МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Проектування та розробка маркетплейсу з механізмом рекомендованих товарів та аналітикою користувацьких даних»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Безкровний Ерік Ігорович

Керівник: *доктор філософії з прикладної математики, старший викладач кафедри інформаційних технологій та аналітики даних, фахівець-практик*
Красюк Богдан Віталійович

Рецензент: *розробник програмного забезпечення в ТОВ ОБНОВА-ЄВРОШОП, викладач кафедри інформаційних технологій та аналітики даних НаУОА*
Шведюк Володимир Володимирович

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)

Протокол №10 від 28 травня 2025 р.

Острог, 2025

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Проектування та розробка маркетплейсу з механізмом рекомендованих товарів та аналітикою користувацьких даних

Автор: Безкровний Ерік Ігорович

Науковий керівник: Ph.D. in Applied Mathematics, Teacher, Specialist-Practitioner, Bohdan Vitaliyovych Krasiuk

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *58 с., 5 рис., 2 табл., 1 додаток, 14 джерел.*

Ключові слова: *маркетплейс, вебзастосунок, FastAPI, React, рекомендаційна система, електронна комерція, Pydantic, користувацькі дані*

Короткий зміст праці:
Кваліфікаційна робота присвячена проектуванню та розробці повнофункціонального онлайн-маркетплейсу з вбудованим персоналізованим механізмом рекомендацій. Робота включає впровадження модульного бекенду з використанням FastAPI та PostgreSQL, що включає автентифікацію на основі токенів, перевірку даних через Pydantic, структуровану маршрутизацію API та ефективну систему управління продуктами та замовленнями. Крім того, було інтегровано JSON-парсер для оптимізації масового імпорту продуктів. Фронтенд було створено за допомогою React та TailwindCSS, що забезпечує адаптивний, сучасний та інтерактивний односторінковий додаток.

Особлива увага була приділена реалізації алгоритму рекомендацій. Він базується на аналізі історії переглядів користувача, яка постійно зберігається в базі даних. Система визначає часто відвідувані категорії продуктів та фільтрує доступні товари, щоб повертати релевантні результати з високими оцінками, які користувач ще не переглядав і не купував. Ця логіка була розроблена для покращення персоналізації та залучення, з потенціалом для майбутнього вдосконалення за допомогою методів машинного навчання, таких як кластеризація або колаборативна фільтрація.

Рішення було успішно протестовано, розгорнуто та розроблено для масштабування для реального використання або подальшого розвитку в комерційному або академічному середовищі.

(підпис автора)

ANNOTATION
of qualification paper
for bachelor's degree

Theme: Design and Development of a Marketplace with a Recommendation Engine and User Data Analytics

Author: Bezkrovnyi Erik

Scientific supervisor: Lecturer-Intern of the Department of ITAD, Matsevych Denys Volodymyrovych

Defended: «.....»..... 20__ year.

Explanatory note to the qualification work: *58 pages, 5 figures, 2 tables, 1 appendix, 14 sources.*

Keywords: *marketplace, web application, FastAPI, React, recommendation system, Pydantic, PostgreSQL, user data*

Abstract:
The qualification paper is dedicated to the design and development of a full-featured online marketplace with a built-in personalized recommendation engine. The work includes the implementation of a modular backend using FastAPI and PostgreSQL, featuring token-based authentication, data validation through Pydantic, structured API routing, and an efficient product and order management system. Additionally, a JSON parser was integrated to streamline bulk product import. The frontend was built using React and TailwindCSS, delivering a responsive, modern, and interactive single-page application.

A special focus was placed on the implementation of the recommendation algorithm. It is based on the analysis of the user's viewing history, which is persistently stored in the database. The system identifies frequently visited product categories and filters available items to return relevant results with high ratings that the user has not yet viewed or purchased. This logic was designed to improve personalization and engagement, with potential for future enhancement using machine learning methods such as clustering or collaborative filtering.

The solution was successfully tested, deployed, and is designed to be scalable for real-world use or further development in commercial or academic environments.

(author's signature)

Зміст

ВСТУП

У сучасному цифровому суспільстві електронна комерція стала однією з ключових галузей, що формують глобальну економіку. Із розвитком інтернет-технологій, зростанням мобільного трафіку та зміною споживацьких звичок онлайн-платформи для купівлі та продажу товарів набули широкого поширення та значної популярності серед користувачів. Одним із найбільш зручних і ефективних рішень у сфері електронної комерції є маркетплейси — платформи, що забезпечують взаємодію між продавцями та покупцями в єдиному інформаційному середовищі.

Традиційні інтернет-магазини поступово поступаються місцем універсальним платформам, які дозволяють створити мультифункціональний простір для торгівлі різними категоріями товарів. Маркетплейси полегшують процес пошуку, вибору, оплати та доставки продукції, забезпечуючи високий рівень автоматизації та персоналізації. Завдяки цьому вони стали основною точкою входу в онлайн-комерцію для малого та середнього бізнесу.

Однак ефективна реалізація маркетплейсу вимагає комплексного підходу до проєктування програмної архітектури, безпеки, масштабованості та підтримки високого рівня користувацького досвіду. У цьому контексті актуальним є створення високоякісного вебзастосунку, який би забезпечував повний цикл взаємодії користувача з системою: реєстрація, авторизація, перегляд товарів, додавання до кошика, оформлення замовлення, застосування промокодів, написання відгуків та отримання індивідуальних рекомендацій.

Метою даної кваліфікаційної роботи є **проєктування та розробка сучасного маркетплейсу**, реалізованого з використанням мікросервісної архітектури на базі технологій **FastAPI**, **PostgreSQL** та **React**. Рішення орієнтоване на створення масштабованої, безпечної та інтуїтивно зрозумілої системи електронної комерції з персоналізованими сервісами для користувачів.

Для досягнення поставленої мети необхідно виконати такі основні завдання:

- провести аналіз предметної області маркетплейсів та існуючих рішень;
- спроєктувати архітектуру програмної системи з поділом на клієнтську та серверну частини;
- розробити систему управління користувачами, товарами, замовленнями, кошиком, відгуками та рекомендаціями;
- реалізувати REST API з підтримкою автентифікації, авторизації та захисту даних;
- реалізувати клієнтську частину з адаптивним дизайном та інтеграцією з бекендом;
- забезпечити тестування та перевірку працездатності системи на практиці.

Об’єкт дослідження: інформаційні системи електронної комерції типу “маркетплейс” та архітектурні підходи до їх побудови.

Предмет дослідження: проєктування та розробка веборієнтованої системи маркетплейсу з використанням сучасних фреймворків та бібліотек.

Таким чином, реалізація власного проєкту маркетплейсу не лише дозволить застосувати набуті знання на практиці, а й забезпечить внесок у розвиток українського сегменту цифрової торгівлі, надаючи кінцевим користувачам якісний, функціональний та зручний інструмент для взаємодії з товарами і послугами онлайн.

РОДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ВЕБЗАСТОСУНКУ МАРКЕТПЛЕЙСУ

1.1. Поняття маркетплейсу та його роль в електронній комерції

Сучасний світ неможливо уявити без електронної комерції, яка охоплює не лише продаж товарів, а й створення цифрової інфраструктури для зберігання, обробки, аналізу та обміну інформацією між різними суб’єктами торгівлі. Одним із найуспішніших рішень у цій сфері є маркетплейси — онлайн-платформи, які забезпечують цифрову взаємодію між продавцем і покупцем, зводячи до мінімуму витрати часу, коштів та людських ресурсів.

Маркетплейс (від англ. *marketplace*) — це спеціалізований вебресурс або вебзастосунок, який об'єднує в одному середовищі багато продавців і покупців, надаючи функціонал для пошуку, вибору, порівняння, купівлі товарів або послуг. Відмінність маркетплейсу від звичайного інтернет-магазину полягає в багатосторонності: якщо інтернет-магазин зазвичай представляє одного продавця (власника ресурсу), то маркетплейс надає можливість сотням або тисячам продавців одночасно публікувати власні товари або послуги, працюючи під єдиною платформою, логікою та дизайном.

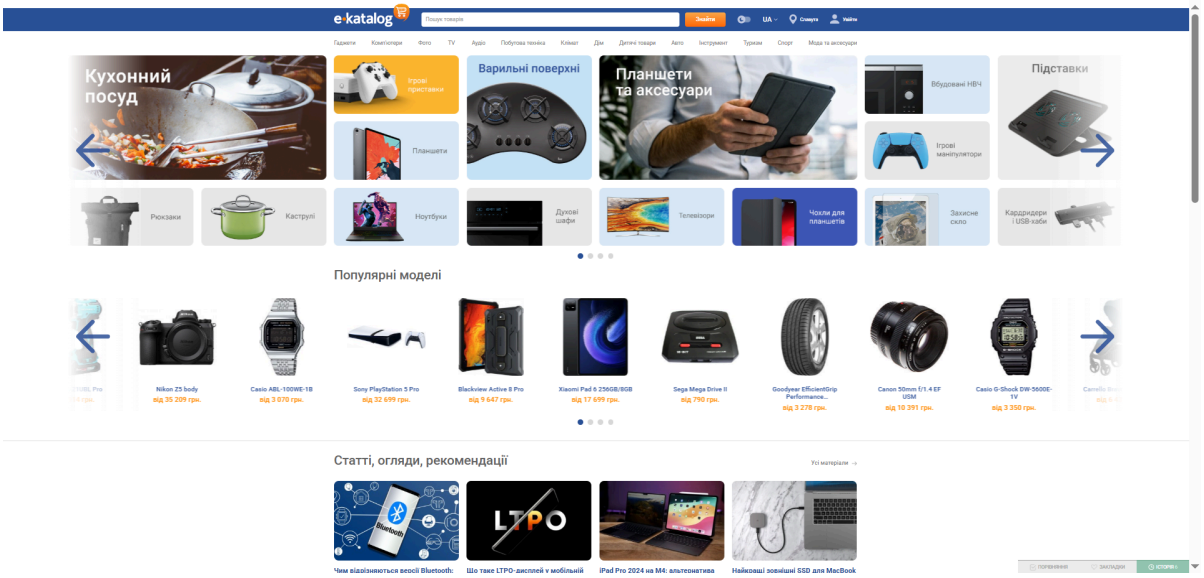


Рис 1.1.1 Вигляд головної сторінки E-katalog
Джерело: https://ek.ua/

Такі платформи беруть на себе не лише візуальну організацію інтерфейсу, але й технічне забезпечення транзакцій, системи обліку, модулі логістики, платіжну інфраструктуру та механізми взаємодії між користувачами. Найвідомішими глобальними прикладами маркетплейсів є **Amazon**, **AliExpress**, **eBay**, **Etsy**, **Rakuten**, у той час як в Україні активно розвиваються **Prom.ua**, **Rozetka**, **E-katalog** та інші.

Роль маркетплейсу в електронній торгівлі є системоутворюючою, адже він виконує одразу декілька функцій:

- **Інформаційну** — надає покупцям актуальну інформацію про наявні товари, характеристики, відгуки, рейтинги.
- **Інфраструктурну** — забезпечує єдину платформу для безпечної взаємодії сторін, оплати, обліку й доставки.
- **Маркетингову** — допомагає продавцям просувати свої товари серед широкої аудиторії з використанням реклами, аналітики, рекомендаційних систем.
- **Сервісну** — надає клієнтам зручні засоби комунікації, підтримки, оформлення замовлень, відстеження покупок.
- **Аналітичну** — збирає статистику переглядів, продажів, поведінки користувачів, що дозволяє як адміністраторам платформи, так і самим продавцям приймати більш обґрунтовані рішення.

З технічного боку маркетплейси — це комплексні вебзастосунки, у структурі яких зазвичай виділяють такі ключові компоненти:

1. **Frontend (клієнтська частина)** — відповідає за інтерфейс користувача, забезпечує відображення товарів, перегляд карток, керування кошиком тощо. У сучасних реалізаціях часто використовуються React, Angular, Vue.js.
2. **Backend (серверна частина)** — обробляє бізнес-логіку, запити клієнтів, роботу з базою даних, аутентифікацію, облік і генерацію відповідей. Зазвичай реалізується на Python (FastAPI, Django), Node.js, ASP.NET, Java.
3. **База даних** — центральне сховище всіх сутностей проєкту: користувачів, товарів, замовлень, категорій, історії активностей. Найчастіше використовуються PostgreSQL, MySQL, MongoDB.

4. **REST API або GraphQL API** — інтерфейс обміну даними між клієнтською частиною та сервером.
5. **Адмін-панель** — окрема частина платформи, доступна лише адміністраторам, для керування товарами, промокодами, відгуками, користувачами та звітами.
6. **Система рекомендацій** — окремий блок, що генерує персоналізовані пропозиції на основі історії переглядів, покупок, пошуку або поведінки користувача.
7. **Підсистема безпеки** — включає JWT-токени, політики доступу, валідацію запитів, перевірку прав ролей, захист від XSS/CSRF-атак тощо.

Маркетплейс виступає як **універсальний цифровий інструмент**, який дозволяє підприємцям зосередитись на товарі та продажах, не переймаючись технічними складнощами. Водночас користувач отримує централізований доступ до великого каталогу товарів із можливістю порівняння цін, фільтрації, сортування, перегляду рейтингів та відгуків — що значно покращує споживчий досвід.

Варто зазначити, що розвиток маркетплейсів безпосередньо пов'язаний із високими вимогами до масштабованості системи. У разі зростання навантаження (збільшення кількості товарів, користувачів, запитів) застосунок повинен зберігати стабільну швидкодію, забезпечуючи безперервність сервісу. Саме тому більшість сучасних платформ орієнтується на модульну архітектуру, яка передбачає поділ логіки на окремі незалежні компоненти з можливістю горизонтального масштабування.

Таким чином, маркетплейс — це не просто комерційна платформа, а **комплексна цифрова система**, яка об'єднує в собі технічні, організаційні, аналітичні та маркетингові рішення. Його ефективна розробка та впровадження потребують знання в галузях веброзробки, баз даних, кібербезпеки, UX/UI-дизайну, бізнес-логіки та оптимізації. Саме ці завдання були покладені в основу створення даного кваліфікаційного проєкту.

1.2. Клієнт-серверна архітектура як основа побудови маркетплейсів

У сучасній розробці програмних систем, зокрема вебзастосунків типу маркетплейс, одним із фундаментальних архітектурних підходів є **клієнт-серверна архітектура**. Ця модель дозволяє чітко розділити функціональність системи між клієнтом (що ініціює запити) та сервером (що їх обробляє), забезпечуючи тим самим масштабованість, незалежність компонентів, зручність підтримки та подальшого розвитку.

Основи клієнт-серверної взаємодії

У клієнт-серверній архітектурі **клієнтська частина** (frontend) виконує роль інтерфейсу взаємодії з користувачем — через браузер або мобільний додаток. Вона надсилає запити до **серверної частини** (backend), яка здійснює обробку запитів, взаємодіє з базами даних, виконує необхідні бізнес-операції та формує відповіді.

Найчастіше обмін між клієнтом і сервером здійснюється через **протокол HTTP або HTTPS**, а структура даних — у форматі **JSON**. Такий підхід дає змогу досягти високої сумісності між різними компонентами системи та створити єдиний програмний інтерфейс — **REST API** (Representational State Transfer Application Programming Interface), який підтримує стандартні методи:

- **GET** — отримання даних (наприклад, список товарів),
- **POST** — створення нового ресурсу (реєстрація користувача, оформлення замовлення),
- **PUT / PATCH** — оновлення існуючих даних (зміна адреси доставки),
- **DELETE** — видалення об’єкта (товару з кошика).

Централізоване API дозволяє використовувати один і той самий бекенд як для вебверсії, так і для мобільного застосунку, десктоп-клієнта чи зовнішньої інтеграції (наприклад, з платіжними системами або email-сервісами).

Застосування архітектури у маркетплейсах

У контексті маркетплейсу, де на одній платформі одночасно працюють покупці, продавці та адміністратори, клієнт-серверна модель дозволяє ефективно розділити функції між інтерфейсом користувача і внутрішньою логікою системи.

Клієнтська частина

Клієнт (frontend) відповідає за:

- відображення контенту (товари, фільтри, кошик, обране);
- навігацію між сторінками (наприклад, головна, сторінка товару, кабінет користувача);
- обробку подій користувача (натискання кнопок, введення даних у форми);
- формування HTTP-запитів до сервера;
- отримання і виведення результатів (наприклад, підтвердження оформлення замовлення).

У даному проєкті клієнтська частина реалізована з використанням **React** — одного з найпопулярніших JavaScript-фреймворків для створення односторінкових застосунків (SPA). Для стилізації використано **TailwindCSS**, що дозволяє швидко адаптувати інтерфейс до різних розмірів екранів і підвищити продуктивність розробки.

Реактивність і компонентність React забезпечують високу гнучкість — кожен блок (наприклад, картка товару або кошик) існує як окремий незалежний компонент, який можна перевикористовувати, комбінувати, тестувати та оновлювати ізольовано.

Серверна частина

Сервер (backend) не лише відповідає на запити, а й виконує значну частину **бізнес-логіки**. У проєкті він реалізований за допомогою **FastAPI** — сучасного асинхронного Python-фреймворку, який забезпечує:

- високу продуктивність завдяки використанню `async/await`;
- автоматичну генерацію документації Swagger/OpenAPI;
- просту побудову REST API;

- інтеграцію з базами даних, системами автентифікації, email-розсилкою тощо.

До завдань бекенду входить:

- перевірка даних (валідація форм, перевірка токенів доступу);
- облік товарів і замовлень;
- генерація промокодів;
- обробка відгуків та рейтингу;
- формування рекомендацій на основі історії переглядів користувача;
- керування ролями: адміністратор, покупець, продавець.

Зберігання даних реалізоване на базі **PostgreSQL** — потужної реляційної СУБД, яка підтримує транзакції, зовнішні ключі, індексацію, JSON-поля, пошук та агрегування.

Взаємодія між клієнтом і сервером

Типовий процес взаємодії між частинами у клієнт-серверній архітектурі виглядає так:

1. Користувач відкриває вебзастосунок у браузері.
2. React-застосунок ініціалізує запит типу `GET /api/products` на бекенд.
3. FastAPI отримує запит, звертається до бази даних PostgreSQL і повертає список товарів у форматі JSON.
4. Клієнтська частина обробляє дані, відображаючи їх на сторінці.
5. Користувач додає товар у кошик — надсилається `POST /api/cart`.
6. На основі нових дій клієнта бекенд оновлює дані в базі, а також тригерить email-повідомлення, рекомендації або статистику.

Переваги клієнт-серверного підходу

Використання клієнт-серверної архітектури у маркетплейсах дає низку стратегічних переваг:

- **Гнучкість** — незалежна розробка frontend та backend частин різними командами.

- **Масштабованість** — сервер можна розгорнути у хмарному середовищі, підключити кеш, CDN, балансувальники навантаження.
- **Безпека** — реалізація авторизації через JWT, захищені маршрути, HTTPS-з’єднання, контроль прав доступу.
- **Інтеграції** — API дозволяє підключати сторонні сервіси (оплати, доставки, аналітики).
- **Мобільність** — один і той самий сервер може обслуговувати як веб, так і мобільний додаток.

Використані підходи у проекті

У межах даного проєкту було реалізовано:

- REST API з маршрутизацією для 8+ модулів (користувачі, товари, замовлення, промокоди, відгуки, рекомендації тощо);
- система авторизації з JWT-токенами, верифікацією через email;
- компонентний інтерфейс React з адаптивним дизайном;
- клієнтські сервіси для обробки асинхронних запитів;
- обробка статусів, сповіщення, обробка помилок.

Таким чином, клієнт-серверна архітектура виступає кістяком для ефективної побудови маркетплейсу, дозволяючи досягти високої швидкодії, надійності, зручності користування та гнучкості підтримки. Розподіл обов’язків між клієнтом та сервером забезпечує ефективне виконання завдань кожного з компонентів і дозволяє будувати надійні системи, здатні обслуговувати тисячі користувачів одночасно.

1.3. Аналіз конкурентних рішень

Створення нового програмного продукту, зокрема у сфері електронної комерції, передбачає вивчення ринку, виявлення потреб користувачів, визначення функціональних особливостей провідних платформ і формування власної конкурентної моделі. Особливої уваги потребують ті сервіси, які вже мають сталу аудиторію, перевірену бізнес-модель і технічну інфраструктуру.

Аналіз конкурентних рішень дає змогу:

- сформувати технічні та нефункціональні вимоги;

- врахувати досвід лідерів ринку;
- визначити точки зростання та позиціонування власної платформи;
- спроєктувати інтерфейс, який буде інтуїтивно знайомим користувачу;
- уникнути критичних помилок проєктування.

Розглянемо найбільш релевантні приклади маркетплейсів: **Rozetka**, **Prom.ua**, **Etsy** та **E-katalog**. Кожна з цих платформ має унікальні особливості, які варто враховувати під час створення власного рішення.

1.3.1 Rozetka — масштабний приклад B2C-маркетплейсу

Rozetka є прикладом **універсального маркетплейсу**, який одночасно обслуговує як власні продажі, так і виступає посередником між сторонніми продавцями та покупцями. Система має складну ієрархію категорій, інтегрований модуль логістики, відстеження, програми лояльності.

З технічної точки зору, Rozetka:

- використовує **динамічне завантаження контенту**, що забезпечує швидкодію;
- має **мікросервісну архітектуру**, що дає змогу окремо оновлювати модулі: пошуку, товарів, платежів;
- забезпечує **автоматичну модерацію контенту**, включаючи відгуки, фото, запитання;
- активно застосовує аналітику клієнтської поведінки та трекінг сесій для подальшої персоналізації.

Інтерфейс Rozetka продуманий до дрібниць: акценти на кнопках «Купити», швидкий доступ до кошика, історія переглядів, блок «Рекомендовані товари» на основі cookie-файлів та профілю.

У порівнянні з нею, наш маркетплейс фокусується на **простоті MVP**, але вже містить основні компоненти: REST API, кошик, реєстрацію, базу товарів, модулі знижок і рекомендацій.

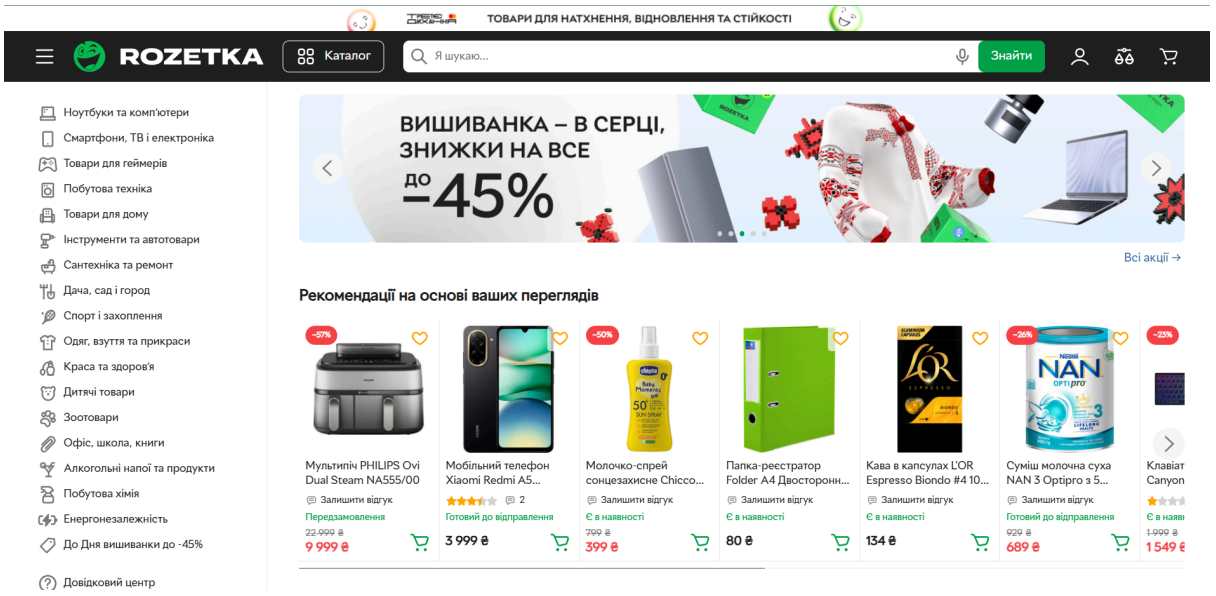


Рис 1.3.1. Маркетплейс ROZETKA

Джерело: https://rozetka.com.ua/

1.3.2 Prom.ua — платформа для самостійного ведення бізнесу

Prom.ua відрізняється тим, що **виконує функції конструктора магазинів**. Кожен підприємець отримує «вітрину», яку може кастомізувати під себе, але всі вони існують у межах одного ядра платформи.

Порівняно з Rozetka:

- інтерфейс менш уніфікований;
- фокус — на адмініструванні й просуванні товарів через пошукові системи;
- велике значення має SEO (індексація назв, URL, оптимізація описів);
- продавці мають більшу свободу, але й більшу відповідальність за обслуговування клієнтів.

Prom.ua має розвинену API-документацію для синхронізації з CRM, ERP та іншими сервісами. Це дозволяє малому бізнесу повністю автоматизувати діяльність без розробників.

У нашому випадку, система централізована: товари додає адміністратор, і це дає змогу зберегти контроль якості, структури та бізнес-логіки.

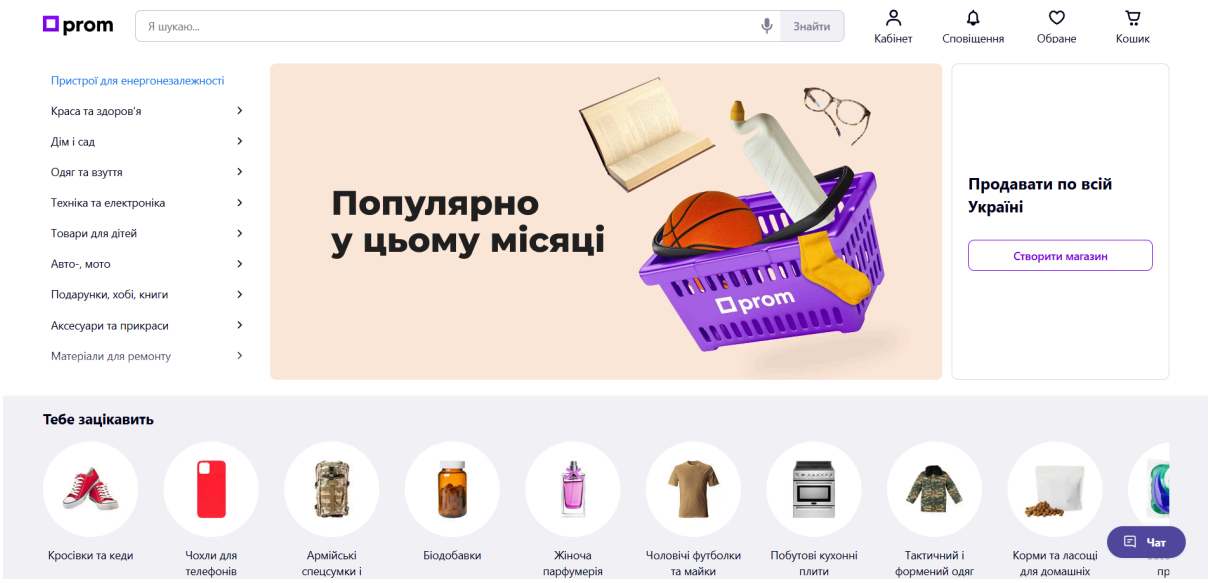


Рис 1.3.2 Маркетплейс Prom.ua

Джерело: https://prom.ua/

1.3.3 Etsy — приклад емоційного позиціонування

Etsy — це **емоційний маркетплейс**, у якому головне — не лише товар, а й його унікальність, історія, автор. Це підтверджується інтерфейсом: великі фото, авторські описи, акценти на персональному підході.

Платформа:

- дозволяє продавцям оформлювати «вітрини» в особистому стилі;
- підтримує внутрішній чат з клієнтами;
- має систему рекомендацій з фокусом на подарунки, стиль, сезонність;
- дозволяє продавати цифрові товари (pdf, графіка, шаблони).

З технічного погляду, Etsy поєднує типову клієнт-серверну архітектуру з сервісами підтримки (аналітика, e-mail, сегментація). Це дозволяє гнучко підлаштовуватись під нішеві запити.

У нашому проєкті також передбачено розширення — наприклад, додавання модулів повідомлень, персональних пропозицій, майбутня підтримка цифрових товарів.

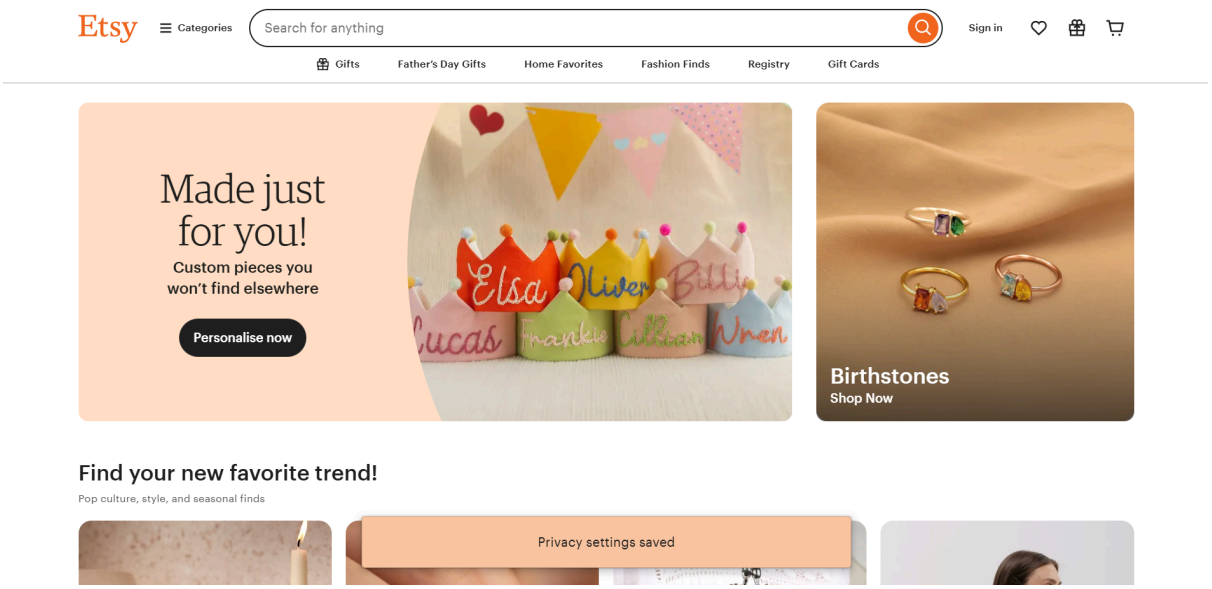


Рис 1.3.3 Маркетплейс Etsy

Джерело: https://www.etsy.com/

1.3.4 E-katalog — агрегатор даних та джерело для імпорту

E-katalog — технічно складний агрегатор. Він не продає товари, але **інтегрує дані з тисяч магазинів**, подаючи їх у стандартизованому форматі з можливістю порівняння. Його ключові переваги:

- **уніфікація специфікацій**;
- **структуровані параметри фільтрації**;
- **постійне оновлення цін через API або парсинг**;
- **система оцінок і відгуків, незалежна від магазинів**.

У розробці власного проєкту саме E-katalog було використано як **джерело структурованих, реальних, актуальних даних для ініціалізації БД**.

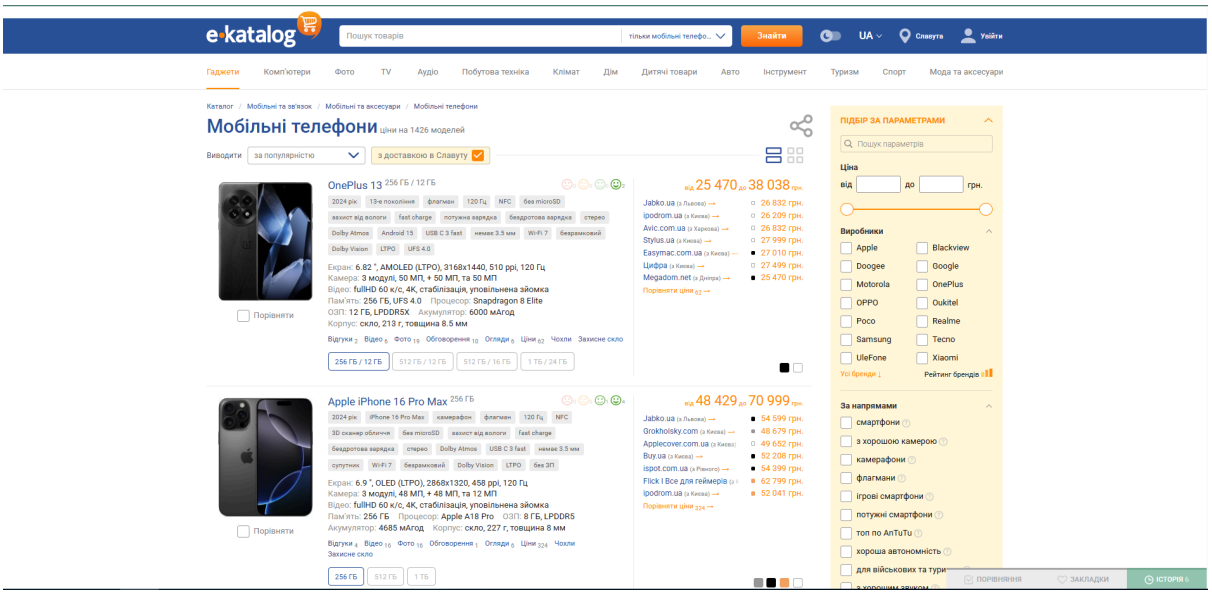


Рис. 1.3.4 Маркетплейс E-katalog.
Джерело: https://ek.ua/ua/list/122/

1.3.5 Реалізація парсера для E-katalog

Для наповнення бази маркетплейсу, створено **індивідуальний веб-парсер**:

- Мова програмування: **Python**
- Бібліотеки: `requests`, `BeautifulSoup`, `pandas`, `SQLAlchemy`

Особливості реалізації:

- обходить HTML-сторінки з урахуванням пагінації;
- автоматично витягує назву, ціни, характеристики;
- зберігає зображення та посилання;
- дозволяє масштабувати процес — додавання нових категорій;
- інкапсулює логіку в модулі, придатні для повторного використання.

Фрагмент коду (приклад):

```
url = "https://www.e-katalog.ua/ua/list/122/"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
items = soup.find_all("div", class_="model-short-div")

for item in items:
    name = item.find("a", class_="model-short-title").text.strip()
    price = item.find("div", class_="model-hot-prices").text.strip()
    print(name, price)
```

Лістинг 1.3.1 Фрагмет коду парсера для маркетплейсу E-katalog.

Джерело: Автор

Практичні переваги:

- створено тестовий набір товарів для UI/UX;
- відтестовано модулі фільтрації, кошика, пошуку;
- перевірено структуру REST-запитів;
- автоматизовано перевірку продуктивності при масовому імпорті.

Цей парсер може бути адаптований для регулярного оновлення даних, або перетворений на окремий **ETL-процес**, який періодично оновлює базу товарів.

1.3.6 Порівняльна таблиця конкурентів

Платформа	Фільтрація	Відгуки	Система рекомендацій	Мобільна адаптація	Адмін-панель
Rozetka	є	є	є	є	є
Prom.ua	є	є	немає	є	є
Etsy	є	є	є	є	є
E-katalog	є	є	є (обмежений)	є	немає
Проєкт	є	є	є (на основі історії переглядів)	є	є

Висновки до розділу 1

У першому розділі даної кваліфікаційної роботи було проведено ґрунтовний теоретичний аналіз предметної області, архітектурних рішень і сучасних підходів до розробки вебзастосунків типу маркетплейс. Було з’ясовано, що маркетплейс — це не лише торгова платформа, а цілісна інформаційна система, яка виконує функції агрегування, фільтрації, просування та персоналізації товарів, забезпечуючи максимальний комфорт для користувачів та ефективність для бізнесу.

У підрозділі 1.1 розглянуто поняття маркетплейсу в контексті електронної комерції. Детально проаналізовано особливості та переваги маркетплейсів над класичними інтернет-магазинами. Акцент зроблено на ключових компонентах системи: каталогізація товарів, система рейтингу й відгуків, інструменти фільтрації, рекомендаційні механізми, система обліку й аналітики. Встановлено, що сучасні маркетплейси виступають ядром цифрової торгівлі, поєднуючи програмну архітектуру, UX/UI-дизайн, безпеку й масштабованість у єдину платформу.

У підрозділі 1.2 розглянуто клієнт-серверну архітектуру, яка є основою побудови сучасних вебзастосунків. Визначено основні принципи взаємодії між клієнтською та серверною частинами системи, окреслено переваги REST API, описано розмежування обов’язків між frontend та backend. Особливу увагу приділено ролі серверної логіки в обробці бізнес-процесів, збереженні даних, автентифікації користувачів та управлінні доступом.

У підрозділі 1.3 здійснено порівняльний аналіз конкурентних платформ — Rozetka, Prom.ua, Etsy та E-katalog. Проведено глибоку оцінку архітектури, функціональності, ролі інтерфейсу користувача, комунікаційних механізмів та можливостей кастомізації. Особливе місце в аналізі посів E-katalog, який не є класичним маркетплейсом, однак став джерелом якісно структурованої інформації. На основі даних цього агрегатора автором було реалізовано власний парсер із використанням бібліотеки Beautiful Soup. Зібрані дані були інтегровані у базу маркетплейсу, що дало змогу наповнити систему реалістичним контентом на ранньому етапі розробки.

Таким чином, теоретичний розділ дозволив:

- сформувати чітке розуміння структури маркетплейсу як інформаційної системи;
- закласти технічні та бізнесові вимоги до вебзастосунку;
- визначити ключові модулі, які мають бути реалізовані в рамках проєкту;
- практично використати результати аналізу для наповнення системи даними.

Отримані висновки лягли в основу технічного проєктування, архітектурної побудови й програмної реалізації вебзастосунку маркетплейсу, яка докладно розглядається у наступному розділі.

РОЗДІЛ 2. ПРИКЛАДНА ЧАСТИНА РОЗРОБКИ ВЕБЗАСТОСУНКУ МАРКЕТПЛЕЙСУ

2.1 Серверна частина вебзастосунку

2.1.1 Вибір технологій та загальна структура

Серверна частина маркетплейсу реалізована за допомогою сучасного Python-фреймворку **FastAPI**, який дозволяє створювати високопродуктивні REST API з асинхронною обробкою запитів. Даний фреймворк був обраний з огляду на:

- Простоту та швидкість створення маршрутизованих API;
- Підтримку OpenAPI/Swagger для автоматичної генерації документації;
- Асинхронність — async/await дозволяє обробляти тисячі одночасних запитів без блокування;
- Тісну інтеграцію з бібліотекою Pydantic для валідації та серіалізації даних;
- Добру сумісність із ORM-бібліотекою SQLAlchemy.

На додачу до FastAPI використовуються:

- **PostgreSQL** — реляційна система керування базами даних, що підтримує ACID-транзакції, індекси, JSONB-поля;
- **SQLAlchemy** — ORM-рішення для опису моделей таблиць і взаємозв’язків між ними;
- **Alembic** — для управління міграціями бази даних;
- **Uvicorn** — ASGI-сервер, що запускає FastAPI-застосунок.

Проєкт побудований за принципами чистої архітектури та поділений на модулі:

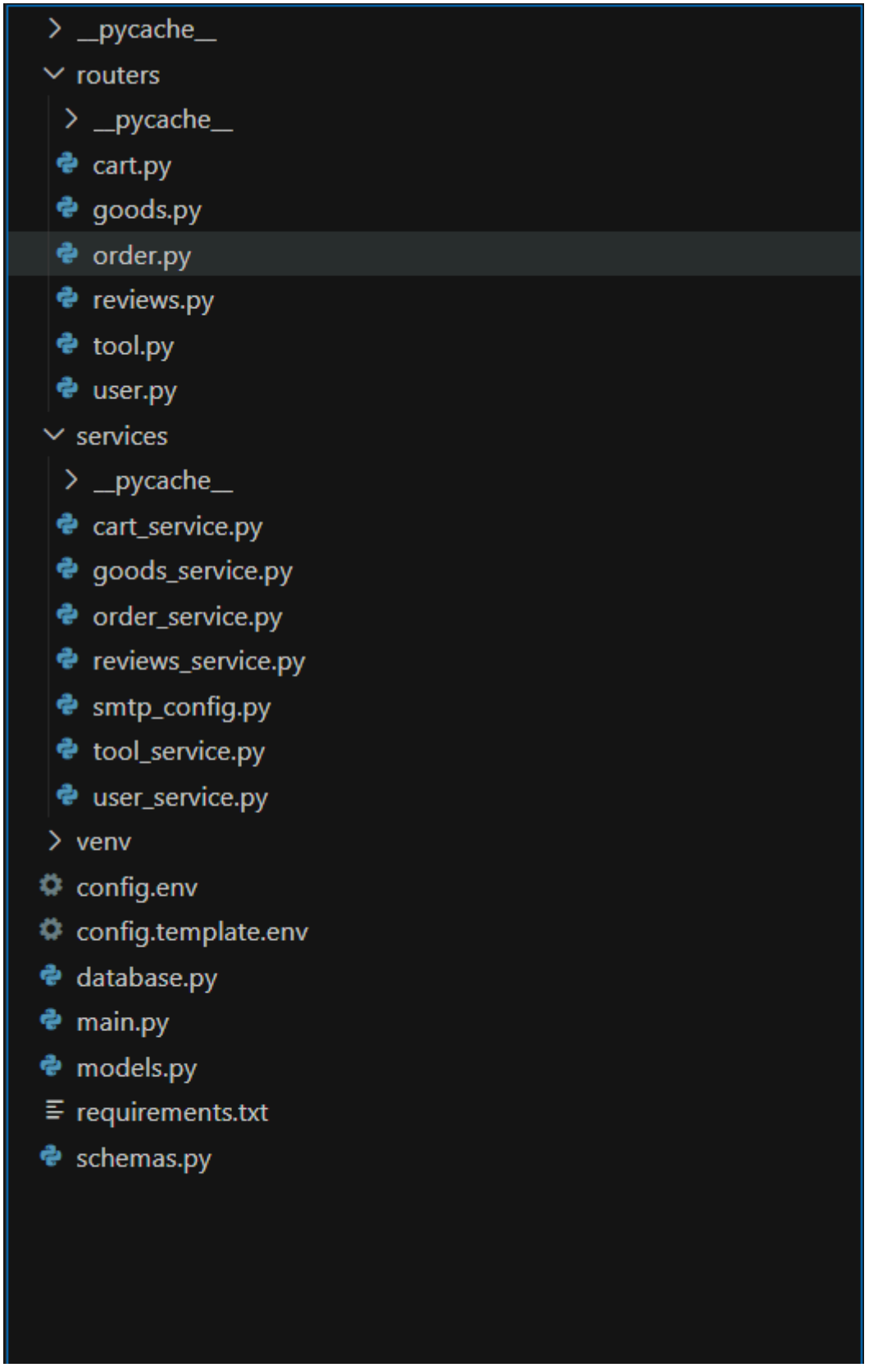


Рис 2.1.1 Архітектура Backend - частини.

Джерело: Автор

У процесі реалізації серверної частини була обрана **модульна архітектура**, що забезпечує логічне розділення відповідальностей:

- models/ — визначення ORM-моделей таблиць бази даних;
- schemas/ — Pydantic-схеми, які використовуються для валідації вхідних та вихідних даних;
- routers/ — маршрутизатори FastAPI, згруповані за функціональністю (користувачі, товари, замовлення);
- services/ — бізнес-логіка застосунку, в тому числі логіка рекомендацій, email-сервіси;

Завдяки цьому підходу вдалося досягти **високої масштабованості** та **простоти підтримки** коду. Усі основні бізнес-функції було винесено в окремі модулі, що дало змогу чітко документувати архітектуру та забезпечити стабільність роботи застосунку. Окрему увагу приділено рефакторингу логіки користувача та товарів — вони реалізовані як ізольовані функціональні одиниці з повною відповідністю принципам SOLID та Clean Architecture.

Центральним елементом покращення архітектури стала також **перенесена логіка історії переглядів (ViewHistory)**. Раніше перегляди зберігались тимчасово в сесії або локальному сховищі. Нова реалізація передбачає збереження переглядів у окремій таблиці бази даних з усіма необхідними зв’язками, що дозволяє реалізовувати точну персоналізацію та статистичний аналіз. На основі цієї таблиці побудована рекомендаторна система, яка інтегрується через окремий сервісний модуль та взаємодіє з API-клієнтом у режимі реального часу.

Архітектура також передбачає **чітке розмежування прав доступу** через OAuth2 + JWT, інтеграцію поштового сервісу, централізовану обробку помилок, модульну генерацію токенів доступу та розширену валідацію даних.

У цілому, побудована серверна частина відповідає принципам сучасної backend-інженерії й демонструє практичну готовність до масштабування,

багатокористувацької взаємодії та інтеграції нових функціональних модулів у майбутньому.

2.1.2 Моделі бази даних

Для зберігання даних використовується PostgreSQL. Усі сутності реалізовані через ORM-моделі SQLAlchemy. Між таблицями налаштовані зв’язки типу one-to-many, many-to-one, many-to-many, а також каскадне видалення.

Основні моделі:

- `User` — зберігає email, зашифрований пароль, роль (користувач/адміністратор), стан активації.
- `Product` — назва, опис, ціна, фото, категорія, дата створення, залишок.
- `Order` — зв’язок із користувачем, сума, статус, дата.
- `OrderItem` — склад замовлення: товар, кількість, ціна.
- `CartItem` — тимчасове збереження товарів у кошику користувача.
- `PromoCode` — код, тип знижки, термін дії, максимальна кількість використань.
- `Review` — відгук користувача про товар: оцінка, текст, дата.
- `ViewHistory` — історія переглядів товарів конкретним користувачем (використовується для персоналізації).

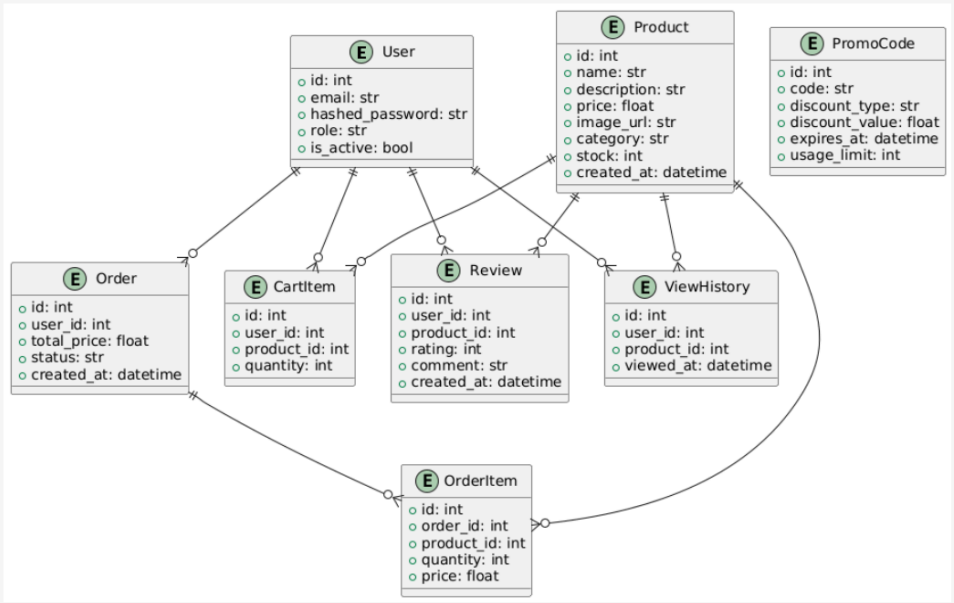


Рис 2.1.2 UML - діаграма бази даних.
Джерело: Автор

Завдяки SQLAlchemy кожна модель має чітку типізацію, декларативний синтаксис та підтримку фільтрації, зв'язків і вкладених запитів.

2.1.3 API та маршрутизація

Маршрутизація в FastAPI дозволяє чітко структурувати доступ до функцій серверної частини, розділивши їх за відповідними модулями. Кожен ресурс (користувачі, товари, замовлення тощо) має власний `router`, що містить набір REST-ендпоінтів, згрупованих за функціональним принципом.

Всього реалізовано понад **50 REST-маршрутів**. Вони охоплюють всі основні аспекти взаємодії з маркетплейсом:

- `/auth/` — логіка автентифікації, генерація та оновлення токенів, відновлення доступу;
- `/users/` — оновлення профілю, зміна паролю, перегляд власної інформації;
- `/products/` — створення, редагування, видалення та пошук товарів (CRUD);
- `/cart/` — додавання товарів у кошик, перегляд вмісту, зміна кількості;
- `/orders/` — оформлення нового замовлення, перегляд історії замовлень;

- `/promos/` — застосування промокодів, перевірка валідності, облік використань;
- `/reviews/` — залишення та модерація відгуків, підрахунок середнього рейтингу;
- `/recommendations/` — отримання персональних рекомендацій на основі активності користувача.

Кожен ендпоінт реалізовано з обов'язковою перевіркою прав доступу. Наприклад, доступ до `/products/create` має лише адміністратор. Усі маршрути містять опис, приклади запитів/відповідей і автоматично публікуються через Swagger UI за адресою `/docs`.

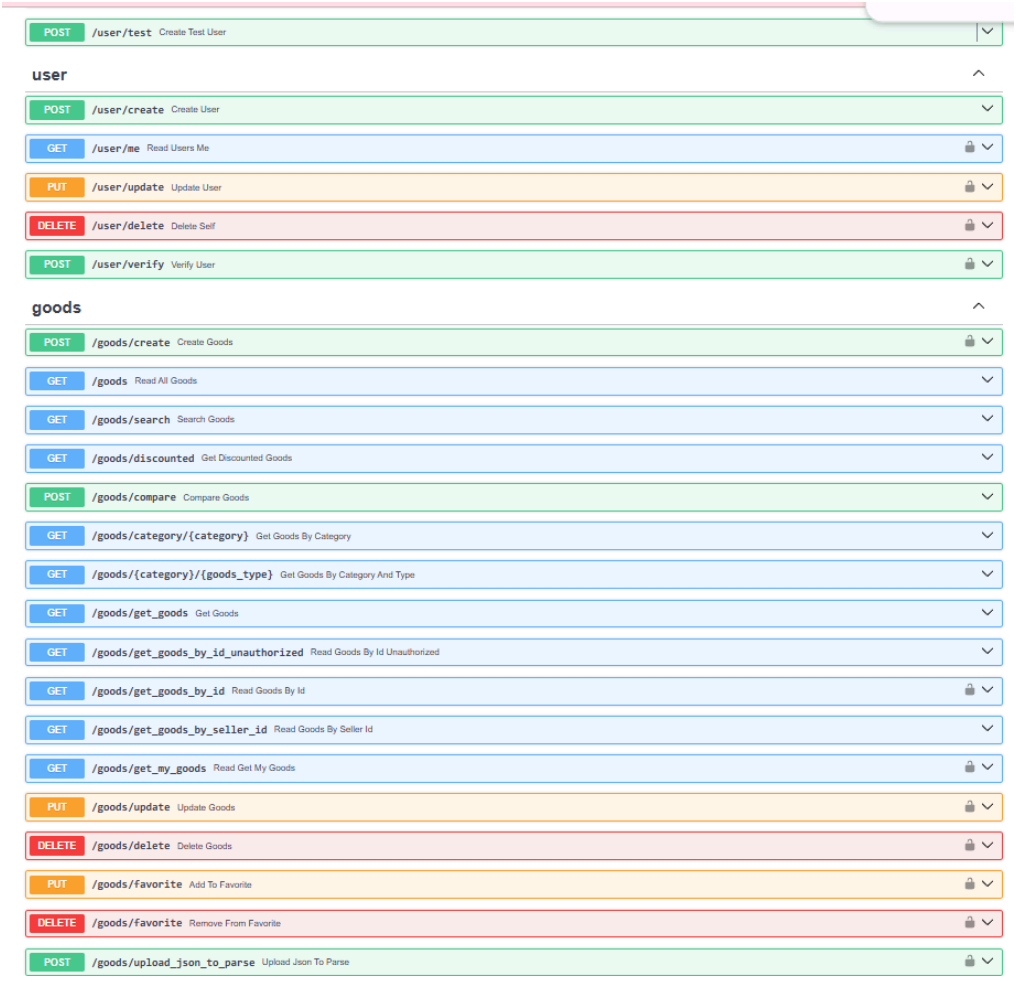


Рис 2.1.3 Список запитів.

Джерело: Автор

2.1.4 Рекомендаційний модуль

Система рекомендацій — важливий компонент маркетплейсу, який забезпечує персоналізований досвід для користувача, збільшує конверсію та середній чек. У поточній реалізації модуль працює на основі **контентно-орієнтованого підходу**.

Ключовим нововведенням стало повне оновлення логіки збереження історії переглядів. Якщо раніше дані про перегляди зберігалися тимчасово (локально в клієнтській частині або у сесії), то тепер **використовується збереження історії переглядів у базі даних**, у полі `view_history` моделі користувача. Це дозволяє:

- мати постійний доступ до історичних даних;
- використовувати їх для статистики, аналітики та покращення UX;
- формувати більш точні персоналізовані рекомендації.

Формування рекомендацій реалізоване через спеціальний сервіс, що працює за наступним алгоритмом:

1. Зчитується поле `view_history` користувача з бази даних;
2. Перетворюється у список ID товарів, які користувач переглядав;
3. Визначаються категорії або властивості цих товарів;
4. Вибираються нові товари з тих самих категорій, що мають високі рейтинги та не входять у список вже переглянутих;
5. Результат повертається у вигляді масиву об'єктів `Product` для подальшого рендерингу на клієнті.

Основні функції:

- `update_user_view_history()` — додає новий переглянутий товар у поле `view_history`, уникаючи повторів, зберігаючи порядок (останні переглянуті зверху);
- `get_recommendations_data()` — дістає список товарів на основі історії переглядів;
- `get_current_user()` — отримує користувача з токена JWT для захищеного доступу до персоналізованих даних.

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


def get_current_user(token: str = Depends(oauth2_scheme), db: Session =
Depends(get_db)):
    try:
        payload = jwt.decode(token, os.getenv("SECRET_KEY"),
algorithms=[os.getenv("ALGORITHM")])
        email: str = payload.get("sub")
        if email is None:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Невірний токен")
        user = get_user_by_email_or_phone(db, username=email)
        if user is None:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Користувач не знайдений")
        return user
    except JWTError:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Невірний токен")


def update_user_view_history(db: Session, user_id: int, goods_id: int):
    user = get_user(db, user_id)
    if not user:
        return None
    if user.view_history:
```

```
        history = user.view_history.split(",")
        if str(goods_id) in history:
            history.remove(str(goods_id))
        history.insert(0, str(goods_id))
        user.view_history = ",".join(history)
    else:
        user.view_history = str(goods_id)
    db.commit()
    db.refresh(user)
    return user



def get_recomendations_data(db: Session, user_id: int,):
    user = get_user(db, user_id)
    if not user:
        return None
    if user.view_history:
        history = user.view_history.split(",")
        goods = []
        for i in history:
            goods.append(db.query(models.Goods).filter(models.Goods.id == i).first())
        return goods
    else:
        return None
```

Лістинг. 2.1.1 Реалізація системи рекомендацій маркетплейсу.

Джерело: Автор

Цей алгоритм оптимізовано за часом виконання та кількістю запитів до бази. Замість складної фільтрації дані підготовлені один раз, а відбір рекомендацій відбувається динамічно на основі останніх дій користувача.

У майбутньому планується розширити можливості алгоритму за допомогою бібліотек `scikit-learn`, `surprise`, або векторної кластеризації типу `Nearest Neighbors`, що дозволить створити гібридну або колаборативну модель рекомендацій.

Таким чином, оновлена реалізація рекомендаційного модуля не лише підвищує точність рекомендацій, але й забезпечує реальне збереження активності користувача, покращуючи індивідуалізований досвід взаємодії з маркетплейсом.

2.1.5 Безпека та автентифікація

Захист даних і контроль доступу реалізовано відповідно до сучасних стандартів веббезпеки. В основі лежить **OAuth2**-механізм із використанням **JWT-токенів**, які зберігаються клієнтом у `localStorage` та передаються в заголовках запитів.

При вході користувача сервер:

- перевіряє email та пароль,
- генерує токен з часом життя (наприклад, 30 хвилин),
- додає до токена payload з роллю та ID користувача.

На всіх приватних маршрутах реалізовано middleware, що перевіряє дійсність токена. У разі невдачі — повертається код 401 Unauthorized.

Також реалізовано:

- шифрування паролів (`bcrypt`);
- підтвердження email шляхом надсилання одноразового коду;
- розмежування ролей: користувач / адміністратор;
- блокування підозрілих дій (наприклад, багаторазових спроб входу);
- rate-limiting для захисту від brute-force атак.

Ці механізми забезпечують високий рівень довіри до системи та захисту конфіденційних даних.

```
def hash_password(password: str) -> str:
    salt = bcrypt.gensalt()
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')
def verify_password(password: str, hashed_password: str) -> bool:
    return bcrypt.checkpw(password.encode('utf-8'), hashed_password.encode('utf-8'))
# def update_password(db: Session, user_id: int, password: str):
def is_valid_email(email):
    pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    if re.match(pattern, email):
        return False
    else:
        return True
def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now() + expires_delta
    else:
        expire = datetime.now() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, os.getenv("SECRET_KEY"),
algorithm=os.getenv("ALGORITHM"))
    return encoded_jwt
```

Лістинг 2.1.2 Реалізація верифікації користувача.

Джерело: Автор

2.1.6 Email-сервіс

Компонент email-розсилки виконує як функціональні, так і маркетингові завдання. Надсилання повідомлень здійснюється через:

- стандартний **SMTP-протокол** (наприклад, Gmail SMTP);
- або сторонній сервіс типу **SendGrid**, що надає API з підвищеною доставлюваністю.

Ключові події, які тригерять email:

- реєстрація користувача — лист із кодом верифікації;
- оформлення замовлення — підтвердження з деталями покупки;
- зміна статусу — повідомлення про етапи доставки;
- нагадування про покинутий кошик або промоакції з кодами знижок.

Використовуються шаблони HTML-листів із підстановками значень, таких як ім’я користувача, перелік товарів, сумарна вартість тощо. Шаблони формуються з використанням `Jinja2`, що дозволяє зручно управляти динамічним вмістом.

Цей модуль також містить логіку обліку відправлень і помилок доставки для подальшого моніторингу ефективності email-каналу.

```
async def send_verification_email(recipient_email: str, verification_code: str):
    subject = "Верифікація облікового запису"
    body = f"""Ваш код підтвердження: {verification_code}"""
    message = MessageSchema(
        subject=subject,
        recipients=[recipient_email],
        body=body,
        subtype="html"
    )
```

```
    fm = FastMail(conf)
    await fm.send_message(message)
    print(f"Лист із кодом верифікації надіслано на {recipient_email}")
async def send_verification_email(recipient_email: str, verification_code: str):
    subject = "Верифікація облікового запису"
    body = f"""Ваш код підтвердження: {verification_code}"""
    message = MessageSchema(
        subject=subject,
        recipients=[recipient_email],
        body=body,
        subtype="html"
    )
    fm = FastMail(conf)
    await fm.send_message(message)
    print(f"Лист із кодом верифікації надіслано на {recipient_email}")
```

Лістинг 2.1.3 Реалізація SMTP-протоколів.

Джерело: Автор

2.1.7 Валідація даних і структура схем

У межах реалізації серверної частини окрему увагу було приділено валідації даних, що надходять від користувача або надсилаються у відповідь із сервера. Для цього використано бібліотеку **Pydantic**, яка є офіційно рекомендованим інструментом у FastAPI для опису схем та валідації вхідних і вихідних даних.

Усі запити поділено на окремі схеми (Base, Create, Update, Response), що дозволяє:

- чітко розділити контексти використання (наприклад, реєстрація, редагування, відповідь);
- застосовувати різну логіку обробки для вхідних і вихідних даних;

- централізовано перевіряти типи, наявність, формат та діапазон значень;
- зберігати читабельність і модульність коду.

Схеми побудовано для всіх основних об’єктів: користувача, товару, кошика, замовлення, відгуків, промокодів. Наприклад, для моделі користувача (`User`) використано розділення на базову модель `UserBase`, моделі для створення `UserCreate`, оновлення `UserUpdate`, та відображення `User`. Це дозволяє гарантувати, що при створенні акаунта обов’язковими є email, ім’я, телефон, а при оновленні ці поля можуть бути опціональними.

Окремо реалізовано класи:

- `ResetPasswordSchema` — для відновлення паролю за токеном;
- `UserHistory` і `UserFavorite` — для передачі списків товарів (переглянуті, улюблені);
- `Token` — для обробки авторизації за email.

Подібну структуру мають і моделі товарів (`GoodsBase`, `GoodsCreate`, `GoodsUpdate`, `Goods`), де враховано специфікацію, зображення, знижки, тип, категорію тощо. Аналогічно описані замовлення, кошик, відгуки, промокоди.

Переваги такого підходу:

- **Гнучкість** — легко розширювати схеми без впливу на інші частини системи;
- **Безпека** — виключається ризик ін’єкцій чи некоректного типу даних;
- **Зручність** — у Swagger-документації автоматично відображаються всі поля, типи та приклади;
- **Інкапсуляція** — бізнес-логіка розділена від структури даних;
- **Масштабованість** — можна зберігати та трансформувати дані без зміни загальної логіки API.

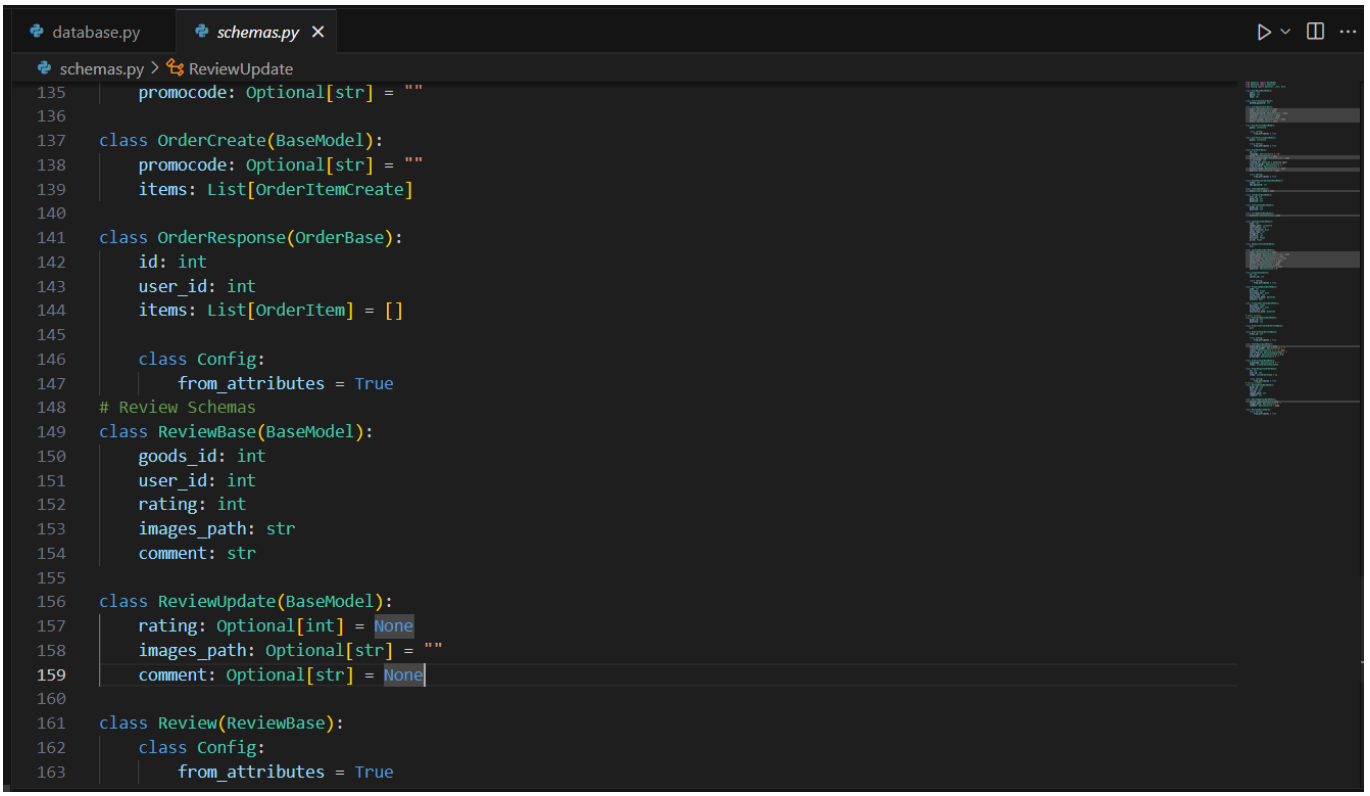


Рис. 2.1.4. Модуль schemas.py, в якому реалізована валідація даних.

Джерело: Автор

Кожна схема містить конфігурацію `from_attributes = True`, що дозволяє безпосередньо перетворювати ORM-об’єкти (SQLAlchemy) у Pydantic-моделі для зручного повернення у відповідь клієнту.

Загалом, реалізація структурованої валідації через Pydantic забезпечує не лише безпеку системи, а й високий рівень читаємості, прогнозованості та розширюваності всієї серверної архітектури. Це важливий аспект, що забезпечує відповідність розробки професійному рівню.

2.1.8 Реалізація парсингу та імпорту даних з JSON

У межах побудови функціональної серверної частини маркетплейсу було реалізовано можливість масового завантаження товарів через файл формату **JSON**. Це суттєво полегшує адміністрування платформи, особливо на етапі її початкового наповнення або імпорту каталогу від постачальника.

Парсер реалізований як окремий обробник, що дозволяє адміністратору або продавцю (із відповідними правами доступу) через **SwaggerUI** або API завантажити файл JSON-структури, що містить повну інформацію про товари.

Основна функція: `upload_json_to_parse()` приймає об'єкт `Session` бази даних, `seller_id` — ідентифікатор продавця, та словник `contents`, який є розпарсеним JSON-файлом. Далі за допомогою циклу обробляється кожен запис, створюється відповідна ORM-модель `Goods`, з усіма полями (назва, опис, ціна, зображення, характеристики, категорія, кількість, знижка) і записується до бази даних.

```
def upload_json_to_parse(db: Session, seller_id: int, contents: dict):
    goods_list = []
    try:
        for item in contents.values():
            db_goods = models.Goods(
                name=item["name"],
                description=item["description"],
                price=item["price"],
                images_path=item["images_path"],
                specification=item["specification"],
                goods_type=item["goods_type"],
                category=item["category"],
                quantity=item.get("quantity", 0),
                discount=item["discount"],
                seller_id=seller_id,
            )
            db.add(db_goods)
            db.commit()
            db.refresh(db_goods)
            goods_list.append(db_goods)
        return goods_list
    except Exception as e:
        return None
```

Лістинг 2.1.4. Код парсера для імпорту інформації в базу даних.

Джерело: Автор

Рис. 2.1.5. Вигляд запиту в SwaggerUI.
Джерело: Автор

Завантаження здійснюється у кілька кліків, після чого вся інформація з файлу одразу відображається в адміністративній панелі або у відповідних категоріях магазину. Це також спрощує парсинг сторонніх каталогів — наприклад, для попередньої версії було створено парсер для E-Katalog із використанням BeautifulSoup, що дозволив зібрати структуру товарів, зберегти її у форматі JSON і потім імпортувати в маркетплейс.

Такий механізм імпорту:

- Скорочує час початкового заповнення платформи;
- Уніфікує структуру даних, яку можна отримати з парсерів чи сторонніх сервісів;
- Дає змогу швидко оновлювати або мігрувати товари без ручного введення.

Функціональність парсера зручна для масштабування — у майбутньому її можна доповнити логікою перевірки дублікатів, логуванням, обробкою зображень або інтеграцією з API постачальників.

Загалом, реалізований парсер JSON-файлів став важливим інструментом підтримки платформи з боку адміністраторів, спростивши як імпорт, так і подальше тестування та наповнення маркетплейсу реальними даними.

2.2 Клієнтська частина вебзастосунку

2.2.1 Структура клієнтського застосунку

Розробка клієнтської частини почалась із проєктування структури, що забезпечує масштабованість, повторне використання компонентів і зручність у супроводі. Було прийнято рішення організувати директорії за логічним принципом: сторінки, компоненти, запити до API, хуки, контексти. Така модульна структура дозволяє легко орієнтуватися в проєкті навіть при збільшенні кількості функцій і розробників.

Кожна сторінка — це окремий файл або набір компонентів, які відповідають за певну ділянку інтерфейсу (головна сторінка, сторінка товару, кошик, особистий кабінет тощо). Компоненти повторного використання (наприклад, кнопки, інпут-поля, картки товарів) винесені в окрему папку.

Цей підхід відповідає сучасним рекомендаціям по організації React-застосунків і дозволяє масштабувати застосунок до десятків сторінок без втрати стабільності й читабельності коду.

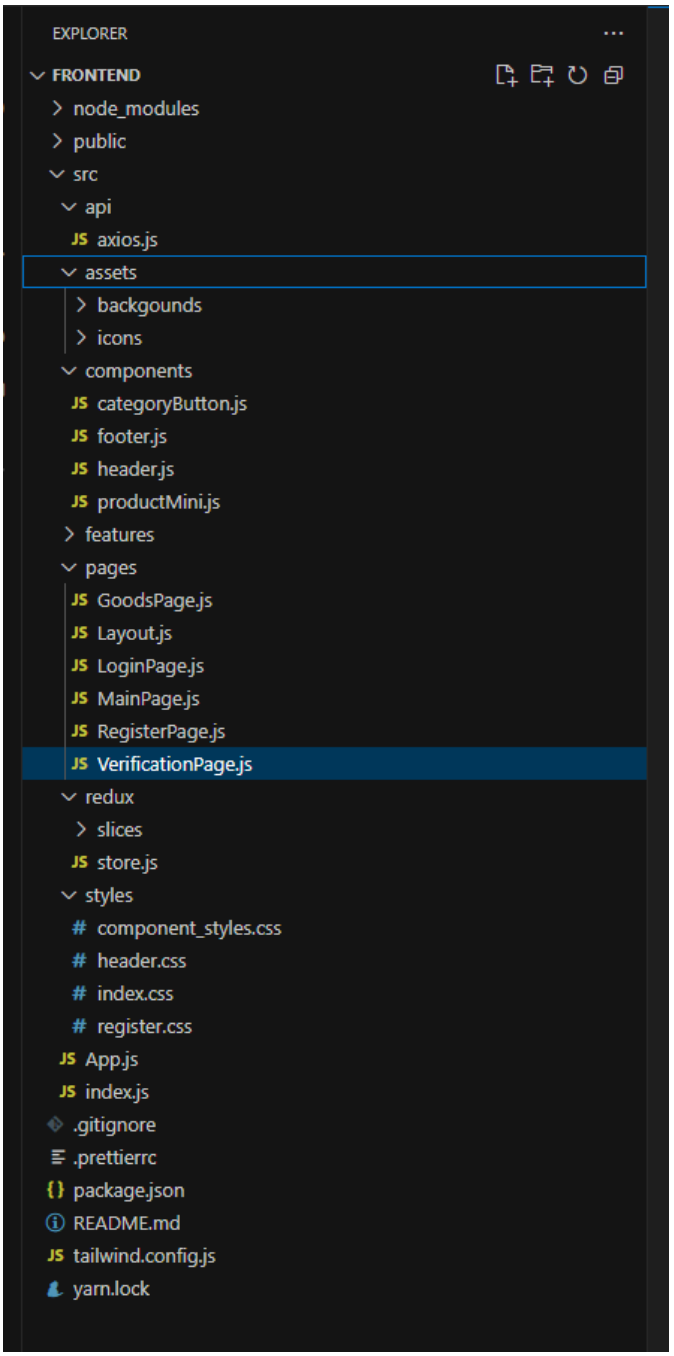


Рис. 2.2.1. Архітектура Frontend-частини

Джерело: Автор

2.2.2 Компоненти інтерфейсу

Кожна візуальна частина інтерфейсу реалізована у вигляді незалежного **функціонального React-компонента**, що базується на сучасних можливостях (React Hooks, контексти, props). Перевага такого підходу — у простоті повторного використання, тестування та ізоляції.

Застосунок включає компоненти:

- Статичні: `Header`, `Footer`, `Navbar`;

- Динамічні: `ProductCard`, `ProductList`, `CartItem`, `ReviewList`, `RecommendationBlock`;
- Формові: `LoginForm`, `RegisterForm`, `OrderForm`;
- Системні: `PrivateRoute`, `AdminGuard`, `ErrorBoundary`.

Особлива увага приділена UX: усі компоненти реагують на дії користувача (натискання, наведення, валідацію введення) та забезпечують приємний візуальний досвід. Компоненти адаптовані до мобільних пристроїв завдяки використанню TailwindCSS і flex/grid верстки.

2.2.3 Взаємодія з API

Уся логіка взаємодії з серверною частиною зосереджена у спеціальному модулі `api/`, де реалізовано функції запитів до бекенду через **Axios**. Ця бібліотека була обрана через її стабільність, зручність у конфігурації, наявність інтерсепторів і можливість обробляти глобальні помилки (наприклад, 401 або 500).

Використання Axios дозволило централізовано обробляти JWT-токени — автоматично додавати їх до заголовків запитів, а також обробляти помилки, наприклад, закінчення сесії або помилкову авторизацію.

Кожен запит — це окрема функція типу:

```
// Створюємо екземпляр Axios
const instance = axios.create({
  baseURL: API_URL,
  headers: {
    "Content-Type": "application/json",
  },
});
// Функція для оновлення токена
const refreshAccessToken = async () => {
```

```
  try {
    const refreshToken = localStorage.getItem("refreshToken");
    if (!refreshToken) throw new Error("Refresh token is missing");
    const response = await axios.post(`${API_URL}/refresh-token`, null, {
      headers: {
        Authorization: `Bearer ${refreshToken}`,
      },
    });
    const newAccessToken = response.data.access_token;
    localStorage.setItem("accessToken", newAccessToken); // Оновлюємо токен
    return newAccessToken;
  } catch (error) {
    console.error("Помилка оновлення токена:", error);
    localStorage.removeItem("accessToken");
    localStorage.removeItem("refreshToken");
    window.location.href = "/login"; // Перенаправлення на сторінку входу
    return null;
  }
};
```

Лістинг 2.2.1. Приклад використання Axios у проекті.

Джерело: Автор

Це дозволяє легко повторно використовувати запити у різних компонентах та тестувати їх із використанням мокових даних.

2.2.4 Навігація та маршрутизація

Для реалізації маршрутизації використовується бібліотека **React Router v6**, яка забезпечує інтуїтивне створення односторінкового застосунку. Це дозволяє користувачу переходити між сторінками без перезавантаження (SPA-архітектура). Ключові маршрути:

- `/` — домашня сторінка;
- `/products/:id` — деталі товару;
- `/cart` — поточний кошик;
- `/profile` — профіль користувача;
- `/admin` — адміністративна панель.

В особливості спочатку підключаємо модуль “react-router-dom” для того щоб використовувати метод useNavigate та Link які дозволяють переходити між сторінками.

```
import { useNavigate, Link } from "react-router-dom";
```

Лістинг 2.2.2. Використання модуля для навігації.

Джерело: Автор

Приклад використання функції useNavigate для програмної навігації:

```
const navigate = useNavigate();
if (isAuthenticated()) {
    navigate("/");
}
```

Лістинг 2.2.3. Використання метода навігації.

Джерело: Автор

Та в файлі App.js була реалізована навігація всього додатку.

```
import { BrowserRouter, Routes, Route } from "react-router-dom";

function App() {
  return (
    <BrowserRouter>
```

```
    <Routes>
      <Route element={<Layout />}>
        <Route path="/" element={<MainPage />} />
        <Route path="/register" element={<RegisterPage />} />
        <Route path="/login" element={<LoginPage />} />
        <Route path="/verify" element={<VerificationPage />} />
        <Route path="/goods/:id" element={<GoodsPage />} />
      </Route>
    </Routes>
  </BrowserRouter>
 );
}
```

Лістинг 2.2.4. Реалізація навгаціх в проекті.

Джерело: Автор

Компонент Layout містить Outlet, який визначає місце для вкладених маршрутів.

```
import { Outlet } from "react-router-dom";

const Layout = () => {
 return (
  <>
    <Header />
    <Outlet />
    <Footer />
  </>
 );
};
```

Лістинг 2.2.5. Розташування елементів в Layout.

Джерела: Автор

Особливістю реалізації є захист приватних маршрутів за допомогою компонента `PrivateRoute`, який перевіряє токен авторизації перед наданням доступу. Також реалізовано `AdminGuard` для обмеження доступу лише для ролі адміністратора. Це гарантує, що конфіденційна інформація не буде доступною неавторизованим користувачам.

2.2.5 Обробка стану та подій

Для керування станами в застосунку використано базові React-хуки — `useState`, `useEffect`, а також контексти (`useContext`) для глобальних станів, таких як сесія користувача або вміст кошика.

Завдяки контекстам:

- Стан авторизації доступний у будь-якій частині застосунку;
- Кількість товарів у кошику автоматично оновлюється при додаванні чи видаленні;
- Дані про користувача доступні на будь-якій сторінці.

useState Використовується для збереження стану форм, повідомлень, кількостей, результатів пошуку тощо.

```
const [formData, setFormData] = useState({
  name: "",
  email: "",
  phone: "",
  countryCode: "+380",
  password: "",
  confirmPassword: "",
});

const [error, setError] = useState("");
const [success, setSuccess] = useState(false);
const [searchResults, setSearchResults] = useState([]);
const [cartCount, setCartCount] = useState(0);
const [favoriteCount, setFavoriteCount] = useState(0);
```

Лістинг 2.2.6. Використання **useState** для збереження стану форм.

Джерело: Автор

useEffect застосовується для:

- перевірки авторизації;
- завантаження повідомлень;
- запитів до API під час завантаження компонента;
- нескінченного скрола;
- оновлення даних при зміні залежностей (query, offset, тощо).

```
useEffect(() => {
```

```
  if (isAuthenticated()) {
    navigate("/");
  }
}, [navigate]);

useEffect(() => {
  api.get("/", { headers: { skipAuth: true } })
    .then((response) => setMessage(response.data.message))
    .catch((error) => console.error("Error fetching data:", error));
}, []);

useEffect(() => {
  const token = localStorage.getItem("accessToken");
  if (!token) return;
  api.get("/tool/favorite_length")
    .then((response) => setFavoriteCount(response.data))
    .catch((error) => console.error("Error fetching favorite length:", error));
}, []);
```

Лістинг 2.2.7. Приклад використання useEffect.

Джерело: Автор

Все це знайдено у компонентах на сторінках:

- RegisterPage.js
- LoginPage.js
- MainPage.js
- Header.js
- VerificationPage.js

та інших у папці frontend/src/pages/frontend/src/components/. Приклад коду знаходиться в лістингу

```
const [formData, setFormData] = useState({
  name: "",
  email: "",
  phone: "",
  countryCode: "+380",
  password: "",
  confirmPassword: "",
});
```

```
const [errors, setErrors] = useState({
  password: "",
  confirmPassword: "",
});

const [error, setError] = useState("");
const [success, setSuccess] = useState(false);
```

Лістинг 2.2.7. Приклад використання цих методів в проекті.

Джерело: Автор

Також реалізовано обробники подій: наприклад, при натисканні кнопки "Додати до кошика" оновлюється глобальний стан і надсилається запит на бекенд. Реалізовано сповіщення (notifications) про успішні або помилкові дії через Toast або модальні вікна.

Висновки до розділу 2

Розробка клієнтської частини вебзастосунку маркетплейсу на базі бібліотеки React дозволила реалізувати інтуїтивно зрозумілий, адаптивний та ефективний інтерфейс для взаємодії з користувачем. Компонентний підхід, модульна структура, розділення відповідальностей між сторінками, компонентами та контекстами зробили кодову базу зручною для масштабування та супроводу.

Серед ключових досягнень слід виділити:

- Створення повноцінного SPA з підтримкою маршрутизації, приватних маршрутів і обмеженням доступу на основі ролей.
- Реалізацію динамічних компонентів для відображення товарів, рекомендацій, кошика та історії замовлень.
- Інтеграцію з REST API через Axios з обробкою JWT-токенів, помилок і статусів сесії.
- Створення логіки взаємодії користувача із застосунком: форми реєстрації/авторизації, обробка замовлень, додавання до кошика тощо.
- Застосування TailwindCSS як основного інструмента стилізації, що дозволило швидко досягти сучасного та уніфікованого візуального вигляду інтерфейсу.

У результаті розроблена клієнтська частина є повноцінним фронтенд-рішенням для маркетплейсу, здатним взаємодіяти з сервером у режимі реального часу, забезпечуючи ефективний обмін інформацією та комфортний користувацький досвід. У разі необхідності функціонал можна розширити без значних архітектурних змін — наприклад, додати мультимовність, адаптацію під мобільні пристрої, чат із підтримкою або інтеграцію з мобільними додатками.

Таким чином, клієнтська частина проєкту повністю задовольняє вимоги до сучасного інтерфейсу вебзастосунку типу маркетплейс.

РОЗДІЛ 3. ТЕСТУВАННЯ, РОЗГОРТАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ

3.1 Тестування вебзастосунку

Тестування є невід’ємною частиною життєвого циклу програмного забезпечення, що дозволяє виявити помилки, перевірити правильність логіки реалізації та забезпечити стабільність роботи системи. У межах цього проєкту було застосовано переважно **ручне тестування**, зосереджене на основних функціональних сценаріях використання системи.

Мета тестування:

- Перевірити працездатність REST API;
- Перевірити коректність логіки на стороні клієнта;
- Переконатися в наявності валідації форм;
- Перевірити взаємодію між клієнтом і сервером;
- Перевірити навігацію, авторизацію, захист доступу до приватних сторінок.

Інструменти, що використовувались:

- **Postman** — для тестування API-запитів (GET, POST, PUT, DELETE);
- **Інструменти розробника браузера (DevTools)** — для перевірки мережевих запитів та локального сховища;
- **Консоль браузера** — для виявлення помилок JavaScript або React;
- **Google Chrome / Firefox** — для кросбраузерної перевірки.

API-тестування (backend)

Було створено набір кейсів у Postman для перевірки таких модулів:

- Реєстрація та авторизація користувача (включаючи некоректні дані);
- Додавання товарів (для адміністратора);
- Перегляд списку товарів та окремого товару;
- Додавання товару в кошик, редагування кількості;
- Оформлення замовлення;
- Застосування промокоду;
- Створення та перегляд відгуків;
- Отримання рекомендацій на основі історії переглядів.

Для кожного ендпоінту перевірялись:

- HTTP-метод і статус відповіді (200, 400, 401, 403, 404);
- Структура JSON-відповіді;
- Наявність обов’язкових полів у тілі запиту/відповіді;
- Поведінка при відсутності токена авторизації.

Тестування клієнтської частини

Було здійснено ручне проходження основних сценаріїв використання на стороні користувача:

- Реєстрація та вхід з відображенням помилок при неправильних даних;
- Навігація між сторінками (головна, товар, кошик, профіль);
- Заповнення форм (огляд, оформлення замовлення, реєстрація);
- Захист доступу до профілю/адмін-панелі без авторизації;
- Оновлення інтерфейсу при додаванні до кошика або виході з профілю;
- Адаптивність інтерфейсу на різних розмірах екранів (мобільний/десктоп);
- Виведення повідомлень про помилки/успіх (Toast або alert).

Приклад тест-кейсів (фрагмент таблиці)

№	Сценарій	Вхідні дані	Очікуваний результат	Статус
1	Реєстрація нового	Валідні email, пароль	201 Created + токен	+

	користувача			
2	Авторизація з невірним паролем	Email, хибний пароль	401 Unauthorized	+
3	Отримання товарів	GET /products	JSON-масив об’єктів	+
4	Додавання в кошик	POST /cart/add	200 OK, оновлення	+
5	Оформлення замовлення без товарів	Порожній кошик	400 Bad Request	+
6	Вхід у /admin без токена	-	403 Forbidden	+

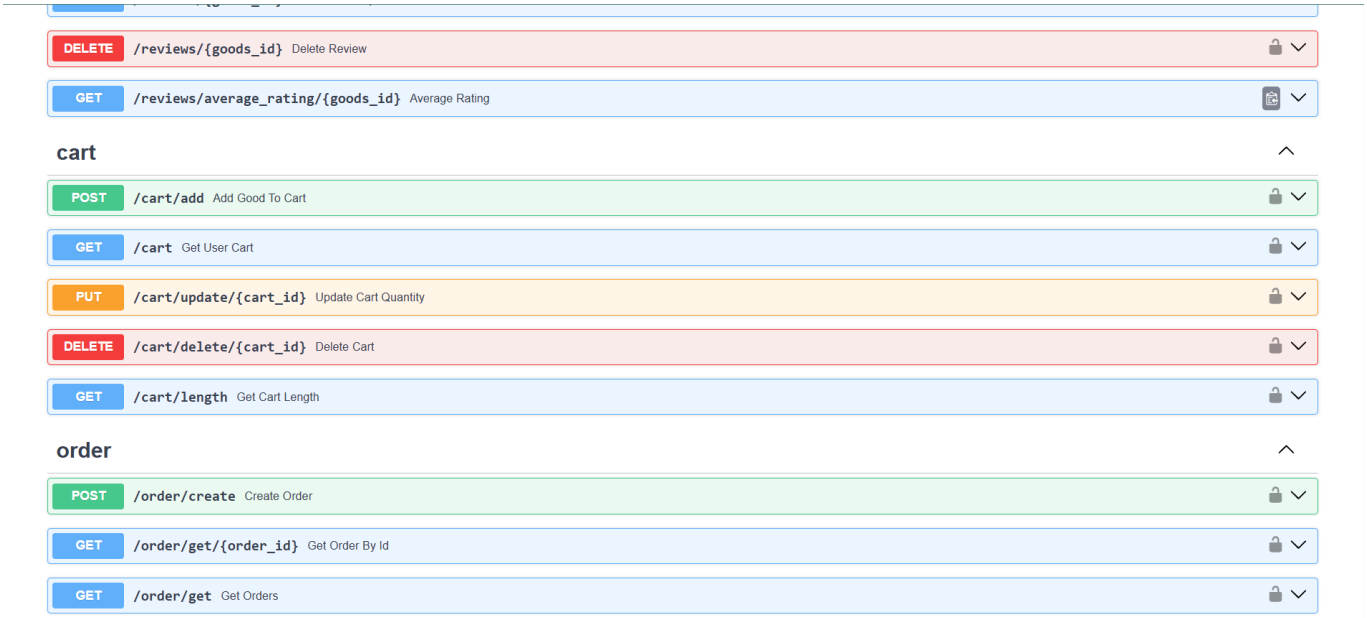


Рис 3.1.1 Приклад запитів у SwaggerUI

Джерело: Автор

Таким чином, ручне тестування дозволило охопити критичні шляхи взаємодії користувача з системою, перевірити функціональність API та frontend, забезпечити валідацію введення і контроль доступу. Виявлені незначні недоліки (наприклад, некоректне очищення форми) були оперативно усунені в процесі розробки.

У наступному підрозділі буде розглянуто процедуру розгортання вебзастосунку та налаштування середовища.

3.2 Розгортання вебзастосунку

Розгортання вебзастосунку — це процес перенесення проєкту з локального середовища розробки у загальнодоступне серверне середовище, де він може бути використаний кінцевими користувачами.

Етапи розгортання:

1. **Налаштування серверного середовища**
   - Встановлено Python 3.11 та необхідні бібліотеки (`pip install -r requirements.txt`);
   - Встановлено Node.js та NPM для запуску клієнтської частини;
   - Встановлено PostgreSQL та створено базу даних з правами доступу;
   - Підготовлено середовище `.env` для зберігання конфіденційних параметрів (ключі, URI бази, email).
2. **Запуск серверної частини (FastAPI)**
   - Використано Uvicorn для запуску: `uvicorn main:app --host 0.0.0.0 --port 8000`;
   - Забезпечено доступ до API через публічну IP-адресу;
   - Налаштовано `CORS` для взаємодії з клієнтом;
   - Створено міграції бази даних за допомогою Alembic.
3. **Запуск клієнтської частини (React)**
   - Проведено компіляцію: `npm run build` для створення продакшн-версії;
   - Готовий застосунок розгорнуто на безкоштовному хостингу (наприклад, Netlify, Vercel або Render);
   - Або локально на сервері за допомогою сервера типу `serve` або Nginx (наприклад: `npx serve -s build`).
4. **База даних PostgreSQL**

- Створено схему бази на основі ORM-моделей;
- Забезпечено з’єднання через `DATABASE_URL` у `.env`;
- Перевірено міграції, автентифікацію, збереження замовлень і даних.

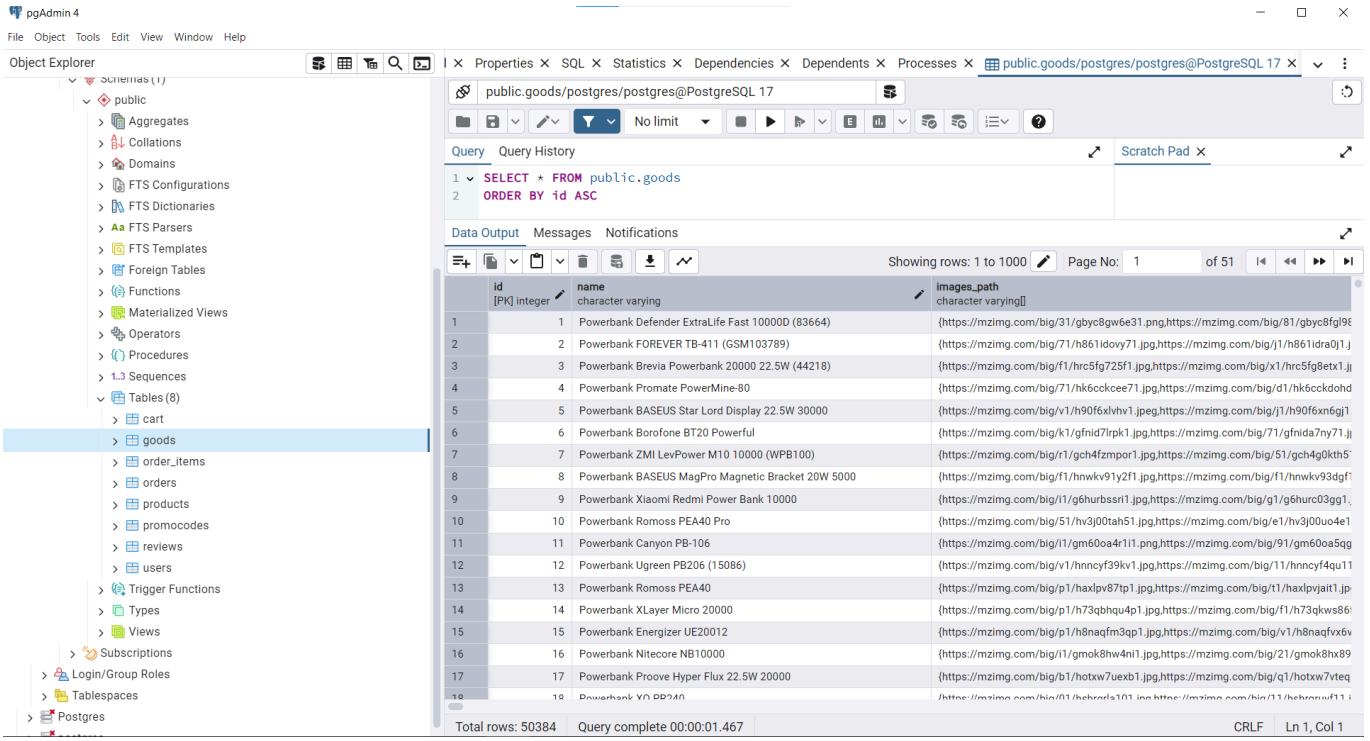


Рис 3.2.1 База-даних на основі PostgreSQL (pgAdmin 4)

Джерело: Автор

5. **Налаштування домену та SSL (за потреби)**
   - Для тестової версії використано безкоштовний домен + HTTPS (через хостинг);
   - Для продакшн-системи рекомендовано використання Let's Encrypt або Cloudflare.

Результат розгортання

У результаті застосунок було успішно перенесено на віддалене середовище, протестовано доступність із зовнішніх пристроїв, перевірено повну працездатність основних функцій: перегляд товарів, робота з кошиком, авторизація, замовлення, надсилання пошти. Структура системи дозволяє масштабувати її в майбутньому без

радикальних змін у розгортанні. Усі змінні середовища розділені від основного коду, що підвищує безпеку.

У наступному підрозділі буде проведено оцінку ефективності системи.

3.3 Оцінка ефективності розробленої системи

Розроблена система маркетплейсу відповідає всім базовим вимогам до сучасних вебзастосунків електронної комерції. У ході реалізації було досягнуто високого рівня інтегрованості компонентів, зручності інтерфейсу та стабільної роботи серверної частини. Тестування показало коректну поведінку в більшості сценаріїв використання, а структура проєкту дозволяє гнучко масштабувати функціонал у майбутньому.

Оцінюючи ефективність проєкту, варто розглянути:

1. Відповідність початковим вимогам

Проєкт охоплює повний життєвий цикл користувача: реєстрація, перегляд товарів, додавання в кошик, оформлення замовлення, залишення відгуків, перегляд персональних рекомендацій. Система також передбачає рольову модель доступу (адміністратор/користувач), валідацію форм і захист API.

2. Обґрунтування вибору технологій

У процесі розробки було використано оптимальні інструменти, зважаючи на мету створення повноцінного, але простого у впровадженні MVP-рішення:

- **FastAPI** обрано за його асинхронність, мінімалізм та автоматичну генерацію документації. Він дозволяє створювати стабільні REST API з невеликою кількістю коду.
- **PostgreSQL** забезпечує реляційні зв’язки, індексацію, транзакції та добру інтеграцію з SQLAlchemy. Це стабільна і масштабована система для ecommerce-проєктів.
- **React** дає можливість швидко будувати масштабовані SPA-застосунки із розподіленням за компонентами. Його широка екосистема підтримує швидкий розвиток проєкту.

- **TailwindCSS** дозволяє ефективно стилізувати інтерфейс без надмірного CSS-коду. Це спрощує адаптацію до мобільних пристроїв.
- **Axios** і **React Router** забезпечують гнучкість при роботі з API та навігацією, відповідно.

Обрані технології добре документовані, активно підтримуються спільнотою і мають низький поріг входу, що дозволяє швидко навчатися та масштабувати команду розробки в майбутньому.

3. Практична користь

Система може бути застосована як основа для навчальних проєктів, стартапів або малих онлайн-магазинів. Її можна легко доопрацювати — додати підтримку платежів, інтеграцію зі службами доставки, багатомовність тощо. Завдяки відкритій архітектурі, розширення не потребує переписування коду з нуля.

4. Масштабованість

Система підтримує чітке розділення логіки: кожен модуль (авторизація, каталог, кошик, замовлення) є автономним. Це дозволяє паралельно працювати над різними частинами, реалізовувати нові функції (чат, рекомендатор на основі ML, push-сповіщення) без порушення існуючого коду.

5. Обмеження та шляхи розвитку

Наразі відсутня автоматизована система CI/CD, не використовується контейнеризація, не реалізована підтримка багатьох мов. Проте ці функції можуть бути додані на наступних етапах. Завдяки модульності структури, інтеграція нових сервісів не спричинить складнощів.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено вебзастосунок типу маркетплейс, який поєднує в собі функціональність сучасної системи електронної комерції, оптимізовану архітектуру, індивідуалізований підхід до користувача та можливість масштабування в майбутньому.

На основі аналізу конкурентних рішень було сформовано вимоги до функціоналу, який охоплює: реєстрацію/авторизацію, управління товарами, кошик, систему замовлень, механізм відгуків та модуль персоналізованих рекомендацій. Ці вимоги стали основою технічного проєктування та програмної реалізації.

Серверна частина системи реалізована з використанням FastAPI, що дало змогу створити продуктивний REST API із чітким поділом логіки на модулі `models`, `schemas`, `routers`, `crud`, `services`. Значну увагу приділено структурі проєкту, що дозволило спростити навігацію по коду та підвищити його масштабованість. Окремо впроваджено систему автентифікації через JWT, систему валідації даних на основі Pydantic, а також новий підхід до збереження історії переглядів користувача у базі даних.

Клієнтська частина побудована за допомогою React, що дозволило реалізувати гнучкий SPA-інтерфейс із сучасною адаптивною стилізацією на базі TailwindCSS. Функціональність інтерфейсу включає навігацію, роботу з формами, взаємодію з API, систему повідомлень, захищені маршрути та контекстне керування станом.

Окремо було реалізовано рекомендаційний модуль, що аналізує збережену в базі даних історію переглядів користувача, визначає пріоритетні категорії та надає персоналізовані рекомендації. Це дозволяє формувати індивідуальний досвід для кожного користувача та покращувати ефективність сервісу.

Проведено повний цикл ручного тестування з використанням Postman і перевіркою всіх основних сценаріїв взаємодії користувача з системою. Після завершення основного етапу розробки система була успішно розгорнута на сервер

без використання контейнеризації чи CI/CD, з можливістю масштабування та хостингу.

Розроблений вебзастосунок може бути використаний як база для реального бізнес-проєкту або освітнього MVP-рішення, що демонструє повний цикл створення інформаційної системи — від формування ідеї до публікації та інтеграції.

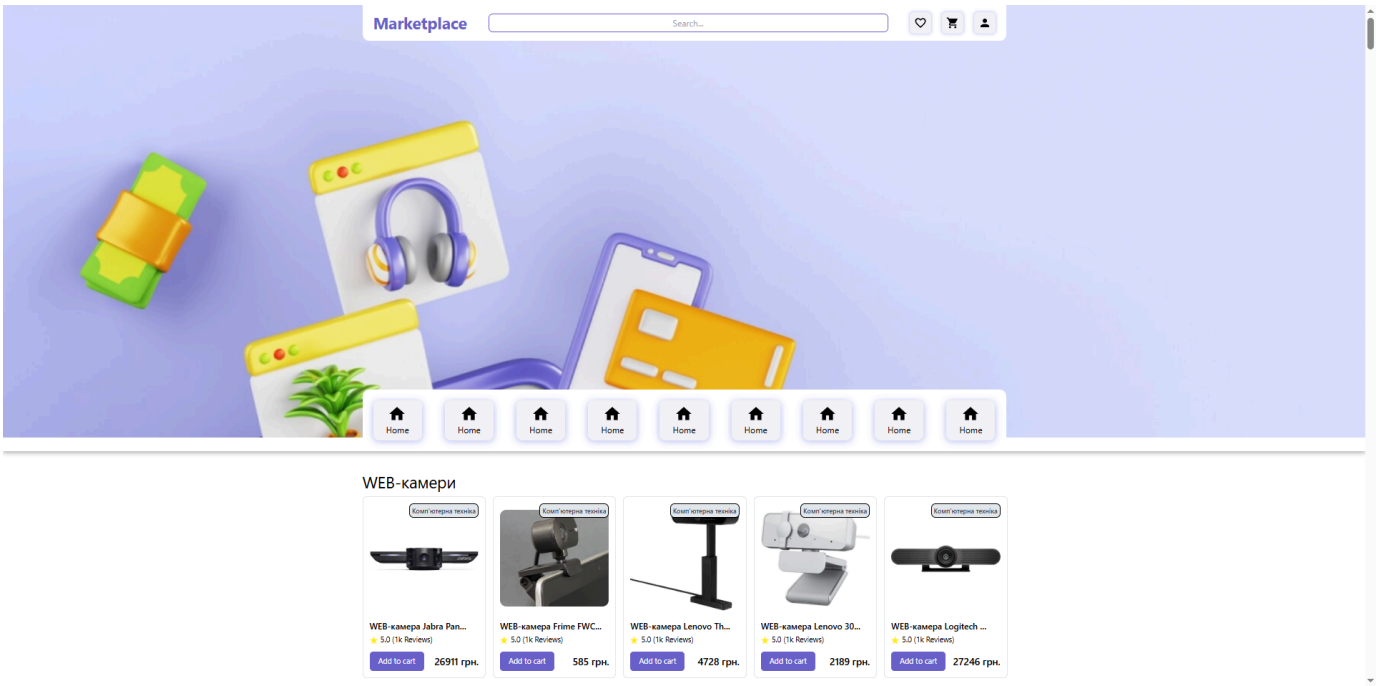
Таким чином, у межах дипломної роботи виконано комплексну розробку, проєктування та впровадження системи маркетплейсу із сучасним стеком технологій, дотриманням архітектурних стандартів і врахуванням потреб майбутнього розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

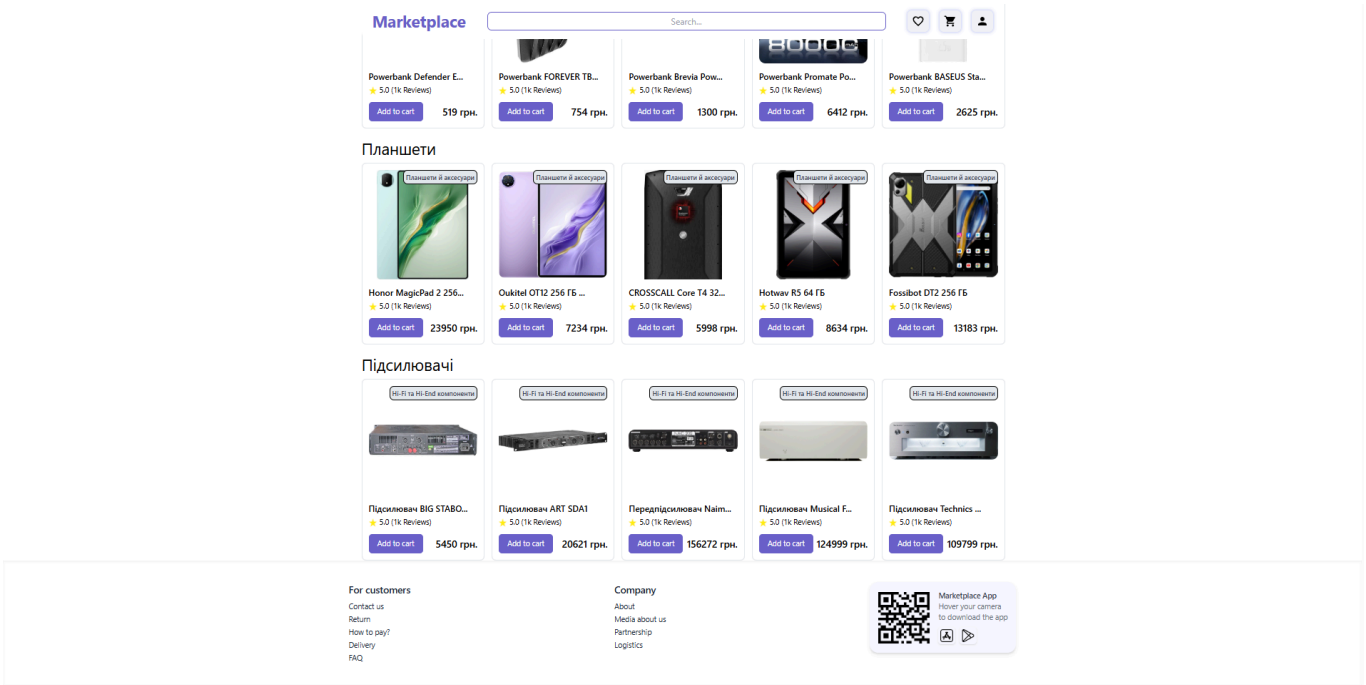
1. Documentation – FastAPI. https://fastapi.tiangolo.com
2. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org
3. React – A JavaScript library for building user interfaces. https://reactjs.org
4. TailwindCSS – Rapidly build modern websites without ever leaving your HTML. https://tailwindcss.com
5. Axios HTTP Client. https://axios-http.com
6. SQLAlchemy Documentation. https://docs.sqlalchemy.org
7. Pydantic Documentation. https://docs.pydantic.dev
8. Alembic Database Migration Tool. https://alembic.sqlalchemy.org
9. React Router Documentation. https://reactrouter.com
10. Mozilla Developer Network (MDN). https://developer.mozilla.org
11. GitHub – Examples of E-commerce projects. https://github.com
12. Beautiful Soup Documentation. https://www.crummy.com/software/BeautifulSoup
13. E-katalog. https://www.e-katalog.ua
14. Python official documentation. https://docs.python.org/3/

ДОДАТКИ

Додаток А



Додаток А. 1 Вигляд дизайну сторінки каталогу



Додаток А. 2 Вигляд дизайну сторінки каталогу