НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук

_____/Голуб Б.Л., доц., к.т.н. /
підпис ПІБ, вчене звання і ступінь

«___»_____2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Програмне забезпечення для інформаційної системи з продажу комп'ютерної техніки»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми
_____к.т.н., доцент_____/ Вайганг Г.О./
Науковий ступень та вчене звання підпис ПІБ

Керівник бакалаврської кваліфікаційної роботи :_____/ Кириченко В.В/
підпис ПІБ

Виконав:_____/ Тищенко Н.В./
підпис ПІБ

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп’ютерних наук

_____ / Голуб Б.Л., доцент, к.т.н /
підпис

“ ” 2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

студенту Тищенко Нікіті Володимировичу

Спеціальність 121 «Інженерія програмного забезпечення»

1. Тема роботи: Програмне забезпечення для інформаційної системи з продажу комп'ютерної техніки

Затверджена наказом ректора НУБіП України № 2249 “С” від 16.12.2024

2. Термін подання завершеної роботи на кафедру 2025 . __ . ___
рік, місяць, число

3. Вихідні дані до роботи: опис програмного забезпечення

4. Перелік питань що розглядаються:

1. Аналіз пердметної області.
2. Проєктування інформаційного та програмного забезпечення.
3. Прикладне програмне забезпечення.
4. Рекомендації щодо впровадження та експлуатації системи.
5. Висновки.

Керівник бакалаврської кваліфікаційної роботи _____ / Кириченко В.В. /
підпис ініціали та прізвище

Завдання прийняв до виконання _____ / Тищенко Н.В. /
підпис ініціали та прізвище

Дата отримання завдання 2024 . 12 . 16
рік, місяць, число

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД — база даних;

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

API — програмний інтерфейс прикладного програмування (Application Programming Interface);

GUI — графічний інтерфейс користувача (Graphical User Interface);

UX — досвід користувача (User Experience);

UI — інтерфейс користувача (User Interface);

SMTP — протокол надсилання електронної пошти (Simple Mail Transfer Protocol);

REST — архітектурний стиль обміну даними між клієнтом і сервером (Representational State Transfer);

CRUD — створення, читання, оновлення, видалення (Create, Read, Update, Delete);

CI/CD — безперервна інтеграція та розгортання (Continuous Integration / Continuous Delivery);

JWT — JSON Web Token (формат передачі токенів автентифікації);

UML — уніфікована мова моделювання (Unified Modeling Language);

ER-діаграма — діаграма "сутність-зв’язок" (Entity-Relationship diagram);

SPA — односторінковий додаток (Single Page Application);

IDE — інтегроване середовище розробки (Integrated Development Environment);

JSON — формат обміну даними (JavaScript Object Notation);

CRUD-операції — базові операції взаємодії з даними: створення, читання, редагування та видалення.

ЗМІСТ

ВСТУП 4

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛСТІ 6

1.1 Опис предметної області 6

1.2 Аналіз вимог до програмної системи 7

1.3 Моделювання предметної області 9

1.4 Огляд інформаційних джерел та існуючих рішень 13

1.5 Постановка завдання 15

2. ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 18

2.1 Логічна модель даних у вигляді ER-діаграми 18

2.2 Вибір системи управління інформаційною базою 20

2.3 Створення інформаційної бази 21

3. ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ 23

3.1 Вибір інструментарію для розробки програмного забезпечення 23

3.2 Принципи роботи у середовищі візуальної розробки програм 25

4. РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ 34

4.1 Тестування системи 34

4.2 Вимоги до апаратного та програмного забезпечення 35

4.3 Склад інсталяційного пакету 37

ВИСНОВКИ 40

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 43

ДОДАТОК А 45

ДОДАТОК Б 51

ДОДАТОК В 59

ДОДАТОК Г 67

ВСТУП

У сучасних умовах інтенсивного розвитку інформаційних технологій торгівля стрімко переходить в онлайн-простір. Продаж комп'ютерної техніки — одна з галузей, що особливо активно розвивається в електронній комерції, оскільки комп'ютери, ноутбуки, комплектуючі та інші цифрові пристрої є невід'ємною частиною життя кожної сучасної людини. Водночас для ефективного управління процесами продажу, обліку товарів, взаємодії з клієнтами та організації доставки необхідна гнучка, масштабована та зручна у користуванні інформаційна система.

Актуальність теми зумовлена потребою малого та середнього бізнесу в інструментах, які дозволяють автоматизувати торгівельні процеси, підвищити ефективність обробки замовлень і покращити клієнтський сервіс. У сучасному цифровому середовищі підприємства, які не мають зручної та функціональної системи онлайн-продажів, поступаються конкурентам. Відтак, розробка універсального рішення для продажу комп'ютерної техніки має важливе практичне значення.

Метою дипломної роботи є розробка прикладного програмного забезпечення для інформаційної системи, яка автоматизує основні бізнес-процеси магазину комп'ютерної техніки — від реєстрації клієнта, перегляду товарів, збірка комп'ютерів онлайн, формування замовлення і його оплати до обробки замовлень менеджерами та адміністрування товарного асортименту. Програмний продукт повинен бути зручним у використанні, безпечним, масштабованим і придатним для розгортання у реальних умовах.

Для реалізації проєкту використовуються сучасні технології розробки вебзастосунків:

фреймворк Django на мові Python — для побудови серверної частини та API;

бібліотека React — для створення адаптивного, динамічного інтерфейсу користувача;

система керування базами даних PostgreSQL;

технології Docker, Celery, Redis, Stripe — для контейнеризації, обробки асинхронних задач, кешування і прийому платежів.

Структура пояснювальної записки складається з 5 розділів, обсяг документу становить 58 сторінок. У записці використано 19 джерел, надано 3 UML-діаграм та 13 скріншотів реалізованої системи. У першому розділі подано аналіз предметної області, описано функціональні вимоги до системи, наведено постановку задачі. У другому — розглянуто структуру бази даних, виконано її проєктування та описано логічну модель у вигляді ER-діаграми. Третій розділ присвячено програмній реалізації, опису архітектури, вибору інструментів та реалізації REST API. У четвертому подано рекомендації щодо впровадження, описано процес тестування та структуру інсталяційного пакету.

Таким чином, розроблене програмне забезпечення є комплексним технічним рішенням, здатним суттєво покращити ефективність торгового процесу, зменшити навантаження на персонал і забезпечити високий рівень обслуговування клієнтів.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛСТІ

1.1 Опис предметної області

Розвиток цифрових технологій, а також постійне зростання попиту на комп’ютерну техніку зумовлює необхідність створення інструментів, які дозволяють автоматизувати процес продажу та обслуговування клієнтів. У традиційній моделі взаємодії з покупцями часто виникають труднощі, пов’язані з ручним веденням обліку товарів, затримками в обробці замовлень, недостатньою прозорістю у взаємодії між клієнтами, менеджерами та адміністраторами. Це, у свою чергу, знижує ефективність бізнесу та рівень задоволеності клієнтів.

Саме тому основна задача, яка стоїть перед розробником, полягає у створенні сучасної інформаційної системи для продажу комп’ютерної техніки, яка буде охоплювати всі ключові процеси, починаючи з управління товарним асортиментом і завершуючи обслуговуванням клієнтів після покупки. Така система повинна забезпечувати надійне зберігання даних, зручний інтерфейс для користувачів з різними ролями та можливість гнучкого розширення функціоналу.

Система передбачає три основні ролі користувачів: **Клієнт**, **Менеджер** та **Адміністратор**. Кожна з цих ролей виконує свою частину функцій, що дозволяє розділити обов’язки та зменшити навантаження на окремих учасників процесу (див. таблиця 1).

Таблиця 1

Роль	Опис	Основні функції
Клієнт	Користується веб-інтерфейсом для перегляду та покупки товарів. Має простий і зручний доступ до функцій магазину.	- Реєстрація та авторизація - Перегляд каталогу товарів - Додавання товарів у кошик - Оформлення замовлень - Оплата - Перегляд історії замовлень
Менеджер	Відповідає за обробку замовлень, взаємодіє з клієнтами при потребі уточнень. Контролює наявність товарів і статуси замовлень.	- Перевірка правильності оформлення замовлення - Контроль залишків товарів - Зміна статусів замовлень - Спілкування з клієнтами
Адміністратор	Керує асортиментом товарів у системі. Забезпечує їхню актуальність, редагує або видаляє інформацію про наявні позиції.	- Додавання нових товарів - Редагування існуючих товарів - Видалення недоступних або застарілих товарів

Система повинна також реалізовувати механізми фільтрації й пошуку, що значно полегшують навігацію у великій кількості товарів. Крім того, важливою складовою є можливість експорту даних, зокрема створення звітів і накладних у форматі Excel чи PDF, що спрощує внутрішній документообіг.

У результаті, інформаційна система повинна вирішувати такі основні задачі: оптимізувати процес продажу комп’ютерної техніки, скоротити витрати часу на обробку замовлень, зменшити кількість помилок, пов’язаних з людським фактором, та покращити взаємодію з клієнтами. Завдяки автоматизації рутинних процесів і зручному інтерфейсу для всіх типів користувачів, система сприятиме

підвищенню ефективності бізнесу та забезпечить якісніший сервіс для кінцевого споживача.

1.2 Аналіз вимог до програмної системи

Розробка інформаційної системи з продажу комп’ютерної техніки вимагає чіткого формулювання вимог, які визначають її основну функціональність, поведінку, якість, ефективність та зручність для користувача. На основі аналізу предметної області, цільової аудиторії та стандартів розробки сучасного програмного забезпечення, були сформульовані як функціональні, так і нефункціональні вимоги до системи.

На основі аналізу ролей, процесів і взаємодій було сформульовано перелік функціональних (див. таблицю 2) та нефункціональних (див. таблицю 3) вимог, які має реалізовувати система.

Функціональні вимоги

Таблиця 2

№	Вимога	Опис
1	Реєстрація та авторизація	Користувачі мають змогу створювати облікові записи та входити до системи через логін і пароль.
2	Перегляд товарів	Каталог товарів доступний для перегляду всім зареєстрованим користувачам із повною інформацією про кожен товар.
3	Пошук та фільтрація	Пошук товарів за назвою, категорією, виробником, ціною.
4	Кошик	Можливість додавати товари до кошика, змінювати кількість та видаляти їх перед оформленням замовлення.
5	Оформлення замовлення	Користувач формує замовлення, вказує контактні дані, спосіб доставки, та переходить до оплати.
6	Оплата	Інтеграція з платіжними системами (банківські картки, онлайн-оплата).
7	Перегляд історії замовлень	Користувач має доступ до попередніх замовлень, їх статусів та деталей.
8	Обробка замовлень (Менеджер)	Перевірка замовлень, зміна статусу, зв’язок із клієнтом при необхідності.
9	Управління товарами (Адміністратор)	Додавання, редагування, видалення товарів, оновлення цін, фото, залишків.
10	Експорт даних	Вивантаження звітів (Excel, CSV) про товари, клієнтів або замовлення.

Нефункціональні вимоги

Таблиця 3

№	Категорія	Вимога
1	Зручність	Інтерфейс повинен бути інтуїтивно зрозумілим і доступним як для новачків, так і для досвідчених користувачів.
2	Продуктивність	Система повинна працювати швидко
3	Масштабованість	Архітектура дозволяє додавання нових функцій без серйозних змін у системі.
4	Безпека	Захист даних користувачів (HTTPS, шифрування паролів, захист від атак).
5	Надійність	Під час збою система повинна зберігати дані та мінімізувати втрату інформації.
6	Сумісність	Коректна робота на всіх популярних браузерах і пристроях.
7	Обслуговування	Код повинен бути структурованим, документованим, зручним для тестування та оновлення.

Ці вимоги створюють основу для подальшого проєктування архітектури системи, побудови бази даних, вибору технологій реалізації та формування плану тестування. Їх чітке визначення на ранніх етапах дозволяє уникнути багатьох помилок у процесі розробки.

1.3 Моделювання предметної області

Для повного розуміння логіки функціонування інформаційної системи та структури її основних об'єктів було побудовано діаграму класів у нотації UML (див. рис. 1). Вона відображає ключові сутності предметної області, їхні властивості, зв’язки між класами та типи відношень.

У центрі моделі знаходиться клас **SiteUser**, який представляє зареєстрованого користувача системи. Користувач має такі властивості, як електронна пошта, номер телефону та пароль. З кожним користувачем пов’язані кілька об'єктів: адреси доставки (**UserAddress**), кошик (**ShoppingCart**), замовлення (**ShopOrder**), платіжні методи (**UserPaymentMethod**) і залишені відгуки (**UserReview**).

ShoppingCart є віртуальним кошиком користувача, який містить **ShoppingCartItem** — об'єкти, що вказують на конкретні товари (**ProductItem**) та їх кількість. Ці товари є представленням конкретних одиниць продукції, які мають артикул (SKU), кількість на складі, зображення та ціну.

Кожен **ProductItem** пов’язаний із загальною інформацією про товар, яка зберігається в класі **Product**. Тут описані такі властивості, як назва, опис та основне зображення. Товари належать до певної категорії (**ProductCategory**), що дозволяє ієрархічну організацію асортименту. Крім того, товар може мати конфігурацію (**ProductConfiguration**) — варіанти (наприклад, об’єм пам’яті чи колір), представлені у вигляді **VariationOption**, що пов’язані з конкретними **Variation**.

ShopOrder відображає процес покупки та містить дату замовлення та загальну суму. Замовлення пов’язане з кількома сутностями:

OrderLine — позиції замовлення, що містять кількість і ціну кожного товару;

UserPaymentMethod — вибраний користувачем метод оплати, із зазначенням платіжної системи, номера рахунку та дати завершення;

OrderStatus — поточний стан замовлення (наприклад, нове, в обробці, доставлено);

ShippingMethod — спосіб доставки (назва, вартість);

UserAddress — адреса користувача, яка зберігається окремо у класі **Address**.

Address, у свою чергу, описує повну інформацію про адресу: номер будинку, вулицю, місто, регіон, поштовий індекс. Цей клас пов’язаний із **Country**, що дозволяє підтримку міжнародної доставки.

Платіжні методи описані окремим класом **PaymentType**, який дозволяє визначити типи оплати: наприклад, картка, банківський переказ тощо.

Модель також передбачає роботу з акційними товарами через клас **Promotion**, що зв’язаний із товарами через **PromotionCategory**. Це дозволяє реалізувати систему знижок або промо-періодів для обраних категорій товарів.

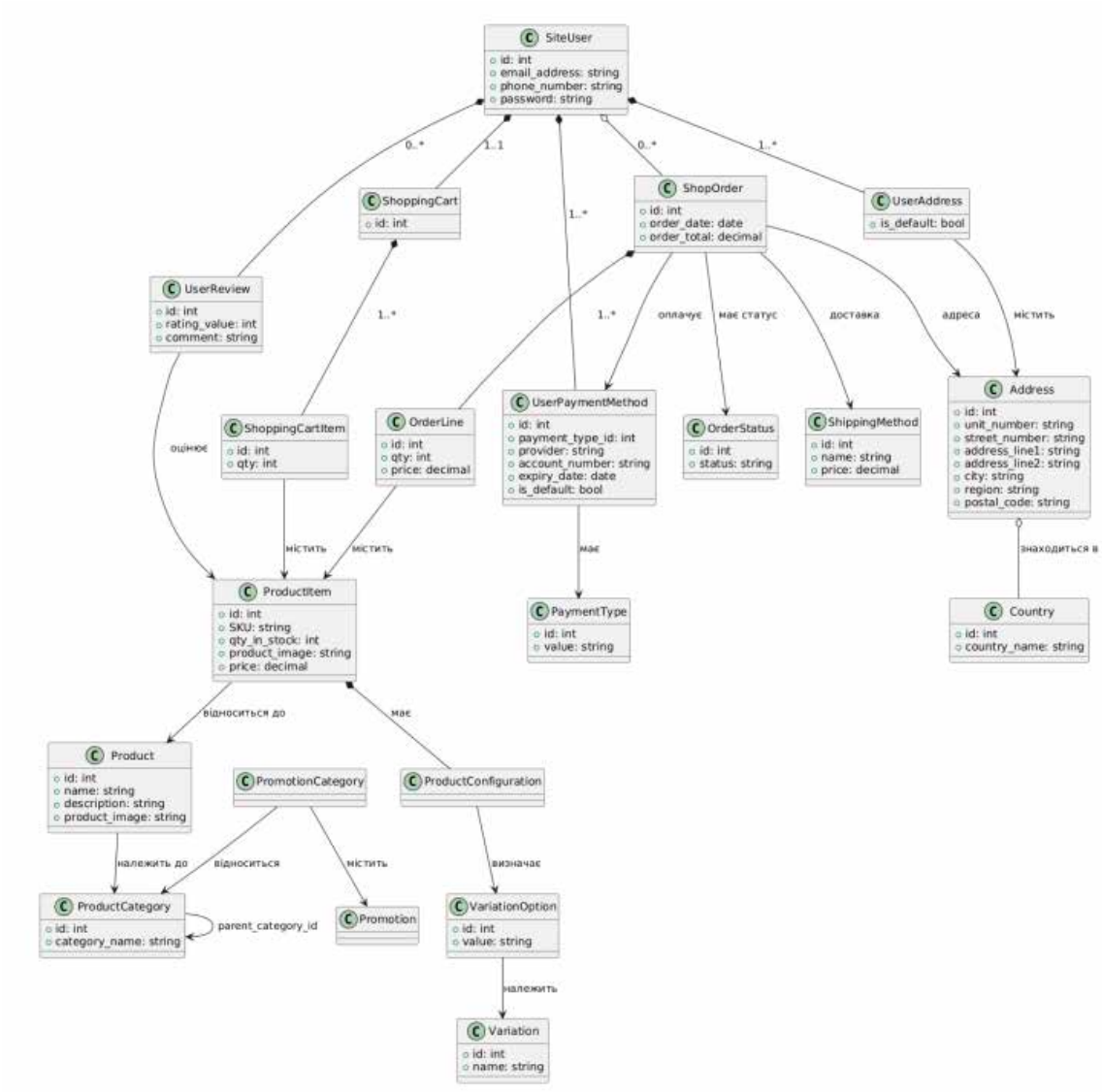

Рис. 1 Діаграма класів

Дана діаграма класів відображає не лише структуру даних, а й логіку взаємозв’язків між користувачами, товарами, замовленнями та іншими елементами системи. Вона демонструє гнучку, масштабовану архітектуру, яка дозволяє:

підтримувати різні типи товарів і конфігурацій;

організувати повний процес покупки з урахуванням способу оплати та доставки;

працювати з декількома адресами користувача;

обробляти динамічні статуси замовлень;

реалізовувати систему знижок, варіацій та категорій.

Використання такої моделі є надійною основою для побудови реляційної бази даних та логіки серверної частини майбутньої системи.

Діаграма кооперацій (див. рис. 2) демонструє бізнес-процес взаємодії між менеджером, клієнтом та основними об'єктами, які беруть участь в обробці замовлення. У центрі процесу — клас **Замовлення**, який пов’язує менеджера, клієнта, товар і доставку.

Клас **Клієнт** містить особисту інформацію та методи для зберігання контактних даних і надання інформації для звітів. **Менеджер** реалізує дії, пов’язані з обслуговуванням клієнтів та управлінням товарами, наприклад, підтвердження замовлення, зміна статусу доставки або консультування клієнтів.

Замовлення агрегує всю інформацію: номер, дату, вартість, пов’язані товари та обрану доставку. Завдяки методам, таким як зберігатиІнформацію() або згенеруватиНакладну(), система може автоматично формувати супровідні документи. Клас **Доставка** містить інформацію про адресу, статус і дати, пов’язані з відправкою, і дозволяє менеджеру відстежувати процес виконання замовлення.

Ця кооперація відображає типову ситуацію в роботі системи: клієнт оформлює замовлення, а менеджер забезпечує його повне і правильне виконання — від підтвердження до передачі на доставку.

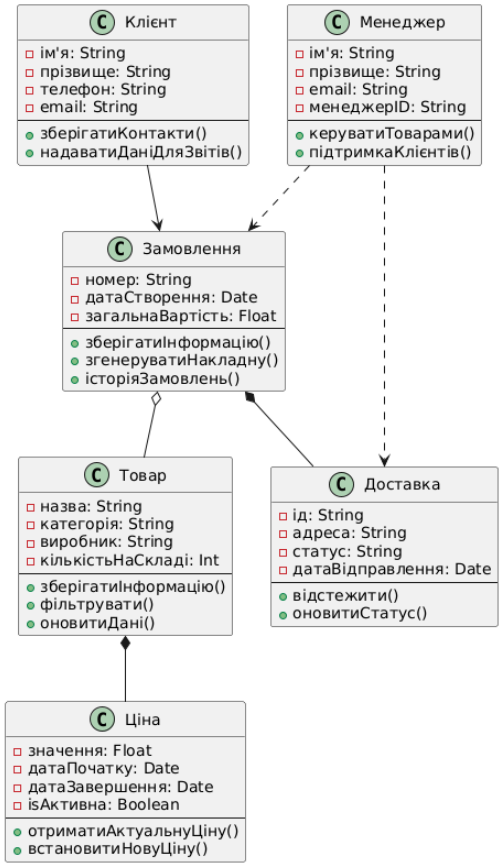

Рис. 2 Діаграма кооперцій

1.4 Огляд інформаційних джерел та існуючих рішень

Перед розробкою програмного забезпечення важливо провести огляд інформаційних джерел, вивчити досвід реалізації подібних систем, ознайомитися з технологіями, які застосовуються у цій галузі, та оцінити ефективність існуючих рішень. Такий аналіз дозволяє не лише обрати найбільш оптимальні інструменти, але й уникнути поширених помилок, які трапляються в типових реалізаціях.

В ході аналізу предметної області було розглянуто низку популярних інформаційних систем для продажу товарів онлайн, зокрема такі платформи, як **Rozetka**, **Foxtrot**, **Comfy**, а також готові CMS-рішення: **OpenCart [17]**, **Magento [18]**, **Shopify [19]** тощо. Ці системи мають певну типову структуру, яка включає каталог товарів, обробку замовлень, особистий кабінет клієнта, панель адміністратора, модулі оплати та доставки, інтеграцію зі складськими системами.

Переваги готових рішень:

швидкий старт: такі системи можна розгорнути за кілька годин і почати продавати продукцію.

багата функціональність: підтримка різних мов, валют, інтеграція з платіжними сервісами, автоматичне створення звітів, seo-інструменти.

підтримка спільноти: наявність плагінів, документації, форумів.

сучасний дизайн та адаптивність: більшість шаблонів оптимізовано під мобільні пристрої.

недоліки готових рішень:

надлишкова складність: системи можуть містити функції, які не потрібні конкретному проєкту, що ускладнює підтримку.

складність кастомізації: глибока модифікація функціоналу може вимагати значних зусиль або повного переписування коду.

безпека: відкриті cms можуть стати мішенню для хакерських атак, якщо не оновлювати їх вчасно.

залежність від сторонніх розробників: оновлення та технічна підтримка не завжди передбачають повну гнучкість.

На відміну від готових рішень, **індивідуальна розробка системи** з урахуванням специфіки предметної області дозволяє створити адаптоване рішення, що точно відповідає потребам користувачів. Наприклад, у випадку продажу комп’ютерної техніки можна врахувати такі аспекти, як складність товарів, наявність технічних характеристик, залежність ціни від комплектації, реалізацію системи технічної підтримки, повернення товару та інше.

У процесі формування технічного завдання також було опрацьовано наукові джерела та статті, які описують процес побудови інформаційних систем, структуру реляційних баз даних, реалізацію багаторівневого доступу користувачів, використання REST API, принципи побудови зручного інтерфейсу користувача та безпечної обробки платежів. Особливу увагу було приділено матеріалам з розробки систем на основі **Django**, **React**, **PostgreSQL**, оскільки ці

технології є одними з найпопулярніших для реалізації веб-додатків із високими вимогами до безпеки, масштабованості та підтримки.

Таким чином, аналіз існуючих рішень дозволив сформулювати чітке бачення майбутньої системи, виокремити функціонал, який буде доцільно реалізовувати, а також обрати набір інструментів, що сприятиме ефективному створенню адаптованої інформаційної системи продажу комп’ютерної техніки.

1.5 Постановка завдання

Розробити систему продажу комп'ютерної техніки з двума основними модулями: клієнтський та серверний. Клієнтський модуль забезпечує інтерфейс для кінцевих користувачів, через який вони можуть переглядати асортимент продукції, додавати товари в кошик, оформлювати замовлення та відстежувати статус доставки. Цей модуль також може включати функції для реєстрації користувачів, їх авторизації, управління персональними даними та взаємодії зі службою підтримки.

Серверний модуль, в свою чергу, обробляє всі запити від клієнтського модуля, управляє базою даних, обробляє платежі, моніторить статуси замовлень, а також здійснює взаємодію з іншими службами, наприклад, для управління інвентарем або для доставки товарів. Серверний модуль також відповідає за безпеку, авторизацію та аутентифікацію користувачів, а також за обробку даних і забезпечення безперебійної роботи системи. Обидва модулі повинні бути оптимізовані для високої продуктивності та безпеки.

Для досягнення цієї мети система повинна реалізовувати наступні ключові функціональні компоненти:

1. модуль обліку товарів, що дозволяє зберігати та редагувати інформацію про продукцію (назва, категорія, виробник, ціна, кількість на складі);

2. модуль реєстрації та авторизації користувачів із розподілом на ролі (клієнт, менеджер, адміністратор);

3. інтерфейс для клієнтів, що забезпечує перегляд каталогу товарів, пошук, фільтрацію, додавання до кошика, оформлення та оплату замовлення;

4. інтерфейс для менеджера з можливістю обробки замовлень, перегляду історії замовлень, зміни їх статусу;

5. інтерфейс адміністратора для управління товарами — додавання, редагування, видалення та експортування даних;

6. система звітності, що дозволяє формувати накладні, переглядати історію покупок, аналізувати залишки на складі.

На основі поставлених функціональних вимог та опису основних сценаріїв взаємодії користувачів із системою, технічне завдання на розробку включає:

розробити структуру бази даних, яка дозволяє зберігати інформацію про товари, замовлення, користувачів, ролі, історію транзакцій;

реалізувати користувацький вебінтерфейс, адаптований для трьох основних ролей — клієнта, менеджера та адміністратора;

забезпечити захищену систему реєстрації та автентифікації з можливістю зберігання персональних даних клієнтів та їх замовлень;

інтегрувати модулі пошуку та фільтрації, які дозволяють зручно знаходити потрібний товар у каталозі;

реалізувати процес оформлення замовлення, що включає додавання товарів до кошика, вказання контактної інформації, підтвердження та збереження замовлення в системі;

забезпечити можливість перегляду статистики продажів та експорту даних в Excel;

побудувати архітектуру системи на сучасних вебтехнологіях, що підтримують масштабованість, модульність і забезпечують високу швидкість обробки запитів.

Таким чином, у межах цієї дипломної роботи ставиться завдання спроєктувати, розробити, протестувати та задокументувати інформаційну

систему, яка реалізує повний цикл онлайн-продажу комп’ютерної техніки, враховуючи потреби користувачів та технічні вимоги до сучасного вебсервісу.

2. ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Проєктування інформаційної системи є одним із ключових етапів розробки, оскільки саме на цьому етапі закладається основа для подальшої реалізації програмного забезпечення. Важливо не лише визначити логіку функціонування окремих компонентів, а й грамотно спроєктувати структуру збереження даних, вибрати надійну систему управління базами даних, а також забезпечити ефективну інтеграцію між кодом і базою даних.

2.1 Логічна модель даних у вигляді ER-діаграми

Для побудови ефективної інформаційної бази було розроблено ER-діаграму (див. рис. 3), яка візуалізує основні сутності системи, їх атрибути та зв’язки між ними. Побудова цієї діаграми допомагає краще зрозуміти логіку предметної області, формалізувати залежності між таблицями та уникнути помилок при фізичному проєктуванні бази даних.

У центрі моделі знаходиться сутність **site_user**, яка представляє користувача системи. Користувач може мати кілька адрес (**user_address**), платіжних методів (**user_payment_method**) та замовлень (**shop_order**). З кожною адресою пов’язана сутність **address**, яка включає місто, вулицю, індекс тощо, а також зовнішній ключ до країни (**country**).

Замовлення (shop_order) має атрибути дати та загальної суми. Воно пов’язане з:

- **order_line** — список товарних позицій у замовленні;
- **shipping_method** — спосіб доставки;
- **order_status** — статус обробки замовлення;
- **user_payment_method** — платіжний метод, обраний користувачем для цього замовлення.

Сутність **shopping_cart** містить товари, які користувач додав до кошика. Кожен кошик складається з одного або кількох **shopping_cart_item**, що вказують на конкретний товар (**product_item**).

product_item описує SKU, наявність, зображення та ціну конкретного товару. Він пов’язаний з базовою сутністю **product**, яка містить загальну інформацію про товар. Кожен товар належить до **product_category** — ієрархічної категорії товарів.

Також підтримуються знижки та акції через сутності **promotion** і **promotion_category**, а також конфігурації товару (**product_configuration**) з можливими варіаціями (**variation**, **variation_option**).

ER-діаграма демонструє складну, але гнучку модель даних, яка дозволяє підтримувати багатофункціональний магазин із варіативними товарами, складною системою замовлень і персоналізованим обслуговуванням користувача.

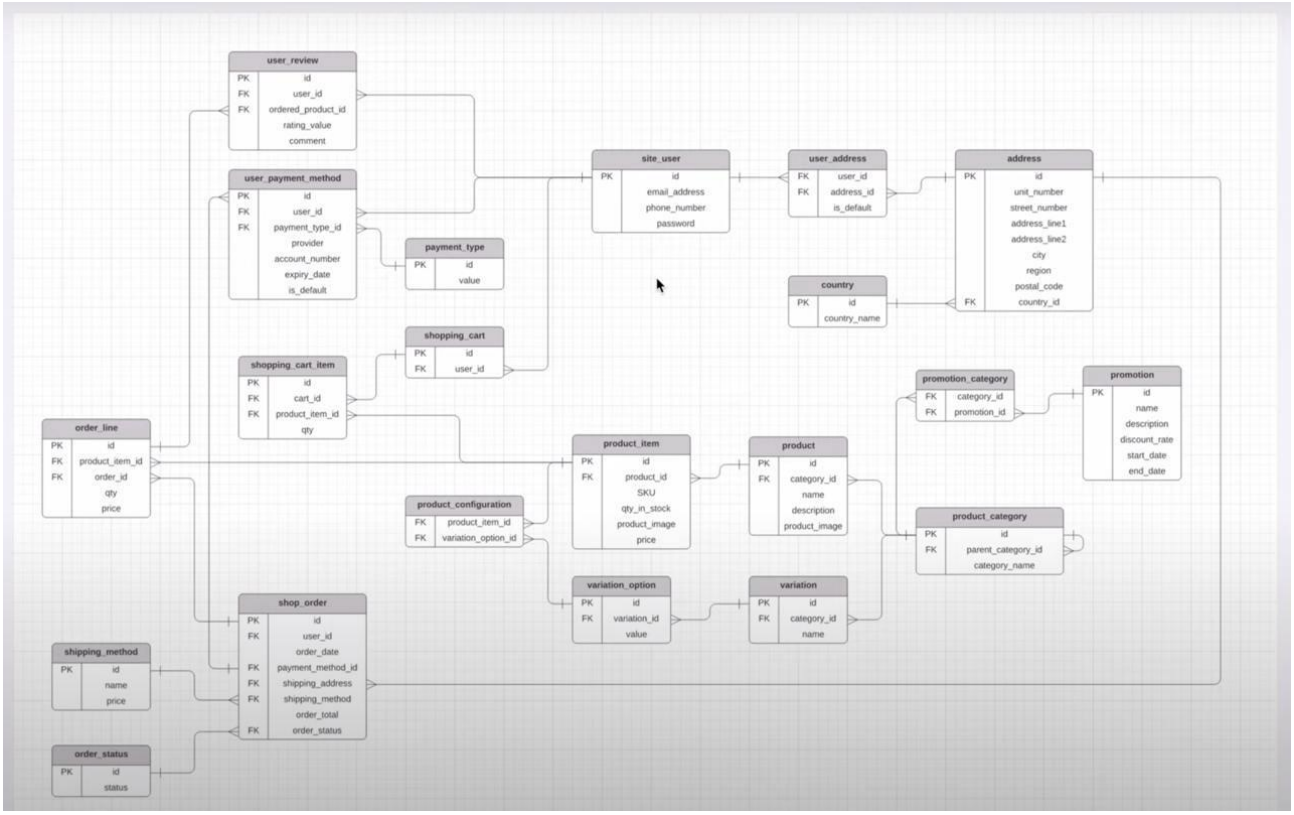

Рис. 3 ER діаграма

2.2 Вибір системи управління інформаційною базою

У якості системи управління базами даних (СУБД) для цього проєкту було обрано **PostgreSQL**. Це об'єктно-реляційна СУБД із відкритим вихідним кодом, яка є однією з найнадійніших і наймасштабованіших систем у світі. Вибір PostgreSQL зумовлений її широкими функціональними можливостями, стабільною роботою в умовах високого навантаження та чудовою інтеграцією з Django ORM.

Переваги PostgreSQL:

підтримка складних запитів: можливість використовувати вкладені підзапити, віконні функції, агрегати тощо.

надійність і транзакційність: підтримка ACID-принципів, забезпечення цілісності даних.

масштабованість: ефективна робота як на малих проєктах, так і в системах корпоративного рівня.

безпека: вбудована система ролей, аутентифікація, підтримка ssl.

можливість розширення: створення користувацьких типів даних, функцій, індексів.

Структура бази даних PostgreSQL:

таблиця (table): основна структура зберігання даних.

схема (schema): логічна група таблиць, що дозволяє структурувати базу даних.

індекс (index): спеціальна структура для оптимізації пошуку.

тригери (trigger): реакція на події в таблицях (insert, update, delete).

вигляди (view): віртуальні таблиці для зручності запитів.

PostgreSQL також чудово підтримує роботу з JSON, що дозволяє зберігати напівструктуровані дані. Це особливо зручно при розширенні функціоналу без зміни схеми бази даних.

У нашій системі PostgreSQL відповідає за зберігання всієї інформації: про користувачів, товари, замовлення, платежі, доставку тощо. Це дозволяє

забезпечити цілісність, стабільність і високу продуктивність системи навіть при зростанні обсягу даних.

2.3 Створення інформаційної бази

Для взаємодії з PostgreSQL у системі використовується **Django ORM** (Object-Relational Mapping) — вбудована система Django для зв’язку об’єктів Python з реляційною базою даних. Вона дозволяє розробнику працювати із даними як із об'єктами Python, не використовуючи SQL-запити напряму.

Що таке ORM? Object-Relational Mapping — це програмний підхід, який дає змогу описати структуру бази даних мовою об'єктів (у нашому випадку Python) і автоматично транслювати операції над об’єктами в SQL-запити до бази даних.

Переваги ORM:

1. Абстрагування від SQL: розробник оперує класами, а не таблицями;
2. Переносимість: можна легко змінити СУБД без зміни коду;
3. Вбудована валідація, обмеження, зв’язки між моделями;
4. Автоматична генерація міграцій і оновлень структури бази.

Процес створення бази даних у Django:

1. **Опис моделей:** для кожної сутності створюється Python-клас (наприклад, Product, Order, SiteUser). Кожен атрибут класу — це поле у таблиці.
2. **Створення міграцій:** команда python manage.py makemigrations створює файли міграцій — опис змін структури бази даних.
3. **Застосування міграцій:** команда python manage.py migrate застосовує ці зміни, створюючи або оновлюючи таблиці в PostgreSQL.
4. **Адміністративна панель:** Django автоматично створює CRUD-інтерфейс для всіх моделей, що дає змогу перевіряти дані, не створюючи окремих сторінок керування.

Взаємодія Django ORM і PostgreSQL: При зверненні до бази даних Django ORM самостійно формує відповідні SQL-запити. Наприклад, метод Product.objects.all() формує SELECT * FROM product, а Product.objects.filter(name="Монітор") — SELECT * FROM product WHERE name='Монітор'. ORM також автоматично підтримує зовнішні ключі, об'єднання (JOIN) та багато інше.

Це значно спрощує розробку, зменшує кількість помилок при роботі з даними і дозволяє розробникам зосередитися на бізнес-логіці, а не на низькорівневому синтаксисі SQL.

Використання Django ORM у зв'язці з PostgreSQL забезпечує гнучке та масштабоване рішення, яке легко підтримується, розширюється і дозволяє реалізовувати як прості, так і складні операції з базою даних без втрати продуктивності. Це дозволяє досягти високого рівня автоматизації в процесі розробки та забезпечити стабільну роботу системи у майбутньому.

3. ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Вибір інструментарію для розробки програмного забезпечення

При створенні інформаційної системи з продажу комп’ютерної техніки важливим етапом є вибір технологічного стеку. Правильно підібрані інструменти дозволяють забезпечити високу продуктивність, безпеку, масштабованість та зручність у подальшому супроводі.

Back-end: Django

Основою серверної частини було обрано фреймворк **Django**, який написаний мовою програмування **Python**. Django — це високорівнева система розробки веб-застосунків, яка дозволяє будувати надійні серверні додатки швидко та безпечно. Django дотримується архітектурного патерну **MTV (Model-Template-View)**, що є логічним продовженням MVC, адаптованим під веб-середовище.

Model відповідає за опис структури даних, взаємодію з базою даних через ORM.

Template — це HTML-шаблони для відображення даних (хоча в проєкті основна відповідальність інтерфейсу лежить на React).

View — обробник запитів, який отримує дані та передає їх далі через серіалізатори або шаблони.

Також Django має потужну адміністративну панель, систему автентифікації, підтримку форм, URL-роутінгу та інтеграцію з REST API через Django REST Framework.

Django REST Framework (DRF)

DRF — це бібліотека для створення RESTful API. Вона забезпечує серіалізацію даних (перетворення моделей у формат JSON), підтримку аутентифікації, пагінацію, фільтрацію, throttle-контроль (обмеження частоти

запитів) тощо. Через DRF реалізовано API для обміну даними між клієнтською та серверною частинами. Це дозволяє frontend-розробникам працювати незалежно, використовуючи React для побудови UI.

REST API (Representational State Transfer) — це архітектурний стиль, який визначає принципи взаємодії клієнт-сервер: без збереження стану, використання стандартних HTTP-методів (GET, POST, PUT, DELETE), ідентифікаторів ресурсів (URL), відповідей у форматі JSON або XML.

Front-end: React

Інтерфейс користувача реалізовано за допомогою JavaScript-бібліотеки **React**, яка дозволяє будувати динамічні, компонентні веб-додатки. Основні переваги React:

1. Віртуальний DOM, що забезпечує швидке оновлення сторінки без повного перезавантаження;
2. Компонентна архітектура — код поділений на повторно використовувані частини (наприклад, кнопки, форми, картки товарів);
3. Підтримка маршрутизації через React Router;
4. Можливість використання сучасних бібліотек (Axios, Redux, TailwindCSS).

PostgreSQL

Як реляційну систему керування базами даних обрано **PostgreSQL**, яка добре інтегрується з Django, підтримує транзакції, типи даних JSON, індексацію та складні SQL-запити. PostgreSQL є безкоштовною, стабільною і розширюваною системою, що ідеально підходить для ecommerce-сервісів.

Інфраструктура

Docker: використовується для контейнеризації всіх сервісів. Завдяки Docker можна створити уніфіковане середовище для розробки, тестування й продакшену.

Celery + Redis: для виконання асинхронних задач, таких як надсилання email, оновлення звітів, опрацювання аналітики.

Sentry: сервіс для моніторингу помилок. Дає змогу в реальному часі відстежувати виключення, логи та інші збої.

3.2 Принципи роботи у середовищі візуальної розробки програм

Архітектура та шаблони Django:

У Django логіка додатку поділяється за файлами:

models.py — моделі даних;

views.py — функції або класи, що обробляють запити;

serializers.py — перетворення моделей у формат JSON (у DRF);

urls.py — маршрутизація запитів;

admin.py — опис адміністраторської панелі;

forms.py — опис форм для HTML-інтерфейсу (опціонально).

Для взаємодії з базою даних використовується **ORM (Object-Relational Mapping)**. Django ORM дозволяє маніпулювати таблицями за допомогою Python-класів. Наприклад, щоб створити новий товар, достатньо викликати Product.objects.create(...). ORM автоматично транслює цю команду в SQL-запит.

Міграції (python manage.py makemigrations, migrate) — механізм, що дозволяє зручно створювати або змінювати структуру БД відповідно до змін у моделях. Django самостійно відстежує зміни в структурах даних і генерує SQL-інструкції для оновлення схеми бази.

Опис діаграми компонентів

Діаграма компонентів (див. рисунок) наочно демонструє, як різні частини системи взаємодіють між собою. Вона показує, як дані проходять від користувача через вебінтерфейс до контролерів API, потрапляють до бізнес-логіки, після чого зберігаються в базі або передаються на обробку через Celery.

Центральним вузлом є **API**, через який здійснюється більшість запитів. Це дозволяє повністю відокремити клієнтську частину від серверної, що полегшує масштабування, підтримку та тестування системи. Подібна структура відповідає

принципам **Clean Architecture** та **Separation of Concerns**, що забезпечують високу якість коду та зниження кількості помилок.

Роль компонентів у загальній структурі проєкту

На діаграмі компонентів, поданій нижче, відображено загальну архітектуру інформаційної системи:

Клієнт (браузер) взаємодіє з вебінтерфейсом, написаним на **React**. Через API запити надсилаються на сервер.

API/Контролери (Serializers, URLs) забезпечують маршрутизацію запитів, авторизацію, фільтрацію та форматування відповідей.

Сервер застосунку (Django Backend) обробляє логіку запитів, у тому числі:

1. **Веб-додаток (Views, Forms, Admin)** — реалізує адміністративний інтерфейс та логіку обробки сторінок.
2. **Сервіси (Оплати, Замовлення, Товари)** — модулі, що відповідають за функціональну частину.
3. **Аутентифікація (Токени, Сесії)** — реалізована через JWT або Session middleware.
4. **База даних** — PostgreSQL, яка зберігає всі структуровані дані.

Інфраструктурні компоненти — **Docker**, **Celery**, **Redis**, **Sentry** — розгорнуті як окремі сервіси, що взаємодіють із основним застосунком, але не залежать від нього прямо.

Інтеграції в системі

Для забезпечення додаткових можливостей було реалізовано кілька інтеграцій:

1. **Stripe API** — для обробки онлайн-платежів;
2. **Email SMTP** — для надсилання повідомлень про підтвердження замовлення;
3. **Postman Collection** — для тестування API;

4. **Swagger/OpenAPI** — генерація документації до API автоматично на основі DRF.

Також система підтримує інтеграцію з хмарними платформами (наприклад, AWS або Heroku) у разі потреби розгортання у продакшені.

Принципи роботи з Git та CI/CD

Проєкт використовує систему контролю версій Git. Код розробляється у гілках develop, feature/* та main. Автоматизація тестування та розгортання можлива через CI/CD-платформи, такі як GitHub Actions або GitLab CI. Контейнери Docker забезпечують ізольованість середовища.

Обраний стек технологій повністю відповідає вимогам сучасної веброзробки, дозволяє реалізувати гнучку, масштабовану і зручну в обслуговуванні систему. Компонентна структура, архітектурні патерни та підтримка автоматизації (CI/CD, Docker) роблять платформу готовою до розширення і використання у реальних умовах.

Головна сторінка (Home Page) (див. рис. 4)

На рисунку 4 зображена головна сторінка інтернет-магазину "Computer Shop". Вона містить вітальне повідомлення для користувача та кнопку «Shop Now», яка перенаправляє відвідувача до каталогу продукції. Верхнє меню містить логотип, посилання на сторінку товарів, іконку кошика та можливість входу в систему. Також на головній сторінці представлені категорії товарів і розділ із продуктами. Це покращує зручність навігації та дозволяє швидко знайти популярні товари.

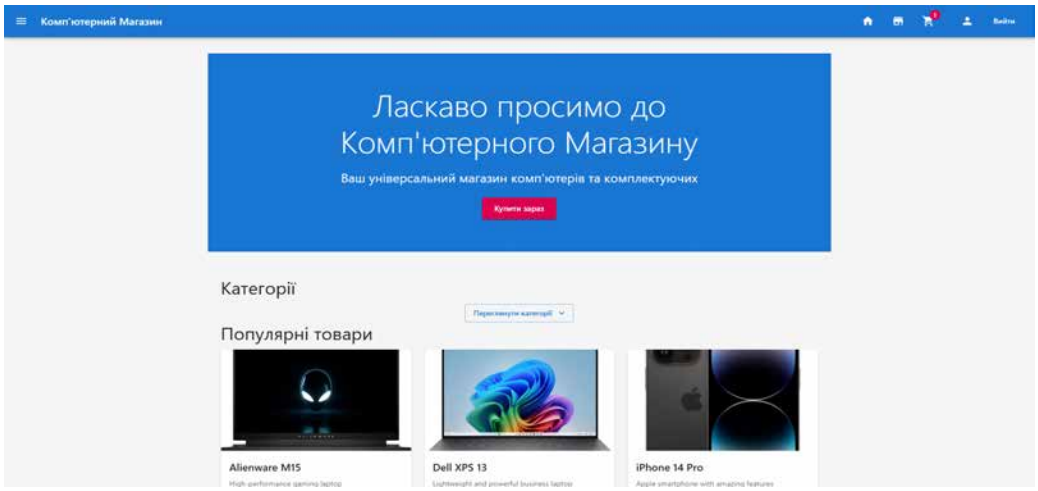

Рис. 4 Головна сторінка

Каталог товарів із фільтрацією (Products Page with Filters) (див. рис. 5)

Цей інтерфейс реалізує повноцінну систему фільтрації товарів за кількома критеріями: назвою, категорією та діапазоном цін. Праворуч виводяться товари, що відповідають обраним параметрам, у вигляді карток. Кожна картка містить зображення, назву, короткий опис та ціну продукту. Також реалізоване сортування (Sort By), що дозволяє впорядкувати результати за назвою або ціною. Завдяки цьому компоненту система забезпечує високу зручність пошуку товарів, особливо при великому асортименті.

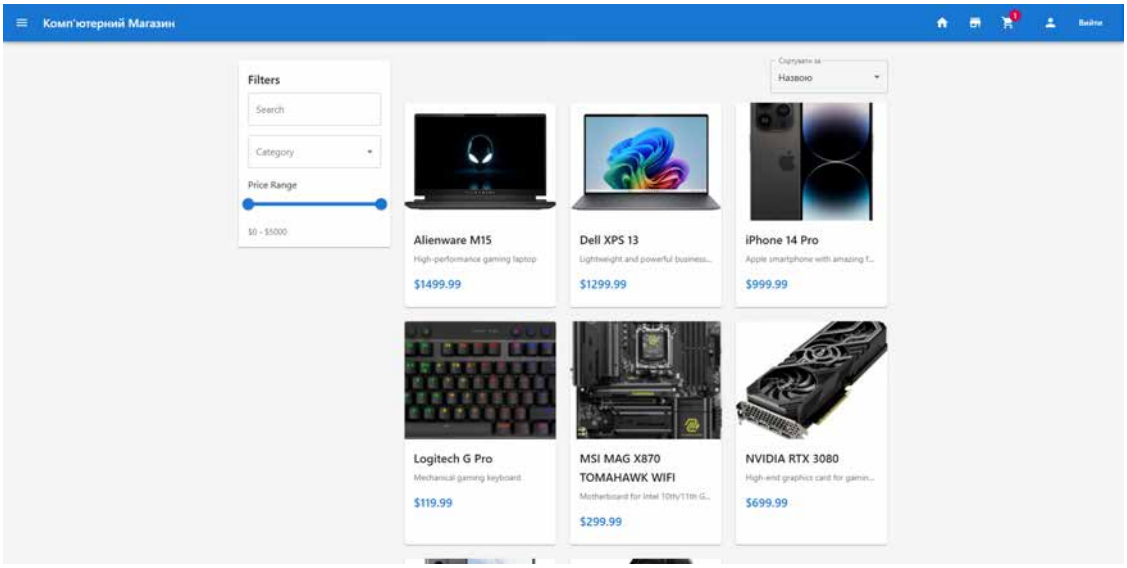

Рис. 5 Каталог товарів з фільтрацією

Сторінка входу (Login Page) (див. рис. 6)

Цей екран дає змогу зареєстрованим користувачам увійти в систему, вказавши електронну адресу та пароль. Інтерфейс реалізовано в мінімалістичному стилі з акцентом на зручність введення даних. Також присутнє посилання для переходу до форми реєстрації, що полегшує залучення нових користувачів. Уся логіка авторизації реалізована через API, що дозволяє масштабувати автентифікацію та підключати сторонні сервіси (наприклад, OAuth 2.0).

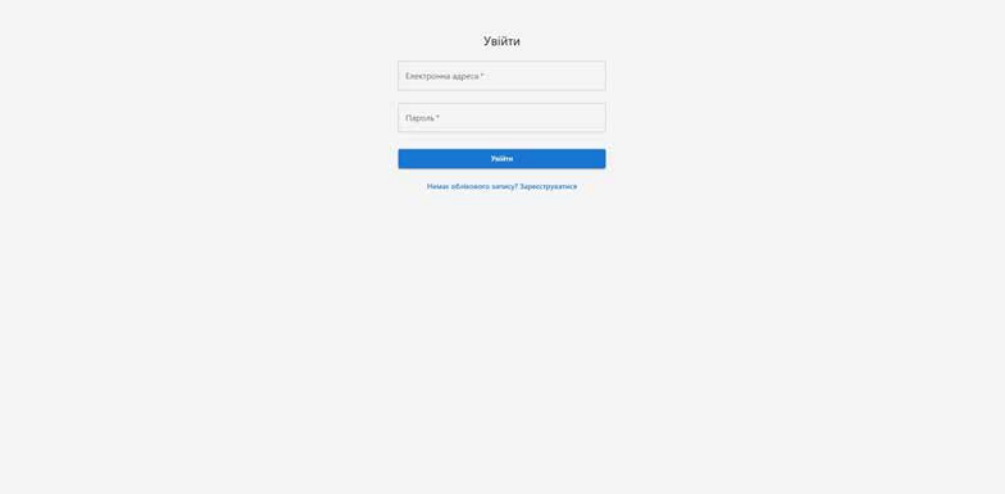

Рис. 6 Сторінка входу

Сторінка реєстрації (Sign Up Page) (див. рис. 7)

Форма реєстрації включає основні поля: ім’я, прізвище, email, номер телефону та пароль із підтвердженням. Система забезпечує базову валідацію введених даних на клієнтському рівні. Після заповнення форми дані надсилаються через API і зберігаються в базі даних PostgreSQL. Це забезпечує зручний вхід для користувачів і підвищує безпеку реєстраційного процесу.

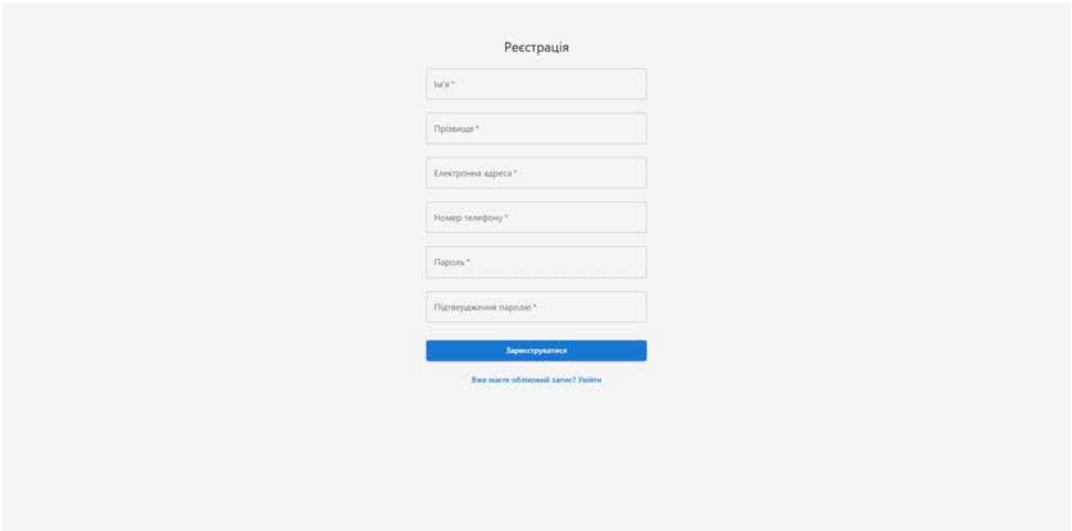

Рис. 7 Сторінка реєстрації

Кошик покупок (Shopping Cart) (див. рис. 8)

У кошику відображається обраний товар із його кількістю, ціною за одиницю та підсумковою вартістю. Користувач може змінити кількість або видалити товар із кошика. Праворуч міститься секція з підсумком замовлення та кнопкою переходу до оформлення (Proceed to Checkout). Це логічне завершення

процесу перегляду й вибору товару, яке веде до подальшого оформлення покупки.

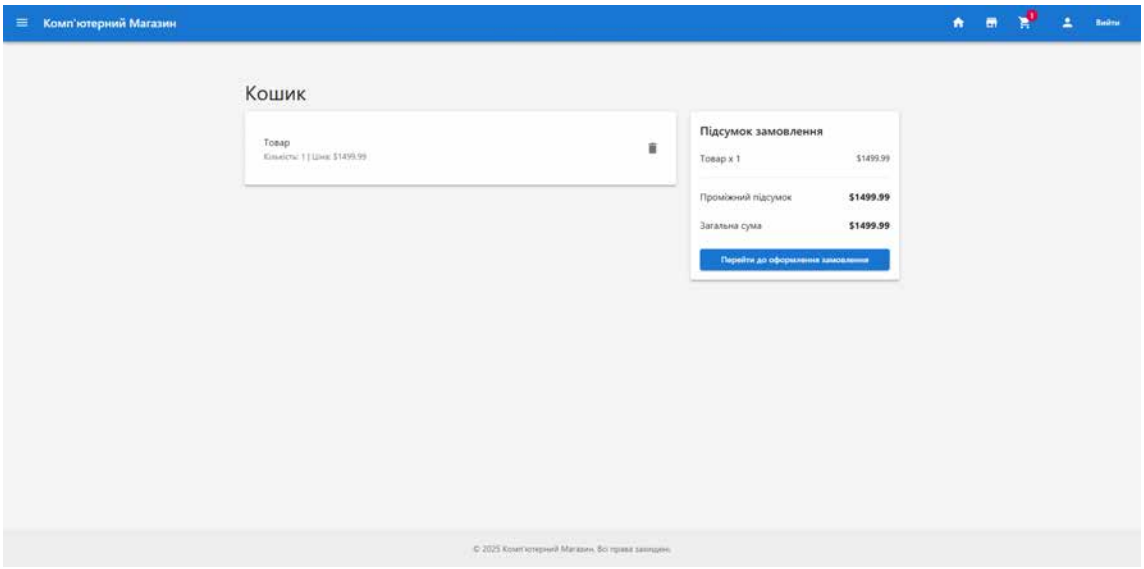

Рис. 8 Кошик покупок

Оформлення замовлення: Доставка (Checkout – Shipping) (див. рис. 9)

Цей етап оформлення замовлення дає змогу ввести адресу доставки та вибрати метод доставки. Інформація вводиться вручну або обирається з доступних адрес, збережених раніше. У правій частині відображається загальна інформація про замовлення та вартість доставки. Це дозволяє прозоро показати клієнту підсумкову вартість до моменту оплати.

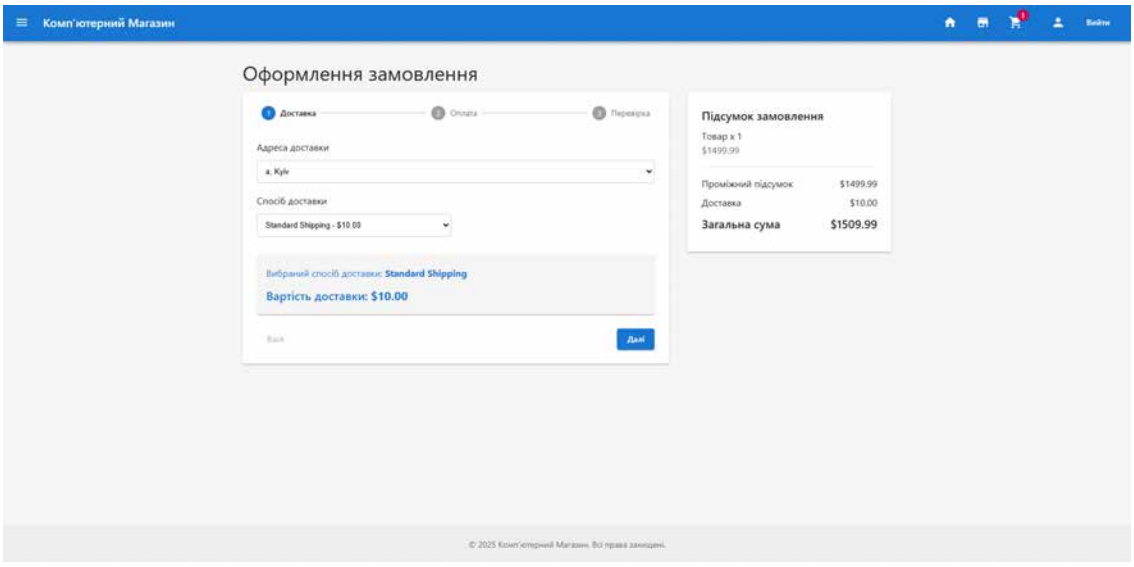

Рис. 9 Оформлення замовлення: Доставка

Оформлення замовлення: Оплата (Checkout – Payment) (див. рис. 10)

На етапі оплати користувач обирає метод платежу (у прикладі — «Credit Card»). Після підтвердження цього кроку система переходить до перегляду замовлення. Така послідовність етапів допомагає уникнути помилок і дозволяє клієнту перевірити всі деталі перед остаточним підтвердженням.

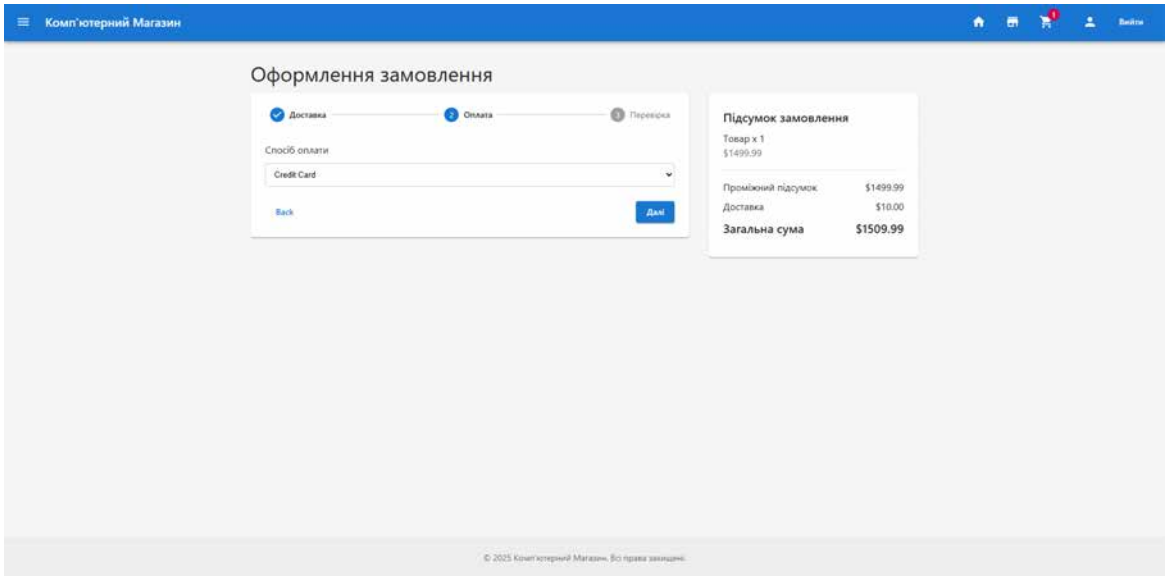

Рис. 10 Оформлення замовлення: Оплата

Оформлення замовлення: Підтвердження (Checkout – Review & Submit) (див. рис. 11)

Користувач бачить повний підсумок замовлення: перелік товарів, кількість, вартість доставки, підсумкова сума. Після натискання кнопки «Place Order» замовлення зберігається в базі, а користувач отримує повідомлення. Це завершує процес покупки.

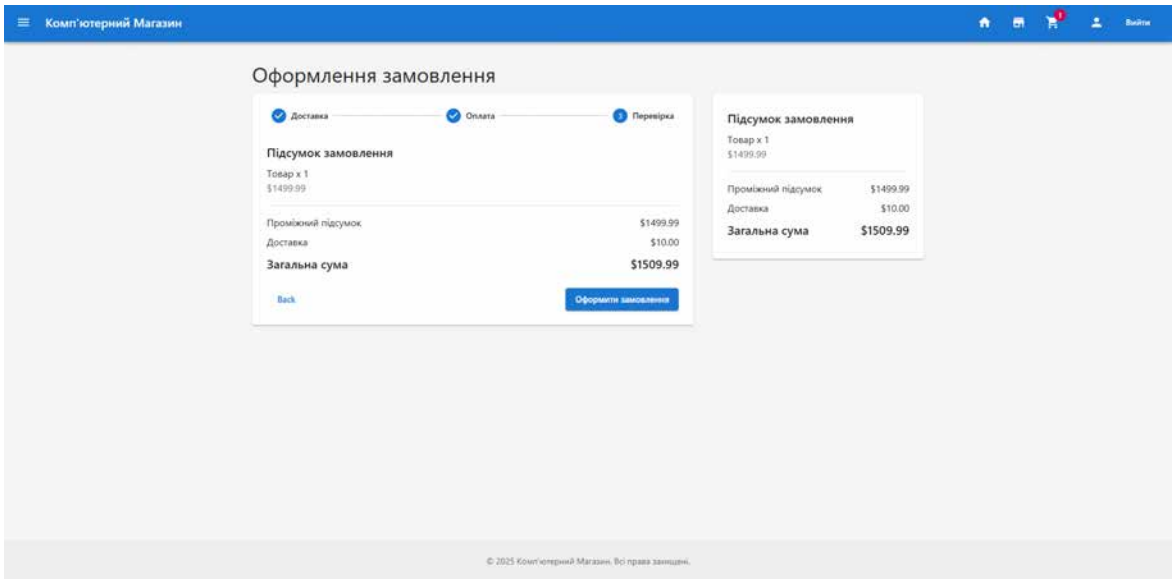

Рис. 11 Оформлення замовлення: Підтвердження

Сторінка оплати через Stripe (Stripe Checkout) (див. рис. 12)

Цей скріншот показує інтеграцію із зовнішнім платіжним сервісом **Stripe**. На екрані користувач вводить платіжні реквізити, обирає валюту та підтверджує платіж. Stripe гарантує безпечну обробку транзакцій, шифрування даних та захист від шахрайства. Інтеграція реалізована через Stripe API та webhook-и, які підтверджують оплату в системі.

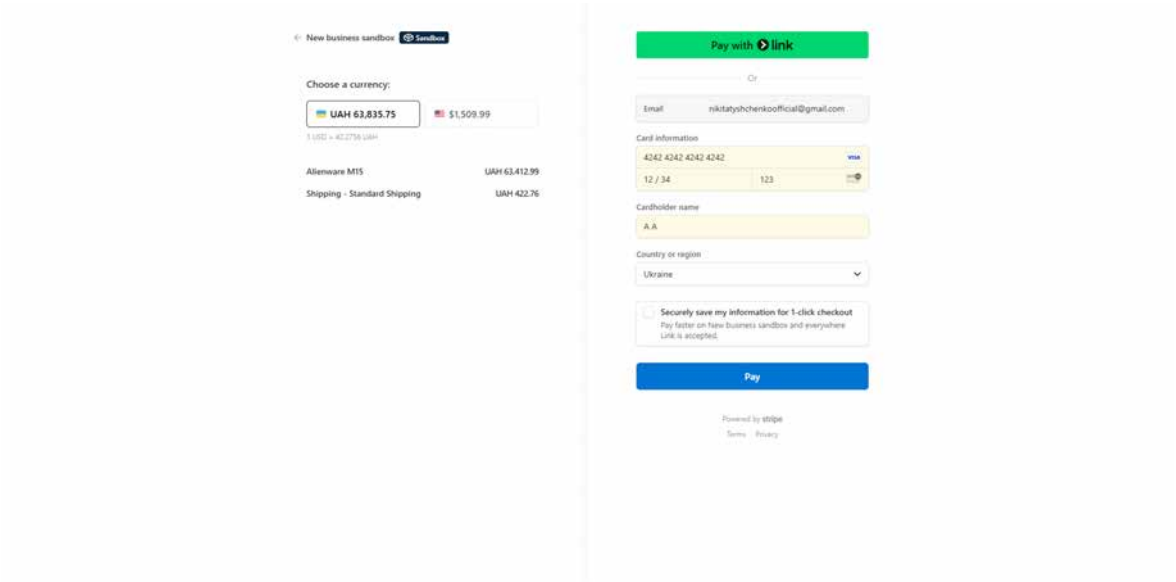

Рис. 12 Сторінка оплати через Stripe

Електронне підтвердження замовлення (Email Notification) (див. рис. 13)

Після успішної оплати користувач отримує автоматичний email із підтвердженням замовлення. У повідомленні вказано деталі: номер замовлення, дата, спосіб доставки, спосіб оплати, адреса та вміст замовлення. Надсилання реалізовано через SMTP-сервер та фонову задачу Celery. Це формує довіру до сервісу та забезпечує збереження квитанції для користувача.

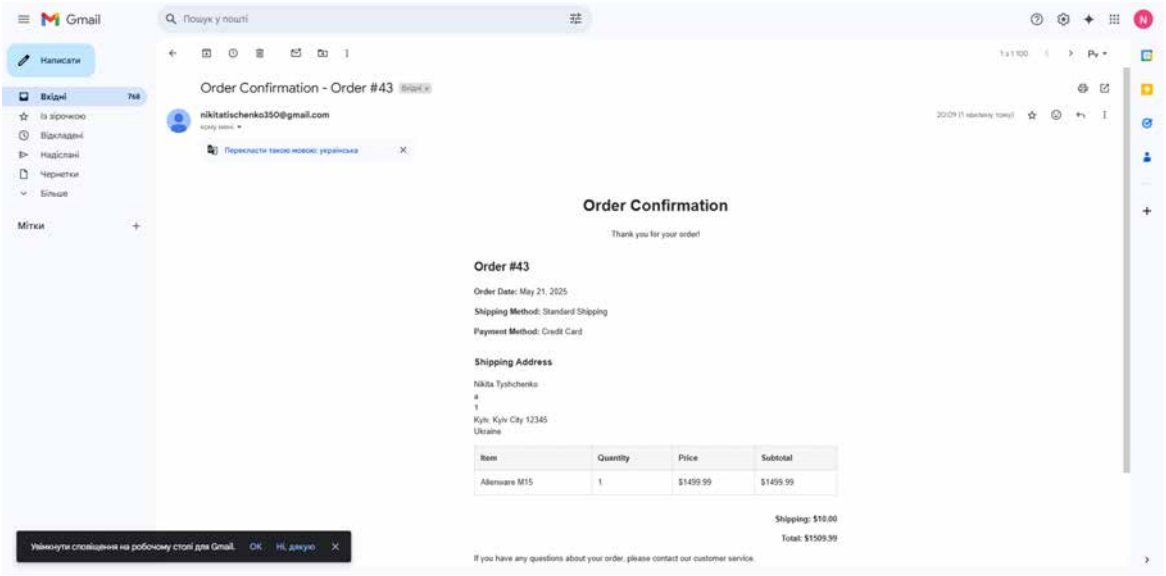

Рис. 13 Електронне підтвердження замовлення

4. РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування інформаційної системи є одним із ключових етапів її життєвого циклу. Метою тестування є виявлення та усунення помилок до впровадження системи у виробниче середовище. Якісне тестування дозволяє гарантувати стабільність, безпечність і відповідність функціоналу системи заявленим вимогам.

У процесі розробки програмного забезпечення для інформаційної системи з продажу комп'ютерної техніки було реалізовано кілька типів тестування, що охоплюють різні рівні роботи системи.

Перш за все, було проведено **модульне тестування (unit testing)**. Його основна мета — перевірка роботи окремих функцій або методів у відриві від решти системи. Це дозволяє локалізовано виявити помилки на ранньому етапі. У проєкті використовувався фреймворк **Pytest**, який дозволяє писати компактні та читабельні тести. Наприклад, тестувались функції підрахунку вартості замовлення, додавання товарів у кошик, валідація введених даних та інші критично важливі частини бізнес-логіки. Завдяки автоматизованим тестам розробники мали можливість швидко перевірити, що зміни у коді не призвели до порушення роботи існуючого функціоналу (регресійне тестування).

Другий етап — **інтеграційне тестування**, яке перевіряє взаємодію між модулями системи. Наприклад, після підтвердження замовлення потрібно пересвідчитись, що інформація коректно передається до бази даних, відбувається зменшення кількості товару на складі, викликається обробка платіжної транзакції та формується запис для логістики. Такі складні бізнес-

ланцюги вимагають тестування на інтеграційному рівні, оскільки помилка на будь-якому етапі може вплинути на роботу всієї системи.

Тестування **API** проводилось за допомогою платформи **Postman**, яка дозволяє моделювати запити до REST-інтерфейсу системи. Було створено набір запитів (колекція), що охоплюють основні сценарії: отримання списку товарів, створення замовлення, додавання клієнтів, реєстрація тощо. Для кожного ендпоінту перевірялися відповіді на валідні та невалідні запити, формат повернених даних, відповідність статус-кодів (200, 400, 404, 500 тощо). Це дозволяє забезпечити надійну взаємодію між фронтендом, мобільним застосунком або зовнішніми сервісами з бекендом.

Також важливим етапом було **тестування інтерфейсу користувача (UI testing)**. Воно здійснювалось вручну, шляхом проходження основних сценаріїв використання системи з боку користувача. Тестувались сторінки реєстрації, входу, перегляду каталогу, додавання товарів до кошика, оформлення замовлення, вибір методу оплати, підтвердження доставки. Особливу увагу було приділено адаптивності інтерфейсу — система має коректно працювати як на великих екранах, так і на мобільних пристроях.

Загалом, усі види тестування дозволили перевірити систему як на рівні окремих функцій, так і в контексті складних процесів. Це забезпечило високу якість продукту, зменшило ймовірність помилок у продакшені, підвищило стабільність і зручність використання системи кінцевим користувачем.

4.2 Вимоги до апаратного та програмного забезпечення

Для ефективної роботи інформаційної системи з продажу комп’ютерної техніки необхідно забезпечити відповідність певним технічним вимогам, як на стороні сервера, так і на стороні клієнта. Ці вимоги гарантують стабільну роботу системи, швидке оброблення запитів і можливість масштабування при збільшенні кількості користувачів.

Серверна частина

Сервер — це центральний компонент системи, на якому розміщується бекенд-програма, база даних, система авторизації, обробка API-запитів, асинхронні задачі та інші фонові процеси. Для роботи серверної частини рекомендуються такі характеристики:

Операційна система: Ubuntu 20.04 LTS або Windows Server 2019. Обидві системи стабільні, підтримуються виробниками, мають великий обсяг документації та спільнот.

Процесор: мінімум 2 фізичних ядра (рекомендовано 4+) — наприклад, Intel Core i5, AMD Ryzen 5 або еквівалент. Це забезпечить обробку паралельних запитів від кількох користувачів.

Оперативна пам’ять: від 4 ГБ (оптимально — 8 ГБ) для комфортного запуску сервісів Django, Celery, PostgreSQL та Redis.

Накопичувач: мінімум 20 ГБ SSD — швидкий диск покращує час відповіді бази даних і швидкість обробки логів.

Мережеве підключення: безперервне інтернет-з'єднання з пропускною здатністю не менше 10 Мбіт/с.

Програмне забезпечення на сервері:

Python 3.12+ — для запуску серверної частини, реалізованої у фреймворку Django.

Django Framework — ядро серверного застосунку, яке відповідає за обробку HTTP-запитів, бізнес-логіку, авторизацію тощо.

PostgreSQL 13+ — реляційна система управління базами даних, яка забезпечує надійне зберігання та обробку структурованої інформації.

Docker та Docker Compose — для контейнеризації компонентів проєкту, що дозволяє запускати систему в ізольованому середовищі.

Celery — для обробки фонових задач (наприклад, надсилання email-повідомлень або генерації звітів).

Redis — брокер повідомлень для Celery та кеш-сховище.

Sentry — для моніторингу та збору логів про помилки у продакшн-середовищі.

Завдяки використанню Docker і Docker Compose встановлення всіх сервісів відбувається автоматично, незалежно від операційної системи або середовища розробки. Це значно полегшує розгортання, зменшує кількість помилок при встановленні та покращує безпеку за рахунок ізоляції компонентів.

Клієнтська частина

Клієнт взаємодіє із системою через браузер. Система є кросплатформенною, тому працює на всіх сучасних операційних системах, таких як Windows, macOS, Linux, Android, iOS.

Вимоги до клієнта:

Браузер: сучасні версії Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.

Підтримка JavaScript (ES6+) — необхідна для коректної роботи React-додатку.

Підключення до інтернету: стабільне з'єднання зі швидкістю від 5 Мбіт/с для комфортного перегляду каталогів, здійснення покупок і роботи з кошиком.

Розширення екрану: мінімум 1024x768 пікселів, підтримується адаптивність для мобільних пристроїв.

Інтерфейс системи створено на основі React, що дозволяє динамічно оновлювати вміст сторінок без повного перезавантаження. Це забезпечує швидку та зручну взаємодію з користувачем, що є важливою вимогою до сучасних e-commerce платформ.

4.3 Склад інсталяційного пакету

Для забезпечення швидкого, зручного та безпечного розгортання інформаційної системи з продажу комп’ютерної техніки був сформований інсталяційний пакет, що включає всі необхідні компоненти для встановлення та запуску системи у будь-якому середовищі — як локальному, так і хмарному (AWS, Heroku тощо).

Інсталяційний пакет є особливо важливим для командної роботи, автоматизації CI/CD, розгортання у продакшені, а також для тестування системи

на нових серверах. Він забезпечує ізольованість компонентів, узгодженість версій і незалежність від середовища, у якому система запускається.

Основні компоненти інсталяційного пакету:

1. **Вихідний код**:

Серверна частина (backend) на Django/Python.

Клієнтська частина (frontend) на React/JavaScript.

Структура проєкту розділена на логічні модулі: orders, products, users, payments.

2. **Файл конфігурації Docker**:

Dockerfile — описує інструкції зі збирання образу системи.

docker-compose.yml — забезпечує запуск кількох сервісів одночасно (бекенд, база даних, Redis, Celery, Sentry, фронтенд).

3. **Інструкція з розгортання (README.md)**:

Покрокова інструкція для запуску системи.

Опис змінних середовища (.env).

Приклади використання API.

Команди для запуску тестів, міграцій, створення суперкористувача.

4. **Міграції бази даних**:

Файли migrations для кожного додатку Django.

Скрипти автоматичного створення таблиць та початкових даних (наприклад, категорій товарів або методів оплати).

5. **Документація API**:

Swagger/OpenAPI — автоматично генерована документація з можливістю тестування запитів прямо у браузері.

Вказано всі доступні ендпоінти, методи, параметри, приклади запитів та відповідей.

6. **Postman Collection**:

Готовий набір запитів до API для швидкого ручного тестування.

Включає авторизацію, створення користувача, перегляд товарів, оформлення замовлення, оплату.

7. **Демонстраційні дані (seed data)**:

JSON або CSV файли з прикладами товарів, категорій, користувачів.

Можуть бути завантажені під час першого запуску.

8. **Інтеграційні сервіси**:

Celery + Redis — для обробки фонових задач (надсилання email, обробка оплати).

Stripe API — попередньо налаштована інтеграція з платіжним сервісом.

SMTP-конфігурація — для надсилання листів підтвердження після замовлення.

Sentry — система моніторингу помилок у режимі реального часу.

9. **Інтерфейс адміністратора Django**:

Включено кастомізовану адмін-панель для управління товарами, замовленнями та користувачами.

Переваги інсталяційного пакету:

автоматизація розгортання: запуск усіх сервісів одним рядком (docker-compose up).

гнучкість: можливість легко налаштувати середовище під потреби конкретного клієнта або сервера.

масштабованість: можливість запускати окремі сервіси (наприклад, celery, фронтенд) в окремих контейнерах або серверах.

простота підтримки: спрощене оновлення, заміна конфігурацій, перевірка логів і моніторинг.

сумісність: незалежність від конкретної ос або типу середовища (можна запускати на локальній машині, сервері чи у хмарі).

Таким чином, інсталяційний пакет відіграє ключову роль у забезпеченні надійного, швидкого і повторюваного розгортання системи у будь-яких умовах — від локальної розробки до масштабованого хмарного продакшн-рішення.

ВИСНОВКИ

У процесі виконання дипломної роботи була повністю розроблена та протестована інформаційна система для автоматизації процесу продажу комп'ютерної техніки. Система охоплює всі ключові аспекти електронної комерції — від зручного веб-інтерфейсу для клієнтів до гнучкого адміністрування товарного каталогу, обробки замовлень та інтеграції з зовнішніми сервісами. Завдяки чіткому структуруванню вимог і системному аналізу предметної області, вдалося реалізувати логічну та ефективну архітектуру системи, що базується на сучасних інженерних практиках.

У вступній частині роботи було обґрунтовано актуальність створення подібного програмного продукту, окреслено мету та завдання, які стояли перед системою. Проведений аналіз предметної області дозволив чітко визначити ролі користувачів, логіку взаємодії між ними та бізнес-процеси, які система мала автоматизувати. Було розроблено відповідні UML-діаграми, які дозволили узагальнити функціональність і структуру системи у вигляді формалізованих моделей.

На етапі проєктування інформаційної бази побудовано логічну модель даних у вигляді ER-діаграми. Було обґрунтовано вибір системи управління базами даних PostgreSQL, яка забезпечує високу продуктивність, надійність і гнучкість. Для реалізації бекенд-частини використано фреймворк Django, що дозволив побудувати чітку логіку взаємодії з базою даних, реалізувати REST API, здійснити автентифікацію користувачів, а також забезпечити зручне адміністрування через вбудований інтерфейс. Клієнтська частина системи була створена на базі бібліотеки React, що забезпечило створення швидкого, динамічного і зручного інтерфейсу користувача.

Особливу увагу було приділено інтеграціям із зовнішніми сервісами: Stripe API для проведення онлайн-платежів, SMTP-сервер для відправки

листів з підтвердженням замовлення, Celery та Redis для обробки фонових задач. Важливо, що було реалізовано тестування системи на кількох рівнях: модульне, інтеграційне, API та UI-тестування. Це дозволило виявити та усунути критичні помилки ще на етапі розробки. Для документування API використано інструмент Swagger, що значно спрощує подальше супроводження проєкту. Крім того, було сформовано інсталяційний пакет із Docker-контейнерами, що забезпечує легке розгортання системи в будь-якому середовищі.

У підсумку, система повністю відповідає функціональним і нефункціональним вимогам, є зручною для користувачів, масштабованою для бізнесу та відкритою до подальшого розвитку. Розроблене програмне забезпечення має високе практичне значення і може бути використане не лише в сфері продажу комп’ютерної техніки, а й адаптоване під інші види електронної торгівлі.

Отже, мету дипломної роботи досягнуто: створено повнофункціональний інформаційний продукт, що вирішує актуальні бізнес-завдання та відповідає сучасним вимогам до якості програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Django Software Foundation. Django documentation [Електронний ресурс]. – Режим доступу: https://docs.djangoproject.com
2. React – A JavaScript library for building user interfaces [Електронний ресурс]. – Meta Platforms, Inc. – Режим доступу: https://react.dev
3. PostgreSQL Global Development Group. PostgreSQL documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/
4. Docker Documentation [Електронний ресурс]. – Docker, Inc. – Режим доступу: https://docs.docker.com
5. Celery Project. Celery documentation [Електронний ресурс]. – Режим доступу: https://docs.celeryq.dev
6. Stripe API Reference [Електронний ресурс]. – Stripe, Inc. – Режим доступу: https://stripe.com/docs/api
7. Unified Modeling Language (UML) Specification, version 2.5 [Електронний ресурс]. – Object Management Group. – Режим доступу: https://www.omg.org/spec/UML
8. Pytest – helps you write better programs [Електронний ресурс]. – Режим доступу: https://docs.pytest.org
9. Postman – API Platform for Developers [Електронний ресурс]. – Режим доступу: https://www.postman.com
10. OpenAPI Specification (Swagger) [Електронний ресурс]. – SmartBear Software. – Режим доступу: https://swagger.io/specification/
11. Git Documentation [Електронний ресурс]. – Режим доступу: https://git-scm.com/doc
12. GitHub Actions Documentation [Електронний ресурс]. – GitHub, Inc. – Режим доступу: https://docs.github.com/en/actions

13. GitLab CI/CD Documentation [Електронний ресурс]. – GitLab, Inc. – Режим доступу: https://docs.gitlab.com/ee/ci/
14. Sentry – Application Monitoring and Error Tracking Software [Електронний ресурс]. – Режим доступу: https://sentry.io/welcome
15. Redis Documentation [Електронний ресурс]. – Redis Ltd. – Режим доступу: https://redis.io/docs
16. ДСТУ 8302:2015. Бібліографічне посилання. Загальні положення та правила складання. – [Чинний від 01.07.2016]. – Київ: ДП «УкрНДНЦ», 2016. – 16 с.
17. OpenCart [Електронний ресурс]. – Режим доступу: https://opencart.ua/
18. Magento [Електронний ресурс]. – Режим доступу: https://business.adobe.com/products/magento/magento-commerce.html
19. Shopify [Електронний ресурс]. – Режим доступу: https://www.shopify.com/

ДОДАТОК А

База даних

Сторінок 6

Київ-2025

```
import django.core.validators
import django.db.models.deletion
import uuid
from django.conf import settings
from django.db import migrations, models

class Migration(migrations.Migration):

    initial = True

    dependencies = [
        ('cities_light', '0011_alter_city_country_alter_city_region_and_more'),
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
    ]

    operations = [
        migrations.CreateModel(
            name='Client',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('modified_at', models.DateTimeField(auto_now=True)),
                ('uuid', models.UUIDField(default=uuid.uuid4, editable=False, unique=True)),
                ('first_name', models.CharField(blank=True, max_length=50, verbose_name='First
name')),
                ('last_name', models.CharField(blank=True, max_length=50, verbose_name='Last
name')),
                ('middle_name', models.CharField(blank=True, max_length=50, verbose_name='Middle
name')),
                ('email_address', models.EmailField(max_length=254, unique=True,
validators=[django.core.validators.EmailValidator], verbose_name='Email')),
                ('phone_number', models.CharField(max_length=20)),
```

```
            ('user', models.OneToOneField(on_delete=django.db.models.deletion.CASCADE,
to=settings.AUTH_USER_MODEL)),
        ],
        options={
            'abstract': False,
        },
    ),
    migrations.CreateModel(
        name='Address',
        fields=[
            ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
            ('unit_number', models.CharField(blank=True, max_length=10, null=True)),
            ('street_number', models.CharField(max_length=10)),
            ('address_line1', models.CharField(max_length=255)),
            ('address_line2', models.CharField(blank=True, max_length=255, null=True)),
            ('postal_code', models.CharField(max_length=20)),
            ('city', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
to='cities_light.city')),
            ('country', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
to='cities_light.country')),
            ('region', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
to='cities_light.region')),
            ('client', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
to='users.client')),
        ],
    ),
]


import uuid
from django.db import migrations, models

class Migration(migrations.Migration):
```

```
    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='ClientReview',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('modified_at', models.DateTimeField(auto_now=True)),
                ('uuid', models.UUIDField(default=uuid.uuid4, editable=False, unique=True)),
                ('rating_value', models.PositiveSmallIntegerField()),
                ('comment', models.TextField()),
            ],
            options={
                'abstract': False,
            },
        ),
        migrations.CreateModel(
            name='Product',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('modified_at', models.DateTimeField(auto_now=True)),
                ('uuid', models.UUIDField(default=uuid.uuid4, editable=False, unique=True)),
                ('name', models.CharField(max_length=255)),
                ('description', models.TextField()),
                ('product_image', models.ImageField(upload_to='products/')),
            ],
            options={
                'abstract': False,
```

```
            },
        ),
        migrations.CreateModel(
            name='ProductCategory',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('category_name', models.CharField(max_length=100)),
                ('key', models.CharField(max_length=100, unique=True)),
            ],
        ),
        migrations.CreateModel(
            name='ProductConfiguration',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
            ],
        ),
        migrations.CreateModel(
            name='ProductItem',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('quantity_in_stock', models.PositiveIntegerField()),
                ('product_image', models.ImageField(upload_to='product_items/')),
                ('price', models.DecimalField(decimal_places=2, max_digits=10)),
            ],
        ),
        migrations.CreateModel(
            name='Promotion',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('name', models.CharField(max_length=255)),
                ('description', models.TextField()),
```

```
            ('discount_rate', models.DecimalField(decimal_places=2, max_digits=5)),
            ('start_date', models.DateField()),
            ('end_date', models.DateField()),
        ],
    ),
    migrations.CreateModel(
        name='Variation',
        fields=[
            ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
            ('name', models.CharField(max_length=100)),
        ],
    ),
    migrations.CreateModel(
        name='VariationOption',
        fields=[
            ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
            ('value', models.CharField(max_length=100)),
        ],
    ),
]
```

ДОДАТОК Б

Код програми Backend

Сторінок 7

Київ-2025

```
class ProductViewSet(drf_viewsets.ReadOnlyModelViewSet):
    model = product_models.Product
    permission_classes = [permissions.AllowAny, ]
    queryset = product_models.Product.objects.all()
    paginate_by = 10

    def get_serializer_class(self):
        logger.debug(f"Getting serializer for action: {self.action}")
        # Use detail serializer for single product view
        if self.action == 'retrieve':
            return product_serializers.ProductDetailSerializer
        return product_serializers.ProductDetailSerializer

    def get_queryset(self):
        logger.debug(f"Getting queryset with params: {self.request.query_params}")
        queryset = self.queryset
        category = self.request.query_params.get('category')
        pk = self.request.query_params.get('pk')

        if category:
            logger.debug(f"Filtering by category: {category}")
            queryset = queryset.filter(category__key=category)
        if pk:
            logger.debug(f"Filtering by pk: {pk}")
            queryset = queryset.filter(pk=pk)

        return queryset

    def list(self, request, *args, **kwargs):
        logger.info(f"Listing products with params: {request.query_params}")
        return super().list(request, *args, **kwargs)

    def retrieve(self, request, *args, **kwargs):
        logger.info(f"Retrieving product with pk: {kwargs.get('pk')}")
        return super().retrieve(request, *args, **kwargs)
```

```
class ShopOrderViewSet(viewsets.ModelViewSet):
    queryset = order_models.ShopOrder.objects.all()
    serializer_class = order_serializers.ShopOrderSerializer
    permission_classes = [permissions.IsAuthenticated, IsOwnerOrAdmin]
    filter_backends = [DjangoFilterBackend, filters.SearchFilter, filters.OrderingFilter]
    filterset_fields = ['order_status', 'shipping_method']
    search_fields = ['client__first_name', 'client__last_name', 'client__email_address']
    ordering_fields = ['order_date', 'order_total']
    ordering = ['-order_date']

    def get_queryset(self):
        queryset = super().get_queryset()
        if not self.request.user.is_staff:
            # Non-staff users can only see their own orders
            queryset = queryset.filter(client__user=self.request.user)
        return queryset

    @swagger_auto_schema(
        operation_description="List all orders",
        responses={
            200: order_serializers.ShopOrderSerializer(many=True),
            401: "Unauthorized"
        }
    )
    def list(self, request, *args, **kwargs):
        return super().list(request, *args, **kwargs)

    @swagger_auto_schema(
        operation_description="Create a new order",
        request_body=order_serializers.ShopOrderSerializer,
        responses={
            201: openapi.Response(
                description="Order created successfully",
```

```
            schema=openapi.Schema(
                type=openapi.TYPE_OBJECT,
                properties={
                    'order': openapi.Schema(
                        type=openapi.TYPE_OBJECT,
                        properties={
                            'id': openapi.Schema(type=openapi.TYPE_INTEGER),
                            'client': openapi.Schema(type=openapi.TYPE_OBJECT),
                            'shipping_address': openapi.Schema(type=openapi.TYPE_OBJECT),
                            'shipping_method': openapi.Schema(type=openapi.TYPE_OBJECT),
                            'payment_method': openapi.Schema(type=openapi.TYPE_OBJECT),
                            'order_status': openapi.Schema(type=openapi.TYPE_OBJECT),
                            'order_total': openapi.Schema(type=openapi.TYPE_NUMBER),
                            'order_lines': openapi.Schema(type=openapi.TYPE_ARRAY,
items=openapi.Schema(type=openapi.TYPE_OBJECT)),
                            'order_date': openapi.Schema(type=openapi.TYPE_STRING, format='date-
time'),
                        }
                    ),
                    'stripe_session_url': openapi.Schema(
                        type=openapi.TYPE_STRING,
                        description='URL to Stripe checkout page (only for card payments)'
                    )
                }
            )
        ),
        400: "Bad Request",
        401: "Unauthorized"
    }
)
def create(self, request, *args, **kwargs):
    serializer = self.get_serializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    order = serializer.save()
```

```
        # Send order confirmation email
        send_order_confirmation_email.delay(order.id)

        # If payment method is card, create Stripe session
        if order.payment_method.name.lower() == 'credit card':
            try:
                stripe_service = StripeService()
                # Check if we should include shipping
                include_shipping = request.data.get('include_shipping', True)
                session = stripe_service.create_payment_session(order,
include_shipping=include_shipping)
                return Response({
                    'order': serializer.data,
                    'stripe_session_url': session.url
                }, status=status.HTTP_201_CREATED)
            except Exception as e:
                order.delete()  # Rollback order creation if Stripe session fails
                return Response(
                    {'error': str(e)},
                    status=status.HTTP_400_BAD_REQUEST
                )

        return Response({'order': serializer.data}, status=status.HTTP_201_CREATED)

    @swagger_auto_schema(
        operation_description="Retrieve a specific order",
        responses={
            200: order_serializers.ShopOrderSerializer,
            401: "Unauthorized",
            404: "Not Found"
        }
    )
    def retrieve(self, request, *args, **kwargs):
        return super().retrieve(request, *args, **kwargs)
```

```
    @swagger_auto_schema(
        operation_description="Update a specific order",
        request_body=order_serializers.ShopOrderSerializer,
        responses={
            200: order_serializers.ShopOrderSerializer,
            400: "Bad Request",
            401: "Unauthorized",
            404: "Not Found"
        }
    )
    def update(self, request, *args, **kwargs):
        return super().update(request, *args, **kwargs)

    @swagger_auto_schema(
        operation_description="Delete a specific order",
        responses={
            204: "No Content",
            401: "Unauthorized",
            404: "Not Found"
        }
    )
    def destroy(self, request, *args, **kwargs):
        return super().destroy(request, *args, **kwargs)

class StripeService:
    @staticmethod
    def create_payment_session(order, include_shipping=True):
        """
        Create a Stripe Checkout Session for the order
        Args:
            order: The order object
            include_shipping: Whether to include shipping address collection and shipping cost
        """
        try:
            # Create line items for the order
```

```
line_items = []
for order_line in order.order_lines.all():
    line_items.append({
        'price_data': {
            'currency': 'usd',
            'product_data': {
                'name': order_line.product_item.product.name,
            },
            'unit_amount': int(order_line.price * 100),  # Convert to cents
        },
        'quantity': order_line.qty,
    })

# Add shipping cost as a line item if include_shipping is True
if include_shipping and hasattr(order, 'shipping_method'):
    line_items.append({
        'price_data': {
            'currency': 'usd',
            'product_data': {
                'name': f'Shipping - {order.shipping_method.name}',
            },
            'unit_amount': int(order.shipping_method.price * 100),
        },
        'quantity': 1,
    })

# Create the checkout session
session_params = {
    'payment_method_types': ['card'],
    'line_items': line_items,
    'mode': 'payment',
    'success_url': f"{settings.FRONTEND_URL}/payment/success",
    'cancel_url': f"{settings.FRONTEND_URL}/payment/cancel",
    'metadata': {
        'order_id': order.id,
```

```
            },
            'customer_email': order.client.email_address,
        }

        # Add shipping address collection only if include_shipping is True
        # if include_shipping:
        #     session_params['shipping_address_collection'] = {
        #         'allowed_countries': ['US', 'CA', 'UA'],  # Add more countries as needed
        #     }

        session = stripe.checkout.Session.create(**session_params)
        return session
    except Exception as e:
        raise Exception(f"Error creating Stripe session: {str(e)}")
```

ДОДАТОК В

Код програми Frontend

Сторінок 7

Київ-2025

```
import React, { useState } from 'react';
import { useNavigate, useSearchParams } from 'react-router-dom';
import {
  Box,
  Container,
  Grid,
  Card,
  CardContent,
  CardMedia,
  Typography,
  FormControl,
  InputLabel,
  Select,
  MenuItem,
  TextField,
  SelectChangeEvent,
  Pagination,
  Slider,
  Paper,
} from '@mui/material';
import { useQuery } from '@tanstack/react-query';
import { productService } from '../services/api';
import { Product, ProductCategory } from '../types';

const Products: React.FC = () => {
  const navigate = useNavigate();
  const [searchParams, setSearchParams] = useSearchParams();
  const [sortBy, setSortBy] = useState('name');
  const [searchQuery, setSearchQuery] = useState('');
  const [priceRange, setPriceRange] = useState<number[]>([0, 5000]);
  const [page, setPage] = useState(1);
  const itemsPerPage = 9;

  const selectedCategory = searchParams.get('category');
```

```
const { data: products } = useQuery({
  queryKey: ['products', selectedCategory],
  queryFn: () => productService.getProducts(selectedCategory || undefined),
});

const { data: categories } = useQuery({
  queryKey: ['categories'],
  queryFn: productService.getSecondLevelCategories,
});

console.log('Categories data:', categories);
console.log('Selected category:', selectedCategory);
console.log('Products before filtering:', products);

const handleSortChange = (event: SelectChangeEvent) => {
  setSortBy(event.target.value);
};

const handleSearchChange = (event: React.ChangeEvent<HTMLInputElement>) => {
  setSearchQuery(event.target.value);
  setPage(1);
};

const handlePriceChange = (event: Event, newValue: number | number[]) => {
  setPriceRange(newValue as number[]);
  setPage(1);
};

const handlePageChange = (event: React.ChangeEvent<unknown>, value: number) => {
  setPage(value);
};

const filteredProducts = products?.filter((product) => {
  const matchesSearch = product.name.toLowerCase().includes(searchQuery.toLowerCase()) ||
    product.description.toLowerCase().includes(searchQuery.toLowerCase());
```

```
  const matchesPrice = product.product_items?.some(
    (item) => item.price >= priceRange[0] && item.price <= priceRange[1]
  ) ?? false;
  return matchesSearch && matchesPrice;
});

const sortedProducts = filteredProducts?.sort((a, b) => {
  switch (sortBy) {
    case 'price-asc':
      return (a.product_items?.[0]?.price || 0) - (b.product_items?.[0]?.price || 0);
    case 'price-desc':
      return (b.product_items?.[0]?.price || 0) - (a.product_items?.[0]?.price || 0);
    case 'name':
    default:
      return a.name.localeCompare(b.name);
  }
});

const paginatedProducts = sortedProducts?.slice(
  (page - 1) * itemsPerPage,
  page * itemsPerPage
);

return (
  <Container maxWidth="lg">
    <Grid container spacing={3}>
      {/* Filters Sidebar */}
      <Grid item xs={12} md={3}>
        <Paper sx={{ p: 2 }}>
          <Typography variant="h6" gutterBottom>
            Filters
          </Typography>

          {/* Search */}
          <TextField
```

```
  fullWidth
  label="Search"
  variant="outlined"
  value={searchQuery}
  onChange={handleSearchChange}
  sx={{ mb: 2 }}
/>

{/* Categories */}
<FormControl fullWidth sx={{ mb: 2 }}>
  <InputLabel>Category</InputLabel>
  <Select
    value={selectedCategory || ''}
    label="Category"
    onChange={(e) => {
      if (e.target.value) {
        setSearchParams({ category: e.target.value });
      } else {
        setSearchParams({});
      }
      setPage(1);
    }}
    sx={{ color: 'text.primary' }}
  >
    <MenuItem value="">All Categories</MenuItem>
    {categories?.filter(category => category.parent_category !== null).map((category) => (
      <MenuItem key={category.id} value={category.key} sx={{ color: 'text.primary' }}>
        {category.title}
      </MenuItem>
    ))}
  </Select>
</FormControl>

{/* Price Range */}
<Typography gutterBottom>Price Range</Typography>
```

```
    <Slider
      value={priceRange}
      onChange={handlePriceChange}
      valueLabelDisplay="auto"
      min={0}
      max={5000}
      sx={{ mb: 2 }}
    />
    <Typography variant="body2" color="text.secondary">
      ${priceRange[0]} - ${priceRange[1]}
    </Typography>
  </Paper>
</Grid>

{/* Products Grid */}
<Grid item xs={12} md={9}>
  {/* Sort Controls */}
  <Box sx={{ mb: 2, display: 'flex', justifyContent: 'flex-end' }}>
    <FormControl sx={{ minWidth: 200 }}>
      <InputLabel>Сортувати за</InputLabel>
      <Select value={sortBy} label="Сортувати за" onChange={handleSortChange}>
        <MenuItem value="name">Назвою</MenuItem>
        <MenuItem value="price-asc">Ціною: від низької до високої</MenuItem>
        <MenuItem value="price-desc">Ціною: від високої до низької</MenuItem>
      </Select>
    </FormControl>
  </Box>

  {/* Products Grid */}
  <Grid container spacing={3}>
    {paginatedProducts?.map((product) => (
      <Grid item xs={12} sm={6} md={4} key={product.id}>
        <Card
          sx={{
            height: '100%',
```

```
          display: 'flex',
          flexDirection: 'column',
          cursor: 'pointer',
        }}
        onClick={() => navigate(`/products/${product.id}`)}
      >
        <CardMedia
          component="img"
          height="200"
          image={product.product_image}
          alt={product.name}
        />
        <CardContent>
          <Typography variant="h6" component="h3" gutterBottom>
            {product.name}
          </Typography>
          <Typography variant="body2" color="text.secondary" noWrap>
            {product.description}
          </Typography>
          <Typography variant="h6" color="primary" sx={{ mt: 2 }}>
            ${product.product_items[0]?.price || 'N/A'}
          </Typography>
        </CardContent>
      </Card>
    </Grid>
  ))}
</Grid>

{/* Pagination */}
<Box sx={{ mt: 4, display: 'flex', justifyContent: 'center' }}>
  <Pagination
    count={Math.ceil((filteredProducts?.length || 0) / itemsPerPage)}
    page={page}
    onChange={handlePageChange}
    color="primary"
```

```
            />
          </Box>
        </Grid>
      </Grid>
    </Container>
  );
};

export default Products;
```

ДОДАТОК Г

Код програми
Допоміжні файли

Сторінок 4

Київ-2025

```
version: '3.8'

services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    env_file:
      - .env
    environment:
      - DJANGO_SETTINGS_MODULE=config.settings
      - DATABASE_URL=postgres://${USER}:${PASSWORD}@${HOST}:${PORT}/${NAME}
      - REDIS_URL=redis://redis:6379/0
    depends_on:
      - db
      - redis
    networks:
      - app-network

  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - .env
    environment:
      - POSTGRES_DB=${NAME}
      - POSTGRES_USER=${USER}
      - POSTGRES_PASSWORD=${PASSWORD}
      - POSTGRES_HOST=${HOST}
      - POSTGRES_PORT=${PORT}
    ports:
```

```
    - "5432:5432"
  networks:
    - app-network

redis:
  image: redis:7
  ports:
    - "6379:6379"
  networks:
    - app-network

celery:
  build: .
  command: celery -A config worker -l info --pool=solo
  volumes:
    - .:/app
  env_file:
    - .env
  environment:
    - DJANGO_SETTINGS_MODULE=config.settings
    - DATABASE_URL=postgres://${USER}:${PASSWORD}@${HOST}:${PORT}/${NAME}
    - REDIS_URL=redis://redis:6379/0
  depends_on:
    - web
    - db
    - redis
  networks:
    - app-network

celery-beat:
  build: .
  command: celery -A config beat -l info
  volumes:
    - .:/app
  env_file:
```

```
    - .env
  environment:
    - DJANGO_SETTINGS_MODULE=config.settings
    - DATABASE_URL=postgres://${USER}:${PASSWORD}@${HOST}:${PORT}/${NAME}
    - REDIS_URL=redis://redis:6379/0
  depends_on:
    - web
    - db
    - redis
  networks:
    - app-network

networks:
  app-network:
    driver: bridge

volumes:
  postgres_data:
```

```
#Database
HOST=
USER=
PASSWORD=
PORT=
NAME=

SECRET_KEY=

DEBUG=

GOOGLE_OAUTH2_ID=
GOOGLE_OAUTH2_SECRET=
```

```
FRONTEND_URL=

STRIPE_PUBLISHABLE_KEY=
STRIPE_SECRET_KEY=
STRIPE_WEBHOOK_SECRET=

EMAIL_HOST_USER=
EMAIL_HOST_PASSWORD=
```