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

**КВАЛІФІКАЦІЙНА РОБОТА**
на здобуття освітнього ступеня бакалавра

на тему: **«Розробка інтерактивного веб-ресурсу для забезпечення он-лайн продажів»**

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

**Керівник:** ***Шатний С.В., кандидат технічних наук, доцент кафедри ЕММІТ***

**Рецензент:** *кандидат психологічних наук, доцент, доцент кафедри інформаційних технологій та аналітики даних Зубенко Ігор Ростиславович*

***РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ***

Завідувач кафедри інформаційних технологій та аналітики даних

_____ (проф., д.е.н. Кривицька О.Р.)

Протокол № 10 від «28» травня 2025 р.

Острог, 2025

**АНОТАЦІЯ**
**кваліфікаційної роботи**
**на здобуття освітнього ступеня бакалавра**

***Тема: Розробка інтерактивного веб-ресурсу для забезпечення он-лайн продажів.***

***Автор: Богач Віталій Едуардович***

***Науковий керівник: Шатний С.В., кандидат технічних наук, доцент кафедри ІТАД***

***Захищена «……»…… 20__ року.***

***Пояснювальна записка до кваліфікаційної роботи:*** *73 с., 18 рис., 1 табл., 4 додатків, 25 джерел.*

***Ключові слова:*** *інтернет-магазин, сервер, фронтенд, чиста архітектура, API, Swagger, HTML, CSS, стилі*

***Короткий зміст праці:***

*Завданням кваліфікаційної роботи, є створення веб-застосунку для он-лайн продажів одягу. Додаток має надавати простий та зрозумілий інтерфейс, щоб усе виглядало лаконічно та мінмалістично. Головними користувачами он-лайн сервісу "VaviBrand" будуть люди, які хочуть придбати одяг різного типу. Фронтенд частина створювалась за допомогою HTML та CSS на TypeScript. Бекенд частина була створена на ASP.NET Core. Коли велася розробка, були дотримані правила створення чистої архітектури.*

***Keywords:*** *online store, server, frontend, clean architecture, API, Swagger, HTML, CSS, styles*

***Summary of the work:***

*The task of the qualification work is to create a web application for online clothing sales. The application should provide a simple and understandable interface so that everything looks concise and minimalistic. The main users of the online service "VaviBrand" will be people who want to purchase various types of clothing. The front-end part was created using HTML and CSS on TypeScript. The back-end part was created on ASP.NET Core. When developing, the rules for creating a clean architecture were followed.*

_____

# ЗМІСТ

## КЛЮЧОВІ СЛОВА

1. HTML (HyperText Markup Language)
2. CSS (Cascading Style Sheets)
3. JavaScript (JS)
4. SQL (Structured Query Language)
5. API (Application Programming Interface)
6. UI (User Interface) - Користувацький інтерфейс
7. UX (User Experience) - Користувацький досвід
8. HTTP (Hypertext Transfer Protocol)
9. HTTPS (Hypertext Transfer Protocol Secure)
10. CMS (Content Management System) - Система управління контентом (якщо актуально)
11. CRUD (Create, Read, Update, Delete) - (якщо обговорюються операції з даними)
12. IDE (Integrated Development Environment) - Інтегроване середовище розробки (якщо згадуються інструменти)
13. MVC (Model-View-Controller) - (якщо використовується архітектурний шаблон)

# ВСТУП

Сучасний етап розвитку цифрових технологій суттєво впливає на всі сфери життя, особливо на сферу торгівлі. Онлайн-продажі одягу останніми роками демонструють стрімке зростання, що пов'язано зі зручністю, доступністю та широким асортиментом товарів, які пропонують інтернет-магазини. Однак, незважаючи на велику кількість існуючих рішень, багато платформ стикаються з проблемами у сфері юзабіліті, продуктивності та безпеки. Це створює потребу у розробці сучасних, технологічно просунутих веб-рішень, які б ефективно вирішували ці питання.

**Метою даної роботи** є створення інтерактивного веб-додатку для онлайн-продажів одягу під назвою "VaviBrand". Основним завданням є поєднання мінімалістичного дизайну, зручного інтерфейсу та високої продуктивності, що забезпечить комфортний досвід користувачів при виборі та покупці товарів. Важливим аспектом є також реалізація надійних механізмів захисту даних, що дозволить користувачам відчувати себе впевнено під час роботи з системою. Архітектура додатку має бути спроектована з урахуванням принципів масштабованості, що дасть можливість у майбутньому розширювати функціонал без серйозних змін у структурі коду.

**Об'єктом** дослідження виступає процес розробки веб-додатків для електронної комерції, зокрема використання сучасних технологій та підходів у створенні як клієнтської, так і серверної частин. Особлива увага приділяється клієнт-серверній архітектурі, яка є основою для побудови ефективних онлайн-сервісів. Дослідження також охоплює аналіз інструментів та фреймворків, що дозволяють забезпечити високу швидкодію, безпеку та зручність розробки.

**Предметом дослідження** є безпосередньо веб-додаток "VaviBrand", який включає фронтенд-частину з інтуїтивно зрозумілим інтерфейсом, розробленим за допомогою HTML, CSS та TypeScript, і бекенд-частину на основі ASP.NET

Core з інтеграцією бази даних SQL Server. Важливим елементом додатка є використання чистої архітектури, що забезпечує гнучкість та легкість підтримки коду. Система також включає механізми аутентифікації та авторизації, що базуються на JWT-токенах, для забезпечення безпеки даних користувачів.

У процесі роботи було проведено теоретичний аналіз ринку онлайн-продажів одягу, що дозволило визначити основні тенденції та потреби клієнтів. Далі було розроблено архітектуру додатку з урахуванням принципів чистої архітектури, що забезпечує його масштабованість та легкість у підтримці. Серверна частина реалізувалася за допомогою ASP.NET Core Web API, де особлива увага приділилася створенню моделей даних, контролерів та репозиторіїв для ефективної роботи з інформацією. Клієнтська частина розроблялася з акцентом на юзабіліті, що забезпечила зручний та швидкий доступ до товарів. Завершальним етапом було тестування системи, виявлення та усунення можливих помилок, а також оптимізація продуктивності.

**Результатом** роботи є повнофункціональний веб-додаток, який поєднує в собі сучасний дизайн, зручність використання та високу продуктивність. Цей продукт має не лише академічну цінність, але й практичне застосування, оскільки він може бути впроваджений у реальних умовах електронної комерції. Розроблений додаток демонструє ефективність використання сучасних технологій у веб-розробці та може стати основою для подальшого вдосконалення та розширення функціоналу.

# РОЗДІЛ 1
# ОГЛЯД ІСНУЮЧИХ ЗАСОБІВ ПО ПОСТАВЛЕНОМУ ЗАВДАННЮ

## 1.1. Огляд конкурентів за напрямком роботи

Для оцінки конкурентного середовища було проведено SWOT-аналіз та порівняльну характеристику основних гравців українського ринку онлайн-продажів одягу. Дослідження охопило 15 ключових компаній, що працюють у цьому сегменті.

*Категорії конкурентів:*

1.Міжнародні маркетплейси: Zalando, ASOS, H&M. Їхні переваги це глобальний асортимент, міцна репутація, а недоліки це високі ціни через імпорт, довгі терміни доставки.

2.Українські маркетплейси: Rozetka, Prom.ua, Crafta.ua. Їх сильні сторони: локальна присутність, україномовний інтерфейс, а слабкі сторони: недостатня спеціалізація на моді.

3.Нішеві українські бренди: Etnodim, Kachorovska, Varsi. Конкурентні переваги: унікальний дизайн, якість матеріалів, а обмеження: вузький асортимент

4. Українські онлайн-магазини одягу

Це прямі конкуренти, які пропонують схожий асортимент і цінові категорії:

Rozetka (розділ одягу) – великий маркетплейс із широким вибором брендів.

Prom.ua / Shafa – агрегатори, де представлені численні українські виробники.

EVA.ua – популярний магазин жіночого одягу та взуття.

Intertop – спеціалізується на взутті та спортивному одязі.

ModnaKasta – середній і преміальний сегмент, акцент на розпродажах.

Answear.ua – мультибрендовий магазин (є як українські, так і зарубіжні марки).

### 1.2. Огляд технології для розробки

Сучасні веб-додатки, особливо в сфері електронної комерції, базуються на клієнт-серверній архітектурі – моделі, що передбачає чітке розділення обов'язків між клієнтською та серверною частинами.

Клієнт-серверна архітектура ґрунтується на взаємодії двох основних компонентів: клієнта (frontend)[2] і сервера (backend)[3]. Клієнтська частина, яку користувач бачить у своєму браузері або мобільному додатку, відповідає за відображення інтерфейсу, обробку дій користувача та відправку запитів на сервер. Вона реалізована за допомогою сучасних фреймворків, таких як React або Vue.js, що дозволяє створювати динамічні односторінкові додатки (SPA) з інтуїтивно зрозумілим UX[5].

Серверна частина виконує ключову роль у системі – обробляє запити від клієнта, взаємодіє з базою даних, виконує бізнес-логіку та повертає результати у форматі, зрозумілому для frontend. Для цього використали сервери додатків (Node.js, ASP.NET Core) та системи керування базами даних (SQL Server[4]).

Комунікація між клієнтом і сервером відбувається за стандартними протоколами, переважно HTTP/HTTPS, через API (REST). REST API визначає набір ендпоінтів, за допомогою яких frontend може отримувати або відправляти

дані, а також виконувати інші операції[6]. Для забезпечення безпеки використовуються механізми автентифікації JWT (JSON Web Tokens), що дозволяють ідентифікувати користувачів і контролювати доступ до ресурсів.

Існують різні підходи до побудови клієнт-серверних систем, кожен з яких має свої переваги та недоліки.

Для нашого додатку ми використали трирівнева архітектура, яка включає:

- клієнтський рівень (frontend);
- сервер додатків (backend, де виконується бізнес-логіка);
- базу даних.

Для платформи електронної комерції "VaviBrand" було обрано трирівневу архітектуру, оскільки вона оптимально поєднує продуктивність, масштабованість і відносну простоту підтримки.

На клієнтському рівні був розроблений SPA-додаток на React, що забезпечив швидкий та інтерактивний інтерфейс для покупців. Ці технології дозволили реалізувати сучасний UI/UX, швидко завантажувати контент і працювати в офлайн-режимі (PWA).

Серверна частина була побудована на ASP.NET Core – потужному фреймворку для створення високопродуктивних API. Він дозволяє ефективно обробляти запити, інтегруватися з різними платіжними системами та забезпечувати безпеку даних[7].

Для зберігання інформації використовувався SQL Server разом із Entity Framework Core – це дозволило зручно працювати з даними, забезпечувати транзакційність і швидко масштабувати схему бази даних.

Щоб забезпечити стабільну роботу "VaviBrand", було враховано кілька ключових моментів:

1. Оптимізація API: REST-ендпоінти повинні бути добре структуровані, з мінімальною кількістю запитів для отримання даних.

2. Безпека: реалізація JWT-аутентифікації, захист від SQL-ін'єкцій, CSRF-атак та інших загроз.

3. Кешування: використання Redis або подібних технологій для прискорення роботи з частими запитами (наприклад, каталог товарів).

4. Балансування навантаження: можливість додавати нові сервери в разі зростання трафіку.

Отже, клієнт-серверна архітектура стала вибором для "VaviBrand", оскільки вона забезпечила гнучкість, безпеку та можливість масштабування. Використання сучасних технологій (React/Vue.js, ASP.NET Core, SQL Server) дозволило створити швидкий, надійний і зручний для користувачів сервіс, здатний конкурувати на ринку електронної комерції.

### 1.3. Порівняльна таблиця огляду

**Таблиця 1.1. Порівняльна таблиця основних конкурентів**

| **Критерій** | **Міжнародні** | **Українські маркетплейси** | **Нішеві бренди** | **VaviBrand ( розроблений проект)** |
|---|---|---|---|---|
| Ціновий діапазон | 500 доларів | 50-500 доларів | 200-500 доларів | 50-500 доларів |
| Термін доставки | 7-14 днів | 1-3 дні | 1-5 днів | 1-3 дні |
| Українські бренди | <10% | 30-40% | 100% | 80-90% |
| Мобільний додаток | Так | Так | Ні | Так |
| Персоналізація | Середня | Низька | Висока | Висока |

**Таблиця 1.2. SWOT-аналіз онлайн-сервісу "VaviBrand"**

| Сильні сторони | Слабкі сторони | Можливості | Загрози |
|---|---|---|---|
| Свій, український! Ми робимо ставку на українських виробників (80-90% товарів), підтримуючи свій бізнес, особливо в такий складний час. Це дуже подобається людям! | Мало грошей на рекламу. Нам важко змагатися з великими магазинами, у яких є величезні рекламні бюджети. | Мода на українське! Зараз люди активно підтримують українське виробництво, купують наше. Це наш шанс! | Нестабільна економіка. Якщо грошей у людей стає менше, вони менше купують. Також на нас впливає курс валют. |
| Ми дружимо з технологіями! У нас є круті "фішки": штучний інтелект, який допомагає підібрати стиль, і навіть віртуальна примірка. А ще — зручний мобільний додаток і можливість продавати через соцмережі. | Залежимо від українських виробників. Якщо у когось з постачальників виникнуть проблеми, це може вплинути на наш асортимент. І поки що у нас не дуже багато дуже дорогих, "люксових" речей. | Всі сидять у смартфонах. Дедалі більше людей купують через телефони. Ми можемо використовувати це, пропонуючи щось особливе, коли людина поруч з магазином, або цікаві акції прямо в мобільному додатку. | Великі "акули" бізнесу. Великі інтернет-магазини можуть пропонувати дуже низькі ціни і багато реклами, з якими нам важко конкурувати. |
| Ціна і якість – золота середина! Ми пропонуємо хороший одяг за адекватну ціну (середній чек | Технології – це дорого. Всі ці круті штуки на зразок штучного інтелекту потребують | Об'єднуємось з іншими українськими брендами. Можна робити спільні колекції або | Проблеми з доставкою. Війна ускладнює логістику, і доставка може |

| | | | |
|---|---|---|---|
| $30-50). До того ж, у нас є знижки та бонуси для постійних клієнтів. | постійних вкладень, і безпека даних теж коштує грошей. | рекламуватися разом. | дорожчати або затримуватися. |

**Висновки до розділу 1**

Проведений аналіз теоретичних основ розробки онлайн-сервісу для продажу одягу дозволив виявити ключові тенденції та особливості сучасного ринку електронної комерції, зокрема в українському сегменті. Дослідження показало, що українські споживачі дедалі більше орієнтуються на онлайн-шопінг, що зумовлено зручністю, доступністю та широким асортиментом товарів.

Аналіз клієнт-серверної архітектури підтвердив доцільність використання трирівневої моделі для веб-додатку "VaviBrand", що дозволило забезпечити оптимальний баланс між продуктивністю, безпекою та масштабованістю. Особливу увагу приділено принципам чистої архітектури, які дали змогу створити гнучке та легке в підтримці програмне рішення, здатне адаптуватися до змін ринкових умов.

Конкурентний аналіз виявив, що основним перевагою "VaviBrand" є поєднання технологічних інновацій (AI-підбір стилю, віртуальна примірка) з акцентом на українських виробників, що відрізняє сервіс від міжнародних гравців.

Підсумовуючи, отримані результати свідчать про те, що наш додаток "VaviBrand" має добрі перспективи на українському ринку за умови правильного використання конкурентних переваг та подолання існуючих викликів.

# РОЗДІЛ 2
# ПРОЕКТУВАННЯ РОЗРОБЛЮВАНОГО ОНЛАЙН СЕРВІСУ

## 2.1. Серверна частина застосунку “VaviBrand”

### 2.1.1. Розробка структурної схеми продукту

Серверна компонента системи побудована на основі інноваційного технологічного стеку, що поєднує переваги сучасних веб-технологій та надійних рішень від Microsoft. Ядром архітектури виступає ASP.NET Core Web API, що реалізує RESTful підхід, який забезпечує високу продуктивність, масштабованість та гнучкість у розробці.

Інтеграція Swagger[12] через NuGet пакет значно підвищує ефективність процесу розробки, надаючи інтерактивну документацію API у реальному часі, автоматизовані інструменти для тестування ендпоінтів та зручний інтерфейс для взаємодії з API, що спрощує як розробку, так і подальше супроводження системи.

Основним принципом організації API є строга адресність ресурсів, де кожен об'єкт системи має чітко визначений унікальний URI. Наприклад, колекція автомобілів доступна за адресою /api/cars, тоді як конкретний автомобіль ідентифікується через /api/cars/{id}, де {id} представляє собою унікальний GUID ідентифікатор. Така структура дозволяє легко керувати ресурсами та забезпечує прозорість архітектури.

Для реалізації CRUD операцій використовуються стандартні HTTP методи: GET для отримання ресурсів, POST для створення нових записів, PUT для повного оновлення існуючих даних та DELETE для видалення ресурсів. Цей підхід забезпечує стандартизований інтерфейс взаємодії з системою та відповідає сучасним best practices у розробці веб-API.

Робота з даними реалізована через потужний стек технологій Microsoft, де Microsoft SQL Server[13] виступає як високопродуктивна система керування реляційними базами даних, а SQL Server Management Studio (SSMS) надає професійні інструменти для адміністрування БД, оптимізації запитів та налаштування безпеки.

Використання Entity Framework 6 з підходом Code First дозволяє генерувати структуру бази даних безпосередньо з моделей C#, керувати міграціями, оптимізувати продуктивність запитів та реалізовувати складні зв'язки між сутностями, що значно прискорює процес розробки та забезпечує узгодженість між об'єктно-орієнтованою моделлю даних та реляційною схемою бази даних.

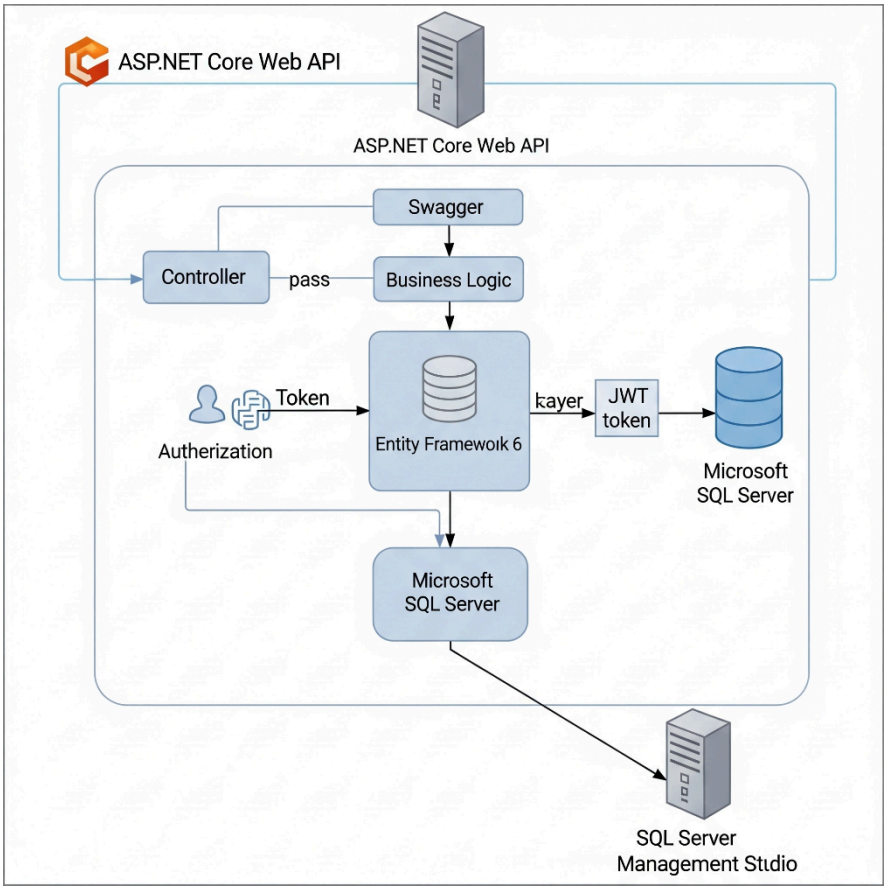


Рис. 2.1. Структурна схема

Джерело:[створено автором]

Обраний технологічний стек пропонує численні переваги, включаючи високу продуктивність з низькими затримками, масштабованість архітектури, сувору типізацію даних, вбудовані механізми безпеки, підтримку асинхронних операцій та крос-платформенність рішення.

Аутентифікацію було виконано через Nuget пакети Microsot.AspNetCore.Identity.EntityFrameworkCore та для генерації JWT токенів Microsoft.AspNetCore.Authentication.JwtBearer.

Також, ще використано AutoMapper – це невелика бібліотека, створена для того, щоб реалізувати співвідношення одного об'єкта до іншого [14].

### 2.1.2. Етапи створення проєкту

Проєкт "Vavibrand" реалізований у Visual Studio 2022 з використанням сучасної модульної архітектури, що включає чотири основні структури:

**Application Layer** виконує ключову роль у програмній архітектурі, виступаючи основним прошарком для реалізації бізнес-логіки програми. Цей рівень відповідає за обробку складних бізнес-правил та координацію взаємодії між різними компонентами системи, забезпечуючи при цьому чітке відокремлення логіки від інфраструктурних деталей.

Важливою характеристикою Application Layer є використання механізму Dependency Injection (DI), який дозволяє ефективно керувати залежностями між компонентами системи. Цей підхід значно підвищує гнучкість архітектури.

Для організації передачі даних між різними частинами системи Application Layer використовує спеціальні об'єкти DTO (Data Transfer Objects). Ці легковісні структури даних оптимізовані для ефективної передачі інформації між рівнями програми, мінімізуючи накладні витрати та забезпечуючи чітку контрактну основу для взаємодії компонентів.

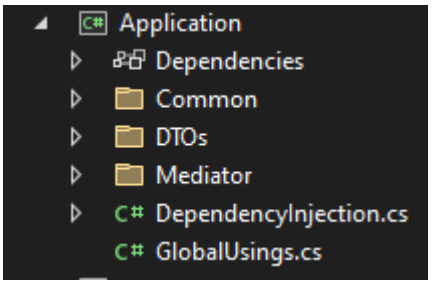


Рис. 2.2. Структура Applicarion Layer

Джерело:[створено автором]

**Domain Layer** становить концептуальну основу всієї системи, втілюючи її фундаментальні бізнес-сутності та ключові доменні моделі. Цей рівень архітектури виступає як інтелектуальне ядро, що інкапсулює найважливіші бізнес-правила та логіку предметної області, незалежно від технічних деталей реалізації.

Важливою особливістю Domain Layer є використання глобальних using-директив, які спрощують доступ до ключових просторів імен та базових типів по всій кодобазі доменного шару.

Архітектурна чистота Domain Layer досяглася нам за рахунок суворого відокремлення бізнес-логіки від інфраструктурних аспектів. Цей шар містить мінімальну кількість технічних залежностей, фокусуючись виключно на реалізації бізнес-правил та підтримці цілісності даних. Всі зовнішні інтеграції та технічні деталі виносяться на інші рівні архітектури, що дозволяє зберігати доменну модель чистою та зрозумілою.

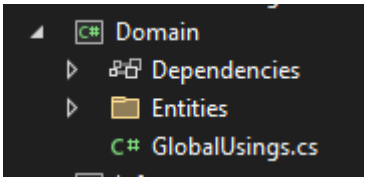


Рис. 2.3. Структура Domain Layer

Джерело:[створено автором]

**Infrastructure Layer** (інфраструктурний рівень) відповідає за зовнішні інтеграції системи. Він забезпечує підключення та взаємодію з базою даних за допомогою ORM-технології Entity Framework. Також цей рівень реалізує механізми автентифікації користувачів, використовуючи систему Identity. Крім того, тут налаштовується й керується процес міграцій бази даних для оновлення її структури без втрати даних. Окрему роль відіграють репозиторії та сервіси, що розміщуються в цій частині архітектури — вони відповідають за доступ до даних і реалізацію бізнес-логіки, яка не пов’язана безпосередньо з користувацьким інтерфейсом.

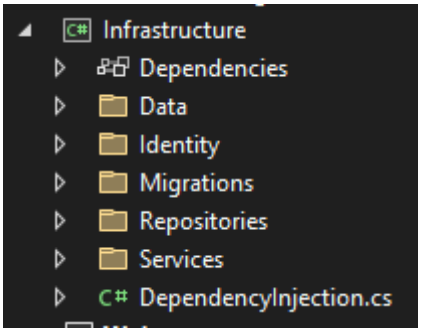


Рис. 2.4. Структура Infrastructure Layer

Джерело:[створено автором]

**Web Layer** є представницьким рівнем системи, який відповідає за обробку HTTP-запитів і взаємодію із зовнішніми користувачами або іншими сервісами. Він містить контролери ASP.NET Core Web API, які реалізують точки входу для API-запитів і взаємодіють із сервісами застосунку. У цьому шарі також розташовані конфігураційні файли, зокрема appsettings.json, який використовується для зберігання налаштувань додатку. Файл Program.cs визначає стартову точку запуску програми та конфігурує основні компоненти ASP.NET Core.

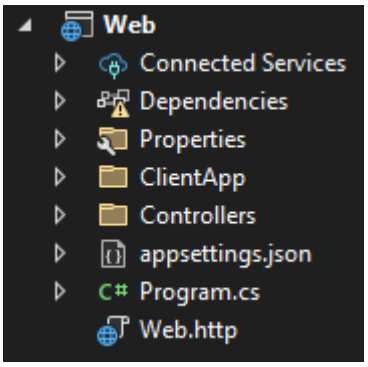


Рис. 2.5. Структура Web Layer

Джерело:[створено автором]

Основні особливості структури складаються з чітких розділень відповідальностей за принципом Clean Architecture**,** модульну організацію з чітко визначеними межами**,** централізоване керування залежностями через DependencyInjection**,** підтримку DDD (Domain-Driven Design) підходу.

**Технологічний стек складається з:**

1. ASP.NET Core 6.0 для Web API
2. Entity Framework Core для роботи з даними
3. MediatR для реалізації CQRS
4. AutoMapper для об'єктно-об'єктного відображення
5. Swagger для документації API

**Процес створення складався з таких етапів:**

Етап 1. Ініціалізація рішення з чотирма проектними слоями

Етап 2. Налаштування міжпроектних залежностей

Етап 3. Конфігурація Dependency Injection

Етап 4. Імплементація базових інтерфейсів та сервісів

Етап 5. Налаштування інфраструктурних компонентів

Етап 6. Підключення клієнтської частини

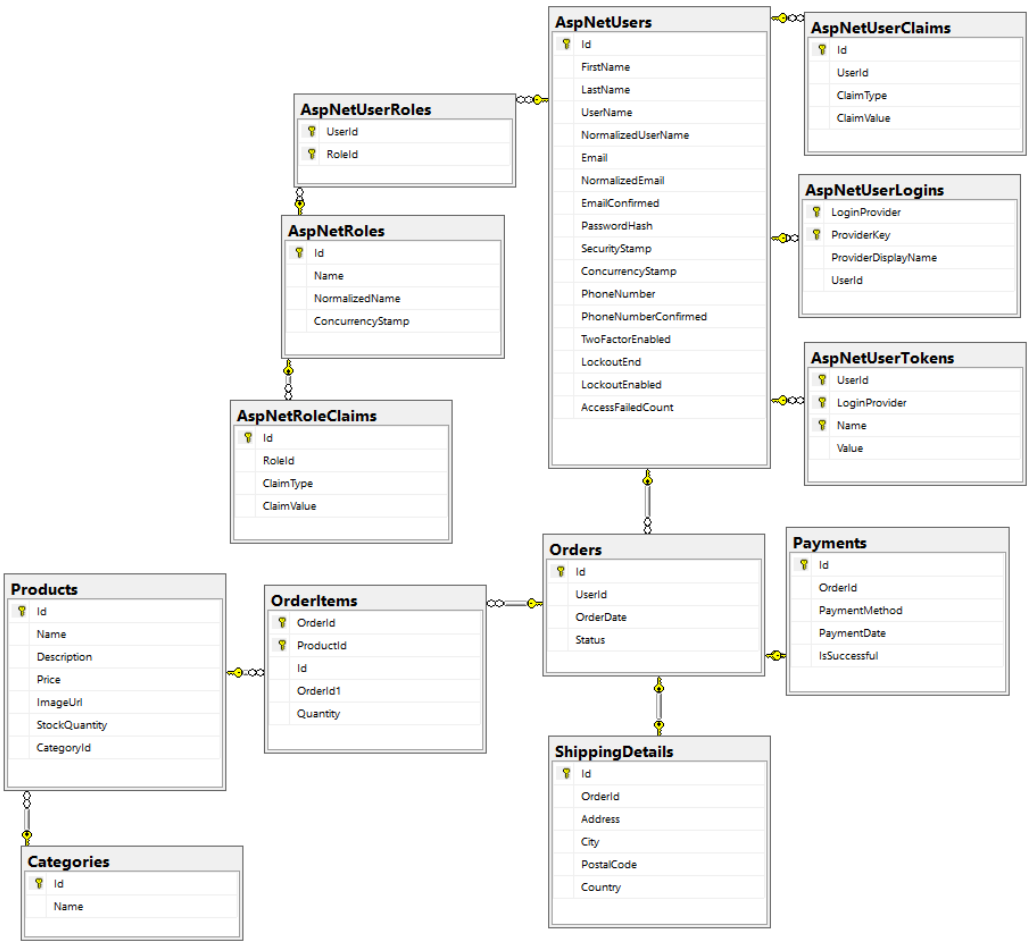


Рис. 2.6. Представлена модель створеної бази даних

Джерело: [створено автором]

Блок-схеми можуть бути використані для загального представлення високорівневої архітектури веб-ресурсу, показуючи основні модулі (наприклад, фронтенд, бекенд, база даних, платіжний шлюз) та потоки даних між ними. Це забезпечує чітке розуміння загального устрою системи.

UML діаграми надають більш деталізоване та стандартизоване моделювання:

**(Use Case Diagram):**

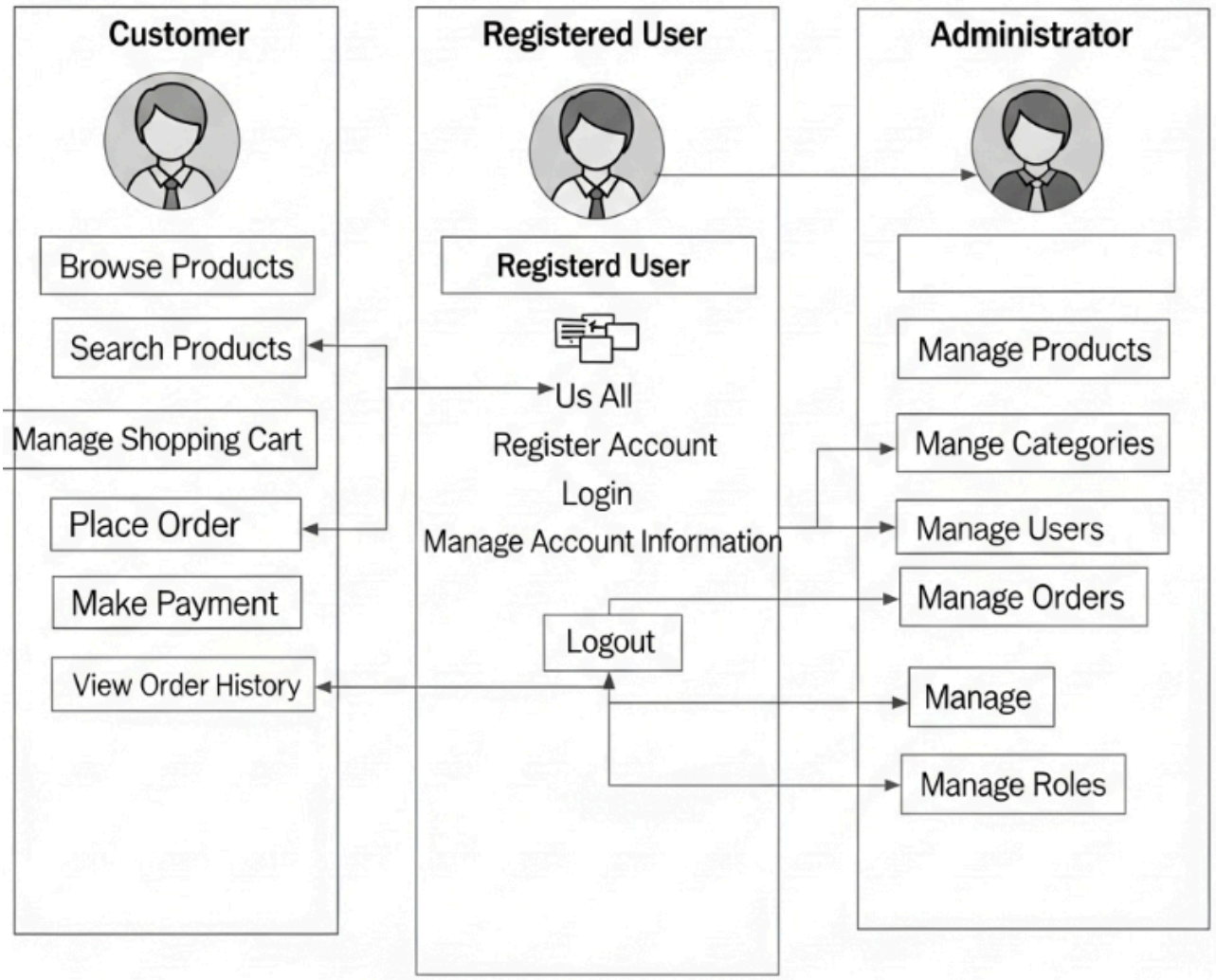


Рис. 2.7. Case diagram

Джерело:[створено автором]

### 2.1.4. Контролери та репозиторії бекенд частини

Цей підрозділ описує роль контролерів та репозиторіїв у серверній частині веб-додатку "VaviBrand", їхню взаємодію та внесок у загальну архітектуру.

Контролери в ASP.NET Core Web API відповідають за обробку HTTP-запитів від клієнтської частини (frontend). Вони діють як посередники між клієнтом та бізнес-логікою програми.

Функції контролерів:

Прийом HTTP-запитів (GET, POST, PUT, DELETE), маршрутизація запитів до відповідних методів обробки, валідація вхідних даних (за потреби), виклик методів репозиторіїв для виконання операцій з даними, повернення HTTP-відповідей (з кодами статусу та даними) у форматі JSON.

Репозиторії є посередниками між контролерами та рівнем доступу до даних (Data Access Layer), де знаходиться контекст бази даних (EF Core). Вони інкапсулюють логіку взаємодії з базою даних, забезпечуючи абстракцію та розділення відповідальностей.

Функції репозиторіїв включають виконання операцій CRUD (Create, Read, Update, Delete) з базою даних, інкапсуляція логіки доступу до даних, щоб контролери не знали, як саме дані зберігаються або отримуються, забезпечення можливості легко замінювати технології доступу до даних (наприклад, зміна ORM або СУБД) без зміни коду контролерів, реалізація складних запитів до бази даних.

Лістинг 2.6. Код файлу IGenericRepository.cs проєкту Application

```
using System.Linq.Expressions;

namespace Application.Common.Interfaces
{
    public interface IGenericRepository<T> where T : class
    {
        Task<T> GetByIdAsync(int id);
        Task<IEnumerable<T>> GetAllAsync();
        Task AddAsync(T item);
        void Update(T item);
        Task DeleteAsync(int id);
        Task<T> FindAsync(Expression<Func<T, bool>> predicate);
        Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> predicate);
    }
}
```

Джерело: [створено автором]

Лістинг 2.7. Код файлу GenericRepository.cs проєкту Infrastructure

```
using Application.Common.Interfaces;
using Infrastructure.Data;
```

```
using Microsoft.EntityFrameworkCore;
using System.Linq.Expressions;

namespace Infrastructure.Repositories
{
    public class GenericRepository<T> : IGenericRepository<T> where T : class
    {
        private readonly DataContext _context;
        private readonly DbSet<T> _dbSet;

        public GenericRepository(DataContext context)
        {
            _context = context;
            _dbSet = context.Set<T>();
        }

        public async Task<T> GetByIdAsync(int id)
        {
            return await _dbSet.FindAsync(id);
        }

        public async Task<IEnumerable<T>> GetAllAsync()
        {
            return await _dbSet.AsNoTracking().ToListAsync();
        }

    }
      }
```

Джерело: [створено автором]

Повний фрагмент коду розташований в додатках

Цей репозиторій є основним і він наслідується від інтерфейсу IGenericRepository, отже він реалізовує усі методи які були створені в нашому інтерфейсі. Весь код репозиторію винесено в ДОДАТОК А.

Наступний крок, це контролер, візьмемо для прикладу обробку категорій. Розглянемо метод GetCategory(int id). Цей метод реалізує виведення категорій по конкретному вказаному ідентифікатору (id). Наглядно видно, що є атрибут [HttpGet] з назвою "get-category/{id}". Назва була вказана для покращення обробки та читання зпитів. Повний код додали в ДОДАТОК Б.

Лістинг 2.8. Код файлу CategoryController.cs проєкту Web

```
[HttpGet("get-category/{id}")]
 public async Task<IActionResult> GetCategory(int id)
 {
     var category = await _mediator.Send(new GetCategoryQuery { Id = id });
     if ( category == null )
     {
         return NotFound(new { message = "Категорія не знайдена" });
```

```
    }
    return Ok(category);
      }
```

Джерело: [створено автором]

Також, наступний можна розглянути метод для створення категорій

Лістинг 2.9. Код файлу CategoryController.cs проєкту Web

```
[HttpPost("create-category")]
public async Task<IActionResult> CreateCategory([FromBody] CreateCategoryCommand
command)
{
    var categoryId = await _mediator.Send(command);
    return Ok(categoryId);
      }
```

Джерело: [створено автором]

Видно, що ідентифікатор реалізовується через медіатор.

## 2.2. Проектування інтерфейсу користувача

### 2.2.1. Загальна інформація про клієнтську частину

Після розробки сервера з базою даних для управління ресурсами, вони відображаються користувачам за допомогою клієнтського застосунку на Angular.

Обмін даними між сервером і клієнтом відбувається у форматі JSON. Для представлення інформації використано HTML (розмітка), CSS (стилізація з Bootstrap) та TypeScript (логіка).

### 2.2.2. Структура клієнтської частини

Архітектура Angular-застосунків ґрунтується на модулях, які агрегують компоненти, сервіси та різноманітні утиліти. Отже, модулі являють собою комплексну структуру, що містить всю логіку функціонування застосунку.

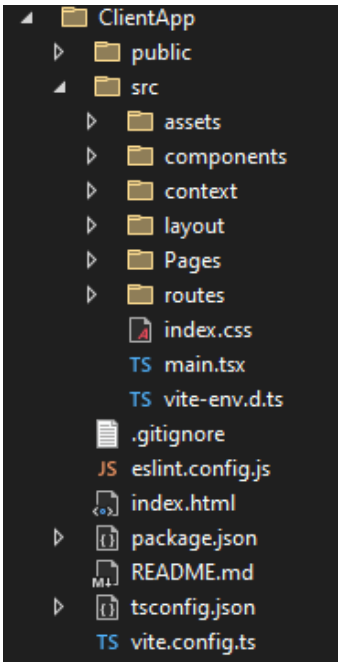


Рис. 2.8. Структура клієнтської частини

Джерело:[створено автором]

У нас є основні папки: assets, components, context, layout, pages, routex.

**assets** містить статичні ресурси, такі як зображення, стилі, шрифти тощо, які використовуються в застосунку. **Components -** це ключові будівельні блоки Angular-застосунку. Кожен компонент відповідає за певну частину інтерфейсу користувача. Наприклад, це можуть бути компоненти для відображення списку товарів, форми авторизації, кошика покупок тощо.

layout містить компоненти, що визначають загальну структуру сторінки (наприклад, header, footer, navigation). **Pages** містить компоненти, які представляють окремі сторінки застосунку (наприклад, головна сторінка, сторінка каталогу, сторінка товару). В Angular часто використовуються компоненти всередині app.component.ts та роутинг. **Routes** це файли, що визначають навігацію між сторінками застосунку. В Angular це конфігурація Angular Router.

В Angular модулі застосовуються для імпортування як сторонніх, так і власних модулів для їх інтеграції в проєкт. Також, кожен модуль може містити в собі різні компоненти, допоміжні функції.

Лістинг 2.10. Фрагмент коду файлу Home.tsx проєкту Web

```
import Swiper from "swiper";
import { Navigation } from "swiper/modules";
import "../../node_modules/swiper/swiper.css";
import "../../node_modules/swiper/modules/pagination.css";
import "../../node_modules/swiper/modules/navigation.css";
import { useEffect } from "react";

export default function Home() {
  useEffect(() => {
    new Swiper(".main-swiper", {
      slidesPerView: 3,
      height: 300,
      modules: [Navigation],
      direction: "horizontal",
      loop: true,
      spaceBetween: 100,
      navigation: {
        nextEl: ".swiper-button-next",
        prevEl: ".swiper-button-prev",
      },
      breakpoints: {
        300: { slidesPerView: 1, spaceBetween: 20 },
        768: { slidesPerView: 2, spaceBetween: 20 },
        1200: { slidesPerView: 3, spaceBetween: 80 },
      },
    });
  }, []);

  return (
    <>
      <section id="billboard" className="bg-light py-5">
        <div className="container">
          <div className="row justify-content-center">
            <h1
              className="section-title text-center mt-4"
              style={{ fontSize: "4.5rem" }}
              data-aos="fade-up"
            >
```

Джерело: [створено автором]

Детальний код представлений в додатку 1

### 2.2.3. Допоміжні компоненти розроблюваного сервісу

Компоненти є фундаментальними будівельними блоками будь-якого Angular-застосунку. Вони відповідають за відображення частин інтерфейсу користувача та обробку взаємодії з ним. Кожен компонент в Angular - це клас TypeScript, який тісно пов'язаний з HTML-шаблоном і, за потреби, CSS-стилями. Клас TypeScript визначає логіку компонента, включаючи дані, методи для обробки подій та іншу функціональність, тоді як HTML-шаблон визначає структуру та вміст інтерфейсу, який компонент відображає. CSS-стилі відповідають за зовнішній вигляд компонента.

Зазвичай, кожен компонент організовується у власній папці, що містить файли: component-name.component.ts (клас TypeScript), component-name.component.html (HTML-шаблон), component-name.component.css (або інші формати, наприклад, .scss, .less - файли зі стилями), і component-name.component.spec.ts (необов'язковий файл з тестами для компонента).

Перш за все, це компоненти макету (Layout Components), які визначають загальну структуру сторінки, наприклад, header (шапка сайту), footer (підвал сайту) та navigation (меню навігації). Далі йдуть компоненти сторінок (Page Components), що представляють окремі сторінки застосунку, такі як home (головна сторінка), catalog (сторінка каталогу товарів), product-details (сторінка з детальною інформацією про товар), cart (кошик покупок), checkout (сторінка оформлення замовлення), а також сторінки аутентифікації login та register. Існують також компоненти, що можуть бути багаторазово використані в різних частинах застосунку (Reusable Components), наприклад, product-card (картка товару), filter (фільтр товарів), search-bar (рядок пошуку) та pagination (посторінкова навігація).

Наприклад, для демонстрації візьмемо код пошукового вікна. SearchPopup, який реалізує функціональність пошукового вікна. Він імпортує

хуки useEffect, useRef та useState з React. Компонент приймає пропси isVisible (булеве значення, що визначає видимість вікна) та onClose (функція для закриття вікна).

Хук useRef використовується для створення посилання на поле введення (inputRef), що дозволяє маніпулювати його фокусом.

Стан searchValue за допомогою хука useState зберігає значення пошукового запиту. Хук useEffect використовується для обробки натискання клавіші Escape: при натисканні цієї клавіші викликається функція onClose для закриття вікна.

Інший useEffect відповідає за встановлення фокусу на поле введення при відкритті вікна (з невеликою затримкою для плавності).

Функція handleOverlayClick обробляє кліки на оверлей вікна, кнопку закриття або елементи всередині неї, викликаючи onClose для закриття вікна. Якщо пропс isVisible має значення false, компонент не рендерить нічого (null). В іншому випадку, JSX відображає контейнер div для вікна пошуку, який містить форму пошуку з полем введення та кнопкою відправки, а також список посилань на категорії.

Форма використовує обробник onChange для оновлення стану searchValue при введенні тексту. Фрагмент (<> ... </>) використовується для групування елементів JSX.

Лістинг 2.11. Код файлу SearchPopup.tsx проєкту Web

```
import { useEffect, useRef, useState } from "react";

interface SearchPopupProps {
  isVisible: boolean;
  onClose: () => void;
}

export default function SearchPopup({ isVisible, onClose }: SearchPopupProps) {
  const inputRef = useRef<HTMLInputElement | null>(null);
  const [searchValue, setSearchValue] = useState<string>("");

  useEffect(() => {
```

```
    const handleKeyUp = (e: KeyboardEvent) => {
      if (e.key === "Escape") {
        onClose();
      }
    };
    document.addEventListener("keyup", handleKeyUp);
    return () => document.removeEventListener("keyup", handleKeyUp);
  }, [onClose]);

  useEffect(() => {
    if (isVisible) {
      const timeout = setTimeout(() => {
        inputRef.current?.focus();
      }, 350);
      return () => clearTimeout(timeout);
    }
  }, [isVisible]);


      }
```

Джерело: [створено автором]

Повний лістинг коду представлено в додатках.

### 2.2.4. Шляхи переходу між сторінками

Усі переходи між сторінками здійснюється за допомогою routes(шляхи), файли для них знаходяться в папці “routes”

Приклад для переходів на сторінки Home, Login, About

Лістинг 2.12. Код файлу AppRouter.tsx проєкту Web

```
import { BrowserRouter, Route, Routes } from "react-router-dom";
import Layout from "../layout/Layout";
import Home from "../Pages/Home";
import Login from "../Pages/Login";

export default function AppRouter() {
    return (
        <BrowserRouter>
            <Routes>
                <Route path="/" element={<Layout />}>
                    <Route index element={<Home />} />
                    <Route path="login" element={<Login />} />
                    <Route path="about" element={<h1>About</h1>} />
                    <Route path="contact" element={<h1>Contact</h1>} />
                </Route>
            </Routes>
        </BrowserRouter>
    );
      }
```

Джерело: [створено автором]

### 2.2.5. Схема графічного інтерфейсу веб-сервісу

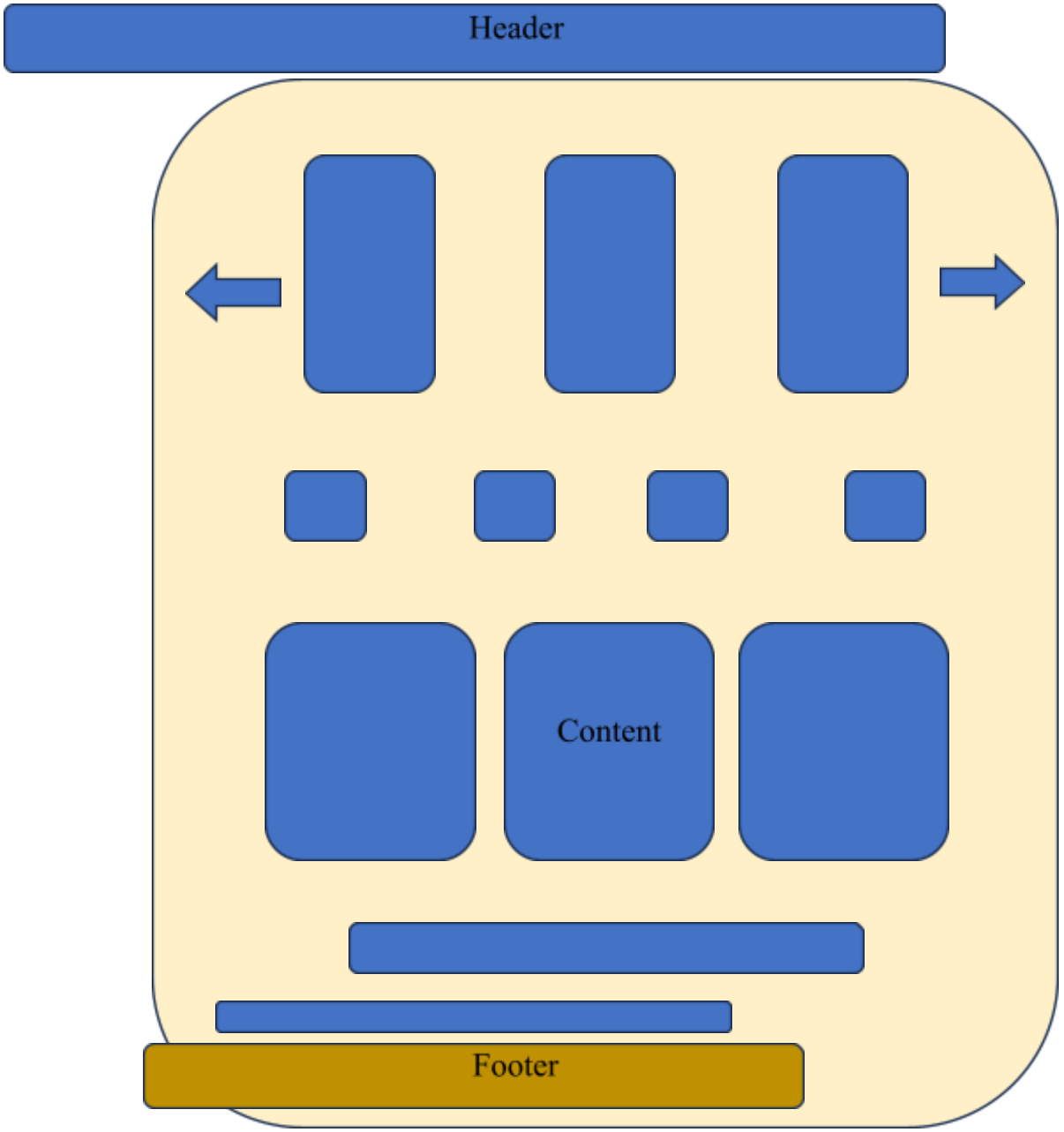


Рис. 2.9. Схема інтерфейсу

Представлена структура відображає класичний підхід до побудови веб-сторінки, орієнтованої на користувача, що дозволяє легко орієнтуватися та взаємодіяти з контентом, і призначена для онлайн-продажів. Уся сторінка логічно розділена на три основні зони: шапку, основний контент та підвал. Шапка сайту, розташована у верхній частині, є ключовим елементом ідентифікації та навігації. Вона типово містить логотип, що слугує для

впізнаваності бренду та швидкого повернення на головну сторінку, а також горизонтальне навігаційне меню, яке надає доступ до основних розділів сайту, таких як "Головна", "Каталог", "Про нас" або "Контакти". Для зручності користувачів у шапці також передбачено поле пошуку, часто позначене іконкою лупи, що дозволяє швидко знайти необхідні товари чи інформацію. Неодмінними елементами електронної комерції у шапці є іконка або кнопка кошика, що відображає кількість обраних товарів та веде до оформлення замовлення, а також функціонал для входу в особистий кабінет або реєстрації. У разі міжнародного охоплення, шапка може включати селектор мови чи регіону.

Найбільшу та найважливішу частину сторінки займає основний контент, який динамічно змінюється залежно від відвідуваної сторінки. Наприклад, на головній сторінці або сторінці категорії, верхню частину основного контенту може займати великий банер або слайдер, який привертає увагу до акційних пропозицій, новинок або ключових меседжів, часто з кнопками дії типу "Дізнатися більше". Далі розташовуються блоки або розділи, що відображають категорії товарів, кожна з яких представлена зображенням та назвою, що веде до детальнішого переліку товарів. Обов'язковими для комерційного сайту є секції з рекомендованими товарами, хітами продажу або новинками, де кожен товар відображається у вигляді картки, яка містить зображення, назву, ціну, можливий рейтинг у зірочках та кнопку "Додати до кошика". Інформація про акції та спеціальні пропозиції також розміщується у виділених блоках, а для підвищення довіри може бути присутня секція з відгуками клієнтів. Додатково, щоб підкреслити цінність, сайт може містити блоки з описом переваг, таких як "Безкоштовна доставка" або "Гарантія якості", а також посилання на інформаційні статті або розділ блогу.

Нарешті, підвал сайту є завершальним блоком сторінки, що містить додаткову, але важливу інформацію. Тут зазвичай розташовані навігаційні посилання на сторінки з умовами використання, політикою конфіденційності, контактними даними або картою сайту. Обов'язковими елементами є іконки з

посиланнями на соціальні мережі бренду. Для маркетингових цілей у підвалі може бути інтегрована форма для підписки на розсилку новин та акцій. Інформація про авторські права, як правило, також знаходиться тут, разом з логотипами підтримуваних платіжних систем, що підвищує довіру до сайту. Вся ця структура передбачає адаптивність для коректного відображення на всіх типах пристроїв та інтерактивність усіх елементів для забезпечення позитивного користувацького досвіду, при цьому оптимізація для швидкого завантаження залишається пріоритетом.

### 2.3. Вибір інструментів та засобів розробки

При розробці інтерактивного веб-ресурсу для забезпечення онлайн-продажів нами було обрано надійний та сучасний стек технологій, який забезпечує високу продуктивність, масштабованість та зручність розробки.

Основним інструментом розробки виступає мова програмування C#, вибір якої обумовлений її потужністю, універсальністю та широкою підтримкою в екосистемі Microsoft. C# є об'єктно-орієнтованою мовою, що сприяє створенню модульного та легкопідтримуваного коду. Веб-ресурс буде розроблений на базі платформи .NET, яка надає багатий набір бібліотек, фреймворків та інструментів для розробки різноманітних додатків, включаючи веб-додатки. Зокрема, для створення веб-API та/або веб-інтерфейсу ймовірно буде використано ASP.NET Core.

Для взаємодії з базою даних обрали Entity Framework Core (ORM). Цей Object-Relational Mapper дозволяє працювати з даними бази даних за допомогою об'єктів C#, що значно спрощує операції CRUD (Create, Read, Update, Delete) та зменшує кількість шаблонного коду. Використання Entity Framework Core також забезпечує незалежність від конкретної системи управління базами даних, дозволяючи легко переключатися між ними за необхідності. Щодо системи управління базами даних (СУБД), її вибір

залежатиме від вимог до проекту, таких як обсяг даних, очікуване навантаження та необхідність масштабування. Популярними варіантами, які добре інтегруються з Entity Framework Core, є SQL Server, PostgreSQL, MySQL або SQLite (для локальної розробки та тестування). Для реалізації цього проекту буде обрано відповідну реляційну базу даних.

Як середовище розробки (IDE) використали Visual Studio. Це повноцінне інтегроване середовище розробки від Microsoft, яке надає широкий спектр інструментів для написання, налагодження та розгортання .NET-додатків.

Крім того, для забезпечення повноцінного та інтерактивного користувацького інтерфейсу, фронтенд розробка базуватиметься на стандартних веб-технологіях, таких як HTML5, CSS3 та JavaScript. Для створення динамічних та односторінкових додатків (SPA) можуть бути задіяні сучасні фреймворки/бібліотеки для фронтенду, наприклад, React, Angular або Vue.js.

Для ефективного управління версіями коду та забезпечення можливості спільної роботи, була використана система контролю версій Git у поєднанні з платформами, такими як GitHub, GitLab або Azure DevOps. Якість та надійність програмного забезпечення забезпечуватиметься через тестування, для якого можуть бути застосовані такі фреймворки, як xUnit, NUnit та бібліотека Moq для юніт-тестування.

**Висновки до розділу 2**

У другому розділі кваліфікаційної роботи було детально розглянуто практичні аспекти розробки веб-додатку "VaviBrand", зокрема його серверної частини. Проведене дослідження дозволило сформулювати низку ключових висновків щодо архітектурних та технологічних рішень проекту.

Створення проекту в середовищі Visual Studio 2022 з використанням сучасної модульної архітектури довело свою ефективність. Чітке розділення на чотири основні шари (Application, Domain, Infrastructure та Web) забезпечило зручність розробки та підтримки коду. Використання принципів Clean

Architecture та Domain-Driven Design дозволило створити гнучку та масштабовану систему, здатну до подальшого розширення.

Реалізація бази даних з використанням підходу Code First на базі Entity Framework Core 6.0 продемонструвала свою ефективність. Створена модель даних, що включає такі ключові сутності як Product, Category, User та Order, повністю відображає бізнес-логіку системи. Використання міграцій забезпечило контроль версій схеми бази даних та спростило процес її оновлення.

Технологічні рішення, прийняті під час розробки, забезпечують:

- Високу продуктивність системи навіть при значному навантаженні
- Легкість підтримки та розширення функціоналу
- Безпеку даних та транзакцій
- Зручність тестування окремих компонентів
- Ефективну співпрацю команди розробників

Отримані результати свідчать про те, що обраний підхід до розробки серверної частини додатку є оптимальним для вирішення поставлених завдань. Реалізована архітектура дозволяє легко адаптувати систему до змінних вимог ринку та забезпечує стабільну роботу всіх компонентів. Це створює міцну основу для подальшого вдосконалення додатку та розширення його функціональних можливостей.

Проведена робота в другому розділі підтвердила ефективність обраних технологій та методологій розробки, що дозволяє переходити до наступних етапів проекту, зокрема тестування системи та впровадження додаткових функцій.

# РОЗДІЛ 3
# РОЗРОБКА ОНЛАЙН СЕРВІСУ

## 3.1. Розробка UML-діаграми

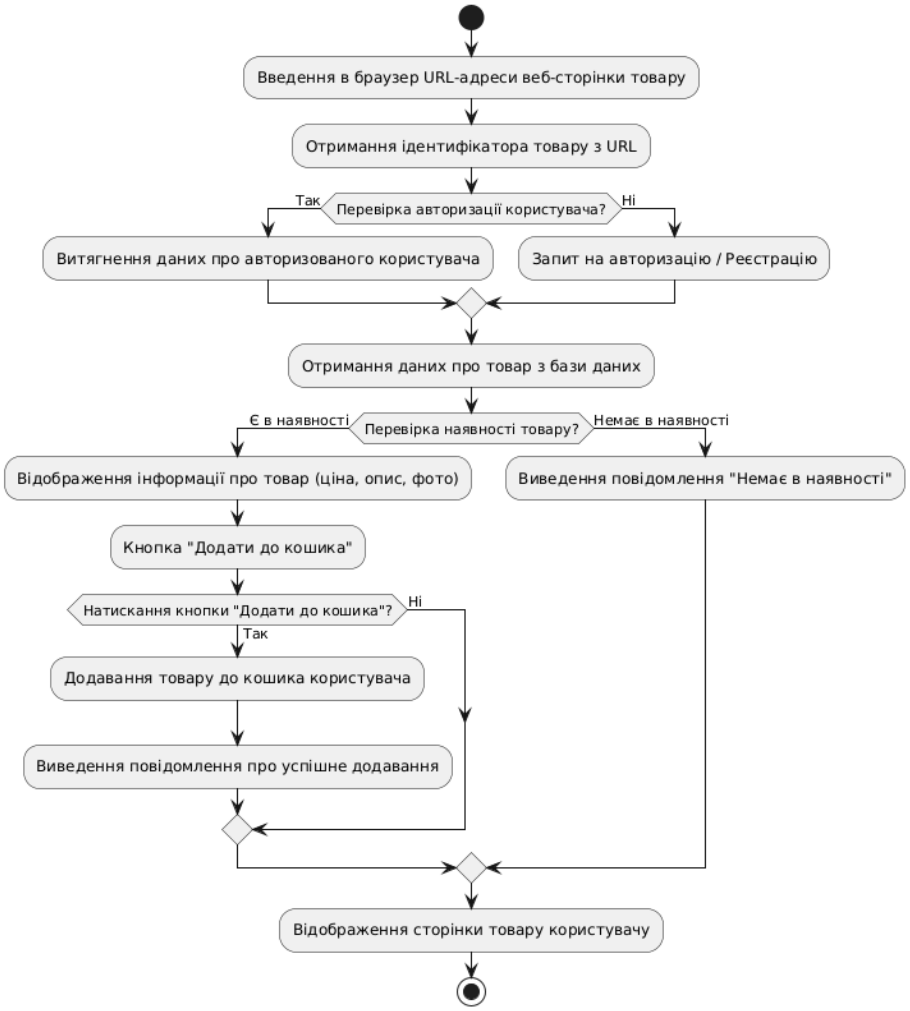


Рис. 3.1. UML діаграма розроблюваного червісу

На етапі розробки онлайн-сервісу для забезпечення онлайн-продажів надзвичайно важливим є детальне візуальне представлення архітектури системи та її функціональних можливостей. Для цього використали блок-схеми та/або діаграми уніфікованої мови моделювання (UML), які дозволили ефективно

описати взаємодію компонентів, логіку роботи та поведінку системи з різних точок зору.

Файл Program.cs відіграє ключову роль у запуску та налаштуванні веб-серверу, визначаючи основні параметри функціонування додатку. У ньому зосереджено логіку ініціалізації сервісів, таких як система автентифікації та авторизації, а також конфігурація з'єднання з базою даних. Наприклад, саме тут прописується підключення до SQL Server або іншої СУБД, а також налаштовується служба IdentityCore, що відповідає за управління обліковими записами користувачів.

Лістинг 3.1. Код файлу Program.cs проєкту Web

```
using Application;
using Infrastructure;
using Infrastructure.Data;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddInfrastructureServices(builder.Configuration);

builder.Services.AddApplicationServices();

// Add services to the container.

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "VaviBrand API",
        Version = "v1",
        Description = "API for VAVI_BRAND web application"
    });
    options.CustomSchemaIds(type => type.FullName);
});

builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAllOrigins",
        policy => policy.WithOrigins("http://localhost:3000")
                .AllowAnyHeader()
                .AllowAnyMethod());
});
```

Джерело: [створено автором]

Повний лістинг поданий в додатках.

### 3.1.1. База даних та моделі даних

Цей підрозділ описує структуру бази даних та моделі даних, які використовуються в серверній частині веб-додатку "VaviBrand".

Для зберігання даних було обрано SQL Server. Це потужна реляційна система керування базами даних (СУБД), яка забезпечує надійність, масштабованість та продуктивність, необхідні для сучасного веб-додатку. SQL Server добре інтегрується з ASP.NET Core та Entity Framework Core, що спрощує розробку та підтримку.

Entity Framework Core (EF Core) використовується як ORM (Object-Relational Mapper). EF Core дозволяє розробникам працювати з базою даних, використовуючи об'єкти C#, замість написання SQL-запитів вручну. Це підвищує продуктивність розробки, зменшує кількість коду та забезпечує більшу безпеку.

Моделі даних представляють сутності предметної області інтернет-магазину "VaviBrand" є Category який описує категорію товару (наприклад, "Одяг", "Взуття", "Аксесуари"), далі клас Order, він представляє замовлення, зроблене покупцем. Клас OrderItem описує конкретний товар у замовленні. Сутність Payment – це інформація про платіж. Клас Product описує товар, який продається в магазині. ShippingDetail містить інформацію про доставку замовлення.

Кожна з цих моделей представлена класом C# у проекті Domain\Entities. Ці класи відображаються на таблиці в базі даних за допомогою EF Core.

Представлено основні зв’язки:

- Категорія може мати багато товарів (One-to-Many).
- Замовлення може містити багато позицій замовлення (One-to-Many).
- Позиція замовлення відноситься до одного замовлення та одного товару (Many-to-Many через проміжну сутність).
- Товар може мати багато позицій замовлення (One-to-Many).
- Замовлення може мати одну інформацію про доставку (One-to-One або One-to-Zero-or-One).
- Замовлення може мати один платіж (One-to-One або One-to-Zero-or-One).

Ці зв'язки реалізовані за допомогою навігаційних властивостей та зовнішніх ключів у EF Core.

Для взаємодії з базою даних у EF Core використовується контекст бази даних (Database Context). Він представляє сесію з базою даних і дозволяє виконувати операції CRUD (Create, Read, Update, Delete). Контекст бази даних був створений у проекті Infrastructure\Data.

**Entity Framework Core Migrations** використовуються для створення та оновлення схеми бази даних. Міграції дозволяють автоматично генерувати SQL-скрипти для внесення змін у структуру бази даних, що спрощує процес розробки та розгортання.

Репозиторії інкапсулюють логіку доступу до даних і надають інтерфейс для отримання, додавання, оновлення та видалення даних з бази даних. Вони використовують контекст бази даних EF Core для виконання цих операцій.

Важливий метод є **OnModelCreating()** в якому присутній метод Seed() для наповлення бази даних та задано усі зв’язки між таблицями.

Лістинг 3.2. Код файлу DataContext.cs

```
using Domain.Entities;
using Infrastructure.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Data
{
    public class DataContext : IdentityDbContext<ApplicationUser>
    {
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<Payment> Payments { get; set; }
        public DbSet<ShippingDetail> ShippingDetails { get; set; }

        public DataContext(DbContextOptions<DataContext> options) : base(options)
        {
        }
    }
}
```

Джерело: [створено автором]

Повний лістинг подано в додатках.

Також, клас OrderItem показує реалізацію зв’язку багато-до-багато з класом Order

Лістинг 3.3. Код файлу OrderItem.cs

```
namespace Domain.Entities
{
    public class OrderItem
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

        public Guid OrderId { get; set; }
        public Order Order { get; set; }

        public int ProductId { get; set; }
        public Product Product { get; set; }

        [Required]
        public int Quantity { get; set; }
    }
}
```

Джерело: [створено автором]

Лістинг 3.4. Код файлу Order.cs

```
namespace Domain.Entities
{
    public class Order
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

        [Required]
        public string UserId { get; set; }

        public DateTime OrderDate { get; set; } = DateTime.UtcNow;

        [Required, MaxLength(20)]
        public string Status { get; set; } = "Обробка";

        public ICollection<OrderItem> OrderItems { get; set; }

        public Payment Payment { get; set; }
        public ShippingDetail ShippingDetail { get; set; }
    }
      }
```

Джерело: [створено автором]

Далі для заповнення бази даних використовуємо статичний метод Seed(), який викликається в DataContext(), в методі OnModelCreating(), який розглядали вище.

Лістинг 3.5. Код файлу DataSeeder.cs

```
using Domain.Entities;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Data
{
    public static class DataSeeder
    {
        public static async Task SeedDataAsync(DataContext context)
        {
            if ( (await context.Database.GetPendingMigrationsAsync()).Any() )
            {
                await context.Database.MigrateAsync();
            }

            SeedCategories(context);
        }

    }
      }
```

Джерело: [створено автором]

### 3.2. Реалізація серверної частини

Спершу відбувається налаштування середовища розробки. На цьому етапі встановлюють та конфігурують необхідне програмне забезпечення: Visual Studio як основне IDE, .NET SDK, а також інструменти для роботи з обраною СУБД (SQL Server Management). Також налаштовували систему контролю версій (Git) для спільної роботи та управління змінами в коді.

Далі, на основі UML діаграм та вимог, формувалася архітектура та структура проекту. Для .NET-додатків це часто передбачає створення багатошарової архітектури (наприклад, Presentation Layer, Business Logic Layer, Data Access Layer), що забезпечує чітке розділення відповідальності та легкість у подальшому масштабуванні та модифікації. На цьому етапі визначалися основні папки, файли та конвенції іменування.

Розробка бекенду є центральним етапом для реалізації бізнес-логіки. За допомогою ASP.NET Core розроблялися RESTful API endpoints для взаємодії з фронтендом та іншими сервісами. Реалізували контролери, які обробляють запити, валідують дані та викликають відповідні сервіси бізнес-логіки. Писався код на C#, що відповідає за основні операції системи: управління користувачами (реєстрація, авторизація, профілі), управління товарами (додавання, редагування, пошук, фільтрація), логіка кошика, формування та обробка замовлень, інтеграція з платіжними системами та сервісами доставки. За допомогою Entity Framework Core створили модель даних (сутності), налаштували міграції для управління схемою бази даних. Розробили репозиторії та сервіси для виконання операцій CRUD над даними, забезпечили ефективне та безпечне зберігання та вибірку інформації.

Далі створили HTML-шаблони для сторінок, використовуючи CSS3 для їх стилізації та забезпечення адаптивного дизайну, щоб веб-ресурс коректно відображався на різних пристроях. За допомогою JavaScript та обраного фреймворку React, Angular або Vue.js розробили динамічний інтерфейс. Це включало обробку подій користувача, асинхронні запити до бекенд API для отримання та відправлення даних, маніпуляції з DOM та відображення даних. Впровадили валідація вхідних даних безпосередньо в браузері для покращення користувацького досвіду та зменшення навантаження на сервер.

Тестування всього розробленого коду є критичним етапом для виявлення та виправлення помилок. Воно охопило у нас модульне тестування (з використанням фреймворків, таких як xUnit, NUnit, для перевірки окремих компонентів), інтеграційне тестування (перевірка взаємодії між модулями), функціональне тестування (відповідність реалізованої функціональності вимогам), а також тестування продуктивності та безпеки (оцінка швидкості роботи під навантаженням та стійкості до загроз).

### 3.3. Реалізація фронтенд частини

Реалізація онлайн-сервісу продажів передбачила послідовне впровадження всіх визначених функціональних можливостей, використовуючи обрані технології та дотримуючись розробленої архітектури. Це комплексний процес, що включив створення бази даних, розробку бекенду та фронтенду, а також їх інтеграцію і тестування.

Для реалізації цього проєкту використовували C# як основну мову програмування. На бекенді застосували ASP.NET Core для розробки API та Entity Framework Core для ефективної взаємодії з базою даних. Фронтенд реалізований за допомогою одного з популярних JavaScript фреймворків, таких як React, Angular або Vue.js. В якості системи управління базами даних ми обрали SQL Server. Управління змінами в коді відбувалося за допомогою Git.

Спершу створили базу даних та її модель. На основі розробленої діаграми класів сформували таблиці в обраній СУБД. Далі, за допомогою Entity Framework Core, розробили C# класи, які відображають структуру цих таблиць. Налаштували міграції, що дозволили автоматично керувати змінами в схемі бази даних. Для виконання операцій над даними (створення, читання, оновлення, видалення) реалізували репозиторії або DAO (Data Access Objects), що забезпечили абстракцію від безпосередньої взаємодії з базою даних.

Потім відбувалася розробка Backend API. За допомогою ASP.NET Core створили RESTful API контролери. Кожен контролер відповідає за певний набір функціональності, наприклад, управління користувачами, товарами чи замовленнями. Далі реалізуються endpoints для всіх ключових варіантів використання, включаючи реєстрацію та авторизацію користувачів, управління товарними позиціями, додавання товарів до кошика, оформлення замовлень та обробку платежів. Бізнес-логіка зроблена в окремих сервісних шарах, які викликаються контролерами. Це охопило валідацію даних, розрахунок вартості замовлень та оновлення статусу товарів. Для захисту API та забезпечення доступу лише авторизованим користувачам налаштували авторизацію та аутентифікацію за допомогою JWT-токенів.

Паралельно з бекендом велася розробка фронтенду, тобто клієнтського інтерфейсу. Він створювався з використанням обраного JavaScript фреймворку. Розроблялися UI компоненти для всіх основних елементів системи: каталогів товарів, карток продуктів, кошика, форм реєстрації та авторизації, сторінок оформлення замовлень та профілю користувача. Для візуального оформлення компонентів та забезпечення адаптивного дизайну, що гарантує коректне відображення на різних пристроях, використовували CSS3. За допомогою асинхронних запитів (AJAX, Fetch API або Axios) реалізували взаємодія з розробленим Backend API для отримання та надсилання даних. Також на стороні клієнта впровадили валідацію форм, що покращило користувацький досвід та зменшило навантаження на сервер.

Протягом всього процесу розробки дотримувалися усіх ключових принципів: модульності (код розбивається на логічні, незалежні частини), чистого коду (дотримання стандартів кодування, використання зрозумілих імен та коментарів), DRY (Don't Repeat Yourself) для уникнення дублювання коду, а також SOLID принципів для створення гнучкої та розширюваної об'єктно-орієнтованої системи.

Спочатку коли користувач заходить на сайт, він одразу бачить «шапку» сайту, у ній розташовані у правій стороні кнопка профілю, товари які були додані до «Вподобані» та кнопка корзини, у ній знаходяться товари які користувач додав.

Також у центральній частині присутні кнопки «ГОЛОВНА», розгортаючий список «КАТАЛОГ», тут можна обрати користувачу, яку категорію товару він хоче переглянути та ще присутній елемент «ПРО НАС»

Демонстрація першої частини сторінки «Головна»:

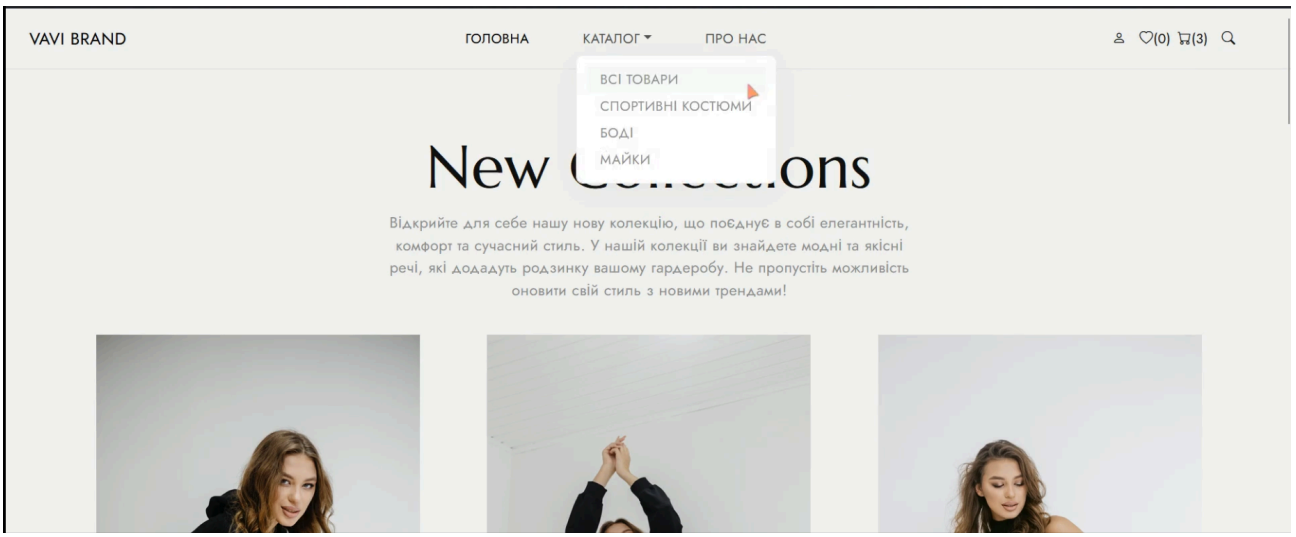


Рис. 3.2. Верхня частина сторінки сайту

Джерело:[створено автором]

Наступна частина, це «карусель», у якій розташовані самі найпоширеніші товари у асортименті

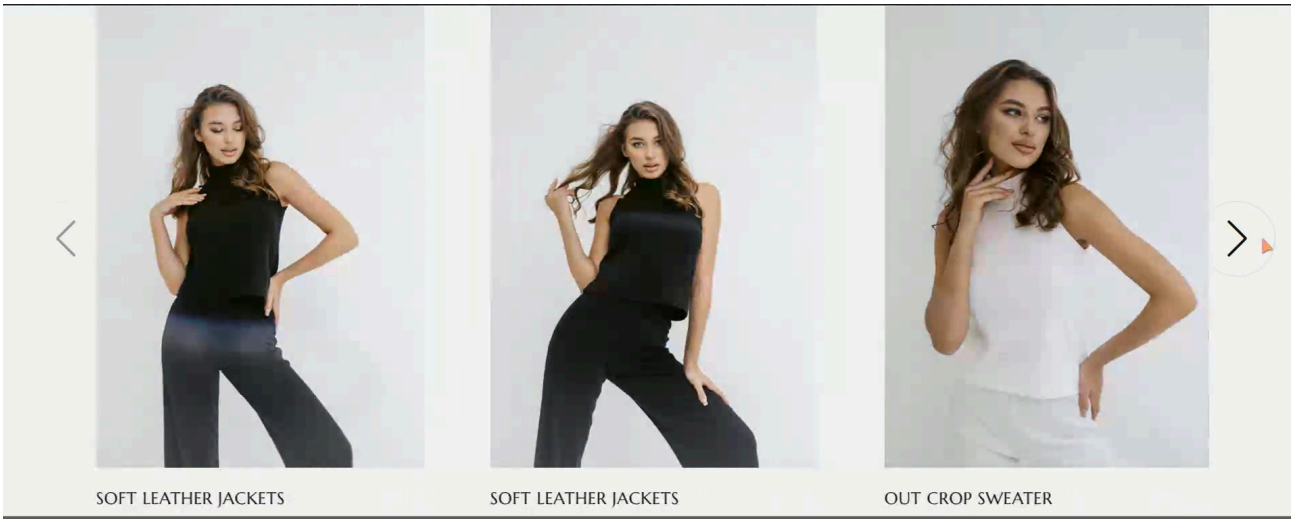


Рис. 3.3. Карусель сторінки «Головна»

Джерело:[створено автором]

Наступна секція відповідає за інформацію про доставку та умови повернення, у ній користувач може детально ознайомитися також з різними видами пакування та умовами як зробити замовлення

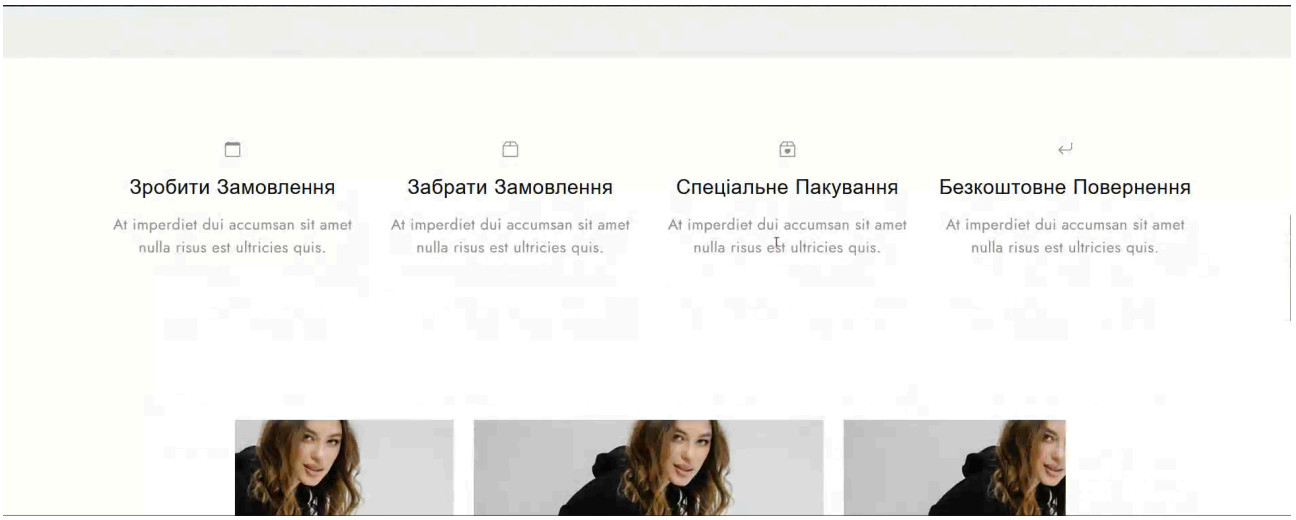


Рис. 3.4. Секція з інформацією про доставку сторінки «Головна»

Джерело:[створено автором]

Ще один елемент, це невелика демонстраційна галерея, у якій можна переглянути фото асортименту товарів

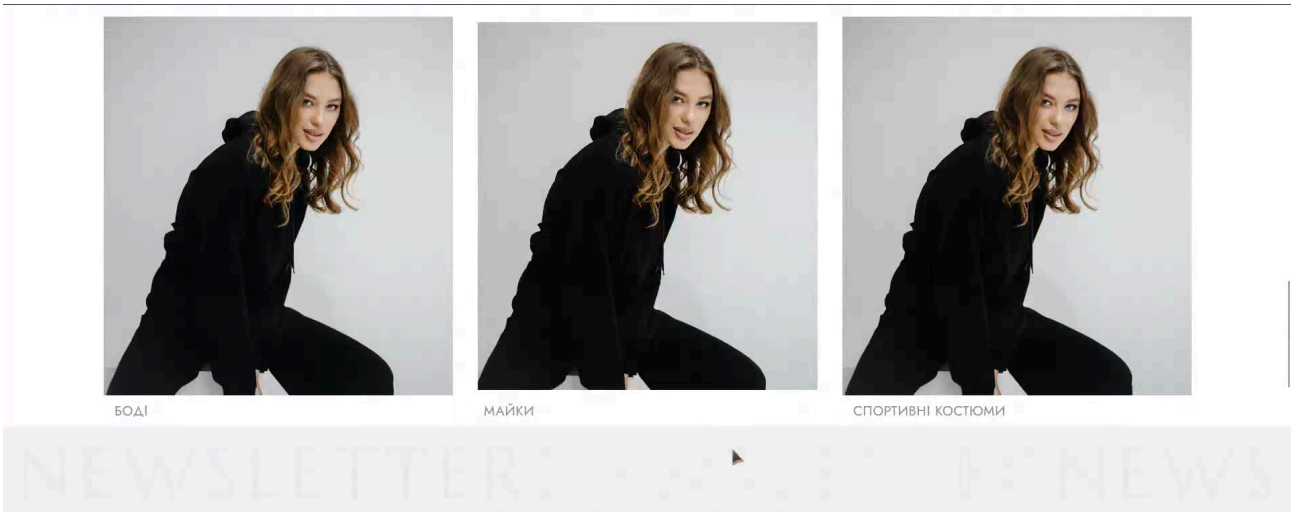


Рис. 3.5. Секція з галереєю сторінки «Головна»

Джерело:[створено автором]

Ще присутній елемент «ОТРИМУВАТИ НОВИНИ ПЕРШИМИ», тобто користувач залишає свій Email адрес і коли з’являються оновлення на сайті або знижки, йому приходять сповіщення на почту і таким чином користувач буде завжди вкурсі подій.

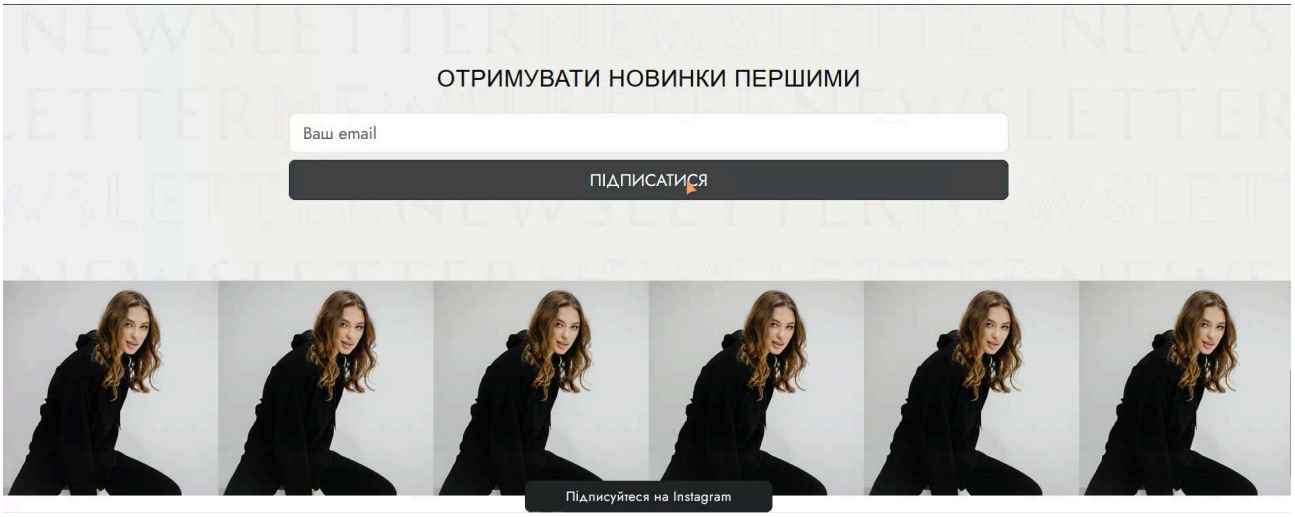


Рис. 3.6. Секція з отриманням новин про нові надходження товарів сторінки «Головна»

Джерело:[створено автором]

На завершення, перед футером ще є секція про наші соціальні мережі, щоб користувач міг легко з нами зв’язатися та задати різні питання

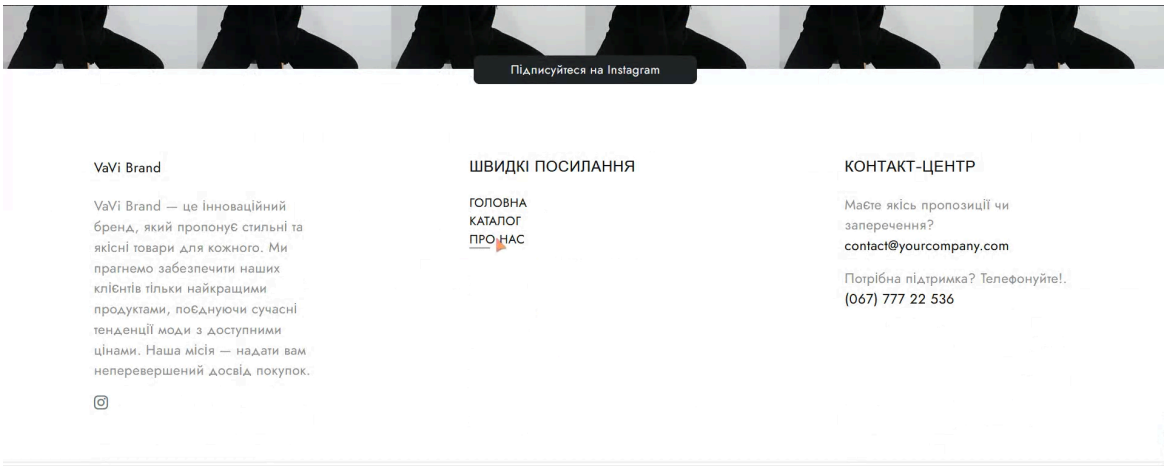


Рис. 3.7. Секція з отриманням новин про нові надходження товарів сторінки «Головна»

Джерело:[створено автором]

Футер складається з невиличкого опису нашого бренду, присутні зручні швидкі посилання на сторінки («ГОЛОВНА», «КАТАЛОГ», «ПРО НАС»), також є ще контактні дані як з нами зв’язатися

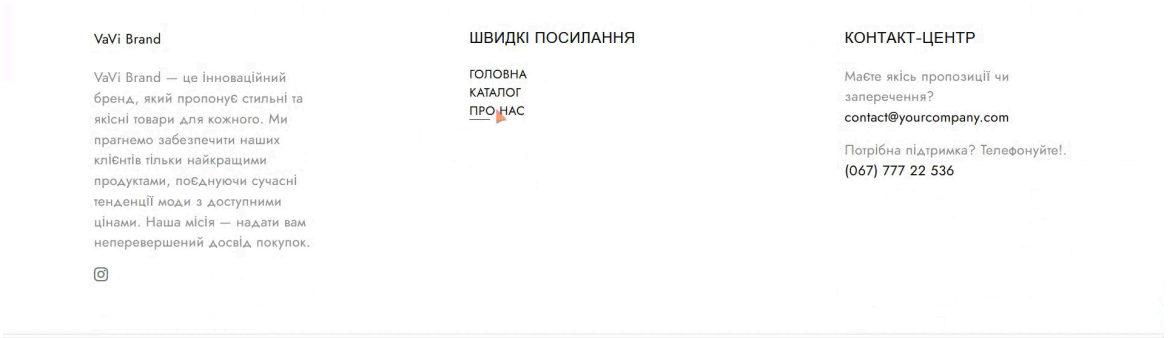


Рис. 3.8. Демонстрація підвалу сайту

Джерело:[створено автором]

**Висновки до розділу 3**

У Розділі 3 було детально розглянуто процес розробки онлайн-сервісу "VaviBrand" для забезпечення онлайн-продажів, охопивши як архітектурні

аспекти, так і практичну реалізацію серверної та клієнтської частин. На початковому етапі особливу увагу було приділено візуальному представленню системи за допомогою UML-діаграм, що дозволило ефективно описати її функціональні можливості та взаємодію компонентів.

Ядром розробки стало налаштування веб-сервера через файл Program.cs, який відповідає за ініціалізацію ключових сервісів, включаючи систему автентифікації, авторизації та конфігурацію з'єднання з базою даних. Для зберігання та управління даними було обрано SQL Server, потужну реляційну СУБД, у поєднанні з Entity Framework Core як ORM. Це рішення забезпечило надійність, масштабованість та спростило роботу з базою даних за допомогою об'єктів C#. Було детально описано моделі даних, такі як Category, Product, Order, OrderItem, Payment та ShippingDetail, а також представлено їхні основні зв'язки (One-to-Many, Many-to-Many), реалізовані через навігаційні властивості та зовнішні ключі. Контекст бази даних (DataContext.cs) та механізм міграцій EF Core були використані для ефективного керування схемою БД та операціями CRUD, а також реалізовано початкове наповнення бази даних за допомогою методу Seed().

Реалізація серверної частини базувалася на ASP.NET Core для розробки RESTful API. Було створено контролери та бізнес-логіку для управління користувачами, товарами, кошиком та замовленнями, а також інтегровано платіжні системи та сервіси доставки. Забезпечення безпеки API було реалізовано за допомогою JWT-токенів для автентифікації та авторизації. Паралельно велася розробка фронтенд частини з використанням одного з популярних JavaScript фреймворків (React/Angular/Vue.js). Були розроблені UI компоненти, забезпечено адаптивний дизайн за допомогою CSS3 та реалізовано взаємодію з бекенд API через асинхронні запити.

Весь процес розробки дотримувався принципів модульності, чистого коду, DRY та SOLID, що сприяло створенню гнучкої, розширюваної та підтримуваної

системи. Завершальним етапом було тестування, яке охопило модульне, інтеграційне, функціональне тестування, а також перевірку продуктивності та безпеки, що забезпечило виявлення та виправлення помилок. Детальна демонстрація структури головної сторінки, включаючи шапку, карусель товарів, інформаційні секції (доставка, галерея, підписка на новини, соціальні мережі) та футер, підтверджує орієнтованість розробки на користувацький досвід та зручність навігації.

## ВИСНОВКИ

Кваліфікаційна робота, присвячена розробці веб-додатку "VaviBrand" для онлайн-продажу одягу, успішно досягла своєї основної мети — створення ефективного, масштабованого та зручного інструменту електронної комерції, орієнтованого на підтримку українських виробників та задоволення потреб сучасних споживачів. Проєкт реалізовано як повноцінне програмне рішення, що поєднує технічну досконалість із практичною цінністю для бізнесу.

У процесі розробки особливу увагу було приділено побудові архітектури веб-додатку на основі сучасного технологічного стеку, який включає ASP.NET Core для серверної частини та React.js для клієнтського інтерфейсу. Такий підхід дозволив створити гнучкий та швидкодіючий додаток із чітким розділенням відповідальностей, що значно полегшує розширення функціоналу в майбутньому. Завдяки використанню принципів Clean Architecture та Domain-Driven Design було забезпечено логічну структуру проєкту, спрощено підтримку коду, а також створено передумови для масштабування системи у разі зростання навантаження або бізнес-вимог.

Функціональна частина веб-додатку побудована на основі RESTful API, що дозволяє ефективно обробляти запити від клієнтської частини та забезпечує високу продуктивність і гнучкість взаємодії. Для організації доступу до даних

використано ORM-фреймворк Entity Framework Core у поєднанні з підходом Code First, що надало змогу зручно проектувати базу даних, адаптувати її до змін та гарантувати цілісність інформації. У системі реалізовано зберігання та обробку даних про товари, користувачів, замовлення, платіжні операції та інші бізнес-об'єкти, що є критично важливими для ефективного функціонування онлайн-магазину.

Велике значення було надано інструментам, які підвищують якість та прозорість розробки. Зокрема, впровадження Swagger дозволило автоматично генерувати документацію до API, що значно полегшує роботу як розробникам, так і майбутнім командам супроводу. Використання бібліотеки MediatR забезпечило реалізацію патерну CQRS, розділивши запити та команди, тим самим спростивши логіку взаємодії між шарами програми. Завдяки AutoMapper було спрощено трансформацію даних між різними шарами архітектури, що також сприяло зменшенню дублювання коду та підвищенню його чистоти.

Особливий акцент у проєкті зроблено на підтримці українського ринку та його специфіки. Усі елементи інтерфейсу адаптовано під україномовного користувача, що дозволяє створити комфортне середовище для взаємодії з додатком. До платформи інтегровано популярні в Україні платіжні системи, зокрема ті, що забезпечують швидкі та безпечні транзакції, а також враховано особливості логістики — як доставки, так і обробки повернень.

Окремо слід згадати впровадження інноваційних функцій, які надають "VaviBrand" конкурентних переваг на ринку. Зокрема, функціонал AI-підбору стилю дає змогу користувачам отримувати персоналізовані рекомендації на основі їхніх уподобань, поведінки та аналізу актуальних трендів. А можливість віртуальної примірки одягу забезпечує новий рівень взаємодії, дозволяючи користувачам візуально оцінити, як обрані речі виглядатимуть на них ще до здійснення покупки.

Загалом, реалізація веб-додатку "VaviBrand" підтвердила здатність поєднати академічні знання з реальними вимогами сучасного електронного бізнесу. Успішне завершення проєкту свідчить про готовність автора до вирішення комплексних прикладних задач, демонструє високий рівень володіння сучасними технологіями програмної інженерії та створює потенціал для подальшого вдосконалення системи, її масштабування та комерціалізації. Таким чином, розроблений додаток є не лише технічно ефективним, а й стратегічно значущим для розвитку онлайн-торгівлі, популяризації українського виробництва та забезпечення високоякісного онлайн-шопінгу для споживачів.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. "Аналіз ринку одягу і взуття в Україні 2023 рік" [Електронний ресурс] – URL: https://pro-consulting.ua/ua/issledovanie-rynka/analiz-rynka-odezhdy-i-obuvi-v-ukraine-2023-god (дата звернення: 16.05.2024)

2. Розробка з боку front-end: що це таке і чим відрізняється від back-end? [Електронний ресурс] – URL: https://dan-it.com.ua/uk/blog/rozrobka-z-boku-front-end-shho-ce-take-i-chim-vidriznjaietsja-vid-back-end/ (дата звернення: 16.05.2024)

3. Що таке backend розробка? [Електронний ресурс] – URL: https://mc.today/uk/shho-take-backend-rozrobka/ (дата звернення: 16.05.2024)

4. Wikipedia. Microsoft SQL Server. [Електронний ресурс] – URL: https://uk.wikipedia.org/wiki/Microsoft_SQL_Server (дата звернення: 16.05.2024)

5. Wikipedia. Дизайн користувацького досвіду. [Електронний ресурс] – URL: https://uk.wikipedia.org/wiki/%D0%94%D0%B8%D0%B7%D0%B0%D0%B9%D0%BD_%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%B2%D1%87%D0%B0%D1%86%D1%8C%D0%BA%D0%BE%D0%B3%D0%BE_

%D0%B4%D0%BE%D1%81%D0%B2%D1%96%D0%B4%D1%83 (дата звернення: 16.05.2024)

6. Що таке REST API? [Електронний ресурс] – URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення: 16.05.2024)

7. Wikipedia. ASP.NET Core. [Електронний ресурс] – URL: https://en.wikipedia.org/wiki/ASP.NET_Core (дата звернення: 16.05.2024)

8. EVA.ua. [Електронний ресурс] – URL: https://eva.ua/ua/ (дата звернення: 16.05.2024)

9. Answear.ua. [Електронний ресурс] – URL: https://answear.ua/?utm_source=google&utm_medium=cpc&utm_campaign=_search_brand_Answear%20%5BMaxClicks%5D&utm_id=21646667206&gad_source=1&gad_campaignid=21646667206&gbraid=0AAAAAB8Djs30nvENNG2AKOJhMa2n6ARny&gclid=Cj0KCQjwoZbBBhDCARIsAOqMEZVFNRb3qS0HxsnlXbAionbKbi8Wo-1Mfjl0u87Di8g1YOpWdVhDePkaAmXKEALw_wcB (дата звернення: 16.05.2024)

10. Kasta.ua. [Електронний ресурс] – URL: https://kasta.ua/?utm_campaign=181229_search_brand_cpc_main_branding_modnakasta_city&utm_medium=cpc&utm_source=google&utm_content=156417424&utm_term=modna%20kasta&gad_source=1&gad_campaignid=156417424&gbraid=0AAAADo7-9BpOHqtL9tGHJ1ixB-wX8Owq&gclid=Cj0KCQjwoZbBBhDCARIsAOqMEZXM336lGIXqL03qE8o6Ccbp9SJe6-Ng2Lc0-Eecq-8NfYLz7bIW9zYaAhG6EALw_wcB (дата звернення: 16.05.2024)

11. CRUD. [Електронний ресурс] – URL: https://goit.global/javascript/ru/v1/module-13/crud.html (дата звернення: 16.05.2024)

12. Swagger. [Електронний ресурс] – URL: https://swagger.io/ (дата звернення: 16.05.2024)

13. Wikipedia. Microsoft SQL Server. [Електронний ресурс] – URL: https://uk.wikipedia.org/wiki/Microsoft_SQL_Server (дата звернення: 16.05.2024)

14. AutoMapper. [Електронний ресурс] – URL: https://automapper.org/ (дата звернення: 16.05.2024)

15. Wikipedia. Аутентифікація. [Електронний ресурс] – URL: https://uk.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F (дата звернення: 16.05.2024)

16. Wikipedia. JSON Web Token. [Електронний ресурс] – URL: https://en.wikipedia.org/wiki/JSON_Web_Token (дата звернення: 16.05.2024)

17. Mozilla Developer Network (MDN Web Docs). JavaScript. [Електронний ресурс] - URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення: 17.05.2025)

18. Mozilla Developer Network (MDN Web Docs). HTML. [Електронний ресурс] - URL: https://developer.mozilla.org/en-US/docs/Web/HTML (дата звернення: 17.05.2025)

19. Mozilla Developer Network (MDN Web Docs). CSS. [Електронний ресурс] - URL: https://developer.mozilla.org/en-US/docs/Web/CSS (дата звернення: 17.05.2025)

20. W3Schools. Web Tutorials. [Електронний ресурс] - URL: https://www.w3schools.com/ (дата звернення: 17.05.2025)

21. Microsoft Learn. ASP.NET Core Documentation. [Електронний ресурс] - URL: https://learn.microsoft.com/en-us/aspnet/core/ (дата звернення: 17.05.2025)

22. Microsoft Learn. Entity Framework Core. [Електронний ресурс] - URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 17.05.2025)

23. Angular. Angular Documentation. [Електронний ресурс] - URL: https://angular.io/docs (дата звернення: 17.05.2025)

24. TypeScript. TypeScript Documentation. [Електронний ресурс] - URL: https://www.typescriptlang.org/docs/ (дата звернення: 17.05.2025)

25. Microsoft Learn. SQL Server Documentation. [Електронний ресурс] - URL: https://learn.microsoft.com/en-us/sql/ (дата звернення: 17.05.2025)

# ДОДАТКИ

ДОДАТОК А.

Приклад реалізації репозиторію IGenericRepository

```
using System.Linq.Expressions;

namespace Application.Common.Interfaces
{
    public interface IGenericRepository<T> where T : class
    {
        Task<T> GetByIdAsync(int id);
        Task<IEnumerable<T>> GetAllAsync();
        Task AddAsync(T item);
        void Update(T item);
        Task DeleteAsync(int id);
        Task<T> FindAsync(Expression<Func<T, bool>> predicate);
        Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> predicate);
    }
}
```

ДОДАТОК Б.

Приклад реалізації контролеру CategoryController

```
using Application.Mediator.Categories.Commands;
using Application.Mediator.Categories.Queries;
using MediatR;
using Microsoft.AspNetCore.Mvc;

namespace Web.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class CategoryController : ControllerBase
    {
        private readonly IMediator _mediator;

        public CategoryController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpGet("get-category/{id}")]
        public async Task<IActionResult> GetCategory(int id)
        {
            var category = await _mediator.Send(new GetCategoryQuery { Id = id });
            if ( category == null )
```

```
            {
                return NotFound(new { message = "Категорія не знайдена" });
            }
            return Ok(category);
        }

        [HttpGet("get-all-categories")]
        public async Task<IActionResult> GetAllCategories()
        {
            var categories = await _mediator.Send(new GetAllCategoriesQuery());
            return Ok(categories);
        }

        [HttpPost("create-category")]
        public async Task<IActionResult> CreateCategory([FromBody]
CreateCategoryCommand command)
        {
            var categoryId = await _mediator.Send(command);
            return Ok(categoryId);
        }

        [HttpDelete("delete-category/{id}")]
        public async Task<IActionResult> DeleteCategory(int id)
        {
            var result = await _mediator.Send(new DeleteCategoryCommand { Id = id });
            if ( !result )
            {
                return NotFound(new { message = "Категорія не знайдена" });
            }
            return Ok(new { message = "Категорія видалена успішно" });
        }

        [HttpPut("update-category/{id}")]
        public async Task<IActionResult> UpdateCategory(int id, [FromBody]
UpdateCategoryCommand command)
        {
            if ( id != command.Id )
            {
                return BadRequest(new { message = "Id не співпадає" });
            }
            var updatedCategory = await _mediator.Send(command);
            if ( updatedCategory == null )
            {
                return NotFound(new { message = "Категорія не знайдена" });
            }
            return Ok(updatedCategory);
        }
    }
      }
```

ДОДАТОК В.

Приклад створення міграцій в базу даних

```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace Infrastructure.Migrations
{
    /// <inheritdoc />
    public partial class Init : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "AspNetRoles",
                columns: table => new
                {
                    Id = table.Column<string>(type: "nvarchar(450)", nullable: false),
                        Name = table.Column<string>(type: "nvarchar(256)", maxLength: 256,
nullable: true),
                    NormalizedName = table.Column<string>(type: "nvarchar(256)", maxLength:
256, nullable: true),
                     ConcurrencyStamp = table.Column<string>(type: "nvarchar(max)", nullable:
true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetRoles", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUsers",
                columns: table => new
                {
```

```
                    Id = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    FirstName = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    LastName = table.Column<string>(type: "nvarchar(max)", nullable: true),
                        UserName = table.Column<string>(type: "nvarchar(256)", maxLength: 256,
nullable: true),
                                NormalizedUserName = table.Column<string>(type: "nvarchar(256)",
maxLength: 256, nullable: true),
                            Email = table.Column<string>(type: "nvarchar(256)", maxLength: 256,
nullable: true),
                        NormalizedEmail = table.Column<string>(type: "nvarchar(256)", maxLength:
256, nullable: true),
                    EmailConfirmed = table.Column<bool>(type: "bit", nullable: false),
                    PasswordHash = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    SecurityStamp = table.Column<string>(type: "nvarchar(max)", nullable: true),
                        ConcurrencyStamp = table.Column<string>(type: "nvarchar(max)", nullable:
true),
                    PhoneNumber = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    PhoneNumberConfirmed = table.Column<bool>(type: "bit", nullable: false),
                    TwoFactorEnabled = table.Column<bool>(type: "bit", nullable: false),
                    LockoutEnd = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
true),
                    LockoutEnabled = table.Column<bool>(type: "bit", nullable: false),
                    AccessFailedCount = table.Column<int>(type: "int", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUsers", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "Categories",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable:
false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Categories", x => x.Id);
                });

            migrationBuilder.CreateTable(
```

```
            name: "AspNetRoleClaims",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                RoleId = table.Column<string>(type: "nvarchar(450)", nullable: false),
                ClaimType = table.Column<string>(type: "nvarchar(max)", nullable: true),
                ClaimValue = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_AspNetRoleClaims", x => x.Id);
                table.ForeignKey(
                    name: "FK_AspNetRoleClaims_AspNetRoles_RoleId",
                    column: x => x.RoleId,
                    principalTable: "AspNetRoles",
                    principalColumn: "Id",
                    onDelete: ReferentialAction.Cascade);
            });

        migrationBuilder.CreateTable(
            name: "AspNetUserClaims",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                UserId = table.Column<string>(type: "nvarchar(450)", nullable: false),
                ClaimType = table.Column<string>(type: "nvarchar(max)", nullable: true),
                ClaimValue = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_AspNetUserClaims", x => x.Id);
                table.ForeignKey(
                    name: "FK_AspNetUserClaims_AspNetUsers_UserId",
                    column: x => x.UserId,
                    principalTable: "AspNetUsers",
                    principalColumn: "Id",
                    onDelete: ReferentialAction.Cascade);
            });

        migrationBuilder.CreateTable(
            name: "AspNetUserLogins",
            columns: table => new
```

```
                {
                    LoginProvider = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    ProviderKey = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    ProviderDisplayName = table.Column<string>(type: "nvarchar(max)", nullable:
true),
                    UserId = table.Column<string>(type: "nvarchar(450)", nullable: false)
                },
                constraints: table =>
                {
                        table.PrimaryKey("PK_AspNetUserLogins", x => new { x.LoginProvider,
x.ProviderKey });
                    table.ForeignKey(
                        name: "FK_AspNetUserLogins_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUserRoles",
                columns: table => new
                {
                    UserId = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    RoleId = table.Column<string>(type: "nvarchar(450)", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUserRoles", x => new { x.UserId, x.RoleId });
                    table.ForeignKey(
                        name: "FK_AspNetUserRoles_AspNetRoles_RoleId",
                        column: x => x.RoleId,
                        principalTable: "AspNetRoles",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_AspNetUserRoles_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
```

```
                name: "AspNetUserTokens",
                columns: table => new
                {
                    UserId = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    LoginProvider = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    Name = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    Value = table.Column<string>(type: "nvarchar(max)", nullable: true)
                },
                constraints: table =>
                {
                                table.PrimaryKey("PK_AspNetUserTokens", x => new { x.UserId,
x.LoginProvider, x.Name });
                    table.ForeignKey(
                        name: "FK_AspNetUserTokens_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
                name: "Orders",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    UserId = table.Column<string>(type: "nvarchar(450)", nullable: false),
                    OrderDate = table.Column<DateTime>(type: "datetime2", nullable: false),
                      Status = table.Column<string>(type: "nvarchar(20)", maxLength: 20, nullable:
false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Orders", x => x.Id);
                    table.ForeignKey(
                        name: "FK_Orders_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });
```

ДОДАТОК Г.

Приклад репозиторію GenericRepository

```
using Application.Common.Interfaces;
using Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
using System.Linq.Expressions;

namespace Infrastructure.Repositories
{
    public class GenericRepository<T> : IGenericRepository<T> where T : class
    {
        private readonly DataContext _context;
        private readonly DbSet<T> _dbSet;

        public GenericRepository(DataContext context)
        {
            _context = context;
            _dbSet = context.Set<T>();
        }

        public async Task<T> GetByIdAsync(int id)
        {
            return await _dbSet.FindAsync(id);
        }

        public async Task<IEnumerable<T>> GetAllAsync()
        {
            return await _dbSet.AsNoTracking().ToListAsync();
        }

        public async Task AddAsync(T item)
        {
            await _dbSet.AddAsync(item);
        }

        public void Update(T item)
        {
            _dbSet.Update(item);
        }

        public async Task DeleteAsync(int id)
        {
            var item = await GetByIdAsync(id);
            if ( item != null )
            {
                _dbSet.Remove(item);
            }
        }

        public async Task<T> FindAsync(Expression<Func<T, bool>> predicate)
        {
```

```
            return await _dbSet.AsNoTracking().FirstOrDefaultAsync(predicate);
        }

        public async Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>>
predicate)
        {
            return await _dbSet.AsNoTracking().Where(predicate).ToListAsync();
        }
    }
      }
```