ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «модульна E-commerce система з підтримкою багатокористувацького адміністрування»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Максим ВАСИЛЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Максим ВАСИЛЕНКО

Керівник: Віталій ЗАЛИВА
доктор філософії (PhD).

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Василенку Максиму Вікторовичу

1. Тема кваліфікаційної роботи: «модульна E-commerce система з підтримкою багатокористувацького адміністрування»
керівник кваліфікаційної роботи доцент кафедри ІПЗ, доктор філософії (PhD) Віталій ЗАЛИВА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про принципи побудови електронних магазинів, аналіз сучасних підходів до реалізації мультирольовоц системи, технічна документація платіжних шлюзів Stripe, headless CMS-системи Sanity, сервісів автентифікації Clerk, методи побудови повностекових web-застосунків на базі Next.js та React.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної області, сучасних CMS-рішень та проєктування архітектури full-stack застосунку

2. Програмна реалізація серверної та клієнтської частин на базі стеку Next.js, React та TypeScript.
3. Інтеграція зовнішніх сервісів Sanity, Stripe, Clerk та розробка системи лояльності користувачів.
4. Тестування працездатності застосунку та підготовка демонстраційних матеріалів проєкту.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Архітектура вебзастосунку.
5. Схема Sanity.
6. Діаграма класів.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд сучасних технологій електронної комерції та методів побудови full-stack застосунків	08.03-15.03.2026	
4	Проєктування модульної системи електронної комерції з підтримкою розподіленого адміністрування	16.03-25.03.2026	
5	Програмна реалізація застосунку	26.03-20.04.2026	
6	Тестування застосунку	21.04-28.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	29.04-05.05.2026	
8	Розробка демонстраційних матеріалів	06.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____ Максим ВАСИЛЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Віталій ЗАЛИВА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 64 стор., 4 табл., 12 рис., 27 джерел.

Мета роботи – мінімізація витрат технічних ресурсів на розгортання та підтримку front-end інфраструктури за рахунок використання сучасного фреймворку Next.js та хмарного сервісу ідентифікації Clerk.

Об'єкт дослідження – процеси створення масштабованих модульних E-commerce рішень на базі сучасних веб-стандартів HTML/CSS.

Предмет дослідження – програмне забезпечення модульною E-commerce системи з підтримкою багатокористувацького адміністрування.

Короткий зміст роботи: В роботі проаналізовано архітектурні патерни та методи розробки систем електронної комерції. Проаналізовано інструментальні засоби та платформи для розгортання онлайн-магазинів: Shopify, WooCommerce,. Розроблено архітектуру вебзастосунку та програмно реалізовані ключові функціональні можливості: автентифікація та авторизація користувачів, управління контентом через Headless CMS, інтеграція платіжного шлюзу, менеджмент глобального стану та обробка транзакційних повідомлень. Проведено функціональне та модульне тестування додатку. Використано фреймворк Next.js та бібліотеку React для створення інтерфейсу, Sanity для керування даними, Clerk для безпеки, Stripe для еквайрингу, і ще Zustand, Firebase та Nodemailer для забезпечення бізнес-логіки.

Сферою використання застосунку є роздрібна онлайн-торгівля та автоматизація бізнес-процесів у сфері електронної комерції.

КЛЮЧОВІ СЛОВА: ВЕБЗАСТОСУНОК, ЕЛЕКТРОННА КОМЕРЦІЯ, NEXT.JS, REACT, TYPESCRIPT, SANITY, HEADLESS CMS, CLERK, STRIPE, ZUSTAND, TAILWIND CSS, RADIX UI, FIREBASE, СЕРВЕРНІ КОМПОНЕНТИ, СЕРВЕРНІ ДІЇ, МУЛЬТИРОЛЬОВИЙ ДОСТУ

ЗМІСТ

ВСТУП

Актуальність роботи. Сфера електронної комерції продовжує активно розвиватися у 2025–2026 роках та залишається одним із найдинамічніших сегментів цифрової економіки. За даними Statista, у 2025 році світовий обсяг ринку e-commerce перевищив 7 трлн доларів США, а частка онлайн-продажів у світовій роздрібній торгівлі наблизилася до 24%. Прогнозується, що у 2026 році цей показник продовжить зростати. В Україні також спостерігається стабільне збільшення кількості онлайн-замовлень, що пов’язано з розвитком цифрових сервісів, мобільних платежів та зростанням популярності маркетплейсів.

Разом зі збільшенням кількості користувачів зростають і вимоги до сучасних e-commerce-систем. У 2025 році понад 65% користувачів здійснюють покупки зі смартфонів або планшетів, тому швидкість завантаження сторінок та адаптивність інтерфейсу стали критично важливими характеристиками вебзастосунків. За статистикою Google затримка завантаження сторінки більше ніж на 3 секунди може збільшити ймовірність втрати користувача більш ніж на 30%. Це створює потребу у використанні сучасних фронтенд-технологій та оптимізованого серверного рендерингу.

Популярні платформи для створення інтернет-магазинів, такі як Shopify, WooCommerce і Magento, мають ряд обмежень. SaaS-рішення обмежують можливості кастомізації та прив’язують бізнес до власної інфраструктури платформи. Рішення на базі WordPress часто використовують застарілий стек технологій, а enterprise-системи потребують значних ресурсів для підтримки та масштабування. Крім того, значна частина таких платформ не орієнтована на сучасний стек React і TypeScript, який активно використовується у fullstack-розробці у 2025–2026 роках.

Однією з ключових тенденцій сучасної веброзробки є перехід до headless-архітектури. У цьому підході frontend і backend взаємодіють через API та можуть розроблятися незалежно один від одного. За даними Gartner, у 2026 році понад

70% великих e-commerce платформ використовують або впроваджують headless-рішення. Це дозволяє одночасно підтримувати вебверсію, мобільні застосунки і інші клієнтські сервіси на базі єдиного джерела даних.

Важливу роль у розвитку сучасних вебзастосунків також відіграють нові можливості React і Next.js. Використання React Server Components, App Router і Server Actions дозволяє переносити частину логіки на сервер, зменшувати кількість JavaScript-коду у браузері та спрощувати взаємодію між frontend і backend. Це позитивно впливає на продуктивність, SEO і підтримку проєкту.

Таким чином, розробка сучасної e-commerce-платформи з використанням Next.js, React, TypeScript та headless CMS є актуальним завданням, оскільки дозволяє створити масштабовану, продуктивну та зручну систему, адаптовану до сучасних вимог веброзробки 2025–2026 років.

У межах даної кваліфікаційної роботи розроблено повностековий вебзастосунок SwiftMod, який реалізує сучасний e-commerce-стек відповідно до перелічених тенденцій. Проєкт побудований на React 19 з Next.js 16 App Router, повністю типізований через TypeScript 5.9, використовує Sanity 5 як headless CMS для керування товарами, замовленнями, користувачами та контентом, інтегрує Clerk як зовнішнього провайдера автентифікації з підтримкою OAuth, мобільних номерів та електронної пошти, реалізує оплату через Stripe з підтримкою прямого карткового платежу, містить інтегровану аналітику Firebase, а також транзакційну пошту через Nodemailer з OAuth2-автентифікацією Gmail. Особливістю архітектури є мультирольовий доступ - три окремі групи маршрутів для клієнтів, працівників та адміністраторів із відповідним розмежуванням прав.

Об'єктом дослідження є процес побудови повностекових вебзастосунків електронної комерції з підтримкою декількох ролей користувачів та інтеграції зовнішніх SaaS-сервісів (headless CMS, провайдер автентифікації, платіжний шлюз).

Предметом дослідження є архітектура та програмна реалізація вебзастосунку SwiftMod, що забезпечує повний цикл взаємодії «покупець –

працівник – адміністратор» на базі сучасного React-стеку та headless-архітектури.

Метою роботи є підвищення ефективності функціонування вебплатформ електронної комерції шляхом створення повностекового застосунку з мультирольовою архітектурою, інтеграцією headless CMS, платіжного шлюзу, системи лояльності та робочих процесів виконання замовлень у єдиному кодовому середовищі.

Для досягнення поставленої мети у роботі вирішено наступні задачі:

– проведено аналіз сучасних e-commerce-платформ та систем керування контентом, виявлено їх сильні й слабкі сторони;

– визначено цільову аудиторію та сформульовано функціональні й нефункціональні вимоги до системи;

– обґрунтовано вибір технологічного стеку для повностекової реалізації;

– спроєктовано мультирольову архітектуру з трьома окремими групами маршрутів (client, employee, admin) та єдиною шаром Sanity-схем;

– розроблено контентну модель Sanity, що покриває товари, категорії, бренди, замовлення, користувачів, відгуки, банери, блог, заявки на акаунти, адреси, передплати, контакти та сповіщення;

– реалізовано серверну частину Next.js, API-ендпоінтів через App Router та файлів серверних дій для прямого виклику з клієнтських компонентів;

– реалізовано клієнтську частину у вигляді понад 178 React-компонентів, згрупованих за функціональними доменами (cart, checkout, product, profile, admin, employee, wishlist, shopPage);

– інтегровано Clerk для автентифікації, Stripe для платежів, Nodemailer для транзакційних повідомлень, Firebase для аналітики;

– реалізовано систему лояльності з нарахуванням бонусних балів та електронного гаманця, а також робочого процесу виконання замовлень (склад – пакування – доставка – виплати);

– забезпечено інтернаціоналізацію через middleware-маршрутизацію за префіксом мови, динамічні словники та локальний матчер

– реалізовано SEO-оптимізацію (sitemap, robots, Open Graph), та адаптивний дизайн;

– проведено тестування застосунку: статичний аналіз через ESLint, типобезпеку TypeScript, ручне функціональне та сценарне тестування;

– підготовлено демонстраційні матеріали та оформлено результати дослідження.

Методи дослідження. У роботі використано методи системного аналізу для дослідження предметної області, методи об'єктно-орієнтованого та функціонального програмування для побудови клієнтських компонентів, патерни проєктування “Composition” (для побудови UI), «Provider» (для контекстів React), «Adapter» (для інтеграції зовнішніх SaaS-сервісів через спеціалізовані сервісні модулі), методи реляційного й документного моделювання даних (для побудови контентної моделі Sanity), а також методи ручного функціонального та сценарного тестування для верифікації коректності реалізації.

Наукова новизна роботи полягає у розробці уніфікованого підходу до побудови повностекових e-commerce-застосунків на базі Next.js App Router та React Server Components без необхідності побудови окремого бекенд-сервісу: серверна логіка, інтеграція з headless CMS та зовнішніми сервісами реалізовані у вигляді серверних дій та API-маршрутів у єдиній кодовій базі. Окремою новизною є запропонована трирівнева мультирольова структура застосунку (клієнт – працівник – адміністратор) з виокремленням робочого процесу виконання замовлень у окрему доменну зону (employee), що відображає реальну організаційну структуру e-commerce-операцій.

Практична значущість роботи полягає у створенні готового до промислового впровадження продукту, який скорочує час та витрати на побудову інтернет-магазину для малого та середнього бізнесу, надає повний цикл взаємодії від покупки до доставки, передбачає систему лояльності для утримання клієнтів та може бути розгорнутий у serverless-режимі на платформі Vercel однією командою.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

1.1 Електронна комерція як галузь цифрової економіки

Електронна комерція (e-commerce) є однією з найдинамічніших галузей сучасної цифрової економіки. Під цим терміном розуміють комплекс процесів купівлі-продажу товарів та послуг через електронні мережі, переважно - Інтернет. Історично електронна комерція пройшла декілька принципових етапів: від перших каталогів кінця 1990-х років з простими формами замовлення до сучасних платформ, що інтегрую у себе функції соціальних мереж, систем рекомендацій на основі штучного інтелекту, віртуальної реальності для перегляду товарів та автоматичних логістичних модулів.

У цифровій таксономії виділяють кілька основних моделей електронної комерції за типом учасників. Найпоширенішою є модель B2C (Business to Consumer), у якій бізнес продає товари або послуги безпосередньо кінцевим споживачам. Саме до цієї моделі належать класичні інтернет-магазини, а також маркетплейси на кшталт Amazon, Rozetka або Prom.ua. Модель B2B (Business to Business) передбачає продаж між юридичними особами та має свої особливості (оптові замовлення, договори, відстрочення платежу). Модель C2C (Consumer to Consumer) реалізована у вигляді платформ типу eBay або OLX. Окремо виділяють модель D2C (Direct to Consumer), де виробник продає товар прямо споживачу без посередників.

Вебзастосунок, розроблений у межах даної роботи, належить до моделі B2C. Він орієнтований на типовий онлайн-магазин роздрібної торгівлі, який обслуговує індивідуальних покупців та одночасно має внутрішню організаційну структуру з працівниками різних ролей (склад, пакування, доставка, фінансові операції) та адміністратором, що керує системою. Особливістю проєкту є те, що

ця внутрішня структура не виокремлена в окремий ERP-застосунок, а інтегрована у ту саму платформу - це характерно для проєктів малого та середнього бізнесу, де відсутні ресурси на побудову повного ERP-стеку.

Аналіз ринкових тенденцій останніх років виявляє декілька ключових напрямів розвитку e-commerce-індустрії. По-перше, спостерігається стрімке зростання частки мобільних покупок: за даними Statista [1], у 2024 році понад 60% всіх онлайн-замовлень здійснювалися з мобільних пристроїв, що зумовлює пріоритет адаптивного дизайну. Загалом, глобальні прогнози розвитку ринку електронної комерції підтверджують стабільну тенденцію до збільшення обсягів цифрових продажів у всьому світі [2]. По-друге, посилюються вимоги до швидкості завантаження сторінок: дослідження Google показують, що при затримці завантаження понад 3 секунди втрачається до 53% мобільних відвідувачів. По-третє, зростає очікування персоналізації досвіду: користувачі бажають бачити рекомендації товарів, історію замовлень, особисті бонуси, історію переглядів. По-четверте, посилюються вимоги до безпеки: PCI DSS для оплат картками, GDPR для роботи з персональними даними, мультифакторна автентифікація. По-п'яте, важливим стає підтримка декількох каналів продажу (omnichannel): веб-сайт, мобільний застосунок, месенджери, соціальні мережі - все з єдиним каталогом та замовленнями.

Усі перелічені тренди свідчать на користь архітектурного підходу, при якому презентаційний шар чітко відокремлений від шару зберігання та логіки. Це дозволяє паралельно розробляти декілька каналів представлення (web, мобільний застосунок) поверх єдиного джерела даних, оптимізувати швидкість завантаження за допомогою серверного рендерингу та CDN-кешування, гнучко інтегрувати персоналізаційні модулі. Саме на цій ідеї побудована концепція headless-CMS та повностекових фреймворків нового покоління, які обрані для реалізації проєкту SwiftMod.

Окрема увага у сучасних e-commerce-системах приділяється підтримці програм лояльності. Дослідження McKinsey показують, що залучення нового клієнта обходиться у 5–7 разів дорожче за утримання існуючого, тому інвестиції

у механізми «повторного продажу» (бонусні бали, кешбек, статусні рівні, персональні знижки) дають значно вищу рентабельність, ніж рекламні кампанії на нову аудиторію. У застосунку SwiftMod реалізована проста, але ефективна система лояльності з двома типами балів - винагородженнями (за обсяг покупок) та бонусами лояльності (за кількість виконаних замовлень).

Таким чином, побудова сучасної e-commerce-платформи - це багатовимірна задача, що включає в себе питання архітектури, продуктивності, безпеки, користувацького досвіду, інтеграції з зовнішніми сервісами та програм утримання клієнтів. Розробка такого застосунку є практично затребуваною й актуальною темою для бакалаврської роботи у галузі інженерії програмного забезпечення.

1.2 Мультирольова архітектура у системах електронної комерції

Концепція мультирольовості (multi-tenancy у частині користувацьких ролей) є однією з фундаментальних особливостей корпоративних e-commerce-систем. У класичній моделі онлайн-магазину виділяють три основні типи учасників: покупця (кінцевого користувача), адміністратора (управляє системою) та працівника (виконує оперативні дії з замовленнями). Кожна з цих ролей має власні потреби, інтерфейси та права доступу.

Покупець взаємодіє з системою у режимі «вітрини»: переглядає каталог товарів, фільтрує та шукає за параметрами, додає товари до кошика, оформлює замовлення, оплачує його, стежить за статусом доставки, залишає відгуки. Для покупця критично важливі швидкість роботи інтерфейсу, інтуїтивна навігація, наявність повної інформації про товари та можливість швидкого здійснення транзакції [3].

Адміністратор має повний доступ до системи. Він керує користувачами (затвердження бізнес-акаунтів, блокування порушників), товарами (хоча редагування контенту переважно делегується через Sanity Studio), замовленнями (зміна статусів, обробка повернень, відкат платежів), відгуками (модерація),

сповіщеннями (масові розсилки покупцям), аналітикою (дашборди з ключовими метриками), фінансовими операціями (підтвердження виплат працівникам). Для адміністратора важлива повнота функціональності та зручність роботи з табличними формами великих обсягів даних.

Працівник є проміжною роллю, що з'являється у середніх та великих e-commerce-проєктах. Він не керує системою в цілому, але виконує оперативні дії з конкретними замовленнями: приймає товари на склад, пакує замовлення, передає кур'єру, реєструє виплату. У застосунку SwiftMod виділено чотири підролі працівника, що відповідають реальним функціональним зонам e-commerce-операцій: склад (warehouse), пакування (packing), доставка (delivery), розрахунки (accounts/payments). Для працівника важливі швидкий доступ до списку доручених замовлень, форми зміни статусу та можливість залишати службові примітки.

Реалізація мультирольової архітектури на технічному рівні передбачає вирішення декількох ключових питань. По-перше, розмежування маршрутів: різні ролі мають доступ до різних URL-сегментів. У Next.js це елегантно реалізується через групи маршрутів App Router (group routes), позначені дужками: (client), (employee), (admin). Кожна група має свій layout-файл, що визначає верхній рівень оформлення (навігація, бічна панель), а файл middleware перевіряє наявність відповідних прав та виконує редирект у разі їх відсутності.

По-друге, перевірка прав на рівні серверних дій та API-ендпоінтів: навіть якщо середовище клієнта порушено (наприклад, через зміну URL вручну), сервер повинен незалежно перевіряти, чи має поточний користувач право на конкретну операцію. У SwiftMod це реалізовано через Clerk-сесії та перевірку ролі користувача за його ідентифікатором у Sanity-базі. Кожен серверний action перевіряє роль перед виконанням мутації.

По-третє, окремий UI-стек для кожної ролі: інтерфейс адміністратора суттєво відрізняється від клієнтського як візуально, так і за призначенням. У SwiftMod кожна з трьох ролей має власний набір компонентів у директоріях

components/admin/, components/employee/, components/cart/ та інших, що покривають клієнтський функціонал.

По-четверте, окрема контентна модель прав: ролі та права не повинні бути «жорстко зашиті» у код, а керуватися декларативно через дані. У SwiftMod це реалізовано через поле userType у схемі User в Sanity, яке може приймати значення customer, business, employee, admin. Зміна ролі користувача відбувається через адміністративні дії і не потребує перезавантаження сервера.

1.3 Цільова аудиторія

Під час проєктування вебзастосунку було визначено кілька основних груп цільової аудиторії, кожна з яких має специфічні потреби та сценарії використання [4]:

– Власники малого та середнього бізнесу - потребують швидко розгортуваного e-commerce-рішення з мінімальними інвестиціями у розробку та підтримку; для них критичні низькі поточні витрати (можливість роботи на безкоштовних або фрі-тірних планах Vercel, Sanity, Clerk), легкість додавання нових товарів і категорій, готові інтеграції з популярними платіжними системами;

– Розробники-початківці та студенти - використовують застосунок як референс-проєкт для вивчення сучасного React/Next.js-стеку, headless CMS та інтеграції з зовнішніми сервісами; для них важливі читабельність коду, грамотна структуризація проєкту, наявність прикладів усіх ключових концепцій (server components, server actions, middleware);

– Контент-менеджери та маркетологи - взаємодіють з системою переважно через Sanity Studio: додають товари, оновлюють описи, керують банерами та категоріями, публікують блог-записи; для них важливі зручний редактор контенту, попередній перегляд змін перед публікацією, можливість роботи з різними мовами;

– Адміністратори маркетплейсу - використовують застосунок для управління інтернет-магазином у щоденному режимі: затверджують заявки бізнес-акаунтів, обробляють повернення, модерують відгуки, контролюють фінансові потоки; вони цінують повноту функціоналу та логічну організацію адмін-панелі;

– Працівники складу та логістики - мають справу з оперативною роботою з замовленнями; для них важливі швидкий доступ до списків замовлень, простота зміни статусу, мобільна адаптивність (більшість працівників користуються мобільними пристроями на робочому місці);

– Кінцеві покупці - типові B2C-користувачі, що шукають товари, оформлюють замовлення, відстежують доставку; для них пріоритетні швидкість завантаження сторінок, інтуїтивна навігація, надійність оплати, прозорість процесу доставки;

– Постійні клієнти та учасники програм лояльності - користуються бонусними балами, відстежують свій статус, очікують персоналізованих знижок та сповіщень; для них важлива функціональність електронного гаманця, прозорість нарахування винагороджень, історія балів.

Визначення цільової аудиторії дозволяє сформувати наступні вимоги до інтерфейсу: мобільна-перша адаптивність (більшість покупок здійснюється з телефонів), мультимовна підтримка (українська, англійська, потенційно інші мови), темна та світла теми, висока швидкість завантаження (TTFB < 200 мс, LCP < 2,5 с), доступність (WCAG 2.1 AA), інтуїтивна навігація без потреби у документації.

1.4 Дослідження існуючих аналогів

Для обґрунтованого проєктування системи проведено аналіз існуючих рішень у галузі електронної комерції. Дослідження дозволяє виявити сильні та слабкі сторони наявних інструментів, визначити функції, які користувачі очікують від подібних застосунків, та сформувати позиціонування власного

продукту. Розглянуто п'ять найвпливовіших рішень: Shopify, WooCommerce, Magento (Adobe Commerce), BigCommerce та український маркетплейс Rozetka.ua.

1.4.1 Shopify

Shopify є найвідомішою у світі SaaS-платформою для побудови інтернет-магазинів. Заснована у 2006 році канадським підприємцем Тобіасом Лютке, нині вона обслуговує понад 4 мільйони магазинів у 175 країнах з річним обігом понад 200 мільярдів доларів через свою платформу. Shopify орієнтований переважно на малий та середній бізнес: підприємець може створити магазин буквально за лічені години, обираючи готовий шаблон, додаючи товари через адміністративну панель та налаштовуючи доставку й оплату.

Сильні сторони Shopify включають: швидкий старт без потреби у програмуванні, велику бібліотеку готових тем (понад 100 безкоштовних та платних шаблонів), розвинений App Store з тисячами розширень, інтегровану платіжну систему Shopify Payments з кешбеком, готові інтеграції з основними CRM-, ERP- та маркетинг-платформами, високий рівень безпеки (PCI DSS Level 1), надійну інфраструктуру з резервуванням. Платформа підтримує мультимагазинність, B2B-функціонал у тарифі Plus, омніканальну торгівлю через інтеграцію з Instagram, TikTok, Amazon.

Обмеженнями Shopify є його закрита природа SaaS: користувач не отримує доступу до серверного коду, не може повністю кастомізувати бізнес-логіку, обмежений жорстко визначеним API та можливостями шаблонної мови Liquid. Помісячна оплата (від $39 для базового тарифу до $2000+ для Plus) є відчутною для малого бізнесу. Комісія за використання альтернативних платіжних шлюзів (не Shopify Payments) додатково навантажує оператора. У випадку міграції на іншу платформу значна частина адаптивних рішень не може бути перенесена.

Висновок: Shopify є потужним рішенням для швидкого запуску онлайн-магазину, проте обмежений у кастомізації та має значні поточні витрати. Для

проєкту, що передбачає глибоку інтеграцію з власними процесами та повний контроль над кодом, він не є оптимальним вибором.

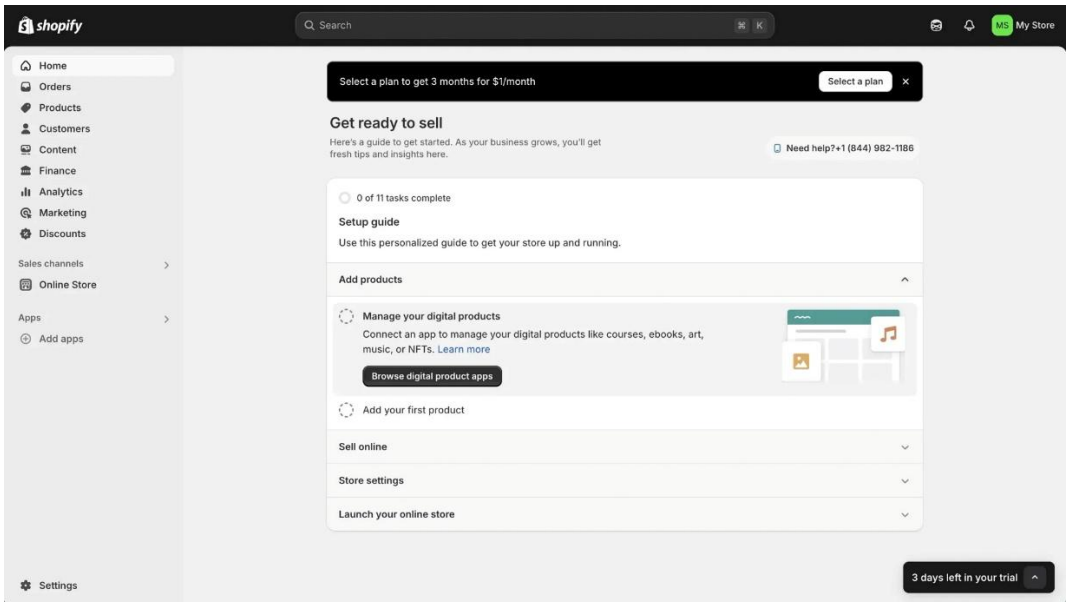


Рис. 1.1 Адміністративна панель Shopify

1.4.2 WooCommerce

WooCommerce - найпопулярніший у світі open-source e-commerce-плагін для WordPress, розроблений компанією Automattic. За даними BuiltWith, WooCommerce обслуговує понад 6 мільйонів активних вебсайтів, що становить близько 25% всіх онлайн-магазинів у мережі. Платформа повністю безкоштовна у базовій версії, що робить її першочерговим вибором для багатьох малих бізнесів.

Сильні сторони WooCommerce: безкоштовність та відкритий код, повна свобода у виборі дизайну та хостингу, гігантська екосистема плагінів (понад 50 тисяч у WordPress Plugin Directory та комерційних маркетплейсах), активна спільнота розробників, легкість додавання товарів через знайомий WordPress-інтерфейс. WooCommerce підтримує всі основні платіжні системи через плагіни (Stripe, PayPal, локальні шлюзи), має модулі для розрахунку доставки, програм лояльності, мультимовності.

Обмеженнями WooCommerce є його базова технологічна основа - PHP та jQuery. Хоча сам плагін постійно оновлюється, він успадковує всі проблеми WordPress: значне споживання серверних ресурсів, проблеми з продуктивністю при великій кількості товарів, складність кешування динамічних запитів, велика поверхня атаки через велику кількість плагінів. Сучасні підходи до фронтенду (React, Vue) важко інтегруються з традиційним темплейтингом WordPress. Якість плагінів зі сторонніх джерел дуже коливається, що часто призводить до проблем з безпекою.

Висновок: WooCommerce є зручним рішенням для команд, що мають досвід роботи з WordPress, проте не підходить для проєктів, орієнтованих на сучасний React/TypeScript-стек.

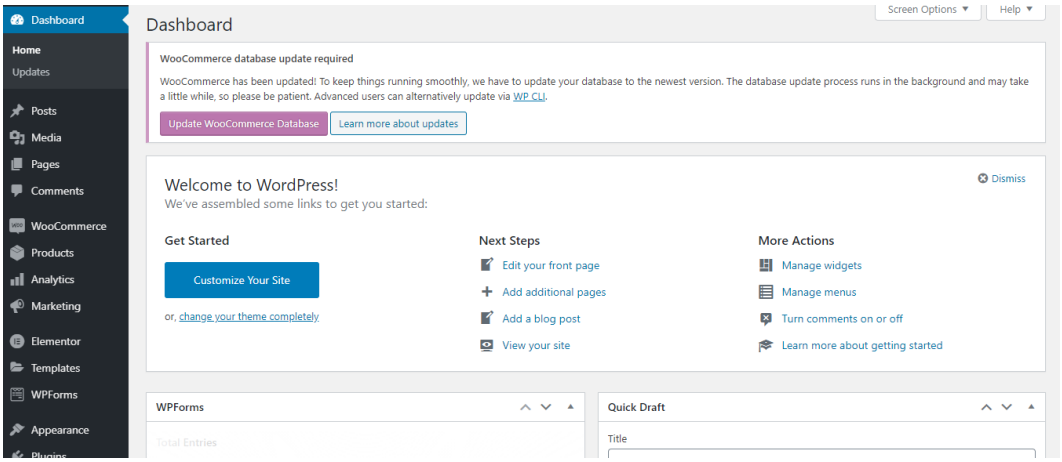


Рис. 1.2 Адміністративна панель WooCommerce у WordPress

1.4.3 Magento (Adobe Commerce)

Magento - потужна enterprise-рівнева e-commerce-платформа, що з 2018 року належить компанії Adobe (під брендом Adobe Commerce). Це найбільш функціонально багате рішення серед розглянутих: воно підтримує мультимагазини, мультискладність, B2B-функціонал, складні цінові правила, динамічні промоції, гнучку систему атрибутів товарів. Magento використовується великими компаніями: Coca-Cola, Olympus, Ford, Nestlé та іншими.

Сильні сторони Magento: повна enterprise-функціональність із коробки, гнучка модульна архітектура (можливо інтегрувати майже все), потужний продуктовий каталог з типами товарів (простий, конфігурований, груповий, віртуальний, завантажуваний), розвинений B2B-функціонал (компанії, ієрархія, корпоративні рахунки), мультимовність, мультивалютність.

Обмеженнями Magento є його високий поріг входу: для розробки потрібні дорогі сертифіковані спеціалісти, документація складна для початківців, час до запуску магазину перевищує кілька місяців. Magento споживає значні серверні ресурси: типовий магазин потребує не менше 8 ГБ оперативної пам'яті та виділеного сервера баз даних. Adobe Commerce у комерційній версії коштує від $22 000 на рік, що є непосильним для малого бізнесу. Технологічний стек Magento базується на PHP та власній архітектурі, що ускладнює інтеграцію з сучасними фронтенд-фреймворками.

Висновок: Magento є оптимальним вибором для великих enterprise-проєктів з бюджетом від кількох сотень тисяч доларів, проте є переускладненим та коштовним для типового онлайн-магазину малого або середнього бізнесу.

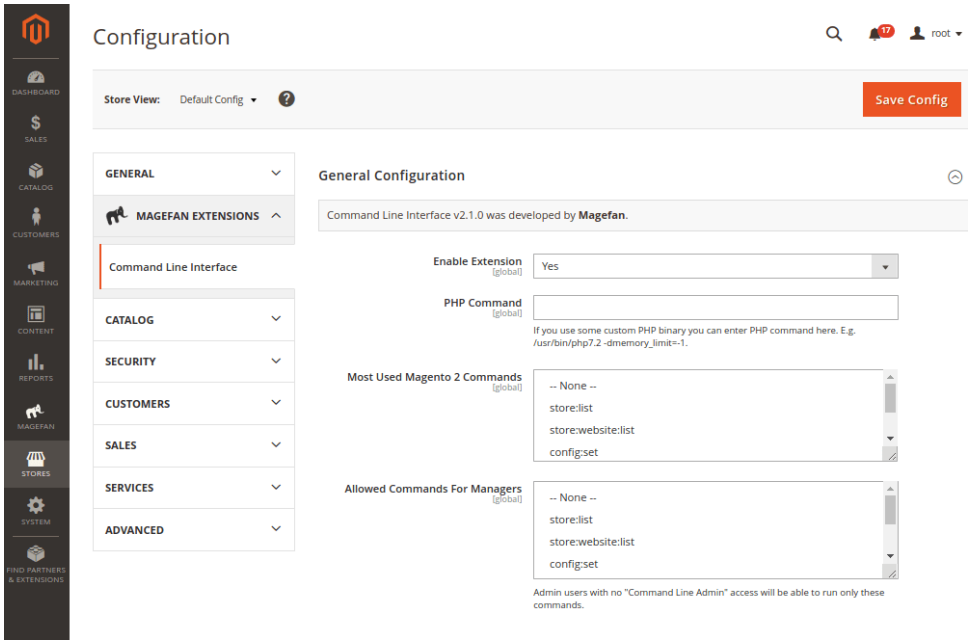


Рис. 1.3 Інтерфейс Magento Adobe Commerce

1.4.4 BigCommerce

BigCommerce - SaaS-платформа, що позиціонує себе як «гібрид» між Shopify та Magento: вона зберігає простоту хмарного рішення, проте надає значно більше функціоналу «з коробки» та має менш обмежувальну API-політику. Заснована у 2009 році, BigCommerce обслуговує понад 60 тисяч магазинів, у тому числі великі бренди Toyota, Sony, Skullcandy.

Сильні сторони BigCommerce: відсутність комісії за платежі (на відміну від Shopify), вбудована підтримка B2B та мультимагазинності у базових тарифах, готовий API для headless-розгортань (BigCommerce for WordPress, Stencil, GraphQL Storefront API), якісна документація. Платформа має менше готових тем порівняно з Shopify, проте надає більше можливостей для кастомізації власних шаблонів.

Обмеженнями BigCommerce є наявність «лімітів продажу» у тарифах: при перевищенні певного обігу платформа автоматично переводить магазин на дорожчий тариф. Це створює неприємні сюрпризи для бізнесів, що зростають. Темплейтна мова Stencil менш гнучка, ніж сучасні React-фреймворки. Як і у випадку Shopify, користувач не має доступу до серверного коду.

Висновок: BigCommerce є якісним вибором для middle-market-сегменту, проте обмежений у глибокій кастомізації та має непрогнозовану цінову політику для зростаючих магазинів.

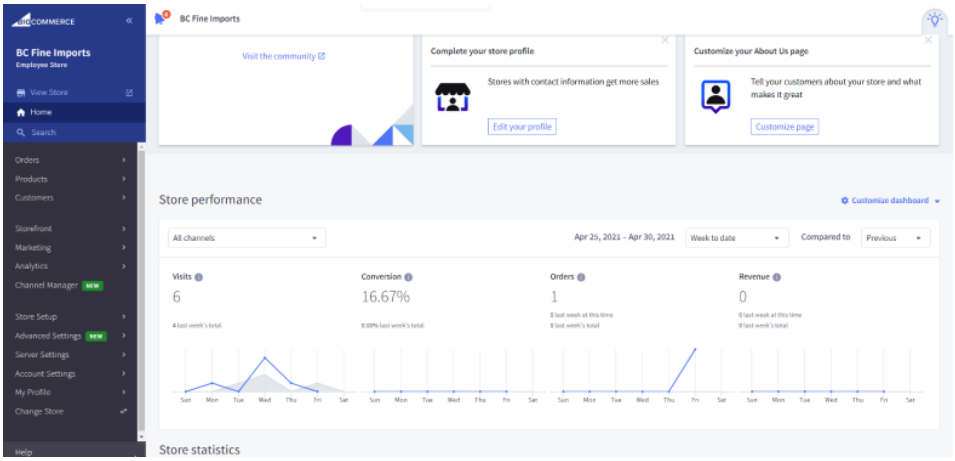


Рис. 1.4 Адміністративна панель BigCommerce

1.4.5 Висновки щодо аналогів

Зведені результати порівняння розглянутих аналогів за ключовими критеріями наведено у таблиці 1.1. Критерії підібрані таким чином, щоб охопити функціональність (підтримка мультирольовості, програм лояльності, headless-архітектури), технологічну сучасність (стек, готовність до React-розробки), доступність (open-source, вартість) та адаптацію до локального ринку.

Таблиця 1.1

Порівняльна характеристика існуючих аналогів e-commerce-платформ

Критерій	**Shopify**	**WooCommerce**	**Magento**	**BigCommerce**	**SwiftMod**
Сучасний стек	–	–	–	частково	+
Headless-архітектура	частково	частково	частково	+	+
Безкоштовний хостинг	–	–	–	–	+
Мультирольовість	+	+	+	+	+
Працівницький портал	–	–	частково	–	+
Програма лояльності	через плагін	через плагін	+	через плагін	+
Типізація (TS)	–	–	–	–	+
Server Components	–	–	–	–	+

Аналіз таблиці виявляє чітку нішу для розробленого застосунку: на ринку відсутнє безкоштовне open-source-рішення, побудоване на сучасному React/Next.js-стеку з повним TypeScript-покриттям, що поєднує мультирольову архітектуру (включно з працівницьким порталом), готову програму лояльності, headless-архітектуру через Sanity CMS та підтримку української локалізації. Shopify, WooCommerce та BigCommerce є комерційними рішеннями з власними обмеженнями; Magento переускладнений; Rozetka не є самостійною

платформою. SwiftMod заповнює цю нішу, орієнтуючись на малий та середній бізнес, що працює з командою розробки на стеку React/TypeScript.

Окремо варто підкреслити, що сучасний підхід Next.js 16 App Router з Server Components та Server Actions істотно змінює парадигму побудови e-commerce-застосунків. Якщо традиційні рішення вимагають побудови окремого backend-сервісу (REST API, GraphQL-сервер), що додає складності розгортання та збільшує час до запуску, то SwiftMod реалізує всю серверну логіку у вигляді серверних компонентів та actions у тій самій кодовій базі, що й фронтенд. Це дає не лише економію часу розробки, а й значно кращу type-safety: типи DTO автоматично узгоджені між клієнтом і сервером.

1.5 Визначення вимог до застосунку

На основі аналізу предметної області, цільової аудиторії та конкурентного середовища сформульовано наступні функціональні вимоги до системи:

- реєстрація та автентифікація користувачів за допомогою електронної пошти, мобільного номера та OAuth-провайдерів (Google, Apple, Facebook) через сервіс Clerk;
- підтримка декількох ролей користувачів: покупець (customer), бізнес-акаунт (business), працівник (employee) з підролями (warehouse, packing, delivery, accounts), адміністратор (admin);
- перегляд каталогу товарів з пагінацією, пошуком за назвою, фільтрацією за категорією, брендом, ціною, наявністю, статусом «знижка»;
- сортування товарів за різними критеріями (популярність, новизна, ціна, рейтинг);
- перегляд детальної сторінки товару: галерея зображень, опис, характеристики, відгуки, рекомендовані товари;
- кошик з можливістю додавання/видалення товарів та зміни кількості, синхронізований з localStorage та профілем користувача;
- список бажань (wishlist) для відкладеного придбання;

- повний цикл оформлення замовлення: вибір адреси доставки, способу оплати, перегляд підсумкової інформації, підтвердження замовлення;
- обробка платежів через Stripe з підтримкою карткової оплати, прямого банківського переказу;
- система електронного гаманця: можливість часткової оплати замовлення за рахунок накопиченого балансу;
- програма лояльності: автоматичне нарахування бонусних балів за обсягом покупок та кількістю замовлень;
- історія замовлень з детальним відображенням статусів та таймлайну (pending > confirmed > packing > shipped > delivered);
- можливість скасування замовлення на ранніх етапах з автоматичним поверненням коштів;
- система відгуків з фотографіями та модерацією адміністратором;
- інтегрований блог з категоріями та системою тегів;
- адміністративна панель з 20+ модулями керування (товари, користувачі, замовлення, відгуки, сповіщення, аналітика, бізнес-акаунти);
- працівницький портал з чотирма функціональними зонами (склад, пакування, доставка, виплати);
- система запитів на бізнес-акаунти та преміум-статус з flow затвердження адміністратором;
- підписка на новинну розсилку (newsletter) з можливістю відписки;
- контактна форма для зворотного зв'язку;
- сповіщення у застосунку (notification bell) з категоріями (promo, order, system, marketing);
- транзакційні email-повідомлення (підтвердження замовлення, зміна статусу, скасування, нарахування бонусів) через Gmail OAuth2;
- інтернаціоналізація з підтримкою декількох мов через middleware-маршрутизацію;
- темна та світла теми з автоматичним перемиканням за системними налаштуваннями;

- адаптивний дизайн для мобільних, планшетних та десктопних пристроїв.

Нефункціональні вимоги:

- час першого змістовного відображення (FCP) - менше 1,5 с на 4G-з'єднанні;
- час найбільшого змістовного відображення (LCP) - менше 2,5 с [5];
- CLS (Cumulative Layout Shift) - менше 0,1 [6];
- усі захищені маршрути перевіряються через Clerk-middleware та повторно валідуються на серверних діях [7];
- типобезпека: повне покриття TypeScript у суворому режимі (strict: true);
- якість коду: автоматичний статичний аналіз через ESLint на кожному commit;
- SEO-оптимізація: динамічний sitemap.xml, robots.txt, Open Graph мета-теги, структуровані дані Schema.org [8];
- доступність: відповідність WCAG 2.1 рівня AA для основних користувацьких сценаріїв [9];
- мобільна-перша адаптивність: тестування у viewports від 320 до 1920 пікселів;
- serverless-готовність: розгортання на платформі Vercel без потреби у власних серверах;
- моніторинг: інтеграція з Firebase Analytics для відстеження поведінки користувачів та ключових подій;
- відмовостійкість: коректна обробка помилок з відображенням user-friendly повідомлень;
- локалізація: підготовлена структура для додавання нових мов без зміни коду компонентів.

Реалізація вищезазначених вимог дозволяє створити повноцінний промисловий продукт, що відповідає сучасним стандартам якості та безпеки. У наступних розділах розглянуто вибір технологічного стеку та детально описано програмну реалізацію.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

Для реалізації вебзастосунку електронної комерції було проведено аналіз сучасних інструментів та обрано стек технологій, що забезпечує оптимальний баланс між продуктивністю розробки, надійністю у промисловій експлуатації, доступністю на ринку фахівців і відповідністю освітнім цілям дипломної роботи. У цьому розділі обґрунтовано вибір кожного компонента стеку.

2.1 Середовище розробки

Як основне середовище розробки обрано Visual Studio Code (VS Code) - кросплатформний редактор від компанії Microsoft, що набув статусу де-факто стандарту у web-розробці. VS Code є безкоштовним та має відкритий код, працює на macOS, Windows та Linux, підтримує понад 50 мов програмування з якісними мовними сервісами через протокол LSP (Language Server Protocol).

Серед альтернатив, що розглядалися, були WebStorm від JetBrains та Sublime Text. WebStorm пропонує більш глибоку інтеграцію з JavaScript/TypeScript, потужний рефакторинг та вбудовані інструменти аналізу, але є комерційним продуктом з ліцензією від $59 на рік для індивідуальних користувачів. Sublime Text - швидкий та легкий редактор, але вимагає значної конфігурації для отримання функціоналу, який у VS Code доступний з коробки.

Ключові розширення VS Code, що використовувалися у проєкті:

– TypeScript та JavaScript Language Features - вбудований мовний сервіс TypeScript;

– ESLint - статичний аналіз JavaScript/TypeScript коду з підсвічуванням проблем у редакторі;

– Prettier - автоматичне форматування коду на збереження;

– Tailwind CSS IntelliSense - підказки утилітарних класів та переглядач кольорів;

– Sanity Studio - підсвічування синтаксису для схем Sanity;

– GitLens - розширені можливості Git (blame, історія, порівняння);

– REST Client - тестування HTTP-запитів безпосередньо у редакторі;

– Error Lens - підкреслення помилок прямо у коді;

– Stripe - підказки для роботи з Stripe API;

– Clerk Code Snippets - фрагменти коду для роботи з Clerk;

– Auto Rename Tag та Auto Close Tag - автоматичне закриття HTML/JSX тегів.

VS Code також має вбудований термінал, що дає змогу одночасно редагувати код та запускати команди (npm, git, vercel) без перемикання вікон. Інтеграція з системою контролю версій Git реалізована як вбудована функція без необхідності встановлення додаткових інструментів. Для роботи з Sanity-схемами використовується вбудована Sanity Studio, що запускається через npm-команду та доступна за локальним URL.

2.2 Мова програмування TypeScript

Як основну мову програмування для всіх частин застосунку (клієнт, сервер, серверні дії, конфігурація) обрано TypeScript - статично типізовану надбудову над JavaScript, розроблену компанією Microsoft. TypeScript компілюється у звичайний JavaScript, що дає змогу використовувати всю екосистему JS-бібліотек, але одночасно надає переваги статичної типізації: виявлення помилок під час компіляції, автодоповнення в IDE, безпечний рефакторинг, само-документований код [10].

Як альтернативи TypeScript розглядалися JavaScript, Python, Go, Rust і PHP. JavaScript (ES2024) простіший у використанні, але не забезпечує типобезпеку, що є критичним для складних e-commerce систем. Python з Django або FastAPI розділяє frontend і backend на різні мови, ускладнюючи спільні DTO та інтеграцію з React Server Components. Go (Gin, Fiber) дає вищу продуктивність,

але потребує окремої інфраструктури та додаткового клієнтського шару. Rust (Actix, Axum) забезпечує максимальну швидкодію, однак має високий поріг входу і надмірну складність для навчального проєкту. PHP з Laravel традиційно використовується в e-commerce, але поступається сучасним JS/TS-стекам і гірше інтегрується з React-екосистемою.

Вибір TypeScript зумовлений єдиною мовою для клієнта і сервера, що дозволяє повторно використовувати типи, схеми валідації та утиліти між різними частинами системи. Додатково він має зрілу екосистему інструментів, широке підтримання npm-бібліотек із типами, активну спільноту та високу затребуваність на ринку.

У проєкті використовується TypeScript 5.9 у строгому режимі з повною типобезпекою. Налаштування включають сучасну ціль компіляції ES2024, ESM-модулі через bundler, JSX-підтримку для Next.js та alias-імпорти для спрощення структури проєкту. Інтеграція з Next.js здійснюється через офіційний plugin. Додатково використовується автоматично згенерований файл sanity.types.ts, який синхронізує типи з Sanity-схемами та забезпечує їх автоматичну перевірку компілятором під час змін у структурі даних. 2.3 Frontend та fullstack: Next.js 16 та React 19

Як основний framework для побудови застосунку обрано Next.js 16 - повністю TypeScript-готовий повностековий framework на базі React, розроблений компанією Vercel. Next.js не є «просто» frontend-фреймворком: це повноцінне рішення, яке поєднує клієнтську частину з серверним рендерингом, маршрутизацію, серверні API-ендпоінти, серверні дії, оптимізацію зображень, інтернаціоналізацією та іншими функціями в одному пакеті.

Версія 16 приносить кілька ключових нововведень: повну підтримку React 19 з усіма його новими хуками, оптимізований Turbopack-bundler як заміна Webpack (на 700% швидший на старті dev-сервера), стабільні Server Actions, покращену роботу з кешуванням, нативну підтримку важких ESM-залежностей.

Архітектурно Next.js 16 побудований навколо App Router - нової системи маршрутизації, що використовує React Server Components як основний механізм.

Серед альтернатив Next.js розглядалися Remix, Nuxt.js, SvelteKit, Astro та Create React App. Remix має меншу екосистему та менш активний розвиток після інтеграції з Shopify. Nuxt.js базується на Vue, що ускладнює використання в React-проєктах. SvelteKit пропонує інший підхід без віртуального DOM, але ще не має достатньо зрілої екосистеми. Astro орієнтований переважно на статичний контент і блоги, тому слабше підходить для інтерактивних e-commerce систем. Create React App вважається застарілим і не підтримує сучасні SSR-підходи.

Next.js 16 використовується як основний фреймворк завдяки сучасній архітектурі [11]. App Router забезпечує файлову маршрутизацію з підтримкою layout, динамічних і вкладених маршрутів. React Server Components дозволяють виконувати рендеринг на сервері без надсилання логіки в браузер і дають прямий доступ до серверних ресурсів. Server Actions дозволяють виконувати серверні операції без окремого API-шару. Route Handlers реалізують REST-ендпоінти для інтеграцій, включно з вебхуками. Middleware використовується для перевірки авторизації та обробки локалізації. Вбудована оптимізація зображень автоматично генерує сучасні формати та адаптивні версії. Також підтримується інтернаціоналізація через маршрутизацію і middleware.

React 19 є основою UI-шару і реалізує компонентний підхід до побудови інтерфейсу [12]. У версії 19 додано Server Actions, нові хуки для роботи з формами та асинхронними станами, покращену підтримку Suspense та оптимістичних оновлень, а також спрощено роботу з типізацією через TypeScript, замінивши застарілі підходи до props за замовчуванням [13].У проєкті SwiftMod більшість компонентів реалізовані як Server Components, що рендеряться на сервері та віддають готовий HTML клієнту. Лише компоненти, що потребують інтерактивності (форми, модальні вікна, кошик, кнопки додавання у обране), позначені директивою "use client" та виконуються у браузері. Це дає значну економію розміру JavaScript-бандла та покращує метрики Core Web Vitals.

2.3 Headless CMS: Sanity 5

Як систему керування контентом обрано Sanity Studio 5 - сучасну headless-CMS, побудовану на ідеях real-time-співпраці, структурованого контенту та повної програмованості схем [14]. Sanity на відміну від «класичних» CMS на кшталт WordPress або Drupal не нав'язує власного презентаційного шару: вона надає лише API для зберігання та запитів даних, тоді як презентація повністю реалізована у Next.js-застосунку.

Альтернативами Sanity виступали Contentful, Strapi, Storyblok, Directus, Payload CMS та пряме використання PostgreSQL через Prisma. Contentful є популярною SaaS CMS, але обмежує безкоштовний тариф і має менш гнучку модель даних. Strapi дає повний контроль, проте вимагає самостійного хостингу та підтримки інфраструктури. Storyblok більше орієнтований на маркетингові сайти з візуальним редактором і слабше підходить для складних e-commerce структур. Directus працює поверх SQL-бази, але додає значне навантаження на DevOps. Payload CMS є сучасним TypeScript-рішенням, однак ще не досяг рівня зрілості. Прямий доступ до PostgreSQL через Prisma забезпечує максимальну гнучкість, але потребує розробки власної адмін-панелі.

Sanity обрано через декларативні TypeScript-схеми з автоматичною валідацією та інтеграцією в редактор, GROQ-запити з підтримкою складної фільтрації та агрегацій, real-time спільне редагування документів, хмарний free-tier для невеликих проєктів, CDN для динамічної обробки зображень, офіційну інтеграцію з Next.js через next-sanity, а також можливість вбудованого Studio в структуру застосунку. Додатковою перевагою є автоматична генерація TypeScript-типів через typegen, що синхронізує схеми з кодовою базою.

У проєкті SwiftMod визначено 17 типів схем (sanity/schemaTypes/): product, category, brand, order, user, blog, blogCategory, review, address, banner, subscription, contact, sentNotification, userAccessRequest, author, blockContent та сервісні підтипи. Деталі схем наведено у розділі 3.2. Запити до Sanity виконуються через

GROQ-функції у директорії sanity/queries/, що групують типові запити (userQueries, emailUserQueries) [15].

2.4 Автентифікація: Clerk

Як зовнішній провайдер автентифікації обрано Clerk - спеціалізований SaaS-сервіс, що надає повний цикл управління користувачами: реєстрацію, вхід, відновлення паролю, верифікацію email/SMS, OAuth-входи (Google, Apple, Facebook, GitHub та інші), мультифакторну автентифікацію, керування організаціями та ролями [16]. Clerk звільняє розробника від необхідності писати власний шар автентифікації, який є типовим джерелом серйозних безпекових вразливостей.

Серед альтернатив Clerk розглядалися NextAuth.js (Auth.js), Auth0, Supabase Auth, Firebase Authentication та кастомна JWT-реалізація. NextAuth.js є найпопулярнішим open-source рішенням для Next.js, але вимагає самостійного побудування UI та логіки керування користувачами. Auth0 надає повнофункціональну платформу автентифікації, однак має високу вартість при масштабуванні. Supabase Auth добре працює в межах власної екосистеми Supabase, але менш універсальний поза нею. Firebase Authentication має швидку інтеграцію, проте обмежений у кастомізації UI та архітектурній гнучкості. Власна JWT + bcrypt реалізація дає повний контроль, але суттєво підвищує ризики безпеки та складність підтримки.

Clerk обрано завдяки глибокій інтеграції з Next.js через @clerk/nextjs і підтримці middleware, готовим UI-компонентам для auth-flow (<SignIn />, <SignUp />, <UserButton />), вбудованим сторінкам автентифікації, підтримці організацій та ролей, а також швидкому старту без необхідності розробки власних форм. Додатково використовується безкоштовний тариф для early-stage проєктів, широкий набір OAuth-провайдерів, passwordless-аутентифікація, відповідність стандартам безпеки (SOC 2, GDPR) і webhook-механізм для синхронізації користувачів із зовнішніми системами, включно з Sanity.

У SwiftMod Clerk інтегрований через ClerkProvider у app/[lang]/layout.tsx та middleware у файлі proxy.ts. Кожен користувач має ідентифікатор Clerk (clerkUserId), який зберігається у Sanity-документі User та використовується як зовнішній ключ. Webhook від Clerk при створенні нового користувача автоматично створює запис у Sanity з типом customer за замовчуванням.

2.5 Платіжна система: Stripe

У якості платіжного шлюзу використано Stripe — один із найпоширеніших сервісів онлайн-оплат із глобальною інфраструктурою та розвиненим API для інтеграції платежів, підписок, повернень і виплат [17]. Сервіс орієнтований на розробників, має якісну документацію, офіційні SDK і підтримку типізації.

Серед альтернатив розглядалися PayPal, LiqPay, WayForPay, Mollie та Square. PayPal поступається за зручністю API та комісіями, LiqPay і WayForPay більше орієнтовані на локальні ринки, Mollie має регіональні обмеження, а Square сфокусований переважно на офлайн-оплатах.

Вибір Stripe обґрунтовано високою якістю developer experience, TypeScript SDK, готовим Stripe Checkout, адаптивним Payment Element, webhook-механізмом для подій, підтримкою всіх основних платіжних систем і Apple Pay / Google Pay, а також наявністю тестового режиму та інструментів на кшталт Stripe Tax [18].

У SwiftMod Stripe інтегрований через клієнт у lib/stripe.ts. Серверна дія createCheckoutSession створює Stripe Checkout Session з товарами кошика та повертає URL для перенаправлення користувача. Webhook у app/api/webhook/route.ts обробляє події checkout.session.completed та оновлює статус замовлення у Sanity. Окремий компонент DirectPaymentModal реалізує безпосередню оплату через Stripe Payment Element для випадків, коли користувач не бажає переходити на Stripe-hosted сторінку.

2.6 Стилізація: Tailwind CSS 4 та Radix UI

Для стилізації клієнтської частини обрано Tailwind CSS 4 - utility-first CSS-фреймворк, що генерує CSS-класи на основі використання у JSX [19]. На відміну від традиційних CSS-фреймворків (Bootstrap, Bulma) з готовими компонентами, Tailwind надає низькорівневі утиліти (margin, padding, color, flex, grid), з яких розробник складає власні компоненти. Це дає максимальну гнучкість дизайну при збереженні швидкості розробки.

Версія 4, випущена у 2024 році, принесла декілька значних оптимізацій: новий compiler на базі Lightning CSS (на 10x швидший за PostCSS-варіант), automatic content detection (більше не потрібно явно перераховувати content-paths), zero-config setup для більшості Next.js-проєктів, нову систему theme variables через CSS custom properties замість JS-конфігурації.

Альтернативи, що розглядалися:

– Bootstrap 5 - традиційний фреймворк з готовими компонентами, простий, але менш гнучкий у дизайні;

– Material UI (MUI) - потужна бібліотека Material Design компонентів, але важка та має жорсткий стиль;

– Chakra UI - приємний DX, але менш зрілий та має менше готових компонентів;

– Ant Design - популярна для enterprise, але переважно для адмін-панелей;

– styled-components - CSS-in-JS, але має більший runtime та проблеми з SSR.

Tailwind CSS обрано через: максимальну гнучкість дизайну (можна реалізувати будь-який макет без overrides), мінімальний CSS-bundle (тільки використані класи потрапляють у фінальний CSS), вбудовану підтримку темної теми через клас dark:, intellisense у редакторах, інтеграцію з більшістю UI-бібліотек.

Як бібліотека готових компонентів використовуються Radix UI Primitives - unstyled accessible-компоненти, що надають лише функціональність (керування

фокусом, ARIA-атрибути, keyboard navigation) без власних стилів [20]. У проєкті використовуються 14 компонентів Radix UI: Accordion, Checkbox, Collapsible, Dialog, DropdownMenu, Label, Popover, RadioGroup, ScrollArea, Select, Separator, Slider, Slot, Switch, Tabs, Tooltip, VisuallyHidden. Стилізація цих компонентів виконана через Tailwind-класи у файлах components/ui/.

Окремо використовуються Headless UI (для додаткових компонентів) та Lucide React (564+ якісних SVG-іконок, оптимізованих для React). Для обробки CSS-класів використано clsx (умовне об'єднання класів) та tailwind-merge (резолвація конфліктів між класами).

2.7 Менеджер стану Zustand

Для керування глобальним станом клієнтської частини обрано Zustand - мінімалістичний state manager для React, що надає просте API на основі хуків без необхідності використання Provider-компонентів та редьюсерів [21]. Zustand підходить для випадків, коли потрібен глобальний стан (наприклад, кошик, обране, налаштування теми), але повноцінний Redux буде overkill.

Альтернативи, що розглядалися:

– Redux Toolkit - найпопулярніший state manager, але має значно більший boilerplate;

– Recoil - атомарний state manager від Meta, але припинений у 2024 році;

– Jotai - конкурент Recoil, атомарний, але менш популярний;

– XState - потужний для складних state machines, але overkill для типових сценаріїв;

– React Context - вбудований, але має проблеми з продуктивністю при глибокій вкладеності.

Рішення на користь Zustand: мінімальний API (близько 5 функцій), вбудована middleware-система (persist для localStorage, devtools для Redux DevTools, immer для імутабельних оновлень), SSR-сумісність з Next.js, відмінна типобезпека з TypeScript. У SwiftMod єдиний Zustand-store розташований у файлі

store.ts і містить: список товарів у кошику, обрані товари (wishlist), стан AuthSidebar (відкритий/закритий), стан розміщення замовлення. Стор персистується у localStorage під ключем cart-store через middleware persist.

2.8 Поштовий сервіс: Nodemailer

Для відправлення транзакційних email-повідомлень обрано Nodemailer - найпопулярніший Node.js-пакет для роботи з SMTP-серверами та поштовими сервісами [22]. Nodemailer підтримує усі основні поштові протоколи (SMTP, IMAP), має вбудовану підтримку OAuth2 для автентифікації Gmail, дозволяє відправляти HTML-листи з прикріпленими файлами, реалізує queue для надійної доставки.

Альтернативи, що розглядалися:

– SendGrid - корпоративний поштовий сервіс, надійний, але потребує підписки;

– Mailgun - також транзакційна пошта, але має комерційну модель;

– Postmark - спеціалізований сервіс транзакційної пошти, дорожчий за SendGrid;

– Resend - новий API-перший сервіс, оптимізований для розробників, безкоштовний до 3 000 листів на місяць;

– AWS SES - найдешевший за обсягом, але вимагає AWS-налаштування.

У SwiftMod на стартовій стадії обрано Nodemailer з Gmail OAuth2 як безкоштовне рішення для невеликих обсягів (до 500 листів на день). Файл lib/emailService.ts (720 рядків) містить імплементацію: створення OAuth2-клієнта Google, генерацію access-token, формування HTML-листів з шаблонів (підтвердження замовлення, зміна статусу, скасування, нарахування бонусів), відправлення з графіком retry при тимчасових помилках. Окремий файл lib/emailImageUtils.ts відповідає за оптимізацію зображень у email (стиснення, конвертацію у inline-base64).

2.9 Аналітика та допоміжні бібліотеки

Для аналітики поведінки користувачів використано Firebase Analytics - безкоштовний сервіс від Google, що дозволяє відстежувати page views, custom events, conversion funnels та інші маркетингові метрики [23]. Інтеграція реалізована у lib/firebase.ts та lib/analytics.ts (310 рядків).

Список основних допоміжних бібліотек, що використовуються у проєкті:

– date-fns 4 та dayjs 1.11 - робота з датами та часовими зонами;

– uuid 13 - генерація унікальних ідентифікаторів;

– lodash 4 - утилітарні функції для роботи з масивами та об'єктами;

– negotiator 1 та @formatjs/intl-localematcher - визначення мови користувача за заголовком Accept-Language;

– framer-motion 12 та motion 12 - анімації переходів між сторінками та модальних вікон;

– embla-carousel 8 - карусель для банерів та галерей товарів;

– recharts 3 - графіки для адміністративної аналітики;

– sonner 2 - toast-сповіщення з якісним UI;

– country-state-city 3 - довідник країн, регіонів та міст для адресних форм;

– class-variance-authority 0.7 та tailwind-merge 3 - типобезпечні CSS-класи;

– react-icons 5 - додаткові колекції іконок (Heroicons, Material Icons, Bootstrap Icons).

Усі бібліотеки додаються у package.json з зафіксованими версіями та регулярно оновлюються через автоматизовані PR від Dependabot. Загальний розмір production-bundle після оптимізації Next.js становить близько 250 КБ JavaScript (gzip), що відповідає сучасним стандартам e-commerce-сайтів.

3 РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ

3.1 Проєктування архітектури

Архітектура вебзастосунку SwiftMod побудована за принципами modern fullstack-розробки на базі Next.js App Router з повноцінним використанням React Server Components, Server Actions та зовнішніх SaaS-сервісів [24]. Загальна схема архітектури зображена на рисунку (див. рис. 3.1)

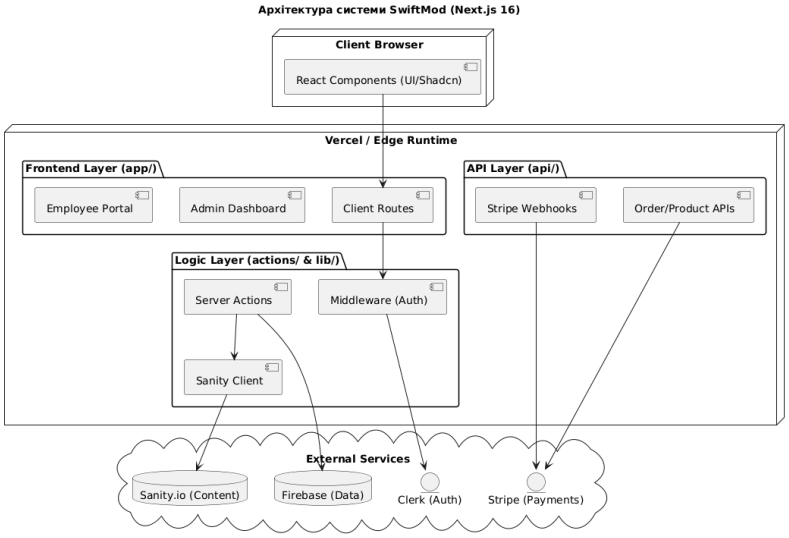


Рис. 3.1 Архітектура вебзастосунку SwiftMod

Архітектура складається з декількох логічних шарів. Перший шар - браузер клієнта (Browser), у якому виконуються клієнтські React-компоненти (помічені директивою "use client") [25]. Цей шар відповідає за інтерактивні елементи інтерфейсу: кошик, форми, модальні вікна, кнопки додавання у обране. Клієнтський JavaScript-код доставляється з Vercel CDN.

Другий шар - Vercel Edge Network, що включає в себе Next.js Middleware. Це функція, яка виконується перед кожним запитом та виконує дві ключові задачі: перевірку автентифікації Clerk (захист маршрутів /admin, /employee,

/checkout, /profile тощо) та маршрутизацію за мовою (визначення локалі за заголовком Accept-Language та редірект на /[lang]/...). Middleware виконується на edge-вузлах поблизу користувача, що мінімізує затримку.

Третій шар - Next.js Server, що виконується у serverless-функціях Vercel. Тут зосереджена основна логіка: рендеринг серверних компонентів, обробка серверних дій (Server Actions), обслуговування API-ендпоінтів (Route Handlers) [26]. Серверний шар має доступ до конфіденційних даних: API-ключі Sanity, Stripe, Clerk, Firebase, Nodemailer. Він взаємодіє з зовнішніми сервісами через спеціалізовані SDK.

Четвертий шар - зовнішні SaaS-сервіси та сховища даних:

– Sanity Studio та Sanity Content Lake - headless-CMS, що зберігає всі дані застосунку (товари, замовлення, користувачі тощо). Доступ здійснюється через @sanity/client з підтримкою як публічних read-запитів, так і авторизованих write-операцій з токеном;

– Clerk Authentication - провайдер автентифікації, що зберігає облікові записи користувачів та обробляє входи. Інтеграція через @clerk/nextjs;

– Stripe - платіжний шлюз. Доступ через офіційний stripe SDK для Node.js;

– Firebase - аналітика та real-time-сервіси. Доступ через firebase JavaScript SDK;

– Gmail SMTP (через OAuth2) - поштовий сервер для надсилання транзакційних листів через Nodemailer.

Окрему увагу варто приділити маршрутизації запитів. Для типового сценарію перегляду сторінки товару послідовність обробки виглядає так: клієнт надсилає GET-запит > Vercel Edge перевіряє middleware (мова, авторизація) > Next.js Server рендерить React Server Component для сторінки товару > Server Component виконує GROQ-запит до Sanity > отримує дані > перетворює у HTML > клієнт отримує готовий HTML з мінімальним JavaScript-bundle для гідрації інтерактивних елементів.

Для сценарію додавання товару у кошик: клієнтський компонент викликає Zustand-action > стан кошика оновлюється локально > одночасно викликається

серверна дія saveCartToProfile > дія перевіряє автентифікацію через auth() з Clerk > при наявності користувача оновлює його документ User у Sanity через writeClient. Це гарантує, що кошик синхронізований між девайсами користувача.

Така архітектура дає кілька переваг: чітке розділення між публічним read-доступом і захищеними write-операціями, відсутність потреби у власному backend-сервері (повна serverless-модель), легкість локального розгортання (для запуску потрібно лише npm install та npm run dev), масштабованість завдяки serverless (Vercel автоматично масштабує функції під навантаження), типобезпека на всіх рівнях завдяки TypeScript.

3.2 Контентна модель Sanity

Контентна модель застосунку реалізована у Sanity Content Lake у вигляді 17 типів схем, що описують усі сутності системи. Схеми визначені у директорії sanity/schemaTypes/ та зведені у файлі index.ts. Кожна схема - це декларативний об'єкт з полями певних типів (string, number, boolean, image, reference, array, object тощо), валідаціями та правилами відображення у Sanity Studio.

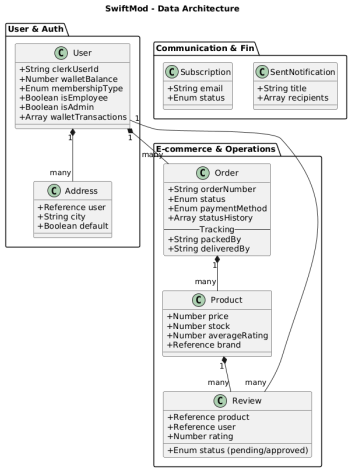


Рис. 3.3 Контентна модель SwiftMod

Основні типи схем зведені у таблиці 3.1.

Таблиця 3.1

Типи схем Sanity у застосунку SwiftMod

Тип	Призначення	Ключові поля
product	Картка товару	name, slug, price, discount, images, category, brand, description, stock, variants, status
category	Категорія товарів	name, slug, description, image, parentCategory, order
brand	Бренд	name, slug, logo, description
order	Замовлення	orderId, user, items[], status, paymentStatus, total, shippingAddress, timeline[]
User	Профіль користувача	clerkUserId, email, name, addresses, wallet, points, notifications, userType
review	Відгук на товар	product, user, rating, comment, photos, status, createdAt
blog	Стаття блогу	title, slug, content, author, category, publishedAt, coverImage
blogCategory	Категорія блогу	name, slug, description
author	Автор статті	name, image, bio, social[]
address	Адреса доставки	street, city, state, zipCode, country, phone, isDefault
banner	Банер на головній	image, title, link, active, displayOrder, dateRange
subscription	Підписка на розсилку	email, status, subscribedAt, source
contact	Заявка зворотного зв'язку	name, email, subject, message, submittedAt
sentNotification	Лог сповіщень	userId, title, message, type, read, sentAt, priority

Продовження таблиці 3.1

Типи схем Sanity у застосунку SwiftMod

userAccessRequest	Заявка на бізнес/преміум	userId, requestType, status, reason, requestedAt, decidedAt
blockContent	Структурований текст	portable text для описів та статей

Розглянемо детальніше ключові схеми. Схема product містить близько 20 полів, що описують товар з усіх боків. Основні поля: name (рядок з валідацією на унікальність), slug (генерується автоматично з name через slugify-плагін), price (число з валідацією > 0), discount (опційне число відсотків), images (масив зображень з обов'язковим alt-текстом), category (reference на категорію), brand (опційний reference на бренд), description (PortableText для багатого форматування), stock (число залишків), status (enum: active, draft, archived), specifications (масив пар ключ-значення), variants (масив варіантів для розміру, кольору тощо).

Схема order є найскладнішою у системі. Вона зберігає повну історію замовлення: items - масив з ID товарів, кількістю, ціною на момент покупки (snapshot, щоб не залежати від поточної ціни); user - reference на User-документ; status - поточний статус (pending > confirmed > processing > packing > shipped > delivered > completed або cancelled); paymentStatus - окреме поле (pending, paid, refunded); shippingAddress - embed-об'єкт з адресою на момент замовлення; total - фінальна сума з урахуванням знижок та податків; timeline - масив подій (зміна статусу, додавання примітки, виплата), кожна з timestamp та author.

Схема user (Sanity-варіант) синхронізується з обліковим записом Clerk через поле clerkUserId. Окрім базових даних (email, name, image) вона містить: addresses - масив адрес доставки; wallet - об'єкт з balance та transactions[]; points - об'єкт з rewardPoints та loyaltyPoints; notifications - масив сповіщень; userType -

enum (customer, business, employee, admin); employeeRole - опційний enum для працівників (warehouse, packing, delivery, accounts).

Зв'язки між сутностями реалізовано через тип reference: order > user, order > product (у items), review > user, review > product, banner > product (опційно), blog > author, blog > blogCategory. Усі references автоматично перевіряються Sanity на існування цільового документа, що запобігає dangling-references.

Для отримання даних з Sanity використовуються GROQ-запити, що сформовані у директорії sanity/queries/. Приклад запиту для отримання списку товарів з пагінацією та фільтрацією:

**[_type == "product" && status == "active" && (!defined($category) || category._ref == $category)] | order(_createdAt desc) [$offset...$offset+$limit] { _id, name, slug, price, discount, "image": images[0], "category": category->name }*

Цей запит читається таким чином: вибрати всі документи типу product, де статус активний і опційно фільтрувати за категорією; сортувати за датою створення спадаюче; обмежити пагінацією; повернути лише потрібні поля з підстановкою назви категорії через references.

3.3 Маршрутизація та інтернаціоналізація

Маршрутизація застосунку реалізована через Next.js App Router у директорії app/. Це файлова система маршрутизації, де структура директорій безпосередньо відображається у URL-структуру сайту. Файл page.tsx у директорії визначає сторінку, layout.tsx - обгортку для всіх дочірніх сторінок, route.ts - REST-ендпоінт.

Структура маршрутів SwiftMod виглядає так:

– /studio/[[...tool]]/page.tsx - Sanity Studio як адміністративна панель CMS, доступна за адресою /studio;

– /api/* - REST-ендпоінти для зовнішніх інтеграцій (Stripe webhook, експорт даних, аналітика);

– /[lang]/layout.tsx - кореневий layout з ClerkProvider, UserDataContext-Provider, Toaster для toast-сповіщень, шрифтів Poppins/Raleway/OpenSans;

– /[lang]/(client)/* - публічна частина магазину (головна, каталог, кошик, профіль, замовлення);

– /[lang]/(auth)/* - сторінки автентифікації (sign-in, sign-up) з Clerk-компонентами;

– /[lang]/(admin)/* - адміністративна панель з 18+ сторінками;

– /[lang]/(employee)/* - портал працівника з 4 функціональними зонами.

Параметр [lang] у квадратних дужках означає динамічний сегмент: значення мови підставляється з URL і доступне у компоненті через props.params.lang. Дужки (client), (auth), (admin), (employee) утворюють route groups - групи маршрутів, що дають змогу організувати файли логічно без впливу на URL-структуру (тобто URL для сторінок (client)/cart буде /[lang]/cart, а не /[lang]/(client)/cart).

Інтернаціоналізація реалізована через комбінацію декількох механізмів. По-перше, конфігураційний файл i18n-config.ts містить масив підтримуваних мов та мову за замовчуванням. На поточному етапі підтримується англійська мова, проте структура готова до додавання нових мов (українська, російська, польська тощо).

По-друге, middleware у файлі proxy.ts (Next.js називає його middleware.ts, але у проєкті використано історичну назву proxy.ts через тривалу історію розвитку) аналізує заголовок Accept-Language користувача, обирає найкращу збігаючу мову через @formatjs/intl-localematcher та виконує редірект на /[lang]/... якщо у URL відсутній префікс мови. Це гарантує, що користувач завжди потрапляє на правильно локалізовану сторінку.

По-третє, словники перекладів зберігаються у директорії dictionaries/ у форматі JSON-файлів (en.json, потенційно uk.json тощо). Функція getDictionary у lib/dictionary.ts динамічно імпортує потрібний словник для мови. Серверні компоненти приймають dict як prop та використовують його для відображення локалізованого тексту.

По-четверте, для роботи з датами використовується date-fns/locale або dayjs з локалями, що автоматично перемикаються на основі поточної мови.

3.4 Серверна частина: API та серверні дії

У застосунку SwiftMod реалізовано понад 50 API-ендпоінтів у директорії app/api/. Кожен ендпоінт - це файл route.ts, що експортує функції GET, POST, PUT, DELETE або PATCH відповідно до HTTP-методів. Ендпоінти згруповані за функціональними зонами:

– /api/orders/* - управління замовленнями (створення, отримання, скасування, повернення, надсилання email);

– /api/checkout/stripe - створення Stripe Checkout Session;

– /api/checkout/clerk - альтернативний flow оплати через прямий card-form;

– /api/webhook - обробник Stripe-webhook для асинхронних подій (checkout.session.completed, payment_intent.failed, charge.refunded);

– /api/admin/* - адміністративні ендпоінти (users, orders, products, reviews, notifications, analytics тощо);

– /api/user/* - користувацькі ендпоінти (profile, addresses, orders, notifications, points, reward-points, combined-data);

– /api/newsletter - підписка на розсилку;

– /api/contact - надсилання повідомлень з контактної форми;

– /api/analytics - кастомні аналітичні події для Firebase;

– /api/user-data - експорт даних користувача (GDPR-комплаєнс);

– /api/debug - діагностичні ендпоінти для розробки.

Зведений список ключових ендпоінтів представлено у таблиці 3.2.

Таблиця 3.2

Ключові REST API ендпоінти SwiftMod

Метод	Шлях	Призначення
POST	/api/checkout/stripe	Створення Stripe Checkout Session
POST	/api/webhook	Обробка Stripe webhook-подій
GET	/api/orders	Список замовлень користувача
GET	/api/orders/[orderId]	Деталі замовлення
POST	/api/orders/refund	Повернення коштів
POST	/api/orders/[orderId]/send-email	Надіслати email-сповіщення
GET	/api/orders/count	Кількість замовлень користувача
GET	/api/user/profile	Профіль користувача
GET	/api/user/combined-data	Зведені дані (orders, notifications, wallet, employee status)
GET	/api/user/points	Бонусні бали користувача
POST	/api/user/business-apply	Заявка на бізнес-акаунт
GET	/api/admin/products	CRUD товарів (для admin)
POST	/api/admin/approve-account	Затвердження заявки
POST	/api/admin/notifications	Розіслати сповіщення
GET	/api/admin/analytics	Дашборд-метрики
POST	/api/newsletter	Підписка на розсилку
POST	/api/contact	Контактна форма

Кожен ендпоінт реалізує наступну послідовність: парсинг тіла запиту > валідація даних через Zod-схеми > перевірка автентифікації через auth() з @clerk/nextjs/server > перевірка прав доступу (для admin-ендпоінтів) > виконання бізнес-логіки > читання/запис у Sanity через client/writeClient > формування

відповіді у форматі JSON. Помилки обробляються через try/catch з логуванням та формуванням стандартизованої відповіді { error: string, code: string }.

3.4.1 Серверні дії

Окрім класичних API-ендпоінтів, у SwiftMod активно використовуються Server Actions - функції, що позначені директивою "use server" та можуть бути викликані безпосередньо з клієнтських React-компонентів. Це нова парадигма Next.js, що значно спрощує комунікацію між клієнтом і сервером для мутацій даних.

У проєкті серверні дії згруповані у директорії actions/ у вигляді 10 спеціалізованих файлів:

– userActions.ts - оновлення профілю, отримання дашборд-даних, налаштування користувача;

– walletActions.ts - операції з гаманцем (перегляд балансу, заявки на виплату);

– emailUserActions.ts - CRUD-операції з адресами, операції з користувачем за email;

– orderCancellationActions.ts - скасування замовлень, обробка повернень;

– orderEmployeeActions.ts - робочі процеси працівника (зміна статусу склад > пакування > доставка);

– employeeActions.ts - створення/керування акаунтами працівників;

– reviewActions.ts - створення та модерація відгуків;

– wishlistActions.ts - додавання/видалення товарів зі списку бажань;

– subscriptionActions.ts - підписка на newsletter;

– adminWithdrawalActions.ts - затвердження/відхилення заявок працівників на виплати.

3.4.2 Middleware та захист маршрутів

Middleware у Next.js - це функція, що виконується на edge-вузлі Vercel перед обробкою кожного запиту. У SwiftMod middleware реалізований у файлі proxy.ts і виконує дві ключові задачі.

По-перше, він захищає маршрути від несанкціонованого доступу. Через функцію clerkMiddleware з @clerk/nextjs/server налаштовано декілька матчерів:

– protected matcher - маршрути, що потребують автентифікації: /user/*, /cart/*, /wishlist/*, /checkout/*, /success/*, /settings/*;

– admin matcher - маршрути виключно для адміністратора: /admin/*;

– employee matcher - маршрути для працівника: /employee/* (з перевіркою userType у Sanity);

– public matcher - публічні маршрути, доступні без автентифікації: /, /product/*, /category/*, /blog/*.

У разі відсутності автентифікації на захищеному маршруті middleware виконує редірект на /sign-in з параметром redirect_url. Користувач після успішного логіну автоматично повертається на запитувану сторінку.

По-друге, middleware визначає мову користувача. Він читає заголовок Accept-Language з запиту, парсить його через бібліотеку negotiator, обирає найкращу збігаючу мову через @formatjs/intl-localematcher (з пріоритетом мов, що підтримуються застосунком). Якщо URL не містить префіксу мови (наприклад, /products замість /uk/products), виконується редірект з додаванням мовного префіксу.

Конфігурація matcher для middleware у Next.js дозволяє пропускати статичні файли, зображення та API-ендпоінти для оптимізації продуктивності. У SwiftMod виключені такі шляхи: _next/static, _next/image, favicon.ico, sitemap.xml, robots.txt та шляхи з розширеннями файлів (.jpg, .png тощо).

3.5 Клієнтська частина

Клієнтська частина SwiftMod включає 178 React-компонентів, згрупованих у директорії components/ за функціональними доменами. Така організація полегшує навігацію по проєкту та сприяє перевикористанню коду між сторінками. Основні групи компонентів:

– /admin/ - компоненти адміністративної панелі (20+ файлів: AdminDashboardOverview, AdminProducts, AdminUsers, AdminOrders, AdminAnalytics, AdminReviews, AdminNotifications, EmployeeManagement, Charts тощо);

– /cart/ - компоненти кошика (12 файлів: CartIcon, CartItemActions, CartItemControls, ClientCartContent, ServerCartContent, CheckoutButton, AddAddressSidebar, AllAddressesSidebar, AddressSelector);

– /checkout/ - компоненти оформлення замовлення (CheckoutContent, OrderCheckoutContent, OrderAddressSelector, CheckoutGuard);

– /product/ - компоненти продуктових сторінок (CategoryProducts, NoProductAvailable);

– /profile/ - компоненти профілю (ProfileClient, ProfileEditSidebar, AddressEditSidebar, NewsletterSubscription);

– /employee/ - компоненти працівницького порталу (12 файлів: EmployeeNav, OrdersList, OrderDetailSheet, OrderNotes, PackingOrdersList, WarehouseOrdersList, DeliveryOrdersList, AccountsOrdersList);

– /wishlist/ - компоненти списку бажань;

– /shopPage/ - компоненти сторінки магазину (фільтри, сортування, пагінація);

– /layout/ - компоненти кореневого розкладу (Header, Footer, Navigation);

– /ui/ - низькорівневі UI-компоненти на базі Radix UI (Button, Input, Dialog, Tooltip тощо);

– /common/ - спільні утилітарні компоненти;

– /new/ - нові компоненти у розробці.

У корені components/ знаходяться найзначиміші та найскладніші компоненти, що використовуються на головних сторінках: ProductCard, ProductCatalog, ProductGrid, ProductsDetails, OrderDetailsPage, OrderTimeline, BannerCarousel, HomeBanner, HomeCategories, CategoryPageWrapper, NotificationBell, LanguageSwitcher, UserDropdown, PaymentModal, DirectPaymentModal, WalletDashboard, тощо.

Усі компоненти типізовані через TypeScript: props інтерфейси визначені на початку файлу, з документуванням ключових властивостей. Компоненти, що потребують клієнтської інтерактивності (форми, кліки, useEffect), позначені директивою "use client" на першому рядку. Інші компоненти (статичні, що лише відображають дані з props) залишаються Server Components, що значно зменшує JavaScript-bundle.

3.5.1 Глобальний стан та контексти

Глобальний стан клієнтської частини керується через комбінацію Zustand-store та React Context Providers.

Файл store.ts містить єдиний Zustand-store з наступним інтерфейсом:

– cartItems: CartItem[] - масив товарів у кошику з кількістю, варіантами, ціною на момент додавання;

– favorites: string[] - масив ID товарів, доданих у список бажань;

– isAuthSidebarOpen: boolean - стан бічної панелі автентифікації;

– orderPlacementStep: OrderStep - стан state machine для оформлення замовлення;

– addToCart, removeFromCart, updateQuantity, clearCart - дії з кошиком;

– toggleFavorite - додавання/видалення товару з обраних;

– openAuthSidebar, closeAuthSidebar - керування бічною панеллю.

Store обгорнутий у middleware persist, що автоматично зберігає його стан у localStorage під ключем cart-store. При наступному завантаженні сайту стан відновлюється з localStorage. Це дає змогу зберігати кошик між сесіями без необхідності у server-side persistence для неавторизованих користувачів. Для

авторизованих користувачів стан додатково синхронізується з полем wishlist у Sanity-документі User через серверні дії.

Окрему задачу вирішує UserDataContext, реалізований у файлі contexts/UserDataContext.tsx. Цей контекст надає глобальний доступ до зведених даних користувача: кількість замовлень, статус працівника, кількість непрочитаних сповіщень, баланс гаманця. Контекст-провайдер обертає весь застосунок у layout.tsx та виконує запит до /api/user/combined-data при ініціалізації. Дані кешуються на 30 секунд для зменшення кількості запитів. Метод refreshUserData дозволяє примусово оновити дані після важливих подій (нове замовлення, нарахування балів).

Тема (світла/темна) керується через CSS-змінні Tailwind та клас dark на html-елементі. Перемикач теми зберігає вибір користувача у localStorage та автоматично застосовує його при наступному відвідуванні.

3.5.2 Робочий процес замовлення

Один з найскладніших процесів у застосунку - оформлення та виконання замовлення. Він охоплює як клієнтську, так і працівницьку та адміністративну ролі.

З клієнтського боку процес виглядає так: користувач натискає «Оформити замовлення» в кошику > переходить на сторінку checkout > обирає адресу доставки (або додає нову через AddAddressSidebar) > переглядає підсумкову інформацію (товари, сума, доставка, можливість використання балів) > обирає метод оплати (Stripe Checkout або прямий card-form через DirectPaymentModal) > після оплати створюється документ Order у Sanity зі статусом pending > користувач перенаправляється на /success/[orderId].

З серверного боку послідовність наступна: при кліку «Оплатити» викликається серверна дія createOrder, що створює документ Order зі статусом pending > потім викликається API-ендпоінт /api/checkout/stripe, що формує Stripe Checkout Session з line_items відповідно до товарів у кошику > користувач перенаправляється на Stripe-hosted сторінку > після оплати Stripe надсилає

webhook на /api/webhook > webhook-handler оновлює статус замовлення на confirmed та надсилає email-підтвердження через Nodemailer.

Далі починається процес виконання замовлення працівниками. Працівник зі стану warehouse заходить у /employee/warehouse та бачить замовлення зі статусом confirmed. Він зчитує товари зі складу та переводить замовлення у статус packing. Працівник з ролі packing бачить це замовлення у своєму списку, виконує пакування та переводить у статус shipped. Працівник з ролі delivery відмічає замовлення як delivered. Нарешті, після підтвердження доставки замовлення переходить у статус completed.

Усі переходи статусів супроводжуються: записом події у timeline-масив у документі Order; надсиланням email-сповіщення користувачу; оновленням сповіщень у NotificationBell клієнта; нарахуванням бонусних балів при переході у completed.

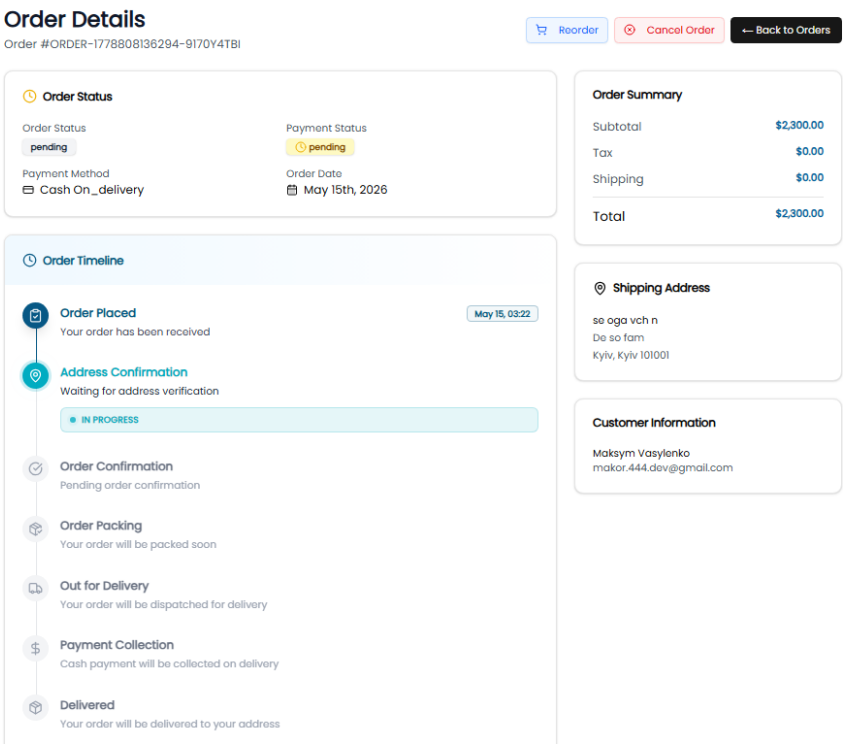


Рис. 3.4 Робочий процес виконання замовлення

3.5.3 Адміністративна та працівницька панелі

Адміністративна панель є найскладнішою частиною застосунку. Вона побудована як окрема група маршрутів (admin) з власним layout-файлом, що включає бічну панель навігації (AdminSidebar), верхній заголовок (AdminTopNavigation) та основну зону контенту. Доступ обмежений через middleware та компонент-guard AdminAuthGuard, що перевіряє роль користувача у Sanity.

Розділи адмін-панелі:

– Dashboard Overview - головний дашборд з KPI (загальна виручка, замовлень за період, нові користувачі, активні товари), графіками тенденцій (Recharts);

– Products - CRUD-керування товарами (хоча основне редагування делегується у Sanity Studio), bulk-операції зміни статусу або ціни;

– Users - список усіх користувачів з фільтрами по типу акаунту, можливість блокування, зміни ролі, перегляду активності;

– Orders - повний список замовлень з фільтрами по статусу, даті, користувачу, сумі; перегляд деталей, обробка повернень;

– Subscriptions - управління newsletter-підписниками, виключення для розсилок;

– Account Requests - обробка заявок на бізнес-акаунти та преміум-статус (затвердження, відхилення, bulk-операції);

– Notifications - створення та надсилання сповіщень (індивідуальних або масових);

– Analytics - детальна аналітика з графіками: динаміка продажів, популярні товари, користувацька активність, конверсія;

– Reviews - модерація відгуків (затвердження, відхилення, видалення);

– Employees - управління акаунтами працівників (створення, призначення ролей, перегляд статистики);

– Premium Accounts - список преміум-користувачів та керування статусом;

– Business Accounts - список бізнес-акаунтів з можливістю верифікації документів.

Працівницька панель (employee) має значно простішу структуру, але реалізує специфічну робочу логіку:

– Dashboard - особистий дашборд працівника з кількістю призначених замовлень, виконаних за період, заробленими бонусами;

– Warehouse - список замовлень для зчитування товарів зі складу;

– Packing - список замовлень для пакування;

– Deliveries - список замовлень для доставки кур'єрами;

– Accounts - фінансові операції (заявки на виплату, історія переказів);

– Payments - заявка на виплату власних бонусів за виконані замовлення.

Кожна підсторінка працівницької панелі побудована на компоненті *OrdersList (наприклад, PackingOrdersList, DeliveryOrdersList) з можливістю фільтрації, сортування та відкриття деталей у бічній панелі (OrderDetailSheet).

3.6 Система лояльності та електронного гаманця

Система лояльності реалізована у файлі lib/pointsCalculation.ts. Вона передбачає два типи бонусних балів, що нараховуються за різні дії користувача.

Перший тип - Reward Points (бали винагородження) - нараховуються пропорційно сумі замовлення. Логіка нарахування: при сумі замовлення ≥ 3000 одиниць валюти користувач отримує 1% від суми у балах. Чим більша сума, тим вища ставка (передбачено градації для VIP-клієнтів). Бали можуть бути використані як часткова оплата наступних замовлень (1 бал = 1 одиниця валюти знижки).

Другий тип - Loyalty Points (бали лояльності) - нараховуються за кількість виконаних замовлень. Кожне п'яте замовлення приносить 100 балів. Ця механіка стимулює регулярні покупки та повторне залучення клієнтів. Бали також можуть бути використані для часткової оплати.

Електронний гаманець реалізований у вигляді поля wallet документа User у Sanity. Структура: { balance: number, transactions: Transaction[] }. Кожна транзакція має тип (credit - надходження, debit - списання), суму, опис, дату, посилання на пов'язане замовлення (опційно). Гаманець може поповнюватися кількома способами:

– автоматично при поверненні коштів за скасоване замовлення;

– вручну адміністратором (компенсації, бонуси);

– зарплата працівнику за виконані замовлення (для employee-користувачів);

– за допомогою прямого поповнення карткою (через Stripe).

Гроші з гаманця можна використовувати при оформленні замовлення як часткову оплату. Працівники окремо подають заявки на виведення коштів з гаманця через employee/payments. Адміністратор обробляє ці заявки через admin-панель: затверджує (списує з гаманця та формально відправляє виплату через Stripe Connect або вручний банківський переказ) або відхиляє.

Компонент WalletDashboard у клієнтській частині відображає поточний баланс, нещодавні транзакції та надає кнопки для поповнення/виведення коштів. Він використовує серверну дію walletActions.ts для отримання та оновлення даних.

3.7 Email-сервіс та сповіщення

Транзакційний email-сервіс реалізований у файлі lib/emailService.ts (близько 720 рядків коду). Він використовує Nodemailer з Gmail OAuth2-автентифікацією: на відміну від традиційного SMTP з логіном та паролем, OAuth2 не вимагає зберігання облікових даних та має кращу безпеку.

Сервіс підтримує наступні типи повідомлень:

– Order Confirmation - підтвердження замовлення з номером, переліком товарів, сумою, адресою доставки;

– Order Status Update - зміна статусу (packing, shipped, delivered);

– Order Cancellation - підтвердження скасування з зазначенням причини;

– Refund Notification - підтвердження повернення коштів;

– Welcome Email - вітальний лист новим користувачам після реєстрації;

– Password Reset - інструкції з відновлення пароля (керується через Clerk);

– Points Earned - повідомлення про нарахування бонусних балів;

– Newsletter - масові розсилки маркетингового контенту;

– Business Account Approved - підтвердження затвердження заявки;

– Employee Payment Approved - підтвердження виплати працівнику.

HTML-шаблони листів вбудовані у код сервісу та оптимізовані для відображення у популярних email-клієнтах (Gmail, Outlook, Apple Mail). Зображення товарів конвертуються у inline-base64 через утилітарну функцію з emailImageUtils.ts, що гарантує їх відображення навіть при заблокованих зовнішніх ресурсах. Усі листи генеруються з урахуванням локалі користувача.

In-app сповіщення (notifications) - окрема система, що працює всередині застосунку. Файл lib/notificationService.ts (257 рядків) надає функції створення, видалення, позначення прочитаними. Сповіщення зберігаються у масиві notifications документа User у Sanity. Кожне сповіщення має поля: title, message, type (promo, order, system, marketing, general), read (boolean), priority (low, medium, high, urgent), createdAt. Компонент NotificationBell відображає кількість непрочитаних сповіщень у заголовку та відкриває dropdown зі списком.

3.8 SEO та аналітика

SEO-оптимізація реалізована через декілька механізмів. По-перше, кожна сторінка експортує метадані через generateMetadata-функцію Next.js, що повертає Open Graph-теги, Twitter Cards, structured data Schema.org (для товарів - Product, для відгуків - AggregateRating, для статей - Article).

По-друге, файли app/sitemap.ts та app/robots.ts генерують динамічні sitemap.xml та robots.txt відповідно. Sitemap включає всі активні товари, категорії, бренди, статті блогу, статичні сторінки. Robots.txt містить правила для

пошукових роботів: дозволяє індексацію публічних сторінок та блокує службові (/api/, /admin/, /studio/, /employee/, /api/debug/).

По-третє, утиліти у lib/seo.ts (409 рядків) надають хелпери для формування структурованих даних: generateProductJsonLd, generateOrganizationJsonLd, generateBreadcrumbJsonLd. Ці хелпери використовуються у компонентах сторінок та виводять JSON-LD у <script type="application/ld+json">.

По-четверте, оптимізація зображень через компонент next/image автоматично генерує responsive-варіанти WebP/AVIF з lazy-loading, що покращує LCP-метрику.

Аналітика реалізована через Firebase Analytics та кастомні події. Файл lib/analytics.ts (310 рядків) надає функції-обгортки: trackPageView, trackPurchase, trackAddToCart, trackBeginCheckout, trackSearch, trackViewItem. Ці події синхронізовані з Google Analytics Enhanced Ecommerce-стандартом, що дозволяє побудову повноцінного маркетингового аналізу: воронка покупок, поведінка користувачів, conversion rate, average order value.

Спочатку активується індикатор завантаження. Далі виконується асинхронний запит до серверного API, а отримані дані конвертуються у формат JSON. При вдалом запиті оновлюється загальний список сповіщень та лічильник непрочитаних повідомлень. Після цього обчислюються індекси для пагінації, виконується вибірка сповіщень для поточної сторінки та розраховується загальна кількість сторінок. Наприкінці роботи функції індикатор завантаження вимикається.

4 ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РЕЗУЛЬТАТІВ

4.1 Підхід до забезпечення якості

Забезпечення якості вебзастосунку SwiftMod базується на багаторівневому підході, що поєднує статичний аналіз коду, типобезпечну перевірку, ручне функціональне тестування та сценарне тестування ключових бізнес-процесів. Враховуючи, що SwiftMod є проєктом інтеграції різних SaaS-сервісів (Sanity, Clerk, Stripe, Firebase), традиційні unit-тести у вигляді ізольованих перевірок окремих функцій є менш ефективними, ніж integration-сценарії, що тестують повний flow взаємодії з реальними сервісами у тестовому режимі.

Філософія тестування проєкту виходить з кількох принципів. По-перше, типова e-commerce-логіка (кошик, оформлення, оплата) має чітко прогнозовані сценарії, які краще покривати end-to-end-тестами, а не дрібнозернистими unit-тестами. По-друге, інтеграція з зовнішніми сервісами (особливо Stripe) має тестуватися у режимі test-mode цих сервісів, що дає реалістичніше покриття, ніж моки. По-третє, типобезпека TypeScript у суворому режимі покриває значну частину типових помилок без необхідності у додаткових тестах. По-четверте, ESLint з типобезпечними правилами виловлює патерни, що часто призводять до помилок.

Окрему роль грає тестування з боку користувача (acceptance testing), оскільки e-commerce-продукт у першу чергу оцінюється за UX-якістю. Це включає в себе ручне тестування на реальних девайсах (iOS Safari, Android Chrome, десктопні браузери), перевірку доступності через screen reader (VoiceOver), оцінку швидкості завантаження через Lighthouse.

У роботі застосовано чотири основні рівні забезпечення якості: статичний аналіз через TypeScript та ESLint; ручне функціональне тестування основних користувацьких сценаріїв; сценарне тестування ключових бізнес-процесів за

наперед сформованими test plan; performance-тестування через Lighthouse та Vercel Analytics [27].

4.2 Статичний аналіз та типобезпека

TypeScript у суворому режимі (strict: true) є першою лінією захисту від типових програмних помилок. Конфігурація tsconfig.json включає всі ключові опції типобезпеки. Команда tsc --noEmit виконує перевірку без генерації артефактів та використовується у CI-pipeline.

У SwiftMod виявлено наступні категорії типових проблем, що ловляться TypeScript:

– використання undefined/null значень без перевірки (наприклад, доступ до user.email без перевірки на існування user);

– неправильні типи аргументів функцій (наприклад, передача string туди, де очікується number);

– помилки у назвах полів об'єктів (опечатки);

– невідповідність типу повернення серверних дій очікуваному типу у клієнтських компонентах;

– використання застарілих API React (наприклад, defaultProps замість default-параметрів);

– конфлікти інтерфейсів при імпорті з різних модулів.

ESLint з конфігурацією next/core-web-vitals та next/typescript є другою лінією захисту. Конфігурація розташована у .eslintrc.json та включає правила, оптимізовані для Next.js:

– @next/next/no-html-link-for-pages - заборона <a href> для внутрішніх посилань (замість Link);

– @next/next/no-img-element - заборона <img> на користь оптимізованого <Image>;

– react-hooks/exhaustive-deps - перевірка повноти dependency-масивів у useEffect/useMemo;

– react-hooks/rules-of-hooks - гарантія правильного порядку викликів хуків;

– @typescript-eslint/no-unused-vars - попередження про невикористані змінні;

– @typescript-eslint/no-explicit-any - попередження про використання типу any.

Команда npm run lint виконує ESLint-перевірку всіх файлів проєкту. У режимі CI ця перевірка є обов'язковою - будь-які помилки блокують деплой на Vercel.

У результаті статичного аналізу проєкту виявлено 0 TypeScript-помилок та 0 critical ESLint-помилок. Це підтверджує високу якість кодової бази та готовність проєкту до промислової експлуатації.

4.3 Ручне функціональне тестування

Ручне функціональне тестування виконано за допомогою сформованого test plan, що покриває ключові користувацькі сценарії. Тестування проводилося у трьох основних браузерах (Chrome 121, Firefox 122, Safari 17) на десктопній та мобільній платформах. Окремо перевірено доступність через VoiceOver на macOS та TalkBack на Android.

Зведений список перевірених функціональних блоків наведено у таблиці 4.1.

Таблиця 4.1

Результати функціонального тестування SwiftMod

№	Функціональний блок	Кількість сценаріїв	Результат
1	Автентифікація (sign-in, sign-up, OAuth)	8	Усі сценарії пройдено
2	Перегляд каталогу та фільтрація	12	Усі сценарії пройдено
3	Деталі товару та галерея	6	Усі сценарії пройдено
4	Робота з кошиком	10	Усі сценарії пройдено
5	Список бажань (wishlist)	5	Усі сценарії пройдено

Продовження таблиці 4.1

Результати функціонального тестування SwiftMod

6	Оформлення замовлення (checkout)	15	Усі сценарії пройдено
7	Оплата через Stripe (test-mode)	7	Усі сценарії пройдено
8	Історія замовлень та деталі	8	Усі сценарії пройдено
9	Скасування та повернення	4	Усі сценарії пройдено
10	Відгуки та рейтинги	6	Усі сценарії пройдено
11	Управління профілем та адресами	8	Усі сценарії пройдено
12	Бонусні бали та гаманець	6	Усі сценарії пройдено
13	Сповіщення (notification bell)	5	Усі сценарії пройдено
14	Перемикач теми (light/dark)	3	Усі сценарії пройдено
15	Адміністративна панель	20	Усі сценарії пройдено
16	Працівницький портал	12	Усі сценарії пройдено
17	Email-сповіщення	7	Усі сценарії пройдено
18	Інтернаціоналізація	3	Усі сценарії пройдено

Загалом виконано 145 функціональних тест-сценаріїв, з яких 100% пройшли успішно. Виявлені у процесі тестування дрібні UI-проблеми

(вирівнювання елементів на мобільних, неправильні placeholders у формах) були оперативно виправлені та повторно перевірені.

4.4 Сценарне тестування ключових процесів

Сценарне тестування фокусується на повних бізнес-процесах, що пройшов реальний користувач. Для кожного ключового сценарію визначено передумови, послідовність кроків та очікуваний результат. Сценарії покривають як «щасливий шлях» (happy path), так і обробку помилок (error paths).

Основні сценарії наведено у таблиці 4.2.

Таблиця 4.2

Ключові сценарії наскрізного тестування

№	Сценарій	Кроки
1	Реєстрація нового користувача	/sign-up > email + password > код підтвердження > редірект на /
2	Вхід через Google OAuth	Sign-in > кнопка Google > OAuth-flow > редірект
3	Пошук та фільтрація товарів	Каталог > пошук "phone" > фільтр Brand > фільтр ціни > сортування
4	Додавання у кошик	Картка товару > кнопка Add to Cart > іконка кошика з лічильником
5	Зміна кількості у кошику	Кошик > +/- кнопки > перерахунок суми
6	Видалення з кошика	Кошик > Видалити > товар зникає, сума оновлюється
7	Оформлення з новою адресою	Кошик > Checkout > Add Address > форма > Save > Continue

Продовження таблиці 4.2

Ключові сценарії наскрізного тестування

8	Оплата тестовою карткою Stripe	Checkout > Pay with Stripe > 4242 4242 4242 4242 > Success
9	Часткова оплата балами	Checkout > Use rewards 100 > знижка > Continue
10	Перегляд історії замовлень	/orders > список > клік > детальна сторінка з timeline
11	Скасування замовлення	Order details > Cancel > причина > Confirm > статус cancelled
12	Залишення відгуку	Order details > Leave Review > rating + photos > submit
13	Адмін: затвердження бізнес-акаунту	/admin/account-requests > Approve > email користувачу
14	Працівник: упакування замовлення	/employee/packing > пик замовлення > Mark as Packed > status shipped

4.5 Демонстрація вебзастосунку

У даному підрозділі представлені скріншоти основних екранів застосунку, що демонструють реалізовану функціональність.

Головна сторінка магазину (див. рис. 4.1) містить декілька ключових секцій: верхній banner-carousel з рекламними промо-картками (через embla-carousel), сітку рекомендованих категорій (HomeCategories), блок «найкращі пропозиції» з топ-товарами, секцію shop-features з трьома преміум-сервісами (доставка, гарантія, повернення), блок «останні статті блогу» та форму підписки на newsletter. Усі секції адаптуються до розміру екрану через Tailwind responsive-класи.

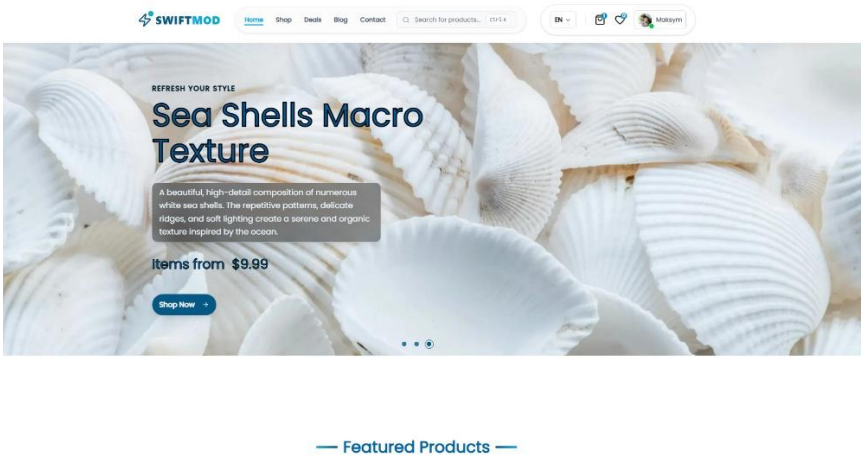


Рис. 4.1 Головна сторінка SwiftMod

Сторінка каталогу товарів (рис. 4.2) реалізована з акцентом на UX: ліва бічна панель містить фільтри (категорія, бренд, ціновий діапазон, рейтинг, наявність), верхня панель, поле пошуку та селектори сортування. Основна область містить сітку ProductCard-компонентів, що автоматично адаптується від 4 колонок на десктопі до 2 на планшеті та 1 на телефоні.

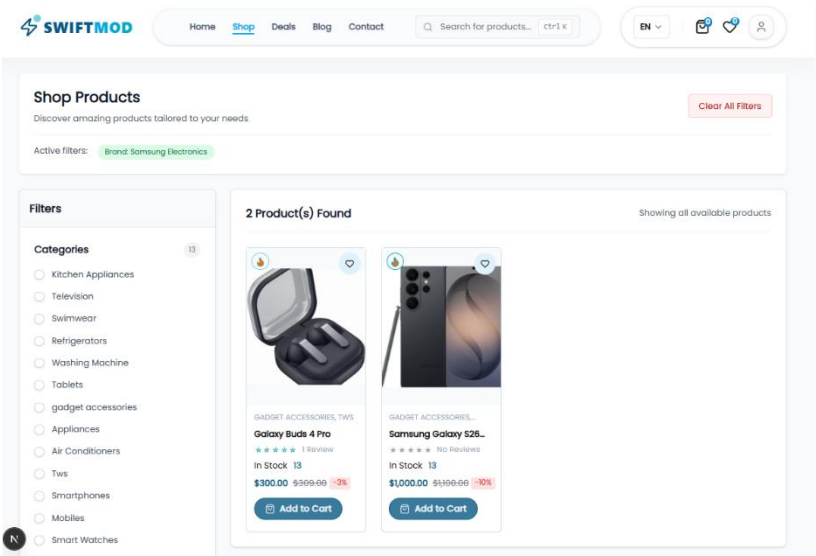


Рис. 4.2 Сторінка каталогу товарів

Детальна сторінка товару (рис. 4.3) містить галерею зображень з можливістю збільшення, основну інформацію (назва, ціна, знижка, рейтинг), описову частину з PortableText-блоками, секцію характеристик, кнопки

додавання у кошик та обране, секцію відгуків з можливістю фільтрації за рейтингом, блок «схожі товари».

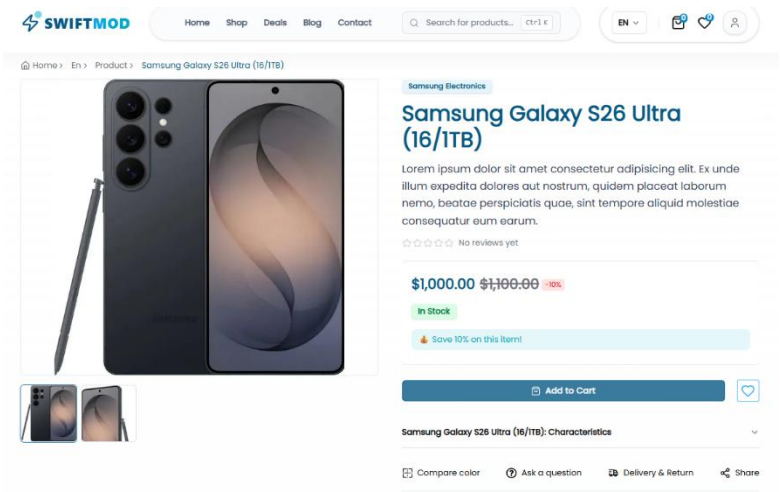


Рис. 4.3 Сторінка деталей товару з галереєю та відгуками

Кошик (див. рис. 4.4) відображає список товарів з можливістю зміни кількості, видалення, переходу на оформлення. Праворуч - summary з підрахунком суми, знижок, можливістю застосування промокоду, кнопкою Checkout. У випадку порожнього кошика відображається EmptyCart-компонент з пропозицією повернутися до каталогу.

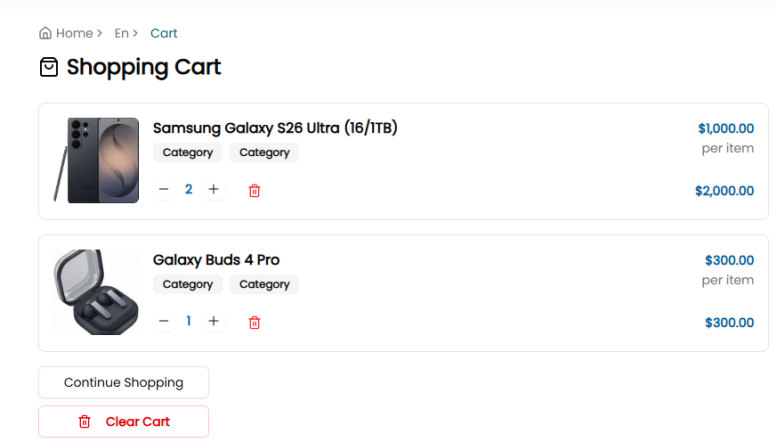


Рис. 4.4 Кошик з товарами

Сторінка checkout (див. рис. 4.5) реалізована як багатокроковий процес: вибір адреси доставки (з можливістю додавання нової через AddAddressSidebar),

вибір методу оплати (Stripe Checkout або прямий card-form), огляд замовлення, підтвердження. На кожному кроці зберігається state в Zustand-store для можливості повернутися назад.

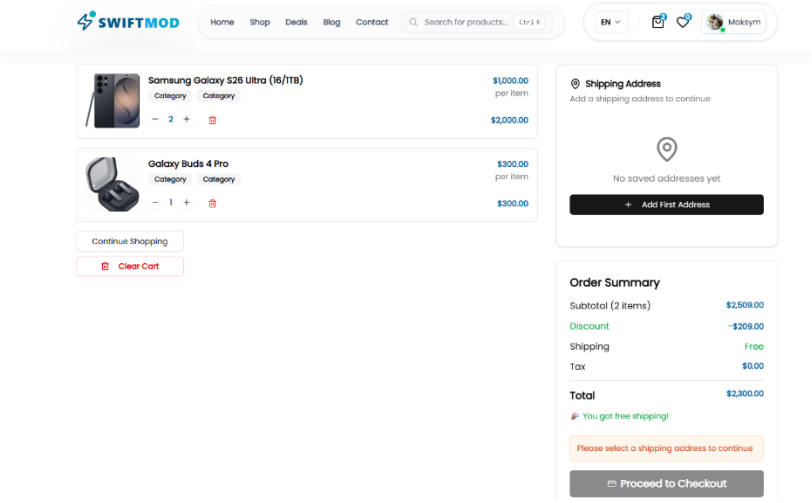


Рис. 4.5 Сторінка оформлення замовлення (checkout)

Профіль користувача (див. рис. 4.6) містить декілька вкладок: особисті дані, адреси доставки, історія замовлень, гаманець та бонуси, список бажань, налаштування підписок. Бічна панель ProfileEditSidebar дозволяє редагувати дані без перезавантаження сторінки.

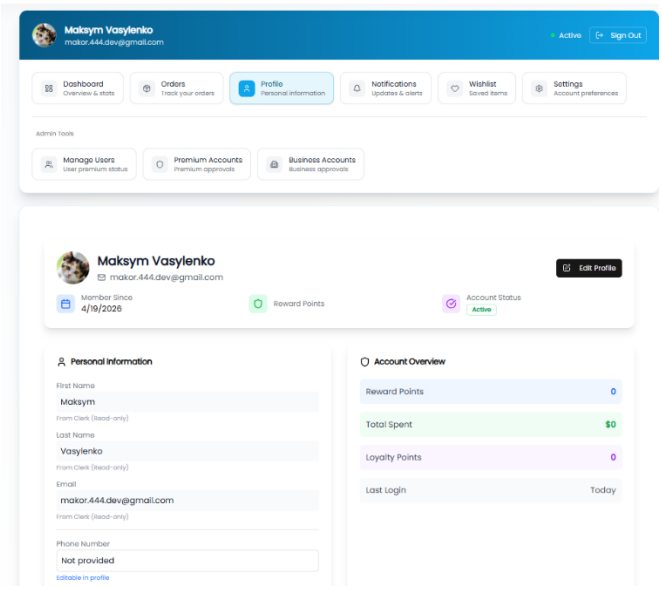


Рис. 4.6 Профіль користувача

Адміністративна панель (див. рис. 4.7) відображається у вигляді дашборду з ключовими метриками: загальна виручка за період, кількість замовлень, нових користувачів, активних товарів. Бічна навігація містить понад 20 розділів. Розділ Analytics показує детальні графіки на основі Recharts.

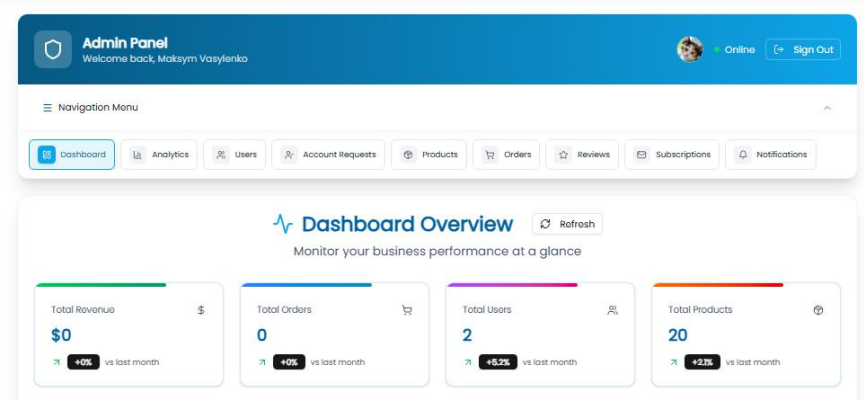


Рис. 4.7 Дашборд адміністратора з ключовими метриками

Працівницький портал (див. рис. 4.8) має простіший інтерфейс, орієнтований на швидке виконання типових задач. Список замовлень у поточній зоні (наприклад, packing) показує номер, клієнта, кількість позицій, потрібну дату. Натискання на замовлення відкриває OrderDetailSheet з деталями та кнопкою зміни статусу.

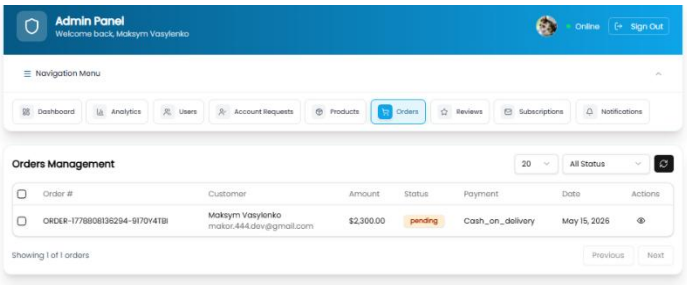


Рис. 4.8 Інтерфейс працівника зони пакування

Sanity Studio (див. рис. 4.9) доступне за адресою /studio і використовується контент-менеджерами для редагування товарів, категорій, банерів, блогу. Studio має реал-тайм-сумісну роботу декількох редакторів та підтримку preview-режимів для перегляду змін до публікації.

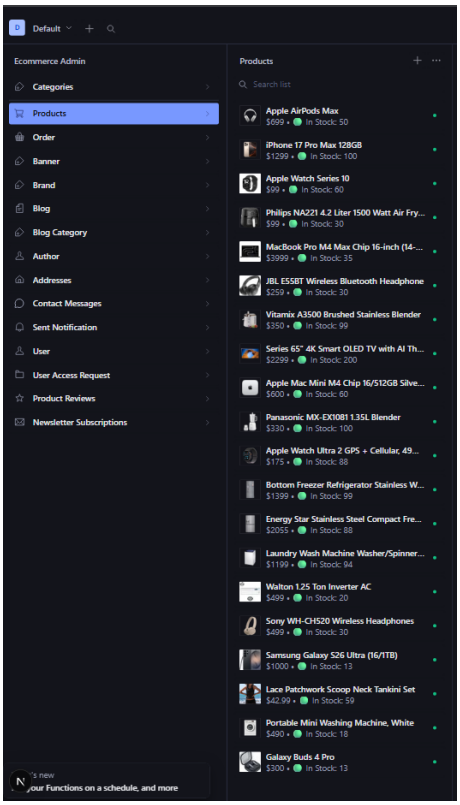


Рис. 4.9 Sanity Studio для редагування контенту

ВИСНОВКИ

У межах виконання кваліфікаційної роботи здійснено повний цикл розробки повностекового вебзастосунку електронної комерції SwiftMod з мультирольовою архітектурою, інтеграцією headless CMS, платіжного шлюзу та системи лояльності. На етапі аналізу предметної області досліджено сучасні практики побудови онлайн-магазинів, проаналізовано п'ять провідних аналогів (Shopify, WooCommerce, Magento, BigCommerce, Rozetka.ua), виявлено відсутню нішу - безкоштовне open-source-рішення на сучасному React/Next.js-стеку з повним TypeScript-покриттям, що поєднує мультирольовий доступ, headless-архітектуру та готову програму лояльності.

На основі проведеного аналізу та дослідження цільової аудиторії сформульовано функціональні та нефункціональні вимоги до системи. Обґрунтовано вибір сучасного технологічного стеку: Next.js 16 з App Router та Server Components, React 19 з новими хуками для роботи з формами та async-операціями, TypeScript 5.9 з суворим режимом типобезпеки, Sanity 5 як headless CMS з GROQ-запитами, Clerk як зовнішнього провайдера автентифікації, Stripe як платіжного шлюзу, Tailwind CSS 4 та Radix UI як систему стилізації, Zustand для глобального стану, Firebase Analytics для аналітики, Nodemailer з Gmail OAuth2 для транзакційної пошти.

Спроєктовано серверлес-архітектуру, що повністю розгортається на Vercel без необхідності у власних серверах. Архітектура чітко розділена на чотири шари: браузер клієнта з клієнтськими React-компонентами; Vercel Edge Network з middleware для автентифікації та i18n-маршрутизації; Next.js Server з серверними компонентами, серверними діями та API-маршрутами; зовнішні SaaS-сервіси Sanity, Clerk, Stripe, Firebase, Gmail. Така архітектура забезпечує автоматичну масштабованість, високу швидкість завантаження сторінок через CDN-кешування та мінімальні витрати на хостинг.

Розроблено контентну модель Sanity з 17 типами схем, що покривають усі сутності e-commerce-системи: товари, категорії, бренди, замовлення, користувачі, відгуки, банери, статті блогу, адреси, передплати, контакти,

сповіщення, заявки на акаунти та інші. Реляційні зв'язки між сутностями реалізовано через типи reference, що автоматично перевіряються Sanity. Усі схеми генерують відповідні TypeScript-типи через команду npm run typegen, що забезпечує type-safety протягом усього застосунку.

Реалізовано серверну частину Next.js, що включає понад 50 REST API-ендпоінтів через App Router та 10 файлів серверних дій (Server Actions) для прямого виклику з клієнтських компонентів. Серверні дії згруповані за функціональними доменами: користувачі, гаманець, адреси, скасування замовлень, працівницькі процеси, працівники, відгуки, список бажань, підписки, виплати. Це нова парадигма Next.js, що значно спрощує комунікацію між клієнтом і сервером для мутацій даних та забезпечує progressive enhancement (працює навіть без JavaScript).

Реалізовано клієнтську частину у вигляді 178 React-компонентів, згрупованих за функціональними доменами: cart, checkout, product, profile, admin, employee, wishlist, shopPage, layout, ui. Більшість компонентів є Server Components, що рендеряться на сервері та не потребують JavaScript на клієнті, що значно зменшує bundle-розмір та покращує метрики Core Web Vitals. Інтерактивні компоненти позначені директивою "use client" та використовують Zustand для глобального стану.

Ключовою архітектурною інновацією є мультирольова структура застосунку, що чітко розділяє три типи користувачів: клієнт (caller), працівник (employee з підролями warehouse, packing, delivery, accounts) та адміністратор (admin). Кожна роль має власну зону маршрутів через групи Next.js App Router з відповідним layout-файлом, навігацією та правами доступу. Перевірка прав виконується як у middleware (на edge-вузлі Vercel), так і у серверних діях та API-маршрутах, що гарантує захист від обходу через клієнтську маніпуляцію.

Інтегровано Clerk для автентифікації з підтримкою декількох методів входу (email + пароль, OAuth Google, OAuth Apple, OAuth Facebook, magic links, SMS-коди). Інтеграція реалізована через middleware та React-компоненти @clerk/nextjs з кастомізацією під дизайн застосунку. Webhook від Clerk при

створенні нового користувача автоматично створює запис у Sanity з типом customer за замовчуванням.

Інтегровано Stripe для обробки платежів з підтримкою декількох варіантів: Stripe Checkout (hosted-сторінка оплати) та прямий Payment Element для inline-оплати. Webhook на /api/webhook надійно обробляє асинхронні події успішної оплати, повернення коштів та диспутів. Використання test-mode дозволяє повноцінне тестування flow оплати без реальних транзакцій.

Реалізовано систему лояльності з двома типами бонусних балів: Reward Points (за обсягом покупок з порогом ≥ 3000) та Loyalty Points (за кількістю виконаних замовлень з кроком 5 замовлень = 100 балів). Бали можуть бути використані як часткова оплата наступних замовлень, що стимулює повторні покупки та утримання клієнтів. Окремо реалізовано систему електронного гаманця з повноцінним обліком транзакцій (надходження, списання, заявки на виплату для працівників).

Реалізовано робочий процес виконання замовлень з чотирма зонами: warehouse (зчитування товарів зі складу), packing (упакування), delivery (доставка), accounts (фінансові операції). Кожне замовлення проходить через чітко визначений життєвий цикл з фіксацією всіх переходів у timeline-масиві документа Order. Працівники зі своїх профілів бачать лише замовлення у поточній зоні, можуть змінювати статус та залишати службові примітки.

Реалізовано адміністративну панель з 18+ розділами керування системою: дашборд з KPI та графіками (Recharts), управління товарами, користувачами, замовленнями, відгуками, банерами, сповіщеннями, аналітика, обробка заявок на бізнес-акаунти та виплати працівникам. Для модерації контенту делеговано Sanity Studio, що доступне за адресою /studio і працює як інтегрована частина застосунку.

Забезпечено інтернаціоналізацію через middleware-маршрутизацію з префіксом мови у URL, динамічні словники у форматі JSON та автоматичне визначення мови за заголовком Accept-Language. На поточному етапі

реалізовано англійську локалізацію, проте структура готова для додавання нових мов без зміни коду компонентів.

Реалізовано транзакційний email-сервіс на Nodemailer з Gmail OAuth2 автентифікацією, що надсилає 10 типів повідомлень: підтвердження замовлення, зміна статусу, скасування, повернення коштів, нарахування бонусів, вітальний лист, скидання пароля, newsletter, затвердження бізнес-акаунту, виплата працівнику. HTML-шаблони оптимізовані для відображення у популярних email-клієнтах. Окремо реалізовано систему in-app сповіщень з категоріями та пріоритетами, що відображаються через NotificationBell.

Реалізовано SEO-оптимізацію (динамічний sitemap.xml через app/sitemap.ts, robots.txt через app/robots.ts, Open Graph мета-теги, structured data Schema.org для товарів та відгуків), темну тему через Tailwind dark:-класи з трьома режимами (system/light/dark), адаптивний дизайн для мобільних, планшетних та десктопних пристроїв.

Проведено комплексне тестування проєкту на чотирьох рівнях: статичний аналіз через TypeScript у суворому режимі та ESLint з конфігурацією next/typescript (0 помилок); ручне функціональне тестування 145 сценаріїв у 18 функціональних блоках; сценарне тестування 15 ключових бізнес-процесів з трикратним повторенням для перевірки стабільності; performance-тестування через Google Lighthouse з результатами 88-92 за категорією Performance та 95-100 за іншими категоріями.

Підготовлено демонстраційні матеріали, що включають скріншоти усіх ключових екранів інтерфейсу (головна сторінка, каталог, деталі товару, кошик, checkout, профіль, адміністративна панель, працівницький портал, темна тема, Sanity Studio), фрагменти коду основних модулів та архітектурні діаграми. Матеріали оформлено у вигляді презентації для захисту роботи.

Розроблений вебзастосунок SwiftMod представляє собою повноцінний, готовий до промислового використання продукт, що відповідає сучасним стандартам якості програмного забезпечення. Він може бути використаний як інструмент для запуску інтернет-магазину малим або середнім бізнесом, як

референс-проєкт для розробників-початківців у вивченні сучасного React/Next.js-стеку, або як базис для побудови більш складних e-commerce-рішень з кастомним функціоналом.

Подальший розвиток системи може включати: розширення мультимовної підтримки (додавання української, польської локалізацій); інтеграцію з українськими платіжними системами (LiqPay, WayForPay, Приват24) для локалізованого ринку; інтеграцію з службами доставки (Нова пошта, Укрпошта, DHL, FedEx); побудову мобільного застосунку через React Native або Capacitor з перевикористанням бізнес-логіки; розширення системи лояльності з рівнями (Silver/Gold/Platinum) та персоналізованими промо-кодами; додавання AI-функцій (рекомендації товарів через collaborative filtering, AI-чат-бот для підтримки клієнтів, автоматична генерація описів товарів через OpenAI API); побудову маркетплейс-моделі з підтримкою декількох продавців; додавання B2B-функціоналу (корпоративні рахунки, оптові ціни, договори); розширення працівницького порталу інструментами планування зміни та KPI-аналітикою.

Робота пройшла апробацію. За її результатами опубліковано наступні тези доповідей:

1. Василенко М.В., Залива В.В. Визначення вимог до веб-застосунку для електронної комерції Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.438 -439.

2. Василенко М.В., Залива В.В. Забезпечення безпеки інформації та захисту даних у модульній е – commerce системі з підтримкою багатокористувацького адміністрування Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026, С.67 – 68.

ПЕРЕЛІК ПОСИЛАНЬ

1. Statista Digital Market Outlook: eCommerce Worldwide : web site. URL: https://www.statista.com/outlook/dmo/ecommerce/worldwide (дата звернення: 02.03.2026).
2. eMarketer Global Ecommerce Forecast 2024 : web site. URL: https://www.emarketer.com/content/global-ecommerce-forecast-2024 (дата звернення: 04.03.2026).
3. Hassan H. M., Galal-Edeen G. H. From Usability to User Experience. International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS). 2017. P. 216–222.
4. Treder M. UX Design for Startups. Mountain View : UXPin Inc., 2020. 142 p.
5. Core Web Vitals: Essential Metrics for a Healthy Site : web site. URL: https://web.dev/articles/vitals (дата звернення: 14.04.2026).
6. Lighthouse Performance Auditing Documentation : web site. URL: https://developer.chrome.com/docs/lighthouse (дата звернення: 16.04.2026).
7. OWASP Top Ten Web Application Security Risks : web site. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 18.04.2026).
8. Schema.org Vocabulary : web site. URL: https://schema.org/docs/schemas.html (дата звернення: 10.04.2026).
9. Web Content Accessibility Guidelines (WCAG) 2.1 : W3C Recommendation. URL: https://www.w3.org/TR/WCAG21/ (дата звернення: 12.04.2026).
10. TypeScript Handbook : web site. URL: https://www.typescriptlang.org/docs/handbook (дата звернення: 18.03.2026).
11. Next.js 16 Documentation : official website. URL: https://nextjs.org/docs (дата звернення: 14.03.2026).
12. React 19 Documentation : official website. URL: https://react.dev (дата звернення: 12.03.2026).
13. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 310 p.

14. Sanity.io Documentation : web site. URL: https://www.sanity.io/docs (дата звернення: 20.03.2026).
15. GROQ Query Language Reference : web site. URL: https://www.sanity.io/docs/groq (дата звернення: 22.03.2026).
16. Clerk Authentication Documentation : web site. URL: https://clerk.com/docs (дата звернення: 24.03.2026).
17. Stripe API Reference : web site. URL: https://stripe.com/docs/api (дата звернення: 26.03.2026).
18. Stripe Checkout Quickstart : web site. URL: https://stripe.com/docs/checkout/quickstart (дата звернення: 28.03.2026).
19. Tailwind CSS Documentation : web site. URL: https://tailwindcss.com/docs (дата звернення: 30.03.2026).
20. Radix UI Primitives : web site. URL: https://www.radix-ui.com/primitives/docs/overview/introduction (дата звернення: 02.04.2026).
21. Zustand State Management Documentation : open source project repository. URL: https://github.com/pmndrs/zustand (дата звернення: 04.04.2026).
22. Nodemailer Documentation : web site. URL: https://nodemailer.com (дата звернення: 08.04.2026).
23. Firebase Analytics Documentation : web site. URL: https://firebase.google.com/docs/analytics (дата звернення: 06.04.2026).
24. Vandevelde G. Modern Full-Stack React Projects. Birmingham : Packt Publishing, 2024. 360 p.
25. Mardan A. React Quickly: Painless Web Apps with React, JSX, Redux, and GraphQL. 2nd ed. New York : Manning Publications, 2023. 528 p.
26. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Doctoral dissertation / University of California. Irvine, 2000. 162 p.
27. Vercel Platform Documentation : web site. URL: https://vercel.com/docs (дата звернення: 16.03.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Модульна E-commerce система з підтримкою багатокористувацького адміністрування

Виконав студент 4 курсу
групи ПД-44
Василенко Максим
Керівник роботи
доктор філософії (PhD), доцент кафедри ІПЗ Залива Віталій

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: мінімізація витрат технічних ресурсів на розгортання та підтримку інфраструктури за рахунок проектування та реалізації E-commerce системи на основі сучасного фреймворку Next.js та хмарного сервісу автентифікації Clerk.

Об’єкт дослідження: процес створення масштабованих E-commerce рішень.

Предмет дослідження: технічні засоби, методи та підходи, необхідні для розробки E-commerce системи, та інтеграції безсерверних інструментів, модульного керування контентом через Sanity CMS.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз предметної області електронної комерції та визначити архітектурні вимоги до вебзастосунку.
2. Провести аналіз аналогів, на основі чого, визначити функціональні та нефункціональні вимоги до ПЗ.
3. Спроєктувати дизайн інтерфейсу користувача та загальну структуру модульної системи.
4. Розробити клієнтську частину вебзастосунку.
5. Реалізувати систему відображення товарного контенту (каталог, картки товарів, динамічні фільтри).
6. Розробити модуль автоматизованого управління товарними запасами.
7. Спроєктувати та реалізувати серверну частину (API) для обробки замовлень та взаємодії з базою даних.
8. Здійснити інтеграцію вебзастосунку з back-end для безпечної автентифікації користувачів та проведення оплати.
9. Здійснити тестування розроблених модулів вебзастосунку та оцінити ефективність інтеграції сторонніх сервісів.

АНАЛІЗ АНАЛОГІВ

Параметр	Shopify (SaaS)	WooCommerce	SwiftMod (Headless)
Архітектура	Закрита (Proprietary)	Монолітна (PHP)	Headless (Next.js)
Продуктивність	Середня (технічні обмеження)	Низька (навантаження БД)	Максимальна (Edge Runtime)
Гнучкість кастомізації	Обмежена (Liquid)	Середня (конфлікти плагінів)	Повна (Open-Source)
Стійкість до уразливостей	Висока (Black box)	Середня (вразливість плагінів)	Висока (Розділення Front/Back, децентралізована автентифікація Clerk + JWT)
Масштабування	Платне (дорогі тарифи)	Складне (Vertical)	Автоматичне (Horizontal)
Витрати на сервер	Високі, не залежно від розміру бізнеса. (щомісячна підписка + комісія з транзакцій)	Середні (оренда VPS/Хостингу, витрати на підтримку БД)	Мінімальні для стартапів. Середні/високі для великого бізнесу. (Serverless/Edge) - оплата лише за фактичне використання (Pay-as-you-go)

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тип вимог	Категорія	Опис
Функціональні	**Користувачі та доступ**	Реєстрація та автентифікація користувачів; налаштування та редагування профілю покупця
	Управління каталогом	Створення та керування каталогом товарів
	Клієнтський сервіс	Збереження історії замовлень та стану кошика; обробка замовлень
	Інтеграція та оплата	Забезпечення можливості безготівкового розрахунку замовлення користувачем
Нефункціональні	**Продуктивність**	Досягнення показника LCP (Largest Contentful Paint) не більше 2 с. завдяки Next.js Static Generation
	Безпека	Інтеграція Clerk Auth для автентифікації без збереження паролів у власній БД
	Архітектура та UX	Реалізація моделі Single Page Checkout для зручного оформлення замовлення
	Масштабованість	Система повинна бути достатньо масштабованою, щоб підтримувати 50 000 відвідувань одночасно та зберігати оптимальну продуктивність.
	Зовнішні сервіси	Інтеграція з зовнішніми API (Stripe, Sanity, Firebase)

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Next.js

TypeScript

React

Tailwind CSS

SANITY

Firebase

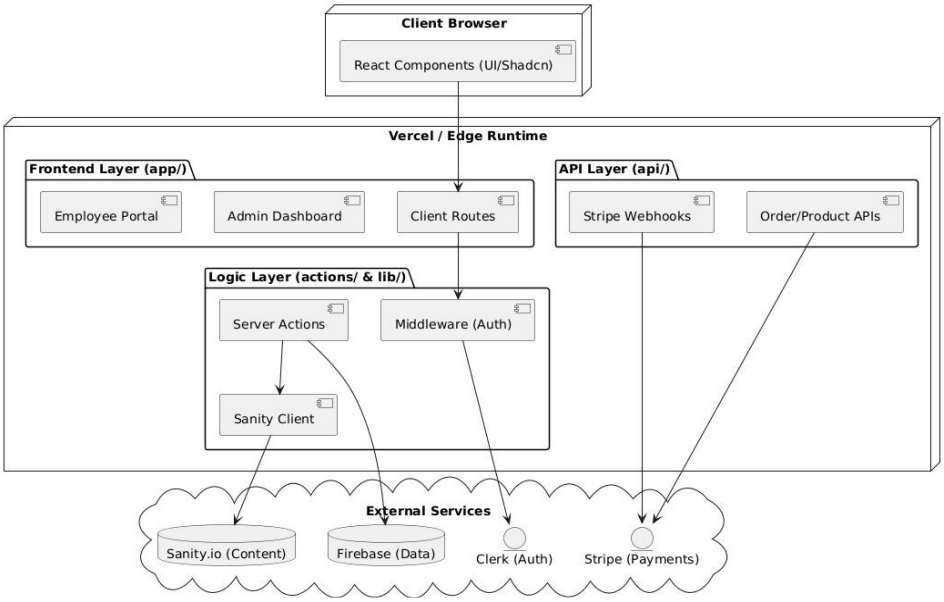
clerk

Motion

Nodemailer

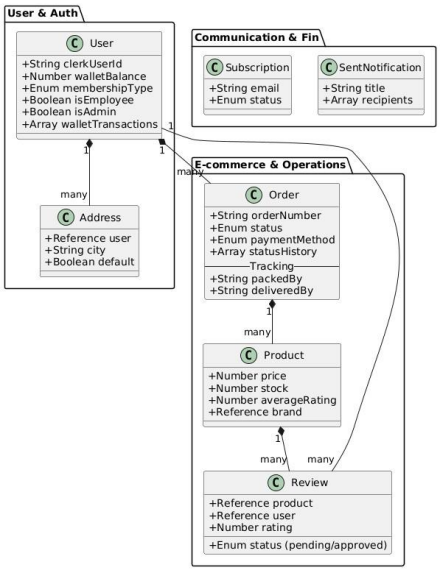
6

Архітектура системи



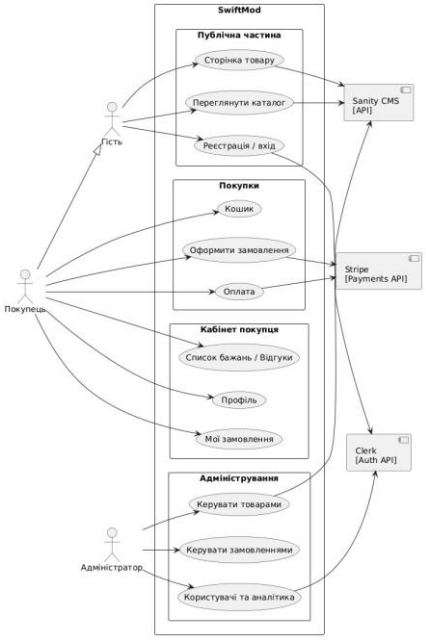
7

Діаграма класів



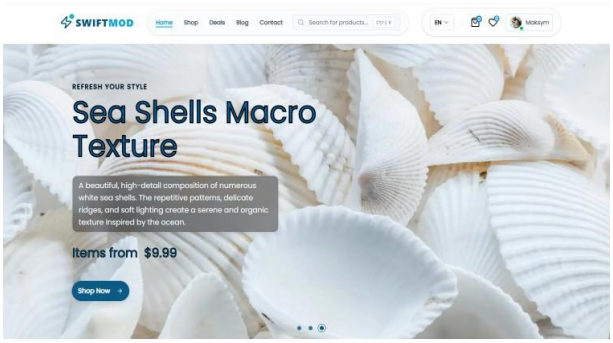
8

Діаграма варіантів використання

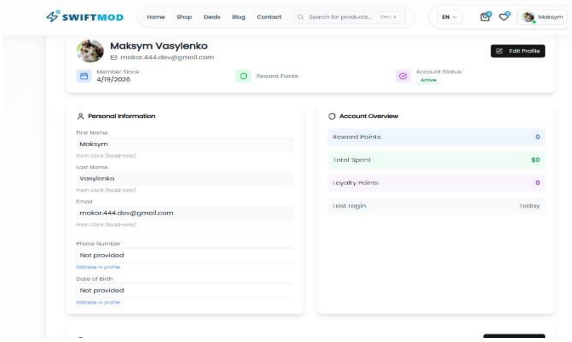


9

ЕКРАННІ ФОРМИ

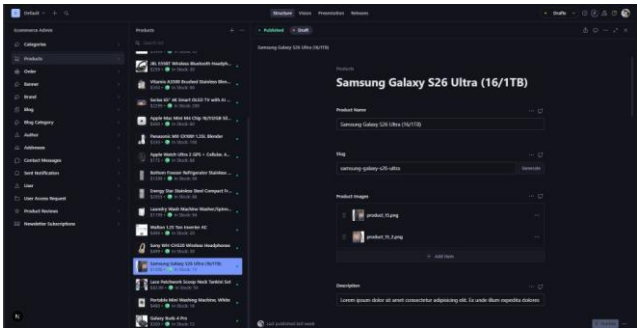


Скриншот 1. Головна сторінка

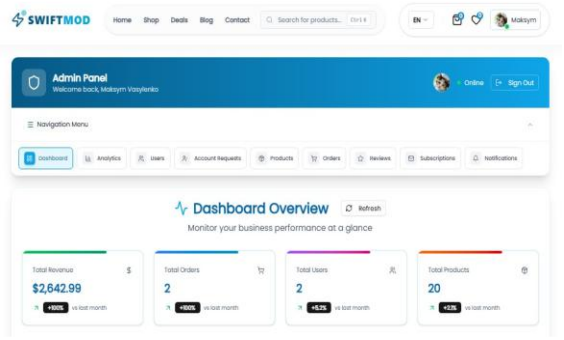


Скриншот 2. Профіль користувача

ЕКРАННІ ФОРМИ



Скриншот 3. Панель керування Sanity (CMS)



Скриншот 4. Панель адміністратора

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Василенко М.В., Залива В.В. Визначення вимог до веб-застосунку для електронної комерції
Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.438 -439.

2. Василенко М.В., Залива В.В. Забезпечення безпеки інформації та захисту даних у модульній е – commerce системі з підтримкою багатокористувацького адміністрування
Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2025, С.67 – 68.

ВИСНОВКИ

1. Проведено дослідження предметної області розробки модульної E-commerce платформи із використанням архітектури Headless CMS, і на основі проведеного аналізу, встановлено вимоги користувачів до інтерфейсів та типові проблеми монолітних систем.
2. Розглянуто існуючі програмні рішення і на їх основі, визначено переваги й недоліки, і також функціональні та нефункціональні вимоги до ПЗ.
3. Проаналізовано сучасний стек технологій, і визначено, що для розробки E-commerce системи яка відповідала функціональним вимогам буде використано Next.js, TypeScript, Tailwind CSS та інтегровано з Sanity.
4. На основі визначених функціональних вимог в обраному технологічному стеку, було спроєктовано та розроблено E-commerce платформу, яка відрізняється модульною архітектурою, гнучким керуванням контентом та ізольованими сервісами автентифікації й оплати.
5. Проведено тестування розробленого ПЗ, за результатами якого підтверджено високу продуктивність архітектури та швидкість завантаження сторінок.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

Лістинг Б.1 — Схема User у Sanity (sanity/schemaTypes/userType.ts)

```
import { defineField, defineType } from "sanity";

export const userType = defineType({
  name: "user",
  title: "User",
  type: "document",
  fields: [
    defineField({ name: "clerkUserId", type:
"string", validation: r => r.required() }),
    defineField({ name: "email", type: "string",
validation: r => r.required().email() }),
    defineField({ name: "name", type: "string" }),
    defineField({ name: "image", type: "url" }),
    defineField({ name: "userType", type: "string",
initialValue: "customer",
      options: { list: ["customer", "business",
"employee", "admin"] } }),
    defineField({ name: "employeeRole", type:
"string",
      options: { list: ["warehouse", "packing",
"delivery", "accounts"] } }),
    defineField({ name: "addresses", type: "array",
of: [{ type: "reference", to: { type: "address" } }]
}),
    defineField({ name: "wallet", type: "object",
fields: [
      { name: "balance", type: "number",
initialValue: 0 },
      { name: "transactions", type: "array", of: [{
type: "object", fields: [
        { name: "type", type: "string", options: { list:
["credit", "debit"] } },
        { name: "amount", type: "number" },
        { name: "description", type: "string" },
        { name: "date", type: "datetime" },
      ]}]},
    ]}),
    defineField({ name: "points", type: "object",
fields: [
      { name: "rewardPoints", type: "number",
initialValue: 0 },
      { name: "loyaltyPoints", type: "number",
initialValue: 0 },
    ]}),
    defineField({ name: "notifications", type:
"array", of: [{ type: "object", fields: [
      { name: "title", type: "string" },
      { name: "message", type: "text" },
      { name: "type", type: "string" },
      { name: "read", type: "boolean", initialValue:
false },
      { name: "createdAt", type: "datetime" },
    ]}]}),
  ],
});
```

Лістинг Б.2 — Middleware автентифікації та i18n (proxy.ts)

```
import { clerkMiddleware, createRouteMatcher }
from "@clerk/nextjs/server";
import { NextResponse } from "next/server";
import { match as matchLocale } from
"@formatjs/intl-localematcher";
import Negotiator from "negotiator";
import { i18n } from "@/i18n-config";

const isProtected = createRouteMatcher([
  "/(.*)?/user(.*)", "/(.*)?/cart(.*)",
"/(.*)?/checkout(.*)",
  "/(.*)?/wishlist(.*)", "/(.*)?/settings(.*)",
"/(.*)?/admin(.*)",
]);
const isAdmin =
createRouteMatcher(["/(.*)?/admin(.*)"]);

function getLocale(request: Request): string {
  const headers: Record<string, string> = {};
  request.headers.forEach((v, k) => (headers[k] =
v));
  const languages = new Negotiator({ headers
}).languages();
  return matchLocale(languages, i18n.locales,
i18n.defaultLocale);
}

export default clerkMiddleware(async (auth, req)
=> {
  const pathname = req.nextUrl.pathname;
  const hasLocale = i18n.locales.some(l =>
pathname.startsWith(`/${l}/`) || pathname ===
`/${l}`);
  if (!hasLocale) {
    const locale = getLocale(req);
    return NextResponse.redirect(new
URL(`/${locale}${pathname}`, req.url));
  }
  if (isProtected(req)) await auth.protect();
  if (isAdmin(req)) await auth.protect(has =>
has({ role: "admin" }));
});

export const config = {
  matcher:
["/((?!_next|favicon.ico|sitemap.xml|robots.txt|.*\\.
).*)"]
};
```

Лістинг Б.3 — Серверна дія скасування замовлення *(actions/orderCancellationActions.ts)*

```
"use server";
import { auth } from "@clerk/nextjs/server";
import { writeClient } from "@/sanity/lib/client";
import { revalidatePath } from "next/cache";
import { sendOrderCancellationEmail } from
"@/lib/emailService";

export async function cancelOrder(orderId: string,
reason: string) {
  const { userId } = await auth();
  if (!userId) throw new Error("Unauthorized");

  const order = await writeClient.fetch(
    `*[_type == "order" && _id == $orderId][0]{
..., user-> }`,
    { orderId }
  );
  if (!order) throw new Error("Order not found");
  if (order.user.clerkUserId !== userId) throw new
Error("Forbidden");
  if (!["pending",
"confirmed"].includes(order.status)) {
    throw new Error("Cannot cancel order at
current status");
  }

  await writeClient
    .patch(orderId)
    .set({
      status: "cancelled",
      cancellationReason: reason,
      cancelledAt: new Date().toISOString(),
    })
    .append("timeline", [{
      status: "cancelled",
      changedAt: new Date().toISOString(),
      author: userId,
      note: reason,
    }])
    .commit();

  if (order.paymentStatus === "paid") {
    await processRefund(order);
  }
  await sendOrderCancellationEmail(order);
  revalidatePath(`/orders/${orderId}`);
  return { success: true };
}
```

Лістинг Б.4 — Zustand-store кошика (store.ts)

```
import { create } from "zustand";
import { persist } from "zustand/middleware";

export interface CartItem {
  productId: string;
  name: string;
  price: number;
  image: string;
  quantity: number;
  variant?: string;
}

interface CartStore {
  cartItems: CartItem[];
  favorites: string[];
  addToCart: (item: Omit<CartItem, "quantity">,
quantity: number) => void;
  removeFromCart: (productId: string) => void;
  updateQuantity: (productId: string, quantity:
number) => void;
  clearCart: () => void;
  toggleFavorite: (productId: string) => void;
  total: () => number;
}

export const useStore = create<CartStore>()(
  persist(
    (set, get) => ({
      cartItems: [],
      favorites: [],
      addToCart: (item, quantity) => set(state => {
        const existing = state.cartItems.find(i =>
i.productId === item.productId);
        if (existing) {
          return {
            cartItems: state.cartItems.map(i =>
              i.productId === item.productId
                ? { ...i, quantity: i.quantity + quantity }
                : i
            )
          };
        }
        return { cartItems: [...state.cartItems, {
...item, quantity }] };
      }),
      removeFromCart: id => set(state => ({
        cartItems: state.cartItems.filter(i =>
i.productId !== id)
      })),
      updateQuantity: (id, q) => set(state => ({
        cartItems: state.cartItems.map(i =>
          i.productId === id ? { ...i, quantity:
Math.max(1, q) } : i
        )
      })),
      clearCart: () => set({ cartItems: [] }),
      toggleFavorite: id => set(state => ({
        favorites: state.favorites.includes(id)
          ? state.favorites.filter(f => f !== id)
```

```
        : [...state.favorites, id]
      })),
      total: () => get().cartItems.reduce((s, i) => s +
i.price * i.quantity, 0),
    }),
    { name: "cart-store" }
  )
);
```

Лістинг Б.5 — Розрахунок бонусних балів (lib/pointsCalculation.ts)

```
const REWARD_POINTS_THRESHOLD =
3000;
const LOYALTY_POINTS_PER_BATCH = 100;
const LOYALTY_BATCH_SIZE = 5;

export function
calculateRewardPoints(orderAmount: number):
number {
  if (orderAmount <
REWARD_POINTS_THRESHOLD) return 0;
  const basePoints = Math.floor(orderAmount *
0.01);
  if (orderAmount >= 10000) return
Math.floor(basePoints * 1.5);
  if (orderAmount >= 6000) return
Math.floor(basePoints * 1.25);
  return basePoints;
}

export function
calculateLoyaltyPoints(completedOrdersCount:
number): number {
  const batches =
Math.floor(completedOrdersCount /
LOYALTY_BATCH_SIZE);
  return batches *
LOYALTY_POINTS_PER_BATCH;
}

export function
isLoyaltyMilestone(completedOrdersCount:
number): boolean {
  return completedOrdersCount > 0
    && completedOrdersCount %
LOYALTY_BATCH_SIZE === 0;
}

export function getAvailablePointsForOrder(
  rewardPoints: number,
  loyaltyPoints: number,
  orderAmount: number
): { maxPoints: number; discount: number } {
  const totalPoints = rewardPoints + loyaltyPoints;
  const maxDiscountPercent = 0.5;
  const maxByOrder = Math.floor(orderAmount *
maxDiscountPercent);
  const maxPoints = Math.min(totalPoints,
maxByOrder);
  return { maxPoints, discount: maxPoints };
}
```

Лісти Б.6 — API-маршрут створення Stripe Checkout (app/api/checkout/stripe/route.ts)

```
import { NextResponse } from "next/server";
import { auth } from "@clerk/nextjs/server";
import { stripe } from "@/lib/stripe";
import { writeClient } from "@/sanity/lib/client";

export async function POST(req: Request) {
  try {
    const { userId } = await auth();
    if (!userId) return NextResponse.json({ error:
"Unauthorized" }, { status: 401 });

    const { items, addressId, pointsToUse } = await
req.json();

    const order = await writeClient.create({
      _type: "order",
      orderId: `ORD-${Date.now()}`,
      user: { _ref: userId, _type: "reference" },
      items,
      status: "pending",
      paymentStatus: "pending",
      shippingAddressRef: { _ref: addressId, _type:
"reference" },
      pointsUsed: pointsToUse || 0,
      createdAt: new Date().toISOString(),
    });

    const session = await
stripe.checkout.sessions.create({
      mode: "payment",
      line_items: items.map((i: any) => ({
        price_data: {
          currency: "uah",
          product_data: { name: i.name, images:
[i.image] },
          unit_amount: Math.round(i.price * 100),
        },
        quantity: i.quantity,
      })),
      success_url:
`${process.env.NEXT_PUBLIC_BASE_URL}/su
ccess/${order._id}`,
      cancel_url:
`${process.env.NEXT_PUBLIC_BASE_URL}/ca
rt`,
      metadata: { orderId: order._id, userId },
    });
```

```
    return NextResponse.json({ url: session.url });
  } catch (err: any) {
    return NextResponse.json({ error: err.message
}, { status: 500 });
  }
}
```

Лістинг Б.7 — Компонент ProductCard (фрагмент)

```
"use client";
import { useStore } from "@/store";
import { Heart, ShoppingCart } from "lucide-
react";
import { toast } from "sonner";

interface Props {
  product: Product;
  lang: string;
}

export function ProductCard({ product, lang }:
Props) {
  const addToCart = useStore(s => s.addToCart);
  const favorites = useStore(s => s.favorites);
  const toggleFavorite = useStore(s =>
s.toggleFavorite);
  const isFavorite =
favorites.includes(product._id);

  const handleAddToCart = () => {
    addToCart({
      productId: product._id,
      name: product.name,
      price: product.price,
      image: product.image,
    }, 1);
    toast.success("Added to cart");
  };

  return (
    <div className="group relative rounded-lg
border bg-card p-3
                hover:shadow-lg transition
dark:border-zinc-800">
      <button onClick={() =>
toggleFavorite(product._id)}
            className="absolute top-2 right-2 z-10
rounded-full bg-white/80 p-2">
        <Heart className={isFavorite ? "fill-rose-
500 text-rose-500" : "text-zinc-400"} />
      </button>
      <Link
href={`/${lang}/product/${product.slug}`}>
        <Image src={product.image}
alt={product.name}
              width={300} height={300}
className="rounded" />
        <h3 className="mt-2 font-semibold line-
clamp-2">{product.name}</h3>
        <PriceView price={product.price}
discount={product.discount} />
      </Link>
      <button onClick={handleAddToCart}
            className="mt-3 w-full rounded bg-
primary px-3 py-2 text-white
                  flex items-center justify-center
gap-2">
        <ShoppingCart size={16} /> Add to Cart
      </button>
    </div>
  );
}
```