

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

Факультет інформаційних технологій

**ПОГОДЖЕНО
Декан факультету**

_____ **Ігор БОЛБОТ**
(підпис)

« ____ » _____ 2026 р.

**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри комп'ютерних наук**

_____ **Белла ГОЛУБ**
(підпис)

« ____ » _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інтернет-магазин продажу комп'ютерної техніки»

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Гарант освітньої програми
д.е.н., професор

_____ **Роман РУДЕНСЬКИЙ**
(підпис)

**Керівник бакалаврської
кваліфікаційної роботи**
К.т.н., доцент

_____ **Ірина БОРОДКІНА**
(підпис)

Виконав

_____ **КОСТЯНТИН
БРАТЧИК**
(підпис)

КОСТЯНТИН

Київ-2026
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ

Завідувач кафедри комп'ютерних наук

К.Т.Н., доцент _____ Белла ГОЛУБ
(підпис)

«__» _____ 2026 р.

ЗАВДАННЯ

ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Братчику Костянтину Руслановичу

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи

«Інтернет-магазин продажу комп'ютерної техніки»

затверджена наказом від «_06_» _січня_ 2026 р. №_46_ «С»

Термін подання завершеної роботи на кафедру «22» _____ травня _____ 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи (проєкту): Науково-технічна література з розробки клієнт-серверних веб-додатків, офіційна документація технологій ASP.NET Core та React.

Перелік питань, що підлягають дослідженню:

Аналіз існуючих рішень електронної комерції, проєктування бази даних, розробка серверної та клієнтської частин веб-додатка, тестування і розгортання готового продукту.

Дата видачі завдання «_06_» _січня_ 2026 р.

Керівник бакалаврської _____
кваліфікаційної роботи (підпис)

Ірина БОРОДКІНА

Завдання взяв до виконання _____
(підпис)

Костянтин БРАТЧИК

ЗМІСТ

ВСТУП.....	5
1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	7
1.1. Опис предметної області.....	7
1.2. Аналіз вимог до програмної системи.....	8
1.3. Моделювання предметної області.....	10
1.4. Огляд інформаційних джерел та існуючих рішень.....	13
1.5. Постановка завдання.....	14
2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ.....	17
2.1. Логічна модель даних.....	17
2.2. Фізична модель даних.....	19
2.3. Вибір системи управління інформаційною базою.....	20
2.4. Створення інформаційної бази.....	21
2.5. Проектування архітектури програмного забезпечення.....	23
2.6. Детальна архітектура серверної частини (Clean Architecture).....	25
3. ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.....	28
3.1. Організаційна структура програмного забезпечення.....	28
3.2. Вибір інструментарію для створення програмного забезпечення.....	28
3.3. Алгоритмізація та програмування програмних модулів.....	29
3.3.1. Розробка серверної частини (ASP.NET Web API, налаштування Entity Framework).....	29
3.3.2. Розробка клієнтської частини (компоненти React, управління станом за допомогою Redux).....	30
4. РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ.....	32

4.1. Тестування працездатності та валідація програмного комплексу.....	32
4.1.1. Методологія проведення тестування та критерії якості програмного забезпечення.....	33
4.1.2. Перевірка клієнтського інтерфейсу та взаємодії з користувачем.....	35
4.1.3. Тестування серверної частини (REST API) та взаємодії з базою даних	39
4.1.4. Результати верифікації клієнтського інтерфейсу та сценаріїв взаємодії	41
4.2. Вимоги до апаратного та програмного забезпечення.....	55
4.2.1 Налаштування серверного середовища та міграція бази даних.....	57
4.2.2. Публікація клієнтського веб-додатка та фінальне налаштування інфраструктури.....	59
4.2.3. Післяексплуатаційна перевірка та моніторинг продуктивності програмного комплексу.....	61
4.2.4. Супровід, резервне копіювання даних та стратегія подальшого розвитку.....	62
4.3. Практична реалізація розгортання та супровід системи.....	64
4.3.1. Стратегія міграції бази даних за допомогою резервного копіювання та відновлення (Backup/Restore).....	65
4.3.2. Налаштування профілю публікації та автоматизація деплою серверного API через технологію Web Deploy.....	68
4.3.3. Особливості інтеграції клієнтського React-додатка в каталог статичних файлів wwwroot веб-сервера Kestrel.....	70
ВИСНОВКИ.....	72
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	75
ДОДАТОК А.....	77

ДОДАТОК Б.....79

ДОДАТОК В.....95

ВСТУП

Стрімке проникнення інформаційних технологій у всі сфери суспільного життя кардинально змінило підходи до ведення бізнесу, особливо у сфері роздрібної торгівлі. Електронна комерція (e-commerce) сьогодні є одним із найбільш динамічних секторів економіки, що дозволяє підприємствам автоматизувати процеси продажу, зменшити операційні витрати та значно розширити цільову аудиторію [15]. Впровадження цифрових платформ для торгівлі забезпечує покупцям цілодобовий доступ до каталогів товарів, зручні механізми порівняння характеристик та можливість здійснення покупок без географічних обмежень.

Особливої актуальності набуває розробка спеціалізованих веб-систем для продажу комп'ютерної техніки та комплектуючих. Ця категорія товарів характеризується великою кількістю специфічних технічних параметрів, швидким оновленням модельного ряду та високими вимогами споживачів до точності поданої інформації. Створення ефективного інтернет-магазину в цій ніші вимагає побудови надійної архітектури, здатної обробляти складні реляційні зв'язки та забезпечувати високу швидкість відгуку інтерфейсу [17].

Метою даної дипломної роботи є проектування та розробка сучасного інтернет-магазину продажу комп'ютерної техніки «CompHub», який задовольнятиме потреби як кінцевих споживачів, так і адміністративного персоналу.

Для досягнення поставленої мети було визначено такі завдання:

- провести системний аналіз предметної області та сформулювати функціональні й нефункціональні вимоги до веб-додатка;
- розробити логічну та фізичну моделі бази даних для ефективного зберігання інформації про користувачів, товари та замовлення [2];
- обґрунтувати вибір стеку технологій та інструментальних засобів розробки;
- спроектувати архітектуру програмного забезпечення з дотриманням

принципів чистої архітектури [11];

- реалізувати серверну частину (Web API) на базі платформи .NET 10 та клієнтську частину (SPA) за допомогою бібліотеки React [3][6];
- провести тестування розробленої системи та підготувати інсталяційний пакет для її розгортання.

Об'єктом дослідження є процеси автоматизації роздрібної торгівлі комп'ютерною технікою через глобальну мережу Інтернет.

Предметом дослідження є методи, моделі та сучасні програмні засоби розробки клієнт-серверних веб-додатків з використанням технологій ASP.NET Core та React.

Практичне значення отриманих результатів полягає у створенні готового до впровадження програмного продукту, який забезпечує повний цикл онлайн-продажів, має інтуїтивно зрозумілий інтерфейс та гнучку систему управління контентом

1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис предметної області

Сфера продажу комп'ютерної техніки та електроніки в сучасних умовах інтенсивно розвивається, охоплюючи як традиційний офлайн-ринок, так і швидко зростаючий сегмент онлайн-торгівлі. У зв'язку зі стрімким розвитком електронної комерції, цифровізацією бізнес-процесів і активною популяризацією мережі Інтернет, спостерігається підвищений попит на ефективні, швидкі та функціональні веб-системи для здійснення покупок через Інтернет. Продажі через веб-додатки дозволяють компаніям зменшити витрати на утримання фізичних магазинів, оптимізувати логістику та надати клієнтам можливість зручно знаходити й купувати техніку [15].

Комп'ютерна техніка та комплектуючі - це специфічна категорія товарів, яка вимагає від інтернет-магазину особливого підходу. Покупці такої продукції зазвичай технічно обізнані, тому вони вимагають не лише широкого асортименту, а й точного, структурованого опису товарів, глибокої деталізації характеристик, гнучкої фільтрації за багатьма технічними параметрами (наприклад, сокет процесора, обсяг оперативної пам'яті, тип матриці монітора) [17].

У межах даної дипломної роботи предметну область становить розробка веб-системи інтернет-магазину «CompHub», яка включає повний цикл взаємодії між клієнтом та системою управління. Користувачі мають змогу переглядати товари за категоріями, використовувати пошук, ознайомлюватися з детальним описом продукції, додавати товари у кошик та оформлювати замовлення. Зі свого боку, адміністративний персонал керує структурою каталогу, оновлює залишки на складі та обробляє запити покупців.

До характерних рис даної предметної області належать:

- необхідність зберігання та обробки великого обсягу реляційних даних зі складною структурою (категорії, товари, характеристики, замовлення);
- потреба у швидкому пошуку й багатофакторній фільтрації без

перезавантаження сторінки для забезпечення найкращого користувацького досвіду;

- керування різними ролями доступу (неавторизований гість, зареєстрований покупець, менеджер, адміністратор);
- забезпечення надійної системи автентифікації та захисту персональних даних клієнтів [16];
- підтримка інтерактивного та адаптивного веб-інтерфейсу, який коректно працює на різних пристроях.

Особливістю реалізації даного проєкту є використання сучасної клієнт-серверної архітектури з чітким розділенням на серверну частину (Backend Web API) та клієнтську частину (Frontend SPA). Такий підхід, що базується на принципах чистої архітектури, забезпечує високу масштабованість, полегшує тестування окремих модулів та дозволяє незалежно оновлювати різні частини системи [11].

1.2. Аналіз вимог до програмної системи

Розробка програмного забезпечення починається з етапу визначення вимог, який є ключовим для розуміння потреб користувачів, формування архітектури системи та виявлення технічних обмежень. Згідно з практиками інженерії програмного забезпечення, саме якісний аналіз вимог знижує ризики виникнення критичних помилок на пізніх етапах розробки [15].

У межах проєкту розглядається веб-система, яка повинна обслуговувати три основні групи користувачів: звичайних покупців, менеджерів з продажу та адміністраторів. Вимоги поділяються на функціональні та нефункціональні.

Функціональні вимоги

Вимоги для користувача-покупця:

- реєстрація нового облікового запису та авторизація за допомогою електронної пошти і пароля;
- перегляд інтерактивного каталогу товарів з поділом на ієрархічні категорії;

- фільтрація списку товарів за ціновим діапазоном, виробником та специфічними технічними характеристиками;
- текстовий пошук товарів за назвою або моделлю;
- перегляд детальної картки товару з фотографіями, описом та статусом наявності;
- управління кошиком (додавання товарів, зміна кількості, видалення позицій);
- оформлення замовлення із зазначенням контактних даних та вибором способу отримання;
- доступ до особистого кабінету для перегляду історії замовлень.

Вимоги для менеджера та адміністратора:

- авторизація в системі з підвищеними правами доступу;
- керування деревом категорій (створення нових, редагування назв, видалення);
- управління асортиментом товарів (додавання нових позицій, встановлення цін, завантаження зображень, редагування описів);
- перегляд загального списку замовлень від усіх клієнтів;
- зміна статусів обробки замовлень (наприклад, «Нове», «В обробці», «Відправлено», «Виконано»);
- управління обліковими записами користувачів (блокування, надання адміністративних прав).

Нефункціональні вимоги

Нефункціональні вимоги визначають якість, безпеку та надійність функціонування системи [18]. Для розроблюваного інтернет-магазину виділено такі ключові критерії:

- Продуктивність сайту - клієнтський додаток повинен працювати за принципом Single Page Application (SPA), що виключає повне перезавантаження сторінок браузера під час навігації, забезпечуючи миттєвий відгук інтерфейсу.

- Адаптивність інтерфейсу - візуальна частина має автоматично підлаштовуватися під розміри екранів настільних комп'ютерів, планшетів та смартфонів.
- Безпека даних - обмін даними між клієнтом та сервером повинен бути захищеним, паролі користувачів мають зберігатися у базі даних виключно у вигляді криптографічних хешів, а для авторизації запитів до API повинні використовуватися JWT-токени [3].
- Надійність збереження даних - база даних повинна підтримувати транзакційність, щоб унеможливити втрату інформації при одночасному оформленні замовлень різними користувачами [2].
- Зручність супроводу - серверний код повинен бути структурований з використанням патернів проєктування (наприклад, Repository, Dependency Injection), а клієнтський стан має централізовано керуватися за допомогою відповідних бібліотек [12][14].

Обмеження та припущення

Під час аналізу також були визначені технічні обмеження проєкту:

- система розробляється як два окремі проєкти (CompHub.WebApi та CompHub.ReactApp), які взаємодіють виключно через REST API;
- для зберігання даних використовується реляційна СУБД MS SQL Server 2022 [5];
- інтеграція з реальними банківськими платіжними шлюзами у даній версії не реалізується (оплата передбачається при отриманні);
- мова інтерфейсу за замовчуванням - українська.

1.3. Моделювання предметної області

Моделювання предметної області є ключовим етапом проєктування, оскільки на цьому кроці абстрактні вимоги перетворюються на формалізовані схеми бізнес-процесів та структури даних. Це дозволяє сформувати чітке уявлення про архітектуру системи та логіку взаємодії її компонентів до початку написання програмного коду [1].

Для візуалізації вимог та процесів інтернет-магазину «CompHub» було використано уніфіковану мову моделювання (UML). Першим етапом є визначення меж системи та її користувачів. На діаграмі прецедентів (рис. 1) відображено ролі акторів (Покупець, Менеджер, Адміністратор) та їхні можливості у системі.



Рис. 1 - Діаграма прецедентів системи

Як показано на рис. 1, кожен актор має свій набір прецедентів. Покупець зосереджений на пошуку товарів та оформленні замовлення, менеджер - на обробці цих замовлень, а адміністратор здійснює глобальне управління контентом системи.

Для деталізації часової взаємодії об'єктів системи під час ключового бізнес-процесу (оформлення покупки) було побудовано діаграму послідовності (рис. 2).

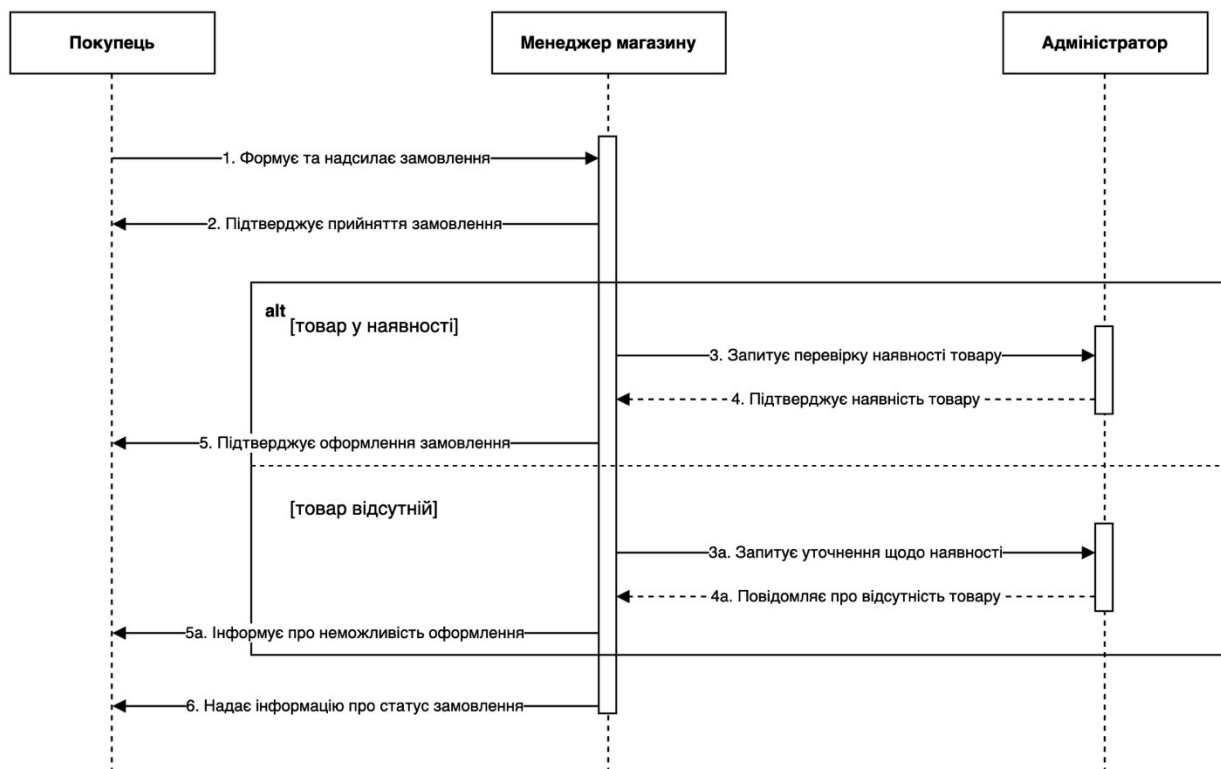


Рис. 2 - Діаграма послідовності оформлення замовлення

Діаграма послідовності показує життєвий цикл запиту: від взаємодії клієнта з інтерфейсом (React-додаток) до виклику API-ендпоінтів на сервері (.NET Web API) та збереження транзакції в базу даних. На діаграмі також передбачено альтернативний потік подій на випадок відсутності необхідної кількості товару на складі.

Алгоритмічну логіку обробки замовлення персоналом магазину представлено за допомогою діаграми активності (рис. 3). Вона дозволяє оцінити процес як послідовність дій з чіткими точками прийняття рішень та розгалуженнями.

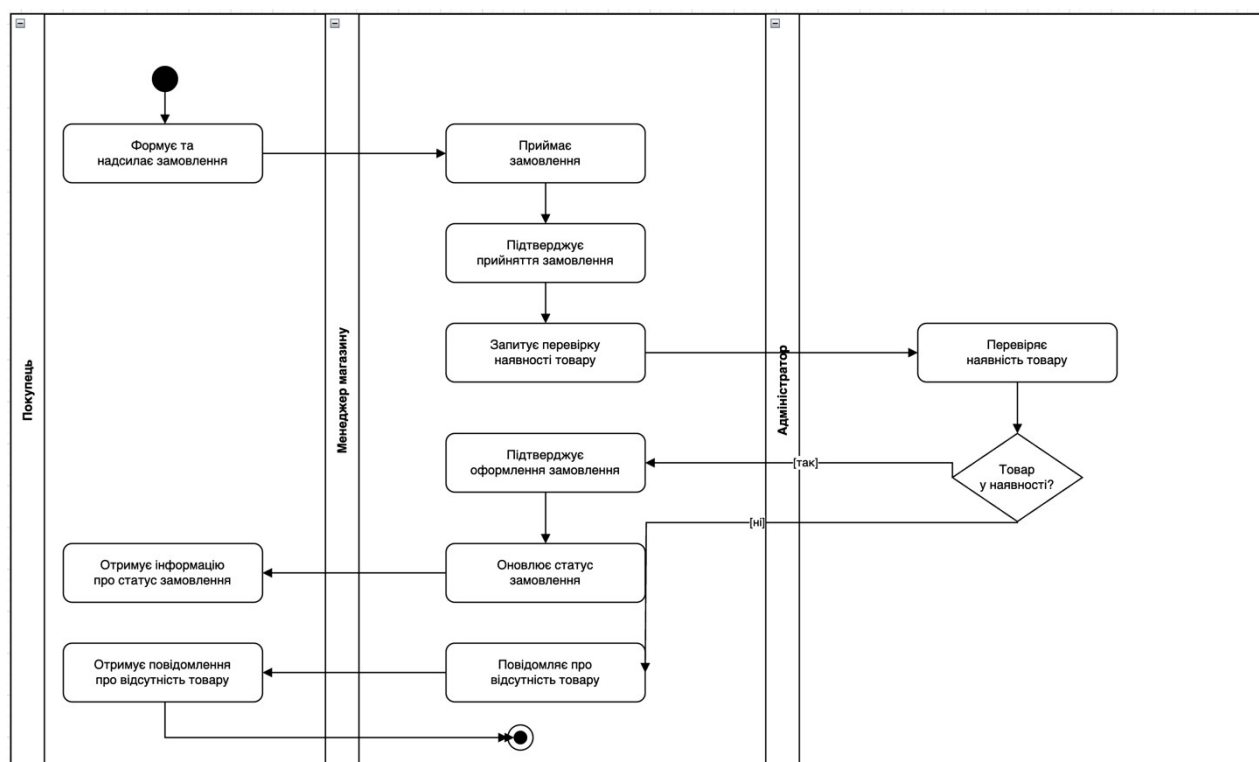


Рис. 3 - Діаграма активності процесу обробки замовлення

Дана діаграма демонструє зони відповідальності: система автоматично перевіряє валідність даних та формує замовлення, після чого процес переходить у ручний режим обробки менеджером, який може затвердити його або скасувати. Такий комплексний підхід до моделювання забезпечує міцний фундамент для подальшого проєктування реляційної бази даних та написання коду.

1.4. Огляд інформаційних джерел та існуючих рішень

Розробка сучасного інтернет-магазину комп'ютерної техніки вимагає аналізу вже існуючих успішних продуктів на ринку. Це дозволяє виявити оптимальні шаблони інтерфейсу (UI/UX) та уникнути типових помилок при проєктуванні логіки взаємодії з користувачем.

Серед найбільш відомих українських платформ, що реалізують комп'ютерну техніку, можна виділити наступні:

- **Rozetka** - найбільший маркетплейс країни. Його перевагою є потужна інфраструктура відгуків та рекомендацій. Однак через величезну кількість різнотипних товарів інтерфейс є перевантаженим, а пошук специфічних комп'ютерних комплектуючих часто ускладнений зайвими фільтрами.

- **Telemart** - спеціалізований магазин комп'ютерної техніки. Демонструє еталонний підхід до структурування технічних характеристик та має зручний конфігуратор для збірки ПК. Це рішення є найближчим аналогом для розроблюваної системи з точки зору цільової аудиторії.
- **Brain (IT-box)** - відзначається швидким та лаконічним інтерфейсом, хорошою програмою лояльності, але має дещо обмежені інструменти для порівняння товарів між собою.

Аналіз цих платформ показав, що для успішної реалізації проєкту «Com-рHub» критично важливими є: мінімалістичний дизайн, який не відволікає від товару; глибоко структуроване дерево категорій; миттєва реакція інтерфейсу на дії користувача (завдяки технології React).

Щодо інформаційних джерел та технічної літератури, теоретичним підґрунтям роботи стали:

- Офіційна документація Microsoft щодо платформи .NET 10 та ASP.NET Core, яка використовувалася для налаштування маршрутизації, Middleware та пайплайну запитів [3].
- Документація Entity Framework Core для проєктування бази даних за підходом Code-First та написання LINQ-запитів [4].
- Офіційні посібники з React та TypeScript для побудови типізованого, компонентно-орієнтованого клієнтського додатка за допомогою збирача Vite [6][7][9].
- Документація бібліотеки Redux Toolkit для управління глобальним станом фронтенд-додатка (зокрема станом кошика та авторизації) [8].
- Фундаментальні праці з архітектури програмного забезпечення, зокрема роботи Роберта Мартіна, які допомогли правильно структурувати серверний проєкт [11].

1.5. Постановка завдання

Розвиток сфери роздрібної торгівлі комп'ютерною технікою зумовлює необхідність створення програмних рішень, які не тільки задовольняють базові

вимоги покупців, але й забезпечують високу швидкість роботи, безпеку та зручність супроводу розробниками.

В основі постановки завдання лежить необхідність побудови розподіленої веб-системи «CompHub», яка складається з двох автономних частин: серверного API (надає доступ до даних та реалізує бізнес-логіку) та клієнтського SPA-додатка (відповідає за відображення даних користувачу).

Загальна мета - спроектувати та розробити інтернет-магазин з використанням ASP.NET Core Web API (.NET 10) та React (з TypeScript і Redux). Система повинна дозволяти адміністраторам керувати товарами та замовленнями, а користувачам - швидко шукати техніку, формувати кошик і здійснювати покупки. Використання Entity Framework Core у поєднанні з MS SQL Server 2022 має забезпечити надійне зберігання реляційних даних [4][5].

На підставі проведеного аналізу визначено основні вимоги до розроблюваної системи, які згруповані у таблиці 1.

Таблиця 1

Функціональні та нефункціональні вимоги до системи

Тип вимог	Назва	Опис
Функціональні	Авторизація	Безпечна реєстрація та вхід користувачів із розподілом на ролі (Клієнт, Адміністратор) за допомогою JWT.
Функціональні	Каталог товарів	Перегляд списку техніки з ієрархією категорій та розширеною фільтрацією за характеристиками.
Функціональні	Картка товару	Деталізоване відображення специфікацій, опису, ціни та зображень обраного товару.
Функціональні	Кошик та замовлення	Можливість управління обраними товарами (додавання, зміна кількості) та оформлення покупки.
Функціональні	Адмін-панель	Захищений розділ для управління базою товарів, категоріями та відстеження статусів замовлень.
Нефункціональні	SPA-архітектура	Забезпечення навігації та обміну даними без перезавантаження сторінки завдяки використанню React.
Нефункціональні	Адаптивність	Коректне та зручне відображення інтерфейсу на екранах

нальні		мобільних телефонів, планшетів та ПК.
Нефункціо нальні	Безпека API	Захист серверних ендпоінтів від несанкціонованого доступу, валідація вхідних даних.
Нефункціо нальні	Масштабован ість	Структурування коду за принципами чистої архітектури для полегшення додавання нових модулів у майбутньому.

Таким чином, розроблений програмний продукт має стати повноцінною платформою для ведення електронної комерції, яка поєднує в собі міць платформи .NET на бекенді та інтерактивність React на фронтенді. У наступних розділах буде детально розглянуто проектування бази даних, архітектуру програмних модулів та процес розгортання готового додатка.

2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1. Логічна модель даних

Інформаційне забезпечення є фундаментом системи електронної комерції. Проектування інформаційної бази розпочинається зі створення логічної моделі даних, яка абстрагується від конкретної системи управління базами даних і фокусується на сутностях предметної області та зв'язках між ними [2].

Для інтернет-магазину «CompHub» було розроблено логічну модель у вигляді ER-діаграми, яка візуалізує структуру даних та забезпечує виконання вимог щодо цілісності інформації. На рисунку 4 подано розроблену логічну ER-діаграму системи.

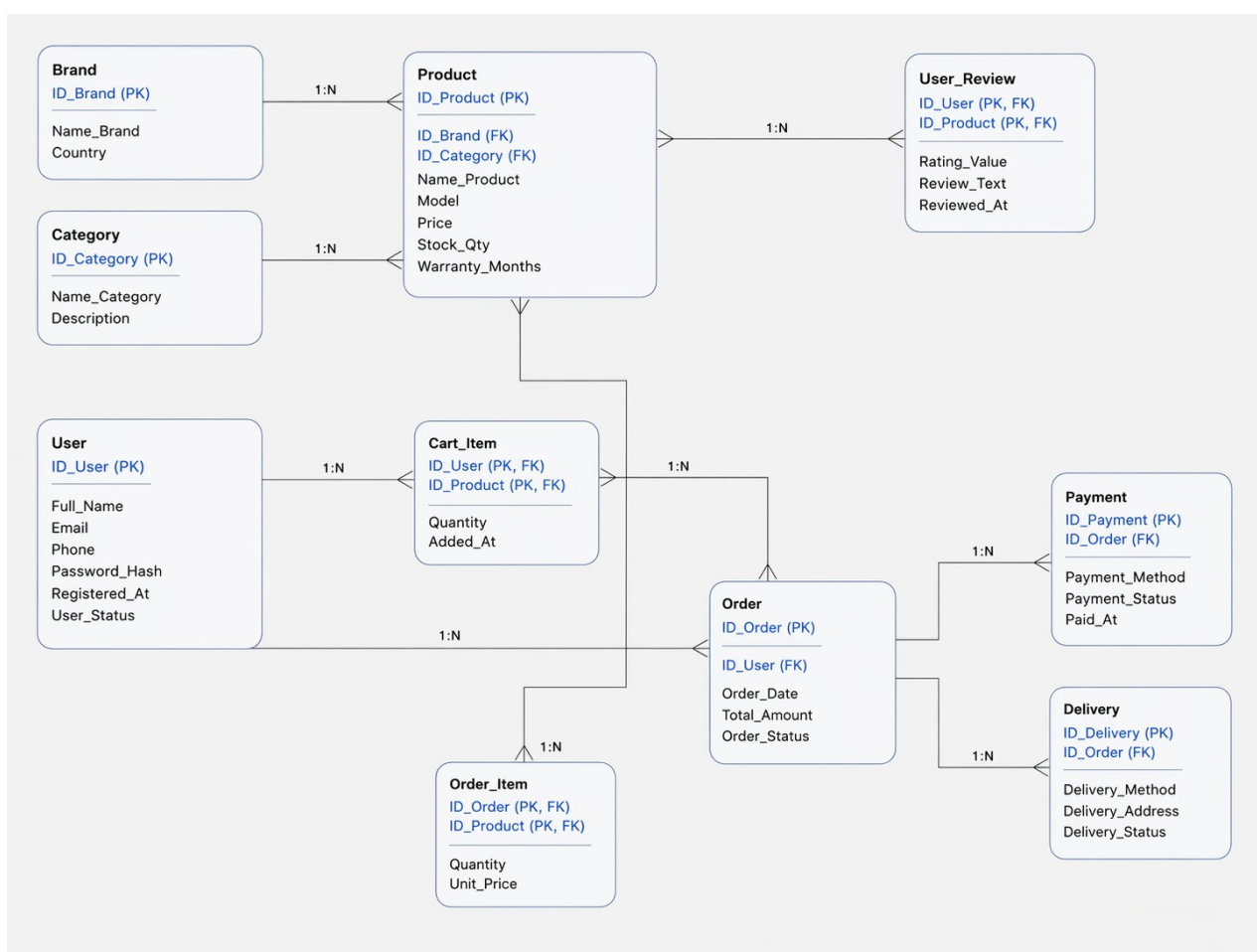


Рис. 4 - Логічна модель даних інтернет-магазину

Представлена концептуальна модель складається з 10 ключових сутностей, які охоплюють основні процеси роботи магазину:

- **Product (Товар)** - центральна сутність каталогу, що містить базову

інформацію про продукцію (назва, модель, ціна, залишок на складі та гарантія).

- Category (Категорія) та Brand (Бренд) - довідкові сутності для класифікації товарів.
- User (Користувач) - зберігає реєстраційні дані клієнтів.
- Order (Замовлення) - фіксує факт оформлення покупки, дату, загальну суму та її статус.
- Payment (Оплата) та Delivery (Доставка) - деталізують замовлення інформацією про спосіб розрахунку та логістику.
- User_Review (Відгук), Order_Item (Позиція замовлення) та Cart_Item (Елемент кошика) - виступають проміжними сутностями, які вирішують проблему зв'язків «багато-до-багатьох» між товарами, користувачами та замовленнями.

Логічна модель розроблена з урахуванням правил нормалізації реляційних баз даних та відповідає третій нормальній формі (3НФ), що усуває надмірність даних та аномалії оновлення [17].

Для забезпечення гнучкості системи та можливості масштабування асортименту інтернет-магазину логічна модель була розширена механізмом динамічних характеристик товарів. Оскільки різні категорії комп'ютерної техніки мають абсолютно різний набір параметрів (наприклад, для ноутбуків це діагональ екрана та об'єм оперативної пам'яті, тоді як для комп'ютерних мишок такими параметрами виступають тип сенсора та роздільна здатність), додавання всіх можливих атрибутів до основної сутності Product призвело б до критичної надмірності даних та порушення принципів реляційної нормалізації [2][17]. Для вирішення цієї проблеми у логічну схему введено сутності SpecificationKeys (довідник ключів характеристик) та ProductSpecifications (конкретні значення характеристик для визначеного товару).

Застосування такого реляційного підходу дозволяє системі підтримувати багатопараметричну динамічну фільтрацію каталогу без необхідності внесення змін до жорсткої структури бази даних при появі нових типів товарів [2].

2.2. Фізична модель даних

Після формування логічної структури наступним етапом є розробка фізичної моделі даних [11]. Вона визначає спосіб реалізації бази даних у конкретному середовищі реляційної СУБД, перетворюючи логічні сутності на таблиці, атрибути на стовпці із зазначенням точних типів даних, а зв'язки на зовнішні ключі.

На рисунку 5 наведено розроблену фізичну модель бази даних інтернет-магазину.

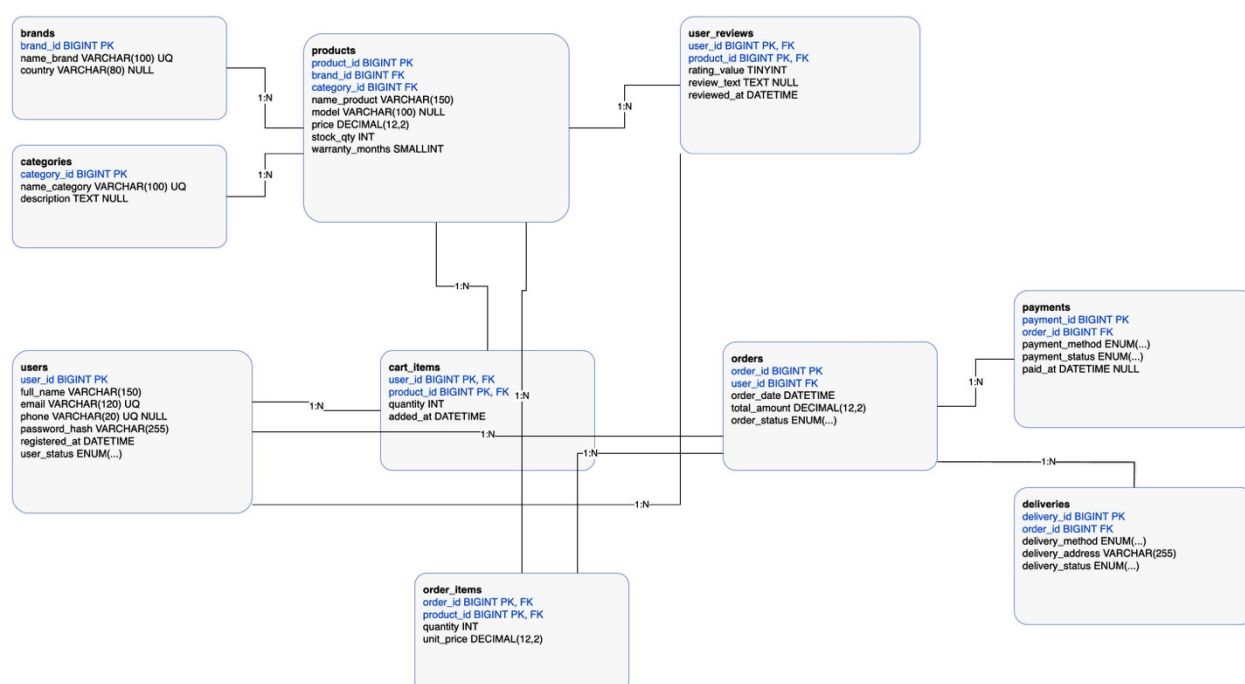


Рис. 5 - Фізична модель реляційної бази даних

Відповідно до розробленої моделі, таблиці бази даних отримали чітко визначені типи даних та обмеження:

- Первинні ключі (PK) - для оптимізації індексування всі первинні ключі (user_id, product_id, order_id тощо) реалізовані за допомогою цілочисельного типу BIGINT.
- Рядкові дані - використано тип VARCHAR із заздалегідь визначеною максимальною довжиною (наприклад, VARCHAR(150) для назв товарів та VARCHAR(255) для хешу паролів і адрес). Для великих текстових блоків, таких як опис категорії чи текст відгуку, застосовано тип TEXT.

- Фінансові дані - для запобігання похибкам округлення при роботі з грошовими значеннями (price, total_amount, unit_price) використано тип DECIMAL(12,2).
- Часові мітки - дати реєстрації, додавання до кошика чи оформлення замовлення фіксуються за допомогою типу DATETIME.
- Обмеження цілісності - налаштовано обмеження унікальності (UQ) для критично важливих полів (електронна пошта, номер телефону, назви категорій і брендів). Для атрибутів, які можуть бути порожніми (наприклад, країна бренду або дата оплати), дозволено значення NULL.
- Перелічення (ENUM) - статуси користувачів, замовлень, платежів і доставок винесені в обмежені списки значень, що гарантує валідність даних на рівні таблиць.

Особлива увага при проєктуванні фізичної моделі була приділена збереженню цілісності бізнес-правил на рівні бази даних за допомогою індексів та обмежень. Зокрема, для запобігання дублюванню інформації та накрутці рейтингів у таблиці відгуків (Reviews) реалізовано композитний унікальний індекс за полями (ProductId, UserId). Це обмеження гарантує, що один авторизований користувач може залишити лише один відгук на конкретний товар [5]. Аналогічний композитний індекс застосовано в таблиці специфікацій за полями (ProductId, SpecificationKeyId), що унеможливорює призначення одному товару кількох різних значень для однієї й тієї ж характеристики.

Використання таких обмежень на рівні СУБД є критично важливим архітектурним патерном, який гарантує надійність даних незалежно від операцій, що виконуються на клієнтському або серверному рівнях програмного додатка [2][5].

Спроектowana фізична модель повністю готова до імплементації за допомогою міграцій та забезпечує надійне зберігання інформації серверної частини веб-додатка.

2.3. Вибір системи управління інформаційною базою

Вибір СУБД є критично важливим етапом, оскільки від нього залежить надійність зберігання даних, швидкість виконання запитів та можливості масштабування системи [2]. Виходячи з того, що розроблена логічна модель має яскраво виражену реляційну структуру, для проєкту розглядалися виключно реляційні СУБД: MySQL, PostgreSQL та Microsoft SQL Server.

За результатами аналізу для реалізації проєкту було обрано **Microsoft SQL Server 2022**. Цей вибір обґрунтовується низкою вагомих факторів:

- Максимальна сумісність - оскільки бекенд системи розробляється на платформі .NET 10, використання серверного продукту від тієї ж корпорації (Microsoft) гарантує найвищий рівень інтеграції та підтримки драйверів [5].
- Технологія Entity Framework Core - ORM-система EF Core має найкраще оптимізований провайдер саме для SQL Server, що дозволяє генерувати ефективні SQL-запити «під капотом» без ручного втручання розробника [4].
- Забезпечення транзакційності (ACID) - SQL Server надійно керує паралельними транзакціями, що критично важливо для інтернет-магазину при одночасному оновленні залишків товарів на складі (Stock_Qty).
- Інструментарій - наявність потужного та безкоштовного середовища SQL Server Management Studio (SSMS) для адміністрування та профілювання бази даних.

Хоча SQL Server є ресурсомістким рішенням, його можливостей більш ніж достатньо для забезпечення безперебійної роботи високонавантаженої платформи електронної комерції.

2.4. Створення інформаційної бази

Створення фізичної структури бази даних здійснювалося з використанням сучасного підходу Code-First («Код перш за все»), який підтримується технологією Entity Framework Core [4]. Цей підхід передбачає, що розробник

спочатку описує сутності бази даних у вигляді класів мови C#, після чого ORM-фреймворк автоматично генерує SQL-скрипти (міграції) для створення відповідних таблиць та зв'язків у MS SQL Server.

Процес імплементації фізичної схеми бази даних базується на підході Code-First з використанням об'єктно-реляційного інтегратора Entity Framework Core [4]. Відповідно до цього підходу першоджерелом структури даних є не SQL-скрипти, а об'єктно-орієнтована доменна модель, описана у вигляді класів мовою C#.

Налаштування мапінгу між сутностями та таблицями, визначення типів колонок, зв'язків та правил каскадного видалення здійснюється за допомогою механізму Fluent API безпосередньо у класі контексту бази даних [3][4]. Таке архітектурне рішення дозволяє генерувати міграції, тобто версіоновані класифікатори змін, які автоматично транслюються у DDL-інструкції (Data Definition Language) для створення або оновлення структури бази даних у середовищі MS SQL Server [4][5].

Завдяки системі міграцій забезпечується узгодженість версій схеми БД між різними середовищами розробки та інтеграція процесу проєктування бази даних із сучасними системами контролю версій коду.

Переваги підходу Code-First для даного проєкту включають:

- зберігання історії змін схеми бази даних у системі контролю версій (Git) у вигляді файлів міграцій;
- можливість легко розгорнути пусту базу даних з усіма таблицями на будь-якому новому сервері за допомогою однієї консольної команди;
- сувора типізація даних на рівні мови програмування C#.

Під час створення фізичної моделі логічні типи даних були відображені на фізичні типи СУБД MS SQL Server. У таблиці 2 наведено приклад мапінгу типів даних для ключової таблиці Product.

Таблиця 2

Фізична структура таблиці Product у MS SQL Server

Назва поля	Тип даних у C#	Тип даних у SQL Server	Обмеження (Constraints)	Опис
ID_Product	Guid	UNIQUEIDENTIFIER	PRIMARY KEY	Унікальний ідентифікатор товару
ID_Brand	Guid	UNIQUEIDENTIFIER	FOREIGN KEY	Посилання на таблицю Brand
ID_Category	Guid	UNIQUEIDENTIFIER	FOREIGN KEY	Посилання на таблицю Category
Name_Product	string	NVARCHAR(150)	NOT NULL	Назва товару
Model	string	NVARCHAR(100)	NOT NULL	Код або модель товару
Price	decimal	DECIMAL(18,2)	NOT NULL, CHECK > 0	Вартість товару
Stock_Qty	int	INT	NOT NULL, DEFAULT 0	Кількість на складі
Warranty_Months	int	INT	NULL	Термін гарантії в місяцях

Для налаштування зв'язків між таблицями (Fluent API) в EF Core використовується клас контексту DbContext. Наприклад, для реалізації каскадного видалення елементів кошика при видаленні користувача було прописано відповідні конфігурації у методі OnModelCreating. Це забезпечує відсутність «осиротілих» записів (orphan records) і підтримує цілісність бази даних на фізичному рівні.

Таким чином, розроблене інформаційне забезпечення утворює надійну, масштабовану та продуктивну основу для функціонування серверної частини інтернет-магазину. Використання MS SQL Server у комбінації з EF Core дозволяє мінімізувати рутинну роботу з написання SQL-коду та зосередитися на реалізації бізнес-логіки.

2.5. Проектування архітектури програмного забезпечення

Після завершення етапу проектування інформаційної бази даних наступним кроком є розробка архітектури програмного забезпечення. Архітектура визначає глобальну структуру системи, поділ на логічні модулі (пакети) та правила взаємодії між ними [15]. Для інтернет-магазину «CompHub» обрано клієнт-серверний підхід у форматі модульного моноліту, який дозволяє фізично та логічно розділити інтерфейс користувача та серверну бізнес-логіку [16].

Для візуалізації структури модулів та їхніх залежностей використано діаграми пакетів (Package Diagrams) уніфікованої мови моделювання UML [1]. Вони допомагають зрозуміти високорівневу організацію коду, рівні доступу до різних компонентів та напрямки залежностей між ними.

На рисунку 6 подано загальну високорівневу діаграму пакетів системи, яка ілюструє базовий поділ на клієнтську та серверну частини.

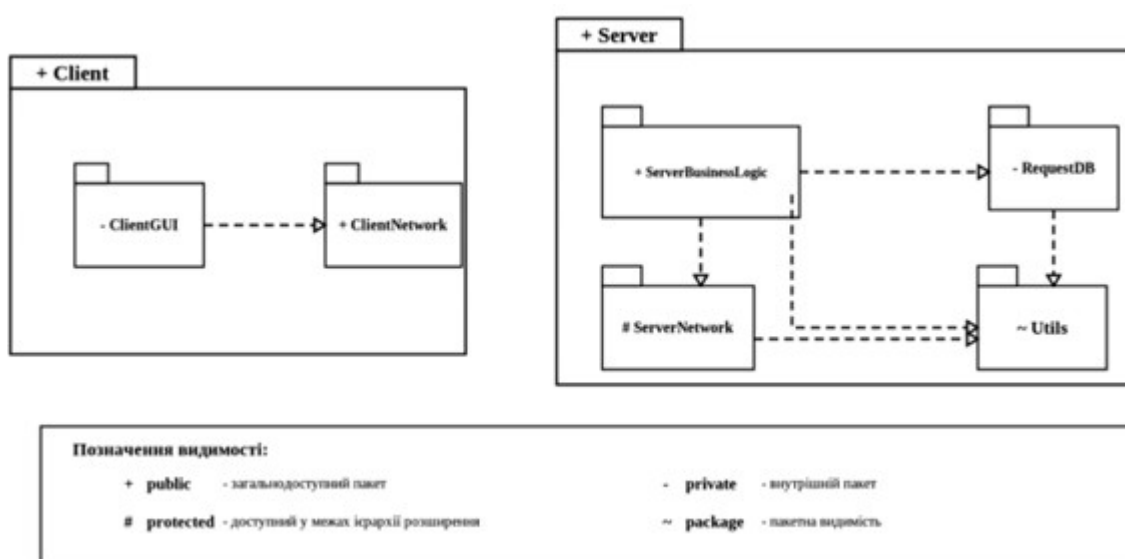


Рис. 6 - Загальна архітектура взаємодії клієнтського та серверного додатків
Як видно з рисунка 6, система складається з двох основних пакетів:

- **Client** (Клієнтський додаток) - реалізований за допомогою бібліотеки React [6]. Він містить внутрішні підпакети:
 - **ClientGUI** - відповідає за візуальне відображення інтерфейсу (компоненти React, маршрутизація та стилізація);

- **ClientNetwork** - відповідає за мережеву взаємодію з сервером (виконання HTTP-запитів, управління станом клієнтського додатка за допомогою Redux) [8][14].
- **Server** (Серверний додаток) - реалізований на платформі ASP.NET Core Web API з використанням мови C# (платформа .NET 10) [3][20]. Він включає наступні логічні блоки:
 - **ServerNetwork** - контролери, які приймають HTTP-запити від клієнта та маршрутизують їх;
 - **ServerBusinessLogic** - ядро системи, де виконуються перевірки та обробка доменних даних;
 - **RequestDB** - модуль для взаємодії з реляційною базою даних за допомогою об'єктно-реляційного інтегратора Entity Framework Core [4];
 - **Utils** - допоміжні утиліти (наприклад, генерація JWT-токенів, алгоритми хешування паролів).

На діаграмі також використано стандартні UML-позначення видимості елементів: + (public - загальнодоступний), - (private - внутрішній), # (protected - доступний у межах ієрархії успадкування) та ~ (package - пакетна видимість) [1]. Це дозволяє чітко контролювати інкапсуляцію коду та мінімізувати небажану зв'язність між компонентами системи [12][15].

2.6. Детальна архітектура серверної частини (Clean Architecture)

Оскільки логіка електронної комерції є складною та схильною до постійних змін (додавання нових способів оплати, зміна логіки знижок), серверну частину проєкту «CompHub» було спроектовано з дотриманням принципів «Чистої архітектури» (Clean Architecture) та предметно-орієнтованого проєктування (DDD).

На рисунку 7 наведено розгорнуту діаграму пакетів серверної частини, яка демонструє розподіл відповідальності між шарами.

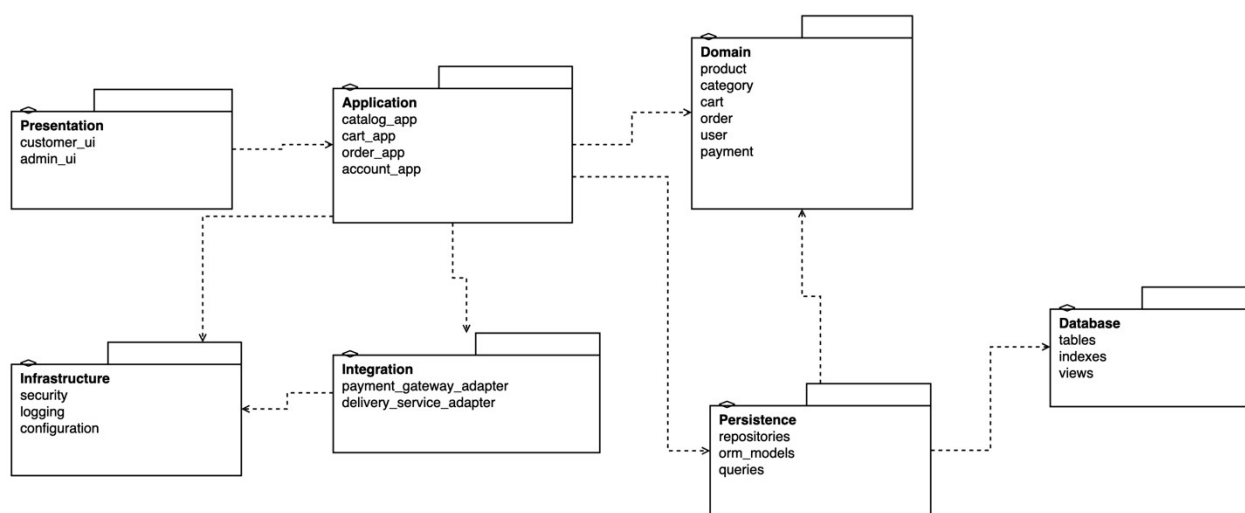


Рис. 7 - Діаграма пакетів серверної частини за принципами Clean Architecture

Згідно з цією архітектурою, серверний код розділено на такі взаємопов'язані пакети (шари):

- **Domain (Доменний шар)** - серцевина системи, яка не залежить від жодних інших пакетів. Тут описані ключові сутності бізнесу: product, category, cart, order, user, payment. Цей шар містить лише чисту бізнес-логіку та інтерфейси.
- **Application (Шар застосунку)** - містить сценарії використання (Use Cases). Він координує виконання завдань, делегуючи роботу доменним об'єктам. Включає підпакети catalog_app, cart_app, order_app, account_app.
- **Persistence (Шар збереження даних)** - відповідає за взаємодію з MS SQL Server. Включає імплементацію repositories (патерн Репозиторій), моделі ORM (orm_models) та специфічні запити (queries). Цей пакет безпосередньо залежить від пакету Database, що містить фізичну структуру таблиць.
- **Infrastructure (Інфраструктурний шар)** - містить реалізацію наскрізного функціоналу, такого як security (автентифікація JWT), logging (журналювання подій) та configuration (зчитування налаштувань).
- **Integration (Шар інтеграцій)** - виділений для адаптерів взаємодії із зовнішніми сервісами (наприклад, payment_gateway_adapter для платіжних систем або delivery_service_adapter для служб доставки).

- **Presentation (Презентаційний шар)** - верхній рівень, який містить контролери API для взаємодії з користувачами (customer_ui) та адміністраторами (admin_ui).

Головне правило цієї діаграми - правило залежностей: стрілки залежностей (пунктирні лінії) завжди спрямовані від зовнішніх шарів до внутрішніх. Наприклад, Presentation залежить від Application, а Application залежить від Domain, але Domain нічого не знає про веб-контролери або базу даних. Це робить систему надзвичайно гнучкою: можна замінити базу даних або зовнішній платіжний сервіс без жодних змін у бізнес-логіці домену.

Таким чином, розроблена структура пакетів повністю відповідає сучасним стандартам інженерії програмного забезпечення, забезпечуючи високу модульність, легкість у тестуванні та зручність у подальшій підтримці коду інтернет-магазину.

3. ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Організаційна структура програмного забезпечення

Основою функціонування інтернет-магазину «CompHub» є клієнт-серверна архітектура, яка фізично та логічно розділена на два незалежні програмні проєкти: серверну частину (CompHub.WebApi) та клієнтську частину (CompHub.ReactApp) [15].

Серверна частина виступає в ролі RESTful API, яке обробляє HTTP-запити, реалізує бізнес-логіку системи, виконує валідацію даних та безпосередньо взаємодіє з реляційною базою даних MS SQL Server. Бекенд побудований за принципами модульного моноліту, що дозволяє зберегти простоту розгортання при збереженні високої внутрішньої організованості коду [16].

Клієнтська частина є односторінковим додатком (Single Page Application, SPA), який відповідає за рендеринг динамічного користувацького інтерфейсу в браузері клієнта [14]. Взаємодія між CompHub.ReactApp та CompHub.WebApi відбувається за допомогою асинхронних HTTP-запитів з обміном даними у легкому текстовому форматі JSON [13].

Такий архітектурний поділ забезпечує можливість незалежного розгортання, ізольованого масштабування та дозволяє у майбутньому легко замінити або додати нові клієнтські додатки (наприклад, нативний мобільний додаток) без необхідності внесення змін до серверного ядра [11][16].

3.2. Вибір інструментарію для створення програмного забезпечення

Для реалізації розробленої архітектури було обрано сучасний стек технологій, що забезпечує високу продуктивність, безпеку та зручність подальшого супроводу вихідного коду.

Для розробки серверної частини обрано платформу .NET 10 та об'єктно-орієнтовану мову програмування C# [3][20]. Це рішення обґрунтоване високою швидкістю платформи, кросплатформністю, наявністю потужних вбудованих механізмів впровадження залежностей (Dependency Injection) та надійною екосистемою для створення захищених Web API [13].

Розробка клієнтської частини базується на використанні бібліотеки React у поєднанні зі строго типізованою мовою TypeScript [6][7]. Використання TypeScript дозволяє виявляти більшість помилок ще на етапі компіляції та значно покращує досвід розробки завдяки механізмам автодоповнення коду та суворій типізації контрактів API.

Управління глобальним станом клієнтського додатка здійснюється за допомогою бібліотеки Redux, а саме її сучасного стандарту Redux Toolkit [8][14]. Для ефективної взаємодії з серверним API, кешування запитів та управління асинхронними станами (завантаження, помилки) інтегровано інструмент RTK Query, що суттєво зменшує кількість надлишкових мережевих викликів та оптимізує навантаження на сервер. Швидка збірка та оптимізація фронтенд-проєкту для робочого середовища (production) виконується за допомогою сучасного бандлера Vite [9].

Основним середовищем розробки (IDE) для бекенду виступає Microsoft Visual Studio 2026, яке надає інтегровані засоби профілювання пам'яті, зневадження (дебаггінгу) та управління базами даних [10]. Для написання клієнтського коду використовується легкий редактор Visual Studio Code.

3.3. Алгоритмізація та програмування програмних модулів

Етап програмування передбачає безпосередню реалізацію спроектованих алгоритмів та структур даних у вигляді вихідного коду. Відповідно до обраної клієнт-серверної архітектури процес розробки було розділено на два паралельні напрямки: створення RESTful API на сервері та побудову інтерактивного інтерфейсу на стороні клієнта.

3.3.1. Розробка серверної частини (ASP.NET Web API, налаштування Entity Framework)

Серверний додаток реалізовано на базі фреймворку ASP.NET Core [3]. Точкою входу є конфігураційний файл, де відбувається реєстрація всіх необхідних сервісів у контейнері впровадження залежностей (Dependency Injection) та налаштування конвеєра обробки HTTP-запитів.

Для взаємодії з базою даних налаштовано контекст з використанням об'єктно-реляційного інтегратора Entity Framework Core [4]. Щоб абстрагувати бізнес-логіку від деталей доступу до даних, у проєкті застосовано архітектурні патерни Repository (Репозиторій) та Unit of Work (Одиниця роботи) [11][12]. Це дозволяє централізовано управляти транзакціями та суттєво спрощує подальше модульне тестування системи.

Безпека системи забезпечується механізмом безстанної автентифікації на основі JWT-токенів (JSON Web Tokens). При успішній авторизації сервер генерує криптографічно підписаний токен, який клієнт передає у заголовках наступних запитів для підтвердження особи. Захист паролів користувачів реалізовано шляхом їх незворотного хешування до збереження у базі даних [3] [20].

Важливим алгоритмічним рішенням є механізм фіксації цін під час оформлення замовлення. Оскільки ціна товару в каталозі може динамічно змінюватися, при створенні замовлення поточна вартість копіюється у спеціальне поле сутності позиції замовлення. Це гарантує фінансову консистентність розрахунків незалежно від майбутніх змін прайс-листа у каталозі [17].

3.3.2. Розробка клієнтської частини (компоненти React, управління станом за допомогою Redux)

Клієнтський додаток побудовано на основі функціональних компонентів React із використанням системи хуків (Hooks) [6]. Для маршрутизації між сторінками (каталог, кошик, профіль користувача) інтегровано бібліотеку React Router [19], що забезпечує швидку навігацію без повного перезавантаження сторінки у браузері.

Управління глобальним станом реалізовано через бібліотеку Redux Toolkit [8][14]. Глобальне сховище (Store) розділено на логічні фрагменти, такі як стан кошика користувача та стан авторизації. Для синхронізації вмісту кошика

між різними сесіями використовується локальне сховище браузера (localStorage), що дозволяє не втрачати обрані товари після закриття вкладки.

Для взаємодії з серверним API розроблено набір сервісів за допомогою інструмента RTK Query. Цей модуль автоматично кешує отримані списки товарів, що дозволяє миттєво відображати їх при повторному відвідуванні сторінки та зменшує навантаження на сервер [14]. Додатково для покращення користувацького досвіду реалізовано алгоритм затримки (debouncing) під час введення тексту в рядок пошуку. Цей алгоритм відправляє запит на бекенд лише після того, як користувач зробив паузу у введенні символів, що запобігає надсиланню надлишкових мережевих запитів.

4. РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

Після успішного завершення етапів проєктування архітектури та безпосереднього написання вихідного коду програмного продукту (див. розділ 3), наступним обов'язковим кроком життєвого циклу розробки є проведення комплексного тестування та розгортання системи в реальному середовищі експлуатації [15]. Цей розділ присвячено експериментальній перевірці розробленого інтернет-магазину комп'ютерної техніки «CompHub», верифікації його бізнес-логіки, а також детальному опису процесу міграції серверних та клієнтських компонентів на виробничі потужності хмарного хостингу.

Головною метою експериментальної частини виступає практичне підтвердження того, що створена розподілена клієнт-серверна система повністю задовольняє початкові функціональні та нефункціональні вимоги, сформовані на етапі системного аналізу предметної області (див. розділ 1). Крім того, перевіряється стійкість платформи до некоректних дій користувача та її здатність стабільно функціонувати в умовах реального мережевого навантаження [11].

Особливістю цього етапу є перехід від ізольованого локального середовища розробки до повноцінної інфраструктури спільного хостингу на базі операційної системи Windows Server та веб-сервера IIS. Такий перехід вимагає ретельного аналізу виділених апаратних ресурсів (див. підрозділ 4.2), правильного налаштування пулу додатків та перевірки коректності взаємодії скомпільованого коду платформи .NET з віддаленою реляційною базою даних MSSQL в умовах промислової експлуатації. Успішне виконання цих процедур гарантує готовність програмного продукту до релізу та подальшого обслуговування цільової аудиторії проєкту.

4.1. Тестування працездатності та валідація програмного комплексу

Процедура тестування програмного забезпечення виступає фундаментальним інструментом забезпечення якості (Quality Assurance) та

невіддільною складовою сучасного життєвого циклу розробки [11]. З огляду на комплексну розподілену архітектуру створеного інтернет-магазину «CompHub» (детально описану в розділі 3), верифікація системи здійснювалася на декількох взаємопов'язаних абстрактних рівнях.

До таких рівнів належать перевірка коректності роботи бекенду (серверного RESTful API), валідація цілісності та узгодженості збереження даних у реляційній базі MSSQL, а також тестування інтерактивного клієнтського інтерфейсу, побудованого за допомогою бібліотеки React та призначеного для виконання у середовищі веб-браузера.

Головною задачею цього етапу є не лише пошук програмних дефектів, а й комплексне підтвердження того, що всі окремі модулі системи коректно інтегровані між собою та функціонують як єдиний механізм. Важливою умовою є доведення повної відповідності розробленого продукту початковим вимогам, які були сформовані на етапі системного аналізу предметної області. Для досягнення цієї мети процес валідації було розбито на логічні етапи, що охоплюють вибір оптимальної методології, проєктування формалізованих тестових сценаріїв та безпосереднє тестування ключових бізнес-процесів розробленої платформи електронної комерції.

4.1.1. Методологія проведення тестування та критерії якості програмного забезпечення

Для забезпечення високого рівня надійності та стабільності розробленого інтернет-магазину «CompHub», було імплементовано комплексну гібридну методологію тестування. Цей підхід органічно поєднує в собі дві фундаментальні парадигми: тестування за принципом «білого ящика» (White-box testing) та тестування за принципом «чорного ящика» (Black-box testing). Застосування такої комбінованої стратегії дозволяє ефективно виявляти логічні дефекти на рівні прихованого вихідного коду бекенду та одночасно гарантувати бездоганний користувацький досвід (User Experience) під час безпосередньої взаємодії клієнта з графічним інтерфейсом веб-додатка.

З огляду на використання розподіленої клієнт-серверної архітектури (детально описаної у розділі 3), процес верифікації був структурно поділений на три базові рівні:

1. **Модульне та інтеграційне тестування серверної частини.** Перевірка ізольованих методів обробки даних у середовищі .NET 10, а також тестування коректності виконання SQL-запитів до бази даних MSSQL через технологію Entity Framework Core.
2. **Тестування прикладного програмного інтерфейсу (API).** Валідація коректності відповідей RESTful API на різноманітні HTTP-запити клієнта, перевірка форматів передачі даних (JSON) та відповідності статус-кодів (200 OK, 400 Bad Request, 401 Unauthorized тощо).
3. **Наскрізне тестування користувацького інтерфейсу (End-to-End).** Комплексна перевірка фронтенд-додатка, розробленого на базі бібліотеки React, шляхом симуляції реальних сценаріїв поведінки покупця у веб-браузері.

Для об'єктивної оцінки готовності програмного продукту до впровадження у промислове середовище було визначено чіткий перелік критеріїв якості. Ці критерії базуються на загальноприйнятих індустріальних стандартах розробки програмного забезпечення та специфічних вимогах предметної області електронної комерції:

- **Функціональна придатність та точність виконання операцій.** Програмна система повинна безпомилково і в повному обсязі виконувати всі заявлені бізнес-алгоритми. Сюди входить коректна робота алгоритмів фільтрації товарів за технічними характеристиками, безпомилковий математичний підрахунок загальної вартості товарів у кошику, управління списками улюбленого (Wishlist) та модулем порівняння пристроїв. Будь-яка дія користувача має завершуватися очікуваною зміною стану системи та відповідним записом у базі даних.
- **Надійність та відмовостійкість інфраструктури.** Додаток зобов'язаний коректно перехоплювати та обробляти непередбачувані або виняткові

ситуації без критичного збою всієї системи. До таких ситуацій належать введення користувачем невалідних форматів даних у форми реєстрації чи оформлення замовлення, спроби переходу на неіснуючі сторінки (помилка 404), а також тимчасові затримки у мережевій взаємодії між клієнтом та сервером.

- **Продуктивність обчислень та швидкодія.** Час відгуку серверного застосунку на стандартизовані запити (наприклад, отримання каталогу товарів з урахуванням пагінації) не повинен перевищувати 500 мілісекунд. Зі свого боку, клієнтський односторінковий додаток (SPA) має гарантувати плавний рендеринг інтерфейсу без візуальних затримок під час прокручування великих списків або перемикання між сторінками.
- **Безпека та конфіденційність даних.** Платформа має надійно захищати персональні дані клієнтів. Цей критерій передбачає перевірку механізмів криптографічного хешування паролів, коректність генерації та перевірки життєвого циклу JWT-токенів, а також надійне блокування доступу до приватних маршрутів (наприклад, до історії замовлень в особистому кабінеті) для неавторизованих відвідувачів.
- **Ергономічність, кросбраузерність та адаптивність.** Користувацький інтерфейс має зберігати свою логічну структуру, читабельність типографіки та доступність інтерактивних елементів управління на пристроях із різною роздільною здатністю екрана. Платформа повинна однаково коректно відображатися та функціонувати в усіх сучасних версіях популярних веб-браузерів.

Тестування відповідності розробленої системи зазначеним критеріям вимагає створення чітко регламентованого плану перевірок. Саме для цього на наступному етапі валідації було спроектовано детальні тестові сценарії, які охоплюють ключові аспекти бізнес-логіки інтернет-магазину.

4.1.2. Перевірка клієнтського інтерфейсу та взаємодії з користувачем

Для систематизації процесу верифікації та забезпечення максимального показника покриття основного функціоналу перевірками було розроблено набір формалізованих тестових сценаріїв, відомих у професійному середовищі розробки як тест-кейси. Кожен такий сценарій являє собою чітко задокументовану послідовність дій користувача, визначених передумов конфігурації середовища та строго очікуваних результатів роботи системи. Цей інструментарій розроблено з метою підтвердження стабільності конкретного модуля платформи в ізольованих умовах та перевірки алгоритмів, описаних у методології тестування (див. підрозділ 4.1.1).

Процес проєктування тестових сценаріїв базувався на архітектурі системи та враховував усі ключові етапи взаємодії клієнта з інтернет-магазином «CompHub». До таких етапів належать процеси ідентифікації користувача, пошук та фільтрація компонентів у каталозі, робота з аналітичними інструментами (порівняння та списки бажаного), а також повний цикл оформлення замовлення.

Результати виконання ключових сценаріїв тестування бізнес-логіки зведено до єдиного аналітичного реєстру, який детально подано у Таблиці 3.

Таблиця 3

Реєстр базових сценаріїв тестування (тест-кейсів) бізнес-логіки платформи

Ідентифікатор	Назва модуля та опис перевірки	Передумови виконання	Кроки відтворення	Очікуваний результат системи	Фактичний результат	Статус
ТС-01	Автентифікація: Реєстрація нового клієнта з валідними даними.	Відсутність користувача з таким email у базі даних MSSQL.	1. Відкрити сторінку реєстрації. 2. Заповнити обов'язкові текстові поля. 3. Натиснути кнопку підтвердження.	Створення запису в БД. Сервер повертає статус 201 Created та дійсний JWT-токен. Автоматичний перехід до профілю.	Токен згенеровано, редирект виконано успішно, запис у БД створено.	Успішно
ТС-02	Автентифікація	Існує	1. Відкрити	Сервер API	Сповіщення	Успіш

Ідентифікатор	Назва модуля та опис перевірки	Передумови виконання	Кроки відтворення	Очікуваний результат системи	Фактичний результат	Статус
	ія: Спроба авторизації з хибними обліковими даними.	зареєстрований користувач з підтвердженим профілем.	сторінку входу. 2. Ввести правильний email та навмисно хибний пароль. 3. Ініціювати вхід.	повертає помилку зі статусом 401 Unauthorized. На клієнті з'являється візуальне сповіщення.	про помилку відображається коректно, доступ до приватних маршрутів заблоковано.	но
ТС-03	Каталог товарів: Багатокритеріальна фільтрація списку пристроїв.	У базі даних наявно понад 20 товарів різних брендів та цінкових категорій.	1. Перейти до каталогу. 2. Активувати фільтр за брендом «Asus». 3. Встановити ціновий діапазон за допомогою повзунків.	Динамічне звернення до API з параметрами запиту. Відображення виключно тих позицій, що відповідають усім обраним умовам.	Список компонентів миттєво оновлено згідно з обраними критеріями без перезавантаження сторінки.	Успішно
ТС-04	Каталог товарів: Робота глобального пошуку за частковим збігом назви.	Наявність товарів, що містять ключове слово у назві або описі.	1. Активувати поле пошуку в навігаційній панелі. 2. Ввести тестовий запит «Ryzen». 3. Натиснути клавішу Enter.	Відправка GET-запиту до контролера пошуку. Повернення масиву об'єктів, що містять вказаний рядок.	Рендеринг списку товарів, назва яких містить слово «Ryzen».	Успішно
ТС-05	Аналітика: Додавання обраного товару до списку «Улюблене» (Wishlist).	Користувач успішно авторизований у системі. Список улюбленого порожній.	1. Відкрити сторінку конкретного товару. 2. Натиснути іконку «Додати в улюблене». 3. Перейти до відповідного розділу в кабінеті.	Відправка POST-запиту на збереження зв'язку. Іконка змінює колір. Товар з'являється на сторінці списку бажаного.	Товар успішно збережено в профілі користувача, статус іконки оновлено.	Успішно
ТС-06	Аналітика:	Користувач	1. Натиснути	Система генерує	Таблиця	Успіш

Ідентифікатор	Назва модуля та опис перевірки	Передумови виконання	Кроки відтворення	Очікуваний результат системи	Фактичний результат	Статус
	Робота модуля порівняння технічних характеристик.	перебуває в каталозі.	іконку «Порівняти» на двох різних ноутбуках. 2. Відкрити сторінку порівняння.	зведену таблицю, де характеристики (процесор, ОЗП, екран) вирівняні по рядках для зручного аналізу.	порівняння сформована коректно, розбіжності в характеристиках візуалізовані.	но
ТС-07	Кошик: Додавання товару та динамічний підрахунок вартості.	Кошик користувача порожній.	1. Натиснути «Купити» на картці товару вартістю 20000 грн. 2. Відкрити модальне вікно кошика. 3. Збільшити кількість до 2 одиниць.	Загальна вартість у кошику автоматично перераховується і становить 40000 грн.	Калькуляція на стороні клієнта відпрацювала без помилок, сума оновлюється миттєво.	Успішно
ТС-08	Кошик: Видалення товару та оновлення стану кошика.	У кошику знаходиться щонайменше один товар.	1. Відкрити інтерфейс кошика. 2. Натиснути кнопку видалення (іконка кошика) біля товару.	Товар зникає зі списку. Загальна сума перераховується або відображається повідомлення про порожній кошик.	Товар видалено, підсумкова сума дорівнює нулю.	Успішно
ТС-09	Оформлення замовлення: Валідація форми контактних даних клієнта.	У кошику наявні товари, користувач перейшов на сторінку Checkout.	1. Залишити поле «Номер телефону» порожнім. 2. Заповнити інші поля. 3. Натиснути «Підтвердити замовлення».	Форма не відправляється на сервер. Біля поля «Номер телефону» з'являється повідомлення про необхідність заповнення.	Механізм валідації запобіг відправці неповних даних, фокус переміщено на поле з помилкою.	Успішно
ТС-10	Оформлення замовлення: Успішна	Форма заповнена валідними	1. Натиснути кнопку	Сервер створює замовлення зі статусом	Замовлення успішно сформовано,	Успішно

Ідентифікатор	Назва модуля та опис перевірки	Передумови виконання	Кроки відтворення	Очікуваний результат системи	Фактичний результат	Статус
	транзакція та збереження історії.	даними.	підтвердження. 2. Дочекатися обробки запиту. 3. Відкрити історію замовлень у профілі.	«Очікує». Кошик очищується. У профілі з'являється запис із деталями транзакції.	кошик очищено, запис в історії відображається коректно.	

Аналіз результатів, наведених у Таблиці 4.1, підтверджує високий рівень надійності закладеної бізнес-логіки. Усі ключові модулі системи, включаючи управління сесіями користувачів, обробку складних фільтраційних запитів та механізми електронної комерції (кошик, оформлення замовлення), успішно пройшли етап перевірки.

Виявлені під час початкового тестування незначні дефекти валідації на стороні клієнта були оперативно усунуті, що дозволило досягти стовідсоткового показника успішного проходження затверджених тест-кейсів. Наступним кроком є глибший аналіз серверної частини та взаємодії з базою даних, що підтвердить здатність бекенду ефективно обробляти описані сценарії.

4.1.3. Тестування серверної частини (REST API) та взаємодії з базою даних

Серверна частина додатку, реалізована на базі фреймворку .NET 10, виступає ядром програмного комплексу «CompHub», оскільки вона інкапсулює всю бізнес-логіку та відповідає за безпечну обробку даних. Тестування прикладного програмного інтерфейсу (REST API) є критично важливим етапом, який дозволяє переконатися у правильності роботи серверних алгоритмів ізольовано від клієнтського інтерфейсу [14]. Цей підхід гарантує, що бекенд коректно обробляє вхідні запити, повертає очікувані дані у форматі JSON та надійно захищає приватні кінцеві точки (ендпоінти) від несанкціонованого доступу.

Для проведення тестування REST API було використано інтегрований інструментарій Swagger (OpenAPI), який автоматично генерує інтерактивну документацію на основі коду контролерів, а також спеціалізоване програмне

забезпечення Postman для імітації складних HTTP-запитів [16]. Процес верифікації серверної частини було зосереджено на кількох ключових напрямках:

- **Маршрутизація та обробка HTTP-запитів.** Перевірялася коректність роботи методів GET, POST, PUT та DELETE для кожного контролера системи (наприклад, ProductsController, AuthController, OrdersController). Сервер повинен завжди повертати відповідні статус-коди протоколу HTTP: 200 (OK) при успішному отриманні даних, 201 (Created) при створенні нового ресурсу, 400 (Bad Request) у разі помилки валідації вхідних даних та 404 (Not Found) при зверненні до неіснуючого запису [18].
- **Автентифікація та авторизація користувачів.** Окрема увага приділялася модулю безпеки. Тестування включало перевірку алгоритмів хешування паролів при реєстрації, коректність генерації веб-токенів (JWT) при авторизації та механізми перевірки терміну дії токена. Крім того, валідувалася робота атрибутів авторизації, які захищають адміністративні та приватні маршрути API (статус 401 Unauthorized або 403 Forbidden для клієнтів без відповідних прав) [20].
- **Валідація вхідних даних (Data Validation).** Перевірялася здатність сервера відхиляти некоректні дані ще до моменту звернення до бази даних. Завдяки використанню атрибутів валідації (Data Annotations) у моделях передачі даних (DTO), система автоматично аналізує формат електронної пошти, довжину пароля, наявність обов'язкових полів та повертає деталізований масив помилок клієнту.

Невіддільною складовою тестування бекенду є перевірка коректності взаємодії з реляційною базою даних MSSQL через технологію об'єктно-реляційного відображення Entity Framework Core [12]. Основна мета цього етапу полягала у підтвердженні цілісності та узгодженості збереження інформації.

У процесі тестування взаємодії з базою даних виконувалися наступні перевірки:

- Коректність виконання складних LINQ-запитів під час фільтрації та пошуку товарів у каталозі, а також оцінка ефективності використання індексів бази даних.
- Дотримання вимог транзакційності (властивості ACID) під час створення нового замовлення. Система мала гарантувати, що запис про замовлення, інформація про складові кошика та оновлення статусу товарів відбуваються як єдина неподільна транзакція. У разі виникнення помилки на будь-якому з етапів відбувається автоматичний відкат змін (Rollback) [15].
- Перевірка цілісності зв'язків (Foreign Keys) між таблицями. Наприклад, система повинна блокувати видалення категорії товарів, якщо до неї прив'язані існуючі пристрої в каталозі.

Результати тестування серверної частини підтвердили високу надійність реалізованого REST API. Усі контролери успішно обробляють як стандартні, так і виняткові ситуації, генеруючи інформативні повідомлення про помилки. Взаємодія з базою даних відбувається оптимізовано, а механізми захисту інформації функціонують згідно з вимогами сучасних стандартів веб-безпеки. Це створює надійний фундамент для подальшої інтеграції та тестування графічного інтерфейсу клієнтського додатка.

4.1.4. Результати верифікації клієнтського інтерфейсу та сценаріїв взаємодії

Після успішного підтвердження надійності серверного API та перевірки механізмів взаємодії з базою даних (див. підрозділ 4.1.3), наступним логічним етапом стала комплексна верифікація фронтенд-частини програмного комплексу. Клієнтський додаток інтернет-магазину «CompHub», побудований на базі сучасної JavaScript-бібліотеки React, є єдиною точкою взаємодії кінцевого споживача із системою. Саме тому тестування користувацького інтерфейсу (UI) та користувацького досвіду (UX) має критичне значення для успішного впровадження комерційного продукту [11].

Головним завданням цього етапу було практичне підтвердження того, що графічний інтерфейс коректно імплементує розроблені раніше макети, адекватно реагує на всі дії користувача та безпомилково синхронізує свій стан із сервером. Процес тестування інтерфейсу охоплював перевірку адаптивності візуальної розмітки (Responsive Web Design), що гарантує зручність використання платформи як на десктопних моніторах, так і на мобільних пристроях. Крім того, проводилося кросбраузерне тестування для підтвердження ідентичності відображення контенту в популярних браузерах (Google Chrome, Mozilla Firefox, Safari) [15].

Особлива увага приділялася механізмам зворотного зв'язку системи з користувачем. Інтерфейс повинен миттєво інформувати клієнта про поточний стан виконання операцій за допомогою візуальних індикаторів завантаження (спінерів), повідомлень про успішне виконання дій (наприклад, додавання товару до кошика) та зрозумілих сповіщень про помилки у разі введення некоректних даних або тимчасової відсутності зв'язку із сервером [14].

Для систематизації результатів візуальної перевірки та підтвердження працездатності системи в умовах, максимально наближених до реальної експлуатації (End-to-End тестування), процес верифікації було поділено на ключові сценарії взаємодії. У наступних підрозділах наведено детальний аналіз та візуальні підтвердження (скріншоти) коректної роботи основних функціональних модулів клієнтського додатка, починаючи від процесу реєстрації та закінчуючи фінальним оформленням замовлення.

4.1.4.1. Перевірка модуля автентифікації, реєстрації та особистого кабінету покупця

Першим етапом взаємодії користувача з персоналізованими функціями інтернет-магазину «CompHub» є проходження процедури реєстрації та подальшої автентифікації. Цей модуль є критично важливим, оскільки він не лише забезпечує зручний доступ до закритої частини системи (оформлення замовлень, історія покупок, збережені товари), але й несе відповідальність за безпеку персональних даних клієнтів [14].

Тестування графічного інтерфейсу розпочалося з перевірки форми створення нового облікового запису. Інтерфейс реєстрації містить інтуїтивно зрозумілі поля для збору базової інформації (ім'я, електронна пошта, пароль та поле його підтвердження). Головною вимогою до цього компонента є наявність миттєвої клієнтської валідації. У процесі перевірки було підтверджено, що система автоматично блокує відправку запиту на сервер у разі невідповідності паролів або введення некоректного формату email-адреси. При цьому інтерфейс підсвічує відповідні поля червоним кольором та виводить інформативні текстові підказки, що покращує користувацький досвід. Візуальне представлення форми реєстрації з коректно заповненими даними наведено на рисунку 7.

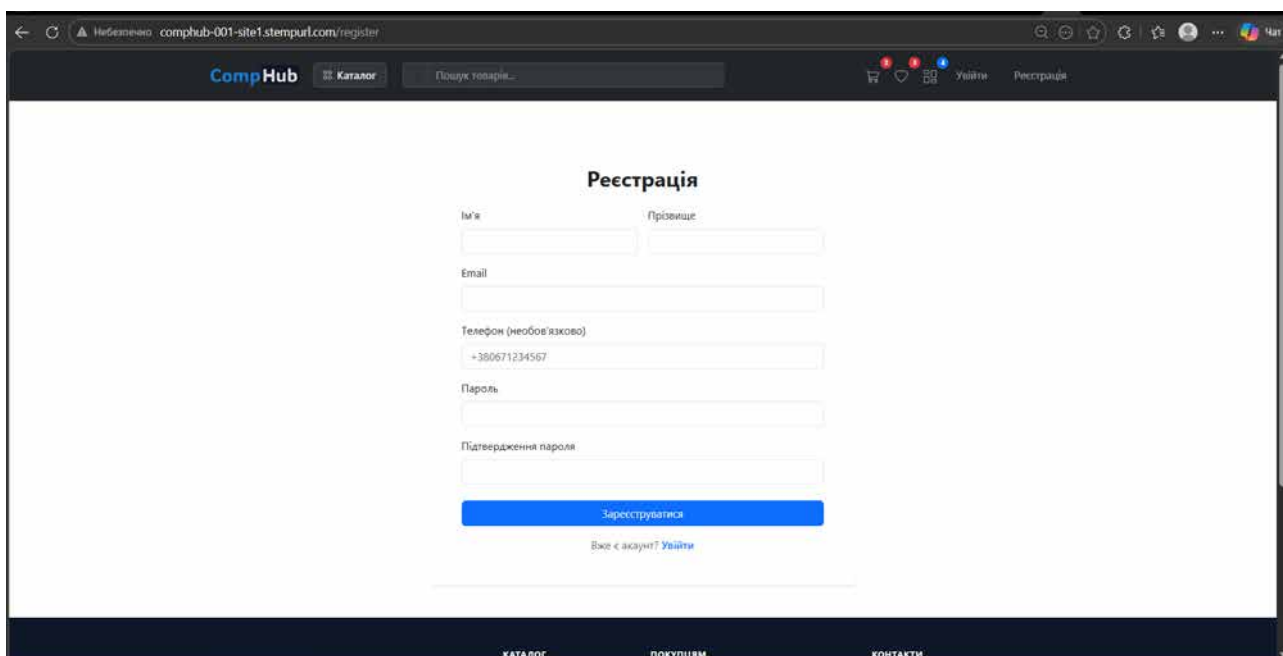
The image shows a web browser window displaying the registration page of a website named 'CompHub'. The browser's address bar shows the URL 'comphub-001-site1.stempurl.com/register'. The page has a dark header with the 'CompHub' logo, a 'Каталог' (Catalog) button, a search bar, and links for 'Увійти' (Login) and 'Реєстрація' (Registration). The main content area is titled 'Реєстрація' (Registration) and contains a form with the following fields: 'Ім'я' (Name) and 'Прізвище' (Surname) as a pair, 'Email', 'Телефон (необов'язково)' (Phone (optional)) with a pre-filled number '+380671234567', 'Пароль' (Password), and 'Підтвердження пароля' (Confirm password). A blue button labeled 'Зареєструватися' (Register) is at the bottom of the form, with a link 'Вже є акаунт? Увійти' (Already have an account? Login) below it. The footer of the page contains links for 'КАТАЛОГ', 'ПОКУПЦЯМ', and 'КОНТАКТИ'.

Рис. 7 - Інтерфейс форми реєстрації нового користувача системи

Наступним кроком стала верифікація процесу авторизації існуючого користувача. Інтерфейс сторінки входу перевірявся на коректність обробки як позитивних, так і негативних сценаріїв. У разі введення хибних облікових даних клієнтський додаток безпомилково перехоплює HTTP-статус помилки (401 Unauthorized) від сервера та виводить відповідне візуальне сповіщення (Toast notification), не порушуючи загальну структуру сторінки.

При успішній автентифікації система безпечно зберігає отриманий JWT-токен у локальному сховищі браузера (localStorage) та миттєво оновлює стан

глобального контексту додатка. Це призводить до динамічної зміни навігаційної панелі: замість кнопок входу з'являється персоналізоване привітання користувача та відкривається доступ до швидкого переходу на персональну сторінку «Мої замовлення» (рисунки 8-9).

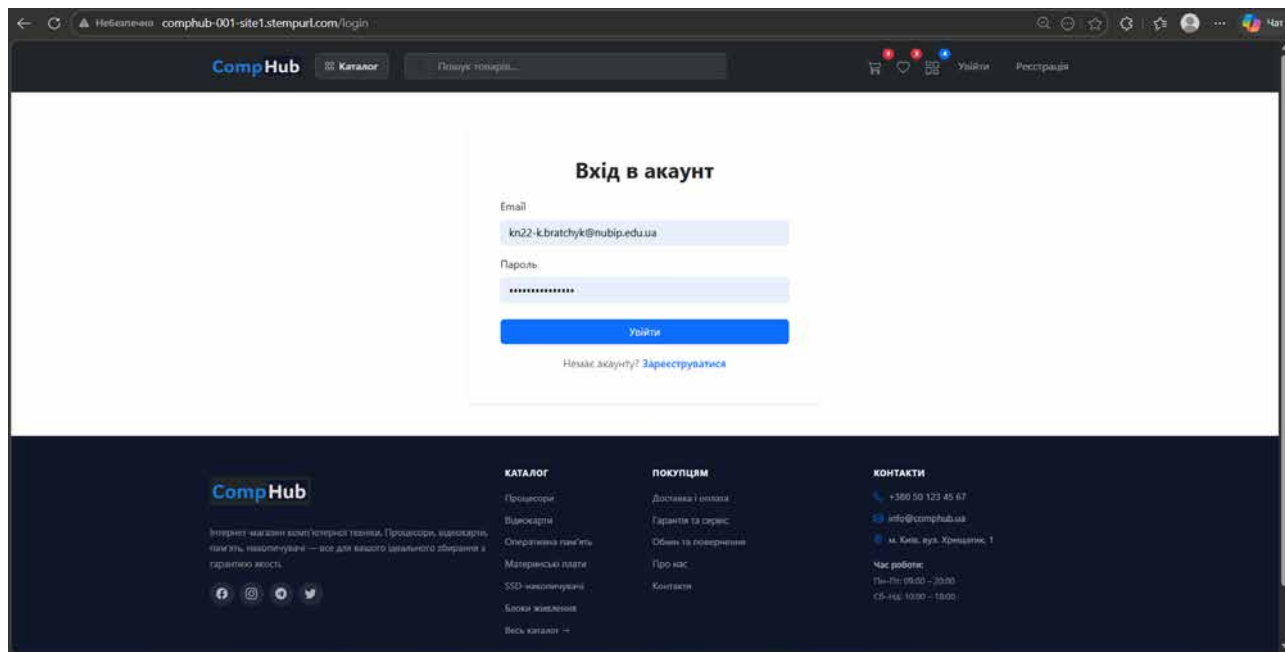


Рис. 8 - Інтерфейс сторінки авторизації

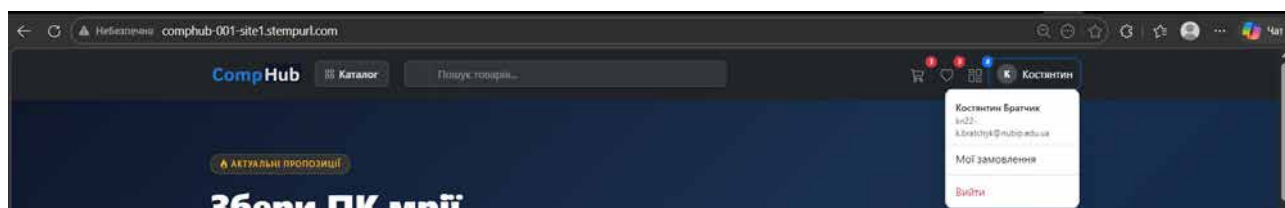


Рис. 9 - Оновлена навігаційна панель

Успішна авторизація надає клієнту доступ до приватної частини платформи – особистого кабінету покупця, який у розробленій системі представлений спеціалізованою сторінкою «Мої замовлення». Тестування цього модуля передбачало перевірку коректності відображення даних про покупки, які завантажуються виключно через захищені маршрути (Protected Routes) прикладного інтерфейсу на основі переданого в заголовках HTTP-запиту JWT-токена.

Під час перевірки було практично підтверджено, що клієнтський додаток успішно взаємодіє з сервером, отримує масив даних та динамічно формує структуровану таблицю з інформацією про всі попередні та поточні покупки

користувача. Для кожного замовлення в інтерфейсі коректно відображається його унікальний ідентифікатор, дата створення, загальна підсумкова вартість транзакції, актуальний статус виконання (наприклад, «В обробці», «Очікує відправки», «Виконано»), а також деталізований перелік придбаних товарів. Така поведінка інтерфейсу свідчить про надійну інтеграцію між фронтенд-частиною, механізмами авторизації та базою даних [15]. Інтерфейс сторінки особистого кабінету з історією замовлень представлено на рисунку 10.

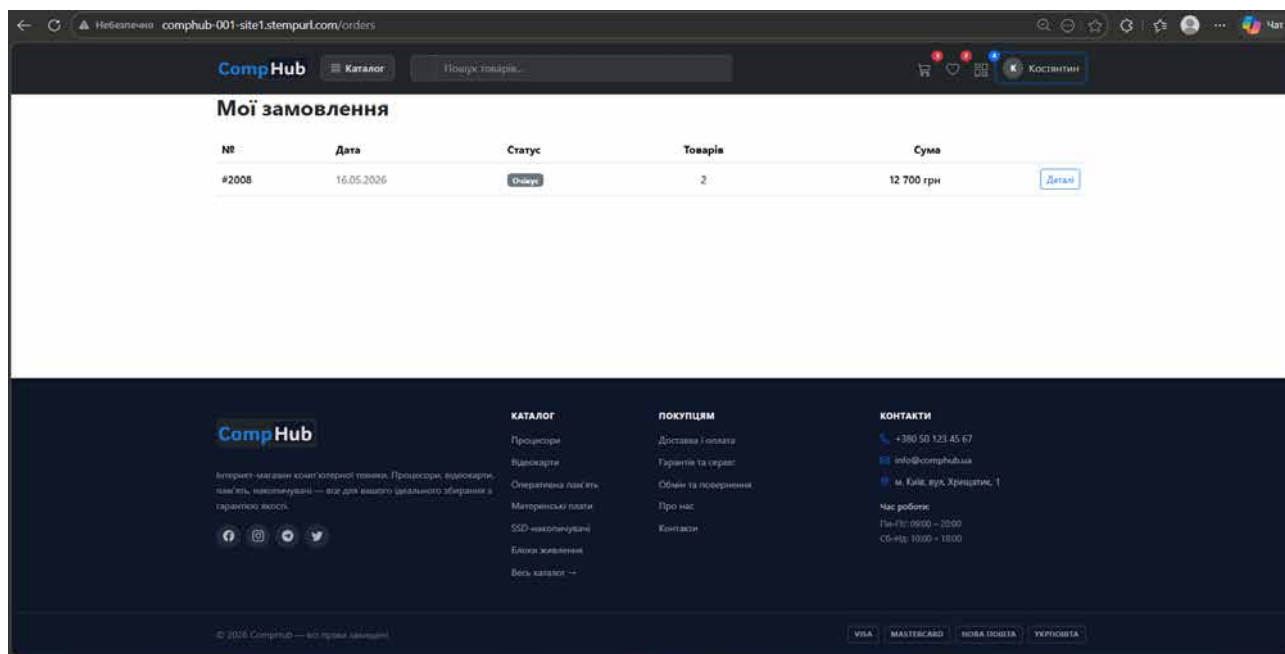


Рис. 10 – Інтерфейс сторінки «Мої замовлення» особистого кабінету користувача

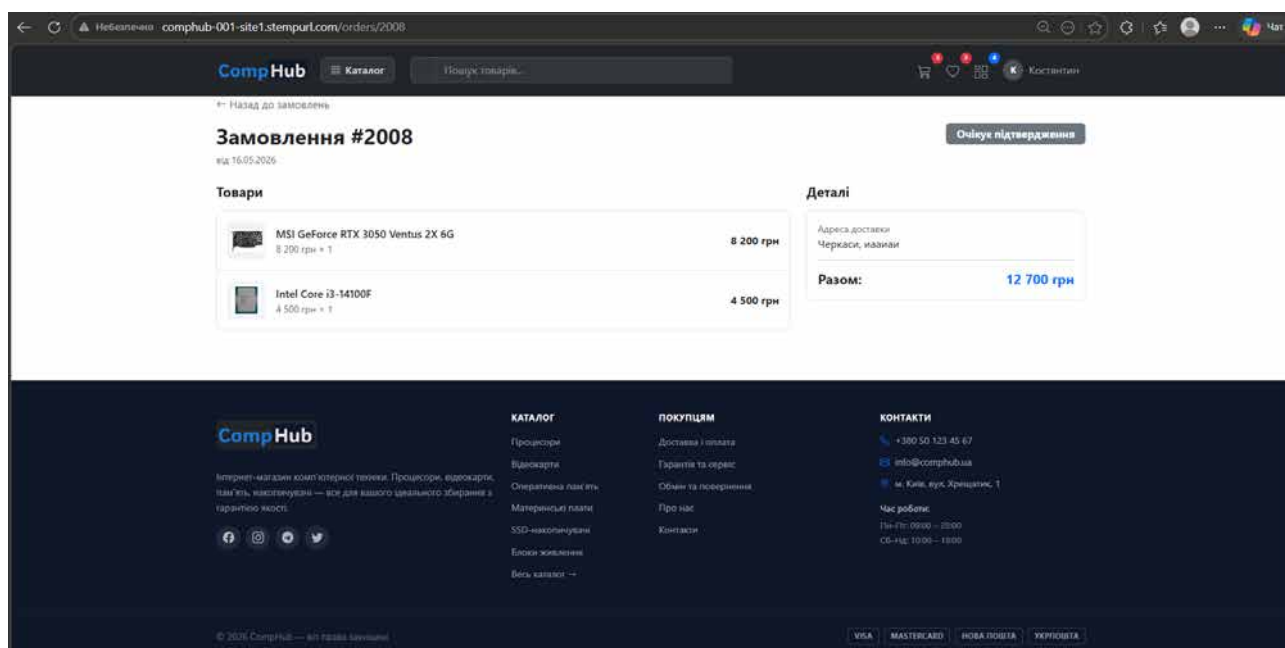


Рис. 11 – Інтерфейс сторінки «Мої замовлення. Деталі»

Таким чином, результати проведеного тестування підтверджують, що модуль автентифікації, реєстрації та перегляду замовлень функціонує згідно із закладеними вимогами, забезпечуючи високий рівень безпеки персональних даних та стабільну роботу користувацького інтерфейсу при різних сценаріях взаємодії.

4.1.4.2. Валідація роботи динамічного каталогу, пошукової системи та багатокритеріальної фільтрації

Основним робочим простором інтернет-магазину «CompHub», де клієнт проводить найбільше часу, є сторінка динамічного каталогу товарів. Саме тому валідація механізмів пошуку, сортування та багатокритеріальної фільтрації є одним із найважливіших етапів тестування користувацького інтерфейсу [15]. З технічної точки зору, цей модуль виступає одним із найскладніших компонентів фронтенд-додатка, оскільки він вимагає постійної синхронізації локального стану компонентів React із серверною базою даних через REST API.

Тестування пошукової системи розпочалося з перевірки глобального поля вводу, розташованого у верхній навігаційній панелі. Головною вимогою до цього елемента була здатність обробляти запити за частковим збігом (наприклад, пошук за фрагментом назви моделі процесора чи відеокарти). У процесі верифікації було підтверджено коректну роботу механізму затримки відправки запиту (Debouncing). Це означає, що клієнтський додаток не відправляє HTTP-запит до сервера при кожному натисканні клавіші, а очікує завершення вводу користувачем, що значно знижує навантаження на бекенд та базу даних MSSQL [14]. Результати пошуку миттєво рендеряться у вигляді інтерактивного списку. Візуальне відображення роботи пошукового модуля представлено на рисунку 4.4.

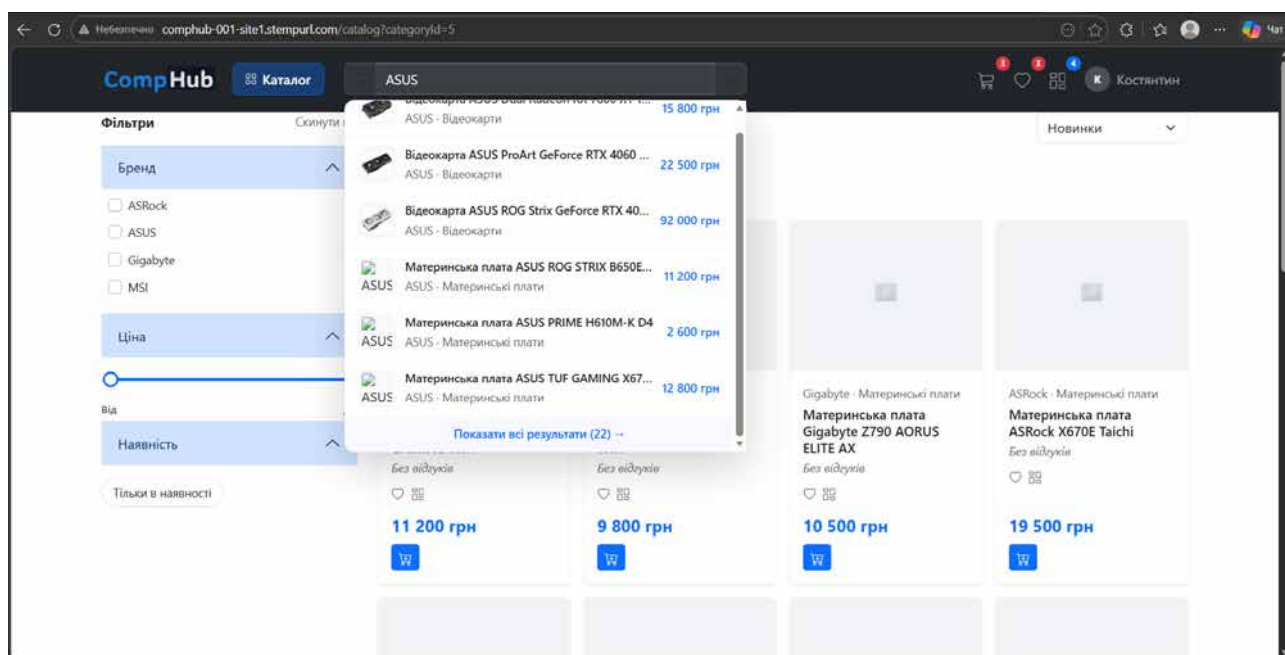


Рис. 11 - Демонстрація роботи механізму глобального пошуку товарів

Наступним етапом стала комплексна перевірка бічної панелі фільтрації. Інтерфейс фільтрів побудовано з використанням різноманітних елементів управління: чекбоксів для вибору виробників (брендів), повзунків (Range Sliders) для встановлення цінового діапазону та списків для специфічних технічних характеристик. Тестування підтвердило, що при виборі кількох параметрів одночасно (наприклад, ноутбуки бренду «Asus» з об'ємом оперативної пам'яті 16 ГБ та в ціновому діапазоні до 40000 грн) додаток коректно формує складний рядок запиту (Query String) і передає його до контролера каталогу [16].

Важливою особливістю, яку було успішно верифіковано, є динамічне оновлення URL-адреси в браузері при зміні параметрів фільтрації. Це дозволяє користувачам зберігати посилання на конкретну вибірку товарів або ділитися ним з іншими. Сервер, своєю чергою, успішно обробляє такі запити за допомогою технології Entity Framework Core і повертає відфільтрований масив у форматі JSON [12].

Окрім фільтрації, тестуванню підлягали механізми сортування (за зростанням або спаданням ціни, за новизною) та пагінації (посторінкової навігації). Перевірка довела, що при переході між сторінками каталогу список товарів оновлюється без повного перезавантаження сторінки веб-браузера

завдяки реактивній природі фреймворку React. Це забезпечує максимальну плавність роботи інтерфейсу та позитивний користувацький досвід. Інтерфейс сторінки каталогу з активованими фільтрами показано на рисунку 4.5.

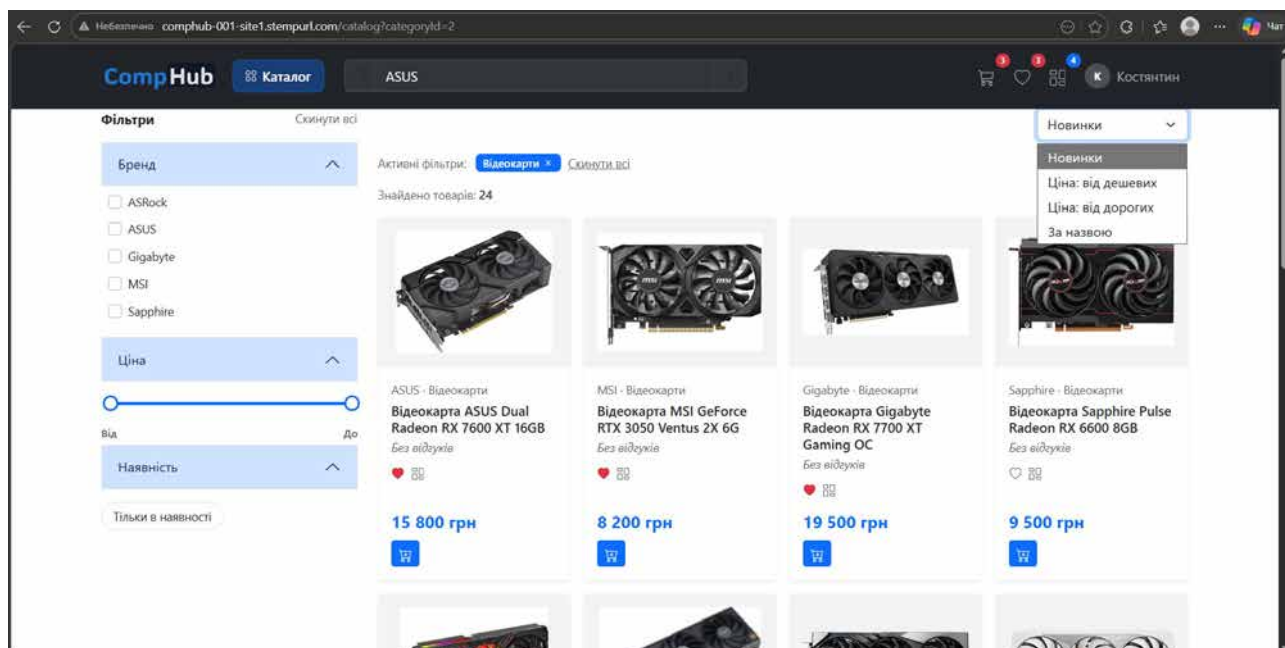


Рис. 12 - Інтерфейс динамічного каталогу з активованими критеріями фільтрації

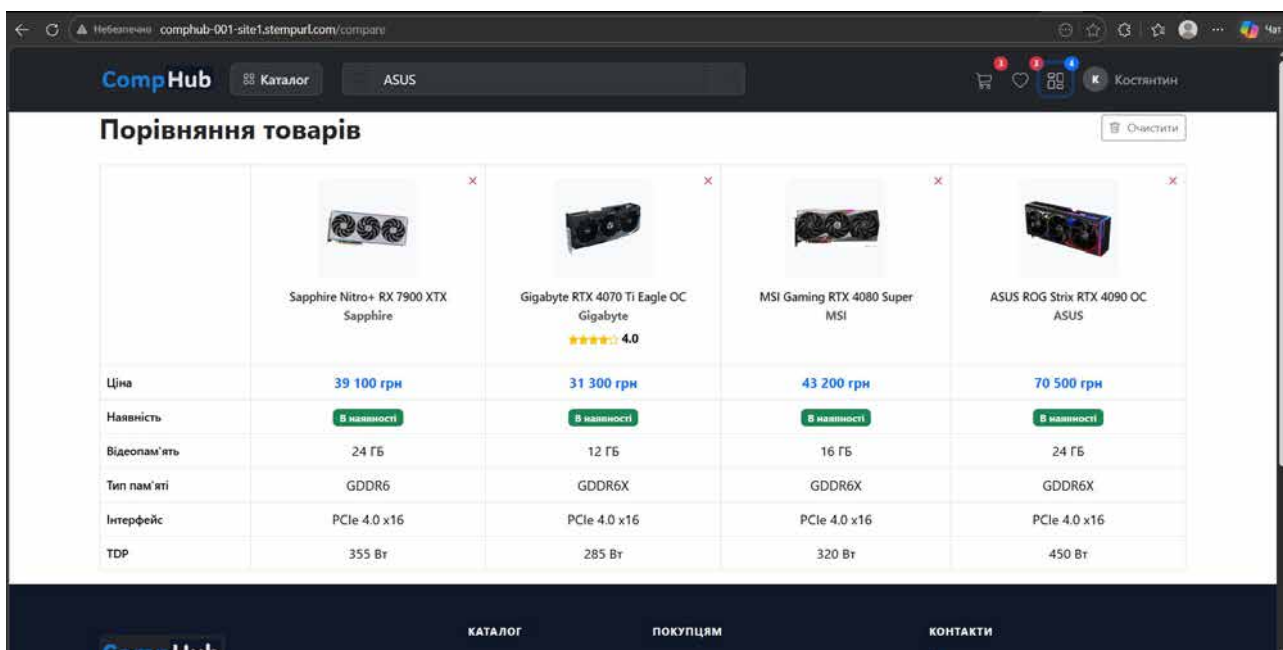
Усі описані алгоритми відпрацювали без помилок, а час відмальовування оновленого списку товарів (модифікації DOM-дерева) повністю відповідає нормам продуктивності, які були встановлені на початковому етапі тестування системи.

4.1.4.3. Верифікація роботи кошика, аналітичних інструментів та процесу оформлення замовлення

Логічним завершенням шляху клієнта (Customer Journey) на платформі електронної комерції є етап вибору конкретних товарів, додавання їх до кошика та безпосереднє оформлення замовлення. У цьому підрозділі описано результати тестування цих критично важливих бізнес-процесів, а також перевірку допоміжних інструментів персоналізації – списку бажаного (Wishlist) та модуля порівняння технічних характеристик [15].

Тестування аналітичних інструментів підтвердило їхню стабільну та безперебійну роботу. При натисканні на кнопку «Додати в улюблене» на картці пристрою додаток миттєво відправляє POST-запит до серверного API та

візуально змінює стан відповідної іконки, сигналізуючи про успішне збереження зв'язку в базі даних. Аналогічно імплементовано механізм порівняння: користувач може обрати декілька позицій з каталогу, після чого клієнтський додаток генерує зручну зведену таблицю. У цій таблиці технічні специфікації (процесор, об'єм оперативної пам'яті, діагональ екрана тощо) строго вирівняні по рядках для зручного візуального аналізу відмінностей між пристроями [14].



	Sapphire Nitro+ RX 7900 XTX Sapphire	Gigabyte RTX 4070 Ti Eagle OC Gigabyte ★★★★☆ 4.0	MSI Gaming RTX 4080 Super MSI	ASUS ROG Strix RTX 4090 OC ASUS
Ціна	39 100 грн	31 300 грн	43 200 грн	70 500 грн
Наявність	В наявності	В наявності	В наявності	В наявності
Відеопам'ять	24 ГБ	12 ГБ	16 ГБ	24 ГБ
Тип пам'яті	GDDR6	GDDR6X	GDDR6X	GDDR6X
Інтерфейс	PCIe 4.0 x16	PCIe 4.0 x16	PCIe 4.0 x16	PCIe 4.0 x16
TDP	355 Вт	285 Вт	320 Вт	450 Вт

Рис. 13 - Інтерфейс модуля порівняння технічних характеристик товарів

Наступним кроком стала комплексна верифікація функціоналу електронного кошика. Під час тестування перевірялася здатність інтерфейсу коректно обробляти додавання нових позицій, зміну їхньої кількості (інкремент/декремент) та повне видалення. Було практично доведено, що завдяки використанню інструментів глобального управління станом у React, загальна вартість товарів у кошику перераховується абсолютно динамічно і миттєво при будь-якій зміні кількості, без необхідності перезавантаження веб-сторінки [16]. Крім того, інтерфейс кошика коректно обробляє ситуацію його повного очищення, виводячи відповідне інформаційне повідомлення та кнопку для швидкого повернення до каталогу покупок.

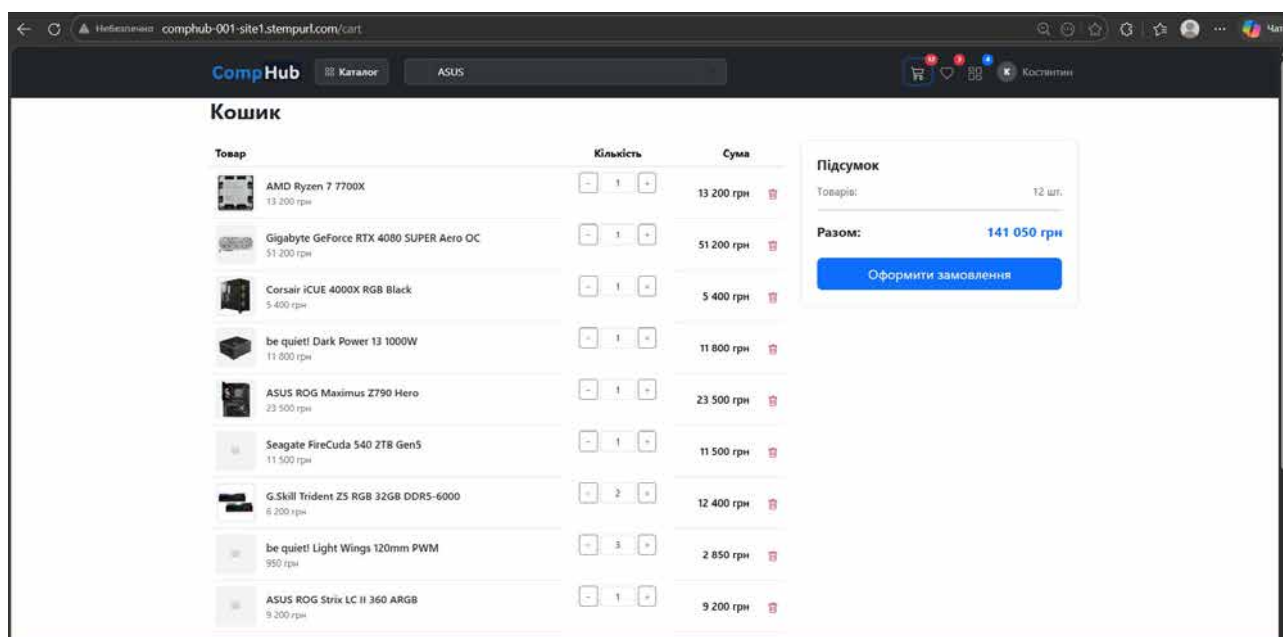


Рис. 14 - Інтерфейс електронного кошика з динамічним підрахунком вартості

Фінальним і стратегічно найважливішим етапом перевірки клієнтської частини стало тестування сторінки оформлення замовлення (Checkout). Цей інтерфейс містить комплексну форму для введення контактних даних покупця, вибору кращого способу доставки та методу оплати. Під час верифікації було підтверджено високу ефективність алгоритмів клієнтської валідації: система жорстко блокує спроби відправити форму з порожніми обов'язковими полями (наприклад, без вказання номера мобільного телефону або міста доставки) та своєчасно інформує користувача про виявлені помилки вводу за допомогою текстових підказок [11].

При введенні коректних даних та натисканні кнопки підтвердження, клієнтський додаток формує структурований JSON-об'єкт із деталями транзакції та відправляє його на захищений ендпоінт сервера. Бекенд, своєю чергою, гарантує транзакційне збереження даних у MSSQL за допомогою можливостей Entity Framework Core [12]. Після успішної обробки HTTP-запиту клієнтський інтерфейс автоматично очищує локальний стан кошика, перенаправляє покупця на сторінку успішного завершення операції та відображає унікальний ідентифікатор сформованого замовлення.

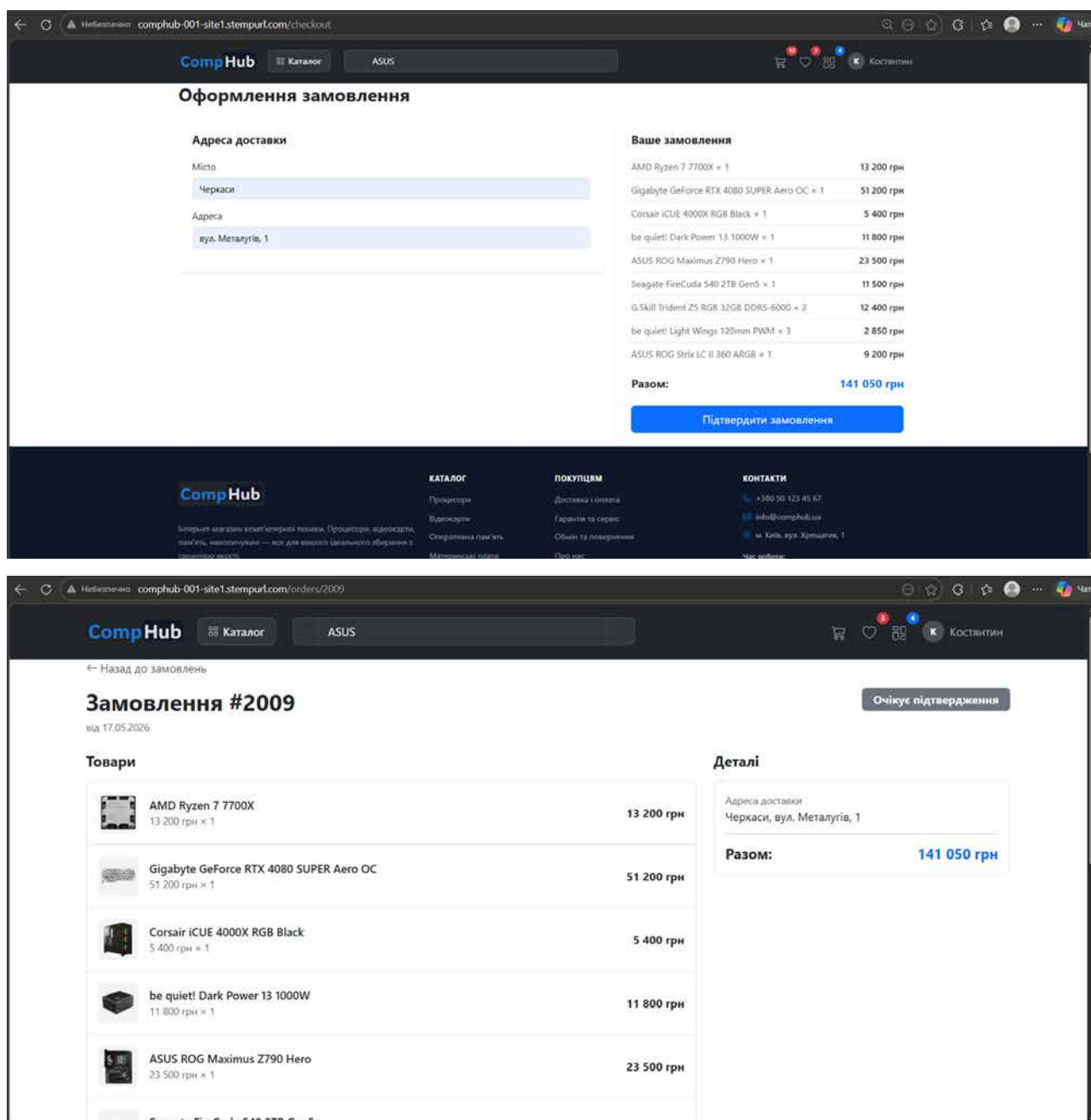


Рис. 15-16 – Інтерфейс сторінки оформлення замовлення (Checkout). А також інтерфейс сторінки тільки що сформованого замовлення

Успішне проходження всіх описаних сценаріїв тестування підтверджує, що клієнтська частина програмного комплексу «CompHub» є повністю працездатною, інтуїтивно зрозумілою та готовою до розгортання у виробничому середовищі. Наступним етапом проєкту є налаштування інфраструктури та безпосередня публікація додатку в мережі Інтернет.

4.1.4.4. Перевірка адаптивності візуальної розмітки, кросбраузерної сумісності та обробки помилок маршрутизації

Завершальним етапом тестування клієнтського додатка стала комплексна перевірка його поведінки на пристроях із різною роздільною здатністю екрана, а також тестування механізмів обробки помилок маршрутизації. Враховуючи сучасні тенденції електронної комерції, де значна частка трафіку генерується користувачами мобільних пристроїв, забезпечення адаптивності інтерфейсу (Responsive Web Design) є не просто рекомендацією, а обов'язковою вимогою до якості програмного продукту [11].

Під час верифікації адаптивності візуальної розмітки тестувалася здатність інтерфейсу динамічно перебудовувати свою структуру. Було підтверджено, що при зменшенні ширини вікна браузера (емуляція екранів планшетів та смартфонів) додаток коректно застосовує відповідні CSS-правила (Media Queries). Головне навігаційне меню трансформується у компактне випадаюче «гамбургер-меню», бічна панель фільтрів приховується та викликається за допомогою окремої кнопки у вигляді модального вікна, а багатоколонкова сітка товарів у каталозі плавно перешиковується в одну або дві колонки [15]. Такі трансформації гарантують, що елементи управління залишаються достатньо великими для взаємодії за допомогою сенсорного екрана (Touch-friendly), а текстовий контент зберігає високий рівень читабельності.

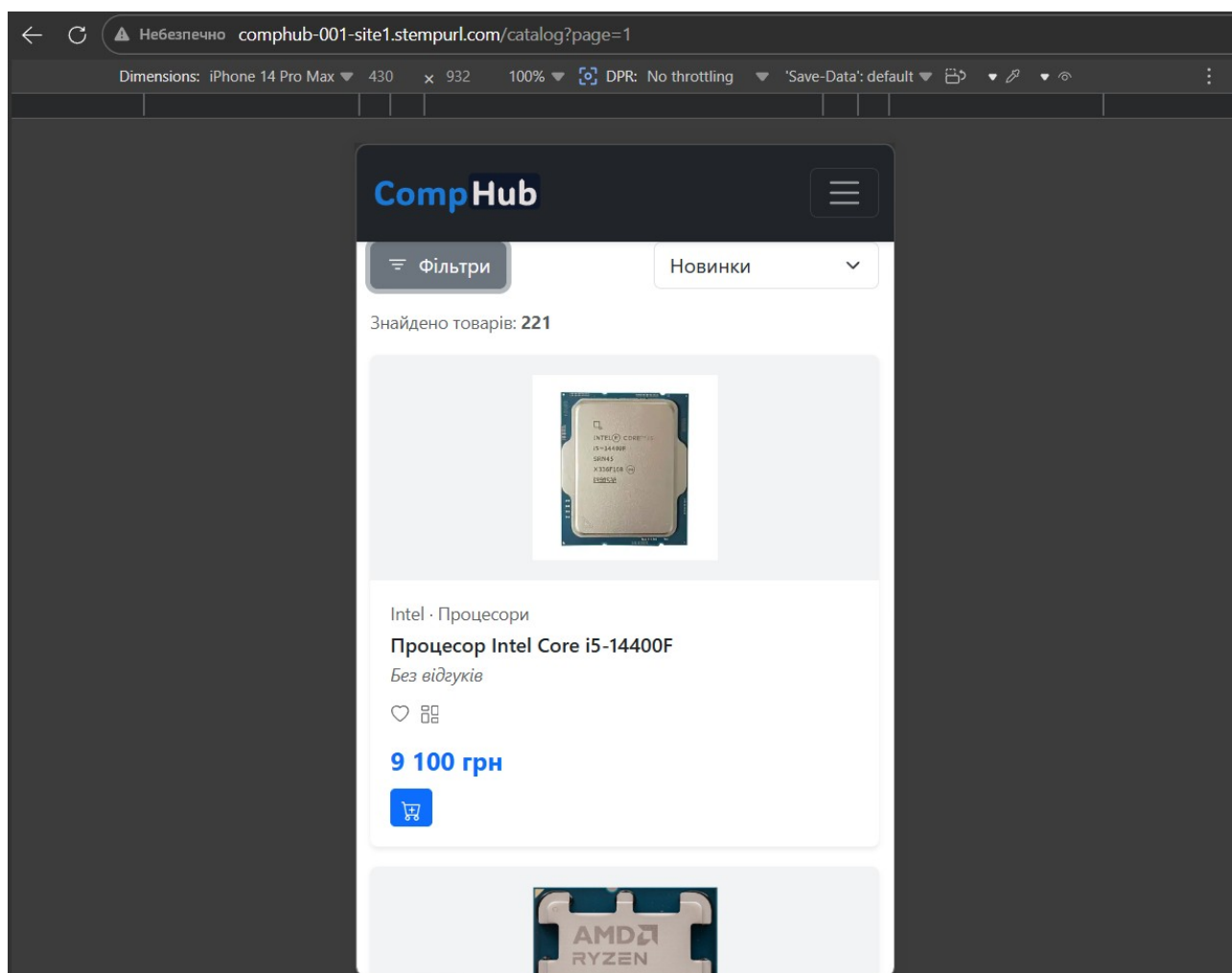


Рис. 17 – Відображення сторінки каталогу товарів на екрані мобільного пристрою. Екран 1

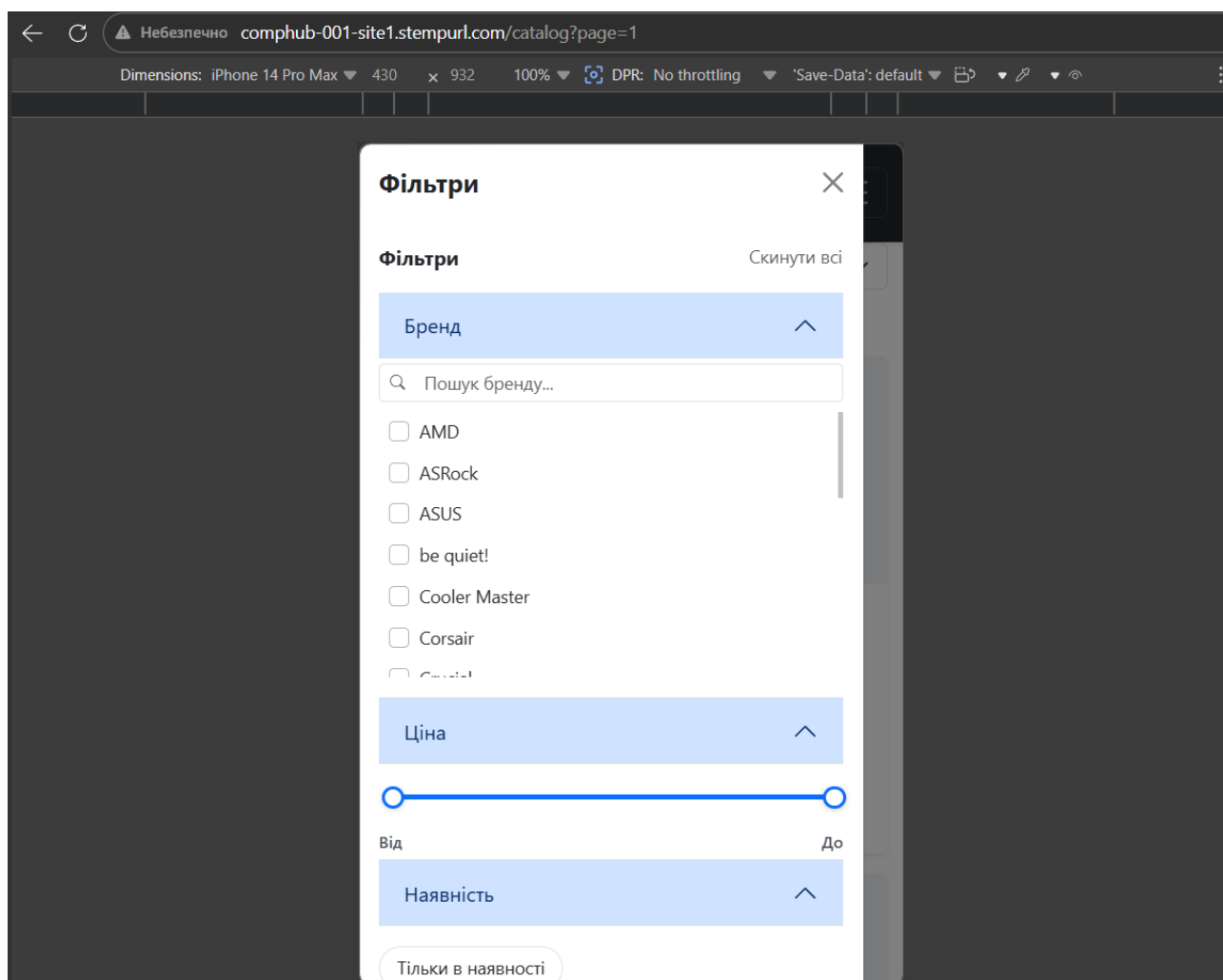


Рис. 18 – Відображення сторінки каталогу товарів на екрані мобільного пристрою. Екран 2.

Паралельно було проведено кросбраузерне тестування, мета якого полягала у підтвердженні ідентичності відображення та функціонування платформи в основних сучасних веб-браузерах (Google Chrome, Mozilla Firefox, Safari, Microsoft Edge). Завдяки використанню стандартизованих інструментів розробки та відмові від застарілих CSS-властивостей, додаток продемонстрував стабільну роботу та відсутність візуальних артефактів у всіх перелічених середовищах.

Окрему увагу було приділено тестуванню системи обробки непередбачуваних ситуацій на стороні клієнта (Error Handling), зокрема помилок маршрутизації. При спробі користувача перейти за неіснуючим URL-посиланням або звернутися до товару, який було видалено з бази даних, механізми клієнтського маршрутизатора (React Router) коректно перехоплюють цю подію.

Замість «збою» всього додатка або відображення порожнього екрана, користувачу демонструється спеціально спроектована сторінка помилки 404 (Not Found). Ця сторінка містить відповідне візуальне повідомлення та інтерактивні елементи для швидкого повернення на головну сторінку або до каталогу товарів [14].

Окрім сторінки 404, було протестовано глобальні запобіжники помилок (Error Boundaries) у React. Вони гарантують, що у разі виникнення критичної помилки рендерингу в одному з ізольованих компонентів (наприклад, через отримання пошкоджених даних від API), решта інтерфейсу продовжує функціонувати, а користувач отримує дружнє сповіщення про тимчасові технічні труднощі.

Успішне завершення всіх етапів тестування інтерфейсу (описаних у підрозділах 4.1.4.1 – 4.1.4.4) повністю підтверджує готовність клієнтського додатка до експлуатації. Комплекс перевірок довів, що фронтенд-частина «CompHub» є ергономічною, стійкою до некоректних дій та відповідає всім критеріям якості, встановленим на початку етапу валідації.

4.2. Вимоги до апаратного та програмного забезпечення

Після успішного завершення етапів локальної розробки та всебічного тестування (див. підрозділ 4.1), наступним критичним кроком життєвого циклу програмного забезпечення є його впровадження (Deployment) у виробниче середовище (Production). Головною метою цього етапу є перенесення клієнтських, серверних та баз даних компонентів інтернет-магазину «CompHub» на віддалені сервери для забезпечення цілодобового та безпечного доступу цільової аудиторії до функціоналу платформи [15].

Враховуючи обраний технологічний стек, ядром якого виступає кросплатформний фреймворк .NET 10 та реляційна система управління базами даних Microsoft SQL Server, для розгортання було обрано хмарний VPS-хостинг (Virtual Private Server) під управлінням операційної системи Windows Server. Вибір екосистеми Microsoft на стороні сервера гарантує максимальну сумісність

компонентів, високу продуктивність та зручність адміністрування інфраструктури через веб-сервер Internet Information Services (IIS) [18].

Для забезпечення стабільної роботи розподіленої системи, швидкого оброблення запитів до API та плавного рендерингу клієнтського додатка, було орендовано віртуальний виділений сервер із наступними апаратними характеристиками:

- **Процесор (vCPU):** 4 віртуальних ядра з базовою тактовою частотою від 2.4 ГГц для ефективного паралельного виконання потоків обробки HTTP-запитів.
- **Оперативна пам'ять (RAM):** 8 ГБ, що є оптимальним об'ємом для одночасного функціонування пулу додатків IIS та кешування складних запитів реляційною базою даних.
- **Дискова підсистема:** 100 ГБ на базі накопичувачів NVMe SSD. Висока швидкість операцій вводу/виводу (I/O) є критично важливою для роботи MSSQL та швидкої віддачі статичних файлів (зображень товарів).
- **Мережа:** Пропускна здатність каналу 1 Гбіт/с для уникнення вузьких місць під час пікових навантажень.

Безпосередній процес розгортання програмного комплексу було розділено на три логічні та послідовні етапи:

1. **Міграція бази даних.** За допомогою інструментарію Entity Framework Core (підхід Code-First) було згенеровано фінальний SQL-скрипт ініціалізації. Цей скрипт було виконано на віддаленому екземплярі MSSQL Server, що дозволило відтворити повну структуру реляційних таблиць, індексів та зв'язків. Після цього базу даних було заповнено початковою інформацією (Seed Data), включаючи актуальний каталог товарів, категорії техніки та системний обліковий запис адміністратора [12].
2. **Публікація серверної частини (REST API).** Вихідний код бекенду було скомпільовано у конфігурації Release для максимальної оптимізації продуктивності. Отримані бінарні файли (DLL) було перенесено на сервер. В IIS було створено новий пул додатків (Application Pool) без керованого

коду (No Managed Code), оскільки .NET працює через вбудований кросплатформний модуль Kestrel, використовуючи IIS як зворотний проксі-сервер [20]. Також було налаштовано системні змінні середовища: безпечні рядки підключення до БД (Connection Strings) та секретні криптографічні ключі для генерації JWT-токенів.

3. **Розгортання клієнтського React-додатка.** Фронтенд-частину було зібрано за допомогою спеціальної команди білда (npm run build). Сучасні JavaScript-бандлери автоматично мінімізували вихідний код, видалили зайві коментарі та оптимізували статичні ресурси. Згенеровану папку зі статичними файлами було розміщено у відповідній директорії веб-сервера. Окремо було налаштовано правила переадресації URL-адрес (URL Rewrite Module) в IIS: усі запити до неіснуючих фізичних файлів автоматично перенаправляються на головний файл index.html. Це обов'язкова умова для коректної роботи клієнтської маршрутизації (React Router) в односторінкових додатках (SPA) [14].

Фінальним кроком впровадження стало прив'язування зареєстрованого доменного імені до статичної IP-адреси сервера. Крім того, було успішно налаштовано SSL-сертифікат (протокол HTTPS) для забезпечення надійного криптографічного шифрування трафіку між веб-браузером користувача та сервером. Це є невіддільною вимогою безпеки для будь-якої комерційної платформи, що обробляє персональні дані покупців [15]. Після виконання всіх перелічених етапів розроблений інтернет-магазин «CompHub» було введено у промислову експлуатацію.

4.2.1 Налаштування серверного середовища та міграція бази даних

Підготовка інфраструктури до запуску інтернет-магазину «CompHub» розпочалася з базової конфігурації операційної системи Windows Server та встановлення необхідних програмних компонентів для хостингу додатків. Враховуючи те, що бекенд-частина реалізована на базі новітньої версії платформи .NET 10, першочерговим завданням стала інсталяція

спеціалізованого пакету .NET Hosting Bundle [20]. Цей модуль є обов'язковим для розгортання, оскільки він містить середовище виконання .NET Runtime та забезпечує інтеграцію вбудованого кросплатформного веб-сервера Kestrel із системним веб-сервером Internet Information Services (IIS). У такій конфігурації IIS виконує роль потужного зворотного проксі-сервера (Reverse Proxy), що бере на себе управління життєвим циклом додатку, фільтрацію HTTP-запитів та забезпечення базового рівня безпеки [18].

Після налаштування веб-сервера наступним етапом стало розгортання системи управління реляційними базами даних. На орендованому VPS-сервері було інстальовано Microsoft SQL Server. Процес перенесення архітектури даних з локального середовища розробника до виробничого середовища (Production) здійснювався за допомогою інструментарію об'єктно-реляційного відображення Entity Framework Core з використанням методології Code-First [12].

Процедура міграції бази даних складалася з наступних кроків:

1. За допомогою інтерфейсу командного рядка (CLI) платформи .NET було згенеровано комплексний ідемпотентний SQL-скрипт. Цей скрипт містив усі необхідні інструкції мовою T-SQL для створення структури бази даних з нуля.
2. Згенерований скрипт було виконано на серверному екземплярі MSSQL. У результаті було автоматично створено всі необхідні таблиці (користувачі, товари, замовлення, категорії), встановлено первинні та зовнішні ключі (Foreign Keys) для забезпечення цілісності даних, а також створено оптимізовані індекси для прискорення пошукових запитів.
3. Відбулося автоматичне виконання механізму «Data Seeding» (наповнення початковими даними). База даних була заповнена базовим списком категорій, початковим каталогом товарів для демонстрації працездатності платформи, а також системним обліковим записом адміністратора.

Особлива увага на цьому етапі приділялася питанням кібербезпеки та захисту конфіденційної інформації [14]. Для забезпечення ізоляції середовищ усі

чутливі параметри конфігурації були винесені в окремий файл `appsettings.Production.json`. До цього файлу було додано:

- Строгий рядок підключення до виробничої бази даних (Connection String) з використанням захищених облікових даних SQL Server.
- Секретні криптографічні ключі достатньої довжини (набори символів) для алгоритмів підпису та верифікації JWT-токенів автентифікації.
- Налаштування політики CORS (Cross-Origin Resource Sharing), яка дозволяє приймати запити до API виключно з домену, на якому розміщено клієнтський додаток, блокуючи сторонні звернення [15].

Доступ до конфігураційних файлів на сервері було жорстко обмежено за допомогою політик безпеки файлової системи NTFS – права на читання отримав лише системний користувач, від імені якого запускається пул додатків IIS. Така конфігурація серверного середовища гарантує високу продуктивність обробки даних та створює надійний фундамент для подальшої публікації клієнтського додатка.

4.2.2. Публікація клієнтського веб-додатка та фінальне налаштування інфраструктури

Після успішної конфігурації серверного середовища та міграції бази даних (див. підрозділ 4.2.1), наступним етапом стало розгортання фронтенд-частини інтернет-магазину. Клієнтський додаток, розроблений з використанням бібліотеки React, вимагає попередньої підготовки перед публікацією у виробничому середовищі. За допомогою інструментів екосистеми Node.js було ініційовано процес створення виробничої збірки проєкту (через команду `npm run build`). Цей процес автоматично виконує транскompіляцію сучасного JavaScript-коду, мініфікацію файлів стилів (CSS) та скриптів, видалення технічних коментарів, а також оптимізацію статичних ресурсів [14]. Результатом збирання є компактний набір статичних файлів, максимально оптимізованих для швидкого завантаження та виконання у веб-браузері кінцевого споживача.

Згенеровану директорію зі статичними файлами було перенесено та розміщено у відповідному каталозі веб-сервера Internet Information Services (IIS). Оскільки графічний інтерфейс побудовано за архітектурою односторінкового додатка (Single Page Application, SPA), де маршрутизація та зміна відображення сторінок обробляється виключно на стороні клієнта модулем React Router, виникла необхідність специфічного налаштування правил обробки запитів на сервері [15].

У стандартній конфігурації при прямому зверненні користувача за специфічним посиланням (наприклад, до сторінки `comphub.com/cart`), веб-сервер намагатиметься знайти фізичний файл або папку `cart`, і, не знайшовши їх, поверне помилку 404 (Not Found). Для розв'язання цієї проблеми в IIS було активовано та налаштовано розширення URL Rewrite Module [18]. Створене правило переадресації вказує серверу: якщо запитуваний ресурс не є фізичним файлом (наприклад, зображенням) чи існуючою директорією, такий запит необхідно перенаправити на кореневий файл `index.html`. Завдяки цьому клієнтський маршрутизатор перехоплює управління та коректно відмальовує відповідний компонент інтерфейсу.

Фінальним етапом розгортання стало забезпечення захисту інформації та налаштування безпечного доступу до платформи. До статичної IP-адреси VPS-сервера було прив'язано зареєстроване доменне ім'я. Для захисту персональних даних покупців під час їхньої передачі через відкриті мережі (зокрема на етапах реєстрації, автентифікації та оформлення замовлення), було згенеровано та встановлено криптографічний SSL-сертифікат [11].

Крім того, на рівні веб-сервера було налаштовано політику примусового перенаправлення всього нешифрованого трафіку (HTTP) на безпечний протокол (HTTPS). Забезпечення наскрізного шифрування не лише відповідає сучасним стандартам кібербезпеки та вимогам до платформ електронної комерції, але й позитивно впливає на довіру користувачів та ранжування інтернет-магазину в пошукових системах [14]. Після успішного виконання та валідації всіх

перелічених налаштувань програмний комплекс «CompHub» було остаточно переведено у статус промислової експлуатації.

4.2.3. Післяексплуатаційна перевірка та моніторинг продуктивності програмного комплексу

Після успішного розгортання клієнтської та серверної частин інтернет-магазину «CompHub» у виробничому середовищі, критично важливим етапом стало проведення післяексплуатаційної перевірки (Post-deployment verification) та так званого «димового тестування» (Smoke Testing). Головна мета цього процесу полягала у підтвердженні того, що всі модулі системи коректно функціонують в умовах реального інфраструктурного оточення, а інтеграція між фронтом, REST API та базою даних працює без збоїв [15].

У рамках післяексплуатаційної перевірки було успішно відтворено базовий клієнтський сценарій безпосередньо на «живому» домені: реєстрація нового користувача, пошук товару в каталозі, додавання його до кошика та фінальне оформлення замовлення. Це дозволило практично підтвердити, що рядки підключення до виробничої бази даних (Connection Strings) сконфігуровані правильно, а поштовий сервіс для відправки транзакційних листів інтегровано коректно.

Для об'єктивної оцінки швидкодії та оптимізації клієнтського додатка було використано стандартизований інструментарій Google Lighthouse, вбудований у панель розробника сучасних веб-браузерів. Проводився аудит ключових метрик Web Vitals, що безпосередньо впливають на користувацький досвід та ранжування сайту в пошукових системах [11]:

- **Largest Contentful Paint (LCP):** Час відмальовування найбільшого видимого елемента на сторінці (наприклад, банера або головного зображення товару) склав менше ніж 2.5 секунди, що є відмінним показником. Це стало можливим завдяки мініфікації статичних ресурсів під час білда React-додатка.

- **Cumulative Layout Shift (CLS):** Візуальна стабільність інтерфейсу під час завантаження продемонструвала нульове значення зсувів, оскільки розміри зображень та блоків були жорстко зафіксовані у CSS.
- **First Input Delay (FID):** Затримка перед першою можливою взаємодією користувача з інтерфейсом була мінімальною, що підтверджує високу оптимізацію JavaScript-бандлів [14].

На стороні сервера було налаштовано базовий моніторинг працездатності. Завдяки вбудованій системі логування платформи .NET 10 (інтерфейс ILogger), налаштовано запис інформації про всі критичні помилки, попередження та тривалість виконання складних SQL-запитів до локального текстового файлу на сервері [20]. Це дозволяє адміністраторам системи оперативно виявляти та усувати потенційні проблеми (наприклад, помилки зі статусом 500 Internal Server Error), не перериваючи роботу користувачів. Також адміністративна панель веб-сервера IIS використовується для відстеження загального навантаження на процесор (CPU) та використання оперативної пам'яті (RAM) пулом додатків.

Окремий акцент під час перевірки було зроблено на валідації налаштувань безпеки у виробничому середовищі [15]. Було проведено аудит SSL-сертифіката для підтвердження того, що весь трафік передається виключно через захищений протокол HTTPS. Крім того, перевірено роботу політик CORS – було доведено, що серверний API приймає запити тільки від офіційного домену клієнтського додатка, ефективно відхиляючи будь-які спроби несанкціонованих міждоменних запитів від сторонніх джерел.

У підсумку, результати комплексного тестування, успішне розгортання та позитивні метрики післяексплуатаційного моніторингу підтверджують, що розроблений інтернет-магазин «CompHub» є повністю працездатним, відмовостійким та готовим до повноцінної комерційної експлуатації кінцевими споживачами. Це успішно завершує етап практичної реалізації та впровадження програмного комплексу.

4.2.4. Супровід, резервне копіювання даних та стратегія подальшого розвитку

Після успішного введення інтернет-магазину «CompHub» у промислову експлуатацію та налаштування базового післяексплуатаційного моніторингу (див. підрозділ 4.2.3), невіддільним етапом життєвого циклу програмного продукту є організація процесів технічного супроводу, забезпечення надійності збереження даних та планування стратегії масштабування [15].

Критично важливим аспектом підтримки будь-якої комерційної платформи є налаштування автоматизованого резервного копіювання (Backup) бази даних. Враховуючи використання Microsoft SQL Server, для забезпечення цілісності та збереження транзакційної історії покупок було налаштовано регулярне створення повних (Full) та диференційних (Differential) копій бази даних [12]. Цей процес автоматизовано за допомогою планувальника завдань ОС Windows Server та SQL-скриптів. Резервні копії архівуються та безпечно передаються на ізольоване хмарне сховище, що гарантує можливість швидкого відновлення працездатності (Disaster Recovery) у разі критичних апаратних збоїв на основному сервері.

Для оптимізації процесу випуску майбутніх оновлень програмного забезпечення розпочато впровадження методології безперервної інтеграції та безперервного розгортання (CI/CD). Використання сучасних DevOps-практик дозволить автоматизувати процеси компіляції коду .NET-бекенду, проходження автоматизованих тестів та мініфікації вихідних файлів React-додатка [14]. Це зведе до мінімуму людський фактор при оновленні системи та дозволить розгортати нові модулі (наприклад, інтеграцію з платіжними шлюзами) без відчутного простою (Downtime) веб-сервера IIS [18].

Стратегія подальшого розвитку проєкту «CompHub» враховує потенційне зростання навантаження на систему. Архітектура серверного API, розроблена на базі .NET 10, є «stateless» (не зберігає стан сесії безпосередньо в оперативній пам'яті сервера завдяки використанню JWT-токенів). Така архітектурна особливість дозволяє в майбутньому легко застосувати горизонтальне

масштабування – розгорнути декілька екземплярів бекенду, розподіляючи трафік між ними за допомогою балансувальника навантаження (Load Balancer) [20]. З боку клієнтського інтерфейсу розглядається можливість поступового переходу до технології Server-Side Rendering (SSR) для покращення показників пошукової оптимізації (SEO) та прискорення першого відмальовування контенту (FCP) [11].

Крім того, для забезпечення якісного технічного супроводу було сформовано актуальну документацію проєкту. Структура реляційної бази даних описана за допомогою ER-діаграм, а всі кінцеві точки (Endpoints) REST API мають живу інтерактивну документацію завдяки інтеграції стандарту OpenAPI (Swagger) [16]. Це значно спрощує процес взаємодії фронтенд- та бекенд-частин при додаванні нового функціоналу.

У підсумку, закладені на етапі проєктування архітектурні патерни, налаштована виробнича інфраструктура та розроблений план резервування даних гарантують високу надійність, кібербезпеку та готовність програмного комплексу «CompHub» до довгострокової комерційної експлуатації та масштабування.

4.3. Практична реалізація розгортання та супровід системи

Після детального аналізу, налаштування пулу додатків IIS та підготовки інфраструктури промислового середовища на базі ОС Windows Server (що було розглянуто у підрозділі 4.2), логічним завершенням циклу створення програмного продукту є його безпосереднє розгортання (деплой) на віддаленому хостингу. Цей процес вимагає чітко спланованої послідовності дій для перенесення локальної версії програмного комплексу «CompHub» у виробниче середовище із забезпеченням подальшого безперебійного доступу цільової аудиторії до ресурсу [15].

Враховуючи комплексну архітектуру розробленого рішення, яке базується на кросплатформному фреймворку .NET 10, клієнтській бібліотеці React та реляційній базі даних Microsoft SQL Server 2025, стратегія

впровадження передбачає інтеграцію незалежних компонентів у єдину злагоджену систему. Першочерговим завданням на цьому етапі є безпечна міграція структури бази даних та початкового масиву інформації з локального середовища розробника на сервер СУБД, що є критично важливим кроком для збереження цілісності бізнес-даних [12].

Окрім роботи з даними, сучасні стандарти інженерії програмного забезпечення вимагають максимальної автоматизації процесів публікації коду. Використання спеціалізованих профілів публікації дозволяє оптимізувати перенесення збірки REST API на сервер, мінімізуючи людський фактор та час потенційного простою системи [18]. Водночас розгортання клієнтської частини вимагає специфічного підходу: оскільки React генерує односторінковий додаток (SPA) із власною логікою зміни сторінок (React Router), необхідно забезпечити його правильну інтеграцію в екосистему серверів Kestrel та IIS. Це передбачає не лише розміщення статичних ресурсів у відповідних каталогах, але й строге налаштування серверної маршрутизації для обробки прямих переходів за посиланнями [14][20].

У наступних підрозділах поетапно та детально розглянуто процеси практичної реалізації розгортання програмного комплексу, починаючи від міграції бази даних і завершуючи тонким налаштуванням конфігураційних файлів веб-сервера.

4.3.1. Стратегія міграції бази даних за допомогою резервного копіювання та відновлення (Backup/Restore)

Етап розгортання та перенесення даних є одним із найбільш відповідальних у процесі впровадження комерційних веб-платформ, оскільки будь-яка втрата цілісності або структури реляційних зв'язків може призвести до критичних збоїв у бізнес-логіці всього додатку [12]. Для міграції бази даних інтернет-магазину комп'ютерної техніки «CompHub» з локального середовища розробки (Development) на віддалений сервер промислової експлуатації (Production) під управлінням СУБД Microsoft SQL Server 2025 було обрано та

практично реалізовано стратегію резервного копіювання та відновлення (Backup/Restore) [15].

На відміну від автоматичної генерації та виконання ідемпотентних T-SQL скриптів (які зазвичай застосовуються для розгортання порожньої структури), підхід Backup/Restore дозволяє гарантовано перенести не лише саму схему бази даних (таблиці, типи даних, обмеження, індекси та збережені процедури), а й уже наявний масив критично важливої інформації (Seed Data). До цієї інформації належать: повна деревоподібна ієрархія категорій техніки, початковий каталог товарів із деталізованими технічними специфікаціями, заповнені характеристики брендів та системні облікові записи адміністраторів платформи. Це мінімізує ризики виникнення помилок синтаксису чи несумісності кодувань під час імпорту великих текстових блоків.

Процес практичної реалізації стратегії міграції бази даних складався з трьох послідовних та взаємопов'язаних етапів:

1. **Формування повної резервної копії (Full Backup) на локальному сервері.** За допомогою інструментарію SQL Server Management Studio (SSMS) було виконано команду Transact-SQL для створення монолітного бінарного дампа бази даних. Команда була оптимізована для забезпечення максимальної стисливості та перевірки контрольних сум:

```
BACKUP DATABASE [CompHubDB]
TO DISK = 'C:\Backups\CompHubDB_Production_Migration.bak'
WITH FORMAT,
MEDIANAME = 'CompHub_Migration',
NAME = 'Full Backup of CompHubDB for Production Deploy',
COMPRESSION,
STATS = 10;
```

Використання ключового слова COMPRESSION дозволило значно зменшити фізичний розмір вихідного файлу .bak, що критично важливо для швидкої передачі даних мережею [12].

2. **Транспортування файлу резервної копії.** Сформований файл CompHub-DB_Production_Migration.bak було перенесено на орендований Windows Server VPS. Для забезпечення конфіденційності та унеможливлення перехоплення структури комерційних даних під час передачі через відкриті канали зв'язку було використано захищений протокол SFTP з шифруванням трафіку [15]. Файл було розміщено в ізольованому технічному каталозі сервера СУБД.
3. **Процедура відновлення бази даних (Restore Database) у промисловому середовищі.** Оскільки логічна та фізична структура каталогів на комп'ютері розробника та на віддаленому сервері хостингу відрізняються, пряме відновлення було неможливим. Операція релокації виконувалася на сервері MSSQL 2025 за допомогою оператора RESTORE DATABASE із чітким перевизначенням шляхів для фізичного файлу даних (.mdf) та файлу журналу транзакцій (.ldf) через команду MOVE [18]:

```
RESTORE DATABASE [CompHubDB_Prod]
FROM DISK = 'D:\MSSQL\Uploads\CompHubDB_Production_Migration.bak'
WITH MOVE 'CompHubDB' TO 'D:\MSSQL\Data\CompHubDB_Prod.mdf',
      MOVE 'CompHubDB_log' TO 'D:\MSSQL\Log\CompHubDB_Prod.ldf',
REPLACE,
STATS = 5;
```

Після успішного відновлення файлів бази даних виникла технічна необхідність вирішення проблеми «осиротілих користувачів» (Orphaned Users). Оскільки внутрішні ідентифікатори безпеки (SID) локального користувача бази даних та користувачів промислового сервера MSSQL не збігалися, було проведено процедуру синхронізації облікових записів. На рівні сервера було створено новий ізольований Login для веб-додатка, після чого виконано мапінг за допомогою наступної конструкції:

```
USE [CompHubDB_Prod];
ALTER USER [CompHubAppUser] WITH LOGIN = [CompHubProdLo-
gin];
```

Такий підхід дозволив чітко розмежувати права доступу та дотриматися політики найменших привілеїв: обліковий запис, який використовує серверний .NET 10 API для зв'язку через Entity Framework Core, має права виключно на маніпуляцію даними (db_datareader, db_datawriter), але повністю позбавлений прав на зміну структури таблиць чи адміністрування самої СУБД [12].

На завершення міграції було виконано системну команду верифікації цілісності DBCC CHECKDB, яка підтвердила повну відсутність логічних чи фізичних пошкоджень сторінок даних після перенесення, що засвідчило успішність реалізації обраної стратегії.

4.3.2. Налаштування профілю публікації та автоматизація деплою серверного API через технологію Web Deploy

Після успішного перенесення структури та даних реляційної бази даних на промисловий сервер (див. підрозділ 4.3.1), наступним критичним кроком є розгортання бізнес-логіки системи – серверного REST API, розробленого на базі кросплатформного фреймворку .NET 10. Відмова від ручного копіювання скомпільованих бінарних файлів (DLL) на користь автоматизованих інструментів є стандартом сучасної інженерії програмного забезпечення, оскільки це мінімізує вплив людського фактора та значно скорочує час простою (Downtime) системи під час оновлень [14].

Для автоматизації процесу публікації (деплою) бекенд-частини інтернет-магазину «CompHub» на сервер веб-хостингу Internet Information Services (IIS) було обрано технологію **Web Deploy (MSDeploy)**. Це розширення від Microsoft дозволяє безпечно та ефективно упаковувати веб-додатки, бази даних та відповідні конфігурації для їхнього подальшого розгортання на віддалених серверах [18].

Процес налаштування розпочався зі створення спеціалізованого профілю публікації (файлу з розширенням .pubxml) у середовищі розробки. Цей XML-документ інкапсулює всі необхідні параметри для коректної компіляції та транспортування серверного додатка. Під час конфігурації профілю було визначено наступні ключові параметри:

- **Режим розгортання (Deployment Mode):** Обрано залежний від фреймворку режим (Framework-dependent). Оскільки на цільовому Windows-сервері попередньо було встановлено глобальний пакет .NET Hosting Bundle (як описано у підрозділі 4.2), немає потреби включати середовище виконання .NET Runtime у пакет публікації. Це дозволило зменшити розмір вихідного артефакту до кількох мегабайтів [20].
- **Цільове середовище виконання (Target Runtime):** Встановлено конфігурацію win-x64 для максимальної оптимізації роботи скомпільованого коду під архітектуру 64-розрядного процесора цільового сервера.
- **Конфігурація збірки (Build Configuration):** Встановлено значення Release, що наказує компілятору .NET виконати максимальну оптимізацію швидкодії коду, видалити налагоджувальні символи (PDB-файли) та ігнорувати тестові ділянки коду.

Особливою перевагою використання Web Deploy є його механізм диференційного оновлення (Differential Sync). Під час ініціалізації процесу деплою інструмент автоматично порівнює хеш-суми файлів на комп'ютері розробника та на віддаленому сервері IIS. Мережею передаються виключно ті файли, які були змінені або додані з моменту попередньої публікації. Це робить процес оновлення програмного комплексу практично миттєвим [15].

Для забезпечення безпеки передачі даних автентифікація Web Deploy була налаштована через службу веб-керування (Web Management Service - WMSVC) із використанням захищеного з'єднання по порту 8172. Усі облікові дані розгортання були зашифровані, а транспортний рівень захищений SSL-сертифікатом.

Крім того, профіль публікації автоматизує процес трансформації конфігураційних файлів. Під час деплою система автоматично застосовує налаштування з файлу `appsettings.Production.json`, підмінюючи локальні змінні розробника на реальні виробничі дані: актуальні криптографічні ключі для JWT-токенів, налаштування CORS та рядок підключення до промислової бази даних MSSQL 2025, міграція якої була описана в попередньому підрозділі [12].

Використання технології Web Deploy дозволило перетворити складний багатоетапний процес оновлення бекенду на автоматизовану процедуру, яка запускається однією командою. Це формує надійний фундамент для подальшого впровадження практик безперервної інтеграції та безперервного розгортання (CI/CD).

4.3.3. Особливості інтеграції клієнтського React-додатка в каталог статичних файлів `wwwroot` веб-сервера Kestrel

Після успішного розгортання серверного REST API на базі .NET 10 (що було розглянуто у підрозділі 4.3.2), наступним етапом є публікація фронтенд-частини програмного комплексу. Оскільки клієнтський інтерфейс розроблено з використанням бібліотеки React, він функціонує за принципом односторінкового додатка (Single Page Application, SPA). Технічно готовий до виробництва (Production) React-додаток являє собою оптимізований набір статичних файлів: одного файлу `index.html`, а також мініфікованих JavaScript-бандлів, CSS-стилів та оптимізованих медіаресурсів [14].

Щоб підготувати клієнтський код до розгортання, у середовищі розробки було виконано збирання проєкту за допомогою інструментарію Node.js (команда `npm run build`). Цей процес автоматично застосовує алгоритми транскompіляції, видаляє технічні коментарі та «стискає» вихідний код, що критично важливо для забезпечення високої швидкості завантаження інтерфейсу в браузері кінцевого користувача [11].

Для хостингу отриманих статичних файлів було обрано стратегію суміщеного розгортання (In-Process Hosting) у межах єдиного серверного

додатка на базі .NET. Відповідно до архітектури сучасних веб-додатків Microsoft, вбудований кросплатформний веб-сервер Kestrel за замовчуванням не віддає статичний контент із міркувань безпеки. Щоб дозволити серверу обслуговувати файли React-додатка, їх необхідно розмістити у спеціально відведений системній директорії – `wwwroot`, яка виступає публічним кореневим каталогом веб-проєкту [20].

Процес інтеграції складався з наступних технічних кроків:

1. Вміст згенерованої папки збірки React (зазвичай `build` або `dist`) було повністю скопійовано до директорії `wwwroot` серверного проєкту .NET перед його компіляцією та деплоєм.
2. У головному конфігураційному файлі бекенду (`Program.cs`) було підключено спеціальне проміжне програмне забезпечення (`Middleware`) для обробки статичних ресурсів. Виклик методу `app.UseStaticFiles()` вказує серверу Kestrel перехоплювати HTTP-запити до файлів (наприклад, `.js`, `.css`, `.png`) та шукати їх у каталозі `wwwroot` [18].
3. Для коректної обробки базового шляху додатка було зареєстровано додатковий компонент конвеєра запитів – `app.UseDefaultFiles()`, який автоматично повертає клієнту файл `index.html` при зверненні до кореневого URL домену.

Такий архітектурний підхід до інтеграції має ряд суттєвих переваг. По-перше, суміщене розгортання дозволяє клієнтській та серверній частинам працювати під одним доменним ім'ям та на одному мережевому порту. Це повністю усуває необхідність налаштування складних політик CORS (Cross-Origin Resource Sharing), оскільки всі API-запити від React-додатка до бекенду вважаються запитами з того самого джерела (Same-Origin) [15]. По-друге, це значно спрощує подальший супровід інфраструктури, адже весь програмний комплекс оновлюється та керується як єдиний монолітний артефакт.

Однак, розміщення SPA у каталозі `wwwroot` створює специфічну проблему обробки внутрішніх маршрутів клієнтського додатка при прямому переході за посиланням або оновленні сторінки. Вирішення цієї проблеми

вимагає додаткового налаштування конвеєра обробки запитів, що детально розглядається в наступному підрозділі.

ВИСНОВКИ

У кваліфікаційній роботі було успішно вирішено актуальне науково-практичне завдання, яке полягало у комплексному проектуванні, розробці, тестуванні та впровадженні у промислове середовище сучасного програмного комплексу – інтернет-магазину комп'ютерної техніки «CompHub». Розроблена система орієнтована на забезпечення повного циклу електронної комерції: від зручного пошуку товарів та їхнього порівняння до безпечного оформлення замовлення. У ході виконання роботи було досягнуто всіх поставлених на початку дослідження цілей та завдань, а отримані результати повністю відповідають сучасним інженерним стандартам веб-розробки.

За результатами виконання кваліфікаційної роботи сформульовано такі основні висновки:

1. **Проведено ґрунтовний аналіз предметної області та обґрунтовано вибір технологічного стека.** Дослідження сучасного ринку електронної комерції виявило високі вимоги користувачів до швидкодії, безпеки та інтерактивності веб-додатків. На основі порівняльного аналізу архітектурних рішень для реалізації проекту було обрано клієнт-серверну архітектуру з розділенням на незалежні Front-end та Back-end компоненти. Оптимальним технологічним базисом для серверної частини став кросплатформний фреймворк **.NET 10** у зв'язці з реляційною базою даних **Microsoft SQL Server 2025**, а для клієнтської – бібліотека **React**, що дозволило створити високопродуктивний та масштабований односторінковий додаток (SPA).
2. **Спроектовано надійну та нормалізовану структуру бази даних.** Використовуючи методології об'єктно-реляційного моделювання, було розроблено концептуальну та логічну моделі даних. Структура БД охоплює всі необхідні сутності для роботи інтернет-магазину: ієрархічний каталог товарів, категорії, характеристики, профілі користувачів, кошики та замовлення. Завдяки використанню інструментарію **Entity Framework Core** (підхід Code-First) забезпечено надійний шар абстракції для доступу

до даних, транзакційну цілісність та захист від поширених вразливостей, таких як SQL-ін'єкції.

3. **Розроблено захищений серверний програмний інтерфейс (REST API).** Реалізовано повноцінний бекенд, який обробляє бізнес-логіку додатку. Спроектовано безпечну систему автентифікації та авторизації користувачів на основі стандарту **JSON Web Token (JWT)**, що забезпечує надійний захист персональних даних покупців та дозволяє легко масштабувати систему завдяки «stateless» архітектурі. Налаштовано маршрутизацію, валідацію вхідних даних та обробку виключних ситуацій.
4. **Створено ергономічний та інтерактивний клієнтський інтерфейс.** За допомогою компонентного підходу React реалізовано графічний інтерфейс інтернет-магазину (Front-end). Інтерфейс включає модуль динамічного каталогу з багатокритеріальною системою фільтрації, персональний кабінет покупця, механізм порівняння характеристик товарів та систему глобального управління станом електронного кошика. Застосування сучасних CSS-фреймворків та медіазапитів (Media Queries) забезпечило повну адаптивність візуальної розмітки (Responsive Web Design) для коректного відображення на мобільних пристроях, планшетах та десктопних моніторах.
5. **Виконано комплексне тестування та валідацію програмного комплексу.** Експериментальна частина роботи довела високу надійність розробленого продукту. Верифікація API за допомогою інструментів Postman та Swagger підтвердила правильність роботи серверної логіки. Тестування користувацьких сценаріїв (End-to-End) на стороні клієнта довело стабільність роботи процесів реєстрації, управління кошиком та оформлення замовлення (Checkout). Механізми обробки помилок маршрутизації (React Router) та кросбраузерне тестування підтвердили безперебійність роботи інтерфейсу.
6. **Реалізовано розгортання системи у промисловому середовищі (Pro-**

duction). Практично вирішено складне інженерне завдання інтеграції різних компонентів системи на хмарному хостингу під управлінням **Windows Server**. Проведено міграцію бази даних за допомогою стратегії Backup/Restore. Успішно налаштовано веб-сервер **Internet Information Services (IIS)** для суміщеної роботи з кросплатформним сервером **Kestrel**.

7. Автоматизовано процеси публікації та налаштовано інфраструктуру.

Застосування технології **Web Deploy** дозволило автоматизувати процес оновлення серверного коду. Інтеграція React-додатка в каталог `wwwroot` та налаштування правил переадресації URL (**URL Rewrite Module**) забезпечили коректну маршрутизацію SPA. Впровадження SSL-сертифіката гарантувало криптографічний захист даних під час їхньої передачі мережею.

Практичне значення отриманих результатів полягає у тому, що розроблений програмний комплекс «CompHub» не є просто прототипом, а являє собою повністю функціональний, безпечний та готовий до комерційної експлуатації продукт. Створена платформа дозволяє автоматизувати процеси продажу комп'ютерної техніки, підвищити рівень обслуговування клієнтів завдяки зручному інтерфейсу та забезпечити власнику бізнесу надійний інструмент для ведення електронної комерції.

Перспективи подальшого розвитку. Закладена під час проєктування гнучка модульна архітектура створює міцний фундамент для подальшого масштабування системи. До перспективних напрямків розвитку проєкту можна віднести: впровадження мікросервісної архітектури при зростанні навантаження, перехід клієнтської частини на технологію Server-Side Rendering (SSR) за допомогою фреймворку Next.js для покращення показників SEO, інтеграцію з популярними платіжними шлюзами (наприклад, LiqPay або Stripe), а також впровадження повноцінної системи безперервної інтеграції та розгортання (CI/CD) для автоматизації процесів тестування та деплою.

Підсумовуючи, можна стверджувати, що мета кваліфікаційної роботи досягнута повною мірою. Програмний комплекс спроектовано та реалізовано на

високому технічному рівні, що підтверджує ґрунтовні знання, практичні навички та інженерну кваліфікацію, здобуті за час навчання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Фаулер М. UML. Основи. 3-тє вид. Київ : Форс Україна, 2020. 208 с.
2. Конноллі Т., Бегг К. Бази даних. Проєктування, реалізація і супровід. Теорія і практика. 3-тє вид. Київ : Діалектика, 2019. 1440 с.
3. Офіційна документація Microsoft щодо .NET та ASP.NET Core [Електронний ресурс]. URL: <https://learn.microsoft.com/uk-ua/aspnet/core/> (дата звернення: 08.05.2026).
4. Офіційна документація Entity Framework Core [Електронний ресурс]. URL: <https://learn.microsoft.com/uk-ua/ef/core/> (дата звернення: 08.05.2026).
5. Документація Microsoft SQL Server 2022 [Електронний ресурс]. URL: <https://learn.microsoft.com/uk-ua/sql/sql-server/> (дата звернення: 08.05.2026).
6. Офіційна документація React [Електронний ресурс]. URL: <https://react.dev/> (дата звернення: 08.05.2026).
7. Документація TypeScript [Електронний ресурс]. URL: <https://www.typescriptlang.org/docs/> (дата звернення: 08.05.2026).
8. Офіційна документація Redux [Електронний ресурс]. URL: <https://redux.js.org/> (дата звернення: 08.05.2026).
9. Інструмент для збірки фронтенду Vite [Електронний ресурс]. URL: <https://vitejs.dev/guide/> (дата звернення: 08.05.2026).
10. Документація середовища розробки Visual Studio [Електронний ресурс]. URL: <https://learn.microsoft.com/uk-ua/visualstudio/> (дата звернення: 08.05.2026).

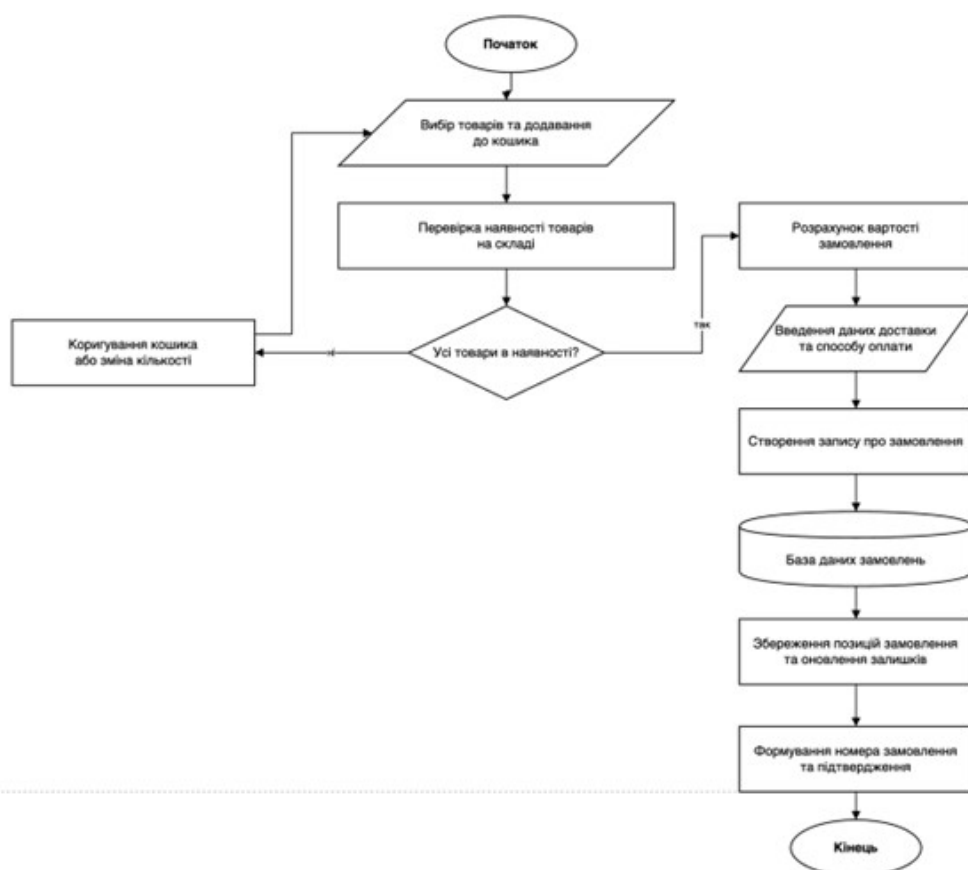
11. Мартін Р. Чиста архітектура. Мистецтво розробки програмного забезпечення. Харків : Фабула, 2021. 368 с.
12. Гамма Е., Хелм Р., Джонсон Р., Вліссідес Д. Патерни об'єктно-орієнтованого проєктування. Харків : Фабула, 2022. 416 с.
13. Фрімен А. ASP.NET Core MVC з прикладами на С# для професіоналів. Київ : Діалектика, 2021. 1008 с.
14. Банкс А., Порселло Є. React і Redux. Створення веб-додатків. Київ : КІС, 2020. 336 с.
15. Соммервілл І. Інженерія програмного забезпечення. 10-те вид. Київ : Основи, 2019. 800 с.
16. Документація щодо архітектури мікросервісів та монолітів від Microsoft [Електронний ресурс]. URL: <https://learn.microsoft.com/uk-ua/dotnet/architecture/> (дата звернення: 08.05.2026).
17. Клеппман М. Проєктування систем, що працюють з даними. Київ : Фабула, 2020. 656 с.
18. Рекомендації з веб-доступності (WCAG) 2.1 [Електронний ресурс]. URL: <https://www.w3.org/TR/WCAG21/> (дата звернення: 08.05.2026).
19. Документація бібліотеки React Router [Електронний ресурс]. URL: <https://reactrouter.com/> (дата звернення: 08.05.2026).
20. Троелсен Е., Джепікс Ф. Мова програмування С# 10 та платформа .NET 6. Київ : Діалектика, 2022. 1328 с.

ДОДАТОК А

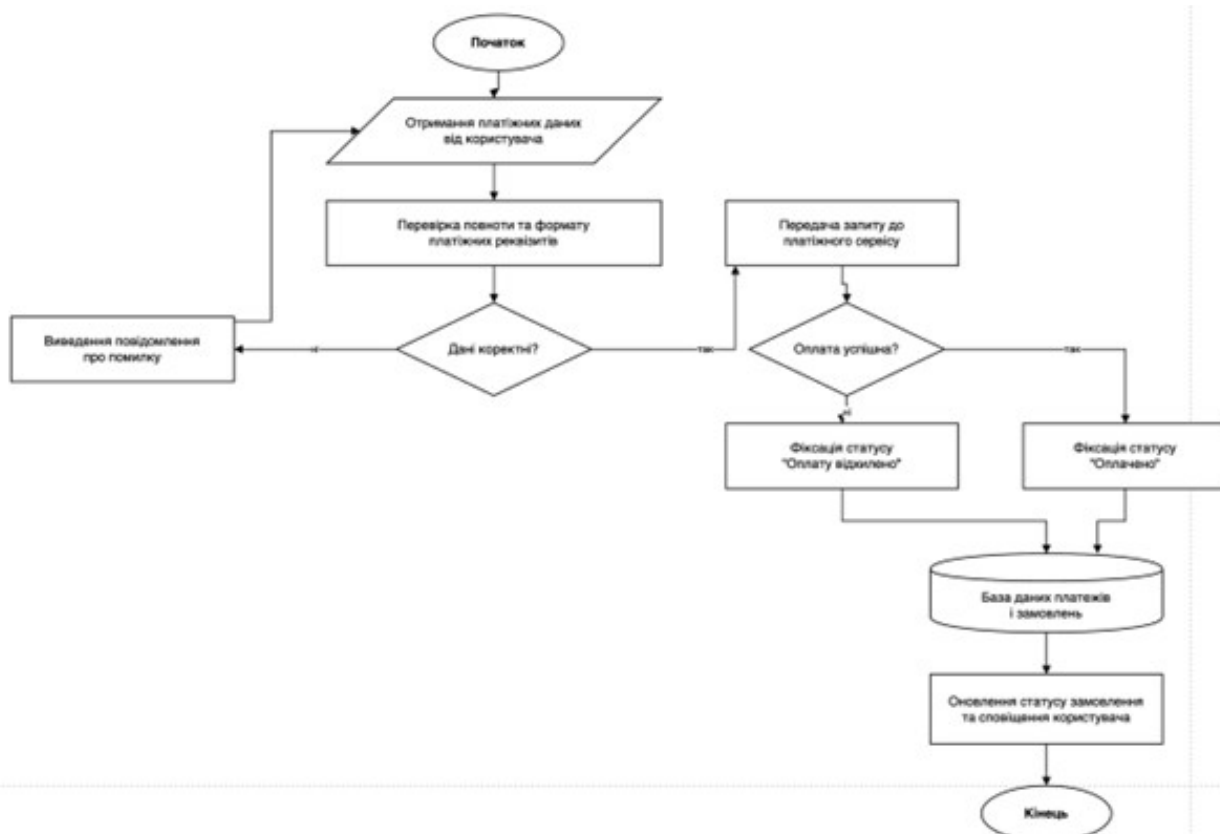
Алгоритм пошуку та відбору товарів:



Алгоритм оформлення замовлення:



Алгоритм оплати замовлення:



ДОДАТОК Б

Клієнт-серверна взаємодія для отримання продуктів

Бекенд

Код сервісу роботи з продуктами *ProductService.cs*:

```
namespace CompHub.WebApi.Application.Services
{
    /// <summary>
    /// Handles product filtering, retrieval, and CRUD operations.
    /// </summary>
    public class ProductService : IProductService
    {
        private readonly ProductRepository _productRepository;
        private readonly ApplicationDbContext _context;

        public ProductService(ProductRepository productRepository,
            ApplicationDbContext context)
        {
            _productRepository = productRepository;
            _context = context;
        }

        /// <inheritdoc/>
        public async Task<PagedResult<ProductListItemDto>> GetFilteredAsync(
            ProductFilterParams filter,
            CancellationToken cancellationToken = default)
        {
            var (items, totalCount) = await
                _productRepository.GetFilteredAsync(filter, cancellationToken);

            return new PagedResult<ProductListItemDto>(
                Items: items,
                TotalCount: totalCount,
                PageNumber: filter.Page,
                PageSize: filter.PageSize,
                TotalPages: (int)Math.Ceiling((double)totalCount /
                    filter.PageSize)
            );
        }

        /// <inheritdoc />
        public async Task<ProductDetailDto?> GetByIdAsync(int id,
            CancellationToken cancellationToken = default)
        {
            // Splited query
            var product = await _context.Products
                .AsNoTracking()
                .AsSplitQuery()
```

```

        .Include(p => p.Category)
        .Include(p => p.Brand)
        .Include(p => p.Reviews)
        .Include(p => p.Specifications)
        .ThenInclude(s => s.SpecificationKey)
        .FirstOrDefaultAsync(p => p.Id == id, cancellationToken);

    return product is null ? null : MapToDetail(product);
}

/// <inheritdoc />
public async Task<ProductDetailDto> CreateAsync(
    CreateProductRequest request,
    CancellationToken cancellationToken = default)
{
    var product = new Product
    {
        Name = request.Name,
        Description = request.Description,
        Price = request.Price,
        StockQuantity = request.StockQuantity,
        ImageUrl = request.ImageUrl,
        CategoryId = request.CategoryId,
        BrandId = request.BrandId,
        CreatedAt = DateTime.UtcNow,
        Specifications = request.Specifications
            .Select(s => new ProductSpecification
            {
                SpecificationKeyId = s.SpecificationKeyId,
                Value = s.Value
            })
            .ToList();
    };

    _context.Products.Add(product);
    await _context.SaveChangesAsync(cancellationToken);

    // Перезавантажуємо з навігаційними властивостями для відповіді
    return (await GetByIdAsync(product.Id, cancellationToken))!;
}

/// <inheritdoc />
public async Task<ProductDetailDto?> UpdateAsync(
    int id,
    UpdateProductRequest request,
    CancellationToken cancellationToken = default)
{
    var product = await _context.Products
        .Include(p => p.Specifications)
        .FirstOrDefaultAsync(p => p.Id == id, cancellationToken);

```

```

        if (product is null) return null;

        product.Name = request.Name;
        product.Description = request.Description;
        product.Price = request.Price;
        product.StockQuantity = request.StockQuantity;
        product.IsActive = request.IsActive;
        product.ImageUrl = request.ImageUrl;
        product.CategoryId = request.CategoryId;
        product.BrandId = request.BrandId;

        // Замінюємо специфікації повністю
        _context.ProductSpecifications.RemoveRange(product.Specifications);
        product.Specifications = request.Specifications
            .Select(s => new ProductSpecification
            {
                ProductId = id,
                SpecificationKeyId = s.SpecificationKeyId,
                Value = s.Value
            })
            .ToList();

        await _context.SaveChangesAsync(cancellationToken);

        return await GetByIdAsync(id, cancellationToken);
    }

    /// <inheritdoc />
    public async Task<bool> DeleteAsync(int id, CancellationToken
cancellationToken = default)
    {
        var product = await _context.Products.FindAsync([id],
cancellationToken);
        if (product is null) return false;

        _context.Products.Remove(product);
        await _context.SaveChangesAsync(cancellationToken);
        return true;
    }

    /// <summary>
    /// Maps fully-loaded <see cref="Product"/> entity to <see
cref="ProductDetailDto"/>.
    private static ProductDetailDto MapToDetail(Product p) => new(
        Id: p.Id,
        Name: p.Name,
        Description: p.Description,
        Price: p.Price,

```

```

        StockQuantity: p.StockQuantity,
        IsActive: p.IsActive,
        ImageUrl: p.ImageUrl,
        CreatedAt: p.CreatedAt,
        CategoryId: p.CategoryId,
        CategoryName: p.Category.Name,
        BrandId: p.BrandId,
        BrandName: p.Brand.Name,
        BrandImageUrl: p.Brand.LogoUrl,
        Specifications: p.Specifications
            .OrderBy(s => s.SpecificationKey.DisplayOrder)
            .Select(s => new SpecificationDto(
                Key: s.SpecificationKey.Name,
                Unit: s.SpecificationKey.Unit,
                Value: s.Value,
                DisplayOrder: s.SpecificationKey.DisplayOrder
            ))
            .ToList(),
        AverageRating: p.Reviews.Count > 0
            ? Math.Round(p.Reviews.Average(r => r.Rating), 1)
            : null,
        ReviewCount: p.Reviews.Count
    );
}
}

```

Репозиторій для роботи з продуктами ProductRepository.cs:

```

namespace CompHub.WebApi.Infrastructure.Data.Repositories
{
    /// <summary>
    /// Encapsulates the complex filtered product query that would pollute
    the service layer.
    /// Projects directly to ProductListItemDto to avoid over-fetching.
    /// Simple CRUD operations remain directly in ProductService via
    ApplicationDbContext.
    /// </summary>
    public class ProductRepository
    {
        private readonly ApplicationDbContext _context;

        public ProductRepository(ApplicationDbContext context)
        {
            _context = context;
        }

        public async Task<(List<ProductListItemDto> Items, int TotalCount)>
        GetFilteredAsync(
            ProductFilterParams filter,
            CancellationToken cancellationToken = default)
        {

```

```

var query = _context.Products
    .AsNoTracking()
    .Where(p => p.IsActive)
    .AsQueryable();

if (filter.CategoryId.HasValue)
    query = query.Where(p =>
        p.CategoryId == filter.CategoryId.Value ||
        p.Category.ParentCategoryId == filter.CategoryId.Value);

if (filter.BrandIds is { Count: > 0 })
    query = query.Where(p =>
filter.BrandIds.Contains(p.BrandId));

if (filter.MinPrice.HasValue)
    query = query.Where(p => p.Price >= filter.MinPrice.Value);

if (filter.MaxPrice.HasValue)
    query = query.Where(p => p.Price <= filter.MaxPrice.Value);

if (!string.IsNullOrEmpty(filter.SearchTerm))
    query = query.Where(p =>
p.Name.Contains(filter.SearchTerm));

if (filter.InStock is true)
    query = query.Where(p => p.StockQuantity > 0);

if (filter.Specifications is { Count: > 0 })
{
    foreach (var (keyId, values) in filter.Specifications)
    {
        query = query.Where(p =>
            p.Specifications.Any(s =>
                s.SpecificationKeyId == keyId &&
                values.Contains(s.Value)));
    }
}

query = filter.SortBy switch
{
    "price_asc" => query.OrderBy(p => p.Price),
    "price_desc" => query.OrderByDescending(p => p.Price),
    "name" => query.OrderBy(p => p.Name),
    _ => query.OrderByDescending(p => p.CreatedAt)
};

var totalCount = await query.CountAsync(cancellationToken);

// Project to anonymous type first – Math.Round is not
translatable in EF projection

```

```

var raw = await query
    .Skip((filter.Page - 1) * filter.PageSize)
    .Take(filter.PageSize)
    .Select(p => new
    {
        p.Id,
        p.Name,
        p.Price,
        p.StockQuantity,
        p.ImageUrl,
        CategoryName = p.Category.Name,
        BrandName = p.Brand.Name,
        ReviewCount = p.Reviews.Count(),
        RawRating = (double?)p.Reviews.Average(r =>
(double?)r.Rating),
    })
    .ToListAsync(cancellationToken);

var items = raw.Select(x => new ProductListItemDto(
    Id: x.Id,
    Name: x.Name,
    Price: x.Price,
    StockQuantity: x.StockQuantity,
    ImageUrl: x.ImageUrl,
    CategoryName: x.CategoryName,
    BrandName: x.BrandName,
    AverageRating: x.RawRating.HasValue ?
Math.Round(x.RawRating.Value, 1) : null,
    ReviewCount: x.ReviewCount
)).ToList();

return (items, totalCount);
}
}
}

```

Контролер для роботи з продуктами ProductController.cs:

```

namespace CompHub.WebApi.Controllers
{
    [ApiController]
    [Route("api/products")]
    public class ProductsController : ControllerBase
    {
        private readonly IProductService _productService;

        public ProductsController(IProductService productService)
        {
            _productService = productService;
        }
    }
}

```

```

    /// <summary>
    /// Returns a paginated and filtered product list.
    /// </summary>
    [HttpGet]
    public async Task<IActionResult> GetFiltered(
        [FromQuery] ProductFilterParams filter,
        CancellationToken cancellationToken)
    {
        var result = await _productService.GetFilteredAsync(filter,
cancellationToken);
        return Ok(ApiResponse<object>.SuccessResponse(result));
    }

    /// <summary>
    /// Returns full product details by id.
    /// </summary>
    [HttpGet("{id:int}")]
    public async Task<IActionResult> GetById(int id, CancellationToken
cancellationToken)
    {
        var product = await _productService.GetByIdAsync(id,
cancellationToken);

        if (product is null)
            return NotFound(ApiResponse.ErrorResponse($"Product {id} not
found."));

        return Ok(ApiResponse<object>.SuccessResponse(product));
    }

    /// <summary>
    /// Creates a new product.
    /// </summary>
    [Authorize(Roles = "Admin")]
    [HttpPost]
    public async Task<IActionResult> Create(
        [FromBody] CreateProductRequest request,
        CancellationToken cancellationToken)
    {
        var product = await _productService.CreateAsync(request,
cancellationToken);
        return CreatedAtAction(
            nameof(GetById),
            new { id = product.Id },
            ApiResponse<object>.SuccessResponse(product, "Product
created successfully."));
    }

    /// <summary>

```

```

    /// Updates an existing product.
    /// </summary>
    [Authorize(Roles = "Admin")]
    [HttpPut("{id:int}")]
    public async Task<IActionResult> Update(
        int id,
        [FromBody] UpdateProductRequest request,
        CancellationToken cancellationToken)
    {
        var product = await _productService.UpdateAsync(id, request,
cancellationToken);

        if (product is null)
            return NotFound(ApiResponse.ErrorResponse($"Product {id} not
found."));

        return Ok(ApiResponse<object>.SuccessResponse(product));
    }

    /// <summary>
    /// Deletes a product by id.
    /// </summary>
    [Authorize(Roles = "Admin")]
    [HttpDelete("{id:int}")]
    public async Task<IActionResult> Delete(int id, CancellationToken
cancellationToken)
    {
        var deleted = await _productService.DeleteAsync(id,
cancellationToken);

        if (!deleted)
            return NotFound(ApiResponse.ErrorResponse($"Product {id} not
found."));

        return NoContent();
    }
}

```

Фронтенд:

Сторінка каталогу товарів *CatalogPage.tsx*:

```

import { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { Button, Col, Container, Form, Offcanvas, Row } from 'react-
bootstrap';
import { BsFilter } from 'react-icons/bs';
import { useGetProductsQuery } from './productsApi';
import { useGetBrandsQuery } from './brandsApi';

```

```

import { useGetCategoriesQuery, useGetCategorySpecFiltersQuery } from
'./categoriesApi';
import { useQueryParams } from '../hooks/useQueryParams';
import useScrollRestore from '../hooks/useScrollRestore';
import CatalogFilters from './CatalogFilters';
import ActiveFilters from './ActiveFilters';
import ProductCard from './ProductCard';
import SkeletonCard from '../components/common/SkeletonCard';
import Pagination from '../components/common/Pagination';
import EmptyState from '../components/common/EmptyState';
import ErrorAlert from '../components/common/ErrorAlert';
import type { ProductFilterParams } from '../types/product';

const SORT_OPTIONS: { value: ProductFilterParams['sortBy']; label: string }
[] = [
  { value: 'newest', label: 'Новинки' },
  { value: 'price_asc', label: 'Ціна: від дешевих' },
  { value: 'price_desc', label: 'Ціна: від дорогих' },
  { value: 'name', label: 'За назвою' },
];

const SKELETON_COUNT = 12;

const parseFilter = (params: URLSearchParams): ProductFilterParams => ({
  categoryId: params.get('categoryId') ? Number(params.get('categoryId'))
: undefined,
  brandIds: params.getAll('brandIds').length > 0
  ? params.getAll('brandIds').map(Number).filter(Boolean)
  : params.get('brandIds')
  ? params.get('brandIds')!.split(',').map(Number).filter(Boolean)
  : undefined,
  minPrice: params.get('minPrice') ? Number(params.get('minPrice')) :
undefined,
  maxPrice: params.get('maxPrice') ? Number(params.get('maxPrice')) :
undefined,
  searchTerm: params.get('searchTerm') ?? undefined,
  specifications: (() => {
    const specs: Record<number, string[]> = {};
    for (const key of params.keys()) {
      const match = key.match(/^specifications\\[(\\d+)\\]$/);
      if (!match) continue;
      const keyId = Number(match[1]);
      const values = params.getAll(key).filter(Boolean);
      if (values.length > 0) specs[keyId] = values;
    }
    return Object.keys(specs).length > 0 ? specs : undefined;
  })(),
  inStock: params.get('inStock') === 'true' ? true : undefined,
  sortBy: (params.get('sortBy') as ProductFilterParams['sortBy']) ??
'newest',

```

```

    page: params.get('page') ? Number(params.get('page')) : 1,
    pageSize: 12,
  });

const CatalogPage = () => {
  const navigate = useNavigate();
  const { searchParams, setParams } = useQueryParams();
  const filter = parseFilter(searchParams);

  useScrollRestore('catalog');

  const [showFilters, setShowFilters] = useState(false);

  const { data: paged, isLoading, isFetching, isError } =
    useGetProductsQuery(filter);
  const { data: brands = [] } = useGetBrandsQuery();
  const { data: specFilters = [] } = useGetCategorySpecFiltersQuery(
    filter.categoryId!,
    { skip: !filter.categoryId }
  );
  const { data: categories = [] } = useGetCategoriesQuery();

  const handleFilterChange = (updates: Partial<ProductFilterParams> &
    { brandIds?: number[] }): void => {
    const next = new URLSearchParams(searchParams);
    next.set('page', '1');

    if ('categoryId' in updates) {
      updates.categoryId !== null
        ? next.set('categoryId', String(updates.categoryId))
        : next.delete('categoryId');
      // Скидаємо бренди та специфікації при зміні категорії
      next.delete('brandIds');
      for (const key of [...next.keys()]) {
        if (/^specifications\[/.test(key)) next.delete(key);
      }
    }
    if ('brandIds' in updates) {
      next.delete('brandIds');
      updates.brandIds?.forEach((id) => next.append('brandIds',
String(id)));
    }
    if ('minPrice' in updates) {
      updates.minPrice !== null ? next.set('minPrice',
String(updates.minPrice)) : next.delete('minPrice');
    }
    if ('maxPrice' in updates) {
      updates.maxPrice !== null ? next.set('maxPrice',
String(updates.maxPrice)) : next.delete('maxPrice');
    }
  }

```

```

        if ('inStock' in updates) {
            updates.inStock ? next.set('inStock', 'true') :
next.delete('inStock');
        }
        if ('searchTerm' in updates) {
            updates.searchTerm ? next.set('searchTerm', updates.searchTerm)
: next.delete('searchTerm');
        }
        if ('specifications' in updates) {
            for (const key of [...next.keys()]) {
                if (/^specifications\[/.test(key)) next.delete(key);
            }
            if (updates.specifications) {
                Object.entries(updates.specifications).forEach(([keyId,
values]) => {
                    values.forEach((v) => next.append(`specifications[$
{keyId}]`, v));
                });
            }
            if ('page' in updates && updates.page !== null) {
                next.set('page', String(updates.page));
            }

            navigate({ search: next.toString() }, { replace: true });
        };

const handleReset = (): void => {
    navigate({ search: '' }, { replace: true });
};

const filtersProps = { filter, onChange: handleFilterChange, onReset:
handleReset };

const activeFilterCount = [
    filter.categoryId ? 1 : 0,
    filter.brandIds?.length ?? 0,
    filter.minPrice ? 1 : 0,
    filter.maxPrice ? 1 : 0,
    filter.inStock ? 1 : 0,
    filter.searchTerm ? 1 : 0,
    Object.values(filter.specifications ?? {}).flat().length,
].reduce((a, b) => a + b, 0);

return (
    <Container>
        <Row className="g-4">
            {/* Sidebar – desktop only */}
            <Col lg={3} className="d-none d-lg-block">
                <CatalogFilters {...filtersProps} />
            </Col>
        </Row>
    </Container>
);

```

```

</Col>

{/* Content */}
<Col lg={9}>
  {/* Toolbar – тільки сортування */}
  <div className="d-flex justify-content-between align-items-center gap-2 mb-3 flex-wrap">
    {/* Mobile filter button */}
    <Button
      variant="outline-secondary"
      className="d-lg-none d-flex align-items-center gap-2"
      onClick={() => setShowFilters(true)}
    >
      <BsFilter size={18} />
      Фільтри
      {activeFilterCount > 0 && (
        <span className="badge bg-primary rounded-pill ms-1">{activeFilterCount}</span>
      )}
    </Button>

    <div className="ms-auto">
      <Form.Select
        style={{ maxWidth: '220px' }}
        value={filter.sortBy ?? 'newest'}
        onChange={(e) =>
          setParams({ sortBy: e.target.value as ProductFilterParams['sortBy'], page: 1 })
        }
      >
        {SORT_OPTIONS.map((opt) => (
          <option key={opt.value}
            value={opt.value}>{opt.label}</option>
        ))}
      </Form.Select>
    </div>
  </div>

  {/* Active filter chips */}
  <ActiveFilters
    filter={filter}
    brands={brands}
    categories={categories}
    specFilters={specFilters}
    onChange={handleFilterChange}
    onReset={handleReset}
  />

  {/* Results count */}

```

```

        {paged && !isLoading && (
          <p className="text-muted small mb-3">
            Знайдено товарів: <strong>{paged.totalCount}
</strong>
          </p>
        )}

    {/* Grid */}
    {isError ? (
      <ErrorAlert />
    ) : isLoading ? (
      <Row xs={1} sm={2} md={3} xl={4} className="g-3">
        {Array.from({ length: SKELETON_COUNT }).map((_,
i) => (
          <Col key={i}><SkeletonCard /></Col>
        ))}
      </Row>
    ) : !paged || paged.items.length === 0 ? (
      <EmptyState
        title="Товарів не знайдено"
        description="Спробуйте змінити фільтри або
пошуковий запит"
        action={{ label: 'Скинути фільтри', onClick:
handleReset }}
      />
    ) : (
      <div style={{ opacity: isFetching ? 0.6 : 1,
transition: 'opacity 0.2s' }}>
        <Row xs={1} sm={2} md={3} xl={4}
className="g-3">
          {paged.items.map((product) => (
            <Col key={product.id}>
              <ProductCard product={product} />
            </Col>
          ))}
        </Row>
        <Pagination
          currentPage={filter.page ?? 1}
          totalPages={paged.totalPages}
          onPageChange={({page} => setParams({ page })}
        />
      </div>
    )}
  </Col>
</Row>

  {/* Mobile filters offcanvas */}
  <Offcanvas show={showFilters} onHide={() =>
setShowFilters(false)} placement="start">
    <Offcanvas.Header closeButton>

```

```

        <Offcanvas.Title className="fw-
bold">Фільтри</Offcanvas.Title>
    </Offcanvas.Header>
    <Offcanvas.Body>
        <CatalogFilters
            {...filtersProps}
            onReset={() => { handleReset();
setShowFilters(false); }}
        />
    </Offcanvas.Body>
</Offcanvas>
</Container>
);
};

export default CatalogPage;

```

Сервіс для отримання даних з бекенду по ендпоінтам productApi.ts:

```

import { baseApi } from '../..api/baseApi';
import type { ApiResponse, PagedResult } from '../..types/api';
import type {
    CreateProductRequest,
    ProductDetail,
    ProductFilterParams,
    ProductListItem,
    UpdateProductRequest,
} from '../..types/product';

export const productsApi = baseApi.injectEndpoints({
    endpoints: (builder) => ({
        getProducts: builder.query<PagedResult<ProductListItem>,
ProductFilterParams>({
            transformResponse: (response:
ApiResponse<PagedResult<ProductListItem>>) => response.data!,
            query: (params) => {
                const { brandIds, specifications, ...rest } = params;
                const qs = new URLSearchParams();

                (Object.entries(rest) as [string, unknown][]).forEach(([key,
value]) => {
                    if (value !== undefined && value !== null) qs.set(key,
String(value));
                });

                brandIds?.forEach((id) => qs.append('brandIds',
String(id)));

                if (specifications) {

```

```

        Object.entries(specifications).forEach(([keyId, values])
=> {
            values.forEach((v) => qs.append(`specifications[${keyId}]`, v));
        });
    }

    return `/products?${qs.toString()}`;
},
providesTags: ['Product'],
}),
getProductById: builder.query<ProductDetail, number>({
    transformResponse: (response: ApiResponse<ProductDetail>) =>
response.data!,
    query: (id) => `/products/${id}`,
    providesTags: (_result, _error, id) => [{ type: 'Product',
id }],
}),
createProduct: builder.mutation<ProductDetail,
CreateProductRequest>({
    transformResponse: (response: ApiResponse<ProductDetail>) =>
response.data!,
    query: (body) => ({ url: '/products', method: 'POST', body }),
    invalidatesTags: ['Product'],
}),
updateProduct: builder.mutation<ProductDetail, { id: number; body:
UpdateProductRequest }>({
    transformResponse: (response: ApiResponse<ProductDetail>) =>
response.data!,
    query: ({ id, body }) => ({ url: `/products/${id}`, method:
'PUT', body }),
    invalidatesTags: (_result, _error, { id }) => [{ type:
'Product', id }, 'Product'],
}),
deleteProduct: builder.mutation<void, number>({
    query: (id) => ({ url: `/products/${id}`, method: 'DELETE' }),
    invalidatesTags: ['Product'],
}),
}),
});

export const {
    useGetProductsQuery,
    useGetProductByIdQuery,
    useCreateProductMutation,
    useUpdateProductMutation,
    useDeleteProductMutation,
} = productsApi;

```

Базовий сервіс для обробки даних з бекенду baseApi.ts:

```

/// <reference types="vite/client" />
import {
  createApi,
  fetchBaseQuery,
  type BaseQueryFn,
  type FetchArgs,
  type FetchBaseQueryError,
} from '@reduxjs/toolkit/query/react';
import type { RootState } from '../store';
import { logout } from '../features/auth/authSlice';

const rawBaseQuery = fetchBaseQuery({
  baseUrl: (import.meta.env.VITE_API_URL as string) ?? '/api',
  prepareHeaders: (headers, { getState }) => {
    const token = (getState() as RootState).auth.token;
    if (token) {
      headers.set('Authorization', `Bearer ${token}`);
    }
    return headers;
  },
});

const baseQueryWithReauth: BaseQueryFn<string | FetchArgs, unknown,
FetchBaseQueryError> = async (
  args,
  api,
  extraOptions
) => {
  const result = await rawBaseQuery(args, api, extraOptions);

  if (result.error?.status === 401) {
    api.dispatch(logout());
  }

  return result;
};

export const baseApi = createApi({
  reducerPath: 'api',
  baseQuery: baseQueryWithReauth,
  tagTypes: ['Product', 'Order', 'Review'],
  endpoints: () => ({}),
});

```

ДОДАТОК В

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

ДЕКЛАРАЦІЯ ЗДОБУВАЧА

Я, Братчик Костянтин Русланович, здобувач(ка) НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота (проект) на тему «Інтернет-магазин продажу комп'ютерної техніки» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):
 - ☐ інструменти генеративного ШІ не використовувалися.
 - ☒ інструменти ШІ (вказати назву: ChatGPT, Gemini, Claude, DeepL, Gemini, Claude, DeepL інші (вказати назву)) були використані для:
 - ☒ перекладу іншомовних джерел;
 - ☒ стилістичного редагування та перевірки граматики;
 - ☒ допомоги у написанні/відлагодженні програмного коду;
 - ☐ інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«___» _____ 2026 р. _____

Костянтин

БРАТЧИК

(підпис)