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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка інтернет-магазину з продажу жалюзі»

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

_____ Олег КОВАЛЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Олег КОВАЛЕНКО

Керівник: Наталя АНДРУСЕВИЧ
ст.викладач

Рецензент: _____

Київ 2025

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
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Коваленку Олегу Михайловичу

1. Тема кваліфікаційної роботи: «Розробка інтернет-магазину з продажу жалюзі» керівник кваліфікаційної роботи старший викладач кафедри ІПЗ Наталя АНДРУСЕВИЧ, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки вебзастосунків, опис роботи вебдодатків, документація фреймворків NestJS та Nuxt.JS.

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

1. Огляд та аналіз існуючих методів та технологій розробки інтернет-магазинів.
2. Проектування вебзастосунку для автоматизованої роботи інтерент-магазину з продажу жалюзі.
3. Програмна реалізація та опис функціонування застосунку для інтернет-магазину з продажу жалюзі.

4. Тестування застосунку.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Архітектура вебзастосунку.
5. Діаграма варіантів використання.
6. Схема бази даних.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд та аналіз засобів реалізації застосунку	14.03-18.03.2025	
4	Проектування застосунку	19.03-23.03.2025	
5	Програмна реалізація застосунку	23.03-13.04.2025	
6	Тестування застосунку	14.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Олег КОВАЛЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Наталя АНДРУСЕВИЧ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 43 стор., 1 табл., 21 рис., 12 джерел.

Мета роботи – автоматизація процесу продажу жалюзі через вебзастосунок.

Об'єкт дослідження – процес розробки вебплатформи для продажу товарів.

Предмет дослідження – засоби та технології розробки інтернет-магазинів для продажу жалюзі.

Короткий зміст роботи: В роботі проаналізовано предметну галузь та методи реалізації застосунку для продажу жалюзі. Проведено огляд вже існуючих засобів для продажу товарів в інтернет-магазині. Розроблено вебсайт та програмно реалізовані основні функціональні можливості, такі як: авторизація, створення товарів, та замовлень, написання коментарів, система пошуку товарів та фільтрації, швидкий інтерфейс. В роботі використано мову програмування JavaScript, фреймворки NestJS та Nuxt.JS для розробки бекенду та фронтенду відповідно, та PostgreSQL для зберігання даних.

Сферою використання застосунку є електронна комерція, що займається продажем товарів через інтернет, таких як жалюзі.

КЛЮЧОВІ СЛОВА: JAVASCRIPT, NESTJS, NUXT.JS, POSTGRESQL, ВЕБДОДАТОК,

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

IDE – Integrated Development Environment

SSR – Server Side Rendering

API – Application Programming Interface

SSG – Static Site Generation

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

SQL – Structured Query Language

ORM – Object-Relational Mapping

JWT – JSON Web Token

JSON – JavaScript Object Notation

CI/CD – Continuous Integration/Continuous Deployment

ВСТУП

Актуальність: У сучасному бізнесі все більше уваги приділяється автоматизації продажів і розвитку електронної комерції. Онлайн-торгівля жалюзі дає змогу компаніям розширити коло клієнтів поза межами локального ринку, скоротити витрати на утримання офлайн-точок і підвищити швидкість обробки замовлень. Для споживачів вебзастосунок із зручною навігацією та налаштуванням параметрів виробу забезпечує можливість вибору оптимального варіанту в будь-який час без необхідності особистого візиту до магазину. Саме тому розробка ефективного інтернет-магазину для продажу жалюзі є своєчасним і важливим завданням, яке сприятиме зростанню продажів і задоволеності клієнтів.

Об'єктом дослідження є процес розробки вебплатформи для продажу товарів.

Предметом дослідження є засоби та технології розробки інтернет-магазинів для продажу жалюзі.

Метою роботи є сприяння процесу продажу жалюзі за допомогою інтернет-магазину.

Методи дослідження: У роботі спочатку проведено детальний аналіз існуючих інтернет-магазинів жалюзі, при цьому виявлено, що більшість платформ створена на базі готових CMS-рішень (наприклад, WordPress, OpenCart, Shopify), це дало змогу відокремити ключові обмеження та переваги готових систем і сформулювати функціональні і нефункціональні вимоги до платформи. На їхній основі обрано стек технологій: NestJS для серверної логіки, Nuxt.JS для клієнтської частини, PostgreSQL для зберігання даних. Далі спроектовано схему бази даних із основними сутностями і їхніми зв'язками. Дослідження реалізації застосунку проводилося під час безпосередньої розробки.

Наукова новизна роботи полягає в створенні власного вебзастосунку для продажу жалюзі, який без використання готових CMS поєднує параметричний підбір товарів із миттєвим оновленням результатів пошуку, розширений механізм

фільтрації та пошуку, адаптований під особливості жалюзі, вбудовану систему коментарів і рейтингу для підвищення довіри користувачів та оптимізоване кешування даних для забезпечення швидкої роботи інтерфейсу навіть за високого навантаження.

Практична значущість результатів полягає у можливості впровадження розробленого вебзастосунку для повної автоматизації прийому і обробки замовлень жалюзі, управління каталогом продукції та взаємодії з клієнтами. Адаптивний інтерфейс забезпечує стабільну роботу на ПК, планшетах і смартфонах, що скорочує час обробки замовлень і підвищує якість обслуговування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО ВЕБЗАСТОСУНКУ ДЛЯ ПРОДАЖУ ЖАЛЮЗІ

1.1 Огляд ринку жалюзі та специфіка онлайн-торгівлі

Ринок жалюзі та інших віконних систем сьогодні демонструє стійку динаміку зростання[1], що обумовлено загальною тенденцією до підвищення комфорту житлових і офісних приміщень. Завдяки появі нових матеріалів і технологій виробництва, споживачі мають змогу обирати не лише між різними видовими рішеннями (горизонтальні й вертикальні жалюзі, ролети, штори–день-ніч), а й гнучко налаштовувати розміри, колір і фактуру полотна.

Онлайн-торгівля жалюзі становить близько третини від загального обсягу продажів віконних виробів: дедалі більше підприємств переходять на електронні майданчики, щоб уникнути витрат на орендовані приміщення й оптимізувати логістику. При цьому середня сума замовлення в інтернет-магазинах жалюзі вища, ніж у звичайних роздрібних точках, що пояснюється індивідуалізацією параметрів і додатковими додатками (керуючі системи, монтажні профілі, декоративні елементи).

Особливістю онлайн-торгівлі жалюзі є потреба в точності замірів і візуалізації кінцевого результату.

Ще одним важливим аспектом є швидка й зручна фільтрація: клієнти шукають товари за кількома критеріями одночасно — ширина, висота, матеріал (пластик, тканина, алюміній), тип кріплення, спосіб керування (ланцюжок, пульт тощо). Відповідно, архітектура каталогу та інтерфейс пошуку повинні забезпечувати миттєву видачу релевантних результатів навіть при великій кількості позицій.

Таким чином, поєднання високих вимог до точності замірів, візуалізації та швидкодії формує специфіку інтернет-магазинів жалюзі. Створення кастомної платформи, яка враховує ці особливості з нуля, а не покладається на готові CMS-

рішення, здатне забезпечити кращу продуктивність, гнучкість і комфорт для кінцевого користувача.

1.2 Цільова аудиторія

Цільова аудиторія вебзастосунку для продажу жалюзі охоплює кілька основних категорій користувачів, кожна з яких має власні потреби та очікування щодо функціональності й зручності інтерфейсу.

Першу категорію становлять приватні користувачі, які здійснюють покупки для власних потреб - вдома або в офісі. Для них важливо мати доступ до зрозумілого каталогу продукції, зручного фільтрування за розмірами, кольорами, типами жалюзі, а також до візуалізацій або прикладів встановлення. Такі користувачі цінують простоту оформлення замовлення, швидку доставку та можливість отримати консультацію.

Другою важливою групою є дизайнери інтер'єрів. Їм необхідний гнучкий інструмент для підбору жалюзі в рамках проєктів, що враховують кольорову гаму, матеріали й стиль приміщення.

Третьою категорією виступають невеликі монтажні компанії та будівельні бригади, які закуповують жалюзі оптом або під конкретні об'єкти. Для них важливо мати можливість швидко сформувати замовлення з декількох позицій, відстежити статус кожного, а також оформлювати рахунки-фактури та інші супровідні документи.

Визначення цільової аудиторії є ключовим етапом розробки вебзастосунку, оскільки саме воно впливає на логіку інтерфейсу, спосіб подання інформації, механізми пошуку, фільтрації, структуру каталогу та маркетингову стратегію. Кожна група користувачів має відображення у функціональних вимогах до платформи та визначає її користувацький сценарій.

1.3 Дослідження існуючих аналогів

У цьому розділі наведено детальний аналіз існуючих програмних засобів, які можна використовувати для створення інтернет-магазину з продажу жалюзі.

Метод дослідження є не тільки порівняння популярних CMS та SaaS-рішень, але й обґрунтування вибору оптимальної платформи з урахуванням технічних, організаційних і фінансових вимог малого бізнесу. Кожне рішення розглядається з точки зору того, наскільки воно відповідає конкретному об'єкту: відмінностями процесу обробки заявок без онлайн-оплати до потреби єдиного монолітного розгортання на стандартному VPS.

1.3.1 OpenCart

OpenCart є однією з найпопулярніших систем керування вмістом (CMS) з відкритим вихідним кодом, що призначена для створення інтернет-магазинів [2]. Платформа вирізняється простотою встановлення, модульною архітектурою та широкими можливостями для налаштування. Інтерфейс адміністративної частини є інтуїтивно зрозумілим і дозволяє швидко адаптувати систему під потреби бізнесу.

Серед ключових функціональних можливостей OpenCart варто виокремити наступні:

- підтримка багатомовності та мультивітринності, що дозволяє створювати магазини для різних мовних груп і регіонів;
- наявність великої кількості безкоштовних і платних розширень;
- модульна структура, яка забезпечує зручне додавання нових функцій.

В контексті реалізації вебзастосунку для продажу жалюзі OpenCart має ряд переваг:

- швидкий старт: можливість розгорнути повноцінний інтернет-магазин у стислі строки, базовий функціонал працює «з коробки»;
- гнучкість розширення: за допомогою додаткових модулів або кастомного плагіна можна реалізувати калькулятор вартості жалюзі;
- активна спільнота: велика кількість користувачів та розробників, доступна

документація, форуми, відеоуроки.

Однак, OpenCart має і певні обмеження, що варто врахувати під час розробки:

- зниження продуктивності при великих навантаженнях: у разі збільшення обсягу даних може виникнути потреба в додаткових інструментах кешування (наприклад, Redis або Varnish);
- складність кастомізації інтерфейсу: зміна нетипових макетів потребує ручного редагування шаблонів, що реалізовані на мові шаблонів Twig;
- якість сторонніх модулів: деякі додатки мають низький рівень підтримки або можуть викликати конфлікти з іншими плагінами, що ускладнює супровід системи.

Загалом, OpenCart є придатною платформою для створення магазину продажу жалюзі на початковому етапі, однак при масштабуванні та розширенні функціоналу може виникнути потреба у глибшій технічній оптимізації.

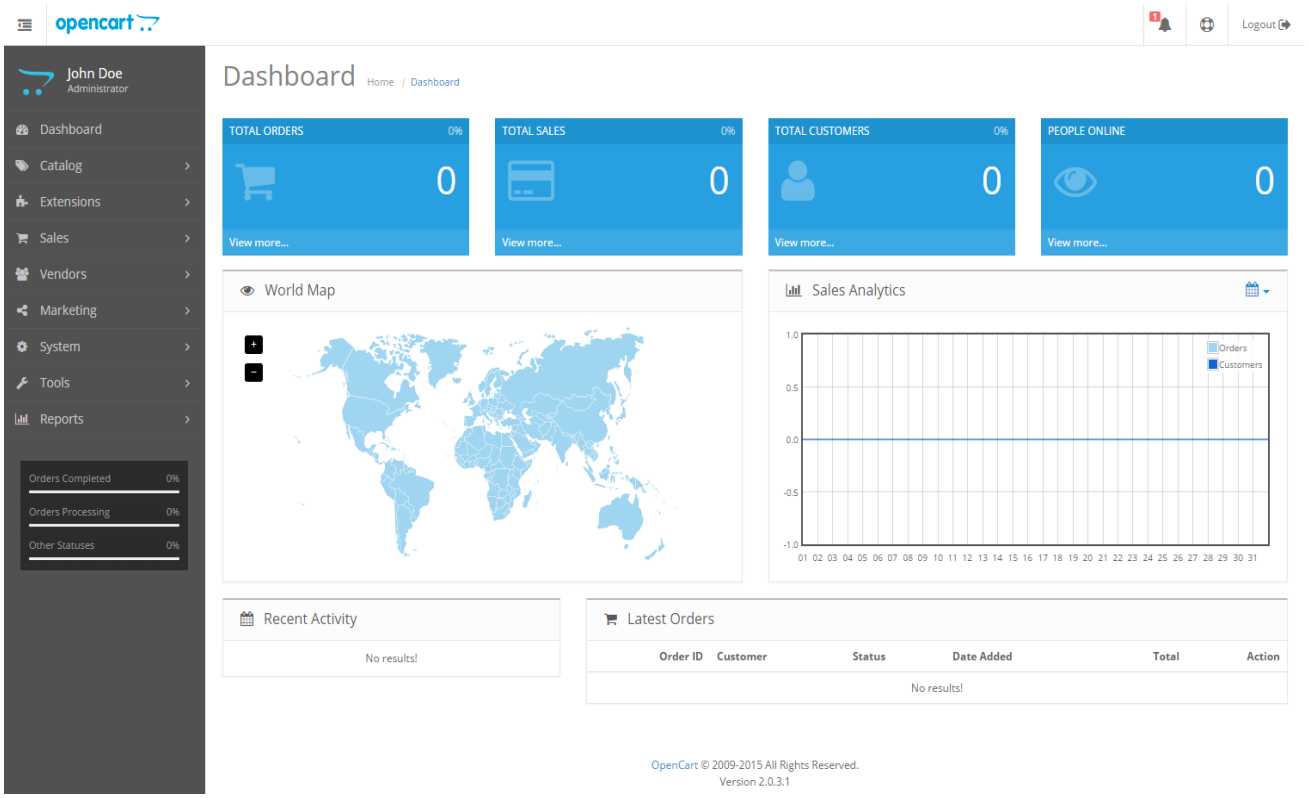


Рис. 1.1 Приклад інтерфейсу адміністратора OpenCart

1.3.2 Shopify

Shopify — це одна з найпопулярніших SaaS-платформ для створення інтернет-магазинів, що дозволяє запускати повноцінні онлайн-продажі без необхідності самостійного розгортання та обслуговування серверної інфраструктури [3]. Уся система працює в хмарному середовищі, а користувач отримує доступ до готової екосистеми інструментів для керування товарами, обробки замовлень, прийому платежів і налаштування зовнішнього вигляду магазину. Зручний інтерфейс адміністративної панелі Shopify.

Shopify підтримує понад сотню шаблонів дизайну, більшість з яких адаптовані під мобільні пристрої, що особливо актуально з огляду на тенденцію збільшення мобільного трафіку в електронній комерції. Платформа має багатомовний інтерфейс, підтримує різні валюти та інтеграції з популярними платіжними сервісами, що робить її універсальним рішенням для бізнесу різного масштабу.

Основні переваги Shopify:

- швидкий запуск: створення інтернет-магазину можливе за кілька годин навіть без технічних знань, завдяки покроковому майстру налаштувань та широкому вибору шаблонів;
- надійність і безпека - Shopify автоматично оновлює платформу, гарантує захист даних користувачів, сертифікацію PCI DSS та забезпечує безперебійність роботи;
- інтеграції та розширення - понад 3000 застосунків у фірмовому App Store дозволяють інтегрувати CRM, аналітику, логістику, email-маркетинг, програми лояльності тощо;
- технічна підтримка - доступна цілодобово через чат, електронну пошту або телефон, що є важливою перевагою у разі виникнення технічних проблем.

Проте при розгляді платформи слід враховувати і її обмеження:

- обмежена кастомізація: значні зміни в логіці або зовнішньому вигляді магазину вимагають знань Liquid: шаблонної мови програмування, яку

використовує Shopify;

- незалежність від хостингу: оскільки платформа повністю управляється Shopify, неможливо змінити серверне середовище чи оптимізувати його вручну для конкретних бізнес-задач.

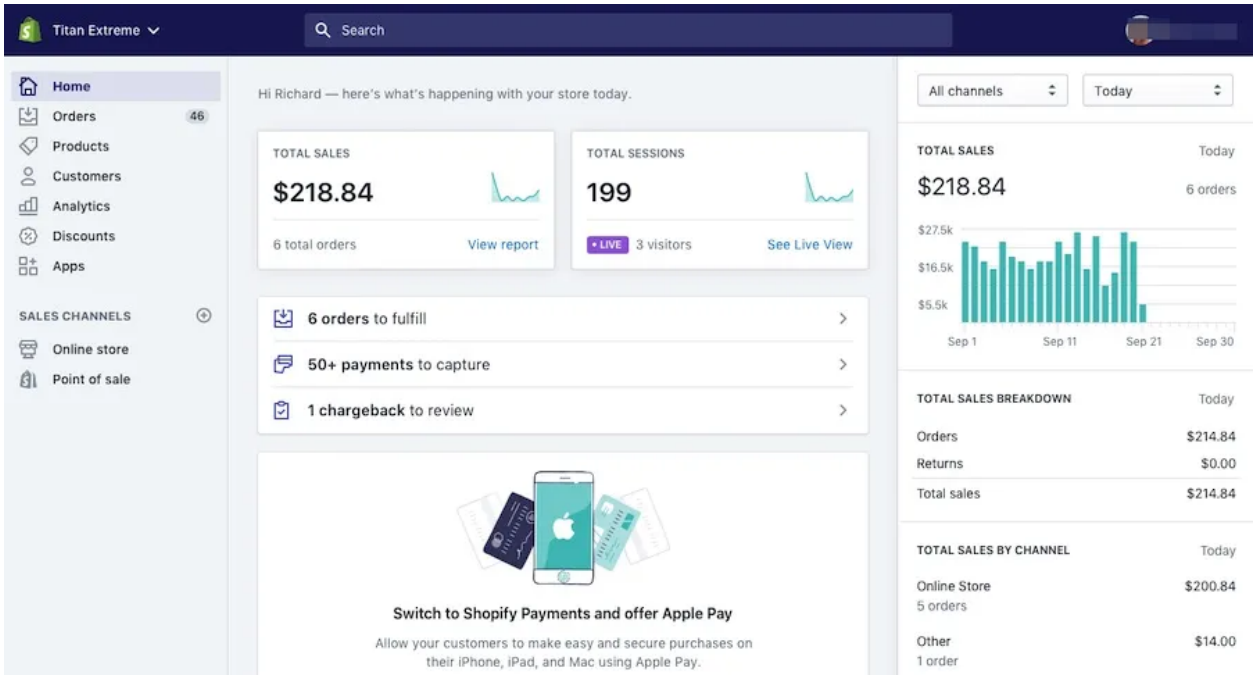


Рис. 1.2 Приклад адмін-панелі Shopify

1.3.3 WordPress (WooCommerce)

WordPress у поєднанні з плагіном WooCommerce є одним із найпопулярніших рішень для створення інтернет-магазинів малого та середнього масштабу [4]. Система дозволяє швидко розгорнути магазин на базі звичайного сайту WordPress, що дає змогу об’єднати контентну та торговельну частини.

Переваги WooCommerce:

- велика гнучкість: можливість повністю змінювати структуру, вигляд і логіку роботи магазину;
- велика кількість тем і плагінів: доступні як безкоштовні, так і платні розширення, що значно пришвидшують розробку;
- відкритий код: дозволяє повністю контролювати систему та модифікувати

її відповідно до потреб проєкту.

Це рішення є доцільним для реалізації магазину жалюзі у разі потреби у глибокій кастомізації або поєднання з блогом чи інформаційним сайтом. WooCommerce підтримує численні платіжні шлюзи, системи доставки та SEO-інструменти.

До недоліків належать:

- потреба в постійному обслуговуванні: регулярне оновлення ядра, плагінів і тем для підтримки безпеки та стабільної роботи;
- навантаження на хостинг: при збільшенні кількості товарів або відвідувачів сайт може потребувати потужнішого серверного середовища;
- конфлікти плагінів: інколи встановлення великої кількості розширень може призводити до несумісностей або зниження швидкодії.

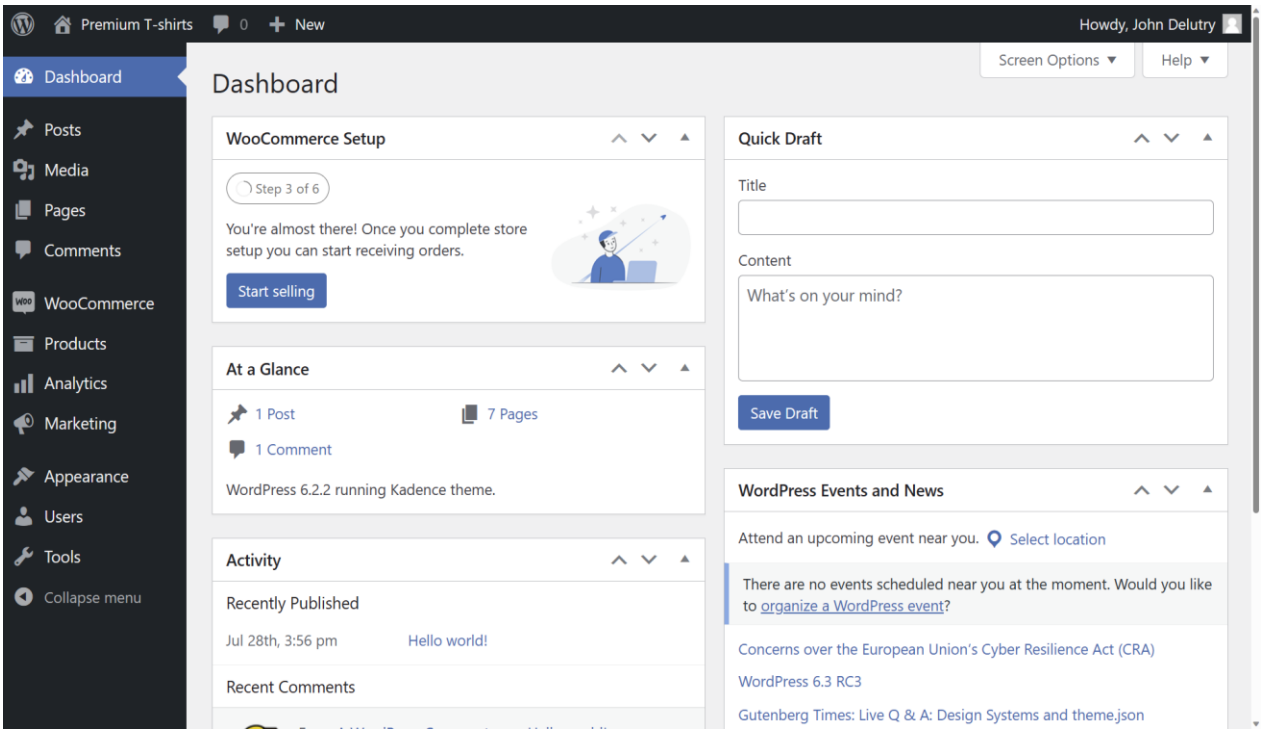


Рис. 1.3 Приклад адмін-панелі

1.3.4 BlindWave

Вибір власної платформи на базі NestJS та Next.js дає змогу створити інтернет-магазин жалюзів, ідеально адаптований до ваших бізнес-процесів, без

зайвих обмежень готових CMS чи SaaS-рішень. Також отримуєте зручний інтерфейс котрий увібрав все найкраще із вищезазначених систем зображено.

Ключові особливості:

- монолітна архітектура: весь код лежить в одному репозиторії, що суттєво спрощує розгортання та підтримку;
- серверний рендеринг і статична генерація: Next.js забезпечує швидке завантаження сторінок та відмінне SEO без додаткових налаштувань;
- модульна структура Nest JS: Чіткий поділ на модулі: каталог товарів, обробка заявок, автентифікація, адміністрування і логування

Переваги для проєкту:

- проста форма заявки: користувач обирає тип жалюзів, колір і розміри, зазначає свої контакти - і заявка миттєво надходить менеджеру;
- гнучкий дизайн і UX: жодних обмежень у макетах: будь-які інтерактивні елементи, анімації або спецефекти реалізуються без проблем;
- швидка інтеграція з CRM і месенджерами - вбудовані REST/GraphQL API дозволяють підключити Bitrix24, Telegram-бота, Google Sheets або будь-який інший сервіс;
- висока продуктивність: оптимізований код і час відповіді сервера ≤ 500 мс, що забезпечує плавну роботу навіть під навантаженням.

Недоліки та обмеження:

- необхідність фахівця: потрібен розробник з досвідом для створення і подальшої підтримки;
- повна відповідальність: вся інфраструктура лежить на вашій відповідальності;
- відсутність готових плагінів: кожна додаткова інтеграція та функція потребує власної розробки.

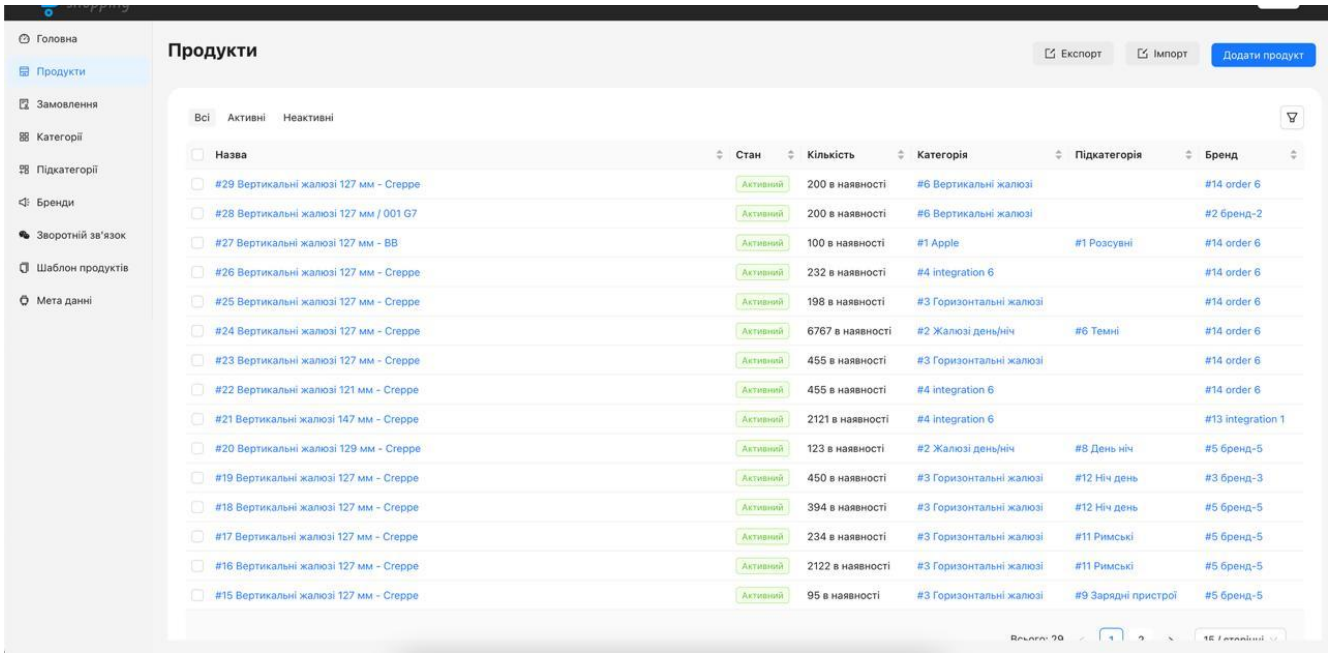


Рис. 1.4 Приклад адмін-панелі

Зведені результату аналізу характеристик розглянутих додатків наведено в таблиці 1.1.

Таблиця 1.1

Зведена таблиця аналізу існуючих застосунків

Показник	**OpenCart**	**Shopify**	**WooCommerce**	**BlindWave**
Гнучкість у зміні логіки роботи	Обмежена CMS	Мінімальна	Середня	Максимальна
Адмін-панель	проста, не завжди інтуїтивна	Зручна, сучасна	Залежить від WP теми	Реалізована під потреби замовника
Швидкість роботи	Залежить від кількості модулів	Залежить від кількості модулів	Багато зайвих функцій	Оптимізована під конкретний проєкт

Продовження таблиці 1.1

Зведена таблиця аналізу існуючих застосунків

Показники	**OpenCart**	**Shopify**	**WooCommerce**	**BlindWave**
Кастомізація дизайну	Залежить від шаблонів	Мінімальна	Відносно гнучка	свобода дизайну
Залежність від сторонніх сервісів	Висока	Повна залежність	Часткова	Без залежностей

1.4 Визначення функціональних і нефункціональних вимог

На основі аналізу предметної області, дослідження цільової аудиторії та вивчення існуючих рішень, було сформульовано перелік функціональних та нефункціональних вимог до вебзастосунку для продажу жалюзі. Їхнє чітке визначення є необхідним етапом, що забезпечує успішне проєктування, реалізацію та тестування програмного продукту.

1.4.1 Функціональні вимог

Функціональні вимоги описують основні можливості системи з точки зору взаємодії користувача з нею. До таких вимог належать:

- реєстрація та авторизація: реалізація системи реєстрації та входу в обліковий запис, що передбачає авторизацію адміністратора для керування контентом;
- пошук товарів: можливість здійснювати пошук товарів за ключовими словами
- фільтрація товарів: впровадження фільтрів за категоріями, типом, та іншими параметрами продукції;
- оформлення замовлення: додавання товарів у кошик, вказання контактної інформації, та підтвердження замовлення;

1.4.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики вебзастосунку та обмеження, що накладаються на його роботу. Основні з них включають:

- безпека: забезпечення конфіденційності та цілісності персональних даних користувачів за допомогою шифрування, аутентифікації та налаштувань прав доступу;
- продуктивність: час завантаження сторінки повинен бути не більшим за 3 секунди при нормальному навантаженні;
- адаптивність: інтерфейс вебзастосунку має бути оптимізованим для коректного відображення на різних типах пристроїв: комп’ютерах, планшетах, смартфонах.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Середовище розробки

Інтегроване середовище розробки (IDE) – це комплексне програмне забезпечення, що надає програмістам інструменти для створення, редагування, компіляції, налагодження та тестування програмного коду. IDE зазвичай включає текстовий редактор, засоби для автоматизації збірки, інтегровані інструменти налагодження (debugger), а також підтримку систем керування версіями, що дозволяє розробникам ефективно організовувати процес розробки в одному середовищі.

У процесі створення вебзастосунку для онлайн-продажу жалюзі було використано два основні інструменти: Sublime Text та WebStorm. Обидва середовища мають свої переваги, що зумовило їх використання на різних етапах розробки проєкту.

2.1.1 Sublime Text

Sublime Text — це легкий, швидкий та високопродуктивний текстовий редактор, який широко застосовується розробниками завдяки своїй простоті, гнучкості та підтримці великої кількості мов програмування [5]. Редактор має багату екосистему плагінів, які дозволяють розширювати його функціональність до рівня повноцінного середовища розробки.

Серед ключових переваг Sublime Text можна виділити: швидкий запуск, мінімалістичний інтерфейс, підтримку кількох курсорів для одночасного редагування, зручну навігацію по проєкту, а також можливість налаштування через JSON-конфігурації. Крім того, Sublime Text підтримує інтеграцію з Git, системами збірки та автоматичного тестування.

Для проєкту було використано Sublime Text на початкових етапах розробки для швидкого створення шаблонів, написання HTML/CSS/JavaScript коду та базових скриптів.

Його легкість і стабільність дозволили зосередитися на логіці програми без потреби в ресурсоємному середовищі.

2.1.2 WebStorm

WebStorm — це повнофункціональне інтегроване середовище розробки від компанії JetBrains, спеціалізоване на JavaScript і сучасній веброзробці [6]. Воно надає повну підтримку таких технологій, як HTML, CSS, JavaScript, TypeScript, а також фреймворків Angular, React, Vue.js тощо.

Однією з головних переваг WebStorm є потужні інструменти для аналізу коду, рефакторингу, автоматичного доповнення, а також інтегрований інструмент налагодження. WebStorm дозволяє ефективно працювати з проєктами Node.js, REST API, а також має підтримку роботи з базами даних і системами контролю версій (зокрема Git).

Інтерфейс WebStorm забезпечує комфортну навігацію, підтримку шаблонів коду, конфігурацій запуску та тестування, що сприяє зменшенню часу на реалізацію функціоналу та покращенню якості коду. Крім того, WebStorm має гнучку систему налаштувань, яка дозволяє адаптувати середовище до конкретних потреб розробника.

Використання WebStorm було доцільним на етапі розробки клієнтської частини застосунку з використанням Angular, що дозволило досягти високої якості реалізації інтерфейсу користувача.

2.2 Мови програмування

Мова програмування — це формальна система правил і символів, що використовується для створення програмного коду. Вона служить засобом спілкування між людиною і комп’ютером, дозволяючи описувати логіку роботи програм і керувати обробкою даних. Вибір мови програмування залежить від характеру проєкту, цілей розробки та особливостей платформи.

У сучасній веброзробці особливе значення мають мови, що дозволяють створювати як інтерфейс користувача, так і логіку роботи застосунку. В даній роботі основними мовами програмування є JavaScript і TypeScript, які широко застосовуються для створення інтерактивних веб застосунків з високим рівнем надійності та підтримки.

2.2.1 JavaScript

JavaScript є динамічною, інтерпретованою мовою програмування, яка вперше була розроблена для створення інтерактивних елементів у веббраузерах. З того часу вона розвинулася в потужний інструмент, який підтримує широке коло можливостей, включаючи роботу з DOM, обробку подій, асинхронне програмування та роботу з мережевими запитами.

Виконання коду JavaScript безпосередньо у браузері дозволяє створювати динамічні та адаптивні інтерфейси, що підвищує зручність використання веб застосунків. Крім того, JavaScript може застосовуватись і на серверній стороні, завдяки таким платформам, як Node.js, що робить його універсальною мовою для повного циклу розробки.

JavaScript підтримує різні парадигми програмування, зокрема імперативну, об’єктно-орієнтовану та функціональну, що дає розробникам гнучкість у виборі стилю кодування відповідно до вимог проєкту.

2.2.2 TypeScript

TypeScript - це мова програмування, створена компанією Microsoft як надбудова над JavaScript. Вона розширює JavaScript, додаючи строгий статичний контроль типів, що суттєво знижує ймовірність помилок на етапі розробки [7].

Основною перевагою TypeScript є можливість явно вказувати типи змінних, параметрів функцій та їхніх результатів, що підвищує надійність і підтримуваність коду. Це особливо важливо в масштабних проєктах, де кількість рядків коду і кількість розробників можуть бути великими.

TypeScript компілюється у стандартний JavaScript, що гарантує сумісність із усіма браузерами та середовищами, де виконується JavaScript. Це дає змогу використовувати всі переваги сучасної типізованої мови, не втрачаючи сумісності з існуючою інфраструктурою.

Використання TypeScript є рекомендованим підходом у багатьох сучасних вебфреймворках, таких як Angular, React та Vue.js (починаючи з версії 3), що сприяє підвищенню якості розробки і полегшує подальшу підтримку програмного забезпечення.

2.3 Вебфреймворки

Вебфреймворки — це програмні комплекси, які надають стандартизовану структуру для розробки вебзастосунків. Вони спрощують створення складної логіки, полегшують управління проектом і пришвидшують розробку завдяки готовим рішенням для типових задач, таких як маршрутизація, робота з базою даних, автентифікація та інше.

У рамках розробки даного застосунку були використані два сучасні вебфреймворки — NestJS для бекенд-частини та NuxtJS для фронтенду.

2.3.1 NestJS

NestJS — це прогресивний бекенд-фреймворк для Node.js [20], який побудований на TypeScript і використовує архітектурні підходи, притаманні Angular, такі як ін’єкція залежностей, модулі, контролери та провайдери [9]. Це дозволяє створювати масштабовані, підтримувані та гнучкі серверні додатки.

Основною перевагою NestJS є його модульність та підтримка різних бібліотек і технологій, зокрема Express або Fastify в якості HTTP-сервера. Фреймворк має вбудовану підтримку для роботи з базами даних через ORM (TypeORM, Sequelize), а також зручний механізм для обробки запитів, маршрутизації, валідації і аутентифікації.

NestJS підходить для розробки як невеликих сервісів, так і складних

корпоративних систем, що робить його ідеальним вибором для проєктів з вимогами масштабованості та безпеки.

2.3.2 NuxtJS

NuxtJS - це фреймворк на основі Vue.js, який спрощує розробку універсальних (server-side rendered) вебзастосунків, а також SPA (single-page application). Nuxt забезпечує автоматичну маршрутизацію, оптимізацію продуктивності та генерацію статичних сайтів [8].

Основна особливість Nuxt полягає у здатності рендерити сторінки на сервері, що покращує SEO, швидкість завантаження і користувацький досвід. Фреймворк підтримує розширення, модулі, а також інтеграцію з різними API та бекенд-сервісами.

Використання NuxtJS дозволяє розробникам зосередитись на логіці додатку, уникаючи ручного налаштування складних аспектів конфігурації, завдяки чому збільшується продуктивність розробки.

2.4 Система управління базою даних

Система управління базою даних (СУБД) — це програмне забезпечення, яке забезпечує зберігання, організацію та управління даними в інформаційних системах.

У цьому проєкті використовується PostgreSQL — об'єктно-реляційна система управління базами даних з відкритим вихідним кодом. Вона забезпечує:

- підтримку стандарту ACID (атомарність, узгодженість, ізоляція, довговічність), що гарантує надійність транзакцій;
- розширений набір функцій SQL, включно зі складними запитами, зовнішніми ключами, тригерами та користувацькими типами даних;
- підтримку формату JSON для зручної роботи з напівструктурованими даними, що важливо для вебзастосунків.

PostgreSQL є надійною, масштабованою та продуктивною СУБД, що

дозволяє ефективно управляти великими обсягами даних і забезпечувати їх цілісність.

2.5 Система контролю версій

GitHub — це вебплатформа для хостингу коду, яка базується на системі контролю версій Git [10]. GitHub швидко став однією з найбільш впливових платформ у світі розробки програмного забезпечення, забезпечуючи інструменти для спільної роботи, контролю версій та розподіленого доступу до коду.
Центральним елементом GitHub є репозиторії, де зберігається програмний код.

Користувачі можуть створювати публічні або приватні репозиторії для своїх проєктів, у яких реалізовано управління версіями за допомогою Git.

GitHub надає можливість "форкати" (створювати копії) чужих репозиторіїв, вносити зміни та пропонувати ці зміни авторам оригінальних репозиторіїв через пул-реквести (pull request). Ця функція є ключовою для співпраці над відкритими проєктами.

Завдяки GitHub розробники можуть ефективно контролювати версії, відстежувати зміни, обговорювати правки та спільно працювати над кодом, що значно підвищує якість і швидкість розробки програмних продуктів.

3 ПРОЄКТУВАННЯ

3.1 Проєктування архітектури застосунку

Додаток реалізований за моделлю клієнт-сервер, що забезпечує розподіл обробки даних між клієнтською та серверною частинами. Таке розділення дозволяє оптимізувати продуктивність системи, підвищити масштабованість і спростити підтримку.

Клієнтська частина розроблена з використанням фреймворку Nuxt.js, що базується на Vue.js [11]. Вона відповідає за взаємодію з користувачем через інтуїтивний графічний інтерфейс. Комунікація з сервером здійснюється через HTTP(S) протокол, з обміном даних у форматі JSON. Використання асинхронних запитів дозволяє оновлювати вміст сторінок без повного перезавантаження, забезпечуючи плавний і зручний користувацький досвід.

Серверна частина побудована на основі фреймворку NestJS — сучасного прогресивного Node.js фреймворку, який підтримує модульну структуру та використовує TypeScript. Сервер приймає запити від клієнта, обробляє бізнес-логіку, взаємодіє з базою даних PostgreSQL і повертає необхідні відповіді. NestJS забезпечує масштабованість та зручність тестування коду, а також підтримує аутентифікацію і авторизацію користувачів, захист даних і безпеку транзакцій.

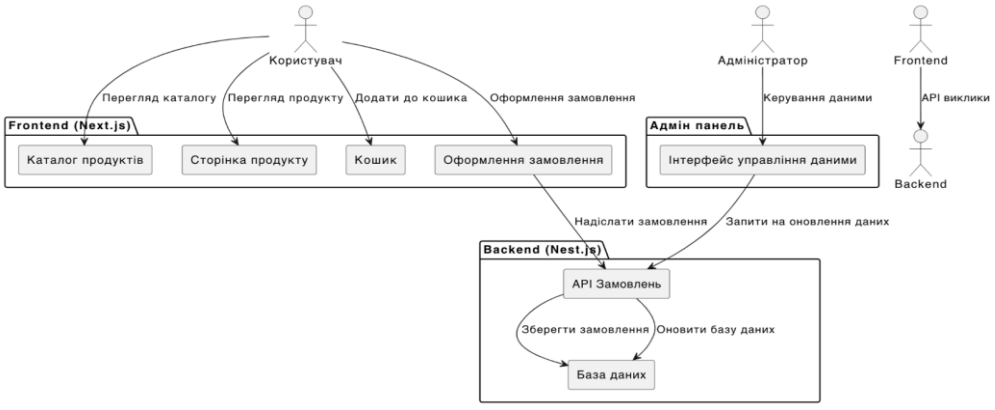


Рис. 3.1 Діаграма розгортання клієнт-серверної моделі

3.2 Архітектура клієнтської частини

Структура Nuxt.js-проєкту організована у три основні частини: компоненти, сторінки та плагіни. Такий підхід забезпечує чітке розділення відповідальностей між різними частинами додатку, що сприяє кращій організації коду та полегшує його підтримку, масштабування і тестування.

Компоненти у Nuxt.js є повторно використовуваними будівельними блоками інтерфейсу, що містять логіку управління і шаблони для відображення користувацького інтерфейсу. Кожен компонент зазвичай має власний Vue-файл, де описано HTML-структуру, стилі та JavaScript-код [12]. Компоненти можуть включати кнопки, форми, модальні вікна тощо, які застосовуються у різних частинах додатку.

Сторінки формують основні маршрути (routes) додатку і складаються з груп компонентів, що разом утворюють повноцінні екранні відображення.

Наприклад, сторінка профілю користувача може містити компоненти інформації про користувача, налаштувань і історії активності. Сторінки відповідають за навігацію в додатку і забезпечують структуру користувацького досвіду.

Плагіни в Nuxt.js використовуються для підключення зовнішніх бібліотек та сервісів, а також для глобальної ініціалізації функціоналу, наприклад, налаштування HTTP-клієнта або авторизації. Вони забезпечують централізовану інтеграцію функцій, які повторно використовуються у всьому додатку.

Такий підхід до організації клієнтської частини забезпечує високу модульність, зручність у розвитку і підтримці коду, а також покращує масштабованість додатку в майбутньому.

3.3 Архітектура серверної частини

Серверна частина програмного забезпечення реалізована із використанням сучасного серверного фреймворку NestJS, що працює поверх платформи Node.js та побудований за принципами архітектури мікросервісів і інверсії управління (IoC).

NestJS забезпечує масштабованість, модульність і підтримку TypeScript, що значно підвищує безпеку типів та стабільність коду.

Серверна логіка організована у вигляді модулів (modules), які містять у собі пов’язані компоненти: контролери, сервіси та схеми моделей. Кожен модуль відповідає за окрему функціональність системи, зокрема: обробку запитів користувачів, аутентифікацію, керування даними тощо. Така організація сприяє розділенню відповідальностей, підвищує читабельність коду та полегшує тестування [13].

Комунікація клієнта з сервером здійснюється за протоколом HTTP, з дотриманням принципів REST-архітектури. Контролери відповідають за обробку вхідних HTTP-запитів та викликають відповідні методи сервісів. Сервіси реалізують бізнес-логіку застосунку та взаємодіють з базою даних. Взаємодія з базою даних PostgreSQL реалізується через об’єктно-реляційний мапер (ORM) TypeORM, який забезпечує зручну роботу з транзакціями, зв’язками між таблицями та валідацією даних.

Сервер підтримує аутентифікацію та авторизацію користувачів, реалізовану за допомогою JWT (JSON Web Token). Токен використовується для зберігання інформації про користувача та передається у кожному запиті для забезпечення безпеки доступу до захищених ресурсів [14].

Для забезпечення стабільної роботи та обробки виняткових ситуацій реалізовано систему глобального оброблення помилок, яка забезпечує стандартизовані відповіді при виникненні помилок або винятків. Також реалізовано middleware-компоненти, які забезпечують логування, парсинг запитів та перевірку прав доступу.

Додатково сервер інтегрується з зовнішніми API при потребі (наприклад, сервіси сторонніх постачальників), використовуючи HTTP або WebSocket протоколи. Усі запити фіксуються у журналах, що дозволяє забезпечити аудит дій користувачів і адміністраторів.

Завдяки застосуванню NestJS як основи серверної частини досягається висока модульність та масштабованість проєкту. Архітектурний підхід, що

базується на модулях та сервісах, забезпечує гнучкість системи, можливість її розвитку та подальшого інтегрування з іншими сервісами без потреби у кардинальній перебудові системи.

3.4 Архітектура бази даних

Архітектура бази даних розробленого проєкту базується на реляційній моделі, реалізованій за допомогою системи управління базами даних PostgreSQL. Використання реляційної моделі дозволяє забезпечити ефективне зберігання, організацію та керування великими обсягами структурованої інформації.
База даних складається з таблиць, які відображають сутності предметної області проєкту. Таблиці містять колонки (поля) та рядки (записи):

- колонки визначають тип даних, які можуть зберігатися (рядкові, числові, датовані тощо). У рамках NestJS опис структури таблиць реалізовано за допомогою ORM-бібліотеки TypeORM, яка відображає таблиці у вигляді класів, а колонки – у вигляді атрибутів класів;
- рядки містять конкретні екземпляри сутностей, наприклад, дані про користувачів, курси або результати тестів.

Для керування даними застосовується мова структурованих запитів SQL (Structured Query Language), що забезпечує основні операції зі створення, читання, оновлення та видалення інформації (CREATE, READ, UPDATE, DELETE). Використання ORM дозволяє виконувати ці операції на рівні об’єктів, що спрощує розробку і підтримку коду.

В реляційній моделі встановлено зв’язки між таблицями, які класифікуються за типом:

- «один до одного» (One-to-One), коли одному запису в одній таблиці відповідає один запис в іншій;
- «один до багатьох» (One-to-Many), що передбачає зв’язок одного запису з багатьма іншими;

- «багато до багатьох» (Many-to-Many), який реалізується через додаткові проміжні таблиці для збереження множинних зв’язків.

До основних переваг реляційної архітектури бази даних належать:

- забезпечення цілісності даних через встановлення обмежень (первинні ключі, унікальність, зовнішні ключі), що гарантує достовірність і послідовність інформації [15];
- можливість виконання складних аналітичних запитів за допомогою SQL, що сприяє гнучкому доступу до даних.

Запропонована архітектура (див. рисунок 3.2) відповідає вимогам продуктивності, масштабованості та надійності, що забезпечує стабільну роботу вебдодатку з великою кількістю користувачів і обсягом інформації.

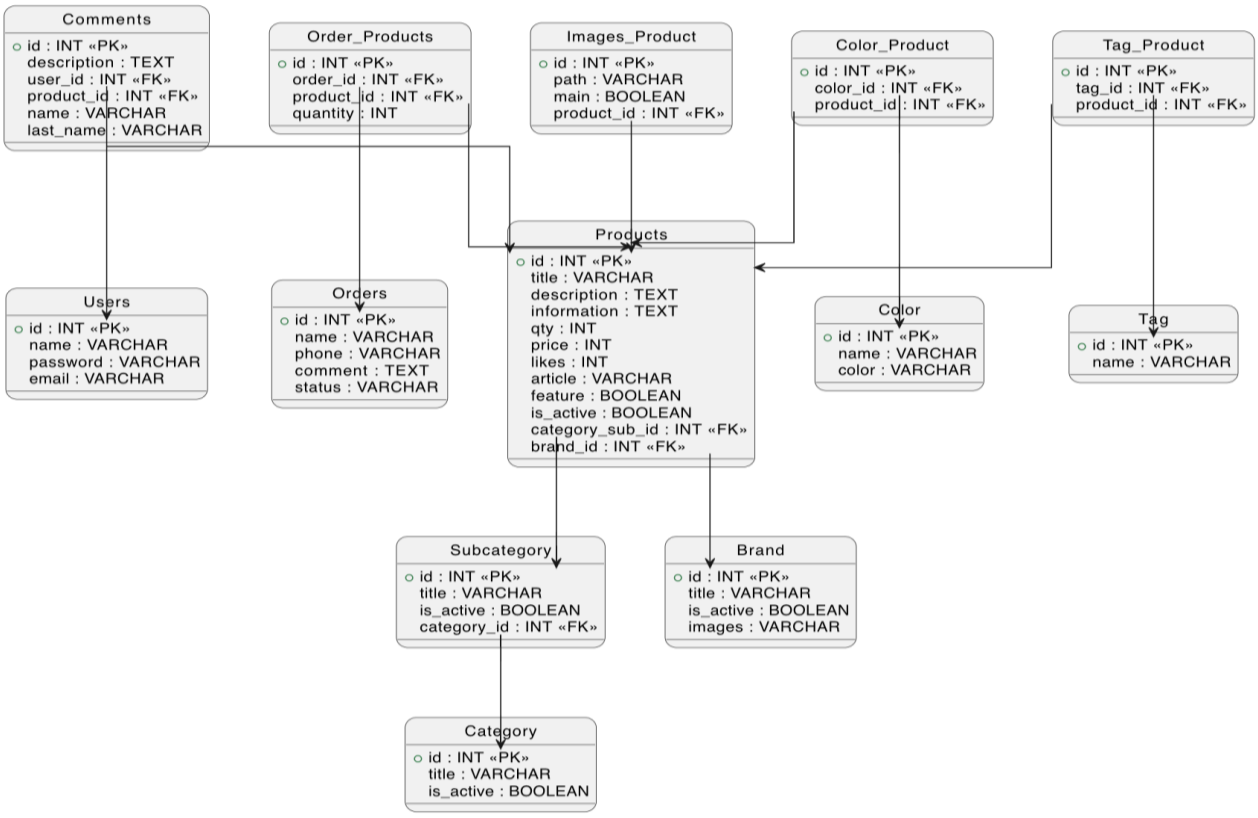


Рис. 3.2 Діаграма класів бази даних

Для зображення взаємодії користувача з системою була розроблена діаграма варіантів використання (див. рисунок 3.3).

Рис. 3.3 Діаграма варіантів використання

4 РЕАЛІЗАЦІЯ

4.1 Реалізація клієнтської частини

Для реалізації клієнтської частини проєкту використовувався фреймворк Nuxt.js, який базується на Vue.js і призначений для розробки універсальних вебдодатків.

Розробка клієнтської частини включала кілька етапів.

Першим кроком була ініціалізація нового проекту Nuxt.js із використанням офіційного шаблону. Для цього у командному рядку було виконано команду:

- npx create-nuxt-app frontend

Внаслідок цього створюється стандартна структура каталогів і файлів, що забезпечує основу для подальшої розробки (див. рисунок 4.1).

Основні каталоги і файли, які використовуються у проекті, наведено нижче:

- pages: каталог, що містить файли сторінок, які автоматично відповідають маршрутам вебдодатку;
- components: каталог для зберігання повторно використовуваних Vue-компонентів;
- assets: каталог для статичних ресурсів, таких як стилі та зображення;
- plugins: каталог для підключення сторонніх бібліотек і плагінів;
- store: каталог для управління станом додатку.

Другим кроком було створення компонентів, які є базовими елементами інтерфейсу користувача. Компоненти в Nuxt.js оформлюються у вигляді .vue файлів, які містять три основні складові: шаблон (template), скрипт (script) і стилі (style). Для кожного компонента описуються його структура, логіка та оформлення, що забезпечує повторне використання та підтримку коду (див. рисунок 4.2).

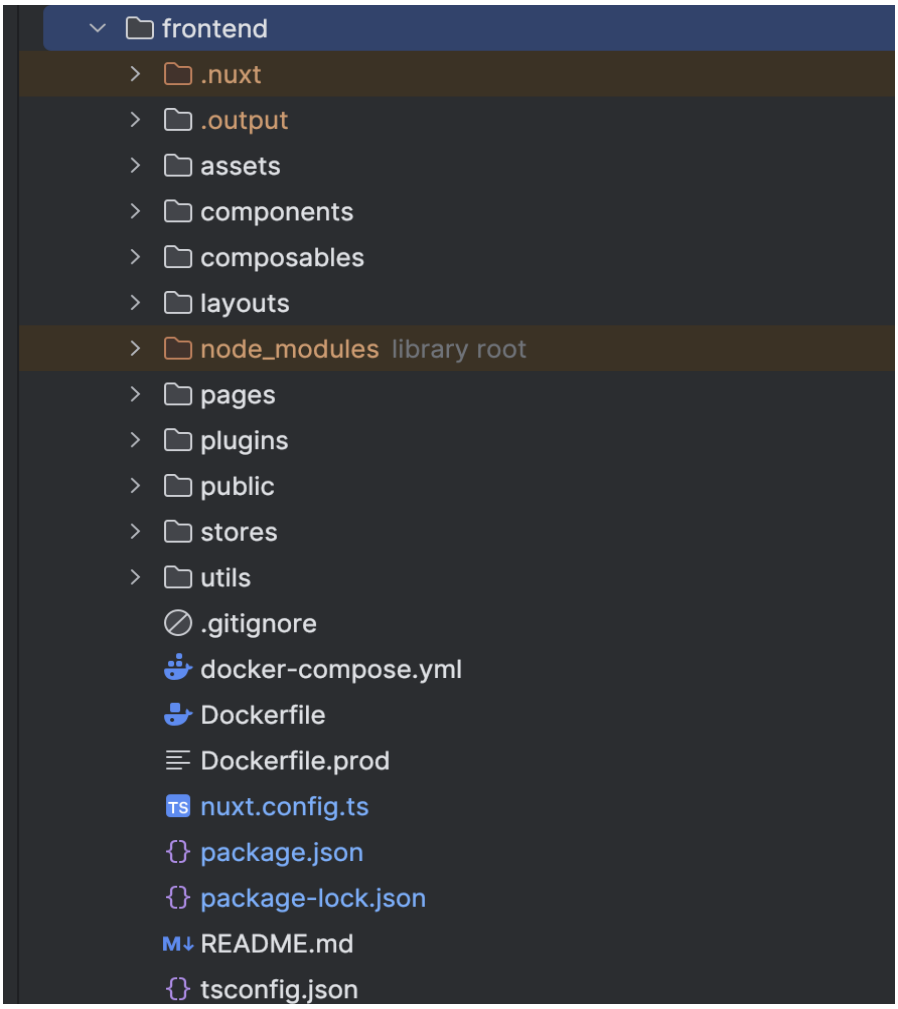


Рис. 4.1 Приклад структури Nuxt.js проєкту після ініціалізації

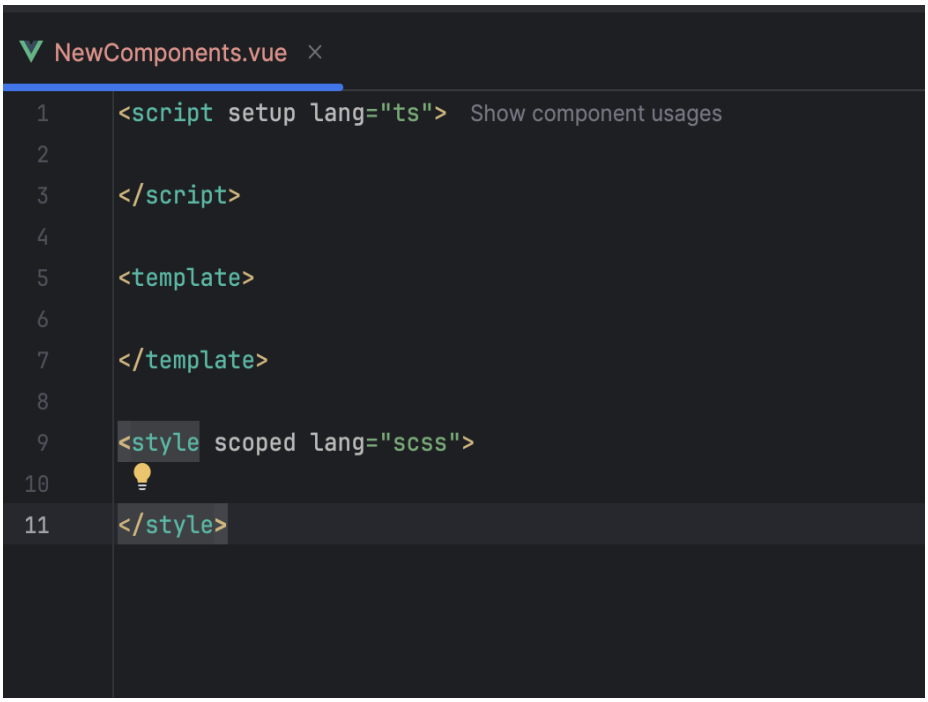


Рис. 4.2 Приклад створеного компоненту

Третім етапом стало впровадження механізму взаємодії клієнтської частини із сервером через REST API.

Для обміну даними з сервером використовувався axios, що дозволяє виконувати HTTP-запити до сервера. Використання axios дозволяє реалізувати основні CRUD-операції (створення, читання, оновлення, видалення) для обробки даних на стороні клієнта. Завдяки axios ми можемо налаштувати baseURL та конфігурації та створити hook який потім можемо використовувати по всій системі (див. рисунок 4.3, 4.4).

useApi.ts

```
1   import axios from 'axios'
2
3   export const useApi = () => {  Show usages  Deadshotgo
4       const config : RuntimeConfig  = useRuntimeConfig()
5       const baseURL : string  = config.public.apiBase
6       return axios.create({
7           baseURL,
8           headers: {
9               "Content-Type": 'application/json',
10              Accept:  'application/json',
11          }
12      })
13  }
```

Рис. 4.3 Приклад створеного hook

```
async getActionProducts(obj?: object) {
    const api = useApi()
    const filters = createFilterObject(obj)
    const { data } = await api.get( url: `products?${filters}`);
    this.products = data
    return data
},
```

Рис. 4.4 Приклад використання

Четвертим етапом було налаштування маршрутизації.

У Nuxt.js маршрути автоматично генеруються на основі файлів у каталозі /pages/. Кожен файл у цьому каталозі відповідає окремому маршруту, що спрощує організацію навігації в додатку. Для реалізації динамічних маршрутів використовуються файли з параметрами у квадратних дужках, наприклад, pages/course/[id].vue.

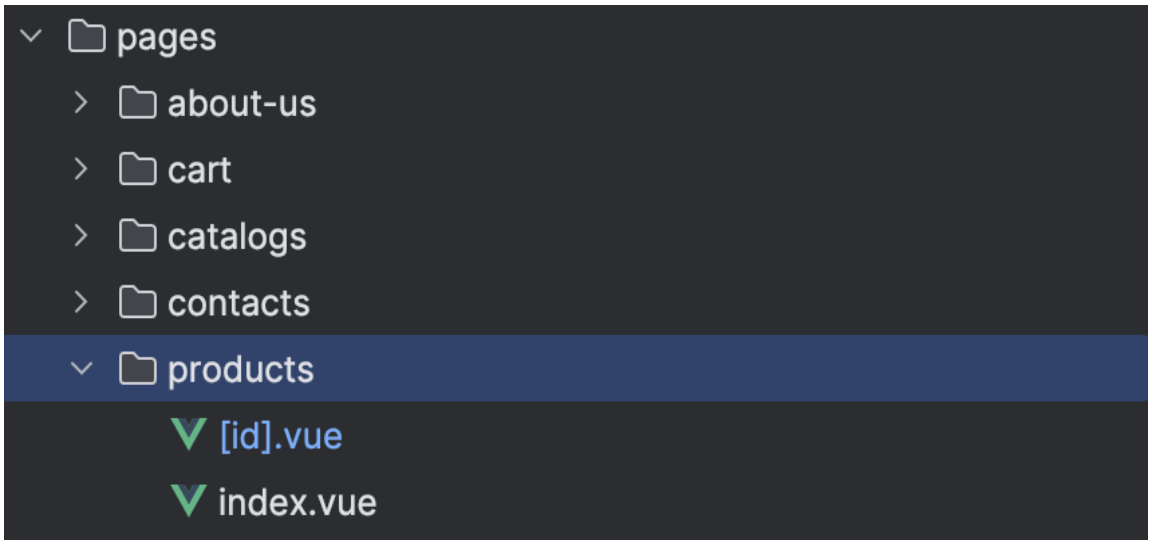


Рис. 4.5 Приклад маршрутизації в Nuxt.js

Для навігації між сторінками використовуються компоненти <nuxt-link>, які забезпечують переходи без перезавантаження сторінки.

П’ятим і заключним кроком є запуск проєкту у режимі розробки для перевірки коректності роботи та тестування інтерфейсу. Для цього використовується команда:

- npm run dev.

Після виконання команди запускається локальний вебсервер, доступний за адресою http://localhost:3000. Вебсервер слідкує за змінами у файлах джерельного коду та автоматично перекомпілює додаток та перезавантажує вікно браузера, коли зміни вносяться в код.

На рисунку 4.6 наведено приклад екранної форми сторінки головного екрану.

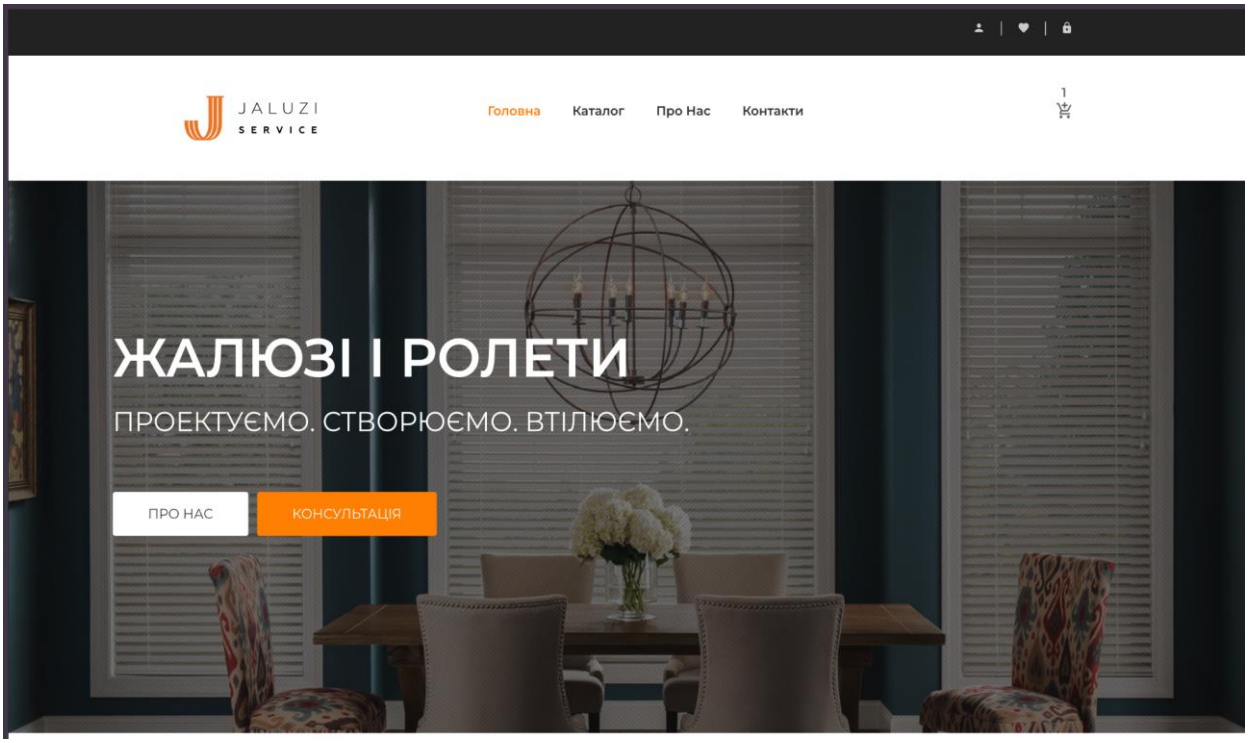


Рис. 4.6 Екранна форма головної сторінки

4.2 Реалізація серверної частини

Першим етапом реалізації серверної частини є ініціалізація нового проєкту за допомогою NestJS CLI. Для цього у терміналі виконується команда:

- nest new nest-shop.

У результаті виконання цієї команди створюється базова структура NestJS-проєкту з наступними основними елементами:

- src/: каталог із вихідним кодом;
- main.ts: файл для запуску серверу;
- app.module.ts: головний модуль проекту;
- package.json: конфігурація залежностей та скриптів.

Другим етапом є підключення бази даних. У проєкті використовується реляційна база даних PostgreSQL, а підключення реалізується через модуль TypeOrmModule, що імпортується у app.module.ts. Конфігурація включає в себе параметри підключення до бази даних, а також шлях до сутностей.

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: (configService: ConfigService) : { type: 'postgres'; host: string; port: n... => ({
    type: 'postgres',
    host: configService.get('DATABASE_HOST'),
    port: +configService.get('DATABASE_PORT'),
    username: configService.get('POSTGRES_USER'),
    password: configService.get('POSTGRES_PASSWORD'),
    database: configService.get('POSTGRES_DB'),
    entities,
    synchronize: false,
    logging: false,
    ssl: undefined,
  }),
  inject: [ConfigService],
}),
```

Рис. 4.6 Приклад підключення бази даних.

Третім кроком є генерація ресурсу (модуля) для роботи з категоріями. За допомогою CLI NestJS виконується команда:

- nest generate resource category.

У результаті генерації створюється повноцінний ресурс, що включає:

- category.module.ts – модуль для групування компонентів;
- category.controller.ts – контролер, що обробляє HTTP-запити;
- category.service.ts – сервіс для реалізації бізнес-логіки;
- DTO-файли для визначення вхідних даних;
- сутність category.entity.ts для опису структури таблиці в базі даних.

У контролері (category.controller.ts) реалізуються кінцеві точки (endpoints), які обробляють запити, та делегують виконання відповідних операцій сервісу.

Прикладом такої кінцевої точки є метод для отримання списку категорій з фільтрацією та пагінацією.

```
@Get()  Show usages  olegKovalenko *
@ApiPaginated(Category)
findAll(@Paginate() query: PaginateQuery): Promise<Paginated<Category>> {
  return this.categoryService.findAll(query);
}
```

Рис. 4.7 Приклад кінцевої точки

У свою чергу, у файлі category.service.ts реалізовується вся бізнес-логіка. Для реалізації пагінації та фільтрації використовується бібліотека nestjs-paginate, яка дозволяє легко обробляти запити зі змінними параметрами (limit, page, filter тощо).

```
findAll(query: any): Promise<Paginated<Category>> {  Show usages  olegKovalenko +2
  return paginate(query, this.categoryRepository, {
    sortableColumns: ['id', 'name', 'createdAt', 'updatedAt'],
    defaultSortBy: [['id', 'DESC']],
    filterableColumns: {
      id: [FilterOperator.IN],
      name: [FilterOperator.ILIKE],
      price: [FilterOperator.EQ],
      isActive: [FilterOperator.EQ],
      createdAt: [FilterOperator.BTW],
    },
    relations: {
      subCategory: true,
    },
  });
}
```

Рис. 4.8 Приклад метода для повернення всіх категорій

Останнім кроком є запуск серверу за допомогою команди:

- npm run start:dev.

Після цього сервер стає доступним за адресою http://localhost:3000 і готовий обробляти HTTP-запити згідно з реалізованою логікою.

4.3 Реалізація авторизації з використанням токенів JWT

У розробленому програмному додатку забезпеченні реалізовано механізм на основі токенів JWT (JSON Web Tokens). Цей підхід забезпечує захист маршрутизованих HTTP-запитів до серверної частини додатку та дозволяє здійснювати контроль доступу до функціональності відповідно до ролей користувача.

Механізм авторизації реалізовано за допомогою наступних структурних компонентів: контролера “AuthController”, сервісу “AuthService”, AuthGuard.

4.3.1 Контролер авторизації

Контролер відповідає за обробку HTTP-запитів на вхід він приймає дані користувача у вигляді об’єкта (електронна пошта та пароль), які передаються у відповідний метод сервісу авторизації:

```
@Post('login')
signIn(@Body() signInDto: AuthDto):
Promise<LoginResponseDto> {
  return this.authService.signIn(signInDto.email,
signInDto.password);
}
```

Метод повертає об’єкт, що містить згенерований токен доступу.

4.3.2 Сервіс авторизації

Сервіс виконує безпосередню логіку аутентифікації:

- пошук користувача за електроною поштою;
- перевірка пароля з використанням функції bcrypt.compare;
- формування токена JWT з такими полями (sub, username, isAdmin, role):

```
const payload = {
  sub: user.id,
  username: user.name,
```

```
    isAdmin: user.isAdmin,
    role: user.role,
  };
  const token = await
  this.jwtService.signAsync(payload);
```

Згенерований токен повертається клієнтові.

4.3.3 AuthGuard

Клас AuthGuard реалізує механізм перевірки наявності та валідності токена у вхідному запиті. Він виконує наступні дії:

- витягує токен із заголовку Authorization;
- перевіряє валідність токена за допомогою секретного ключа;
- зберігає розшифрований payload у request.user;
- за необхідності – перевіряє роль користувача для обмеження доступу до маршруту.

Приклад програмного коду:

```
const allowedRoles =
this.reflector.get<string[]>('roles',
context.getHandler());
if (
  allowedRoles &&
  !allowedRoles.includes(payload.role) &&
  payload?.role !== 'SUPER_ADMIN' &&
  !payload?.isAdmin
) {
  throw new ForbiddenException();
}
```

У разі відсутності прав доступу генерується виключення типу ForbiddenException.

4.3.4 Механізм обмеження доступу за ролями

Задання ролей реалізовано за допомогою рефлексії метаданих та декораторів. Кожен маршрут може мати обмеження за роллю, що забезпечує гнучке управління правами доступу в системі.

4.4 Завантаження проєкту на GitHub

Для того щоб додати свій проєкт до вже створеного репозиторію на GitHub, необхідно виконати послідовність стандартних кроків, які дозволяють ініціалізувати Git у проєкті, підключити віддалений репозиторій і завантажити файли.

Насамперед потрібно відкрити термінал у кореневій папці проєкту, У цьому каталозі виконується команда для створення локального Git-репозиторію:

- git init.

Після цього потрібно додати віддалений репозиторій GitHub, до якого планується завантаження файлів. Це робиться за допомогою команди з URL-адресою відповідного репозиторію:

- git remote add origin https://github.com/KovalenkoOleg/Nest-Shop.

Наступним етапом є додавання всіх файлів і папок проєкту до індексу Git. Це означає, що Git відстежуватиме ці файли при створенні коміту:

- git add.

Після додавання файлів необхідно створити перший, що фіксує поточний стан проєкту. До коміту додається коротке повідомлення, яке описує внесені зміни:

- git commit –m “init”.

Завершальним кроком є завантаження локального коміту до підключеного віддаленого репозиторію на GitHub, з одночасним прив’язуванням гілки main як основної:

- git push –u origin main.

4.5 Налаштування контейнеризації за допомогою Docker

Контейнеризація застосунку була реалізована з орієнтацією на розгортання у production-середовищі. Для цього використано Docker і Docker Compose, що забезпечило ізоляцію компонентів, спрощення розгортання та уніфіковане середовище незалежно від операційної системи [16].

Першим кроком було створення окремих Dockerfile.prod для кожного сервісу: бекенду, фронтенду, адміністративної панелі. У кожному з них прописано необхідні залежності, базовий образ, інструкції для збирання та запуску у production-режимі.

```
FROM node:19.9.0-alpine3.17
WORKDIR /nest-back
COPY package*.json ./
RUN npm install
RUN apk add --no-cache \
      nss \
      freetype \
      harfbuzz \
      ca-certificates \
      ttf-freefont

RUN npm install bull-repl -g
RUN echo 'alias migrations="npm run typeorm:run-migrations"' >> /etc/profile
ENV ENV="/etc/profile"

COPY . .
RUN npm run build
CMD ["node", "dist/src/main.js"]
```

Рис. 4.9 Приклад Dockerfile.prod для сервісної частини

Далі було сформовано головний файл docker-compose.yml, у якому описано всі сервіси системи, їх залежності та середовища виконання. У production-версії

система містить наступні сервіси:

- backend: NestJS-сервер, який обробляє запити. Підключається до бази даних через змінні середовища;
- front: користувацький інтерфейс, що збирається в статичні файли;
- admin: адміністративна панель, яка запускається окремо, але також є frontend-застосунком;
- nest-db: база даних PostgreSQL, що використовує зовнішній том для збереження даних;
- nginx: HTTP-проксі-сервер, що маршрутизує запити до відповідних сервісів.

```yaml
version: '3'

services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile.prod
    depends_on:
      - nest-db
    restart: unless-stopped
    hostname: backend
    container_name: backend
    environment:
      - DATABASE_HOST=${DATABASE_HOST}
      - DATABASE_PORT=${DATABASE_PORT}
      - DATABASE_USERNAME=${DATABASE_USERNAME}
      - DATABASE_PASSWORD=${DATABASE_PASSWORD}
      - DATABASE_NAME=${DATABASE_NAME}
      - HTTP_PORT=${HTTP_PORT}
      - AWS_SESSION_ID=${AWS_SESSION_ID}
      - AWS_ACCESS_KEY=${AWS_ACCESS_KEY}
      - AWS_DEFAULT_REGION=${AWS_DEFAULT_REGION}
      - AWS_S3_BUCKET=${AWS_S3_BUCKET}
      - JWT_SECRET_KEY=${JWT_SECRET_KEY}
    networks:
      - default

  front:
    build:
      dockerfile: Dockerfile.prod
      context: ./frontend
    container_name: front
    hostname: front
    restart: unless-stopped
    networks:
      - default

  admin:
    build:
      dockerfile: Dockerfile.prod
      context: ./admin
```

Рис. 4.10 Приклад Dockerfile.prod для бекенд-сервісу

Також створено конфігураційний файл nginx.conf.prod, у якому прописано правила маршрутизації запитів. Це дозволяє централізовано перенаправляти трафік:

- /api → на бекенд;
- /admin → на адмін-панель;
- / → на користувацький інтерфейс;

```
server {
    listen 80;
    server_name serene-screens.site;

    location / {
        proxy_pass http://front:3000;
        proxy_http_version 1.1;
    }

}

server {
    listen 80;
    server_name admin.serene-screens.site;

    location / {
        proxy_pass http://admin:3001;
        proxy_http_version 1.1;
    }

}

server {
    listen 80;
    server_name api.serene-screens.site;

    location / {
        proxy_pass http://backend:4001;
    }
}
```

Рис. 4.11 Фрагмент nginx.conf.prod

Для запуску production-застосунку використовується команда:

- docker-compose up –build.

Таким чином, було реалізовано повну production-контейнеризацію застосунку з використанням NestJS, PostgreSQL, NGINX і Docker.

4.6 Впровадження CI/CD

Для автоматизації процесу безперервної інтеграції та доставки (CI/CD) був створений workflow на базі GitHub Actions. Цей workflow налаштований на запуск при пуші у гілку main репозиторію:

```
on:
  push:
    branches: [ "main" ]
```

Першим етапом є клонування репозиторію на self-hosted runner — сервер, що виконує роботу з деплою:

```
jobs:
 checkout-repo:
   runs-on: self-hosted
   steps:
     - name: checkout repo
       uses: actions/checkout@v3
```

Це дозволяє повністю контролювати середовище виконання та уникнути обмежень публічних CI-сервісів.

Другий крок полягає у підготовці середовища для нового деплою. Виконується зупинка усіх контейнерів за допомогою команди docker compose down з параметром --remove-orphans, що гарантує видалення непотрібних або застарілих контейнерів, які можуть заважати оновленню:

```
prepare-environment:
    runs-on: self-hosted
    needs: checkout-repo
```

```
    steps:
      - name: Stop containers
        run: docker compose -f ./docker-
compose.prod.yml down --remove-orphans
```

Наступним кроком відбувається створення необхідних Docker volumes, що відповідають за збереження даних бази, та запуск усіх контейнерів з оновленими образами у фоновому режимі за допомогою docker compose up -d --build, використовуючи конфігурацію для продакшен середовища.

```yaml
deploy:
  runs-on: self-hosted
  needs: prepare-environment
  env:
    DATABASE_HOST: ${{ secrets.DATABASE_HOST }}
    DATABASE_PORT: ${{ secrets.DATABASE_PORT }}
    DATABASE_USERNAME: ${{ secrets.DATABASE_USERNAME }}
    DATABASE_PASSWORD: ${{ secrets.DATABASE_PASSWORD }}
    DATABASE_NAME: ${{ secrets.DATABASE_NAME }}
    HTTP_PORT: ${{ secrets.HTTP_PORT }}
    JWT_SECRET_KEY: ${{ secrets.JWT_SECRET_KEY }}
    AWS_SESSION_ID: ${{ secrets.AWS_SESSION_ID }}
    AWS_ACCESS_KEY: ${{ secrets.AWS_ACCESS_KEY }}
    AWS_DEFAULT_REGION: ${{ secrets.AWS_DEFAULT_REGION }}
    AWS_S3_BUCKET: ${{ secrets.AWS_S3_BUCKET }}
  steps:
    - name: create-env-vue
      working-directory: ./frontend
      run: |
        touch .env
        echo  VUE_APP_BACK_HOST_URL=${{ secrets.BACK_HOST_URL }} >> .env
    - name: create-env-admin
      working-directory: ./admin
      run: |
        touch .env
        echo  REACT_APP_BACK_HOST_URL=${{ secrets.BACK_HOST_URL }} >> .env
    - name: create-volumes
      run: |
        docker volume create db
    - name: docker-compose-up
      run: |
        docker compose -f ./docker-compose.prod.yml up -d --build
```

Рис. 4.12 Приклад запуску контейнерів

Останнім етапом виконується застосування міграцій бази даних за допомогою команди npm run typeorm:run-migrations всередині контейнера бекенду, що гарантує актуальність структури бази відповідно до змін у коді:

```
post-deploy:
     runs-on: self-hosted
     needs: deploy
     steps:
       - name: run-migrations-nest
         run: docker exec backend npm run typeorm:run-
migrations
```

Даний pipeline забезпечує надійне та автоматизоване розгортання застосунку, мінімізуючи людський фактор і ризики помилок під час оновлення системи.

4.7 Документація API (Swagger, ComDocs)

Для забезпечення зручної та зрозумілої роботи з серверним API була реалізована документація за допомогою Swagger - одного з найпопулярніших інструментів для опису RESTful сервісів [17]. Використання Swagger дозволяє не лише автоматично генерувати інтерактивний інтерфейс для тестування API, але й забезпечує чіткий опис кожної кінцевої точки, методів, параметрів і форматів даних.

Першим кроком було інтегрувати модуль Swagger у серверну частину NestJS проекту. За допомогою бібліотеки @nestjs/swagger було налаштовано генерацію документації на основі існуючих контролерів, DTO і серіалізаторів, що забезпечує відповідність між кодом і документацією.

Наступним етапом було створення конфігурації Swagger у файлі main.ts, де вказано базові параметри документації: назву API, версію, опис, а також базовий URL, під яким доступна документація. Документація автоматично оновлюється при внесенні змін у код, що значно полегшує підтримку і тестування [18].

Для покращення читабельності і деталізації опису, у DTO і контролерах було

використано відповідні декоратори, які визначають приклади вхідних і вихідних даних, типи параметрів, а також обов'язкові поля. Це дозволяє автоматично формувати точну і зрозумілу документацію без додаткових зусиль.

Кінцевою точкою доступу до документації є маршрут /api, де користувач може візуально ознайомитися з усіма доступними кінцевими точками API, їх параметрами та протестувати запити у браузері.

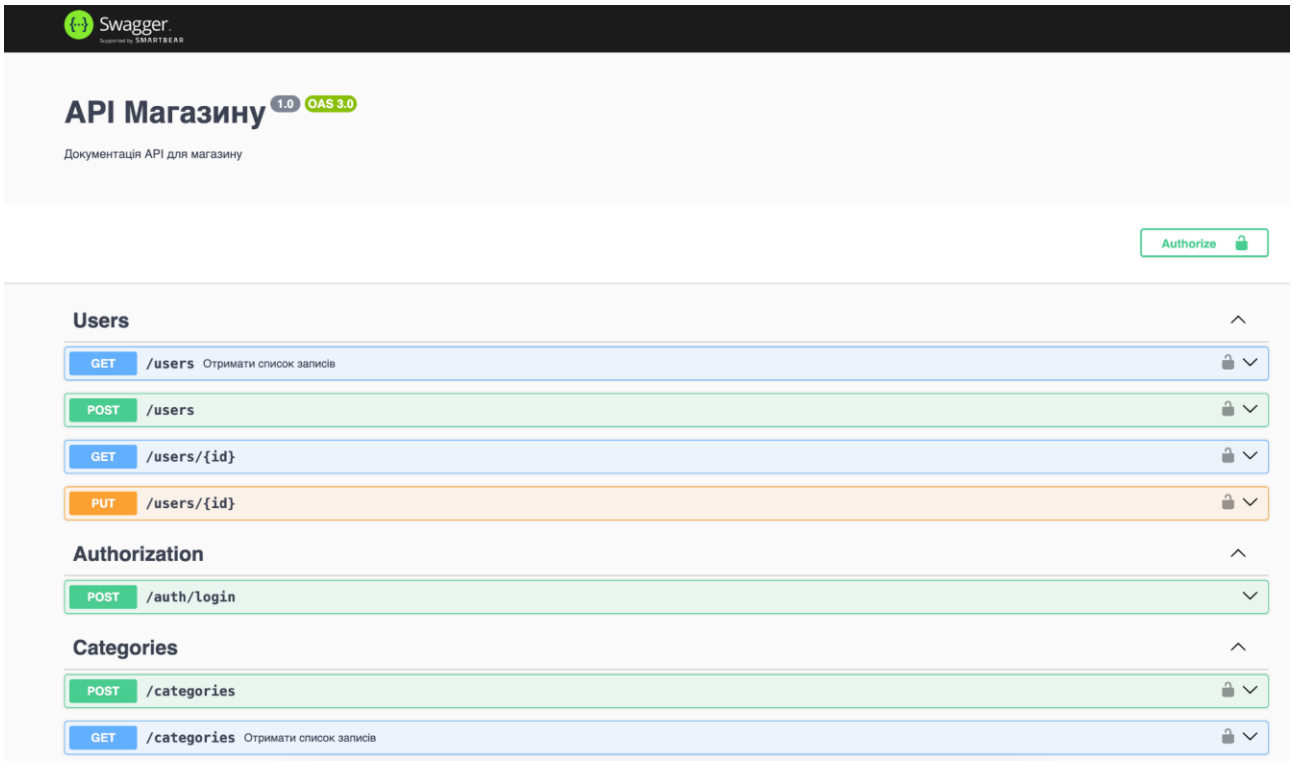


Рис. 4.13 Головний екран документації Swagger

Паралельно для опису внутрішньої структури серверної частини, модулів, сервісів і класів, а також для генерації повної технічної документації коду був застосований Compodoc. Цей інструмент автоматично аналізує TypeScript-код проекту, генерує графи залежностей, діаграми модулів і детальний опис кожного компонента, включно з його публічними методами, параметрами, властивостями та типами [19].

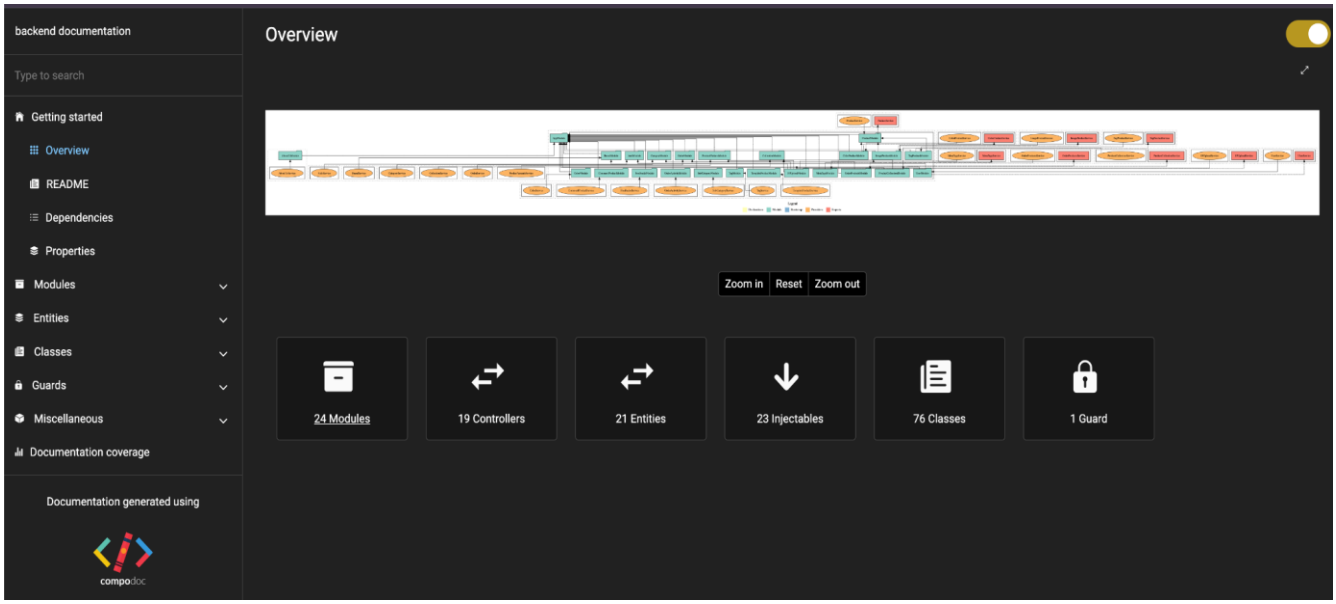


Рис. 4.14 Приклад документації оформленої через CompDocs

Завдяки такому підходу було досягнуто високого рівня автоматизації та якості документації, що сприяє ефективній роботі як розробників, так і зовнішніх інтеграторів сервісу.

ВИСНОВКИ

Результатом кваліфікаційної роботи стала розробка повнофункціонального вебзастосунку для продажу жалюзі, що дозволяє автоматизувати процес оформлення замовлень, здійснювати параметричний підбір товарів та ефективно взаємодіяти з клієнтами без використання готових CMS-систем.

У ході виконання роботи було реалізовано наступні задачі:

1. Проведено аналіз предметної області: досліджено ринок інтернет-магазинів жалюзі, виявлено основні недоліки та переваги платформ, створених на основі готових CMS-рішень, таких як WordPress, OpenCart та Shopify. На основі цього аналізу було сформульовано функціональні та нефункціональні вимоги до власного застосунку.
2. Обрано технологічний стек для реалізації платформи: NestJS для серверної частини, Nuxt.JS для фронтенду, PostgreSQL як система управління базою даних і підвищення швидкодії.
3. Розроблено архітектуру застосунку, спроєктовано схему бази даних, яка охоплює ключові сутності та зв’язки між ними.
4. Реалізовано функціонал інтернет-магазину: система параметричного підбору жалюзі з миттєвим оновленням результатів, розширений механізм фільтрації та пошуку, адаптований до специфіки товару, інтеграція з системою коментарів і рейтингу.
5. Забезпечено адаптивний інтерфейс для стабільної роботи на різних пристроях (ПК, планшетах, смартфонах).

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

1. Коваленко О.М., Андрусевич Н.В. Продуктивність PostgreSQL у порівнянні із MySQL. // Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24 квітня 2025 року, ДУІКТ, Київ. С.463-466.

2. Коваленко О.М., Андрусевич Н.В. JSON WEB TOKENS інструменти для забезпечення авторизації. // Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24 квітня 2025 року, ДУІКТ, Київ. С.46-49.

ПЕРЕЛІК ПОСИЛАНЬ

1. OpenCart vs shopify [2025]: all features comparison!. *LitExtension Blog*. URL: https://litextension.com/blog/opencart-vs-shopify/ (date of access: 25.05.2025).
2. OpenCart documentation. *OpenCart Documentation*. URL: https://docs.opencart.com/en-gb/introduction/ (date of access: 27.05.2025).
3. Shopify API, libraries, and tools. *Shopify*. URL: https://shopify.dev/docs/api (date of access: 27.05.2025).
4. WooCommerce archives. *WooCommerce*. URL: https://woocommerce.com/documentation/woocommerce/ (date of access: 27.05.2025).
5. Introduction | sublime text community documentation. *Sublime Text Community Documentation*. URL: https://docs.sublimetext.io/guide/ (date of access: 27.05.2025).
6. Meet WebStorm | WebStorm. *WebStorm Help*. URL: https://www.jetbrains.com/help/webstorm/meet-webstorm.html (date of access: 27.05.2025).
7. The starting point for learning TypeScript. *TypeScript: JavaScript With Syntax For Types*. URL: https://www.typescriptlang.org/docs/ (date of access: 25.05.2025).
8. What is nuxt.js? Learn more about the intuitive vue framework. *Kinsta®*. URL: https://kinsta.com/knowledgebase/nuxt-js/ (date of access: 25.05.2025).
9. Documentation | NestJS - A progressive Node.js framework. *Documentation | NestJS - A progressive Node.js framework*. URL: https://docs.nestjs.com/ (date of access: 25.05.2025).
10. GitHub Actions documentation - GitHub Docs. *GitHub Docs*. URL: https://docs.github.com/en/actions (date of access: 25.05.2025).
11. Contributors to Wikimedia projects. Nuxt - wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Nuxt (date of access: 25.05.2025).
12. Introduction · get started with nuxt. *Nuxt*. URL: https://nuxt.com/docs (date of access: 25.05.2025).

13. What is nest.js? Why should you use it? | turing. *Turn AGI Research into Real-World Impact | Turing*. URL: https://www.turing.com/blog/what-is-nest-js-why-use-it (date of access: 25.05.2025).
14. JSON web token introduction - jwt.io. *JSON Web Tokens - jwt.io*. URL: https://jwt.io/introduction (date of access: 25.05.2025).
15. PostgreSQL: documentation. *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/docs/ (date of access: 25.05.2025).
16. Docker engine. *Docker Documentation*. URL: https://docs.docker.com/engine/ (date of access: 25.05.2025).
17. API documentation tools | swagger. *API Documentation & Design Tools for Teams | Swagger*. URL: https://swagger.io/solutions/api-documentation/ (date of access: 27.05.2025).
18. Sayany R. Integrating Swagger with NestJS: A Step-by-Step Guide. *Medium*. URL: https://rehmat-sayany.medium.com/integrating-swagger-with-nestjs-a-step-by-step-guide-abd532743c43 (date of access: 27.05.2025).
19. Compodoc - The missing documentation tool for your Angular application. *Compodoc - The missing documentation tool for your Angular application*. URL: https://compodoc.app/guides/getting-started.html (date of access: 27.05.2025).
20. Index | node.js v24.1.0 documentation. *Node.js – Run JavaScript Everywhere*. URL: https://nodejs.org/docs/latest/api/ (date of access: 27.05.2025).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка інтернет-магазину з продажу жалюзі

Виконав студент 4 курсу
група ПД-43
Коваленко Олег Михайлович
Керівник роботи
старший викладач кафедри ІПЗ Андрусевич Наталя Володимирівна

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи**: сприяння процесу продажу жалюзі за допомогою інтернет-магазину
- **Об’єкт дослідження:** процес розробки веб-платформи для продажу товарів.
- **Предмет дослідження:** засоби та технології розробки інтернет-магазинів для продажу жалюзі

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Аналіз аналогів та підходів для продажу товарів в електронній комерції
2. Визначення вимог та проєктування web-застосунку для продажу жалюзів
3. Аналіз інструментів для створення та документації web-застосунку
4. Реалізація клієнтської частини застосунку з використанням Nuxt.JS
5. Створення серверної частини застосунку з використанням NestJS та бази даних PostgreSQL.
6. Документування серверної системи за допомогою Swagger та Compodoc
7. Тестування адаптивності та функціональної частини

3

АНАЛІЗ АНАЛОГІВ

Текст слайда

Продукт /функціональність	OpenCart	Shopify	WooCommerce	BlindWave
Гнучкість у зміні логіки роботи	Обмежена CMS	Мінімальна	Середня	Максимальна
Адмін-панель	проста, не завжди інтуїтивна	Зручна, сучасна	Залежить від WP теми	Реалізована під потреби замовника
Швидкість роботи	Залежить від кількості модулів	Залежить від кількості модулів	Багато зайвих функцій	Оптимізована під конкретний проєкт
Кастомізація дизайну	Залежить від шаблонів	Мінімальна	Відносно гнучка	Повна свобода дизайну
Залежність від сторонніх сервісів	Висока	Повна залежність	Часткова	Без залежностей

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- Реєстрація та авторизація - передбачити можливість реєстрації/авторизації для адміністратора сайту.
- Пошук та фільтрація товарів - можливість шукати товари за ключовими словами, категоріями, а також фільтрувати за їх параметрами.
- Оформлення замовлення - додавання товарів у кошик та оформлення замовлення.

Нефункціональні вимоги:

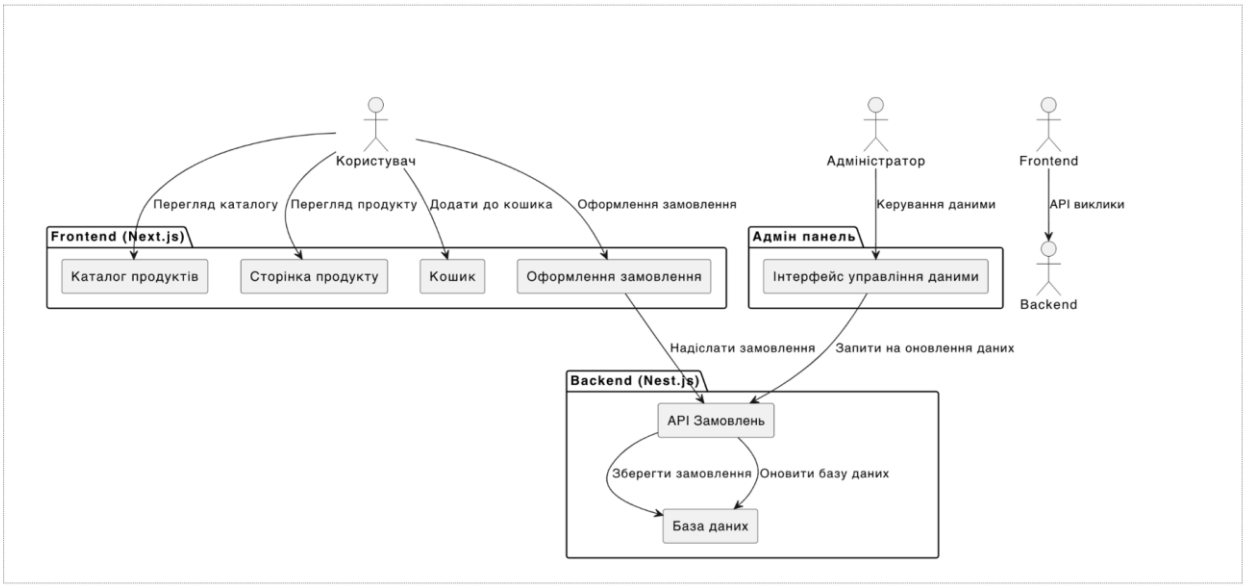
- Безпека - забезпечення захисту персональних даних.
- Швидкість роботи - сторінка повинна завантажуватися не більше 3 секунд.
- Адаптивність - адаптивний інтерфейс під різні пристрої

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

6

Архітектура web-застосунку

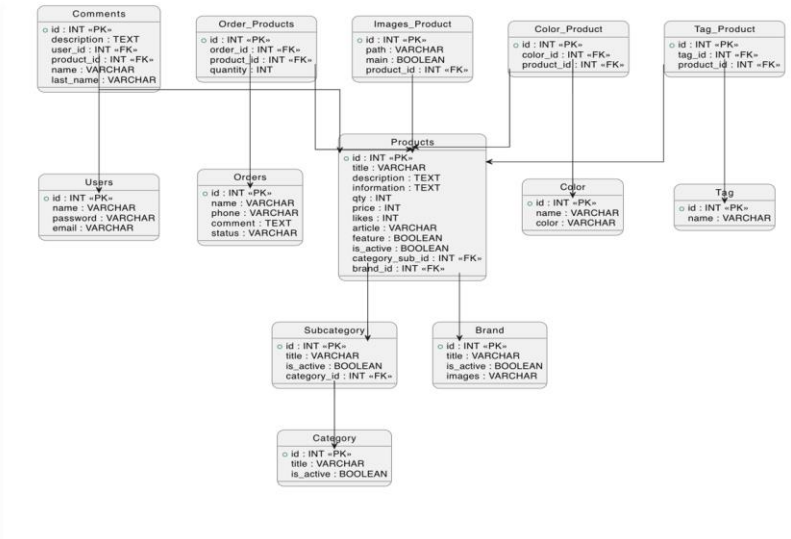


7

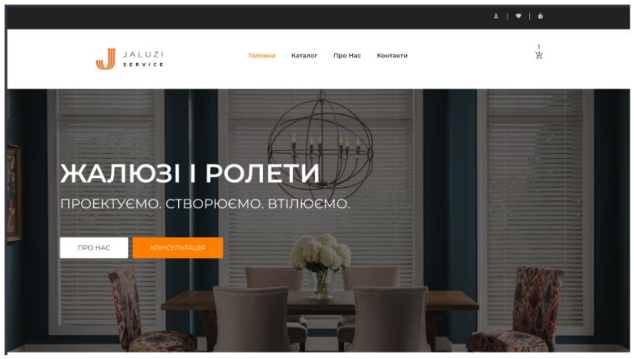
Діаграми варіантів використання

8

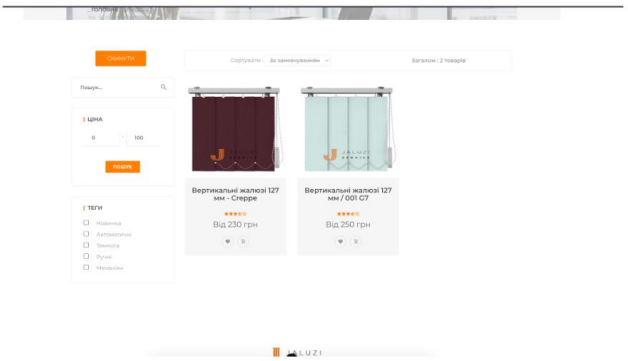
Схема бази даних



ЕКРАННІ ФОРМИ

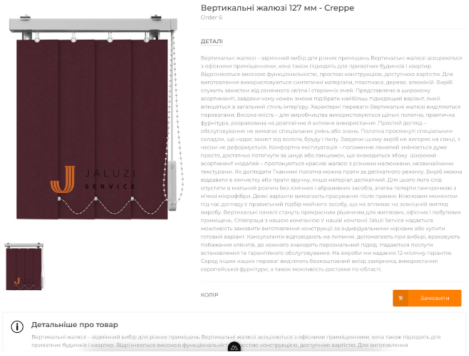


Головний екран



Каталог

ЕКРАННІ ФОРМИ



Продукт

Оформлення замовлення

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Коваленко О.М., Андрусевич Н.В. Продуктивність PostgreSQL у порівнянні із MySQL. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24 квітня 2025 року, ДУІКТ, м.Київ. С.423.
2. Коваленко О.М. ., Андрусевич Н.В. JSON WEB TOKENS інструменти для забезпечення авторизації. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24 квітня 2025 року, ДУІКТ, м.Київ. С.45.

ВИСНОВКИ

1. Проведено аналіз сучасних підходів до реалізації інтернет-магазинів жалюзі, що дало змогу визначити вимоги до функціональності та інтерфейсу додатку.
2. Досліджено існуючі рішення й типові архітектурні моделі, на основі чого обрано оптимальну структуру системи та підходи до реалізації інтерфейсної частини.
3. Визначено функціональні та нефункціональні вимоги, які лягли в основу подальшого проєктування застосунку.
4. Клієнтська частина реалізована з використанням Nuxt.js.
5. Розроблено серверну частину за допомогою NestJS та базу даних PostgreSQL.
6. Підготовлено документацію за допомогою Swagger і Compodoc, що спростить подальший розвиток і масштабування системи.
7. Проведено тестування, яке підтвердило відповідність розробленого web-застосунка вимогам, що були визначені.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Вхідний код файлу NestJS

```
import {
  BadRequestException,
  forwardRef,
  Inject,
  Injectable,
} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { InsertResult, Repository } from 'typeorm';
import { Product } from './entities/product.entity';
import {
  FilterOperator,
  FilterSuffix,
  paginate,
  Paginated,
  PaginateQuery,
} from 'nestjs-paginate';
import { TagProductService } from '../tag-product/tag-
product.service';
import { ColorProductService } from '../color-product/color-
product.service';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import slugify from 'slugify';
import { queryBuilderProducts } from
'../utils/queryBuilderProducts';
import { MetaTagsService } from '../meta-tags/meta-
tags.service';
import { ImageProductService } from '../image-product/image-
product.service';
const filterableColumns = {
  id: [FilterOperator.IN, FilterSuffix.NOT],
  title: [FilterOperator.ILIKE],
  available: [FilterOperator.ILIKE],
  'attributes[stan]': [FilterOperator.ILIKE],
  price: [FilterOperator.BTW],
  article: [FilterOperator.ILIKE],
  isActive: [FilterOperator.EQ, FilterOperator.IN],
  subCategoryId: [FilterOperator.IN],
  categoryId: [FilterOperator.IN],
  'tags.id': [FilterOperator.IN],
  'category.id': [FilterOperator.IN],
  'subCategory.id': [FilterOperator.IN],
  'brand.id': [FilterOperator.IN],
  createdAt: [FilterOperator.BTW],
};

@Injectable()
export class ProductService {
  constructor(
    @InjectRepository(Product)
    private productRepository: Repository<Product>,
    private tagsService: TagProductService,
    private colorsService: ColorProductService,
    @Inject(forwardRef(() => MetaTagsService))
    private metaTagService: MetaTagsService,
    private imageProductService: ImageProductService,
  ) {}
  async create(
    createProductDto: CreateProductDto,
    images: Array<Express.Multer.File>,
  ): Promise<InsertResult> {
    const {
      tags,
      colors,
      metaTitle,
      metaDescription,
      metaKeywords,
      otherMetaTags,
      attributes,
      ...productData
    } = createProductDto;
    const slug = slugify(createProductDto.title, '_');

    const isExistSlugProduct = await
this.productRepository.findOneBy({ slug });

    if (isExistSlugProduct)
      throw new BadRequestException('Product title already
exists');

    const product = await this.productRepository.insert({
      ...productData,
      attributes: attributes ? JSON.parse(attributes) : null,
      slug,
    });
    const id = product.identifiers[0].id;

    if (colors?.length) {
      for (const color of colors) {
        await this.colorsService.create({ productId: id, colorId:
color });
      }
    }

    if (tags?.length) {
      for (const tag of tags) {
        await this.tagsService.create({ productId: id, tagId: tag });
      }
    }

    await this.metaTagService.createInProduct({
      productId: id,
      metaKeywords,
      metaTitle,
      metaDescription,
      otherMetaTags: otherMetaTags ?
JSON.parse(otherMetaTags) : null,
    });

    await this.imageProductService.create(images, id);

    return product;
  }

  findAll(query: PaginateQuery):
Promise<Paginated<Product>> {
```

```
    const queryBuilder =
this.productRepository.createQueryBuilder('lead');
    queryBuilderProducts(queryBuilder, query);
    return paginate(query, queryBuilder, {
      sortableColumns: ['id', 'price', 'createdAt'],
      defaultSortBy: [['id', 'DESC']],
      filterableColumns: filterableColumns,
      relations: {
        subCategory: true,
        category: true,
        brand: true,
        images: true,
        colors: true,
        tags: true,
        comments: true,
        metaTags: true,
        productVariants: true,
      },
    });
  }

  findAllInSelectFilter(query: PaginateQuery):
Promise<Paginated<Product>> {
    return paginate(query, this.productRepository, {
      sortableColumns: ['id'],
      defaultSortBy: [['id', 'DESC']],
      filterableColumns: filterableColumns,
      select: ['id', 'title', 'slug'],
    });
  }

  findAllOrderProduct(query: PaginateQuery):
Promise<Paginated<Product>> {
    return paginate(query, this.productRepository, {
      sortableColumns: ['id'],
      defaultSortBy: [['id', 'DESC']],
      filterableColumns: filterableColumns,
      select: [
        'id',
        'title',
        'price',
        'salePrice',
        'available',
        'category.id',
        'category.name',
        'subCategory.id',
        'subCategory.name',
        'images.path',
      ],
      relations: {
        subCategory: true,
        category: true,
        images: true,
      },
    });
  }

  findOne(id: number): Promise<Product> {
    return this.productRepository.findOne({
      where: { id },
      relations: {
        subCategory: true,
        category: true,
        brand: true,
        colors: true,
        images: true,
        tags: true,
        comments: true,
        metaTags: true,
        productVariants: true,
      },
    });
  }

  findOneByMetaTags(id: number): Promise<Product> {
    return this.productRepository.findOneOrFail({
      where: { id },
      select: ['slug', 'id'],
    });
  }

  async update(
    id: number,
    updateProductDto: UpdateProductDto,
    images?: Array<Express.Multer.File>,
  ) {
    const {
      tags,
      colors,
      metaId,
      metaTitle,
      metaDescription,
      metaKeywords,
      otherMetaTags,
      oldImages,
      deletedItems,
      attributes,
      ...productData
    } = updateProductDto;

    const oldImagesParse = JSON.parse(oldImages);
    const deletedImages = deletedItems ?
JSON.parse(deletedItems) : [];
    await
this.imageProductService.changeImagePosition(oldImagesPars
e);
    if (deletedImages.length) {
      for (const imageId of deletedImages) {
        await this.imageProductService.remove(imageId);
      }
    }

    if (images && images.length) {
      await this.imageProductService.create(images, id);
    }
    if (updateProductDto['metaTag']) {
      delete updateProductDto['metaTag'];
    }

    if (colors) {
      const existingColors = await
this.colorsService.getProductColors(id); // Отримуємо
існуючі кольори
      const newColors = updateProductDto.colors;

      for (const existingColor of existingColors) {
        if (!newColors.includes(existingColor.colorId)) {
          await this.colorsService.removeColorFromProduct(
            id,
            existingColor.colorId,
          );
        }
      }

      for (const color of newColors) {
        if (
          !existingColors.some(
            (existingColor) => existingColor.colorId === color,
```

```
        )
       ) {
         await this.colorsService.create({ productId: id, colorId:
color });
       }
     }
   }
   if (tags) {
     const existingTags = await
this.tagsService.getProductTags(id); // Отримуємо існуючі
теги
     const newTags = updateProductDto.tags;

     for (const existingTag of existingTags) {
       if (!newTags.includes(existingTag.tagId)) {
         await this.tagsService.removeTagFromProduct(id,
existingTag.tagId);
       }
     }

     for (const tag of newTags) {
       if (!existingTags.some((existingTag) => existingTag.tagId
=== tag)) {
         await this.tagsService.create({ productId: id, tagId: tag
});
       }
     }
   }

   await this.metaTagService.updateInProduct(+metaId, {
     keywords: metaKeywords,
     title: metaTitle,
     description: metaDescription,
     otherMetaTags: otherMetaTags ?
JSON.parse(otherMetaTags) : null,
   });

   return this.productRepository.update(id, {
     ...productData,
     attributes: attributes ? JSON.parse(attributes) : null,
   });
  }
}
```

Вхідний код Next JS

```
<template>
 <div>
  <Seo />
  <div
    class="bg-opacity-black-30 dotted-overlay mb-10 mb_YTPlayer main-image-block"
    id="YTP_1690710410000"
  >
   <div class="container ptb-160" style="">
    <div class="row">
     <div style="display: flex; flex-direction: column; justify-content: flex-end" class="col-md-12">
      <div class="wow fadeInUp">
       <h1 style="color: #fff; font-size: 59px; font-weight: 600; text-transform: uppercase;" class="h1-p">Жалюзі і ролети</h1>
      </div>
      <div style="margin-top: 10px" class="wow fadeInUp">
       <h2 style="color: #fff; " class="slider2-title-3 h1-p">Проектуємо. Створюємо. Втілюємо.</h2>
      </div>
      <div class="slider-button wow fadeInUp">
       <nuxt-link to="/about-us" class="button small button-white">
        <span class="text-uppercase">Про нас</span>
       </nuxt-link>
       <a @click="openWindow('Отримати консультацію')" class="button small button">
        <span class="text-uppercase">Консультація</span>
       </a>
      </div>
     </div>
    </div>
   </div>
   <section id="page-content" class="page-wrapper">
    <!-- PRODUCT TAB SECTION START -->
    <div class="product-tab-section">
     <div style="align-items: center" class="call-to-actions">
      <div class="container-c">
       <div class="call-to-actions-item">
        <div class="call-to-actions-item__content">
         <div class="call-to-actions-item__icon"><CalculationSvg /></div>
         <h3 class="title-calculate">Розрахуйте вартість виробів за 2 хвилини</h3>
         <p class="call-to-actions-item__subtitle">Отримайте безкоштовний прорахунок, від наших фахівців</p>
         <a @click="openWindow('Отримати розрахунок')" class="button small button custom-width">
          <span class="text-uppercase">Отримати розрахунок</span>
         </a>
        </div>
       </div>
       <div class="call-to-actions-item">
        <div class="call-to-actions-item__content">
         <div class="call-to-actions-item__icon"><MeasureSvg /></div>
         <h3  class="title-calculate">Викликати замірника з каталогом матеріалів</h3>
         <p class="call-to-actions-item__subtitle">Наш фахівець виїжджає на заміри з каталогом матеріалів і підбирає найкращі варіанти для вас</p>
         <a @click="openWindow('Викликати замірника')" class="button small button custom-width">
          <span class="text-uppercase">Викликати замірника</span>
         </a>
        </div>
       </div>
      </div>
     </div>
    </div>
    <div style="margin-top: 50px" class="product-tab-section section-bg-tb pt-80 pb-55">
     <div class="container">
      <div class="row">
       <div class="col-md-6 col-sm-12 col-xs-12">
        <div class="section-title text-left mb-40">
         <h2 class="uppercase">Каталог</h2>
         <h6>Знайдіть своє ідеальне рішення: огляньте наш різноманітний каталог</h6>
        </div>
       </div>
      </div>
      <div class="product-tab">
       <div class="tab-content">
        <div class="tab-pane active" id="special-offer">
         <div class="row">
          <!-- product-item start -->
          <div v-for="category in categories" :key="category.id" class="col-md-4 col-sm-6 col-xs-12">
           <CategoryCard :data="category" />
          </div>
          <!-- product-item end -->
         </div>
        </div>
       </div>
      </div>
     </div>
    </div>

    <div class="product-tab-section mt-50 ">
     <div class="container">
      <div class="row ">
       <div class="col-md-6 col-sm-12 col-xs-12">
        <div class="section-title text-left mb-40">
         <h2 class="uppercase">Наші послуги</h2>
         <h6>Ми тут, щоб допомогти: дізнайтесь, що ми пропонуємо</h6>
        </div>
       </div>
      </div>
      <div class="row services">
       <div class="services__items">
        <li class="services-item">
         <a class="services-item__body">
          <div class="services-item__icon">
           <img src="/zamir_win.png" alt="Замеры окон" loading="lazy">
          </div>
          <span class="services-item__title">Замір вікон</span>
```

```
            </a>
           </li>
           <li class="services-item">
            <a class="services-item__body">
             <div class="services-item__icon">
              <img src="/mount.png" alt="Монтаж окон"
loading="lazy" width="112" height="100">
             </div>
             <span class="services-item__title">Монтаж
вікон</span>
            </a>
           </li>
           <li class="services-item">
            <a class="services-item__body">
             <div class="services-item__icon">
              <img src="/mot_window.png" alt="Демонтаж
окон" loading="lazy" width="112" height="100">
             </div>
             <span class="services-item__title">Демонтаж
вікон</span>
            </a>
           </li>
           <li class="services-item">
            <a class="services-item__body">
             <div class="services-item__icon">
              <img src="/mount_jaluzie.png" alt="Установка
ролетов" loading="lazy" width="112" height="100">
             </div>
             <span class="services-item__title">Встановлення
ролетів</span>
            </a>
           </li>
           <li class="services-item">
            <a class="services-item__body">
             <div class="services-item__icon">
              <img src="/zamir_rolet.png" alt="Замер ролетов"
loading="lazy" width="112" height="100">
             </div>
             <span class="services-item__title">Замір
ролетів</span>
            </a>
           </li>
          </div>
         </div>
        </div>
    <div style="margin-top: 50px; margin-bottom: 50px"
class="newsletter-section section-bg-tb pt-60 mt-50 mb-50 pb-
80">
      <div class="container-fluid">
       <div class="row">
        <div class="col-md-8 col-md-offset-2 col-xs-12">
         <div class="newsletter">
          <div class="newsletter-info text-center">
           <h2 class="newsletter-title">Як ми
працюємо</h2>
           <p>Прозорість та співпраця: ключі до нашого
успіху</p>
          </div>
          <div class="content " style="margin-top: 20px">
           <div class="btgrid">
            <div class="row row-1">
             <div class="col col-md-4 col-sm-4 col-xs-12 m-
custom-w">
              <div class="content">
               <p class="rtecenter text-align-center"><img
style="width: 70px;" class="mb-15" alt=""
src="/work/phone.png"><br><strong class="strong-
style">Створюємо заявку</strong></p>
              </div>
             </div>
             <div class="col col-md-4 col-sm-4 col-xs-12 m-
custom-w">
              <div class="content">
               <p class="rtecenter text-align-center"><img
style="width: 70px;" class="mb-15" alt=""
src="/work/inst.png"><br><strong class="strong-
style">Проводм заміри</strong></p>
              </div>
             </div>
             <div class="col col-md-4 col-sm-4 col-xs-12 m-
custom-w">
              <div class="content">
               <p class="rtecenter text-align-center"><img
style="width: 70px;" class="mb-15" alt=""
src="/work/mont.png"><br><strong class="strong-
style">Монтуємо жалюзі</strong></p>
              </div>
             </div>
            </div>
           </div>
           <p> </p>
          </div>
         </div>
        </div>
       </div>
      </div>
     <FeedbackModal @closeModal="onIsOpen"
:textBtn="textInBtn" :isOpen="isOpen"/>
    </section>
    <!-- End page content -->
   </div>
</template>
<script>
import CustomCarousel from
"~/components/CustomCarousel.vue";
import Calculation from
"~/components/Svg/CalculationSvg.vue";
import CalculationSvg from
"~/components/Svg/CalculationSvg.vue";
import MeasureSvg from
"~/components/Svg/MeasureSvg.vue";
import FeedbackModal from
"~/components/Modals/FeedbackModal.vue";
import { useMetaTagStore } from "~/stores/metaTag.ts";

export default {
 components: {FeedbackModal, MeasureSvg, CalculationSvg,
Calculation, CustomCarousel},
 async setup() {
  let isOpen = ref(false)
  let textInBtn = ref('')
  const onIsOpen = () => {
   isOpen.value = !isOpen.value
  }
  const openWindow = (text) => {
   onIsOpen();
   textInBtn.value = text
  }
  const categoryStore = useCategoriesStore();
  await useAsyncData(async () => {
   const [{data: category} ] = await Promise.all([
    categoryStore.getActionCategories({
     filters: { isActive: { value: true, type: "$eq" } },
    }),
   ])
   return { category }
  })
```

```
    const categories = computed(() =>
categoryStore.getCategories )
    return {
      categories,
      openWindow,
      isOpen,
      onIsOpen,
      textInBtn
    }
  },
}

</script>

<style lang="scss">

.main-image-block {
  position: relative;
  background-image: url('/home-image.jpg');
  background-repeat: no-repeat;
  height: 600px;
  background-position: center;
  background-size: cover;
}
.vertical-line {
  border-left: 1px solid #761616; /* Цвет и стиль черты */
  height: 100%; /* Устанавливаем высоту элемента */
}
.vertical-line {
  position: relative;
}

.vertical-line::before {
  content: '';
  position: absolute;
  top: 0;
  left: 50%;
  width: 1px; /* Ширина вертикальной линии */
  height: 100%; /* Высота вертикальной линии */
  background-color: #dc1818; /* Цвет вертикальной линии */
  transform: translateX(-50%);
}

.call-to-actions {
  position: relative
}
.call-to-actions .container-c {
  display: flex;
  justify-content: center;
  align-items: flex-start;
  flex-wrap: wrap;
}

.call-to-actions-item {
  width: calc(50% - 1px)
}

</style>
```