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

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавра

на тему: «**Проєктування та розробка Backend частини веб-сервісу для продажу кави»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Лук'янчук Михайло Юрійович

Керівник: *науковий ступінь, вчене звання, посада,*
Клебан Юрій Вікторович

Рецензент: *кандидат технічних наук, доцент,*
доцент кафедри прикладної математики
Донецького національного університету
імені Василя Стуса
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 20 » травня 2026 р.

Острог, 2026

З А В Д А Н Н Я
на кваліфікаційну роботу/проєкт студента

Лук'янчук Михайло Юрійович

1. *Тема роботи:* “Проєктування та розробка Backend частини веб -сервісу для продажу кави”.

Керівник проєкту: Клебан Юрій Вікторович, старший викладач кафедри економіко-математичного моделювання та інформаційних технологій.

Затверджено наказом ректора НаУОА від 05 жовтня 2026 року.

2. *Термін здачі студентом закінченої роботи/проєкту:* _____.

3. *Вихідні дані до роботи/проєкту:* середовище розробки JetBrains Rider, інструменти тестування Postman та Swagger, Docker, GitHub. Стек технологій: мова програмування C#, .NET 8 (ASP.NET Core), Entity Framework Core, PostgreSQL, MediatR (CQRS), JWT, Stripe API.

4. *Перелік завдань, які належить виконати:* спроєктувати архітектуру серверної частини (Backend) на основі Clean Architecture, розробити схему бази даних, реалізувати RESTful API з використанням патерну CQRS, налаштувати автентифікацію користувачів, інтегрувати сторонні сервіси (платежі Stripe та AI), задокументувати інтерфейс за допомогою Swagger, налаштувати розгортання проєкту через Docker.

5. *Перелік графічного матеріалу:* рисунки, діаграми, лістинги.

6. *Консультанти розділів роботи:*

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Клебан Ю. В.		
2	Клебан Ю. В.		
3	Клебан Ю. В.		

7. Дата видачі завдання: _____ р.

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: *Проєктування та розробка Backend частини веб -сервісу для продажу кави*
Автор: *Лук'янчук Михайло Юрійович*

Науковий керівник: *Клебан Юрій Вікторович старший викладач кафедри економіко-математичного моделювання та інформаційних технологій.*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *____(кількість сторінок роботи) с., ____ (кількість рисунків) рис., _____ (кількість таблиць) табл., _____ (кількість додатків) додатків, _____ (кількість джерел) джерел.*

Ключові слова: *BACKEND, ASP.NET CORE, C#, WEB API, REST API, CLEAN ARCHITECTURE, ENTITY FRAMEWORK CORE, POSTGRESQL, DOCKER, JWT, ЕЛЕКТРОННА КОМЕРЦІЯ, ВЕБ-СЕРВІС, УПРАВЛІННЯ ТОВАРАМИ, АВТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ.*

Короткий зміст праці:

У кваліфікаційній роботі розглянуто процес проєктування та розробки backend-частини вебсервісу, призначеного для продажу кави. Актуальність теми полягає у зростаючій потребі створення надійних, безпечних та масштабованих платформ для сучасного сектору електронної комерції. Метою роботи було створення комплексної серверної логіки, яка забезпечує повний життєвий цикл взаємодії користувача з інтернет-магазином, починаючи від реєстрації і закінчуючи успішним оформленням замовлення. У теоретичній частині обґрунтовано вибір технологічного стеку та сучасних архітектурних підходів. В якості фундаментальної платформи для розробки було обрано середовище .NET 8 та фреймворк ASP.NET Core для створення Web API. Архітектура системи побудована згідно з принципами Clean Architecture (чиста архітектура), що забезпечує жорстке розділення відповідальності між програмними шарами: рівнем представлення (Api), рівнем бізнес-логіки (Application), інфраструктурним рівнем доступу до даних (DataAccessLayer) та ядром доменної моделі (Domain). Для взаємодії з реляційною базою даних використано технологію об'єктно-реляційного відображення Entity Framework Core із провайдером для портування у PostgreSQL. Контейнеризація розробленої системи здійснюється за допомогою платформи Docker, що гарантує стабільність розгортання, портативність та легкість подальшого масштабування. Практична частина роботи присвячена безпосередній реалізації ключового функціоналу серверної частини вебсервісу. В рамках розробки RESTful API створено надійну систему автентифікації та авторизації на базі JSON Web Tokens

(JWT) для захисту кінцевих точок застосунку. Також було алгоритмізовано процеси управління каталогом, що включає повноцінний набір CRUD-операцій для товарів та їх категорій, дозволяючи адміністраторам ефективно керувати асортиментом. Важливим етапом стала програмна реалізація механізмів клієнтської взаємодії, що охоплює процеси управління віртуальним кошиком, алгоритми оформлення та відстеження статусів замовлень, формування персоналізованих списків обраної продукції, а також систему створення користувацьких відгуків. Розроблена архітектура системи спроєктована з урахуванням високих вимог до гнучкості, що гарантує надійну та ефективну основу для повноцінної інтеграції з frontend-додатком.

The qualification work considers the process of designing and developing the backend part of a web service intended for selling coffee. The relevance of the topic lies in the growing need to create reliable, secure, and scalable platforms for the modern e-commerce sector. The main purpose of the work was to create a comprehensive server logic that ensures the full lifecycle of user interaction with the online store, from initial registration to successful order placement. The theoretical part justifies the choice of the technology stack and modern architectural approaches. The .NET 8 platform and the ASP.NET Core framework were chosen as the fundamental basis for developing the Web API. The system architecture is built according to the principles of Clean Architecture, which ensures a strict separation of concerns between software layers: presentation layer (Api), business logic layer (Application), data access infrastructure layer (DataAccessLayer), and the domain model core (Domain). Entity Framework Core with a PostgreSQL provider was utilized for seamless interaction with the relational database. The deployment of the developed system is containerized using Docker, which guarantees its portability, stability across environments, and ease of scaling. The practical part of the work is devoted to the direct implementation of the key functionality of the web service's backend. Within the framework of the RESTful API development, a robust authentication and authorization system based on JSON Web Tokens (JWT) was created to secure application endpoints. Catalog management processes were also algorithmized, including a full set of CRUD operations for products and their categories, thus allowing administrators to manage the assortment effectively. An essential stage was the software implementation of client interaction mechanisms, covering shopping cart management, order processing and status tracking algorithms, the creation of personalized favorite product lists, and a robust user review system. The designed system architecture takes into account rigorous requirements for structural flexibility, ensuring a reliable and highly efficient foundation for seamless integration with the frontend application.

ЗМІСТ

ВСТУП

Інтенсивний розвиток інформаційно-комунікаційних технологій та глобальне проникнення інтерактивних мережевих рішень у всі сфери суспільного життя кардинально змінили фундаментальну парадигму функціонування сучасних торговельних підприємств. Традиційні методи реалізації товарів та послуг через фізичні точки продажу поступово поступаються місцем складним розподіленим програмним комплексам, які здатні автономно та безперебійно обслуговувати тисячі клієнтів одночасно, незалежно від їхнього часового поясу чи географічного розташування. Індустрія продажу спеціалізованих продовольчих товарів, зокрема сегмент високоякісної кави європейського та світового зразка, висуває особливо жорсткі та специфічні вимоги до організації процесів електронної комерції. Це зумовлено насамперед складною природою самого продукту: необхідністю врахування та конфігурації безлічі змінних параметрів (таких як сорт зерен, країна походження, ступінь та дата обсмаження, детальний профіль смаку, варіанти помелу тощо). Такі умови вимагають від програмної платформи не лише банального статичного відображення списку товарної номенклатури, але й забезпечення роботи потужних алгоритмів фільтрації, інтелектуального пошуку та управління об’ємними реляційними масивами метаданих. Відповідно, програмні рішення для подібних нішевих ринків повинні володіти екстраординарним ступенем технічної адаптивності та бездоганною швидкістю відгуку, щоб задовольнити вимоги сучасного споживача, який очікує від взаємодії з вебдодатком такого ж високого рівня персоналізації та надійності, як і від обслуговування професійним баристою у фізичному закладі.

Фундаментальним ядром, що забезпечує життєдіяльність, відмовостійкість та загальну безпеку будь-якої сучасної системи електронної комерції, є її серверна інфраструктура, відома в інженерній практиці як backend-частина застосунку. Цей архітектурний рівень виступає в ролі невидимого для роздрібного клієнта, але критично важливого центрального координаційного вузла всієї платформи. Саме в межах бекенду сконцентрована реалізація всієї багаторівневої бізнес-логіки,

алгоритми гарантування суворої транзакційної цілісності бази даних, багатокритеріальна валідація агрегованих вхідних інформаційних потоків, розв’язання складних конфліктів при паралельному чи асинхронному доступі до обмежених ресурсів, а також безпрецедентний криптографічний захист конфіденційної персональної і платіжної інформації. Глобальний історичний досвід інженерії програмного забезпечення наочно демонструє, що застосування застарілих монолітних підходів або відсутність чіткої структурної ізоляції програмного коду неминуче призводить до швидкого накопичення критичного технічного боргу. У таких системах найменша модифікація одного функціонального модуля може стати тригером для каскаду непередбачуваних критичних збоїв у геть інших підсистемах. Для мінімізації подібних катастрофічних сценаріїв та забезпечення можливостей подальшого безперешкодного горизонтального чи вертикального масштабування архітектури, надзвичайно важливо імплементувати передові парадигми проєктування на ранніх етапах розробки. Серед них домінуючу позицію посідає концепція «Чистої архітектури» (Clean Architecture) в тісній технічній синергії з патерном суворого розділення команд та запитів (Command Query Responsibility Segregation – CQRS). Такий концептуальний інженерний тандем дозволяє досягти стовідсоткової ізоляції доменної області застосунку від будь-яких зовнішніх інфраструктурних залежностей, сторонніх фреймворків чи конкретних постачальників систем управління базами даних, чим гарантує високу тестованість, модульність та довговічність створюваного програмного забезпечення.

З огляду на комплекс вищевикладених факторів, головною метою даної кваліфікаційної роботи є системне проєктування, концептуалізація та практична розробка відмовостійкого програмного продукту — повноцінної backend-частини вебсервісу, спеціалізованого на реалізації кавової продукції. Досягнення визначеної глобальної мети передбачає декомпозицію та поетапне виконання низки взаємопов'язаних науково-інженерних і практичних завдань, що охоплюють увесь багатогранний життєвий цикл створення серверного програмного забезпечення. Насамперед, пріоритетним завданням є проєктування надійної, слабкозв'язаної архітектури застосунку, яка здатна ефективно абсорбувати та розподіляти високі

мережеві навантаження, забезпечуючи при цьому безперебійну операційну діяльність магазину. Наступним стратегічним кроком є розробка ретельно нормалізованої реляційної схеми бази даних, що мінімізує будь-які ризики виникнення аномалій модифікації та забезпечує довгострокове, структуроване зберігання масивів даних про зареєстрованих користувачів, їхні фінансові транзакції, каталог продукції та інвентаризаційні записи. Крім того, передбачається створення високопродуктивного та семантично коректного RESTful API, який слугуватиме універсальним стандартизованим мережевим інтерфейсом для інформаційного обміну між інтерфейсом користувача (фронтендом) та серверними потужностями. Окремим ключовим завданням виступає програмна імплементація багаторівневих механізмів автентифікації та авторизації з використанням безпечних протоколів криптографічного цифрового підпису для генерації та перевірки токенів доступу (JSON Web Tokens), що унеможливить несанкціоноване втручання в особисті кабінети клієнтів. Разом із цим, необхідно спроєктувати життєвий цикл цифрового кошика, інструменти оформлення замовлень, а також здійснити фінальне інфраструктурне налаштування платформи шляхом створення конфігурацій для контейнеризації продукту за допомогою засобів Docker, що переведе розроблений проєкт у стан готовності до промислової (production) експлуатації у хмарних середовищах.

Об'єктом дослідження у межах даної кваліфікаційної роботи виступають глибинні процеси алгоритмізації обробки даних, забезпечення комплексного інформаційного захисту та імплементації складної розгалуженої бізнес-логіки в розподілених мережевих інформаційних системах сектору B2C (електронної комерції). Поняттєвий обсяг об'єкта дослідження охоплює аналіз етапів життєвого циклу електронного замовлення як програмної транзакційної сутності, методи маршрутизації користувацьких запитів, алгоритми асинхронної обробки інформації та механізми збереження інформаційної консистентності системи під час пікових навантажень — від моменту ініціації сесії браузером до успішного збереження фінальної транзакції на фізичний носій віддаленого сервера.

Предметом дослідження є інженерні архітектурні моделі, патерни проєктування рівня підприємства, концептуальні засади управління станами інформаційної системи, а також найсучасніші інструментальні та технологічні програмні засоби розробки високопродуктивних серверних застосунків в межах екосистеми .NET. У роботі детально препарується та обґрунтовується застосування принципів чистої архітектури (Clean Architecture) з інтегрованим принципом CQRS, практична реалізація якого здійснюється через впровадження медіаторних абстракцій за допомогою спеціалізованої бібліотеки MediatR. Технологічним підґрунтям для реалізації поставлених практичних цілей виступає об'єктно-орієнтована мова програмування C# та високопродуктивний кросплатформний фреймворк ASP.NET Core, що є визнаними індустріальними стандартами для конструювання масштабованих веб-сервісів. Класична проблема імпедансу, тобто структурного неузгодження доменної об'єктно-орієнтованої моделі застосунку з реляційною математичною природою бази даних PostgreSQL, віртуозно вирішується шляхом підключення та конфігурації технології об'єктно-реляційного відображення (ORM) Entity Framework Core. Важливий акцент у дослідженні також ставиться на використанні інструментарію системної контейнеризації Docker, що дає змогу повністю абстрагувати розроблений функціонал від мінливих апаратних особливостей та операційної системи хост-машини, забезпечуючи ідентичність ізольованих середовищ для розробника, тестувальника та кінцевого сервера розгортання.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАСОБІВ РОЗРОБКИ

1.1. Опис предметного середовища

Предметним середовищем даної кваліфікаційної роботи є сфера електронної комерції, зокрема спеціалізований ринок продажу кавових зерен, супутніх товарів та послуг через інтернет-платформи. У сучасному світі, який характеризується стрімкою цифровізацією всіх сфер життя, електронна комерція виступає одним із найбільш динамічних сегментів глобальної економіки. Кавова індустрія, зі свого боку, має специфічні риси: споживачі стають все більш обізнаними, вибагливими до сортів, методів обсмаження та походження зерен. Відповідно, традиційні методи продажів поступаються місцем високотехнологічним інформаційним системам, які здатні не лише забезпечити процес транзакції, але й надати клієнту унікальний, персоналізований досвід взаємодії з продуктом. Створення ефективного веб-сервісу вимагає комплексного підходу, що охоплює управління електронним каталогом, обробку фінансових платежів, організацію логістичної інформації та впровадження інтелектуальних систем допомоги під час вибору товару. Основною метою функціонування такого середовища є задоволення потреб кінцевого споживача через надання максимально зручного, швидкого та безпечного інструменту для здійснення покупок, а також автоматизація рутинних завдань менеджменту та адміністрування для власників бізнесу.

1.1.1. Аналіз бізнес-процесів інтернет-магазину кави

Будь-який сучасний інтернет-магазин є складною екосистемою взаємопов'язаних бізнес-процесів, які об'єднують клієнтський досвід (Customer Journey) та внутрішні операційні процедури адміністративного персоналу. Процес діяльності кав’ярні в онлайн-форматі розпочинається з етапу приваблення клієнта та його інформування. Клієнтська подорож стартує з навігації по каталогу товарів, де користувач отримує доступ до деталізованої інформації про кожен продукт. Цей

процес вимагає ефективної системи категоризації та фільтрації, щоб користувач міг швидко знайти потрібний сорт кави, оцінити її характеристики, переглянути фотографії та актуальні ціни. Відтак, життєвий цикл клієнтської взаємодії переходить до етапу додавання товару у віртуальний кошик, який діє як тимчасове сховище обраних позицій до моменту прийняття остаточного рішення про покупку. Важливим аспектом цього процесу є можливість динамічного перерахунку вартості замовлення з урахуванням кількості товару та можливих знижок.

З боку адміністративного управління бізнес-процеси виглядають принципово інакше і базуються на концепції безперервного контролю та оновлення інформації. Адміністратор системи відповідає за управління асортиментом, що включає додавання нових позицій, редагування існуючих описів, завантаження медіаконтенту та приховування товарів, які тимчасово відсутні на складі. Крім того, адміністративний бізнес-процес охоплює моніторинг оформлених замовлень, відстеження їх статусів (наприклад, очікування оплати, оплачено, комплектується, відправлено) та модерацію соціальної складової платформи, такої як відгуки користувачів. Таким чином, бізнес-процеси інтернет-магазину являють собою безперервний цикл інформаційного обміну між клієнтами, які генерують попит, та системою, яка цей попит автоматизовано задовольняє.

1.1.2. Функціональна модель системи та основні актори

Функціональна модель системи визначає сукупність дій, які може виконувати програмний комплекс у відповідь на запити різних категорій користувачів. Для чіткого розмежування прав доступу та забезпечення інформаційної безпеки виокремлюються чотири ключові актори: Гість, Клієнт, Адміністратор та AI-асистент. Актор «Гість» представляє собою неавторизованого відвідувача платформи. Його можливості обмежені базовим функціоналом: перегляд каталогу товарів, використання фільтрів, читання відгуків інших покупців та взаємодія з інформаційними сторінками сайту. Ця категорія є критично важливою, оскільки

більшість конверсій починається саме з гостьового візиту, і система повинна працювати максимально швидко, щоб не втратити потенційного покупця.

Актор «Клієнт» є авторизованим користувачем системи, що пройшов процедуру реєстрації та підтвердив свою особу. Клієнт отримує доступ до розширеного функціонала: формування особистого профілю, збереження історії замовлень, управління адресами доставки, додавання товарів до списку улюблених (Favorites), а також можливість залишати власні відгуки до придбаних товарів. Актор «Адміністратор» володіє найвищим рівнем привілеїв у системі. Його функціональна роль полягає в технічному та контентному супроводі порталу. Це включає повний доступ до CRUD-операцій (створення, читання, оновлення, видалення) над сутностями товарів, категорій, користувачів та відгуків, а також можливість управління правами доступу інших акторів. Новітнім актором у цій екосистемі виступає «AI-асистент», який діє як автономний програмний агент, інтегрований на рівні бекенду. Його роль зводиться до аналізу текстових запитів користувачів і надання персоналізованих рекомендацій щодо вибору кавових сумішей на основі складних алгоритмів обробки природної мови, закладених у зовнішніх великих мовних моделях. Цей актор суттєво розширює функціональну модель, перетворюючи стандартний каталог на інтерактивне середовище спілкування з клієнтом.

1.1.3. Моделювання основного бізнес-процесу "Покупка товару"

Основним бізнес-процесом, який безпосередньо впливає на комерційний успіх системи, є процес здійснення покупки (Checkout). Цей процес є транзакційним і включає чітку послідовність етапів, обов'язкових до виконання. На першому етапі користувач завершує формування кошика і переходить до процедури оформлення замовлення. Система перевіряє факт авторизації; якщо користувач не ідентифікований, йому пропонується увійти в систему або створити новий обліковий запис для збереження гарантій безпеки. Після успішної ідентифікації система пропонує клієнту ввести або обрати з попередньо збережених деталі доставки, які

включають країну, місто, поштовий індекс, вулицю та номер відділення поштового оператора. Ці просторові дані логічно прив’язуються до створюваної чернетки замовлення в базі даних.

Наступним критичним етапом є інтеграція фінансових потоків. Після підтвердження адреси та сумарної вартості товарів (включно з можливими податками та вартістю доставки), бекенд-система самостійно звертається до API платіжного шлюзу для реєстрації наміру платежу (Payment Intent). Користувач вводить реквізити платіжної картки, і ця інформація зашифровано передається безпосередньо до провайдера послуг, минаючи сервери магазину для забезпечення відповідності стандартам PCI DSS. Після успішної транзакції платіжний провайдер асинхронно надсилає на сервер магазину спеціальний сигнал (Webhook) про зміну статусу платежу. Лише після отримання та криптографічної перевірки цього сигналу бекенд переводить статус замовлення у стан «Оплачено», очищує віртуальний кошик користувача та надсилає інформацію про нове замовлення на панель управління Адміністратора для подальшого фізичного пакування та відправлення вантажу. Більш наочно ця послідовність взаємодій під час купівлі товару подана на відповідній діаграмі діяльності, що зображена на Рис. 1.1.

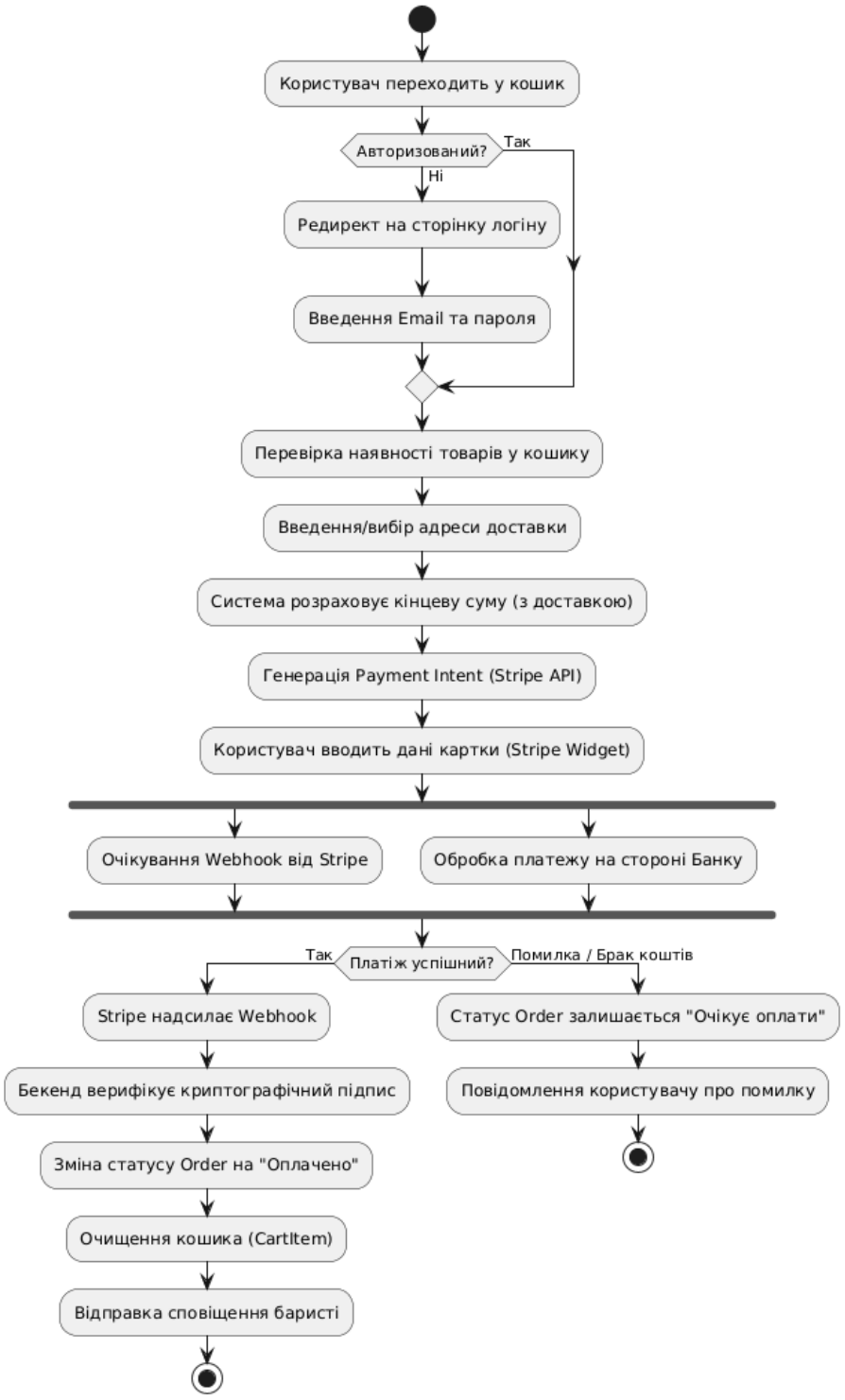

Рис. 1.1. Діаграма діяльності для процесу покупки товару

Джерело: [створено автором]

1.2. Огляд та аналіз аналогічних програмних рішень

На етапі підготовки масштабного програмного проєкту обов'язковим кроком є ретельне вивчення та об'єктивний аналіз існуючих на ринку аналогів, що дозволяє уникнути повторення чужих помилок, адаптувати найкращі індустріальні практики та виділити власні конкурентні переваги. Сучасний український ринок спеціалізованих інтернет-магазинів кави та чаю представлений значною кількістю рішень, проте більшість із них використовують стандартизовані платформні підходи, що накладає певні обмеження на гнучкість їхнього функціонування та розвитку.

1.2.1. Аналіз функціоналу конкурентів (thetea.ua, doppio.ua, coffee-craft.com.ua)

Першим вагомим гравцем на ринку є інтернет-магазин «The Tea» (thetea.ua), який спеціалізується на преміальних сортах чаю та кави. Цей ресурс має широку клієнтську базу та досить потужний каталог продукції. До його функціональних переваг можна віднести наявність деталізованих описів товарів, багаторівневе меню категорій та інтеграцію класичних методів оплати. Однак, інтерфейс системи виглядає дещо перевантаженим, а швидкість завантаження сторінок під час застосування складних фільтрів суттєво знижується. Відсутність персоналізованих розумних рекомендацій змушує клієнта самостійно вивчати величезні масиви тексту для вибору потрібного сорту.

Інший аналог – платформа «Doppio Coffee» (doppio.ua), яка має виражений фокус як на роздрібного, так і на гуртового (B2B) покупця. Функціонал даного сайту оптимізований під швидке повторення попередніх замовлень. Проте основним недоліком Doppio є недостатньо гнучка архітектура обробки динамічних кошиків користувача та відносно складний процес реєстрації. Сервіс також не пропонує передових методів інтеграції з хмарними штучними нейромережами для забезпечення інтерактивної взаємодії з клієнтами.

Третім конкурентом виступає магазин «Coffee Craft» (coffee-craft.com.ua), який позиціонує себе як локальний бренд крафтового обсмаження. Перевагою платформи є привабливий мінімалістичний дизайн, зручне відображення процесів обсмаження та зрозумілий процес додавання до кошика. Очевидним недоліком є обмежена гнучкість в управлінні профілем користувача та неможливість гнучкого масштабування, що часто стає проблемою під час сезонних пікових навантажень. Відсутність асинхронної обробки вебхуків для платежів іноді призводить до десинхронізації статусів транзакцій та реального стану замовлення в базі.

1.2.2. Порівняльний аналіз технологічних стеків та архітектурних підходів аналогів

Спільним знаменником більшості проаналізованих платформ є використання монолітних архітектур та готових рішень на базі популярних CMS (Content Management Systems), таких як OpenCart, WordPress (WooCommerce) чи Magento. Ці системи побудовані переважно з використанням мови програмування PHP та реляційних баз даних сімейства MySQL. Монолітна архітектура таких систем означає, що клієнтська частина (Frontend) та серверна бізнес-логіка (Backend) тісно переплетені в межах єдиної кодової бази та виконуються на одному фізичному чи віртуальному сервері. Хоча цей підхід дозволяє швидко розгорнути базовий магазин, він накладає жорсткі обмеження на продуктивність та масштабованість. Збільшення потоку клієнтів призводить до перевантаження серверів баз даних, оскільки CMS генерують велику кількість надлишкових SQL-запитів до бази. Більше того, монолітна побудова значно ускладнює впровадження новітніх технологій, таких як контейнеризація за допомогою Docker, розгортання окремих мікросервісів або інтеграція з ресурсоємними AI-провайдерами без ризику порушити працездатність основної системи. Натомість, проєктування сучасного сервісу вимагає використання мікросервісної орієнтації або строго структурованої RESTful API архітектури, розгорнутої на продуктивній платформі на кшталт .NET, що забезпечує чіткий поділ відповідальності між клієнтом та сервером.

1.2.3. Виявлення недоліків існуючих рішень та обґрунтування доцільності розробки власного продукту

Ґрунтовний аналіз представлених аналогічних розробок дозволив виявити низку системних недоліків, які є загальними для ринку. По-перше, абсолютна більшість магазинів позбавлена інструментів інтелектуальної допомоги на базі штучного інтелекту, змушуючи покупця покладатися виключно на статичні описи. По-друге, використання застарілих CMS створює загрози інформаційній безпеці та значно сповільнює час реакції сервісу на клієнтські HTTP-запити. По-третє, імплементація платіжних модулів часто відбувається через синхронні редіректи, що підвищує відсоток втрачених чи неоформлених до кінця замовлень у випадках нестабільного інтернет-з'єднання у клієнта.

На основі цих факторів розробка власного спеціалізованого програмного забезпечення є абсолютно доцільною та науково обґрунтованою. Створення масштабованого Backend-сервісу з використанням новітніх архітектурних шаблонів (зокрема, Clean Architecture та CQRS), реалізація надійного та асинхронного механізму оплати на базі Stripe Webhooks, а також впровадження інтелектуального AI-асистента дозволить створити продукт, що якісно перевершує існуючі ринкові аналоги як за швидкістю роботи, так і за рівнем користувацького досвіду та безпеки.

1.3. Постановка задачі на розробку

Створення складного інформаційного середовища вимагає чіткого формалізації вимог, які ставляться перед розроблюваною системою. Постановка задачі на розробку Backend-частини кав’ярні охоплює структуризацію функціональних критеріїв, визначення специфічних бізнес-потреб та ідентифікацію нефункціональних стандартів, яких має дотримуватися програмний код.

1.3.1. Формулювання основних функціональних вимог

Основними функціональними вимогами до серверної частини є забезпечення надійної обробки базових операцій зі створення, читання, оновлення та видалення об’єктів системи (CRUD). Бекенд повинен обробляти реєстрацію нових користувачів, зберігати їх персональні дані у захищеному вигляді за допомогою гешування паролів та реалізувати авторизацію через генерацію токенів доступу (JWT). Крім того, система має підтримувати повноцінну роботу електронного каталогу товарів: можливість категоризації, пошуку, фільтрації кави за различными критеріями. Важливою вимогою є реалізація збереження станів користувача – логіка роботи із кошиком товарів (Cart Items), можливість формування колекцій улюблених товарів (Favorites) та створення користувацького контенту у вигляді можливості залишити та редагувати відгуки до продукції (Reviews). Система повинна координувати створення структури замовлення (Order) з відповідним обчисленням фінансової інформації та географічних даних доставки.

1.3.2. Формулювання специфічних функціональних вимог

До специфічних вимог, які виділяють даний продукт з-поміж стандартних навчальних та комерційних проєктів, належить вимога глибокої інтеграції з міжнародною платіжною системою Stripe. Система повинна не просто перенаправляти користувача на сторінку оплати, а забезпечувати безпечну генерацію клієнтського секрету сесії та реалізувати окремий захищений ендпоінт для прослуховування вхідних повідомлень (Stripe Webhooks). Це необхідно для автоматичної зміни статусу замовлення виключно на основі криптографічно підтверджених сигналів від банку. Другою специфічною вимогою є імплементація AI-асистента в екосистему магазину. Сервер має приймати текстові запити від клієнта через API, попередньо формувати контекстно-залежний системний промпт, звертатися до зовнішніх сервісів обробки природної мови (таких як Nvidia API чи розширені моделі Gemini), отримувати векторну відповідь та повертати її клієнту,

емулюючи живе спілкування з професійним баристою. Програмний код повинен гарантувати приховування технічних ключів цих моделей від клієнта задля інформаційної безпеки.

1.3.3. Формулювання нефункціональних вимог

Нефункціональні вимоги описують якісні характеристики системи та методи забезпечення її життєздатності у довгостроковій перспективі. Продуктивність сервісу має бути оптимізована завдяки використанню асинхронного програмування у всіх запитах до бази даних та зовнішніх веб-служб. Безпека даних клієнтів повинна відповідати сучасним галузевим стандартам: паролі не можуть зберігатися у відкритому вигляді (лише зашифровані хеші), а комунікація між системами має здійснюватися виключно по захищеному протоколу HTTPS. З архітектурної точки зору вимагається суворе дотримання принципів "Чистої архітектури" (Clean Architecture) з метою розділення доменної логіки від інфраструктурних залежностей. Крім того, додаток має бути розроблений із застосуванням патерну CQRS з використанням бібліотеки MediatR для забезпечення максимальної ізоляції обробників запитів та команд. Для забезпечення легкості розгортання та масштабуваності на серверах під управлінням ОС Linux, весь розроблений код, залежності та середовище виконання повинні бути упаковані у Docker-контейнери із відповідними конфігураціями інфраструктури.

РОЗДІЛ 2. ПРОЄКТУВАННЯ BACKEND-СЕРВІСУ

2.1. Проєктування інформаційного забезпечення

Проєктування інформаційного забезпечення є фундаментальним, комплексним етапом розробки будь-якої сучасної інформаційної системи, який закладає базис для подальшого життєвого циклу програмного продукту. В умовах стрімкого розвитку електронної комерції, де обсяги інформації зростають експоненціально, інформаційне забезпечення виступає не просто як сховище даних, а як складна багаторівнева екосистема, що регламентує правила обробки, збереження та верифікації інформаційних потоків. Для спеціалізованого веб-сервісу з продажу кави якість проєктування бази даних безпосередньо впливає на швидкість генерації онлайн-вітрини, надійність фінансових транзакцій під час оформлення замовлень та безпеку конфіденційної інформації клієнтів. Саме на цьому етапі формується структурне розуміння того, яким чином абстрактні бізнес-вимоги перетворюються на жорстко типізовані таблиці реляційної системи управління базами даних. Процес проєктування передбачає системний перехід від узагальненого розуміння сутностей предметної області до створення суворої логічної моделі, яка враховує специфіку вибраної технологічної платформи, а саме ORM Entity Framework Core та СУБД PostgreSQL. Некоректне планування на цій стадії неминуче приведе до інформаційних аномалій, дублювання записів, порушення транзакційної цілісності та втрати даних при масштабуванні кав'ярні на нові ринки.

2.1.1. Концептуальне проєктування бази даних

Концептуальне проєктування бази даних є стартовим кроком у формуванні інформаційної моделі системи, під час якого відбувається виокремлення ключових об'єктів предметної області та визначення їхньої семантичної ролі в загальному процесі електронної торгівлі. Головна мета цього етапу полягає у створенні абстрактного уявлення про дані, повністю відв'язаного від специфіки конкретної

системи управління базами даних. У контексті розроблюваного Backend-сервісу кав’ярні ядром концептуальної моделі виступає сутність User (Користувач), яка абстрагує в собі всі автентифікаційні, авторизаційні та персональні ознаки клієнтів та адміністраторів. Для забезпечення принципу мінімальних привілеїв та розмежування прав доступу поруч із користувачем ідентифіковано сутність Role (Роль), а також специфічну сутність RefreshToken (Токен оновлення), яка концептуально відповідає за пролонгацію безпечних сесій без необхідності передачі критичних даних.

Наступним концептуальним блоком є товарний каталог, який безпосередньо генерує прибуток проєкту. Основною сутністю цього блоку виступає Product (Товар), що символізує фізичний продукт — кавові зерна, обладнання чи аксесуари. Для системної класифікації асортименту вводиться сутність Category (Категорія), що дозволяє формувати логічні ієрархічні дерева товарів. Клієнтський досвід взаємодії з каталогом розширюється завдяки концептуальним сутностям Review (Відгук) для накопичення соціальних оцінок продукції, FavoriteProduct (Улюблений товар) для формування персоналізованих списків бажань, а також ViewedProduct для збереження історії переглядів користувача задля подальшого аналізу алгоритмами рекомендацій. Фінальний, найважливіший блок стосується логіки торгівлі: він складається із сутностей CartItem (Елемент віртуального кошика), що акумулює наміри покупця, Order (Замовлення), яка фіксує момент угоди, та доповнюється специфічними логістичними об'єктами Location (Локація) та Warehouse (Склад) для управління географічними координатами доставки замовлення кінцевому споживачу.

2.1.2. Розробка логічної моделі даних

Перехід від концептуальної до логічної моделі даних передбачає точне визначення атрибутивного складу кожної виокремленої сутності та встановлення жорстких математичних взаємозв'язків між ними згідно з правилами реляційної алгебри та вимогами третьої нормальної форми (3NF). База даних розроблюваної системи спроєктована на основі підходу Code-First з використанням технології Entity

Framework Core, що дозволяє описувати логічну модель безпосередньо за допомогою класів мови програмування C# у шарі DataAccessLayer. Наприклад, логічна сутність UserEntity регламентує наявність унікального первинного ключа типу GUID, обов'язкових полів для збереження електронної пошти та захищеного гешу пароля, а також містить навігаційні властивості, що реалізують відношення «один до багатьох» із таблицями OrderEntity та ReviewEntity. Рівні доступу в системі формуються через багатомножинний зв'язок між таблицями Користувачів та Ролей, реалізований за допомогою неявної або явної проміжної таблиці з’єднання.

Логічна структура таблиці товарів (ProductEntity) включає не лише базові текстові та числові атрибути, але й складні зв'язки з колекціями візуальних матеріалів (ProductPhoto), дозволяючи зберігати динамічну кількість зображень для кожної пачки кави. Між сутностями CategoryEntity та ProductEntity встановлено суворий зв'язок «один до багатьох», що гарантує цілісність каталогу та запобігає появі товарів без визначеної групи. Найвибагливіша логіка звітна у таблиці замовлень (OrderEntity), яка діє як глобальний агрегатор. Кожен об'єкт замовлення зберігає в собі статичний зліпок фінансової інформації на момент здійснення транзакції, статус оплати (наприклад, стан процесингу платіжним шлюзом Stripe) та посилання на деталізовані записи про конкретні замовлені позиції. Така глибока нормалізація та типізація елементів забезпечує абсолютну транзакційну надійність при фінансових розрахунках і мінімізує імовірність критичних збоїв системи управління базами даних під навантаженням.

2.1.3. Схема бази даних

Для візуального представлення описаної складної логічної моделі даних та чіткої математичної демонстрації всіх інформаційних потоків і типів відношень між згенерованими таблицями було розроблено розширену ER-діаграму (Entity-Relationship diagram, модель «сутність-зв’язок») бази даних. Зазначена абстрактна схема наочно ілюструє ієрархію таблиць, наявність зовнішніх (Foreign Key) та первинних ключів (Primary Key), напрямки каскадного видалення записів

(Cascade Delete) задля підтримки посилальної цілісності в реляційній базі PostgreSQL, а також визначає обмеження унікальності для окремих колонок. Детальна концептуальна структура збереження даних та зв'язків між сутностями розробленої системи відображена на ER-діаграмі бази даних веб-сервісу, яку наведено на Рис. 2.1.

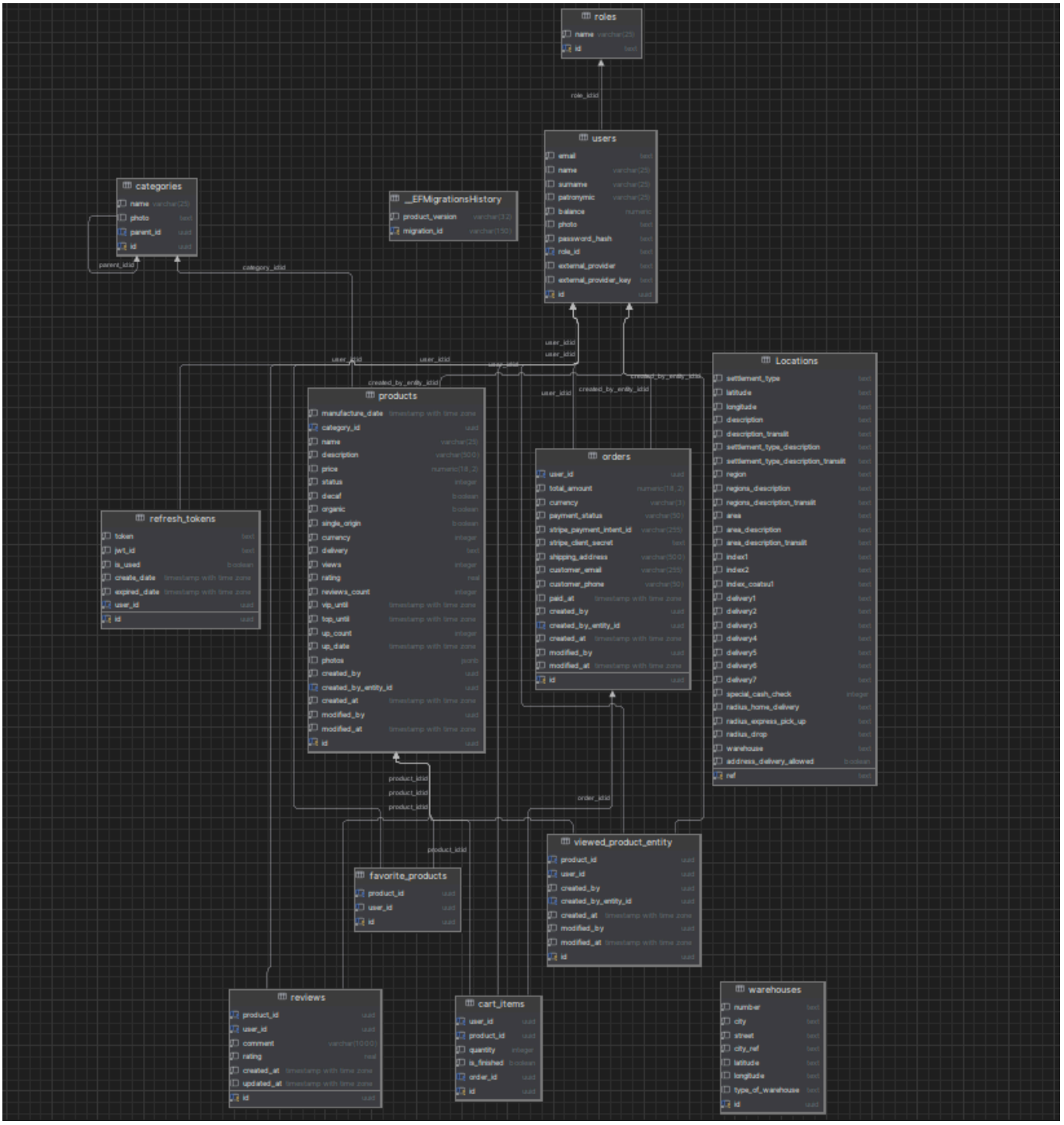

Рис. 2.1. ER-діаграма бази даних веб-сервісу

Джерело: [створено автором]

2.2. Проєктування архітектури програмного продукту

Архітектура програмного продукту є невидимим, але визначальним каркасом, який диктує правила організації вихідного коду, керує напрямками залежностей між окремими компонентами та визначає рівень складності можливого горизонтального масштабування сервісу в майбутньому. Відсутність чітко сформульованого архітектурного бачення на стартових етапах проєктування закономірно призводить до ефекту «спагетті-коду», де зміна логіки відображення даних може критично вплинути на алгоритми фінансових транзакцій або записи у базу даних. Для створення надійної Backend-частини кав’ярні, яка повинна безперебійно витримувати пікові навантаження під час масових розпродажів та рекламних кампаній, вимагався перехід від примітивних монолітних підходів до структурованих, корпоративних еталонів інженерії. Проєктування архітектури охоплює не лише вибір конкретних паттернів взаємодії між класами, але й логічний поділ всього рішення (Solution) на фізично ізольовані проєкти бібліотек класів (.csproj), кожен з яких наділений строгою, єдиною відповідальністю. Це дозволяє команді розробників паралельно працювати над різними модулями системи, легко впроваджувати безперервної інтеграції (CI/CD), що є абсолютним стандартом сучасної індустрії розробки комерційного програмного забезпечення.

2.2.1. Обґрунтування вибору архітектурного підходу Clean Architecture

Успішна технологічна реалізація сучасного потужного бекенд-сервісу немислима без вибору правильного архітектурного фундаменту, який би забезпечував стійкість кодової бази до закономірних ринкових змін. Для даного проєкту було аргументовано обрано методологію "Чистої архітектури" (Clean Architecture), розроблену та популяризовану видатним інженером Робертом С. Мартіном. Головним каталізатором такого вибору стала гостра необхідність повної, безумовної ізоляції бізнес-правил та доменної логіки електронного магазину кави від технологічних деталей швидкої реалізації. В сучасному вебі часто виникають

ситуації, коли бізнес вимагає термінової заміни платіжного провайдера, переходу на іншу систему управління базами даних чи інтеграції нових хмарних інструментів. У традиційних багатошарових (N-Tier) або монолітних архітектурах такі ініціативи викликають ланцюгову реакцію масштабного переписування коду по всьому проєкту, що веде до деградації надійності та появи прихованих помилок. Натомість, Clean Architecture концептуально базується на досконалому принципі інверсії залежностей (Dependency Inversion Principle зі складу SOLID). Суть правила залежностей (Dependency Rule) полягає в тому, що всі ієрархічні посилання в системі повинні вказувати виключно всередину – у бік доменного ядра додатка. Відповідно, зовнішні клієнтські інтерфейси (REST API) або інфраструктурні рішення (PostgreSQL, Docker) стають лише механізмами доставки повідомлень або зберігання даних, які повністю залежать від абстракцій бізнес-логіки, роблячи ядро програми феноменально гнучким, стійким і повністю відв’язаним від сторонніх фреймворків чи бібліотек.

2.2.2. Детальний опис шарів системи та їх взаємодії

Відповідно до імперативної парадигми Чистої архітектури, файлова та логічна структура розробленого рішення поділена на чотири незалежні, суворо регламентовані проєкти (шари). Серцем інженерної композиції є шар Domain. Він концептуалізує класи сутностей реального світу (Product, Category, Order, User), універсальні абстракції (IAuditableEntity, класи Result), а також інкапсулює фундаментальні моделі, домену. Цей шар не має абсолютно жодних посилань на сторонні проєкти в рішенні, гарантуючи свою кристальну чистоту.

Навколо домену обгортається шар Application (Прикладний шар), який містить бізнес-логіку системи та реалізує конкретні сценарії використання (Use Cases). Він визначає абстрактні контракти (інтерфейси) для роботи з базами даних (Repositories), описує об'єкти передачі даних (Data Transfer Objects — DTO), містить папки для Команд та Запитів (Commands/Queries), а також кастомні класи винятків

(Exceptions) та сервіси хешування. Аналогічно до Домену, Прикладний шар не має уявлення про імплементацію бази даних чи методи HTTP валідації.

Третім критичним компонентом є DataAccessLayer (або шар Інфраструктури/Доступу до даних), який розташований на зовнішньому кільці архітектури. Саме тут відбувається фізична віртуалізація уявних інтерфейсів з Application-шару: реалізуються репозиторії, налаштовується комунікація з базою даних через DbContext від Entity Framework Core, декларуються конфігурації табличних зв'язків та автоматично формуються файли міграцій схеми даних бази PostgreSQL.

Презентаційним та комунікаційним фронтиром усієї системи виступає шар Api. Він об'єднує в собі HTTP-контролери (такі як AccountController, ProductsController, AiController), класи централізованого впровадження залежностей (Dependency Injection Injection), механізми перехоплення винятків (Middlewares) та налаштування конвеєра обробки HTTP-запитів програми у файлі Program.cs. Інформаційна взаємодія протікає суворо за встановленим вектором: Api приймає запит клієнта і викликає абстрактні команди з Application, який за необхідності звертається до DataAccessLayer через інжектовані інтерфейси для отримання чи запису сутностей з Domain-шару.

2.2.3. Застосування патерну CQRS для розділення команд та запитів

Крилюче важливою архітектурною модернізацією, застосованою в розробленому бекенд-сервісі, є тотальна імплементація патерну CQRS (Command Query Responsibility Segregation — розділення відповідальності команд і запитів). Відомо, що у сфері електронної комерції статистичне співвідношення запитів на отримання інформації (наприклад, читання каталогу, перегляд кошика, пошук характеристик кави) до операцій безпосередньої модифікації даних (фактичне оформлення замовлення або видалення товару) може сягати пропорції сотень і тисяч до одного. Використання традиційних CRUD-орієнтованих архітектурних структур, де одні й ті самі сервісні класи обробляють як читання, так і запис, неминуче

провокує надмірне розростання коду, порушення принципу єдиної відповідальності та значне падіння продуктивності. Завдяки ідеології CQRS весь потік операцій у шарі Application був семантично та фізично поділений на дві ізольовані гілки.

Перша гілка – Команди (Commands) – включає операції, метою яких є зміна стану системи (операції Create, Update, Delete). Прикладом є класи CreateCartItemCommand, AddCategoryCommand або UpdatePaymentStatusCommand. Друга гілка – Запити (Queries) – відповідає суто за витягування об'єктів з бази без жодних модифікацій стану. В ролі головного диспетчера та внутрішнього системного маршрутизатора між Api-контролерами та обробниками цих об'єктів виступає сторонній медіатор (шаблон Mediator), реалізований за допомогою передової бібліотеки MediatR. Контролер шару презентації просто формує об'єкт потрібної команди і сліпо передає його медіатору, а сам MediatR аналізує тип об'єкта та гарантовано перенаправляє його до відповідного класу-обробника (Handler), де і зосереджена вузьконаправлена прикладна бізнес-логіка. Це робить контролери максимально «тонкими», а код бездоганно структурованим та придатним до паралельного розширення новими функціями.

2.2.4. Моделювання архітектури

З метою академічного обґрунтування застосованих складних архітектурних рішень та наглядної ілюстрації процесу синергічної взаємодії між розрізненими компонентами системи було розроблено набір специфічних діаграм. Графічне моделювання архітектури базується на UML-діаграмі класів, яка надзвичайно детально розкриває статичну структурну будову ієрархічно-підпорядкованого програмного продукту, показуючи інкапсуляцію закритих полів, відкриті методи абстрактних класів та формальну вичерпну реалізацію інтерфейсів репозиторіїв контролерами через проксі-шар медіатора MediatR, що наочно продемонстровано на Рис. 2.2. Своєю чергою, візуалізація класичного динамічного та асинхронного життєвого циклу HTTP-запиту, починаючи від точки входу у відповідний контролер комунікаційного шару API, проходячи через валідаційні конвеєри, виконуючи

збереження даних у реляційній таблиці PostgreSQL та гарантуючи успішне повернення стандартизованого результату клієнту, всебічно відображена на UML-діаграмі послідовності, яку наведено на Рис. 2.3.

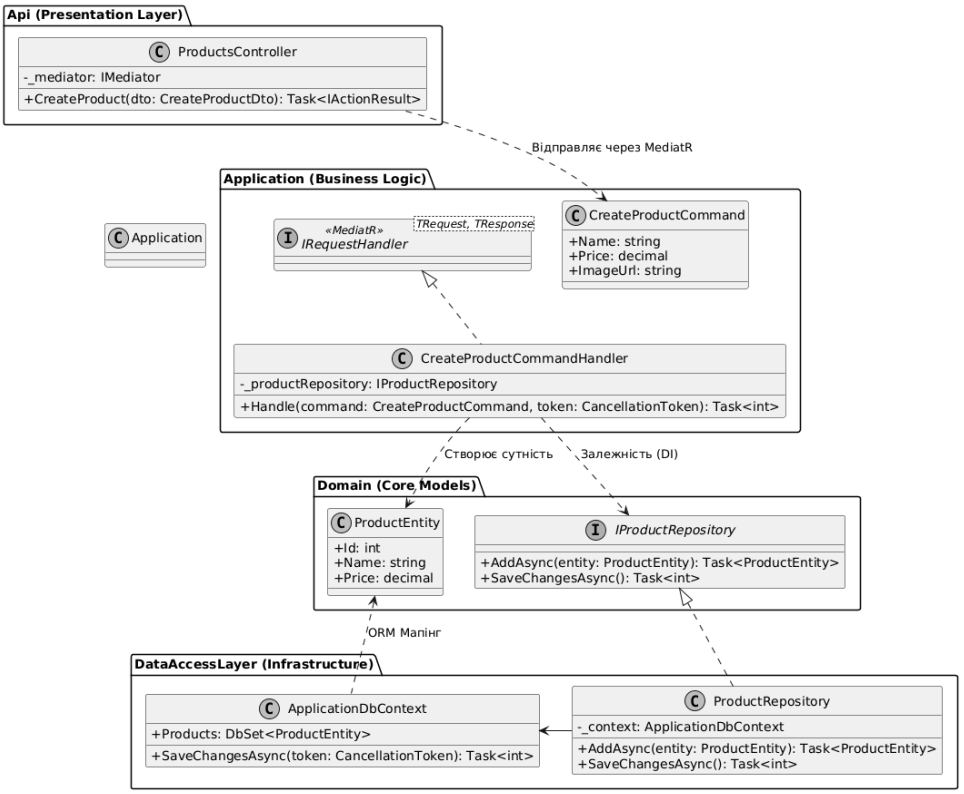

Рис. 2.2. UML-діаграма класів архітектури системи бекенду

Джерело: [створено автором]

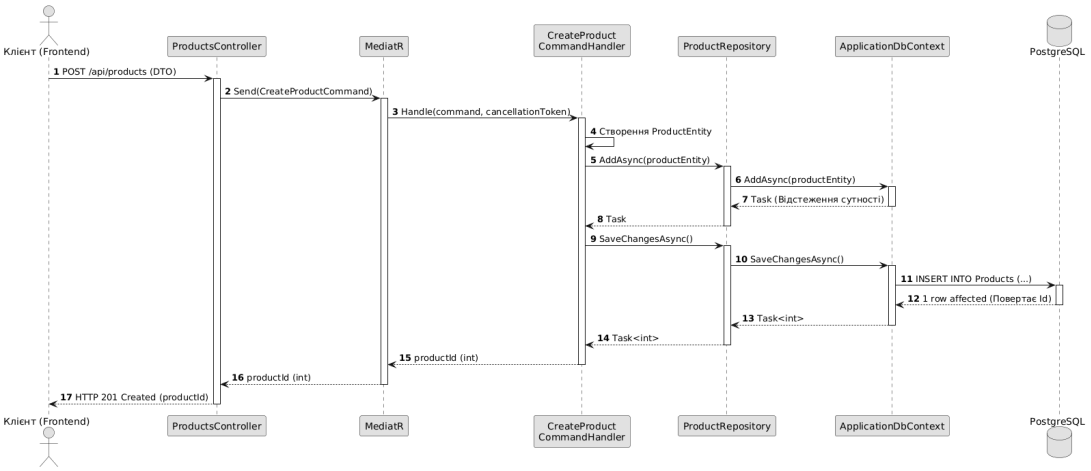

Рис. 2.3. UML-діаграма послідовності асинхронного життєвого циклу обробки запиту

Джерело: [створено автором]

2.3. Проєктування та опис ключових алгоритмів

Робота бекенд-сервісу кав'ярні не обмежується простою передачею інформації з бази даних до клієнта та у зворотному напрямку. Сучасні веб-системи насичені надскладною прикладною бізнес-логікою, яка вимагає строгих алгоритмічних послідовностей для забезпечення криптографічної безпеки, фінансової достовірності та інтелектуальної автоматизації. Алгоритмічне моделювання є необхідним інструментом для переведення загальних вимог бізнесу на мову програмування та формалізації реакцій системи на будь-які непередбачувані або зловмисні дії користувачів. Проєктування алгоритмів у цьому розрізі стосується не стандартного пошуку чи сортування (яке здебільшого виконується силами розроблених СУБД та ORM), а вирішення багатокомпонентних комунікаційних задач між ізольованими зовнішніми та внутрішніми сервісами. Непорушність послідовності кроків під час генерації сеансових ключів доступу, провадження крос-серверних перевірок платежів або обробки природної мови користувача алгоритмами штучного інтелекту є гарантом життєздатності комерційної платформи. Ефективність спроєктованих бізнес-алгоритмів безпосередньо впливає на час відповіді веб-ресурсу на запити (Response Time), його захист від стороннього втручання та унеможливлює виникнення парадоксів, таких як підробка статусу оплати замовлення. У наступних підпунктах деталізовано логічну структуру найважливіших алгоритмів, що визначають унікальність розробленого сервісу.

2.3.1. Алгоритм реєстрації та автентифікації користувача з використанням JWT

Забезпечення суворої безпеки та ідентифікації клієнтських додатків при взаємодії із серверною інфраструктурою базується на передовому алгоритмі stateless-автентифікації (без збереження стану на сервері) із застосуванням захищених JSON Web Tokens (JWT). Математичний алгоритм багатофакторної реєстрації починається з моменту ініціації HTTP POST запиту користувачем до

ендпоінту SignInCommand або SignUpCommand. На первинному етапі проміжні шари системи (Middlewares) проводять жорстку алгоритмічну валідацію вхідних DTO-даних на основі регулярних виразів, перевіряючи допустимість формату електронної пошти та вимоги складності до пароля. За умови успішного проходження цього барикадного фільтру прикладна логіка виконує звернення до бази даних для гарантування відсутності дублікатів ідентичних поштових адрес, що продемонстровано на блок-схемі алгоритму реєстрації нового користувача на Рис. 2.4. Найбільш критичним кроком є виконання одностороннього криптографічного хешування пароля клієнта у спеціалізованому сервісі HashPasswordService — база даних апріорі категорично не може зберігати відкриті текстові паролі.

Щойно сутність успішно зафіксована у масиві даних, ініціюється алгоритм генерації двох суміжних типів криптографічних токенів. Перший — швидкий Access Token (токен доступу), який алгоритмічно підписується секретним приватним ключем конфігурацій сервера і навмисне має вкрай малий термін дії технологічної активності. Зміст цього токена включає закодоване корисливе навантаження з ідентифікатором суб'єкта та матрицею його ролей доступу (наприклад, Admin), що дозволяє пришвидшити маршрутизацію без додаткових дорогих запитів до дискової підсистеми БД під час перевірки прав. Другий елемент пари — ліквідний Refresh Token, який є істинно довготривалим та надійно зберігається всередині таблиці RefreshTokens. Відповідно до алгоритму роботи сесії, якщо короткий Access Token об'єктивно закінчує свій термін системної дії, клієнтський додаток через фоновий процес генерує запит на відправлення утилітарного Refresh Token, тим самим безшовно породжує нову пару ключів та продовжує сеанс користувацької автентифікації.

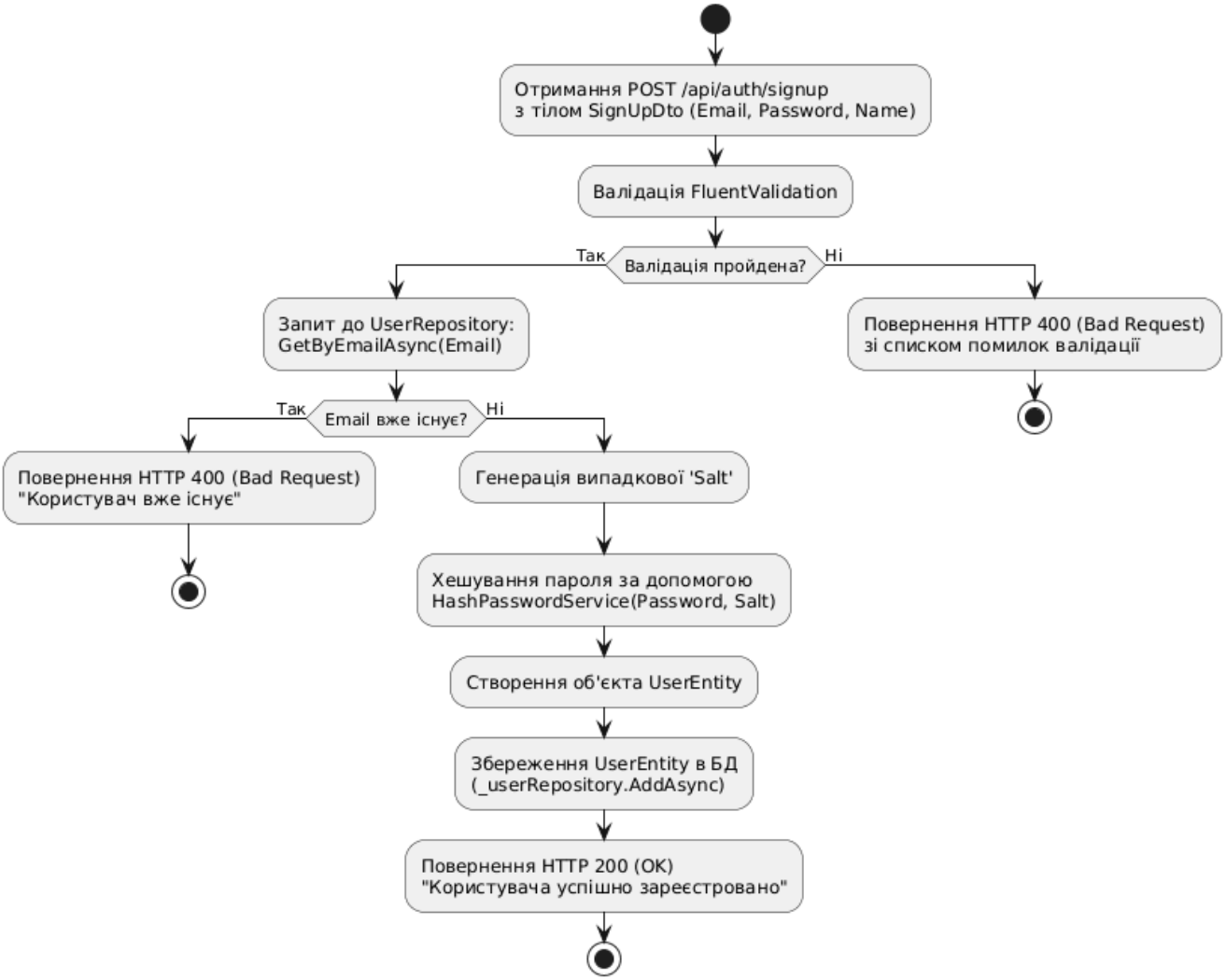

Рис. 2.4. Блок-схема алгоритму реєстрації нового користувача

Джерело: [створено автором]

2.3.2. Алгоритм обробки замовлення та інтеграції зі Stripe (через вебхуки)

Фізичне перетворення віртуальних товарів кошика клієнта в реальну фінансову транзакцію забезпечується через багатокомпонентний подійно-орієнтований алгоритм глибокої інтеграції з міжнародним платіжним провайдером Stripe. Традиційні парадигми інтернет-магазинів базуються на синхронному очікуванні підтвердження платежу прямо на сторінці клієнта, що є вкрай дефективним підходом та неодмінно призводить до величезних втрат замовлень у разі найменшого збою інтернет-зв'язку на мобільному пристрої

користувача. Розроблений бекенд-алгоритм вирішує цю фундаментальну проблему на основі асинхронних архітектурних механізмів Webhooks, повністю гарантуючи цілісність фінансових транзакцій.

Логічний процес стартує у той момент, коли клієнт ініціює команду CreateOrderCommand. Сервер читає і фіксує поточний статичний вміст актуального кошика сутностей CartItemEntity, на його основі генерує агреговану сутність OrderEntity зі статусом "Очікує на оплату", після чого самостійно звертається по закритому тунелю до Stripe API задля генерування закритого секрету наміру платежу (Payment Intent), що детально відображено на блок-схемі алгоритму створення нового замовлення (Рис. 2.5). Користувач здійснює безпечне введення фінансових реквізитів безпосередньо через вбудований віджет Stripe на сторінці фронтенду, стовідсотково минаючи інформаційні ланцюги розробленої кав’ярні (дотримання стандарту PCI DSS). Наступний блок алгоритму активується після фактичного списання коштів із картки банком: сервери еквайрингу Stripe ініціюють захищений асинхронний POST запит (Webhook) до спеціально обумовленого контролера на нашому бекенді. Центральним запобіжником цього алгоритмічного етапу є сувора криптографічна перевірка цифрового підпису даного вхідного запиту за допомогою публічного корпоративного ключа прив'язки Stripe, що виключає можливість імітації факту оплати неавторизованими зловмисниками. Лише після беззаперечного проходження математичної верифікації алгоритм ініціює команду UpdatePaymentStatusCommand, котра оновлює фінансову сутність замовлення у базі даних та сигналізує адміністративній панелі магазину про старт процесу пакування товарів.

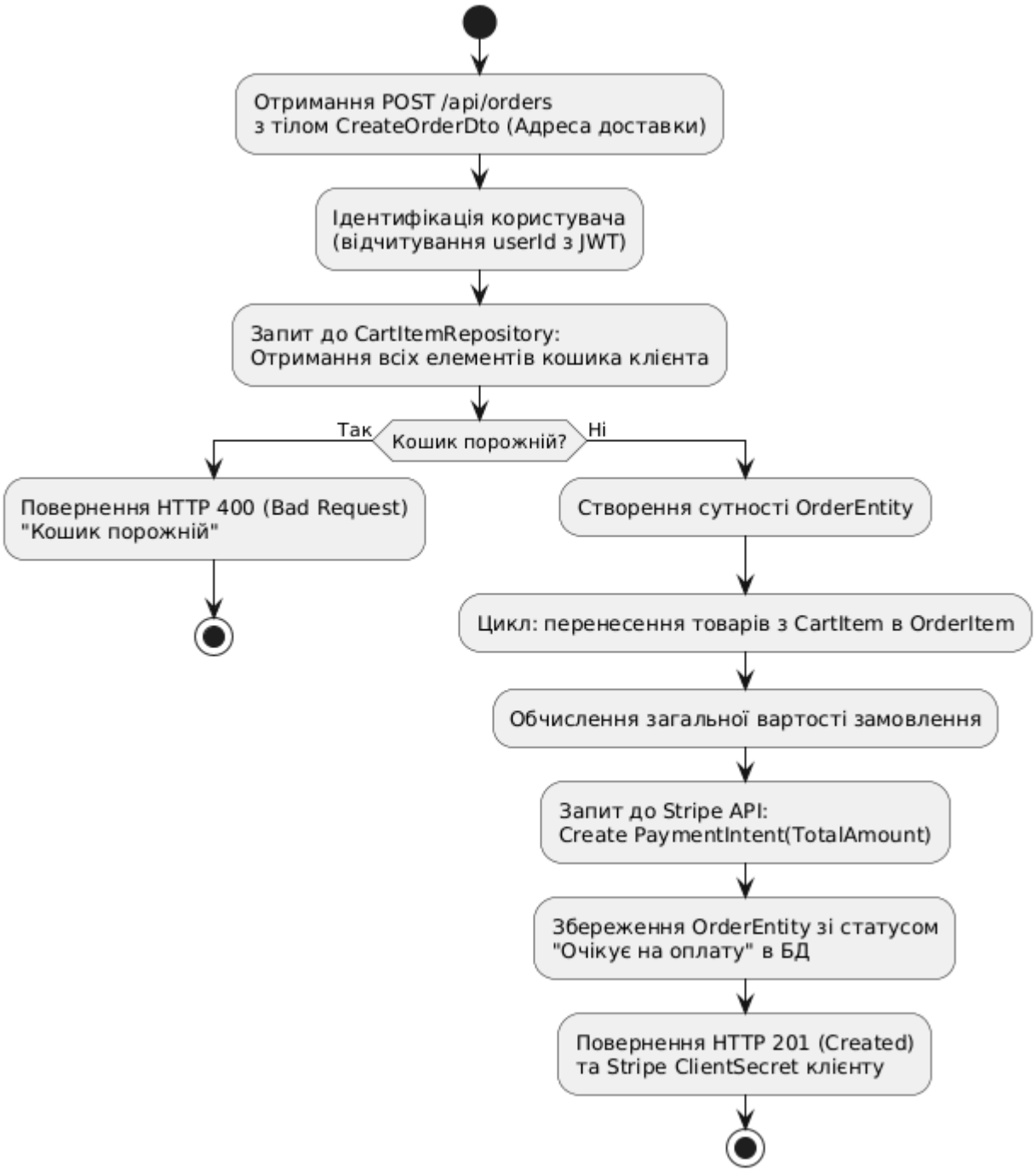

Рис. 2.5. Блок-схема алгоритму створення нового замовлення

Джерело: [створено автором]

2.3.3. Алгоритм взаємодії з AI-асистентом

Задля забезпечення високотехнологічного, унікального та високоякісного користувацького досвіду (User Experience) в середовищі кав’ярні, було синтезовано

унікальний алгоритм інтеграції потужностей штучного інтелекту в консервативну інфраструктуру веб-сервісу. Складний алгоритм реалізується за посередництва контролера AiController і функціонує в якості "розумного шлюзу" або проксі між кінцевим споживачем та віддаленою великою мовною моделлю (LLM), такою як gpt-oss-20b. Головною метою алгоритмізації на цьому етапі є захист ключів інтеграції та попереднє забезпечення контекстуалізації нейромережі.

Послідовність виконання розпочинається з отримання людського текстового запиту (промпта) від клієнта у формі JSON DTO, наприклад інструкції: "я обрав каву темного обсмаження, порекомендуй мені метод заварювання з нашого магазину". Алгоритм на стороні бекенда категорично не відправляє цей запит напряму хмарному провайдеру машинного навчання. Попередньо ініціюється підпрограма формування розширеного семантичного контексту: бекенд зшиває запит користувача зі складними прихованими інструкціями для машини (System Prompt), директивно обмежуючи її модель поведінки виключно амплуа "професійного баристи". Більш того, розроблений алгоритм може динамічно збагачувати цей промпт актуальною інформацією з бази даних (про асортимент, наявність тощо). Масштабний пакет сформованих даних маркується секретним API-ключем авторизації моделі, який надійно схований в середовищі розгортання, і ретранслюється на сервери нейромережі. Після успішного отримання та десеріалізації багаторівневої JSON-відповіді або потоку токенів генерації, алгоритм відсікає непотрібні статистичні та службові метадані від LLM (як у форматі об'єктів GenerateContentResponse), здійснює фінальне форматування структури у клієнтський об'єкт NvidiaResponse або подібний DTO і тільки тоді передає готову текстову рекомендацію назад клієнту. В такий спосіб архітектура бекенду забезпечує абсолютний контроль інформаційної безпеки, ізоляцію векторів діалогу та неперервність інтелектуальної допомоги на платформі.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1. Реалізація архітектурних рішень та базової інфраструктури проєкту

Процес програмної реалізації розробленої системи розпочався з фундаментального етапу конструювання базового каркаса проєкту відповідно до принципів чистої архітектури, що вимагало створення чітко розмежованих та ізольованих просторів імен для кожного логічного рівня застосунку. Використовуючи інструментарій командного рядка платформи .NET та інтегроване середовище розробки, було згенеровано основне рішення, до складу якого увійшли окремі проєкти типу бібліотек класів для доменного рівня, рівня застосунку, інфраструктурного рівня, рівня доступу до даних, а також головний виконавчий проєкт веб-інтерфейсу прикладного програмування, який слугує вхідною точкою для всіх зовнішніх запитів. Доменний рівень був ізольований від будь-яких зовнішніх залежностей, містячи виключно чисті сутності предметної області, такі як товари, категорії, замовлення, елементи кошика та користувачі кав'ярні, що гарантує високу стабільність найважливіших бізнес-правил незалежно від обраних технологій збереження даних чи форматів передачі інформації.

Наступним кроком стало налаштування рівня доступу до даних за допомогою об'єктно-реляційного відображення Entity Framework Core, що дозволило абстрагуватися від низькорівневих SQL-запитів і працювати з базою даних об'єктно-орієнтованим способом. Для забезпечення надійного підключення до реляційної системи управління базами даних PostgreSQL було реалізовано клас контексту бази даних, який успадковує базовий функціонал мікропрограмного фреймворку та конфігурує кожну сутність за допомогою механізму Fluent API. Конфігурація впровадження залежностей для рівня збереження даних була винесена в окремий інфраструктурний клас, який реєструє контекст бази даних та всі реалізації патерну репозиторій у колекції сервісів платформи. Такий підхід забезпечує дотримання принципу інверсії залежностей, оскільки вищі рівні архітектури залежать виключно від інтерфейсів репозиторіїв, а не від їх конкретних

імплементацій, що суттєво полегшує подальше тестування та модифікацію системи без необхідності втручання в бізнес-логіку.

Лістинг 3.1. Конфігурація інфраструктури збереження даних у класі ConfigurePersistence

```
using DataAccessLayer.Data;
using DataAccessLayer.Repositories;
using Domain.Common.Interfaces.Repositories;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace DataAccessLayer
{
    public static class ConfigurePersistence
    {
        public static IServiceCollection AddPersistenceServices(this
IServiceCollection services, IConfiguration configuration)
        {
            services.AddDbContext<ApplicationDbContext>(options =>

options.UseNpgsql(configuration.GetConnectionString("Default"),
                    builder =>
builder.MigrationsAssembly(typeof(ApplicationDbContext).Assembly.FullName))
);

            services.AddScoped<IProductRepository, ProductRepository>();
            services.AddScoped<ICategoryRepository, CategoryRepository>();
            services.AddScoped<IOrderRepository, OrderRepository>();
            services.AddScoped<IUserRepository, UserRepository>();
            services.AddScoped<ICartItemRepository, CartItemRepository>();

            return services;
        }
    }
}
```

Джерело: [створено автором]

Окрім налаштування бази даних, значна увага була приділена організації конвеєра обробки HTTP-запитів у головному конфігураційному файлі застосунку. Тут здійснюється підключення проміжних програм для розпізнавання помилок, валідації вхідних даних, забезпечення механізмів автентифікації на основі

веб-токенів стандарту JSON та маршрутизації контролерів. Кожен етап конвеєра ретельно налаштований для забезпечення максимальної швидкодії та безпеки, зокрема реалізовано глобальний перехоплювач винятків, який замінює стандартні системні повідомлення про помилки на стандартизовані відповіді у форматі, зрозумілому для клієнтських додатків, запобігаючи витоку чутливої системної інформації у виробничому середовищі. Інфраструктурний рівень також був доповнений сервісами для роботи із зовнішніми системами, включаючи сервіси для генерації токенів доступу, обробки зображень товарів та шифрування паролів користувачів кав'ярні.

3.2. Розробка бізнес-логіки за допомогою патерну CQRS

Усвідомлюючи складність процесів електронної комерції та необхідність чіткого структурування операцій модифікації та читання даних, для реалізації рівня бізнес-логіки було обрано архітектурний шаблон відокремлення відповідальності за команди та запити (CQRS). Даний патерн був імплементований за допомогою популярної бібліотеки-медіатора MediatR, яка забезпечує слабку зв'язність між компонентами системи шляхом реалізації патерну посередника. Замість того, щоб контролери безпосередньо взаємодіяли з репозиторіями чи складними сервісними класами, вони лише формують об'єкти команд або запитів і передають їх системному медіатору, який маршрутизує ці об'єкти до відповідних ізольованих обробників, що містять виключно ту логіку, яка стосується конкретної операції.

Кожна операція у системі, наприклад, створення нової категорії товарів для кавового меню, представлена у вигляді окремого класу команди, який інкапсулює всі необхідні для виконання параметри. Обробник цієї команди містить алгоритм перевірки унікальності назви категорії, створення доменного об'єкта, його збереження в репозиторії та повернення ідентифікатора створеного ресурсу. Такий атомарний підхід робить код надзвичайно читабельним та підтримуваним, оскільки будь-яка зміна у бізнес-логіці створення категорії не впливає на інші частини застосунку, зводячи ризик несподіваних побічних ефектів до мінімуму. Важливою

складовою цього процесу є також автоматична валідація вхідних даних перед їх передачею до обробника команди, яка реалізована за допомогою бібліотеки FluentValidation та інтегрована у конвеєр виконання медіатора як окрема поведінка валідації.

Лістинг 3.2. Клас команди та обробника для створення нової категорії товарів

```
using Domain.Categories;
using Domain.Common.Interfaces.Repositories;
using MediatR;
using System.Threading;
using System.Threading.Tasks;

namespace Application.Commands.Categories.Commands
{
    public class AddCategoryCommand : IRequest<int>
    {
        public string Name { get; set; }
        public string Description { get; set; }
        public string ImageUrl { get; set; }
    }

    public class AddCategoryCommandHandler :
IRequestHandler<AddCategoryCommand, int>
    {
        private readonly ICategoryRepository _categoryRepository;

        public AddCategoryCommandHandler(ICategoryRepository
categoryRepository)
        {
            _categoryRepository = categoryRepository;
        }

        public async Task<int> Handle(AddCategoryCommand request,
CancellationToken cancellationToken)
        {
            var categoryEntity = new Category
            {
                Name = request.Name,
                Description = request.Description,
                ImageUrl = request.ImageUrl
            };
```

```
            var createdCategory = await
_categoryRepository.AddAsync(categoryEntity);
            await _categoryRepository.SaveChangesAsync();

            return createdCategory.Id;
        }
    }
}
```

Джерело: [створено автором]

Розв'язання проблеми консистентності даних та транзакційності також здійснюється на рівні обробників команд, що дозволяє групувати декілька операцій над репозиторіями у єдину логічну транзакцію. Використання сучасних можливостей мови програмування C# дозволило створити виразний та лаконічний код, який легко сприймається новими розробниками у випадку масштабування команди. Впровадження патерну медіатора суттєво розвантажило контролери прикладного програмного інтерфейсу, перетворивши їх на тонкі адаптери, єдиним завданням яких є прийом HTTP-запитів, перетворення їхнього тіла на об'єкти команд або запитів та повернення відповідних HTTP-статусів на основі результатів, отриманих від рівня застосунку.

3.3. Інтеграція штучного інтелекту та платіжних систем

Надання конкурентних переваг розробленому вебсервісу кав'ярні зумовило необхідність впровадження інноваційних технологічних рішень, зокрема інтеграції розширених можливостей штучного інтелекту для покращення користувацького досвіду та автоматизованої обробки платежів. Архітектура застосунку передбачає наявність спеціалізованого модуля для комунікації із зовнішніми моделями великих мовних мереж (LLM), який слугує інтелектуальним баристою-асистентом. Цей функціонал реалізований через захищений програмний шлюз на бекенді, де користувацькі повідомлення збагачуються заздалегідь підготовленими системними інструкціями, так званим «системним промптом», який суворо обмежує тематику

генерації тексту виключно аспектами вибору кави, рецептур кавових напоїв та асортименту закладу.

Лістинг 3.3. Об'єкт передачі даних для запиту до зовнішньої нейромережі

```
using System.Collections.Generic;

namespace Api.Dtos.Ai
{
    public class ChatRequest
    {
        public List<Message> Messages { get; set; }
    }

    public class Message
    {
        public string Role { get; set; }
        public string Content { get; set; }
    }

    public class GenerateContentResponse
    {
        public List<Candidate> Candidates { get; set; }
    }

    public class Candidate
    {
        public Message Message { get; set; }
        public string FinishReason { get; set; }
    }
}
```

Джерело: [створено автором]

Не менш критичним етапом стала програмна реалізація безпечного та безперебійного процесу приймання електронних платежів, що була здійснена шляхом інтеграції з глобальною платіжною платформою Stripe. Враховуючи високі вимоги до фінансової безпеки, процес оплати був розділений на два асинхронні етапи: створення наміру платежу (Payment Intent) на стороні сервера перед безпосереднім зняттям коштів та подальша валідація успішності транзакції за допомогою механізму вебхуків. Спеціально розроблений контролер вебхуків прослуховує вхідні події від платіжного шлюзу, застосовує криптографічну

перевірку підпису запиту для підтвердження його автентичності від серверів Stripe, і лише після успішної валідації ініціює команду оновлення статусу замовлення у базі даних, автоматично запускаючи бізнес-процеси формування чека та відправлення повідомлення баристі на приготування замовлення.

3.4. Забезпечення управління контейнеризацією та розгортанням застосунку

Для вирішення класичної проблеми несумісності середовищ розробки та промислового розгортання, а також для гарантії безперебійної роботи всіх компонентів вебсервісу продажів кави, було застосовано технологію апаратної віртуалізації на рівні операційної системи – контейнеризацію на базі платформи Docker. Процес створення образу серверного застосунку базується на багатоетапній збірці, прописаній у конфігураційному файлі Dockerfile, що дозволяє радикально зменшити кінцевий розмір контейнера та підвищити його безпеку шляхом відсічення всіх інструментів компіляції та вихідних кодів з фінального образу, залишаючи лише скомпільовані бінарні файли та середовище виконання .NET Runtime. Запуск контейнера налаштовано здійснювати від імені непривілейованого користувача операційної системи, що блокує потенційні вектори атак, пов'язані з ескалацією привілеїв у випадку компрометації вебзастосунку.

Лістинг 3.4. Конфігурація розгортання застосунку та бази даних у файлі compose.yaml

```yaml
services:
  postgres:
    container_name: db_container
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
    volumes:
      - postgres_data:/var/lib/postgresql
    networks:
```

```
      - dashboard_network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  webapi:
    container_name: api_container
    image: mishalolka299/app_api
    ports:
      - "5001:8080"
    depends_on:
      postgres:
        condition: service_healthy
    environment:
      ASPNETCORE_ENVIRONMENT: "Production"
      ConnectionStrings__Default:
"Server=postgres;Port=5432;Database=postgres;User
Id=postgres;Password=postgres;"
    networks:
      - dashboard_network

networks:
  dashboard_network:

volumes:
  postgres_data:
```

Джерело: [створено автором]

Оркестрація та взаємодія між компонентами системи автоматизована за допомогою інструменту Docker Compose, який дозволяє визначити всю інфраструктурну топологію проєкту в єдиному декларативному конфігураційному файлі. У цьому маніфесті детально описано два ключових сервіси: контейнер реляційної системи управління базами даних PostgreSQL оптимізованої версії на базі Alpine Linux та сам контейнер основного веб-інтерфейсу прикладного програмування. Для забезпечення надійності старту системи налаштовано механізм перевірки працездатності бази даних, який блокує запуск системи управління бізнес-логікою до моменту повної готовності сервера баз даних приймати підключення. Також передбачено ізольовану віртуальну мережу для безпечної

маршрутизації трафіку виключно між компонентами сервісу та зовнішній підключений том даних для забезпечення постійного зберігання інформації при перезавантаженні контейнерів, запобігаючи втраті користувацьких кошиків, замовлень та асортименту кав'ярні.

3.5. Засоби тестування та перевірки функціональності веб-сервісу

Фінальним, але критично необхідним етапом життєвого циклу розробки бекенду стало комплексне тестування розробленої системи для забезпечення її високої якості, надійності та відповідності сформованим бізнес-вимогам. Основним інструментарієм для проведення ручного та функціонального тестування прикладних програмних інтерфейсів стала інтегрована платформа Swagger UI, яка автоматично генерує інтерактивну документацію на основі анотацій маршрутизації контролерів у вихідному коді застосунку відповідно до загальноприйнятого стандарту специфікації OpenAPI. Підключення цього інструменту в конвеєр обробки запитів на етапі запуску додатку дозволило створити зручний візуальний інтерфейс прямо у вікні веббраузера, де розробники або тестувальники можуть переглянути всі доступні кінцеві точки системи обліку кави, детально дослідити очікувані схеми об'єктів запитів та відповідей, а також самостійно ініціювати виконання будь-яких HTTP-команд без необхідності залучення сторонніх клієнтських програм.

Процедура тестування охоплювала послідовну перевірку всіх критичних сценаріїв використання веб-сервісу, починаючи від реєстрації нових клієнтів закладу, авторизації за допомогою генерації та валідації цифрових підписів у токенах доступу формату JSON Web Token. Після успішного підтвердження особи тестувальника у заголовках до авторизаційного шлюзу Swagger, здійснювалися запити до захищених кінцевих точок для тестування процесів створення нових товарних позицій у меню, додавання капілярних фотографій для кавових напоїв, модифікації кошика та формування фінальних замовлень. Кожен відповідний відгук із сервера ретельно аналізувався на предмет відповідності визначеним статусним кодам протоколу передачі гіпертексту, швидкості відгуку пристрою збереження

даних, коректності сериалізації вихідних моделей у формат JSON та адекватності реакції системи на помилкові, неповні чи зловмисні вхідні параметри, гарантуючи стійкість бекенду перед непередбаченими обставинами в умовах реальної експлуатації кінцевими споживачами кав'ярні. Спільне використання контейнерного віртуального середовища разом із автоматизованими засобами опису прикладного програмного інтерфейсу надало змогу провести повну симуляцію промислового середовища та впевнитись у високій готовності продукту до фінального розгортання та виходу на ринок.

3.6. Опис взаємодії з прикладним програмним інтерфейсом (API) через Swagger

З метою забезпечення максимальної зручності під час перевірки, налагодження та демонстрації всіх розроблених сервісів серверної частини системи було застосовано передовий інструментарій інтерактивної документації Swagger, який базується на відкритій специфікації OpenAPI. Цей візуальний інтерфейс відіграє надзвичайно важливу роль у процесі розробки веб-застосунків, оскільки він дозволяє як розробникам, так і потенційним тестувальникам чи клієнтським програмам чітко розуміти логічну структуру запитів, очікувані формати даних, обов'язкові параметри та типи можливих відповідей без необхідності звертатися до вихідного програмного коду напряму. Кожен окремий контролер прикладного програмного інтерфейсу відображається у вигляді інкапсульованого блоку, що містить повний перелік маршрутів (ендпоінтів) із зазначенням конкретного протокольного методу передачі даних: отримання (GET), створення (POST), оновлення (PUT) або ж видалення (DELETE) відповідних ресурсів інформаційної бази.

Фундаментальним аспектом безпеки нашої системи, який яскраво прослідковується під час роботи з документацією Swagger, є процес автентифікації користувачів, інкапсульований у логічному просторі контролера AccountController. Для побудови потужного та стійкого до зовнішніх загроз бар'єру конфіденційності в

екосистемі сервісу використовується механізм передачі авторизаційних даних за допомогою самодостатніх токенів формату JSON Web Token (JWT). Даний токен являє собою оптимізований та криптографічно підписаний рядок символів, що складається із заголовка, корисного навантаження (визначення ідентифікатора клієнта та його авторизаційної ролі) та унікального цифрового підпису, згенерованого закритим серверним ключем. Такий підхід повністю нівелює необхідність створення класичних сесій локального збереження даних, оскільки після успішної верифікації облікових даних клієнта (чи то електронної пошти та пароля, чи використання сторонніх сервісів єдиного входу), сервер видає йому готовий тимчасовий пропуск. Щоразу, коли актор системи намагається отримати доступ до ізольованих чи конфіденційних ресурсів сервісу (наприклад, доступ до історії замовлень), він зобов'язаний закласти отриманий токен у головний заголовок авторизації HTTP-запиту. Інтерфейс Swagger надає зручну кнопку авторизації на верхній панелі, де розробник може одноразово ввести токен, після чого графічна платформа автоматично додаватиме його до всіх наступних звернень; цей механізм контролю ідеально візуалізований на Рис. 3.1.

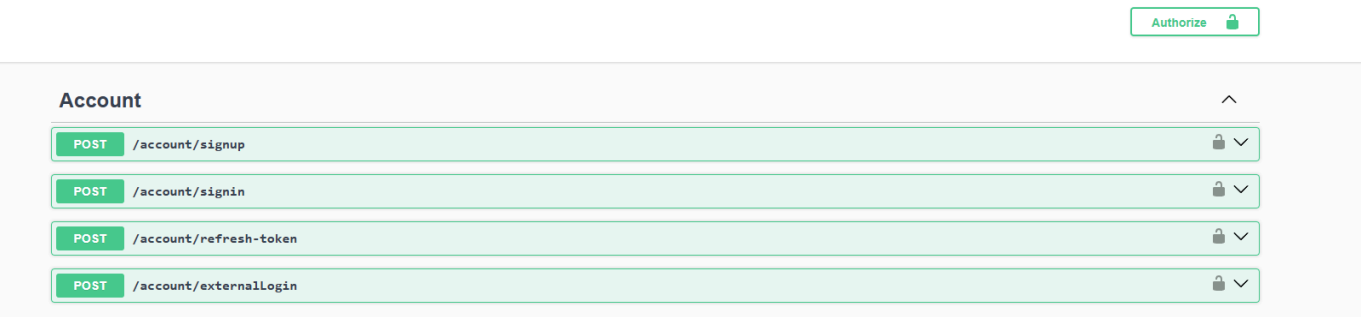

Рис. 3.1. Документація в Swagger контролера AccountController

Джерело: [створено автором]

Для управління товарним асортиментом інтернет-магазину та формування електронної вітрини кав'ярні система покладається на роботу взаємопов'язаних вузлів ProductsController та CategoriesController. Дані контролери формують єдиний простір для взаємодії з каталогом, дозволяючи виконувати глибоку фільтрацію, динамічне сортування кавових зерен за ступенем обсмаження чи країною

походження, а також реалізують пагінацію результатів для уникнення перевантаження мережевого каналу при великій кількості записів. Додатковий рівень адміністрування, який також вимагає наявності відповідних прав доступу в генерованому JWT-токені, розкриває можливості для додавання нових позицій до бази, оновлення текстових специфікацій товару чи прив'язки багатовимірних структур фотографій. Документація цих двох контролерів з повним переліком специфікацій тіл запитів для безперешкодної зміни каталогу наглядно подана на Рис. 3.2.

Рис. 3.2. Документація в Swagger контролерів CategoriesController, ProductsController
Джерело: [створено автором]

Важливим і без перебільшення критичним блоком взаємодії є контролери, що безпосередньо впливають на комерційну життєздатність проєкту: CartItemsController для гнучкого управління електронним кошиком, FavoritesController для збереження намірів улюблених товарів та OrderController для агрегації транзакцій. Процес покупки через ці кінцеві точки виглядає як послідовна естафета даних: від

додавання абстрактного товару у кошик до створення повномасштабного об'єкта замовлення із залученням логістичних модулів системи. Особливістю модуля оформлення фінансових угод є його алгоритмічне підключення до міжнародного платіжного шлюзу, що дозволяє сервісу самостійно створювати наміри платежу (Payment Intent) та гарантувати асинхронне отримання серверних вебхуків після повного зарахування банківських коштів, що виключає ризик втрати інформації через розірвання зв'язку з фронтенд-додатком. Архітектура та функціонал зазначених сервісів задокументовані на Рис. 3.3.

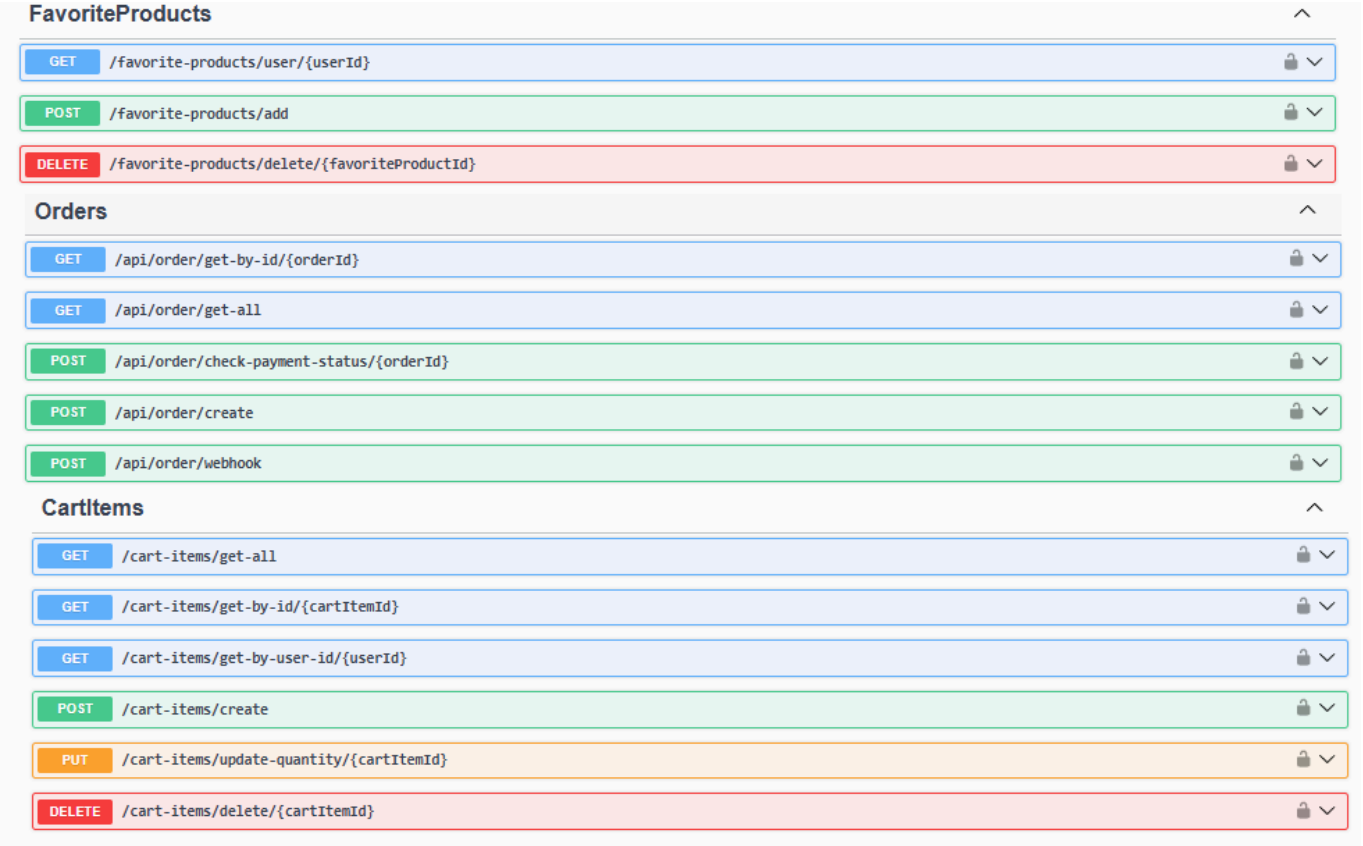

Рис. 3.3. Документація в Swagger контролерів FavoriteProductsController, OrdersController, CartItemsController

Джерело: [створено автором]

Насамкінець, інноваційна сутність проєкту, що кардинально вирізняє його на фоні традиційних ринкових аналогів, імплементована в межах контролера AiController. Саме ця точка доступу приймає текстові повідомлення від клієнта і, виконуючи роль суворого інформаційного менеджера, генерує розширений об'єднаний промпт, додаючи інструкції для дотримання імітаційної моделі поведінки

"професійного баристи". Лише після цього система ініціює прихований та безпечний виклик до зовнішнього хмарного провайдера машинного навчання для генерації комплексної аналітичної відповіді, результати якої серіалізуються та передаються назад користувачеві як звичайний зрозумілий текст. Очевидна потужність цієї інтеграції та схема викликів алгоритмів штучного інтелекту доступна для огляду на Рис. 3.4.

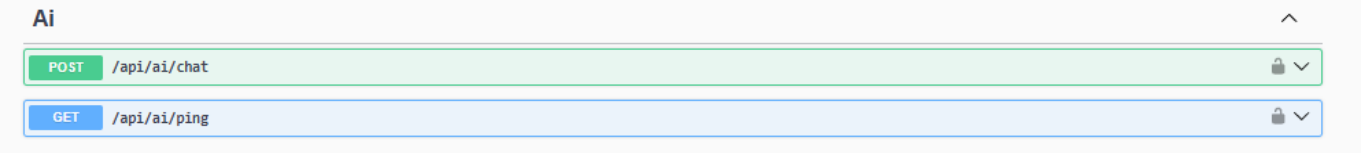

Рис. 3.4. Документація в Swagger контролера AiController

Джерело: [створено автором]

ЗАГАЛЬНІ ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було успішно досягнуто поставлену мету: спроєктовано та розроблено програмний продукт у вигляді backend-частини веб-сервісу для продажу кави. Результатом роботи є повноцінний, надійний та готовий до інтеграції з клієнтськими додатками серверний застосунок, що реалізує весь необхідний функціонал для функціонування сучасного інтернет-магазину.

На першому етапі роботи було проведено комплексний аналіз предметної області. Було досліджено основні бізнес-процеси, характерні для сфери електронної комерції, та виявлено ключові вимоги до системи. Аналіз аналогічних рішень на ринку дозволив визначити стандартний набір функцій, очікуваний користувачами, та обґрунтувати доцільність розробки власного гнучкого та масштабованого рішення, на відміну від використання готових платформ.

На етапі проєктування було закладено архітектурний фундамент системи. В якості основи було обрано підхід Clean Architecture, що дозволило створити слабкозв'язану, модульну та тестовану систему з чітким розділенням відповідальності між шарами: доменним (Domain), прикладним (Application), доступу до даних (DataAccessLayer) та представлення (Api). Було спроєктовано реляційну модель даних, що адекватно відображає всі сутності предметної області, та розроблено детальні алгоритми для ключових бізнес-процесів, таких як реєстрація користувача та створення замовлення.

Практична частина роботи була присвячена безпосередній реалізації спроєктованої системи з використанням сучасного технологічного стеку. В якості платформи було обрано .NET 8 та мову C#, для створення API — фреймворк ASP.NET Core, а для взаємодії з базою даних PostgreSQL — ORM Entity Framework Core. Було реалізовано RESTful API, що включає:

Систему аутентифікації та авторизації на основі JWT з підтримкою ролей.
Повноцінний CRUD-функціонал для управління товарами та категоріями.
Логіку для роботи з кошиком, списком обраних товарів та відгуками.

Механізм створення та відстеження статусу замовлень.

Особливу увагу було приділено якості коду та дотриманню найкращих практик розробки, зокрема патернів CQRS та Repository. Для забезпечення простоти розгортання та портативності проєкт було підготовлено до контейнеризації за допомогою Docker.

Таким чином, у результаті виконання кваліфікаційної роботи було створено програмний продукт, який повністю відповідає поставленим на початку вимогам. Розроблений backend-сервіс є надійною та гнучкою основою для повноцінного веб-сервісу електронної комерції. Подальший розвиток проєкту може включати розширення функціоналу (наприклад, інтеграція з платіжними системами, реалізація системи промокодів), оптимізацію продуктивності під високі навантаження та розробку клієнтських додатків (веб та мобільних), які будуть взаємодіяти з розробленим API.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft Learn. ASP.NET Core Documentation. URL: https://learn.microsoft.com/en-us/aspnet/core/ (Дата звернення: 07.02.2026).
2. Microsoft Learn. Entity Framework Core Documentation. URL: https://learn.microsoft.com/en-us/ef/core/ (Дата звернення: 21.01.2026).
3. Microsoft Learn. C# Guide. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (Дата звернення: 10.03.2026).
4. Microsoft Learn. Clean Architecture and Layered Design Patterns. URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (Дата звернення: 11.03.2026).
5. Microsoft Learn. Tutorial: Create a web API with ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api (Дата звернення: 23.03.2026).
6. Microsoft Learn. Command and Query Responsibility Segregation (CQRS) pattern. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs (Дата звернення: 01.04.2026).
7. Microsoft Learn. Secure APIs with .NET and JWT. URL: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/jwt-authn (Дата звернення: 02.04.2026).
8. Microsoft API-Guidelines. Microsoft REST API Guidelines. URL: https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md (Дата звернення: 09.03.2026).
9. Docker Documentation. Containerizing an ASP.NET Core application. URL: https://docs.docker.com/language/dotnet/ (Дата звернення: 08.03.2026).
10. PostgreSQL. Official PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (Дата звернення: 09.03.2026).
11. Npgsql. .NET Data Provider for PostgreSQL. URL: https://www.npgsql.org/ (Дата звернення: 10.03.2026).

12. Bogard, J. MediatR Library for .NET. URL: https://github.com/jbogard/MediatR (Дата звернення: 11.03.2026).
13. JWT.io. JSON Web Token Introduction. URL: https://jwt.io/introduction/ (Дата звернення: 12.03.2026).
14. Swagger. OpenAPI & Swagger Documentation. URL: https://swagger.io/docs/specification/about/ (Дата звернення: 06.03.2026).
15. Martin, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall. URL: https://www.oreilly.com/library/view/clean-architecture-a/9780134494272/ (Дата звернення: 15.03.2026).
16. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley. URL: https://martinfowler.com/books/eaa.html (Дата звернення: 15.03.2026).
17. Lerman, J., & Miller, R. Programming Entity Framework: Code First. O'Reilly Media. URL: https://www.oreilly.com/library/view/programming-entity-framework/9781449317867/ (Дата звернення: 16.03.2026).
18. Skeet, J. C# in Depth, Fourth Edition. Manning Publications. URL: https://www.manning.com/books/c-sharp-in-depth-fourth-edition (Дата звернення: 17.03.2026).
19. Smith, S. eShopOnWeb Reference Application. URL: https://github.com/dotnet-architecture/eShopOnWeb (Дата звернення: 15.03.2026).
20. Mooney, J. FluentValidation Library. URL: https://fluentvalidation.net/ (Дата звернення: 16.03.2026).
21. W3C. Web Architecture and API Standards. URL: https://www.w3.org/standards/webdesign/ (Дата звернення: 05.03.2026).
22. OWASP. API Security Top 10. URL: https://owasp.org/www-project-api-security/ (Дата звернення: 15.03.2026).
23. RESTful API Tutorial. Understanding RESTful Services. URL: https://restfulapi.net/ (Дата звернення: 13.03.2026).

24. Mozilla Developer Network. HTTP documentation. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP (Дата звернення: 14.03.2026).
25. JSON.org. Introducing JSON. URL: https://www.json.org/json-en.html (Дата звернення: 27.03.2026).
26. OAuth.net. OAuth 2.0 Protocol. URL: https://oauth.net/2/ (Дата звернення: 28.03.2026).
27. Auth0. Authentication and Authorization Docs. URL: https://auth0.com/docs (Дата звернення: 29.03.2026).
28. Postman Learning Center. Testing REST APIs. URL: https://learning.postman.com/docs/sending-requests/requests/ (Дата звернення: 21.03.2026).
29. GitHub Actions. Automating workflows. URL: https://github.com/features/actions (Дата звернення: 15.03.2026).
30. Stripe API Docs. Real-World REST API Example. URL: https://stripe.com/docs/api (Дата звернення: 11.04.2026).
31. Stripe. Stripe .NET SDK Repository. URL: https://github.com/stripe/stripe-dotnet (Дата звернення: 12.04.2026).
32. Refactoring.Guru. Design Patterns (Mediator & Repository in C#). URL: https://refactoring.guru/ (Дата звернення: 14.04.2026).
33. Chapsas, N. Modern REST API Techniques with C#. URL: https://www.youtube.com/@NickChapsas (Дата звернення: 26.04.2026).
34. Quiroz, I. How To Easily Integrate STRIPE Into Your .NET API!. URL: https://www.youtube.com/watch?v=2sZXrUFvZ3E (Дата звернення: 11.04.2026).
35. The .NET Foundation. Homepage. URL: https://dotnetfoundation.org/ (Дата звернення: 15.04.2026).
36. Jovanović, M. Advanced Clean Architecture in .NET 8. URL: https://www.youtube.com/@milanovanovic (Дата звернення: 25.04.2026).

37. Stack Overflow. Questions tagged 'asp.net-core'. URL: https://stackoverflow.com/questions/tagged/asp.net-core (Дата звернення: 02.04.2026).
38. Code-Maze. Ultimate ASP.NET Core Web API. URL: https://code-maze.com/ultimate-aspnet-core-web-api/ (Дата звернення: 03.04.2026).
39. Galloway, S., & Pope, K. ASP.NET Community Standup. URL: https://www.youtube.com/playlist?list=PLdo4fOcmZ0oX-DBuRG4u58ZTAJgBAeQ-t (Дата звернення: 05.04.2026).
40. Mantinband, A. Clean Architecture and Domain-Driven Design in .NET. URL: https://www.youtube.com/@AmichaiMantinband (Дата звернення: 28.04.2026).