

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ УПРАВЛІННЯ ОНЛАЙН-
МАГАЗИНОМ ІЗ ВИКОРИСТАННЯМ ASP.NET CORE ТА CLEAN
ARCHITECTURE**

**DEVELOPMENT OF A WEB APPLICATION FOR MANAGING AN
ONLINE STORE USING ASP.NET CORE AND CLEAN ARCHITECTURE**

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-43
Гончарук Олександра Вадимівна

Керівник:
Андрущак Ігор Євгенович

Кваліфікаційну роботу
допущено до захисту
«__» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

«__» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Гончарук Олександрі Вадимівні

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: «Розробка вебзастосунку для управління онлайн магазином ASP.NET Core та Clean Architecture»

Керівник роботи: д.т.н., професор, Андрущак І. Є.

затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 474/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: Специфікація функціональності інтернет-магазину, вимоги до безпеки та адаптивності інтерфейсу, документація до фреймворків ASP.NET Core та Entity Framework Core, довідкові матеріали з розробки вебзастосунків на C#, вимоги до структури бази даних SQLite, принципи Clean Architecture, методичні вказівки до виконання кваліфікаційної роботи бакалавра.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): Обґрунтувати актуальність теми та необхідність створення вебзастосунку для управління інтернет-магазином; проаналізувати сучасні проблеми розробки подібних систем та існуючі аналоги; визначити функціональні й нефункціональні вимоги до системи; обрати та обґрунтувати технологічний стек на основі ASP.NET Core, Entity Framework Core та SQLite; спроектувати архітектуру застосунку відповідно до принципів Clean Architecture; спроектувати та реалізувати базу даних; розробити модулі каталогу товарів, авторизації, кошика та оформлення замовлень; реалізувати адміністративну панель; протестувати працездатність і коректність функціонування системи; реалізувати заходи для забезпечення безпеки даних.

5. Перелік графічного матеріалу: 9 рисунків, 11 лістингів, 5 таблиць

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	<i>Андрущак І. Є.</i>		
Специфікація вимог до розробленої системи	<i>Андрущак І. Є.</i>		
Розробка об'єкта проектування	<i>Андрущак І. Є.</i>		
Нормоконтроль	<i>Суринович О. М.</i>		
Гарант ОП	<i>Ліщина Н. М.</i>		
Показник запозичень тексту	<u> </u> %		
Академічна доброчесність	<i>Андрущак І. Є.</i>		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.06.2026 р.	

Здобувач вищої освіти _____

Олександра ГОНЧАРУК

Керівник кваліфікаційної роботи _____

Ігор АНДРУЩАК

АНОТАЦІЯ

Гончарук О. В. Розробка вебзастосунку для управління онлайн-магазином із використанням ASP.NET Core та Clean Architecture. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра містить вступ, три розділи, висновки та пропозиції, список використаних джерел.

У першому розділі проведено аналіз предметної області електронної комерції та існуючих рішень для управління онлайн-магазинами. Розглянуто сучасні підходи до архітектури програмного забезпечення, виявлено переваги та обмеження наявних систем, що обґрунтовує доцільність розробки нового вебзастосунку.

У другому розділі сформульовано функціональні та нефункціональні вимоги до системи, обґрунтовано вибір технологічного стеку на основі ASP.NET Core та принципів Clean Architecture. Спроектовано багаторівневу архітектуру застосунку, визначено структуру бази даних, описано механізми автентифікації, авторизації та безпечного зберігання даних.

У третьому розділі описано практичну реалізацію вебзастосунку: розроблено модулі управління товарами, замовленнями та користувачами, реалізовано адміністративну панель і клієнтську частину. Проведено тестування функціональності системи, перевірено її продуктивність у різних умовах навантаження.

Ключові слова: вебзастосунок, онлайн-магазин, ASP.NET Core, Clean Architecture, електронна комерція, REST API, база даних, автентифікація, адміністративна панель, тестування.

ABSTRACT

Honcharuk O. V. Development of a web application for managing an online store using ASP.NET Core and clean architecture. Manuscript. Qualification work of the bachelor's degree program "Software Engineering", specialty "Software Engineering". Lutsk National Technical University. Lutsk, 2026.

A bachelor's qualification thesis contains an introduction, three chapters, conclusions and suggestions, a list of sources used.

The first section analyzes the subject area of e-commerce and existing solutions for managing online stores. Modern approaches to software architecture are considered, the advantages and limitations of existing systems are identified, which justifies the feasibility of developing a new web application.

The second section formulates functional and non-functional requirements for the system, justifies the choice of a technology stack based on ASP.NET Core and Clean Architecture principles. The multi-tier architecture of the application is designed, the database structure is defined, and the mechanisms of authentication, authorization, and secure data storage are described.

The third section describes the practical implementation of the web application: modules for managing products, orders and users were developed, an administrative panel and a client part were implemented. The system functionality was tested, its performance and fault tolerance were checked under different load conditions..

Keywords: web application, online store, ASP.NET Core, Clean Architecture, e-commerce, REST API, database, authentication, admin panel, testing.

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ.....	9
1.1 Аналіз сучасного стану проблеми.....	9
1.2 Постановка завдання на кваліфікаційну роботу бакалавра.....	12
Висновки до розділу 1	12
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЕНОЇ СИСТЕМИ	14
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення	14
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання.....	17
Висновки до розділу 2.....	22
РОЗДІЛ 3 РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ УПРАВЛІННЯ ОНЛАЙН-МАГАЗИНОМ	24
3.1 Практична реалізація об'єкта проектування.....	24
3.2 Тестування та налагодження інформаційно-комп'ютерної системи.....	35
3.3 Розробка бази даних	38
3.4 Адміністративна панель управління магазином.....	39
3.5 Безпека розроблюваного вебзастосунку	40
Висновки до розділу 3.....	43
ВИСНОВКИ.....	44
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	47

ВСТУП

Актуальність теми. Стрімкий розвиток інтернет торгівлі в Україні та світі формує зростаючий попит на надійні й масштабовані системи управління онлайн-магазинами. За останні роки електронна комерція у сфері одягу та взуття перетворилася на один із найбільш конкурентних ринкових сегментів: споживачі очікують зручного каталогу товарів, швидкого оформлення замовлень і безпечної авторизації. Водночас бізнес потребує гнучкого адміністрування асортименту й замовлень без залучення зовнішніх розробників. Це зумовлює необхідність створення спеціалізованого вебзастосунку, здатного ефективно вирішувати зазначені задачі.

Сучасна розробка корпоративного програмного забезпечення дедалі частіше спирається на принципи Clean Architecture – архітектурного підходу, що забезпечує чіткий поділ відповідальності між шарами застосунку, спрощує тестування та підтримку коду. Платформа ASP.NET Core є одним із провідних фреймворків для побудови вебзастосунків на базі мови C#, поєднуючи високу продуктивність, кросплатформеність та широкі можливості для реалізації RESTful-інтерфейсів. Використання цих технологій у парі дозволяє будувати програмні системи, які легко розширюються та адаптуються до мінливих вимог ринку.

Мета роботи – розробка вебзастосунку для управління онлайн-магазином одягу з використанням ASP.NET Core та принципів Clean Architecture, що забезпечує функціонал перегляду каталогу товарів, авторизації й реєстрації користувачів, а також управління кошиком і замовленнями.

Для досягнення поставленої мети визначено такі завдання:

- проаналізувати сучасний стан ринку електронної комерції в сегменті одягу і дослідити існуючі рішення для управління онлайн-магазинами;
- обґрунтувати вибір технологічного стеку та архітектурних рішень на основі ASP.NET Core і Clean Architecture;

- спроектувати структуру бази даних із використанням SQLite та визначити механізми зберігання й захисту даних;
- реалізувати основні модулі системи: каталог товарів, авторизацію та реєстрацію користувачів, кошик і оформлення замовлень;
- провести тестування розробленого застосунку та оцінити його функціональність та продуктивність.

Об'єктом дослідження є процес проєктування та розробки вебзастосунків для автоматизації управління інтернет-магазином. У межах цього процесу розглядаються архітектурні підходи до організації програмного коду, методи забезпечення безпеки даних користувачів, способи реалізації взаємодії між шарами системи, а також підходи до організації клієнтської частини на основі компонентної моделі Blazor.

Предметом дослідження є методи та засоби розробки вебзастосунків на основі платформи ASP.NET Core із використанням принципів Clean Architecture для побудови масштабованих і підтримуваних систем управління онлайн-торгівлею.

Практичне значення отриманих результатів. Розроблений вебзастосунок «ММ Studios» може бути використаний як готове рішення для управління інтернет-магазином одягу та взуття. Реалізована архітектура на основі Clean Architecture є практичним прикладом побудови багаторівневої програмної системи, придатної до масштабування та заміни окремих компонентів без змін у бізнес-логіці. Результати роботи можуть бути застосовані у навчальному процесі як демонстраційний приклад реалізації принципів чистої архітектури засобами .NET, а також як основа для подальшого розвитку системи – додавання платіжного модуля, інтеграції з API служби доставки або розгортання у хмарному середовищі.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз сучасного стану проблеми

Розробка програмних систем для управління інтернет-магазинами є складною інженерною задачею, що охоплює широкий спектр технічних і організаційних викликів. Для виявлення актуальних проблем у цій сфері проведено аналіз існуючих рішень – як великих комерційних платформ, так і конкретних інтернет-магазинів одягу, що функціонують на українському та міжнародному ринках.

Одним із найбільших українських майданчиків у сфері fashion-роздрібу є інтернет-магазин Kasta. Платформа пропонує широкий асортимент одягу та взуття, реалізує каталог із фільтрацією, кошик і оформлення замовлень. Проте з технічної точки зору система демонструє характерні проблеми масштабованих монолітних рішень: при пікових навантаженнях (розпродажі, акції) спостерігається значне уповільнення відповіді сервера. Архітектура платформи не є відкритою, однак за зовнішніми ознаками – відсутністю чіткого API-шару і змішуванням логіки на рівні представлення – можна зробити висновок про наявність типових проблем підтримуваності, характерних для систем без чіткого архітектурного поділу [1].

Міжнародний бренд H&M реалізує повноцінний вебзастосунок для управління онлайн-продажами в Україні. Система забезпечує зручний каталог із розширеною фільтрацією, адаптивний інтерфейс і персоналізацію для авторизованих користувачів. Разом із тим навіть у такого масштабного гравця фіксуються проблеми безпеки: у 2022 році компанія зазнала витоку даних клієнтів у низці регіонів через вразливості у системі зберігання облікових даних. Це підтверджує, що навіть великі платформи не завжди приділяють достатньо уваги безпеці даних на архітектурному рівні [2].

Аналіз зазначених систем дозволяє виокремити спільні проблеми, що виникають при розробці та експлуатації подібних платформ незалежно від їхнього масштабу.

Однією з найгостріших проблем є забезпечення безпеки даних користувачів. Системи онлайн-торгівлі обробляють чутливу інформацію: облікові дані, адреси доставки, історію замовлень. Неналежне зберігання паролів без хешування або з використанням застарілих алгоритмів – створює критичні вразливості. Поширеними загрозами залишаються SQL-ін'єкції, міжсайтовий скриптинг (XSS) та підробка міжсайтових запитів (CSRF). Відсутність належного контролю доступу може призвести до несанкціонованого перегляду чи модифікації замовлень інших користувачів. Проблема загострюється тим, що безпека часто розглядається як другорядна задача і реалізується вже після завершення основної розробки, а не закладається в архітектуру з самого початку.

Не менш актуальною є проблема тестованості програмного коду. У системах, де бізнес-логіка перемішана з кодом доступу до бази даних або залежить від конкретного фреймворку, написання автоматизованих тестів стає вкрай складним. Розробники змушені або повністю відмовлятися від юніт-тестування, або витрачати значні зусилля на мокування залежностей. Як наслідок, помилки виявляються лише на етапі ручного тестування або вже в промисловому середовищі, що суттєво підвищує вартість їх виправлення.

Серйозною проблемою є також підтримуваність і розширюваність коду. Системи управління магазинами з часом неминуче ускладнюються: додаються нові категорії товарів, способи оплати, інтеграції з зовнішніми сервісами. Якщо архітектура застосунку не передбачає чіткого поділу відповідальності між компонентами, кожне нове розширення потребує глибокого втручання у вже існуючий код. Це збільшує ризик регресійних помилок і поступово перетворює систему на важкокеровану структуру, внесення змін до якої стає дорогим і ризикованим.

Окремої уваги заслуговує проблема масштабованості. Зі зростанням кількості користувачів і обсягу даних система починає працювати повільніше,

якщо не була спроектована з урахуванням навантаження. Неоптимізовані запити до бази даних, відсутність кешування, синхронна обробка операцій, що мають виконуватися асинхронно, – усе це призводить до деградації продуктивності. Для розробника це означає необхідність переписувати значні частини коду, тоді як для користувача – тривале очікування відповіді від сервера та погіршення загального досвіду взаємодії з магазином.

Суттєвою є і проблема сумісності та залежності від зовнішніх компонентів. Сучасні вебзастосунки, як правило, використовують велику кількість сторонніх бібліотек і пакетів. Кожне оновлення залежностей потенційно може порушити роботу системи через несумісність версій або зміни в публічному API бібліотеки. За відсутності належного управління залежностями система може стати нестабільною, а усунення таких конфліктів потребує значних витрат часу розробника.

Варто виокремити проблему недостатнього розмежування ролей і прав доступу. У системах управління магазином важливо чітко розмежувати можливості звичайного покупця й адміністратора. Відсутність або некоректна реалізація авторизаційної моделі може дозволити неавторизованому користувачу отримати доступ до адміністративних функцій, що є критичною вразливістю. Крім того, надмірно широкі права доступу навіть для авторизованих користувачів підвищують ризики внаслідок випадкових або зловмисних дій.

Таким чином, актуальні проблеми у сфері розробки систем управління інтернет-магазинами охоплюють безпеку даних, складність тестування, низьку підтримуваність коду, проблеми масштабованості, залежність від зовнішніх компонентів і некоректне розмежування прав доступу. Вирішення цих проблем потребує застосування сучасних архітектурних підходів і перевірених технологій, що і є метою даної кваліфікаційної роботи бакалавра.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

На підставі проведеного аналізу проблемної області сформульовано такі завдання кваліфікаційної роботи бакалавра:

- проаналізувати архітектурний підхід Clean Architecture та обґрунтувати його застосування для побудови вебзастосунку з чітким поділом на шари Domain, Application, Infrastructure та Presentation;
- обрати технологічний стек на основі ASP.NET Core, Entity Framework Core та SQLite і обґрунтувати доцільність кожного інструменту відповідно до вимог системи;
- спроектувати структуру бази даних для зберігання відомостей про товари, користувачів і замовлення, визначити механізми безпечного зберігання облікових даних виконати тестування працездатності та коректності роботи сайту;
- реалізувати функціональні модулі системи: каталог товарів, реєстрацію та авторизацію користувачів, кошик і оформлення замовлень;
- провести тестування застосунку, перевірити коректність роботи ключових сценаріїв і оцінити відповідність системи визначеним вимогам.

Висновки до розділу 1

У першому розділі виявлено основні проблеми, з якими стикаються розробники та користувачі систем управління інтернет-магазинами: вразливості у сфері безпеки даних, складність написання автоматизованих тестів, низька підтримуваність коду за відсутності чіткої архітектури, проблеми масштабованості при зростанні навантаження, залежність від сторонніх компонентів і некоректне розмежування прав доступу.

Встановлено, що вразливості у сфері безпеки даних залишаються однією з найгостріших проблем: неналежне зберігання паролів, відсутність захисту від SQL-ін'єкцій та XSS-атак, а також некоректна реалізація авторизаційної моделі

створюють критичні ризики для персональних даних користувачів. Визначено, що проблема складності тестування безпосередньо пов'язана з архітектурними рішеннями: змішування бізнес-логіки з кодом доступу до бази даних унеможлиблює написання ізольованих юніт-тестів, що призводить до виявлення помилок лише на пізніх етапах розробки або вже в промисловому середовищі.

Досліджено, що низька підтримуваність і розширюваність коду є наслідком відсутності чіткого архітектурного поділу: кожне нове розширення функціоналу потребує глибокого втручання в існуючий код і підвищує ризик регресійних помилок. Проблеми масштабованості при зростанні навантаження, залежність від сторонніх компонентів та їх версій, а також некоректне розмежування ролей і прав доступу доповнюють картину системних викликів, що стоять перед розробниками сучасних вебсистем торгівлі.

На підставі проведеного аналізу сформульовано сім конкретних завдань кваліфікаційної роботи бакалавра, виконання яких передбачає вирішення зазначених проблем засобами сучасного технологічного стеку та архітектурних підходів. Реалізація поставлених завдань розкривається у наступних розділах роботи.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЕНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Система передбачає три типи акторів: «відвідувач» (неzareєстрований користувач), «zareєстрований користувач» і «адміністратор». Кожен актор має чітко визначений набір дозволених дій, що забезпечує коректне розмежування прав доступу.

Для моделювання системи обрано нотацію UML як загальновизнаний стандарт у проєктуванні програмного забезпечення. Діаграма варіантів використання (Use Case Diagram) відображає взаємодію між акторами та функціями системи. Діаграма класів описує структуру сутностей і зв'язки між ними. Діаграма компонентів ілюструє архітектурний поділ системи на шари відповідно до принципів Clean Architecture [3].

На основі аналізу предметної області та визначених цілей системи сформовано перелік функціональних вимог. Кожна вимога описує конкретну дію, яку система має забезпечити для відповідного актора, та має визначений пріоритет виконання. Зведений перелік функціональних вимог наведено у таблиці 2.1.

Таблиця 2.1 – Функціональні вимоги до системи

№	Вимога	Актор	Пріоритет
1	Реєстрація нового користувача	Відвідувач	Високий
2	Авторизація zareєстрованого користувача	Користувач	Високий
3	Перегляд каталогу товарів	Відвідувач, Користувач	Високий
4	Фільтрація товарів за розміром, кольором та ціною	Відвідувач, Користувач	Середній
5	Додавання товару до кошика	Користувач	Високий
6	Оформлення замовлення з введенням даних доставки	Користувач	Високий
7	Управління товарами (додавання, редагування, видалення)	Адміністратор	Високий
8	Перегляд і обробка замовлень	Адміністратор	Високий
9	Управління обліковими записами користувачів	Адміністратор	Середній

Окрім функціональних вимог, важливу роль відіграють нефункціональні вимоги (табл. 2.2), що визначають якісні характеристики системи – продуктивність, безпеку, надійність, зручність використання та підтримуваність.

Таблиця 2.2 – Нефункціональні вимоги до системи

Категорія	Вимога	Показник
Продуктивність	Час відповіді сервера на типовий запит	Не більше 500 мс
Безпека	Зберігання паролів з хешуванням	BCrypt / SHA-256
Надійність	Доступність системи	Не менше 99 %
Зручність	Адаптивний інтерфейс	Обов'язково
Підтримуваність	Покриття бізнес-логіки юніт-тестами	Не менше 70 %
Масштабованість	Архітектура повинна дозволяти заміну БД без зміни бізнес логіки	Clean Architecture

Діаграма варіантів використання (рис. 2.1) описує взаємодію акторів із системою. Актор «Відвідувач» може переглядати каталог товарів, фільтрувати їх та реєструватися. Актор «Користувач» успадковує можливості відвідувача й додатково може авторизуватися, додавати товари до кошика, оформлювати замовлення та переглядати їх історію. Актор «Адміністратор» має доступ до управління товарами (створення, редагування, видалення), перегляду й обробки замовлень та управління обліковими записами користувачів.



Рисунок 2.1 – Діаграма варіантів використання

Діаграма класів (рис. 2.2) відображає основні сутності системи та зв'язки між ними. Клас User містить атрибути: Id, Email, PasswordHash, Role, CreatedAt. Клас Product – Id, Name, Description, Price, Category, Size, ImageUrl, Stock. Клас Order – Id, UserId, CreatedAt, Status, DeliveryAddress, TotalAmount. Клас OrderItem – Id, OrderId, ProductId, Quantity, UnitPrice. Між User та Order існує зв'язок «один до багатьох». Order і OrderItem пов'язані аналогічно, а Product і OrderItem – зв'язком «один до багатьох».

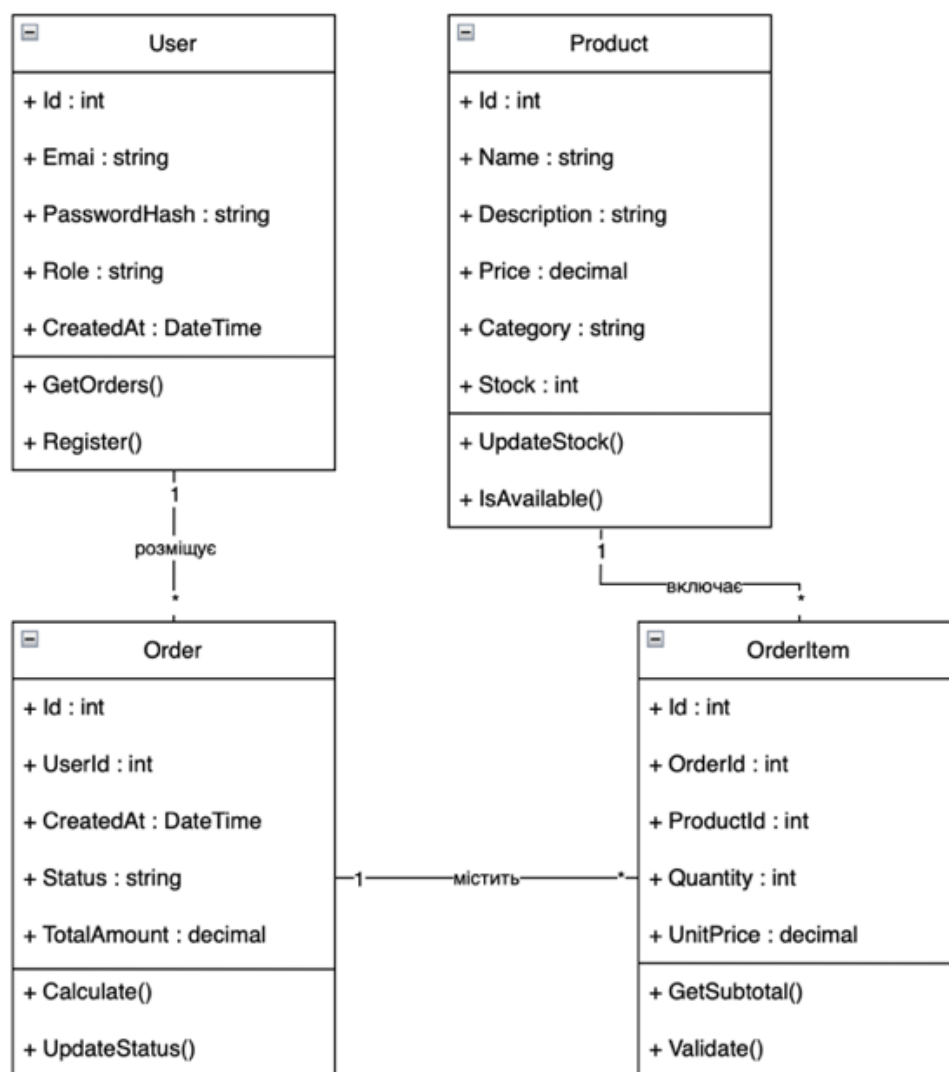


Рисунок 2.2 – Діаграма класів

Діаграма компонентів (рис. 2.3) відображає чотири шари системи відповідно до Clean Architecture. Шар Presentation містить контролери ASP.NET

Core MVC та представлення Razor Pages. Шар Application включає інтерфейси сервісів, команди та обробники. Шар Infrastructure реалізує репозиторії, контекст бази даних EF Core та SQLite. Шар Domain містить сутності та бізнес-правила.

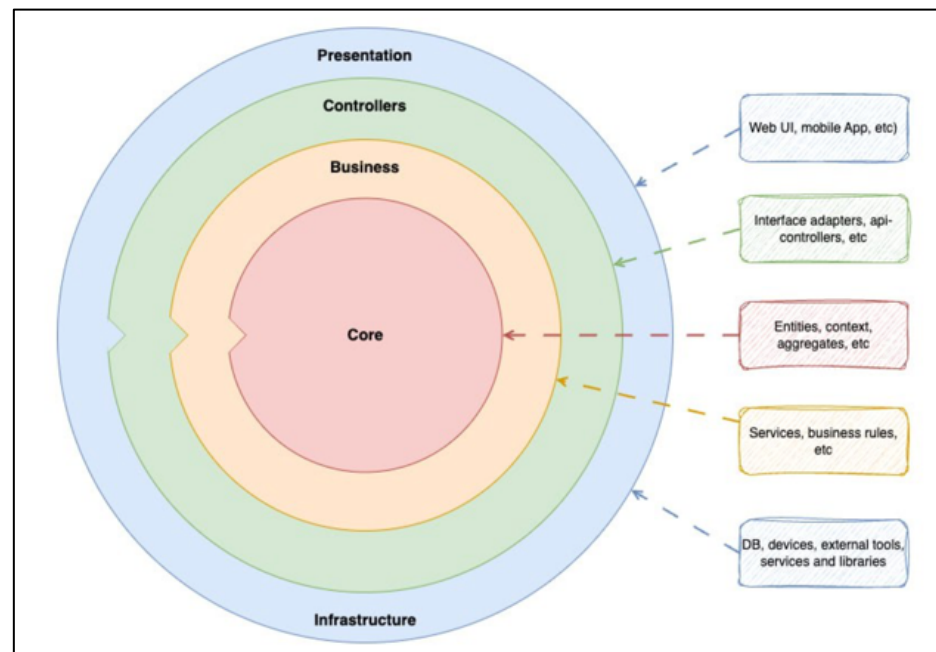


Рисунок 2.3 – Діаграма компонентів

Інформаційні потоки системи організовано таким чином: запит від користувача надходить до контролера рівня Presentation, який делегує обробку до відповідного сервісу рівня Application. Сервіс звертається до репозиторію рівня Infrastructure для отримання або збереження даних у SQLite через Entity Framework Core. Результат повертається зворотнім шляхом і відображається користувачеві. Такий підхід забезпечує незалежність бізнес-логіки від деталей реалізації зберігання даних.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір технологічного стеку є одним із ключових архітектурних рішень, що безпосередньо впливає на якість, підтримуваність і масштабованість системи. З

метою обґрунтованого вибору інструментів було проведено порівняльний аналіз наявних альтернатив за критеріями продуктивності, сумісності з обраним архітектурним підходом, складності налаштування та ліцензійної політики.

Для реалізації серверної частини вебзастосунку розглядалися чотири альтернативи: ASP.NET Core, Spring Boot, Django та Node.js/Express . Порівняння здійснювалося за такими критеріями: мова програмування, продуктивність, наявність вбудованого механізму впровадження залежностей (DI), кросплатформеність, рівень підтримки Clean Architecture, зрілість екосистеми та тип ліцензії (табл. 2.3).

Таблиця 2.3 – Порівняння серверних фреймворків

Критерій	ASP.NET Core	Spring Boot	Django	Node.js/Express
Мова Програмування	C#	Java	Python	JavaScript
Продуктивність	Висока	Висока	Середня	Висока
Підтримка DI	Вбудована	Вбудована	Часткова	Стороння
Підтримка Clean Architecture	Відмінна	Добра	Обмежена	Обмежена
Зрілість та документація	Висока	Висока	Висока	Середня
Кросплатформеність	Так	Так	Так	Так
Ліцензія	MIT	Apache 2.0	BSD	MIT

ASP.NET Core – відкритий кросплатформений фреймворк від Microsoft для розробки вебзастосунків і API на мові C#. Його ключовою перевагою є вбудований механізм впровадження залежностей, який є фундаментальним для реалізації Clean Architecture: саме через DI-контейнер реєструються та підключаються реалізації інтерфейсів між шарами. Фреймворк має зрілу екосистему, активно підтримується і регулярно оновлюється. За показниками продуктивності ASP.NET Core стабільно входить до лідерів серед вебфреймворків згідно з незалежними бенчмарками TechEmpower .

Spring Boot (Java) є потужною альтернативою з подібними можливостями, проте вимагає знання мови Java та значно важчого середовища виконання JVM, що підвищує споживання оперативної пам'яті. Django (Python) має просту архітектуру і зручний ORM, однак слабо підтримує принципи Clean Architecture через жорстку прив'язку до власних шаблонів проектування. Node.js/Express є

легковісним рішенням із високою продуктивністю, але відсутність вбудованого DI і менш структурована екосистема ускладнюють реалізацію чіткого архітектурного поділу.

За результатами порівняння для розробки обрано ASP.NET Core як фреймворк, що найповніше відповідає вимогам проєкту: забезпечує вбудований DI, відмінно підходить для реалізації Clean Architecture, має відкриту ліцензію MIT і широку документацію.

Для роботи з базою даних розглядалися три інструменти: Entity Framework Core, Dapper та NHibernate. Порівняння здійснювалося за типом інструменту, підтримкою підходу Code First, наявністю вбудованих міграцій, швидкістю розробки та складністю початкового налаштування (табл 2.4).

Таблиця 2.4 – Порівняння інструментів для роботи з базою даних

Критерій	Entity Framework Core	Dapper	NHibernate
Тип	Повноцінний ORM	Micro-ORM	Повноцінний ORM
Підхід Code First	Так	Ні	Частково
Міграції БД	Вбудовані	Відсутні	Сторонні
Швидкість розробки	Висока	Середня	Низька
Складність налаштування	Низька	Низька	Висока
SQLite	Так	Так	Так

Entity Framework Core – повноцінний ORM від Microsoft, що забезпечує роботу з базою даних через об'єктну модель C# без написання SQL-запитів вручну. Підхід Code First дозволяє описувати структуру таблиць безпосередньо у класах-сутностях домену, що повністю відповідає принципам Clean Architecture: модель бази даних впливає з бізнес-сутностей, а не навпаки. Система міграцій EF Core автоматично відстежує зміни у структурі даних і генерує відповідні SQL-скрипти, що суттєво спрощує еволюцію схеми бази даних протягом розробки.

Dapper є легковісним Micro-ORM, що виконує маппінг результатів SQL-запитів на C#-об'єкти. Він забезпечує вищу продуктивність порівняно з EF Core у сценаріях зі складними запитамі, однак потребує написання SQL вручну і не підтримує Code First та міграції, що збільшує обсяг ручної роботи і знижує

швидкість розробки. NHibernate є зрілим ORM із багатими можливостями, але має складну конфігурацію через XML-файли або атрибути, застарілий API і значно меншу активність спільноти порівняно з EF Core.

За результатами порівняння обрано Entity Framework Core. Вбудована підтримка Code First і міграцій, висока швидкість розробки та мінімальна складність налаштування роблять його оптимальним вибором для даного проєкту.

Для зберігання даних застосунку розглядалися чотири СУБД: SQLite, PostgreSQL, MySQL та MS SQL Server (табл. 2.5). Аналіз проводився за необхідністю серверного процесу, складністю налаштування, підтримкою з боку EF Core, ліцензійною політикою та придатністю для використання в умовах розробки [7].

Таблиця 2.5 – Порівняння баз даних

Критерій	SQLite	PostgreSQL	MySQL	MS SQL Server
Серверний процес	Не потрібен	Потрібен	Потрібен	Потрібен
Складність налаштування	Мінімальна	Середня	Середня	Висока
Підтримка EF Core	Так	Так	Так	Так
Ліцензія	Відкрита	Відкрита	Відкрита	Комерційна
Придатність для розробки	Відмінна	Добра	Добра	Обмежена

SQLite – вбудована реляційна СУБД, що зберігає всі дані в єдиному файлі без потреби у запуску окремого серверного процесу. Це суттєво спрощує розгортання та налагодження застосунку: для початку роботи достатньо вказати шлях до файлу бази даних у рядку підключення. SQLite повністю підтримується Entity Framework Core, включаючи всі можливості міграцій і LINQ-запитів. Відкрита ліцензія дозволяє вільне використання без будь-яких обмежень.

PostgreSQL та MySQL є потужними серверними СУБД з відкритим кодом і широкими можливостями. Вони є оптимальним вибором для продуктивних систем із високим навантаженням, однак потребують окремого встановлення, налаштування та адміністрування сервера бази даних, що ускладнює процес розробки. MS SQL Server – комерційна СУБД від Microsoft із відмінною

інтеграцією з екосистемою .NET, проте її використання у безкоштовній версії (Developer/Express) має обмеження, а комерційна ліцензія є значно дорожчою.

За результатами порівняння обрано SQLite. Відсутність необхідності у серверному процесі, мінімальна складність налаштування та повна підтримка EF Core роблять її ідеальним вибором для розробки. Водночас архітектура Clean Architecture гарантує, що за потреби SQLite може бути замінена на PostgreSQL або MS SQL Server без жодних змін у бізнес-логіці системи – достатньо лише змінити провайдер у шарі Infrastructure [8].

Для реалізації системи входу та розмежування прав доступу обрано механізм Cookie Authentication, вбудований в ASP.NET Core. Цей підхід добре підходить для серверних вебзастосунків на Razor Pages, оскільки не потребує налаштування окремого JWT-сервера. Паролі користувачів зберігаються виключно у захешованому вигляді з використанням BCrypt – сучасного стандарту безпечного зберігання облікових даних, що є стійким до атак перебором завдяки регульованому коефіцієнту складності (work factor). Розмежування доступу між роллю «Користувач» та роллю «Адміністратор» реалізується через атрибут [Authorize(Roles = "...")], що дозволяє декларативно обмежувати доступ до окремих контролерів і дій [9].

Одним із ключових бізнес-процесів системи є оформлення замовлення. Алгоритм реалізується у такій послідовності:

- користувач додає обрані товари до кошика; кошик зберігається у серверній сесії і прив'язаний до поточного облікового запису;
- під час ініціювання оформлення замовлення система перевіряє наявність кожного товару на складі відповідно до зазначеної кількості;
- якщо перевірка успішна, у межах однієї транзакції бази даних створюється запис Order із даними доставки та пов'язані записи OrderItem для кожної позиції кошика;
- залишок товару на складі (поле Stock у сутності Product) зменшується відповідно до замовленої кількості;
- кошик очищається, користувач отримує підтвердження успішного

оформлення замовлення;

- у разі виникнення помилки на будь-якому кроці транзакція відкочується повністю, що гарантує цілісність даних.

Транзакційна обробка замовлення реалізується засобами Entity Framework Core через метод `SaveChangesAsync()`, що виконує всі операції як єдину атомарну одиницю роботи. Такий підхід виключає ситуацію часткового збереження даних і забезпечує узгодженість стану бази даних.

Для розробки інтерфейсу користувача обрано Razor Pages – серверний рушій шаблонів, що є частиною ASP.NET Core. Razor Pages генерує HTML на стороні сервера, що позбавляє необхідності розробляти окремий SPA-застосунок і спрощує загальну архітектуру проєкту. Для стилізації та забезпечення адаптивності інтерфейсу використовується Bootstrap 5 – один із найпоширеніших CSS-фреймворків. Його сіткова система (grid) і готові компоненти дозволяють швидко реалізувати коректне відображення сторінок на пристроях різних розмірів – від смартфонів до широкоформатних моніторів [10].

Висновки до розділу 2

У другому розділі визначено функціональні та нефункціональні вимоги до розроблюваної системи. Функціональні вимоги охоплюють десять ключових сценаріїв використання для трьох типів акторів: відвідувача, зареєстрованого користувача та адміністратора. Нефункціональні вимоги регламентують показники продуктивності, безпеки, надійності та підтримуваності системи.

Для моделювання системи обрано нотацію UML. Описано три ключові діаграми: діаграму варіантів використання, що відображає взаємодію акторів із функціями системи; діаграму класів основних сутностей домену (User, Product, Order, OrderItem) із зв'язками «один до багатьох»; діаграму компонентів, що ілюструє чотиришарову архітектуру Clean Architecture з односпрямованими залежностями між шарами Domain, Application, Infrastructure та Presentation.

Проведено порівняльний аналіз альтернативних інструментів розробки за трьома напрямками. У категорії серверних фреймворків порівнювалися ASP.NET Core, Spring Boot, Django та Node.js/Express – обрано ASP.NET Core завдяки відмінній підтримці Clean Architecture через вбудований DI-контейнер і найвищій продуктивності серед розглянутих альтернатив. У категорії ORM порівнювалися Entity Framework Core, Dapper та NHibernate – обрано EF Core завдяки підходу Code First, вбудованій системі міграцій і мінімальній складності налаштування. У категорії СУБД порівнювалися SQLite, PostgreSQL, MySQL та MS SQL Server – обрано SQLite як оптимальне рішення для розробки завдяки відсутності потреби у серверному процесі та повній підтримці EF Core.

РОЗДІЛ 3

РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ УПРАВЛІННЯ ОНЛАЙН-МАГАЗИНОМ

3.1 Практична реалізація об'єкта проєктування

Практична реалізація вебзастосунку здійснювалася відповідно до архітектурних рішень, визначених у другому розділі. Solution складається з чотирьох проєктів: OnlineShop.Domain, OnlineShop.Application, OnlineShop.Infrastructure та OnlineShop.Web, кожен із яких відповідає окремому шару Clean Architecture. Такий поділ гарантує, що бізнес-логіка залишається незалежною від фреймворку та деталей зберігання даних.

Шар Domain містить сутності предметної області та бізнес-правила. Шар Application визначає інтерфейси репозиторіїв та реалізує сервіси застосунку. Шар Infrastructure містить реалізації репозиторіїв, контекст бази даних ApplicationDbContext і міграції. Шар Web – це Blazor Server-застосунок, що реалізує інтерфейс користувача через Razor-компоненти [11].

Основою системи є сутності доменного шару, що описують ключові об'єкти предметної області. Сутність Product представляє товар інтернет-магазину і містить ідентифікатор, назву, опис, ціну, категорію та залишок на складі. Окрім властивостей, клас інкапсулює бізнес-поведінку: метод IsAvailable() перевіряє наявність товару на складі, а метод UpdateStock() змінює його кількість (ліст. 3.1).

Лістинг 3.1 – Сутність Product

```
namespace OnlineShop.Domain.Entities;

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Description { get; set; } = string.Empty;
    public decimal Price { get; set; }
    public string Category { get; set; } = string.Empty;
    public int Stock { get; set; }
```

```

    public int Type { get; set; }
    public bool IsAvailable() => Stock > 0;
    public void UpdateStock(int quantity) => Stock += quantity;
}

```

кінець лістингу 3.1

Аналогічно організовано інші сутності доменного шару. Клас `Order` зберігає інформацію про замовлення: ідентифікатор, посилання на користувача, дату створення, статус, адресу доставки та загальну суму. Клас `OrderItem` описує окрему позицію замовлення з посиланнями на замовлення і товар, кількість та ціну на момент оформлення. Клас `User` містить облікові дані користувача: електронну адресу, хеш пароля та роль у системі.

Шар `Infrastructure` реалізує доступ до даних засобами `Entity Framework Core`. Клас `AppDbContext` успадковує `DbContext` і оголошує чотири набори даних (`DbSet`) – по одному для кожної сутності домену. Метод `OnModelCreating` налаштовує зв'язки між таблицями: користувач може мати багато замовлень, замовлення містить багато позицій, кожна позиція посилається на конкретний товар. Також задається тип колонки `TEXT` для `decimal`-полів, оскільки `SQLite` не має вбудованого числового типу для дробових значень (ліст. 3.2).

Лістинг 3.2 – Контекст бази даних

```

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }

    public DbSet<User> Users => Set<User>();
    public DbSet<Product> Products => Set<Product>();
    public DbSet<Order> Orders => Set<Order>();
    public DbSet<OrderItem> OrderItems => Set<OrderItem>();

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Order>()
            .HasOne(o => o.User)
            .WithMany(u => u.Orders)
            .HasForeignKey(o => o.UserId);

        modelBuilder.Entity<OrderItem>()
    }
}

```

```

        .HasOne(i => i.Order)
        .WithMany(o => o.Items)
        .HasForeignKey(i => i.OrderId);

modelBuilder.Entity<OrderItem>()
    .HasOne(i => i.Product)
    .WithMany()
    .HasForeignKey(i => i.ProductId);
    }
}

```

кінець лістингу 3.2

Клас `ProductRepository` реалізує інтерфейс `IPProductRepository`, визначений у шарі `Application` (ліст. 3.3). Всі операції виконуються асинхронно з використанням `async/await` та методів `Entity Framework Core`: `ToListAsync()` для отримання колекцій, `FindAsync()` для пошуку за ідентифікатором, `Add()/Update()/Remove()` для модифікації даних із подальшим збереженням через `SaveChangesAsync()`. Фільтрація товарів за категорією реалізована через LINQ-вираз `Where()`. Такий підхід забезпечує неблокуючу обробку запитів і коректну роботу Blazor-компонентів у режимі `InteractiveServer`.

Лістинг 3.3 – Репозиторій товарів

```

public class ProductRepository : IPProductRepository
{
    private readonly AppDbContext _ctx;
    public ProductRepository(AppDbContext ctx) => _ctx = ctx;

    public async Task<IEnumerable<Product>> GetAllAsync() =>
        await _ctx.Products.ToListAsync();

    public async Task<Product?> GetByIdAsync(int id) =>
        await _ctx.Products.FindAsync(id);

    public async Task AddAsync(Product product)
    {
        _ctx.Products.Add(product);
        await _ctx.SaveChangesAsync();
    }

    public async Task UpdateAsync(Product product)
    {
        _ctx.Products.Update(product);
        await _ctx.SaveChangesAsync();
    }
}

```

```

    }

    public async Task DeleteAsync(int id)
    {
        var product = await _ctx.Products.FindAsync(id);
        if (product != null)
        {
            _ctx.Products.Remove(product);
            await _ctx.SaveChangesAsync();
        }
    }

    public async Task<IEnumerable<Product>> GetByCategoryAsync(string
category) =>
        await _ctx.Products
            .Where(p => p.Category == category)
            .ToListAsync();
}

```

кінець лістингу 3.3

ProductService є сервісом шару Application і виступає посередником між шаром Presentation і репозиторієм (ліст. 3.4). Сервіс отримує залежність IProductRepository через конструктор – це забезпечує принцип інверсії залежностей: сервіс залежить від абстракції, а не від конкретної реалізації. Метод CreateAsync() містить бізнес-валідацію: перед збереженням перевіряється, що назва товару не є порожньою, і у разі порушення правила кидається виняток ArgumentException.

Лістинг 3.4 – Сервіс з товарами

```

public class ProductService
{
    private readonly IProductRepository _repo;
    public ProductService(IProductRepository repo) => _repo = repo;

    public Task<IEnumerable<Product>> GetAllAsync() =>
        _repo.GetAllAsync();

    public Task<IEnumerable<Product>> GetByCategoryAsync(string category)
=>
        _repo.GetByCategoryAsync(category);

    public async Task CreateAsync(Product product)
    {

```

```

        if (string.IsNullOrEmpty(product.Name))
            throw new ArgumentException("Назва не може бути порожньою");
        await _repo.AddAsync(product);
    }

    public Task UpdateAsync(Product product) =>
        _repo.UpdateAsync(product);
    public Task DeleteAsync(int id) => _repo.DeleteAsync(id);
    public Task<Product?> GetByIdAsync(int id) => _repo.GetByIdAsync(id);
}

```

кінець лістингу 3.4

Клієнтська частина реалізована на Blazor Server із використанням Razor-компонентів. Головна сторінка Home.razor є точкою входу для користувача: вона завантажує список товарів через впроваджений ProductService, відображає категорійну сітку та забезпечує фільтрацію (ліст. 3.5). Директива @rendermode InteractiveServer вмикає двосторонній зв'язок між компонентом і сервером через SignalR, що дозволяє реагувати на події користувача в реальному часі. Властивість FilteredProducts обчислюється реактивно: при зміні activeCategory Blazor автоматично перемальовує відповідну частину DOM [12].

Лістинг 3.5 – Сервіс з товарами

```

@page "/"
@inject ProductService ProductService
@rendermode InteractiveServer
@using OnlineShop.Application.Services
@using OnlineShop.Domain.Entities

<div class="cat-grid">
    <div class="cat-item" @onclick='() => SetCategory("Woman")'>
        
    </div>
    <div class="cat-item" @onclick='() => SetCategory("Man")'>
        
    </div>
</div>

@code {
    private List<Product>? products;
    private string activeCategory = "";

    private IEnumerable<Product> FilteredProducts =>
        products?.Where(p => activeCategory == "" ||

```

```

        p.Category == activeCategory) ?? [];

protected override async Task OnInitializedAsync()
{
    products = (await ProductService.GetAllAsync()).ToList();
}

private void SetCategory(string cat)
{
    activeCategory = cat == activeCategory ? "" : cat;
}
}

```

кінець лістингу 3.5

Результатом реалізації компонента Home.razor є головна сторінка застосунку «ММ Studios», що відображається при першому відкритті сайту. Сторінка містить навігаційне меню з категоріями товарів, hero-банер із зображенням поточної колекції та категорійну сітку для швидкого переходу до відповідного розділу каталогу (рис. 3.1).

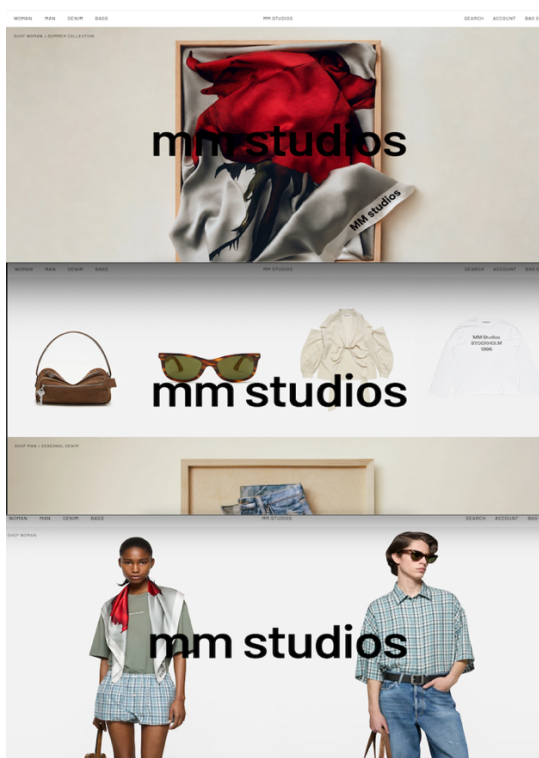


Рисунок 3.1 – Головна сторінка вебзастосунку

Сторінка `Catalog.razor` реалізує повний перелік товарів із можливістю фільтрації за категорією та переходу на сторінку деталей. `ProductDetail.razor` відображає детальну інформацію про обраний товар і забезпечує додавання до кошика. Навігація між сторінками організована через компонент `Routes.razor` із використанням стандартного роутингу Blazor на основі атрибуту `@page`.

Реєстрація залежностей здійснюється у файлі `Program.cs`. Там підключається `AppDbContext` з провайдером `SQLite`, реалізації репозиторіїв реєструються як реалізації відповідних інтерфейсів через `AddScoped<IProductRepository, ProductRepository>()`, а сервіси – через `AddScoped<ProductService>()`. Це забезпечує коректне управління часом життя об'єктів у межах одного HTTP-запиту (ліст. 3.6).

Лістинг 3.6 – Реєстрація залежностей та міграції

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite(
        builder.Configuration.GetConnectionString("Default")));

builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();

builder.Services.AddScoped<ProductService>();
builder.Services.AddScoped<OrderService>();

using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    db.Database.Migrate();
}
```

кінець лістингу 3.6

Компонент `MainLayout.razor` визначає загальну структуру застосунку: верхню навігаційну панель, основну область вмісту та футер. Навігаційна панель містить посилання на категорії товарів, логотип і блок дій користувача. Логіка відображення мега-меню реалізована засобами Blazor через змінну стану

activeMenu та обробники подій @onmouseenter і @onmouseleave без використання JavaScript (ліст. 3.7).

Лістинг 3.7 – Компонент MainLayout.razor

```

@inherits LayoutComponentBase
@inject NavigationManager Nav

<div class="shop-navbar">
    <nav class="nav-left">
        <div class="nav-item-dropdown"
            @mouseenter='() => ShowMenu("Woman")'
            @mouseleave="HideMenu">
            <a href="/catalog?category=Woman">Woman</a>
        </div>
        <div class="nav-item-dropdown"
            @mouseenter='() => ShowMenu("Man")'
            @mouseleave="HideMenu">
            <a href="/catalog?category=Man">Man</a>
        </div>
    </nav>
    <a href="/" class="navbar-logo">MM Studios</a>
    <div class="nav-right">
        <a href="/search">Search</a>
        <a href="/account">Account</a>
        <a href="/cart">Bag 00</a>
    </div>
</div>

@if (activeMenu != "")
{
    <div class="mega-menu"
        @mouseenter='() => ShowMenu(activeMenu)'
        @mouseleave="HideMenu">
        @if (activeMenu == "Woman")
        {
            <div class="mega-menu-col">
                <div class="mega-menu-title">Clothing</div>
                <a href="/catalog?category=Woman">View all</a>
                <a href="/catalog?category=Denim">Jeans</a>
                <a href="/catalog?category=Woman">T-shirts</a>
                <!-- інші підкатегорії -->
            </div>
        }
        <div class="mega-menu-close" @onclick="HideMenu">X CLOSE</div>
    </div>
}
<main>@Body</main>
@code {

```

```

private string activeMenu = "";
private void ShowMenu(string menu) => activeMenu = menu;
private void HideMenu() => activeMenu = "";
}

```

кінець лістингу 3.7

Результатом реалізації компонента MainLayout.razor є навігаційна панель із категоріями та адаптивним підменю. При активації категорії «Woman» відображається горизонтальне підменю з посиланнями на підрозділи колекції (рис. 3.2).

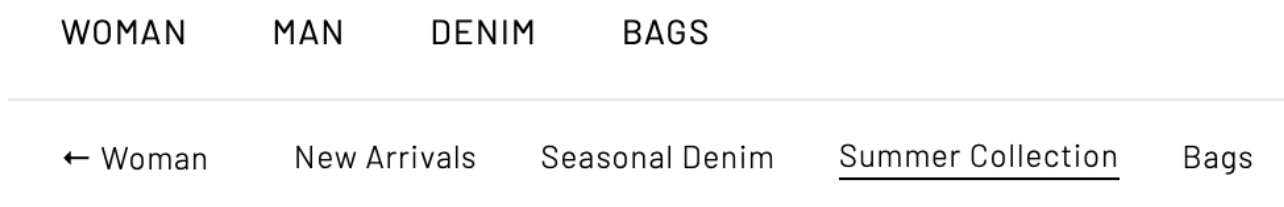


Рисунок 3.2 – Навігаційна панель

Сторінка деталей товару реалізована у компоненті ProductDetail.razor і відображається при переході користувача на конкретну позицію каталогу. Сторінка побудована за двоколонковим макетом: ліва частина містить фотографію товару, права – усю необхідну інформацію для прийняття рішення про покупку. У правій колонці відображається назва товару, ціна, сітка вибору розміру з візуальним виділенням активного варіанту, кнопка додавання до кошика та підтвердження успішного додавання «Added to bag». Нижче розташовані розкривні блоки з додатковою інформацією: умови доставки та повернення, подарункове пакування і можливість отримання у магазині. Також відображається актуальний залишок товару на складі «In stock: 10», що зчитується безпосередньо з поля Stock сутності Product у базі даних. Рисунок 3.3 демонструє сторінку деталей товару «1996 Logo T-Shirt».

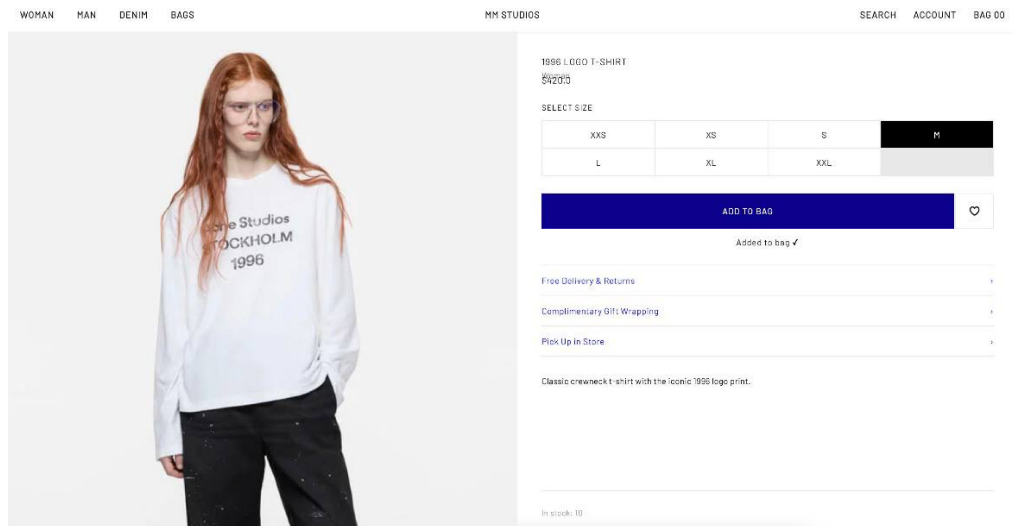


Рисунок 3.3 – Відображення деталей товару

Кошик реалізовано як бічна панель, що відкривається поверх основного вмісту без переходу на окрему сторінку. Компонент інжектує `CartService` і підписується на його подію `OnChange` – це дозволяє автоматично оновлювати відображення при будь-якій зміні вмісту кошика з будь-якого місця застосунку. Лічильник товарів у навігаційній панелі оновлюється реактивно через властивість `CartService.Count`. Для кожної позиції кошика відображається зображення товару, назва, обраний розмір та ціна. У нижній частині панелі розраховується проміжна сума, вартість доставки та загальна сума замовлення. Підписка на подію знімається у методі `Dispose()` відповідно до інтерфейсу `IDisposable`, що запобігає витокам пам'яті. Лістинг 3.8 демонструє реалізацію компонента кошика.

Лістинг 3.8 – Компонент кошика

```
@inject CartService CartService
@rendermode InteractiveServer
@implements IDisposable
<span class="nav-bag-btn" @onclick="ToggleCart">
    Bag @(CartService.Count.ToString("D2"))
</span>
<div class="cart-panel @(cartOpen ? open : empty)">
    <div class="cart-panel-header">
        <span>BAG @(CartService.Count.ToString("D2"))</span>
        <span @onclick="ToggleCart">X CLOSE</span>
    </div>
    <div class="cart-panel-items">
```

```

@foreach (var item in CartService.Items)
{
    <div class="cart-item">
        <div class="cart-item-info">
            <div>@item.Product.Name</div>
            <div>Size: @item.Size</div>
            <div>$@item.Product.Price</div>
            <div class="cart-item-qty">
                <span @onclick='() => CartService.UpdateQuantity(
                    item.Product.Id, item.Size, -1)'>-</span>
                <span>@item.Quantity</span>
                <span @onclick='() => CartService.UpdateQuantity(
                    item.Product.Id, item.Size, 1)'>+</span>
            </div>
        </div>
        <span @onclick='() => CartService.Remove(
            item.Product.Id, item.Size)'>X</span>
    </div>
}
</div>
</div>

```

кінець лістингу 3.8

Результатом реалізації компонента кошика є бічна панель, що відкривається при натисканні на «Bag» у навігації. Панель відображає список доданих товарів із зображеннями, розмірами та контролами кількості, а також підсумкову інформацію про вартість замовлення та кнопку переходу до оформлення (рис. 3.4).

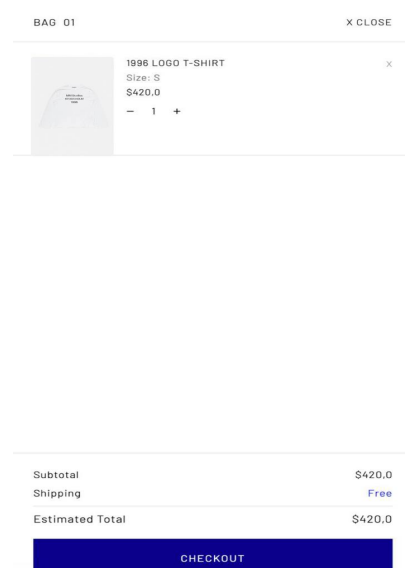


Рисунок 3.4 – Панель кошика із доданим товаром

У межах підрозділу було реалізовано практичну частину вебзастосунку відповідно до принципів Clean Architecture. Використання багат шарової структури дозволило чітко розмежувати відповідальність між компонентами системи, забезпечити незалежність бізнес-логіки від інтерфейсу користувача та механізмів збереження даних. Реалізована архітектура спростила підтримку, тестування та подальше розширення функціональності застосунку. Використання Blazor Server і Tailwind CSS дало змогу створити сучасний адаптивний інтерфейс, а застосування Entity Framework Core та SQLite забезпечило стабільну роботу із даними [14].

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування застосунку проводилося на кількох рівнях: перевірка коректності роботи окремих методів бізнес-логіки, функціональне тестування сценаріїв використання та ручне тестування інтерфейсу користувача на пристроях різних розмірів.

Оскільки бізнес-логіка сконцентрована у шарі Application і залежить виключно від інтерфейсів, її можна тестувати ізольовано без звернення до реальної бази даних. Для цього застосовується техніка підміни реалізацій (mock-об'єкти): у тесті створюється фіктивна реалізація IProductRepository, яка повертає заздалегідь визначені дані. Метод CreateAsync() перевіряється на коректне кидання винятку ArgumentException при порожній назві та на успішне збереження при валідних даних.

Тестування методу IsAvailable() сутності Product підтвердило коректну поведінку: при Stock > 0 метод повертає true, при Stock = 0 – false. Метод UpdateStock() правильно збільшує та зменшує залишок при передачі позитивного або від'ємного значення відповідно (ліст. 3.9).

Лістинг 3.9 – Юніт тестування сервісу ProductService

```

using Moq;
using OnlineShop.Application.Interfaces;
using OnlineShop.Application.Services;
using OnlineShop.Domain.Entities;
using Xunit;
namespace OnlineShop.Tests;
public class ProductServiceTests
{
    private readonly Mock<IProductRepository> _repoMock;
    private readonly ProductService _service;
    public ProductServiceTests()
    {
        _repoMock = new Mock<IProductRepository>();
        _service = new ProductService(_repoMock.Object);
    }
    [Fact]
    public async Task CreateAsync_ShouldThrowException_WhenNameIsEmpty()
    {
        var product = new Product
        {
            Name = "",
            Price = 100,
            Stock = 5
        };
        await Assert.ThrowsAsync<ArgumentException>(() =>
            _service.CreateAsync(product));
    }
    [Fact]
    public async Task CreateAsync_ShouldCallRepository_WhenDataIsValid()
    {
        var product = new Product
        {
            Name = "Laptop",
            Price = 25000,
            Stock = 10
        };
    }
}

```

кінець лістингу 3.9

Коректність схеми бази даних перевірялася за допомогою Database Console у середовищі розробки Rider. Перевірялася цілісність зовнішніх ключів між таблицями Orders і Users, OrderItems і Orders, OrderItems і Products. Тестувалося каскадне відображення: при видаленні замовлення всі пов'язані позиції OrderItem видалялися коректно. Файл onlineshop.db успішно зберігав і

відновлював дані між перезапусками застосунку.

Інтерфейс застосунку перевірявся на коректне відображення у браузерях Chrome, та Safari на пристроях із різною шириною екрана: смартфон (375 px), планшет (768 px) та десктоп (1440 px). Стили з файлу app.css та компонентний CSS MainLayout.razor.css забезпечували коректне масштабування сіткового макету та читабельність тексту на всіх розмірах (рис. 3.5).

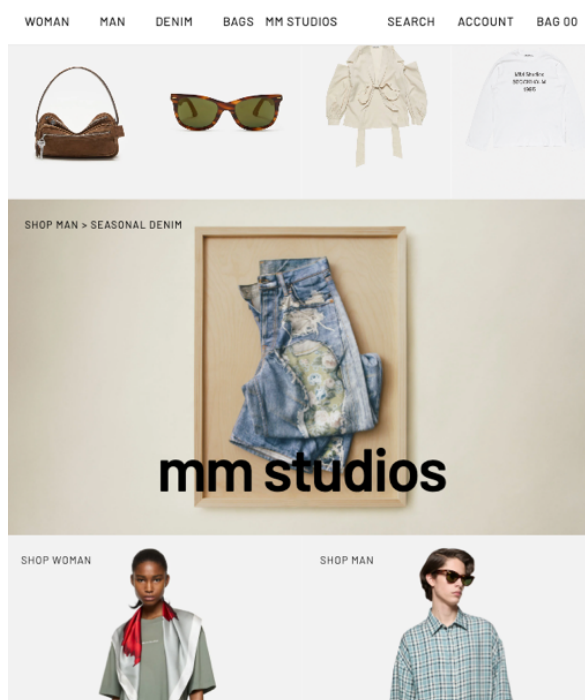


Рисунок 3.5 – Відображення вебзастосунку на екрані шириною 375 пікселів

За результатами тестування встановлено, що вебзастосунок коректно відображається на пристроях із різними розмірами екранів. Інтерфейс адаптувався до зміни ширини без порушення структури чи спотворення елементів, а компоненти зберігали зручне та логічне розташування як на ноутбуках, так і на планшетах і смартфонах.

Крім візуального аналізу, було також вручну протестовано функціональність основних елементів сайту. Перевірці підлягали навігаційне меню, картки товарів, бічна панель кошика, форма оформлення замовлення та інтеграція з API Нової пошти. Усі функції працюють стабільно та без збоїв: додаються та видаляються товари з кошика, коректно обраховується загальна

сума, валідація форми на основі бібліотеки «Zod» виконується згідно з заданими умовами, а динамічне завантаження списку відділень здійснюється згідно з обраним містом. Помилки у роботі ключових компонентів не виявлено, що підтверджує надійність реалізованої логіки та високий рівень готовності проєкту до реального використання.

3.3 Розробка бази даних

База даних застосунку реалізована з використанням SQLite та підходу Code First засобами Entity Framework Core. Структура бази даних повністю визначається класами сутностей доменного шару та конфігурацією у методі OnModelCreating класу ApplicationDbContext. Зміни у структурі відстежуються через систему міграцій і застосовуються командою `dotnet ef database update` [13].

База даних складається з чотирьох таблиць: Users, Products, Orders та OrderItems. Таблиця Users зберігає облікові записи користувачів: унікальний ідентифікатор (INTEGER PRIMARY KEY), електронну адресу (TEXT, унікальна), хеш пароля (TEXT), роль (TEXT) і дату реєстрації (TEXT). Поле PasswordHash ніколи не містить відкритого пароля – лише результат хешування BCrypt.

Таблиця Products містить інформацію про товари: ідентифікатор, назву, опис, ціну (TEXT через обмеження SQLite для decimal), категорію та залишок на складі (INTEGER). Таблиця Orders зберігає дані замовлень з посиланням на користувача через зовнішній ключ UserId, статус замовлення, адресу доставки та загальну суму. Таблиця OrderItems є зв'язуючою між Orders та Products і додатково зберігає кількість і ціну на момент оформлення замовлення – це важливо, оскільки ціна товару може змінитися після розміщення замовлення.

Зв'язки між таблицями налаштовано у методі OnModelCreating за принципом Fluent API. Зв'язок між Users і Orders є «один до багатьох»: один користувач може мати необмежену кількість замовлень, але кожне замовлення належить рівно одному користувачу. Аналогічний зв'язок між Orders і

OrderItems. Зв'язок між Products і OrderItems – «один до багатьох» без навігаційної колекції на боці Product, що спрощує модель і запобігає надмірному завантаженню даних.

Ініціалізація бази даних відбувається автоматично при першому запуску застосунку через виклик `context.Database.EnsureCreated()` або через застосування міграцій. Для тестування та демонстрації роботи системи до бази додаються початкові дані (seed data) – кілька категорій товарів одягу та взуття з тестовими позиціями, що дозволяє одразу перевірити роботу каталогу та фільтрації без ручного введення даних.

Файл бази даних `onlineshop.db` розміщується у кореневій директорії проєкту `OnlineShop.Web` і підключається через рядок підключення у файлі `appsettings.json`. Така організація забезпечує простоту розгортання: для перенесення застосунку на інший сервер достатньо скопіювати скомпільовані файли разом із файлом бази даних.

3.4 Адміністративна панель управління магазином

Адміністративна панель є окремим захищеним розділом застосунку, доступним виключно для користувачів із роллю адміністратора. Вона забезпечує повний цикл управління асортиментом інтернет-магазину та надає можливість відстежувати замовлення покупців без необхідності прямого доступу до бази даних.

Функціонал адміністративної панелі охоплює два основних модулі: управління товарами та перегляд замовлень. Модуль управління товарами дозволяє переглядати повний каталог, додавати нові позиції із заповненням усіх необхідних полів – назва, опис, ціна, категорія, залишок на складі, – редагувати існуючі товари та видаляти їх. Усі операції виконуються через `ProductService`, що забезпечує проходження бізнес-валідації перед збереженням у базі даних.

Модуль перегляду замовлень відображає список усіх замовлень покупців із зазначенням дати, статусу, загальної суми та даних доставки. Адміністратор

може переглядати деталі кожного замовлення, включаючи перелік позицій із кількостями та цінами на момент оформлення. Захист адміністративних маршрутів реалізовано через перевірку стану автентифікації у `SessionAuthService`: при спробі неавторизованого доступу відбувається перенаправлення на сторінку авторизації.

Розмежування між звичайним користувачем і адміністратором здійснюється на рівні JWT-токена: при генерації токена до нього включаються клеймси з даними користувача. Компоненти адміністративної панелі перевіряють стан автентифікації через `SessionAuthService.IsAuthenticated` і приховують або блокують доступ до захищених елементів інтерфейсу для неавторизованих користувачів.

3.5 Безпека розроблюваного вебзастосунку

Забезпечення безпеки є одним із ключових аспектів розробки вебзастосунку для управління інтернет-магазином, оскільки система обробляє персональні дані користувачів та інформацію про замовлення.

Клас `User` у доменному шарі містить поле `PasswordHash` замість відкритого пароля – це фундаментальний принцип безпечного зберігання облікових даних. Поля `FirstName` і `LastName` дозволяють персоналізувати взаємодію без збереження зайвих чутливих даних (ліст. 3.10).

Лістинг 3.10 – Сутність користувача

```
namespace OnlineShop.Domain.Entities;
public class User
{
    public int Id { get; set; }
    public string Email { get; set; } = string.Empty;
    public string PasswordHash { get; set; } = string.Empty;
    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
}
```

кінець лістингу 3.10

AuthService реалізує логіку реєстрації та входу користувача. Пароль ніколи не зберігається у відкритому вигляді – перед збереженням він хешується алгоритмом SHA-256 із додаванням солі, що захищає від атак за допомогою райдужних таблиць. При вході система порівнює хеш введеного пароля з хешем у базі даних. У разі автентифікації генерується JWT-токен (ліст 3.11) [15].

Лістинг 3.11 – Сервіс автентифікації

```
public class AuthService
{
    private readonly IUserRepository _userRepository;
    private readonly TokenService _tokenService;
    public async Task<AuthResult> RegisterAsync(
        string email, string password,
        string firstName, string lastName)
    {
        if (await _userRepository.ExistsAsync(email))
            return new AuthResult(false,
                "Користувач з таким email вже існує.");
        if (password.Length < 6)
            return new AuthResult(false);
        var user = new User
        {
            Email = email.ToLower().Trim(),
            PasswordHash = HashPassword(password),
            FirstName = firstName.Trim(),
        };
        await _userRepository.CreateAsync(user);
        var token = _tokenService.GenerateToken(user);
        return new AuthResult(true, 'Реєстрація успішна!', token);
    }
    public async Task<AuthResult> LoginAsync(
        string email, string password)
    {
        var user = await _userRepository
            .GetByEmailAsync(email.ToLower().Trim());
        if (user == null ||
            HashPassword(password) != user.PasswordHash)
            return new AuthResult(false,
                "Невірний email або пароль.");
        var token = _tokenService.GenerateToken(user);
        return new AuthResult(true, "Вхід успішний!", token);
    }
}
```

кінець лістингу 3.11

Для автентифікації без збереження стану сесії на сервері застосовується механізм JSON Web Tokens. TokenService генерує підписаний токен, що містить клеймси користувача: ідентифікатор, електронну адресу, ім'я та прізвище. Токен підписується симетричним ключем за алгоритмом HMAC-SHA256 і має термін дії 7 днів. При валідації перевіряється підпис, видавець, аудиторія та термін дії – у разі будь-якої невідповідності метод ValidateToken повертає null [16].

Сторінка Login.razor реалізує триетапний процес автентифікації. На першому етапі користувач вводить електронну адресу, після чого система перевіряє її наявність у базі даних: якщо адреса вже зареєстрована – відображається форма введення пароля, якщо ні – форма реєстрації нового акаунту. Такий підхід є зручним з точки зору UX і водночас не розкриває зломиснику перелік зареєстрованих адрес до моменту введення пароля. Форма реєстрації вимагає введення імені, прізвища, пароля та його підтвердження. На рисунку 3.6 вигляд сторінки Login.razor [17].

The image shows a web form titled "Sign in or register". Below the title is a link for "United States" and a blue link "Change location". A list of five bullet points describes the benefits of signing in: "Priority access to new collections", "Eligible to exclusive events and gifting", "Easily track your orders and book returns", "Pre access to sale", and "Priority shipping". Below this list is a light gray input field labeled "EMAIL". Underneath the email field is a solid black button with the word "CONTINUE" in white capital letters. Below the button is a thin horizontal line with the word "OR" centered. At the bottom is a white button with a thin gray border, containing the Google logo on the left and the word "GOOGLE" in capital letters on the right.

Рисунок 3.6 – Вигляд сторінки Login.razor

Висновки до розділу 3

У третьому розділі описано практичну реалізацію вебзастосунку для управління інтернет-магазином. Застосунок побудовано з дотриманням принципів Clean Architecture: solution розподілено на чотири проєкти – Domain, Application, Infrastructure та Web, між якими виконується правило односпрямованості залежностей.

Реалізовано ключові компоненти системи: сутності доменного шару з інкапсульованою бізнес-поведінкою, репозиторії на основі Entity Framework Core з асинхронними операціями, сервіси шару Application із бізнес-валідацією, контекст бази даних із налаштуванням зв'язків через Fluent API, а також Razor-компоненти клієнтської частини на Blazor Server із реактивною фільтрацією товарів.

У доменному шарі визначено сутності Product, Order, OrderItem та User з інкапсульованою бізнес-поведінкою. Шар Infrastructure містить репозиторії на основі Entity Framework Core з повністю асинхронними операціями та контекст бази даних ApplicationDbContext із налаштуванням зв'язків через Fluent API. Шар Application реалізує сервіси ProductService, OrderService та AuthService із бізнес-валідацією вхідних даних. Клієнтська частина побудована на Blazor Server із використанням Razor-компонентів.

Забезпечено багаторівневий захист даних: паролі зберігаються у хешованому вигляді алгоритмом SHA-256, автентифікація реалізована через JWT-токени з підписом HMAC-SHA256 і терміном дії 7 днів, управління станом сесії здійснюється через SessionAuthService із подією OnAuthStateChanged. Сторінка авторизації реалізує триетапний UX-процес входу та реєстрації.

Проведено багаторівневе тестування системи: перевірено бізнес-логіку сервісів, функціональність основних сценаріїв використання, цілісність схеми бази даних та адаптивність інтерфейсу на пристроях різних розмірів. Виявлені в процесі тестування дефекти усунено.

ВИСНОВКИ

Підсумовуючи результати виконаної кваліфікаційної роботи бакалавра, можна зробити висновок, що поставлена мета – розробка вебзастосунку для управління онлайн-магазином із використанням сучасних архітектурних підходів та технологій – була повністю досягнута. У процесі виконання роботи проведено комплексний аналіз предметної області, визначено функціональні та нефункціональні вимоги до системи, обґрунтовано вибір технологічного стеку та реалізовано програмний продукт, що забезпечує зручне управління товарами, замовленнями та взаємодією користувача із системою.

У першому розділі роботи було досліджено особливості сучасних систем управління інтернет-магазинами. Проаналізовано реальні платформи у сфері онлайн-торгівлі одягом – Kasta та H&M. Виявлено характерні проблеми підтримуваності, масштабованості та безпеки даних, притаманні як великим комерційним рішенням, так і системам меншого масштабу. Особливу увагу приділено питанням архітектурної організації програмного забезпечення, автоматизованого тестування та забезпечення цілісності даних. Проведений аналіз дозволив сформулювати основні завдання кваліфікаційної роботи бакалавра та визначити вимоги до майбутньої системи.

У другому розділі виконано проектування програмного забезпечення. Було визначено функціональні можливості системи для трьох типів користувачів: «відвідувача», «zareєстрованого користувача» та «адміністратора». Описано нефункціональні вимоги, що стосуються продуктивності, безпеки, адаптивності інтерфейсу та підтримуваності коду. Для моделювання структури системи використано UML-діаграми варіантів використання, класів і компонентів, які відобразили взаємодію між елементами системи та архітектурний поділ на окремі шари.

У межах другого розділу також проведено порівняльний аналіз альтернативних технологій розробки. За результатами аналізу обрано технологічний стек, до складу якого входять ASP.NET Core, Entity Framework

Core та SQLite. Використання ASP.NET Core дозволило реалізувати вебзастосунок із підтримкою сучасних підходів до розробки, зокрема механізму впровадження залежностей та багатошарової архітектури. Entity Framework Core забезпечив зручну роботу з базою даних через ORM-підхід і підтримку міграцій, а SQLite став оптимальним рішенням для локального зберігання даних завдяки простоті налаштування та інтеграції з платформою .NET.

У третьому розділі здійснено практичну реалізацію вебзастосунку відповідно до принципів Clean Architecture. Структура рішення була поділена на окремі шари Domain, Application, Infrastructure та Web, що дозволило ізолювати бізнес-логіку від деталей реалізації інтерфейсу користувача та механізмів доступу до даних. У доменному шарі реалізовано основні сутності системи та бізнес-правила, у шарі Application – сервіси та інтерфейси репозиторіїв, а в шарі Infrastructure – роботу з базою даних через Entity Framework Core. Спроектовано структуру бази даних із використанням SQLite та визначено механізми зберігання й захисту даних.

Клієнтська частина застосунку реалізована на основі Blazor Server із використанням Razor-компонентів. Було створено головну сторінку, каталог товарів, сторінку деталей товару, компонент кошика, авторизацію та реєстрацію користувачів та навігаційне меню з адаптивним відображенням. Інтерфейс забезпечує інтерактивну взаємодію без повного перезавантаження сторінок, що позитивно впливає на зручність користування системою. Для стилізації та адаптивності інтерфейсу застосовано Tailwind CSS, що дозволило забезпечити коректне відображення вебзастосунку на пристроях із різними розмірами екранів.

У межах роботи також проведено тестування програмного забезпечення. Виконано unit-тестування бізнес-логіки із використанням mock-об'єктів для перевірки сервісів застосунку без звернення до реальної бази даних. Перевірено коректність роботи методів доменних сутностей, зокрема перевірку наявності товару та зміну залишку на складі. Окремо протестовано схему бази даних, цілісність зовнішніх ключів і каскадне видалення пов'язаних записів. Крім того,

здійснено ручне тестування інтерфейсу користувача в різних браузерх і на пристроях із різною шириною екрана, що підтвердило правильність реалізації адаптивного дизайну та стабільність роботи інтерфейсу.

Результатом виконання кваліфікаційної роботи бакалавра став повноцінний вебзастосунок для управління онлайн-магазином, який відповідає поставленим вимогам і демонструє можливість практичного застосування сучасних технологій та архітектурних підходів у розробці інформаційних систем. Реалізована система є масштабованою, підтримуваною та придатною для подальшого розширення функціональності, зокрема інтеграції платіжних сервісів, систем аналітики, авторизації через зовнішні сервіси або переходу на продуктивні серверні СУБД.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. KASTA Kasta Superapp. URL: <https://kasta.ua> (дата звернення: 10.05.2026).
2. H&M offers fashion and quality at the best price. URL: https://www.hm.com/entrance.ahtml?orguri=%2Fuk_ua%2Findex.html (дата звернення: 10.05.2026).
3. UML 2.5 Specification. Object Management Group. URL: <https://www.omg.org/spec/UML/2.5/> (дата звернення: 10.05.2026).
4. ASP.NET Core documentation. Microsoft Docs. URL: <https://learn.microsoft.com/en-us/aspnet/core/> (дата звернення: 10.05.2026).
5. Entity Framework Core documentation. Microsoft Docs. URL: <https://learn.microsoft.com/en-us/ef/core> (дата звернення: 10.05.2026).
6. Code First Migrations in Entity Framework Core. URL: <https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/> (дата звернення: 11.05.2026).
7. SQLite official documentation. URL: <https://www.sqlite.org/docs.html> (дата звернення: 11.05.2026).
8. JSON Web Tokens (JWT) – Introduction. URL: <https://adwservice.com.ua/en/weebly-website-builder-review> (дата звернення: 11.05.2026).
9. Authentication and Authorization in ASP.NET Core. URL: <https://learn.microsoft.com/en-us/aspnet/core/security/> (дата звернення: 11.05.2026).
10. Bootstrap 5 official documentation. URL: <https://getbootstrap.com/docs/5.3/> (дата звернення: 12.05.2026).
11. Blazor Server vs Blazor WebAssembly: Choosing the Right Model. URL: <https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models> (дата звернення: 12.05.2026).
12. Roth D. Blazor component lifecycle. Microsoft Docs. URL: <https://learn.microsoft.com/en-us/aspnet/core/blazor/components/lifecycle> (дата звернення: 12.05.2026).

13. SQLite provider for Entity Framework Core. URL: <https://brander.ua/blog/perevahy-freymvorku-laravel> (дата звернення: 15.05.2026).
14. Introduction to Tailwind CSS. URL: <https://www.geeksforgeeks.org/introduction-to-tailwind-css/> (дата звернення: 17.05.2026).
15. SHA-256 and Cryptographic Hash Functions. URL: <https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha256> (дата звернення: 17.05.2026).
16. Microsoft Learn: Build with answers in reach. URL: <https://learn.microsoft.com/dotnet/api/system.identitymodel.tokens.securitytokenhandler.validateToken?view=netframework-4.8.1> (дата звернення: 20.05.2026).
17. Microsoft Learn: Simple authorization in ASP.NET Core. URL: <https://learn.microsoft.com/aspnet/core/security/authorization/simple?view=aspnetcore-10.0> (дата звернення: 20.05.2026).