ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-платформи для дистрибуції навчальних курсів та навчальних матеріалів»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Вадим БАГАЧОК
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Вадим БАГАЧОК

Керівник: Олександр ЯРОШЕВСЬКИЙ
асист.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____Ірина ЗАМРІЙ
« ____» _____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Багачку Вадиму Дмитровичу

1. Тема кваліфікаційної роботи: «Розробка web-платформи для дистрибуції навчальних курсів та навчальних матеріалів»
керівник кваліфікаційної роботи асистент кафедри ІПЗ Олександр ЯРОШЕВСЬКИЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про принципи побудови web-платформ, опис сучасних підходів до розробки web-застосунків із використанням технологій ASP.NET Core MVC, а також засобів роботи з базами даних Entity Framework. Технічна документація з використання контейнеризації, системи автентифікації та авторизації , а також засобів валідації даних.

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

1. Провести аналіз предметної області та існуючих рішень у сфері електронної комерції й платформ для навчального контенту, а також

визначити функціональні та нефункціональні вимоги до системи LearnStore.
2. Розробити архітектуру системи на основі принципів Clean Architecture з урахуванням вимог масштабованості та підтримуваності.
3. Спроєктувати структуру бази даних та створити UML-діаграми класів, компонентів і взаємодії модулів системи.
4. Реалізувати серверну частину вебзастосунку на платформі ASP.NET Core із використанням мови C#.
5. Інтегрувати ORM Entity Framework Core, реалізувати механізм міграцій бази даних та забезпечити коректну роботу з даними.
6. Реалізувати систему автентифікації та авторизації користувачів на основі JWT і ASP.NET Identity.
7. Впровадити архітектурний підхід CQRS із використанням бібліотеки MediatR для організації обробки команд і запитів.
8. Розробити основні функціональні модулі системи (управління товарами, користувачами, замовленнями, авторами, категоріями та ролями).
9. Забезпечити якість програмного коду шляхом дотримання принципів SOLID, реалізації обробки винятків і валідації даних.
10. Провести тестування системи.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма класів.
5. ER-діаграма бази даних проекту.
6. ER-діаграма таблиць автентифікації та авторизації.
7. Діаграма компонентів.
8. Діаграма варіантів використання.
9. Екранні форми.
10. Апробація результатів дослідження.

6. Дата видачі завдання«23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.	
2	Аналіз та дослідження існуючих аналогів	01.03-11.03.	
3	Огляд популярних платформ онлайн-навчання та їх функціоналу	12.03.2026 – 17.03.	

4	Дослідження способів дистрибуції навчального контенту	18.03– 22.03	
5	Дослідження технологій розробки web-застосунків	23.03– 25.03	
6	Аналіз архітектурних підходів	26.03	
7	Аналіз та формування функціональних і нефункціональних вимог	27.03– 30.03	
8	Розподіл проєкту на архітектурні шари	31.03	
9	Розробка шару Domain	01.04	
10	Реалізація шару Application	02.04– 04.04	
11	Розробка шару Infrastructure	04.04– 05.04	
12	Створення проєкту модульного тестування (Tests.Unit)	05.04– 06.04	
13	Розробка та покриття застосунку юніт-тестами	06.04– 08.04	
14	Проєктування структури та створення бази даних	9.04– 15.04	
15	Налаштування та застосування міграцій бази даних	16.04– 17.04	
16	Проєктування та налаштування UI-рівня	18.04– 25.04	
17	Контейнеризація та налаштування розгортання (Docker)	26.04	
18	Узагальнення результатів, побудова UML-діаграм	04.05	
19	Оформлення звітної документації	09.05	
20	Попередній захист роботи	11.05-22.05	

Здобувач вищої освіти _____ Вадим БАГАЧОК
(підпис)

Керівник
кваліфікаційної роботи _____ Олександр ЯРОШЕВСЬКИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 65стор. ,1табл., 8 рис., 19 джерел.

Мета роботи – Метою роботи є підвищення ефективності процесу дистрибуції та придбання навчальних курсів у web-середовищі за рахунок розробки зручної, масштабованої та функціонально орієнтованої web-платформи, що забезпечує спрощення доступу до навчальних матеріалів, зменшення часу пошуку та придбання курсів.

Об'єкт дослідження – Об'єктом дослідження є процес дистрибуції та придбання навчальних курсів у середовищі дистанційного навчання.

Предмет дослідження – Предметом дослідження є методи, моделі та програмні засоби розробки web-платформи для дистрибуції та придбання навчальних курсів

Короткий зміст роботи: У кваліфікаційній роботі проведено аналіз сучасних підходів до розробки web-платформ для дистрибуції навчальних курсів та навчальних матеріалів, а також досліджено особливості функціонування систем дистанційного навчання. Виконано порівняльний аналіз існуючих платформ Udemy, Coursera та Prometheus, визначено їх переваги, недоліки та обмеження у контексті організації самоосвіти та використання структурованого текстового контенту. На основі проведеного аналізу сформовано функціональні та нефункціональні вимоги до розроблюваної системи.

У межах роботи спроєктовано архітектуру web-платформи LearnStore із використанням принципів шарової архітектури та Clean Architecture. Розроблено доменну модель системи, ER-діаграми бази даних і підсистеми автентифікації та авторизації, а також діаграми структури проєкту. Для реалізації серверної частини застосунку використано технології C#, ASP.NET Core MVC, Entity Framework Core та MS SQL Server. Реалізовано механізми управління

навчальними курсами, категоріями, користувачами, замовленнями та платежами[1].

У роботі застосовано ASP.NET Core Identity та JWT для реалізації автентифікації й авторизації користувачів. Для забезпечення структурованості та підтримуваності програмного забезпечення використано MediatR, FluentValidation, Dependency Injection і репозиторний підхід. Для контейнеризації та розгортання застосунку використано Docker і Docker Compose.

Окрему увагу приділено забезпеченню масштабованості, слабкої зв’язаності компонентів системи та розділенню відповідальностей між рівнями архітектури. Проведено модульне тестування основних компонентів системи із використанням xUnit, NSubstitute та FluentAssertions. Результатом роботи є реалізована web-платформа для дистрибуції навчального контенту, орієнтована на підтримку самоосвіти, швидкий доступ до структурованих навчальних матеріалів та зручну взаємодію користувачів із системою.

КЛЮЧОВІ СЛОВА: WEB-ПЛАТФОРМА, ДИСТРИБУЦІЯ НАВЧАЛЬНОГО КОНТЕНТУ, ДИСТАНЦІЙНЕ НАВЧАННЯ, ASP.NET CORE MVC, C#, ENTITY FRAMEWORK CORE, MS SQL SERVER, CLEAN ARCHITECTURE, JWT, ASP.NET CORE IDENTITY, DOCKER, FLUENTVALIDATION, MEDIATR, MVC, WEB-ЗАСТОСУНОК, САМООСВІТА, НАВЧАЛЬНІ КУРСИ, REST API.

ЗМІСТ

ВСТУП

Актуальність: У сучасних умовах цифровізації освіти значного поширення набувають системи дистанційного навчання, які забезпечують користувачам можливість отримання знань у зручному форматі незалежно від місця та часу. Особливої актуальності набувають web-платформи, орієнтовані на самоосвіту, що дозволяють користувачам самостійно визначати темп навчання та обирати необхідні освітні матеріали.

Існуючі платформи, такі як Udemy, Coursera та Prometheus, надають широкий спектр курсів, проте здебільшого орієнтовані на відеоформат навчання та комплексні програми. Водночас зростає потреба у більш гнучких рішеннях, що забезпечують швидкий доступ до структурованого текстового контенту, зручну навігацію та ефективну взаємодію користувача із системою.

У зв’язку з цим розробка спеціалізованої web-платформи для дистрибуції навчальних курсів і матеріалів, орієнтованої на самоосвіту, є актуальним завданням, що відповідає сучасним тенденціям розвитку освітніх технологій.

Така платформа має стати альтернативним рішенням існуючим сервісам, роблячи акцент на доступності, структурованості та зручності використання навчального контенту.

Об’єкт дослідження є процес дистрибуції та споживання навчальних курсів у середовищі дистанційного навчання.

Предмет дослідження є методи, моделі та програмні засоби розробки web-платформи для дистрибуції навчальних курсів і навчальних матеріалів.

Метою роботи є підвищення ефективності процесу дистрибуції та придбання навчальних курсів у web-середовищі за рахунок розробки зручної, масштабованої та функціонально орієнтованої web-платформи, що забезпечує спрощення доступу до навчальних матеріалів, зменшення часу пошуку та придбання курсів.

Методи дослідження: Першим етапом дослідження стало ознайомлення з сучасними підходами до розробки web-застосунків та аналіз існуючих ПЗ у сфері платформ для дистрибуції навчального контенту. Це дозволило визначити основні функціональні та нефункціональні вимоги до майбутньої системи, а також сформувати загальні принципи її реалізації.

Додатково під час проєктування системи було застосовано принципи чистої архітектури (Clean Architecture), що забезпечує чітке розділення бізнес-логіки, рівня представлення та рівня доступу до даних. Використання даного підходу підвищує масштабованість, тестованість та підтримуваність програмного забезпечення. Визначена архітектура стала основою для подальшого підбору технологічного стеку, який повинен був відповідати закладеним принципам та вимогам до системи.

Відповідно до сформульованих вимог було здійснено вибір технологій для реалізації програмного забезпечення. Для серверної частини використано мову програмування C#, яка входить до платформи .NET та широко застосовується для створення сучасних web-застосунків. Вона забезпечує надійну реалізацію бізнес-логіки, строгий контроль типів та високу стабільність ПЗ.

Для побудови web-застосунку використано фреймворк ASP.NET Core MVC, що реалізує архітектурний шаблон MVC . Такий підхід дозволяє розділити систему на логічні компоненти, що спрощує розробку, тестування та супровід ПЗ. Додатково застосовано сучасні механізми, зокрема Dependency Injection, що підвищує гнучкість архітектури[3].

Для роботи з базою даних використано технологію Entity Framework, яка забезпечує об’єктно-реляційне відображення та дозволяє взаємодіяти з даними без використання великої кількості SQL-запитів. Це спрощує розробку та підвищує читабельність коду. Для управління структурою бази даних у процесі розробки застосовано механізм міграцій.

У якості системи управління базами даних обрано MS SQL Server, що забезпечує надійне зберігання інформації, підтримує складні SQL-запити та має

повну інтеграцію з технологіями платформи .NET. Для адміністрування та управління базою даних використовується SSMS.

Для реалізації механізмів автентифікації та авторизації користувачів застосовано ASP.NET Core Identity, що забезпечує безпечне управління обліковими записами, ролями користувачів та захист даних. У межах API-рівня для організації токен-орієнтованої авторизації використовується JWT , що дозволяє забезпечити захищений обмін даними між клієнтом і сервером. У даному проєкті JWT застосовується переважно для перевірки коректності реалізації сценаріїв використання та тестування роботи захищених API-ендпоінтів.

Для валідації даних використано бібліотеку FluentValidation, яка дозволяє відокремити правила перевірки від бізнес-логіки та підвищує структурованість ПЗ.

На завершальному етапі для розгортання застосунку використано технологію Docker, яка забезпечує контейнеризацію системи, спрощує процес деплою та гарантує стабільну роботу ПЗ в різних середовищах.

Наукова новизна роботи полягає у розробці web-платформи, орієнтованої на самоосвіту, яка забезпечує ефективну дистрибуцію навчального контенту з акцентом на структурований текстовий матеріал.

Практична значущість результатів полягає у можливості використання розробленої платформи для організації дистанційного навчання, що дозволяє користувачам отримувати доступ до навчальних матеріалів у зручний час, а авторам — ефективно створювати та поширювати освітній контент.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Особливості платформ для дистрибуції навчального контенту

Сучасний розвиток інформаційних технологій та цифровізація освітнього середовища сприяють активному поширенню платформ дистанційного навчання. Системи забезпечують користувачам можливість отримання знань у зручному форматі незалежно від місця перебування, а також часу доступу до навчальних матеріалів. Особливої актуальності набувають web-платформи, орієнтовані на самоосвіту, які дозволяють користувачам самостійно визначати темп та послідовність навчання.

Системи дистрибуції навчального контенту призначені для організації доступу до курсів, навчальних матеріалів та інших освітніх ресурсів. Основною задачею таких платформ є забезпечення взаємодії між продавцями освітнього контенту та користувачами системи.

У межах подібних платформ автори та продавці курсів здійснюють створення, публікацію й оновлення навчальних матеріалів, тоді як користувачі отримують доступ до перегляду, придбання та проходження курсів відповідно до власних потреб і цілей навчання.

На сьогодні існує велика кількість платформ дистанційного навчання, серед яких найбільш відомими є Udemy, Coursera та Prometheus. Дані платформи надають доступ до значної кількості навчальних курсів та освітніх програм у різних галузях знань. Більшість сучасних рішень орієнтована переважно на відеоформат навчання, комплексні освітні програми та проходження повноцінних онлайн-курсів.

Разом із тим, у сучасних умовах зростає потреба у більш гнучких системах, орієнтованих на швидкий доступ до структурованого текстового контенту. Окремо, цей підхід дозволяє спростити пошук необхідної інформації, підвищити

швидкість засвоєння матеріалу та забезпечити зручну навігацію між навчальними ресурсами.

Особливістю предметної області є також необхідність забезпечення стабільної роботи системи, зручного інтерфейсу користувача, структурованого зберігання навчальних матеріалів та підтримки взаємодії між різними компонентами платформи.

1.2 Платформа Udemy

Udemy є однією з найбільших платформ для дистрибуції онлайн-курсів, яка функціонує за принципом маркетплейсу навчального контенту. Платформа надає авторам можливість самостійно створювати, публікувати та оновлювати навчальні курси, а користувачам — отримувати доступ до освітніх матеріалів у дистанційному форматі.

Основним форматом подачі контенту на платформі є відеолекції, які можуть доповнюватися текстовими матеріалами, тестами, практичними завданнями та додатковими ресурсами. Після придбання курсу користувач отримує необмежений доступ до навчальних матеріалів, що дозволяє проходити навчання у зручному темпі.

Платформа забезпечує засоби пошуку та фільтрації курсів за категоріями, рівнем складності, тематикою та рейтингом. Додатково користувачі мають можливість переглядати відгуки, оцінки інших студентів та історію власних придбань. Для авторів курсів Udemy надає інструменти управління контентом, редагування матеріалів та взаємодії зі студентами.

До основних переваг платформи можна віднести:

1. простоту використання;
2. зручний механізм публікації курсів;
3. велику кількість доступного навчального контенту;
4. наявність вбудованої системи продажу курсів.

До основних недоліків платформи можна віднести:

1. орієнтацію переважно на відеоформат навчання;

2. обмежені можливості використання структурованого текстового контенту;

3. залежність авторів курсів від внутрішньої політики платформи;

4. обмежений контроль над даними користувачів та механізмами взаємодії із системою.

Udemy є однією з найбільш популярних платформ для дистрибуції навчального контенту та забезпечує зручні засоби для публікації й продажу онлайн-курсів. Водночас особливості організації навчального процесу та орієнтація переважно на відеоформат не повністю відповідають концепції швидкого доступу до структурованих текстових матеріалів, що враховується під час проєктування власної web-платформи.

1.3 Платформа Coursera

Coursera є освітньою платформою, орієнтованою на співпрацю з університетами, навчальними закладами та міжнародними організаціями. Платформа надає доступ до великої кількості онлайн-курсів, професійних програм та спеціалізацій у різних галузях знань.

Навчальні курси на Coursera мають чітко визначену структуру та передбачають послідовне проходження окремих модулів. Основу навчального контенту складають відеолекції, які доповнюються тестами, практичними завданнями, системою оцінювання та додатковими матеріалами. Після успішного завершення курсу користувач може отримати сертифікат, що підтверджує проходження навчання.

Платформа забезпечує високий рівень організації навчального процесу, підтримує контроль прогресу користувачів та надає можливість проходження спеціалізованих освітніх програм. Значна частина курсів створюється у співпраці

з провідними університетами та компаніями, що позитивно впливає на якість навчального контенту та рівень довіри з боку користувачів.

До основних переваг платформи можна віднести:

1. високу якість навчальних матеріалів;
2. структурованість курсів;
3. наявність системи сертифікації;
4. співпрацю з університетами та міжнародними організаціями.

До основних недоліків платформи можна віднести:

1. обмежену гнучкість навчального процесу;
2. орієнтацію переважно на комплексні освітні програми;
3. залежність частини функціональності від платної підписки;
4. складність створення та публікації курсів для незалежних авторів.

Coursera є однією з найбільш відомих платформ дистанційного навчання та забезпечує високий рівень якості освітнього контенту. Водночас орієнтація на формалізовані навчальні програми та переважне використання відеоформату не повністю відповідають концепції швидкого доступу до структурованих текстових матеріалів, що враховується під час розробки власної web-платформи.

1.4 Платформа Prometheus

Prometheus є українською платформою дистанційного навчання, яка надає користувачам доступ до онлайн-курсів, створених у співпраці з університетами, освітніми установами та громадськими організаціями. Платформа орієнтована на поширення доступної освіти та підтримку самоосвіти користувачів у різних галузях знань.

Навчальний контент на платформі представлений переважно у форматі відеолекцій, які доповнюються тестами, практичними завданнями та додатковими матеріалами.

Курси мають чітко визначену структуру та передбачають послідовне проходження окремих навчальних модулів. Користувачі можуть проходити

навчання у зручний для себе час, переглядати власний прогрес та отримувати сертифікати після завершення курсів.

Однією з особливостей платформи є орієнтація на україномовний освітній контент та співпраця з українськими навчальними закладами. Це сприяє популяризації дистанційної освіти та забезпечує доступ до навчальних матеріалів для широкого кола користувачів.

До основних переваг платформи можна віднести:

1. безкоштовний доступ до більшості курсів;
2. наявність україномовного контенту;
3. співпрацю з освітніми установами;
4. структуровану організацію навчального процесу.

До основних недоліків платформи можна віднести:

1. орієнтацію переважно на відеоформат навчання;
2. обмежені можливості швидкого доступу до окремих фрагментів інформації;
3. невисокий рівень інтерактивної взаємодії з контентом;
4. обмежену гнучкість навігації між навчальними матеріалами.

Prometheus є популярною українською платформою дистанційного навчання, яка забезпечує широкий доступ до освітнього контенту та сприяє розвитку самоосвіти. Водночас особливості організації навчального процесу та орієнтація переважно на відеокурси створюють потребу у більш гнучких рішеннях, орієнтованих на швидкий доступ до структурованих текстових матеріалів.

1.5 Порівняльний аналіз

Для визначення особливостей сучасних систем дистрибуції навчального контенту було проведено порівняльний аналіз платформ Udemy, Coursera та Prometheus. Аналіз здійснювався за критеріями, які є важливими у контексті

розробки web-платформи для дистрибуції навчальних курсів та навчальних матеріалів.

У таблиці оцінювалися такі характеристики:

1. тип системи — загальна модель функціонування платформи;

2. основний формат — спосіб подачі навчального контенту;

3. текстовий контент — рівень підтримки структурованих текстових матеріалів;

4. гнучкість навчання — можливість самостійного вибору темпу та послідовності проходження курсів;

5. структурованість — рівень організації та впорядкованості навчальних матеріалів;

6. швидкий доступ до інформації — можливість швидкого отримання необхідних знань без проходження повного курсу;

7. монетизація — наявність механізмів продажу та придбання курсів;

8. контроль над даними — рівень контролю авторів над користувацькими даними та контентом;

9. самоосвіта — підтримка моделі самостійного навчання;

10. орієнтація на користувача — зручність взаємодії користувача із системою.

Проведений аналіз показав, що більшість сучасних платформ для дистрибуції навчального контенту мають спільну особливість — орієнтацію переважно на відеоформат подачі матеріалу. Це забезпечує послідовне проходження курсів, однак ускладнює швидкий доступ до окремих фрагментів інформації та знижує гнучкість процесу самоосвіти.

У таблиці використано такі позначення для оцінювання рівня реалізації характеристик: знак «+» означає, що характеристика реалізована на високому рівні; знак «±» вказує на часткову реалізацію характеристики; знак «-» свідчить про слабку реалізацію характеристики або її відсутність.

Таблиця 1.1

Порівняльний аналіз

№	Характеристика	Udemy	Coursera	Prometheus
1	Тип системи	Маркетплейс	Освітня платформа	Освітня платформа
2	Основний формат	Відео	Відео	Відео
3	Текстовий контент	±	±	±
4	Гнучкість навчання	+	-	-
5	Структурованість	±	+	+
6	Швидкий доступ до інформації	-	-	-
7	Контроль над даними	-	-	-
8	Самоосвіта	±	-	-
9	Орієнтація на користувача	+	+	+

Платформи Coursera та Prometheus орієнтовані пере8важно на структуровані освітні програми з фіксованою послідовністю проходження навчальних модулів. Це забезпечує високий рівень організації навчального процесу, проте обмежує можливість користувача самостійно визначати порядок вивчення матеріалів. Udemy, у свою чергу, надає більшу свободу у виборі курсів та підтримує механізми продажу навчального контенту, однак також переважно використовує відеоформат як основний спосіб подачі інформації.

Додатково було встановлено, що більшість існуючих платформ мають обмежені можливості використання структурованого текстового контенту як основного джерела навчальної інформації. Крім того, автори курсів часто залежать від внутрішньої політики платформ та мають обмежений контроль над даними користувачів і механізмами взаємодії із системою. У межах розроблюваної web-платформи основний акцент робиться на ефективній дистрибуції структурованих навчальних матеріалів та підтримці моделі самоосвіти користувачів.

2 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

2.1 Середовище розробки

Інтегроване середовище розробки (IDE) — це програмне забезпечення, призначене для забезпечення повного циклу створення програмного продукту. Воно надає засоби для написання, тестування, налагодження та супроводу програмного коду. Сучасні IDE також дозволяють автоматизувати процеси збірки проєкту, керування залежностями, інтеграції із системами контролю версій та аналізу помилок.

У межах розробки web-платформи для дистрибуції навчальних курсів і навчальних матеріалів було використано Visual Studio. Дане середовище забезпечує підтримку платформи .NET, мови програмування C# та технології ASP.NET Core, а також надає зручні інструменти для налагодження, роботи з базою даних і управління структурою проєкту[1].

2.1.1 Visual Studio

Visual Studio — це інтегроване середовище розробки, створене для платформи .NET та мови програмування C#. У межах проєкту воно використовувалося для реалізації серверної логіки web-платформи, розробки API, взаємодії з базою даних та побудови бізнес-логіки застосунку.

Перевагою Visual Studio є ефективна підтримка веб-фреймворку ASP.NET Core MVC, що забезпечує організацію структури проєкту та розподіл застосунку на моделі, представлення та контролери. Це сприяє чіткому розділенню відповідальності між компонентами системи та спрощує їх взаємодію.

Середовище забезпечує аналіз структури проєкту, зручну навігацію по коду, повнотекстовий пошук та вбудовані інструменти налагодження. Також підтримується робота з NuGet пакетами та інтеграція із системами контролю версій.

Для роботи з базою даних використовувалась MS SQL Server. Оскільки Visual Studio, MS SQL Server та Entity Framework є частиною екосистеми Microsoft, це забезпечує їх повну інтеграцію та спрощує процес розробки. Середовище дозволяє працювати з Entity Framework, створювати міграції, виконувати SQL-запити та налагоджувати взаємодію з базою даних.

Додатково Visual Studio підтримує інтеграцію із сучасними AI-інструментами, що допомагають у написанні коду, пошуку помилок та контролі якості програмного забезпечення.

Враховуючи всі вищезазначені фактори, було прийнято рішення використовувати Visual Studio як основне середовище розробки.

2.2 Мови програмування

Мови програмування — це формалізовані засоби опису алгоритмів, що виконуються комп'ютером. Вони використовуються для реалізації бізнес-логіки, роботи з даними, взаємодії з базами даних, створення інтерфейсів користувача та побудови програмних систем різної складності. Сучасні мови підтримують різні парадигми програмування, зокрема об'єктно-орієнтовану та імперативну, що забезпечує гнучкість при розробці програмного забезпечення.

У межах даного проєкту основною мовою програмування було використано C#, яка входить до платформи .NET. Вона забезпечує високий рівень продуктивності, строгий контроль типів та широкі можливості для розробки сучасних web-застосунків.

2.2.1 Мова програмування C#

C# — це сучасна об'єктно-орієнтована мова програмування загального призначення, розроблена для платформи .NET. Вона поєднує продуктивність, строгий контроль типів та зручний синтаксис, що робить її ефективним інструментом для створення вебзастосунків, десктопних рішень та серверних систем. Мова підтримує основні парадигми програмування, зокрема об'єктно-

орієнтовану та імперативну, що забезпечує побудову структурованої та масштабованої архітектури застосунків.

У межах проєкту C# використовувалася для реалізації серверної частини web-платформи. На її основі було побудовано бізнес-логіку системи, API, обробку запитів та взаємодію з базою даних. Завдяки інтеграції з .NET вона забезпечує зручну роботу з технологіями ASP.NET Core та Entity Framework.

Використання цієї мови забезпечило стабільність застосунку, покращення організації коду та підвищення ефективності розробки і тестування основних компонентів системи.

2.3 Веб-фреймворки

Веб-фреймворки — це програмні платформи, які спрощують процес розробки вебзастосунків шляхом надання готових компонентів, бібліотек та архітектурних рішень. Вони дозволяють ефективно реалізовувати взаємодію з базою даних, обробку запитів, маршрутизацію, а також інші типові задачі веброзробки.

У межах даного проєкту для розробки серверної частини було використано фреймворк ASP.NET Core. Він забезпечує створення сучасних вебзастосунків на платформі .NET та надає зручні інструменти для реалізації бізнес-логіки та обробки запитів користувачів[3]. Для організації структури застосунку застосовувався архітектурний шаблон MVC, що дозволяє розділити систему на логічні компоненти та забезпечити їх ефективну взаємодію.

2.3.1 ASP.NET Core MVC

ASP.NET Core MVC — це веб-фреймворк платформи .NET, призначений для створення сучасних вебзастосунків із використанням архітектурного шаблону Model–View–Controller (MVC). Даний підхід забезпечує розділення логіки застосунку, представлення даних та керування взаємодією з

користувачем, що підвищує структурованість коду, його підтримуваність та масштабованість[1].

У межах даного проєкту ASP.NET Core MVC використовується в окремих частинах системи, зокрема в адміністративному модулі (LearnStore.Admin) та веб-інтерфейсі користувача (LearnStore.Web.MVC). Це дозволяє реалізувати різні рівні взаємодії із системою: адміністрування контенту, управління користувачами та відображення навчальних матеріалів.

Архітектура MVC забезпечує поділ застосунку на три основні компоненти: моделі, які відповідають за роботу з даними; представлення, що формують інтерфейс користувача; та контролери, які обробляють запити і керують логікою взаємодії між компонентами системи.

2.4 Система управління базами даних

Система управління базами даних — це програмне забезпечення, призначене для створення, зберігання, обробки та управління структурованими даними. СУБД є важливою складовою сучасних програмних систем, оскільки забезпечує ефективну роботу з великими обсягами інформації, підтримку зв’язків між даними, а також їх цілісність, надійність і безпеку.

У межах даного пректу було використано систему управління базами даних MS SQL Server, яка входить до екосистеми Microsoft та широко застосовується для розробки застосунків.

2.4.1 MS SQL

MS SQL Server — це реляційна система управління базами даних, розроблена компанією Microsoft, яка забезпечує високу продуктивність, надійність та масштабованість при роботі з великими обсягами даних. Дана СУБД широко використовується для створення корпоративних систем і web-застосунків завдяки тісній інтеграції з технологіями платформи .NET та високому рівню безпеки.

Основні особливості MS SQL Server:

1. Підтримка складних SQL-запитів: дозволяє реалізовувати фільтрацію, сортування, агрегацію даних та зв’язки між таблицями.

2. Підтримка транзакцій та відповідність принципам ACID: забезпечує цілісність, узгодженість і надійність збереження даних навіть у разі збоїв системи.

3. Інтеграція з платформою .NET: забезпечує ефективну взаємодію з Entity Framework та спрощує роботу з базою даних на рівні програмного коду.

4. Засоби адміністрування та безпеки: MS SQL Server підтримує механізми резервного копіювання, управління доступом, автентифікацію користувачів та захист даних від несанкціонованого доступу.

5. Масштабованість та стабільність: система забезпечує стабільну роботу застосунку при збільшенні обсягів даних і навантаження на сервер.

У межах даного проєкту MS SQL Server використовувалась для збереження інформації про користувачів, навчальні курси, уроки, категорії, замовлення та інші сутності системи. Надійність, стабільність та повноцінна інтеграція з екосистемою Microsoft зробили дану СУБД оптимальним рішенням для реалізації серверної частини web-платформи.

2.5 Система контролю версій

Система контролю версій — це програмний інструмент, призначений для збереження історії змін програмного коду та організації процесу розробки програмного забезпечення. Вона дозволяє відстежувати внесені зміни, повертатися до попередніх версій проєкту, контролювати структуру коду та спрощує процес спільної роботи над застосунком.

У межах розробки даного проєкту було використано систему контролю версій Git та платформу GitHub для зберігання віддаленого репозиторію. Використання Git дозволило ефективно керувати версіями проєкту, контролювати зміни в коді та забезпечити стабільність процесу розробки.

GitHub, у свою чергу, забезпечив зручне зберігання репозиторію, синхронізацію змін та можливість резервного збереження програмного коду.

2.5.1 Git

Git — це розподілена система контролю версій, призначена для відстеження змін у програмному коді та організації процесу розробки програмного забезпечення. Вона дозволяє створювати локальні репозиторії, працювати з ними незалежно від підключення до мережі та синхронізувати зміни з віддаленими сховищами. Кожна зміна в Git зберігається у вигляді окремого коміту з унікальним ідентифікатором, що забезпечує можливість контролю історії змін і відновлення попередніх версій проєкту.

У межах даного проєкту Git використовувався для:

1. фіксації основних етапів розробки;
2. створення окремих гілок для реалізації нового функціоналу;
3. контролю змін у програмному коді;
4. об’єднання змін між гілками;
5. відстеження історії змін та відновлення попередніх версій проєкту у разі потреби.

Використання Git дозволило підвищити надійність роботи з кодовою базою, спростити процес розробки та забезпечити контроль над усіма змінами в проєкті.

2.5.2 GitHub

GitHub — це хмарна платформа для зберігання та управління Git-репозиторіями, яка забезпечує зручні засоби для спільної роботи над програмними проєктами. Платформа надає вебінтерфейс для перегляду структури проєкту, історії комітів, гілок та внесених змін, а також підтримує механізми контролю версій і взаємодії між розробниками.

У межах даного проєкту GitHub використовувався як основна платформа для зберігання програмного коду та синхронізації локального репозиторію з

віддаленим сховищем. Використання GitHub забезпечило централізоване зберігання проєкту, резервне копіювання кодової бази та контроль історії змін.

Крім того, платформа підтримує створення pull request, перегляд і аналіз змін у коді. Це дозволило забезпечити структуровану організацію процесу розробки та спростити управління проєктом.

2.6 Система розгортання застосунку

Система розгортання застосунку — це набір програмних засобів та технологій, призначених для автоматизації процесу запуску, налаштування та підтримки програмного забезпечення у робочому середовищі. Використання сучасних засобів розгортання дозволяє забезпечити стабільність роботи застосунку, спростити перенесення системи між різними середовищами та зменшити кількість помилок, пов’язаних із налаштуванням залежностей [15, 16, 19].

У межах даного проєкту для розгортання web-платформи використовувалась технологія Docker. Docker забезпечує контейнеризацію застосунку, що дозволяє запускати програмне забезпечення разом із усіма необхідними залежностями та компонентами в ізольованому середовищі.

2.6.1 Docker

Docker — це платформа для контейнеризації програмного забезпечення, яка дозволяє запускати застосунки в ізольованих середовищах — контейнерах. Контейнери містять усі необхідні компоненти для роботи застосунку, зокрема програмний код, бібліотеки, залежності та конфігураційні файли, що забезпечує стабільність роботи системи незалежно від середовища запуску.

Він забезпечує зручне управління контейнерами, ізоляцію компонентів системи та спрощує процес оновлення й масштабування застосунку. Для конфігурації середовища та запуску сервісів використовувалися Dockerfile і

Docker Compose, що дозволило автоматизувати процес розгортання системи та взаємодію між її компонентами.

2.7 Засоби розробки користувацького інтерфейсу

Засоби розробки користувацького інтерфейсу призначені для створення візуальної частини web-застосунку та забезпечення взаємодії користувача із системою. Вони дозволяють реалізовувати структуру сторінок, стилізацію елементів інтерфейсу, адаптивний дизайн та динамічну поведінку компонентів застосунку.

У межах даного проєкту для реалізації користувацького інтерфейсу використовувались технології Razor та CSS. Razor застосовувався для створення динамічних web-сторінок і поєднання HTML-розмітки із серверною логікою ASP.NET Core MVC, що дозволило ефективно формувати вміст сторінок та відображати дані користувачу.

CSS використовувався для стилізації елементів інтерфейсу, налаштування зовнішнього вигляду сторінок та реалізації адаптивного дизайну застосунку.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Архітектурні принципи та організація системи

Під час проєктування web-платформи для дистрибуції навчальних курсів та навчальних матеріалів особлива увага приділялась побудові масштабованої, підтримуваної та структурованої архітектури застосунку. Для реалізації системи було використано принципи шарової архітектури (Layered Architecture) та Clean Architecture, які забезпечують чітке розмежування відповідальності між окремими компонентами програмного забезпечення.

Основна ідея шарової архітектури полягає у поділі системи на окремі логічні рівні, кожен з яких виконує власний набір задач. Даний підхід дозволяє ізолювати бізнес-логіку від деталей реалізації інтерфейсу користувача, роботи з базою даних та зовнішніх сервісів. У межах даного проєкту архітектура системи була розподілена на декілька окремих проєктів, кожен з яких відповідає за власну область функціональності.

Проєкт LearnStore.Domain містить доменні сутності, моделі та основні бізнес-правила системи. Даний рівень не залежить від зовнішніх технологій та використовується як центральна частина архітектури застосунку [4, 5].

Проєкт LearnStore.Application відповідає за реалізацію прикладної логіки системи. У ньому реалізовано механізми обробки команд і запитів, DTO-моделі, валідацію даних та сервіси взаємодії між компонентами системи. Саме на цьому рівні реалізується основна логіка сценаріїв використання застосунку [4, 5, 12].

Проєкт LearnStore.Infrastructure забезпечує роботу із зовнішніми залежностями та інфраструктурними компонентами. У даному проєкті реалізовано роботу з базою даних через Entity Framework Core, репозиторії, механізми автентифікації, конфігурацію JWT та інші сервіси, необхідні для функціонування системи [1, 10].

Рівень представлення реалізовано через декілька окремих web-проєктів. Проєкт LearnStore.Web.MVC відповідає за користувацький web-інтерфейс платформи, побудований на основі ASP.NET Core MVC. Проєкт LearnStore.Web.Api використовується для реалізації API та забезпечення взаємодії між клієнтською та серверною частинами системи. Окремо також використовується проєкт LearnStore.Admin, який відповідає за реалізацію адміністративної частини системи та забезпечує функціональність для управління компонентами платформи.

Під час побудови архітектури використовувалися принципи Clean Architecture, основною метою яких є мінімізація залежностей між компонентами системи та забезпечення незалежності бізнес-логіки від конкретних технологій. Відповідно до даного підходу внутрішні компоненти застосунку не залежать від реалізації інтерфейсу користувача, системи збереження даних або зовнішніх бібліотек. Це дозволяє змінювати окремі технологічні компоненти без необхідності зміни основної логіки системи.

Для організації взаємодії між компонентами застосунку використовувався принцип інверсії залежностей (Dependency Inversion), відповідно до якого прикладний рівень визначає інтерфейси взаємодії, а їх реалізація знаходиться на рівні інфраструктури. Інверсія залежностей зменшує зв’язність між компонентами системи та спрощує тестування програмного забезпечення.

У межах проєкту також було використано низку додаткових архітектурних підходів та бібліотек. Для реалізації механізму обробки команд і запитів застосовувалась бібліотека MediatR, що дозволило розділити логіку обробки функціональних сценаріїв системи[12]. Для перетворення моделей і DTO використовувався Mapster, а для централізованої валідації даних — FluentValidation. Доступ до бази даних реалізовано за допомогою Entity Framework Core та репозиторного підходу.

Окрему увагу було приділено централізованому налаштуванню залежностей через механізм Dependency Injection. Реєстрація сервісів, репозиторіїв, валідаторів та інших компонентів системи виконується під час запуску застосунку, що забезпечує гнучкість конфігурації та спрощує підтримку програмного забезпечення.

Використання принципів шарової архітектури та Clean Architecture дозволило забезпечити чітку структуру проєкту, підвищити підтримуваність системи, спростити тестування окремих компонентів та створити основу для подальшого масштабування web-платформи.

3.2 Діаграма класів доменної моделі системи

Діаграма класів доменної моделі відображає основні сутності системи LearnStore, їх атрибути та взаємозв’язки між ними. Основною метою діаграми є демонстрація структури предметної області, організації бізнес-сутностей та логіки взаємодії між компонентами системи.

Центральним елементом доменної моделі є сутність Order, яка виступає коренем агрегату замовлення. Вона містить основну інформацію про замовлення, дату створення, дату оновлення, поточний стан замовлення, а також колекції позицій замовлення та платежів. Для представлення стану замовлення використовується перерахування OrderState, яке визначає можливі етапи життєвого циклу замовлення.

Сутність OrderItem представляє окрему позицію замовлення та містить інформацію про товар, його кількість і ціну на момент оформлення замовлення. Кожен елемент замовлення пов’язаний із відповідним товаром (Product).

Сутність Product описує навчальний продукт або навчальний матеріал, що розміщується на платформі. Вона містить основні атрибути товару, зокрема назву, вартість та ознаку активності.

Додатково продукт пов’язаний із сутностями Author, Seller та ProductCategory, які описують автора контенту, продавця та категорію

навчального матеріалу відповідно. Також передбачено самозв’язок між продуктами для реалізації дочірніх або пов’язаних матеріалів.

Сутність Customer представляє користувача платформи, який здійснює придбання навчальних матеріалів. Між Customer та Product реалізовано зв’язок типу багато-до-багатьох , що дозволяє зберігати інформацію про придбані користувачем продукти.

Сутність Payment використовується для збереження інформації про оплату замовлення, включаючи суму платежу та дату його виконання. Одне замовлення може містити декілька платежів.

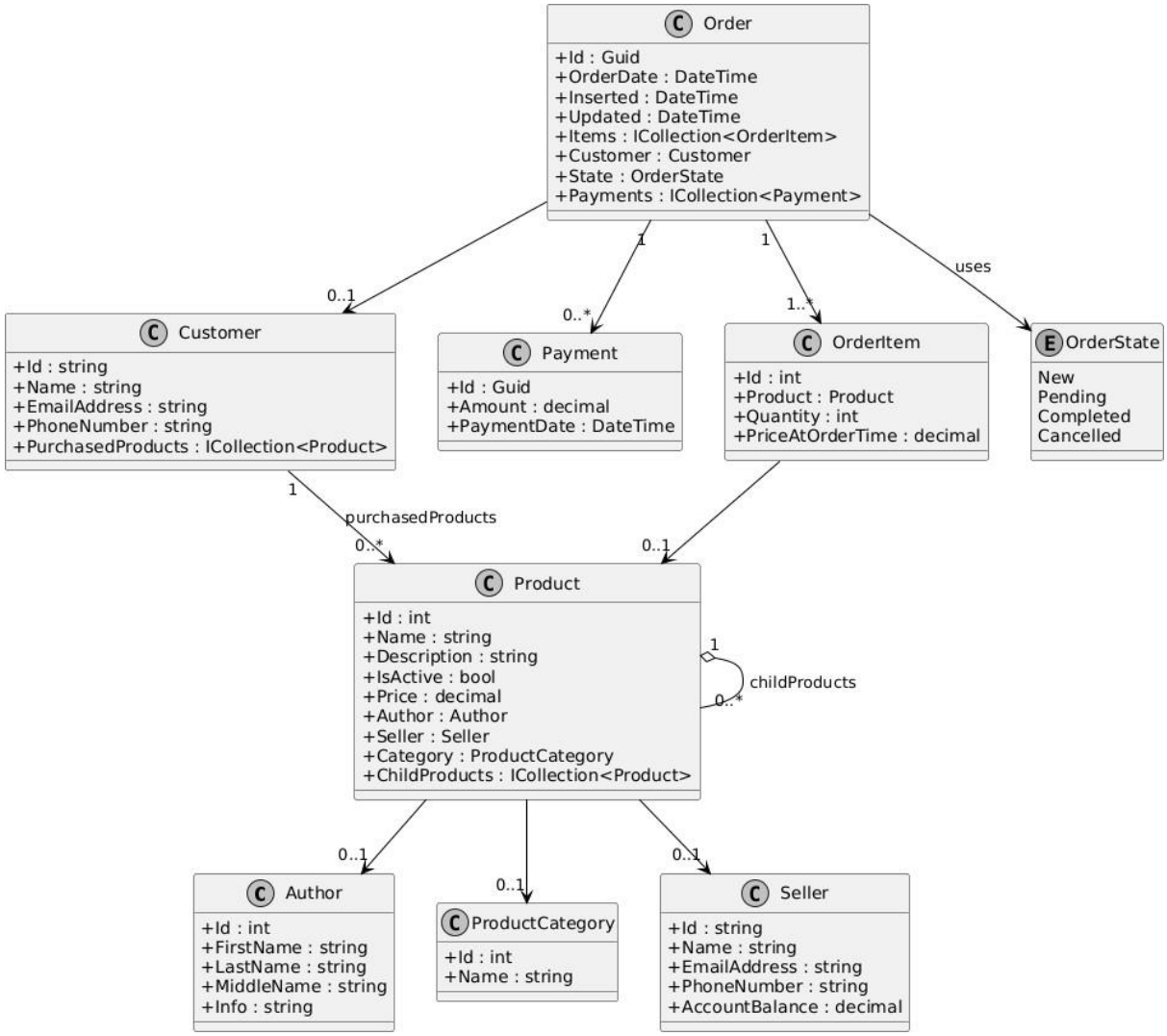


Рис. 3.1 Діаграма класів доменної моделі системи

Між основними сутностями реалізовано такі типи зв’язків:

1. Order — OrderItem : зв’язок один-до-багатьох;

2. Order — Payment : зв’язок один-до-багатьох;

3. Order — Customer : зв’язок один-до-одного;

4. OrderItem — Product : асоціативний зв’язок із товаром;

5. Product — Author, Seller, ProductCategory : зв’язки з довідковими сутностями;

6. Customer — Product : зв’язок багато-до-багатьох.

У межах доменної моделі реалізовано низку бізнес-інваріантів. Зокрема, під час створення замовлення перевіряється наявність покупця та хоча б одного елемента замовлення. Такий підхід дозволяє забезпечити цілісність бізнес-логіки незалежно від рівня інфраструктури або способу взаємодії із системою.

Діаграма класів відображає концептуальну структуру предметної області та використовується як основа для реалізації доменної моделі, конфігурації Entity Framework Core та побудови взаємодії між компонентами системи LearnStore.

3.3 Діаграма компонентів

Діаграма компонентів архітектури системи LearnStore відображає внутрішню структуру застосунку, взаємодію між основними шарами системи та залежності між компонентами відповідно до принципів Clean Architecture. Основною метою діаграми є демонстрація розподілу відповідальностей між шарами Presentation, Application, Domain та Infrastructure, а також організації взаємодії із базою даних SQL Server, сервісом міграцій і модулем тестування.

Шар Presentation Layer представлений компонентами LearnStore.Admin, LearnStore.Web.MVC та LearnStore.Web.Api. Даний шар виступає точкою входу до системи та забезпечує взаємодію користувачів із застосунком через HTTP-запити. Компонент LearnStore.Web.Api реалізує REST API для клієнтських застосунків, LearnStore.Web.MVC відповідає за веб-інтерфейс системи, а

LearnStore.Admin забезпечує функціональність адміністративної панелі. Presentation Layer взаємодіє із Application Layer для виконання бізнес-операцій та отримання результатів обробки запитів.

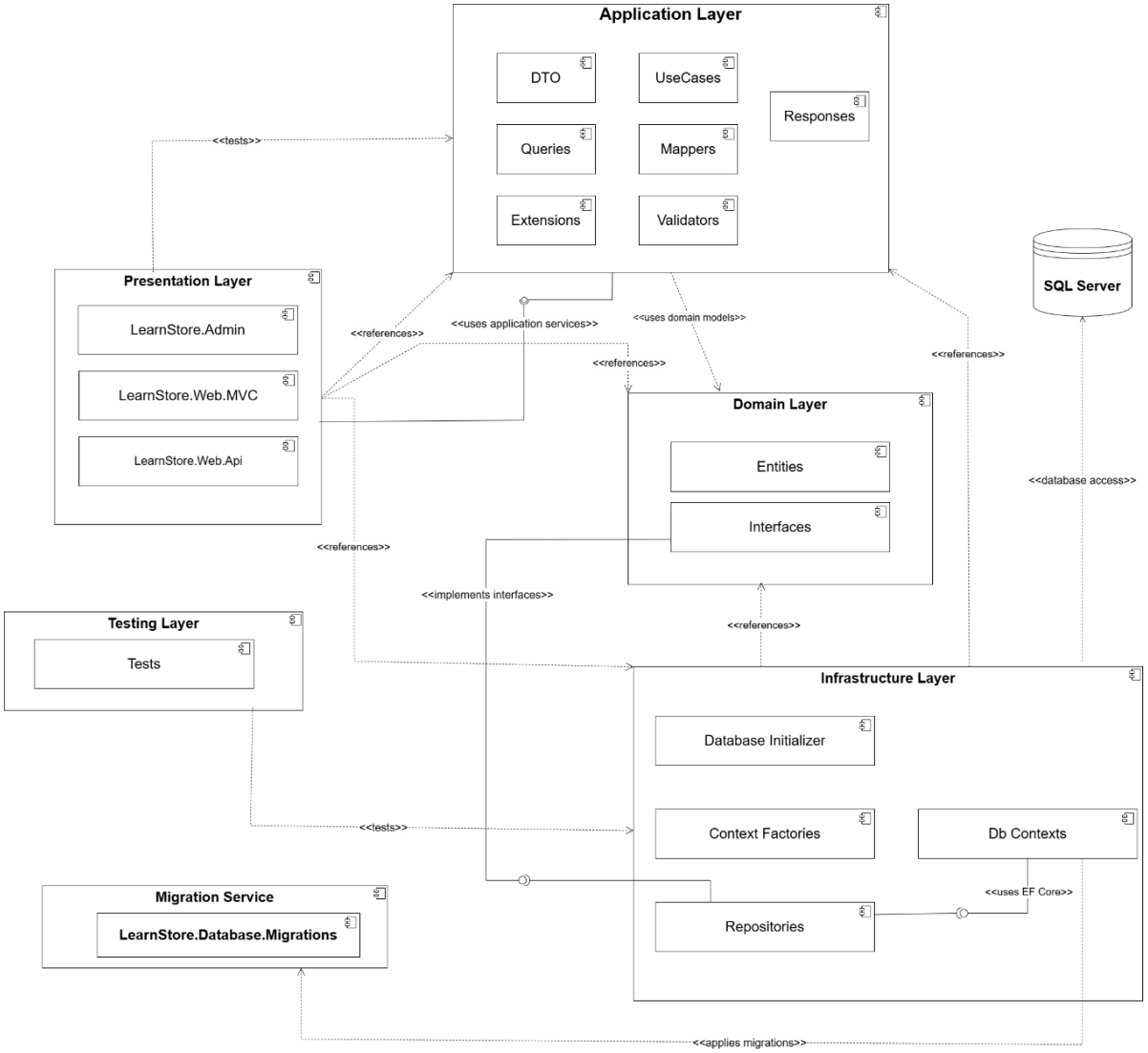


Рис. 3.2 Діаграма компонентів

Шар Application Layer представлений набором компонентів DTO, Queries, Extensions, UseCases, Mappers, Validators та Responses. Даний шар реалізує прикладну логіку системи та координує виконання бізнес-сценаріїв. Компонент DTO використовується для передачі даних між шарами без прямої роботи із доменними сутностями. Queries реалізують механізми отримання даних із системи без зміни її стану. Компонент UseCases містить сценарії виконання бізнес-операцій. Validators забезпечують перевірку коректності вхідних даних

перед виконанням логіки застосунку. Mappers відповідають за перетворення моделей даних між різними шарами системи. Responses використовуються для формування стандартизованих результатів виконання операцій, а Extensions забезпечують конфігурацію сервісів та механізмів залежностей Application шару.

Шар Domain Layer є центральним ядром предметної області системи. У ньому розташовані компоненти Entities та Interfaces. Entities містять основні бізнес-сутності системи та інкапсулюють бізнес-логіку предметної області. Interfaces визначають контракти взаємодії із механізмами доступу до даних та забезпечують принцип інверсії залежностей. Завдяки цьому бізнес-логіка не залежить від конкретної реалізації інфраструктурних сервісів або бази даних.

Шар Infrastructure Layer реалізує технічну інфраструктуру системи. До його складу входять компоненти DatabaseInitializer, Context Factories, Repositories та Db Contexts. Компонент Db Contexts забезпечує взаємодію із базою даних SQL Server через Entity Framework Core. Context Factories відповідають за створення екземплярів контексту бази даних. Repositories реалізують механізми доступу до даних та працюють через Db Contexts. DatabaseInitializer забезпечує початкову ініціалізацію бази даних та налаштування структури сховища даних.

Окремим компонентом системи виступає Migration Service, представлений модулем LearnStore.Database.Migrations. Даний сервіс відповідає за автоматичне застосування міграцій бази даних та оновлення структури SQL Server під час запуску або розгортання системи. Migration Service взаємодіє з Infrastructure Layer та Db Contexts для виконання операцій міграції.

Testing Layer представлений компонентом Tests, який забезпечує модульне тестування системи. Даний шар використовується для перевірки коректності роботи бізнес-логіки, прикладних сценаріїв, репозиторіїв та механізмів взаємодії із базою даних.

База даних SQL Server виступає централізованим сховищем інформації системи LearnStore. Взаємодія з базою даних здійснюється через Infrastructure

Layer та компоненти Db Contexts і Repositories. Такий підхід забезпечує ізоляцію бізнес-логіки від конкретної реалізації механізмів доступу до даних.

Потік виконання системи починається з надходження HTTP-запиту до компонентів Presentation Layer. Далі запит передається до Application Layer, де виконується обробка бізнес-сценарію, валідація даних та формування результату. Для доступу до даних використовуються інтерфейси Domain Layer, реалізація яких знаходиться в Infrastructure Layer. Після завершення виконання операції результат повертається до Presentation Layer та надсилається користувачу.

Таким чином, діаграма компонентів демонструє реалізацію багатошарової архітектури LearnStore із чітким розділенням відповідальностей між модулями системи, використанням принципів Clean Architecture, інверсії залежностей, слабкого зв’язування та ізоляції бізнес-логіки від інфраструктурних компонентів.

3.4 ER-діаграма бази даних системи LearnStore

ER-діаграма бази даних системи LearnStore відображає структуру фізичної моделі даних, основні таблиці, їх атрибути та зв’язки між ними. Основною метою діаграми є демонстрація організації зберігання даних у реляційній базі даних, а також забезпечення цілісності та нормалізації структури даних [13, 14].

Центральною таблицею є Orders, яка зберігає інформацію про замовлення користувачів. Вона містить дату створення замовлення, дату оновлення, поточний стан замовлення та зовнішній ключ CustomerId, що вказує на клієнта, який створив замовлення. Додатково використовується поле State для збереження статусу замовлення у вигляді числового значення (див. Рис. 3.3).

Таблиця OrderItems використовується для зберігання позицій замовлення. Вона містить інформацію про продукт, кількість товару та ціну на момент оформлення замовлення. Такий підхід дозволяє зберігати історичні дані

незалежно від подальших змін у таблиці Products. Кожен запис пов’язаний із конкретним замовленням через зовнішній ключ OrderId.

Таблиця Payment призначена для збереження інформації про платежі, пов’язані із замовленнями. Вона містить суму платежу та дату його здійснення. Один запис у таблиці Orders може бути пов’язаний з декількома записами в Payment, що дозволяє реалізувати часткову оплату замовлення.

Таблиця Products є основною таблицею для зберігання інформації про товари. Вона містить назву, опис, ціну, ознаку активності, а також зовнішні ключі AuthorId, SellerId та CategoryId, які пов’язують продукт із відповідними довідниковими таблицями. Додатково присутній самозв’язок через поле ProductId для реалізації ієрархічних або пов’язаних продуктів.

Таблиця Authors зберігає інформацію про авторів навчального контенту, включаючи ім’я та додаткові дані. Таблиця Sellers містить інформацію про продавців, а таблиця Customers — про користувачів системи, які здійснюють замовлення.

Таблиця ProductCategories використовується для класифікації продуктів за категоріями та містить базову довідкову інформацію.

Зв’язувальна таблиця CustomerProduct реалізує зв’язок багато-до-багатьох між Customers та Products і використовується для збереження інформації про придбані користувачем продукти.

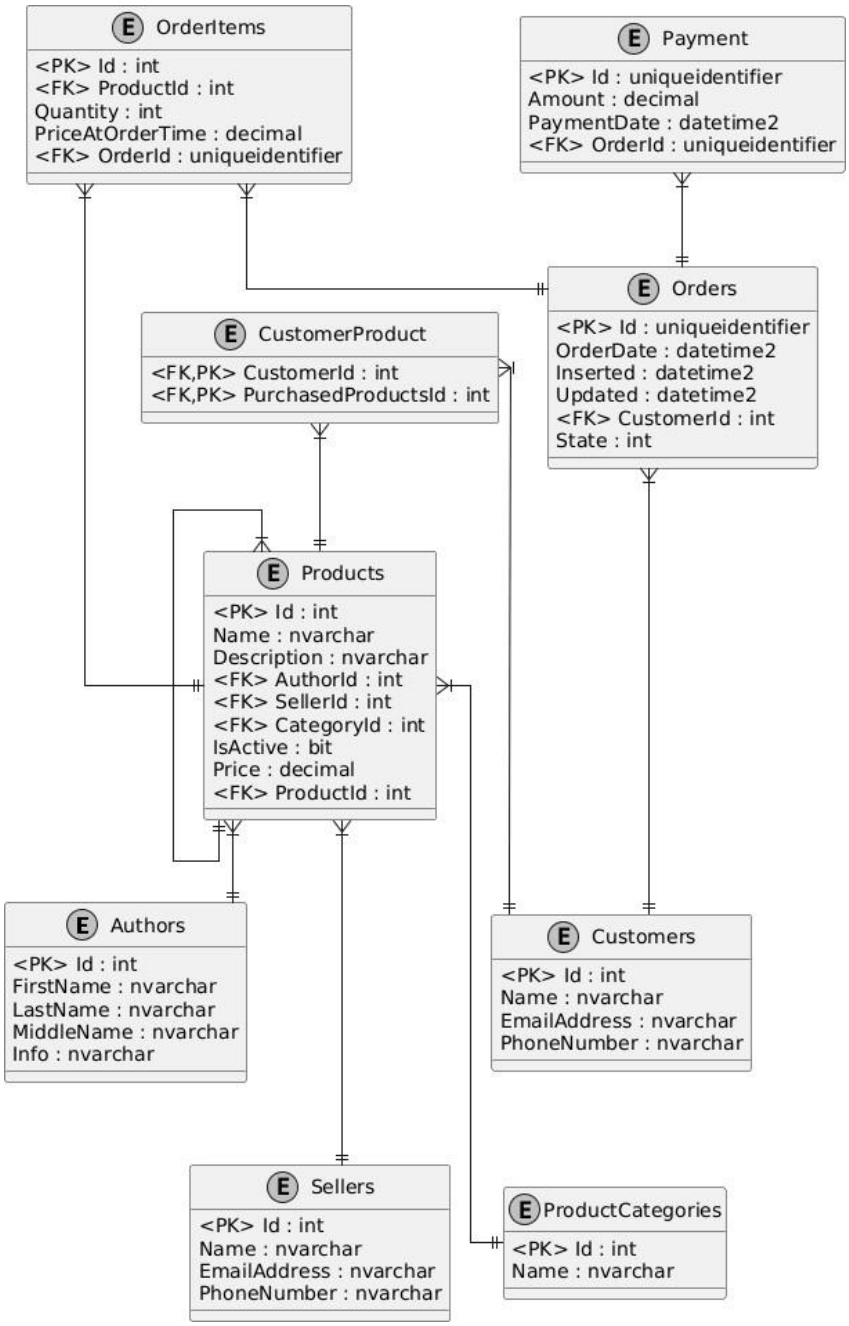


Рис. 3.3 ER-діаграма бази даних LearnStore

Між таблицями бази даних реалізовано такі основні зв’язки:

1. Customers — Orders: зв’язок один-до-багатьох через CustomerId;
2. Orders — OrderItems: зв’язок один-до-багатьох через OrderId;
3. Orders — Payment: зв’язок один-до-багатьох через OrderId;
4. Products — Authors: зв’язок багато-до-одного через AuthorId;
5. Products — Sellers: зв’язок багато-до-одного через SellerId;
6. Products — ProductCategories: зв’язок багато-до-одного через CategoryId;

7. OrderItems — Products: зв’язок багато-до-одного через ProductId;

8. Customers — Products: зв’язок багато-до-багатьох через таблицю CustomerProduct.

У межах ER-моделі бази даних реалізовано ключові обмеження цілісності, зокрема використання зовнішніх ключів для забезпечення зв’язності даних та збереження історичної ціни продукту в OrderItems [13, 14]. Це дозволяє гарантувати коректність історичних замовлень навіть у разі зміни даних у таблиці Products.

ER-діаграма бази даних системи LearnStore відображає фізичну структуру зберігання даних та використовується як основа для реалізації міграцій Entity Framework Core, конфігурації зв’язків між таблицями та забезпечення цілісності даних у реляційній моделі.

3.5 ER-діаграма системних таблиць автентифікації та авторизації

ER-діаграма системних таблиць автентифікації та авторизації системи LearnStore відображає структуру стандартної моделі ASP.NET Identity, яка використовується для управління користувачами, ролями, правами доступу та механізмами автентифікації. Основною метою діаграми є демонстрація організації безпекового шару системи та взаємозв’язків між користувачами, ролями та їхніми атрибутами доступу.

Центральною сутністю є таблиця AspNetUsers (Users), яка зберігає інформацію про користувачів системи. Вона містить основні дані профілю, включаючи ім’я користувача, email, підтвердження пошти, хеш пароля, параметри безпеки, а також дані для підтримки блокування, двофакторної автентифікації та контролю доступу.

Таблиця AspNetRoles (Roles) використовується для зберігання ролей користувачів. Вона містить назву ролі, нормалізоване ім’я та службові поля для контролю конкурентності змін. Ролі визначають рівні доступу в системі та використовуються для реалізації рольової моделі авторизації.

Таблиця AspNetUserRoles (UserRoles) реалізує зв’язок багато-до-багатьох між користувачами та ролями, дозволяючи одному користувачу мати декілька ролей, а одній ролі — бути призначеною багатьом користувачам.

Таблиця AspNetUserClaims (UserClaims) використовується для збереження додаткових прав доступу користувачів у вигляді claims. Вона дозволяє реалізувати більш гнучку модель авторизації на рівні конкретних дозволів.

Таблиця AspNetRoleClaims (RoleClaims) виконує аналогічну функцію, але для ролей, дозволяючи призначати права доступу не окремим користувачам, а групам користувачів через ролі.

Таблиця AspNetUserLogins (UserLogins) використовується для підтримки зовнішніх провайдерів автентифікації (наприклад, Google або Microsoft). Вона зберігає інформацію про зовнішні логіни та їх зв’язок із користувачами системи.

Таблиця AspNetUserTokens (UserTokens) призначена для зберігання токенів користувачів, які використовуються для різних сценаріїв автентифікації, зокрема для відновлення доступу, запам’ятовування сесій або інтеграції з зовнішніми сервісами.

Між основними таблицями реалізовано такі зв’язки:

1. AspNetUsers — AspNetUserRoles: зв’язок один-до-багатьох через UserId;
2. AspNetRoles — AspNetUserRoles: зв’язок один-до-багатьох через RoleId;
3. AspNetUsers — AspNetUserClaims: зв’язок один-до-багатьох через UserId;
4. AspNetRoles — AspNetRoleClaims: зв’язок один-до-багатьох через RoleId;
5. AspNetUsers — AspNetUserLogins: зв’язок один-до-багатьох через UserId;
6. AspNetUsers — AspNetUserTokens: зв’язок один-до-багатьох через UserId.

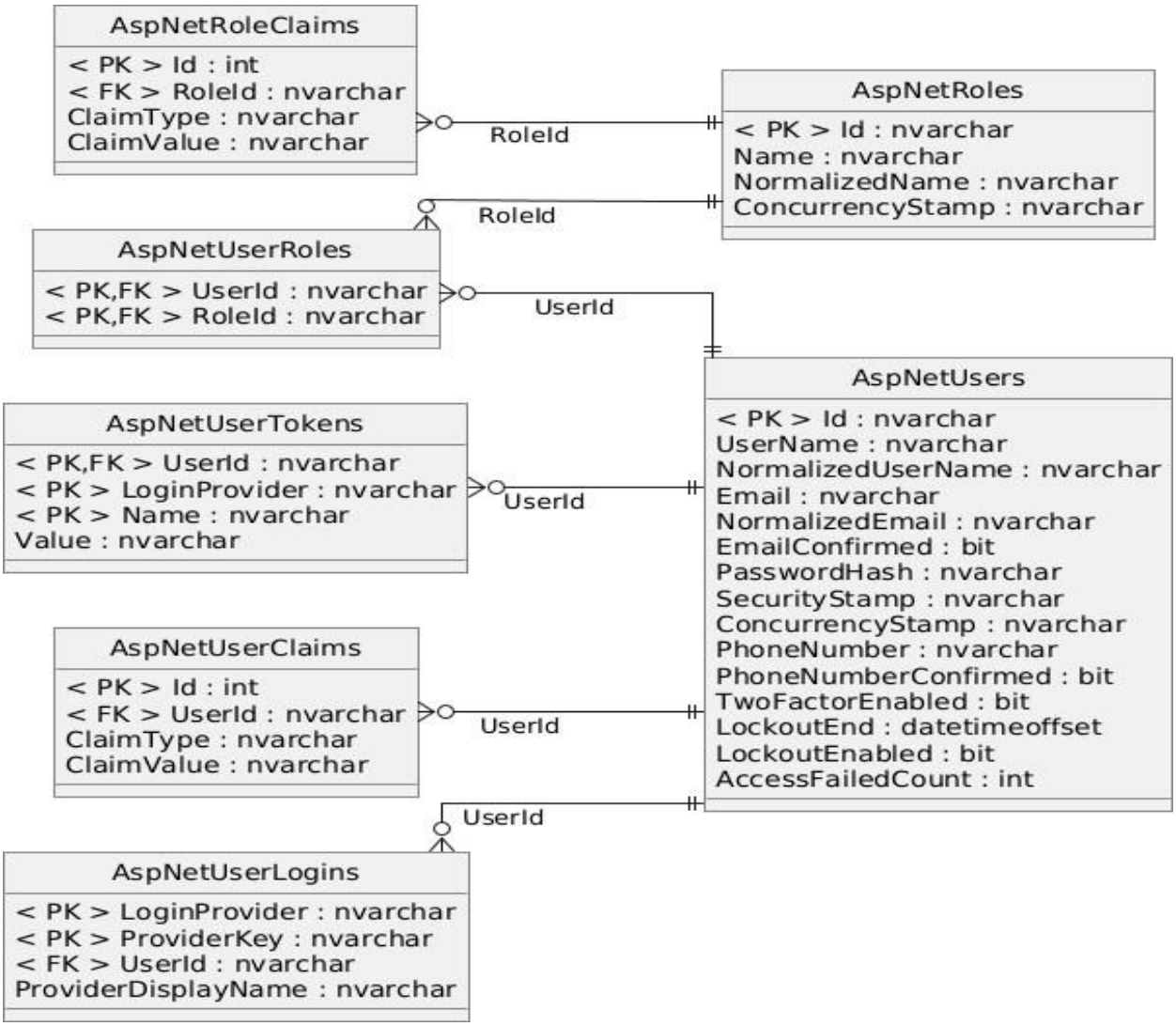


Рис. 3.4 ER-діаграма системних таблиць автентифікації та авторизації

У межах ER-моделі системи автентифікації реалізовано повноцінну рольову модель доступу (RBAC), яка забезпечує гнучке керування правами користувачів та масштабовану систему авторизації.

Використання стандартної структури ASP.NET Identity дозволяє інтегрувати механізми безпеки на рівні фреймворку та забезпечує надійне управління користувачами.

ER-діаграма підсистеми автентифікації та авторизації системи LearnStore відображає фізичну структуру зберігання даних безпеки та використовується як основа для реалізації механізмів Identity, JWT-автентифікації та контролю доступу до ресурсів системи.

3.6 Діаграма варіантів використання для адміністратора

Діаграма варіантів використання системи LearnStore для адміністратора відображає основні сценарії взаємодії користувача типу «Адміністратор» із системою. Основною метою діаграми є демонстрація функціональних можливостей, пов’язаних з автентифікацією, керуванням ролями користувачів та адмініструванням товарів у системі. Основним актором діаграми є «Адміністратор», який взаємодіє із системою LearnStore через набір адміністративних функцій. Діаграма структурована за пакетами відповідно до логічного поділу функціональності системи.

Пакет «Аутентифікація» містить варіанти використання, що забезпечують доступ адміністратора до системи. Адміністратор може зареєструвати обліковий запис, виконати вхід до системи та завершити роботу шляхом виходу з облікового запису. Дані функції забезпечують механізми автентифікації та контролю доступу до адміністративної частини платформи.

Пакет «Авторизація» реалізує функціональність керування ролями користувачів у системі. Адміністратор має можливість призначати ролі користувачам та видаляти наявні ролі. Це дозволяє централізовано керувати правами доступу та забезпечувати розмежування функціональності між різними типами користувачів системи.

Варіант використання «Призначення ролі користувачу» використовується для надання користувачу певного рівня доступу або функціональних можливостей у системі. Варіант «Видалення ролі користувача» забезпечує можливість скасування раніше призначених прав доступу.

Пакет «Управління товарами» містить функціональність, пов’язану з адмініструванням продуктів платформи LearnStore. Центральним сценарієм є «Керування товарами», який об’єднує процеси створення, оновлення та видалення товарів.

Варіант використання «Створення товару» забезпечує можливість додавання нового навчального продукту до системи із зазначенням необхідних характеристик, таких як назва, опис, категорія та вартість.

Варіант використання «Оновлення товару» використовується для редагування інформації про вже існуючі продукти, що дозволяє підтримувати актуальність даних у системі.

Варіант використання «Видалення товару» забезпечує можливість видалення продуктів із системи LearnStore у разі необхідності їх деактивації або повного вилучення.

Між варіантами використання реалізовано зв’язки типу <<include>>, які відображають включення допоміжних сценаріїв до основного бізнес-процесу. Зокрема, варіант використання «Керування товарами» включає:

1. «Створення товару»;
2. «Оновлення товару»;
3. «Видалення товару».

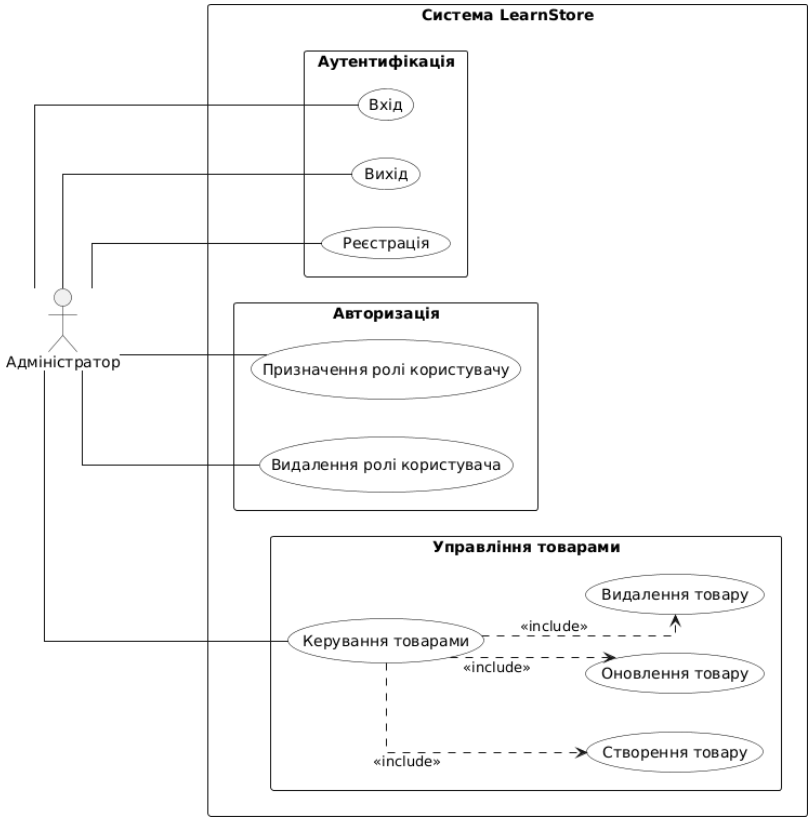


Рис. 3.5 Діаграма варіантів використання для адміністратора

Діаграма варіантів використання системи LearnStore для адміністратора відображає ключові адміністративні можливості системи та демонструє взаємозв’язки між основними процесами керування користувачами і товарами. Вона використовується як основа для подальшого проєктування функціональної архітектури системи, реалізації механізмів авторизації та побудови бізнес-логіки адміністративної частини платформи.

3.7 Діаграма варіантів використання для продавця

Діаграма варіантів використання системи LearnStore для продавця відображає основні сценарії взаємодії користувача типу «Продавець» із системою. Основною метою діаграми є демонстрація функціональних можливостей, пов’язаних з автентифікацією, керуванням товарами та фінансовими операціями продавця.

Основним актором діаграми є «Продавець», який взаємодіє із системою LearnStore через набір функціональних підсистем. Діаграма організована за пакетами відповідно до логічного поділу функціональності системи.

Пакет «Аутентифікація» містить варіанти використання, що забезпечують доступ продавця до системи. Користувач може зареєструвати новий обліковий запис, виконати вхід до системи та завершити роботу шляхом виходу з облікового запису. Дані функції забезпечують механізми ідентифікації та контролю доступу до функціональності платформи.

Пакет «Управління товарами» реалізує основну функціональність продавця, пов’язану з керуванням навчальними продуктами. Продавець має можливість переглядати товари, а також виконувати їх створення та оновлення. Центральним варіантом використання даного пакета є «Керування товарами», який об’єднує процеси створення та редагування продуктів.

Варіант використання «Створення товару» використовується для додавання нового навчального продукту до системи. Під час виконання цього

сценарію продавець може вказати назву продукту, опис, ціну, категорію та інші характеристики товару.

Варіант використання «Оновлення товару» забезпечує можливість редагування інформації про вже існуючі продукти. Це дозволяє змінювати опис, ціну, доступність або інші параметри навчального контенту.

Пакет «Фінанси» містить варіант використання «Виведення заробітку», який відповідає за виконання фінансових операцій із виведення коштів, отриманих від продажу продуктів через платформу LearnStore.

Між варіантами використання реалізовано зв’язки типу <<include>>, які відображають включення допоміжних сценаріїв до основного бізнес-процесу. Зокрема, варіант використання «Керування товарами» включає:

1. «Створення товару»;
2. «Оновлення товару».

Таким чином, сценарій керування товарами складається з декількох взаємопов’язаних функцій, необхідних для повноцінної роботи продавця із навчальним контентом.

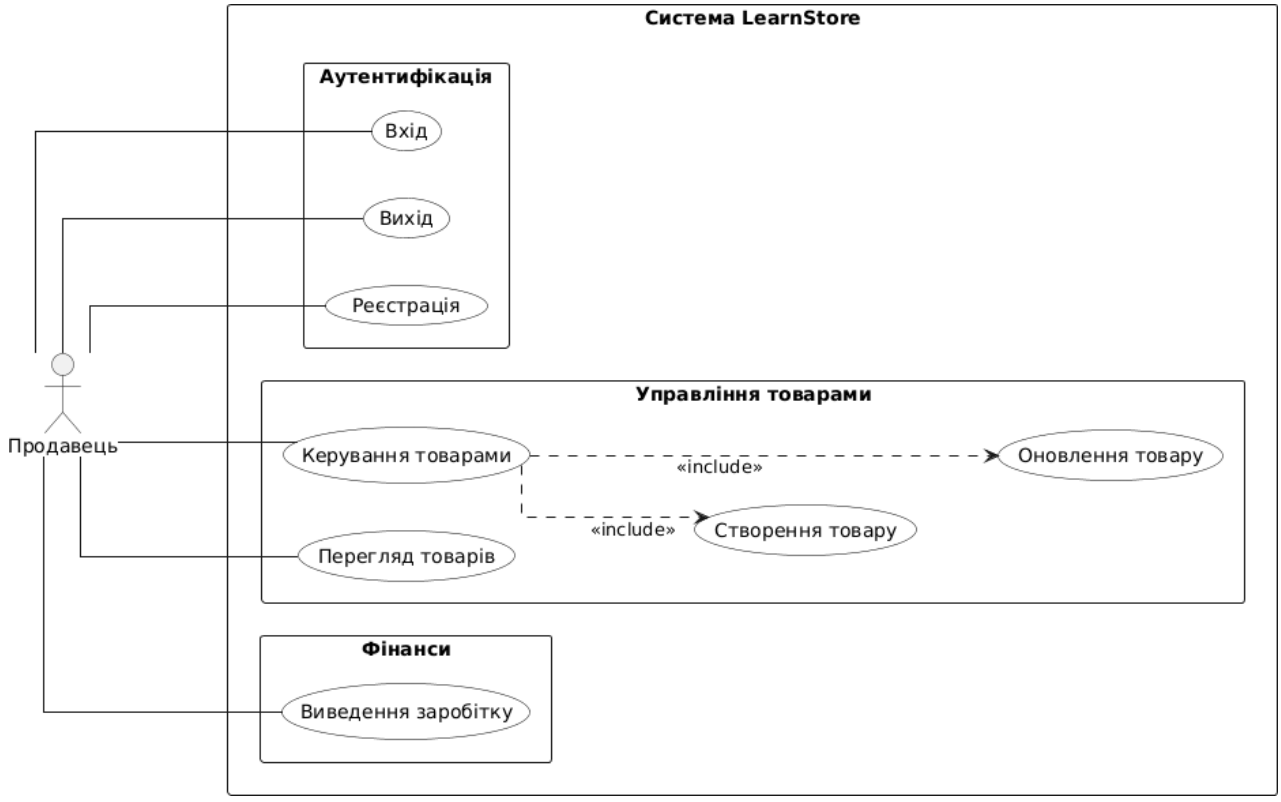


Рис. 3.6 Діаграма варіантів використання для продавця

Діаграма варіантів використання системи LearnStore для продавця відображає ключові можливості користувача типу «Продавець» та демонструє взаємозв’язки між основними бізнес-процесами платформи. Вона використовується як основа для подальшого проєктування функціональної архітектури системи та реалізації бізнес-логіки застосунку.

3.8 Діаграма варіантів використання для незареєстрованого користувача

Діаграма варіантів використання системи LearnStore для незареєстрованого користувача відображає основні сценарії взаємодії користувача із системою без попередньої автентифікації. Основною метою діаграми є демонстрація функціональних можливостей, доступних відвідувачам платформи до створення облікового запису або входу в систему.

Основним актором діаграми є «Незареєстрований користувач», який взаємодіє із системою LearnStore через функціональність публічного доступу. Діаграма структурована у вигляді окремого пакета, що об’єднує сценарії, доступні без авторизації.

Пакет «Публічний доступ» містить варіанти використання, пов’язані з переглядом навчального контенту та виконанням базових операцій автентифікації. Незареєстрований користувач має можливість переглядати доступні товари, ознайомлюватися з деталями конкретного продукту, а також виконувати реєстрацію нового облікового запису або вхід до системи.

Варіант використання «Перегляд товарів» забезпечує доступ до каталогу навчальних курсів та інших цифрових продуктів, доступних на платформі LearnStore. Користувач може переглядати список товарів, ознайомлюватися з їх основними характеристиками та обирати необхідний контент.

Варіант використання «Перегляд деталей товару» дозволяє отримати детальну інформацію про конкретний продукт, зокрема опис, вартість, автора або інші характеристики навчального матеріалу.

Варіант використання «Реєстрація» призначений для створення нового облікового запису користувача в системі. Під час виконання цього сценарію користувач вводить необхідні реєстраційні дані для подальшого доступу до функціональності платформи.

Варіант використання «Вхід» забезпечує автентифікацію користувача в системі LearnStore та надає доступ до функцій, доступних зареєстрованим користувачам.

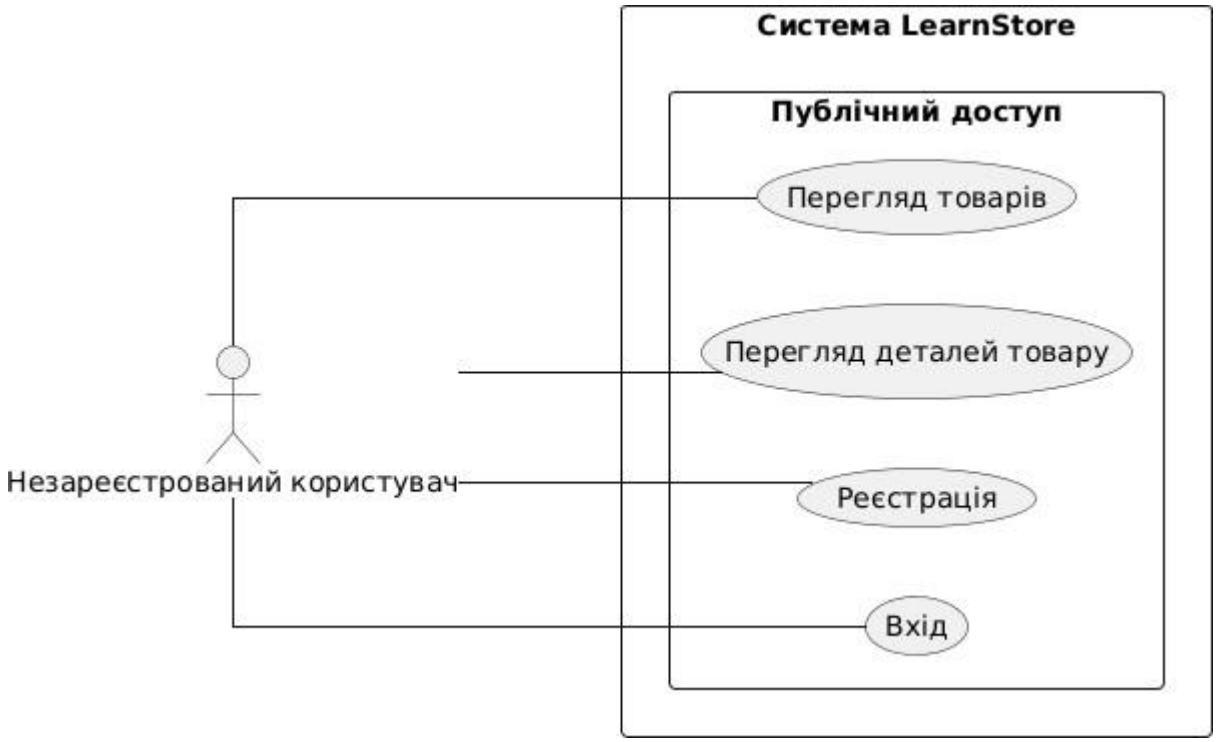


Рис. 3.7 Діаграма варіантів використання для незареєстрованого користувача

Діаграма варіантів використання системи LearnStore для незареєстрованого користувача демонструє базову функціональність публічної частини платформи та визначає основні сценарії взаємодії користувачів із системою до проходження автентифікації. Вона використовується для формування функціональних вимог до публічного інтерфейсу системи та подальшого проєктування механізмів доступу й автентифікації користувачів.

3.9 Діаграма варіантів використання для зареєстрованого користувач

Діаграма варіантів використання системи LearnStore для клієнта відображає основні сценарії взаємодії користувача із системою, пов’язані з автентифікацією, переглядом придбаного контенту, оформленням замовлень та здійсненням оплати. Основною метою діаграми є демонстрація функціональних можливостей, доступних клієнту системи, а також взаємозв’язків між окремими варіантами використання.

Основним актором діаграми є користувач типу «Клієнт», який взаємодіє із системою LearnStore через набір функціональних модулів. Діаграма структурована за пакетами відповідно до логічних підсистем застосунку.

Пакет «Аутентифікація» містить функції реєстрації, входу та виходу із системи, що забезпечують автентифікацію користувачів. Пакет «Товари» реалізує можливість перегляду бібліотеки придбаного навчального контенту та отримання детальної інформації про курси або інші цифрові продукти.

Пакет «Замовлення» містить основні бізнес-процеси, пов’язані з оформленням покупок. Користувач може створювати, переглядати та оновлювати замовлення. Під час створення замовлення система виконує додавання товарів до замовлення, перевірку його коректності та здійснення оплати.

Варіант використання «Додавання товарів до замовлення» забезпечує формування списку продуктів для покупки, а «Перевірка замовлення» відповідає за валідацію даних перед підтвердженням оформлення.

Пакет «Оплата» містить сценарій «Здійснення оплати», який забезпечує виконання платіжної операції для оформленого замовлення.

Між варіантами використання реалізовано зв’язки типу <<include>>, які відображають обов’язкове включення допоміжних сценаріїв до процесу створення замовлення. Зокрема, сценарій «Створення замовлення» включає:

1. «Додавання товарів до замовлення»;
2. «Перевірка замовлення»;

3. «Здійснення оплати».

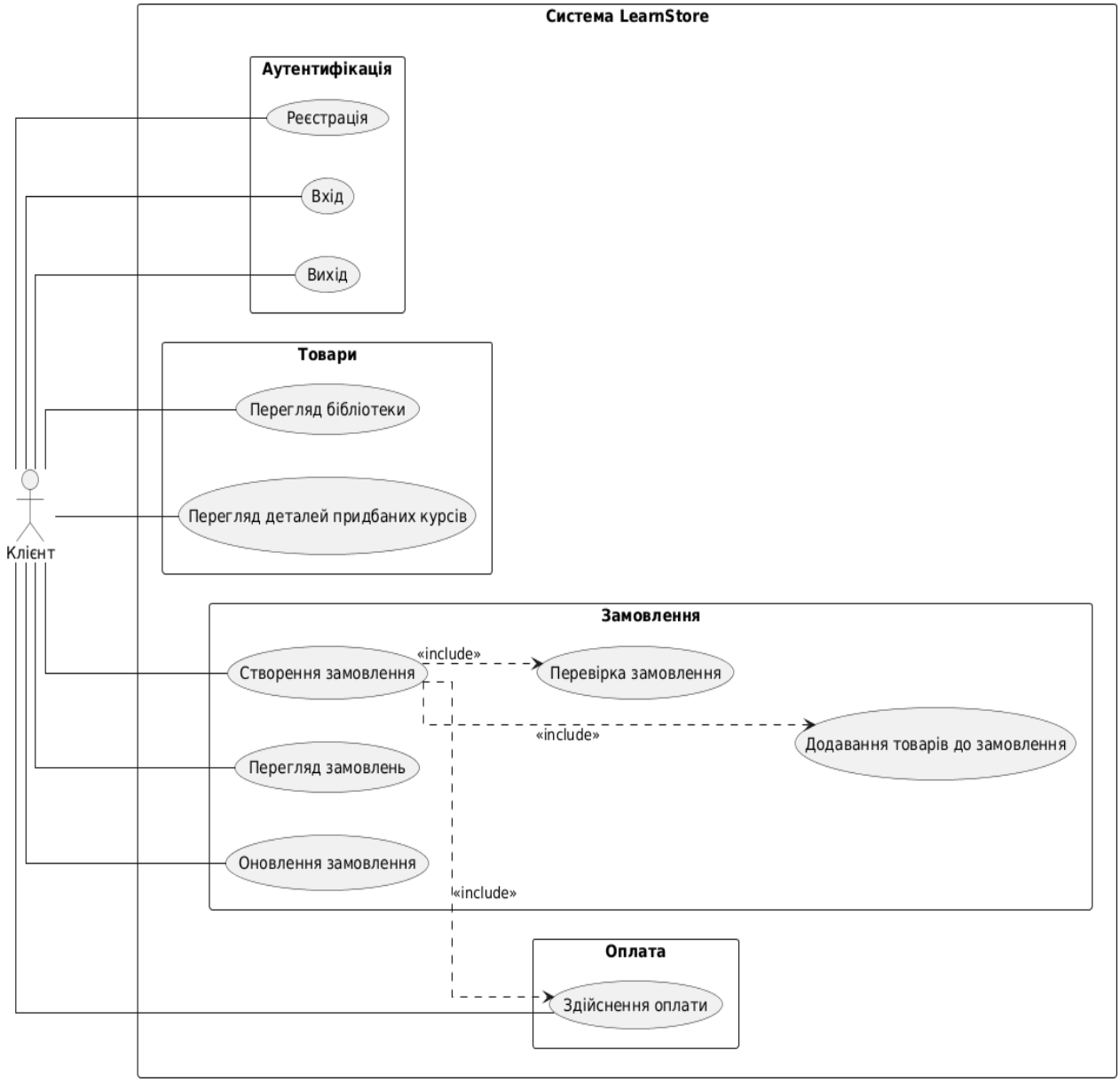


Рис. 3.8 Діаграма варіантів використання для зареєстрованого користувач

Діаграма варіантів використання системи LearnStore для клієнта демонструє основні функціональні можливості користувача та використовується як основа для подальшого проєктування архітектури й бізнес-логіки системи.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Проєкт LearnStore.Domain

Проєкт LearnStore.Domain є центральною частиною архітектури застосунку та відповідає за реалізацію предметної області системи. У даному проєкті зосереджені основні бізнес-моделі, правила роботи системи, доменні сутності та абстракції, що описують логіку функціонування web-платформи. Шар домену не залежить від інфраструктурних компонентів, бази даних, фреймворків або інтерфейсу користувача, що відповідає принципам Clean Architecture.

Основу проєкту складають доменні сутності, розміщені у просторі імен LearnStore.Domain.Entities. До них належать сутності користувачів, курсів, категорій, замовлень, авторів контенту та інших компонентів системи. Доменні сутності інкапсулюють стан і поведінку предметної області, а також забезпечують виконання основних бізнес-правил застосунку.

Однією з ключових сутностей системи є Order, яка виступає коренем агрегату та об’єднує пов’язані компоненти замовлення, зокрема елементи замовлення, інформацію про користувача та платежі. Використання агрегатів дозволяє забезпечити цілісність даних і контроль бізнес-логіки в межах окремих функціональних модулів системи.

У проєкті також використовуються значущі об’єкти (Value Objects) [4, 5], розташовані у просторі імен LearnStore.Domain.ValueObjects. Вони застосовуються для реалізації допоміжних структур даних, зокрема параметрів пошуку та фільтрації контенту. Такі об’єкти не мають окремого життєвого циклу та використовуються виключно як складові бізнес-логіки.

Для опису фіксованих станів і типів у системі використовуються перерахування (Enums). Наприклад, перелік станів замовлення дозволяє

стандартизувати обробку життєвого циклу замовлень та спрощує реалізацію бізнес-логіки застосунку.

Окрему роль у структурі проєкту відіграють інтерфейси, розміщені у просторі імен LearnStore.Domain.Interfaces. Вони визначають контракти взаємодії між доменним та інфраструктурним рівнями системи. У даному проєкті інтерфейси репозиторіїв описують операції отримання, збереження та оновлення даних без прив’язки до конкретної технології реалізації. Їх реалізація виконується на рівні проєкту LearnStore.Infrastructure, що забезпечує дотримання принципу інверсії залежностей.

Важливою особливістю шару домену є реалізація бізнес-правил безпосередньо всередині доменних сутностей. Це дає змогу гарантувати коректність роботи системи незалежно від способу взаємодії із застосунком — через MVC-інтерфейс, API або інші компоненти системи.

Завдяки ізоляції від зовнішніх технологій, використанню інтерфейсів та чіткому розподілу відповідальності проєкт LearnStore.Domain забезпечує високу тестованість, підтримуваність та масштабованість програмного забезпечення.

4.2 Шар прикладної логіки LearnStore.Application

Проєкт LearnStore.Application реалізує прикладну логіку системи та відповідає за координацію виконання основних сценаріїв роботи web-платформи. Даний шар забезпечує взаємодію між рівнем представлення та доменною моделлю, організовує обробку запитів, валідацію даних, перетворення моделей та виклики інфраструктурних компонентів. Основним призначенням шару є реалізація use-case сценаріїв системи без прямої залежності від деталей інфраструктури або засобів збереження даних.

У межах проєкту LearnStore.Application містить обробники команд і запитів, DTO-моделі, валідатори, мапери та конфігурацію сервісів прикладного рівня. Для реалізації взаємодії між компонентами використовується бібліотека MediatR, яка забезпечує централізовану обробку команд і запитів відповідно до

принципів CQRS[12]. Обробники команд та запитів відповідають за виконання окремих бізнес-сценаріїв системи, зокрема створення, оновлення, отримання та видалення даних.

Для передачі даних між шарами використовуються DTO-моделі, які дозволяють відокремити внутрішню доменну модель від моделей представлення. Перетворення DTO у доменні сутності та у зворотному напрямку виконується за допомогою маперів, що централізує логіку конвертації даних та спрощує взаємодію між компонентами системи.

Окрему роль у шарі Application відіграє механізм валідації даних. Для цього використовується бібліотека FluentValidation, яка дозволяє централізовано описувати правила перевірки команд і запитів перед їх виконанням. Такий підхід забезпечує коректність вхідних даних та зменшує кількість помилок під час роботи застосунку.

Шар прикладної логіки взаємодіє з інтерфейсами репозиторіїв, визначеними у LearnStore.Domain. Завдяки цьому Application не залежить від конкретної реалізації доступу до даних, а всі інфраструктурні компоненти підключаються через механізм Dependency Injection. Реалізації репозиторіїв знаходяться у проєкті LearnStore.Infrastructure та підключаються під час запуску застосунку.

Для централізованої реєстрації сервісів у проєкті використовується файл ApplicationRegistration.cs, у якому виконується підключення MediatR, валідаторів, маперів та інших компонентів прикладного шару. Це забезпечує зручне управління залежностями та спрощує конфігурацію системи.

Організація шару LearnStore.Application дозволяє відокремити логіку виконання сценаріїв системи від рівня представлення та інфраструктури, забезпечує високу тестованість компонентів і спрощує подальше розширення функціональності web-платформи.

4.3 Шар інфраструктури LearnStore.Infrastructure

Проєкт LearnStore.Infrastructure реалізує інфраструктурний шар системи та відповідає за роботу з технічними компонентами застосунку. До основних задач даного шару належать організація доступу до бази даних, реалізація механізмів автентифікації та авторизації, управління конфігурацією, робота із зовнішніми сервісами та забезпечення взаємодії із середовищем виконання. Інфраструктурний шар реалізує інтерфейси, визначені у Domain та Application, і не містить безпосередньої бізнес-логіки системи [4, 5].

Для роботи з базою даних у проєкті використовується технологія Entity Framework Core. Основним компонентом доступу до даних є AppDbContext, який містить набори сутностей системи та конфігурації зв’язків між ними. Через DbContext виконується взаємодія з таблицями бази даних, формування запитів та збереження змін.

У межах шару Infrastructure реалізовано репозиторії для роботи з основними сутностями системи, зокрема користувачами, курсами, категоріями, замовленнями та авторами навчального контенту. Репозиторії реалізують інтерфейси, визначені у шарі Domain, та забезпечують ізоляцію прикладної логіки від конкретної технології доступу до даних. Така реалізація дозволяє дотримуватись принципу Dependency Inversion та спрощує подальшу зміну або розширення інфраструктурних компонентів [4, 5, 10].

Для організації автентифікації та управління користувачами використовується ASP.NET Core Identity. У проєкті реалізовано механізми реєстрації користувачів, авторизації, роботи з ролями та генерації JWT-токенів для захищеної взаємодії із системою. Конфігурація JWT-автентифікації та параметри безпеки централізовано налаштовуються у спеціалізованих сервісах інфраструктурного шару.

Окрему роль у LearnStore.Infrastructure відіграють механізми роботи з конфігурацією та секретними даними. Для цього використовуються спеціальні провайдери, які відповідають за отримання паролів, ключів доступу та інших

конфіденційних параметрів. Провайдери відокремлюють бізнес-логіку від способів зберігання конфігураційних даних та забезпечують підвищення безпеки системи.

У проєкті також реалізовано механізми ініціалізації бази даних та підтримки міграцій Entity Framework Core. Це дозволяє автоматизувати процес створення структури бази даних, оновлення схем таблиць та підготовку середовища виконання застосунку.

Для централізованого налаштування інфраструктурних компонентів використовується клас ServiceCollectionExtensions, у якому виконується реєстрація сервісів, репозиторіїв, механізмів автентифікації та інших залежностей через Dependency Injection. Його реалізація забезпечує зручне управління конфігурацією застосунку та спрощує масштабування системи.

Організація шару LearnStore.Infrastructure забезпечує ізоляцію технічних деталей від бізнес-логіки системи, спрощує підтримку застосунку та дозволяє гнучко змінювати інфраструктурні рішення без впливу на інші частини web-платформи.

4.4 Шар представлення LearnStore.Web

Шар представлення у проєкті LearnStore відповідає за взаємодію користувачів та зовнішніх клієнтів із функціональністю web-платформи. Основним призначенням даного шару є прийом HTTP-запитів, обробка взаємодії з користувацьким інтерфейсом, передача даних до прикладного шару та формування відповідей системи. Presentation-рівень побудований таким чином, щоб мінімізувати кількість бізнес-логіки у web-компонентах та забезпечити чітке розмежування відповідальностей між інтерфейсом користувача та прикладною логікою системи.

У межах проєкту шар представлення складається з декількох web-проєктів: LearnStore.Web.Api, LearnStore.Web.MVC та LearnStore.Admin. Кожен із них

виконує окрему роль у структурі системи та відповідає за конкретний тип взаємодії із застосунком.

Проєкт LearnStore.Web.Api реалізує HTTP API для взаємодії із зовнішніми клієнтами та іншими компонентами системи. Контролери API приймають HTTP-запити, виконують первинну перевірку вхідних даних, формують DTO або команди та передають їх до прикладного шару через MediatR. Після обробки запиту система формує HTTP-відповіді з відповідними кодами стану та результатами виконання операцій. Для документування та тестування API використовується Swagger/OpenAPI, що спрощує взаємодію із сервісом під час розробки та налагодження.

Проєкт LearnStore.Web.MVC відповідає за реалізацію користувацької частини web-платформи. Для побудови інтерфейсу використовуються ASP.NET Core MVC та Razor Views. Контролери даного проєкту виконують роль посередника між користувацьким інтерфейсом та прикладною логікою системи, формують ViewModel-моделі та передають дані у представлення.

Окремо використовується проєкт LearnStore.Admin, який відповідає за адміністративну частину системи. Даний проєкт забезпечує засоби управління курсами, користувачами та іншими компонентами платформи. Для доступу до адміністративного функціоналу використовуються механізми автентифікації та авторизації на основі ролей користувачів.

Для взаємодії між Presentation та Application шарами використовується MediatR, що дозволяє передавати команди та запити без прямої залежності web-компонентів від реалізацій бізнес-логіки[12]. Така технологія забезпечує слабке зв’язування між компонентами системи та підвищує тестованість застосунку.

У web-проєктах також реалізовано механізми серверної валідації даних, обробки помилок, автентифікації користувачів та захисту маршрутів. Централізована конфігурація сервісів виконується у файлі Program.cs через підключення Application та Infrastructure шарів за допомогою Dependency Injection.

Організація шару представлення забезпечує чітке відокремлення інтерфейсу користувача від бізнес-логіки системи, спрощує підтримку web-платформи та дозволяє масштабувати окремі компоненти застосунку незалежно один від одного.

4.5 Модуль LearnStore.Database.Migrations

Проєкт LearnStore.Database.Migrations відповідає за централізоване застосування міграцій бази даних у середовищах виконання застосунку. Основним призначенням даного модуля є автоматична підготовка структури бази даних та схеми Identity перед запуском web-платформи або під час процесів розгортання системи.

У межах проєкту реалізовано окремий Host-сервіс, який виконує застосування EF Core міграцій для основної бази даних та системи автентифікації користувачів [10, 11]. Для цього використовуються контексти AppDbContext та AppIdentityDbContext, які створюються через спеціалізовані фабрики контекстів.

Основним компонентом модуля є файл Program.cs, у якому виконується конфігурація середовища, реєстрація сервісів прикладного та інфраструктурного шарів, а також підключення механізмів роботи із секретами та конфігурацією застосунку. Під час запуску сервіс використовує ті самі налаштування та механізми Dependency Injection, що й основні web-проєкти системи.

Для автоматичного запуску міграцій використовується сервіс MigrationService, який реалізує інтерфейс IHostedService. Під час запуску застосунку сервіс виконує перевірку середовища виконання та застосовує міграції до бази даних за допомогою методу Database.MigrateAsync(). Такий підхід дозволяє автоматизувати оновлення структури бази даних під час розгортання системи [3, 9] .

У режимі розробки застосування міграцій виконується окремо від автоматичного запуску сервісу, що дозволяє уникнути небажаних змін

локального середовища розробника. У Production та CI/CD середовищах модуль забезпечує автоматичне оновлення структури бази даних відповідно до актуального стану проєкту.

Модуль LearnStore.Database.Migrations інтегрується з шарами Application та Infrastructure, використовуючи їх конфігурацію, сервіси та механізми роботи із секретами. Модуль дозволяє забезпечити єдиний підхід до налаштування середовища виконання та узгодженість конфігурації між усіма компонентами системи [3, 5, 19].

Використання окремого сервісу для управління міграціями забезпечує централізоване та контрольоване оновлення структури бази даних, спрощує процес розгортання застосунку та зменшує ризик виникнення розбіжностей між різними середовищами виконання системи.

4.6 Контейнеризація та розгортання застосунку з використанням Docker

Для забезпечення спрощеного розгортання, ізоляції середовища виконання та стандартизації процесу запуску застосунку у проєкті LearnStore використовується платформа Docker. Контейнеризація дозволяє запускати всі компоненти системи в ізольованому середовищі незалежно від конфігурації операційної системи або локального оточення розробника.

У межах проєкту Docker використовується для контейнеризації web-застосунків, бази даних MS SQL Server, сервісу застосування міграцій та допоміжних компонентів інфраструктури. Платформ разгортання забезпечує однакові умови виконання застосунку в локальному середовищі, тестових середовищах та під час production-розгортання [6, 19].

Для кожного web-проєкту реалізовано окремий Dockerfile із використанням multi-stage build. На першому етапі використовується SDK-образ .NET для компіляції та публікації застосунку, а на фінальному — lightweight runtime-образ ASP.NET Core для запуску вже зібраного застосунку. Це дозволяє

зменшити розмір фінального контейнера та оптимізувати процес розгортання [1, 3, 9].

Контейнери web-проєктів працюють у середовищі Production та використовують централізовану систему конфігурації. Для запуску застосунків відкриваються відповідні HTTP-порти, а виконання контейнерів здійснюється від окремого користувача, що підвищує рівень безпеки системи.

У проєкті також реалізовано механізм роботи із секретами через Docker Secrets. Конфіденційні дані, зокрема паролі бази даних і JWT-ключі, не зберігаються безпосередньо у конфігураційних файлах застосунку. У production-середовищі система використовує спеціалізований DockerSecretProvider, який отримує секрети з каталогу /run/secrets/. У режимі локальної розробки застосовується механізм User Secrets.

Для автоматизації роботи з базою даних реалізовано окремий сервіс LearnStore.Database.Migrations, який відповідає за автоматичне застосування EF Core міграцій у production-середовищі та під час CI/CD-розгортання. Це дозволяє централізовано підтримувати актуальну структуру бази даних без необхідності ручного запуску міграцій.

Окремо використовується контейнер MS SQL Server, конфігурація якого виконується через Docker Swarm. Для збереження даних застосовується Docker Volume, що забезпечує збереження інформації незалежно від життєвого циклу контейнера [6, 10, 14].

Використання Docker у проєкті LearnStore дозволило стандартизувати процес розгортання, спростити перенесення системи між різними середовищами виконання, підвищити ізоляцію компонентів та забезпечити безпечне зберігання конфіденційних даних.

4.7 Модульне тестування системи

Для забезпечення стабільності роботи застосунку, перевірки коректності бізнес-логіки та зменшення ризику виникнення помилок у проєкті LearnStore використовується окремий тестовий проєкт LearnStore.Tests.Unit. Модульні тести дозволяють перевіряти роботу окремих компонентів системи в ізоляції від зовнішніх залежностей, зокрема бази даних, файлової системи або мережевих сервісів [4,15,16].

Основною задачею модульного тестування є перевірка поведінки окремих функціональних елементів системи, таких як мапери, валідатори, обробники команд і запитів, а також допоміжні компоненти прикладного та інфраструктурного шарів. При цьому інтеграційні сценарії взаємодії з реальною базою даних не входять до складу unit-тестів і виконуються окремо.

Структура тестового проєкту організована відповідно до архітектури основного застосунку. Окремі директорії відповідають певним функціональним зонам системи:

1. Application/Mappers — тести маперів DTO та доменних моделей;
2. Application/UseCases — тести обробників команд і запитів;
3. Application/Validators — тести валідаторів FluentValidation;
4. Infrastructure — тести інфраструктурних сервісів та допоміжних компонентів.

Для реалізації тестування у проєкті використовуються сучасні бібліотеки та інструменти:

1. xUnit — основний фреймворк для написання unit-тестів;
2. NSubstitute — бібліотека для створення mock-об’єктів і підміни залежностей;
3. FluentAssertions — засіб для формування читабельних перевірок результатів тестування;
4. AutoFixture — бібліотека для автоматичної генерації тестових даних.

Під час написання тестів використовується підхід Arrange — Act — Assert, який передбачає розділення підготовки даних, виконання дії та перевірки результату [15, 16]. Такий підхід підвищує читабельність тестів і спрощує їх підтримку.

Зовнішні залежності системи ізолюються за допомогою mock-об’єктів. Репозиторії, сервіси та інші компоненти передаються у тестовані класи через Dependency Injection та замінюються підставними реалізаціями за допомогою NSubstitute. Для перевірки взаємодії між компонентами використовуються методи Received() та DidNotReceive().

Окрема увага приділяється перевірці коректності обробки некоректних або null-значень, а також тестуванню сценаріїв валідації та трансформації даних між DTO та доменними моделями. Для порівняння складних об’єктів використовується метод BeEquivalentTo() бібліотеки FluentAssertions.

Використання окремого проєкту модульного тестування дозволяє підвищити надійність системи, спростити підтримку коду та забезпечити контроль коректності роботи критичних компонентів застосунку LearnStore.

ПЕРЕЛІК ПОСИЛАНЬ

1. Freeman A. Pro ASP.NET Core 8. 11th ed. New York : Apress, 2024.1045 p.
Price M. C# 12 and .NET 8 – Modern Cross-Platform Development
2. Fundamentals. 8th ed. Birmingham : Packt Publishing, 2023. 826 p.
Lock A. ASP.NET Core in Action. 3rd ed. Shelter Island : Manning Publications, 2023. 950 p.
3. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Boston : Pearson Education, 2018. 432 p.
4. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2019. 560 p.
5. Richardson C. Microservices Patterns. Shelter Island : Manning Publications, 2018. 520 p.
6. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O’Reilly Media, 2017. 616 p.
7. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston : Addison-Wesley, 2019. 416 p.
8. Microsoft Corporation. ASP.NET Core documentation. URL: https://learn.microsoft.com/aspnet/core/ (date of access: 17.05.2026).
9. Microsoft Corporation. Entity Framework Core documentation. URL: https://learn.microsoft.com/ef/core/ (date of access: 17.05.2026).
10. Microsoft Corporation. ASP.NET Core Identity documentation. URL: https://learn.microsoft.com/aspnet/core/security/authentication/identity/ (date of access:17.05.2026).

11. MediatR documentation. URL: https://github.com/jbogard/MediatR (date of access: 17.05.2026).

12. Date C. J. An Introduction to Database Systems. 8th ed. Boston : Pearson, 2019. 1024 p.

13. Silberschatz A., Korth H. F.,Sudarshan S. Database System Concepts.7th ed.

14. New York : McGraw-Hill Education, 2019. 1376 p.

15. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner’s Approach. 9th ed. New York : McGraw-Hill, 2019. 880 p.

16. Sommerville I. Software Engineering. 10th ed. Boston : Pearson, 2020. z816 p.

17. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Boston : Addison-Wesley, 2018. 208 p.

18. Gamma E., Beck K. Concurrency in C# Cookbook. Sebastopol : O’Reilly Media, 2022. 410 p.

19. Brown S. Software Architecture for Developers. London : Leanpub, 2022. 310 p.

20. Price M. C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals. 8th ed. Birmingham : Packt Publishing, 2023. 826 p.

ВИСНОВКИ

У результаті виконання дипломної роботи було розроблено інформаційну систему LearnStore, призначену для автоматизації процесів управління навчальними матеріалами, користувачами, замовленнями та контентом платформи. Основною метою роботи було створення сучасного програмного рішення, яке забезпечує зручне адміністрування даних, підтримує розширення функціональності та відповідає сучасним вимогам до вебзастосунків.

У процесі виконання дипломної роботи було проведено аналіз предметної області та існуючих підходів до побудови систем електронної комерції й платформ для керування навчальним контентом. На основі проведеного аналізу були визначені основні функціональні вимоги до системи, а також сформовано архітектурні рішення, що забезпечують гнучкість і масштабованість програмного забезпечення.

Під час проєктування системи було розроблено структуру бази даних, UML-діаграми класів, компонентів та взаємодії модулів. Для побудови програмного рішення використано принципи Clean Architecture, що дозволило чітко розділити рівні відповідальності між компонентами системи та спростити подальшу підтримку проєкту. Також було застосовано підхід CQRS із використанням бібліотеки MediatR для організації обробки команд і запитів.

Серверну частину застосунку реалізовано на платформі ASP.NET Core із використанням мови програмування C#. Для роботи з базою даних застосовано Entity Framework Core та механізм міграцій, що забезпечує контроль версій структури бази даних і спрощує процес розгортання системи. Реалізовано механізми автентифікації та авторизації користувачів на основі JWT-токенів і ASP.NET Identity, що дозволяє забезпечити належний рівень безпеки доступу до функціоналу системи.

У межах роботи було реалізовано функціональні модулі управління товарами, авторами, категоріями, замовленнями, ролями користувачів та

звітністю. Крім цього, система підтримує розширення функціоналу без необхідності суттєвої зміни вже реалізованої логіки, що є важливою перевагою при подальшому розвитку програмного продукту.

Під час розробки значну увагу було приділено якості програмного коду, дотриманню принципів SOLID та забезпеченню слабкої зв’язаності між компонентами системи. Це дозволило створити структуру застосунку, яка є зрозумілою для підтримки, тестування та подальшого масштабування. Також було реалізовано систему обробки винятків, логування та перевірки коректності даних.

Отримані результати підтверджують, що поставлена мета дипломної роботи була досягнута, а всі визначені завдання — виконані. Розроблена система може бути використана як основа для створення повноцінної комерційної або навчальної онлайн-платформи.

У подальшому можливе розширення функціоналу шляхом додавання рекомендаційних сервісів, інтеграції з хмарними сервісами, мобільного клієнта та систем аналітики користувацької активності.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Багачок В.Д., Ярошевський О.В. Порівняльний аналіз архітектур MVC і MVVM у контексті проектування та розробки веб-застосунків. Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії"– Київ: ДУІКТ, 2025. – С. 429-432.

2. Багачок В.Д., Ярошевський О.В. Застосування принципів предметно-орієнтованого проєктування (DDD) для побудови масштабованих web-платформ. VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» – Київ: ДУІКТ, 2026. – С. 328-330.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

РОЗРОБКА WEB-ПЛАТФОРМИ ДЛЯ ДИСТРИБУЦІЇ НАВЧАЛЬНИХ КУРСІВ ТА НАВЧАЛЬНИХ МАТЕРІАЛІВ

Виконав студент 4 курсу
групи ПД-44
Багачок Вадим Дмитрович
Керівник роботи
асист.каф. Ярошевський Олександр Вікторович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення ефективності процесу дистрибуції та придбання навчальних курсів у web-середовищі за рахунок розробки web-платформи.

Об’gкт дослідження: процес дистрибуції та придбання навчальних курсів у середовищі дистанційного навчання.

Предмет дослідження: методи, моделі та програмні засоби розробки web-платформи для дистрибуції та придбання навчальних курсів.

ЗАДАЧІ КВАЛІФІКАЦІМНОЇ РОБОТИ

1. Провести аналіз предметної області та існуючих рішень у сфері електронної комерції й платформ для навчального контенту, а також визначити функціональні та нефункціональні вимоги до системи LearnStore.
2. Розробити архітектуру системи на основі принципів Clean Architecture з урахуванням вимог масштабованості та підтримуваності.
3. Спроєктувати структуру бази даних та створити UML-діаграми класів, компонентів і взаємодії модулів системи.
4. Реалізувати серверну частину вебплатформи на платформі ASP.NET Core із використанням мови C#.
5. Інтегрувати ORM Entity Framework Core, реалізувати механізм міграцій бази даних та забезпечити коректну роботу з даними вебплатформи.
6. Реалізувати систему автентифікації та авторизації користувачів вебплатформи на основі JWT і ASP.NET Identity.
7. Впровадити архітектурний підхід CQRS із використанням бібліотеки MediatR для організації обробки команд і запитів у мехах вебплатформи.
8. Розробити основні функціональні модулі системи вебплатформи (управління товарами, користувачами, замовленнями, авторами, категоріями та ролями).
9. Провести тестування системи.

3

АНАЛІЗ АНАЛОГІВ

№	Характеристика	Udemy	Coursera	Prometheus	LearnStore
1	Модель платформи	Маркетплейс онлайн-курсів	Академічна освітня платформа	Освітня платформа дистанційного навчання	Маркетплейс
2	Формат навчального контенту	Перевахно відеокурси	Відеолекції та навчальні програми	Відеокурси та тести	Структурований текстовий навчальний контент та супровідні матеріали
3	Навігація всередині курсу	Перевахно послідовне проходхення	Проходхення відповідно до програми	Послідовна структура курсу	Вільний перехід міх будь-якими темами та розділами курсу
4	Пошук та фільтрація курсів	Базова фільтрація за категоріями	Пошук за напрямами навчання	Категорійний пошук	Категорійний пошуку та фільтрації курсів
5	Розміщення власних курсів	Доступно авторам після модерації	Обмехено партнерами платформи	Обмехено організаціями-партнерами	Будь-який зареєстрований продавець мохе створювати та публікувати курси відповідно до правил платформи
6	Контроль над контентом	Обмехений політикою сервісу	Обмехений вимогами платформи	Частковий контроль	Повний контроль автора над структурою та наповненням курсу

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні :

1. Реєстрація, авторизація користувачів та керування ролями (Customer, Seller, Admin) із розмехуванням доступу до функціоналу системи.
2. Перегляд каталогу товарів із мохливістю пошуку та фільтрації за назвою, автором та категорією.
3. Перегляд детальної інформації про товар із відображенням опису, ціни, продавця.
4. Додавання товарів у кошик, зміна їх кількості та оформлення замовлення.
5. Обробка замовлень із створенням Order та позицій товарів із перевіркою коректності даних.
6. Проведення оплати замовлення з фіксацією результату транзакції у системі.
7. Управління хиттєвим циклом замовлення зі зміною статусів (New, Pending, Completed, Cancelled).
8. Керування товарами продавцем, включаючи створення, редагування, видалення та перегляд замовлень, пов'язаних із його товарами.
9. Редагування профілю користувача та перегляд історії замовлень.

Нефункціональні :

1. Час відповіді API ≤ 300 мс
2. Безпека даних через JWT, ASP.NET Identity та захищене зберігання секретів
3. Транзакційна цілісність даних і контроль доменних інваріантів
4. Покриття критичних компонентів тестами ≥ 70%
5. Підтримка Docker-контейнеризації
6. Архітектура на основі Clean Architecture з чітким поділом шарів
7. Надання REST API із документуванням через Swagger та стандартизованою обробкою помилок.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Технології розробки клієнтської частини:
- CSS
- HTML5
- Blazor

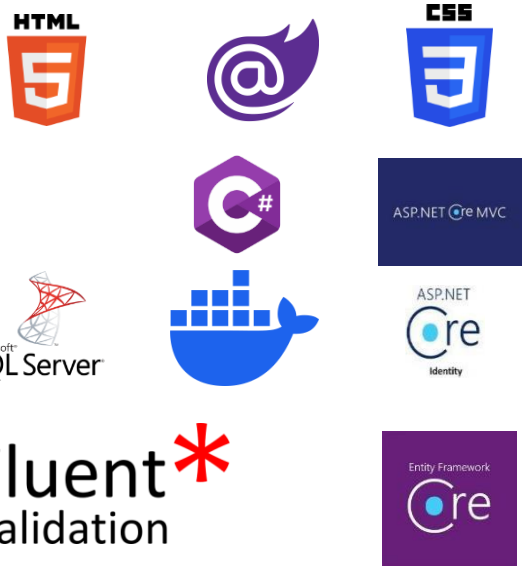
Технології серверної частини:
- C#
- ASP.NET Core MVC

Технології доступу до даних:
- Entity Framework Core
- Microsoft SQL Server

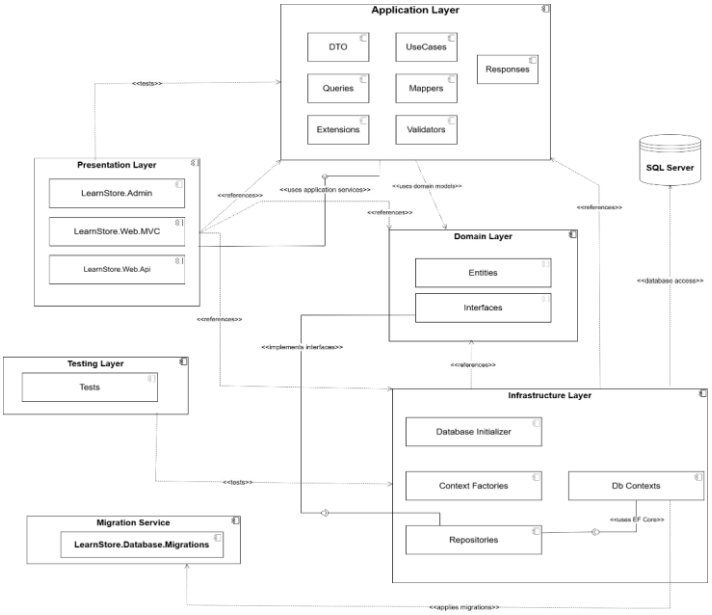
Технології безпеки та керування користувачами:
- ASP.NET Core Identity

Технології забезпечення якості даних :
- FluentValidation

Технології контейнеризації та розгортання:
- Docker

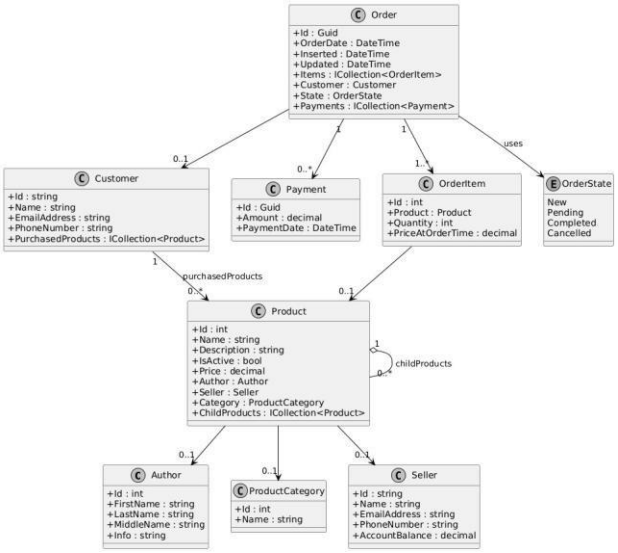


ДІАГРАМА КОМПОНЕНТІВ



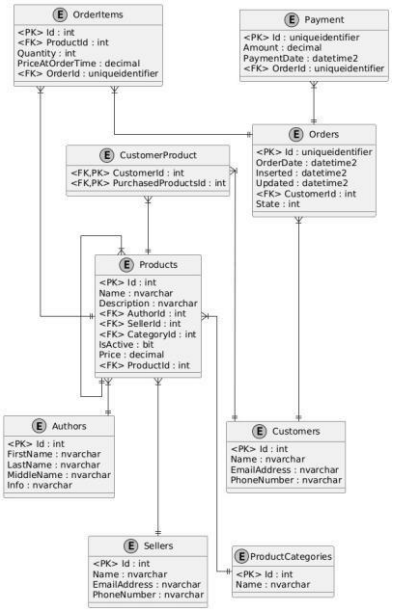
7

ДІАГРАМА КЛАСІВ ДОМЕННОГО СЛОЯ



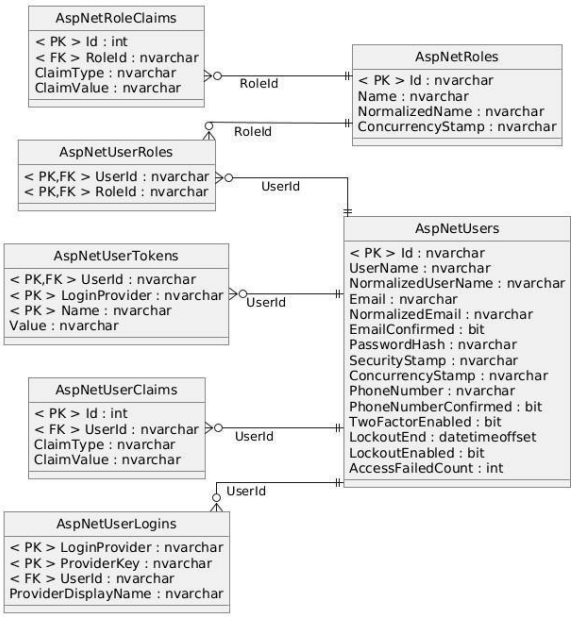
8

ER-ДІАГРАМА БАЗИ ДАНИХ СИСТЕМИ



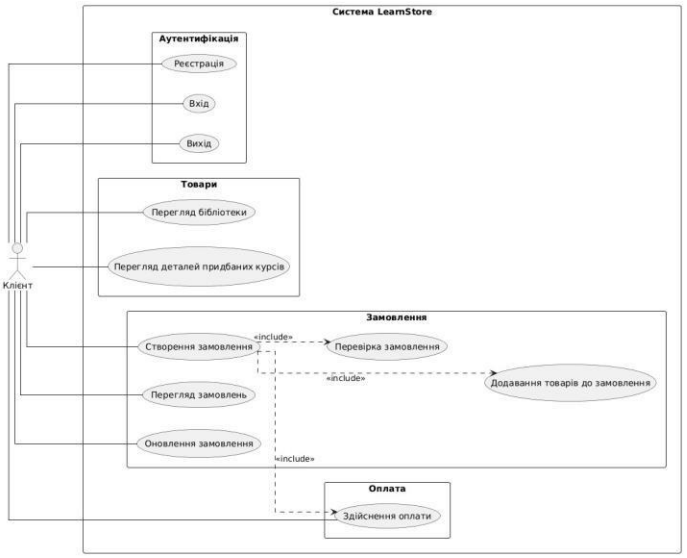
9

ДІАГРАМА СИСТЕМНИХ ТАБЛИЦЬ АВТЕНТИФІКАЦІЇ ТА АВТОРИЗАЦІЇ



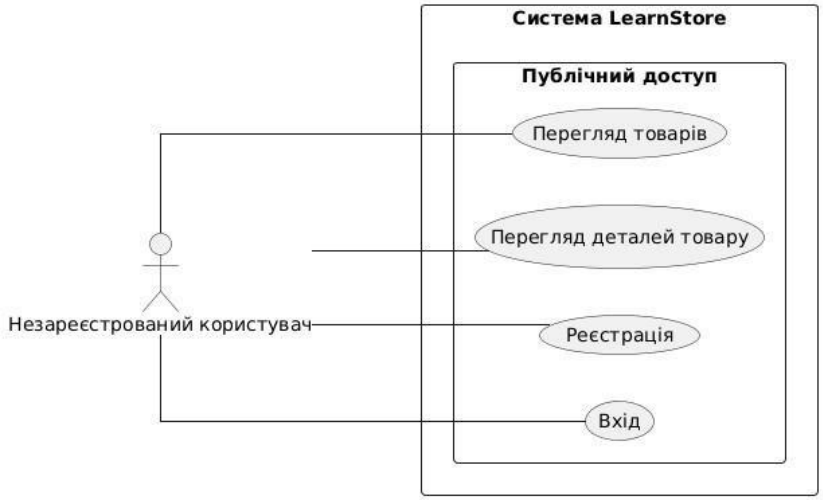
10

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ ДЛЯ ЗАРЕЄСТРОВАНОГО КОРИСТУВАЧА



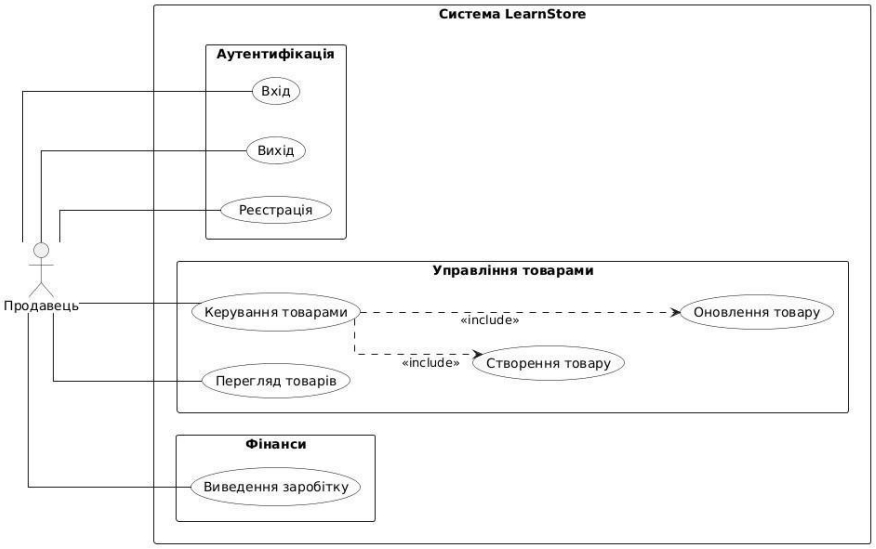
11

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ ДЛЯ НЕЗАРЕЄСТРОВАНОГО КОРИСТУВАЧА



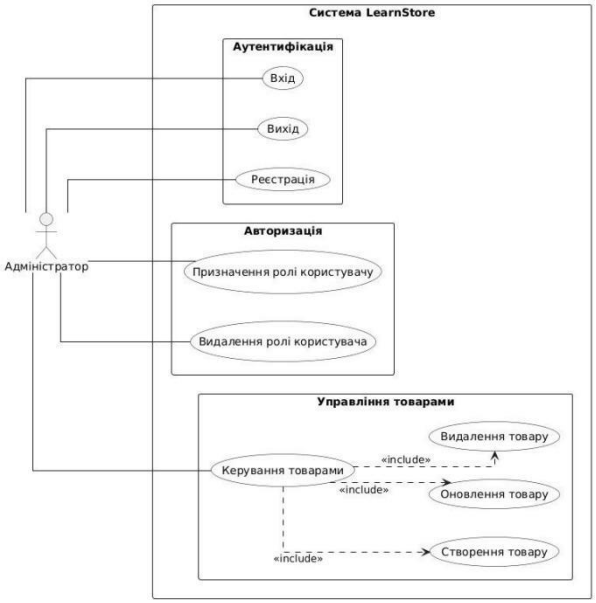
12

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ ДЛЯ ПРОДАВЦЯ



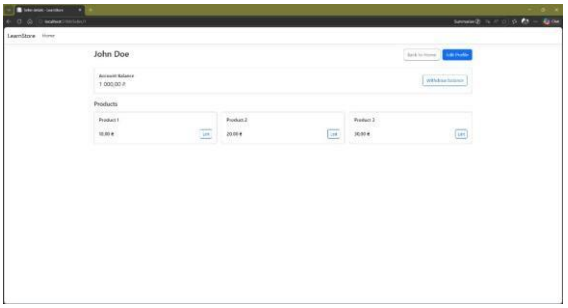
13

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ ДЛЯ АДМІНІСТРАТОРА

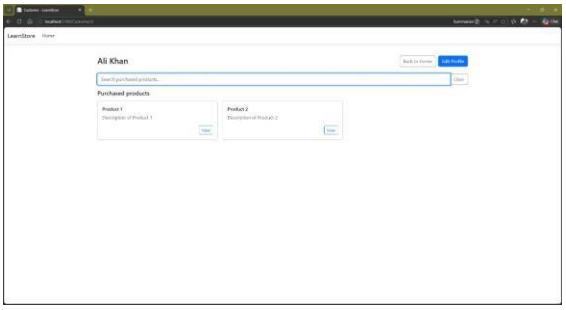


14

ЕКРАННІ ФОРМИ



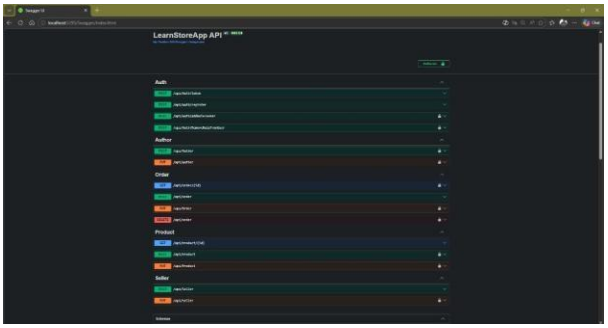
Кабінет продавця



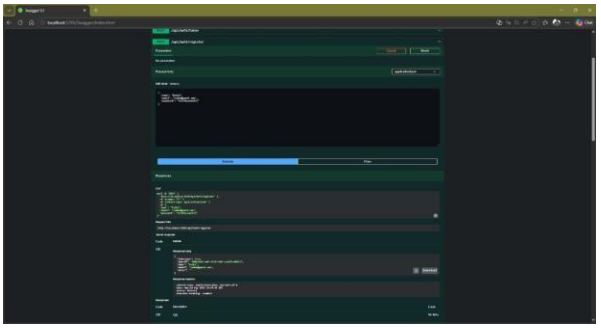
Кабінет користувача

15

ЕКРАННІ ФОРМИ



Інтерфейс API системи



Виконання HTTP-запиту в Swagger

16

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Багачок В.Д., Ярошевський О.В. Порівняльний аналіз архітектур MVC і MVVM у контексті проектування та розробки веб-застосунків. Всеукраїнська науково -технічна конференція "Виклики та рішення в програмній інхенерії"– Київ: ДУІКТ, 2025. – С. 429-432.

Багачок В.Д., Ярошевський О.В. Застосування принципів предметно-орієнтованого проєктування (DDD) для побудови масштабованих web-платформ. VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно - комунікаційних технологіях» – Київ: ДУІКТ, 2026. – С. 328-330.

17

ВИСНОВКИ

- Проведено аналіз предметної області, існуючих платформ дистанційного навчання та електронної комерції, що дозволило визначити ключові функціональні та нефункціональні вимоги до системи LearnStore.
- Розроблено архітектуру вебплатформи на основі принципів Clean Architecture, яка забезпечує розділення відповідальностей міх компонентами системи, спрощує її підтримку та подальше масштабування.
- Спроєктовано структуру бази даних і побудовано UML-діаграми, що відобрахають основні сутності, взаємозв'язки та логіку взаємодії компонентів системи.
- Реалізовано серверну частину вебплатформи з використанням ASP.NET Core, Entity Framework Core, MediatR та підходу CQRS, що забезпечує ефективну обробку даних і бізнес-логіки.
- Впровадхено механізми автентифікації та авторизації на основі JWT та ASP.NET Identity, що дозволяє забезпечити безпечний доступ користувачів до ресурсів системи.
- Розроблено основні функціональні модулі платформи, зокрема управління курсами, авторами, категоріями, користувачами, замовленнями та ролями.
- Реалізовано модель маркетплейсу навчального контенту, яка надає авторам мохливість самостійно створювати та публікувати курси, а користувачам — здійснювати їх пошук, придбання та проходження.

18

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
public class Order
{
    public Guid Id { get; set; }
    public DateTime OrderDate { get; set; } =
DateTime.UtcNow;
    public DateTime Inserted { get; set; } =
DateTime.UtcNow;
    public DateTime Updated { get; set; } =
DateTime.UtcNow;
    public ICollection<OrderItem> Items { get;
init; } = [];
    public Customer Customer { get; init; }
    public OrderState State { get; set; } =
OrderState.New;
    public ICollection<Payment> Payments { get;
set; } = [];

    public Order()
    {
    }
    public Order(Customer
customer,ICollection<OrderItem> orderItems)
    {

ArgumentNullException.ThrowIfNull(customer,
nameof(customer));

ArgumentNullException.ThrowIfNull(orderItems,
nameof(orderItems));
        if(orderItems.Count == 0)
            throw new ArgumentException("Order
must have at least one item.",
nameof(orderItems));
        Items = orderItems;
        Customer = customer;
    }

}
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;

 public string EmailAddress { get; set; }=
string.Empty;
    public string PhoneNumber { get; set; } =
string.Empty;
    public ICollection<Product>
PurchasedProducts { get; set; } = [];
}
public record class OrderDto
 {
     public Guid Id { get; set; }
public DateTime OrderDate { get; set; } =
DateTime.UtcNow;
     public DateTime Inserted { get; set; } =
DateTime.UtcNow;
     public DateTime Updated { get; set; } =
DateTime.UtcNow;
     public ICollection<OrderItemDto> Items { get;
set; } = [];
     public CustomerDto? Customer { get; set; }
     public OrderState State { get; set; } =
OrderState.New;
     public ICollection<PaymentDto> Payments {
get; set; } = [];
 }
   public record class CustomerDto
   {
       public int Id { get; set; }
       public string Name { get; set; } =
string.Empty;
       public string EmailAddress { get; set; } =
string.Empty;
       public string PhoneNumber { get; set; } =
string.Empty;
       public ICollection<ProductDto>
PurchasedProducts { get; set; } = [];
   }
public interface IMapper
{
    object ToDto(object domain);
    object ToDomain(object dto);
}
public interface IMapper<TDomain, TDto> :
IMapper
{
    TDto ToDto(TDomain domain);
    TDomain ToDomain(TDto dto);
}
   public class OrderMapper : IMapper<Order,
OrderDto>
   {
       private IMapper<OrderItem,OrderItemDto>
_orderItemMapper ;
       private IMapper<Payment,PaymentDto>
_paymentMapper;
       private IMapper<Customer,CustomerDto>
_customerMapper;

       public OrderMapper(IMapperRegistry
mapperRegistry)
       {
           _customerMapper=
mapperRegistry.Get<Customer, CustomerDto>();
```

```
        _orderItemMapper=
mapperRegistry.Get<OrderItem,
OrderItemDto>();
        _paymentMapper=
mapperRegistry.Get<Payment, PaymentDto>();
    }

    public Order ToDomain(OrderDto orderDto)
    {

ArgumentNullException.ThrowIfNull(orderDto,
nameof(orderDto));
        return new Order()
        {
            Id = orderDto.Id,
            OrderDate = orderDto.OrderDate,
            Inserted = orderDto.Inserted,
            Updated = orderDto.Updated,
  Items = [.. orderDto.Items.Select(i =>
_orderItemMapper.ToDomain(i))],
             Customer = orderDto.Customer != null ?
 _customerMapper.ToDomain(orderDto.Customer
) : null,
            State = orderDto.State,
            Payments = [..
orderDto.Payments.Select(p  =>
_paymentMapper.ToDomain(p))]
        };
    }

    public object ToDomain(object dto)
    {
        return ToDomain((OrderDto)dto);
    }

    public OrderDto ToDto (Order order)
    {

ArgumentNullException.ThrowIfNull(order,name
of(order));
        return new OrderDto
        {
            Id = order.Id,
            OrderDate = order.OrderDate,
            Inserted = order.Inserted,
            Updated = order.Updated,
            Items = [..order.Items.Select(i =>
_orderItemMapper.ToDto(i))],
            Customer = order.Customer != null ?
_customerMapper.ToDto(order.Customer) : null,
            State = order.State,
            Payments = [..order.Payments.Select(p
=> _paymentMapper.ToDto(p))]
        };
    }

    public object ToDto(object domain)
    {
        return ToDto((Order)domain);
    }
public class CustomerMapper:
IMapper<Customer, CustomerDto>
 {
    private IMapper<Product, ProductDto>
_productMapper;
    public CustomerMapper(IMapperRegistry
mapperRegistry)
    {
        _productMapper=
mapperRegistry.Get<Product, ProductDto>();
    }
    public Customer ToDomain(CustomerDto
customerDto)
    {

ArgumentNullException.ThrowIfNull(customerDt
o, nameof(customerDto));
        return new Customer
        {
            Id = customerDto.Id,
            Name = customerDto.Name,
            EmailAddress =
customerDto.EmailAddress,
            PhoneNumber =
customerDto.PhoneNumber,
            PurchasedProducts =
customerDto.PurchasedProducts.Select(p  =>
_productMapper.ToDomain(p)).ToList()

        };
    }

    public object ToDomain(object dto)
    {
      return ToDomain((CustomerDto)dto);
    }

    public CustomerDto ToDto (Customer
customer)
    {

ArgumentNullException.ThrowIfNull(customer,n
ameof(customer));
        return new CustomerDto
        {
            Id = customer.Id,
            Name = customer.Name,
            EmailAddress = customer.EmailAddress,
            PhoneNumber = customer.PhoneNumber,
            PurchasedProducts =
customer.PurchasedProducts.Select(p  =>
_productMapper.ToDto(p)).ToList()
```

```
        };
    }

    public object ToDto(object domain)
    {
        return ToDto((Customer)domain);
    }

 }
public class
CreateOrderCommandValidator:AbstractValidato
r<CreateOrderCommand>
 {
    public CreateOrderCommandValidator()
    {
        RuleFor(o => o.Customer)
            .NotNull().ChildRules(customer =>
            {
                customer.RuleFor(c =>
c!.Id).GreaterThan(0);
            });

        RuleFor(o =>
o.Items).NotEmpty().ForEach(item=>
        {
            item.NotNull().ChildRules(orderItem =>
            {
                orderItem.RuleFor(i =>
i!.Id).GreaterThan(0);
                orderItem.RuleFor(i =>
i!.Product).NotNull().ChildRules(product =>
                {
                    product.RuleFor(p =>
p!.Id).GreaterThan(0);
                });
                orderItem.RuleFor(i =>
i!.Quantity).GreaterThan(0);
            });
        });

    }
 }
public UpdateOrderCommandValidator()
 {
    RuleFor(o => o!.Id).NotEmpty();
    RuleFor(o => o!.Items).NotEmpty();
    RuleForEach(o =>
o!.Items).NotNull().ChildRules(item =>
    {
        item.RuleFor(i => i!.Id).NotEmpty();
    });
    RuleFor(o =>
o!.Customer).NotNull().ChildRules(customer =>
    {
        customer.RuleFor(c => c!.Id).NotEmpty();
    });
    RuleFor(o=>o!.Payments).NotNull();
    RuleForEach(o =>
o!.Payments).NotNull().ChildRules(payment =>
    {
        payment.RuleFor(p => p!.Id).NotEmpty();
    });
 }
  public class UpdateOrderCommand:
IRequest<Unit>
  {
      public Guid Id { get; set; }

      public ICollection<OrderItemDto> Items {
get; set; } = [];
      public CustomerDto? Customer { get; set; }

      public ICollection<PaymentDto> Payments {
get; set; } = [];
  }
public record class
CreateOrderCommand:IRequest<Guid>
 {
    public CustomerDto? Customer { get; init; }
    public ICollection<OrderItemDto> Items { get;
init; } = [];

 }
public async Task<Guid>
Handle(CreateOrderCommand command,
CancellationToken cancellationToken)
{

ArgumentNullException.ThrowIfNull(command,
nameof(command));

ArgumentNullException.ThrowIfNull(command.I
tems, nameof(command.Items));

ArgumentNullException.ThrowIfNull(command.
Customer, nameof(command.Customer));
   await
ValidationRules.ValidateAndThrowAsync(comm
and, cancellationToken);

   var customerMapper =
Mappers.OfType<IMapper<Customer,
CustomerDto>>().First();
   var orderItemMapper =
Mappers.OfType<IMapper<OrderItem,OrderItem
Dto>>().First();

   var orderItems = command.Items.Select(item
=> orderItemMapper.ToDomain(item)).ToList();
   Order order =
new(customerMapper.ToDomain(command.Custo
mer), orderItems)
```

```
    {

        OrderDate = DateTime.UtcNow,
        Inserted = DateTime.UtcNow,
        Updated = DateTime.UtcNow,
        State = OrderState.New
    };

    await OrderRepository.SaveOrderAsync(order,
cancellationToken);
    return order.Id;
}
public async Task<Unit>
Handle(UpdateOrderCommand command,
CancellationToken cancellationToken)
 {

ArgumentNullException.ThrowIfNull(command,
nameof(command));

ArgumentNullException.ThrowIfNull(command.
Customer, nameof(command.Customer));
    await
Validator.ValidateAndThrowAsync(command,
cancellationToken);

    var customerMapper =
mapperRegistry.Get<Customer,CustomerDto>()
?? throw new ArgumentException("Customer
mapper not found", nameof(mapperRegistry));
    var orderItemMapper =
mapperRegistry.Get<OrderItem,OrderItemDto>()
?? throw new ArgumentException("OrderItem
mapper not found", nameof(mapperRegistry));
    var paymentMapper =
mapperRegistry.Get<Payment,PaymentDto>() ??
throw new ArgumentException("Payment mapper
not found", nameof(mapperRegistry));

    var customer=
customerMapper.ToDomain(command.Customer)
;
     var orderItems = command.Items.Select(item
 => orderItemMapper.ToDomain(item)).ToList();


    var order = new Order(customer,orderItems)
    {
        Id = command.Id,
        Payments = command.Payments.Select(p =>
paymentMapper.ToDomain(p)).ToList(),
        Items = orderItems,
        Updated = DateTime.UtcNow
    };

    await
OrderRepository.SaveOrderAsync(order,cancellat
ionToken);
    return Unit.Value;
 }
public class AppDbContext : DbContext
 {
    public
AppDbContext(DbContextOptions<AppDbConte
xt> options) : base(options)
    {
    }
    public DbSet<Domain.Entities.Customer>
Customers { get; set; }
    public DbSet<Domain.Entities.Order> Orders
{ get; set; }
    public DbSet<Domain.Entities.Product>
Products { get; set; }
    public DbSet<OrderItem> OrderItems { get;
set; }
    public DbSet<Domain.Entities.Seller> Sellers
{ get; set; }

    public
DbSet<Domain.Entities.ProductCategory>
ProductCategories { get; set; }

    public DbSet<Domain.Entities.Author >
Authors { get; set; }

    protected override void
OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);

builder.Entity<Customer>().HasMany(c=>c.Purc
hasedProducts).WithMany();

builder.Entity<Order>().HasMany(o=>o.Items).W
ithOne().HasForeignKey("OrderId").OnDelete(De
leteBehavior.Cascade);
    }
  public class AppDbContextFactory :
IDesignTimeDbContextFactory<AppDbContext>
  {

     private readonly
IDatabaseInitializer<AppDbContext>?
_initializer;
     private readonly IConfiguration
_configuration;
     private readonly IPasswordProvider
_passwordProvider;
     public AppDbContextFactory()
     {
```

```
        var environment =
Environment.GetEnvironmentVariable("ASPNET
CORE_ENVIRONMENT") ?? "Production";
        _configuration = new
ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json",
optional: false)

.AddJsonFile($"appsettings.{environment}.json",
optional: true)

.AddUserSecrets<AppDbContextFactory>(option
al: true)
            .AddEnvironmentVariables()
            .Build();
        _passwordProvider= new
PasswordProvider(new
WindowsUserSecretsProvider(_configuration));
    }
    public
AppDbContextFactory(IDatabaseInitializer<App
DbContext> initializer, IConfiguration
configuration, IPasswordProvider
passwordProvider)
    {
        _initializer = initializer ?? throw new
ArgumentNullException(nameof(initializer));
        _configuration = configuration ?? throw
new
ArgumentNullException(nameof(configuration));
        _passwordProvider = passwordProvider ??
throw new
ArgumentNullException(nameof(passwordProvid
er));
    }

    public AppDbContext
CreateDbContext(string[] args)
    {
        var connectionString =
GetConnectionString(_configuration);
        var optionsBuilder = new
DbContextOptionsBuilder<AppDbContext>();

optionsBuilder.UseSqlServer(connectionString);

        return new
AppDbContext(optionsBuilder.Options);
    }
    private string
GetConnectionString(IConfiguration
configuration)
    {
        var connectionString =
configuration.GetConnectionString("LearnStoreM
igration")
                    ?? throw new
InvalidOperationException("Conection string
'LearnStoreMigration' not found.");
        var builder = new
SqlConnectionStringBuilder(connectionString);
        var migrationUserName =
configuration["MigrationUser:UserName"];

        if
(!string.IsNullOrEmpty(migrationUserName) &&
migrationUserName == builder.UserID &&
string.IsNullOrEmpty(builder.Password) &&
!builder.IntegratedSecurity)
        {
            _initializer?.EnsureDatabaseAndUser();

            builder.Password =
_passwordProvider.GetPassword("migrator_pass
word");
        }

        return builder.ConnectionString;
    }


 }
public class DatabaseInitializer<TContext> :
IDatabaseInitializer<TContext> where TContext :
DbContext
{
   private readonly IPasswordProvider
_passwordProvider;
   private readonly
DatabaseInitializerOptions<TContext>  _options;

   public DatabaseInitializer(IPasswordProvider
passwordProvider, IConfiguration configuration,
IOptions<DatabaseInitializerOptions<TContext>>
options)
   {
      _passwordProvider = passwordProvider ??
throw new
ArgumentNullException(nameof(passwordProvid
er));
      _options = options?.Value ?? throw new
ArgumentNullException(nameof(options));
   }

   public void EnsureDatabaseAndUser()
   {
      var saPassword =
_passwordProvider.GetPassword(_options.SaPass
wordKey);
```

```
        if(string.IsNullOrEmpty(saPassword))
            throw new
InvalidOperationException("SA password is
required to ensure midration user exists");

        var connectionString =
 _options.ConnectionString
            ?? throw new
InvalidOperationException("Connection string
'LearnStoreMigration' not found ");
        var builder = new
SqlConnectionStringBuilder(connectionString)
        {
            UserID = "SA",
            Password = saPassword,
            IntegratedSecurity = false
        };

        var builderForSa = new
SqlConnectionStringBuilder(builder.ConnectionSt
ring)
        {
            InitialCatalog = "master"
        };
        var connection= new
SqlConnection(builderForSa.ToString());


EnsureDatabaseExists(builderForSa.ConnectionSt
ring, builder.InitialCatalog);

EnsureMigrationUserExists(builder.ConnectionSt
ring);

EnsureAppUserExists(builder.ConnectionString);
    }

    private void EnsureAppUserExists(string
connectionString)
    {
        var appUserName =
 _options.AppUserName;
        if (string.IsNullOrEmpty(appUserName))
            throw new
InvalidOperationException("App user name is not
configured.");

        var password =
_passwordProvider.GetPassword(_options.AppPa
sswordKey);
        var roles = "EXEC sp_addrolemember
'db_datareader', N'" + appUserName + "'; EXEC
sp_addrolemember 'db_datawriter', N'" +
appUserName + "';";
        EnsureUserExists(connectionString,
appUserName, password, roles);
    }
    private void EnsureMigrationUserExists(string
connectionString)
    {
        var migrationUserName =
_options.MigrationUserName;
        if
(string.IsNullOrEmpty(migrationUserName))
            throw new
InvalidOperationException("Migration user name
is not configured.");
        var password =
_passwordProvider.GetPassword(_options.Migrat
ionPasswordKey);
        var roles = "EXEC sp_addrolemember
'db_owner', N'" + migrationUserName + "';";
        EnsureUserExists(connectionString,
migrationUserName, password, roles);
    }
    private void EnsureUserExists(string
connectionString, string userName, string
password,string roles)
    {
        using var connection = new
SqlConnection(connectionString);
        var sql = $@"
                IF NOT EXISTS (SELECT * FROM
sys.server_principals WHERE name =
N'{userName}')
                BEGIN
                    CREATE LOGIN [{userName}]
WITH PASSWORD = '{password}';
                END

                IF NOT EXISTS (SELECT * FROM
sys.database_principals WHERE name =
N'{userName}')
                BEGIN
                    CREATE USER [{userName}] FOR
LOGIN [{userName}];
                    {roles}
                END";

        using var command = new SqlCommand(sql,
connection);
        connection.Open();
        command.ExecuteNonQuery();
    }

    private void EnsureDatabaseExists(string
connectionString, string databaseName)
    {
        using var connection = new
SqlConnection(connectionString);
        var sql = $@"IF NOT EXISTS (SELECT
name FROM sys.databases WHERE name =
```

```
N'{databaseName}') BEGIN  CREATE
DATABASE [{databaseName}] END";

      using var command = new SqlCommand(sql,
connection);
      connection.Open();
      command.ExecuteNonQuery();
   }
}

public record class
DatabaseInitializerOptions<TContext> where
TContext : DbContext
{
   public string ConnectionString { get; set; } =
string.Empty;
   public string SaPasswordKey { get; set; } =
"mssql_sa_password";
   public string MigrationUserName { get; set; } =
string.Empty;
   public string AppUserName { get; set; } =
string.Empty;
   public string MigrationPasswordKey { get; set;
} = "migrator_password";
   public string AppPasswordKey { get; set; } =
"app_password";
}
public class EFOrderRepository (AppDbContext
Context): IOrderRepository
{
   private readonly AppDbContext _context =
Context ?? throw new
ArgumentNullException(nameof(Context));
   public IEnumerable<Order> Orders =>
_context.Orders.Include(o=>o.Customer).Include(
o=>o.Items)

.Include(o=>o.Payments).Include(o=>o.State);
   public async Task<Order>
DeleteOrderAsync(Guid orderId,
CancellationToken cancellationToken)
   {
      var order = await
_context.Orders.FindAsync(new object[] {
orderId }, cancellationToken);
      if (order is not null)
      {
         _context.Orders.Remove(order);
         await
_context.SaveChangesAsync(cancellationToken);

      }
      return order;
   }

   public async Task<IEnumerable<Order>>
GetOrdersAsync(OrderSearchCriteria criteria,
CancellationToken cancellationToken)
   {
      var query = _context.Orders.Include(o =>
o.Customer)
         .Include(o => o.Items).ThenInclude(i =>
i.Product).ThenInclude(p => p.Category)
         .Include(o => o.Payments)
         .AsEnumerable();

      if (criteria.OrderId.HasValue)
         query = query.Where(o => o.Id ==
criteria.OrderId.Value);

      if (criteria.CustomerId.HasValue)
         query = query.Where(o => o.Customer !=
null && o.Customer.Id ==
criteria.CustomerId.Value);

      return query.ToList();
   }

   public async Task SaveOrderAsync(Order
order, CancellationToken cancellationToken)
   {
      ArgumentNullException.ThrowIfNull(order,
nameof(order));

ArgumentNullException.ThrowIfNull(order.Cust
omer, nameof(order.Customer));

ArgumentNullException.ThrowIfNull(order.Items
, nameof(order.Items));


ArgumentNullException.ThrowIfNull(order.Paym
ents, nameof(order.Payments));

      _context.Attach(order.Customer);


      foreach (var item in order.Items)
      {
         _context.Attach(item.Product);
      }

      var existingOrder = await _context.Orders
.Include(o => o.Customer).Include(o =>
o.Items).Include(o =>
o.Payments).FirstOrDefaultAsync(o => o.Id ==
order.Id, cancellationToken);

      if (existingOrder == null)
         _context.Orders.Add(order);
      else
```

```
        {

_context.Entry(existingOrder).CurrentValues.Set
Values(order);


UpdateOrderPayments(existingOrder,order);
            UpdateOrderItems(existingOrder,
 order);
        }
        await
_context.SaveChangesAsync(cancellationToken);
    }
    private void UpdateOrderItems(Order
existingOrder, Order updatedOrder)
    {
        foreach (var updatedItem in
updatedOrder.Items)
        {
            var existingItem = existingOrder.Items
                .FirstOrDefault(i => i.Id ==
updatedItem.Id);

            if (existingItem == null)
                existingOrder.Items.Add(updatedItem);
            else
            {

_context.Entry(existingItem).CurrentValues.SetV
alues(updatedItem);
                existingItem.Product =
updatedItem.Product;
            }
        }

        var itemsToRemove =
existingOrder.Items.Where(i =>
!updatedOrder.Items.Any(ui => ui.Id ==
i.Id)).ToList();

        foreach (var item in itemsToRemove)
            existingOrder.Items.Remove(item);


    }
    private void UpdateOrderPayments(Order
existingOrder, Order updatedOrder)
    {
        foreach (var updatedPayment in
updatedOrder.Payments)
        {
            var existingPayment =
existingOrder.Payments .FirstOrDefault(p => p.Id
== updatedPayment.Id);

            if (existingPayment == null)
existingOrder.Payments.Add(updatedPayment);
            else

_context.Entry(existingPayment).CurrentValues.S
etValues(updatedPayment);
        }

        var paymentsToRemove =
existingOrder.Payments.Where(p =>
!updatedOrder.Payments.Any(up => up.Id ==
p.Id)).ToList();

        foreach (var payment in
paymentsToRemove)

existingOrder.Payments.Remove(payment);
    }

    public async Task<bool>
UpdateOrderStageAsync(Guid orderId,
OrderState newStage, CancellationToken
cancellationToken)
    {
        var order = await
_context.Orders.FindAsync(new object[] {
orderId }, cancellationToken);

        if (order is null)
            return false;

        order.State = newStage;

        await
_context.SaveChangesAsync(cancellationToken);
        return true;
    }
}
public static class ServiceCollectionExtensions
 {
    public static void AddMsSqlDataAccess(this
IServiceCollection services, IConfiguration
configuration)
    {

ArgumentNullException.ThrowIfNull(configurati
on, nameof(configuration));
        var connectionString =
configuration.GetConnectionString("LearnStoreA
pp") ?? throw new
InvalidOperationException("Conection string
'LearnStoreMigration' not found.");


services.AddDbContext<AppDbContext>((sp,opti
ons) =>
```

```
        {
            var builder = new
SqlConnectionStringBuilder(connectionString);
            if (!builder.IntegratedSecurity &&
string.IsNullOrEmpty(builder.Password))
            {
                var passwordProvider =
sp.GetRequiredService<IPasswordProvider>();
                var password =
passwordProvider.GetPassword(GetConfigOrDef
ault(configuration, "AppUser:PasswordKey",
"app_password"));
                builder.Password =
!string.IsNullOrEmpty(password) ? password :
throw new InvalidOperationException("Password
for database connection is not provided.");
            }

options.UseSqlServer(builder.ConnectionString);
        });


services.AddScoped<ICustomerRepository,EFCu
stomerRepository>();
        services.AddScoped<IOrderRepository,
EFOrderRepository>();
        services.AddScoped<IProductRepository,
EFProductRepository>();
        services.AddScoped<ISellerRepository,
EFSellerRepository>();

services.TryAddSingleton<IPasswordProvider,
PasswordProvider>();


services.Configure<DatabaseInitializerOptions<A
ppDbContext>>(opt =>
        {
            opt.ConnectionString = connectionString;
            opt.MigrationUserName =
GetConfigOrDefault(configuration,
"MigrationUser:UserName",
"learnStore_migrator");
            opt.MigrationPasswordKey =
GetConfigOrDefault(configuration,
"MigrationUser:PasswordKey",
"learnStore_password");
            opt.AppUserName =
GetConfigOrDefault(configuration,
"AppUser:UserName", "learnStore_app");
            opt.AppPasswordKey =
GetConfigOrDefault(configuration,
"AppUser:PasswordKey", "app_password");
        });

services.AddTransient<IDatabaseInitializer<App
DbContext>,
DatabaseInitializer<AppDbContext>>();

services.AddSingleton<IDesignTimeDbContextFa
ctory<AppDbContext>,
AppDbContextFactory>();
    }
    public static void AddIdentity(this
IServiceCollection services, IConfiguration
configuration)
    {

ArgumentNullException.ThrowIfNull(configurati
on, nameof(configuration));
        var connectionString =
configuration.GetConnectionString("LearnStoreId
entity") ?? throw new
InvalidOperationException("Identity connection
string not found.");
        services.AddIdentity<IdentityUser,
IdentityRole>().AddEntityFrameworkStores<App
IdentityDbContext>();

services.AddDbContext<AppIdentityDbContext>
((sp, options) =>
        {
            var builder = new
SqlConnectionStringBuilder(connectionString);
            if (!builder.IntegratedSecurity &&
string.IsNullOrEmpty(builder.Password))
            {
                var passwordProvider =
sp.GetRequiredService<IPasswordProvider>();
                var password =
passwordProvider.GetPassword(GetConfigOrDef
ault(configuration, "IdentityUser:PasswordKey",
"identity_password"));
                builder.Password =
!string.IsNullOrEmpty(password) ? password :
throw new InvalidOperationException("Password
for database connection is not provided.");
            }

options.UseSqlServer(builder.ConnectionString);
        });


services.Configure<DatabaseInitializerOptions<A
ppIdentityDbContext>>(opt =>
        {
            opt.ConnectionString = connectionString;
            opt.MigrationUserName =
GetConfigOrDefault(configuration,
```

```
"IdentityMigrationUser:UserName",
"learnStore_migrator");
            opt.MigrationPasswordKey =
GetConfigOrDefault(configuration,
"IdentityMigrationUser:PasswordKey",
"migrator_password");
            opt.AppUserName =
GetConfigOrDefault(configuration,
"IdentityUser:UserName", "learnStoreIdentity");
            opt.AppPasswordKey =
GetConfigOrDefault(configuration,
"IdentityUser:PasswordKey",
"identity_password");
        });

        services.AddScoped<IAuthService,
IdentityAuthService>();

services.AddTransient<IDatabaseInitializer<AppI
dentityDbContext>,
DatabaseInitializer<AppIdentityDbContext>>();

services.AddSingleton<IDesignTimeDbContextFa
ctory<AppIdentityDbContext>,
AppIdentityDbContextFactory>();

services.TryAddSingleton<IPasswordProvider,
PasswordProvider>();
    }

    public static void AddJwtAuthentication(this
IServiceCollection services, IConfiguration
configuration)
    {

services.AddSingleton<IJwtTokenGenerator,
JwtTokenGenerator>();

services.AddSingleton<IJwtOptionsProvider,
JwtOptionsProvider>();

        services.Configure<JwtOptions>(opt =>
        {
            var provider = new
JwtOptionsProvider(configuration,
services.BuildServiceProvider().GetRequiredServ
ice<ISecretProvider>());
            var options = provider.GetOptions();
            opt.Issuer = options.Issuer;
            opt.Audience = options.Audience;
            opt.Key = options.Key;
            opt.ExpireHours = options.ExpireHours;
        });
        services.AddAuthentication(options =>
        {
            options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme =
JwtBearerDefaults.AuthenticationScheme;
        }).AddJwtBearer(options =>
        {
            var sp= services.BuildServiceProvider();
            var jwtOptions =
sp.GetRequiredService<IOptions<JwtOptions>>()
.Value;
            options.TokenValidationParameters =
new TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                ValidIssuer = jwtOptions.Issuer,
                ValidAudience = jwtOptions.Audience,
                IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes
(jwtOptions.Key))
            };
        });
    }
    public static void
UseWindowsUserSecrets<TMarker>(this
IServiceCollection services,
IConfigurationBuilder configuration) where
TMarker : class
    {
        configuration.AddUserSecrets<TMarker>();
        services.AddSingleton<ISecretProvider,
WindowsUserSecretsProvider>();
    }
    public static void
UseWindowsUserSecrets(this IServiceCollection
services)
    {
        services.AddSingleton<ISecretProvider,
WindowsUserSecretsProvider>();
    }
    public static void UseDockerSecrets(this
IServiceCollection services)
    {
        services.AddSingleton<ISecretProvider,
DockerSecretProvider>();
    }
    private static string
GetConfigOrDefault(IConfiguration
configuration, string path, string defaultValue)
    {
        var raw = configuration[path];
        return string.IsNullOrWhiteSpace(raw) ?
defaultValue : raw;
    }
```

```
 }

var builder =
WebApplication.CreateBuilder(args);



builder.Services.AddControllers();

builder.Services.AddOpenApi();
builder.Services.AddMapster();
builder.Services.AddApplication();
var connectionString =
builder.Configuration.GetConnectionString("Lear
nStoreApp") ?? throw new
InvalidOperationException("Connection string
'LearnStoreAppUser' not found.");
if (builder.Environment.IsDevelopment())

builder.Services.UseWindowsUserSecrets<Progra
m>(builder.Configuration);
else
   builder.Services.UseDockerSecrets();
builder.Services.AddMsSqlDataAccess(builder.C
onfiguration);

builder.Services.AddIdentity(builder.Configuratio
n);
builder.Services.AddJwtAuthentication(builder.C
onfiguration);
builder.Services.AddSwaggerGen(options =>
{
   options.SwaggerDoc("v1", new OpenApiInfo {
Title = "Contrabanda API", Version = "v1" });


options.AddSecurityDefinition(JwtBearerDefaults
.AuthenticationScheme, new
OpenApiSecurityScheme
   {
      Description = "JWT Authorization header
using the Bearer scheme. Example: \"Bearer
{token}\"",
      Name = "Authorization",
      In = ParameterLocation.Header,
      Type = SecuritySchemeType.Http,
      Scheme = "bearer"
   });


options.OperationFilter<SecurityRequirementsOp
erationFilter>(true,
JwtBearerDefaults.AuthenticationScheme);
});

var app = builder.Build();

app.UseExceptionHandler(errorApp  =>
{
   errorApp.Run(async context =>
   {
      context.Response.StatusCode =
StatusCodes.Status500InternalServerError;
      context.Response.ContentType =
"application/json";

      var errorFeature =
context.Features.Get<IExceptionHandlerFeature>
();
      if (errorFeature != null)
      {
         var ex = errorFeature.Error;
         var result = new
         {
            Message = "An unexpected error
occurred",
            Details = ex.Message
         };
         await
context.Response.WriteAsJsonAsync(result);
      }
   });

});
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
   app.MapOpenApi();
   app.UseSwagger();
   app.UseSwaggerUI();
}

app.UseAuthentication();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();


if (!app.Environment.IsDevelopment())
   ApplyDatabaseMigrations(app);


app.Run();

static void
ApplyDatabaseMigrations(WebApplication  app)
{
   using (var scope = app.Services.CreateScope())
   {
```

```
        var dbContext =
scope.ServiceProvider.GetRequiredService<IDesi
gnTimeDbContextFactory<AppDbContext>>().C
reateDbContext([]);
        dbContext.Database.Migrate();
    }
}
public class PasswordProvider:IPasswordProvider
{
    private readonly ISecretProvider
_secretProvider;
    public PasswordProvider(ISecretProvider
secretProvider)
    {
        _secretProvider = secretProvider;
    }
    public string GetPassword(string
key)=>_secretProvider.GetSecret(key);
}
public class WindowsUserSecretsProvider :
ISecretProvider
{

  private readonly IConfiguration _configuration;
    public
WindowsUserSecretsProvider(IConfiguration
configuration)
    {
        _configuration = configuration ?? throw new
ArgumentNullException(nameof(configuration));
    }
    public string GetSecret(string key)
    {
        var value = _configuration[key];
        if (string.IsNullOrEmpty(value))
            throw new
InvalidOperationException($"Password for key
'{key}' not found in configuration under
'Secrets:{key}'.");
        return value;
```