ЛЬВІВСЬКИЙ ТОРГОВЕЛЬНО–ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

Кафедра комп’ютерних наук, прикладної та вищої математики

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ САЙТУ ДЛЯ ПОСТАЧАЛЬНИКІВ ОПТОВИХ ТОВАРІВ

Виконав: здобувач першого
(бакалаврського)
рівня вищої освіти
Спеціальність
122 "Комп’ютерні науки"
ОПП "Комп’ютерні науки"
_____ Тицейко Святослав Богданович
(підпис) (прізвище, ім'я, по батькові)

Науковий керівник: ст. викл.
_____ Плеша В.І.
(підпис) (прізвище та ініціали)

Львів – 2026

ЦЕНТРАЛЬНА СПІЛКА СПОЖИВЧИХ ТОВАРИСТВ УКРАЇНИ
ЛЬВІВСЬКИЙ ТОРГОВЕЛЬНО–ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

Факультет міжнародних економічних відносин та інформаційних технологій
Кафедра комп'ютерних наук, прикладної та вищої математики
Перший (бакалаврський) рівень вищої освіти
Освітньо-професійна програма "Комп'ютерні науки"
Спеціальність 122 "Комп'ютерні науки"

“ЗАТВЕРДЖУЮ”

Завідувач кафедри комп'ютерних наук,
прикладної та вищої математики

_____ З. Г. Новосад
28.10. 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу студенту
Тицейко Святославу Богдановичу
(прізвище, ім’я, по батькові)

1. Тема кваліфікаційної роботи: Проєктування та реалізація сайту для постачальників оптових товарів.
керівник проєкту: Плеша Василь Іванович, старший викладач.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ректора Львівського торговельно–економічного університету № 991/03 від 06 жовтня 2025 р.

2. Термін подання студентом роботи: 5 червня 2026 р.

3. Вихідні дані для роботи. Література:

1. About Node.js. [Електронний ресурс]. — Режим доступу: https://nodejs.org/en/about
2. Controllers in ExpressJS. [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/web-tech/controllers-in-expressjs/.
3. Express - Node.js web application framework. [Електронний ресурс]. — Режим доступу: https://expressjs.com/.
4. Functions: useRouter. [Електронний ресурс]. — Режим доступу: https://nextjs.org/docs/pages/api-reference/functions/use-router.

4. Зміст розрахунково–пояснювальної записки (перелік питань, які потрібно розробити):
Поняття та особливості сучасних web-застосунків.
Особливості проєктування web-застосунків для оптової торгівлі. Визначення предметної області інтернет-магазину оптових товарів. Формування функціональних вимог до системи. Проєктування архітектури web-застосунку (клієнт, сервер, база даних). Принципи трирівневої архітектури та взаємодія компонентів системи. Поняття та принципи роботи REST API. Методи HTTP (GET, POST, PUT, DELETE) та їх використання у застосунку. Технологія розробки клієнтської частини із використанням Next.js. Технологія розробки серверної частини із використанням Express.js. Використання TypeScript для типізації даних у full-stack застосунку.

Дата видачі завдання: 28.10.2025 р.

Календарний план

№ з/п	Найменування етапів робіт	Терміни виконання	Примітка
1	Ознайомлення з рекомендованою літературою та сучасними технологіями web-розробки.	до 11.12.2025	
2	Аналіз предметної області оптової торгівлі та формування вимог до системи	до 14.01.2025	
3	Проєктування архітектури web-застосунку (клієнт, сервер, база даних)	до 17.02.2026	
4	Побудова структури бази даних та моделі предметної області	до 24.02.2026	
5	Розробка серверної частини (REST API на Express.js)	до 28.02.2026	
6	Реалізація механізмів автентифікації та авторизації користувачів	до 03.03.2026	
7	Розробка клієнтської частини застосунку (Next.js)	до 08.03.2026	
8	Реалізація функціоналу каталогу товарів, кошика та оформлення замовлення	до 17.03.2026	
9	Інтеграція клієнтської та серверної частин. Тестування взаємодії через REST API	до 31.03.2026	
10	Забезпечення безпеки та оптимізація роботи застосунку	до 05.05.2026	
11	Написання розділів кваліфікаційної роботи	до 09.05.2026	
13	Представлення кваліфікаційної роботи керівникові	до 22.05.2026	
14	Доопрацювання кваліфікаційної роботи з урахуванням зауважень керівника. Подання роботи для перевірки в системі “антиплагіат”	до 26.05.2026	
15	Подання кваліфікаційної роботи для повторної перевірки і отримання відгуку	до 01.06.2026	
16	Представлення кваліфікаційної роботи на зовнішнє рецензування. Подання роботи на кафедру комп’ютерних наук.	до 05.06.2026	
17	Попередній захист кваліфікаційної роботи	09.06.2026 - 10.06.2026	
18	Захист кваліфікаційної роботи на засіданні кваліфікаційної комісії	Згідно графіку роботи КК	

Здобувач вищої освіти _____ Тицейко С.Б..

Керівник роботи _____ Плеша В.І.

Реферат

Кваліфікаційна робота складається із вступу, трьох розділів, висновків, списку використаних джерел з 17 посиланнями. Текст кваліфікаційної роботи розташований на 64 сторінках та включає 1 рисунок і 49 прикладів з текстами програми. Робота містить web-застосунок, який показує процес розробки інтернет магазину для оптових постачальників.

Робота присвячена дослідженню сучасних підходів до проєктування та розробки web-застосунку для інтернет-магазину оптових товарів. У межах дослідження розглянуто особливості створення повноцінної клієнт-серверної системи, що забезпечує взаємодію між постачальниками та бізнес-клієнтами. У роботі проаналізовано ключові технологічні рішення, що забезпечують продуктивність, безпеку та масштабованість системи.

Ключові слова: web-застосунок, інтернет-магазин, оптова торгівля, клієнт-серверна архітектура, база даних, frontend, backend, Node.js, Next.js, Express.js.

Summary

The qualification paper consists of an introduction, three sections, conclusions, a list of sources used with 17 references. The text of the qualification work is located on 63 pages and includes 1 figure and 49 examples with program texts. The work contains a web application that shows the process of developing an online store for wholesale suppliers.

The work is devoted to the study of modern approaches to the design and development of a web application for an online store of wholesale goods. The study considers the features of creating a full-fledged client-server system that ensures interaction between suppliers and business customers. The work analyzes key technological solutions that ensure the performance, security and scalability of the system.

Keywords: web application, online store, wholesale trade, client-server architecture, database, frontend, backend, Node.js, Next.js, Express.js.

ЗМІСТ

ВСТУП 6
1 ПРОЕКТУВАННЯ СУЧАСНИХ WEB-ЗАСТОСУНКІВ 8
1.1 Поняття проєктування web-застосунків 8
1.2 Розробка структури взаємодії компонентів web-застосунку 10
1.3 Вибір технологій для розробки 13
1.4 Побудова предметної області та проєктування структури бази даних 17
2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ WEB-ЗАСТОСУНКУ 22
2.1 Ініціалізація серверного застосунку Express.js 22
2.2 Розробка авторизації та автентифікації користувача 24
2.3 Розробка функціоналу фільтрації та отримання каталогу товарів 29
2.4 Розробка функціоналу операцій над замовленнями 34
2.5 Розробка функціоналу збереження файлів 39
3 РОЗРОБКА КЛІЄНСЬКОЇ ЧАСТИНИ WEB-ЗАСТОСУНКУ 45
3.1 Розробка сторінки авторизації 45
3.2 Розробка сторінки товарів 52
3.3 Розробка сторінки покупок 57
ВИСНОВКИ 63
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 64

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у розвитку бізнесу. Більшість компаній переходять у цифровий простір, автоматизують внутрішні процеси та створюють онлайн-платформи для взаємодії з клієнтами.

Однією з перспективних сфер для впровадження web-технологій є оптова торгівля. На відміну від роздрібного продажу, де покупцем виступає кінцевий споживач, у оптовій торгівлі взаємодіють компанії. Це означає, що система повинна враховувати специфіку бізнес-процесів: великі обсяги замовлень, мінімальні партії товару, гнучке ціноутворення, перевірку реквізитів компаній, збереження історії співпраці та обробку значної кількості даних. Звичайний інтернет-магазин для фізичних осіб не завжди відповідає таким вимогам, тому виникає потреба у створенні спеціалізованого web-застосунку.

Метою даної роботи є проєктування та розробка сучасного web-застосунку для оптового інтернет-магазину, який забезпечує взаємодію між постачальниками та бізнес-клієнтами. У межах проєкту передбачається створення повноцінної системи з клієнтською та серверною частинами, базою даних, механізмами авторизації, управління товарами та оформлення замовлень. Особлива увага приділяється архітектурі застосунку, структурі бази даних, безпеці та зручності користування.

У роботі розглядаються всі основні етапи створення web-застосунку: від проєктування та визначення предметної області до реалізації серверної логіки та структури бази даних. Описується архітектура системи, принципи взаємодії між клієнтом, сервером і базою даних, вибір технологій та обґрунтування цього вибору.

Таким чином, розробка сучасного web-застосунку для оптового інтернет-магазину є комплексним завданням, що поєднує теоретичні знання з практичною реалізацією. Створена система повинна бути зручною для користувачів,

безпечною для роботи з комерційними даними та гнучкою для подальшого розвитку. Саме ці аспекти і розкриваються в наступних розділах роботи.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку з 17 використаних джерел. Основний зміст викладено на 63 сторінках і містить 3 рисунки.

1 ПРОЕКТУВАННЯ СУЧАСНИХ WEB-ЗАСТОСУНКІВ

1.1 Поняття проєктування web-застосунків

Проєктування web-застосунків — це процес детального планування майбутнього онлайн-сервісу ще до початку його технічної реалізації. Простими словами, це створення чіткого плану того, як буде працювати система, які можливості вона матиме та яким буде її зовнішній вигляд.

Web-застосунок — це програма, яка працює через браузер і не потребує встановлення на пристрій користувача. Достатньо мати доступ до інтернету, щоб користуватися таким сервісом.

У нашому випадку мова йде про створення інтернет-магазину для постачальників оптових товарів. Оптова торгівля означає продаж великої кількості товарів зазвичай для інших підприємців, а не для кінцевого споживача. Тому такий магазин має свої особливості, які потрібно врахувати ще на етапі проєктування. Наприклад, покупці можуть замовляти великі партії товару, ціни можуть залежати від обсягу замовлення, а також може існувати система перевірки компаній перед наданням доступу до покупок.

Проєктування в цьому випадку починається з визначення мети та основних завдань майбутнього магазину. Головна мета — створити зручну онлайн-платформу, яка дозволить постачальникам розміщувати свої товари, а бізнес-клієнтам швидко знаходити необхідну продукцію та оформлювати замовлення великими обсягами. Важливо одразу зрозуміти, хто буде користувачем системи: це можуть бути представники компаній, менеджери із продажу або власники бізнесу. Від цього залежить структура та функціональні можливості магазину.

Під час проєктування визначається перелік функцій, які має виконувати система. До таких функцій належить реєстрація та вхід користувачів, створення особистого кабінету, перегляд каталогу товарів, фільтрація за категоріями, оформлення замовлення та перегляд історії покупок. Каталог — це структурований список товарів, розподілений за категоріями для зручного

пошуку. А також сторінки на яких користувач може переглядати свої дані, замовлення та налаштування.

Оскільки мова йде про оптову торгівлю, необхідно передбачити можливість встановлення мінімальної кількості товару для замовлення. Це означає, що система повинна контролювати, щоб покупець не міг придбати меншу кількість, ніж дозволено постачальником. Також може бути реалізована система різних цін залежно від обсягу покупки. Наприклад, при замовленні більшої кількості товару ціна за одиницю може бути нижчою. Усе це потрібно детально продумати ще до початку розробки.

Важливим етапом є створення логічної структури застосунку. Структура — це організація сторінок та розділів і зв’язків між ними. Головна сторінка може містити загальну інформацію про компанію та популярні товари. Окремо створюються сторінки категорій, сторінка конкретного товару з детальним описом, кошик для формування замовлення та сторінка оформлення покупки. Всі ці елементи мають бути пов’язані між собою таким чином, щоб користувач міг легко переходити між ними без зайвих труднощів.

Особливу увагу потрібно приділити інтерфейсу, тобто зовнішньому вигляду застосунку. Інтерфейс включає кнопки, форми для введення даних, меню навігації, зображення товарів і текстову інформацію. Він має бути зрозумілим і зручним. Наприклад, кнопка додавання товару до кошика повинна бути добре помітною, а форма оформлення замовлення — максимально простою. Якщо система буде складною або перевантаженою інформацією, користувачам буде важко з нею працювати.

Окремо під час проєктування необхідно продумати роботу з даними. Уся інформація про користувачів, товари, замовлення та постачальників зберігається в базі даних. База даних — це спеціальна система для зберігання та впорядкування інформації. Важливо правильно визначити, які саме дані потрібно зберігати та як вони будуть пов’язані між собою. Наприклад, кожне замовлення

повинно бути пов’язане з конкретним користувачем і містити список обраних товарів.

Також значну роль відіграє безпека. Оскільки магазин працює з комерційною інформацією та даними компаній, необхідно забезпечити захист особистих даних і фінансових операцій. Система повинна перевіряти правильність введених даних, обмежувати доступ до службової інформації та захищати облікові записи користувачів.

Таким чином, проєктування web-застосунку для оптового інтернет-магазину є складним і відповідальним процесом. Воно включає визначення мети, формування функцій, створення структури, продумування інтерфейсу та організацію зберігання даних. Якісно виконане проєктування дозволяє створити зручний, надійний і ефективний онлайн-магазин, який відповідатиме потребам постачальників і бізнес-клієнтів.

1.2 Розробка структури взаємодії компонентів web-застосунку

Під час розробки web-застосунку одним із найважливіших етапів є проєктування структури взаємодії між його компонентами. Саме на цьому етапі визначається, як користувач буде взаємодіяти із системою, як оброблятимуться запити, де зберігатимуться дані та яким чином забезпечується їх передача між частинами застосунку. Правильно побудована структура взаємодії гарантує стабільність роботи системи, масштабованість та зручність подальшої підтримки.

На зображенні представлено базову схему взаємодії компонентів web-застосунку. Вона складається з трьох основних рівнів: клієнта, сервера та бази даних. Ця трирівнева архітектура є класичною для більшості сучасних систем і дозволяє чітко розділити відповідальність кожного компонента (рисунок 1.1).

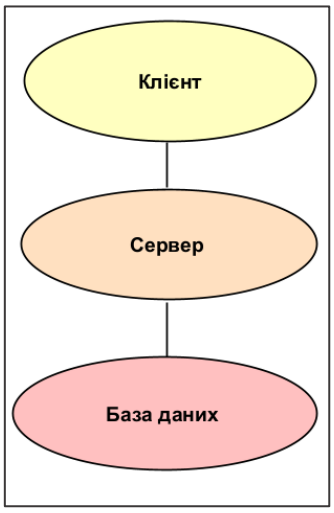

Рисунок 1.1. Схема архітектури web-застосунку

Першим рівнем є клієнт. Клієнтська частина — це інтерфейс, з яким безпосередньо працює користувач. Зазвичай це web-браузер, у якому відкривається застосунок. Саме тут користувач вводить дані, натискає кнопки, переглядає сторінки, створює або редагує записи. Клієнт не зберігає основні дані системи та не виконує складної бізнес-логіки. Його основне завдання — зібрати інформацію від користувача та передати її на сервер, а також відобразити відповідь, отриману від сервера.

Другим рівнем є сервер. Серверна частина відповідає за обробку запитів, виконання бізнес-логіки та взаємодію з базою даних. Коли клієнт надсилає запит, сервер приймає його, перевіряє правильність даних, виконує необхідні обчислення або перевірки та звертається до бази даних для отримання чи збереження інформації. Після цього сервер формує відповідь і повертає її клієнту. Таким чином, сервер виступає центральною ланкою, яка координує всю роботу системи.

Третім рівнем є база даних. Вона відповідає за збереження всієї інформації застосунку: даних користувачів, контенту, налаштувань, історії операцій тощо. База даних не взаємодіє безпосередньо з клієнтом. Усі запити до неї проходять через сервер, що забезпечує безпеку та контроль доступу. Такий підхід дозволяє захистити дані від несанкціонованого доступу та централізовано керувати логікою роботи з ними.

Взаємодія між клієнтом і сервером у сучасних web-застосунках найчастіше здійснюється за допомогою *REST API* [11]. *REST API* — це програмний інтерфейс, який дозволяє різним частинам системи обмінюватися даними через *HTTP*-запити. Завдяки *REST API* клієнт може звертатися до сервера за допомогою стандартних методів, таких як *GET, POST, PUT, DELETE*. Кожен із цих методів відповідає певній дії: отримання даних, створення нового запису, оновлення або видалення інформації.

Використання *REST API* має декілька важливих переваг. По-перше, воно забезпечує чітке розмежування між клієнтською та серверною частинами. Клієнт не знає, як саме реалізована внутрішня логіка сервера або структура бази даних. Він працює лише з визначеними маршрутами та форматами даних. По-друге, *REST API* дозволяє легко масштабувати систему. Наприклад, до одного і того ж серверного *API* можуть підключатися різні клієнти: web-версія, мобільний застосунок або навіть інша зовнішня система. По-третє, така архітектура спрощує тестування та підтримку, оскільки кожен рівень можна розвивати незалежно.

Розглядаючи схему на зображенні, можна описати типовий сценарій взаємодії. Користувач через клієнтський інтерфейс виконує певну дію, наприклад створює новий запис. Клієнт формує *HTTP*-запит і надсилає його на сервер через *REST API.* Сервер отримує цей запит, перевіряє дані, застосовує бізнес-логіку та формує відповідний запит до бази даних для збереження інформації. База даних зберігає дані та повертає результат виконання операції серверу. Після цього сервер формує відповідь у форматі, наприклад *JSON* [6], і надсилає її назад клієнту. Клієнт, отримавши відповідь, оновлює інтерфейс і відображає результат користувачу.

Важливим аспектом розробки структури взаємодії є забезпечення безпеки. Оскільки всі запити проходять через сервер, саме тут реалізується автентифікація та авторизація користувачів. Сервер перевіряє, чи має користувач право

виконувати певну дію, перш ніж звертатися до бази даних. Це дозволяє запобігти несанкціонованому доступу до інформації.

Крім того, чітке розділення на клієнт, сервер і базу даних сприяє масштабованості. У разі зростання кількості користувачів можна збільшити ресурси сервера або додати декілька серверів, не змінюючи клієнтську частину. Аналогічно, базу даних можна оптимізувати або перенести на потужніше обладнання без зміни логіки взаємодії.

Таким чином, розробка структури взаємодії компонентів web-застосунку є ключовим етапом створення системи. Використання трирівневої архітектури з клієнтом, сервером і базою даних забезпечує чіткий розподіл відповідальності між компонентами. Застосування *REST API* дозволяє організувати стандартизований, зрозумілий і масштабований механізм обміну даними між клієнтською та серверною частинами. У результаті формується гнучка, безпечна та зручна для подальшого розвитку архітектура web-застосунку.

1.3 Вибір технологій для розробки

Під час розробки інтернет-магазину для продажу та замовлення оптових товарів надзвичайно важливим етапом є вибір технологій. Від цього рішення залежить продуктивність системи, зручність користування, можливість масштабування, безпека даних та подальша підтримка проєкту. Оскільки мова йде про повноцінний web-застосунок із клієнтською та серверною частинами, обробкою замовлень, роботою з великою кількістю товарів та користувачів, було обрано сучасний стек технологій: *Next.js* [10] для клієнтської частини, *Express.js* [3] для серверної частини, *TypeScript* як основну мову розробки та *Tailwind CSS* для стилізації інтерфейсу.

Інтернет-магазин оптових товарів має свою специфіку. На відміну від звичайного роздрібного магазину, тут часто працюють зареєстровані користувачі — представники компаній або підприємці, які здійснюють великі замовлення, формують кошики з десятків позицій, переглядають історію закупівель та

працюють із персональними цінами. Тому застосунок повинен бути швидким, стабільним і зручним, а також підтримувати складну бізнес-логіку. Саме це і стало основою для вибору відповідних технологій.

Для реалізації клієнтської частини було обрано *Next.js*. Це сучасний фреймворк на базі *React* [10], який дозволяє створювати продуктивні та оптимізовані web-застосунки. Однією з головних причин вибору *Next.js* є підтримка серверного рендерингу *(SSR)* [12]. Для інтернет-магазину це особливо важливо, адже сторінки товарів повинні швидко завантажуватися, а також добре індексуватися пошуковими системами. Оптимізація для пошукових систем безпосередньо впливає на видимість магазину та залучення нових клієнтів.

Крім того, *Next.js* забезпечує зручну маршрутизацію на основі файлової структури, що спрощує організацію сторінок каталогу товарів, сторінок категорій, профілю користувача, кошика та оформлення замовлення. Такий підхід робить проєкт структурованим і зрозумілим. Важливою перевагою є також можливість розділення клієнтських і серверних компонентів, що дозволяє ефективно керувати логікою відображення даних і зменшувати навантаження на браузер користувача.

Next.js добре підходить для масштабування. У випадку розширення функціоналу, наприклад додавання особистих кабінетів для постачальників або інтеграції з платіжними системами, архітектура фреймворку дозволяє гнучко розширювати застосунок без повної перебудови структури. Також він має активну спільноту та велику кількість готових рішень, що пришвидшує розробку.

Для реалізації серверної частини було обрано *Express.js*. Це мінімалістичний і гнучкий фреймворк на базі платформи *Node.js* [1], який дозволяє швидко створювати *REST API*. Оскільки інтернет-магазин потребує обробки великої кількості запитів — авторизації користувачів, отримання списків товарів, створення замовлень, оновлення статусів, роботи з кошиком — важливо мати стабільний та зрозумілий інструмент для реалізації серверної логіки.

Express.js дозволяє чітко структурувати маршрути *API*, розділяти логіку на контролери, сервіси та *middleware* [17]. Це особливо важливо для застосунку з великою кількістю бізнес-процесів. Наприклад, під час створення замовлення сервер повинен перевірити наявність товару, розрахувати загальну суму, врахувати оптові знижки, зберегти дані в базі та повернути відповідь клієнту. Завдяки Express.js така логіка може бути реалізована прозоро та структуровано.

Ще однією перевагою Express.js є його сумісність із великою кількістю бібліотек для роботи з автентифікацією, валідацією даних, логуванням та безпекою. Для інтернет-магазину це критично важливо, оскільки система працює з персональними даними користувачів та інформацією про замовлення. Сервер виступає центральною точкою контролю доступу та перевірки прав користувачів.

Як основну мову розробки було обрано *TypeScript* [13]. Використання *TypeScript* у повному стеку — як на клієнті, так і на сервері — забезпечує типізацію даних та зменшує кількість помилок під час розробки. В інтернет-магазині існує багато сутностей: товар, категорія, користувач, замовлення, позиція замовлення, кошик. Кожна з них має чітку структуру. Завдяки *TypeScript* можна описати ці структури у вигляді інтерфейсів або типів і гарантувати, що дані відповідають очікуваному формату.

Це особливо важливо при взаємодії між клієнтом і сервером. Наприклад, коли сервер повертає список товарів, клієнт повинен точно знати, які поля містить кожен об’єкт. *TypeScript* дозволяє синхронізувати типи між частинами застосунку, що значно зменшує ризик помилок. Крім того, типізація покращує читабельність коду та полегшує підтримку проєкту в майбутньому.

Ще однією важливою перевагою *TypeScript* є покращена підтримка з боку середовищ розробки. Автодоповнення, підказки та перевірка типів у реальному часі прискорюють написання коду та зменшують кількість логічних помилок. Для великого проєкту, яким є оптовий інтернет-магазин, це суттєво підвищує якість програмного забезпечення.

Для стилізації інтерфейсу було обрано *Tailwind CSS* [5]. Це сучасний *CSS*-фреймворк, який працює за принципом *utility-first*. Замість написання великої кількості окремих *CSS*-файлів, стилі формуються безпосередньо в розмітці за допомогою готових класів. Такий підхід значно пришвидшує розробку інтерфейсу.

В інтернет-магазині важливо забезпечити зручний та зрозумілий інтерфейс, оскільки користувачі повинні швидко знаходити товари, переглядати характеристики, формувати замовлення та оформлювати покупку. *Tailwind CSS* дозволяє швидко створювати адаптивний дизайн, що правильно відображається на різних пристроях — від комп'ютерів до планшетів [5]. Це особливо актуально, оскільки частина користувачів може працювати із системою з мобільних пристроїв.

Tailwind також сприяє підтримці єдиного стилю в усьому застосунку. Завдяки конфігурації можна задати корпоративні кольори, шрифти та відступи, що забезпечує цілісність дизайну. Крім того, відсутність великої кількості кастомних *CSS*-файлів спрощує підтримку та зменшує ризик конфліктів стилів.

Обраний стек технологій формує сучасну, гнучку та продуктивну архітектуру. *Next.js* відповідає за швидкий та оптимізований інтерфейс користувача, *Express.js* — за надійну серверну логіку та обробку запитів через *REST API, TypeScript* забезпечує безпечну типізацію та якість коду, а *Tailwind CSS* дозволяє швидко створити адаптивний та зручний дизайн.

Таким чином, вибір саме цього набору технологій є обґрунтованим з точки зору функціональних вимог інтернет-магазину оптових товарів. Він дозволяє створити масштабований, безпечний та зручний у використанні web-застосунок, який може розвиватися та доповнюватися новими можливостями без суттєвих архітектурних змін.

1.4 Побудова предметної області та проєктування структури бази даних

Під час розробки інтернет-магазину для продажу та замовлення оптових товарів одним із ключових етапів є побудова предметної області та проєктування структури бази даних. Саме на цьому етапі визначається, які об’єкти існують у системі, які характеристики вони мають та як пов’язані між собою. Правильно спроєктована база даних забезпечує логічність зберігання інформації, швидкість роботи застосунку та можливість подальшого масштабування.

Предметна область у межах даного проєкту охоплює взаємодію компаній-постачальників і компаній-покупців. Кожна компанія реєструється в системі, додає інформацію про себе, публікує товари або здійснює закупівлі. Основними сутностями є користувач (компанія), товар та замовлення. Саме ці об’єкти формують основу структури бази даних.

У проєкті використовується *MongoDB* — це нереляційна база даних, яка зберігає інформацію у вигляді документів [8]. Документи мають формат, подібний до *JSON*, що дозволяє зберігати вкладені об’єкти та масиви без необхідності створювати багато окремих таблиць. Такий підхід добре підходить для систем, де структура даних може бути гнучкою та змінюватися з часом.

Першою ключовою сутністю є користувач *(User).* У даному випадку користувачем є компанія. Документ користувача містить як дані для авторизації *(email, password, refreshToken),* так і розширену інформацію про компанію. Зокрема, зберігається назва компанії, опис діяльності, контактні номери та контактна особа. Контактна особа є вкладеним об’єктом, який містить ім’я, посаду, телефон та електронну пошту. Це дозволяє правильно згрупувати дані та зробити структуру зрозумілою (приклад 1.1).

```
export interface IUser extends Document {
  _id: mongoose.Types.ObjectId;
  email: string;
  password: string;
  refreshToken?: string | null;
  companyName: string;
```

```
  companyNameEn?: string;
  description?: string;
  website?: string;
  logo?: string;
  photos?: string[];
  phone: string;
  additionalPhones?: string[];
  contactPerson: IContactPerson;
  legalAddress: IAddress;
  actualAddress?: IAddress;
  edrpou?: string; // ЄДРПОУ
  taxId?: string; // Податковий номер
  registrationDate?: Date;
  legalForm?: string;
   businessTerms?: IBusinessTerms;
  productCategories?: IProductCategory[];
  specialization?: string[];
    isVerified: boolean;
  isActive: boolean;
  rating?: number;
  reviewsCount?: number;

  createdAt: Date;
  updatedAt: Date;
}
```

Приклад 1.1 — Схема колекції користувачів

Окремо в структурі користувача передбачено юридичну та фактичну адресу. Адреса також реалізована як вкладений об’єкт із полями вулиці, міста, області, поштового індексу та країни. Такий підхід відповідає реальній бізнес-логіці, оскільки адреса є невід’ємною характеристикою компанії.

У системі також зберігаються реєстраційні дані компанії, такі як ЄДРПОУ, податковий номер, дата реєстрації та організаційно-правова форма (ТОВ, ФОП, ПП тощо). Це важливо для оптового сегменту, оскільки співпраця між компаніями потребує офіційних реквізитів.

Ще одним важливим блоком є бізнес-умови компанії. Вони містять мінімальну суму замовлення, можливі способи оплати, умови доставки, регіони доставки та графік роботи. У *MongoDB* такі дані зручно зберігати у вигляді вкладених об’єктів і масивів, що дозволяє уникнути створення окремих колекцій для кожного типу умов.

Також у структурі користувача передбачено категорії товарів та спеціалізацію. Це дає можливість класифікувати компанії за напрямами діяльності та реалізувати зручний пошук. Для оптимізації пошуку створено індекси, зокрема текстовий індекс для назви компанії та опису.

Другою важливою сутністю є товар *(Good)*. Товар належить конкретному постачальнику, що реалізується через збереження *ObjectId* з посиланням на користувача. У документі товару зберігається назва, опис, категорія, підкатегорія, ціна, валюта, одиниця виміру та мінімальна кількість для замовлення. Також передбачено поле *inStock* та кількість на складі, що дозволяє контролювати доступність товару (приклад 1.2).

```
export interface IGood extends Document {
  _id: mongoose.Types.ObjectId;
  name: string;
  nameEn?: string;
  description?: string;
  category: string;
  subcategory?: string;
  price: number;
  currency: string;
  unit: string; // 'шт', 'кг', 'м', 'м²', 'м³', 'л' тощо
  minimumOrderQuantity: number;
  inStock: boolean;
  stockQuantity?: number;
  photos: string[];
  specifications?: Record<string, string | number>;
  licenses?: {
    type: string; // 'стандартна', 'преміум', 'enterprise' тощо
    description?: string;
    validUntil?: Date;
    certificateNumber?: string;
    issuer?: string;
  }[];
  supplier: mongoose.Types.ObjectId;
  isActive: boolean;
  createdAt: Date;
  updatedAt: Date;
}
```

Приклад 1.2 — Схема колекції товарів

Особливістю структури товару є можливість зберігати специфікації у вигляді *Map*, що дозволяє додавати довільні характеристики (наприклад, розмір, матеріал, потужність). Це демонструє гнучкість *MongoDB*, оскільки структура характеристик може відрізнятися залежно від категорії товару. Крім того, у товарі можуть зберігатися дані про ліцензії або сертифікати, що є важливим для деяких видів продукції.

Для пришвидшення пошуку створено індекси за назвою, категорією, ціною, постачальником та статусом активності. Це забезпечує швидке формування каталогу товарів навіть при великій кількості записів.

Третьою ключовою сутністю є замовлення *(Order)*. Замовлення відображає факт взаємодії між покупцем і постачальником. У структурі замовлення зберігається посилання на покупця, постачальника та товар (приклад 1.3). Таким чином реалізується зв’язок між колекціями через *ObjectId*.

```
export interface IOrder extends Document {
  _id: mongoose.Types.ObjectId;
  buyer: mongoose.Types.ObjectId
  supplier: mongoose.Types.ObjectId;
  good: mongoose.Types.ObjectId
  quantity: number;
  unitPrice: number;
  totalPrice: number;
  currency: string;
  status: 'pending' | 'confirmed' | 'processing' | 'shipped' | 'delivered' | 'cancelled';
  deliveryAddress?: {
    city: string;
    region: string;
    postalCode: string;
    country: string;  };}
```

Приклад 1.3 — Схема колекції замовлень

Замовлення містить кількість товару, ціну за одиницю на момент покупки та загальну суму. Збереження ціни безпосередньо в замовленні є важливим, оскільки ціна товару може змінюватися, але історія замовлень повинна залишатися незмінною. Також у замовленні зберігається валюта та статус

виконання. Статус дозволяє відстежувати етап обробки: очікує підтвердження, підтверджено, в обробці, відправлено, доставлено або скасовано.

Додатково замовлення може містити адресу доставки та контактну особу для конкретної операції. Це дозволяє зберегти актуальні дані саме для цього замовлення, навіть якщо інформація в профілі компанії зміниться.

Проєктування структури бази даних також включало створення індексів для оптимізації запитів. Зокрема, індекси створено для пошуку замовлень за покупцем, постачальником, статусом та датою створення. Це дозволяє швидко формувати списки замовлень у кабінеті користувача.

Таким чином, побудова предметної області базується на трьох основних сутностях: компанія, товар і замовлення. Взаємозв’язки між ними реалізовані через посилання на *ObjectId*, а вкладені об’єкти дозволяють логічно групувати дані. Використання *MongoDB* забезпечує гнучкість структури, простоту розширення та ефективну роботу з великим обсягом інформації.

У результаті було сформовано цілісну, масштабовану та логічну модель даних, яка відповідає бізнес-процесам оптового інтернет-магазину та забезпечує стабільну роботу всієї системи.

2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ WEB-ЗАСТОСУНКУ

2.1 Ініціалізація серверного застосунку Express.js

Ініціалізація серверної частини є одним із ключових етапів розробки інтернет-магазину оптових товарів. Саме на цьому етапі формується базова структура застосунку, підключаються необхідні інструменти та налаштовується взаємодія між клієнтською частиною і базою даних. Правильно організований процес запуску сервера забезпечує стабільну роботу системи, можливість масштабування та зручність подальшої підтримки.

Для реалізації серверної логіки використовується фреймворк *Express.js* у середовищі *Node.js* [1] з використанням *TypeScript*. Такий стек дозволяє створити швидкий, зрозумілий і структурований серверний застосунок.

Для початку створимо новий проєкт та ініціалізуємо файл *package.json* командою (приклад 2.1).

```
npm install express mongoose cors cookie-parser bcryptjs jsonwebtoken multer
```

Приклад 2.1 — Команда встановлення базових пакетів

Тут *express* використовується для створення *HTTP*-сервера, *mongoose* — для підключення до бази даних *MongoDB* та роботи з моделями, *cors* — для дозволу запитів із клієнтської частини, *cookie-parser* — для роботи з *cookies*, *bcryptjs* — для хешування паролів, *jsonwebtoken* — для створення токенів авторизації, а *multer* — для завантаження файлів (наприклад, зображень товарів).

Після встановлення залежностей додаємо зручні команди в розділ *scripts* файлу *package.json.*

Команда *build* компілює *TypeScript* у *JavaScript*, *start* запускає вже скомпільований сервер, а *dev* використовується під час розробки — вона автоматично перезапускає сервер при зміні файлів.

Далі створимо основний файл сервера — *src/server.ts*. У ньому спочатку імпортуємо необхідні модулі (приклад 2.2).

```
"build": "tsc",
"start": "node build/server.js",
"dev": "nodemon --watch 'src/**/*.ts' --exec 'tsx' src/server.ts
```

Приклад 2.2 — Команди запуску проєкту

Після цього створюємо застосунок. У прикладі 2.3 продемонстровано, що ми створили сервер, який буде приймати *HTTP*-запити від клієнтської частини.

```
const app = express();
```

Приклад 2.3 — Команди запуску проєкту

Наступним кроком налаштовуємо *CORS* — механізм безпеки, який дозволяє або забороняє запити з інших адрес (приклад 2.4).

```
app.use(cors({
  origin: ['http://localhost:3000'],
  credentials: true
}));
```

Приклад 2.4 — Налаштування *CORS*

Після цього додаємо обробники для роботи з даними запиту. *express.json()* дозволяє серверу приймати дані у форматі *JSON*, *express.urlencoded()* — обробляти дані з форм, а *cookieParser()* — працювати з *cookies* (приклад 2.5).

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(cookieParser());
```

Приклад 2.5 — Налаштування обробників даних запиту

Далі підключаємося до бази даних (приклад 2.6).

```
mongoose.connect(process.env.MONGO_URI as string)
  .then(() => console.log('Підключено до MongoDB'))
  .catch(err => console.error(err));
```

Приклад 2.6 — Код підключення до бази даних

На завершальному етапі запускаємо сервер. Метод *listen* запускає сервер і вказує порт, на якому він буде приймати запити (приклад 2.7).

```
const PORT = 5000;

app.listen(PORT, () => {
  console.log(`Сервер запущено на порту ${PORT}`);
});
```

Приклад 2.7 — Програмний код запуску сервера

Таким чином, процес ініціалізації серверного застосунку включає встановлення залежностей, налаштування команд запуску, створення основного

файлу сервера, підключення проміжних обробників, встановлення з’єднання з базою даних та реєстрацію маршрутів. Така структура забезпечує логічну організацію коду, зручність розширення функціоналу та стабільну роботу інтернет-магазину.

2.2 Розробка авторизації та автентифікації користувача

У будь-якому сучасному web-застосунку дуже важливо забезпечити безпечний доступ користувачів до системи. Саме для цього реалізовується механізм автентифікації та авторизації. Автентифікація відповідає за перевірку особи користувача — тобто система повинна переконатися, що людина дійсно є тим, за кого себе видає. Авторизація, у свою чергу, визначає, які саме дії дозволені цьому користувачу після входу в систему.

У розробленому застосунку механізм доступу реалізовано за допомогою токенів формату *JSON Web Token (JWT)*. *JWT* — це спеціальний зашифрований токен, який містить інформацію про користувача (наприклад, його ідентифікатор та *email*) [6]. Цей токен створюється сервером після успішного входу та надалі використовується для підтвердження особи користувача без повторного введення пароля при кожному запиті.

Архітектура реалізації побудована за принципом розділення відповідальності. Для цього використовуються такі основні складові: router, *controller* та *service*.

Router — це частина застосунку, яка відповідає за обробку *HTTP*-запитів. Він визначає, яка функція буде виконана при зверненні до певного маршруту, наприклад */register*, */login*, */refresh* або */logout*. Іншими словами, *router* приймає запит від клієнта і передає його далі у відповідний контролер (приклад 2.8).

```
import { Router } from 'express';
import { authController } from '../controllers/auth.contoller.js';

const router = Router();

router.post('/register', (req, res) => authController.register(req, res));
```

```
router.post('/login', (req, res) => authController.login(req, res));
router.post('/refresh', (req, res) => authController.refresh(req, res));
router.post('/logout', (req, res) => authController.logout(req, res));

export default router;
```

Приклад 2.8 — *Router* для авторизації

Controller — це компонент, який приймає запит від *router*, обробляє вхідні дані, виконує базову перевірку (валідацію) та викликає необхідні методи сервісу [2]. Контролер також формує відповідь і повертає її клієнту. Він не містить складної бізнес-логіки, а лише координує роботу інших частин системи (приклад 2.9).

```
import type { Request, Response } from 'express';
import { authService } from '../services/auth.service.js';
import config from '../config/config.js';

export class AuthController {
  /**
   * POST /auth/register
   * Реєстрація нового користувача
   */
}
```

Приклад 2.9 — Контролер авторизації

Service — це рівень, де реалізована основна бізнес-логіка. Саме тут відбувається робота з базою даних, хешування паролів, генерація токенів та їх перевірка [15]. Такий підхід робить код структурованим, зрозумілим та зручним для підтримки (приклад 2.10).

```
import bcrypt from 'bcryptjs';
import jwt, { type SignOptions } from 'jsonwebtoken';
import type { IUser } from '../models/user.model.js';
import { User } from '../models/user.model.js';
import config from '../config/config.js';

export interface TokenPayload {
  userId: string;
  email: string;
}

export interface AuthTokens {
```

```
  accessToken: string;
  refreshToken: string;
}

export class AuthService {
  /**
   * Реєстрація нового користувача
   */
}
```

Приклад 2.10 — Реалізація сервісу авторизації

Процес реєстрації користувача працює наступним чином. Користувач надсилає *email* та пароль. Контролер перевіряє, чи заповнені обов'язкові поля та чи відповідає пароль мінімальним вимогам. Далі сервіс перевіряє, чи не існує вже користувач із таким *email*. Якщо ні — пароль хешується за допомогою бібліотеки *bcrypt*, після чого створюється новий запис у базі даних. Після успішної реєстрації генеруються два токени: *access token* та *refresh token*. *Access token* використовується для доступу до захищених маршрутів, а *refresh token* — для отримання нового *access token* без повторного введення пароля (приклад 2.11).

```
  async register(req: Request, res: Response): Promise<void> {
    try {
      const { email, password, businessProfile } = req.body;
      if (password.length < 6) {
        res.status(400).json({ error: 'Пароль повинен містити мінімум 6 символів' });
        return;
      }
      const user = await authService.register(email, password, businessProfile);

      // Генерація токенів
      const tokens = authService.generateTokens(user._id.toString(), user.email);

      // Збереження refresh токена в базі даних
      user.refreshToken = tokens.refreshToken;
      await user.save();

      // Встановлення cookies
      this.setRefreshTokenCookie(res, tokens.refreshToken);

      res.status(201).json({
        message: 'Користувач успішно зареєстрований',
        user: {
          id: user._id.toString(),
```

```
        email: user.email,
      },
      accessToken: tokens.accessToken,
    });
  } catch (error) {
      const errorMessage = error instanceof Error ? error.message : 'Помилка при реєстрації';
    res.status(400).json({ error: errorMessage });
  }
 }
```

Приклад 2.11 — Бізнес-логіка реєстрації

Під час авторизації (входу в систему) користувач вводить *email* і пароль. Сервіс знаходить користувача в базі даних та порівнює введений пароль із збереженим хешем. Якщо дані правильні, генеруються нові токени. *Refresh token* зберігається в базі даних та додатково передається клієнту через *HTTP-only* cookie. Це означає, що токен недоступний для *JavaScript* у браузері, що підвищує рівень безпеки (приклад 2.12).

```
 async login(req: Request, res: Response): Promise<void> {
  try {
    const { email, password } = req.body;

    // Валідація вхідних даних
    if (!email || !password) {
      res.status(400).json({ error: 'Email та пароль обов\'язкові' });
      return;
    }

    // Авторизація
    const { user, tokens } = await authService.login(email, password);

    // Встановлення refresh токена в cookie
    this.setRefreshTokenCookie(res, tokens.refreshToken);

    res.status(200).json({
      message: 'Успішна авторизація',
      user: {
        id: user._id.toString(),
        email: user.email,
      },
      accessToken: tokens.accessToken,
    });
  } catch (error) {
```

```
      const errorMessage = error instanceof Error ? error.message : 'Помилка при авторизації';
    res.status(401).json({ error: errorMessage });
   }
  }
```

Приклад 2.12 — Бізнес-логіка входу в систему

Механізм оновлення токенів працює так: коли *access token* закінчує свій термін дії, клієнт надсилає запит на оновлення. Сервер перевіряє *refresh token*, і якщо він дійсний та збігається з тим, що збережений у базі, генерує нову пару токенів. Старий *refresh token* замінюється новим (приклад 2.13).

```
 async refresh(req: Request, res: Response): Promise<void> {
  try {
   // Отримання refresh токена з cookie
   const refreshToken = req.cookies?.refreshToken;

   if (!refreshToken) {
    res.status(401).json({ error: 'Refresh токен відсутній' });
    return;
   }

   // Оновлення токенів
   const tokens = await authService.refresh(refreshToken);

   // Встановлення нового refresh токена в cookie
   this.setRefreshTokenCookie(res, tokens.refreshToken);

   res.status(200).json({
    message: 'Токени успішно оновлені',
    accessToken: tokens.accessToken,
   });
  } catch (error) {
   const errorMessage = error instanceof Error ? error.message : 'Помилка при оновленні токенів';
   res.status(401).json({ error: errorMessage });
  }
 }
```

Приклад 2.13 — Програмний код поновлення токенів

Процес виходу з системи передбачає видалення *refresh token* з бази даних та очищення *cookie*. Таким чином, навіть якщо хтось отримає старий токен, він більше не зможе ним скористатися.

```
 async logout(req: Request, res: Response): Promise<void> {
  try {
   // Отримання refresh токена з cookie
   const refreshToken = req.cookies?.refreshToken;

   if (refreshToken) {
    try {
     // Верифікація refresh токена для отримання userId
     const jwt = await import('jsonwebtoken');
     const decoded = jwt.default.verify(refreshToken, config.jwtRefreshSecret) as { userId:
string; email: string };
     await authService.logout(decoded.userId);
    } catch (error) {
     // Якщо токен невалідний, просто очищаємо cookie
    }}
```

Приклад 2.14 — Бізнес-логіка виходу з системи

```
   // Очищення refresh токена з cookie
   this.clearRefreshTokenCookie(res);

   res.status(200).json({ message: 'Успішний вихід' });
  } catch (error) {
   const errorMessage = error instanceof Error ? error.message : 'Помилка при виході';
   res.status(500).json({ error: errorMessage });
  }
 }
```

Реалізований механізм дозволяє забезпечити безпечний та зручний доступ користувачів до системи. Використання *JWT* зменшує навантаження на сервер, оскільки не потрібно зберігати сесії, а поділ логіки на *router*, *controller* та *service* робить код структурованим, зрозумілим та готовим до масштабування.

2.3 Розробка функціоналу фільтрації та отримання каталогу товарів

У межах проєкту було реалізовано повноцінний функціонал роботи з товарами, що дозволяє створювати, редагувати, видаляти та переглядати товари, а також отримувати категорії і підкатегорії для зручного відображення на клієнтській частині застосунку. Кожна операція побудована на чіткій бізнес-логіці та надає контроль над коректністю даних. Основна ідея полягала в тому, щоб кожна взаємодія користувача із товарами була прозорою та передбачуваною, а система гарантувала стабільність і безпеку даних.

Створення нового товару відбувається через ендпоінт *POST /goods*. Користувач надсилає запит із параметрами товару, серед яких обов'язковими є назва, категорія і ціна. Інші поля, такі як опис, підкатегорія, одиниця виміру, мінімальна кількість замовлення, наявність на складі, фотографії або специфікації, є необов'язковими. Якщо постачальник не вказаний, система автоматично підставляє ідентифікатор користувача, який робить запит, що дозволяє прив'язати товар до конкретного акаунту.

Перед збереженням у базу даних сервіс перевіряє коректність даних. Наприклад, якщо ціна не була передана або має некоректний формат, система повертає помилку. Для полів, які не були вказані, встановлюються значення за замовчуванням: валюта — *UAH*, одиниця виміру — шт, мінімальна кількість замовлення — 1, параметр *inStock — true*. Такий підхід дозволяє стандартизувати дані і запобігати некоректному збереженню товарів. Після цього створюється новий запис у базі даних *MongoDB*, і користувач отримує у відповідь інформацію про створений товар, включно з деталями постачальника, що забезпечує прозорість та контроль.

Перегляд конкретного товару реалізовано через ендпоінт *GET /goods/:id.* При отриманні запиту система звертається до бази даних для пошуку товару за унікальним ідентифікатором. Після знаходження товару система повертає деталі самого товару та інформацію про постачальника: назву компанії, контактний телефон, *email*, логотип та інші доступні дані. Якщо товар не знайдено або він був видалений, система повертає помилку з кодом 404. Така реалізація гарантує, що користувач бачить тільки актуальні та доступні товари.

Отримання списку товарів здійснюється через ендпоінт *GET /goods* з підтримкою гнучкої фільтрації, сортування та пагінації. Фільтри дозволяють обмежувати результати за категорією, підкатегорією, ціною, наявністю на складі, постачальником або ключовими словами у назві та описі. При отриманні запиту система динамічно будує запит до бази даних, додаючи лише ті умови, які були передані користувачем. Якщо вказана мінімальна або максимальна ціна, для

формування запиту використовується оператор *MongoDB $gte* та *$lte*. Якщо переданий параметр *inStock*, система враховує тільки доступні на складі товари. Для пошуку за ключовими словами застосовується текстовий індекс *$text*, що дозволяє здійснювати пошук по назві та опису товару одночасно.

Сортування товарів реалізоване за трьома параметрами: ціною, назвою та датою створення. Користувач може вказати порядок сортування, який може бути як зростаючим, так і спадним. Для обмеження обсягу даних реалізовано пагінацію, де користувач вказує номер сторінки і кількість товарів на сторінці. Система обчислює пропуск *(skip)* та обмеження *(limit)* для запиту до бази і додатково підраховує загальну кількість товарів та сторінок, що дозволяє побудувати навігацію по каталогу.

Редагування товару здійснюється через ендпоінт *PUT /goods/:id*. Користувач передає поля, які потрібно змінити, і сервіс оновлює їх у базі даних. При цьому застосовується валідація, що перевіряє формат та коректність нових даних. Після оновлення система повертає нову версію товару разом з підтвердженням успішної операції. Якщо товар не знайдено, повертається помилка. Такий підхід дозволяє оновлювати лише існуючі записи і підтримує цілісність даних.

Видалення товару реалізовано як м’яке видалення через ендпоінт *DELETE /goods/:id*. Замість повного видалення з бази даних система змінює поле *isActive* на *false*. Це дозволяє зберігати історію товарів і уникати втрати даних, якщо їх потрібно буде відновити або перевірити в майбутньому. Всі запити на отримання списку товарів або перегляд конкретного товару фільтрують неактивні записи, щоб вони не відображалися користувачеві (приклад 2.15).

```
async getGoods(filters: FilterGoodsDto = {}): Promise<{
  goods: IGood[];
  total: number;
  page: number;
  limit: number;
  totalPages: number;
}> {
  const {
```

```
  category,
  subcategory,
  minPrice,
  maxPrice,
  inStock,
  supplier,
  search,
  page = 1,
  limit = 20,
  sortBy = 'createdAt',
  sortOrder = 'desc',
} = filters;
const query: any = { isActive: true };
if (category) {
  query.category = category;
}
if (subcategory) {
  query.subcategory = subcategory;
}

if (minPrice !== undefined || maxPrice !== undefined) {
  query.price = {};
  if (minPrice !== undefined) {
    query.price.$gte = minPrice;
  }
  if (maxPrice !== undefined) {
    query.price.$lte = maxPrice;
  }
}
if (inStock !== undefined) {
  query.inStock = inStock;
}
if (supplier) {
  query.supplier = supplier;
}
if (search) {
  query.$text = { $search: search };
}
const sortOptions: any = {};
if (sortBy === 'price') {
  sortOptions.price = sortOrder === 'asc' ? 1 : -1;
} else if (sortBy === 'name') {
  sortOptions.name = sortOrder === 'asc' ? 1 : -1;
} else {
  sortOptions.createdAt = sortOrder === 'asc' ? 1 : -1;
}
const total = await Good.countDocuments(query);
```

```
  const skip = (page - 1) * limit;
  const goods = await Good.find(query)
    .sort(sortOptions)
    .skip(skip)
    .limit(limit)
    .populate('supplier', 'companyName email phone logo')
    .exec();

  return {
    goods,
    total,
    page,
    limit,
    totalPages: Math.ceil(total / limit),
  };
 }
```

Приклад 2.15 — Ендпоінт для отримання та фільтрації товарів

Для зручності роботи користувачів було реалізовано можливість отримання списку категорій і підкатегорій через ендпоінти *GET /goods/categories/list* та GET */goods/categories/:category/subcategories* (приклад 2.6). Сервіс збирає унікальні значення активних категорій і підкатегорій за допомогою методів *distinct*. Це дозволяє формувати динамічні меню та фільтри на клієнтській частині, що автоматично оновлюються при додаванні нових товарів. Наприклад, якщо платформа містить категорії «Електроніка» та «Одяг», підкатегорії будуть динамічно сформовані на основі даних у базі, і користувач зможе відразу фільтрувати товари без додаткових налаштувань.

```
 async getCategories(): Promise<string[]> {
  const categories = await Good.distinct('category', { isActive: true }).exec();
  return categories;
 }

 /**
  * Отримання підкатегорій для категорії
  */
 async getSubcategories(category: string): Promise<string[]> {
  const subcategories = await Good.distinct('subcategory', {
    category,
    isActive: true,
    subcategory: { $exists: true, $ne: null },
  }).exec();
  return subcategories;
```

```
  }
}
```

Приклад 2.16 — Функції отримання категорій та підкатегорій

Бізнес-логіка реалізованого функціоналу передбачає контроль цілісності даних на всіх етапах. Параметри, які не були вказані під час створення товару, отримують значення за замовчуванням, що забезпечує стандартизацію. Під час фільтрації та пошуку система використовує індекси та оптимізовані запити до *MongoDB*, що дозволяє швидко обробляти великі обсяги даних. Всі операції на оновлення і видалення перевіряють існування товару, щоб уникнути помилок і некоректних змін у базі даних. Видимість товарів контролюється через поле *isActive*, що гарантує, що користувач бачить лише актуальні та активні товари.

Розроблений функціонал дозволяє масштабувати платформу, забезпечує інтеграцію з іншими сервісами, наприклад, з модулем користувачів для прив’язки товарів до конкретного постачальника, або з модулем автентифікації для контролю доступу до операцій створення і редагування товарів. Всі відповіді від сервісу мають стандартизовану структуру, що дозволяє клієнтській частині легко обробляти дані, відображати товари у каталогах, формувати меню фільтрів та відсортовані списки, а також підтримувати багатосторінкові перегляди.

Завдяки такому підходу забезпечується повноцінна підтримка основних бізнес-процесів платформи: створення нових товарів із контролем даних, перегляд і пошук по каталогу з гнучкими фільтрами, оновлення і м’яке видалення, а також динамічне отримання категорій і підкатегорій. Всі ці елементи інтегровані в єдину систему, яка підтримує масштабування, контролює цілісність даних і забезпечує стабільну роботу користувацького інтерфейсу, що робить платформу зручною та надійною для користувачів і адміністраторів.

2.4 Розробка функціоналу операцій над замовленнями

Замовлення у веб-застосунку відіграють ключову роль, оскільки саме через них відбувається основна взаємодія між покупцем і постачальником. Якщо товар — це одиниця каталогу, то замовлення — це вже конкретна дія користувача, яка

запускає бізнес-процес. У системі для оптової торгівлі замовлення фіксує намір придбати певну кількість товару, передає контактні дані, адресу доставки та дозволяє постачальнику прийняти або відхилити заявку. Тому реалізація ендпоінтів для роботи із замовленнями повинна бути логічною, зрозумілою та безпечною.

Перший ендпоінт — це *POST /orders*, який відповідає за створення нового замовлення. Коли клієнт надсилає запит на цей маршрут, сервер спочатку виводить у консоль інформацію про отриманий запит та його тіло. Це корисно для перевірки того, які саме дані приходять із клієнтської частини. Далі виконується перевірка, чи існує об’єкт контролера та метод *createOrder*. Якщо з якихось причин контролер не ініціалізований, сервер повертає помилку з кодом 500, що означає внутрішню помилку сервера. Якщо все працює правильно, викликається метод *createOrder*, який обробляє логіку створення замовлення (приклад 2.17). У цьому місці можна вставити приклад коду обробки *POST*-запиту та виклику методу контролера.

```
async createOrder(req: Request, res: Response): Promise<void> {
   try {
     const userId = req.body.userId || req.user?.id; // Можна передати userId в body або з
middleware
     if (!userId) {
       res.status(401).json({ error: 'Користувач не авторизований' });
       return;
     }

     const { goodId, quantity, deliveryAddress, contactPerson, notes } = req.body;

     if (!goodId || quantity === undefined || quantity === null) {
       res.status(400).json({ error: 'ID товару та кількість обов\'язкові' });
       return;
     }

     if (typeof quantity !== 'number' || quantity <= 0) {
       res.status(400).json({ error: 'Кількість повинна бути додатним числом' });
       return;
     }

     const order = await orderService.createOrder(userId, {
       goodId,
```

```
      quantity: Number(quantity),
      deliveryAddress,
      contactPerson,
      notes,
    });

    res.status(201).json({
      message: 'Замовлення успішно створено',
      order,
    });
  } catch (error) {
    console.error('Помилка при створенні замовлення:', error);
    const errorMessage = error instanceof Error ? error.message : 'Помилка при створенні замовлення';
    res.status(400).json({ error: errorMessage });
  }
 }
```

Приклад 2.17 — Функція створення замовлення

Другий ендпоінт — *GET /orders/buyer/:buyerId*. Він призначений для отримання всіх замовлень конкретного покупця. Параметр *buyerId* передається в адресі запиту. Коли сервер отримує запит, він викликає метод *getBuyerOrders* у контролері. У свою чергу, контролер звертається до сервісу, який формує запит до бази даних і повертає список замовлень, що належать цьому покупцю. Важливо, що цей маршрут розміщено перед маршрутом */:id*, оскільки інакше система могла б сприйняти слово *buyer* як ідентифікатор замовлення (приклад 2.18). У цьому фрагменті можна вставити приклад коду виклику методу отримання замовлень покупця.

```
 async getBuyerOrders(buyerId: string, filters?: { status?: string }): Promise<IOrder[]> {
  const query: any = { buyer: buyerId };
  if (filters?.status) {
    query.status = filters.status;
  }

  return Order.find(query)
    .populate('supplier', 'companyName email phone logo')
    .populate('good', 'name photos price currency unit')
    .sort({ createdAt: -1 })
    .exec();
 }
```

Приклад 2.18 — Функція отримання замовлення покупця

Третій ендпоінт — *GET /orders/supplier/:supplierId*. Його призначення аналогічне попередньому, але він повертає список замовлень для постачальника. Параметр *supplierId* також передається в адресі. Контролер викликає метод *getSupplierOrders*, який формує відповідний запит до бази даних і повертає всі замовлення, де цей користувач є постачальником. Як і у випадку з покупцем, цей маршрут розташований перед загальним маршрутом */:id,* щоб уникнути конфлікту шляхів (приклад 2.19). Тут можна вставити приклад коду отримання замовлень постачальника.

```
    async getSupplierOrders(supplierId: string, filters?: { status?: string }): Promise<IOrder[]> {
   const query: any = { supplier: supplierId };
   if (filters?.status) {
     query.status = filters.status;
   }

   return Order.find(query)
     .populate('buyer', 'companyName email phone')
     .populate('good', 'name photos price currency unit')
     .sort({ createdAt: -1 })
     .exec();
 }
```

Приклад 2.19 — Функція отримання замовлень продавця

Четвертий ендпоінт — *GET /orders/:id*. Він використовується для отримання одного конкретного замовлення за його ідентифікатором. Після отримання запиту контролер викликає метод *getOrderById*, який знаходить замовлення у базі даних. Додатково перевіряється, чи користувач має право доступу до цього замовлення. Перегляд дозволений лише покупцю або постачальнику, які беруть участь у цій угоді. Якщо замовлення не знайдено, повертається помилка з кодом 404. Якщо користувач не має доступу — повертається помилка з кодом 403. У цьому місці можна вставити приклад коду пошуку замовлення за *ID* та перевірки прав доступу (приклад 2.20).

```
 async getOrderById(req: Request, res: Response): Promise<void> {
   try {
     const { id } = req.params;
     const userId = req.query.userId as string || req.user?.id;
     if (!userId) {
```

```
      res.status(401).json({ error: 'Користувач не авторизований' });
      return;
    }

    const order = await orderService.getOrderById(id, userId);
    if (!order) {
      res.status(404).json({ error: 'Замовлення не знайдено' });
      return;
    }

    res.status(200).json({ order });
  } catch (error) {
    const errorMessage = error instanceof Error ? error.message : 'Помилка при отриманні замовлення';
    if (errorMessage.includes('Немає доступу')) {
      res.status(403).json({ error: errorMessage });
    } else {
      res.status(500).json({ error: errorMessage });
    }
  }
 }
```

Приклад 2.20 — Функція отримання замовлення по ідентифікатору

Останній ендпоінт — *PATCH /orders/:id/status*. Він призначений для оновлення статусу замовлення. Зазвичай змінювати статус може лише постачальник, який отримав замовлення. У запиті передається новий статус, а також за потреби коментар постачальника. Після отримання запиту контролер викликає метод *updateOrderStatus*, який перевіряє, чи належить це замовлення даному постачальнику. Якщо перевірка проходить успішно, статус оновлюється та зберігається в базі даних. У відповідь клієнту повертається оновлений об’єкт замовлення. У цьому фрагменті можна вставити приклад коду оновлення поля статусу та збереження змін (приклад 2.21).

```
 async updateOrderStatus(req: Request, res: Response): Promise<void> {
  try {
    const { id } = req.params;
    const { status, supplierNotes } = req.body;
    const supplierId = req.body.supplierId || req.user?.id;

    if (!supplierId) {
      res.status(401).json({ error: 'Постачальник не авторизований' });
      return;
    }
```

```
    if (!status) {
      res.status(400).json({ error: 'Статус обов\'язковий' });
      return;
    }
        const order = await orderService.updateOrderStatus(id, supplierId, status, supplierNotes);
    if (!order) {
      res.status(404).json({ error: 'Замовлення не знайдено' });
      return;
    }

    res.status(200).json({
      message: 'Статус замовлення оновлено',
      order,
    });
  } catch (error) {
    const errorMessage = error instanceof Error ? error.message : 'Помилка при оновленні статусу';
    if (errorMessage.includes('Немає доступу')) {
      res.status(403).json({ error: errorMessage });
    } else {
      res.status(500).json({ error: errorMessage });
    }
  }}
```

Приклад 2.21 — Функція оновлення статусу замовлення

Таким чином, реалізований набір ендпоінтів забезпечує повний набір операцій над замовленнями: створення, перегляд одного замовлення, отримання списку замовлень для різних ролей та оновлення статусу. Правильний порядок оголошення маршрутів, перевірки наявності контролера і зручність підтримки системи. Завдяки цьому модуль замовлень працює стабільно та відповідає потребам оптового інтернет-магазину.

2.5 Розробка функціоналу збереження файлів

У інтернет-магазині збереження файлів є важливою частиною функціоналу. Особливо це стосується зображень товарів, адже саме вони допомагають покупцеві оцінити продукцію перед оформленням замовлення. У

нашій системі реалізовано можливість завантаження одного або кількох зображень на сервер із подальшим збереженням їх у файловій системі.

Основна мета цього модуля — прийняти файл від клієнта, перевірити його тип і розмір, зберегти у визначеній папці на сервері та повернути у відповіді шлях до цього файлу. Для реалізації цього функціоналу використано проміжну функцію, яка відповідає за обробку файлів, та окремий контролер, який повертає відповідь клієнту.

Почнемо з налаштування зберігання файлів. Спочатку визначається шлях до папки, в якій будуть зберігатися зображення. Для цього використовується модуль *path*, щоб правильно сформувати шлях незалежно від операційної системи. Також використовується функція створення папки, яка перевіряє, чи існує потрібна директорія, і якщо ні — створює її автоматично. Це важливо, щоб уникнути помилки при першому завантаженні файлу. У цьому місці можна вставити приклад коду визначення шляху до папки та створення директорії через *mkdir* (приклад 2.22).

```
import multer from 'multer';
import { fileURLToPath } from 'url';
import { dirname, join } from 'path';
import { mkdir } from 'fs/promises';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

// Шлях до папки з зображеннями
const imagesDir = join(__dirname, '../utils/images');

// Налаштування зберігання файлів
const storage = multer.diskStorage({
  destination: async (req, file, cb) => {
    try {
      // Переконаємося, що папка існує
      await mkdir(imagesDir, { recursive: true });
      cb(null, imagesDir);
    } catch (error) {
      cb(error instanceof Error ? error : new Error('Помилка створення папки'), '');
    }
  },
export const upload = multer({
```

```
  storage,
  fileFilter,
  limits: {
    fileSize: 5 * 1024 * 1024, // 5MB максимум
  },
});
export const uploadSingle = upload.single('photo');
export const uploadMultiple = upload.array('photos', 10); // Максимум 10 файлів
```

Приклад 2.22 — Функція отримання файлів із запиту

Далі налаштовується механізм зберігання файлів. Визначається, куди саме зберігати файл і як його називати. Для уникнення конфліктів імен кожному файлу генерується унікальна назва. Вона складається з назви поля форми, поточного часу та випадкового числа. Це гарантує, що навіть якщо два користувачі завантажать файл з однаковою назвою, вони не перезапишуть один одного. У цьому місці можна вставити приклад коду з функцією *filename*, де формується унікальне ім’я файлу (при).

```
  filename: (req, file, cb) => {
    // Генеруємо унікальне ім'я файлу
    const uniqueSuffix = `${Date.now()}-${Math.round(Math.random() * 1E9)}`;
    const extension = file.originalname.split('.').pop();
    cb(null, `${file.fieldname}-${uniqueSuffix}.${extension}`);
  },});
```

Приклад 2.23 — Генерація унікального імені файлу

Наступним важливим етапом є перевірка типу файлу. Оскільки система призначена для завантаження зображень товарів, необхідно обмежити можливість завантаження інших типів файлів. Для цього використовується фільтр, який перевіряє *MIME*-тип файлу (приклад 2.23). Якщо тип не починається зі слова *image*, сервер повертає помилку. Це підвищує безпеку застосунку та запобігає завантаженню небажаного контенту.

```
const fileFilter = (req: any, file: Express.Multer.File, cb: multer.FileFilterCallback) => {
  // Дозволяємо тільки зображення
  if (file.mimetype.startsWith('image/')) {
    cb(null, true);
  } else {
    cb(new Error('Тільки зображення дозволені!'));
  }
};
```

Приклад 2.24 — Перевірка типу файлу

Окрім типу файлу, обмежується також його розмір. У налаштуваннях встановлено максимальний розмір 5 мегабайт. Якщо користувач спробує завантажити файл більшого розміру, сервер автоматично поверне помилку. Такий підхід допомагає уникнути перевантаження сервера та економить дисковий простір (приклад 2.25).

```
export const upload = multer({
  storage,
  fileFilter,
  limits: {
    fileSize: 5 * 1024 * 1024, // 5MB максимум
  },
});
```

Приклад 2.25 — Перевірка розміру файлу

Після налаштування проміжної функції створюється контролер, який обробляє запити на завантаження файлів. Перший ендпоінт відповідає за завантаження одного зображення. Коли клієнт надсилає POST-запит на відповідний маршрут, спочатку обробляє файл, а потім керування передається до функції контролера. Якщо виникає помилка під час завантаження, наприклад файл перевищує допустимий розмір або має неправильний тип, сервер повертає статус 400 з описом помилки. Якщо файл не було передано, також повертається відповідне повідомлення. У цьому місці можна вставити приклад коду з методу *uploadSingle*, де перевіряється наявність *req.file* і формується відповідь (приклад 2.26).

```
async uploadSingle(req: Request, res: Response): Promise<void> {
   uploadSingle(req, res, (err) => {
     if (err) {
       res.status(400).json({ error: err.message });
       return;
     }

     if (!req.file) {
       res.status(400).json({ error: 'Файл не завантажено' });
       return;
     }
```

```
    // Повертаємо шлях до файлу відносно папки images
    const fileName = req.file.filename;
    res.status(200).json({
      message: 'Файл успішно завантажено',
      filename: fileName,
      path: `/images/${fileName}`,
    });
  });
}
```

Приклад 2.26 — Функція завантаження файлу

Якщо файл успішно збережений, сервер повертає повідомлення про успішне завантаження, а також ім’я файлу і шлях до нього. Шлях формується відносно папки, яка використовується для зберігання зображень. Це дозволяє клієнтській частині легко відобразити зображення, використовуючи повернений *URL*.

Другий ендпоінт реалізує завантаження кількох зображень одночасно. Це особливо зручно для товарів, які мають декілька фотографій. Проміжна функція приймає масив файлів, після чого контролер перевіряє, чи дійсно файли були передані. Якщо масив порожній, повертається помилка. Якщо файли присутні, формується список об’єктів, кожен з яких містить ім’я файлу та шлях до нього. У цьому місці можна вставити приклад коду з методу *uploadMultiple*, де формується масив *uploadedFiles* (приклад 2.27).

```
async uploadMultiple(req: Request, res: Response): Promise<void> {
  uploadMultiple(req, res, (err) => {
    if (err) {
      res.status(400).json({ error: err.message });
      return;
    }

    if (!req.files || (Array.isArray(req.files) && req.files.length === 0)) {
      res.status(400).json({ error: 'Файли не завантажено' });
      return;
    }

    const files = Array.isArray(req.files) ? req.files : [req.files];
    const uploadedFiles = files.map((file) => ({
      filename: file.filename,
      path: `/images/${file.filename}`,
    }));
```

```
    res.status(200).json({
      message: 'Файли успішно завантажено',
      files: uploadedFiles,
    });
  })}
```

Приклад 2.27 — Функція завантаження кількох файлів

Таким чином, реалізований функціонал дозволяє безпечно та зручно завантажувати файли на сервер. Кожен файл проходить перевірку типу та розміру, отримує унікальне ім’я та зберігається у визначеній папці. У відповідь клієнт отримує інформацію, необхідну для подальшого використання файлу, наприклад для відображення зображення товару на сторінці каталогу.

Розроблений модуль легко розширюється. У майбутньому можна додати підтримку хмарного зберігання, автоматичну оптимізацію зображень або створення зменшених копій для попереднього перегляду. Проте навіть у поточному вигляді система повністю покриває потреби інтернет-магазину та забезпечує стабільну й безпечну роботу із завантаженням файлів.

3 РОЗРОБКА КЛІЄНСЬКОЇ ЧАСТИНИ WEB-ЗАСТОСУНКУ

3.1 Розробка сторінки авторизації

Сторінка авторизації є однією з найважливіших частин будь-якого веб-застосунку, оскільки саме через неї користувач отримує доступ до функціоналу системи. У нашому випадку платформа призначена для взаємодії постачальників та покупців, тому авторизація виконує не лише роль входу до системи, а й забезпечує безпечну ідентифікацію бізнес-користувачів. Реалізація цієї сторінки передбачає підтримку двох режимів: входу в існуючий обліковий запис та реєстрації нового користувача з багатокроковою формою.

Сторінка реалізована як клієнтський компонент, що дозволяє використовувати локальний стан та обробляти події без перезавантаження сторінки. На початку файлу оголошується директива *'use client'*, яка вказує, що компонент *виконується* на стороні браузера. Далі імпортуються необхідні хуки та допоміжні компоненти. У цьому місці можна вставити приклад початку компонента сторінки авторизації з використанням *useState* [14] та *useRouter* [4].

Для керування логікою сторінки використовується два основних стани: режим авторизації та поточний крок реєстрації. Змінна *mode* визначає, чи знаходиться користувач у режимі входу *(login)*, чи реєстрації *(register)*. Це дозволяє динамічно перемикати інтерфейс без переходу на іншу сторінку. Змінна *currentStep* відповідає за номер кроку у багатокроковій формі реєстрації. Такий підхід робить інтерфейс більш зручним, оскільки замість однієї великої форми користувач заповнює інформацію поступово (приклад 3.1).

```
'use client';
import { useState } from 'react';
import { useRouter } from 'next/navigation';
import LoginForm from '@/app/components/auth/LoginForm';

type AuthMode = 'login' | 'register';
export default function AuthPage() {
  const router = useRouter();
  const [mode, setMode] = useState<AuthMode>('login');
  const [currentStep, setCurrentStep] = useState(1);
```

Приклад 3.1 — Початкові налаштування сторінки

Окремо зберігається стан *formData*, який містить усі дані форми реєстрації. У ньому зберігаються *email*, пароль, назва компанії, телефон, контактна особа та юридична адреса. Структура вкладених об’єктів дозволяє логічно згрупувати інформацію та підготувати її для передачі на сервер у зручному форматі. У цьому місці можна вставити приклад оголошення об’єкта *formData* зі всіма вкладеними полями (приклад 3.2).

```
const [formData, setFormData] = useState({
  email: '',
  password: '',
  confirmPassword: '',
  companyName: '',
  phone: '',
  contactPerson: {
    fullName: '',
    position: '',
    phone: '',
    email: '',
  },
  legalAddress: {
    street: '',
    city: '',
    region: '',
    postalCode: '',
    country: 'Україна',
  },
});
```

Приклад 3.2 — Поля форми

Для перемикання між кроками використовуються функції *handleNext* та *handleBack*. Вони змінюють значення *currentStep*, не дозволяючи виходити за межі допустимої кількості кроків [9]. Такий підхід робить форму контрольованою та передбачуваною. У тексті роботи можна вставити приклад реалізації цих функцій (приклад 3.3).

```
const handleNext = () => {
  if (currentStep < totalSteps) {
    setCurrentStep(currentStep + 1); }};
const handleBack = () => {if (currentStep > 1) {
    setCurrentStep(currentStep - 1);} };
```

Приклад 3.3 — Методи для просування між кроками

Процес реєстрації завершується функцією *handleSubmit*. Перед відправкою даних на сервер виконується перевірка обов’язкових полів. Це важливо для покращення користувацького досвіду, оскільки помилки виявляються одразу, без зайвого запиту до сервера. Перевіряється наявність назви компанії, телефону, даних контактної особи та юридичної адреси. Якщо будь-яке поле не заповнене, генерується помилка (приклад 3.4).

```
// Валідація обов'язкових полів перед відправкою
if (!formData.companyName?.trim()) {
  throw new Error('Назва компанії обов\'язкова');
}
if (!formData.phone?.trim()) {
  throw new Error('Телефон компанії обов\'язковий');
}
if (!formData.contactPerson?.fullName?.trim()) {
  throw new Error('ПІБ контактної особи обов\'язковий');
}
if (!formData.contactPerson?.position?.trim()) {
  throw new Error('Посада контактної особи обов\'язкова');
}
if (!formData.contactPerson?.phone?.trim()) {
  throw new Error('Телефон контактної особи обов\'язковий');
}
if (!formData.legalAddress?.street?.trim()) {
  throw new Error('Вулиця юридичної адреси обов\'язкова');
}
if (!formData.legalAddress?.city?.trim()) {
  throw new Error('Місто обов\'язкове');
}
if (!formData.legalAddress?.region?.trim()) {
  throw new Error('Область обов\'язкова');
}
if (!formData.legalAddress?.postalCode?.trim()) {
  throw new Error('Поштовий індекс обов\'язковий');
}
if (!formData.legalAddress?.country?.trim()) {
  throw new Error('Країна обов\'язкова');
}
```

Приклад 3.4 — Валідація на наявність помилок у формі

Після успішної валідації формується об’єкт *registrationData*, який відповідає структурі, що очікується сервером. Дані групуються у вкладений

об’єкт *businessProfile*, що містить інформацію про компанію. Такий підхід дозволяє чітко відокремити облікові дані від бізнес-профілю (приклад 3.5).

```
const registrationData = {
    email: formData.email,
    password: formData.password,
    businessProfile: {
     companyName: formData.companyName.trim(),
     phone: formData.phone.trim(),
     contactPerson: {
      fullName: formData.contactPerson.fullName.trim(),
      position: formData.contactPerson.position.trim(),
      phone: formData.contactPerson.phone.trim(),
      email: formData.contactPerson.email?.trim() || undefined,
     },
     legalAddress: {
      street: formData.legalAddress.street.trim(),
      city: formData.legalAddress.city.trim(),
      region: formData.legalAddress.region.trim(),
      postalCode: formData.legalAddress.postalCode.trim(),
      country: formData.legalAddress.country.trim(),
     },
    },
   };
```

Приклад 3.5 — Об’єкт для відправки на сервер

Відправка запиту виконується за допомогою функції *fetch*. Вказується метод *POST*, заголовок *Content-Type* та тіло запиту у форматі *JSON*. Після отримання відповіді додатково перевіряється тип контенту, щоб переконатися, що сервер повернув *JSON*. Якщо сервер повернув некоректну відповідь, генерується помилка. У цьому місці можна вставити приклад виконання *POST*-запиту до */auth/register* (приклад 3.6).

```
const response = await fetch(`${API_URL}/auth/register`, {
  method: 'POST',
  headers: {
   'Content-Type': 'application/json',
  },
  credentials: 'include', // Для отримання cookies
  body: JSON.stringify(registrationData),
 });
```

Приклад 3.6 — Запит на сервер

Після успішної реєстрації зберігається ідентифікатор користувача та токен доступу у *localStorage* [16]. Це дозволяє зберегти стан авторизації навіть після перезавантаження сторінки. Далі виконується перенаправлення користувача на сторінку товарів за допомогою *router.push*. У текст можна вставити приклад збереження токена та виконання перенаправлення (приклад 3.7).

```
if (!response.ok) {
    const error = await response.json();
    throw new Error(error.error || 'Помилка реєстрації');
  }

  const data = await response.json();
  console.log('Реєстрація успішна:', data);

  // Зберігаємо дані користувача
  if (data.user?.id) {
    localStorage.setItem('userId', data.user.id);
  }
  if (data.accessToken) {
    localStorage.setItem('accessToken', data.accessToken);
  }
```

Приклад 3.7 — Перенаправлення до списку після успішної реєстрації

Режим входу реалізований окремим компонентом *LoginForm*. Такий поділ коду дозволяє зробити сторінку більш структурованою та зрозумілою. У компоненті зберігаються локальні стани для *email*, пароля, помилок та індикатора завантаження (приклад 3.8).

```
<form onSubmit={handleSubmit} className="space-y-4">
    <div>
      <label htmlFor="login-email" className="block text-sm font-medium text-gray-700 mb-2">
        Email
      </label>
      <input
        id="login-email"
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        className={`w-full px-4 py-3 border rounded-lg bg-white text-gray-900 placeholder-gray-500 focus:outline-none focus:ring-2 transition-colors ${
          errors.email
            ? 'border-red-500 focus:ring-red-500'
```

```
            : 'border-gray-300 focus:ring-[#ff6b35]'
          }`}
          placeholder="your@email.com"
          disabled={isLoading}
        </div>
        <div>
          <label htmlFor="login-password" className="block text-sm font-medium text-gray-700 mb-2">
            Пароль
          </label>
          <input
            id="login-password"
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
           className={`w-full px-4 py-3 border rounded-lg bg-white text-gray-900 placeholder-gray-500 focus:outline-none focus:ring-2 transition-colors ${
              errors.password
                ? 'border-red-500 focus:ring-red-500'
                : 'border-gray-300 focus:ring-[#ff6b35]'
            }`}
            placeholder="Введіть пароль"
            disabled={isLoading}
          />
        </div>

        <div className="flex items-center justify-between">
          <label className="flex items-center">
            <input
              type="checkbox"
              className="w-4 h-4 text-[#ff6b35] border-gray-300 rounded focus:ring-[#ff6b35]"
            />
            <span className="ml-2 text-sm text-gray-600">Запам'ятати мене</span>
          </label>
        </div>
        <button
          type="submit"
          disabled={isLoading}
          className="w-full bg-[#ff6b35] hover:bg-[#ff8555] text-white font-semibold py-3 px-6 rounded-lg transition-colors duration-200 disabled:opacity-50 disabled:cursor-not-allowed flex items-center justify-center gap-2"
        >
          <>
            Увійти)}
        </button>
      </form>
```

Приклад 3.8 — *HTML*-розмітка форми логіну

Обробник *handleSubmit* у формі входу запобігає стандартному перезавантаженню сторінки та викликає передану з батьківського компонента функцію *onLogin*. У разі помилки від сервера вона відображається під полем пароля. Таким чином користувач одразу бачить причину невдалого входу. Тут можна вставити приклад реалізації *handleSubmit* у компоненті входу (приклад 3.9).

```
const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault();

  if (!validate()) {
    return;
  }

  setIsLoading(true);
  try {
    await onLogin(email, password);
  } catch (error) {
    setErrors({
      password: error instanceof Error ? error.message : 'Помилка входу',
    });
  } finally {
    setIsLoading(false);
  }
};
```

Приклад 3.9 — Відправка форми входу

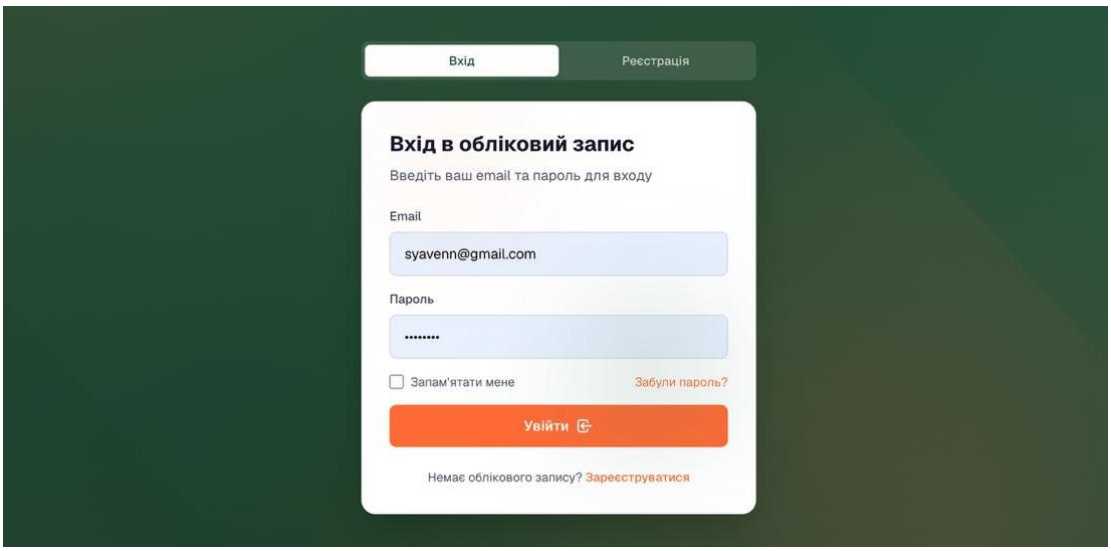

Рисунок 3.1 – Сторінка авторизації

Запит на вхід виконується аналогічно до реєстрації. Дані передаються на сервер за адресою */auth/login*, після чого перевіряється відповідь. Якщо авторизація успішна, зберігається токен доступу та виконується перехід на сторінку товарів. У цьому місці можна вставити приклад коду функції *handleLogin*.

Окрему увагу приділено інтерфейсу користувача. Для перемикання між режимами входу та реєстрації використовується кнопковий перемикач у верхній частині сторінки. Активний режим підсвічується, що дозволяє користувачу чітко розуміти, в якому стані він знаходиться. Під час реєстрації відображається прогрес-бар, який показує поточний крок і загальну кількість кроків. Це покращує зручність користування та зменшує відчуття перевантаженості форми.

Кнопки мають стан завантаження, який відображається під час виконання запиту. Це запобігає повторному надсиланню форми та інформує користувача, що система обробляє запит. Для цього використовується локальний стан *isLoading*, який блокує поля введення та змінює текст кнопки. У текст можна вставити приклад умовного відображення знаку завантаження.

Таким чином, розроблена сторінка авторизації забезпечує повний цикл взаємодії користувача з системою: від створення нового бізнес-акаунта до входу в існуючий профіль. Вона поєднує багатокрокову реєстрацію, клієнтську валідацію, обробку помилок сервера та збереження стану авторизації. Архітектурне розділення на основний компонент сторінки та окремий компонент форми входу робить код зрозумілим і зручним для підтримки. Реалізований функціонал відповідає вимогам безпеки та зручності використання, що є критично важливим для платформи електронної комерції.

3.2 Розробка сторінки товарів

Сторінка товарів є однією з центральних частин усього web-застосунку, оскільки саме через неї користувач взаємодіє з основним функціоналом системи — переглядом, пошуком та додаванням товарів. Під час її розробки головна увага

приділялася зручності користування, логічній структурі та зрозумілості інтерфейсу навіть для користувача, який вперше відкрив сайт.

Для коректної роботи з даними було створено інтерфейс, який описує структуру товару. Кожен товар має унікальний ідентифікатор, назву, опис, категорію, підкатегорію, ціну, валюту, одиницю вимірювання, інформацію про наявність на складі, фотографії, характеристики, ліцензії та дані про постачальника. Такий детальний опис дозволяє чітко визначити, які саме дані приходять із сервера та як вони повинні відображатися на сторінці. Типізація також допомагає уникнути помилок під час розробки та спрощує підтримку коду в майбутньому.

Сторінка будується навколо основного компонента, який відповідає за завантаження даних з серверної частини та керування станом інтерфейсу. При першому відкритті сторінки відбувається автоматичний запит до *API* для отримання списку товарів. У цей момент користувач бачить індикатор завантаження, що повідомляє про те, що система працює та отримує дані. Після успішного отримання відповіді список товарів відображається на екрані. Якщо ж виникає помилка, користувач отримує зрозуміле повідомлення про проблему.

Особливу роль на сторінці відіграє система фільтрів. Вона дозволяє користувачу швидко знайти потрібний товар серед великої кількості позицій. Фільтрація реалізована за кількома параметрами: пошук за назвою або описом, вибір категорії та підкатегорії, обмеження за мінімальною та максимальною ціною, а також перевірка наявності товару на складі. Коли користувач змінює будь-який фільтр, сторінка автоматично оновлює список товарів відповідно до нових умов. Це відбувається завдяки повторному запиту до серверної частини з передачею параметрів у вигляді рядка запиту (приклад 3.10).

```
useEffect(() => {
  const fetchGoods = async () => {
    setLoading(true);
    setError(null);

    try {
```

```
      const params = new URLSearchParams();
      if (filters.search) params.append('search', filters.search);
      if (filters.category) params.append('category', filters.category);
      if (filters.subcategory) params.append('subcategory', filters.subcategory);
      if (filters.minPrice) params.append('minPrice', filters.minPrice);
      if (filters.maxPrice) params.append('maxPrice', filters.maxPrice);
      if (filters.inStock) params.append('inStock', filters.inStock);
      params.append('sortBy', filters.sortBy);
      params.append('sortOrder', filters.sortOrder);
      params.append('page', currentPage.toString());
      params.append('limit', '12');

      const response = await fetch(`${API_URL}/goods?${params.toString()}`);

      if (!response.ok) {
        throw new Error('Помилка завантаження товарів');
      }

      const data: GoodsResponse = await response.json();
      setGoods(data.goods);
      setTotalPages(data.totalPages);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Невідома помилка');
    } finally {
      setLoading(false);
    }
  };
  fetchGoods();
```

Приклад 3.10 — Отримання списку товарів

Окремо реалізовано логіку завантаження категорій та підкатегорій. Спочатку система отримує список усіх доступних категорій. Коли користувач обирає конкретну категорію, додатково виконується запит для отримання відповідних підкатегорій. Якщо категорію скасовано, список підкатегорій автоматично очищується. Такий підхід забезпечує логічний зв’язок між даними та запобігає вибору некоректних значень (приклад 3.11).

```
 useEffect(() => {
   const fetchCategories = async () => {
     try {
       const response = await fetch(`${API_URL}/goods/categories/list`);
       if (response.ok) {
         const data = await response.json();
         setCategories(data.categories || []);
```

```
      }
    } catch (err) {
      console.error('Помилка завантаження категорій:', err);
    }
  };
  fetchCategories();
 }, [API_URL]);

 // Завантаження підкатегорій
 useEffect(() => {
  const fetchSubcategories = async () => {
    if (!filters.category) {
      setSubcategories([]);
      return;
    }
    try {
const response = await
fetch(`${API_URL}/goods/categories/${encodeURIComponent(filters.category)}/subcatego
ries`);
      if (response.ok) {
        const data = await response.json();
        setSubcategories(data.subcategories || []);
      }
    } catch (err) {
      console.error('Помилка завантаження підкатегорій:', err);
    }
  };
  fetchSubcategories();
 }, [filters.category, API_URL]);
```

Приклад 3.11 — Завантаження категорій і підкатегорій

Крім фільтрації, сторінка підтримує сортування результатів. Користувач може обрати, за яким параметром сортувати товари — за датою створення, ціною або назвою. Також можна змінити порядок відображення — від нових до старих або навпаки. Сортування реалізується через передачу відповідних параметрів до серверної частини, що дозволяє виконувати обробку без перевантаження клієнта зайвими обчисленнями (приклад 3.12).

```
      params.append('sortBy', filters.sortBy);
      params.append('sortOrder', filters.sortOrder);
      params.append('page', currentPage.toString());
      params.append('limit', '12');
```

Приклад 3.12 — Сортування товарів

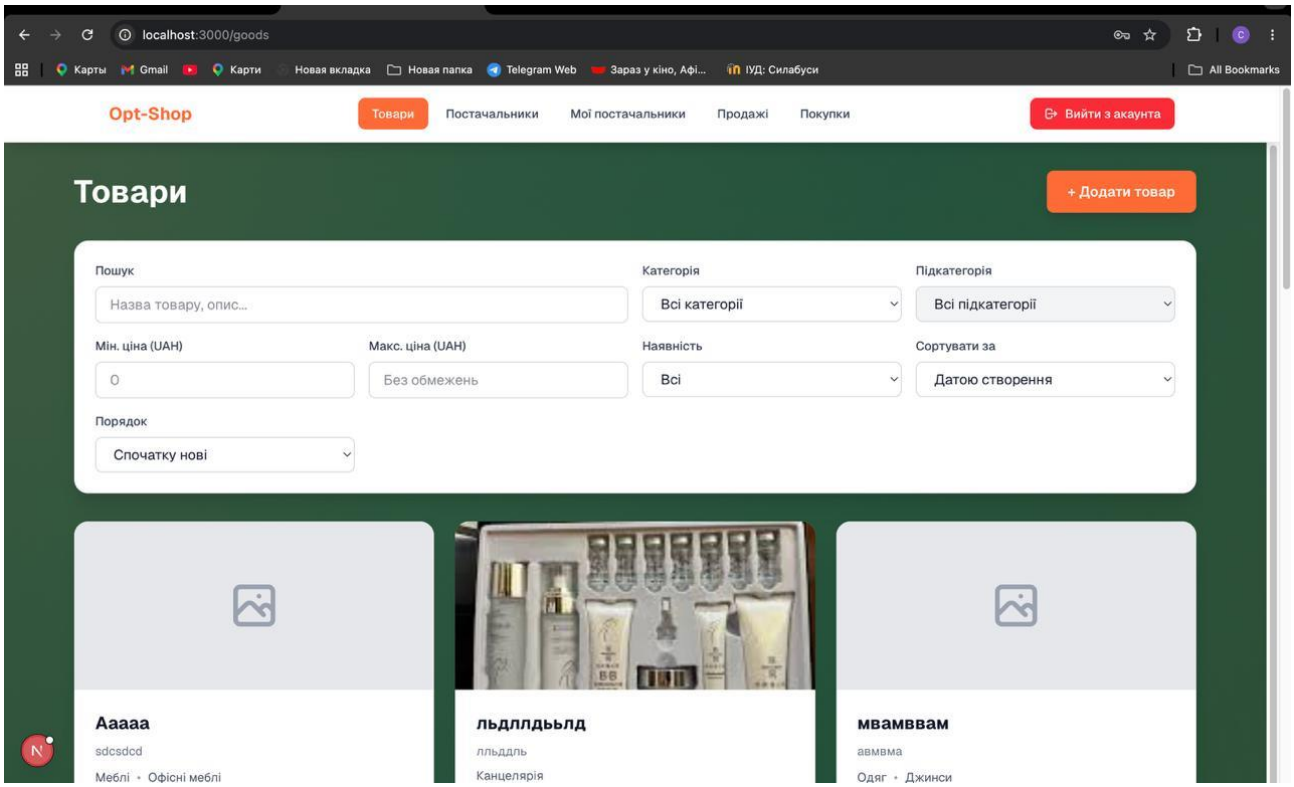

Рисунок 3.2 – Сторінка товарів

Щоб не перевантажувати сторінку великою кількістю даних, реалізовано механізм пагінації. Це означає, що товари відображаються частинами, наприклад по 12 позицій на сторінку. Користувач може переходити між сторінками за допомогою кнопок «Попередня» та «Наступна». Номер поточної сторінки та загальна кількість сторінок також відображаються в інтерфейсі. При переході на іншу сторінку система знову виконує запит до серверу з відповідним номером сторінки (приклад 3.13).

```
{/* Пагінація */}
      {totalPages > 1 && (
        <div className="flex justify-center gap-2 mt-8">
          <button
            onClick={() => setCurrentPage(prev => Math.max(1, prev - 1))}
            disabled={currentPage === 1}
              className="px-4 py-2 bg-white/10 hover:bg-white/20 text-white rounded-lg disabled:opacity-50 disabled:cursor-not-allowed transition-colors"
          >
            Попередня
          </button>
          <span className="px-4 py-2 bg-white/10 text-white rounded-lg">
            Сторінка {currentPage} з {totalPages}</span>
          <button
```

```
            onClick={() => setCurrentPage(prev => Math.min(totalPages, prev + 1))}
            disabled={currentPage === totalPages}
               className="px-4 py-2 bg-white/10 hover:bg-white/20 text-white rounded-lg
disabled:opacity-50 disabled:cursor-not-allowed transition-colors"
          >Наступна
          </button></div>
```

Приклад 3.13 — Кнопки пагінації

Таким чином, сторінка товарів поєднує в собі зручний інтерфейс, гнучку систему фільтрації та сортування та підтримку пагінації. Завдяки розділенню логіки на окремі компоненти та чіткій структурі даних забезпечується масштабованість і простота подальшого розвитку функціоналу. Навіть користувач, який вперше відкриває сторінку, може легко зрозуміти її структуру та швидко знайти необхідний товар.

3.3 Розробка сторінки покупок

Сторінка «Мої покупки» призначена для відображення всіх замовлень, які користувач зробив у системі. Вона є важливою частиною функціоналу, оскільки дозволяє покупцю контролювати свої замовлення, відстежувати їх статус, переглядати деталі доставки та комунікацію з постачальником. Під час розробки цієї сторінки основна увага приділялася зрозумілості інтерфейсу, простоті навігації та логічній структурі відображення інформації.

Сторінка реалізована як клієнтський компонент, що означає, що всі дані завантажуються динамічно після відкриття сторінки без її повного перезавантаження. Коли користувач переходить до розділу «Мої покупки», система автоматично отримує його ідентифікатор із *localStorage*. Це потрібно для того, щоб визначити, які саме замовлення належать конкретному користувачу. Якщо ідентифікатор відсутній, система повідомляє про необхідність авторизації. Таким чином забезпечується базовий контроль доступу до персональних даних (приклад 3.14).

```
useEffect(() => {
  const checkAuth = () => {
    const userId = localStorage.getItem('userId');
```

```
    if (!userId) {
      router.push('/auth');
      return false;}
    return true;};
  const isAuth = checkAuth();
  setIsAuthenticated(isAuth);
  setIsChecking(false);
 }, [router]);
```

Приклад 3.14 — Перевірка авторизації

Після перевірки авторизації відбувається запит до серверної частини для отримання списку замовлень. Поки дані завантажуються, користувач бачить індикатор завантаження. Це важливий елемент інтерфейсу, який показує, що система працює та обробляє запит. Якщо виникає помилка під час отримання даних, замість списку відображається повідомлення з поясненням проблеми. Якщо ж замовлень немає, користувач бачить відповідний текст, який інформує про відсутність покупок (приклад 3.15).

```
 useEffect(() => {
  const fetchOrders = async () => {
    const userId = localStorage.getItem('userId');
    if (!userId) {
      setError('Користувач не авторизований');
      setLoading(false);
      return;
    }

    setLoading(true);
    setError(null);

    try {
      const url = statusFilter
        ? `${API_URL}/orders/buyer/${userId}?status=${statusFilter}`
        : `${API_URL}/orders/buyer/${userId}`;

      const response = await fetch(url, {
        credentials: 'include',
      });

      if (!response.ok) {
        throw new Error('Помилка завантаження замовлень');
      }
```

```
      const data = await response.json();
      setOrders(data.orders || []);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Невідома помилка');
    } finally {
      setLoading(false);
    }
  };
  fetchOrders();
 }, [statusFilter, API_URL]);
```

Приклад 3.15 — Завантаження замовлень

Для коректної роботи з даними було описано структуру замовлення. Кожне замовлення містить інформацію про покупця, постачальника, товар, кількість, ціну за одиницю, загальну суму, валюту, статус, адресу доставки, контактну особу, примітки та дату створення і оновлення (приклад 3.16).

```
  <div className="flex gap-4 flex-1">
              {/* Фото товару */}
              {order.good.photos && order.good.photos.length > 0 ? (
                <img
                  src={`${API_URL}${order.good.photos[0]}`}
                  alt={order.good.name}
                  className="w-24 h-24 object-cover rounded-lg"
                />
              ) : (
              <div className="w-24 h-24 bg-gray-200 rounded-lg flex items-center justify-
center">
                  <span className="text-gray-400 text-xs">Немає фото</span>
                </div>
              )}

              {/* Інформація про товар */}
              <div className="flex-1">
                                  <h3 className="text-xl font-bold text-gray-900 mb-
2">{order.good.name}</h3>
                <div className="flex items-center gap-4 text-sm text-gray-600 mb-2">
                  <span>Кількість: {order.quantity} {order.good.unit}</span>
                  <span>Ціна за одиницю: {order.unitPrice} {order.currency}</span>
                </div>
                <div className="flex items-center gap-4">
                  <span className="text-lg font-bold text-[#ff6b35]">
                    Загальна сума: {order.totalPrice} {order.currency}
                  </span>
                </div>
```

```
        </div>
      </div>
```

Приклад 3.16 — *HTML*-розмітка замовлення

Окрему увагу приділено статусам замовлення. Кожне замовлення може мати один із кількох станів: очікує підтвердження, підтверджено, обробляється, відправлено, доставлено або скасовано. Для зручності користувача кожному статусу відповідає текстове пояснення та окреме кольорове оформлення. Наприклад, статус «доставлено» відображається зеленим кольором, а «скасовано» — червоним. Це дозволяє швидко зорієнтуватися в ситуації навіть без детального перегляду інформації.

```
<div className="flex flex-col items-end gap-2">
          <span
                        className={`px-3 py-1 rounded-full text-sm font-medium
${statusColors[order.status]}`}
          >
            {statusLabels[order.status]}
          </span>
          <span className="text-sm text-gray-600">
            {formatDate(order.createdAt)}
          </span>
        </div>
      </div>
```

Приклад 3.17 — HTML-розмітка статусу замовлення

На сторінці також реалізовано можливість фільтрації замовлень за статусом. У верхній частині розміщено випадаючий список, де користувач може обрати конкретний стан замовлення або переглянути всі замовлення одразу. Коли користувач змінює фільтр, система автоматично виконує новий запит до сервера з відповідним параметром. Таким чином список оновлюється відповідно до вибраного статусу. Це особливо зручно, коли кількість замовлень велика і потрібно швидко знайти, наприклад, лише ті, що знаходяться в процесі доставки (приклад 3.18).

```
    <div className="bg-white rounded-2xl shadow-xl p-6 mb-8">
      <div className="flex items-center gap-4">
            <label className="text-sm font-medium text-gray-700">Фільтр за
статусом:</label><select
        value={statusFilter}
```

```
          onChange={(e) => setStatusFilter(e.target.value)}
            className="px-4 py-2 border border-gray-300 rounded-lg focus:outline-none
focus:ring-2 focus:ring-[#ff6b35] bg-white text-gray-900">
          <option value="">Всі замовлення</option>
          <option value="pending">Очікує підтвердження</option>
          <option value="confirmed">Підтверджено</option>
          <option value="processing">Обробляється</option>
          <option value="shipped">Відправлено</option>
          <option value="delivered">Доставлено</option>
          <option value="cancelled">Скасовано</option>
        </select>
      </div> </div>
```

Приклад 3.18 — Фільтри замовлень за статусом

Нижче основної інформації відображаються дані про постачальника. Користувач бачить назву компанії, контактний телефон, а за наявності — логотип. Це важливо для швидкої ідентифікації, особливо якщо покупець співпрацює з кількома постачальниками одночасно.

Для покращення зручності реалізовано можливість розгортання детальної інформації про замовлення. Коли користувач натискає на картку, вона відкриває додатковий блок із розширеними даними. У цьому блоці може відображатися адреса доставки, контактна особа, примітки покупця та примітки від постачальника. Також показується дата останнього оновлення замовлення. Такий підхід дозволяє не перевантажувати інтерфейс зайвою інформацією, але водночас надає доступ до повних деталей за потреби (приклад 3.19).

```
 {selectedOrder?._id === order._id && (
      <div className="mt-6 pt-6 border-t border-gray-200 space-y-4">
          {order.deliveryAddress && ( <div>
  <h4 className="text-sm font-semibold text-gray-700 mb-2">Адреса доставки:</h4>
       <p className="text-sm text-gray-600">
{order.deliveryAddress.street}, {order.deliveryAddress.city},
{order.deliveryAddress.region}, {order.deliveryAddress.postalCode},
{order.deliveryAddress.country}</p></div>  )}
                {order.contactPerson && (<div>
  <h4 className="text-sm font-semibold text-gray-700 mb-2">Контактна особа:</h4>
                  <p className="text-sm text-gray-600">
                   {order.contactPerson.fullName}, {order.contactPerson.phone}
                   {order.contactPerson.email && `, ${order.contactPerson.email}`}
                  </p>
                </div> )}
```

```
                {order.notes && (
                  <div>
        <h4 className="text-sm font-semibold text-gray-700 mb-2">Ваші примітки:</h4>
                    <p className="text-sm text-gray-600">{order.notes}</p>
                  </div>)}
                {order.supplierNotes && (
                  <div>
                    <h4 className="text-sm font-semibold text-gray-700 mb-2">Примітки
від постачальника:</h4>
                    <p className="text-sm text-gray-600">{order.supplierNotes}</p>
                  </div>  )}
                <div className="text-xs text-gray-500">
                  Оновлено: {formatDate(order.updatedAt)}
                </div></div> )}
```

Приклад 3.19 — HTML-розмітка детальної інформації по замовленню

Інтерфейс сторінки адаптивний, тобто правильно відображається як на великих екранах комп’ютера, так і на мобільних пристроях. Розташування блоків змінюється залежно від ширини екрану, що забезпечує комфортне використання на різних пристроях.

Таким чином, сторінка «Мої покупки» дозволяє користувачу повністю контролювати процес своїх замовлень: переглядати їх, фільтрувати за статусом, відкривати деталі та відстежувати зміни. Інтерфейс побудований логічно та зрозуміло навіть для людини, яка вперше користується системою. Чітка структура даних, динамічне оновлення інформації та зручний дизайн роблять роботу з замовленнями простою та ефективною.

ВИСНОВКИ

У результаті виконання даної роботи було розроблено повноцінний web-застосунок для оптового інтернет-магазину, який забезпечує взаємодію між постачальниками та бізнес-клієнтами в онлайн-середовищі. У процесі реалізації проєкту було проаналізовано предметну область, визначено основні вимоги до системи та спроєктовано її загальну архітектуру.

Під час виконання роботи було реалізовано клієнтську частину застосунку з інтуїтивно зрозумілим інтерфейсом, що забезпечує зручну навігацію, перегляд товарів, оформлення замовлень та контроль за здійсненими покупками. Було створено сторінки реєстрації та авторизації користувачів, каталог товарів, сторінку детального перегляду товару, кошик, а також розділ «Мої покупки» для відстеження статусу замовлень.

Окрім клієнтської частини, було реалізовано серверну логіку, яка забезпечує обробку запитів, взаємодію з базою даних та збереження інформації про користувачів, товари й замовлення. Розроблено структуру бази даних з урахуванням особливостей оптової торгівлі, що дозволяє ефективно зберігати великі обсяги даних і підтримувати зв’язки між різними сутностями web-застосунку. Також було впроваджено механізми перевірки доступу до персональних даних і базову обробку помилок для підвищення надійності роботи застосунку.

У ході роботи було застосовано сучасні web-технології, що забезпечують швидкодію, масштабованість та гнучкість системи. Реалізована архітектура дозволяє у майбутньому розширювати функціонал, додавати нові модулі або інтегрувати систему з іншими сервісами без суттєвої зміни основної структури.

Таким чином, поставлена мета була досягнута, а всі основні завдання виконані. Створений web-застосунок є функціональним, правильно організованим та готовим до подальшого розвитку. Отримані результати демонструють практичну цінність розробленої системи та можливість її використання в реальних умовах оптової торгівлі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. About Node.js. [Електронний ресурс]. — Режим доступу: https://nodejs.org/en/about
2. Controllers in ExpressJS. [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/web-tech/controllers-in-expressjs/.
3. Express - Node.js web application framework. [Електронний ресурс]. — Режим доступу: https://expressjs.com/.
4. Functions: useRouter. [Електронний ресурс]. — Режим доступу: https://nextjs.org/docs/pages/api-reference/functions/use-router.
5. Introduction to Tailwind CSS. [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/css/introduction-to-tailwind-css/.
6. JSON Web Token (JWT). [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/web-tech/json-web-token-jwt/.
7. JSON: що це за формат і як з ним працювати. [Електронний ресурс]. — Режим доступу: https://goit.global/ua/articles/json-shcho-tse-za-format-i-iak-z-nym-pratsiuvaty/.
8. MongoDB: що це за СУБД, переваги та недоліки. [Електронний ресурс]. — Режим доступу: https://goit.global/ua/articles/mongodb-shcho-tse-take-ta-yak-prazuie/.
9. Pagination Design Pattern. [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/blogs/pagination-design-pattern/.
10. React Foundations: About React and Next.js. [Електронний ресурс]. — Режим доступу: https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs.
11. REST API Introduction. [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/node-js/rest-api-introduction/.
12. Server-side rendering (SSR) - Glossary - MDN Web Docs. [Електронний ресурс]. — Режим доступу: https://developer.mozilla.org/en-US/docs/Glossary/SSR.

13. TypeScript Introduction. [Електронний ресурс]. — Режим доступу: https://www.w3schools.com/typescript/typescript_intro.php.
14. useState – React. [Електронний ресурс]. — Режим доступу: https://react.dev/reference/react/useState.
15. What Are Back-End Services, and What Do They Include? [Електронний ресурс]. — Режим доступу: What Are Back-End Services, and What Do They Include?.
16. What is Local Storage? [Електронний ресурс]. — Режим доступу: https://docs.streamlit.io/develop/api-reference.
17. What is Middleware? [Електронний ресурс]. — Режим доступу: https://aws.amazon.com/what-is/middleware/.