НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

_____комп'ютерних наук_____

_____ /Голуб Б. Л., доц.,

к.т.н./

(підпис) (ПІБ, вчене звання і ступінь)

«___»_____2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Програмне забезпечення веб системи продажу електроніки»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми

_____к.т.н., доцент_____ _____ _____/ Вайганг Г.О./

(науковий ступень та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи

_____к.т.н., доцент_____ _____ ____/ Боярінова Ю. Є/

(науковий ступень та вчене звання) (підпис) (ПІБ)

Виконав _____ _____/ Коноваленко Б. А./

(підпис) (ПІБ студента)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ

Завідувач кафедри

комп'ютерних наук

к.т.н.. доцент _____ /Голуб Б. Л.
(науковий ступінь, вчене звання) (підпис) (ПІБ)

«___»_____2025 р

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи студенту

Коноваленко Богдану Анатолійовичу

(прізвище, ім'я, по-батькові)

Спеціальність 121 «Інженерія програмного забезпечення»

Тема бакалаврської роботи Програмне забезпечення системи веб системи продажу електроніки затверджена наказом ректора НУБіП України від «16» грудня 2024р. №2248

Термін подання завершеної роботи на кафедру 2025.05.25

(рік, місяць, число)

Вихідні дані до бакалаврської кваліфікаційної роботи:

опис програмного забезпечення

програмне забезпечення у вигляді веб-додатку

Перелік питань що розглядаються

1. Аналіз проблемної області.
2. Вибір та обґрунтування засобів для розробки системи.
3. Проєктування інформаційної системи.
4. Висновки.

Дата видачі завдання «15» грудня 2025р.

Керівник бакалаврської кваліфікаційної роботи _____ / Боярінова Ю. Є. /

(підпис) (прізвище та ініціали)

Завдання прийняла до виконання _____ / Коноваленко Б.А./

(підпис) (прізвище та ініціали студента)

ЗМІСТ

ВСТУП

У наш час інформаційні технології стрімко проникають у всі сфери життя, суттєво змінюючи принципи ведення бізнесу. Зокрема, галузь роздрібної торгівлі активно трансформується завдяки електронній комерції, яка дозволяє автоматизувати процеси замовлення, продажу, обробки запитів та обслуговування клієнтів. Сучасні бізнес-моделі активно впроваджують цифрові технології, щоб зменшити витрати й забезпечити конкурентоспроможність на ринку [1]. Особливо актуальним є розвиток веб-систем для продажу електроніки, яка має широкий попит серед населення та потребує постійного оновлення асортименту, гнучких механізмів пошуку, фільтрації та оформлення замовлень.

Створення ефективного програмного забезпечення для цієї галузі дозволяє оптимізувати взаємодію між продавцем і покупцем, зменшити витрати на обслуговування та збільшити обсяг продажів.

Метою даної дипломної роботи є розробка повнофункціонального веб-додатку для продажу електронної техніки, який задовольняє потреби як користувачів, так і адміністраторів. Система має забезпечити можливість реєстрації та авторизації користувачів, перегляду та фільтрації товарів за

категоріями, додавання товарів до кошика, а також керування контентом з боку адміністратора.

Для побудови системи було використано ASP.NET Core — сучасний фреймворк, що підтримує шаблон MVC і забезпечує високу продуктивність веб-застосунків [2]. А розробка структури системи буде здійснювалась із дотриманням принципів Clean Architecture, яка дозволяє відокремити бізнес-логіку від інфраструктурних компонентів і забезпечує незалежність від зовнішніх фреймворків [3].

Тож, додаток має бути адаптивним, зручним у використанні, безпечним і відповідати сучасним стандартам веб-розробки.

У процесі реалізації програмного забезпечення було використано такі технології та інструменти:

- ASP.NET Core MVC — як основний фреймворк для побудови веб-додатку;
- C# — мова програмування для реалізації серверної логіки;
- Entity Framework Core — ORM для взаємодії з базою даних;
- SQL Server — система управління реляційною базою даних;
- Razor Pages — для динамічного створення HTML-контенту на стороні сервера;
- Bootstrap 5 — фреймворк для реалізації адаптивного та стилізованого інтерфейсу;
- HTML5 та CSS3 — базові технології для розмітки та стилізації сторінок;
- JavaScript — для клієнтської взаємодії та динамічного оновлення інтерфейсу;
- Fetch API + FormData — для обміну даними між клієнтом і сервером без перезавантаження сторінки;
- AutoMapper — бібліотека для приведення одних сутностей до інших;
- Clean Architecture (чиста архітектура) — структурна основа для організації коду та розділення відповідальності;
- Monolithic Architecture — обрана архітектурна модель, що об’єднує всі компоненти в одному проєкті.

Робота представлена на перевірку науковому керівнику дипломного проєкту, який здійснює рецензування функціональності, якості реалізації та відповідності технічним і методичним вимогам. За результатами представлення внесено низку покращень, що підвищили зручність використання та узгодженість програмної логіки.

Структура пояснювальної записки відповідає загальним вимогам до дипломних робіт і містить чотири основні розділи:

1. «Системний аналіз предметної області» — розглянуто специфіку електронної комерції, описано предметну область, сформульовано вимоги до майбутньої системи, побудовано моделі та проведено аналіз існуючих рішень.
2. «Проєктування інформаційного та програмного забезпечення» — присвячений побудові ER-діаграми, діаграм класів, кооперацій, компонентів та інших моделей, які описують архітектуру додатку.
3. «Розробка інформаційного та програмного забезпечення» — детально описано процес реалізації системи, вибір технологій, структуру бази даних та основні програмні модулі.
4. «Рекомендації щодо впровадження та експлуатації системи» — містить результати тестування, опис системних вимог та рекомендації щодо встановлення програмного продукту.

Пояснювальна записка створена з дотриманням методичних вимог, містить необхідні діаграми, фрагменти коду, опис бази даних, приклади інтерфейсу та інші допоміжні матеріали, що дозволяють повноцінно оцінити розроблене програмне забезпечення.

1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис предметної області

Сфера продажу електронної техніки в сучасних умовах інтенсивно розвивається, охоплюючи як традиційний офлайн-ринок, так і швидко зростаючий сегмент онлайн-торгівлі. У зв’язку зі стрімким розвитком електронної комерції, цифровізацією бізнес-процесів і активною популяризацією мережі Інтернет серед населення, спостерігається підвищений попит на ефективні, зручні та функціональні веб-системи для здійснення покупок електроніки через Інтернет. Продажі через веб-додатки дозволяють компаніям зменшити витрати на оренду приміщень, найм персоналу, покращити логістику та, що важливо, надати клієнтам можливість швидко знаходити, порівнювати та купувати продукцію, не виходячи з дому [4].

Електронна техніка — одна з найактивніших категорій товарів в електронній комерції, адже потреба в смартфонах, ноутбуках, периферійних пристроях та побутовій техніці зростає паралельно з розвитком інформаційних технологій. Покупці вимагають не лише широкого асортименту, а й точного, структурованого опису товарів, можливості фільтрації за технічними параметрами, зображень продукції, відгуків, швидкої обробки замовлень, зручного способу оплати й доставки.

У межах даної дипломної роботи предметну область становить веб-система продажу електроніки, яка включає повний цикл взаємодії між користувачем та адміністратором платформи. Користувачі мають змогу переглядати товари за категоріями (наприклад, "Смартфони", "Ноутбуки", "Аксесуари"), використовувати пошук та фільтрацію, ознайомлюватися з описом товарів, здійснювати реєстрацію, авторизацію, оформлення замовлень. Зі свого боку, адміністратор керує структурою каталогу: додає, редагує, видаляє категорії, підкатегорії та товари, оновлює інформацію про характеристики, зображення, бренди та ціни.

До характерних рис предметної області належать:

- необхідність зберігання великого обсягу структурованої інформації про товари;
- потреба у швидкому пошуку й фільтрації по багатьох критеріях (категорія, ціна, бренд тощо);
- керування ролями користувачів (звичайний покупець та адміністратор);
- забезпечення надійної реєстрації та автентифікації;
- наявність інструментів для гнучкого адміністрування даних без потреби доступу до бази даних напряму;
- підтримка адаптивного веб-інтерфейсу для роботи з різних пристроїв.

Особливістю реалізації даної предметної області є застосування принципів чистої архітектури Роберта Мартіна, що забезпечує логічне розділення шарів відповідальності: від інтерфейсу користувача до бізнес-логіки та доступу до даних [3]. Такий підхід дозволяє досягти гнучкості, масштабованості та тестованості розробленого веб-застосунку.

Крім того, розв’язання поставлених завдань в межах цієї предметної області вимагає інтеграції базових знань із програмування, проєктування баз даних, інженерії ПЗ, а також глибокого розуміння потреб кінцевих користувачів, що забезпечує реальну практичну цінність дипломної розробки [4].

1.2. Аналіз вимог до програмної системи

Розробка програмного забезпечення починається з етапу визначення вимог, який є ключовим для розуміння потреб користувачів, формування обсягу робіт, виявлення обмежень та проєктних ризиків. Згідно з практиками інженерії програмного забезпечення, саме вимоги впливають на структуру архітектури, вибір технологій та модель життєвого циклу системи [6]. А аналіз вимог дозволяє зрозуміти очікування користувачів, визначити обсяг робіт і виявити потенційні ризики ще на ранньому етапі створення веб-системи.

У межах дипломної роботи розглядається веб-система, призначена для

продажу електроніки через інтернет. Вона повинна забезпечувати дві основні групи користувачів — звичайних користувачів (покупців) та адміністратора.

Функціональні вимоги

До функціональних вимог системи належать: вимоги для користувача- покупця та адміністратора.

Вимоги для користувача-покупця:

- реєстрація, авторизація та автентифікація;
- перегляд списку товарів за категоріями й підкатегоріями;
- фільтрація товарів за ціною, брендом, категорією;
- пошук товарів за ключовими словами;
- перегляд детальної інформації про товар (опис, фото, характеристики);
- оформлення замовлення;
- перегляд власного профілю.

Вимоги для адміністратора:

- керування категоріями та підкатегоріями (створення, редагування, видалення);
- керування товарами (додавання, редагування, видалення);
- завантаження зображень товарів;
- перегляд списку замовлень;
- перегляд та редагування інформації про користувачів.

Нефункціональні вимоги

Нефункціональні вимоги визначають якість та надійність функціонування системи. Серед них — адаптивність, продуктивність, безпека, масштабованість та підтримуваність. Як зазначено в дослідженнях, ці аспекти є вирішальними для успішності e-commerce систем, зокрема через вимоги до безперервної доступності та високої швидкодії [7].

Окрема увага приділяється зручності користування інтерфейсом (usability), оскільки саме від неї залежить досвід взаємодії клієнта з системою. За даними наукових оглядів, врахування нефункціональних вимог на ранніх етапах знижує витрати на доопрацювання системи в майбутньому [8].

Розглянемо детальніше нефункціональні вимоги, які забезпечують якість роботи системи:

- Адаптивність — веб-інтерфейс повинен коректно відображатися на екранах різного розміру (ПК, планшети, смартфони).
- Продуктивність — система повинна забезпечувати швидкий відгук при роботі з каталогом товарів та базою даних.
- Безпека — дані користувачів мають зберігатися захищено; необхідно використовувати хешування паролів та перевірку автентичності.
- Масштабованість — архітектура має дозволяти майбутнє розширення функціоналу без кардинальної перебудови.
- Зручність використання (Usability) — інтерфейс має бути інтуїтивно зрозумілим для кінцевого користувача.
- Підтримуваність — структура коду повинна бути розділена за принципами чистої архітектури, щоб полегшити супровід.

Обмеження та припущення

Під час аналізу також були визначені певні обмеження:

- система розробляється як монолітний застосунок з використанням ASP.NET Core MVC;
- база даних зберігається локально або на сервері SQL Server;
- система не передбачає інтеграції з платіжними сервісами (оплата — поза межами функціоналу);
- підтримка лише української мови на рівні інтерфейсу;
- роль адміністратора реалізована через окремий інтерфейс доступу.

Всі вищезазначені вимоги були визначені на основі загальноприйнятих стандартів веб-розробки, а також виходячи з практичних міркувань зручності, безпеки та стабільної роботи системи. На їх основі у подальших розділах буде здійснено проєктування логічної структури, реалізацію модулів та тестування розробленого веб-додатку.

1.3. Моделювання предметної області

Моделювання предметної області є ключовим етапом проєктування програмного забезпечення, оскільки саме на цьому етапі розробник формалізує структуру об’єктів, зв’язків та бізнес-процесів, що мають бути відображені у системі. Метою моделювання є створення абстрактного, але точного представлення тих понять і дій, які визначають функціонування системи в реальному середовищі [6]. Це дозволяє сформувати структуроване уявлення про функціонування предметної області, на основі якого у подальшому проєктується інформаційна та програмна частини системи.

У випадку веб-системи продажу електроніки предметна область охоплює низку процесів: формування асортименту, організація каталогу товарів, робота з користувачами, управління замовленнями, обробка оплати, облік доступності товарів, зворотний зв'язок тощо. Ці процеси реалізуються через відповідні програмні компоненти, які мають бути логічно пов’язані між собою й відповідати реальним бізнес-процесам. Таке моделювання дозволяє розробнику створити систему, яка відповідає бізнес-логіці та задовольняє вимоги кінцевих користувачів [9].

Одним із важливих завдань на цьому етапі є розділення системи на окремі концептуальні області відповідальності. Це дозволяє чітко визначити ролі учасників, взаємодію між ними, життєвий цикл об’єктів у системі, а також передбачити розширення функціоналу в майбутньому. Для цього доцільно використовувати різні підходи до моделювання, зокрема побудову концептуальних схем, діаграм взаємодії, потоків даних тощо.

У ході моделювання враховуються вимоги до гнучкості, масштабованості та розширюваності системи. Саме тому для побудови архітектури використовується принцип доменного розмежування, який сприяє чіткому відокремленню логіки предметної області від технічних аспектів реалізації (збереження даних, передачі запитів тощо). Такий підхід дозволяє зосередитись на суті бізнес-процесів, не обтяжуючи їх інфраструктурними деталями.

Важливим моментом є забезпечення незалежності моделей предметної області від технологій. У процесі моделювання застосовуються концепції розділення відповідальності, що відповідає принципам чистої архітектури Роберта Мартіна. Згідно з нею, моделі предметної області мають бути незалежними від зовнішніх технологій і фреймворків. Це дозволяє досягти високого рівня масштабованості та повторного використання компонентів системи [3].

Узагальнення моделювання предметної області у даному проєкті сприяє побудові системи, яка не лише відображає ключові процеси електронної торгівлі, але й здатна адаптуватися до змін у вимогах, масштабах, ролях користувачів та умовах використання.

1.4. Огляд інформаційних джерел та існуючих рішень

Розробка програмного забезпечення для веб-системи продажу електроніки вимагає попереднього ознайомлення з уже існуючими рішеннями у цій галузі. Аналіз аналогічних проєктів дає змогу виявити загальноприйняті функціональні моделі, знайти сильні та слабкі сторони реалізацій конкурентів, а також краще зрозуміти очікування користувачів щодо зручності, швидкодії та логіки інтерфейсу. Крім того, огляд сучасних технічних підходів дозволяє спроєктувати систему, яка буде відповідати поточним тенденціям у галузі розробки програмного забезпечення.

Серед найбільш відомих рішень на ринку України можна виокремити такі онлайн-магазини:

- Rozetka — один із найпопулярніших та наймасштабніших маркетплейсів, що забезпечує широкий функціонал: каталог продукції, фільтрацію товарів, особистий кабінет, систему відгуків, порівняння товарів, корзину, оплату онлайн та доставку. Архітектурно платформа поєднує класичну клієнт-серверну модель із великою кількістю інтеграцій;
- Comfy — спеціалізований онлайн-магазин побутової техніки та

електроніки, що пропонує зручний фільтр товарів, адаптивний інтерфейс, підтримку кількох типів замовлення (доставка, самовивіз), а також базову авторизацію користувача;

- Foxtrot — мережа з офлайн- і онлайн-продажем електроніки. Основний акцент зроблено на зручну навігацію, простоту інтерфейсу й потужний механізм знижок та акцій;
- Allo — електронний магазин з акцентом на мобільні телефони та ґаджети, що забезпечує широкі функціональні можливості: фільтрацію за категоріями, відгуки, рейтинг товарів, лояльність для зареєстрованих користувачів.

Аналізуючи ці системи, можна зробити висновок, що важливою складовою успішного інтернет-магазину є:

- інтуїтивно зрозумілий інтерфейс користувача;
- швидкий доступ до великої кількості товарів;
- потужна система пошуку й фільтрації;
- персоналізовані сервіси (кабінет користувача, історія замовлень, списки бажань);
- підтримка різних методів оплати та доставки;
- адаптивний дизайн для різних пристроїв.

Щодо інформаційних джерел, під час написання даного проєкту використовувались такі типи ресурсів:навчальні матеріали з програмування на платформах ASP.NET Core MVC, офіційна документація Microsoft (.NET, Entity Framework Core); книги з архітектури ПЗ, зокрема «Чиста архітектура» Роберта Мартіна, що значною мірою вплинула на вибір архітектурного стилю системи; онлайн-курси та статті, які висвітлюють сучасні практики побудови веб-додатків, патерни проектування, забезпечення безпеки, розгортання і тестування;репозиторії з відкритим кодом (GitHub), де аналізувались подібні системи для вивчення практичних рішень у побудові електронної комерції.

У сукупності аналіз існуючих рішень і вивчення відповідної теоретичної бази забезпечили належне обґрунтування архітектурних і функціональних

рішень у рамках розробки власного програмного продукту.

1.5. Постановка завдання

Розробка веб-системи продажу електроніки передбачає створення повноцінного інструменту для здійснення електронної комерції в межах предметної області роздрібної торгівлі електронікою та комплектуючими до неї. Зростання конкуренції в цьому секторі зумовлює необхідність створення таких рішень, які не тільки задовольняють базові вимоги покупців, але й забезпечують зручність, швидкість та масштабованість сервісу.

В основі постановки завдання лежить необхідність побудови централізованої веб-системи, яка дозволяє адміністраторам керувати товарами та категоріями, а користувачам — переглядати, фільтрувати й купувати товари. Система повинна мати чітку архітектуру, бути зручною для супроводу та масштабування, забезпечувати безпечну авторизацію й автентифікацію, і відповідати загальноприйнятим вимогам до сучасних веб-додатків.

Загальна мета — розробити веб-застосунок з використанням ASP.NET Core MVC у межах архітектурного підходу «Чиста архітектура» Роберта Мартіна. Така структура забезпечує розділення відповідальностей, гнучкість модифікацій та полегшує підтримку коду.

На підставі проведеного аналізу можна визначити основні функціональні та нефункціональні вимоги до системи, згруповані у таблиці:

Таблиця 1.1

Функціональних та нефункціональних вимог до системи

Тип вимог	Опис
Функціональні	
Авторизація	Реєстрація та вхід користувачів з ролями (користувач, адміністратор)
Каталог товарів	Перегляд списку товарів з фільтрацією за категоріями, підкатегоріями, брендами
Картка товару	Відображення опису, ціни, фото та характеристик окремого товару
Кошик	Додавання/видалення товарів у кошик, формування замовлення
Адмін-панель	Управління товарами, категоріями, підкатегоріями через зручний інтерфейс
Нефункціональні	
Адаптивність	Коректне відображення на мобільних пристроях, планшетах і десктопах
Безпека	Захист даних користувача, захист адміністративних дій
Масштабованість	Можливість подальшого розширення функціоналу без зміни основної архітектури
Продуктивність	Мінімізація часу завантаження сторінок, ефективна робота при великій кількості користувачів
Технологічність	Впровадження перевірених шаблонів проєктування, модульності та повторного використання коду

Таким чином, у процесі розробки має бути реалізовано ядро функціональності електронного магазину, яке можна буде згодом розширити без шкоди для загальної структури системи. Враховуючи вимоги до якості ПЗ, архітектура, вибрані технології й інтерфейс мають бути підпорядковані ідеї створення надійного та зручного сервісу.

2. Проєктування інформаційного та програмного забезпечення

2.1. Логічна модель даних у вигляді ER-діаграми

Логічне проєктування бази даних є критичним етапом при створенні інформаційної системи. На цьому етапі формується уявлення про структуру даних, їх взаємозв’язки та правила цілісності. Метою цього процесу є створення такої моделі, яка точно відображає вимоги предметної області й у подальшому стане основою для реалізації фізичної бази даних.

Для побудови логічної моделі даних було використано інструмент ERwin Data Modeler, який дозволяє створювати Entity-Relationship (ER) діаграми з урахуванням типів зв’язків та атрибутів сутностей.

У результаті моделювання було виділено ключові сутності системи, які відображають основні об’єкти предметної області: користувачі, товари, замовлення, кошики, списки бажаного, а також супутні сутності, що забезпечують підтримку бізнес-процесів, зокрема доставку. Кожна з цих сутностей має набір властивостей (атрибутів), які визначають її унікальність, логіку функціонування та взаємодію з іншими сутностями.

На рисунку 1 зображено ER-діаграму даної системи

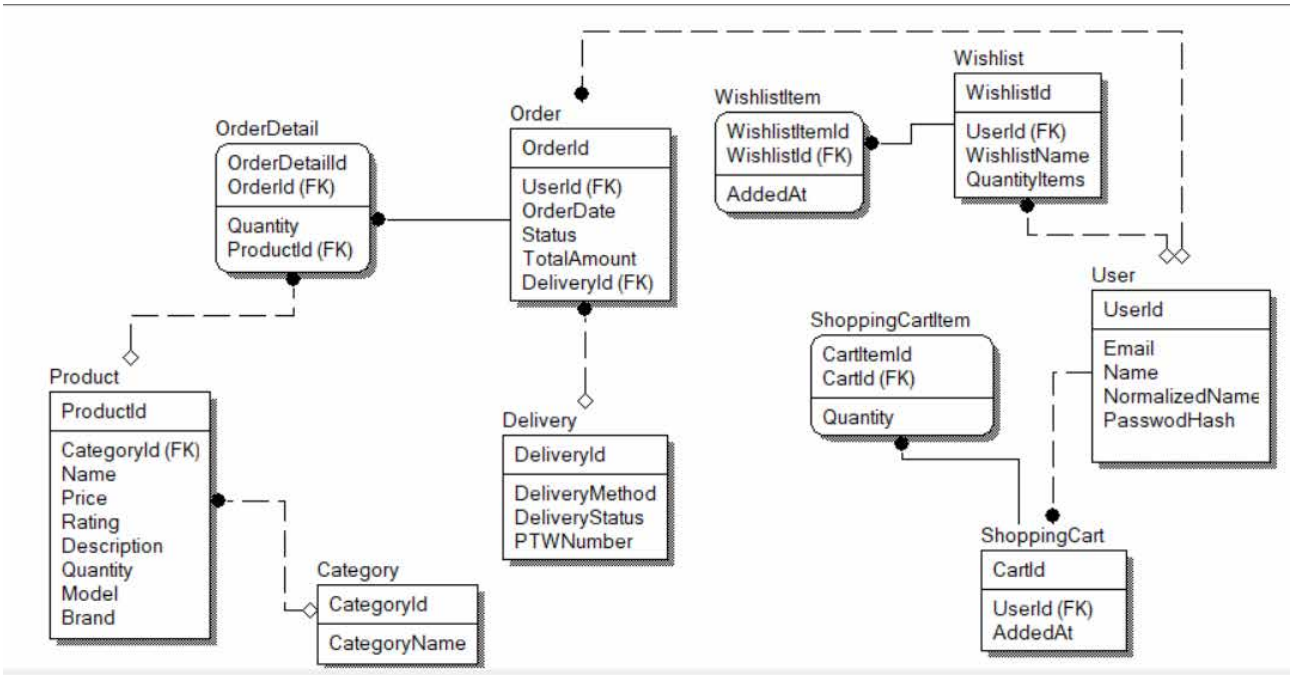


Рис. 1 – ER-діаграма системи

Нижче подано узагальнену таблицю, що описує основні сутності та типи їхніх зв’язків у логічній моделі:

Таблиця 2.1

Сутності та їх описи

Сутність	Ключова роль у системі	Основні атрибути	Зв'язки з іншими сутностями
Users	Представляє зареєстрованого користувача	Email, Name, PasswordHash	Має: кошики, списки бажаного, замовлення
Category	Відповідає за класифікацію товарів	CategoryName	Пов’язана з товарами: один-до-багатьох
Product	Основний об’єкт продажу	Name, Price, Rating, Description, Quantity	Пов’язаний з категорією, кошиками, замовленнями, списками бажаного
ShoppingCart	Представляє кошик користувача	AddedAt	Прив’язаний до користувача, містить елементи
ShoppingCartItem	Описує товар у кошику користувача	Quantity	Належить до кошика, містить посилання на товар
Wishlist	Дає змогу користувачу зберігати обрані товари	WishlistName, QuantityItems	Належить до користувача, містить елементи
WishlistItem	Описує окремий товар у списку бажаного	AddedAt	Зв’язок між Wishlist і Product

Delivery	Містить інформацію про способи та статуси доставки	DeliveryMethod, DeliveryStatus, PTWNumber	Пов’язана із замовленнями
Orders	Відповідає за підтверджені покупки користувача	OrderDate, Status, TotalAmount	Зв’язана з користувачем, доставкою, деталями замовлення
OrderDetail	Представляє конкретну позицію в замовленні	Quantity	Належить до замовлення, містить посилання на товар

У контексті логічного проєктування між сутностями використовуються такі типи зв’язків:

- Один-до-багатьох (1:N) – наприклад, один користувач може мати кілька замовлень, кошиків або списків бажаного.
- Багато-до-багатьох (N:M) – неявно реалізовано через проміжні таблиці, як-от ShoppingCartItem, WishlistItem або OrderDetail.
- Один-до-одного (1:1) – фактично не застосовується безпосередньо, однак можливі варіації у подальшій деталізації (наприклад, профіль користувача).

Зв’язки між сутностями побудовано з дотриманням вимог референтної цілісності: у разі видалення головного об’єкта дочірні об’єкти також видаляються (каскадне видалення), що забезпечує консистентність бази даних.

Логічна модель, відображена через ER-діаграму, стала важливою основою для реалізації схеми бази даних. Вона дала змогу впорядкувати об'єкти предметної області, чітко визначити їхню взаємодію, уникнути надлишковості даних та підготувати ґрунт для побудови ефективної, масштабованої та підтримуваної структури зберігання даних.

2.2. Діаграма класів та кооперацій

2.2.1. Діаграма класів

У процесі проєктування об’єктно-орієнтованої структури програмного забезпечення важливо сформувати діаграму класів, яка ілюструє ключові сутності предметної області, їхні атрибути, методи та взаємозв’язки. Така діаграма є основою для подальшої реалізації системи в межах парадигми об’єктно-орієнтованого програмування, забезпечуючи візуалізацію структури системи на концептуальному рівні [6]. Вона дозволяє розробнику краще зрозуміти логіку бізнес-процесів, знизити складність розробки та забезпечити можливість повторного використання коду.

У розробленій системі продажу електроніки класова модель була створена з урахуванням адаптації логічної моделі даних до об'єктно-орієнтованого середовища. Деякі сутності, що на рівні ER-діаграми подані окремо, у класовій моделі були об'єднані, що дозволило уникнути надмірної деталізації та дублювання логіки. Такий підхід відповідає принципу абстрагування, рекомендованому в об’єктно-орієнтованому аналізі [10].

Зокрема, у діаграмі класів були реалізовані такі класи:

- Користувач (User) — модель, що містить ключову інформацію про зареєстрованих користувачів системи, включаючи ім’я, електронну пошту та захешований пароль. Цей клас асоціюється з усіма функціями, які виконуються від імені користувача, зокрема взаємодією з товарами, замовленнями та списками улюбленого.
- Категорія (Category) — містить назву категорії та асоціюється з кількома товарами. Категорії використовуються для зручної навігації й фільтрації продукції.
- Продукт (Product) — основна бізнес-сутність, яка включає назву, опис, ціну, рейтинг, кількість, модель та бренд. Цей клас пов’язаний із категорією та бере участь в операціях, пов’язаних із замовленнями,

кошиком та улюбленим.

- Замовлення (Order) — агрегований клас, який об’єднує замовлення й деталі замовлення. Він містить інформацію про дату, статус, загальну суму й доставку. Через агрегацію реалізовано зв’язок із продуктами, які входять до складу замовлення.
- Кошик (ShoppingCart) — ще один агрегований клас, що поєднує основну сутність кошика та його елементи. Зберігає інформацію про продукти, які користувач додав перед оформленням замовлення.
- Улюблене (Wishlist) — агрегує список бажаних товарів, який користувач може створити. Зв’язується з класом продуктів.
- Доставка (Delivery) — окрема сутність, що описує спосіб доставки, статус та унікальний номер ТТН. Використовується в межах класу «Замовлення».

З метою забезпечення відповідності вимогам інкапсуляції, усі класи містять закриті поля з відкритими гетерами/сеттерами, або ж автоматичні властивості, залежно від реалізації.

Зв’язки між класами подані у вигляді асоціацій, агрегацій або композицій, залежно від характеру взаємозалежностей між об’єктами. Кожен клас було розроблено з дотриманням принципів SOLID, що дозволяє гарантувати хорошу масштабованість та модифікованість проєкту[3].

На рисунку 2 представлено діаграму класів даної системи.

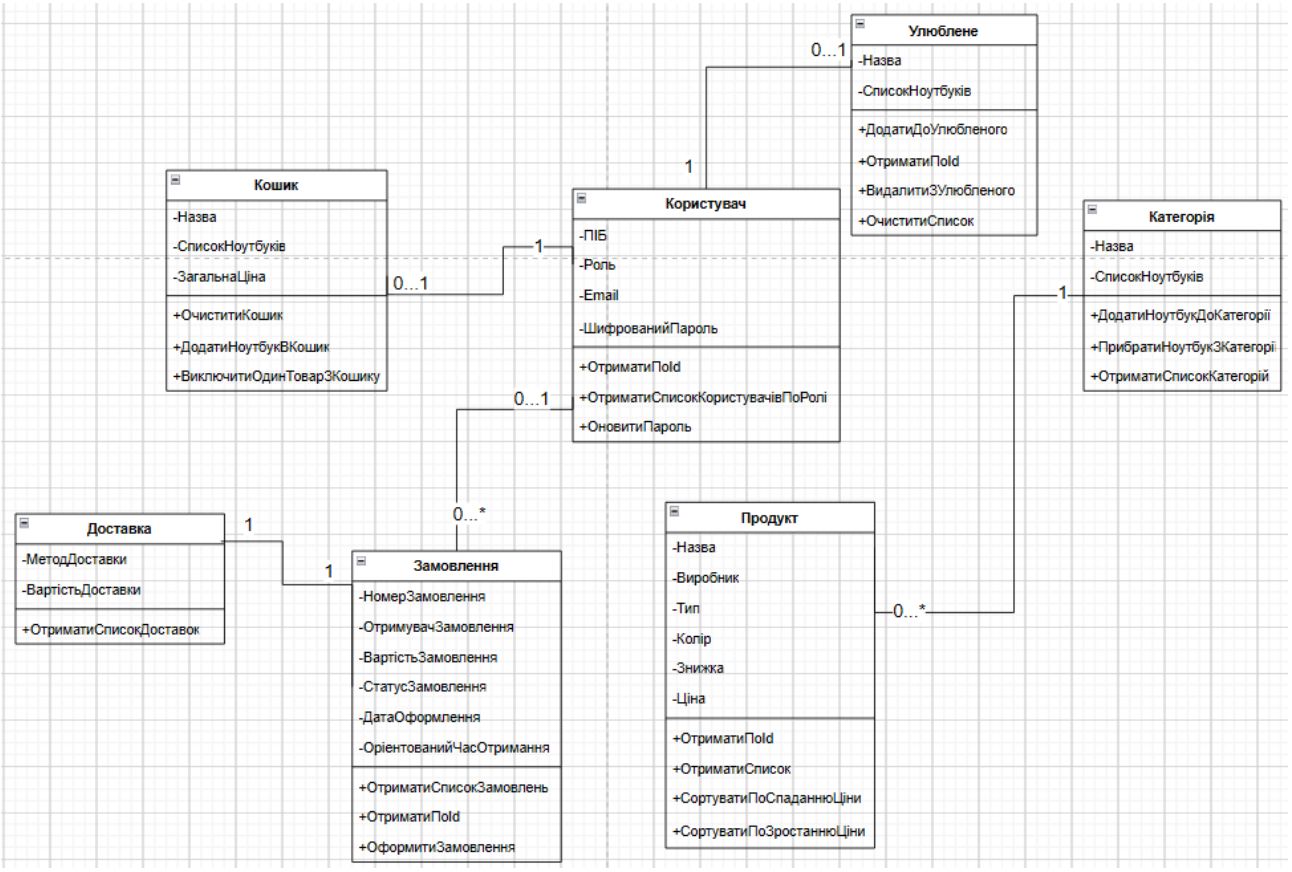


Рис. 2 – Діаграма класів

2.2.2. Діаграма кооперацій

Діаграми кооперації (cooperation diagrams, також відомі як communication diagrams) є важливим інструментом моделювання взаємодії об’єктів у межах конкретних сценаріїв використання системи. Вони зосереджені на зображенні об'єктів та повідомлень, які передаються між ними, показуючи як логіка взаємодії реалізується у відповідь на дії користувача. Основна перевага таких діаграм полягає в їхній здатності поєднувати структурні та поведінкові аспекти системи, забезпечуючи краще розуміння архітектури виконання.

На відміну від діаграм послідовностей, де головний акцент зроблено на часову послідовність подій, діаграми кооперації акцентують увагу на просторовому розміщенні об’єктів і маршруті передачі повідомлень. Це особливо корисно для виявлення прихованих залежностей між компонентами та візуалізації ролей об'єктів у контексті виконання окремих бізнес-процесів.

У даному проєкті було побудовано три діаграми кооперації(подані у

«Додатку А»), які охоплюють ключові функціональні сценарії взаємодії користувача з системою:

1. Управління улюбленим та кошиком

Ця діаграма моделює процеси, пов’язані з взаємодією користувача із системами збереження товарів для подальшого розгляду та формування списку покупок.

Вона зображає:

- додавання товару до переліку «Улюблене» за ініціативою користувача;
- перенесення обраного товару до кошика;
- оновлення кількості товарів у кошику;
- взаємодію між об'єктами: Користувач, Улюблене, Кошик, Продукт.

Цей сценарій дозволяє прослідкувати логіку дій користувача на етапі ознайомлення з асортиментом і підготовки до покупки, а також проєктне визначити точки обробки подій для динамічного оновлення відповідних колекцій.

2. Управління замовленнями та доставкою

Друга діаграма охоплює логіку, що реалізується під час оформлення замовлення, його обробки та відстеження стану доставки.

Зокрема, вона демонструє: ініціацію замовлення користувачем після заповнення кошика; формування замовлення в системі з обчисленням загальної вартості; прив'язку до методу доставки; оновлення статусу доставки (відправлено, доставлено тощо); взаємодію між об'єктами: Користувач, Замовлення, Доставка.

Цей сценарій допомагає моделювати повний цикл обслуговування замовлення — від моменту підтвердження покупки до її фізичного отримання. У такий спосіб забезпечується проєктне уявлення про реалізацію транзакційного механізму та зв’язку з логістичним блоком.

3. Управління категоріями

Остання діаграма охоплює сценарій, пов’язаний із адмініструванням товарного каталогу, що є частиною функціоналу адміністратора або менеджера

контенту.

Вона включає: додавання нової категорії товарів; наповнення категорії продуктами; редагування або видалення елементів каталогу; взаємодію між об'єктами: Користувач, Категорія, Продукт.

Дана діаграма ілюструє, як користувач з відповідними повноваженнями взаємодіє з моделями даних для формування структури інтернет-магазину. Це дозволяє забезпечити централізоване керування пропозицією, адаптацію каталогу до змін у асортименті, а також формалізувати обробку відповідних операцій у межах системи.

Загалом, діаграми кооперації в цьому проєкті сприяють кращому розумінню бізнес-логіки, уточнюють сценарії використання, і дозволяють ефективніше реалізовувати поведінкові вимоги в архітектурі системи. Вони також дають змогу наочно виявити зв’язки між об’єктами, що не завжди очевидні на статичних діаграмах, таких як ER-модель чи діаграма класів.

2.3. Діаграма пакетів

Діаграма пакетів (package diagram) є одним із типів структурних діаграм у нотації UML та використовується для візуального відображення високорівневої організації програмного забезпечення. Вона дозволяє структурувати систему у вигляді логічно впорядкованих модулів (пакетів), які об’єднують пов’язані між собою класи, інтерфейси, сервіси, репозиторії, контролери тощо. Це дає змогу розробникам краще орієнтуватися в архітектурі проєкту, спрощує супровід, масштабування та повторне використання коду.

У контексті розробленої веб-системи продажу електроніки, яка побудована відповідно до принципів "Чистої архітектури" Роберта Мартіна, діаграма пакетів відіграє надзвичайно важливу роль. Завдяки суворому поділу системи на шари (рівні абстракції), забезпечується інверсія залежностей — один із ключових принципів SOLID. Цей підхід дозволяє уникнути жорсткої зв’язаності між компонентами та забезпечити гнучкість у зміні окремих частин

системи.

Діаграма пакетів для розробленої системи демонструє поділ на такі основні пакети:

- Domain (Доменна модель) — ядро системи, що містить сутності та інтерфейси (контракти). Тут немає залежностей від зовнішніх технологій або фреймворків. Це дозволяє відокремити бізнес-логіку від реалізацій.
- Application (Застосунковий рівень) — реалізує бізнес-процеси через сервіси, які взаємодіють із доменними сутностями та оголошеними інтерфейсами. У цьому пакеті можуть бути реалізовані DTO-об’єкти, мапери, сервіси, сценарії використання тощо.
- Infrastructure (Інфраструктурний рівень) — включає реалізації інтерфейсів для доступу до бази даних, обробки файлів, надсилання електронної пошти тощо. Саме тут зосереджена взаємодія з зовнішніми системами.
- WebUI (Презентаційний рівень) — відповідає за взаємодію з користувачем. У цьому пакеті зосереджено MVC-компоненти: контролери, подання, моделі представлення (ViewModel), JavaScript-файли, CSS, статичні ресурси тощо.

Таке логічне групування сприяє: зниженню складності супроводу системи; уникненню циклічних залежностей; покращенню тестованості за рахунок інтерфейсного розділення; гнучкій заміні або оновленню окремих компонентів системи без порушення загальної архітектури.

У результаті, діаграма пакетів не лише візуалізує структуру системи, але й допомагає дотримуватися принципів чистої архітектури впродовж усього життєвого циклу проєкту. Вона є важливим інструментом для спілкування між учасниками команди, зокрема між архітекторами, розробниками та тестувальниками.

2.4. Діаграма компонентів

Діаграма компонентів (Component Diagram) відображає фізичну структуру

програмної системи через компоненти (модулі) та їхні залежності. Такий тип діаграми дозволяє проєктувальнику системи зрозуміти, як побудовані взаємозв’язки між різними частинами програми на рівні шарів архітектури, а також визначити, які саме елементи реалізують бізнес-логіку, інтерфейси взаємодії та доступ до даних.

У розробленій системі інтернет-магазину електроніки структура логічно поділена на чотири архітектурні шари. Вони описані на діаграмі компонентів, яка демонструє, як саме ці шари взаємодіють між собою. Такий підхід спрощує розуміння архітектури як розробникам, так і стороннім учасникам проєкту, і дозволяє гнучко розвивати систему.

Основні компоненти, представлені на діаграмі:

1. Web (Веб-шар). Цей компонент відповідає за взаємодію з користувачем. У ньому розташовані контролери, які приймають HTTP-запити, обробляють їх та викликають необхідні сервіси з інших шарів. Контролери напряму взаємодіють із сервісами, реалізованими в Infrastructure, без використання шаблону репозиторіїв. Саме на цьому рівні реалізуються сценарії інтерфейсу користувача та взаємодія з представленнями (Views).
2. Infrastructure (Інфраструктурний шар). Містить сервіси, що реалізують бізнес-функціональність системи. Вони напряму викликаються з Web-шару. Основна роль інфраструктури — зв’язати вхідні запити з логікою застосування: сервіси використовують DTO, які оголошено у Application, для доступу до бази даних сервіси застосовують DbContext на основі Entity Framework, у цьому шарі також реалізовано збереження, оновлення, видалення й отримання даних з бази.
3. Application (Шар застосунку). Містить DTO (Data Transfer Objects), які служать проміжною формою передачі даних між різними шарами. DTO є важливою частиною взаємодії між інфраструктурою і моделями домену. Вони дозволяють відокремити логіку домену від конкретної реалізації зберігання або подання. У цьому шарі не міститься логіки — лише структура даних для переносу.

4. Domain (Доменний шар). Визначає основні сутності (моделі предметної області), які відображають реальні об’єкти системи: Користувач, Продукт, Замовлення, Категорія, тощо. Цей шар є незалежним від зовнішніх технологій, таких як Entity Framework чи HTTP. DTO з Application шару взаємодіють з цими доменними моделями при мапінгу даних.

Взаємозв’язки між компонентами:

- Контролери з Web-шару викликають сервіси з Infrastructure.
- Сервіси з Infrastructure використовують: DTO з Application для обміну даними, DbContext для взаємодії з базою даних, мапінг DTO ↔ Domain model.
- DTO напряму пов’язані з доменними моделями з Domain, дозволяючи адаптувати формат даних до потреб конкретного запиту або відповіді.
- Загальні характеристики:
- Компоненти зв’язані між собою через залежності, але чітко розділені логічно, що полегшує підтримку системи.
- Діаграма не містить зовнішніх бібліотек або API — вся увага зосереджена на внутрішній архітектурі.
- Шари не мають циклічних залежностей, що відповідає принципам багаторівневої архітектури.

Переваги такої структури:

- Забезпечується висока модульність та масштабованість системи.
- Кожен компонент має чітко визначену відповідальність.
- Спрощується тестування, адже можна ізолювати будь-який шар.
- Полегшується підтримка й розширення функціональності.
- Реалізовано принцип розділення обов’язків (SoC).

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Система управління інформаційною базою

Вибір типу СУБД

У межах розробки інтернет-магазину було обрано реляційну систему управління базами даних (Relational Database Management System, RDBMS). Це рішення обґрунтоване особливостями структури та збереження даних у подібних системах. Як показано в дослідженнях, реляційна модель оптимально підходить для платформ, що оперують із транзакційними замовленнями, пов’язаними сутностями й великою кількістю довідкових таблиць [10].

Основні причини вибору реляційного типу СУБД:

- Нормалізована структура даних: сутності у базі (на кшталт «Користувач», «Продукт», «Замовлення») мають чітко визначені атрибути та зв’язки, що легко моделюються за допомогою таблиць і зовнішніх ключів.
- Складні транзакції: обробка замовлень, збереження даних кошика, оновлення залишків товару — все це вимагає надійних транзакцій з гарантією цілісності даних.
- Стандартизована мова SQL: забезпечує незалежність від конкретного постачальника БД, зручність створення запитів, звітності, аналітики та фільтрації.
- Ієрархічна логіка бізнес-даних: наприклад, кожен товар належить до підкатегорії, яка належить до категорії — таку багаторівневу структуру ефективно реалізовувати через зовнішні ключі.

Таким чином, саме реляційна модель найкраще відповідає задачам інтернет-магазину, що передбачає глибоку зв’язність між об’єктами, високу узгодженість даних та масштабованість логіки збереження.

Обґрунтування вибору Microsoft SQL Server

З-поміж реляційних систем було обрано Microsoft SQL Server, оскільки вона ідеально інтегрується з технологічним стеком ASP.NET Core MVC та ORM-бібліотекою Entity Framework Core [2]. Такий вибір дозволяє мінімізувати конфігураційні бар’єри, зменшити технічну складність при взаємодії між шарами системи та забезпечити продуктивність і масштабованість [6].

Причини такого вибору:

- Повна сумісність із .NET-стеком. Проєкт реалізовано на ASP.NET Core MVC, а для доступу до даних використовується ORM-бібліотека Entity Framework Core. MS SQL є нативною СУБД для цього технологічного стека, що гарантує стабільність, максимальну продуктивність та відсутність необхідності використовувати сторонні адаптери.
- Зручна інтеграція в архітектуру проєкту. У проєкті реалізовано багаторівневу структуру (шари Domain, Application, Infrastructure, Web), і MS SQL дозволяє легко створювати інтерфейси взаємодії з базою через контекст даних (DbContext), не ускладнюючи взаємодію між шарами.
- Високий рівень безпеки. MS SQL Server підтримує розширене керування доступом, автентифікацію через Active Directory, шифрування даних, ролі користувачів та інші інструменти для захисту персональної інформації.
- Масштабованість і надійність. MS SQL Server активно використовується у великих комерційних системах. У проєкті інтернет-магазину важливо враховувати перспективу розширення функціоналу, навантаження від користувачів і наявність великої кількості транзакцій — з чим ця СУБД справляється без проблем.
- Наявність розвинених засобів адміністрування. Інструменти на зразок SQL Server Management Studio (SSMS) дозволяють зручно працювати з таблицями, функціями, представленнями, тригерами, а також відслідковувати ефективність запитів, що особливо цінно при оптимізації продуктивності.

Крім того, MS SQL Express є безкоштовною версією, що дає можливість використовувати її без додаткових витрат у рамках локальної розробки, а при

потребі перейти на повну комерційну редакцію — без зміни коду або логіки застосунку.

Порівняння з іншими популярними СУБД

Таблиця 3.1

Критерій	MS SQL Server	MySQL	PostgreSQL	MongoDB (NoSQL)
Тип СУБД	Реляційна (RDBMS)	Реляційна (RDBMS)	Реляційна (RDBMS)	Документоорієнтована (NoSQL)
Підтримка транзакцій	Повна	Повна	Повна	Часткова
Робота з .NET / EF	Ідеальна	Добра (через з’єднувачі)	Добра (через сторонні провайдери)	Обмежена
Продуктивність при складних запитах	Висока	Середня	Висока	Низька (у складних зв’язках)
Підтримка зв’язків	Повна (PK, FK, constraints)	Повна	Повна	Відсутня (денормалізована модель)
Безпека	Розширена	Базова	Розширена	Залежить від конфігурації
Інструменти адміністрування	SSMS, Azure Data Studio	MySQL Workbench	pgAdmin, DBeaver	Compass, Shell
Ліцензія	Пропрієтарна / Express — безкоштовна	Open Source	Open Source	Open Source

Враховуючи архітектуру проєкту, інтеграцію з .NET та потребу в реляційних зв’язках між даними, Microsoft SQL Server є оптимальним вибором

для реалізації інформаційної бази інтернет-магазину. Його гнучкість, надійність та сумісність з технологіями, що використовуються у проєкті, значно спрощують розробку, супровід і масштабування системи.

3.2Розробка інформаційної бази

3.2.1 Вибір підходу до створення бази даних: Code First

У розробці програмного забезпечення важливо не лише спроєктувати правильну логіку програми, а й ефективно організувати зберігання й обробку даних. Одним із ключових рішень у цьому процесі є вибір підходу до створення бази даних. У сучасному середовищі .NET доступні три основні підходи:

- Database First — коли база даних створюється вручну або автоматично за допомогою SQL-скриптів, а потім на основі цієї бази генеруються класи моделі.
- Model First — коли спочатку створюється візуальна модель у дизайнері, з якої потім генеруються як SQL-скрипти, так і класи.
- Code First — коли вся структура моделі описується в коді (у вигляді класів), а база даних створюється автоматично на основі цього опису.

У межах цієї дипломної роботи обрано саме підхід Code First, що відповідає сучасним практикам побудови програмних систем з використанням Domain-Driven Design (DDD) [12]. У цьому підході вся логіка моделювання переноситься в код, і база даних створюється автоматично — що забезпечує цілісність, контроль версій і узгодженість між шарами застосунку.

3.2.2 Причини вибору підходу Code First

1. Підтримка багаторівневої архітектури

У проєкті реалізовано чітке розділення на шари: Domain, Application, Infrastructure, Web. У такій архітектурі доменна модель є центральним елементом, а база даних має лише відображати структуру доменних сутностей.

Code First дозволяє напряму будувати модель у доменному шарі, а база даних лише віддзеркалює структуру цих класів, що повністю відповідає практикам архітектури Clean Architecture та підходу DDD [3].

2. Легкість внесення змін і розширень

Однією з ключових переваг Code First є те, що уся структура моделі описана у вигляді класів, які легко модифікувати. Якщо з'являється потреба додати нову властивість, змінити тип поля або додати зв'язок — достатньо лише змінити відповідний клас і застосувати нову міграцію. Це дозволяє не відволікатися на ручне оновлення схеми БД і мінімізує кількість помилок.

3. Механізм міграцій

EF Core підтримує потужну систему міграцій, яка дозволяє поетапно оновлювати схему бази даних. Це дає змогу зберігати історію змін у системі контролю версій (Git) і зручно розгортати систему на різних середовищах [2]. Міграції зберігаються у вигляді окремих файлів, які можна легко контролювати і розгортати на різних середовищах (розробка, тестування, випуск).

4. Єдність мовного середовища

Весь процес моделювання відбувається в єдиному мовному контексті — у коді C#. Це дозволяє уникати необхідності перемикання між SQL, XML, EDMX або дизайнером баз даних. Програміст працює лише з мовою, яку добре знає, і не витрачає час на синхронізацію моделей між інструментами.

5. Простота командного використання

У випадку командної роботи над проєктом Code First спрощує спільну розробку: всі моделі, зв'язки, обмеження і зміни зберігаються в коді, тому немає проблем із синхронізацією версій бази даних між учасниками. Після отримання оновленого коду достатньо виконати команду update-database, і всі зміни автоматично застосовуються.

6. Висока відповідність принципам Domain-Driven Design (DDD)

DDD вимагає точного відображення предметної області в моделі коду. Code First дозволяє створювати чисті, незалежні від інфраструктури доменні сутності, що визначають логіку системи, а не її спосіб збереження [12].

3.2.3 Вибір технології ORM (Object-Relational Mapping)

У процесі розробки сучасних програмних систем, що працюють із реляційними базами даних, постає необхідність у зручному та ефективному механізмі відображення об’єктів предметної області на таблиці бази даних. Для цього використовуються технології ORM (Object-Relational Mapping), які дозволяють розробнику оперувати сутностями як звичайними об’єктами, не звертаючись безпосередньо до SQL-запитів. Вони дають змогу працювати з сутностями на рівні об'єктно-орієнтованого коду без написання прямого SQL, що значно спрощує розробку та знижує ризики помилок при зміні структури БД [2].

Обрання конкретного ORM-рішення є важливим етапом, що має безпосередній вплив на структуру проєкту, рівень продуктивності, складність підтримки та потенційну масштабованість системи. У межах цього проєкту розглядалися найбільш поширені та підтримувані ORM-технології в середовищі .NET: Entity Framework Core, NHibernate та Dapper. Їхні відмінності проаналізовано за низкою критеріїв: рівень абстракції, підтримка Code First, міграцій, LINQ, складність конфігурації, тощо [13].

Таблиця 3.2

Порівняльна характеристика ORM-технологій

Параметр порівняння	Entity Framework Core	NHibernate	Dapper
Рівень абстракції	Високий	Високий	Низький
Підтримка Code First підходу	Повна	Часткова (переважно Fluent Configuration)	Відсутня
Автоматичне створення міграцій	Так	Так (але складна конфігурація)	Ні
Простота у використанні	Висока	Середня	Висока (але потребує явного SQL)
Продуктивність при великих даних	Середня	Висока	Висока
Підтримка складних зв’язків	Повна	Повна	Обмежена (потрібна ручна реалізація)
Документація та підтримка спільноти	Широка, офіційна підтримка Microsoft	Добра, але менш активна спільнота	Широка, активно підтримується
Гнучкість налаштування	Висока (Fluent API, Data Annotations)	Висока	Низька
Підтримка LINQ-запитів	Повна	Часткова	Відсутня
Типове використання	Повноцінні CRUD-системи	Системи з ускладненими схемами	Окремі запити з високою продуктивністю

3.2.4 Обґрунтування вибору

Зіставивши переваги та недоліки кожної технології, доцільним стало використання Entity Framework Core як основного ORM для реалізації інформаційної бази. Основними аргументами на користь цього вибору є:

- Повна підтримка Code First підходу, що відповідає загальній архітектурі проєкту та забезпечує зручність еволюції схеми бази даних через механізм міграцій;
- Гармонійна інтеграція з платформою .NET, зокрема ASP.NET Core та інструментами розробника Visual Studio;
- Широка документація, наявність численних прикладів та офіційна підтримка з боку Microsoft, що спрощує розробку та забезпечує надійність;
- Гнучке налаштування моделей та зв’язків за допомогою Fluent API або анотацій;
- Можливість зручно працювати із запитами завдяки LINQ, що підвищує читабельність та безпеку коду.

Згідно з дослідженнями, EF Core є найкращим варіантом для створення CRUD-додатків із багаторівневою архітектурою. Його можливості відповідають вимогам практичного застосування в системах із помірною складністю бізнес-логіки, особливо якщо дотримуватися принципів Domain-Driven Design (DDD) [11].

У порівнянні з NHibernate, Entity Framework є легшим у вивченні, краще документованим і менш вибагливим у конфігурації, що є перевагою при розробці дипломного проєкту. Dapper, хоча й показує найкращу продуктивність при виконанні великих обсягів однотипних запитів, не забезпечує повноцінної підтримки об'єктного моделювання та зв’язків, тому його використання обмежується специфічними випадками, а не загальною реалізацією моделі.

З огляду на вищезазначене, Entity Framework Core є найбільш збалансованим та ефективним вибором для побудови інформаційної бази в контексті цього програмного продукту.

3.2.5 Інформаційна база

Інформаційна база є критично важливою частиною прикладної системи, що забезпечує зберігання, доступ і цілісність даних, які стосуються продукції, користувачів, замовлень, категорій товарів та інших сутностей, пов’язаних з функціонуванням онлайн-магазину комп’ютерної техніки. У даному проєкті її реалізовано за допомогою Entity Framework Core із використанням підходу Code First, що дозволяє спочатку створити доменну модель у вигляді C#-класів, а вже на її основі — автоматично згенерувати структуру бази даних [2].

Відображення доменної моделі

Розроблена інформаційна база відображає предметну область, спираючись на логіку бізнес-процесів системи. Кожна сутність у проєкті є репрезентацією реального об’єкта — товару, категорії, підкатегорії, користувача тощо. Наприклад, модель Product відображає основні характеристики товару: назву, опис, ціну, бренд, а також належність до підкатегорії. Така структура підтримує цілісне відображення бізнес-процесів, що повністю відповідає принципам Domain-Driven Design [12].

Налаштування структури даних

У проєкті структура таблиць, зв’язків між ними, а також поведінка при видаленні або оновленні сутностей конфігурується централізовано через перевизначення методу OnModelCreating у класі контексту AppDbContext. Такий підхід дозволяє гнучко налаштувати поведінку моделей без необхідності створення окремих конфігураційних класів, що спрощує підтримку коду на невеликому або середньому проєкті.

Для прикладу, конфігурація сутності Product у методі OnModelCreating може виглядати так:

```
modelBuilder.Entity<Product>(builder =>
{
    builder.HasKey(p => p.Id);
    builder.Property(p => p.Name).IsRequired().HasMaxLength(200);
    builder.Property(p => p.Price).HasColumnType("decimal(18,2)");
    builder.HasOne(p => p.SubCategory)
```

```
            .WithMany(sc => sc.Products)
            .HasForeignKey(p => p.SubCategoryId)
            .OnDelete(DeleteBehavior.Cascade);
});
```

Тут демонструється базова конфігурація: визначення первинного ключа, обов’язкових полів, типу даних для цін, а також зв’язку між товарами та підкатегоріями.

Особливості проєктування інформаційної бази

Проєктуючи інформаційну базу, були враховані такі принципи:

- Нормалізація даних з метою усунення надлишковості та підвищення узгодженості;
- Забезпечення цілісності на рівні бази через зовнішні ключі та каскадні обмеження;
- Логічна ієрархічна структура, що відповідає предметній області (наприклад, категорія → підкатегорія → товар);
- Розширюваність, що забезпечує легке додавання нових полів або сутностей без необхідності перебудови всієї схеми.

Усі поля, що зберігають часові мітки (CreatedAt, UpdatedAt), конфігуруються із вказівкою значення за замовчуванням у вигляді SQL-функції GETUTCDATE(). Це дозволяє зберігати коректні часові дані навіть у випадку взаємодії з базою ззовні (наприклад, при міграції чи зовнішньому імпорті даних).

Цілісність та ефективність

Завдяки строгій конфігурації в методі OnModelCreating забезпечується:

- Контроль над усіма аспектами структури БД (типи даних, обмеження, індекси, поведінка при видаленні);
- Відсутність дублікації логіки при масштабуванні структури;
- Можливість автоматичного створення міграцій, які відображають зміни в класах моделі.

Цей підхід дозволяє уникнути помилок, пов’язаних із розбіжностями між кодом і базою даних, що особливо критично при підтримці проєкту в

довгостроковій перспективі.

Розроблена інформаційна база є невід’ємною складовою прикладного програмного забезпечення та спроєктована відповідно до сучасних принципів гнучкої архітектури. Завдяки використанню централізованої конфігурації моделей у методі OnModelCreating досягнуто високого рівня узгодженості, прозорості й керованості структури зберігання даних. Такий підхід забезпечує масштабованість системи та її готовність до подальшого розширення функціоналу.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

3.3.1 Вибір архітектури програмного забезпечення

Архітектура програмного забезпечення є одним із ключових аспектів при створенні складних прикладних систем. Вона визначає структуру застосунку, спосіб поділу на модулі, залежності між компонентами, їхню відповідальність та можливості масштабування, тестування і подальшого супроводу. Саме архітектурний підхід визначає поділ системи на модулі, залежності між компонентами та рівень модульності, що має прямий вплив на ефективність розробки і подальший супровід [3].

У розробленому програмному забезпеченні реалізовано монолітну архітектуру із дотриманням принципів Clean Architecture, у якій веб-шар побудований на основі MVC-підходу (Model-View-Controller). Подібна структура рекомендується в офіційній документації ASP.NET Core як базова для корпоративних веб-рішень [2]. Така модель організації дозволяє ефективно структурувати проєкт, забезпечуючи суворе розділення обов’язків між рівнями: представлення (View), керування (Controller) та логіка (Model), а також ізольованість бізнес-логіки від деталей реалізації (інфраструктури, технологій тощо).

У структурі застосунку виділено кілька логічних шарів:

- Domain (доменний шар) — містить сутності, інваріанти, інтерфейси, які не мають залежностей від зовнішніх технологій;
- Application (застосунковий шар) — координує бізнес-операції, працює з сервісами, DTO, інтерфейсами репозиторіїв;
- Infrastructure (інфраструктурний шар) — реалізує залежності: взаємодію з базами даних через ORM (Entity Framework), збереження файлів, логування тощо;
- Web (презентаційний шар) — реалізує вхідну точку взаємодії з

користувачем. Побудований за принципами MVC.

Ця модель дає змогу змінювати інфраструктурні компоненти (ORM, базу, інтерфейс користувача) без впливу на бізнес-логіку. Вона також повністю узгоджується з підходом Domain-Driven Design, що передбачає виділення логіки предметної області в окрему, незалежну від інфраструктури частину [12].

Монолітна структура означає, що всі компоненти зібрані в єдиному виконуваному середовищі (єдиній збірці), але дотримання принципів чистої архітектури дозволяє зберігати чіткий розподіл обов’язків та високий рівень модульності. Важливо наголосити, що така архітектура може реалізовуватись не тільки як «класичний моноліт», а й у формі модульного моноліту — це модель, у якій система фізично розгортається як єдине ціле, але логічно структурована на незалежні, ізольовані функціональні модулі. Саме цей підхід фактично реалізований у проєкті, оскільки між модулями (напр., управління товарами, категоріями, користувачами) чітко визначені межі, і взаємодія здійснюється через абстракції.

Таблиця 3.3

Порівняльна таблиця архітектурних стилів

Критерій	Моноліт (чистий)	Модульний моноліт	Класичний моноліт	Layered (шарова)	Microservices
Поділ на шари	Так (Domain, App, Infra, Web)	Так (з модулями)	Часто відсутній	Так, формальний	Повний, сервісний
Межі відповідальності	Чіткі, формалізовані	Чіткі, логічно ізольовані	Розмиті	Частково визначені	Чітко визначені
Фізична структура	Єдина збірка	Єдина збірка	Єдина збірка	Єдина збірка	Кілька незалежних сервісів
Тестованість	Висока	Висока	Низька	Середня	Висока
Масштабованість логіки	В межах проєкту	В межах модулів	Обмежена	Помірна	Висока
Гнучкість розвитку	Висока	Висока	Низька	Середня	Висока
Складність початкової реалізації	Середня	Середня	Низька	Середня	Висока

Залежності між компонентами	Інверсія залежностей	Мінімальні між модулями	Високі, тісно пов’язані	Часто циклічні	Відсутні
Придатність для командної роботи	Висока	Висока	Обмежена	Помірна	Висока

Аналіз показує, що модульний моноліт з використанням принципів чистої архітектури забезпечує структурованість і гнучкість без надмірної складності, притаманної мікросервісам. На відміну від класичного моноліту або формальної шарової архітектури, цей підхід дозволяє створити масштабовану та підтримувану систему, де кожен функціональний блок може розвиватись ізольовано, при цьому зберігаючи переваги централізованого розгортання (відсутність мережевої взаємодії між сервісами, проста стратегія розгортання, спільна база даних).

Така архітектура ідеально підходить для дипломного проєкту, де важливо продемонструвати інженерну культуру, архітектурне мислення і вміння працювати з сучасними підходами, але без перевантаження проєкту інфраструктурною складністю.

3.3.2 Інструментарій розробника: Visual Studio, C# та ASP.NET

У процесі створення прикладного програмного забезпечення важливу роль відіграє вибір середовища розробки та мов програмування, що прямо впливають на продуктивність праці, стабільність коду, масштабованість рішень і швидкість впровадження нових функцій.

Visual Studio як середовище розробки

У проєкті використано Visual Studio 2022 — одне з найпотужніших IDE для розробки .NET-додатків. Ця платформа забезпечує глибоку інтеграцію з .NET SDK, підтримує керування проєктами, систему контролю версій (Git), автоматичне рефакторинг, налагодження, профілювання продуктивності та

розгортання [2]. Visual Studio також підтримує роботу з Razor Pages, базами даних, тестуванням і міграціями баз даних у середовищі Entity Framework [14]. Його стабільність та глибока інтеграція з ASP.NET робить його оптимальним вибором для розробки сучасних веб-систем на .NET платформі.

Мова програмування C#

У якості основної мови програмування використовувано **C#** — об'єктно-орієнтовану, типізовану мову з багатою екосистемою і гнучкими засобами для написання надійного, безпечного та читабельного коду. Завдяки підтримці сучасних парадигм програмування (асинхронність через async/await, записи record, лямбда-вирази, pattern matching тощо), C# дозволяє писати зрозумілий та масштабований код, що важливо при роботі над багаторівневими проєктами, як-от цей. Також її синтаксис є строгим, але виразним, що робить її зручною для проєктів будь-якої складності, включно з багаторівневими [14].

3.3.2.1 ASP.NET як основна платформа розробки веб-шару

У процесі розробки веб-шару системи було обрано платформу ASP.NET, яка на сьогодні є одним із найпотужніших і найгнучкіших фреймворків для створення веб-застосунків. ASP.NET забезпечує високий рівень продуктивності, масштабованість і підтримку сучасних архітектурних підходів, зокрема чистої архітектури з поділом на шари. У реалізації цього проєкту ASP.NET виступає основою для побудови веб-інтерфейсу на основі підходу MVC, де фронтенд реалізується через Razor Pages.

Особливості Razor Pages

Razor Pages — це сучасна модель створення сторінок у ASP.NET, яка поєднує подання (HTML + Razor) із пов’язаною логікою обробки (C#) у єдиному файлі чи структурі (.cshtml + .cshtml.cs). Razor Pages суттєво спрощують розробку невеликих та середніх проєктів, забезпечуючи: зручну прив’язку даних до форм (через Model Binding), розділення відповідальностей між поданням і логікою обробки, інтеграцію з сервісами ASP.NET, такими як DI, Model

Validation, Middleware, оптимізацію маршрутизації, що дозволяє уніфікувати логіку для конкретних сторінок без зайвого дублювання

Цей підхід сприяє кращому контролю над логікою кожної сторінки, що особливо корисно в адміністративних панелях, профілях користувачів, а також при роботі з формами. Також підхід рекомендовано Microsoft як найзручніший для адмін-панелей, профілів користувачів, невеликих веб-додатків або швидкого прототипування [2][14].

HTML, CSS та JavaScript у Razor Pages

Використання Razor Pages не виключає традиційних фронтенд-технологій. Навпаки, HTML, CSS і JavaScript залишаються основними засобами побудови структури, стилів та динамічної поведінки сторінок. Важливо, що Razor синтаксис дозволяє:вбудовувати динамічні дані у HTML прямо зі C# моделей (@Model.Property), формувати умови відображення або цикли для елементів інтерфейсу, працювати з Layout-сторінками для спільних шаблонів (навігація, футери, модальні вікна)

CSS у цьому проєкті організовано у вигляді окремих стилів у файлі site.css, що забезпечує гнучке оформлення і підтримку адаптивного дизайну.

JavaScript і використання fetch API

Для взаємодії із сервером без повного перезавантаження сторінки активно використовується JavaScript, зокрема fetch API. Цей інтерфейс дає змогу:

- Надсилати асинхронні запити до контролерів ASP.NET
- Завантажувати часткові подання (PartialView) динамічно
- Працювати з формами через FormData, не порушуючи MVC-парадигму
- Реалізовувати адаптивні інтерфейси з реакцією на дії користувача (відкриття модальних вікон, фільтрація товарів, оновлення DOM)

Таке розмежування дозволяє зберегти структуру Razor Pages як основної логіки, а також реалізувати багату клієнтську поведінку без використання сторонніх JavaScript-фреймворків.

Переваги обраного підходу

Об’єднання Razor Pages з HTML/CSS/JS дозволяє зберігати високу

швидкість розробки, гнучку структуру, спрощену підтримку коду та розширюваність, що критично важливо для дипломного проєкту. Завдяки цьому платформа забезпечує достатню функціональність навіть без використання SPA-фреймворків на кшталт React чи Angular.

3.3.3 Інструментарій розробника: Entity Framework Core, AutoMapper, Swagger

Під час реалізації програмного забезпечення було обрано низку перевірених інструментів, які забезпечують ефективне управління даними, спрощення логіки відображення, а також сучасну документацію для зручної розробки, налагодження та тестування. Основними компонентами цього технічного стеку стали Entity Framework Core, AutoMapper та Swagger (Swashbuckle). Усі ці компоненти інтегруються в екосистему ASP.NET Core, підтримуються Microsoft і широко застосовуються в корпоративних .NET-рішеннях [2].

Entity Framework Core: керування базою даних через об'єкти

Entity Framework Core — це потужна ORM (Object-Relational Mapping) бібліотека від Microsoft, яка забезпечує можливість працювати з базою даних через об’єкти мовою C# без необхідності написання SQL-запитів. У рамках розробки застосунку EF Core використовується в режимі **Code First**, що дозволяє описувати модель предметної області у вигляді C#-класів (сутностей), а потім автоматично генерувати відповідну схему бази даних[2].

Завдяки використанню Fluent API у методі OnModelCreating розробник отримує повний контроль над конфігурацією зв’язків між таблицями, типами даних, індексами, обмеженнями та іншими аспектами структури бази. Такий підхід забезпечує гнучкість при зміні структури даних і водночас дозволяє підтримувати модель у синхронізації з базою через механізм міграцій.

Крім того, EF Core підтримує LINQ-запити, які транслюються у SQL-інструкції оптимального вигляду. Це дозволяє виконувати складні вибірки, фільтрації, сортування та обчислення безпосередньо на стороні бази даних, не

втрачаючи при цьому «безпеки типів» й зручності мови C#. Такий підхід значно покращує читаємість та підтримуваність коду порівняно з ручним написанням SQL-запитів, особливо в проєктах середнього масштабу, де продуктивність і контроль є критично важливими.

AutoMapper: уніфікація перетворення даних між шарами

AutoMapper є незамінним інструментом у проєктах, де реалізовано чіткий поділ на шари за принципами чистої архітектури. Замість ручного копіювання даних між сутностями, DTO-об’єктами (Data Transfer Objects) та ViewModel-структурами, AutoMapper забезпечує автоматизоване і централізоване прокладення маршрутів моделей. Це дозволяє уникнути повторення коду та мінімізує ймовірність помилок, що виникають при великій кількості трансформаційних операцій[14].

Окремі конфігураційні класи-профілі дозволяють явно описати логіку перетворення між відповідними об'єктами. У проєкті реалізовано два основні профілі: перший — для мапінгу між сутностями та DTO (Application Layer), другий — між DTO та ViewModel (Presentation Layer). Таким чином, дані ніколи не передаються "напряму" між шарами, а завжди проходять через чітко визначені мапінг-процеси, що відповідає принципам інкапсуляції та слабкого зв'язку між компонентами.

Swagger: автоматизована документація та тестування API

Swagger (у вигляді реалізації через бібліотеку Swashbuckle.AspNetCore) був інтегрований у проєкт для автоматизованої генерації специфікацій OpenAPI на основі атрибутів, заданих у контролерах. Хоча основна частина взаємодії у веб-шарі реалізована через Razor-сторінки та формові POST-запити, Swagger виявився надзвичайно корисним для відлагодження внутрішніх API, які використовуються, зокрема, через fetch API у JavaScript.

Swagger дає змогу наочно бачити всі HTTP-ендпоінти, їхні параметри, структуру запитів та відповідей[2]. Це спрощує процес тестування на ранніх етапах, а також дозволяє оперативно перевіряти працездатність серверної логіки без необхідності залучення додаткових інструментів (типу Postman). У

майбутньому, якщо проєкт буде масштабуватися або відкриватиметься для інтеграції з зовнішніми системами, наявність специфікацій OpenAPI дозволить без додаткових витрат часу реалізувати взаємодію з іншими застосунками.

Загалом, обрані інструменти забезпечують стійку, масштабовану та підтримувану архітектуру прикладного програмного забезпечення. Вони дозволяють мінімізувати обсяг "технічного боргу", автоматизувати повторювані операції та дотримуватися принципів архітектурної чистоти без втрати гнучкості.

3.4 Алгоритмізація та програмування програмних модулів

3.4.1 Опис алгоритму «Реєстрація користувача в системі»

Реєстрація нового користувача у системі реалізована шляхом класичної взаємодії між формою Razor (у частковому поданні) та контролером, який, у свою чергу, викликає відповідний сервіс для виконання бізнес-логіки. У цій реалізації не застосовуються JavaScript або асинхронні запити через fetch API: уся передача даних виконується традиційним HTTP POST-запитом, з подальшим повним оновленням сторінки після обробки.

На клієнтському рівні інтерфейс реєстрації реалізовано за допомогою модального вікна Bootstrap, вміст якого визначено у файлі _SignUpPartial.cshtml. У вікні розміщено HTML-форму, сформовану через Html.BeginForm, що забезпечує автоматичну генерацію POST-запиту на метод SignUp контролера Auth. У формі присутні три обов’язкові поля: повне ім’я (FullName), адреса електронної пошти (Email) та пароль (Password). Кожне з них має відповідну обгортку у вигляді form-floating, що сприяє зручному відображенню підписів до полів. Додатково, для забезпечення захисту від CSRF-атак, у форму включено @Html.AntiForgeryToken() — стандартний механізм безпеки ASP.NET MVC.

Форма містить перевірку на наявність помилок у моделі. Якщо дані користувача є некоректними або не відповідають валідаційним вимогам, вони не передаються до сервісу, і відповідні повідомлення виводяться на інтерфейсі. Це

забезпечується через перевірку ViewData.ModelState["SignUp"], яка може містити помилки, встановлені контролером. Вони виводяться у вигляді червоного блоку Bootstrap-типу alert-danger. Таким чином, помилки передаються через класичний механізм ModelState, без використання динамічного JavaScript-оброблення.

Після натискання кнопки “Зареєструватися” браузер надсилає дані за допомогою HTTP POST до методу SignUp у контролері AuthController. На цьому рівні реалізується основна серверна логіка первинної обробки. Контролер перевіряє коректність отриманої моделі: якщо вона не проходить валідацію, наприклад, через порожні поля або неправильний формат електронної адреси, користувач перенаправляється назад на головну сторінку (Home/Index), а у TempData зберігається повідомлення про помилку. Аналогічним чином формується реакція у разі помилок під час створення користувача.

Якщо модель є коректною, контролер викликає метод SignUpAsync сервісу AuthService, який реалізує бізнес-логіку. Спочатку перевіряється, чи існує вже у системі користувач з наданою адресою електронної пошти. Для цього використовується метод _userManager.FindByEmailAsync(...), що звертається до бази даних через стандартні механізми ASP.NET Identity. Якщо такий користувач вже існує, сервіс повертає об’єкт відповіді AuthenticationResponse із позначкою про помилку, і контролер відповідно перенаправляє користувача назад.

У випадку, коли новий користувач може бути створений, DTO-об’єкт SignUpRequest мапиться на доменну сутність AppUser за допомогою AutoMapper. Далі викликається метод _userManager.CreateAsync(...), який зберігає нового користувача до бази даних із зазначеним паролем. Якщо збереження не вдається, наприклад, через порушення політик безпеки (наприклад, занадто короткий або ненадійний пароль), сервіс повертає список описів помилок.

У разі успішного створення користувача, сервіс додатково генерує JWT-токен, що може бути використано для подальшої авторизації, хоча в даній реалізації він не передається у відповідь або cookies. Після цього сервіс повертає

успішну відповідь, а контролер, у свою чергу, перенаправляє користувача на сторінку його профілю (Profile/Index), де може відображатися базова інформація.

Уся взаємодія між клієнтом і сервером реалізована у межах парадигми “submit-redirect”, що є типовим для традиційних MVC-додатків. Кожне відправлення форми призводить до повного оновлення сторінки, а результат дій користувача — успіх або помилка — передається через механізми TempData або ViewData.ModelState.

У цілому, реалізація реєстрації відповідає стандартам побудови безпечних форм у ASP.NET MVC: вона забезпечує повну серверну валідацію, обробку помилок, захист від CSRF, а також застосовує шаблон “контролер → сервіс → база даних” для чіткого розділення обов’язків. Незважаючи на відсутність асинхронної JavaScript-взаємодії, така реалізація є надійною та легко підтримуваною у рамках загальної архітектури проєкту.

Блок-схема дії алгоритму представлена на рис.3. А інші блок-схеми наведені у додатку Б.

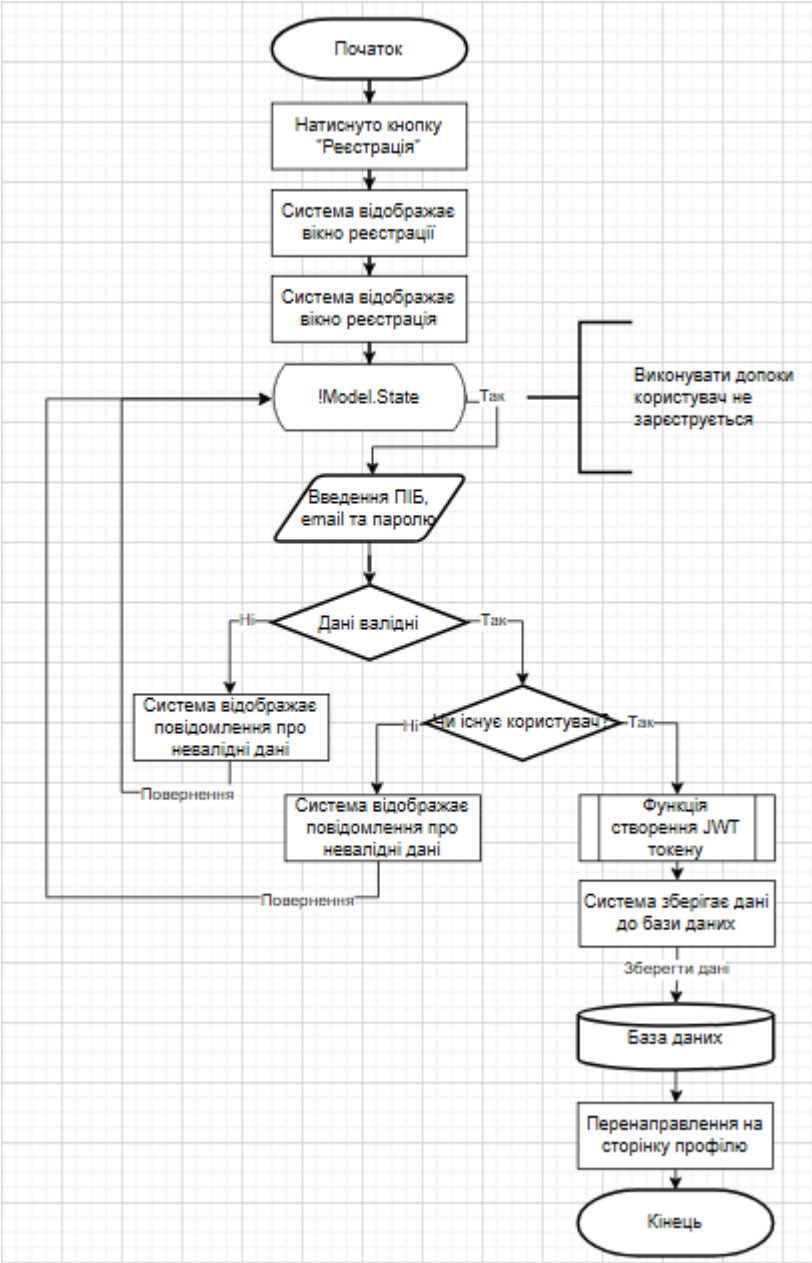


Рис. 3 – Блок-схема алгоритму «Реєстрація користувача в системі»

3.4.2 Опис алгоритму «Авторизації користувача у системі»

Процес авторизації користувача в системі реалізовано за допомогою класичної MVC-моделі, яка передбачає передачу даних з клієнтської HTML-форми до серверного контролера через HTTP POST-запит. Авторизаційна форма розміщена у частковому поданні _SignInPartial.cshtml і, подібно до форми реєстрації, інтегрована до головного макета через Bootstrap-модальне вікно.

Користувач взаємодіє з формою входу, яка містить два основних поля — адресу електронної пошти та пароль. Обидва поля є обов’язковими для

заповнення, що забезпечується атрибутом required. Для візуального оформлення полів використано Bootstrap-компонент form-floating, який дозволяє зручно відображати підписи до полів. У разі помилок валідації або некоректних даних, на інтерфейсі може бути виведене повідомлення про помилку у вигляді червоного блоку alert-danger, сформованого за допомогою механізму ModelState.

Коли користувач натискає кнопку “Вхід”, форма надсилає POST-запит до методу SignIn у AuthController. Першим етапом обробки на сервері є перевірка коректності вхідної моделі. Якщо дані не відповідають вимогам (наприклад, пусті поля, некоректний формат електронної адреси), контролер встановлює у TempData повідомлення про помилку (SignInErrors) та назву модального вікна (authModal), яке потрібно повторно відкрити. Після цього користувача перенаправляють назад на головну сторінку (Home/Index). Цей підхід дозволяє повторно показати модальне вікно входу з уже виведеним повідомленням про помилку.

Якщо модель є валідною, контролер передає запит до сервісу AuthService, який реалізує основну бізнес-логіку авторизації. Спочатку сервіс виконує пошук користувача за вказаною електронною адресою через _userManager.FindByEmailAsync(...). Якщо користувача не знайдено, повертається об’єкт AuthenticationResponse з відповідним повідомленням про помилку.

У разі, якщо користувач існує, сервіс викликає метод _signInManager.PasswordSignInAsync(...), який перевіряє правильність введеного пароля. Якщо автентифікація неуспішна (наприклад, введено неправильний пароль), повертається стандартне повідомлення про недійсні облікові дані.

У випадку успішного входу, сервіс генерує JWT-токен через приватний метод GenerateJwtTokenAsync, який створює підписаний маркер з інформацією про користувача. Даний токен повертається до контролера в об’єкті AuthenticationResponse.

Контролер отримує JWT-токен і зберігає його у cookie з назвою access-token, використовуючи об’єкт Response.Cookies.Append(...). Додатково

застосовуються параметри безпеки: cookie є доступним лише через HTTP (HttpOnly), працює тільки на HTTPS (Secure) і має політику ізоляції SameSite=Strict. Термін дії cookie визначається значенням, отриманим із сервісу через GetJwtExpirationMinutes(), і за замовчуванням становить 60 хвилин.
Після цього користувач перенаправляється на сторінку свого профілю (Profile/Index). Завдяки збереженому токену, система може ідентифікувати авторизованого користувача протягом усього періоду дії маркера, зберігаючи контекст сесії.

Таким чином, уся логіка авторизації користувача побудована за чіткою схемою: валідація → перевірка існування користувача → автентифікація → генерація токена → збереження у cookie → перенаправлення. Такий підхід дозволяє ефективно контролювати процес входу, обробляти помилки через механізм TempData, забезпечувати безпечне зберігання токена і підтримувати інтеграцію із стандартними механізмами ASP.NET Identity без використання JavaScript чи асинхронних API.

3.4.3 Опис алгоритму «Збереження змін при редагуванні категорій»

Алгоритм збереження змін при редагуванні категорій у адміністративній панелі побудований на основі розділення обов’язків між представленням (представленим через Razor-часткове подання), клієнтською логікою (JavaScript) та серверною логікою, реалізованою через контролер AdminPanelController та сервіс ProductService. Основною метою цього алгоритму є забезпечення можливості редагування назв категорій та підкатегорій без перезавантаження сторінки, із подальшим збереженням змін у базі даних.

Початкове відображення списку категорій та їхніх підкатегорій здійснюється через часткове подання _ManageCategoriesPartial.cshtml, яке завантажується у відповідну вкладку сторінки AdminPanel/Index. Це часткове подання приймає модель типу ManageCategoriesVM, яка містить список об'єктів CategoryItemVM, кожен з яких включає ідентифікатор, назву та список

вкладених підкатегорій (SubCategoryItemVM). Для кожної категорії та підкатегорії виводяться окремі текстові поля (<input>), які за замовчуванням мають атрибут readonly, щоб уникнути ненавмисного редагування.

Користувач може активувати редагування конкретної категорії або підкатегорії, натиснувши кнопку з класом editCategoryBtn. Ця кнопка активує JavaScript-обробник подій у файлі edit-categories.js, який знімає атрибут readonly з відповідних полів вводу в DOM-структурі, а також додає до DOM кнопку збереження (saveCategoryBtn) та кнопку видалення підкатегорії (removeSubCategoryBtn), що з'являється тільки в режимі редагування. Таким чином, користувач може змінити назву категорії або її підкатегорій без потреби перезавантаження сторінки або переходу на іншу вкладку.

Після внесення змін користувач натискає кнопку збереження змін (наприклад, .saveChangesBtn, яка присутня на панелі керування). Ця подія перехоплюється JavaScript-скриптом, який здійснює зчитування всіх даних, що були змінені в DOM. Оскільки проєкт не використовує JSON-формат для передачі даних, скрипт створює об'єкт FormData, у якому формуються пари ключ–значення у вигляді стандартного HTML-представлення, що сумісне з прив’язкою даних у MVC-контролері. Структура FormData дозволяє інкапсулювати складну вкладену структуру категорій із відповідними індексами, що узгоджуються з моделлю ManageCategoriesVM у контролері.

Запит відправляється через метод fetch на дію SaveCategories у AdminPanelController. Ця дія приймає параметр ManageCategoriesVM categoriesVM, який завдяки узгодженим іменам полів автоматично зв’язується з даними, що були передані з клієнта. Таким чином, на рівні контролера повністю відтворюється ієрархічна структура: список категорій, кожна з яких має власний список підкатегорій, причому збережено як їхні ідентифікатори, так і оновлені назви.

Після отримання цієї моделі контролер передає її до методу SaveCategoriesAsync у сервісі ProductService, де реалізовано основну бізнес-логіку. Всередині цього методу спочатку виконується зчитування поточних

даних з бази (через AppDbContext) для усіх категорій і підкатегорій, представлених у ViewModel. Для кожної категорії сервіс виконує порівняння між старими та новими назвами. Якщо назва змінилася — відповідний запис оновлюється, і фіксується зміна. Так само аналізуються всі вкладені підкатегорії: якщо змінилась назва підкатегорії або якщо вона була видалена користувачем (через видалення в DOM), то сервіс або оновлює, або видаляє відповідний запис у базі даних.

Після обробки всіх змін і виклику context.SaveChangesAsync() відбувається збереження змін у базі. У разі успішного виконання операції, контролер виконує редирект назад на AdminPanel/Index, що забезпечує оновлене відображення всіх категорій і підкатегорій. Завдяки такому підходу забезпечується цілісність даних, ефективна взаємодія між інтерфейсом та бізнес-логікою, а також гнучкість у подальшій модифікації структури даних (наприклад, додавання нових атрибутів до категорій).

Узагальнюючи, алгоритм збереження змін при редагуванні категорій охоплює усі ключові рівні MVC-архітектури: подання (ManageCategoriesPartial.cshtml) забезпечує інтерактивний інтерфейс; JavaScript (edit-categories.js) — клієнтську логіку збору та передачі даних; контролер (AdminPanelController) — координацію запиту; а сервіс (ProductService) — обробку та збереження змін у базі. Така багаторівнева реалізація дозволяє легко масштабувати систему, розширювати функціональність і підтримувати високу продуктивність навіть при роботі з великою кількістю категорій та підкатегорій.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування є невід'ємним етапом життєвого циклу розробки програмного забезпечення, метою якого є виявлення помилок, невідповідностей вимогам користувача та забезпечення належної якості функціоналу системи. У загальному випадку тестування дозволяє перевірити, наскільки реалізована система відповідає функціональним і нефункціональним вимогам, закладеним на етапах проєктування [15]. Саме завдяки тестуванню можна мінімізувати ризики появи критичних помилок на етапі експлуатації та забезпечити стабільну роботу програмного продукту.

Існує кілька видів тестування, які умовно поділяються за критеріями рівня деталізації, способу виконання та цільового призначення. Найбільш поширені з них:

- Модульне тестування (Unit Testing) — перевіряє окремі функції або методи в ізоляції, дозволяючи виявити помилки ще до інтеграції. У .NET-проєктах це реалізується за допомогою MSTest, xUnit або NUnit [2].
- Інтеграційне тестування (Integration Testing) — перевіряє взаємодію між окремими модулями системи, наприклад, між сервісами та контролерами.
- Системне тестування (System Testing) — охоплює перевірку всієї системи як єдиного цілого. Забезпечує контроль за відповідністю до вимог замовника.
- Функціональне тестування (Functional Testing) — спрямоване на перевірку функціоналу програми відповідно до заданих специфікацій.
- Тестування інтерфейсу користувача (UI Testing) — перевіряє коректність відображення та взаємодії елементів інтерфейсу.
- Регресійне тестування (Regression Testing) — здійснюється після змін у системі, щоб переконатися, що новий код не спричинив збоїв у вже

працездатному функціоналі.

- Ручне тестування (Manual Testing) — полягає у послідовному виконанні дій користувача вручну з метою виявлення помилок.
- Автоматизоване тестування (Automated Testing) — застосовується для автоматичного запуску сценаріїв тестування, зокрема при CI/CD.

У межах створення інформаційної системи для електронної торгівлі тестування відігравало важливу роль на кожному етапі розробки. З огляду на обмеженість у часових та ресурсних рамках, основна увага була зосереджена на ручному функціональному тестуванні інтерфейсу користувача, а також інтеграційному тестуванні взаємодії між основними компонентами архітектури: контролерами, сервісами та базою даних [15].

Ручне тестування проводилося безпосередньо у браузері шляхом проходження типових сценаріїв використання, зокрема:

- реєстрація та авторизація користувача;
- додавання, редагування та видалення товарів (з боку адміністратора);
- перегляд товарів за категоріями та підкатегоріями;
- оформлення фільтрації та перегляду детальної інформації;
- динамічне оновлення категорій у модальному інтерфейсі;
- завантаження зображень товарів та їх відображення в картках.

З технічної точки зору перевірялася також коректність передачі даних між рівнями — зокрема, відповідність ViewModel та DTO, стабільність роботи AutoMapper, правильна прив’язка FormData у запитах, а також надійність логіки збереження в базу даних через Entity Framework Core.

Інтеграційне тестування дозволило переконатися у тому, що взаємодія між компонентами (наприклад, AdminPanelController ↔ ProductService ↔ AppDbContext) відбувається згідно з очікуваннями, без втрати даних та помилок у логіці обробки. Було особливо важливо переконатися, що операції з категоріями та продуктами є узгодженими та зберігають структурну цілісність бази даних.

Таким чином, тестування виступило важливою частиною розробки

системи, дозволивши на практиці виявити і усунути низку логічних та технічних помилок, забезпечити стабільну роботу функціоналу та підготувати систему до етапу експлуатації. Також було забезпечено узгодженість передачі даних і збереження структури у відповідності до моделі предметної області, що особливо важливо при використанні Entity Framework Core [14].

4.1.1 Приклади тест-кейсів

Для перевірки працездатності основних компонентів інформаційної системи було проведено ручне функціональне тестування за допомогою набору тест-кейсів. Тест-кейс — це формалізований сценарій перевірки певної функціональності, який містить вхідні дані, кроки виконання, очікуваний та фактичний результат. Основна мета полягає у виявленні логічних, інтерфейсних або функціональних помилок у системі.

У ході тестування було охоплено основні функціональні модулі, зокрема: реєстрація користувача, автентифікація, адміністративне керування категоріями, підкатегоріями й товарами, робота із завантаженням зображень, динамічна взаємодія між формами та навігація на основі фільтрації. Результати тестування узагальнено в таблиці нижче:

Таблиця 4.1

Тест кейси та їх результати

№	Назва тесту	Вхідні дані	Кроки виконання	Очікуваний результат
1	Реєстрація нового користувача	Ім’я, email, пароль	Перейти до модального вікна реєстрації → заповнити поля → натиснути "Зареєструватися"	Користувач створений, перенаправлення до особистого кабінету
2	Вхід із правильними обліковими даними	Email: існуючий, Пароль: правильний	Відкрити форму входу → ввести облікові дані → натиснути "Увійти"	Перехід до профілю, динамічна зміна стану інтерфейсу
3	Вхід з неправильним паролем	Email: існуючий, Пароль: неправильний	Ввести дані → натиснути "Увійти"	Відображення повідомлення про помилку авторизації
4	Створення категорії у адмін-панелі	Назва категорії	Перейти до "Категорії" → натиснути "Додати категорію" → ввести дані → "Зберегти"	Категорія з’являється в таблиці, оновлення View
5	Додавання підкатегорії	Назва підкатегорії, категорія	Обрати категорію → ввести назву підкатегорії → "Додати"	Підкатегорія додається у вкладену таблицю

6	Редагування назви підкатегорії	Нова назва	Натиснути "Редагувати" → внести зміни → "Зберегти зміни"	Назва підкатегорії оновлюється без перезавантаження сторінки
7	Додавання нового товару	Назва, опис, ціна, підкатегорія, зображення	Увійти як адмін → перейти до вкладки "Товари" → натиснути "Додати"	Новий товар з’являється в таблиці товарів
8	Валідація форми при порожніх обов’язкових полях	Порожнє поле назви	Ввести лише частину даних → натиснути "Зберегти"	Виведення повідомлення про обов’язковість заповнення поля
9	Завантаження файлу з некоректним форматом	Зображення формату .exe	Обрати файл → натиснути "Зберегти"	Відображення повідомлення про помилку завантаження
10	Оновлення списку підкатегорій динамічно	Зміна категорії	Вибрати іншу категорію у формі товару	Список підкатегорій змінюється без перезавантаження сторінки
11	Відображення товарів на головній сторінці	-	Перейти на сторінку /Product/Index	Відображаються картки товарів з анімацією наведення

12	Фільтрація товарів за підкатегорією	Клік по підкатегорії в Layout	Натиснути на пункт меню категорії або підкатегорії	Відображаються тільки товари обраної підкатегорії

Цей розширений набір тест-кейсів охоплює як позитивні сценарії (успішне додавання, реєстрація, вхід), так і негативні (валидаційні помилки, некоректні дані, порожні поля). Значна частина тестів стосується взаємодії з адміністративною частиною системи, оскільки саме вона є найбільш динамічною і функціонально навантаженою.

Кожен тест-кейс був виконаний вручну під час різних етапів розробки з метою забезпечення поступового впровадження змін та уникнення регресійних помилок. Успішне проходження тест-кейсів стало підтвердженням готовності системи до практичного використання.

4.1.2 Тестові ситуації

Розглянуті нижче тестові ситуації описують складніші випадки експлуатації, які охоплюють декілька функціональних модулів системи, включаючи роботу з базою даних, інтерактивні інтерфейси, взаємодію між користувачами різних ролей, перевірку обробки помилок, перевірку валідації, а також поведінку інтерфейсу при некоректному або агресивному введенні.

Ситуація 1: створення некоректного товару, обхід валідації та відмова у збереженні.

Адміністратор додає новий товар через модальне вікно, але вводить у полі "Ціна" негативне значення та залишає поле "Назва" порожнім. Він натискає кнопку "Зберегти" й очікує, що дані не буде прийнято. Додатково він намагається вручну змінити HTML-код сторінки (через інструменти розробника), щоб зняти обмеження на ці поля.

Очікуваний результат: серверна валідація не пропускає запит, навіть якщо клієнтська перевірка обійдена. Користувач отримує повідомлення про помилку,

збереження не відбувається, а форма залишається відкритою з позначеними некоректними полями.

Ситуація 2: взаємодія адміністраторських CRUD-функцій з відображенням на головній сторінці.

Адміністратор додає нову підкатегорію до існуючої категорії, створює кілька товарів, завантажує для кожного відповідні зображення, перевіряє, як ці товари з'являються на публічній сторінці з картками товарів. Потім адміністратор змінює назву підкатегорії та перевіряє, чи оновилася ця інформація в інтерфейсі.

Очікуваний результат: усі створені товари прив’язані до нової підкатегорії, їх можна знайти через фільтрацію, зміна назви підкатегорії одразу відображається у випадаючих списках і картках без необхідності ручного оновлення кешу або сторінки.

Ситуація 3: виявлення колізії під час редагування кількома адміністраторами.

Два адміністратори одночасно відкривають адмін-панель. Один редагує товар і натискає "Зберегти", інший редагує той самий товар через 10 секунд і також намагається зберегти свою версію.

Очікуваний результат: система виявляє колізію та попереджає другого адміністратора, що товар було змінено іншим користувачем. Впроваджено елементарний контроль версій або мітку часу останнього редагування.

Ситуація 4: вплив зміни структури категорій на вже існуючі товари.

Адміністратор видаляє підкатегорію, до якої було прив’язано декілька товарів. Потім відкриває список товарів, щоб пересвідчитися, чи ці записи не порушують цілісність таблиці.

Очікуваний результат: перед видаленням система попереджає про наявні зв'язки, й не дозволяє видалити підкатегорію без перенесення або видалення відповідних товарів. Дані не пошкоджені, усі об'єкти залишаються узгодженими.

Ситуація 5: Редагування профілю користувача з частковим оновленням даних.

Користувач переходить до особистого кабінету, відкриває вкладку редагування та змінює лише номер телефону, залишаючи інші поля без змін. Система має оновити тільки вказане поле.

Очікуваний результат: після збереження змін в базу передається тільки модифіковане значення, інші дані залишаються без змін. Відповідь сервера підтверджує часткове оновлення. Користувач отримує повідомлення про успішне збереження.

Можна стверджувати, що один з найбільш показових сценаріїв — це інтеграційна перевірка "від адміністративної панелі до відображення", яка дозволяє переконатися, що всі зміни в системі одразу і коректно транслюються в публічну частину сайту. Впровадження подібних тестів є важливою частиною забезпечення цілісності системи та мінімізації людських помилок на рівні адміністрування.

4.2 Вимоги до апаратного та програмного забезпечення

Для забезпечення ефективного функціонування розробленої системи необхідно визначити вимоги до апаратного та програмного забезпечення як з боку сервера, так і з боку клієнтських пристроїв. Враховуючи використання сучасного веб-стека з підтримкою Razor Pages, Entity Framework, Bootstrap та інтерактивного JavaScript, вимоги залишаються відносно помірними, однак повинні забезпечувати стабільну продуктивність при розширенні обсягу даних та навантаження на систему [2].

Зокрема, серверна частина системи повинна мати ресурси для обробки запитів, доступу до бази даних, обробки зображень, а також підтримки кількох одночасних адміністративних або користувацьких сесій. За практичними порадами, навіть для невеликих веб-додатків на базі ASP.NET Core з Razor Pages рекомендовані мінімальні характеристики: 2 ядра CPU, 4–8 GB RAM, SSD та встановлений .NET SDK. На стороні клієнта (користувача) достатньо мати сучасний браузер та стабільний інтернет[].

Нижче наведено узагальнені вимоги у табличному форматі:

Таблиця 4.2

Вимоги до апаратного та програмного забезпечення

Категорія	Апаратне забезпечення	Програмне забезпечення
Сервер (хостинг)	- CPU: 2 ядра (наприклад, Intel Xeon або AMD EPYC) - RAM: 4–8 GB - SSD: 50+ GB	- ОС: Windows Server 2019 / Linux (Ubuntu Server) - .NET SDK 7.0 або вище - IIS / Nginx - MS SQL Server / PostgreSQL
Клієнт (користувач)	- CPU: 2 ядра - RAM: 2 GB - Дисплей: ≥ 1280x720 - Інтернет-підключення ≥ 2 Мбіт/с	- Сучасний браузер: Chrome, Firefox, Edge, Safari - ОС: Windows, Linux, macOS, Android, iOS
Розробник / Адміністратор	- CPU: 4 ядра - RAM: 8+ GB - SSD: 100 GB	- Visual Studio 2022 / JetBrains Rider - .NET SDK 7.0+ - Git - SQL Server Management Studio - Postman / браузер з DevTools

Додаткові зауваження.

Безпека та надійність сервера. Рекомендується використовувати SSL-сертифікати, регулярні резервні копії та моніторинг доступності [2].

Масштабування. У разі зростання кількості користувачів можливо використання віртуалізації або хмарних сервісів з автоматичним масштабуванням (наприклад, Azure App Service або AWS Elastic Beanstalk) [16].

Сумісність. Уся система розроблена з дотриманням принципів кросбраузерності та адаптивного дизайну, тому може працювати на більшості

сучасних пристроїв [14].

Таким чином, система є досить гнучкою до середовища запуску, що дозволяє її розгортання як на локальному сервері, так і в хмарній інфраструктурі, за умови дотримання мінімальних рекомендованих характеристик.

4.3 Склад інсталяційного пакету

Інсталяційний пакет призначений для забезпечення швидкого та безпечного розгортання веб-системи BasedTechStore на стороні користувача або адміністратора системи. Його зміст має бути структурованим, мінімальним і при цьому достатнім для повноцінного запуску застосунку у виробничому середовищі. Усі компоненти пакету формуються з урахуванням вимог до безпеки, сумісності з різними операційними системами, а також зручності використання.

Основним елементом пакету є папка з назвою BasedTechStore, що містить усі необхідні файли для функціонування системи. Усередині цієї папки зберігаються лише ті компоненти, що є безпосередньо потрібними для публічного розгортання:

1. Файли компіляції та зібрані компоненти: Включають згенеровані .dll-файли, необхідні для запуску ASP.NET Core-додатку. Ці файли є результатом компіляції проекту і не потребують наявності вихідного коду на стороні користувача.
2. Папка wwwroot: Містить статичні ресурси — стилі CSS, скрипти JavaScript, зображення, шрифти тощо. Це єдиний публічно доступний каталог, з якого сервер надає доступ до ресурсів користувачу.
3. Файл README.md: Документаційний файл, який містить інструкції щодо розгортання системи, налаштування середовища, оновлення бази даних, запуску серверу та базові вимоги до платформи. Це особливо зручно як для Windows-користувачів, так і для користувачів macOS чи Linux.
4. Файл BasedTechStore.runtimeconfig.json: Містить конфігурацію

середовища запуску .NET — версію середовища виконання та інші необхідні залежності. Цей файл необхідний для правильної ініціалізації додатку на сервері.

5. Файл BasedTechStore.deps.json: Містить перелік усіх залежностей, які потрібні для запуску додатку. Завдяки цьому система динамічно підключає всі необхідні бібліотеки під час виконання.
6. Скрипт запуску для Windows (start.bat): Автоматизує запуск додатку через CLI (наприклад, dotnet BasedTechStore.dll) для операційної системи Windows. Такий скрипт є зручним для адміністраторів, які не мають досвіду роботи з терміналом.
7. Скрипт запуску для Unix-систем (start.sh): Аналогічно до .bat-файлу, цей shell-скрипт забезпечує запуск додатку у середовищах на базі Linux або macOS. Завдяки цьому інсталяційний пакет є кросплатформним.
8. Файл .gitignore (опціонально): Якщо система буде розгортатися через репозиторій, файл .gitignore дозволяє уникнути випадкового коміту небажаних файлів (наприклад, appsettings.Development.json, *.db, *.log тощо).

У рамках формування інсталяційного пакету були дотримані наступні принципи безпеки:

Виключення з пакету вихідного коду та службових директорій. У фінальну збірку не включаються такі елементи, як папка Migrations, конфігураційні файли з обліковими даними (appsettings.json, .env), а також інші внутрішні директорії, які можуть містити чутливу інформацію або структуру бази даних. Таким чином, виключається можливість несанкціонованого доступу до конфігурацій або логіки міграцій.

Анонімізація підключення до БД. У публічному пакеті немає жодної інформації про підключення до бази даних. Адміністратор має самостійно створити необхідні конфігураційні файли після розгортання, дотримуючись інструкцій із README.md.

Можливість контейнеризації. Усі компоненти пакету можуть бути легко

інтегровані в Docker-контейнер. При цьому користувач отримує повну автономність та гнучкість при розгортанні.

Отже, інсталяційний пакет є лаконічною, безпечною та портативною збіркою, яка не містить зайвих або ризикованих компонентів. Його формування базується на принципах мінімізації, кросплатформеності та простоти в користуванні, що дозволяє ефективно використовувати систему в будь-якому сучасному серверному середовищі.

4.3.1. Діаграма розгортання та її зв’язок з інсталяційним пакетом

Діаграма розгортання (Deployment Diagram) є частиною специфікації системи, що дозволяє відобразити фізичне розміщення компонентів програмного забезпечення у цільовому середовищі. Вона ілюструє структуру розгортання програмної системи у вигляді вузлів (серверів або клієнтських пристроїв), які з'єднані між собою мережевими каналами, а також демонструє, які саме артефакти (наприклад, файли, виконувані компоненти, бази даних) розміщуються на кожному з них. Така діаграма дозволяє формалізувати і спростити процес розгортання, обслуговування та масштабування системи.

У випадку веб-застосунку BasedTechStore діаграма розгортання включає три основні вузли:

1. Веб-браузер (Клієнтська частина)

Це пристрій кінцевого користувача (ПК, ноутбук, планшет тощо), який за допомогою браузера ініціює запити до веб-сервера. У цьому сегменті функціонують наступні артефакти:

- HTML5-файли — формують структуру сторінки.
- CSS-стилі — забезпечують візуальне оформлення та адаптивність інтерфейсу.
- JavaScript-файли — відповідають за клієнтську логіку (обробка подій, анімація, запити через Fetch API тощо).

Ці ресурси доставляються з веб-сервера та виконуються безпосередньо у

браузері користувача. Вони не входять до інсталяційного пакету у вигляді окремих файлів, але є результатом збірки проєкту і присутні в директорії /wwwroot.

2. Веб-сервер (Серверна частина)

На цьому вузлі відбувається обробка запитів, реалізація бізнес-логіки, управління маршрутами, доступ до бази даних тощо. Тут розміщується інсталяційний пакет, сформований на основі зібраного проєкту ASP.NET Core MVC.

Склад цього пакета включає:

Файли .dll — скомпільовані бібліотеки, які відповідають за: контролери, моделі даних, сервіси, конфігураційні класи тощо.

Каталог wwwroot/ — містить статичні файли: зображення, CSS, JavaScript.

Файли представлень (*.cshtml) — шаблони Razor для формування сторінок.

Файл appsettings.Production.json (умовно) — конфігураційний файл, що може бути включений під час розгортання, але без чутливих даних (наприклад, рядків підключення).

README.md — текстовий супровід, який містить інструкції для розгортання, залежності та вимоги до оточення.

start.bat або аналогічний скрипт запуску — опціонально використовується для автоматичного запуску у Windows. Для користувачів Linux/macOS інструкція в README.md передбачає запуск через CLI.

Важливо зазначити, що інсталяційний пакет не містить жодних Migrations, .env-файлів чи відкритих конфігурацій із доступом до бази даних — задля безпеки та захисту даних.

3. Сервер бази даних

Цей вузол відповідає за зберігання та обробку даних. Він включає:

- Середовище MS SQL Server — як рушій для роботи із запитами.
- База даних BasedTechStore — створена за допомогою Entity Framework Core на етапі ініціалізації (міграції виконуються лише адміністративно).

Інсталяційний пакет напряму не містить цієї бази, однак README.md містить інструкції для її створення через застосунок, або з попереднього дампу чи скрипту ініціалізації. Підключення до БД виконується на сервері згідно з рядком підключення, який задається через змінні середовища або конфігураційні файли, що не входять до загального дистрибутиву.

Діаграму розгортання наведено на рисунку нижче.

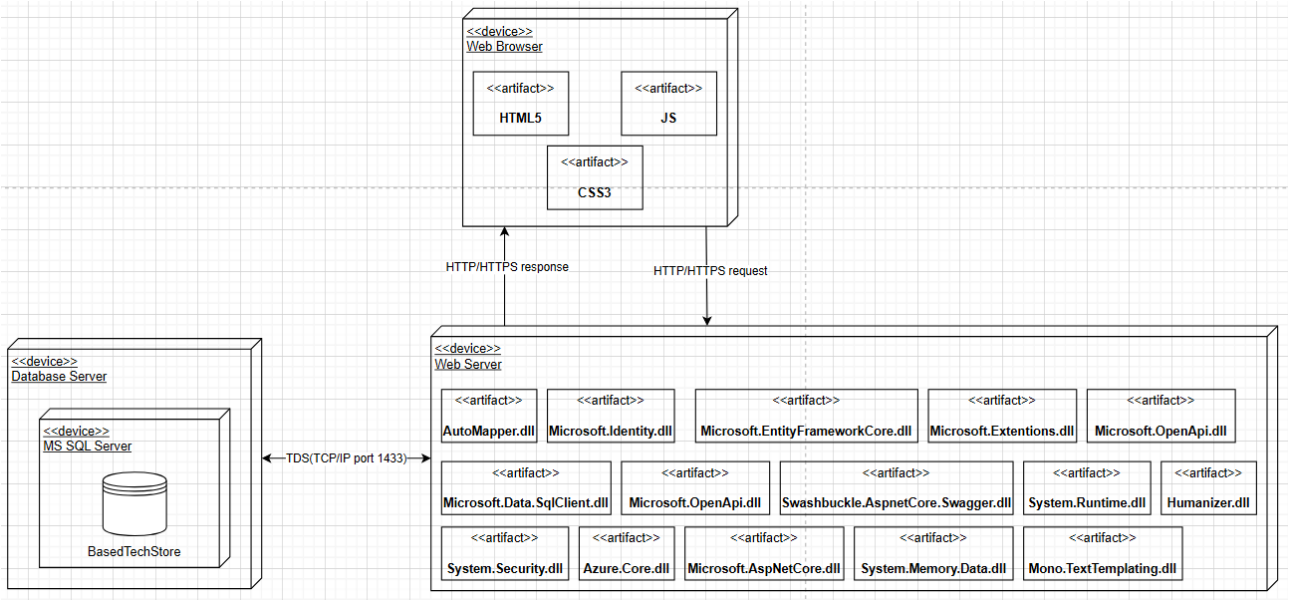


Рис. 4 – Діаграма розгортання системи

Таким чином, діаграма розгортання не лише ілюструє компоненти та їх взаємозв’язки, а й допомагає чітко зрозуміти, які частини інсталяційного пакету потрапляють на який вузол та яка їхня функціональна роль. Це дозволяє організувати коректний процес розгортання, виключити помилки налаштування та дотримуватись принципів безпеки при публікації системи.

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи досягнуто основної мети дослідження — спроєктовано та реалізовано інформаційну систему, призначену для автоматизації обліку та управління товарами в умовах діяльності онлайн-магазину з продажу електротехніки. Реалізоване програмне забезпечення забезпечує зручну взаємодію користувача з інтерфейсом, ефективне керування товарами, категоріями, підкатегоріями, обробку зображень, а також адміністрування системи зі сторони персоналу.

У першому розділі проведено аналіз предметної області, вивчено особливості роботи сучасних електронних комерційних систем та визначено недоліки типових підходів до управління товарами та взаємодії користувачів. Сформульовано вимоги до функціоналу, а також визначено основні задачі, які мала б вирішувати система. Для моделювання основних процесів та взаємозв'язків було застосовано нотацію UML, зокрема побудовано діаграму випадків використання, яка відображає ключові сценарії взаємодії користувача з системою.

У другому розділі розглянуто питання інформаційного та технічного забезпечення системи. Здійснено побудову концептуальної та логічної моделі бази даних у вигляді ER-діаграми, на основі якої було створено структуру таблиць із дотриманням принципів нормалізації. Розроблено зручну структуру взаємопов’язаних сутностей: продукти, категорії, підкатегорії, користувачі, ролі, замовлення тощо. База даних реалізована за допомогою підходу Code First із використанням технології Entity Framework Core, що дозволяє забезпечити гнучкість у розробці та супроводі системи.

У третьому розділі описано програмну реалізацію системи. Для створення веб-застосунку було використано технології ASP.NET Core MVC у поєднанні з Razor Pages для динамічного відображення контенту. Інтерфейс користувача реалізовано із застосуванням сучасних засобів верстки, зокрема HTML5, CSS3 (включаючи Bootstrap) та JavaScript. Усі стилі згруповано у папці wwwroot/css/, а логіка взаємодії на стороні клієнта за допомогою JS у wwwroot/js/. Для обробки

подій та взаємодії з контролерами використано fetch API(частина базового функціоналу JavaScript), який дозволяє здійснювати асинхронні запити з використанням різних форматів таких як JSON та HTML (можливо доступні і інші, але використовувалися лише ці).

Особливу увагу приділено адаптивному дизайну інтерфейсу, зокрема реалізовано перемикач теми, зручну навігацію та адаптивні елементи для різних типів пристроїв. Розроблено інтерфейс користувача з особистим кабінетом, який дозволяє переглядати та редагувати особисту інформацію. Також створено окремий інтерфейс адміністратора з вкладками для управління товарами та категоріями. Модальні вікна, реалізовані з використанням Bootstrap, забезпечують зручне додавання й редагування даних без необхідності перезавантаження сторінки.

У процесі реалізації функціоналу управління товарами використано AutoMapper для мапінгу між сутностями, DTO та ViewModel, що значно спростило перенесення даних між шарами додатку та зменшило дублювання коду. Для збереження даних реалізовано сервісний рівень (ProductService), який містить усю бізнес-логіку, згідно з принципами розподілу відповідальності. Усі дії адміністратора — додавання, редагування, видалення товарів, категорій та підкатегорій — реалізовано через модальні форми з подальшою валідацією та обробкою на сервері.

У четвертому розділі проведено тестування основних модулів системи з метою перевірки їх коректного функціонування та відповідності заявленим вимогам. Проведено аналіз вимог до апаратного та програмного забезпечення, необхідного для повноцінної роботи застосунку.

Отже, у межах виконаної роботи вдалося комплексно вирішити поставлені задачі, об'єднати на практиці теоретичні знання з програмування, моделювання даних, розробки інтерфейсів та організації взаємодії між компонентами веб-застосунку. Отриманий результат може бути використаний як основа для реального онлайн-магазину або масштабований під інші комерційні проєкти.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

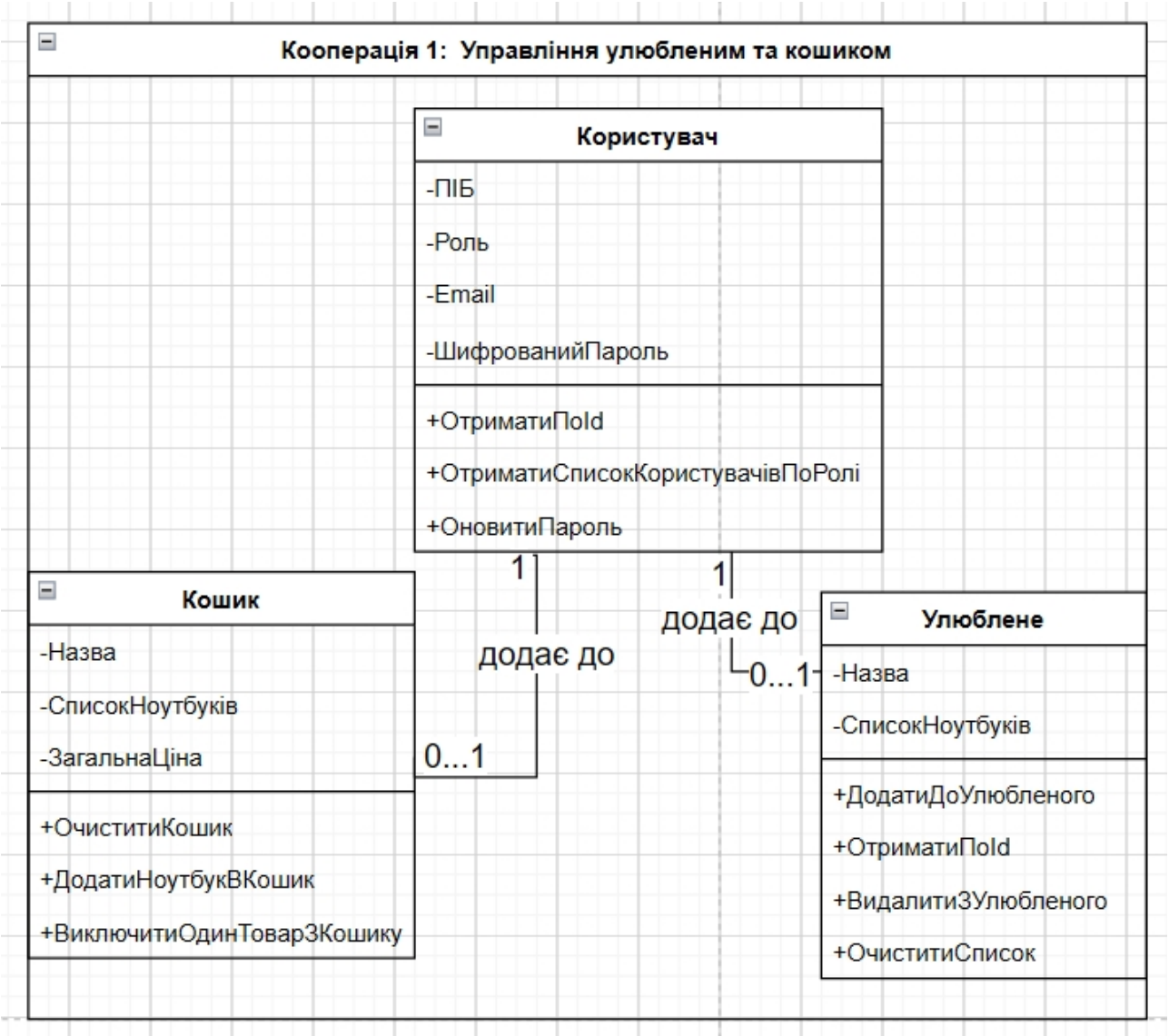
1. Котлер Ф., Армстронг Г. *Основи маркетингу : підручник* / пер. з англ. ; наук. ред. С. В. Воронкова. — К. : Вільямс, 2010. — 1024 с.
2. Microsoft. ASP.NET Core: офіційна документація. URL: https://learn.microsoft.com/aspnet/core/ (дата звернення: 10.05.2025).
3. Мартін Р. *Чиста архітектура. Мистецтво розробки програмного забезпечення* / Роберт С. Мартін ; пер. з англ. — Київ : Наш Формат, 2021. — 336 с.
4. Бідголі Х. *Електронна комерція: принципи та практика*. Сан-Дієго : Academic Press, 2001. — 650 с.
5. Чидійва О. *Аналіз найкращих практик розробки ПЗ для електронної комерції на прикладі методологій Agile та Traditional*. URL: https://uwcscholar.uwc.ac.za/items/d8791c65-426d-41b3-ad0a-d4cd82841d00 (дата звернення: 10.05.2025).
6. Benslimane Y., Cysneiros L. M., Bahli B. Оцінювання критичних функціональних і нефункціональних вимог для веб-систем закупівель. *Requirements Engineering*. 2007. № 12. URL: https://link.springer.com/article/10.1007/s00766-007-0050-4 (дата звернення: 10.05.2025).
7. Yusop N., Zowghi D., Lowe D. Вплив нефункціональних вимог на проєкти веб-систем. *IJVCM*. 2008. № 3(3). URL: https://opus.lib.uts.edu.au/bitstream/10453/10267/1/2007002839.pdf (дата звернення: 10.05.2025).
8. Khatter K., Kalia A. Цільовий аналіз нефункціональних вимог веб-систем. URL: https://www.academia.edu/download/57666720/GoalbasedAnalysisofNon-functionalRequirementsofWeb-basedSystems.pdf (дата звернення: 10.05.2025).
9. Дністров Н. С. Вебсистема управління контентом з використанням Next.js : кваліфікаційна робота. СумДУ, 2024. URL:

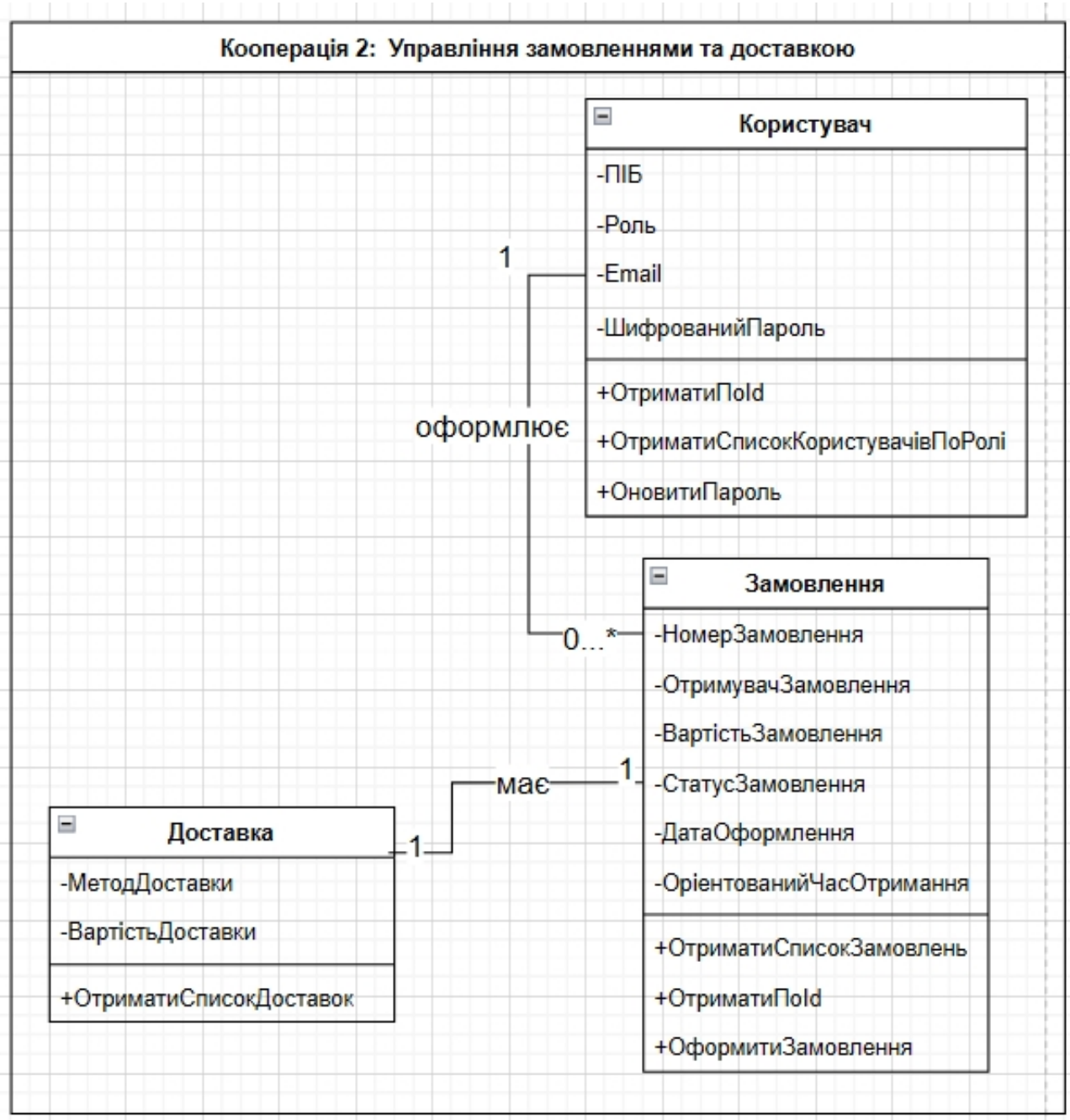
https://essuir.sumdu.edu.ua/handle/123456789/98127 (дата звернення: 11.05.2025).

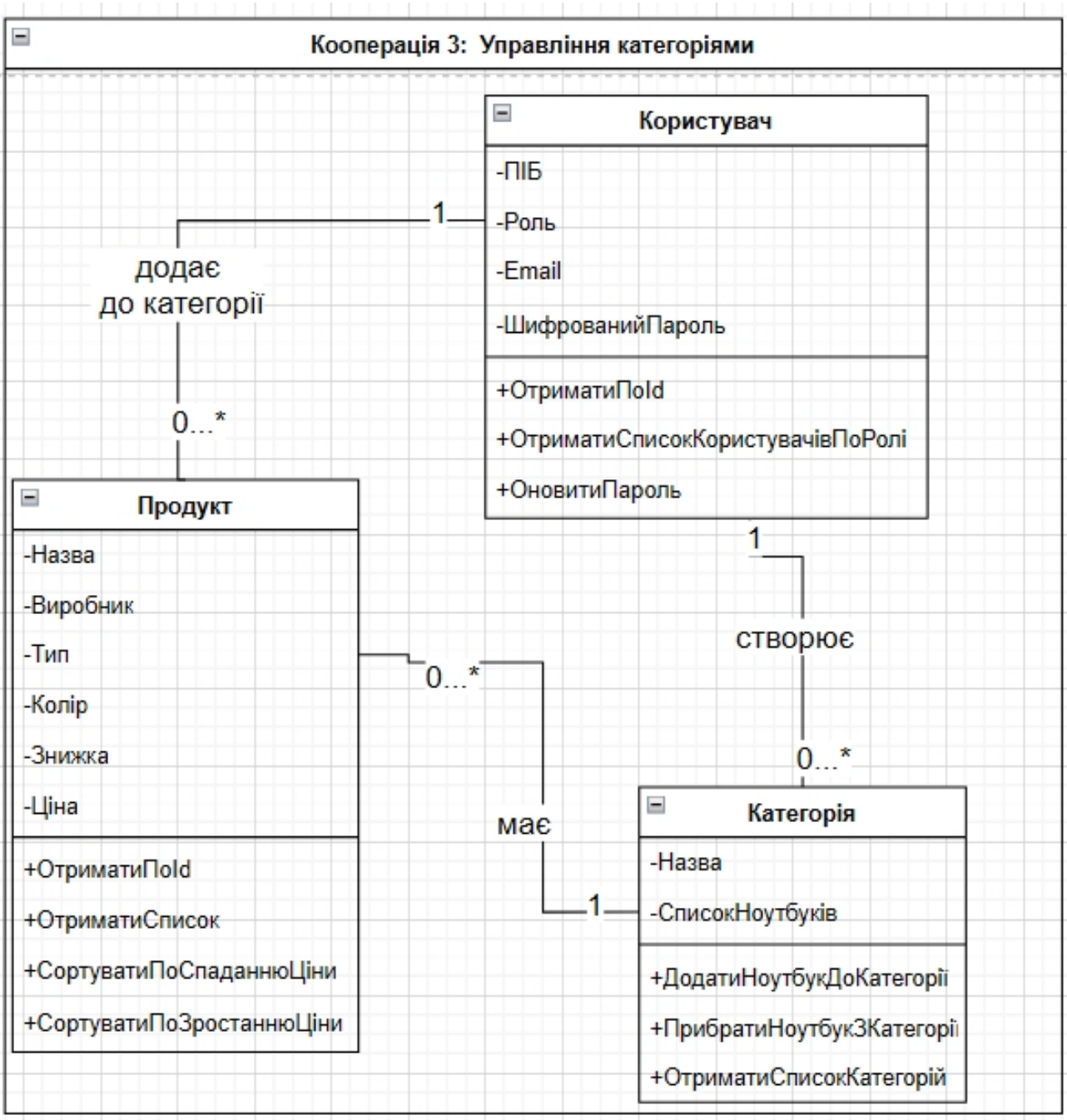
10. Брюгге Б., Дутуа А. Об’єктно-орієнтована інженерія програмного забезпечення з використанням UML, шаблонів та Java. — К. : Діалектика, 2012. — 680 с. URL: https://nibmehub.com/opac-service/pdf/read/Object%20Oriented%20Software%20Engineering%20_%20using%20UML-patterns%20and%20Java.pdf (дата звернення: 11.05.2025).MDN Web Docs. JavaScript Guide. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide (дата звернення: 11.05.2025).
11. Золотий Д. М. Розробка системи для автоматизації управління Git-репозиторіями з використанням ASP.NET Core та Entity Framework. — Тернопіль : ТНТУ, 2022. URL: https://elartu.tntu.edu.ua/handle/lib/38299 (дата звернення: 11.05.2025).
12. Тихоход В., Пасічнюк А. Моделювання та реалізація доменних подій у архітектурі Domain-Driven Design на платформі .NET Core. *Комп’ютерні системи та інформаційні технології*. 2023. №2. URL: https://csitjournal.khmnu.edu.ua/index.php/csit/article/download/229/151 (дата звернення: 11.05.2025).
13. Таран Н. М. Рекомендаційна система планування рекламної кампанії на основі аналізу даних веб-трафіку та поведінки користувачів. *Журнал якісної моделі та технологій*. 2024. URL: https://jqmth.donnu.edu.ua/article/view/15219 (дата звернення: 12.05.2025).
14. Brind M. *ASP.NET Core Razor Pages in Action*. Manning Publications, 2023. URL: https://books.google.com/books?id=nK6eEAAAQBAJ (дата звернення: 12.05.2025).
15. ISTQB. *Foundations of Software Testing: ISTQB Certification*. 4th ed. London: Wiley, 2021. 312 p.
16. Фрімен А. ASP.NET Core 3. Розробка хмароорієнтованих веб-застосунків із використанням MVC, Blazor і Razor Pages. Берклі : Apress, 2020. URL:

https://books.google.com/books?hl=en&lr=&id=o5npDwAAQBAJ (дата звернення: 13.05.2025).

ДОДАТОК А

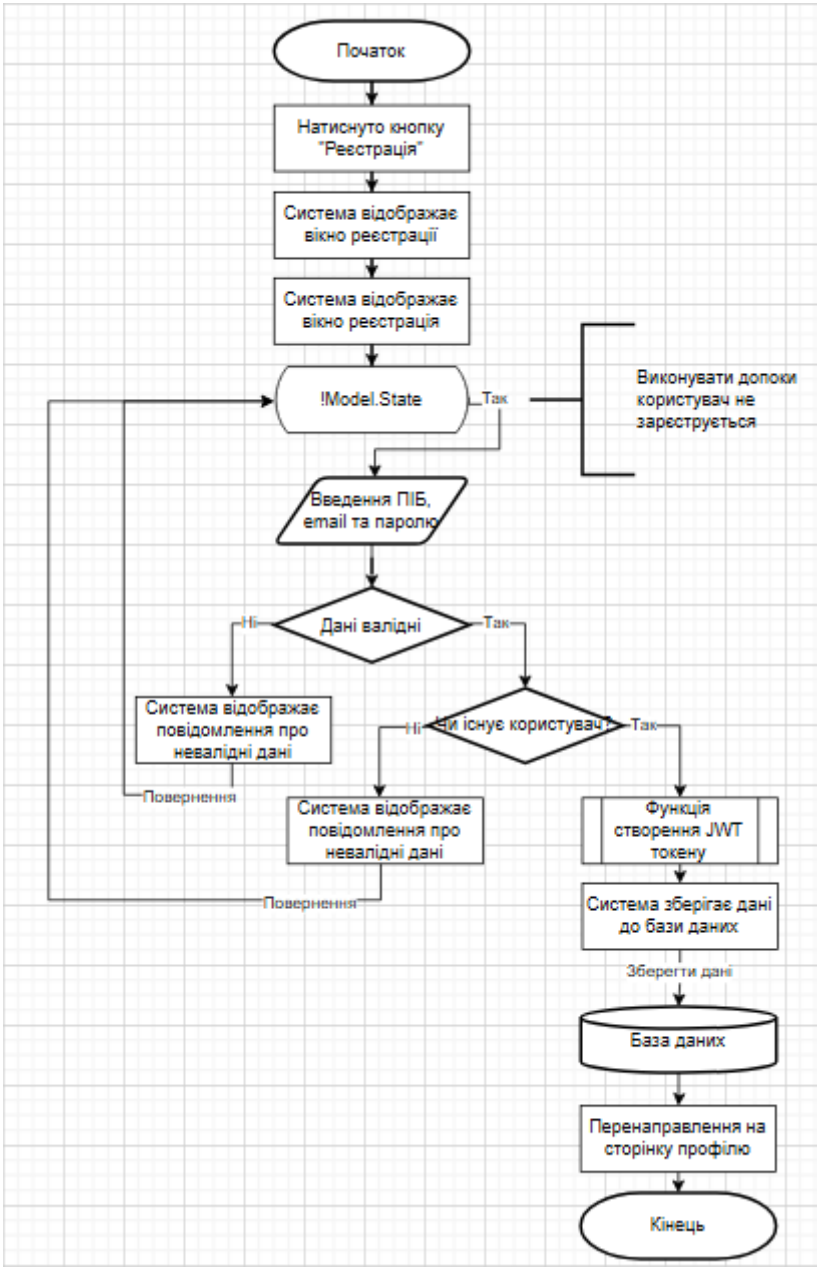

Кооперація 1: Управління улюбленим та кошиком
Користувач
-ПІБ
-Роль
-Email
-ШифрованийПароль
+ОтриматиПоId
+ОтриматиСписокКористувачівПоРолі
+ОновитиПароль
1
додає до
0...1
1
додає до
0...1
Кошик
-Назва
-СписокНоутбуків
-ЗагальнаЦіна
+ОчиститиКошик
+ДодатиНоутбукВКошик
+ВиключитиОдинТоварЗКошику
Улюблене
-Назва
-СписокНоутбуків
+ДодатиДоУлюбленого
+ОтриматиПоId
+ВидалитиЗУлюбленого
+ОчиститиСписок


Кооперація 2: Управління замовленнями та доставкою
Користувач
-ПІБ
-Роль
-Email
-ШифрованийПароль
+ОтриматиПоId
+ОтриматиСписокКористувачівПоРолі
+ОновитиПароль
1
оформлює
0...*
Замовлення
-НомерЗамовлення
-ОтримувачЗамовлення
-ВартістьЗамовлення
-СтатусЗамовлення
-ДатаОформлення
-ОріентованийЧасОтримання
+ОтриматиСписокЗамовлень
+ОтриматиПоId
+ОформитиЗамовлення
має
1
1
Доставка
-МетодДоставки
-ВартістьДоставки
+ОтриматиСписокДоставок

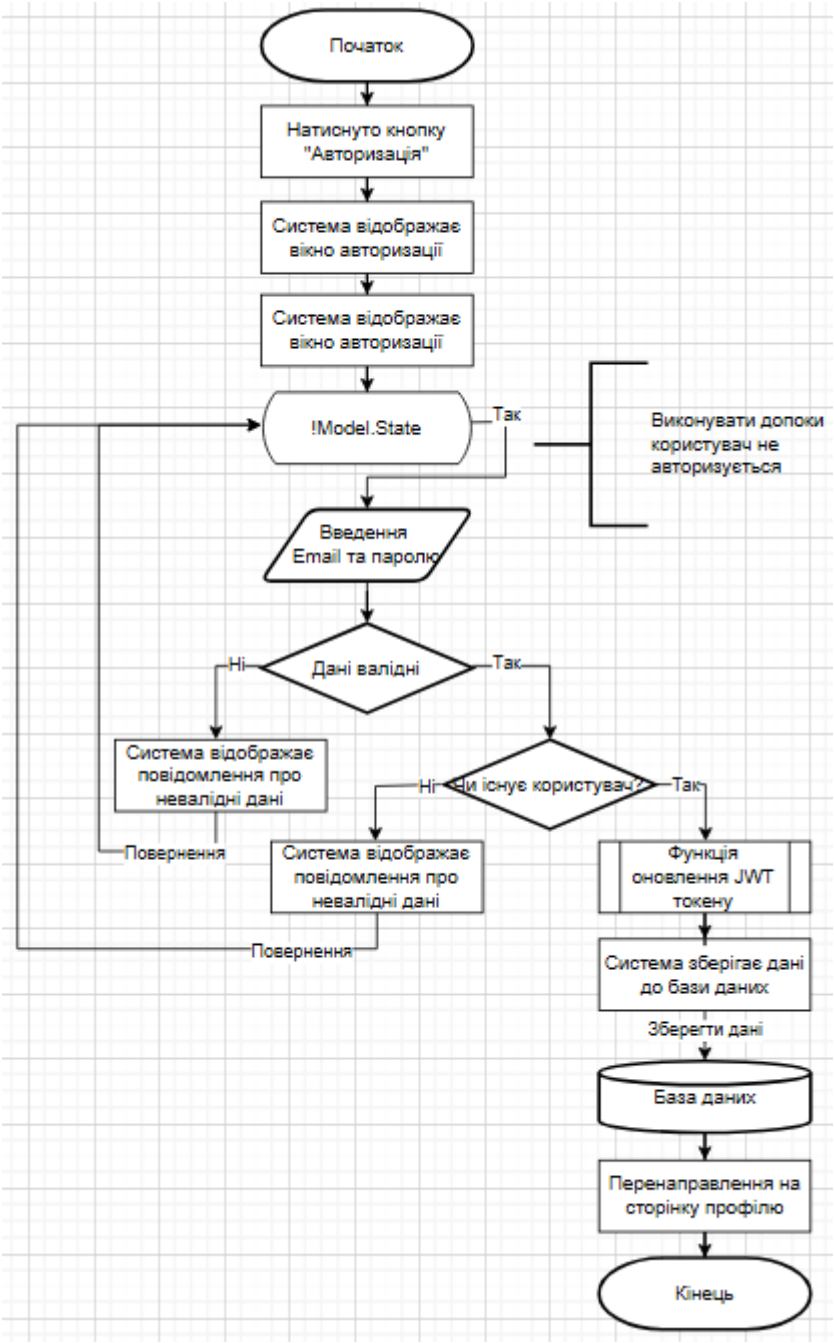

Кооперація 3: Управління категоріями
Користувач
-ПІБ
-Роль
-Email
-ШифрованийПароль
+ОтриматиПоId
+ОтриматиСписокКористувачівПоРолі
+ОновитиПароль
1
додає
до категорії
0...*
Продукт
-Назва
-Виробник
-Тип
-Колір
-Знижка
-Ціна
+ОтриматиПоId
+ОтриматиСписок
+СортуватиПоСпаданнюЦіни
+СортуватиПоЗростаннюЦіни
0...*
має
1
1
створює
0...*
Категорія
-Назва
-СписокНоутбуків
+ДодатиНоутбукДоКатегорії
+ПрибратиНоутбукЗКатегорії
+ОтриматиСписокКатегорій

ДОДАТОК Б

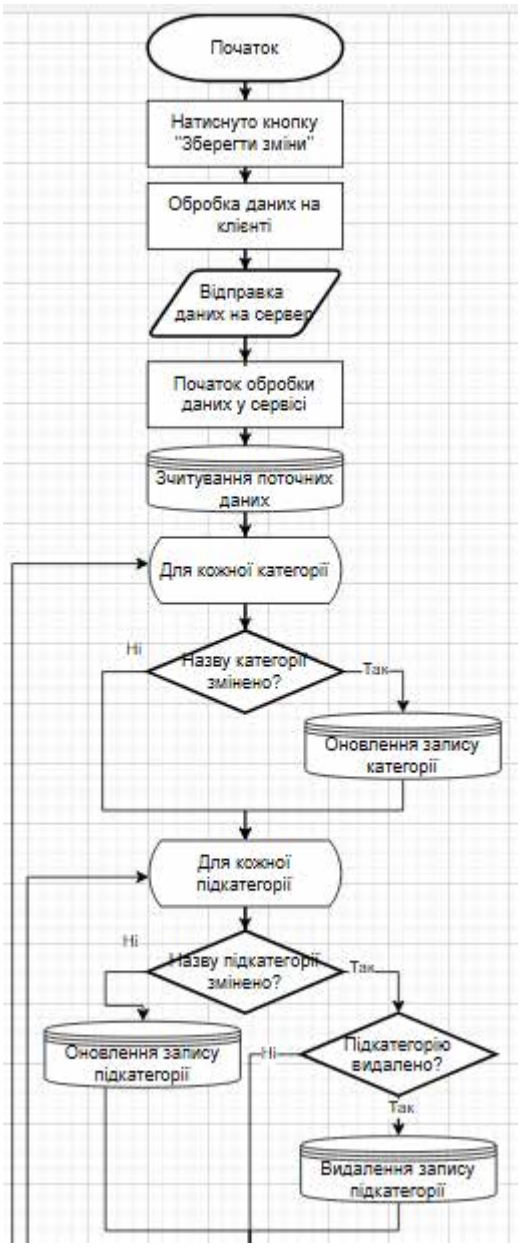
Блок-схема алгоритму «Реєстрація користувача в системі»



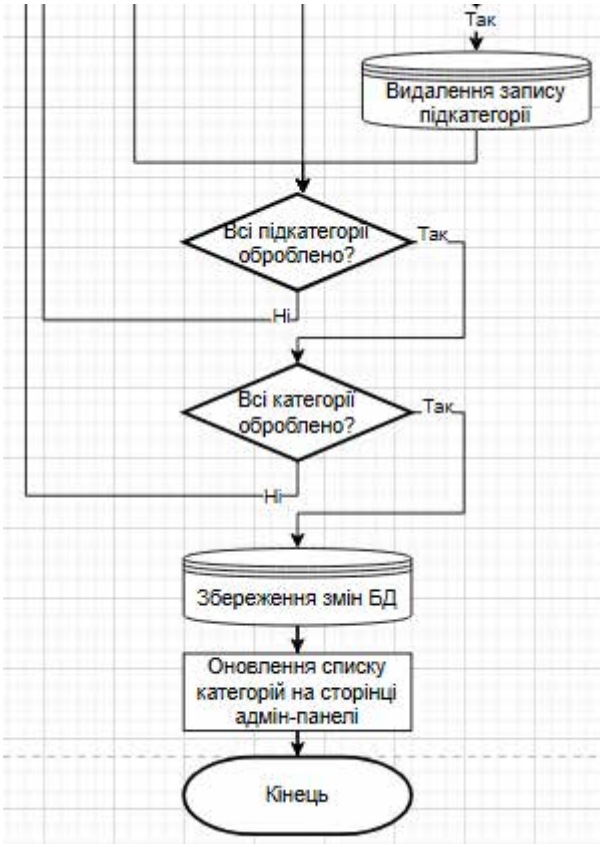
Блок-схема алгоритму «Авторизація користувача в системі»



Блок-схема алгоритму «Збереження змін при редагуванні категорій»



Продовження діаграми



ДОДАТОК В

Код алгоритму «Реєстрація користувача»

Фронтенд

Код файлу _SignUpPartial.cshtml

```
@model BasedTechStore.Application.DTOs.Identity.Request.SignUpRequest

<div class="modal fade" id="regModal" tabindex="-1" aria-
labelledby="regModalTitle" aria-hidden="true">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h1 class="modal-title align-content-center"
id="regModalTitle">Реєстрація</h1>
                <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-label="Close"></button>
            </div>
            <div class="modal-body">
                @if (ViewData.ModelState["SignUp"]?.Errors?.Count > 0)
                {
                    <div class="alert alert-danger">
                        @foreach (var error in
ViewData.ModelState["SignUp"].Errors)
                        {
                            <div>@error.ErrorMessage</div>
                        }
                    </div>
                }
                @using (Html.BeginForm("SignUp", "Auth", FormMethod.Post,
new { @class = "form-horizontal" }))
                {
                    @Html.AntiForgeryToken()
                    <div class="form-floating mb-3">
                        <input type="text" class="form-control"
id="FullName" name="FullName" placeholder="Full name" required>
                        <label for="FullName">ПІБ</label>
                    </div>
                    <div class="form-floating mb-3">
                        <input type="email" class="form-control"
id="Email" name="Email" placeholder="Email" required>
                        <label for="Email">Email</label>
                    </div>
                    <div class="form-floating mb-3">
                        <input type="password" class="form-control"
```

```
id="Password" name="Password" placeholder="Password" required>
                        <label for="Password">Пароль</label>
                    </div>

                    <div class="modal-footer">
                        <button type="button" class="btn btn-secondary"
data-bs-dismiss="modal" name="Close">Закрити</button>
                        <button type="submit" class="btn btn-
primary">Зареєструватися</button>
                    </div>
                }
            </div>
        </div>
    </div>
</div>
```

Бекенд

Частковий код файлу AuthController.cs

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> SignUp(SignUpRequest signUpRequest)
{
    if (!ModelState.IsValid)
    {
        TempData["SignUpErrors"] = "Invalid fields";
        return RedirectToAction("Index", "Home");
    }

    var response = await _authService.SignUpAsync(signUpRequest);
    if (!response.IsSuccess)
    {
        TempData["SignUpErrors"] = string.Join(", ", response.Errors);
        return RedirectToAction("Index", "Home");
    }

    return RedirectToAction("Index", "Profile");
}
```

Частковий код файлу AuthService.cs

```
public async Task<AuthenticationResponse> SignUpAsync(SignUpRequest
signUpRequest)
{
    var existingUser = await
_userManager.FindByEmailAsync(signUpRequest.Email);
    if (existingUser != null)
    {
        return AuthenticationResponse.CreateFailure(new[] { "User already
exists" });
```

```
    }

    var newUser = _mapper.Map<AppUser>(signUpRequest);
    var result = await _userManager.CreateAsync(newUser,
signUpRequest.Password);
    if (!result.Succeeded)
    {
        return
AuthenticationResponse.CreateFailure(result.Errors.Select(e =>
e.Description));
    }

    var jwtToken = await GenerateJwtTokenAsync(newUser);
    return AuthenticationResponse.CreateSuccess(jwtToken);
}
```

Код алгоритму «Авторизація користувача»

Фронтенд

Код файлу _SignInPartial.cshtml

```
@model BasedTechStore.Application.DTOs.Identity.Request.SignInRequest

<div class="modal fade" id="authModal" tabindex="-1" aria-
labelledby="authModalTitle" aria-hidden="true">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h1 class="modal-title"
id="authModalTitle">Авторизація</h1>
                <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-label="Close"></button>
            </div>
            <div class="modal-body">
                @if (ViewData.ModelState["SignIn"]?.Errors?.Count > 0)
                {
                    <div class="alert alert-danger">
                        @foreach (var error in
ViewData.ModelState["SignIn"].Errors)
                        {
                            <div>@error.ErrorMessage</div>
                        }
                    </div>
                }

                @using (Html.BeginForm("SignIn", "Auth", FormMethod.Post,
new { @class = "form-horizontal" }))
                {
                    @Html.AntiForgeryToken()
                    <div class="form-floating mb-3">
                        <input type="email" class="form-control"
id="Email" name="Email" placeholder="Email" required>
                        <label for="Email">Email</label>
```

```
                    </div>
                    <div class="form-floating mb-3">
                        <input type="password" class="form-control"
id="Password" name="Password" placeholder="Password" required>
                        <label for="Password">Пароль</label>
                    </div>

                    <div class="modal-footer">
                        <button type="button" class="btn btn-secondary"
data-bs-dismiss="modal" name="Close">Закрити</button>
                        <button type="submit" class="btn btn-
primary">Вхід</button>
                    </div>
                }
            </div>
        </div>
    </div>
</div>
```

Бекенд

Частковий код файлу AuthController.cs

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> SignIn(SignInRequest signInRequest)
{
    if (!ModelState.IsValid)
    {
        TempData["SignInErrors"] = "Invalid login attempt.";
        TempData["OpenModal"] = "authModal";
        return RedirectToAction("Index", "Home");
    }

    var response = await _authService.SignInAsync(signInRequest);
    if (!response.IsSuccess)
    {
        TempData["SignInErrors"] = string.Join(", ", response.Errors);
        TempData["OpenModal"] = "authModal";
        return RedirectToAction("Index", "Home");
    }

    if (!double.TryParse(await _authService.GetJwtExpirationMinutes(),
out var expirationMinutes))
        expirationMinutes = 60;

    Response.Cookies.Append("access-token", response.Token, new
CookieOptions
    {
        HttpOnly = true,
        Secure = true,
        SameSite = SameSiteMode.Strict,
        Expires = DateTime.UtcNow.AddMinutes(expirationMinutes)
```

```
    });

    return RedirectToAction("Index", "Profile");
}
```

Частковий код файлу AuthService.cs

```
public async Task<AuthenticationResponse> SignInAsync(SignInRequest
signInRequest)
        {
            var user = await
_userManager.FindByEmailAsync(signInRequest.Email);
            if (user == null)
            {
                return AuthenticationResponse.CreateFailure(new[] { "User
not found" });
            }

            var result = await _signInManager.PasswordSignInAsync(user,
signInRequest.Password, false, false);
            if (!result.Succeeded)
            {
                return AuthenticationResponse.CreateFailure(new[] {
"Invalid credentials" });
            }

            var jwtToken = await GenerateJwtTokenAsync(user);
            return AuthenticationResponse.CreateSuccess(jwtToken);
        }
```

Опис алгоритму «Збереження змін в категоріях»

Фронтенд

Код файлу _ManageCategoriesPartial.cshtml

```
@using BasedTechStore.Web.ViewModels.Categories
@model ManageCategoriesVM

<form asp-action="SaveCategories" asp-controller="AdminPanel"
method="post" id="saveCategoriesForm">
    @Html.AntiForgeryToken()
    <h4>Управління категоріями</h4>
    <div class="d-flex justify-content-between align-items-center mb-3">
        <div>
            <button type="button" class="btn btn-success btn-sm me-2"
id="addCategoryBtn" title="Додати категорію">
                <i class="bi bi-plus-lg"></i>
            </button>
            <button type="button" class="btn btn-danger btn-sm d-none"
id="deleteSelectedCategoriesBtn" title="Видалити обране">
                <i class="bi bi-trash"></i>
            </button>
            <button type="button" class="btn btn-primary btn-sm"
id="saveCategoriesButton" title="Зберегти зміни">
```

```
                <i class="bi bi-save"></i>
            </button>
        </div>
    </div>

    <table class="table table-bordered" id="categoriesTable">
        <thead>
            <tr>
                <th style="width: 40px;"><input type="checkbox"
id="selectAllCategoriesCheckbox" /></th>
                <th>Категорія</th>
                <th>Підкатегорії</th>
                <th style="width: 70px;">Дії</th>
            </tr>
        </thead>
        <tbody>
            @for (int i = 0; i < Model.Categories.Count; i++)
            {
                <tr>
                    <td>
                        <input type="checkbox"
class="rowCategoriesCheckbox" />
                    </td>
                    <td>
                        <input type="hidden" asp-for="Categories[@i].Id"
/>
                        <input class="form-control form-control-sm
category-name" asp-for="Categories[@i].Name" readonly />
                    </td>
                    <td>
                        @for (int j = 0; j <
Model.Categories[i].SubCategories.Count; j++)
                        {
                            <div class="d-flex align-items-center mb-1
subcategory-row">
                                <input type="hidden"
name="Categories[@i].SubCategories[@j].Id"
value="@Model.Categories[i].SubCategories[j].Id" />
                                <input class="form-control form-control-
sm me-2 subcategory-name"

name="Categories[@i].SubCategories[@j].Name"

value="@Model.Categories[i].SubCategories[j].Name" readonly />
                                <button type="button" class="btn btn-sm
btn-outline-danger removeSubCategoryBtn d-none" title="Видалити
підкатегорію">
                                    <i class="bi bi-x"></i>
                                </button>
                            </div>
                        }
```

```
                        <button type="button" class="btn btn-outline-
success btn-sm addSubCategoryBtn mt-1 d-none" data-category-index="@i"
title="Додати підкатегорію">
                            <i class="bi bi-plus"></i>
                        </button>
                    </td>
                    <td>
                        <button type="button" class="btn btn-sm btn-
outline-primary editCategoryBtn" title="Редагувати">
                            <i class="bi bi-pencil"></i>
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</form>

<!-- Confirm delete modal -->
<div class="modal fade" id="confirmCategoriesDeleteModal" tabindex="-1"
aria-labelledby="confirmCategoriesDeleteModalLabel" aria-hidden="true">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">Підтвердження видалення</h5>
                <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-label="Закрити"></button>
            </div>
            <div class="modal-body">
                Ви впевнені, що хочете видалити вибрані категорії? Цю дію
не можна скасувати.
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-secondary btn-sm"
data-bs-dismiss="modal">Скасувати</button>
                <button type="button" class="btn btn-danger btn-sm"
id="confirmCategoriesDeleteBtn">Видалити</button>
            </div>
        </div>
    </div>
</div>

@await Component.InvokeAsync("InfoModal", new
BasedTechStore.Web.ViewModels.Modals.InfoModalViewModel
{
    Id = "categoriesWarningOnLeave",
    Title = "Незбережені зміни",
    Message = "На сторінці є незбережені зміни категорій. Ви справді
хочете покинути сторінку?",
    SubMessage = "Всі незбережені зміни будуть втрачені.",
```

```
    IsWarning = true,
    ShowSaveButton = true,
    ShowConfirmButton = true,
    ShowCancelButton = true,
    ConfirmButtonText = "Вийти без збереження",
    SaveButtonText = "Зберегти та вийти",
    StaticBackdrop = true,
    TrackChanges = true,
    ChangeTrackingFormId = "saveCategoriesForm"
})
```

Частковий код файлу edit-categories.js

```
if (saveCategoryButton) {
    saveCategoryButton.addEventListener('click', async function (e) {
        e.preventDefault();

        const form = document.getElementById('saveCategoriesForm');
        const formData = new FormData(form);

        fetch(form.action, {
            method: 'POST',
            body: formData,
            headers: {
                'X-Requested-With': 'XMLHttpRequest'
            }
        })
            .then(response => {
                if (!response.ok) {
                    throw new Error('Помилка збереження категорій.');
                }
                return response.text(); // HTML response
            })
            .then(html => {
                const container =
document.getElementById('categoriesContainer');
                if (container) {
                    container.innerHTML = html;
                    registerAllCategoryEvents();
                }
            })
            .catch(error => {
                console.error('Помилка:', error);
                alert('Сталася помилка під час збереження категорій.');
            });
    });
}
```

Бекенд

Частковий код файлу AdminPanelController.cs

```
[HttpPost]
[Authorize(Roles = "Manager")]
```

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> SaveCategories(ManageCategoriesVM
manageCategoriesVM)
{
    if (!ModelState.IsValid)
    {
        return PartialView("_ManageCategoriesPartial",
manageCategoriesVM);
    }

    var categoriesDto =
_mapper.Map<List<CategoryDto>>(manageCategoriesVM.Categories);

    categoriesDto.SelectMany(c =>
        c.SubCategories.Select(sc =>
        {
            sc.CategoryId = c.Id;
            return sc;
        }))
        .ToList();

    var updatedCategoriesDto = await
_productService.SaveCategoriesAsync(categoriesDto);
    var updatedCategoriesVM = new ManageCategoriesVM
    {
        Categories =
_mapper.Map<List<CategoryItemVM>>(updatedCategoriesDto)
    };

    return PartialView("_ManageCategoriesPartial", updatedCategoriesVM);
}
```

Частковий код файлу ProductService.cs

```
public async Task<List<CategoryDto>>
SaveCategoriesAsync(List<CategoryDto> categoryDtos)
{
    foreach (var categoryDto in categoryDtos)
    {

        if (categoryDto.Id == Guid.Empty)
        {
            var newCategory = _mapper.Map<Category>(categoryDto);
            newCategory.Id = Guid.NewGuid();

            foreach (var subCategory in newCategory.SubCategories)
            {
                subCategory.Id = Guid.NewGuid();
                subCategory.CategoryId = newCategory.Id;
            }
            await _context.Categories.AddAsync(newCategory);
            continue;
```

```
        }
        else
        {
            var existingCategory = await _context.Categories
                .FirstOrDefaultAsync(c => c.Id == categoryDto.Id);

            if (existingCategory == null)
                continue;

            existingCategory.Name = categoryDto.Name;

            var existingSubCategories = await _context.SubCategories
                .Where(sc => sc.CategoryId == categoryDto.Id)
                .ToListAsync();

            var dtoSubIds = categoryDto.SubCategories.Select(x =>
x.Id).ToHashSet();

            // Remove subcategories that are not in the DTO
            var toRemove = existingSubCategories.Where(sc =>
!dtoSubIds.Contains(sc.Id)).ToList();
            _context.SubCategories.RemoveRange(toRemove);

            // Renew existing subcategories / add new ones
            foreach (var subDto in categoryDto.SubCategories)
            {
                if (subDto.Id == Guid.Empty)
                {
                    var newSubCategory =
_mapper.Map<SubCategory>(subDto);
                    newSubCategory.Id = Guid.NewGuid();
                    newSubCategory.CategoryId = categoryDto.Id;
                    _context.SubCategories.Add(newSubCategory);
                }
                else
                {
                    var existingSubCategory =
existingSubCategories.FirstOrDefault(sc => sc.Id == subDto.Id);

                    if (existingSubCategory != null)
                    {
                        existingSubCategory.Name = subDto.Name;
                    }
                }
            }
        }
    }

    await _context.SaveChangesAsync();

    var updatedCategories = await _context.Categories
```

```
            .Include(c => c.SubCategories)
            .ToListAsync();

        return _mapper.Map<List<CategoryDto>>(updatedCategories);
    }
```

Додаток Д

Календарний план

№ з/п	Назва етапів виконання бакалаврської кваліфікаційної роботи	Строк виконання етапів бакалаврської кваліфікаційної роботи	Примітка
1	Ознайомлення з вимогами до кваліфікаційної роботи, вибір теми, формування мети та завдань	12.09.2024 – 15.10.2024	Затвердження теми
2	Аналіз предметної області, вивчення аналогів, постановка задачі	01.11.2024 – 15.11.2024	Розділ 1
3	Вибір технологій, інструментів розробки, побудова архітектури системи	16.11.2024 – 30.11.2024	Обґрунтування вибору
4	Проєктування бази даних: створення ER-діаграми, логічної моделі	01.12.2024 – 15.12.2024	Розділ 2
5	Реалізація базової структури застосунку (інтерфейс, маршрутизація, база даних)	16.12.2024 – 10.01.2025	Початкове ядро системи
6	Розробка модуля аутентифікації та авторизації користувачів	11.01.2025 – 25.01.2025	Модуль безпеки
7	Реалізація функціоналу профілю користувача та управління обліковими записами	26.01.2025 – 10.02.2025	Розділ 3 (частково)
8	Розробка адміністративного інтерфейсу для керування товарами та категоріями	11.02.2025 – 25.02.2025	Основний функціонал
9	Додавання підтримки завантаження зображень, реалізація пошуку та фільтрації	26.02.2025 – 10.03.2025	Оптимізація роботи
10	Тестування функціоналу, валідація, виправлення помилок	11.03.2025 – 28.05.2025	Розділ 4 (частково)
11	Оформлення пояснювальної записки, включаючи всі розділи та додатки	01.04.2025 – 15.05.2025	Підготовка документації

12	Здача пояснювальної записки керівнику дипломної роботи для перевірки та отримання рекомендацій	15.05.2025	Здача чорнового варіанту записки
13	Очікування на результати перевірки пояснювальної записки керівником дипломної роботи	15.05.2025 – 28.05.2025	Результати перевірки записки
14	Проведення фінального тестування, підготовка демонстраційних матеріалів	26.04.2025 – 28.05.2025	Готовність до захисту
15	Підготовка до захисту, оформлення презентації, доповіді	21.05.2025 – 28.05.2025	Презентація
16	Захист бакалаврської кваліфікаційної роботи	28.05.2025 – 16.06.2025	Захист