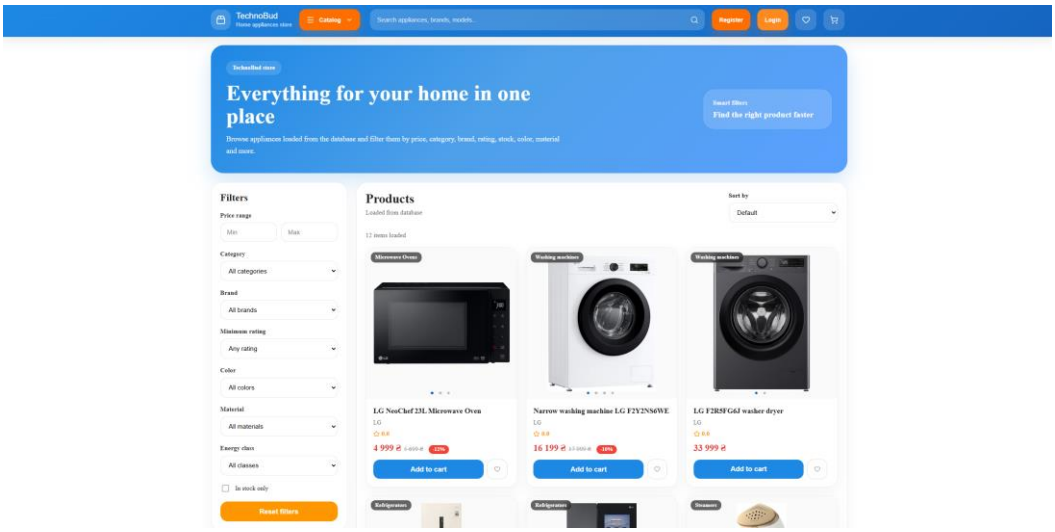

Рисунок 5.4 — Скріншот головної сторінки магазину з товарами

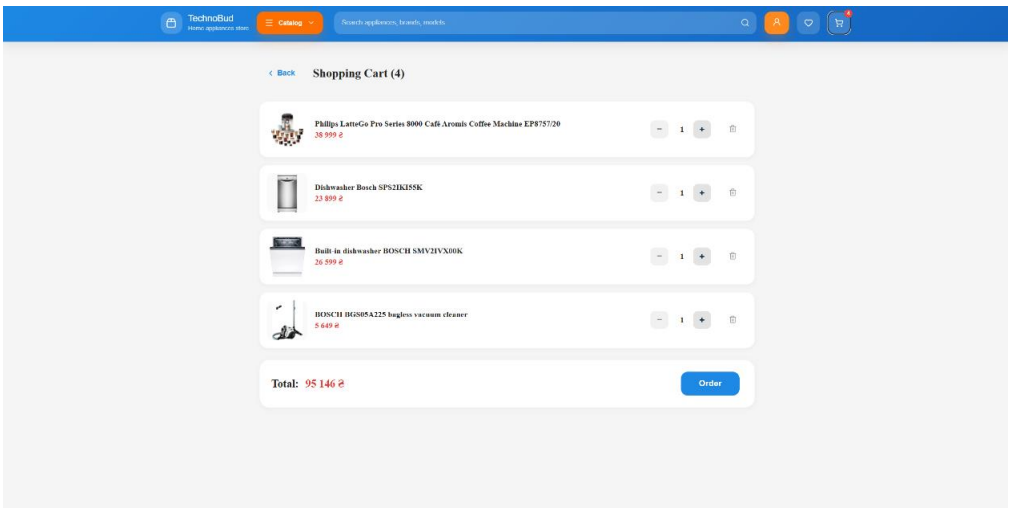

Рисунок 5.5 — Скріншот сторінки кошика з товарами

Форми автентифікації — сторінка логіну (рисунок 5.6) та сторінка реєстрації (рисунок 5.7) — містять поля для введення email, паролю та додаткових даних, забезпечуючи валідацію вхідних даних на клієнтському боці.

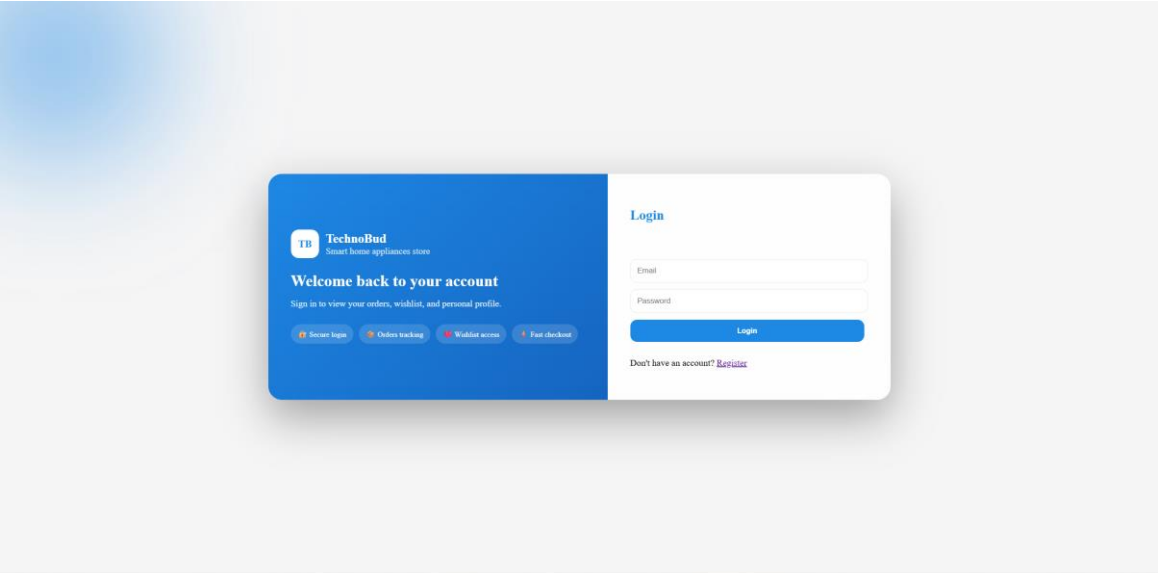

Рисунок 5.6 — Скріншот сторінки логіну

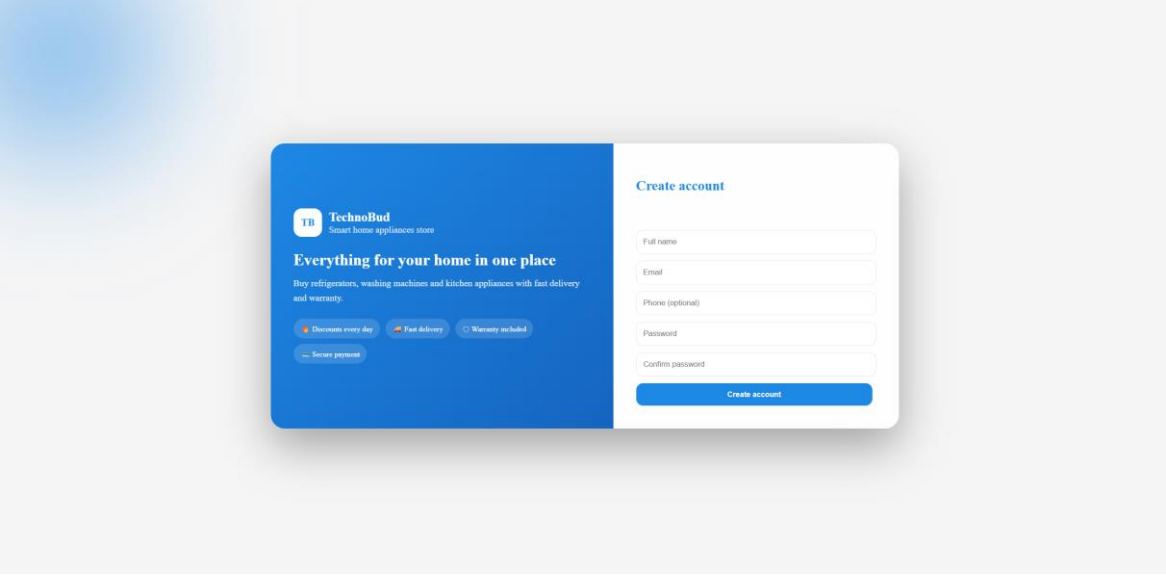

Рисунок 5.7 — Скріншот сторінки реєстрації

Сторінка бренду (рисунок 5.8) містить інформацію про виробника (назву, логотип, опис, посилання на офіційний сайт) та динамічно формує список товарів, що належать до цього бренду.

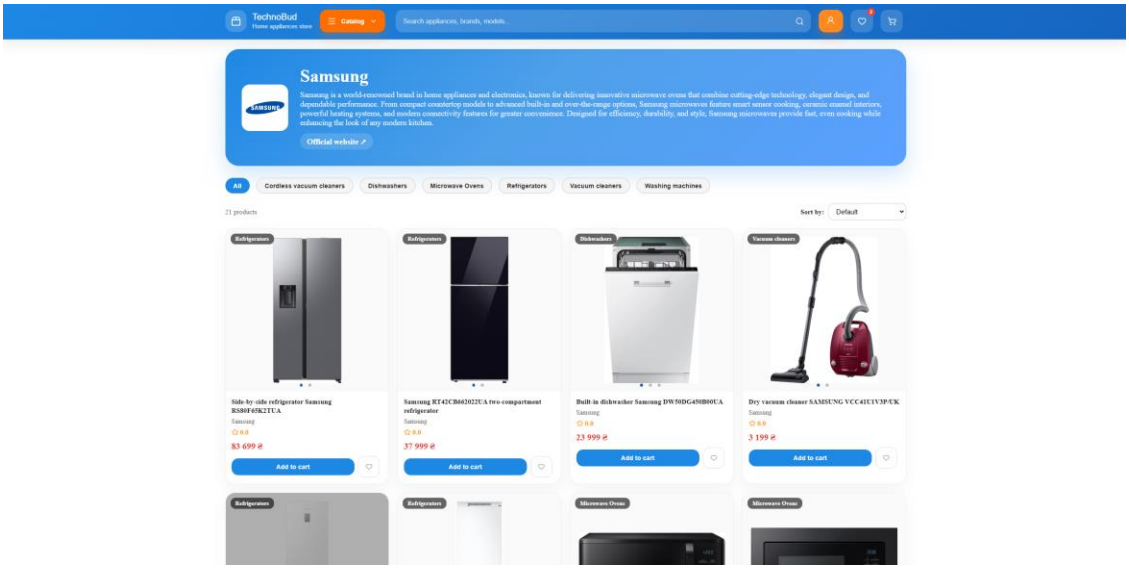

Рисунок 5.8 — Скріншот сторінки бренду з інформацією та товарами

Сторінка списку бажань (рисунок 5.9) дозволяє користувачеві переглядати відкладені товари, видаляти окремі позиції та одним натисканням додавати всі доступні товари до кошика.

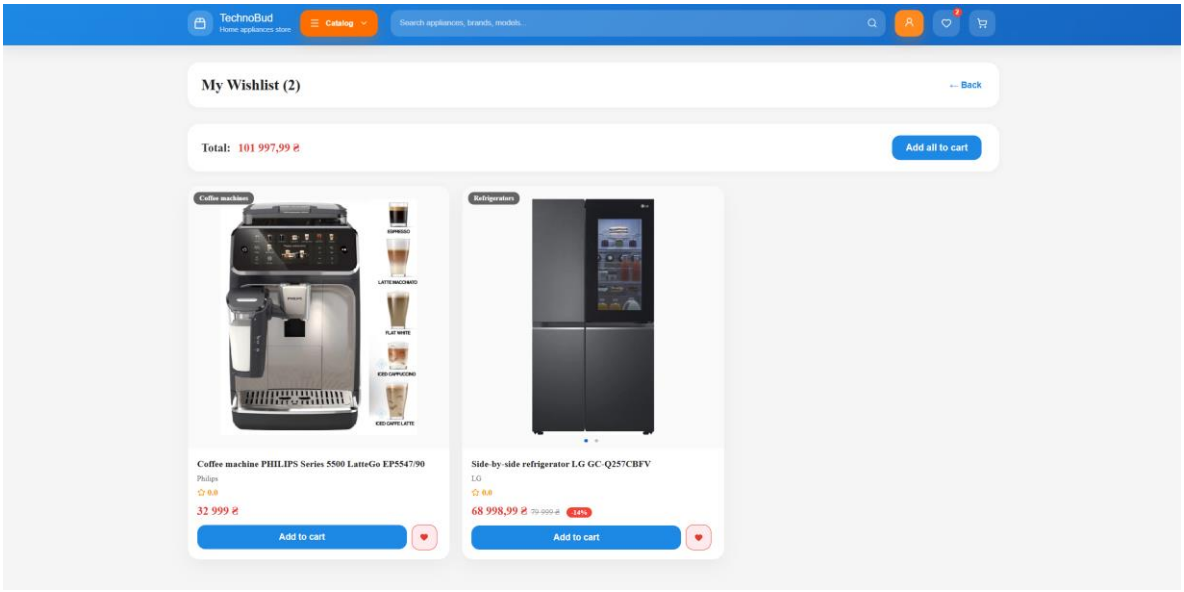

Рисунок 5.9 — Скріншот сторінки списку бажань

Сторінка товару містить вкладку з відгуками (рисунок 5.10), де користувачі можуть залишати оцінку (1–5 зірок) та текстовий коментар, переглядати відгуки інших покупців і відповіді адміністратора, реалізовані у вигляді деревоподібної структури.

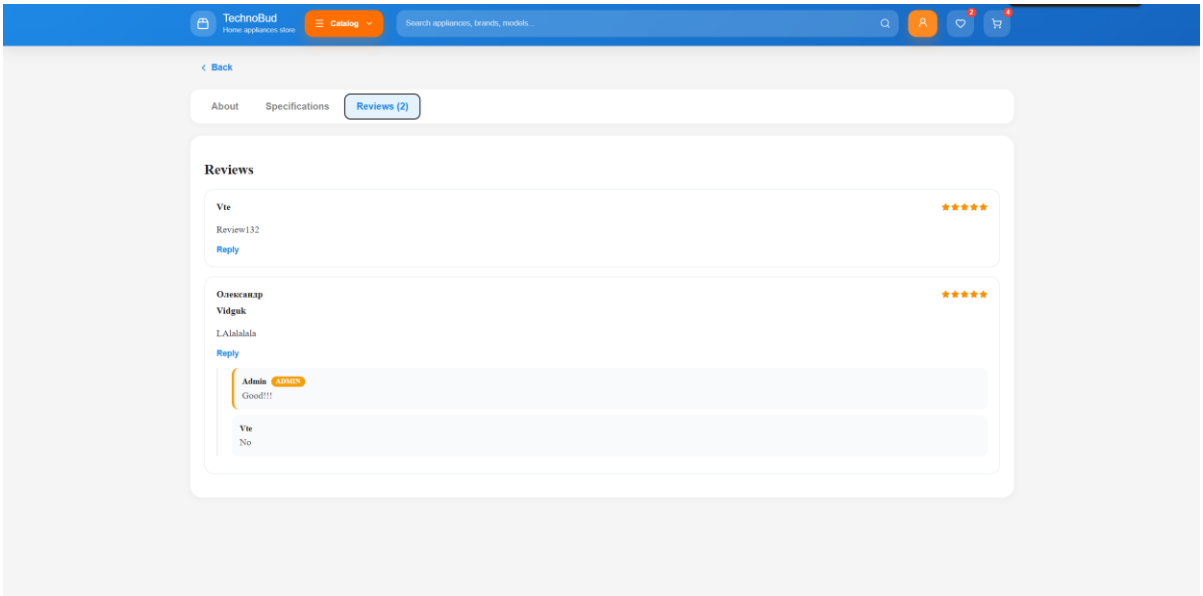

Рисунок 5.10 — Скріншот сторінки відгуків до товару

Сторінка замовлення (рисунок 5.11) відображає повну інформацію про оформлене замовлення: номер, статус, контактні дані покупця, адресу доставки, перелік позицій із назвами, кількістю та цінами, а також підсумкову суму, що дозволяє відстежувати історію покупок.

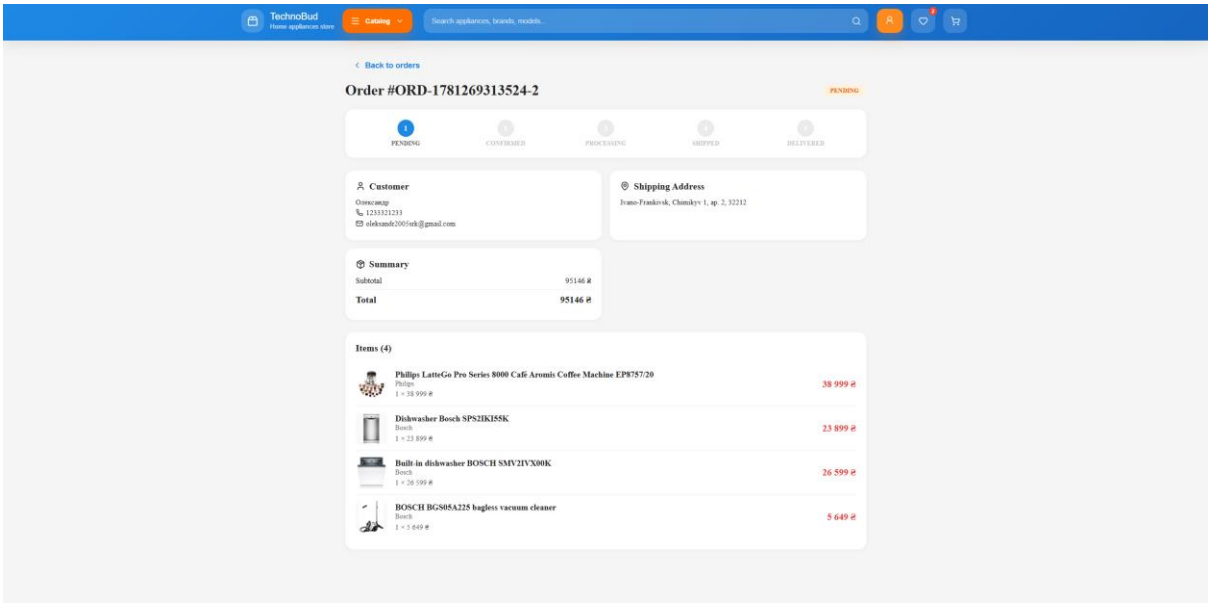

Рисунок 5.11 — Скріншот сторінки замовлення з даними

Адміністративна панель (рисунок 5.12) надає зручний доступ до таблиць із фільтрацією для всіх основних сутностей (товари, категорії, бренди, користувачі,

замовлення); модальні вікна дозволяють створювати та редагувати записи а також змінювати статуси замовлень.

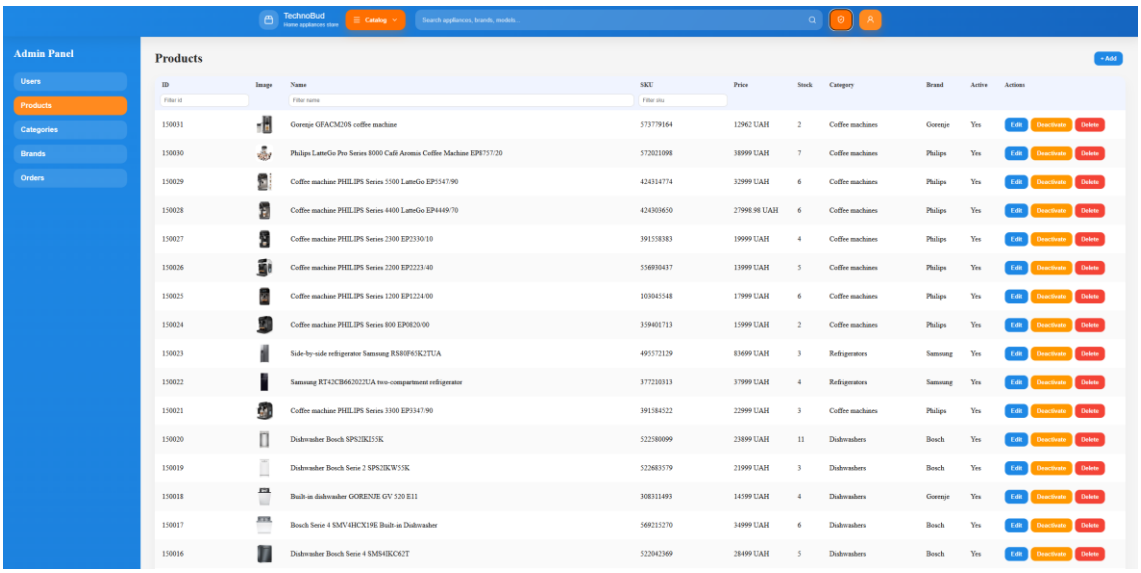

Рисунок 5.12 — Скріншот сторінки адміністратора (список товарів)

Тестування продуктивності, детально описане в підрозділі 4.3, підтвердило суттєве зниження навантаження на базу даних після впровадження курсорної пагінації у поєднанні з Infinite Scroll. Частота запитів зменшилася майже в 5 разів (з 0,367 до 0,075 QPS Per DB), середній час виконання SELECT-запитів скоротився вдвічі (з 4,95 мс до 2,45 мс), а показник Idle Connection Duration (P99) знизився більш ніж на 50% (з 358,97 мс до 166,60 мс). Ці результати узагальнено в таблиці 4.2 та проілюстровано на рисунках 4.1-4.3.

Тестування безпеки засвідчило, що система надійно захищена від несанкціонованого доступу. Усі спроби звернення до захищених маршрутів без JWT-токену отримували статус 401 Unauthorized, а спроби виконання адміністративних дій із роллю CUSTOMER, статус 403 Forbidden. Паролі користувачів зберігаються в базі даних виключно у вигляді bcrypt-хешів, а з'єднання з хмарною СУБД TiDB Cloud виконується через SSL/TLS із використанням CA-сертифіката.

У підсумку, результати тестування за всіма шістьма напрямками підтвердили відповідність розробленого веб-застосунку функціональним і

нефункціональним вимогам, сформульованим у розділі 2. Система демонструє коректну роботу автентифікації та авторизації, надійний захист від несанкціонованого доступу, цілісність транзакцій під час оформлення замовлень, стабільну продуктивність при роботі з каталогом товарів і зручний користувацький інтерфейс. Жодних критичних помилок, пов'язаних із підключенням до бази даних або несумісністю версій бібліотек, не виявлено. Отримані результати дозволяють перейти до етапу розгортання системи в хмарному середовищі.

5.3. Розгортання у хмарному середовищі

Останнім етапом життєвого циклу розробки веб-застосунку «TechnoBud» стало його розгортання в хмарній інфраструктурі, що забезпечує цілодобову доступність системи для кінцевих користувачів без необхідності адміністрування власного серверного обладнання. Завдяки обраній архітектурі з чітким розділенням на фронтенд і бекенд, кожна частина застосунку розгорталася незалежно, що спростило процес деплою та налаштування змінних оточення.

Вибір хмарних платформ

Для розміщення серверної частини (Express) та клієнтської частини (React SPA) було обрано хмарну платформу Render. Вона надає безкоштовний тариф для Web Service (сервер) і Static Site (фронтенд), підтримує автоматичний деплой із GitHub-репозиторію, що значно спрощує оновлення коду при кожному коміті у відповідну гілку. Як виробничу СУБД використано TiDB Cloud, розподілену MySQL-сумісну базу даних із безкоштовним рівнем Starter, який забезпечує горизонтальне масштабування та високу доступність без потреби в ручному налаштуванні шардування чи реплікації.

Розгортання серверної частини

Серверна частина розгорталася як Render Web Service. Для коректного запуску Node.js-застосунку в кореневому package.json було додано скрипти: "build": "tsc" для компіляції TypeScript у JavaScript та "start": "node dist/server.js"

для запуску скомпільованого сервера. У налаштуваннях Web Service на Render вказано:

- Build Command: npm install && npm run build
- Start Command: npm start
- Root Directory: корінь репозиторію
- Змінні оточення: DATABASE_URL (рядок підключення до TiDB Cloud), JWT_SECRET (секретний ключ для підпису JWT-токенів)

Під час перших спроб деплою виникла проблема з виконанням prisma generate: через обмеження безпеки на стороні Render бінарний файл Prisma CLI не мав прав на виконання. Для її усунення було прийнято рішення згенерувати Prisma Client локально (командою npx prisma generate) та закомітити папку generated/prisma до репозиторію, прибравши виклик prisma generate із процесу збірки на Render. Крім того, через використання Render версії TypeScript 6+ виникла помилка moduleResolution=node10 is deprecated. Замість переходу на Node16, який потребував би змін у коді, до tsconfig.json було додано параметр "ignoreDeprecations": "6.0", що дозволило зберегти звичний формат CommonJS. Після цих налаштувань серверна частина успішно стартувала, про що свідчить статус «Live» на дашборді Render (рисунок 5.7).

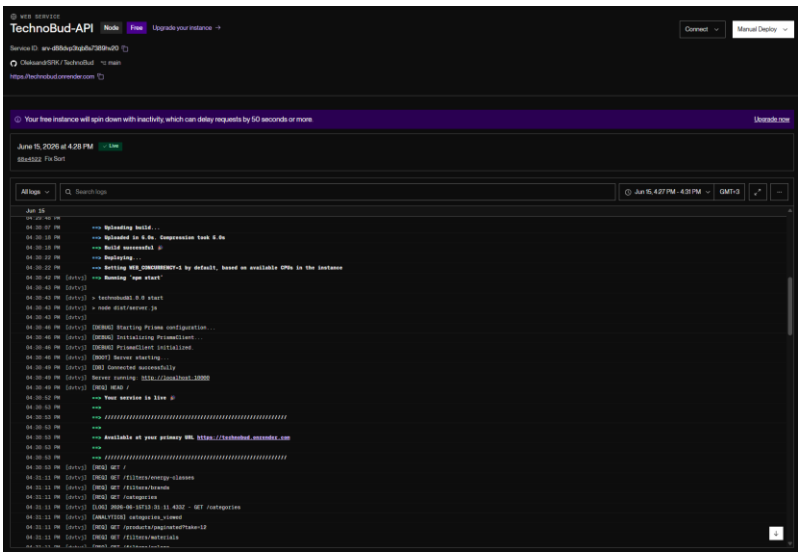

Рисунок 5.13 — Скріншот дашборду Render із успішним деплоєм серверної частини (Web Service)

Перед першим запуском у виробничому середовищі було виконано команду npx prisma migrate deploy, яка створила всі необхідні таблиці та індекси в TiDB Cloud відповідно до актуальної схеми. Початкове наповнення бази даних тестовими категоріями, брендами та товарами здійснювалося через Prisma Studio, підключену до хмарної СУБД.

Розгортання клієнтської частини

Фронтенд розгортався як Render Static Site, оптимізований для обслуговування статичних файлів (HTML, CSS, JavaScript). У налаштуваннях було зазначено:

- Root Directory: frontend
- Build Command: npm install && node ./node_modules/vite/bin/vite.js build
- Publish Directory: dist
- Змінна оточення: VITE_API_URL, що вказує на URL серверної частини (https://technobud.onrender.com)

Початково скрипт build у frontend/package.json містив прямий виклик vite build, який на локальному середовищі виконувався бездоганно. Однак після завантаження коду на Render і запуску автоматичної збірки з'ясувалося, що бінарний файл Vite, встановлений через npm install, не отримує прав на виконання через політику безпеки платформи. Система логувала помилку Permission denied, через що процес складання зупинявся на стадії виклику vite, не доходячи до генерації статичних файлів. Спроби надати права через додаткові команди в Build Command не дали результату, оскільки Render не дозволяє змінювати атрибути виконання для встановлених пакетів. Вирішенням стало редагування скрипту build у frontend/package.json — замість прямого виклику vite build було використано node ./node_modules/vite/bin/vite.js build, що дозволило обійти обмеження прав безпеки, запускаючи Vite через Node.js, який гарантовано має права на виконання скриптів. Успішний деплой клієнтської частини підтверджується статусом «Live» на дашборді Render (рисунок 5.8).

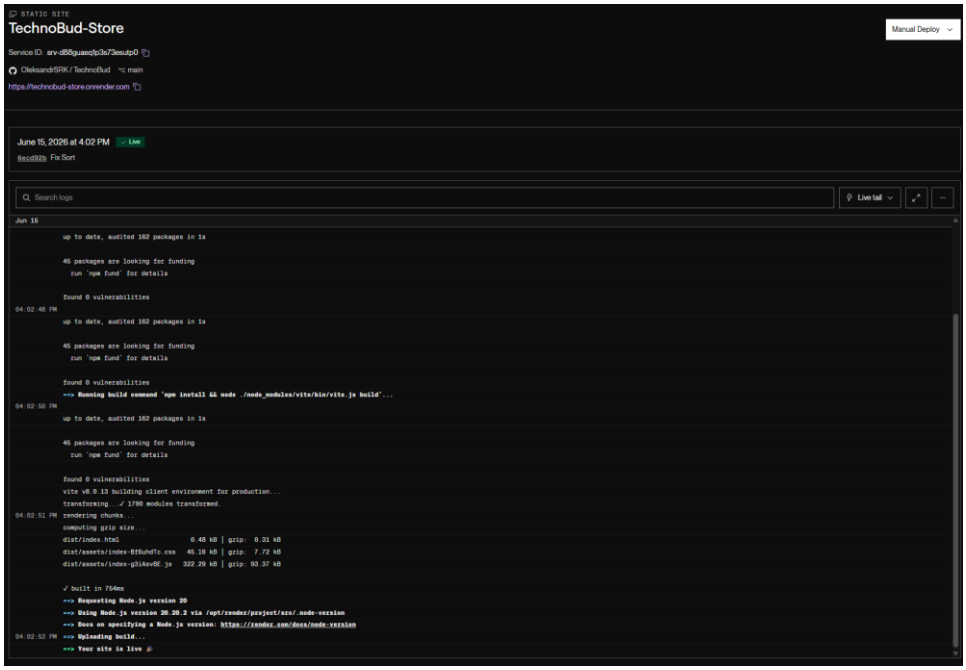

Рисунок 5.14 — Скріншот дашборду Render із успішним деплоєм клієнтської частини (Static Site)

Схема розгортання

Загальну схему хмарного розгортання веб-застосунку «TechnoBud» представлено на рисунку 5.9. Код проєкту зберігається в GitHub-репозиторії. Після кожного коміту Render автоматично запускає процес збірки та деплою для обох сервісів. Клієнтська частина (Static Site) завантажується в браузер користувача, звідки надсилає HTTP-запити до Web Service (Express). Сервер, у свою чергу, через Prisma Client та адаптер @prisma/adapter-mariadb з'єднується з хмарною TiDB, використовуючи SSL-шифрування та пул з'єднань (connectionLimit: 10). Для забезпечення захищеного з'єднання в репозиторії зберігається CA-сертифікат платформи (prisma/ca.pem), який завантажується під час ініціалізації адаптера PrismaMariaDb.

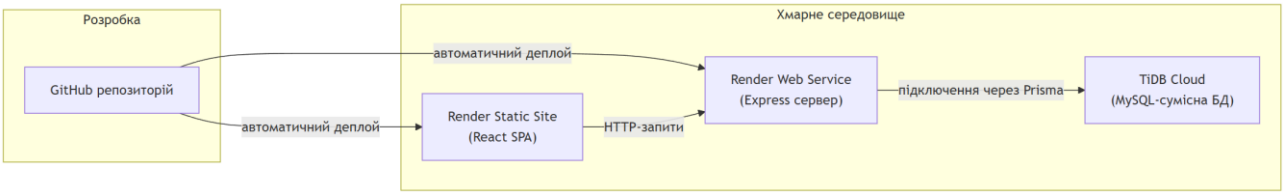

Рисунок 5.15 — Схема хмарного розгортання веб-застосунку

Особливості експлуатації

Після перенесення бази даних із локальної MariaDB до TiDB Cloud виявилися дві особливості, пов'язані з відмінностями в налаштуваннях. По-перше, TiDB за замовчуванням встановлює SQL-режим ONLY_FULL_GROUP_BY, через що Prisma Studio не могла завантажити метадані схеми. Проблему було вирішено виконанням команди SET GLOBAL sql_mode = '...' через SQL-редактор TiDB Cloud. По-друге, під час додавання товарів через адміністративну панель виникали помилки переповнення стовпців типу VARCHAR(191) для полів shortDescription і logoUrl, оскільки окремі значення перевищували це обмеження. Для виправлення довелося змінити типи цих полів на TEXT за допомогою SQL-команди ALTER TABLE ... MODIFY ... TEXT, після чого оновити Prisma-схему та перегенерувати клієнт.

Розгорнутий веб-застосунок доступний за адресою https://technobud.onrender.com і демонструє стабільну роботу в умовах безкоштовного хмарного середовища. Завдяки використанню адаптера @prisma/adapter-mariadb система зберігає повну сумісність із локальною MariaDB, що дозволяє в будь-який момент перенести базу даних назад у локальне оточення або на інший хмарний сервіс без жодних змін у прикладному коді.

5.4. Висновки по розділу

У п'ятому розділі було проведено всебічне тестування веб-застосунку «TechnoBud» та успішно виконано його розгортання в хмарному середовищі. Тестування охопило шість ключових напрямків: функціональне, REST API, автоматизоване модульне, інтерфейсу користувача, продуктивності та безпеки. Це дозволило перевірити відповідність системи всім вимогам, сформульованим на етапі проєктування. Поєднання ручних перевірок (Postman, безпосередня взаємодія з інтерфейсом) із автоматизованим набором із 10 тестів на базі Jest та Supertest забезпечило високу достовірність отриманих результатів.

Автоматизоване модульне тестування підтвердило коректність роботи всіх критичних ендпоінтів: публічних, автентифікаційних, користувацьких та адміністративних. Успішне проходження тестів засвідчило правильність налаштування middleware автентифікації та авторизації, обробки фільтрів у контролерах, валідації вхідних даних і розмежування прав доступу. Ручне тестування через Postman дозволило візуально зафіксувати ключові сценарії, успішну реєстрацію користувача та захист від несанкціонованого доступу.

Тестування інтерфейсу підтвердило стабільну роботу Infinite Scroll, кошика, оформлення замовлень і адміністративної панелі. Перевірка продуктивності на основі метрик TiDB Cloud засвідчила зниження частоти запитів майже в 5 разів та скорочення часу виконання SELECT-запитів удвічі після впровадження курсорної пагінації.

Фінальним етапом стало розгортання системи на хмарних платформах: серверна частина, як Render Web Service, клієнтська, як Render Static Site, а база даних у TiDB Cloud. У процесі деплою було вирішено низку практичних проблем, пов'язаних із правами виконання Prisma CLI та Vite, сумісністю SQL-режимів і переповненням VARCHAR-полів, що дало цінний досвід експлуатації хмарних сервісів.

Отримані результати свідчать, що веб-застосунок повністю готовий до дослідної експлуатації, а закладена архітектурна гнучкість дозволяє масштабувати систему в майбутньому без перегляду ядра схеми. Практичне значення виконаної роботи полягає в тому, що створений веб-застосунок не лише демонструє працездатність запропонованих архітектурних рішень, але й може слугувати готовим шаблоном для швидкого розгортання подібних систем електронної комерції. Завдяки автоматизації деплою через GitHub-інтеграцію з Render, будь-яке оновлення коду миттєво потрапляє у виробниче середовище, що відповідає принципам безперервної доставки (Continuous Delivery). Таким чином, успішне завершення етапів тестування та розгортання логічно підводить до формулювання загальних висновків за результатами всієї дипломної роботи.

ВИСНОВКИ

У результаті виконання дипломної роботи було спроєктовано, оптимізовано та реалізовано реляційну базу даних для високонавантаженого веб-застосунку інтернет-магазину побутової техніки «TechnoBud», а також створено на її основі повнофункціональний клієнт-серверний додаток із розгортанням у хмарному середовищі. Проведене дослідження охопило повний цикл розробки, від аналізу предметної області та збору вимог до практичного впровадження й тестування готової системи.

На етапі аналізу було визначено ключові вимоги до бази даних: цілісність, швидкодія, масштабованість, гнучкість, безпека та транзакційність. Проведено порівняльний аналіз сучасних СУБД (MySQL, MariaDB, PostgreSQL, TiDB), ORM-систем (Prisma, TypeORM, Sequelize, Drizzle) та архітектурних підходів (моноліт, мікросервіси, SPA + REST API). На основі цього аналізу обґрунтовано вибір технологічного стеку: TypeScript, React, Express, Prisma, MariaDB (для локальної розробки) та TiDB Cloud (для виробничого середовища), а також JWT-автентифікацію та хмарну платформу Render.

У рамках проєктування бази даних побудовано ER-модель із 15 сутностей, що охоплюють ієрархічний каталог, товари з гнучкими характеристиками, користувачів із розмежуванням ролей, кошик, список бажань, замовлення зі снапшотами для історичної точності та відгуки з деревоподібною структурою. Схему приведено до третьої нормальної форми з контрольованою денормалізацією позицій замовлення. Визначено оптимальні типи даних (Decimal для грошових сум, Text для довгих описів, enum для статусів), створено понад 30 індексів для прискорення пошукових запитів і налаштовано каскадні обмеження цілісності.

Практична реалізація включала розробку серверної частини на Express.js із використанням Prisma Client, клієнтської частини на React із застосуванням Vite, а також адміністративної панелі для керування контентом. Особливу увагу приділено оптимізації продуктивності: метод include в Prisma усунув проблему

N+1 запитів, а впровадження курсорної пагінації у поєднанні з Infinite Scroll на клієнті дозволило знизити пікову частоту запитів до бази даних майже в 5 разів (з 0,367 до 0,075 QPS), скоротити середній час виконання SELECT-запитів удвічі та зменшити показник Idle Connection Duration (P99) більш ніж на 50%.

Тестування системи проводилося на шести рівнях: функціональне, REST API, автоматизоване модульне (Jest + Supertest, 10 тестів), інтерфейсу користувача, продуктивності та безпеки. Усі тести пройшли успішно, підтвердивши відповідність системи вимогам. Розгортання веб-застосунку виконано на платформі Render із підключенням до TiDB Cloud, що забезпечило цілодобову доступність сервісу. Практичний досвід вирішення проблем сумісності (SQL-режими TiDB, переповнення VARCHAR-полів) підтвердив правильність вибору адаптера @prisma/adapter-mariadb для безшовної міграції між локальною та хмарною СУБД.

Отримані результати свідчать, що розроблена система є готовим до використання прототипом інтернет-магазину, який демонструє стабільну роботу та високу продуктивність навіть в умовах обмежених ресурсів безкоштовного хмарного тарифу. Подальше вдосконалення системи може включати: інтеграцію з платіжними шлюзами (наприклад, Stripe, LiqPay) для автоматизації фінансових транзакцій; підключення до логістичних API («Нова Пошта», «Укрпошта») для автоматичного розрахунку вартості доставки; впровадження персоналізованих рекомендацій на основі історії переглядів та покупок; розробку мобільного додатку з використанням React Native; міграцію окремих модулів на мікросервісну архітектуру за умови зростання навантаження; а також додавання системи масових email- та SMS-розсилок для нотифікації клієнтів про статуси замовлень та акційні пропозиції. Запропонована архітектура та обраний стек технологій забезпечують достатню гнучкість для реалізації цих розширень без перегляду ядра системи.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Що таке RDBMS (relational database management system / реляційна система управління базами даних)? [Електронний ресурс]. — Режим доступу: https://tseivo.com/b/memecode/t/orml9pedyg/shcho-take-rdbms-relational-database-management-system-reliatsiina-systema-upravlinnia-bazamy-dany[kh

2. Understanding NewSQL: The Next Generation of Database Technology [Електронний ресурс]. — Режим доступу: https://www.linkedin.com/pulse/understanding-newsql-next-generation-database-technology-shiv-iyer-kkekc/

3. Інструменти ORM для об'єктно-реляційного відображення [Електронний ресурс]. — Режим доступу: https://www.hostragons.com/uk/%D0%B1%D0%BB%D0%BE%D0%B3/%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B8-orm-%D0%B4%D0%BB%D1%8F-%D1%80%D0%B5%D0%BB%D1%8F%D1%86%D1%96%D0%B9%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B2%D1%96%D0%B4%D0%BE%D0%B1%D1%80%D0%B0/

4. Cursor-based Pagination [Електронний ресурс]. — Режим доступу: https://medium.com/@nimmikrishnab/cursor-based-pagination-37f5fae9f482

5. Paginating Real-Time Data with Cursor Based Pagination [Електронний ресурс]. — Режим доступу: https://www.sitepoint.com/paginating-real-time-data-cursor-based-pagination /

6. Продуктивність індексу бази даних MySQL [Електронний ресурс]. — Режим доступу: https://www.hostragons.com/uk/%D0%B1%D0%BB%D0%BE%D0%B3/%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D1%96%D1%81%D1%82%D1%8C-%D1%96%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D1%83-

%D0%B1%D0%B0%D0%B7%D0%B8-
%D0%B4%D0%B0%D0%BD%D0%B8%D1%85-mysql/

7. Повний огляд REST: нюанси, поради, приклади [Електронний ресурс]. — Режим доступу: https://dou.ua/forums/topic/50364/

8 Типи баз даних: особливості, відмінності та приклади [Електронний ресурс]. — Режим доступу: https://dou.ua/lenta/articles/types-of-databases/

9. TiDB Documentation [Електронний ресурс]. — Режим доступу: https://docs.pingcap.com/

10. MySQL Documentation [Електронний ресурс]. — Режим доступу: https://dev.mysql.com/doc/

11. MariaDB Documentation [Електронний ресурс]. — Режим доступу: https://mariadb.com/docs

12. PostgreSQL: Documentation [Електронний ресурс]. — Режим доступу: https://www.postgresql.org/docs/

13. Prisma Documentation [Електронний ресурс]. — Режим доступу: https://www.prisma.io/docs

14. TypeORM Docs [Електронний ресурс]. — Режим доступу: https://typeorm.io/docs/getting-started

15. Sequelize Documentation [Електронний ресурс]. — Режим доступу: https://sequelize.org/docs/v6/getting-started/

16. Drizzle ORM Documentation [Електронний ресурс]. — Режим доступу: https://orm.drizzle.team/docs/overview

17. Monolithic Architecture in Software Development: A testament to the power of unified creation. [Електронний ресурс]. — Режим доступу: https://www.linkedin.com/pulse/monolithic-architecture-software-development-power--awave/

18. Що таке мікросервісна архітектура: шлях до гнучкого та масштабованого середовища розробки [Електронний ресурс]. — Режим доступу: https://blog.colobridge.net/uk/2024/01/what-is-microservices-architecture-ua/

19. Мікросервісна архітектура [Електронний ресурс]. — Режим доступу: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d

20. Що таке Single Page Application та як працює SPA сайт [Електронний ресурс]. — Режим доступу: http://wezom.com.ua/ua/blog/chto-takoe-spa-prilozheniya

21. Веб-фреймворк Express (Node.js/JavaScript) [Електронний ресурс]. — Режим доступу: https://developer.mozilla.org/ru/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs

22. Why You Should Use Vite Instead of Create React App for Your React Projects [Електронний ресурс]. — Режим доступу: https://www.linkedin.com/pulse/why-you-should-use-vite-instead-create-react-app-your-ramachandra-edu5c/

23. Сувора типізація: Typescript, Flow, Javascript — бути чи не бути? [Електронний ресурс]. — Режим доступу: https://blog.ithillel.ua/articles/suvora-typizatsiia-typescript-flow-javascript

24. @prisma/adapter-mariadb [Електронний ресурс]. — Режим доступу: https://www.npmjs.com/package/@prisma/adapter-mariadb

25. Understanding JSON Web Tokens (JWT) and How They Work [Електронний ресурс]. — Режим доступу: https://www.linkedin.com/pulse/understanding-json-web-tokens-jwt-how-work-dhawal-patel-7rrqf/

26. Render Documentation [Електронний ресурс]. — Режим доступу: https://render.com/docs

27. Нормалізація баз даних [Електронний ресурс]. — Режим доступу: https://greenhouse.cv.ua/?p=697

28. Нормалізація схем баз даних [Електронний ресурс]. — Режим доступу: https://web.posibnyky.vntu.edu.ua/fitki/10savchuk_organizaciya_bazdanih_znan/gl_43.html

29. Нормалізація даних: третя нормальна форма (3НФ) є важливою для вашої бази даних [Електронний ресурс]. — Режим доступу: https://uk.itpedia.nl/2025/05/02/datanormalisatie-de-derde-normaalvorm-3nf-is-essentieel-voor-je-database/

30. Фізичне проектування структури бази даних [Електронний ресурс]. — Режим доступу: https://rdb.dp.ua/uk/chapter_04

31. react-infinite-scroll-component [Електронний ресурс]. — Режим доступу: https://www.npmjs.com/package/react-infinite-scroll-component

32. Express middleware [Електронний ресурс]. — Режим доступу: https://expressjs.com/en/resources/middleware/

33. Jest Documentation [Електронний ресурс]. — Режим доступу: https://jestjs.io/uk/docs/next/getting-started

34. Automated Testing with Jest and Supertest in Node.js (Express + TypeScript) [Електронний ресурс]. — Режим доступу: https://medium.com/@etosin70/automated-testing-with-jest-and-supertest-in-node-js-express-typescript-953683d3f5fb

35. React Router Official Documentation [Електронний ресурс]. — Режим доступу: https://reactrouter.com/home

ДОДАТКИ

ДОДАТОК А

```
Table user {
  id int [pk, increment]
  email varchar [unique]
  passwordHash varchar
  fullName varchar
  phone varchar [unique]
  role user_role
  isActive boolean
  createdAt datetime
  updatedAt datetime
}

Table address {
  id int [pk, increment]
  label varchar
  country varchar
  city varchar
  street varchar
  house varchar
  apartment varchar
  postalCode varchar
  isDefault boolean
  userId int
  createdAt datetime
  updatedAt datetime
}

Table brand {
  id int [pk, increment]
  name varchar
  slug varchar [unique]
  logoUrl varchar
  description text
  isActive boolean
  websiteUrl varchar
  createdAt datetime
  updatedAt datetime
}

Table category {
  id int [pk, increment]
  name varchar
  slug varchar [unique]
  description text
  imageUrl varchar
  isActive boolean
```

```
  parentId int
  createdAt datetime
  updatedAt datetime
}

Table product {
  id int [pk, increment]
  name varchar
  sku varchar [unique]
  slug varchar [unique]
  price decimal
  oldPrice decimal
  currency varchar
  description text
  shortDescription text
  stock int
  reservedStock int
  rating float
  reviewCount int
  color varchar
  material varchar
  energyClass varchar
  powerW int
  warrantyMonths int
  weightKg decimal
  isActive boolean
  isFeatured boolean
  brandId int
  categoryId int
  createdAt datetime
  updatedAt datetime
}

Table productimage {
  id int [pk, increment]
  url varchar
  alt varchar
  sortOrder int
  isMain boolean
  productId int
  createdAt datetime
}

Table productspecification {
  id int [pk, increment]
  name varchar
  value varchar
  unit varchar
  sortOrder int
  isHighlighted boolean
```

```
  productId int
}

Table inventorymovement {
  id int [pk, increment]
  type inventorymovement_type
  quantity int
  note text
  productId int
  createdAt datetime
}

Table cart {
  id int [pk, increment]
  status cart_status
  sessionId varchar
  userId int
  createdAt datetime
  updatedAt datetime
}

Table cartitem {
  id int [pk, increment]
  quantity int
  unitPrice decimal
  cartId int
  productId int
  createdAt datetime
  updatedAt datetime
}

Table wishlist {
  id int [pk, increment]
  userId int [unique]
  createdAt datetime
  updatedAt datetime
}

Table wishlistitem {
  id int [pk, increment]
  wishlistId int
  productId int
  createdAt datetime
}

Table review {
  id int [pk, increment]
  rating int
  title varchar
  comment text
```

```
  isApproved boolean
  userId int
  productId int
  parentId int
  createdAt datetime
  updatedAt datetime
}

Table order {
  id int [pk, increment]
  orderNumber varchar [unique]
  status order_status
  customerName varchar
  customerPhone varchar
  customerEmail varchar
  shippingAddress text
  subtotal decimal
  discountAmount decimal
  shippingAmount decimal
  totalAmount decimal
  notes text
  userId int
  addressId int
  createdAt datetime
  updatedAt datetime
}

Table orderitem {
  id int [pk, increment]
  quantity int
  unitPrice decimal
  totalPrice decimal
  orderId int
  productId int
  productNameSnapshot varchar
  brandNameSnapshot varchar
  skuSnapshot varchar
}

Ref: address.userId > user.id

Ref: cart.userId - user.id

Ref: wishlist.userId - user.id

Ref: order.userId > user.id

Ref: order.addressId > address.id

Ref: cartitem.cartId > cart.id
```

```
Ref: cartitem.productId > product.id

Ref: wishlistitem.wishlistId > wishlist.id
Ref: wishlistitem.productId > product.id

Ref: review.userId > user.id
Ref: review.productId > product.id
Ref: review.parentId > review.id

Ref: orderitem.orderId > order.id
Ref: orderitem.productId > product.id

Ref: product.brandId > brand.id
Ref: product.categoryId > category.id

Ref: productimage.productId > product.id

Ref: productspecification.productId > product.id

Ref: inventorymovement.productId > product.id

Ref: category.parentId > category.id
```

ДОДАТОК Б

```
process.env.JWT_SECRET = 'dev_secret';

import request from 'supertest';
import express from 'express';
import cors from 'cors';
import jwt from 'jsonwebtoken';
import productsRoutes from '../src/routes/products.routes';
import authRouter from '../src/routes/auth.routes';
import cartRoutes from '../src/routes/cart.routes';
import { prisma } from '../src/prisma';

const app = express();
app.use(cors());
app.use(express.json());
app.use('/products', productsRoutes);
app.use('/auth', authRouter);
app.use('/cart', cartRoutes);

describe('REST API Tests', () => {
  let customerToken: string;
  let adminToken: string;
  let adminEmail: string;

  beforeAll(async () => {
    const uniqueEmail = `testuser${Date.now()}@example.com`;
    const registerRes = await request(app)
      .post('/auth/register')
      .send({ fullName: 'Test Customer', email: uniqueEmail, password: 'test123' });
    customerToken = registerRes.body.token;

    adminEmail = `admin${Date.now()}@example.com`;
    const bcryptjs = require('bcryptjs');
    const passwordHash = await bcryptjs.hash('admin123', 10);
    const adminUser = await prisma.user.create({
      data: {
        email: adminEmail,
        passwordHash,
        fullName: 'Test Admin',
        role: 'ADMIN',
      },
    });

    adminToken = jwt.sign(
      {
        id: adminUser.id,
        email: adminUser.email,
        role: adminUser.role,
      },
```

```
        process.env.JWT_SECRET || 'dev_secret',
        { expiresIn: '1h' }
    );
});

afterAll(async () => {
    await prisma.user.deleteMany({
        where: {
            email: { contains: 'testuser' },
        },
    });
    await prisma.user.deleteMany({
        where: {
            email: { contains: 'admin' },
        },
    });
    await prisma.$disconnect();
});

it('GET /products повертає список товарів (200)', async () => {
    const res = await request(app).get('/products');
    expect(res.status).toBe(200);
    expect(Array.isArray(res.body)).toBeTruthy();
});

it('GET /products/paginated?category=coffee-machines фільтрує за категорією', async () => {
    const res = await request(app).get('/products/paginated?category=coffee-machines&take=5');
    expect(res.status).toBe(200);
    res.body.forEach((product: any) => {
        expect(product.category?.slug).toBe('coffee-machines');
    });
});

it('POST /auth/register створює нового користувача (201)', async () => {
    const newEmail = `newuser${Date.now()}@example.com`;
    const res = await request(app)
        .post('/auth/register')
        .send({ fullName: 'New User', email: newEmail, password: 'test123' });
    expect(res.status).toBe(201);
    expect(res.body.token).toBeDefined();
});

it('POST /auth/register без email повертає 400', async () => {
    const res = await request(app)
        .post('/auth/register')
        .send({ fullName: 'No Email', password: 'test123' });
    expect(res.status).toBe(400);
});

it('GET /cart без токену повертає 401', async () => {
    const res = await request(app).get('/cart');
    expect(res.status).toBe(401);
```

```
});

it('POST /auth/login адміністратора повертає 200 та токен', async () => {
  const res = await request(app)
    .post('/auth/login')
    .send({ email: adminEmail, password: 'admin123' });
  expect(res.status).toBe(200);
  expect(res.body.token).toBeDefined();
});

it('GET /auth/me з токеном покупця повертає 200 та профіль', async () => {
  const res = await request(app)
    .get('/auth/me')
    .set('Authorization', `Bearer ${customerToken}`);
  expect(res.status).toBe(200);
  expect(res.body.email).toBeDefined();
});

it('POST /products з роллю CUSTOMER повертає 403', async () => {
  const res = await request(app)
    .post('/products')
    .set('Authorization', `Bearer ${customerToken}`)
    .send({ name: 'Test Product', price: 100, categoryId: 1, brandId: 1 });
  expect(res.status).toBe(403);
});

it('POST /products з роллю ADMIN створює товар (201)', async () => {
  const res = await request(app)
    .post('/products')
    .set('Authorization', `Bearer ${adminToken}`)
    .send({
      name: 'Jest Product',
      price: 999,
      categoryId: 30005,
      brandId: 30001,
      sku: `JEST-${Date.now()}`,
      slug: `jest-product-${Date.now()}`,
      description: 'Created by automated test',
    });
  expect(res.status).toBe(201);
  expect(res.body.id).toBeDefined();
});

it('DELETE /products/:id з роллю ADMIN видаляє товар (200)', async () => {
  const createRes = await request(app)
    .post('/products')
    .set('Authorization', `Bearer ${adminToken}`)
    .send({
      name: 'Product to delete',
      price: 50,
      categoryId: 30005,
      brandId: 30001,
```

```
            sku: `DEL-${Date.now()}`,
            slug: `delete-me-${Date.now()}`,
            description: 'Will be deleted',
          });
      const productId = createRes.body.id;

      const deleteRes = await request(app)
        .delete(`/products/${productId}`)
        .set('Authorization', `Bearer ${adminToken}`);
      expect(deleteRes.status).toBe(200);
    });
});
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Проєктування та оптимізація бази даних для високонавантаженого веб-застосунку інтернет-магазину "

Обсяг пояснювальної записки: 75____ аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____