**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

**До захисту допущено:**
**Завідувач кафедри**
_____Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

# Дипломний проєкт

**на здобуття ступеня бакалавра**

**за освітньо-професійною програмою “Комп’ютерні системи та мережі”**

**спеціальності 123 “Комп’ютерна інженерія”**

на тему: Веб-застосунок управління магазином військового спорядження

Виконав : студент __4__ курсу, групи _ІО-12_
(шифр групи)

_____Добровольський Федір Володимирович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент Меленчуков М. Є.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль)доктор філософії Міщенко Л.Д., ас. кафедри ОТ
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали)
_____
(підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

**ЗАТВЕРДЖУЮ**
**Завідувач кафедри**
_____Михайло НОВОТАРСЬКИЙ
(підпис)
“__” _____2025 р.

**ЗАВДАННЯ**
на бакалаврський дипломний проєкт студента

Добровольського Федора Володимировича

1. Тема проєкту *Веб-застосунок управління магазином військового спорядження*
керівник проєкту Меленчуков Микита Євгенович, Асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від _____

2. Термін здачі студентом закінченого проєкту *30 травня 2025 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Аналіз предметної області та існуючих рішень.

Розділ 2. Огляд технологій для розробки.

Розділ 3. Реалізація проєкту.

Розділ 4. Тестування та оцінка ефективності розробленої веб-платформи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема, схема бази даних.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| Нормоконтроль | Міщенко Л. Д. | | |
| | | | |

7. Дата видачі завдання «3» січня 2025 р.

Календарний план

| № п/п | Найменування етапів дипломного проєкту | Терміни виконання етапів проєкту | Примітки |
|---|---|---|---|
| 1. | *Затвердження теми проєкту* | *24.12.2024-30.12.2024* | |
| 2. | *Вивчення та аналіз завдання* | *31.12.2024-10.03.2025* | |
| 3. | *Розробка архітектури та загальної структури системи* | *10.03.2025-20.03.2025* | |
| 4. | *Розробка структур окремих підсистем* | *20.03.2025-05.04.2025* | |
| 5. | *Програмна реалізація системи* | *05.04.2025-10.05.2025* | |
| 6. | *Оформлення пояснювальної записки* | *15.05.2025-06.05.2025* | |
| 7. | *Захист програмного продукту* | *06.06.2025* | |
| 8. | *Передзахист* | *11.06.2025* | |
| 9. | *Захист* | *18.06.2025* | |

Студент-дипломник _____ Федір ДОБРОВОЛЬСЬКИЙ
(підпис)

Керівник проєкту _____ Микита МЕЛЕНЧУКОВ
(підпис)

## АНОТАЦІЯ

Дипломний проект присвячено розробці веб-застосунку для інтернет-магазину тактичного спорядження. В умовах війни, зростаючого попиту на спеціалізоване оснащення та необхідності зручного і дешевого доступу до нього, створення такої платформи є актуальним. Метою роботи є реалізація інтуїтивно зрозумілого та функціонального веб-сервісу, що надасть користувачам можливість легко знаходити та купувати товари, а адміністраторам і власникам – ефективно керувати процесом.



**Ключові слова:** веб-застосунок, інтернет-магазин, тактичне спорядження, спеціалізоване спорядження, електронна комерція, Angular, TypeScript, управління товарами, інтерфейс користувача.



## ANNOTATION

The diploma project is dedicated to the development of a web application for the online store of tactical equipment. In the conditions of ongoing war, growing demand for specialized equipment and the need for convenient and cheap access to it, the creation of such a platform is relevant. The goal of the work is to implement an intuitive and functional web service that will allow users to easily find and purchase products, for administrators and owners to effectively manage the assortment.



**Keywords**: web application, online store, tactical equipment, specialized equipment, e-commerce, Angular, TypeScript, product management, user interface.

| справки | Формат | *Значення* | *Найменування* | Кіл. листів | № екземпля | *Додаток* |
|---|---|---|---|---|---|---|
| | | | Документація загальна | | | |
| | | | Знову розроблена | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.002 ТЗ*** | Веб-застосунок | 4 | | |
| | | | управління магазином | | | |
| | | | військового спорядження | | | |
| | | | Технічне завдання | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.003 ПЗ*** | Веб-застосунок | 66 | | |
| | | | управління магазином | | | |
| | | | військового спорядження | | | |
| | | | Пояснювальна записка | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.004 Д1*** | Веб-застосунок | 1 | | |
| | | | управління магазином | | | |
| | | | військового спорядження | | | |
| | | | Структурна схема | | | |
| | | | системи | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.4672008.005 Д2*** | Веб-застосунок | 1 | | |
| | | | управління магазином | | | |
| | | | військового спорядження | | | |
| | | | Функціональна схема | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.006 Д3*** | Веб-застосунок | 1 | | |
| | | | управління магазином | | | |
| | | | військового спорядження | | | |
| | | | Схема бази даних | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.007 Д4*** | Веб-застосунок | 15 | | |
| | | | управління магазином | | | |
| | | | військового спорядження | | | |
| | | | Текст програмного коду | | | |
| | | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| *Зм* | *Лист* | *№ докум.* | *Підп* | *Дата* | |

# *ІАЛЦ.467200.001 ОА*

| | | | | | |
|---|---|---|---|---|---|
| ***Розроб*** | | Добровольський Ф.В. | | | |
| ***Перев*** | | Меленчуков М.Є. | | | |
| | | | | | |

*Веб-застосунок управління магазином військового спорядження*
Опис альбому

| Літ. | Аркуш | Аркушів |
|---|---|---|
| | 1 | 1 |

***НТУУ "КПІ" ФІОТ***
***ІО-12***

# ТЕХНІЧНЕ ЗАВДАННЯ
# ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок управління магазином військового спорядження*»

Київ – 2025

# ЗМІСТ



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.002 ТЗ | | |
| | | № докум. | Підпис | Дата | | | |
| Розробив | | Добровольський Ф.В. | | | **Веб-застосунок управління магазином військового спорядження** Технічне завдання | Літ. | Аркуш 1 / Аркушів 4 |
| Перевірив | | Меленчуков М.Є. | | | | | |
| Н. Контр. | | Міщенко Л.Д. | | | | НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-12 | |
| Затвердив | | | | | | | |

# 1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання стосується розробки веб-застосунку для управління інтернет-магазином військового та тактичного спорядження

Областю застосування системи є сфера електронної комерції, зокрема роздрібна онлайн-торгівля спеціалізованим екіпіруванням та супутніми товарами. Платформа призначена для використання потенційними покупцями (військовослужбовці, співробітники силових структур, мисливці, туристи, учасники тактичних ігор), а також адміністративним персоналом магазину для управління товарним асортиментом. Вона забезпечує інструменти для перегляду каталогу товарів, їх детального опису, додавання до кошика та управління адміністратором товарними позиціями. Система може бути використана як онлайн-вітрина для існуючого фізичного магазину або як самостійний інтернет-магазин.

# 2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання дипломного проєкту по освітньо-професійній програмі «Комп’ютерні системи та мережі» спеціальності 123 «Комп’ютерна інженерія», затверджене кафедрою Обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут імені Ігоря Сікорського».

# 3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка веб-застосунку інтернет-магазину, що дозволятиме покупцям переглядати каталог товарів військового спорядження, ознайомлюватися з їх характеристиками, формувати кошик та оформлювати замовлення, а адміністраторам – керувати товарним асортиментом.

# 4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є публікації та статті в мережі Інтернет на тему електронної комерції та розробки веб-застосунків, офіційна документація технологій, що використовуються (Angular, TypeScript), науково-технічна література, а також аналіз функціоналу існуючих інтернет-магазинів.

# 5 ТЕХНІЧНІ ВИМОГИ

## 5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Мати простий та інтуїтивно зрозумілий інтерфейс як для покупців, так і для адміністраторів.
- Надати можливість усім відвідувачам переглядати каталог товарів та детальну інформацію про кожен товар (назва, опис, ціна, зображення, категорія).
- Забезпечити можливість фільтрації товарів за категоріями.
- Надати можливість додавати товари до віртуального кошика.
- Надати можливість переглядати вміст кошика, змінювати кількість обраних товарів та видаляти товари з кошика.
- Забезпечити захищену систему входу для адміністратора магазину.
- Надати адміністратору можливість додавати нові товари до каталогу, вказуючи їх назву, категорію, ціну, опис та назву файлу зображення.
- Надати адміністратору можливість переглядати список усіх наявних товарів.
- Надати адміністратору можливість видаляти товари з каталогу.
- Забезпечити збереження даних про товари та стан кошика.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

### 5.2. Вимоги до програмного забезпечення

- **Клієнтська частина (браузер користувача):** Сучасний веб-браузер з підтримкою актуальних веб-стандартів (наприклад, Google Chrome, Mozilla Firefox, Safari, Microsoft Edge останніх версій).
- **Середовище розробки:**
  - Операційна система: Windows 10/11, macOS, або дистрибутив Linux.
  - Node.js (рекомендована версія LTS).
  - Angular CLI (останньої стабільної версії).
  - Інтегроване середовище розробки: Visual Studio Code (рекомендовано) або інше IDE з підтримкою TypeScript та Angular.

### 5.3. Вимоги до апаратної частини

- **Для користувача:** Пристрій (ПК, ноутбук, планшет, смартфон) з доступом до мережі Інтернет та встановленим сучасним веб-браузером.
- **Для розробника:** ЦП Intel Core i3/AMD Ryzen 3 (або аналогічний); RAM 8 GB (рекомендовано 16 GB); SSD/HDD 20 GB вільного місця.

# 6 ЕТАПИ РОЗРОБКИ

| Назва етапів виконання | Термін виконання |
|---|---|
| Затвердження теми роботи | *24.12.2024-30.12.2024* |
| Вивчення та аналіз завдання | *31.12.2024-10.03.2025* |
| Розробка архітектури та загальної структури системи | *10.03.2025-20.03.2025* |
| Розробка структур окремих частин системи | *20.03.2025-05.04.2025* |
| Програмна реалізація системи | *05.04.2025-10.05.2025* |
| Виправлення помилок | *15.05.2025-06.05.2025* |
| Оформлення пояснювальної записки | *15.05.2025-06.05.2025* |

# ПОЯСНЮВАЛЬНА ЗАПИСКА
# ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок управління магазином військового спорядження*»

Київ – 2025

ЗМІСТ

# ПЕРЕЛІК СКОРОЧЕНЬ

| | |
|---|---|
| API | (Application Programming Interface) Інтерфейс прикладного програмування |
| CLI | (Command Line Interface) Інтерфейс командного рядка |
| CRUD | (Create, Read, Update, Delete) Базові операції для роботи з даними |
| CSS | (Cascading Style Sheets) Каскадні таблиці стилів |
| DB | (Database) База даних |
| DOM | (Document Object Model) Об'єктна модель документа |
| HTML | (HyperText Markup Language) Мова гіпертекстової розмітки |
| HTTP | (Hypertext Transfer Protocol) Протокол передачі гіпертексту |
| IDE | (Integrated Development Environment) Інтегроване середовище розробки |
| JSON | (JavaScript Object Notation) Нотація об'єктів JavaScript |
| LTS | (Long-Term Support) Довготривала підтримка (наприклад, для версій Node.js) |
| ORM | (Object-Relational Mapper) Технологія об'єктно-реляційного відображення |
| SPA | (Single Page Application) Односторінковий веб-застосунок |

# ВСТУП

Забезпечення доступу до якісного та спеціалізованого спорядження є критично важливим, особливо в сучасних умовах, що характеризуються підвищеними вимогами до індивідуального екіпірування як для професійного використання, так і для активного відпочинку. З розвитком електронної комерції та Інтернету, можливості для придбання таких товарів значно розширилися, проте водночас зріс ризик натрапити на неякісну продукцію або незручні сервіси. Особливої актуальності це набуває в Україні, де через війну існує сталий попит на надійне тактичне та військове спорядження.

У зв'язку з цим виникає потреба у створенні спеціалізованих, функціональних та зручних для користувача вітчизняних інтернет-магазинів. Розробка веб-застосунку, призначеного для управління магазином військового спорядження, має на меті не лише комерційну функцію, але й сприяння оперативному забезпеченню споживачів необхідним екіпіруванням. Цей розділ присвячений аналізу предметної області електронної комерції у даній ніші, вивченню потреб цільової аудиторії та огляду існуючих рішень.

Потенційними користувачами розроблюваного веб-застосунку є військовослужбовці, співробітники силових структур, волонтери, учасники спортивно-тактичних ігор, мисливці, туристи, а також адміністратори магазину, відповідальні за управління товарним асортиментом та взаємодію з клієнтами. Платформа має на меті надати їм інтуїтивно зрозумілий інструмент для вибору, замовлення та управління спеціалізованими товарами.

# РОЗДІЛ 1
# АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

## 1.1 Опис предметної області: магазини військового спорядження

### 1.1.1 Сучасний стан сфери продажу військового спорядження

Сфера продажу військового та тактичного спорядження в останні роки демонструє суттєве зростання та трансформацію, що зумовлено низкою глобальних та регіональних факторів. Особливо помітні ці зміни в Україні, де безпекові виклики та підвищений інтерес суспільства до питань оборони та готовності до надзвичайних ситуацій стимулювали значне збільшення попиту на відповідну продукцію. Якщо раніше основними споживачами такого спорядження були переважно представники силових структур та вузьке коло ентузіастів (мисливці, учасники військово-тактичних ігор), то сьогодні аудиторія значно розширилася, включивши волонтерів, резервістів, та й просто громадян, що прагнуть мати базовий набір для особистої безпеки. Цей підвищений попит стимулює як зростання кількості продавців, так і розширення асортименту пропонованих товарів.

Важливою тенденцією є поступовий, але впевнений перехід торгівлі цим специфічним видом товарів у онлайн-середовище. Електронна комерція надає значні переваги як для продавців, так і для покупців. Продавці отримують можливість охопити значно ширшу географію клієнтів, зменшити витрати на утримання фізичних торгових площ та оптимізувати логістичні процеси. Покупці, в свою чергу, отримують доступ до великого вибору товарів з будь-якої точки країни, можливість порівнювати ціни та характеристики, ознайомлюватися з відгуками та робити замовлення у зручний для себе час. Для багатьох категорій тактичного спорядження, особливо для рідкісних або

вузькоспеціалізованих позицій, інтернет-магазини часто є єдиним доступним каналом придбання.

Разом з тим, онлайн-торгівля військовим спорядженням стикається з певними викликами. Одним з ключових є питання довіри та гарантії якості. Оскільки спорядження часто використовується в екстремальних умовах, його надійність є першочерговою. В онлайн-середовищі покупцеві складніше перевірити якість товару до покупки, тому репутація магазину, наявність сертифікатів (де це доречно), детальні описи та якісні зображення продукції відіграють надзвичайно важливу роль. Проблема контрафактної або низькоякісної продукції також присутня на ринку, що вимагає від сумлінних продавців додаткових зусиль для підтвердження автентичності своїх товарів.

Іншим важливим аспектом є специфіка самого товару. Багато позицій тактичного спорядження (наприклад, бронежилети, спеціалізований одяг, оптика) потребують певних знань для правильного вибору та експлуатації. Тому інтернет-магазини, що прагнуть бути успішними, часто надають розширені консультації, публікують огляди, статті та відеоматеріали, допомагаючи клієнтам зробити усвідомлений вибір. Це вимагає від персоналу магазину високого рівня компетенції у товарній групі.

Логістика також є викликом, особливо для габаритних товарів або товарів, що мають певні обмеження на перевезення. Забезпечення швидкої та надійної доставки по всій країні є важливим конкурентним фактором. Крім того, не можна ігнорувати вплив технологій на саму продукцію – постійно з'являються нові матеріали, конструктивні рішення, що підвищують функціональність та надійність спорядження. Інтернет-магазин має оперативно реагувати на ці зміни, оновлюючи свій асортимент та надаючи актуальну інформацію про новинки.

Зростає роль онлайн-спільнот, форумів та соціальних мереж, де користувачі обмінюються досвідом використання спорядження, залишають відгуки та рекомендації. Інтеграція з такими спільнотами або створення власної

може стати ефективним інструментом для просування магазину та формування лояльної клієнтської бази. Таким чином, сучасний стан сфери продажу військового спорядження характеризується динамічним розвитком онлайн-сегменту, підвищеними вимогами до якості товарів та сервісу, а також необхідністю постійної адаптації до мінливих потреб ринку та технологічних інновацій.

### 1.1.2 Визначення ключових понять

Для чіткого розуміння предметної області та подальшого проектування веб-застосунку "Recon Depot" необхідно визначити ряд ключових понять, що використовуються у контексті інтернет-магазинів військового та тактичного спорядження. Ці визначення допоможуть встановити єдину термінологічну базу та уникнути неоднозначностей.

Перш за все, військове спорядження охоплює широкий спектр предметів, призначених для використання збройними силами та іншими військовими формуваннями. Це можуть бути елементи особистого екіпірування солдата (форма, взуття, головні убори), засоби індивідуального балістичного захисту (бронежилети, шоломи), тактичне розвантажувальне спорядження (плитоноски, ремінно-плечові системи, підсумки), засоби маскування, спеціальні інструменти, оптика, засоби зв'язку та навігації, а також інші предмети, що забезпечують виконання бойових та навчальних завдань. Важливо зазначити, що в рамках цивільного інтернет-магазину продаж летальної зброї та деяких категорій військового обладнання може бути обмежений або заборонений законодавством.

Поняття тактичне спорядження часто перетинається з військовим, проте має і свої особливості. Воно розробляється з акцентом на максимальну функціональність, надійність, довговічність та зручність використання в складних та екстремальних умовах. Окрім військових, тактичне спорядження

активно використовується співробітниками правоохоронних органів, рятувальних служб, приватних охоронних структур, а також цивільними особами для специфічних потреб, таких як полювання, екстремальний туризм, військово-тактичні ігри (страйкбол, пейнтбол) або для забезпечення особистої готовності до надзвичайних ситуацій. До тактичного спорядження можна віднести спеціалізований одяг, взуття, рюкзаки, ножі, мультитули, ліхтарі, засоби для виживання та багато іншого.

Інтернет-магазин у даному контексті – це веб-застосунок, що функціонує як віртуальна торгова площадка, дозволяючи продавцю демонструвати товари, а покупцям – ознайомлюватися з асортиментом, обирати необхідні позиції, формувати замовлення та (в ідеалі) здійснювати оплату онлайн. Ключовими атрибутами ефективного інтернет-магазину є зручний каталог, детальний опис товарів, система пошуку та фільтрації, кошик для покупок та процес оформлення замовлення.

Каталог товарів є центральною частиною інтернет-магазину. Це структурована сукупність усіх товарних позицій, доступних для продажу, згрупованих за категоріями для полегшення навігації та пошуку. Кожен товар у каталозі супроводжується відповідною інформацією, такою як назва, опис, характеристики, ціна, зображення.

Користувацький інтерфейс (UI) та досвід користувача (UX) є критично важливими для успіху будь-якого інтернет-магазину, особливо для такого, що продає специфічні товари. UI – це те, як виглядає сайт і як користувач взаємодіє з його елементами (кнопки, меню, форми). UX – це загальне враження та задоволеність користувача від взаємодії з сайтом, включаючи легкість знаходження інформації, простоту оформлення замовлення та загальну зручність.

Платформа електронної комерції або система управління контентом (CMS) для e-commerce – це програмне забезпечення, яке лежить в основі інтернет-магазину і надає інструменти для його створення та управління. Це

можуть бути готові рішення (наприклад, Shopify, WooCommerce, Magento) або кастомно розроблені системи, як у випадку проекту "Recon Depot", де для клієнтської частини використовується Angular, а для серверної частини передбачається розробка на Node.js.

База даних (БД) відіграє ключову роль у зберіганні всієї інформації інтернет-магазину. У контексті "Recon Depot", припускаючи використання SQL, це буде реляційна база даних, де зберігатимуться дані про товари (назви, описи, ціни, категорії, залишки на складі), замовлення, клієнтів (у разі реалізації реєстрації) та інша службова інформація. Структурована мова запитів SQL (Structured Query Language) використовується для взаємодії з такими базами даних.

Чітке розуміння цих термінів дозволяє сформувати єдиний контекст для подальшого обговорення вимог, проектування та реалізації веб-застосунку.

### 1.1.3. Цільова аудиторія

Визначення та розуміння цільової аудиторії є фундаментальним етапом у розробці будь-якого успішного продукту, і веб-застосунок "Recon Depot" не є винятком. Потреби, очікування та поведінкові характеристики потенційних користувачів безпосередньо впливають на формування функціональних вимог, дизайн інтерфейсу та загальну стратегію розвитку інтернет-магазину. Для "Recon Depot" можна виділити декілька ключових сегментів цільової аудиторії, кожен з яких має свої специфічні потреби.

Основною та найширшою групою є покупці – кінцеві споживачі товарів. Цю групу можна далі сегментувати. По-перше, це професійні користувачі, до яких належать військовослужбовці Збройних Сил України, бійці Національної гвардії, Сил спеціальних операцій, а також співробітники інших силових відомств, таких як Служба безпеки України, Державна прикордонна служба та Національна поліція. Для цього сегменту пріоритетними є найвища якість,

надійність, довговічність та відповідність спорядження встановленим стандартам (наприклад, ДСТУ або стандартам NATO). Вони шукають перевірені бренди, вимагають точних технічних характеристик, інформації про матеріали виготовлення, клас захисту (для бронежилетів та шоломів), сумісність з іншим екіпіруванням (наприклад, наявність системи MOLLE). Часовий фактор для них часто є критичним, тому важлива швидкість пошуку потрібного товару, чітка інформація про наявність та оперативна доставка. Також до професіоналів можна віднести співробітників приватних охоронних компаній, чиї потреби можуть дещо відрізнятися, наприклад, у бік менш помітного спорядження для прихованого носіння.

Другий важливий сегмент покупців – це волонтери та представники волонтерських організацій. Їхня основна мета – забезпечення потреб військових підрозділів або інших груп, що потребують спеціалізованого спорядження. Для волонтерів важлива можливість швидкого пошуку товарів за конкретними запитами, оптимальне співвідношення ціни та якості, а також, потенційно, можливість робити гуртові замовлення або отримувати спеціальні умови. Прозорість процесу закупівлі та надійність постачальника для них є ключовими.

Третій сегмент – це цивільні ентузіасти та особи, що ведуть активний спосіб життя. Сюди входять учасники військово-тактичних ігор, таких як страйкбол та пейнтбол, для яких важлива автентичність реплік спорядження, його функціональність в ігрових умовах та відповідність екіпіруванню певних підрозділів для реконструкторів. Також це мисливці та рибалки, які потребують спеціалізованого одягу, взуття, маскувальних засобів, ножів, оптики, що відповідають умовам полювання чи риболовлі. До цієї ж групи належать туристи, альпіністи та прихильники екстремальних видів відпочинку, для яких пріоритетом є легкість, міцність, компактність та надійність спорядження для виживання в складних умовах. Окремо можна виділити осіб, що цікавляться тематикою "виживання" (preppers) або просто дбають про особисту безпеку та

готовність до надзвичайних ситуацій, купуючи аптечки, тривожні рюкзаки, інструменти та засоби індивідуального захисту в межах чинного законодавства.

Окрім покупців, іншою важливою групою цільової аудиторії є адміністратори магазину. Це персонал, відповідальний за наповнення сайту контентом, оновлення каталогу товарів, управління цінами та залишками, а в перспективі – за обробку замовлень та комунікацію з клієнтами. Для них ключовими потребами є наявність зручної, інтуїтивно зрозумілої та функціональної адміністративної панелі, яка дозволить швидко та ефективно виконувати всі необхідні операції з управління інтернет-магазином. Важлива швидкість роботи адмін-панелі, можливість легко додавати, редагувати та видаляти товари, управляти категоріями та переглядати інформацію про активність на сайті.

Таким чином, при розробці "Recon Depot" необхідно враховувати потреби всіх цих сегментів, прагнучи створити універсальну платформу, що буде однаково зручною як для досвідченого професіонала, який точно знає, що шукає, так і для новачка, який потребує консультації та детальної інформації, а також для адміністратора, що забезпечує її безперебійну роботу.

## 1.2 Аналіз існуючих підходів

### 1.2.1 Огляд загальних платформ продажу військового спорядження

Загальні платформи електронної комерції, також відомі як маркетплейси, є великими онлайн-майданчиками, що агрегують пропозиції від численних продавців у різноманітних товарних категоріях. Прикладами таких платформ на українському ринку можуть слугувати Rozetka, Prom.ua, OLX, а на міжнародному – Amazon, eBay, Alibaba. У контексті продажу військового та тактичного спорядження, ці платформи часто мають відповідні розділи або категорії, де продавці можуть розміщувати свої товари.

Однією з ключових переваг використання таких загальних платформ для продавців є доступ до вже сформованої, великої аудиторії потенційних покупців. Користувачі часто звертаються до відомих маркетплейсів через звичку, довіру до бренду платформи та зручність здійснення покупок різних товарів в одному місці. Це може значно знизити початкові витрати на маркетинг та залучення клієнтів для окремого продавця. Крім того, великі платформи зазвичай надають готову інфраструктуру для торгівлі, включаючи системи управління товарами, інтегровані платіжні шлюзи, а іноді й власні логістичні рішення або партнерські програми доставки. Це може спростити запуск онлайн-продажів, особливо для невеликих компаній або підприємців-початківців.

Однак, продаж спеціалізованого військового та тактичного спорядження через загальні маркетплейси має і суттєві недоліки. По-перше, це відсутність глибокої спеціалізації самої платформи. Інструменти пошуку, фільтрації та категоризації на загальних маркетплейсах зазвичай є універсальними і не завжди враховують специфіку тактичного спорядження, де важливі конкретні стандарти, матеріали, системи кріплень (наприклад, MOLLE/PALS), класи захисту тощо. Це може ускладнювати для покупця пошук потрібного товару з необхідними характеристиками серед великої кількості інших пропозицій, включаючи товари широкого вжитку або низькоякісні аналоги.

По-друге, на великих маркетплейсах часто виникає проблема контролю якості та автентичності товарів, особливо якщо на платформі присутня велика кількість дрібних або неперевірених продавців. Для покупця військового спорядження, де надійність є критичним фактором, ризик придбання підробки або товару, що не відповідає заявленим характеристикам, є неприйнятним. Відсутність можливості отримати кваліфіковану консультацію від фахівця безпосередньо на платформі також є суттєвим мінусом, оскільки вибір багатьох видів тактичного спорядження потребує специфічних знань.

По-третє, висока конкуренція серед продавців на маркетплейсі часто призводить до цінових війн, що може негативно впливати на маржинальність

бізнесу. Крім того, продавці обмежені можливостями брендування власної сторінки та формування унікального клієнтського досвіду, оскільки дизайн та функціонал платформи є стандартизованими. Це ускладнює побудову довгострокових відносин з клієнтами та формування лояльності саме до конкретного продавця чи бренду спорядження, а не до маркетплейсу загалом. Комісійні збори платформи за продажі або розміщення товарів також можуть суттєво впливати на кінцеву вартість та прибутковість. Таким чином, хоча загальні платформи надають певні переваги у вигляді широкого охоплення аудиторії та готової інфраструктури, вони не завжди є оптимальним рішенням для ефективного продажу та якісного представлення спеціалізованого військового та тактичного спорядження.

### 1.2.2 Огляд спеціалізованих платформ продажу військового спорядження

На противагу загальним маркетплейсам, існують спеціалізовані інтернет-магазини та онлайн-платформи, які цілеспрямовано фокусуються на продажу військового, тактичного та аутдор спорядження. Такі платформи, як правило, створюються компаніями або ентузіастами, що мають глибокі знання у цій сфері та розуміють специфічні потреби цільової аудиторії. В Україні прикладами таких ресурсів є інтернет-магазини відомих дистриб'юторів та виробників, а також мультибрендові магазини тактичного спрямування. На міжнародному рівні також існує велика кількість подібних спеціалізованих гравців, від великих ритейлерів до нішевих бутиків.

Однією з головних переваг спеціалізованих платформ є глибина та релевантність асортименту. Вони пропонують ретельно підібраний каталог товарів, що відповідає потребам професіоналів та ентузіастів. Часто такі магазини є офіційними дистриб'юторами відомих брендів, що гарантує

автентичність та якість продукції. На відміну від загальних маркетплейсів, тут значно менший ризик натрапити на низькоякісні підробки.

Другою важливою перевагою є експертиза та якість інформаційного супроводу. Картки товарів на спеціалізованих сайтах зазвичай містять значно детальніші описи, повні технічні характеристики, інформацію про використані матеріали, стандарти, яким відповідає виріб, та особливості його застосування. Часто присутні якісні фотографії з різних ракурсів, відеоогляди, а також можливість отримати кваліфіковану консультацію від менеджерів магазину, які добре знаються на специфіці товарів. Деякі платформи ведуть власні блоги, публікують статті та посібники з вибору та використання спорядження, що підвищує їхню цінність для покупця.

Функціонал каталогу на спеціалізованих платформах також зазвичай краще адаптований до потреб цільової аудиторії. Системи фільтрації дозволяють відбирати товари за специфічними параметрами, такими як тип камуфляжу, стандарт кріплення (MOLLE/PALS), матеріал (наприклад, Cordura, Gore-Tex), виробник, клас захисту, розмір, колір тощо. Це значно спрощує процес пошуку та вибору потрібного товару порівняно з універсальними фільтрами загальних маркетплейсів.

Користувацький досвід (UX) на таких платформах може суттєво варіюватися. Деякі спеціалізовані магазини інвестують у сучасний дизайн та зручну навігацію, тоді як інші можуть мати дещо застарілий або перевантажений інтерфейс. Однак, навіть при менш досконалому дизайні, цільова аудиторія часто надає перевагу таким магазинам через їхню спеціалізацію, асортимент та експертизу. Мобільна адаптація також є важливим фактором, і більшість сучасних спеціалізованих магазинів прагнуть забезпечити зручне користування зі смартфонів та планшетів.

Додаткові сервіси, такі як програми лояльності, спеціальні умови для постійних клієнтів або представників силових структур, можливість замовлення специфічних товарів, яких немає в наявності, а також активні

спільноти навколо бренду магазину (наприклад, у соціальних мережах) сприяють формуванню довгострокових відносин з клієнтами.

Незважаючи на переваги, спеціалізовані платформи також мають свої виклики. Зазвичай вони мають менший маркетинговий бюджет порівняно з великими маркетплейсами, тому залучення нової аудиторії потребує значних зусиль у сфері SEO, контент-маркетингу та прямої реклами. Ціни на товари можуть бути дещо вищими через менші обсяги продажів або орієнтацію на преміальні бренди. Логістична інфраструктура також може бути менш розвиненою, ніж у гігантів електронної комерції.

Проте, для цільової аудиторії, що шукає специфічне військове або тактичне спорядження та цінує якість, надійність і професійну консультацію, спеціалізовані інтернет-магазини залишаються пріоритетним вибором. Розробка власного веб-застосунку в цій ніші дозволяє поєднати переваги спеціалізації з можливістю створення унікального користувацького досвіду та повного контролю над усіма аспектами онлайн-торгівлі.

# ВИСНОВОК ДО РОЗДІЛУ

У першому розділі дипломного проекту було здійснено всебічний аналіз предметної області, пов'язаної з розробкою інтернет-магазину, що спеціалізується на військовому та тактичному спорядженні. Було підкреслено зростаючу актуальність таких платформ, особливо в контексті сучасних українських реалій, де попит на якісне та надійне екіпірування значно зріс. Визначено ключові виклики, що стоять перед онлайн-торгівлею у цій специфічній ніші, включаючи необхідність забезпечення високої довіри клієнтів, надання детальної та достовірної інформації про товари, а також ефективне управління логістикою та безпекою.

Детально охарактеризовано цільову аудиторію потенційного інтернет-магазину, виділено основні сегменти споживачів, такі як професійні військові та правоохоронці, волонтери, а також цивільні ентузіасти (мисливці, туристи, учасники тактичних ігор). Проаналізовано їхні специфічні потреби та очікування від функціоналу та асортименту товарів. Також розглянуто потреби адміністративного персоналу магазину, для якого важлива зручна та ефективна система управління контентом та товарними позиціями.

Проведено аналіз існуючих підходів до онлайн-продажу військового спорядження, зокрема розглянуто переваги та недоліки використання великих загальних торгових майданчиків (маркетплейсів) та функціонування вузькоспеціалізованих інтернет-магазинів. Зроблено висновок, що спеціалізовані платформи, незважаючи на потенційно менше охоплення аудиторії, краще задовольняють специфічні потреби цільових клієнтів завдяки глибині асортименту, експертизі та якості інформаційного супроводу.

Результати проведеного аналізу предметної області та існуючих рішень підтверджують доцільність розробки власного веб-застосунку для інтернет-

магазину військового спорядження. Такий підхід дозволить створити платформу, максимально адаптовану до потреб цільової аудиторії, забезпечити високий рівень користувацького досвіду та ефективно управляти бізнес-процесами. Отримані дані та висновки слугуватимуть основою для подальшого обґрунтування вибору технологій, проектування архітектури та реалізації функціоналу запланованого веб-застосунку.

# РОЗДІЛ 2
# ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ

## 2.1 Технології клієнтської частини (Frontend)

### 2.1.1 Мова розмітки HTML та стилізація: CSS3

Основою будь-якого веб-інтерфейсу, з яким взаємодіє користувач у браузері, є мова гіпертекстової розмітки HTML та каскадні таблиці стилів CSS. Ці технології є фундаментальними для створення структури та візуального оформлення веб-сторінок. У розроблюваному веб-застосунку використовується стандарт HTML5, який надає широкий набір семантичних елементів для логічного структурування контенту. Використання таких тегів, як <header>, <footer>, <main>, <nav>, <aside>, <section>, <article>, дозволяє не тільки створити чітку та зрозумілу структуру документа для браузерів та пошукових систем, але й покращити доступність веб-застосунку для користувачів з обмеженими можливостями, які використовують допоміжні технології. HTML5 також визначає структуру форм введення даних, які є невід'ємною частиною інтернет-магазину, наприклад, у формах авторизації адміністратора, додавання та редагування товарів, а також у формі оформлення замовлення. Ці форми реалізовані за допомогою стандартних елементів, таких як <form>, <input>, <textarea>, <button> та <select>, що забезпечує їхню кросбраузерну сумісність та передбачувану поведінку. Шаблони компонентів Angular, які є основними будівельними блоками застосунку, використовують HTML для визначення своєї візуальної структури, доповнюючи її спеціальним синтаксисом Angular для динамічного відображення даних.

Для візуального оформлення та стилізації всіх елементів інтерфейсу застосовується технологія CSS3 (Cascading Style Sheets, рівень 3). CSS3 надає розробникам потужні інструменти для керування зовнішнім виглядом веб-

сторінок, значно розширюючи можливості попередніх версій. У проекті інтернет-магазину CSS3 використовується для визначення кольорової схеми, шрифтів, відступів, розмірів елементів, а також для реалізації більш складних аспектів дизайну. Ключовими можливостями CSS3, задіяними в проекті, є модулі компонування Flexbox та CSS Grid. Ці технології дозволили створити гнучкі та адаптивні макети сторінок, такі як сітка товарів у каталозі, яка автоматично підлаштовується під доступну ширину екрану, та загальна структура сторінок з бічною панеллю для категорій та основною областю для контенту. Важливим аспектом є реалізація адаптивного дизайну (Responsive Web Design) за допомогою медіа-запитів (@media). Це дозволяє веб-застосунку коректно відображатися та бути зручним у використанні на пристроях з різними розмірами екранів – від великих десктопних моніторів до планшетів та мобільних телефонів, що є критично важливим для сучасного інтернет-магазину, оскільки значна частина користувачів здійснює покупки саме з мобільних пристроїв.

CSS3 також використовується для створення візуальних ефектів та мікровзаємодій, що покращують досвід користувача. Наприклад, плавні переходи (transitions) при наведенні курсора на кнопки чи картки товарів, анімація появи повного опису товару, зміна тіней та рамок елементів – усе це реалізовано засобами CSS3. Стилі в проекті інкапсулюються на рівні окремих Angular-компонентів, що є однією з переваг фреймворку. Це запобігає конфліктам стилів між різними частинами застосунку та полегшує їхню підтримку. Крім того, використовуються глобальні стилі, визначені у файлі styles.css, для встановлення загальних правил оформлення, базових шрифтів, скидання стандартних стилів браузера та визначення загальновживаних класів, таких як стилі для активних посилань навігації. У проекті не використовувалися великі сторонні CSS-фреймворки на кшталт Bootstrap чи Materialize, що дозволило досягти більшого контролю над фінальним виглядом та оптимізувати розмір завантажуваних стилів, проте архітектура Angular

дозволяє їх легку інтеграцію за потреби. Поєднання семантично коректної розмітки HTML5 та потужних можливостей CSS3 є основою для створення візуально привабливого, функціонального, доступного та адаптивного користувацького інтерфейсу для розроблюваного інтернет-магазину військового спорядження.

### 2.1.2 Фреймворк Angular та мова TypeScript: основи та переваги для SPA

Для розробки клієнтської частини (frontend) інтернет-магазину військового спорядження було обрано платформу Angular у поєднанні з мовою програмування TypeScript. Angular, розроблений та підтримуваний компанією Google, є одним із провідних сучасних фреймворків для створення складних та високопродуктивних односторінкових веб-застосунків (SPA - Single Page Application). Вибір на користь Angular був зумовлений низкою його архітектурних переваг та можливостей, що якнайкраще відповідають вимогам проекту.

Концепція SPA полягає у створенні веб-застосунку, який завантажує основну частину ресурсів (HTML, CSS, JavaScript) один раз при першому вході користувача, а подальша навігація між різними розділами або "сторінками" відбувається динамічно на стороні клієнта, без повного перезавантаження сторінки з сервера. Це забезпечує значно швидший відгук інтерфейсу, плавні переходи та загалом покращує досвід користувача, наближаючи його до роботи з нативними десктопними або мобільними додатками. Для інтернет-магазину, де користувачі часто переглядають багато товарів, переходять між категоріями та кошиком, швидкість та плавність SPA є суттєвою перевагою.

Мова TypeScript, яка є стандартною для розробки на Angular, стала ще одним важливим фактором вибору. TypeScript є надбудовою над JavaScript, що розширює його можливостями статичної типізації, класами, інтерфейсами та

іншими конструкціями, притаманними об'єктно-орієнтованому програмуванню. Головна перевага статичної типізації полягає у можливості виявлення багатьох типів помилок ще на етапі компіляції коду, а не під час його виконання в браузері. Це значно підвищує надійність розроблюваного програмного забезпечення, спрощує процес рефакторингу та підтримки коду, особливо в міру зростання проекту. Чітко визначені типи даних для моделей (наприклад, Product, Order, CartItem), сервісів та компонентів роблять код більш зрозумілим, самодокументованим та менш схильним до помилок, пов'язаних з неправильним використанням даних. Крім того, TypeScript забезпечує чудову підтримку з боку сучасних інтегрованих середовищ розробки (IDE), таких як Visual Studio Code, надаючи розширені можливості автодоповнення, перевірки типів "на льоту" та навігації по коду.

Ключовою особливістю Angular є його компонентна архітектура. Весь застосунок будується з набору незалежних, інкапсульованих та повторно використовуваних компонентів. Кожен компонент відповідає за певну частину користувацького інтерфейсу та має власну логіку (написану на TypeScript), HTML-шаблон для відображення та CSS-стилі для оформлення. Такий підхід дозволяє розбивати складний інтерфейс на менші, керовані частини, що спрощує розробку, тестування та подальшу модифікацію. У розроблюваному інтернет-магазині компонентами є, наприклад, список товарів (ProductListComponent), меню категорій (CategoryMenuComponent), кошик (ShoppingCartComponent), адміністративна панель (AdminDashboardComponent) та інші. Важливою перевагою є те, що Angular, починаючи з новіших версій, активно просуває концепцію standalone-компонентів, яка була використана в даному проекті. Це дозволяє визначати компоненти, директиви та пайпи без необхідності їхньої явної декларації в традиційних NgModule, що спрощує структуру проекту та зменшує кількість шаблонного коду, особливо для невеликих та середніх за розміром застосунків.

Angular надає потужний механізм впровадження залежностей (Dependency Injection, DI), який дозволяє легко управляти залежностями між різними частинами застосунку. Наприклад, сервіси, що інкапсулюють бізнес-логіку або взаємодію з даними (такі як ProductService, CartService, AuthService, OrderService у нашому проекті), можуть бути легко впроваджені в компоненти через їхні конструктори. Це робить компоненти менш зв'язаними, більш тестованими та сприяє дотриманню принципу єдиної відповідальності.

Система маршрутизації (Routing) в Angular дозволяє створювати навігацію між різними логічними "сторінками" (видами) SPA, використовуючи URL-адреси браузера. Кожен маршрут пов'язаний з певним компонентом, який відображається при активації цього маршруту. У проекті інтернет-магазину маршрутизація використовується для переходу між головною сторінкою, каталогом товарів, кошиком, сторінкою оформлення замовлення та адміністративною панеллю. Також реалізовано захист маршрутів (route guards), наприклад, AuthGuard, який перевіряє, чи авторизований користувач (адміністратор) перед наданням доступу до захищених розділів, таких як панель адміністратора.

Для роботи з асинхронними операціями, такими як HTTP-запити до сервера або обробка подій користувача, Angular інтегрований з бібліотекою RxJS (Reactive Extensions for JavaScript). RxJS надає потужні інструменти для роботи з потоками даних (Observables) та подіями. У проекті Observables та BehaviorSubject (особливий тип Subject з RxJS, що зберігає поточне значення та видає його новим підписникам) використовуються в сервісах (ProductService, CartService, OrderService) для організації реактивного оновлення даних у компонентах, що на них підписані. Наприклад, коли список товарів оновлюється в ProductService після запиту до бекенду, всі компоненти, підписані на потік products$, автоматично отримують оновлені дані та перемальовують відповідні частини інтерфейсу.

Для взаємодії з серверною частиною Angular надає сервіс HttpClient, який дозволяє легко відправляти HTTP-запити (GET, POST, PUT, DELETE тощо) на API ендпоінти та отримувати відповіді. У нашому проекті HttpClient використовується в ProductService для завантаження списку товарів, додавання, редагування та видалення товарів через взаємодію з розробленим Node.js/Express.js бекендом, а також в OrderService для відправки даних замовлення.

Нарешті, інструмент командного рядка Angular CLI (Command Line Interface) значно спрощує та прискорює процес розробки. Він використовується для генерації нових проектів, компонентів, сервісів, guard-ів та інших частин застосунку, а також для компіляції, збірки, тестування та запуску сервера для розробки. Використання Angular CLI забезпечує дотримання рекомендованих практик та структури проекту.

Отже, вибір фреймворку Angular у поєднанні з мовою TypeScript надав міцну та сучасну технологічну основу для розробки клієнтської частини інтернет-магазину військового спорядження, забезпечивши можливості для створення швидкого, інтерактивного, масштабованого та легко підтримуваного веб-застосунку.

## 2.2 Технології серверної частини (Backend)

### 2.2.1 Node.js як платформа виконання

Для реалізації серверної частини веб-застосунку було обрано платформу Node.js. Node.js – це середовище виконання JavaScript, побудоване на високопродуктивному рушії JavaScript V8, тому самому, що використовується у веб-браузері Google Chrome. Ключовою особливістю Node.js, що робить його популярним вибором для розробки веб-серверів та API, є його подієво-орієнтована архітектура та неблокуюча модель операцій вводу-виводу (I/O). На відміну від традиційних багатопотокових серверних платформ, де кожен запит

може оброблятися в окремому потоці, що призводить до значних накладних витрат на управління цими потоками, Node.js використовує однопотокову модель з циклом подій (event loop). Це означає, що Node.js може ефективно обробляти велику кількість одночасних з'єднань, не створюючи новий потік для кожного з них. Коли надходить запит, що потребує виконання тривалої операції вводу-виводу (наприклад, читання файлу, запит до бази даних або до зовнішнього API), Node.js не блокує основний потік, очікуючи завершення цієї операції. Замість цього, він реєструє функцію зворотного виклику (callback), яка буде виконана після завершення операції, і продовжує обробляти інші запити. Такий підхід робить Node.js особливо ефективним для I/O-інтенсивних застосунків, якими є більшість веб-серверів, що постійно взаємодіють з базами даних та файловою системою.

Вибір Node.js для серверної частини інтернет-магазину військового спорядження був зумовлений кількома перевагами. По-перше, це висока продуктивність для типових завдань веб-сервера, таких як обробка HTTP-запитів та відповіді, взаємодія з базою даних. По-друге, це величезна екосистема модулів, доступних через менеджер пакунків npm (Node Package Manager). npm є найбільшим у світі репозиторієм програмних пакетів, що надає готові рішення для практично будь-яких завдань – від веб-фреймворків (як Express.js, що використовується в проекті) до драйверів баз даних, інструментів для автентифікації, логування та багато іншого. Це дозволяє значно прискорити процес розробки та використовувати перевірені спільнотою рішення. По-третє, використання JavaScript (або TypeScript, який компілюється в JavaScript) на серверній стороні дозволяє розробникам, знайомим з JavaScript з досвіду розробки фронтенду (наприклад, на Angular з TypeScript, як у даному проекті), легше адаптуватися до розробки бекенду, а також потенційно спільно використовувати деякі частини коду або логіки (наприклад, моделі даних чи валідаційні правила) між клієнтською та серверною частинами. Хоча для даного проекту на бекенді використовувався JavaScript, можливість легкого

переходу або інтеграції з TypeScript є додатковим плюсом. Концептуально, Node.js також надає хороші можливості для масштабування застосунку в майбутньому, наприклад, шляхом використання кластеризації або побудови мікросервісної архітектури. У контексті розробленого веб-застосунку, Node.js став основою для сервера на базі Express.js, який відповідає за обробку API-запитів від клієнтського Angular-додатку для управління товарами та, в перспективі, замовленнями.

### 2.2.2 Мова програмування для бекенду: JavaScript

Мовою програмування для реалізації серверної логіки на платформі Node.js було обрано JavaScript. Як основна мова Node.js, JavaScript надає всі необхідні інструменти для створення повноцінних серверних застосунків, включаючи веб-сервери та API. Вибір JavaScript для бекенду розроблюваного інтернет-магазину був зумовлений його широким розповсюдженням, великою спільнотою розробників та величезною кількістю доступних бібліотек і фреймворків.

Однією з ключових особливостей JavaScript, яка активно використовується в Node.js, є його асинхронна природа. Багато операцій, особливо пов'язаних із вводом-виводом (робота з файловою системою, мережеві запити, взаємодія з базою даних), є асинхронними. Це означає, що програма не зупиняє своє виконання, очікуючи завершення такої операції, а продовжує працювати, отримуючи результат операції пізніше через механізми зворотних викликів (callbacks), промісів (Promises) або синтаксису async/await. У коді серверної частини інтернет-магазину, зокрема в обробниках маршрутів Express.js, активно використовується синтаксис async/await для роботи з асинхронними операціями ORM Sequelize (яка повертає Promises). Це дозволяє писати асинхронний код у більш читабельному, послідовному стилі, що нагадує синхронний код, уникаючи так званого "пекла колбеків" (callback hell) та

роблячи логіку більш зрозумілою. Наприклад, отримання списку товарів з бази даних (await Product.findAll();) або створення нового товару (await Product.create(...);) є асинхронними операціями, результати яких очікуються без блокування основного потоку Node.js.

JavaScript є динамічно типізованою мовою, що надає гнучкість у розробці, особливо на етапі прототипування та для невеликих проектів. Хоча для великих та складних систем часто надають перевагу статично типізованим мовам (або TypeScript, який компілюється в JavaScript), для реалізації базового API управління товарами та замовленнями в рамках даного дипломного проекту можливостей JavaScript виявилося достатньо. Динамічна типізація дозволила швидко реалізувати необхідні ендпоінти та логіку взаємодії з базою даних.

Використання JavaScript на бекенді також має перевагу в тому, що розробники, які працюють над клієнтською частиною на Angular (де використовується TypeScript, що є розширенням JavaScript), вже знайомі з основними концепціями та синтаксисом мови. Це спрощує розуміння коду обох частин застосунку та потенційно дозволяє одній команді або розробнику працювати над усім стеком технологій (full-stack development). Для розроблюваного інтернет-магазину JavaScript у поєднанні з Node.js та фреймворком Express.js став ефективним інструментом для швидкої розробки RESTful API, необхідного для взаємодії з клієнтською частиною.

## 2.3 Технології баз даних

### 2.3.1 СУБД SQLite

Для зберігання даних на етапі розробки веб-застосунку було обрано систему управління базами даних SQLite. SQLite – це вбудовувана реляційна СУБД, реалізована як програмна бібліотека на мові C. Ключовою особливістю SQLite, що відрізняє її від більшості інших SQL СУБД (таких як MySQL,

PostgreSQL, Oracle), є її серверна незалежність (serverless). Це означає, що SQLite не потребує окремого серверного процесу для своєї роботи; замість цього, вона читає та записує дані безпосередньо у звичайний дисковий файл. Один файл бази даних SQLite може містити повну базу даних з таблицями, індексами, тригерами та представленнями.

Такий підхід надає низку значних переваг, особливо для локальної розробки, прототипування та для застосунків, що не потребують високої конкурентності записів або складних механізмів масштабування. По-перше, це простота налаштування та використання. Для початку роботи з SQLite не потрібно встановлювати та конфігурувати окремий сервер баз даних. Достатньо підключити відповідну бібліотеку (у випадку Node.js – пакет sqlite3), і база даних буде створена автоматично у вигляді файлу (у нашому проекті це recon_depot.sqlite в корені бекенд-частини), якщо вона ще не існує. Це значно спрощує розгортання середовища розробки та зменшує поріг входження.

По-друге, SQLite є легковажною та портативною. Вся база даних зберігається в одному файлі, що дозволяє легко її копіювати, переміщувати між різними середовищами або передавати іншим розробникам. Незважаючи на свою простоту, SQLite є повноцінною транзакційною СУБД, що підтримує властивості ACID (Atomicity, Consistency, Isolation, Durability). Це забезпечує цілісність та надійність даних при виконанні операцій, що є важливим для будь-якого інтернет-магазину, навіть на етапі розробки. SQLite використовує стандартний діалект SQL, що робить його сумісним з багатьма інструментами та бібліотеками, розробленими для роботи з реляційними базами даних.

Для проекту інтернет-магазину військового спорядження, на поточному етапі розробки, можливостей SQLite виявилося цілком достатньо. Вона використовується для зберігання таблиць з інформацією про товари (products), замовлення (orders) та позиції замовлень (order_items). ORM Sequelize успішно взаємодіє з SQLite, абстрагуючи деталі роботи з файловою базою даних.

Водночас, важливо розуміти й обмеження SQLite. Вона не призначена для сценаріїв з високим рівнем паралелізму записів, оскільки використовує блокування на рівні файлу, що може стати вузьким місцем при великій кількості одночасних операцій модифікації даних. Також SQLite має менше вбудованих інструментів для адміністрування, моніторингу та реплікації порівняно з потужними серверними СУБД. Тому, хоча SQLite є чудовим вибором для розробки та невеликих застосунків, для розгортання інтернет-магазину у продуктивному середовищі з очікуваним значним навантаженням та потребою у високій доступності, рекомендувалося б розглянути перехід на більш потужну клієнт-серверну SQL СУБД, таку як PostgreSQL або MySQL. Проте, використання ORM Sequelize значно спростить такий перехід у майбутньому.

### 2.3.2 Sequelize та RESTful API

Для взаємодії серверної частини веб-застосунку на Node.js з обраною SQL базою даних (SQLite на етапі розробки) було використано бібліотеку Sequelize, яка є потужною та популярною ORM (Object-Relational Mapper). Одночасно, для забезпечення комунікації між клієнтською частиною на Angular та серверною логікою було спроектовано та реалізовано RESTful API за допомогою фреймворку Express.js.

Sequelize ORM відіграє ключову роль у спрощенні роботи з базою даних. ORM – це технологія програмування, яка дозволяє перетворювати дані між несумісними системами типів, зокрема між об'єктами в коді застосунку (наприклад, JavaScript/TypeScript об'єктами) та таблицями в реляційній базі даних. Замість написання "сирих" SQL-запитів, розробник може визначати моделі даних як класи або об'єкти у своїй мові програмування та використовувати методи цих моделей для виконання операцій CRUD (Create, Read, Update, Delete) та інших маніпуляцій з даними. Sequelize, будучи проміс-орієнтованою ORM для Node.js, підтримує різні діалекти SQL, включаючи

PostgreSQL, MySQL, MariaDB, SQLite та Microsoft SQL Server, що забезпечує певну гнучкість у виборі СУБД.

У проекті інтернет-магазину військового спорядження Sequelize використовується для кількох ключових завдань. По-перше, це визначення моделей даних (Product, Order, OrderItem). Кожна модель описує структуру відповідної таблиці в базі даних, включаючи назви полів, їх типи даних (наприклад, DataTypes.STRING, DataTypes.INTEGER, DataTypes.FLOAT, DataTypes.TEXT, DataTypes.DATE), а також обмеження, такі як первинні ключі, зовнішні ключі, дозволені чи ні значення NULL, значення за замовчуванням тощо. По-друге, Sequelize дозволяє легко визначати зв'язки (асоціації) між моделями, такі як "один-до-багатьох" (hasMany, belongsTo) або "багато-до-багатьох" (belongsToMany). У нашому проекті визначено зв'язки між замовленнями та їх позиціями, а також між позиціями замовлень та товарами, що дозволяє Sequelize автоматично управляти зовнішніми ключами та спрощує отримання пов'язаних даних. По-третє, Sequelize забезпечує механізм синхронізації бази даних (sequelize.sync()). Цей метод аналізує визначені моделі та автоматично створює або оновлює відповідні таблиці в базі даних, що є дуже зручним на етапі розробки. По-четверте, Sequelize надає багатий набір методів для вибірки та маніпуляції даними (наприклад, findAll, findByPk, findOne, create, bulkCreate, update, destroy), які дозволяють виконувати всі необхідні операції з базою даних, використовуючи об'єктно-орієнтований синтаксис. Наприклад, для отримання всіх товарів використовується Product.findAll(), а для створення нового замовлення – Order.create(). Нарешті, Sequelize підтримує транзакції (sequelize.transaction(), commit(), rollback()), що було використано при реалізації логіки створення замовлення для забезпечення атомарності операцій запису в декілька пов'язаних таблиць (orders та order_items).

RESTful API (Representational State Transfer application programming interface) слугує мостом між клієнтською частиною на Angular та серверною частиною на Node.js/Express.js. REST – це архітектурний стиль для

проектування мережевих застосунків, який базується на використанні стандартних HTTP-методів для взаємодії з ресурсами. У проекті інтернет-магазину ресурсами є товари (products) та замовлення (orders). Для управління цими ресурсами було створено відповідні API ендпоінти:

- GET /api/products – для отримання списку всіх товарів.
- POST /api/products – для створення нового товару (використовується адмін-панеллю).
- PUT /api/products/:id – для оновлення існуючого товару за його ідентифікатором.
- DELETE /api/products/:id – для видалення товару за його ідентифікатором.
- POST /api/orders – для створення нового замовлення (використовується при оформленні покупки клієнтом).
- GET /api/orders – для отримання списку всіх замовлень (використовується адмін-панеллю).
- DELETE /api/orders/:id – для видалення замовлення за його ідентифікатором (використовується адмін-панеллю).

Фреймворк Express.js використовується для визначення цих маршрутів та обробки відповідних HTTP-запитів. Обробники маршрутів отримують запит від клієнта, використовують методи Sequelize для взаємодії з базою даних SQLite, формують відповідь та надсилають її назад клієнту. Дані між клієнтом та сервером передаються у форматі JSON (JavaScript Object Notation), для чого на сервері використовується вбудований в Express.js middleware express.json() для парсингу тіла запитів. Також на сервері реалізовано базовий механізм CORS (Cross-Origin Resource Sharing) для дозволу запитів від Angular-застосунку, що працює на іншому порту (localhost:4200). Таким чином, поєднання Sequelize для роботи з базою даних та RESTful API, реалізованого на Express.js, забезпечує структурований, стандартизований та ефективний спосіб взаємодії між різними частинами розроблюваного веб-застосунку.

# ВИСНОВОК ДО РОЗДІЛУ

У другому розділі даної дипломної роботи було проведено детальний огляд та обґрунтування вибору технологічного стеку, необхідного для розробки веб-застосунку управління магазином військового спорядження. Вибір технологій базувався на аналізі сучасних підходів до веб-розробки, вимог до функціональності майбутнього інтернет-магазину, а також на необхідності створення продуктивного, масштабованого та легко підтримуваного програмного рішення.

Для реалізації клієнтської частини (frontend) було обрано фреймворк Angular у поєднанні з мовою програмування TypeScript. Таке рішення дозволяє створити динамічний односторінковий застосунок (SPA) з багатим користувацьким інтерфейсом, використовуючи компонентний підхід, потужну систему маршрутизації та сервісів для управління станом і бізнес-логікою. Мова TypeScript забезпечує переваги статичної типізації, підвищуючи надійність коду та спрощуючи процес розробки. Основою для структурування та візуального оформлення інтерфейсу слугують стандарти HTML5 та CSS3.

Для серверної частини (backend), що відповідає за обробку API-запитів та взаємодію з базою даних, було розглянуто та обрано платформу Node.js з використанням мови програмування JavaScript та веб-фреймворку Express.js. Цей вибір зумовлений високою продуктивністю Node.js для операцій вводу-виводу, великою екосистемою npm та можливістю використання JavaScript у всьому стеку розробки, що спрощує процес.

Управління даними проекту реалізовано за допомогою реляційної системи управління базами даних SQLite, яка є легкою та простою у налаштуванні для етапу розробки. Взаємодія з базою даних з боку Node.js-

застосунку здійснюється через ORM (Object-Relational Mapper) Sequelize, що абстрагує роботу з SQL-запитами та дозволяє оперувати даними як об'єктами. Доступ до серверної логіки та даних з клієнтської частини забезпечується через розроблений RESTful API.

Процес розробки підтримується сучасними інструментами, такими як інтегроване середовище розробки Visual Studio Code, система контролю версій Git, менеджер пакунків npm та інструмент командного рядка Angular CLI, що сукупно сприяють підвищенню ефективності та якості розробки.

Таким чином, обраний комплекс технологій для клієнтської та серверної частин, а також для управління базами даних, формує збалансовану та сучасну програмну платформу, що дозволяє ефективно реалізувати поставлені завдання щодо створення функціонального веб-застосунку для управління магазином військового спорядження, описаного в наступних розділах.

# РОЗДІЛ 3
# РЕАЛІЗАЦІЯ ПРОЄКТУ

## 3.1. Огляд застосованої архітектури проєкту

Для розробки веб-застосунку управління магазином військового спорядження було обрано класичну та перевірену часом клієнт-серверну архітектуру. Такий підхід передбачає чітке розділення функціональних обов'язків між двома основними компонентами системи: клієнтською частиною (фронтендом), яка виконується у веб-браузері користувача, та серверною частиною (бекендом), що працює на віддаленому сервері та відповідає за обробку даних і бізнес-логіку.

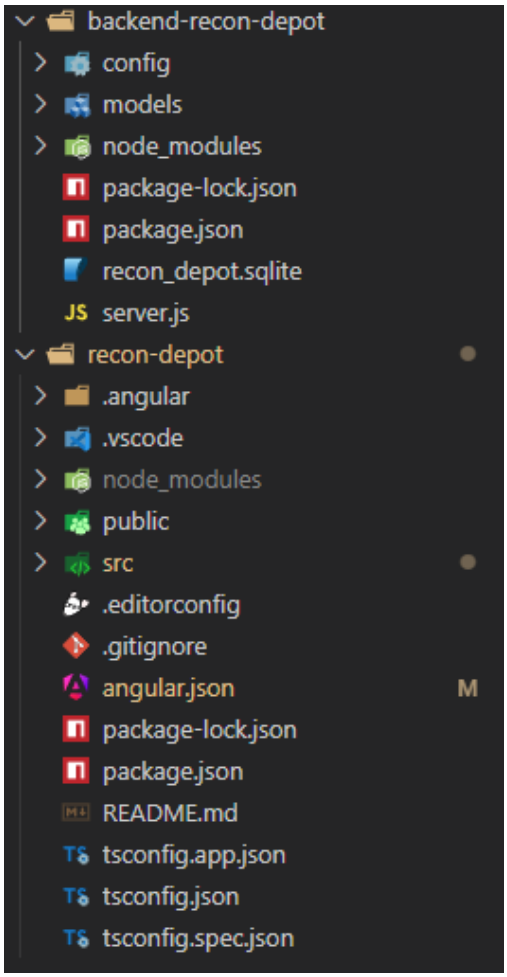



Рисунок 3.1 – Структура проєкту у Visual Studio Code

Клієнтська частина реалізована як односторінковий застосунок (Single Page Application - SPA) з використанням фреймворку Angular. Концепція SPA означає, що більшість ресурсів (HTML, CSS, JavaScript код) завантажуються один раз при першому відкритті сайту, а подальша навігація між різними розділами та оновлення контенту відбуваються динамічно, без повного перезавантаження веб-сторінки. Це забезпечує високу швидкість відгуку інтерфейсу та плавний користувацький досвід, що є особливо важливим для інтернет-магазинів, де користувачі часто переглядають багато сторінок товарів та категорій. Angular, як комплексна платформа, надає всі необхідні інструменти для побудови складних SPA, включаючи компонентну архітектуру, систему маршрутизації, сервіси для управління станом та взаємодії з сервером.

Серверна частина розроблена на платформі Node.js з використанням веб-фреймворку Express.js. Вона відповідає за обробку запитів від клієнтської частини, взаємодію з базою даних, реалізацію бізнес-логіки (наприклад, створення, оновлення, видалення товарів та замовлень) та надання даних клієнту через RESTful API. RESTful API визначає набір стандартизованих правил для взаємодії між клієнтом та сервером за допомогою HTTP-протоколу. Такий підхід дозволяє чітко розділити відповідальність та забезпечити слабку зв'язаність між фронтендом та бекендом, що спрощує їх незалежну розробку та тестування.

Для зберігання даних застосунку, таких як інформація про товари, категорії та замовлення, на серверній стороні використовується реляційна база даних SQLite. Взаємодія з базою даних з Node.js/Express.js застосунку здійснюється через ORM (Object-Relational Mapper) Sequelize, яка абстрагує роботу з SQL-запитами та дозволяє оперувати даними як об'єктами.

Взаємодія між клієнтською частиною на Angular та серверною частиною на Node.js відбувається через HTTP-запити до визначених API ендпоінтів. Клієнт відправляє запити на отримання або модифікацію даних, а сервер

обробляє ці запити, взаємодіє з базою даних і повертає відповідь, зазвичай у форматі JSON (JavaScript Object Notation), який є легким та зручним для обміну даними у веб-середовищі. Таким чином, обрана клієнт-серверна архітектура з чітким розділенням на SPA-фронтенд та RESTful API-бекенд є сучасним та ефективним підходом для реалізації функціонального та масштабованого інтернет-магазину.

### 3.2. Реалізація серверної частини

Серверна частина веб-застосунку є його ядром, відповідальним за обробку даних, реалізацію бізнес-логіки та забезпечення взаємодії з клієнтською частиною через програмний інтерфейс (API). Для інтернет-магазину військового спорядження було розроблено серверну частину з використанням платформи Node.js та веб-фреймворку Express.js. Такий вибір дозволив створити легковаговий, швидкий та ефективний бекенд, здатний обробляти запити від клієнтського Angular-застосунку. Основними завданнями реалізованої серверної частини є управління каталогом товарів, включаючи їх додавання, редагування та видалення, а також обробка та збереження інформації про замовлення клієнтів. Взаємодія з базою даних SQLite здійснюється за допомогою ORM Sequelize.

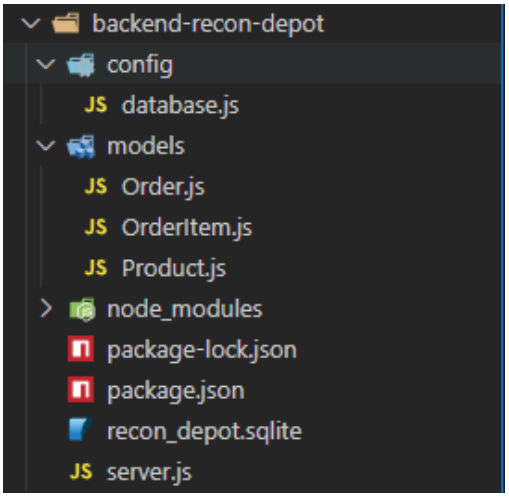



Рисунок 3.2 – Структура серверної частини

### 3.2.1. Модель Product

Модель Product, визначена у файлі backend-recon-depot/models/Product.js, є ключовою для функціонування інтернет-магазину, оскільки вона представляє товари, що пропонуються для продажу. Ця модель описує структуру таблиці products у базі даних SQLite та надає інтерфейс для взаємодії з цією таблицею через методи Sequelize.

Визначення моделі Product включає набір атрибутів, що відповідають стовпцям у таблиці бази даних. Поле id визначено як первинний ключ (primaryKey: true) таблиці. Воно має тип DataTypes.INTEGER та налаштоване на автоматичне збільшення значення (autoIncrement: true) при додаванні нових товарів, що гарантує унікальність кожного запису. Також це поле не може бути порожнім (allowNull: false).

Атрибут name типу DataTypes.STRING призначений для зберігання назви товару і є обов'язковим для заповнення (allowNull: false), оскільки назва є одним з основних ідентифікаторів товару для користувача. Аналогічно, поле category типу DataTypes.STRING також є обов'язковим (allowNull: false) і використовується для групування товарів за відповідними категоріями, що полегшує навігацію та фільтрацію в каталозі.

Поле price визначено як DataTypes.FLOAT та є обов'язковим (allowNull: false). Воно зберігає ціну товару. Хоча для фінансових розрахунків іноді рекомендують використовувати тип DECIMAL для уникнення потенційних проблем з точністю чисел з плаваючою комою, для даного проекту в рамках SQLite тип FLOAT є прийнятним для представлення ціни.

Для детального опису товару призначено поле description типу DataTypes.TEXT. Цей тип даних дозволяє зберігати довгі текстові рядки. Опис товару може бути необов'язковим, тому для цього поля встановлено allowNull: true.

Атрибут imageName типу DataTypes.STRING зберігає назву файлу зображення товару (наприклад, backpack-assault-40.jpg). Це поле також може бути необов'язковим (allowNull: true), якщо для якогось товару зображення відсутнє. Передбачається, що самі файли зображень зберігаються в окремій директорії на сервері (або в клієнтській частині в assets/images, як у поточній реалізації фронтенду), а в базі даних зберігається лише посилання на назву файлу. Повний шлях до зображення (imageUrl) для відображення на клієнті формується динамічно.

При ініціалізації моделі Product за допомогою методу Product.init() також вказані додаткові опції. Параметр sequelize передає екземпляр підключення до бази даних. modelName: 'Product' визначає назву моделі в Sequelize, а tableName: 'products' явно вказує, що відповідна таблиця в базі даних буде називатися products (у множині, що є поширеною практикою). Опція timestamps: true автоматично додає до таблиці два стовпці: createdAt (час створення запису) та updatedAt (час останнього оновлення запису), значення яких Sequelize буде оновлювати автоматично.

Ця модель Product використовується в обробниках API-запитів у файлі server.js для виконання CRUD-операцій (Create, Read, Update, Delete) з товарами. Наприклад, Product.findAll() для отримання списку всіх товарів, Product.findByPk() для пошуку товару за ідентифікатором, Product.create() для додавання нового товару, product.update() для оновлення існуючого та product.destroy() для видалення. Модель Product також пов'язана з моделлю OrderItem відношенням "один-до-багатьох", що означає, що один товар може бути присутнім у багатьох позиціях різних замовлень. Цей зв'язок визначається при ініціалізації сервера.

### 3.2.2. Модель Order

Наступною важливою сутністю в системі інтернет-магазину є замовлення. Модель Order, визначена у файлі backend-recon-depot/models/Order.js, представляє інформацію про кожне окреме замовлення, зроблене покупцем. Вона описує структуру таблиці orders у базі даних SQLite і слугує для збереження та управління даними, пов'язаними з процесом покупки. Як і для інших моделей Sequelize, Order успадковує функціональність від базового класу Model. Первинним ключем таблиці є поле id типу DataTypes.INTEGER, яке автоматично інкрементується (autoIncrement: true) та не може бути порожнім (allowNull: false), забезпечуючи унікальну ідентифікацію кожного замовлення.

Для зберігання інформації про клієнта, який зробив замовлення, модель Order включає декілька текстових полів. Атрибут customerFullName типу DataTypes.STRING призначений для повного імені покупця. Поле customerPhone типу DataTypes.STRING зберігає контактний номер телефону клієнта, а customerEmail того ж типу DataTypes.STRING – його електронну адресу. Всі ці три поля є обов'язковими для заповнення (allowNull: false), оскільки вони необхідні для зв'язку з клієнтом та підтвердження замовлення. Детальна адреса доставки зберігається в полі deliveryAddress типу DataTypes.TEXT, яке також є обов'язковим. Для додаткових побажань або інструкцій від клієнта передбачено поле customerComment типу DataTypes.TEXT, яке може бути порожнім (allowNull: true).

Фінансова інформація про замовлення представлена полем totalAmount типу DataTypes.FLOAT, що зберігає загальну суму замовлення і є обов'язковим (allowNull: false). Дата та час створення замовлення фіксуються у полі orderDate типу DataTypes.DATE. Для нього встановлено значення за замовчуванням DataTypes.NOW, що означає, що при створенні нового запису про замовлення це поле автоматично заповнюватиметься поточним часом сервера, якщо

значення не передано явно. Це поле також не може бути порожнім. Поточний стан обробки замовлення зберігається в атрибуті status типу DataTypes.STRING, який є обов'язковим (allowNull: false) і має значення за замовчуванням 'new', що відповідає новому, ще не обробленому замовленню. У майбутньому це поле може приймати інші значення, такі як 'processing', 'shipped', 'completed', 'cancelled', для відстеження життєвого циклу замовлення.

При ініціалізації моделі Order також вказано опцію timestamps: true, завдяки чому Sequelize автоматично додає та управляє полями createdAt та updatedAt, які фіксують час створення та останнього оновлення запису в базі даних. Назва моделі в Sequelize визначена як 'Order', а відповідна таблиця в базі даних називається 'orders'. Модель Order має зв'язок типу "один-до-багатьох" з моделлю OrderItem, оскільки одне замовлення може містити декілька товарних позицій. Цей зв'язок, визначений як Order.hasMany(OrderItem, { as: 'items', foreignKey: 'orderId', onDelete: 'CASCADE' }), дозволяє легко отримувати всі товари, що належать до конкретного замовлення, а також забезпечує каскадне видалення позицій замовлення при видаленні самого замовлення.

### 3.2.3. Модель OrderItem

Для деталізації вмісту кожного замовлення використовується модель OrderItem, визначена у файлі backend-recon-depot/models/OrderItem.js. Ця модель представляє окрему товарну позицію в конкретному замовленні, включаючи інформацію про товар, його кількість та ціну на момент покупки. Вона описує структуру таблиці order_items у базі даних.

Як і інші моделі, OrderItem наслідує клас Model з Sequelize. Поле id типу DataTypes.INTEGER є первинним ключем (primaryKey: true) з автоматичним інкрементом (autoIncrement: true) і не може бути порожнім (allowNull: false). Ключовими атрибутами моделі OrderItem є ті, що описують сам товар у контексті замовлення. Поле productName типу DataTypes.STRING зберігає

назву товару такою, якою вона була на момент оформлення замовлення. Це важливо, оскільки назва товару в основному каталозі може з часом змінитися, але в історії замовлення має залишитися оригінальна назва. Це поле є обов'язковим (allowNull: false). Кількість замовлених одиниць товару зберігається в полі quantity типу DataTypes.INTEGER, яке також є обов'язковим. Ціна за одну одиницю товару на момент покупки фіксується в полі priceAtOrder типу DataTypes.FLOAT і є обов'язковою. Збереження ціни на момент замовлення є критично важливим, оскільки ціни в магазині можуть змінюватися, але для конкретного замовлення має бути зафіксована та ціна, за якою товар було придбано.

Для зв'язку з оригінальним товаром у каталозі (таблиця products) модель OrderItem містить поле productId типу DataTypes.INTEGER. Це поле є зовнішнім ключем. У поточній реалізації воно визначене як allowNull: true, що означає, що теоретично позиція замовлення може існувати без прямого зв'язку з товаром у каталозі (наприклад, якщо товар пізніше було видалено з каталогу, а історію замовлень потрібно зберегти). Зв'язок OrderItem.belongsTo(Product, { foreignKey: 'productId' }) встановлює цей зв'язок, а опція onDelete: 'SET NULL' для відповідного Product.hasMany(OrderItem) означає, що при видаленні товару з каталогу значення productId в пов'язаних позиціях замовлень стане NULL.
Також модель OrderItem містить зовнішній ключ orderId (автоматично створюється Sequelize при визначенні зв'язку OrderItem.belongsTo(Order, { foreignKey: 'orderId' })), який пов'язує кожну позицію з конкретним замовленням у таблиці orders. Цей зв'язок є критичним для формування повного складу замовлення.

При ініціалізації моделі OrderItem вказано modelName: 'OrderItem' та tableName: 'order_items'. Для цієї моделі опція timestamps встановлена в false, оскільки час створення та оновлення позиції замовлення зазвичай визначається часом створення/оновлення самого замовлення і окремі мітки часу для кожної позиції не є необхідними. Модель OrderItem використовується на серверній

стороні при створенні нового замовлення: для кожної товарної позиції з кошика покупця створюється відповідний запис в таблиці order_items, пов'язаний із щойно створеним записом у таблиці orders.

### 3.3. Реалізація клієнтської частини

Клієнтська частина, або фронтенд, є обличчям веб-застосунку, забезпечуючи безпосередню взаємодію користувача з усіма функціональними можливостями інтернет-магазину військового спорядження. Для розробки цього компонента було обрано сучасний та потужний фреймворк Angular у поєднанні з мовою програмування TypeScript. Такий вибір дозволив реалізувати архітектуру односторінкового застосунку (SPA), що забезпечує високу швидкість відгуку, плавність переходів між розділами та багатий інтерактивний досвід як для покупців, так і для адміністраторів. У цьому підрозділі детально розглянуто структуру проекту клієнтської частини.

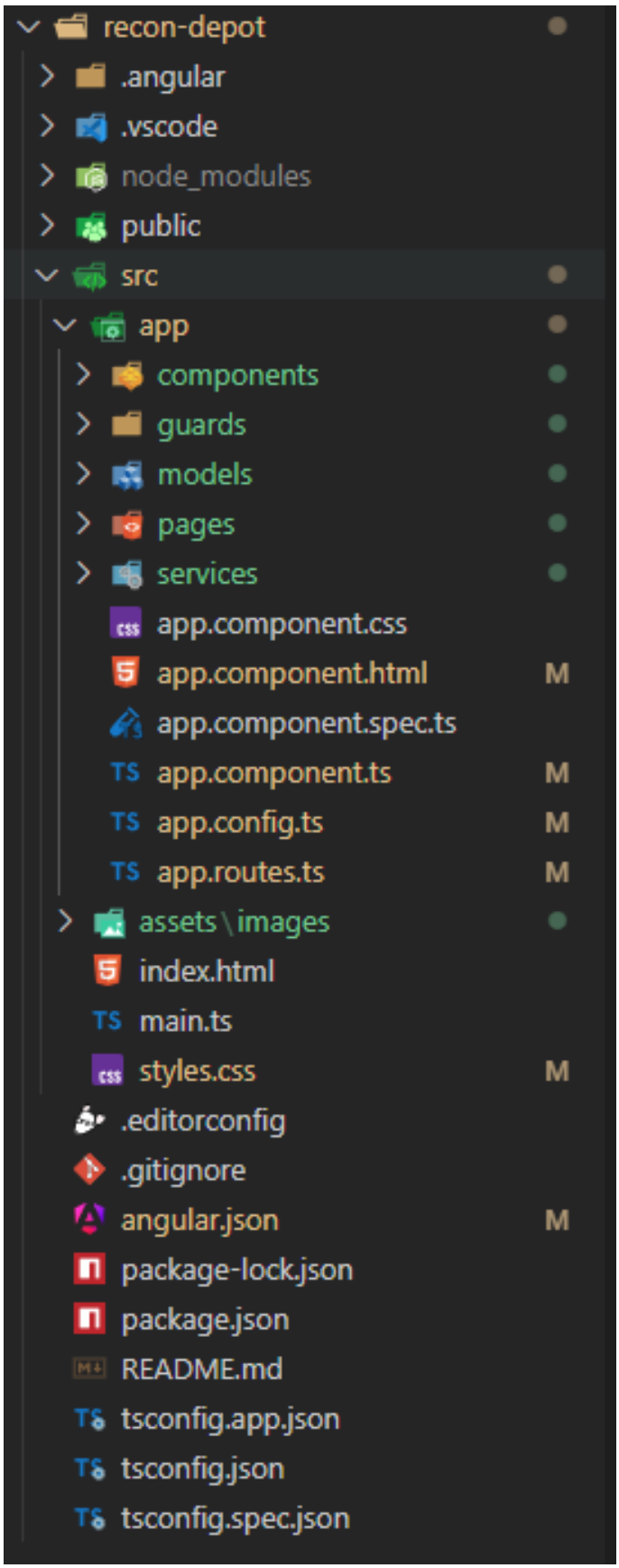



Рисунок 3.3 – Структура клієнтської частини

### 3.3.1. Опис директорії components

Директорія src/app/components/ призначена для зберігання перевикористовуваних UI-компонентів. Це такі елементи інтерфейсу, які не є повноцінними сторінками, але використовуються як будівельні блоки на різних сторінках або всередині інших, більш складних компонентів. Винесення таких елементів в окремі компоненти дозволяє уникнути дублювання коду, спростити їх тестування та модифікацію, а також підвищити загальну модульність застосунку. У розробленому інтернет-магазині ця директорія містить два ключові компоненти, що відіграють важливу роль у формуванні користувацького інтерфейсу каталогу товарів.

Першим з них є компонент category-menu (src/app/components/category-menu/). Його головне завдання – відображення списку доступних категорій товарів, що дозволяє користувачам легко фільтрувати каталог. Цей компонент отримує дані про категорії з ProductService і динамічно формує меню. Він має вихідну подію (categorySelected), за допомогою якої повідомляє батьківський компонент (наприклад, HomePageComponent) про вибір користувачем певної категорії або опції "Всі категорії". Також він може приймати вхідні дані, наприклад, для підсвічування поточної активної категорії. У шаблоні цього компонента також розміщено кнопку для переходу до панелі адміністратора.

Другим важливим компонентом у цій директорії є product-list (src/app/components/product-list/). Цей компонент відповідає безпосередньо за відображення списку товарів у вигляді сітки карток. Він отримує масив товарів (вже відфільтрованих за категорією, якщо фільтр активний) та дані про поточне сортування. Кожна картка товару в його шаблоні містить зображення, назву, категорію, коротку частину опису, ціну та кнопку "Додати у кошик". Реалізовано також функціонал показу повного опису товару при наведенні курсора на картку, що покращує користувацький досвід, оскільки окремі сторінки для кожного товару на даному етапі не передбачені. При натисканні

на кнопку "Додати у кошик", компонент взаємодіє з CartService для додавання відповідного товару до кошика. Також у цьому компоненті реалізована логіка сортування товарів за різними критеріями (назва, ціна) на стороні клієнта. Таким чином, компоненти з директорії components забезпечують ключовий функціонал відображення та взаємодії з каталогом товарів.

### 3.3.2. Опис директорії guards

Директорія src/app/guards/ використовується для зберігання захисників маршрутів (route guards) в Angular-застосунку. Захисники маршрутів – це спеціальні сервіси, які реалізують один або декілька інтерфейсів (наприклад, CanActivate, CanLoad, CanDeactivate) і дозволяють контролювати доступ до певних маршрутів або можливість переходу з них. Це важливий інструмент для забезпечення безпеки та реалізації умовної навігації в застосунку.

У проекті інтернет-магазину військового спорядження було створено один ключовий захисник маршруту – auth.guard.ts, розташований у цій директорії. Його основне призначення – захист адміністративної панелі (/admin/dashboard) від несанкціонованого доступу. AuthGuard реалізує інтерфейс CanActivate, який вимагає наявності методу canActivate(). Цей метод викликається Angular-маршрутизатором щоразу, коли користувач намагається перейти на маршрут, захищений цим guard-ом.

Всередині методу canActivate(), AuthGuard взаємодіє з сервісом автентифікації AuthService. Він викликає метод AuthService.isAdmin(), який перевіряє, чи поточний користувач є залогіненим адміністратором (нагадаємо, що стан логіну адміністратора зберігається у sessionStorage і управляється через AuthService). Якщо AuthService.isAdmin() повертає true, це означає, що користувач має права доступу, і AuthGuard також повертає true, дозволяючи активацію маршруту та перехід на сторінку адміністративної панелі. У протилежному випадку, якщо користувач не є залогіненим адміністратором,

AuthGuard повертає false, забороняючи доступ до захищеного маршруту. Крім того, для покращення користувацького досвіду, у випадку відмови в доступі, AuthGuard автоматично перенаправляє користувача на сторінку входу для адміністратора (/admin-login) за допомогою сервісу Router.

Таким чином, AuthGuard відіграє критично важливу роль у забезпеченні безпеки адміністративної частини веб-застосунку, гарантуючи, що тільки авторизовані особи можуть отримати доступ до функцій управління магазином. Використання функціонального guard-а, як це було зроблено в проекті, є сучасним підходом в Angular, що дозволяє писати більш компактний та легкий для тестування код захисників.

### 3.3.3. Опис директорії models

Директорія src/app/models/ у структурі Angular-проекту призначена для визначення моделей даних, тобто TypeScript інтерфейсів або класів, які описують структуру об'єктів, з якими працює застосунок. Використання чітко визначених моделей є важливою практикою, оскільки це забезпечує типізацію, покращує читабельність коду, полегшує розуміння очікуваної структури даних при взаємодії між різними частинами системи (наприклад, між сервісами та компонентами) та дозволяє використовувати переваги статичної перевірки типів TypeScript для виявлення помилок на ранніх етапах розробки.

У проекті інтернет-магазину військового спорядження ключовою моделлю даних, винесеною в цю директорію, є модель товару, визначена у файлі product.model.ts. Цей файл експортує інтерфейс Product, який описує набір полів для кожного товару в системі. До цих полів належать id (унікальний ідентифікатор товару, типу number), name (назва товару, типу string), category (категорія, до якої належить товар, типу string), price (ціна товару, типу number), description (детальний опис товару, типу string), imageName (назва файлу

зображення товару, типу string) та опціональне поле imageUrl (повний шлях до зображення, типу string, яке зазвичай формується динамічно на основі imageName). Цей інтерфейс Product широко використовується у всьому клієнтському застосунку: сервіс ProductService оперує масивами об'єктів цього типу при отриманні даних з бекенду та наданні їх компонентам; компоненти, такі як ProductListComponent та AdminDashboardComponent, використовують цей інтерфейс для коректного відображення інформації про товари та для зв'язування даних у формах редагування чи додавання. Також цей тип використовується в CartService для представлення товару всередині позиції кошика.

Окрім централізованої моделі Product, у проекті також використовуються інші інтерфейси, що описують специфічні структури даних, хоча вони можуть бути визначені безпосередньо у файлах відповідних сервісів або компонентів для локального використання. Наприклад, в CartService визначено інтерфейс CartItem, який об'єднує об'єкт Product та властивість quantity (кількість даного товару в кошику). У компоненті CheckoutComponent визначено інтерфейс OrderDetails, що структурує дані, які користувач вводить у форму оформлення замовлення (ПІБ, телефон, email, адреса доставки, коментар). В OrderService, у свою чергу, визначено інтерфейси Order та OrderApiResponse, які описують повну структуру замовлення, включаючи деталі клієнта, список замовлених позицій, загальну суму, дату та статус замовлення, а також очікувану структуру відповіді від серверного API при створенні замовлення. Для форми додавання/редагування товарів в AdminDashboardComponent також використовується локальний інтерфейс ProductFormModel для управління даними форми. Використання таких чітко визначених структур даних значно підвищує надійність та підтримуваність коду клієнтської частини застосунку.

### 3.3.4. Опис директорії pages

Директорія src/app/pages/ в Angular-проекті призначена для розміщення компонентів, які представляють собою окремі логічні сторінки або повноцінні розділи веб-застосунку. Ці компоненти зазвичай прив'язані до конкретних маршрутів у системі навігації застосунку і часто слугують контейнерами, що організовують та відображають декілька менших, перевикористовуваних UI-компонентів, а також взаємодіють із сервісами для отримання даних та реалізації бізнес-логіки, специфічної для даної сторінки. У структурі інтернет-магазину військового спорядження ця директорія містить кілька ключових компонентів-сторінок.

Компонент home (src/app/pages/home/) відповідає за відображення головної сторінки інтернет-магазину. Це перша сторінка, яку зазвичай бачить користувач. Вона виконує роль вітрини, представляючи каталог товарів. У своїй структурі HomeComponent використовує два перевикористовувані компоненти: CategoryMenuComponent для відображення меню категорій товарів та ProductListComponent для відображення сітки товарів. HomeComponent також координує взаємодію між цими дочірніми компонентами, наприклад, передаючи обрану категорію з меню до списку товарів для фільтрації.

Компонент shopping-cart (src/app/pages/shopping-cart/) реалізує функціонал сторінки кошика. Тут користувач може переглянути всі товари, які він додав до кошика, змінити кількість кожної позиції, видалити окремі товари або очистити весь кошик. Компонент відображає загальну вартість замовлення та надає кнопку для переходу до сторінки оформлення замовлення. Для отримання даних про вміст кошика та виконання операцій з ним ShoppingCartComponent активно взаємодіє з сервісом CartService.

Сторінка оформлення замовлення реалізована у компоненті checkout (src/app/pages/checkout/). На цій сторінці користувач вводить свої контактні дані, адресу доставки та іншу необхідну інформацію для завершення покупки.

Компонент відображає підсумок товарів з кошика та їх загальну вартість. Після заповнення та відправки форми, CheckoutComponent взаємодіє з OrderService для передачі даних замовлення на серверну частину (або для імітації цього процесу, як у поточній реалізації).

Для адміністративної частини сайту в директорії pages створено два компоненти. Компонент admin-login (src/app/pages/admin-login/) надає інтерфейс для входу адміністратора в систему. Він містить форму для введення логіна та пароля і взаємодіє з AuthService для перевірки облікових даних та авторизації. У випадку успішного входу користувач перенаправляється до основної адміністративної панелі.

Компонент admin-dashboard (src/app/pages/admin-dashboard/) є головною сторінкою панелі адміністратора. Тут реалізовано основний функціонал для управління магазином: відображення списку всіх товарів, форма для додавання нових товарів та редагування існуючих, можливість видалення товарів. Також цей компонент відображає список отриманих замовлень з можливістю їх перегляду та видалення. AdminDashboardComponent активно взаємодіє з ProductService для управління товарами та з OrderService для управління замовленнями, а також з AuthService для реалізації функції виходу з системи. Кожен з цих компонентів-сторінок є самостійним, standalone-компонентом Angular, що спрощує їхню інтеграцію та управління залежностями.

### 3.3.5. Опис директорії services

Директорія src/app/services/ відіграє центральну роль в архітектурі клієнтської частини Angular-застосунку. Вона призначена для розміщення сервісів – спеціальних класів, які інкапсулюють бізнес-логіку, взаємодію з зовнішніми API (в нашому випадку, з бекендом), управління спільними даними та надання перевикористовуваного функціоналу для різних компонентів застосунку. В Angular сервіси зазвичай є одинаками (singletons), що

створюються та надаються через механізм впровадження залежностей (Dependency Injection). Це дозволяє компонентам бути "тоншими", зосереджуючись на відображенні та взаємодії з користувачем, тоді як складна логіка виноситься в сервіси. У проекті інтернет-магазину військового спорядження ця директорія містить декілька ключових сервісів.

Сервіс ProductService (src/app/services/product.service.ts) відповідає за всю логіку, пов'язану з товарами. Його основними завданнями є отримання списку товарів з серверної частини через HTTP GET-запит до ендпоінту /api/products, а також реалізація CRUD-операцій для товарів: додавання нових товарів (через POST-запит), оновлення існуючих (через PUT-запит) та їх видалення (через DELETE-запит). ProductService використовує BehaviorSubject з бібліотеки RxJS (products$ та categories$) для зберігання поточного списку товарів та похідного списку категорій, що дозволяє компонентам (наприклад, ProductListComponent, CategoryMenuComponent, AdminDashboardComponent) підписуватися на ці потоки даних та автоматично оновлювати свій стан при їх зміні. Сервіс також містить логіку для обробки помилок HTTP-запитів та формування повних URL-адрес для зображень товарів.

Сервіс CartService (src/app/services/cart.service.ts) реалізує всю функціональність клієнтського кошика для покупок. Він зберігає список товарів, доданих користувачем до кошика (у вигляді масиву об'єктів CartItem), та надає методи для маніпулювання цим списком: додавання товару, видалення товару, зміна кількості одиниць товару. CartService також розраховує загальну кількість товарів у кошику та їх загальну вартість. Подібно до ProductService, він використовує BehaviorSubject (cartItemsSource та cartItemCount$) для надання компонентам (наприклад, ShoppingCartComponent та лічильнику кошика в AppComponent) можливості реактивно відстежувати зміни у вмісті кошика. На поточному етапі дані кошика зберігаються в пам'яті сервісу і не персистентні між сесіями браузера (хоча раніше розглядалася можливість використання localStorage).

Сервіс OrderService (src/app/services/order.service.ts) відповідає за операції, пов'язані із замовленнями. Його ключовий метод placeOrder приймає деталі замовлення від клієнта (з CheckoutComponent) та список товарів з кошика, формує відповідний запит та відправляє його на серверний ендпоінт /api/orders для створення нового замовлення в базі даних. Крім того, для потреб адміністративної панелі, OrderService містить методи loadAdminOrders (для отримання списку всіх замовлень з бекенду через GET /api/orders) та deleteOrder (для видалення замовлення через DELETE /api/orders/:id). Для динамічного оновлення списку замовлень в адмін-панелі використовується BehaviorSubject (adminOrders$).

Нарешті, сервіс AuthService (src/app/services/auth.service.ts) реалізує логіку автентифікації та авторизації адміністратора магазину. Він містить методи для входу в систему (login), перевіряючи надані облікові дані (на поточному етапі – порівнюючи з жорстко закодованими значеннями, але в перспективі це може бути запит до бекенду), та для виходу з системи (logout). Стан автентифікації адміністратора зберігається в sessionStorage браузера для забезпечення персистентності сесії в межах одного вікна/вкладки та оновлення сторінки. AuthService також надає Observable (isLoggedIn$) для відстеження стану входу та метод isAdmin() для використання захисниками маршрутів та компонентами для перевірки прав доступу.

Таким чином, сервіси в директорії services є важливим архітектурним шаром, що забезпечує чітке розділення відповідальності, перевикористання коду та ефективне управління даними та бізнес-логікою в клієнтській частині інтернет-магазину.

### 3.4. Реалізація роботи з базою даних

Робота з базою даних є центральним аспектом серверної частини розробленого веб-застосунку, оскільки саме база даних відповідає за надійне зберігання та структурування всієї інформації про товари, замовлення та їх складові. Для проекту було обрано систему управління базами даних SQLite, а взаємодія з нею з боку Node.js/Express.js застосунку реалізована за допомогою ORM (Object-Relational Mapper) Sequelize. На початковому етапі ініціалізації сервера відбувається підключення до бази даних, визначеної у конфігураційному файлі config/database.js. Далі, за допомогою методу sequelize.sync(), виконується синхронізація визначених моделей Sequelize (Product, Order, OrderItem) зі схемою бази даних. Цей процес автоматично створює відповідні таблиці (products, orders, order_items) у файлі бази даних recon_depot.sqlite, якщо вони ще не існують, а також встановлює необхідні зв'язки між ними на основі визначених асоціацій (наприклад, Order.hasMany(OrderItem)).

Після успішної синхронізації, серверна частина готова до виконання операцій з даними. Реалізовані API ендпоінти використовують методи, надані Sequelize, для виконання CRUD-операцій (Create, Read, Update, Delete). Наприклад, для отримання списку товарів використовується Product.findAll(), для створення нового товару – Product.create(), для оновлення – product.update(), а для видалення – product.destroy(). Аналогічно, при оформленні нового замовлення через ендпоінт POST /api/orders, створюються записи в таблицях orders та order_items за допомогою методів Order.create() та OrderItem.bulkCreate(). Важливим аспектом реалізації є використання транзакцій Sequelize при створенні замовлення, що гарантує атомарність операцій запису в декілька пов'язаних таблиць: або всі дані зберігаються успішно, або, у випадку помилки, всі зміни відкочуються, забезпечуючи цілісність даних. Логування SQL-запитів, активоване в конфігурації Sequelize,

допомагало на етапі розробки відстежувати та налагоджувати взаємодію з базою даних.

### 3.5. Реалізація роботи зі сховищем

Окрім серверної бази даних, у клієнтській частині веб-застосунку також використовуються механізми зберігання даних безпосередньо у веб-браузері користувача. Ці механізми застосовуються для підвищення зручності користування та збереження певного стану застосунку між сесіями або перезавантаженнями сторінки. У розробленому інтернет-магазині для таких цілей було задіяно sessionStorage. Зокрема, сервіс AuthService, який відповідає за автентифікацію адміністратора, використовує sessionStorage для збереження інформації про успішний вхід адміністратора в систему. При успішній автентифікації встановлюється спеціальний прапорець у sessionStorage, а при виході з системи або закритті сесії браузера (вкладки) цей прапорець видаляється. Це дозволяє адміністратору залишатися залогіненим при оновленні сторінки адмін-панелі, уникаючи необхідності повторного введення облікових даних протягом поточної сесії роботи з браузером, що покращує загальний досвід користування адміністративною частиною.

Варто зазначити, що на ранніх етапах розробки localStorage також використовувався в ProductService для тимчасового зберігання даних про товари, доки не було реалізовано повноцінну взаємодію з серверною базою даних. Однак, з впровадженням бекенду, відповідальність за персистентне зберігання та управління каталогом товарів повністю перейшла на серверну сторону та базу даних SQLite.

# ВИСНОВОК ДО РОЗДІЛУ

У третьому розділі дипломної роботи було детально описано процес практичної реалізації веб-застосунку для управління магазином військового спорядження. Було представлено огляд застосованої клієнт-серверної архітектури, де клієнтська частина реалізована як односторінковий застосунок (SPA) на базі фреймворку Angular, а серверна частина – на платформі Node.js з використанням фреймворку Express.js, що надає RESTful API для взаємодії.

Було детально розглянуто ключові аспекти реалізації серверної частини, включаючи налаштування середовища, конфігурацію підключення до бази даних SQLite за допомогою ORM Sequelize, а також розробку моделей даних Product, Order та OrderItem та визначення зв'язків між ними. Описано процес створення API ендпоінтів для виконання CRUD-операцій з товарами та для обробки й збереження замовлень. Також було висвітлено структуру клієнтської частини, включаючи опис основних директорій (компоненти, сторінки, сервіси, моделі, захисники маршрутів) та призначення ключових файлів, що формують логіку та інтерфейс користувача. Окрему увагу приділено реалізації роботи з базою даних на сервері та використанню клієнтських сховищ, зокрема sessionStorage для управління сесією адміністратора.

Реалізований програмний продукт демонструє практичне застосування обраних технологій для створення функціонального веб-застосунку, що відповідає основним поставленим вимогам. Розроблена система має чітко визначену структуру, забезпечує взаємодію між клієнтською та серверною частинами, а також реалізує ключовий функціонал інтернет-магазину, включаючи каталог товарів, кошик, оформлення замовлення (з імітацією відправки та збереженням на бекенді) та адміністративну панель для управління товарами та перегляду замовлень.

# РОЗДІЛ 4
# ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ ВЕБ-ПЛАТФОРМИ

## 4.1. Опис реалізації та тестування основних користувацьких сценаріїв та перевірка сценаріїв взаємодії кінцевих користувачів

### 4.1.1. Сценарій входу та роботи адміністратора

Почнемо огляд типових сценаріїв взаємодії клієнтів зі входу адміністратора.

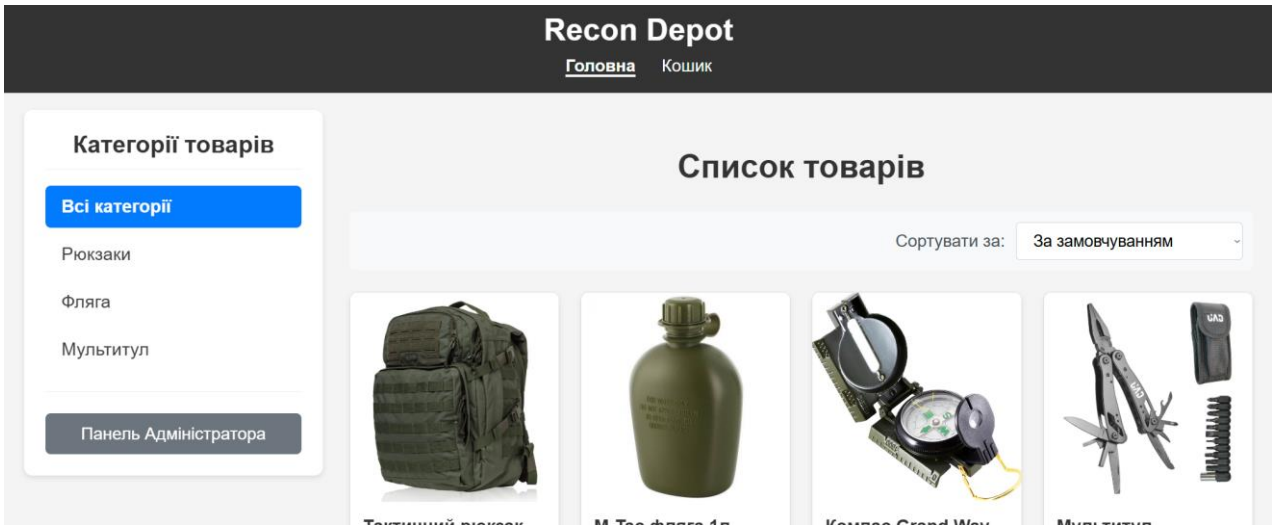



Рисунок 4.1 – Головна сторінка сайту

Попадаючи на веб-сторінку, вказану на рисунку 4.1, адміністратор обирає кнопку «Панель Адміністратора».

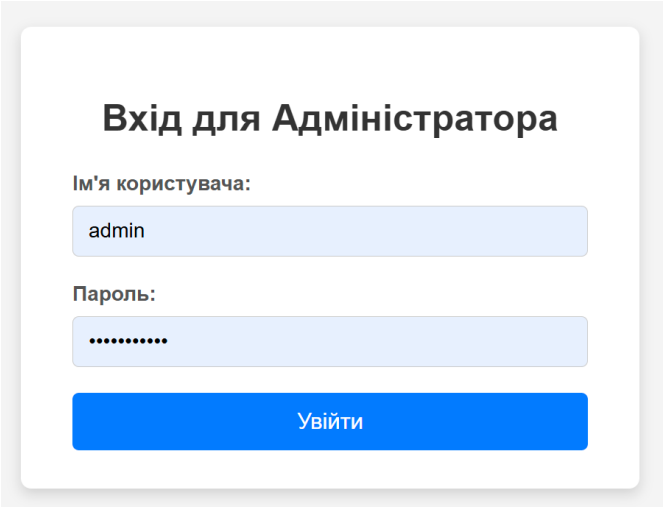



Рисунок 4.2 – Сторінка входу Адміністратора

Ввівши неправильні ім’я користувача та пароль на сторінці входу, зображеній на рисунку 4.2, співробітник не зможе пройти авторизацію та отримає повідомлення про помилкові дані, зображене на рисунку 4.3.

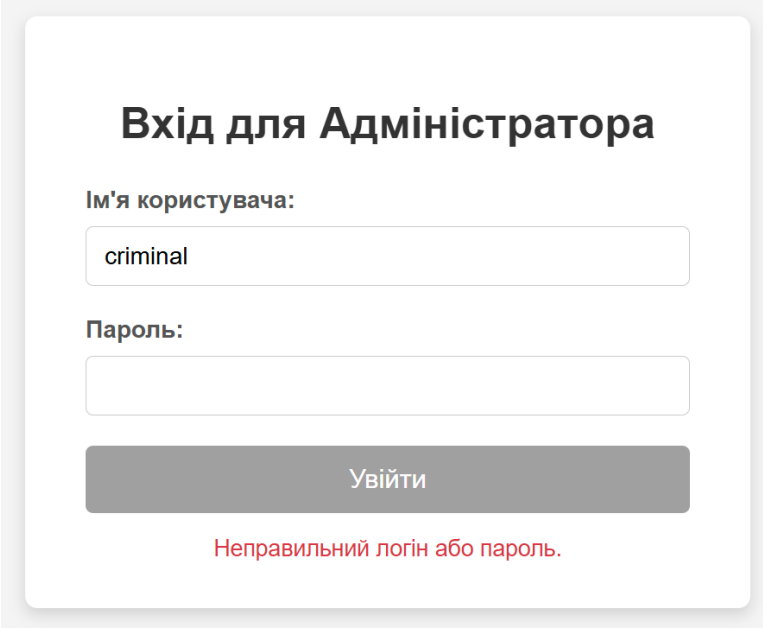



Рисунок 4.3 – Адміністратор ввів неправильні логін або пароль

Ввівши правильні дані авторизації, співробітник проходить у панель Адміністратора, зображену на рисунку 4.4 і яка містить форму для додавання нового товару, список існуючих товарів та отриманих замовлень. Звідси адміністратор може додавати до асортименту нові товари, редагувати існуючі та опрацьовувати отримані замовлення (рисунки 4.5-4.9).

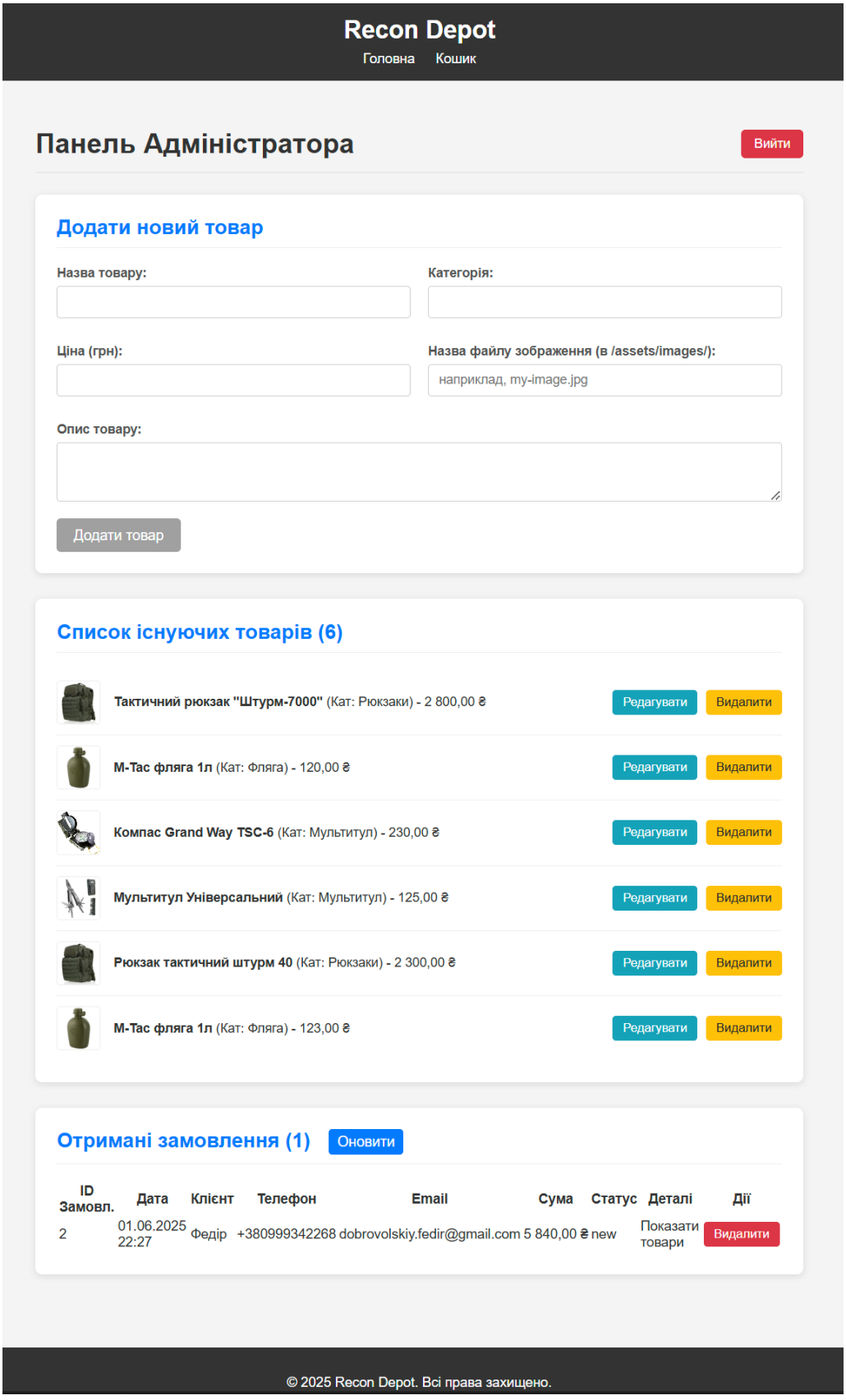



Рисунок 4.4 – Панель Адміністратора

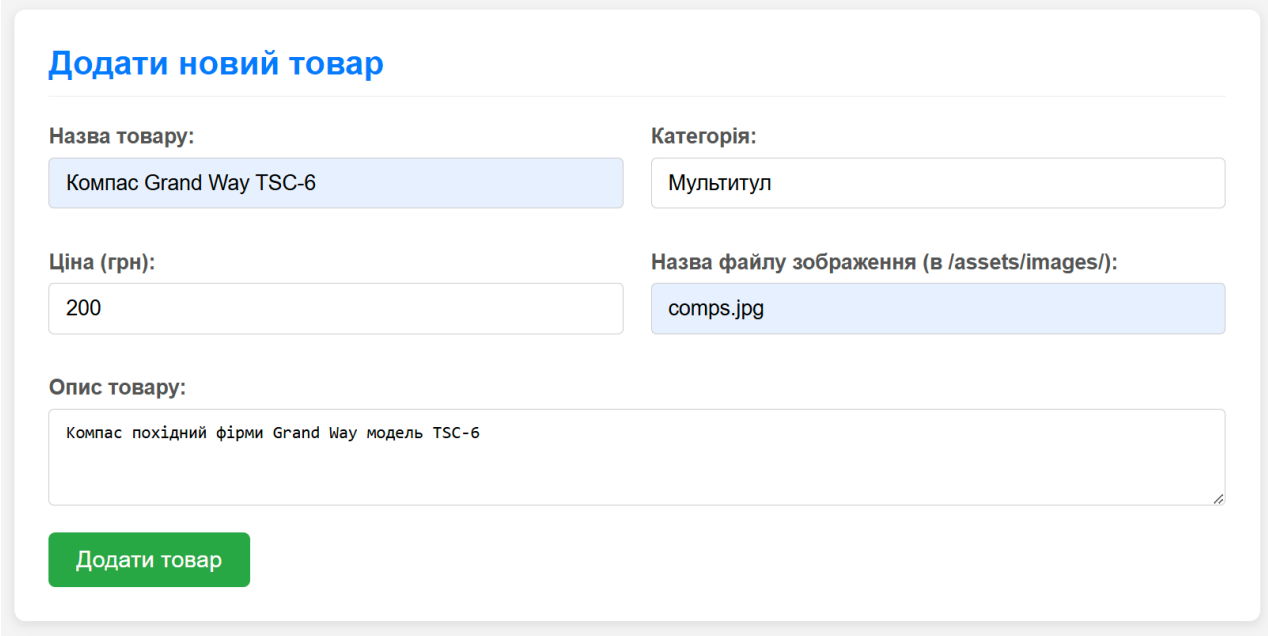



Рисунок 4.5 – Форма додавання нового товару

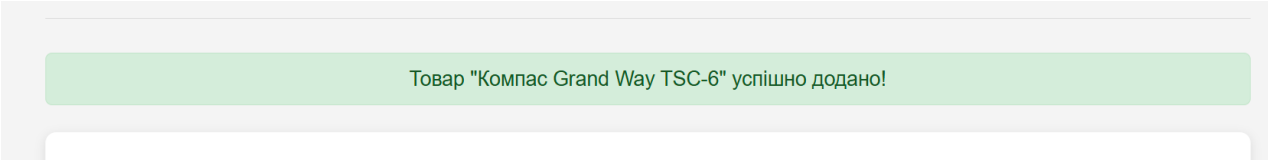



Рисунок 4.6 – Повідомлення про успішне додавання нового товару

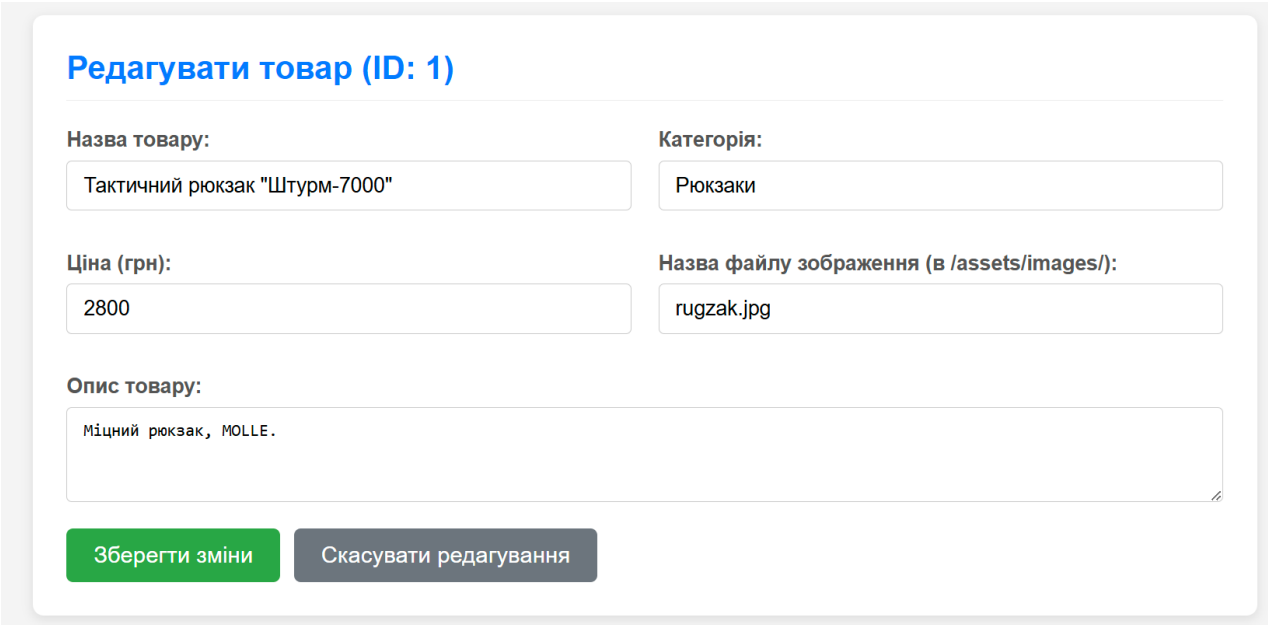



Рисунок 4.7 – Форма редагування товару

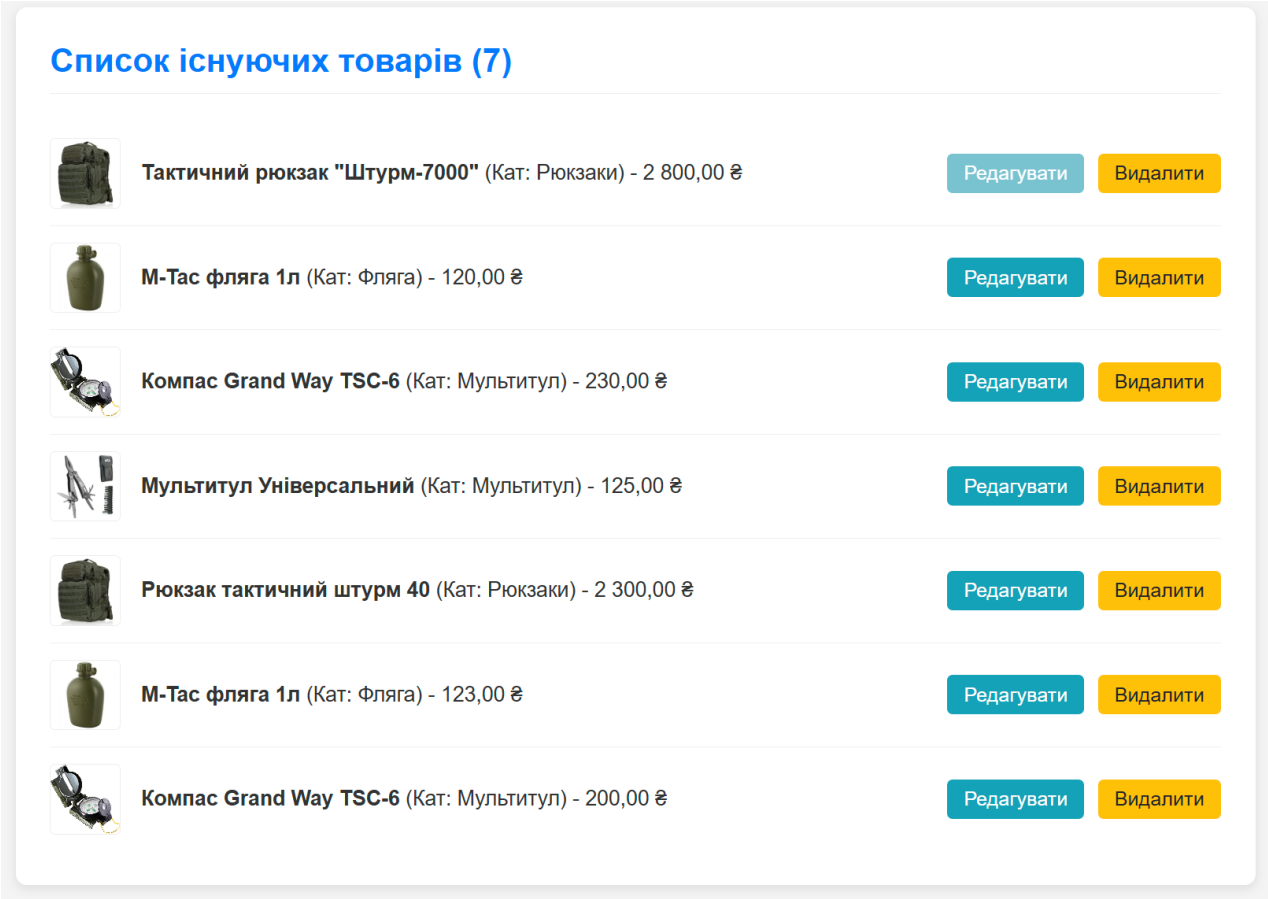



Рисунок 4.8 – Список існуючих товарів

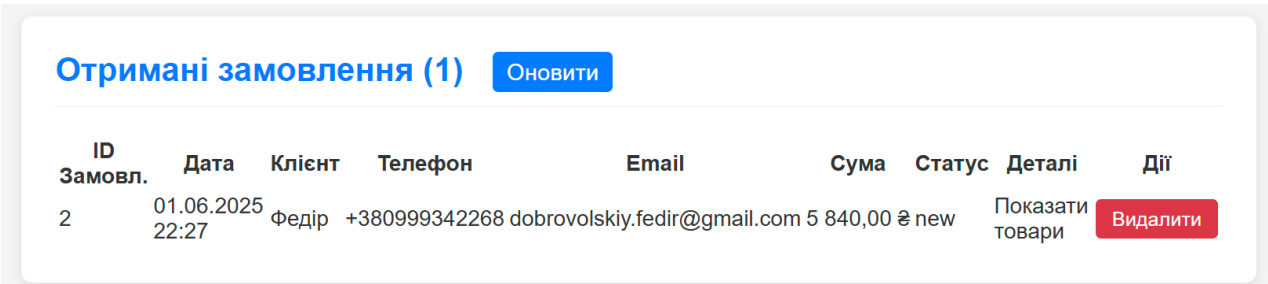



Рисунок 4.9 – Список отриманих замовлень

### 4.1.2. Сценарій ознайомлення з асортиментом користувачем

Розглянемо сценарій використання клієнтом веб-застосунку для ознайомлення з асортиментом інтернет-магазину. При відвідуванні сайту, клієнт потрапляє на головну сторінку (рисунок 4.10), на якій він бачить асортимент товарів, Список

категорій товарів, header сайту, опцію посортувати відображення товару (рисунок 4.11) та посилання на кошик.

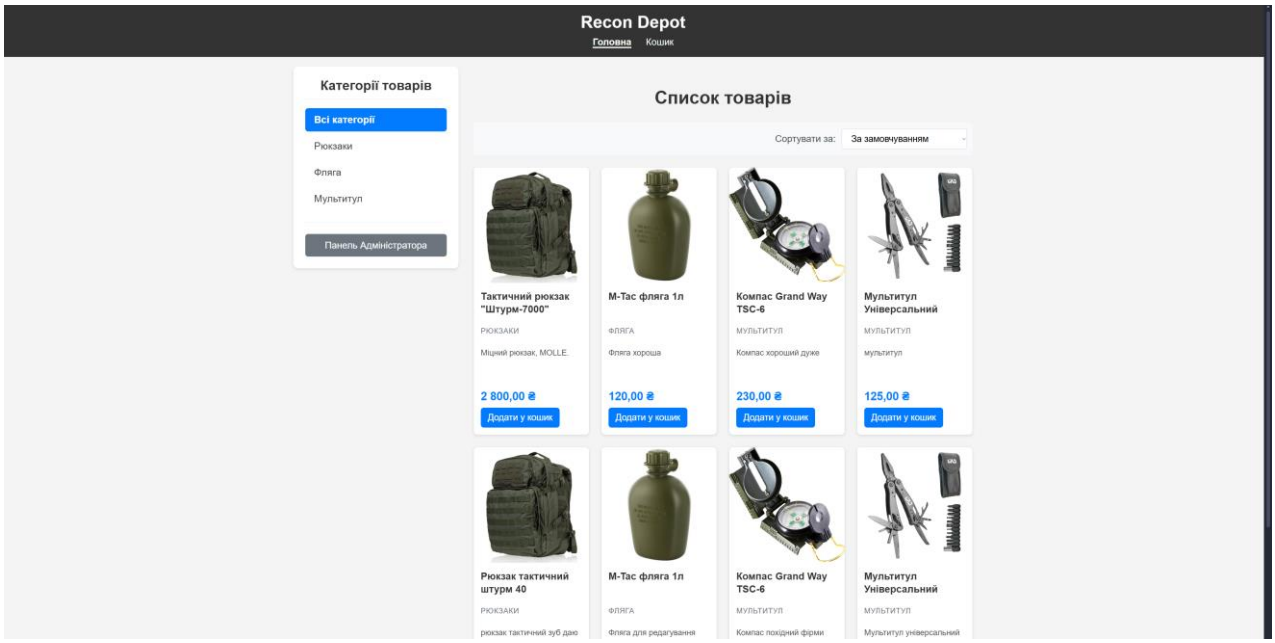



Рисунок 4.10 – Головна сторінка сайту

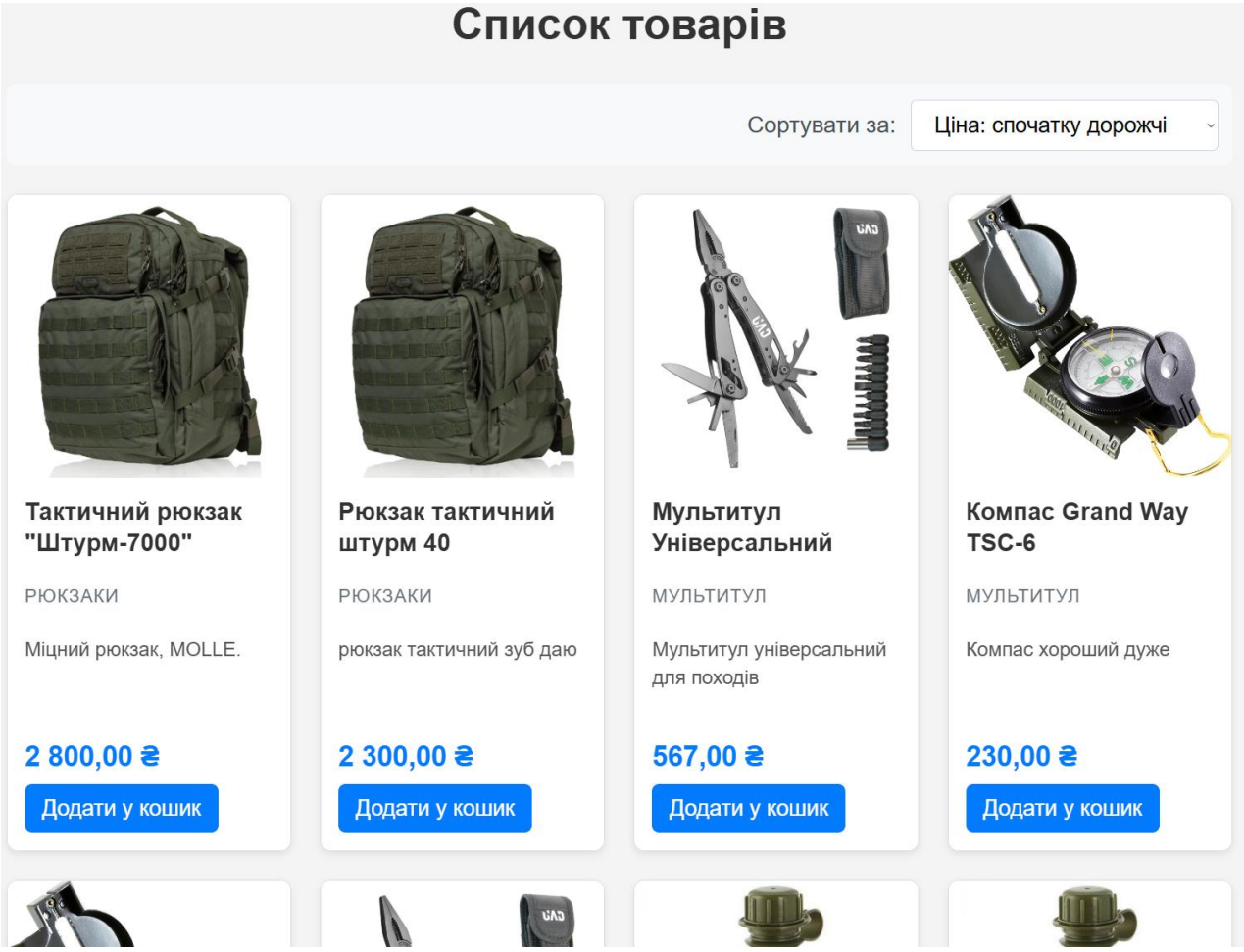



Рисунок 4.11 – Відсортовані товари однією з опцій сортування.

### 4.1.3 Сценарій оформлення замовлення користувачем

Розглянемо сценарій коли клієнт, ознайомившись з асортиментом, приступає до оформлення замовлень. Натиснувши кнопку «Додати у кошик», відповідний товар додається до списку підготовки до оформлення замовлення (рисунок 4.12), що знаходиться за посиланням «Кошик».

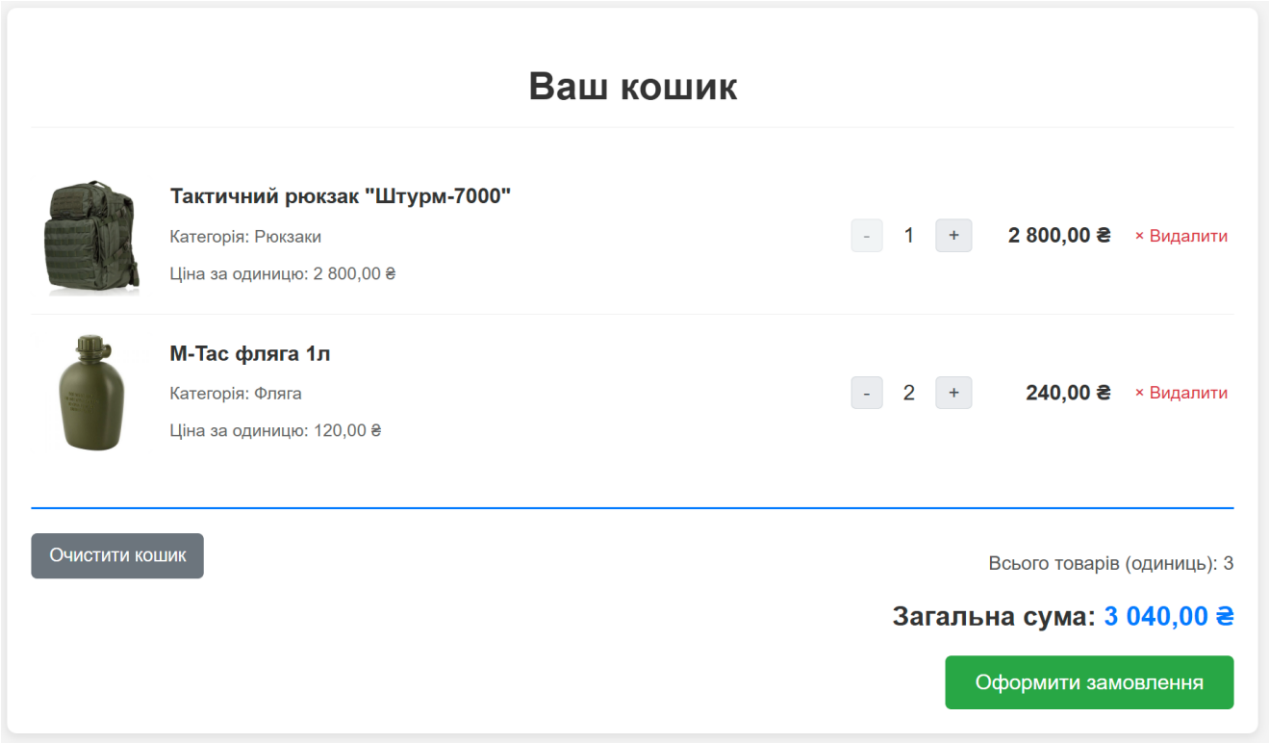



Рисунок 4.12 - «Кошик»

При натисканні кнопки «Оформити замовлення», клієнт переходить до форми оформлення замовлення (рисунок 4.13), де після вводу персональних даних і натиснувши «Підтвердити замовлення», відповідний запит відправляється на сервер та додається у список замолень на панелі адміністратора, який в свою чергу може перейти до опрацювання замовлення.

### 4.2. Оцінка ефективності та якості розробленої платформи

Оцінка ефективності розробленого веб-застосунку для управління магазином військового спорядження показала, що обраний технологічний стек та архітектурні рішення забезпечують належний рівень продуктивності для поставлених завдань. Клієнтська частина, реалізована на Angular, демонструє швидкий відгук інтерфейсу та плавну навігацію завдяки архітектурі

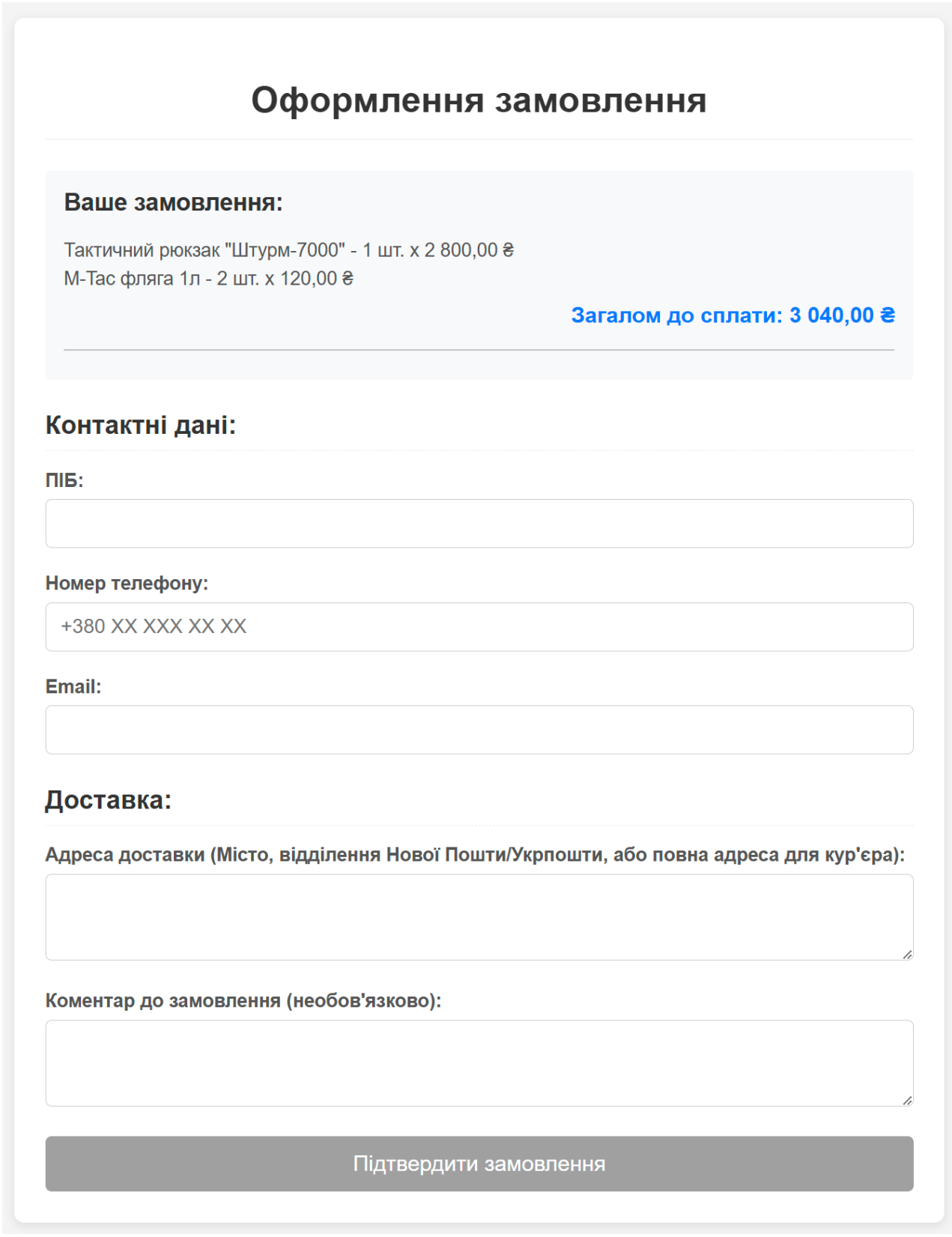



Рисунок 4.13 – Форма оформлення замовлення

односторінкового застосунку та компонентному підходу. Серверна частина на Node.js з Express.js ефективно обробляє API-запити для CRUD-операцій з

товарами та замовленнями, а використання SQLite через Sequelize забезпечує достатню швидкість доступу до даних для обсягу дипломного проекту. Якість програмного коду підтримується за рахунок використання TypeScript, чіткого розділення логіки на сервіси та компоненти, що сприяє його читабельності та подальшій підтримуваності.

### 4.3. Аналіз відповідності функціональним та нефункціональним вимогам

Проведений аналіз підтверджує, що розроблений веб-застосунок відповідає основним функціональним вимогам, визначеним на етапі проектування. Реалізовано каталог товарів з можливістю фільтрації за категоріями та сортування, функціонал кошика для покупок, процес оформлення замовлення зі збереженням даних на сервері, а також адміністративну панель із захищеним доступом для управління товарами (додавання, редагування, видалення) та перегляду й видалення замовлень. Також задоволено ключові нефункціональні вимоги: забезпечено інтуїтивно зрозумілий користувацький інтерфейс, належну швидкість відгуку клієнтської частини, базовий рівень безпеки для адміністративної панелі та збереження цілісності даних у базі.

# ВИСНОВОК ДО РОЗДІЛУ

У четвертому розділі було представлено підходи до тестування розробленого веб-застосунку та проведено оцінку його ефективності, якості та відповідності поставленим вимогам. Результати тестування ключових функціональних модулів, таких як каталог товарів, кошик, оформлення замовлення та адміністративна панель, підтвердили їх працездатність та коректність виконання операцій. Застосунок демонструє задовільну швидкість роботи, стабільність та відповідність основним функціональним і нефункціональним вимогам, що робить його життєздатним прототипом для інтернет-магазину військового спорядження та успішним результатом виконання завдань дипломного проектування.

# ВИСНОВКИ

У ході виконання дипломної роботи було успішно завершено повний цикл розробки веб-застосунку для управління магазином військового спорядження "Recon Depot". Початковий етап включав глибокий аналіз предметної області електронної комерції у цій специфічній ніші, дослідження потреб цільової аудиторії (покупців та адміністраторів) та огляд існуючих аналогічних платформ. Це дозволило чітко сформулювати вимоги до системи та підтвердити актуальність розробки.

На основі проведеного аналізу було обґрунтовано вибір технологічного стеку: для клієнтської частини використано фреймворк Angular з TypeScript, що забезпечило створення динамічного односторінкового застосунку (SPA), а для серверної частини – платформу Node.js з Express.js та JavaScript для реалізації RESTful API. Зберігання даних про товари та замовлення здійснюється у базі даних SQLite за допомогою ORM Sequelize. Було детально описано архітектуру проекту та реалізовано ключові модулі, включаючи каталог товарів з фільтрацією та сортуванням, кошик, процес оформлення замовлення, а також адміністративну панель для управління товарами та замовленнями.

Завершальний етап включав тестування розробленого функціоналу. Перевірка основних користувацьких сценаріїв підтвердила коректність роботи ключових модулів та відповідність застосунку поставленим функціональним і нефункціональним вимогам. Таким чином, було створено дієздатний прототип інтернет-магазину військового спорядження, що демонструє практичне застосування обраних технологій та готовий до можливого подальшого розвитку.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Офіційна документація Angular [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://angular.io/docs
2. Офіційна документація TypeScript [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://www.typescriptlang.org/docs/
3. Node.js Документація та Посібники [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://nodejs.org/uk/docs/guides/
4. SQLBolt: Інтерактивні уроки SQL [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://sqlbolt.com/
5. Nielsen Norman Group: Статті про юзабіліті та UX [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://www.nngroup.com/articles/
6. MDN Web Docs: Посібники з веб-розробки [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://developer.mozilla.org/uk/docs/Web
7. Офіційна документація Express.js [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://expressjs.com/
8. Офіційна документація Sequelize ORM [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://sequelize.org/
9. Pro Git book (Скотт Шакон та Бен Страуб) / Документація Git [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://git-scm.com/book/uk/v2 або https://git-scm.com/doc
10. OWASP Top Ten Project [Електронний ресурс] – 2025. – Режим доступу до ресурсу: https://owasp.org/www-project-top-ten/

# ДОДАТОК А

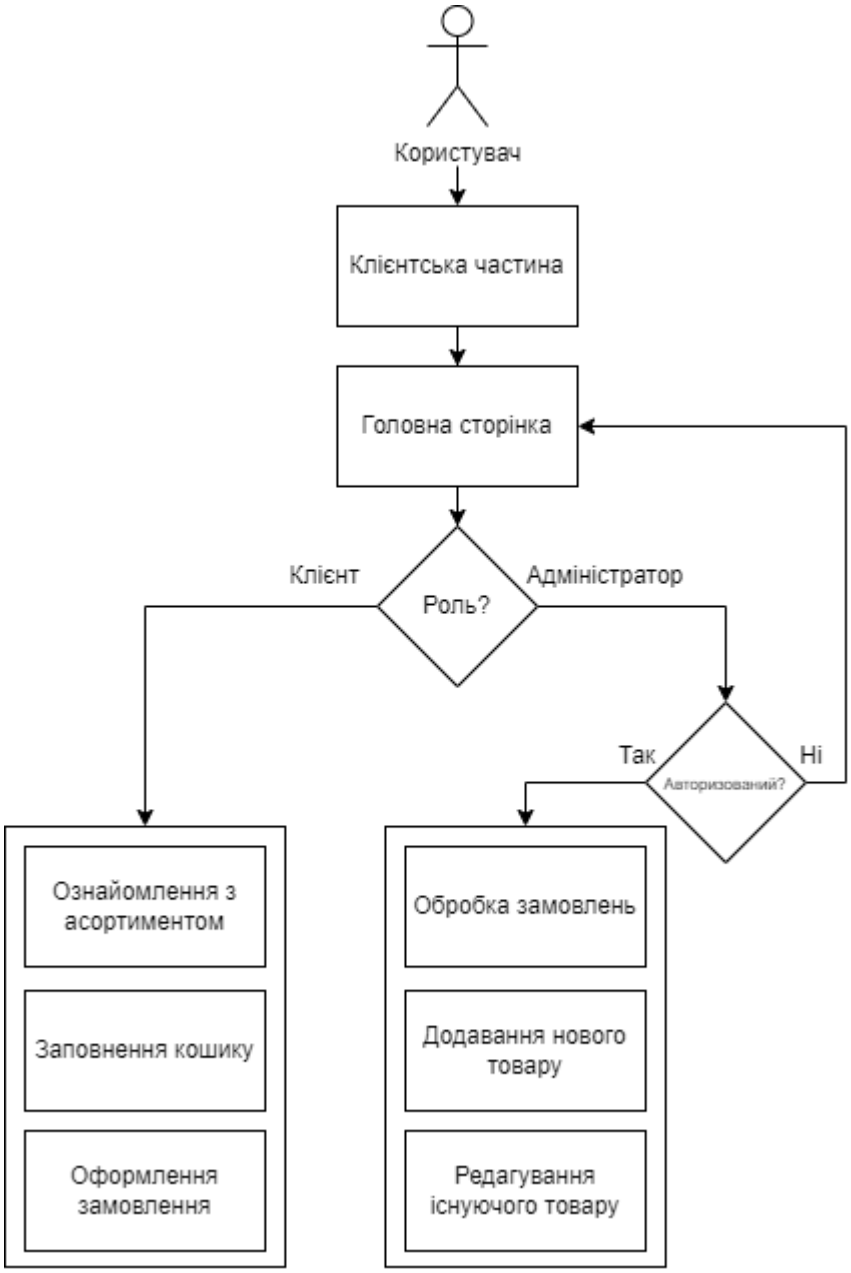
Веб-застосунок управління магазином військового спорядження

**Структура системи**

ІАЛЦ.467200.004 Д1

Аркушів 1

**Київ 2025 р**




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.004 Д1 | | |
| | | № докум. | Підпис | Дата | | | |
| Розробив | | Добровольський Ф.В. | | | **Веб-застосунок управління магазином військового спорядження** | Літ. | Аркуш 1 / Аркушів 1 |
| Перевірив | | Меленчуков М.Є. | | | | | |
| | | | | | Структурна системи | НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-12 | |
| Н. Контр. | | Міщенко Л.Д. | | | | | |
| Затвердив | | | | | | | |

# ДОДАТОК Б

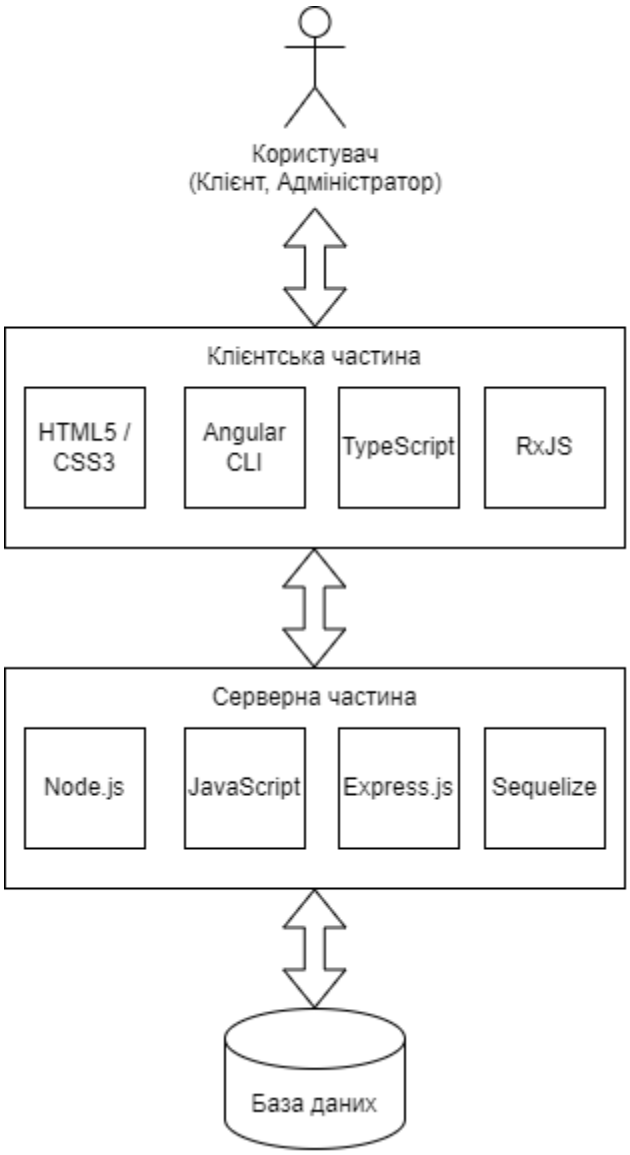
Веб-застосунок управління магазином військового спорядження

**Функціональна структура**

ІАЛЦ.467200.005 Д2

Аркушів 1

**Київ 2025 р**




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.005 Д2 | | |
| | | № докум. | Підпис | Дата | | | |
| Розробив | | Добровольський Ф.В. | | | **Веб-застосунок управління магазином військового спорядження** Функціональна структура | Літ. | Аркуш 1 / Аркушів 1 |
| Перевірив | | Меленчуков М.Є. | | | | | |
| | | | | | | НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-12 | |
| Н. Контр. | | Міщенко Л.Д. | | | | | |
| Затвердив | | | | | | | |

# ДОДАТОК В

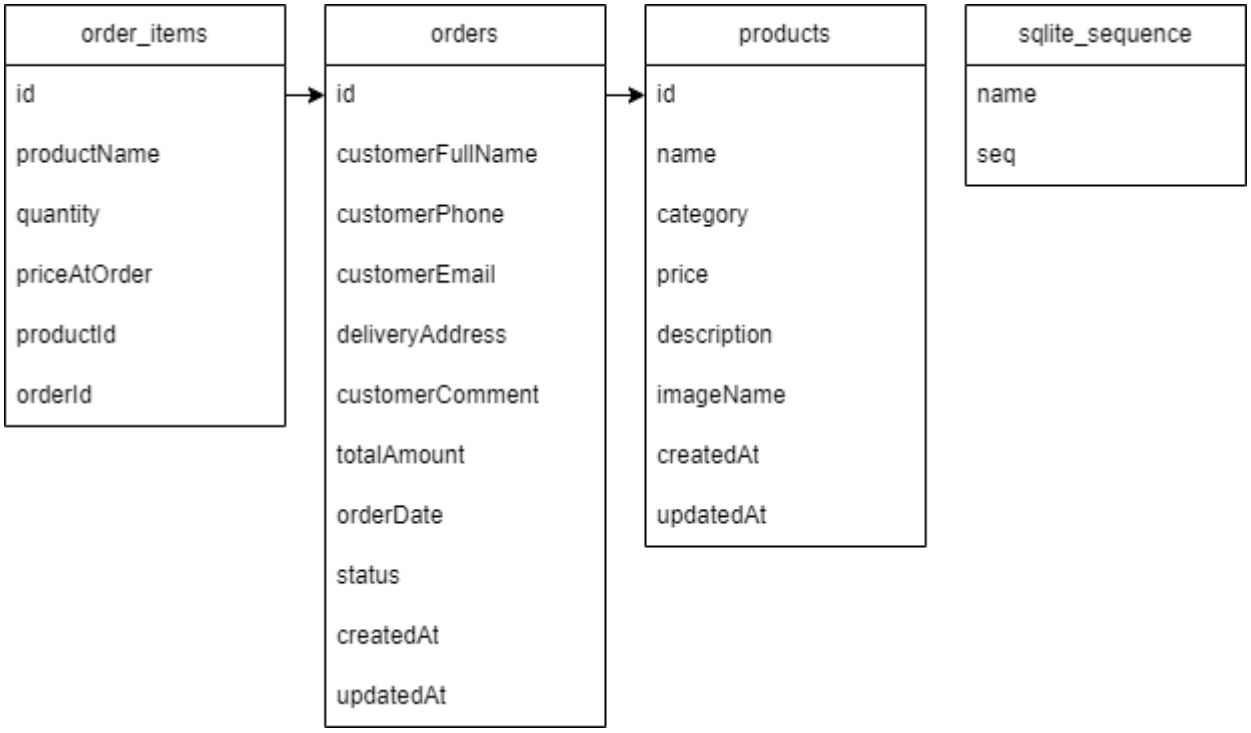
Веб-застосунок управління магазином військового спорядження

**Структура бази даних**

ІАЛЦ.467200.006 Д3

Аркушів 1

**Київ 2025 р**




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.006 Д3 | | |
| | | № докум. | Підпис | Дата | | | |
| Розробив | | Добровольський Ф.В. | | | **Веб-застосунок управління магазином військового спорядження** Структура бази даних | Літ. | Аркуш 1 / Аркушів 1 |
| Перевірив | | Меленчуков М.Є. | | | | | |
| | | | | | | НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-12 | |
| Н. Контр. | | Міщенко Л.Д. | | | | | |
| Затвердив | | | | | | | |

# ДОДАТОК Г

Веб-застосунок управління магазином військового спорядження

**Текст програмного коду**

ІАЛЦ.467200.007 Д4

Аркушів 15

**Київ 2025 р**

**Backend**

**server.js**

```
const express = require('express');
const sequelize = require('./config/database');
const Product = require('./models/Product');
const Order = require('./models/Order');
const OrderItem = require('./models/OrderItem');

const app = express();
const PORT = process.env.PORT || 3000;

app.use(express.json());

app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', 'http://localhost:4200');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS');
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');
  if (req.method === 'OPTIONS') {
    return res.sendStatus(200);
  }
  next();
});

async function initializeDatabaseAndStartServer() {
  try {
    await sequelize.authenticate();
    console.log('З\'єднання з базою даних SQLite успішно встановлено.');

    Product.hasMany(OrderItem, { foreignKey: 'productId', allowNull: true, onDelete: 'SET
NULL' });
    OrderItem.belongsTo(Product, { foreignKey: 'productId' });

    Order.hasMany(OrderItem, { as: 'items', foreignKey: 'orderId', onDelete: 'CASCADE' });
    OrderItem.belongsTo(Order, { foreignKey: 'orderId' });

    await sequelize.sync();
    console.log('Всі моделі успішно синхронізовано з базою даних.');

    const productsCount = await Product.count();
    if (productsCount === 0) {
      await Product.bulkCreate([
       { name: 'Тактичний рюкзак "Штурм-40"', category: 'Рюкзаки', price: 2800, description:
'Міцний рюкзак, MOLLE.', imageName: 'backpack-assault-40.jpg' },
         { name: 'Ліхтар "Промінь-500"', category: 'Ліхтарі', price: 950, description:
'Яскравий тактичний ліхтар.', imageName: 'flashlight-beam-500.jpg' },
        { name: 'Мультитул "Універсал"', category: 'Інструменти', price: 1200, description:
'Багатофункціональний інструмент.', imageName: 'multitool-universal.jpg' }
      ]);
      console.log('Додано тестові товари до бази даних.');
    }

    app.get('/', (req, res) => {
        res.send('Привіт! Це бекенд для Recon Depot! База даних підключена, моделі
синхронізовані.');
    });
```

```
    app.get('/api/products', async (req, res) => {
      try {
        const products = await Product.findAll();
        const productsWithFullUrl = products.map(product => ({
          ...product.toJSON(),
          imageUrl: product.imageName ? `/assets/images/${product.imageName}` : null
        }));
        res.json(productsWithFullUrl);
      } catch (error) {
        console.error('Помилка при отриманні товарів:', error);
        res.status(500).json({ message: 'Помилка на сервері при отриманні товарів' });
      }
    });
    app.post('/api/products', async (req, res) => {
      try {
        const { name, category, price, description, imageName } = req.body;
        if (!name || !category || price === undefined || price === null) {
           return res.status(400).json({ message: 'Будь ласка, надайте назву, категорію та
ціну товару.' });
        }
        if (typeof price !== 'number' || price <= 0) {
            return res.status(400).json({ message: 'Ціна має бути позитивним числом.' });
        }
        const newProduct = await Product.create({
          name,
          category,
          price,
          description: description || null,
          imageName: imageName || null
        });
        const productResponse = {
            ...newProduct.toJSON(),
              imageUrl: newProduct.imageName ? `/assets/images/${newProduct.imageName}` :
null
        };
        res.status(201).json(productResponse);
        console.log('Товар успішно додано через API:', productResponse);
      } catch (error) {
        console.error('Помилка при додаванні товару:', error);
        if (error.name === 'SequelizeValidationError') {
          const messages = error.errors.map(e => e.message);
          return res.status(400).json({ message: 'Помилка валідації', errors: messages });
        }
        res.status(500).json({ message: 'Помилка на сервері при додаванні товару' });
      }
    });
    app.delete('/api/products/:id', async (req, res) => {
      try {
        const productId = parseInt(req.params.id, 10);
        if (isNaN(productId)) {
          return res.status(400).json({ message: 'Некоректний ID товару.' });
        }
        const product = await Product.findByPk(productId);
        if (!product) {
          return res.status(404).json({ message: 'Товар не знайдено.' });
        }
        await product.destroy();
        res.status(200).json({ message: `Товар з ID ${productId} успішно видалено.` });
        console.log(`Товар з ID ${productId} успішно видалено через API.`);
      } catch (error) {
        console.error(`Помилка при видаленні товару з ID ${req.params.id}:`, error);
        res.status(500).json({ message: 'Помилка на сервері при видаленні товару' });
      }
    });

    app.put('/api/products/:id', async (req, res) => {
      try {
        const productId = parseInt(req.params.id, 10);
```

```
        if (isNaN(productId)) {
          return res.status(400).json({ message: 'Некоректний ID товару.' });
        }
        const product = await Product.findByPk(productId);
        if (!product) {
          return res.status(404).json({ message: 'Товар для оновлення не знайдено.' });
        }
        const { name, category, price, description, imageName } = req.body;
        if (price !== undefined && (typeof price !== 'number' || price <= 0)) {
          return res.status(400).json({ message: 'Ціна має бути позитивним числом.' });
        }
        const updatedFields = {};
        if (name !== undefined) updatedFields.name = name;
        if (category !== undefined) updatedFields.category = category;
        if (price !== undefined) updatedFields.price = price;
        if (description !== undefined) updatedFields.description = description;
        if (imageName !== undefined) updatedFields.imageName = imageName;
        if (Object.keys(updatedFields).length === 0) {
            return res.status(400).json({ message: 'Не надано даних для оновлення.' });
        }
        await product.update(updatedFields);
        const productResponse = {
          ...product.toJSON(),
          imageUrl: product.imageName ? `/assets/images/${product.imageName}` : null
        };
        res.status(200).json(productResponse);
       console.log(`Товар з ID ${productId} успішно оновлено через API:`, productResponse);
      } catch (error) {
        console.error(`Помилка при оновленні товару з ID ${req.params.id}:`, error);
        if (error.name === 'SequelizeValidationError') {
          const messages = error.errors.map(e => e.message);
          return res.status(400).json({ message: 'Помилка валідації при оновленні', errors:
messages });
        }
        res.status(500).json({ message: 'Помилка на сервері при оновленні товару' });
      }
    });

        app.post('/api/orders', async (req, res) => {
          const t = await sequelize.transaction();

          try {
            const { customerDetails, items, totalAmount } = req.body;

             if (!customerDetails || !items || !Array.isArray(items) || items.length === 0
|| totalAmount === undefined) {
              await t.rollback();
              return res.status(400).json({ message: 'Некоректні дані замовлення. Потрібні
деталі клієнта, товари та загальна сума.' });
            }
                        if   (!customerDetails.fullName   ||   !customerDetails.phone   ||
!customerDetails.email || !customerDetails.deliveryAddress) {
                await t.rollback();
                 return res.status(400).json({ message: 'Будь ласка, надайте повне ім\'я,
телефон, email та адресу доставки.' });
            }

            const newOrder = await Order.create({
              customerFullName: customerDetails.fullName,
              customerPhone: customerDetails.phone,
              customerEmail: customerDetails.email,
              deliveryAddress: customerDetails.deliveryAddress,
              customerComment: customerDetails.comment || null,
              totalAmount: totalAmount,
              orderDate: new Date(),
              status: 'new'
            }, { transaction: t });
```

```
            const orderItemsData = items.map(item => ({
              orderId: newOrder.id,
              productId: item.product.id,
              productName: item.product.name,
              quantity: item.quantity,
              priceAtOrder: item.product.price
            }));

            await OrderItem.bulkCreate(orderItemsData, { transaction: t });

            await t.commit();

            const createdOrderWithItems = await Order.findByPk(newOrder.id, {
                include: [{ model: OrderItem, as: 'items' }]
            });

            res.status(201).json(createdOrderWithItems);
                        console.log('Замовлення успішно створено через API:',
createdOrderWithItems.toJSON());

          } catch (error) {
            await t.rollback();
            console.error('Помилка при створенні замовлення:', error);
            if (error.name === 'SequelizeValidationError') {
              const messages = error.errors.map(e => e.message);
              return res.status(400).json({ message: 'Помилка валідації даних замовлення',
errors: messages });
            }
             res.status(500).json({ message: 'Помилка на сервері при створенні замовлення'
});
          }
        });

app.get('/api/orders', async (req, res) => {
  try {
    const orders = await Order.findAll({
      include: [{
        model: OrderItem,
        as: 'items',

      }],
      order: [['orderDate', 'DESC']]
    });

    res.status(200).json(orders);
    console.log('Успішно отримано список замовлень через API.');

  } catch (error) {
    console.error('Помилка при отриманні списку замовлень:', error);
    res.status(500).json({ message: 'Помилка на сервері при отриманні списку замовлень' });
  }
});

app.delete('/api/orders/:id', async (req, res) => {
  const t = await sequelize.transaction();
  try {
    const orderId = parseInt(req.params.id, 10);

    if (isNaN(orderId)) {
      await t.rollback();
      return res.status(400).json({ message: 'Некоректний ID замовлення.' });
    }

    const order = await Order.findByPk(orderId, { transaction: t });

    if (!order) {
      await t.rollback();
      return res.status(404).json({ message: 'Замовлення не знайдено.' });
```

```
        }

        await order.destroy({ transaction: t });

        await t.commit();
        res.status(200).json({ message: `Замовлення з ID ${orderId} успішно видалено.` });
        console.log(`Замовлення з ID ${orderId} успішно видалено через API.`);

      } catch (error) {
        await t.rollback();
        console.error(`Помилка при видаленні замовлення з ID ${req.params.id}:`, error);
        res.status(500).json({ message: 'Помилка на сервері при видаленні замовлення' });
      }
    });

        app.listen(PORT, () => {
          console.log(`Бекенд-сервер Recon Depot запущено на порту ${PORT}`);
        });

      } catch (error) {
        console.error('Не вдалося ініціалізувати базу даних або запустити сервер:', error);
      }
    }

    initializeDatabaseAndStartServer();
```

**Order.js**

```
    const { DataTypes, Model } = require('sequelize');
    const sequelize = require('../config/database');

    class Order extends Model {}

    Order.init({
      id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true,
        allowNull: false
      },
      customerFullName: {
        type: DataTypes.STRING,
        allowNull: false
      },
      customerPhone: {
        type: DataTypes.STRING,
        allowNull: false
      },
      customerEmail: {
        type: DataTypes.STRING,
        allowNull: false
      },
      deliveryAddress: {
        type: DataTypes.TEXT,
        allowNull: false
      },
      customerComment: {
        type: DataTypes.TEXT,
        allowNull: true
      },
      totalAmount: {
        type: DataTypes.FLOAT,
        allowNull: false
      },
      orderDate: {
        type: DataTypes.DATE,
        defaultValue: DataTypes.NOW,
```

```
    allowNull: false
  },
  status: {
    type: DataTypes.STRING,
    allowNull: false,
    defaultValue: 'new'
  }
}, {
  sequelize,
  modelName: 'Order',
  tableName: 'orders',
  timestamps: true
});

module.exports = Order;
```

**OrderItem.js**

```
const { DataTypes, Model } = require('sequelize');
const sequelize = require('../config/database');
class OrderItem extends Model {}
OrderItem.init({
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true,
    allowNull: false
  },
  productName: {
    type: DataTypes.STRING,
    allowNull: false
  },
  quantity: {
    type: DataTypes.INTEGER,
    allowNull: false
  },
  priceAtOrder: {
    type: DataTypes.FLOAT,
    allowNull: false
  },
  productId: {
    type: DataTypes.INTEGER,
    allowNull: true
  }
}, {
  sequelize,
  modelName: 'OrderItem',
  tableName: 'order_items',
  timestamps: false
});

module.exports = OrderItem;
```

**Product.js**

```
const { DataTypes, Model } = require('sequelize');
const sequelize = require('../config/database');

class Product extends Model {}

Product.init({
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true,
    allowNull: false
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false
```

```
  },
  category: {
    type: DataTypes.STRING,
    allowNull: false
  },
  price: {
    type: DataTypes.FLOAT,
    allowNull: false
  },
  description: {
    type: DataTypes.TEXT,
    allowNull: true
  },
  imageName: {
    type: DataTypes.STRING,
    allowNull: true
  }

}, {
  sequelize,
  modelName: 'Product',
  tableName: 'products',
  timestamps: true
});

module.exports = Product;
```

**Frontend**
**/src/**
**main.ts**

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

**index.html**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ReconDepot</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

**/src/app**
**app.component.html**

```
<div class="app-container">
<header class="app-header">
  <h1>Recon Depot</h1>
<nav class="app-nav">
    <a routerLink="/" routerLinkActive="active-link" [routerLinkActiveOptions]="{exact:
true}">Головна</a>
    <a routerLink="/cart" routerLinkActive="active-link">
      Кошик <span *ngIf="cartItemCount > 0">({{ cartItemCount }})</span>
    </a>
  </nav>
</header>
```

```
  <main class="main-content">
    <router-outlet></router-outlet>
  </main>

  <footer class="app-footer">
    <p>&copy; {{ currentYear }} Recon Depot. Всі права захищено.</p>
  </footer>
</div>
```

**app.component.ts**

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { RouterOutlet, RouterLink, RouterLinkActive } from '@angular/router';
import { CommonModule } from '@angular/common';
import { CartService } from './services/cart.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    CommonModule,
    RouterOutlet,
    RouterLink,
    RouterLinkActive
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent implements OnInit, OnDestroy {
  title = 'recon-depot';
  currentYear: number = new Date().getFullYear();
  cartItemCount: number = 0;

  private cartCountSubscription!: Subscription;

  constructor(private cartService: CartService) {}

  ngOnInit(): void {

    this.cartCountSubscription = this.cartService.cartItemCount$.subscribe(count => {
      this.cartItemCount = count;
      console.log('AppComponent: Cart count updated to', count);
    });
  }

  ngOnDestroy(): void {

    if (this.cartCountSubscription) {
      this.cartCountSubscription.unsubscribe();
    }
  }
}
```

**app.config.ts**

```
import { ApplicationConfig, provideZoneChangeDetection, LOCALE_ID, importProvidersFrom }
from '@angular/core';
import { provideRouter } from '@angular/router';
import { FormsModule } from '@angular/forms';
import { routes } from './app.routes';

import { registerLocaleData } from '@angular/common';
import localeUk from '@angular/common/locales/uk';

import { provideHttpClient } from '@angular/common/http';

registerLocaleData(localeUk);
```

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    { provide: LOCALE_ID, useValue: 'uk-UA' },
    importProvidersFrom(FormsModule),
    provideHttpClient()
  ]
};
```

**app.routes.ts**

```
import { Routes } from '@angular/router';
import { HomeComponent } from './pages/home/home.component';
import { ShoppingCartComponent } from './pages/shopping-cart/shopping-cart.component';
import { AdminLoginComponent } from './pages/admin-login/admin-login.component';
import { AdminDashboardComponent } from './pages/admin-dashboard/admin-
dashboard.component';
import { authGuard } from './guards/auth.guard';
import { CheckoutComponent } from './pages/checkout/checkout.component';

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'cart', component: ShoppingCartComponent },
  { path: 'checkout', component: CheckoutComponent },
  { path: 'admin-login', component: AdminLoginComponent },
  {
    path: 'admin/dashboard',
    component: AdminDashboardComponent,
    canActivate: [authGuard]
  },
  {
    path: 'admin',
    redirectTo: '/admin/dashboard',
    pathMatch: 'full'
  }
];
```

**/src/app/services**

**auth.service.ts**

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private loggedIn = new BehaviorSubject<boolean>(this.checkInitialLoginState());
  public isLoggedIn$ = this.loggedIn.asObservable();

  private readonly ADMIN_USERNAME = 'admin';
  private readonly ADMIN_PASSWORD = 'password123';

  constructor(private router: Router) {}

  private checkInitialLoginState(): boolean {

    return sessionStorage.getItem('isAdminLoggedIn') === 'true';
  }

  login(username: string, password: string):boolean {
    if (username === this.ADMIN_USERNAME && password === this.ADMIN_PASSWORD) {
      sessionStorage.setItem('isAdminLoggedIn', 'true');
      this.loggedIn.next(true);

      console.log('Admin logged in');
```

```
      return true;

    }
    console.log('Admin login failed');
    this.loggedIn.next(false);
    return false;
  }

  logout(): void {
    sessionStorage.removeItem('isAdminLoggedIn');
    this.loggedIn.next(false);
    console.log('Admin logged out');
    this.router.navigate(['/admin-login']);
  }

  isAdmin(): boolean {
    return this.loggedIn.value;
  }
}
```

**product.service.ts**

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable, of, BehaviorSubject } from 'rxjs';
import { catchError, tap, map, delay } from 'rxjs/operators';
import { Product } from '../models/product.model';

type ProductDataPayload = Omit<Product, 'id' | 'imageUrl' | 'createdAt' | 'updatedAt'>;

interface DeleteResponse {
  message: string;
}

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private productsApiUrl = 'http://localhost:3000/api/products';

  private productsSource = new BehaviorSubject<Product[]>([]);
  public products$ = this.productsSource.asObservable();

  private categoriesSource = new BehaviorSubject<string[]>([]);
  public categories$ = this.categoriesSource.asObservable();

  constructor(private http: HttpClient) {
    this.loadInitialProducts();
  }

  private loadInitialProducts(): void {
    this.http.get<Product[]>(this.productsApiUrl).pipe(
      tap(products => {
        this.productsSource.next(products);
        this.updateCategories(products);
        console.log('Товари успішно завантажено з бекенду:', products);
      }),
      catchError(this.handleError<Product[]>('завантаження товарів', []))
    ).subscribe();
  }

  public refreshProducts(): void {
    this.loadInitialProducts();
  }

  getProducts(): Product[] {
    return this.productsSource.getValue();
  }
```

```
  private updateCategories(products: Product[]): void {
      const categories = products.map(product => product.category).filter(Boolean) as
string[];
    this.categoriesSource.next([...new Set(categories)]);
  }

  getCategories(): string[] {
    return this.categoriesSource.getValue();
  }

  getProductById(id: number): Observable<Product | undefined> {

    const product = this.productsSource.getValue().find(p => p.id === id);
    if (product) {
        return of(product);
    } else {
         console.warn(`ProductService: Товар з ID ${id} не знайдено локально. Розгляньте
запит до API.`);
        return of(undefined);
    }
  }

  addProduct(productData: ProductDataPayload): Observable<Product> {
    console.log('ProductService: відправка нового товару на бекенд:', productData);
    return this.http.post<Product>(this.productsApiUrl, productData).pipe(
      tap((newProductFromBackend) => {
        console.log('Товар успішно додано на бекенді:', newProductFromBackend);
        this.refreshProducts();
      }),
      catchError(this.handleError<Product>('додавання товару'))
    );
  }

  deleteProduct(productId: number): Observable<boolean | DeleteResponse> {
    console.log(`ProductService: відправка запиту на видалення товару ID: ${productId} на
бекенд.`);
    const url = `${this.productsApiUrl}/${productId}`;

    return this.http.delete<DeleteResponse>(url).pipe(
      tap((response) => {
        console.log('Товар успішно видалено на бекенді:', response.message);
        this.refreshProducts();
      }),
      map(() => true),
      catchError(this.handleError<boolean>('видалення товару', false))
    );
  }

     updateProduct(productId: number, productData: Partial<ProductDataPayload>):
Observable<Product> {
    console.log(`ProductService: відправка запиту на оновлення товару ID: ${productId} на
бекенд:`, productData);
    const url = `${this.productsApiUrl}/${productId}`;

    return this.http.put<Product>(url, productData).pipe(
      tap((updatedProductFromBackend) => {
        console.log('Товар успішно оновлено на бекенді:', updatedProductFromBackend);
        this.refreshProducts();
      }),
      catchError(this.handleError<Product>('оновлення товару'))
    );
  }

  private handleError<T>(operation = 'operation', result?: T) {
    return (error: any): Observable<T> => {
        console.error(`${operation} не вдалося: ${error.message || error.statusText ||
error}`, error);
      return new Observable(observer => {
```

```
            observer.error(error.error || { message: `Помилка під час операції "${operation}"`
      });
            observer.complete();
          });
        };
      }
    }
```

**order.service.ts**

```
      import { Injectable } from '@angular/core';
      import { HttpClient, HttpHeaders } from '@angular/common/http';
      import { Observable, of, BehaviorSubject } from 'rxjs';
      import { catchError, tap } from 'rxjs/operators';
      import { CartItem } from './cart.service';
      import { OrderDetails } from '../pages/checkout/checkout.component';

      export interface Order {
        id: number | string;
        customerFullName?: string;
        customerPhone?: string;
        customerEmail?: string;
        deliveryAddress?: string;
        customerComment?: string;
        items: {
          id?: number;
          orderId?: number | string;
          productId: number | null;
          productName: string;
          quantity: number;
          priceAtOrder: number;
        }[];
        totalAmount: number;
        orderDate: string | Date;
        status: string;
        createdAt?: string | Date;
        updatedAt?: string | Date;
      }

      export interface OrderApiResponse extends Order {
      }

      @Injectable({
        providedIn: 'root'
      })
      export class OrderService {
        private ordersApiUrl = 'http://localhost:3000/api/orders';

        private adminOrdersSource = new BehaviorSubject<Order[]>([]);
        public adminOrders$ = this.adminOrdersSource.asObservable();

        constructor(private http: HttpClient) {
        }

         placeOrder(orderDetails: OrderDetails, cartItems: CartItem[], totalAmount: number):
      Observable<OrderApiResponse> {
          const payload = {
            customerDetails: orderDetails,
            items: cartItems.map(item => ({
              product: {
                id: item.product.id,
                name: item.product.name,
                price: item.product.price
              },
              quantity: item.quantity
            })),
            totalAmount: totalAmount
          };
```

```
        console.log('OrderService: Відправка замовлення на бекенд:', payload);
        console.log('URL:', this.ordersApiUrl);

        return this.http.post<OrderApiResponse>(this.ordersApiUrl, payload).pipe(
          tap((createdOrder) => {
            console.log('Замовлення успішно створено на бекенді:', createdOrder);
          }),
          catchError(this.handleError<OrderApiResponse>('розміщення замовлення'))
        );
      }

      loadAdminOrders(): Observable<Order[]> {
        console.log('OrderService: Запит на отримання всіх замовлень з бекенду');
        return this.http.get<Order[]>(this.ordersApiUrl).pipe(
          tap(orders => {
            const processedOrders = orders.map(order => ({
              ...order,
              orderDate: new Date(order.orderDate),
              createdAt: order.createdAt ? new Date(order.createdAt) : undefined,
              updatedAt: order.updatedAt ? new Date(order.updatedAt) : undefined,
            }));
            this.adminOrdersSource.next(processedOrders);
            console.log('Замовлення для адмін-панелі завантажено:', processedOrders);
          }),
          catchError(this.handleError<Order[]>('завантаження замовлень для адміна', []))
        );
      }

      deleteOrder(orderId: number | string): Observable<any> {
        const url = `${this.ordersApiUrl}/${orderId}`;
        console.log(`OrderService: Відправка запиту на видалення замовлення ID: ${orderId}`);
        return this.http.delete<any>(url).pipe(
          tap(() => {
            console.log(`Замовлення ID: ${orderId} успішно видалено на бекенді.`);
            const currentOrders = this.adminOrdersSource.getValue();
            this.adminOrdersSource.next(currentOrders.filter(order => order.id !== orderId));
          }),
          catchError(this.handleError<any>('видалення замовлення'))
        );
      }

      private handleError<T>(operation = 'operation', result?: T) {
        return (error: any): Observable<T> => {
          console.error(`Помилка під час операції "${operation}":`, error);
          const errorMessage = error.error?.message || error.message || `Серверна помилка під
час "${operation}"`;

          return new Observable(observer => {
            observer.error({
              message: errorMessage,
              status: error.status,
              details: error.error
            });
          });
        };
      }
    }
```

**cart.service.ts**

```
    import { Injectable } from '@angular/core';
    import { Product } from '../models/product.model';
    import { BehaviorSubject, Observable } from 'rxjs';

    export interface CartItem {
      product: Product;
      quantity: number;
    }
```

```
@Injectable({
  providedIn: 'root'
})
export class CartService {
  private items: CartItem[] = [];

  private cartItemsSource = new BehaviorSubject<CartItem[]>([]);
  public cartItems$ = this.cartItemsSource.asObservable();

  private cartItemCount = new BehaviorSubject<number>(0);
  public cartItemCount$ = this.cartItemCount.asObservable();

  constructor() {
    this.updateCartState();
  }

  addItem(productToAdd: Product): void {
    const existingItem = this.items.find(item => item.product.id === productToAdd.id);

    if (existingItem) {
      existingItem.quantity++;
    } else {
      this.items.push({ product: productToAdd, quantity: 1 });
    }
    this.updateCartState();
    console.log('Кошик оновлено:', this.items);
  }

  getItems(): CartItem[] {
    return this.items;
  }

  getTotalItemCount(): number {
    return this.items.reduce((total, item) => total + item.quantity, 0);
  }

  getTotalPrice(): number {
    return this.items.reduce((total, item) => total + (item.product.price * item.quantity),
0);
  }

  decrementItemQuantity(productId: number): void {
    const existingItem = this.items.find(item => item.product.id === productId);
    if (existingItem) {
      existingItem.quantity--;
      if (existingItem.quantity <= 0) {
        this.removeItem(productId);
      }
    }
    this.updateCartState();
  }

  incrementItemQuantity(productId: number): void {
    const existingItem = this.items.find(item => item.product.id === productId);
    if (existingItem) {
      existingItem.quantity++;
    }
    this.updateCartState();
  }

  removeItem(productId: number): void {
    this.items = this.items.filter(item => item.product.id !== productId);
    this.updateCartState();
  }

  clearCart(): void {
    this.items = [];
    this.updateCartState();
```

```
  }

  private updateCartState(): void {
    this.cartItemsSource.next([...this.items]);
    this.cartItemCount.next(this.getTotalItemCount());
  }

}
```