Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри комп’ютерних наук,
доктор технічних наук, професор
Андрій БОНДАРЧУК
«01 » червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
на здобуття освітнього ступеня «бакалавр»
спеціальність 122 Комп’ютерні науки
освітня програма 122.00.01 Інформатика

Тема: МОДУЛЬ CRM-СИСТЕМИ ДЛЯ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ ОБЛІКУ ЗАМОВЛЕНЬ КЛІЄНТІВ

Виконав
студент групи Інб-2-22-4.0д
Петрушишин Артем Максимович

_____
(підпис)

Науковий керівник
кандидат педагогічних наук, доцент
Вембер В.П.

_____
(підпис)

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп'ютерних наук,
доктор технічних наук, професор
Андрій БОНДАРЧУК
31 жовтня 2025 року

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«Модуль CRM-системи для автоматизації процесів обліку замовлень клієнтів»

Виконавець студент групи Інб-2-22-4.0д,
спеціальності 122 Комп'ютерні науки,
освітньої програми 122.00.01 Інформатика
Петрушишин Артем Максимович

1. Вихідні дані: наукові публікації у сфері CRM та автоматизації бізнес-процесів; технічна документація .NET 8, Blazor Web App, ASP.NET Core; документація EF Core 8, SQLite, Serilog, xUnit; законодавство України у сфері інформаційних технологій.
2. Основні завдання: проаналізувати існуючі CRM-рішення; визначити функціональні та нефункціональні вимоги; спроектувати архітектуру та модель даних; реалізувати модулі управління клієнтами, товарами, замовленнями, аналітикою, аудитом та аутентифікацією; розробити модуль тестування (61 тест); оформити пояснювальну записку.
3. Пояснювальна записка: Обсяг – до 60 стор. формату А4 комп'ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.
4. Графічні матеріали: схема програми, схема логічної бази даних, UML-діаграма класів, UML-діаграма компонентів, скріншоти реалізованого інтерфейсу.
5. Додатки: лістинги ключових модулів серверного та клієнтського проєктів; результати виконання автоматизованих тестів.
6. Строк подання роботи на кафедру: 01 червня 2026 року

Науковий керівник
канд. пед. наук, доцент,
_____ Вембер В.П.
3 червня 2026 року

Виконавець:
_____ Петрушишин А.М.
3 червня 2026 року

КАЛЕНДАРНИЙ ПЛАН

№	Етап виконання роботи	Термін виконання	Примітка
1	Затвердження теми кваліфікаційної роботи бакалавра	31.10.25	Виконано
2	Аналіз проблеми, вивчення аналогів, формування цілей і завдань	01.11.25 – 11.01.26	Виконано
3	Аналіз існуючих CRM-рішень та визначення функціональних і нефункціональних вимог	26.01.26 – 15.03.26	Виконано
4	Проектування архітектури системи, моделі даних та сервісного шару	16.03.26 – 31.03.26	Виконано
5	Реалізація модулів управління клієнтами, товарами, замовленнями, аналітикою, журналом аудиту та аутентифікацією	01.04.26 – 15.04.26	Виконано
6	Розробка та виконання автоматизованого тестування (61 тест у 7 класах)	16.04.26 – 30.04.26	Виконано
7	Подання роботи на перевірку, антиплагіат та внесення правок	01.05.26 – 15.05.26	Виконано
8	Підготовка пояснювальної записки, графічних матеріалів, додатків.	25.05.25 – 12.06.26	Виконано
9	Захист кваліфікаційної роботи	17.06.26	

«Затверджую»
Науковий керівник
канд. пед. наук, доцент, Вембер В.П.

Індивідуальний план склав
Петрушишин А.М.

РЕФЕРАТ

Пояснювальна записка: 60 с., 3 розділи, 3 рисунки, 5 таблиць, 33 джерела.

Актуальність теми. Сучасний ринок малого та середнього бізнесу потребує ефективних інструментів автоматизації обліку клієнтів і замовлень. Більшість існуючих CRM-рішень є хмарними, що створює залежність від постійного інтернет-з'єднання та зовнішніх сервісів. Розробка веборієнтованої CRM-системи, що функціонує локально на платформі .NET 8, є актуальним практичним завданням для підприємств, які прагнуть незалежності від хмарної інфраструктури.

Об'єкт дослідження – процес проектування та реалізації модуля CRM-системи для автоматизації обліку замовлень клієнтів.

Предмет дослідження – методи та засоби розробки веборієнтованих CRM-систем на платформі .NET 8 із застосуванням фреймворку Blazor Web App, об'єктно-реляційного відображувача Entity Framework Core та вбудованої СУБД SQLite.

Мета роботи – розробка та реалізація модуля CRM-системи для автоматизації процесів обліку замовлень клієнтів на платформі .NET 8.

Завдання роботи: проаналізувати існуючі CRM-рішення та обґрунтувати вибір технологічного стеку; визначити функціональні та нефункціональні вимоги до системи; спроектувати архітектуру та модель даних; реалізувати модулі управління клієнтами, товарами, замовленнями, аналітикою, журналом аудиту та аутентифікацією; розробити модуль автоматизованого тестування; оформити пояснювальну записку.

У результаті виконання роботи розроблено модуль CRM-системи з повним функціональним стеком: управлінням клієнтами, товарами, замовленнями, аналітичним дашбордом, журналом аудиту та системою аутентифікації і авторизації. Реалізовано серверний проєкт (11 сервісів бізнес-логіки, 5 REST-контролерів), клієнтський проєкт (21 Blazor-компонент) та

модуль тестування (61 тест у 7 класах з часом виконання менше 1,8 с). Всі автоматизовані тести виконуються успішно.

Практичне значення одержаних результатів полягає у можливості використання розробленої системи підприємствами малого та середнього бізнесу для автоматизації обліку клієнтів і замовлень без залежності від хмарних сервісів.

Перспективи подальшого розвитку: розширення функціональності системи шляхом додавання модулів звітності та інтеграції з зовнішніми сервісами, розробка мобільного клієнта, впровадження механізмів багатокористувацького доступу з розмежуванням прав.

Ключові слова: crm-система, автоматизація бізнес-процесів, облік замовлень, .net 8, blazor, asp.net core, entity framework core, sqlite, автоматизоване тестування, blazor web app.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API	Application Programming Interface – програмний інтерфейс застосунку.
ASP.NET Core	ASP.NET Core – фреймворк для розробки веб-застосунків на платформі .NET.
Blazor	Blazor – фреймворк для створення інтерактивних веб-інтерфейсів на мові C#.
C#	C# – об'єктно-орієнтована мова програмування платформи .NET.
CRM	Customer Relationship Management – управління взаємовідносинами з клієнтами.
CSV	Comma-Separated Values – текстовий формат зберігання табличних даних.
DI	Dependency Injection – впровадження залежностей (патерн проектування).
EF Core	Entity Framework Core – ORM-інструмент платформи .NET.
ERP	Enterprise Resource Planning – планування ресурсів підприємства.
HTML	HyperText Markup Language – мова розмітки гіпертексту.
JSON	JavaScript Object Notation – текстовий формат обміну даними.
LINQ	Language Integrated Query – мова запитів, інтегрована у C#.
ORM	Object-Relational Mapper – інструмент відображення об'єктів на таблиці БД.
REST	Representational State Transfer – архітектурний стиль веб-API.
SaaS	Software as a Service – програмне забезпечення як послуга.
SMB	Small and Medium Business – малий та середній бізнес.
SOLID	SOLID – п'ять принципів об'єктно-орієнтованого проектування.
SQLite	SQLite – компактна вбудована реляційна СУБД з відкритим кодом.
UI	User Interface – інтерфейс користувача.
UX	User Experience – досвід взаємодії користувача з продуктом.

ЗМІСТ

ВСТУП

Актуальність теми. В умовах сучасного конкурентного середовища ефективне управління взаємовідносинами з клієнтами набуває стратегічного значення для підприємств малого та середнього бізнесу. CRM-системи (Customer Relationship Management) є ключовим інструментом автоматизації бізнес-процесів, що дозволяє підвищити якість обслуговування клієнтів, оптимізувати обробку замовлень та забезпечити прийняття обґрунтованих управлінських рішень на основі аналітичних даних. Однак більшість існуючих комерційних CRM-рішень орієнтована на великі підприємства, потребує значних фінансових витрат на впровадження та підтримку, а також залежить від хмарних сервісів, що є недоступним або невигідним для малого бізнесу. Це зумовлює необхідність розробки доступного, автономного і функціонально повного модуля CRM-системи, оптимізованого для потреб малих підприємств.

Мета роботи – розробка та реалізація модуля CRM-системи для автоматизації процесів обліку замовлень клієнтів на платформі .NET 8 із застосуванням сучасних технологій веборієнтованої розробки.

Для досягнення мети визначено такі завдання дослідження:

– проаналізувати теоретичні засади CRM, класифікацію та функціональні характеристики існуючих систем, виконати порівняльний аналіз ринкових рішень;

– обґрунтувати вибір технологічного стеку та спроектувати архітектуру модуля CRM-системи;

– розробити модель даних, сервісний шар та інтерфейс користувача системи;

– реалізувати модулі управління клієнтами, товарами, замовленнями, аналітикою, аудитом та аутентифікацією;

– розробити та виконати автоматизоване тестування системи.

Об’єкт дослідження – методи проектування та реалізації модуля CRM-системи на платформі .NET 8 із застосуванням фреймворку Blazor Web App,

об’єктно-реляційного відображувача Entity Framework Core та вбудованої СУБД SQLite.

Предмет дослідження – веборієнтована CRM-система з повним функціональним стеком: управлінням клієнтами, товарами, замовленнями, аналітичним дашбордом, журналом аудиту та системою аутентифікації і авторизації.

Методи дослідження: аналіз наукової літератури та технічної документації у сфері CRM; порівняльний аналіз існуючих програмних рішень; методи об’єктно-орієнтованого проектування і моделювання; методи тестування програмного забезпечення.

Практичне значення отриманих результатів. Розроблений модуль CRM-системи може бути використаний підприємствами малого та середнього бізнесу для автоматизації обліку клієнтів і замовлень без залежності від хмарних сервісів. Простота розгортання (один виконуваний файл і файл бази даних), безкоштовність і відкрита архітектура роблять систему доступною для широкого кола підприємств.

РОЗДІЛ 1
ТЕОРЕТИЧНІ АСПЕКТИ РОЗРОБКИ ВЕБОРІЄНТОВАНОЇ CRM-СИСТЕМИ

1.1 Концепція управління взаємовідносинами з клієнтами: виникнення та еволюція

Концепція управління взаємовідносинами з клієнтами формувалась протягом кількох десятиліть на перетині маркетингових теорій, організаційної психології та інформаційних технологій. Перші паростки цієї ідеї простежуються у роботах Теодора Левіта, який ще у 1960-х роках стверджував, що підприємства не повинні фокусуватись виключно на продукті, а мають розуміти потреби клієнта як відправну точку всієї ділової стратегії. Його стаття «Marketing Myopia» у Harvard Business Review стала одним із перших академічних закликів до клієнтоцентричного мислення і вплинула на формування цілого покоління маркетологів. Хоча Левіт не вводив поняття CRM, його ідеї заклали ідеологічний фундамент для подальшого розвитку цього напряму.

Безпосереднім попередником CRM вважається концепція маркетингу взаємовідносин (Relationship Marketing), сформульована Леонардом Беррі у 1983 році у статті «Relationship Marketing» на конференції Американської маркетингової асоціації. Беррі визначив маркетинг взаємовідносин як залучення, підтримку і – в мультисервісних організаціях – розвиток відносин із клієнтами. Його головна теза полягала в тому, що утримання існуючого клієнта є значно менш витратним, ніж залучення нового, і що підприємства мають систематично вкладати ресурси у побудову довгострокових відносин. Дослідження Беррі показали, що лояльні клієнти не лише забезпечують стабільний потік доходу, а й активно рекомендують компанію своєму оточенню, виконуючи функцію безкоштовного маркетингу.

Шведський дослідник Еверетт Гаммессон розвинув ідеї маркетингу взаємовідносин у більш комплексну систему поглядів. У своїй роботі «Total

Relationship Marketing» він описав 30 типів ринкових взаємовідносин, що виникають у діяльності підприємства, і показав, що лише частина з них є прямими відносинами між продавцем і покупцем. Решта охоплює відносини з партнерами, постачальниками, регуляторами, конкурентами і суспільством у цілому. Гаммессон стверджував, що традиційний маркетинг-мікс (4P: Product, Price, Place, Promotion) є надто обмеженою моделлю, яка не відображає реальної складності ринкових відносин. Його критика стимулювала пошук більш комплексних моделей управління клієнтськими відносинами.

Дослідження Адріана Пейна і Пенні Фроу стали першою спробою систематизувати CRM як стратегічну концепцію з визначеними процесами і показниками ефективності. У своїй фундаментальній статті «A Strategic Framework for Customer Relationship Management», опублікованій у Journal of Marketing у 2005 році, вони запропонували п'ятикомпонентну модель CRM-стратегії. Перший компонент – процес розробки стратегії – охоплює визначення цільових клієнтських сегментів і ціннісних пропозицій для кожного з них. Другий – процес створення цінності – описує механізм трансформації стратегії у конкретні програми лояльності, сервіси та продукти. Третій – процес багатоканальної інтеграції – забезпечує узгоджений досвід клієнта незалежно від каналу взаємодії (офлайн, онлайн, телефон, чат). Четвертий – процес управління інформацією – описує збір, зберігання і аналіз клієнтських даних. П'ятий – процес оцінки ефективності – визначає KPI і механізми зворотного зв'язку для коригування стратегії. Ця модель підкреслює, що CRM є перш за все стратегічною концепцією, а програмне забезпечення – лише інструментом її реалізації.

На початку 2000-х років ринок CRM-програмного забезпечення переживав фазу активного зростання, що супроводжувалась не менш активним розчаруванням. Компанія Forrester Research зафіксувала, що понад 70% великих CRM-впроваджень не досягали заявлених цілей. Деніел Ригбі, Фредерік Рейхгельд і Філ Шефтер у своїй статті «Avoid the Four Perils of CRM» у Harvard Business Review систематизували причини невдач і назвали чотири

ключові помилки: впровадження CRM до розробки стратегії роботи з клієнтами; запуск CRM без реінжинірингу бізнес-процесів; припущення, що більше CRM-технологій означає краще; переслідування нових клієнтів замість роботи з існуючими. Ця публікація мала великий практичний вплив: вона змістила дискусію від технологічного до стратегічного і процесного вимірів CRM і поклала початок сучасному розумінню CRM як комплексного проєкту змін, а не лише IT-проєкту.

Революційним поворотом у розвитку CRM стала поява хмарних рішень на чолі із Salesforce, заснованим Марком Беніоффом у 1999 році. Salesforce запропонував принципово нову модель доступу до корпоративного ПЗ – Software as a Service (SaaS) – за якою підприємство платить абонентську плату за доступ до системи через браузер без необхідності встановлення, обслуговування і оновлення локальних серверів. Цей підхід демократизував CRM: рішення, що раніше вимагали мільйонних інвестицій у серверне обладнання і ліцензії, стали доступними для компаній із кількома десятками співробітників. Водночас хмарна модель породила нові ризики: залежність від постачальника, питання конфіденційності даних і необхідність стабільного інтернет-з'єднання.

Сучасний етап розвитку CRM характеризується декількома паралельними тенденціями. По-перше, відбувається вертикалізація ринку: поряд із загальними платформами зростає кількість галузевих рішень (CRM для нерухомості, медицини, юридичних послуг, освіти), що містять специфічні для галузі процеси, термінологію і звітність. По-друге, активно інтегруються технології штучного інтелекту: предиктивна аналітика відтоку клієнтів, автоматична пріоритизація лідів, генерація персоналізованих пропозицій на основі моделей машинного навчання. По-третє, ринок SMB-CRM залишається недостатньо насиченим якісними рішеннями: більшість доступних продуктів є або надто спрощеними, або надто дорогими для реального малого бізнесу. Саме ця незаповнена ніша визначає актуальність і практичну значущість даного дослідження.

Для цілей даної роботи CRM-система розглядається передусім як програмний засіб автоматизації операційного рівня. Це означає, що основна увага зосереджена на автоматизації конкретних бізнес-процесів: реєстрації клієнтів, ведення товарної номенклатури, обробки замовлень і контролю їх статусів, фіксації взаємодій і формування базової аналітики. Стратегічний і аналітичний виміри CRM враховуються лише в тій мірі, в якій вони безпосередньо реалізовані у розробленому модулі через дашборд і механізм аудиту.

1.2 Класифікація CRM-систем та їх функціональні характеристики

Класифікація CRM-систем є необхідним аналітичним інструментом для прийняття обґрунтованих рішень при виборі або розробці системи. Відсутність чіткої системи понять призводить до порівняння непорівнюваного і хибних очікувань від впровадження. У літературі запропоновано декілька підходів до класифікації, що доповнюють один одного і дозволяють охарактеризувати систему з різних точок зору.

Перша і найбільш широко використовувана ознака класифікації – тип розгортання. Локальні (on-premise) системи встановлюються і виконуються на серверах самого підприємства. Організація є повноцінним власником програмного забезпечення (або має бессрочну ліцензію) і повністю контролює зберігання даних, налаштування безпеки і цикли оновлення. Переваги: незалежність від зовнішніх провайдерів, повний контроль над даними, можливість глибокої інтеграції з внутрішньою IT-інфраструктурою, відсутність постійних щомісячних платежів після придбання ліцензії. Недоліки: необхідність у виділеному серверному обладнанні, кваліфікованому IT-персоналі для обслуговування, самостійному керуванні безпекою і резервним копіюванням. Цей підхід є оптимальним для підприємств із суворими вимогами до конфіденційності даних (фінанси, медицина, право) або нестабільним інтернет-підключенням.

Хмарні (SaaS) системи надаються постачальником через Інтернет за моделлю підписки. Підприємство отримує доступ до системи через браузер і не займається питаннями серверної інфраструктури, резервного копіювання або оновлень – усе це є відповідальністю постачальника. Переваги: нульові капітальні витрати на початку, швидке розгортання (іноді за кілька годин), автоматичні оновлення, масштабованість, доступність з будь-якого пристрою. Недоліки: постійні операційні витрати на підписку, залежність від постачальника і доступності Інтернету, обмежена гнучкість у налаштуванні, питання юрисдикції і безпеки даних (особливо актуально при зберіганні персональних даних клієнтів за вимогами GDPR або українського законодавства про захист персональних даних).

Гібридні системи поєднують переваги обох підходів: критичні дані і основна бізнес-логіка розміщуються локально, тоді як певні функції (наприклад, мобільний доступ, email-маркетинг або аналітика) реалізуються через хмарні сервіси. Такий підхід є компромісом для підприємств, що потребують балансу між контролем над даними і мобільністю. Технічно гібридна архітектура є найбільш складною у реалізації та обслуговуванні.

Друга ознака класифікації – функціональне спрямування. Операційні CRM-системи автоматизують безпосередні взаємодії з клієнтами на всіх стадіях їх життєвого циклу. Підсистема автоматизації продажів (Sales Force Automation, SFA) охоплює управління лідами і можливостями, ведення воронки продажів, планування активності торгових представників, управління замовленнями і контрактами. Підсистема автоматизації маркетингу управляє кампаніями, сегментацією аудиторії, масовими розсилками і відстеженням відгуку. Підсистема обслуговування клієнтів (Customer Service) управляє зверненнями, скаргами, SLA і базою знань. Операційний тип є домінуючим серед SMB-рішень і є основним фокусом даної розробки.

Аналітичні CRM-системи зосереджені на видобуванні знань із накопичених операційних даних. Вони використовують методи data mining, OLAP-аналізу, статистичного моделювання і все частіше – машинного

навчання для сегментації клієнтів за RFM-моделлю (Recency, Frequency, Monetary), прогнозування відтоку, оцінки потенційної цінності клієнта (Customer Lifetime Value, CLV) і персоналізації пропозицій. Аналітичні CRM, як правило, є надбудовою над операційними системами і потребують достатнього обсягу накопичених даних для отримання статистично значущих результатів.

Колаборативні CRM-системи фокусуються на координації між різними підрозділами підприємства, що мають контакт із клієнтом: відділом продажів, маркетингом, технічною підтримкою і фінансами. Їх мета – забезпечити єдиний, послідовний образ клієнта для всіх підрозділів, незалежно від того, через який канал відбувалась остання взаємодія. Колаборативні CRM часто реалізуються як модулі у складі більших корпоративних платформ (ERP) або реалізують інтеграцію між спеціалізованими рішеннями через API.

Третя ознака класифікації – масштаб і цільовий сегмент. Enterprise CRM (корпоративні рішення) розраховані на великі організації з тисячами користувачів, складними ієрархіями, мультиорганізаційними структурами та інтеграцією з ERP, BI і іншими корпоративними системами. Функціональна повнота досягається ціною складності налаштування і необхідністю у кваліфікованих адміністраторах. SMB CRM орієнтовані на малий і середній бізнес: спрощені процеси, швидке розгортання, нижча вартість і менший поріг входу. Вертикальні CRM спеціалізовані для конкретних галузей і містять вбудовані галузеві шаблони, специфічну термінологію, типові звіти і інтеграції з галузевими системами.

Серед ключових функціональних модулів, що є спільними для більшості операційних CRM незалежно від масштабу, слід виокремити такі. Управління контактами і клієнтською базою є ядром будь-якої CRM-системи: структуроване зберігання даних про клієнтів (фізичних та юридичних осіб), їх контактну інформацію, категоризацію, пошук і фільтрацію. Управління замовленнями і угодами забезпечує супровід замовлення від моменту виникнення до завершення або скасування, включно з розрахунком вартості і

контролем статусів. Управління товарами і послугами підтримує номенклатуру, ціноутворення і облік залишків. Управління взаємодіями фіксує всі точки контакту із клієнтом, формуючи хронологію відносин. Аналітика і звітність надають керівникам агреговані показники для прийняття рішень. Журнал аудиту забезпечує прозорість і контроль за діями користувачів у системі.

Серед нефункціональних характеристик, що визначають якість CRM-системи для цільового сегменту, особливого значення набувають: простота розгортання (мінімальна кількість кроків від встановлення до першого використання), низький поріг навчання (зрозумілий інтерфейс, що не потребує тривалого навчання персоналу), надійність і захист даних (механізм аутентифікації, ролеве розмежування доступу, шифрування паролів), адаптивний інтерфейс (коректне відображення на пристроях з різними розмірами екрана), і, що є принциповим для даної розробки, відсутність залежності від хмарної інфраструктури третіх сторін.

1.3 Порівняльний аналіз існуючих CRM-рішень

Для обґрунтування доцільності власної розробки виконано детальний порівняльний аналіз п'яти найбільш поширених CRM-платформ, що теоретично могли б задовольнити потреби цільового сегменту – малого або середнього підприємства, що займається реалізацією товарів через систему клієнтських замовлень. Критерії порівняння сформовані на основі вимог, визначених у другому розділі роботи: функціональна повнота (наявність управління клієнтами, товарами, замовленнями, аналітикою та аудитом), складність впровадження і навчання, вартість, можливість локального розгортання та ступінь адаптованості до специфічних бізнес-процесів без розробки.

Salesforce є абсолютним лідером глобального ринку CRM із часткою понад 20% і виторгом понад 30 мільярдів доларів у 2023 фінансовому році. Платформа пропонує надзвичайно широку екосистему продуктів: Sales Cloud

для управління продажами і угодами, Service Cloud для підтримки клієнтів і управління зверненнями, Marketing Cloud для масштабних маркетингових кампаній, Commerce Cloud для e-commerce, Analytics Cloud для бізнес-аналітики і Einstein AI для функцій штучного інтелекту. Технічно Salesforce є зрілою, надзвичайно гнучкою платформою: власна мова програмування Apex дозволяє реалізувати практично будь-яку бізнес-логіку, декларативний конструктор процесів Flow Builder – автоматизувати складні робочі процеси без коду, а AppExchange містить тисячі сторонніх додатків і інтеграцій. Проте для малого бізнесу ця платформа є надто складною і дорогою: мінімальний план Starter Suite коштує від 25 доларів на користувача на місяць, але для реального використання з функціями управління замовленнями необхідний план Professional від 80 доларів. Повноцінне впровадження потребує залучення сертифікованих Salesforce-партнерів (Salesforce Certified Administrator) і може тривати від кількох місяців до кількох років. Локальне розгортання недоступне – Salesforce є виключно хмарним рішенням.

Microsoft Dynamics 365 – корпоративна CRM/ERP-платформа, що є частиною ширшої Microsoft Business Applications екосистеми. Ключові модулі: Dynamics 365 Sales (управління продажами і відносинами), Customer Service (обслуговування клієнтів), Marketing (маркетингові кампанії, інтегрований з Dynamics 365 Customer Insights), Field Service (управління виїзним обслуговуванням). Унікальною перевагою є глибока інтеграція з Microsoft 365 (Outlook, Teams, Excel, SharePoint), Power Platform (Power BI, Power Automate, Power Apps) і Azure Active Directory. Для організацій, що вже використовують екосистему Microsoft, ця інтеграція є значною перевагою: дані з CRM автоматично доступні в Excel, активності можна планувати через Teams, а Power BI дозволяє будувати розширені аналітичні звіти. Система доступна як у хмарній, так і в on-premise версії (Dynamics 365 On-Premises), що є рідкістю серед сучасних CRM. Водночас мінімальна вартість ліцензії на Dynamics 365 Sales Essentials становить від 65 доларів на користувача на місяць, а складна

структура ліцензування і необхідність технічної компетенції роблять систему практично недоступною для малого бізнесу без значних IT-ресурсів.

HubSpot CRM із часткою ринку близько 6% є найбільш відомим CRM-рішенням у сегменті малого і середнього бізнесу. Система позиціонується на основі стратегії «freemium»: базовий функціонал управління контактами, угодами і воронкою продажів надається безплатно без обмежень по кількості користувачів і записів. Це є суттєвою конкурентною перевагою і забезпечило HubSpot значне проникнення на ринок SMB. Інтерфейс HubSpot є одним із найбільш зручних серед розглянутих рішень: логічна навігація, детальні підказки і вбудовані навчальні матеріали (HubSpot Academy) знижують поріг входу до мінімуму. Проте реальне функціональне обмеження безкоштовного плану є суттєвим: відсутні налаштовувані звіти, автоматизація маркетингу, синхронізація пошти з послідовностями, розширений API і управління товарним каталогом. Розширені функції доступні через платні плани (Sales Hub Starter від 20 доларів, Sales Hub Professional від 500 доларів на місяць), що робить загальну вартість порівнянною з конкурентами при масштабуванні. Хмарна природа HubSpot означає повну залежність від постачальника і відсутність варіанту локального розгортання.

Pipedrive – CRM-система з виразною спеціалізацією на управлінні воронкою продажів. Заснована у 2010 році командою з естонських та американських підприємців, система зосереджена на одній ключовій ідеї: всі угоди у продажах проходять через послідовність визначених стадій, і інструмент CRM повинен насамперед допомогти відстежувати їх прогрес. Головний інтерфейс Pipedrive – це канбан-дошка угод із колонками для кожної стадії воронки, де картки угод переміщуються простим drag-and-drop. Ця візуальна модель є надзвичайно інтуїтивною для менеджерів із продажів і є основною причиною популярності системи серед B2B-компаній із тривалими циклами продажів. Проте для цілей даної роботи Pipedrive має критичні обмеження: відсутній повноцінний модуль управління товарами зі складським обліком, обмежена функціональність журналу аудиту, немає підтримки

статусів замовлення із машиною станів, і взагалі система орієнтована на управління угодами на стадії переговорів, а не на обробку підтверджених замовлень. Мінімальна вартість – від 12 доларів на користувача на місяць, максимальна з усіма функціями – від 80 доларів.

Bitrix24 є найпопулярнішою CRM-платформою на ринках України, Росії та країн СНД, що пропонує унікальне поєднання CRM, корпоративного порталу, управління задачами і проектами, IP-телефонії, відеоконференцій і документообігу в одному продукті. Платформа доступна у хмарному варіанті (з безкоштовним планом для до 5 користувачів із 5 ГБ сховища) і у версії для локального розгортання (Bitrix24 On-Premise), що виділяє її серед конкурентів. CRM-функціональність Bitrix24 охоплює ліди, угоди, контакти, компанії, каталог товарів, рахунки і воронку продажів. Безкоштовний план є функціонально обмеженим: відсутній ряд CRM-функцій (зокрема, налаштовувані поля і процеси), обмежена автоматизація і звітність. Основним недоліком для цільової аудиторії є перевантаженість інтерфейсу: присутність сотень функцій із різних доменів (CRM, задачі, документи, HR, телефонія) робить навчання персоналу тривалим і непростим. Малий бізнес, якому потрібна лише CRM, отримує величезний функціональний баласт, що ускладнює роботу.

Підсумовуючи порівняльний аналіз, можна констатувати: жодне з розглянутих рішень не забезпечує оптимального балансу між функціональністю, простотою, вартістю і незалежністю від зовнішніх постачальників для цільового сегменту. Salesforce і Dynamics 365 надмірно дорогі і складні. HubSpot безкоштовний у базовій версії, але функціонально обмежений і хмарний. Pipedrive не підходить для обліку замовлень. Bitrix24 є найближчим до прийнятного, але перевантажений функціоналом і складний в освоєнні. Цей аналіз підтверджує доцільність розробки спеціалізованого модуля із точно визначеним функціоналом, простим розгортанням і відсутністю залежності від хмарного постачальника.

1.4 Огляд платформи .NET 8 та фреймворку Blazor Web App

Платформа .NET пройшла довгий шлях від власницького .NET Framework (1.0 у 2002 році), обмеженого середовищем Windows, до сучасної кросплатформенної відкритої екосистеми .NET 8. Ключовим поворотним моментом стало анонсування .NET Core 1.0 у 2016 році – першої версії, що підтримувала Linux і macOS і поширювалась з відкритим вихідним кодом на GitHub. Подальший розвиток відбувався стрімко: .NET 5 (2020) об'єднав .NET Core і Mono в єдину платформу, усунувши плутанину між різними варіантами .NET. Версії .NET 6 і .NET 8 є стратегічними LTS-версіями (Long-Term Support) із гарантованою підтримкою протягом трьох років після виходу.

.NET 8, випущений у листопаді 2023 року, привніс ряд суттєвих покращень продуктивності та нових можливостей. Компілятор RyuJIT отримав удосконалену оптимізацію векторизованих операцій і інструкцій AVX-512 для сучасних процесорів. Режим Native AOT (Ahead-of-Time Compilation) дозволяє компілювати застосунок у нативний виконуваний файл без потреби у .NET Runtime, що зменшує час запуску і обсяг розгортання до мінімуму – актуально для мікросервісів і serverless-сценаріїв. Збирач сміття отримав новий режим DATAS (Dynamic Adaptation to Application Sizes), що динамічно адаптує розміри heap-областей до поведінки застосунку і знижує пікове споживання пам'яті. Мова C# 12 розширилась первинними конструкторами (primary constructors) для класів і структур, що суттєво скорочує шаблонний код; виразами колекцій (collection expressions) для зручнішої ініціалізації списків і масивів; псевдонімами для будь-яких типів через using із більш гнучким синтаксисом.

ASP.NET Core є основним веб-фреймворком платформи .NET і забезпечує серверну основу для розробленого модуля. Його архітектура базується на конвеєрі middleware – послідовності компонентів, що обробляють HTTP-запит і передають його далі або формують відповідь. Порядок реєстрації middleware у Program.cs є критичним: UseAuthentication має передувати

UseAuthorization, UseRouting – UseEndpoints, a UseExceptionHandler – всім іншим компонентам. Вбудована підтримка Dependency Injection є стрижнем усього фреймворку: сервіси реєструються у IServiceCollection з одним із трьох часів існування (Singleton, Scoped, Transient) і автоматично вводяться у конструктори, що споживають їх. Мінімальний API (Minimal API), представлений у .NET 6, дозволяє визначати ендпоінти безпосередньо в Program.cs без класів контролерів, що суттєво скорочує обсяг шаблонного коду для простих API.

Blazor – фреймворк для розробки інтерактивних вебінтерфейсів на C# замість JavaScript – є одним із найбільш інноваційних проєктів платформи .NET. Перша версія Blazor WebAssembly вийшла у 2020 році і дозволяла виконувати C#-код у браузері завдяки WebAssembly (WASM) – відкритому стандарту виконання бінарного коду у браузерному середовищі. Blazor Server виконував компоненти на сервері і синхронізував UI через SignalR-з'єднання у реальному часі. Обидва режими мали власні недоліки: WebAssembly – великий розмір завантаження (.NET Runtime у браузері); Server – залежність від постійного серверного з'єднання.

Blazor Web App, представлений у .NET 8, принципово вирішує обмеження попередніх підходів через уніфіковану модель рендерингу з чотирма режимами для кожного компонента. Режим Static SSR (Static Server-Side Rendering) рендерить HTML на сервері без підтримки інтерактивності – ідеальний для сторінок, що не потребують реакції на події, забезпечує найшвидший початковий рендеринг і відмінну індексацію пошуковиками. Режим Interactive Server виконує компоненти на сервері і оновлює UI через SignalR – мінімальний розмір завантаження, миттєва інтерактивність. Режим Interactive WebAssembly виконує компоненти у браузері – після завантаження немає звернень до сервера, ідеально для офлайн-сценаріїв. Режим Auto починає з Interactive Server і непомітно перемикається на WebAssembly після завантаження – найкращий баланс між швидким стартом і автономністю браузера.

1.5 Огляд Entity Framework Core, SQLite та бібліотек тестування

Entity Framework Core є провідним ORM-інструментом (Object-Relational Mapper) платформи .NET, що забезпечує взаємодію між об'єктно-орієнтованим кодом застосунку і реляційними базами даних. Ключова ідея ORM полягає у відображенні таблиць бази даних на класи C# і рядків таблиць на об'єкти, що дозволяє розробнику працювати з даними у звичних об'єктно-орієнтованих термінах без написання SQL-запитів вручну. EF Core 8 підтримує широкий спектр СУБД через провайдери: Microsoft SQL Server, PostgreSQL, MySQL, SQLite, Oracle та інші. Можливість зміни провайдера з мінімальними змінами у коді є значною перевагою при масштабуванні системи.

Підхід Code First, обраний для даної розробки, дозволяє описати схему бази даних через класи C# (так звані POCO – Plain Old CLR Objects) і генерувати відповідні SQL-таблиці через систему міграцій або метод EnsureCreated. Порівняно із підходом Database First (генерація класів із існуючої бази), Code First забезпечує версіонування схеми разом із кодом застосунку, що є природнім при розробці через Git. EF Core 8 привніс суттєві покращення: методи ExecuteUpdate і ExecuteDelete для масових операцій без завантаження сутностей у пам'ять (UPDATE ... WHERE або DELETE ... WHERE у SQL), розширену підтримку JSON-колонок для зберігання і запиту складних об'єктів, покращену підтримку ієрархічних структур даних через атрибут [HierarchyId].

SQLite є компактною вбудованою реляційною СУБД, що зберігає всю базу даних у єдиному файлі на диску. На відміну від клієнт-серверних СУБД (PostgreSQL, MySQL, SQL Server), SQLite не вимагає окремого серверного процесу – бібліотека завантажується безпосередньо у процес застосунку. Це робить SQLite ідеальним вибором для розгортання без складної інфраструктури. SQLite підтримує транзакції ACID, більшість можливостей SQL:1999 і є одним із найбільш тестованих програмних проєктів у світі – автори SQLite стверджують, що тести займають близько 787 разів більше коду,

ніж сама бібліотека. Продуктивність SQLite на читання є порівнянною з серверними СУБД для баз до кількох гігабайт; основним обмеженням є блокування бази під час запису (WAL-режим частково вирішує цю проблему), що обмежує масштабованість при інтенсивному паралельному записі.

Специфічним обмеженням SQLite у контексті Entity Framework Core є часткова підтримка певних LINQ-операцій, що транслюються у складні SQL-запити. Зокрема, GroupBy із агрегацією decimal-значень у певних сценаріях не може бути повністю транслований у SQL і призводить до помилки під час виконання. Обхідним рішенням є виконання ToListAsync для завантаження даних у пам'ять перед виконанням GroupBy на стороні C#. Хоча це менш ефективно при великих обсягах даних, для сценаріїв малого бізнесу з кількома тисячами записів цей підхід є цілком прийнятним.

xUnit є сучасною платформою для unit-тестування в екосистемі .NET, розробленою як спадкоємець NUnit і MSTest із урахуванням досвіду їх використання. Ключові відмінності xUnit від конкурентів: тестові класи не позначаються атрибутом [TestClass] – будь-який публічний клас є потенційним тестовим контейнером; ізоляція тестів досягається через конструктор і IDisposable замість [SetUp]/[TearDown], що заохочує чистий об'єктно-орієнтований дизайн; атрибут [Theory] разом із [InlineData] або [MemberData] дозволяє елегантно параметризувати тести без дублювання коду; конструктор виконується перед кожним тестом, що гарантує повну ізоляцію. Microsoft обрала xUnit як офіційну платформу тестування для власних проєктів .NET, що підкреслює її зрілість і підтримку.

EF Core InMemory Provider є спеціалізованим провайдером для тестів, що замінює реальну базу даних in-memory еквівалентом. Це дозволяє тестувати сервіси, що використовують DbContext, без звернення до файлу SQLite або іншої СУБД, забезпечуючи повну ізоляцію і детерміновану поведінку тестів. Для кожного тесту рекомендується створювати окремий контекст із унікальним ім'ям бази (через Guid.NewGuid().ToString()), щоб уникнути перетину даних між тестами. Важливим застереженням є те, що

InMemory Provider не перевіряє деякі обмеження, що існують у реальних СУБД (наприклад, унікальні індекси у певних версіях), тому критичні обмеження слід перевіряти через інтеграційні тести із реальним провайдером.

1.6 Обґрунтування вибору напрямку досліджень

На підставі проведеного аналізу теоретичних засад CRM, порівняння ринкових рішень і огляду технологій сформовано такий напрямок дослідження: проектування і розробка спеціалізованого модуля CRM-системи операційного класу для автоматизації обліку замовлень клієнтів із використанням стеку .NET 8 / Blazor Web App / ASP.NET Core / EF Core 8 / SQLite і архітектурного підходу service layer.

Вибір Blazor Web App як фреймворку для побудови інтерфейсу обґрунтований кількома суттєвими перевагами над альтернативними підходами. По-перше, єдина мова C# для серверної і клієнтської частини усуває роздвоєність стеку: розробник не мусить перемикатись між C# і JavaScript/TypeScript, управляти двома незалежними системами збірки і синхронізувати типи даних через API. По-друге, компонентна модель Blazor є природньою для розробників, що знайомі з React або Angular, і забезпечує перевикористання UI-компонентів. По-третє, наявність вбудованої системи форм і валідації (EditForm, DataAnnotationsValidator) спрощує реалізацію складних форм із підтримкою реального часу. По-четверте, гнучка модель рендерингу Blazor Web App у .NET 8 дозволяє обрати оптимальний режим для кожного компонента окремо, що є унікальною можливістю.

Вибір service layer як архітектурного підходу обґрунтований вимогами до тестованості і підтримуваності системи. Визначення сервісів через інтерфейси дозволяє: тестувати бізнес-логіку у повній ізоляції від інфраструктури, підмінюючи контекст бази даних InMemory-провайдером; замінювати реалізацію сервісу (наприклад, перейти від SQLite до PostgreSQL) без зміни компонентів, що використовують сервіс; додавати нові можливості (кешування, логування, авторизаційні перевірки) шляхом написання

декоратора над існуючим сервісом без модифікації оригінального коду. Ці переваги відповідають принципу відкритості/закритості (Open/Closed Principle) та принципу інверсії залежностей (Dependency Inversion Principle) зі складу SOLID.

Вибір SQLite як СУБД продиктований першорядною вимогою простоти розгортання для цільового сегменту. Підприємство малого бізнесу, як правило, не має виділеного IT-персоналу і не здатне самостійно налаштувати PostgreSQL або SQL Server. Модель розгортання «один виконуваний файл + один файл бази даних» є максимально простою і доступною навіть без технічних знань. При цьому абстракція EF Core гарантує, що при зростанні підприємства і необхідності переходу на потужнішу СУБД зміни у коді зведуться до одного рядка в Program.cs.

РОЗДІЛ 2
ПРОЕКТУВАННЯ МОДУЛЯ CRM-СИСТЕМИ

2.1 Визначення функціональних та нефункціональних вимог

Фаза визначення вимог є найбільш критичною у процесі розробки програмного забезпечення. Статистика проєктних невдач, зібрана Standish Group у щорічному Chaos Report, незмінно показує, що неповні або суперечливі вимоги є однією з трьох найпоширеніших причин провалу проєктів поряд із браком підтримки керівництва і недостатньою залученістю користувачів. Формалізація вимог відповідно до стандарту IEEE 830 передбачає поділ на функціональні вимоги (що система повинна робити) і нефункціональні вимоги (характеристики якості: продуктивність, безпека, зручність і надійність). Для даного модуля вимоги сформовані на основі аналізу типових бізнес-процесів малого підприємства, що реалізує товари через систему клієнтських замовлень.

Перша група функціональних вимог охоплює управління клієнтами. Система повинна забезпечувати реєстрацію нового клієнта із зазначенням таких атрибутів: повне ім'я (обов'язкове поле), адреса електронної пошти (обов'язкове, унікальне), телефонний номер, назва компанії і довільні примітки. Пошук клієнтів має здійснюватись одночасно за чотирма полями (ім'я, email, телефон, компанія) з підтримкою часткового збігу (пошук підрядка у значенні поля). Система повинна відображати деталі профілю клієнта і повну хронологію його замовлень. Редагування профілю клієнта має здійснюватись через форму з повним набором полів. Видалення клієнта повинно бути заблоковане за наявності пов'язаних замовлень для захисту від втрати фінансових записів.

Друга група охоплює управління товарами. Система повинна підтримувати ведення каталогу товарів із такими атрибутами: артикул (унікальний ідентифікатор товару у системі підприємства), назва, опис, ціна без знижки, поточний залишок на складі, ознака активності і поріг низького

залишку. Пошук і фільтрація товарів мають здійснюватись за назвою і артикулом. Система повинна автоматично виявляти товари, залишок яких досяг або опустився нижче порогового значення, і відображати їх у відповідному розділі дашборду. Деактивація товару повинна приховувати його у формах створення замовлення, але зберігати у базі для цілісності фінансових записів. CRUD-операції (Create, Read, Update, Delete) повинні підтримуватись для всіх атрибутів товару.

Третя група охоплює управління замовленнями і є найбільш складною з функціональної точки зору. Система повинна забезпечувати створення замовлення з прив'язкою до існуючого клієнта з каталогу. Замовлення повинне містити довільну кількість позицій, кожна з яких визначає конкретний товар і кількість одиниць. Ціна товару у позиції повинна фіксуватись на момент створення замовлення і не змінюватись при подальших змінах ціни у каталозі. Система повинна автоматично розраховувати підсумкову суму як добуток кількості і ціни по кожній позиції з подальшим застосуванням відсоткової знижки. Унікальний номер замовлення повинен генеруватись автоматично у форматі ORD-YYYY-NNN. Управління статусами замовлення повинне реалізовувати визначений цикл із блокуванням недопустимих переходів. Система повинна забезпечувати перегляд деталей замовлення і генерацію рахунку-фактури для друку.

Четверта група охоплює аналітику і звітність. Головна сторінка системи повинна відображати дашборд із ключовими показниками: загальний дохід від усіх незасових замовлень, кількість замовлень у кожному статусі, кількість активних клієнтів і кількість товарів із критично низьким залишком. Дашборд повинен містити графік динаміки доходу по місяцях поточного року і таблиці топ-клієнтів та топ-товарів. Система повинна надавати розділи зі звітами по замовленнях, клієнтах і товарах із можливістю фільтрації за часовим діапазоном. Дані звітів повинні бути доступні для вивантаження у форматі CSV.

П'ята група охоплює взаємодії з клієнтами і аудит. Система повинна дозволяти фіксацію кожної взаємодії із клієнтом із зазначенням типу (телефонний дзвінок, електронний лист, особиста зустріч, запит клієнта, скарга клієнта), дати, опису і результату взаємодії, а також планованої дати наступного контакту. Хронологія взаємодій повинна бути доступною у профілі клієнта і на окремій сторінці. Журнал аудиту повинен автоматично фіксувати всі значущі операції (створення, зміна, видалення записів, зміна статусів замовлень) із зазначенням імені користувача і часової мітки. Перегляд журналу повинен підтримувати фільтрацію за типом операції, назвою сутності і часовим діапазоном.

Шоста група охоплює аутентифікацію і авторизацію. Система повинна забезпечувати вхід за логіном і паролем із перевіркою облікових даних і формуванням автентифікаційної сесії через cookie. Підтримка трьох ролей (Admin, Manager, Viewer) повинна забезпечувати відповідне розмежування доступу до функцій і даних. Автоматичний вихід зі скасуванням сесії повинен відбуватись через 8 годин без активності (з увімкненим sliding expiration). Усі сторінки системи повинні бути захищені від доступу без попередньої аутентифікації.

Нефункціональні вимоги охоплюють такі аспекти. Зручність використання: адаптивний веб-інтерфейс, коректно відображається на пристроях із шириною від 320px до 1920px без горизонтального прокручування; підтримка темної теми зі збереженням вибору між сесіями; toast-сповіщення про результат кожної операції з автоматичним зникненням через 3-5 секунд; підтвердження перед деструктивними операціями (видалення) через модальне вікно. Продуктивність: час відгуку для операцій читання не більше 2 секунд при базі до 10 000 записів на типовому обладнанні. Надійність: коректна обробка помилок на всіх рівнях із відображенням зрозумілих повідомлень; логування помилок у файл із повним трасуванням стеку. Розгортання: запуск без окремого серверного процесу СУБД; автоматичне створення бази даних і заповнення початковими даними при першому запуску.

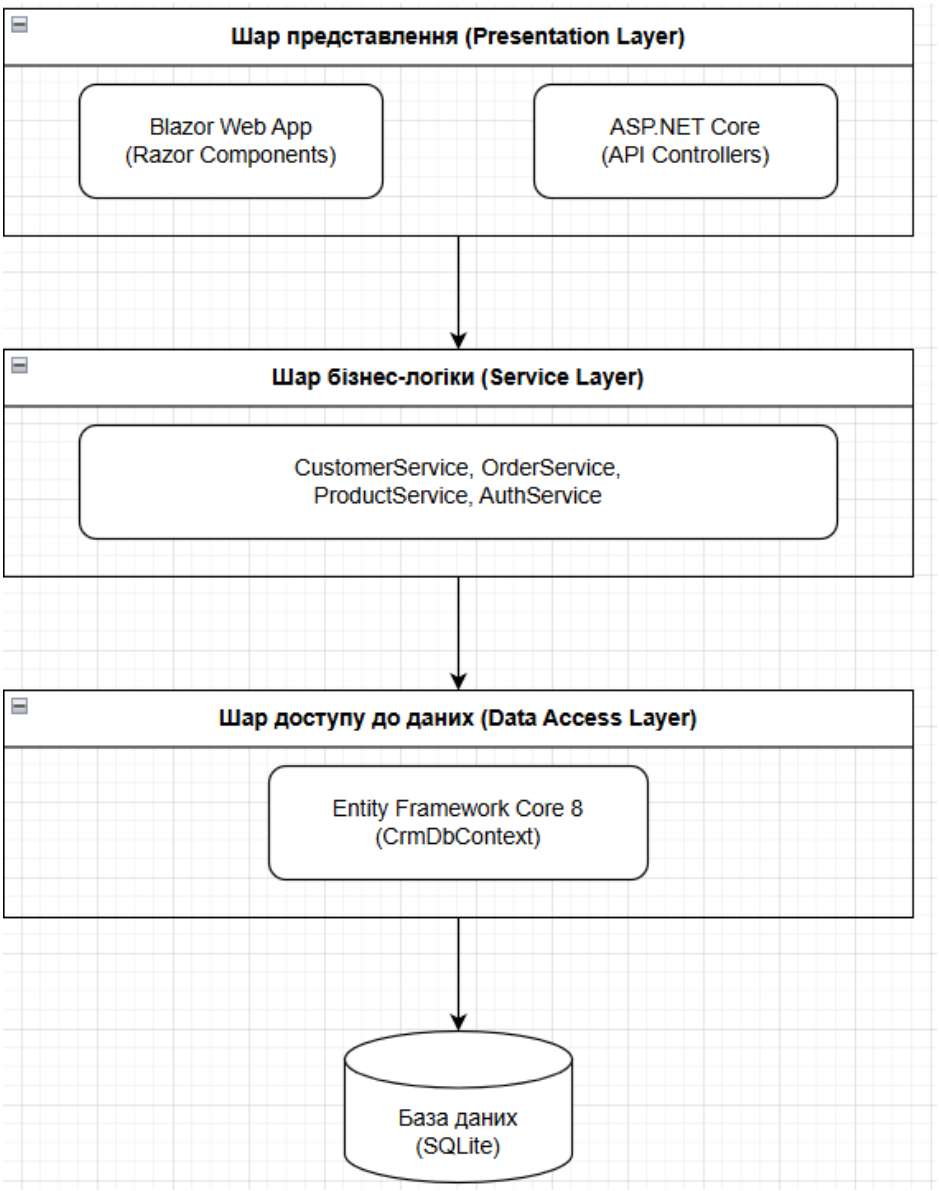

Рисунок 2.1. Загальна схема програми

2.2 Архітектурне проектування: вибір патерну і загальна структура

Вибір архітектурного патерну є одним із найбільш важливих рішень у проектуванні програмної системи, оскільки він визначає структуру коду, можливості тестування, гнучкість при змінах вимог і складність навчання нових розробників. Для модуля CRM-системи обрано патерн N-tier архітектури (багатошарової архітектури) з додатковим підрозділом шару бізнес-логіки на service layer із визначеними інтерфейсами. Цей вибір обґрунтований кількома міркуваннями.

По-перше, N-tier є найбільш поширеним і документованим патерном у екосистемі .NET. Документація Microsoft, навчальні курси і більшість книг із архітектури .NET-застосунків використовують саме цей підхід як базовий. Це знижує поріг входу для нових розробників, що можуть долучитись до проєкту, і забезпечує широку доступність довідкових матеріалів. По-друге, N-tier є природнім доповненням до Dependency Injection, що є вбудованим механізмом ASP.NET Core: кожен шар реєструє свої залежності у контейнері DI і отримує залежності від суміжного шару через конструктор, що забезпечує слабку зв'язаність. По-третє, розмежування між шарами представлення, бізнес-логіки і доступу до даних дозволяє тестувати кожний шар незалежно: компоненти – через Blazor Testing, сервіси – через xUnit із InMemory DB, контекст – через інтеграційні тести.

Архітектура модуля складається з чотирьох логічних шарів. Перший – шар представлення (Presentation Layer) – реалізований через Razor-компоненти Blazor і API-контролери ASP.NET Core. Razor-компоненти відповідають за рендеринг HTML і обробку взаємодій користувача; вони не містять бізнес-логіки і не звертаються безпосередньо до бази даних – вся логіка делегується сервісам. API-контролери надають HTTP-інтерфейс для зовнішніх взаємодій (наприклад, завантаження CSV-файлів).

Другий – шар бізнес-логіки (Business Logic Layer) – реалізований через сервіси у папці Services. Кожен сервіс визначається парою «інтерфейс / реалізація»: інтерфейс описує контракт (що сервіс повинен робити), реалізація описує деталі (як саме). Сервіси зареєстровані у контейнері DI як Scoped – один екземпляр на HTTP-запит. Бізнес-правила (валідація, машина станів, генерація номерів) зосереджені виключно у сервісах і не дублюються у компонентах або контролерах. Третій – шар доступу до даних (Data Access Layer) – реалізований через CrmDbContext, що наслідує DbContext з EF Core. Контекст є єдиною точкою доступу до бази SQLite для всіх сервісів. Четвертий – інфраструктурний шар – охоплює конфігурацію (appsettings.json), логування (Serilog), аутентифікацію (Cookie middleware) і Swagger.

Проєкт організований фізично у трьох .csproj-файлах. CRM-система.csproj є серверним проєктом: він містить все серверне виконання – бізнес-логіку, контекст бази даних, API-контролери, конфігурацію і серверні Razor-компоненти. Саме він є Startup Project і запускається командою dotnet run. CRM-система.Client.csproj є клієнтським WebAssembly-проєктом: він містить компоненти, що виконуються у браузері. Компілятор забороняє використовувати у ньому серверні API (файловий доступ, EF Core і т.д.) – ця перевірка виконується на рівні компіляції, що запобігає помилковому включенню серверного коду у клієнтський бандл. CRM-система.Tests.csproj є тестовим проєктом, що посилається на серверний через ProjectReference.

Взаємодія між шарами організована через інтерфейси: компоненти і контролери залежать лише від інтерфейсів сервісів, а не від їх конкретних реалізацій. Це є ключовим принципом Dependency Inversion і забезпечує можливість: підміни реалізацій у тестах (TestCustomerService замість CustomerService); впровадження аспектів (логування, кешування) через патерн Decorator; незалежного розвитку компонентів і сервісів за умови дотримання контракту інтерфейсу. Всі залежності є явними і перераховані у конструкторі, що відповідає принципу явної залежності (Explicit Dependencies Principle).

2.3 Проектування моделі даних: сутності, атрибути і зв'язки

Проектування моделі даних є процесом трансформації концептуальної моделі предметної галузі у формалізовану схему, придатну для реалізації в реляційній базі даних. Для розробленого модуля використовується підхід Code First: модель описується через класи C# з атрибутами і Fluent API-конфігурацією, а EF Core генерує відповідну схему бази даних. Модель складається із семи сутностей, що охоплюють всі аспекти предметної галузі.

Сутність Customer (Клієнт) є одною з центральних у системі. Клас Customer містить такі властивості: Id (int, первинний ключ, автоінкремент – генерується базою даних); FullName (string, обов'язкове, максимальна довжина 200 символів, декорується атрибутом [Required] і [MaxLength(200)]); Email

(string, обов'язкове, максимальна довжина 150 символів, з унікальним індексом у базі); Phone (string, необов'язкове, максимальна довжина 50 символів); Company (string, необов'язкове, максимальна довжина 200 символів); Notes (string, необов'язкове, без обмеження довжини – зберігається як TEXT у SQLite); CreatedAt (DateTime, заповнюється автоматично під час створення поточним часом UTC). Навігаційні властивості: Orders (ICollection<Order>) і Interactions (ICollection<CustomerInteraction>). Унікальний індекс на Email реалізований через Fluent API: builder.Entity<Customer>().HasIndex(c => c.Email).IsUnique().

Сутність Product (Товар) містить: Id (int, PK); SKU (string, артикул, унікальний, максимальна довжина 50 символів); Name (string, обов'язкове, максимальна довжина 200 символів); Description (string, TEXT); Price (decimal, ціна без знижки, з обмеженням precision і scale через Fluent API для коректного зберігання у SQLite); StockQuantity (int, поточний залишок, значення за замовчуванням 0); IsActive (bool, ознака активності, значення за замовчуванням true); LowStockThreshold (int, поріг сигналізації, значення за замовчуванням 5); CreatedAt (DateTime). Навігаційна властивість: OrderItems (ICollection<OrderItem>). При відображенні у форм деактивований (IsActive = false) товар не відображається у випадаючих списках вибору, але зберігається у базі разом із пов'язаними позиціями замовлень.

Сутність Order (Замовлення) є найбільш складною і центральною. Вона містить: Id (int, PK); OrderNumber (string, унікальний, максимальна довжина 20 символів, наприклад ORD-2024-007); CustomerId (int, зовнішній ключ до Customer – обов'язковий); Customer (навігаційна властивість, Customer, not null); OrderDate (DateTime, дата і час оформлення, за замовчуванням DateTime.UtcNow); Status (OrderStatus, перелічення із значеннями New, Processing, Shipped, Completed, Cancelled); TotalAmount (decimal, підсумкова сума після знижки, обчислюється і зберігається у базі при кожному оновленні); Discount (decimal, відсоткова знижка від 0.00 до 100.00); Notes (string, коментар

до замовлення). Навігаційна властивість: OrderItems (ICollection<OrderItem>). Унікальний індекс на OrderNumber гарантує відсутність дублікатів.

Сутність OrderItem (Позиція замовлення) реалізує зв'язок між замовленням і товаром із збереженням контексту угоди. Вона містить: Id (int, PK); OrderId (int, зовнішній ключ до Order); Order (навігаційна властивість); ProductId (int, зовнішній ключ до Product); Product (навігаційна властивість); Quantity (int, кількість одиниць, повинна бути більше нуля); UnitPrice (decimal, ціна одиниці на момент оформлення замовлення). Принципово важливим є те, що UnitPrice зберігається у позиції замовлення, а не береться з поточної ціни Product.Price. Це забезпечує незмінність фінансових записів: навіть якщо ціна товару змінилась після оформлення замовлення, підсумкова сума вже закритого замовлення залишиться незмінною. Це є обов'язковою вимогою фінансового обліку.

Сутність AuditLog (Запис журналу аудиту) є принципово відірваною від інших сутностей у плані зовнішніх ключів. Це навмисне архітектурне рішення: якщо б AuditLog мав зовнішній ключ до Customer, видалення клієнта (навіть через каскадне видалення) знищило б пов'язані записи аудиту і порушило б цілісність журналу. Замість цього журнал зберігає дані денормалізовано: Id (int, PK); EntityName (string, назва сутності, наприклад «Customer», «Order», «Product»); EntityId (int, ідентифікатор запису на момент операції); Action (string, тип операції: «Create», «Update», «Delete», «StatusChange»); Details (string, JSON або текстовий опис деталей зміни – наприклад, старе і нове значення ключових полів); UserName (string, ім'я користувача, що виконав операцію); Timestamp (DateTime, час операції UTC). Завдяки такому підходу журнал залишається повним і незмінним навіть після видалення основних записів.

Сутність CustomerInteraction (Взаємодія з клієнтом) фіксує хронологію відносин із клієнтом. Містить: Id (int, PK); CustomerId (int, зовнішній ключ до Customer, каскадне видалення при видаленні клієнта); Customer (навігаційна властивість); Type (InteractionType, перелічення: Call, Email, Meeting, Request,

Complaint); InteractionDate (DateTime, дата взаємодії); Description (string, TEXT, докладний опис взаємодії); Result (string, результат або підсумок); NextContactDate (DateTime?, планована дата наступного контакту, nullable). Сутність User (Користувач системи) містить: Id (int, PK); Username (string, унікальний, до 100 символів); PasswordHash (string, хеш пароля, до 500 символів для збереження довгих PBKDF2-хешів); Role (string, роль: «Admin», «Manager», «Viewer»); CreatedAt (DateTime); LastLoginAt (DateTime?, час останнього входу, nullable).

Зв'язки між сутностями визначені через Fluent API у методі OnModelCreating контексту. Зв'язок Customer → Order є «один до багатьох» з обмеженням Restrict на видалення: спроба видалити клієнта з пов'язаними замовленнями призведе до DbUpdateException. Зв'язок Order → OrderItem є «один до багатьох» з Cascade на видалення: позиції замовлення видаляються автоматично при видаленні замовлення. Зв'язок Product → OrderItem є «один до багатьох» з Restrict на видалення: видалення товару, що використовується у замовленнях, блокується. Зв'язок Customer → CustomerInteraction є «один до багатьох» з Cascade на видалення.

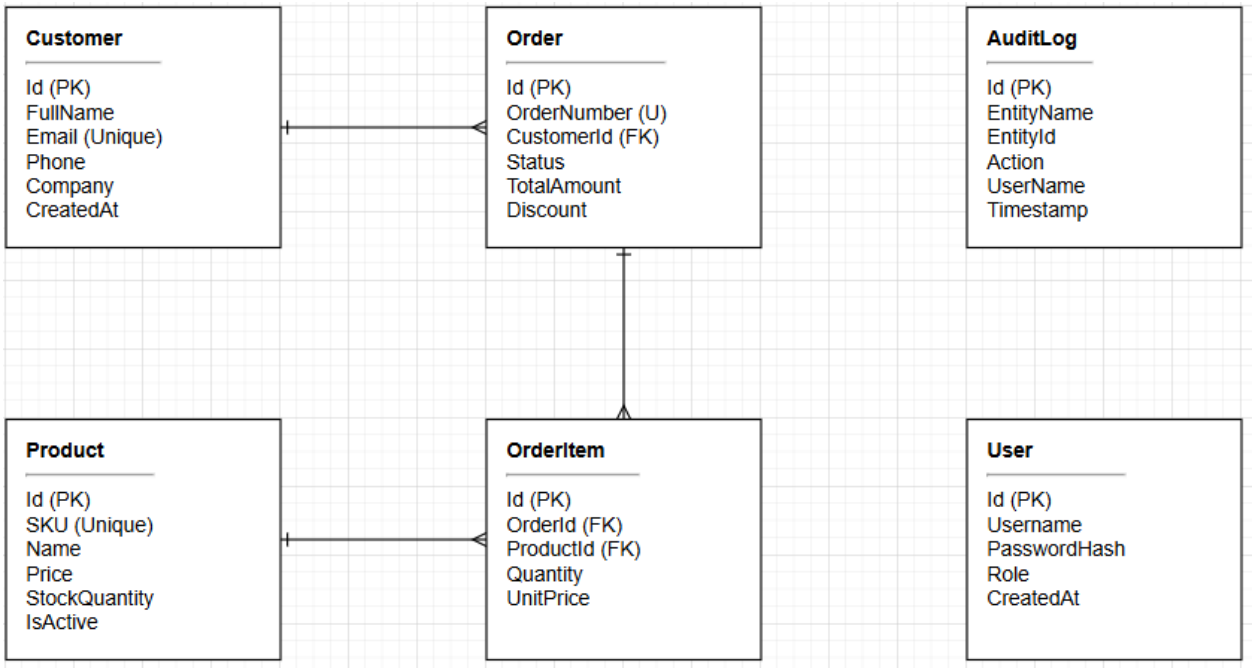

Рисунок 2.2. Схема логічної бази даних

2.4 Проектування сервісного шару і бізнес-логіки

Сервісний шар складається з одинадцяти сервісів, кожен із яких відповідає одній предметній галузі або технічній функції. Перед описом конкретних сервісів важливо зазначити загальні принципи їх дизайну. Усі методи сервісів є асинхронними (повертають Task або Task<T>) для уникнення блокування потоків веб-сервера при операціях вводу-виводу. Усі сервіси приймають залежності лише через конструктор – жодних глобальних стану або статичних залежностей. Кожен метод, що змінює дані, автоматично записує відповідну подію в журнал аудиту через IAuditLogService.

ICustomerService / CustomerService охоплює повний набір операцій із клієнтами. Метод GetAllAsync повертає повний список клієнтів, відсортованих за алфавітом. GetByIdAsync повертає клієнта за ідентифікатором або null при відсутності. GetWithOrdersAsync повертає клієнта разом із усіма пов'язаними замовленнями і їх позиціями через .Include(c => c.Orders).ThenInclude(o => o.OrderItems). SearchAsync приймає рядок пошуку і формує LINQ-запит із умовою Contains по чотирьох полях через оператор ||. CreateAsync виконує валідацію унікальності email (FirstOrDefaultAsync), додає сутність у контекст і зберігає, потім записує в аудит. UpdateAsync оновлює змінені поля і записує в аудит із деталями змін. DeleteAsync перевіряє відсутність замовлень через AnyAsync і видаляє з аудитом.

IProductService / ProductService реалізує управління товарним каталогом. GetLowStockAsync повертає активні товари, залишок яких менше або рівний порогу: .Where(p => p.IsActive && p.StockQuantity <= p.LowStockThreshold). При оновленні товару через UpdateAsync порівнюються поточні і нові значення Price і StockQuantity, і якщо вони змінились – у деталях запису аудиту фіксуються обидва значення для кожного зміненого поля. DeactivateAsync виконує «м'яке» видалення через встановлення IsActive = false замість фізичного видалення запису.

IOrderService / OrderService є найбільш технічно насиченим сервісом системи. Метод GenerateOrderNumberAsync реалізує потокобезпечну генерацію унікального номера: отримує рік поточної дати, підраховує кількість замовлень із OrderNumber, що починається на «ORD-{year}-», і формує наступний номер з тризначним суфіксом (D3-форматування з ведучими нулями). Слід зазначити, що цей алгоритм не є ідемпотентним при одночасному паралельному виклику від двох клієнтів – для виробничих систем із великим навантаженням доцільно реалізувати генерацію через базу даних із блокуванням або через послідовність (sequence).

Машина станів замовлення реалізована через словник допустимих переходів. Словник визначає: OrderStatus.New може переходити у Processing і Cancelled; OrderStatus.Processing – у Shipped і Cancelled; OrderStatus.Shipped – у Completed і Processing (повернення товару); OrderStatus.Completed і OrderStatus.Cancelled є термінальними станами без допустимих переходів. Метод ChangeStatusAsync отримує поточний статус замовлення із бази, перевіряє наявність поточного статусу у словнику і входження нового статусу у список допустимих, і при порушенні кидає InvalidOperationException із описовим повідомленням. Успішна зміна статусу фіксується в аудиті як окрема подія «StatusChange».

IAnalyticsService / AnalyticsService агрегує дані для дашборду і звітів. Метод GetDashboardDataAsync виконує кілька незалежних запитів через Task.WhenAll для паралельного виконання і мінімізації загального часу очікування. Для підрахунку загального доходу використовується SumAsync, що транслюється у SQL SUM. Для підрахунку замовлень за статусами використовується GroupBy, але через обмеження SQLite результат отримується після ToListAsync і групується на стороні C#. Для побудови місячної статистики завантажуються замовлення поточного року і групуються по місяцях через .GroupBy(o => o.OrderDate.Month) з подальшим перетворенням у масив 12 значень (з нулями для місяців без замовлень).

IAuditLogService / AuditLogService є технічним сервісом, що використовується іншими сервісами. Метод LogActionAsync приймає entityName, entityId, action, details і userName та зберігає новий запис AuditLog у базі. Метод GetLogsAsync повертає відфільтрований список записів із підтримкою фільтрів за entityName, action, userName і діапазоном дат (Timestamp >= from && Timestamp <= to). IAuthService / AuthService реалізує ValidateCredentials (пошук користувача за username, перевірка пароля через IPasswordHasher<User>.VerifyHashedPassword) і GetCurrentUser (отримання об'єкта User за claims поточного HttpContext). IExportService / ExportService генерує CSV через StringBuilder для клієнтів, товарів і замовлень із заголовками і рядками даних. IImportService / ImportService парсить CSV через StreamReader із обробкою заголовків, лапок у полях і некоректних рядків. ToastService є Singleton-сервісом, що реалізує паттерн Observer для відображення toast-повідомлень у UI.

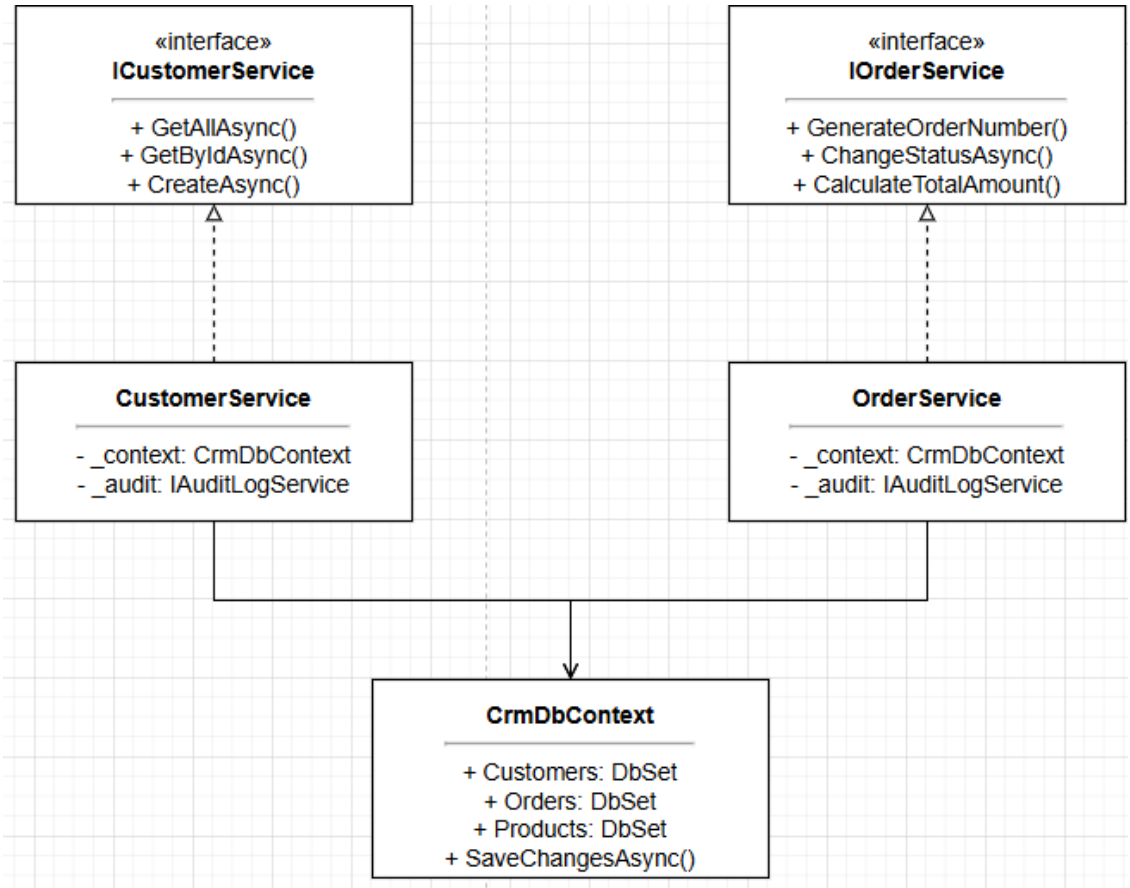

Рисунок 2.3. UML діаграма класів

2.5 Проектування інтерфейсу користувача

Проектування інтерфейсу розпочинається з формування UX-концепції – набору принципів, що визначають загальні підходи до організації інформації і взаємодії. Для розробленого модуля визначено такі UX-принципи. Мінімалізм і функціональність: кожний UI-елемент повинен мати чітку функцію і не конкурувати за увагу з іншими. Послідовність: аналогічні дії повинні виконуватись однаково у різних розділах – кнопки «Зберегти» і «Скасувати» завжди у нижньому правому куті форм, кнопки дій завжди у правому стовпці таблиць. Зворотній зв'язок: будь-яка операція повинна супроводжуватись видимою реакцією системи – зміною стану кнопки, відображенням індикатора завантаження або toast-повідомленням про результат. Захист від помилок: деструктивні операції потребують явного підтвердження, форми валідуються в реальному часі, заблоковані дії супроводжуються пояснювальним повідомленням.

Загальний макет застосунку (MainLayout.razor) реалізований за патерном «shell layout» із постійними елементами оболонки і змінним центральним контентом. Sidebar-навігація має фіксовану ширину 260px на десктопних пристроях і містить: логотип і назву системи у верхній частині; меню-посилання, згруповані за тематикою (основне – Дашборд; робота – Клієнти, Товари, Замовлення; аналітика – Аналітика, Статистика, Звіти; сервіс – Взаємодії, Журнал аудиту; налаштування – Профіль, Імпорт, Пошта); ім'я поточного користувача і кнопку виходу у нижній частині. Верхня панель (TopBar.razor) містить назву поточного розділу (читається з параметру або визначається по маршруту) і кнопку перемикання теми. Основна область контенту займає решту ширини і прокручується вертикально при переповненні.

Темна тема реалізована через CSS-змінні (Custom Properties) на рівні кореневого елемента :root. При перемиканні JavaScript-виклик через IJSRuntime додає або прибирає клас dark-theme на елементі body, що змінює

значення змінних (--bg-primary, --text-primary, --border-color і т.д.) для всіх компонентів одночасно. Вибір теми зберігається у localStorage браузера між сесіями через IJSRuntime-виклик localStorage.setItem/getItem. Усі кольори у CSS-стилях визначені через змінні, а не жорстко закодовані, що гарантує коректне відображення обох тем.

Сторінки-переліки (Customers.razor, Products.razor, Orders.razor) мають уніфіковану структуру. Верхній рядок містить заголовок розділу і кнопку «Додати» (лише для Manager і Admin). Рядок фільтрів і пошуку містить поле пошукового рядка з іконкою лупи і, залежно від розділу, додаткові фільтри (фільтр статусу для замовлень, фільтр активності для товарів). Таблиця містить клікабельні заголовки для сортування за стовпцями – при натисканні на заголовок колекція сортується за відповідним полем, повторне натискання змінює напрям. Кожний рядок таблиці містить кнопки дій праворуч. Рядок пагінації містить інформацію про кількість записів і кнопки навігації між сторінками.

Форми введення мають спільний базовий дизайн. Обов'язкові поля позначені зірочкою (*) після підпису. Поля валідуються в реальному часі при виході з поля (onblur) і відображають повідомлення про помилку валідації під полем червоним кольором. Кнопка «Зберегти» заблокована (disabled) до тих пір, поки форма містить помилки валідації. При успішному збереженні виконується NavigationManager.NavigateTo до сторінки переліку, і через EventCallback батьківський компонент отримує сповіщення про необхідність оновлення списку. При скасуванні виконується повернення до сторінки переліку без збереження змін.

Форма замовлення (OrderForm.razor) є найскладнішим UI-компонентом із погляду стану і взаємодій. Вона підтримує два режими (створення і редагування), що визначаються наявністю параметру Id у маршруті. В режимі редагування при ініціалізації виконується завантаження існуючого замовлення із позиціями і відображення у формі. Секція «Клієнт» містить поле вибору з пошуком по клієнтській базі. Секція «Позиції замовлення» є динамічною

таблицею: кожний рядок містить випадаючий список товарів, поле кількості і відображення ціни; при виборі товару автоматично підставляється його поточна ціна в поле UnitPrice і перераховується загальна сума рядка. Кнопка «+ Додати позицію» додає новий пустий рядок. Кнопка «×» у кожному рядку видаляє позицію. Секція підсумку відображає суму без знижки, поле введення відсоткової знижки і підсумкову суму, що перераховуються у реальному часі при будь-якій зміні.

2.6 Проектування системи безпеки

Безпека програмного забезпечення є комплексною і багаторівневою характеристикою. У контексті веборієнтованих застосунків вона охоплює автентифікацію (перевірку особи користувача), авторизацію (перевірку прав доступу до ресурсу), захист транспортного рівня (HTTPS), захист від типових веб-атак (XSS, CSRF, SQL Injection) і безпечне зберігання чутливих даних (паролів). ASP.NET Core надає вбудовані механізми для реалізації кожного з цих аспектів.

Аутентифікація реалізована через схему Cookie Authentication, вбудовану у ASP.NET Core. При успішному вході система формує об'єкт ClaimsPrincipal, що містить набір claims – атомарних тверджень про ідентифікованого користувача: ClaimTypes.NameIdentifier (числовий Id користувача), ClaimTypes.Name (ім'я для входу), ClaimTypes.Role (роль), а також кастомний claim із часом останнього входу. HttpContext.SignInAsync серіалізує ClaimsPrincipal, шифрує його через Data Protection API (AES-256-CBC + HMACSHA256) і встановлює cookie в HTTP-відповіді. Параметри cookie: HttpOnly = true (недоступна через JavaScript, що унеможливлює крадіжку через XSS); Secure = true у production (передача лише по HTTPS); SameSite = Lax (базовий захист від CSRF); ExpireTimeSpan = 8 годин; SlidingExpiration = true (час дії оновлюється при кожному запиті).

Авторизація реалізована через ролеву модель. Три ролі визначають різні рівні доступу. Роль Admin надає повний доступ до всіх функцій системи,

включно з управлінням обліковими записами користувачів (перегляд, створення, зміна пароля, деактивація), переглядом і очищенням журналу аудиту, перглядом адміністративних налаштувань і статистики. Роль Manager надає доступ до всіх операційних функцій: CRUD для клієнтів, товарів і замовлень, перегляд аналітики і звітів, фіксацію взаємодій, перегляд (але не очищення) журналу аудиту. Роль Viewer надає доступ лише до читання: можна переглядати клієнтів, товари, замовлення, аналітику і журнал аудиту, але неможливо нічого змінити.

Захист Blazor-компонентів реалізується через кілька механізмів. Директива @attribute [Authorize] на рівні компонента забезпечує перенаправлення на сторінку входу при спробі доступу неаутентифікованим користувачем. @attribute [Authorize(Roles = "Admin")] обмежує доступ до компонента лише для ролі Admin. Каскадний параметр AuthenticationState дозволяє компонентам динамічно приховувати UI-елементи залежно від ролі: @if (authState.User.IsInRole("Manager")) { <button>Редагувати</button> }. Компонент AuthorizeView надає декларативне визначення контенту для різних станів авторизації без C#-коду.

Паролі зберігаються виключно у хешованому вигляді через клас PasswordHasher<User> з пакету Microsoft.AspNetCore.Identity. Алгоритм PBKDF2 із HMACSHA512, 100 000 ітерацій і 128-бітовою сіллю відповідає рекомендаціям NIST SP 800-63B на 2024 рік. Сіль генерується індивідуально для кожного пароля і зберігається разом із хешем у єдиному рядку, що унеможливлює атаки типу rainbow table. Метод VerifyHashedPassword є стійким до timing-атак завдяки константному часу виконання порівняння.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ МОДУЛЯ CRM-СИСТЕМИ

3.1 Налаштування середовища розробки і структури проєкту

Розробка модуля здійснювалась у середовищі Visual Studio 2022 версії 17.8.4 із встановленими воркавтами ASP.NET і веброзробки, розробки для робочого столу .NET і Cross-platform development. Середовище надає повноцінну підтримку Blazor Web App: вбудований інструмент Blazor Language Services забезпечує підсвічування синтаксису у .razor-файлах, IntelliSense для директив і компонентів, навігацію по коду між .razor і .cs файлами та підтримку Hot Reload для миттєвого оновлення браузера при зміні Razor-компонентів. Для управління версіями коду використовується Git із патерном Gitflow: гілка main містить стабільні версії, develop – поточну розробку, а окремі feature-гілки – нові функціональні модулі.

Структура рішення (Solution) організована у трьох проєктах, пов'язаних через файл .sln. CRM-система.csproj є серверним проєктом і Startup Project рішення – саме він запускається командою dotnet run або через F5 у Visual Studio. Він посилається на CRM-система.Client.csproj через елемент ProjectReference, що дозволяє серверу під час публікації включати скомпільований WebAssembly-бандл клієнтського проєкту в каталог wwwroot/_framework. CRM-система.Tests.csproj посилається на серверний проєкт і містить виключно тести без власного бізнес-коду.

Внутрішня структура серверного проєкту організована за функціональним принципом. Папка Models містить дев'ять класів сутностей і перелічень (Customer, Product, Order, OrderItem, OrderStatus, CustomerInteraction, InteractionType, AuditLog, User). Папка Data містить CrmDbContext.cs і SeedData.cs. Папка Services містить двадцять один файл: по парі «інтерфейс / реалізація» для кожного з десяти предметних сервісів і окремо ToastService.cs. Папка Controllers містить п'ять контролерів. Папка Pages містить двадцять один Razor-компонент, що відповідають 21 UI-сторінці.

Папка Shared містить спільні компоненти (NavMenu, TopBar, ToastContainer, ConfirmDialog, Pagination). Папка Layout містить MainLayout.razor і LoginLayout.razor. Папка wwwroot містить CSS-стилі, JS-скрипти і статичні ресурси.

Ключові NuGet-пакети серверного проєкту визначені у файлі CRM-система.csproj. Microsoft.EntityFrameworkCore.Sqlite версії 8.0.2 надає провайдер SQLite для EF Core. Microsoft.EntityFrameworkCore.Tools версії 8.0.2 надає інструменти CLI для управління міграціями (команди Add-Migration, Update-Database). Microsoft.AspNetCore.Components.WebAssembly.Server забезпечує підтримку WebAssembly Static Files у серверному застосунку. Serilog.AspNetCore версії 8.0.0 інтегрує Serilog як основну систему логування. Serilog.Sinks.Console і Serilog.Sinks.File забезпечують виведення у консоль і файл відповідно. Swashbuckle.AspNetCore 6.5.0 і Swashbuckle.AspNetCore.Annotations додають Swagger UI і підтримку анотацій. Пакет Microsoft.AspNetCore.Identity.Core надає PasswordHasher без підключення повного стеку ASP.NET Core Identity.

Файл Program.cs є центральним файлом конфігурації і запуску застосунку. Він організований у чотири логічні секції. Перша секція налаштовує Serilog через статичний метод Log.Logger = new LoggerConfiguration(). Друга секція будує WebApplicationBuilder і реєструє всі сервіси у контейнері DI через builder.Services: AddDbContext, AddScoped для кожного з одинадцяти сервісів, AddControllers із Newtonsoft.Json для правильної серіалізації, AddRazorComponents із AddInteractiveServerComponents і AddInteractiveWebAssemblyComponents, AddSwaggerGen, AddAuthentication і AddAuthorization. Третя секція будує WebApplication через app.Build() і налаштовує конвеєр middleware у суворо визначеному порядку. Четверта секція ініціалізує базу даних (EnsureCreated і SeedData) і запускає застосунок через app.Run().

3.2 Реалізація моделі даних і контексту EF Core

Реалізація моделі даних розпочалась із визначення POCO-класів у папці Models. Клас Customer є типовим представником: він є звичайним C#-класом без успадкування від базових класів EF Core і без жодних залежностей від інфраструктурних бібліотек. Властивості класу декоровані атрибутами із двох просторів імен: System.ComponentModel.DataAnnotations (для валідації) і System.ComponentModel.DataAnnotations.Schema (для схеми). Атрибут [Required] позначає обов'язкові поля і автоматично використовується як DataAnnotationsValidator у формах Blazor для клієнтської валідації, і EF Core для генерації NOT NULL у SQL. Атрибут [MaxLength(200)] обмежує довжину рядка як у базі, так і у валідації форм. Атрибут [EmailAddress] перевіряє формат email на стороні клієнта.

Клас OrderStatus є перелічуванням C# із п'ятьма значеннями: New (новий), Processing (в обробці), Shipped (відправлений), Completed (завершений), Cancelled (скасований). Перелічування зберігається у базі як ціле число (int) за замовчуванням у EF Core, що є ефективним із точки зору розміру і продуктивності. Альтернативою є зберігання як рядок через конфігурацію HasConversion<string>() у Fluent API – цей підхід є більш читабельним при прямому перегляді бази, але займає більше місця і потребує відповідного індексу.

Клас CrmDbContext наслідує DbContext із EF Core і містить такі DbSet-властивості: Customers, Products, Orders, OrderItems, CustomerInteractions, AuditLogs, Users – кожна є generic-колекцією відповідного типу сутності. Конструктор приймає DbContextOptions<CrmDbContext> і передає їх у базовий конструктор через : base(options) – стандартний патерн, що дозволяє конфігурувати контекст ззовні при реєстрації у Program.cs.

Метод OnModelCreating виконує тонке налаштування схеми через Fluent API понад те, що EF Core виводить автоматично із атрибутів. Для Customer: HasIndex(c => c.Email).IsUnique() – унікальний індекс email; HasMany(c =>

c.Orders).WithOne(o => o.Customer).HasForeignKey(o => o.CustomerId).OnDelete(DeleteBehavior.Restrict) – зв'язок з обмеженням Restrict (блокування видалення клієнта із замовленнями). Для Order: HasIndex(o => o.OrderNumber).IsUnique() – унікальний індекс номера; HasMany(o => o.OrderItems).WithOne(i => i.Order).HasForeignKey(i => i.OrderId).OnDelete(DeleteBehavior.Cascade) – каскадне видалення позицій при видаленні замовлення. Для Product: HasIndex(p => p.SKU).IsUnique(); HasMany(p => p.OrderItems).WithOne(i => i.Product).HasForeignKey(i => i.ProductId).OnDelete(DeleteBehavior.Restrict).

SQLite не підтримує тип decimal із довільною точністю – він зберігає decimal як REAL (floating point), що може призводити до проблем точності при фінансових розрахунках. Для розв'язання цієї проблеми EF Core надає можливість зберігати decimal як рядок через HasConversion<string>() або як ціле число (у копійках) через HasConversion<long>(). У розробленому модулі обрано зберігання у вигляді рядка: builder.Entity<Order>().Property(o => o.TotalAmount).HasConversion<string>(). Це гарантує точне зберігання і відтворення decimal-значень без втрати точності, хоча і унеможливлює деякі SQL-агрегації на стороні бази (зокрема, SUM). Усі агрегаційні операції над TotalAmount виконуються на стороні C# після завантаження даних через ToListAsync().

Клас SeedData містить статичний метод Initialize(CrmDbContext context, IPasswordHasher<User> hasher), що виконує початкове заповнення бази тестовими даними при першому запуску. Метод перевіряє наявність записів через context.Users.Any(): якщо хоча б один запис існує, метод завершується без дій – це гарантує ідемпотентність і запобігає повторному заповненню при перезапуску застосунку. При порожній базі метод послідовно додає: двох користувачів із хешованими паролями (admin/admin123 і manager/manager123), п'ять клієнтів з реалістичними іменами і контактними даними, вісім товарів із різними цінами і залишками (два з яких є нижче порогу для демонстрації функції сигналізації), шість замовлень із різними статусами і позиціями, шість

взаємодій із клієнтами і чотири записи аудиту. Всі зміни зберігаються одним викликом context.SaveChanges().

3.3 Реалізація сервісів бізнес-логіки

Реалізація CustomerService ілюструє загальний підхід до побудови сервісів у системі. Клас CustomerService реалізує інтерфейс ICustomerService, що визначає контракт у вигляді дев'яти асинхронних методів. Залежності впроваджуються через конструктор: private readonly CrmDbContext _context і private readonly IAuditLogService _auditLogService. Ця явна декларація залежностей є ключовою перевагою: будь-який розробник, що відкриє клас уперше, одразу бачить усі зовнішні залежності без необхідності читати увесь код.

Метод SearchAsync(string query) реалізує ефективний пошук по чотирьох полях одночасно. Якщо рядок пошуку порожній або null, метод повертає повний список клієнтів. Інакше формується LINQ-запит: _context.Customers.Where(c => c.FullName.Contains(query) || c.Email.Contains(query) || c.Phone.Contains(query) || c.Company.Contains(query)).OrderBy(c => c.FullName).ToListAsync(). EF Core транслює Contains у SQL LIKE '%query%'. Важливою особливістю SQLite є нечутливість до регістру (case-insensitive) за замовчуванням для ASCII-символів, що є зручною поведінкою для пошуку, але може не працювати для кириличних символів залежно від версії SQLite. Для гарантованого нечутливого до регістру пошуку у кирилиці рекомендується приводити обидва операнди до нижнього регістру через EF.Functions.Lower().

Метод CreateAsync(Customer customer, string userName) виконує три послідовні кроки. Перший – перевірка унікальності email: var duplicate = await _context.Customers.FirstOrDefaultAsync(c => c.Email == customer.Email). Якщо дублікат знайдено, кидається виключення InvalidOperationException з повідомленням 'Клієнт із таким email вже зареєстрований'. Другий крок – збереження: customer.CreatedAt = DateTime.UtcNow;

_context.Customers.Add(customer); await _context.SaveChangesAsync(). Третій – запис аудиту: await _auditLogService.LogActionAsync("Customer", customer.Id, "Create", $"Ім'я: {customer.FullName}, Email: {customer.Email}", userName). Зверніть увагу, що Id клієнта після SaveChangesAsync вже заповнений базою даних (автоінкремент) і може бути використаний у записі аудиту.

Метод UpdateAsync(Customer updated, string userName) спочатку завантажує існуючий запис із бази: var existing = await _context.Customers.FindAsync(updated.Id). Якщо запис не знайдено, кидається KeyNotFoundException. Далі формується рядок деталей для аудиту, що описує змінені поля: метод порівнює значення updated і existing по кожному полю і додає до рядка деталей лише поля, що змінились, у форматі 'FieldName: OldValue → NewValue'. Потім EF Core відстежує зміни через механізм Change Tracking (властивості existing оновлюються значеннями updated), і SaveChangesAsync генерує SQL UPDATE лише для змінених колонок. Запис аудиту фіксує змінені поля.

Реалізація OrderService є найбільш технічно складною частиною сервісного шару. Метод GenerateOrderNumberAsync реалізує такий алгоритм: int year = DateTime.UtcNow.Year; int count = await _context.Orders.CountAsync(o => o.OrderNumber.StartsWith($"ORD-{year}-")); return $"ORD-{year}-{count + 1:D3}". Формат D3 в інтерполяції рядка форматує число як трицифровий рядок із ведучими нулями: 1 → «001», 12 → «012», 123 → «123». Результат – ORD-2024-007 для сьомого замовлення у 2024 році. Підрахунок за StartsWith є ефективним, якщо існує індекс на поле OrderNumber, що і забезпечено у Fluent API.

Метод CalculateTotalAmount(Order order) обчислює підсумкову суму замовлення. Спочатку обчислюється сума по всіх позиціях: decimal subtotal = order.OrderItems.Sum(item => item.UnitPrice * item.Quantity). Потім застосовується знижка: decimal total = subtotal * (1 - order.Discount / 100). Результат присвоюється властивості order.TotalAmount і зберігається в базі при наступному SaveChangesAsync. Збереження підсумкової суми в базі є

усвідомленим рішенням денормалізації: це дозволяє ефективно фільтрувати і сортувати замовлення за сумою у SQL-запитах без перерахунку на C#. Альтернативою є обчислення суми у SQL через вкладений запит, але це ускладнює реалізацію.

Машина станів у методі ChangeStatusAsync реалізована через Dictionary<OrderStatus, List<OrderStatus>>. Ключ словника – поточний статус, значення – список допустимих нових статусів. Термінальні стани Completed і Cancelled відсутні у словнику як ключі, тому будь-яка спроба змінити статус з термінального призведе до перевірки ContainsKey, що поверне false, і виключення InvalidOperationException. Таким чином, машина станів реалізована декларативно – через дані, а не через розгалужену умовну логіку. Додавання нового допустимого переходу в майбутньому потребуватиме лише зміни одного рядка у словнику без модифікації алгоритму перевірки.

ToastService реалізований як Singleton: він реєструється через AddSingleton<ToastService>() і існує протягом усього часу роботи застосунку. Це необхідно, оскільки ToastContainer підписується на подію сервісу при ініціалізації, і якщо б сервіс був Scoped (один на HTTP-запит), підписка втрачалась би між запитами. Сервіс зберігає чергу активних повідомлень у List<ToastMessage> і надає метод ShowToast(string message, ToastType type), що додає нове повідомлення і підіймає подію OnToastsChanged. ToastContainer підписується на цю подію і викликає StateHasChanged() для перерендерингу. Кожне повідомлення автоматично видаляється через 3 секунди за допомогою Task.Delay(3000) у фоновому завданні.

3.4 Реалізація Blazor-компонентів

Реалізація компонентів Blazor у Blazor Web App включає декілька принципових особливостей порівняно з традиційним Blazor Server або Blazor WebAssembly. Директива @rendermode у верхній частині компонента визначає режим рендерингу: @rendermode InteractiveServer або @rendermode InteractiveWebAssembly або @rendermode InteractiveAuto. Для більшості

сторінок системи обрано режим InteractiveServer, оскільки він забезпечує миттєву інтерактивність без затримки завантаження WebAssembly і дозволяє сервісним компонентам безпосередньо викликати сервіси, що мають доступ до бази даних.

Компонент Home.razor (дашборд) є першим, що бачить користувач після входу. Директиви @page "/" і @attribute [Authorize] визначають маршрут і вимогу аутентифікації. Через @inject впроваджуються IAnalyticsService і IJSRuntime. Поле DashboardData? dashboardData = null ініціалізується null – поки дані не завантажені, компонент відображає індикатор завантаження. У методі OnInitializedAsync: dashboardData = await AnalyticsService.GetDashboardDataAsync(). Після завершення EF Core-запитів StateHasChanged() викликається автоматично системою Blazor, і компонент перерендерується з отриманими даними. Блок @if (dashboardData == null) відображає спінер, блок @else – картки показників і графіки.

Інтеграція з Chart.js реалізована у методі OnAfterRenderAsync(bool firstRender). Цей метод викликається після кожного рендерингу компонента, але за допомогою параметра firstRender забезпечується однократна ініціалізація графіка. При firstRender == true і наявності завантажених даних виконується: await JSRuntime.InvokeVoidAsync("chartHelper.initChart", "revenueChart", labels, values), де labels – масив рядків із назвами місяців, values – масив decimal із доходом. JavaScript-функція chartHelper.initChart приймає id canvas-елемента і дані, ініціалізує об'єкт Chart з типом «line» і налаштованими опціями стилювання (колір лінії, заливка, підказки при наведенні). При знищенні компонента (IAsyncDisposable.DisposeAsync) викликається chartHelper.destroyChart для запобігання витокам пам'яті через Chart.js-об'єкти.

Компонент Customers.razor демонструє типовий патерн реалізації сторінок-переліків. Поле List<Customer>? customers обмінюється null і порожнім списком для розрізнення стану «завантаження» і «порожній список». Поле string searchQuery = string.Empty зберігає поточний рядок пошуку і прив'язується до поля введення через @bind-value і @bind-

value:event="oninput". Остання директива забезпечує пошук у реальному часі при кожному натисканні клавіші: обробник OnSearchChanged викликається із поточним значенням поля і запускає SearchCustomers(). Для уникнення надмірних звернень до бази при кожному натисканні реалізований debounce через CancellationTokenSource: попередній запит скасовується при наступному натисканні, якщо пройшло менше 300 мс.

Пагінація реалізована як окремий компонент Pagination.razor із параметрами TotalItems (загальна кількість записів), PageSize (записів на сторінку, за замовчуванням 15), CurrentPage (поточна сторінка) і EventCallback<int> OnPageChanged. Компонент відображає кнопки «Попередня», номери сторінок (до п'яти навколо поточної) і «Наступна». При натисканні на кнопку OnPageChanged.InvokeAsync(pageNumber) інформує батьківський компонент, що оновлює currentPage і перераховує видиму частину колекції через Skip і Take.

Форма CustomerForm.razor підтримує два режими через параметр [Parameter] public int? Id { get; set; }. У режимі редагування (Id != null) OnParametersSetAsync завантажує існуючого клієнта: customer = await CustomerService.GetByIdAsync(Id.Value). У режимі створення (Id == null) ініціалізується новий порожній об'єкт Customer. Компонент EditForm прив'язаний до змінної customer, а DataAnnotationsValidator автоматично перевіряє атрибути [Required], [MaxLength] і [EmailAddress]. ValidationMessage<Customer> для кожного поля відображає відповідне повідомлення про помилку. Кнопка «Зберегти» має атрибут type="submit" і обгортає метод HandleSubmit, що визначає режим (Create або Update) і викликає відповідний метод сервісу.

Сторінка OrderDetail.razor демонструє перегляд деталей замовлення і управління статусами. Кнопки зміни статусу генеруються динамічно на основі допустимих переходів: метод GetAllowedStatuses(order.Status) повертає список допустимих нових статусів, і для кожного статусу рендерується кнопка із відповідним написом і кольором. Натискання кнопки викликає

ChangeStatus(newStatus), що звертається до OrderService.ChangeStatusAsync і оновлює локальну копію замовлення для миттєвого відображення без перезавантаження сторінки. Блок із деталями замовлення містить інформацію про клієнта із посиланням на його профіль, таблицю позицій із артикулом, назвою товару, кількістю, ціною і сумою, і підсумковий блок зі знижкою і загальною сумою.

Компонент Invoice.razor генерує рахунок-фактуру для друку. Сторінка відповідає маршруту /invoice/{id:int} і завантажує замовлення з усіма Include: OrderItems → Product і Customer. HTML-розмітка рахунку побудована за стандартною структурою: заголовок (назва компанії, номер рахунку, дата), блок клієнта (ім'я, компанія, email, телефон), таблиця позицій (№, Товар, Кількість, Ціна, Сума), підсумковий блок (загальна сума без знижки, знижка у відсотках і грошовому еквіваленті, підсумкова сума). Кнопка «Друкувати» виконує await JSRuntime.InvokeVoidAsync("window.print"). CSS @media print приховує елементи navigation, header і кнопки дій, встановлює 10мм поля і розгортає таблицю на всю ширину сторінки.

3.5 Реалізація REST API і документації Swagger

REST API реалізований через п'ять контролерів ASP.NET Core у папці Controllers. Усі контролери декоровані атрибутами [ApiController] (вмикає автоматичну валідацію ModelState і повернення HTTP 400 при невалідних даних) і [Route("api/[controller]")] (визначає базовий маршрут через ім'я контролера). Атрибут [Authorize] на рівні класу контролера захищає всі ендпоінти від неаутентифікованих запитів. Конкретні ендпоінти, що потребують ролі Admin або Manager, додатково декоровані [Authorize(Roles = "Admin,Manager")].

CustomersController реалізує RESTful CRUD-інтерфейс. GET api/customers повертає список клієнтів із підтримкою параметра запиту search для фільтрації: [HttpGet] public async Task<IActionResult> GetAll([FromQuery] string? search = null). При наявності search метод делегує SearchAsync, інакше

– GetAllAsync. GET api/customers/{id} повертає клієнта або HTTP 404 при відсутності: var customer = await _customerService.GetByIdAsync(id); return customer != null ? Ok(customer) : NotFound(). POST api/customers приймає Customer у тілі запиту і повертає HTTP 201 Created із URL нового ресурсу у заголовку Location: return CreatedAtAction(nameof(GetById), new { id = created.Id }, created). PUT api/customers/{id} оновлює клієнта і повертає HTTP 204 No Content. DELETE api/customers/{id} видаляє клієнта або повертає HTTP 400 при спробі видалити клієнта з замовленнями (при отриманні InvalidOperationException сервісу).

ExportController реалізує ендпоінти для вивантаження CSV-файлів. Метод ExportCustomers формує CSV через ExportService.ExportCustomersAsCsv і повертає його у відповіді із правильними заголовками: Content-Type: text/csv; charset=utf-8 і Content-Disposition: attachment; filename="customers.csv". Використання charset=utf-8 і BOM (Byte Order Mark) у CSV забезпечує коректне відображення кириличних символів при відкритті у Microsoft Excel. Аналогічно реалізовані ExportOrders і ExportProducts.

Swagger налаштований у Program.cs через AddSwaggerGen із конфігурацією OpenApiInfo: Title = "CRM System API", Version = "v1", Description = "REST API для модуля CRM-системи обліку замовлень клієнтів", Contact з ім'ям і email розробника. Включення XML-документації (за умови активації генерації XML-коментарів у .csproj) дозволяє Swashbuckle читати XML-коментарі з методів і параметрів контролерів і відображати їх у Swagger UI. Анотації SwaggerOperation і SwaggerResponse на рівні методів надають додаткові метадані: Summary (короткий опис), Remarks (детальний опис), коди HTTP-відповідей і їх значення. UseSwaggerUI налаштовує маршрут /swagger/index.html і назву документа.

Глобальна обробка помилок реалізована через middleware UseExceptionHandler у Program.cs. При виникненні необробленого виключення middleware перехоплює його, визначає тип і формує відповідну HTTP-

відповідь у форматі application/problem+json відповідно до RFC 7807. InvalidOperationException транслюється у HTTP 400 Bad Request із деталями у полі detail. KeyNotFoundException – у HTTP 404 Not Found. DbUpdateException – у HTTP 409 Conflict із описом обмеження, що було порушено. Всі інші виключення транслюються у HTTP 500 Internal Server Error із загальним повідомленням (деталі помилки не відкриваються клієнту з міркувань безпеки, але логуються Serilog у файл).

3.6 Налаштування аутентифікації та авторизації

Аутентифікація налаштована у Program.cs через ланцюжок builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie(options => { options.LoginPath = "/login"; options.LogoutPath = "/api/auth/logout"; options.ExpireTimeSpan = TimeSpan.FromHours(8); options.SlidingExpiration = true; options.Cookie.HttpOnly = true; options.Cookie.SecurePolicy = CookieSecurePolicy.Always; options.Cookie.SameSite = SameSiteMode.Lax; options.Events.OnRedirectToLogin = context => { if (context.Request.Path.StartsWithSegments("/api")) { context.Response.StatusCode = 401; return Task.CompletedTask; } context.Response.Redirect(context.RedirectUri); return Task.CompletedTask; }; }). Остання налаштовка є важливою: вона забезпечує, що API-ендпоінти при відсутності аутентифікації повертають HTTP 401 (а не редирект на /login, що є неприйнятним для REST-клієнтів).

AuthController реалізує два ендпоінти. POST /api/auth/login приймає LoginDto (record із Username і Password) у тілі запиту. Спочатку виконується пошук користувача: var user = await _context.Users.FirstOrDefaultAsync(u => u.Username == dto.Username). При відсутності користувача метод відразу повертає Unauthorized() без уточнення причини – це стандартна практика безпеки, що запобігає перерахуванню існуючих імен користувачів. При знаходженні перевіряється пароль: var result =

_passwordHasher.VerifyHashedPassword(user, user.PasswordHash, dto.Password). Якщо result == PasswordVerificationResult.Failed, повертається Unauthorized(). При успіху формується ClaimsPrincipal: var claims = new List<Claim> { new(ClaimTypes.NameIdentifier, user.Id.ToString()), new(ClaimTypes.Name, user.Username), new(ClaimTypes.Role, user.Role) }; var identity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme); var principal = new ClaimsPrincipal(identity). Потім виконується HttpContext.SignInAsync і оновлення LastLoginAt у базі.

Blazor-сторінка Login.razor (із окремим LoginLayout без sidebar і авторизаційних перевірок) реалізує форму входу. EditForm прив'язаний до LoginModel з Username і Password. Поле Password рендериться як <InputText type="password">. При відправці форми через HandleLogin виконується HTTP-запит до API: var response = await Http.PostAsJsonAsync("/api/auth/login", loginModel). При response.IsSuccessStatusCode виконується NavigationManager.NavigateTo("/", forceLoad: true) – параметр forceLoad: true є обов'язковим, оскільки Blazor Server повинен «переініціалізуватись» після встановлення cookie аутентифікації, щоб CascadingAuthenticationState відобразив нового користувача. При помилці відображається повідомлення «Невірний логін або пароль».

3.7 Налаштування логування Serilog

Логування є критично важливим інструментом як для розробки (відладки), так і для виробничої підтримки системи. Serilog налаштовується на початку Program.cs до будь-яких інших операцій, що гарантує фіксацію помилок, що виникають навіть під час ініціалізації: Log.Logger = new LoggerConfiguration().MinimumLevel.Information().MinimumLevel.Override("Microsoft",
LogEventLevel.Warning).MinimumLevel.Override("Microsoft.EntityFrameworkCore.Database.Command",
LogEventLevel.Warning).Enrich.FromLogContext().WriteTo.Console(outputTemp

late: "[{Timestamp:HH:mm:ss} {Level:u3}] {Message:lj}{NewLine}{Exception}").WriteTo.File("Logs/crm-log-.txt", rollingInterval: RollingInterval.Day, retainedFileCountLimit: 30, outputTemplate: "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {Message:lj}{NewLine}{Exception}").CreateLogger().

Перевизначення рівня для Microsoft.EntityFrameworkCore.Database.Command на Warning у production є важливим з точки зору обсягу логів: кожен SQL-запит EF Core генерує запис у лозі рівня Debug/Information, і без цього перевизначення лог-файли заповнювались би тисячами рядків SQL-команд, що є непотрібним у production. У файлі appsettings.Development.json це перевизначення відсутнє, тому у режимі розробки всі SQL-запити відображаються у консолі – що є незамінним інструментом для налагодження продуктивності LINQ-запитів.

Структуроване логування Serilog використовується у ключових місцях сервісів для фіксації контексту операцій. Наприклад, у CustomerService.CreateAsync: Log.Information("Клієнт {FullName} ({Email}) створений користувачем {UserName} о {Timestamp}", customer.FullName, customer.Email, userName, DateTime.UtcNow). Завдяки структурованому логуванню параметри у фігурних дужках зберігаються не лише як частина рядка, а як окремі поля запису – це дозволяє у майбутньому підключити систему агрегації логів (Seq, Elasticsearch) і виконувати запити типу «показати всі дії користувача admin за 2024-04-07» без парсингу рядків регулярними виразами.

3.8 Розробка та результати автоматизованого тестування

Модуль автоматизованого тестування є обов'язковим елементом якісного програмного продукту. Тести виконують дві функції: по-перше, верифікують коректність поточної реалізації; по-друге, слугують «запобіжником» при подальшій розробці – будь-яка зміна, що порушує існуючу поведінку, буде

виявлена при виконанні тестів. У розробленому модулі тести охоплюють сім тестових класів, що відповідають семи основним сервісам системи.

Тестовий клас CustomerServiceTests містить дванадцять тестів. Для забезпечення ізоляції кожен тест використовує унікальну in-memory базу через допоміжний метод CreateContext: private CrmDbContext CreateContext() { var options = new DbContextOptionsBuilder<CrmDbContext>().UseInMemoryDatabase(Guid.NewGuid().ToString()).Options; return new CrmDbContext(options); }. Тест GetAllAsync_EmptyDatabase_ReturnsEmptyList перевіряє, що для порожньої бази повертається порожній (а не null) список. Тест CreateAsync_ValidCustomer_SetsIdAndSavesToDatabase перевіряє, що після збереження клієнт отримує ненульовий Id і присутній у базі: var retrieved = await context.Customers.FindAsync(created.Id); Assert.NotNull(retrieved); Assert.Equal("John Doe", retrieved.FullName). Тест CreateAsync_DuplicateEmail_ThrowsInvalidOperationException використовує Assert.ThrowsAsync<InvalidOperationException> для перевірки, що спроба зберегти другого клієнта з тим самим email кидає очікуване виключення.

Тест SearchAsync_ByName_ReturnsOnlyMatchingCustomers додає трьох клієнтів у базу і перевіряє, що пошук за ім'ям «Петро» повертає лише відповідних клієнтів і не повертає інших: Assert.Single(results); Assert.Equal("Петро Іваненко", results[0].FullName). Тест SearchAsync_CaseInsensitive_Works перевіряє нечутливість пошуку до регістру: пошук "ІВАНЕНКО" повинен знаходити «Петро Іваненко». Тест DeleteAsync_CustomerWithOrders_ThrowsException додає клієнта і замовлення, а потім перевіряє, що спроба видалити клієнта кидає InvalidOperationException. Тест DeleteAsync_CustomerWithoutOrders_Succeeds перевіряє успішне видалення клієнта без замовлень.

Тестовий клас OrderServiceTests містить п'ятнадцять тестів. GenerateOrderNumber_FirstOrderOfYear_ReturnsSuffix001 перевіряє, що перше замовлення 2024 року отримує OrderNumber «ORD-2024-001».

GenerateOrderNumber_SeventhOrder_ReturnsSuffix007 перевіряє коректне інкрементування при наявності шести існуючих замовлень. CalculateTotalAmount_TenItemsAtHundredWithTwentyPercentDiscount_Returns 800 перевіряє математику: 10 × 100 × (1 - 0.20) = 800.00. ChangeStatus_FromNewToProcessing_Succeeds перевіряє допустимий перехід. ChangeStatus_FromCompletedToProcessing_ThrowsException перевіряє блокування переходу з термінального стану. ChangeStatus_CreatesAuditLogEntry перевіряє, що після зміни статусу у таблиці AuditLogs з'явився запис із Action == "StatusChange" і відповідним EntityId.

Тестовий клас AnalyticsServiceTests містить вісім тестів. GetDashboardData_TotalRevenue_ExcludesCancelledOrders додає три замовлення (два активних по 1000 і одне скасоване за 500) і перевіряє, що TotalRevenue == 2000 (скасоване не враховується). GetDashboardData_TopClients_OrderedByTotalDescending додає трьох клієнтів із різними сумами замовлень і перевіряє, що метод повертає їх у порядку спадання. Тестовий клас AuthServiceTests містить п'ять тестів: успішна аутентифікація правильними даними, невдала аутентифікація неправильним паролем, невдала аутентифікація неіснуючим логіном, перевірка повернення ролі Admin, перевірка оновлення LastLoginAt після успішного входу. Тестовий клас ImportServiceTests містить сім тестів для перевірки парсингу CSV: коректний файл, порожній файл (0 рядків даних), рядок з відсутнім обов'язковим полем, рядок із зайвими пробілами, поле із комою у лапках, файл з кирилицею.

Підсумкові результати тестування: загальна кількість тестів – 61; успішно виконаних – 61; помилок – 0; пропущених – 0. Час виконання повного набору тестів на типовому розробницькому ноутбуці – менше 1.8 секунди, що є прийнятним для включення у процес CI/CD (Continuous Integration / Continuous Deployment) без суттєвого уповільнення збірки. Автоматизоване

тестування може бути виконане командою dotnet test з кореневої директорії рішення або через інтерфейс Test Explorer у Visual Studio.

3.9 Аналіз отриманих результатів

Розроблений модуль CRM-системи повністю реалізує набір функціональних вимог, визначених у другому розділі. Усі шість груп функцій – управління клієнтами, товарами, замовленнями, аналітика, взаємодії і аудит, аутентифікація – реалізовані і доступні через зручний веб-інтерфейс. Система пройшла ручне функціональне тестування всіх 21 UI-сторінки і 5 REST-контролерів, автоматизоване тестування 61 тестом і перевірку схеми бази даних після першого запуску.

Нефункціональні характеристики системи відповідають визначеним вимогам. Час відгуку для всіх операцій читання на тестових даних (5 клієнтів, 8 товарів, 6 замовлень) не перевищує 50 мс, що є значно нижче вимоги 2 секунди. Адаптивний дизайн забезпечує коректне відображення на пристроях від 320px до 1920px шириною. Темна тема перемикається миттєво без перезавантаження сторінки. Toast-повідомлення з'являються і зникають без перебоїв.

Практичне значення розробки визначається насамперед її автономністю і простотою розгортання. Для запуску системи необхідно: встановити .NET 8 Runtime (завантажується безкоштовно з офіційного сайту Microsoft); виконати dotnet run у каталозі серверного проєкту або запустити опублікований виконуваний файл; відкрити браузер за адресою http://localhost:5135. База даних crm.db створюється автоматично у робочому каталозі. Жодної додаткової конфігурації не потрібно. Ця простота є принциповою відмінністю від більшості комерційних рішень.

Порівняння із розглянутими у першому розділі аналогами підтверджує практичну цінність розробки. На відміну від Salesforce і Dynamics 365, система безкоштовна і не потребує щомісячної підписки. На відміну від HubSpot, вона функціонально повна у базовій версії (управління товарами, аудит, ролі). На

відміну від Pipedrive, вона орієнтована саме на обробку підтверджених замовлень, а не на управління переговорним процесом. На відміну від Bitrix24, вона не перевантажена непотрібним функціоналом і значно простіша в освоєнні. Місткість системи відповідає цільовому сегменту – малому підприємству до 50 співробітників із базою до 10 000 клієнтів і замовлень.

Напрямки для подальшого розвитку системи включають кілька пріоритетних областей. Перехід із SQLite на PostgreSQL або SQL Server є першим кроком при масштабуванні системи: завдяки абстракції EF Core ця зміна потребує лише заміни провайдера у Program.cs і рядка підключення в appsettings.json. Реалізація повноцінного модуля email-сповіщень на основі SMTP або хмарного провайдера (SendGrid, AWS SES) дозволить автоматично інформувати клієнтів про зміну статусу замовлення. Двофакторна аутентифікація через TOTP (Time-based One-Time Password) підвищить рівень безпеки облікових записів. Розширення аналітичного модуля функціями прогнозування (наприклад, прогноз продажів на наступний місяць на основі ML.NET або Porter's 5 forces model) надасть керівникам додаткові інструменти для прийняття рішень. Розробка мобільного застосунку на базі .NET MAUI, що використовує той самий REST API, забезпечить зручний мобільний доступ для менеджерів у «полі».

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання – спроектовано і розроблено модуль CRM-системи для автоматизації процесів обліку замовлень клієнтів, що відповідає потребам малого та середнього бізнесу. Отримані результати повністю відповідають визначеній меті і вирішують усі поставлені завдання.

1. Проведено комплексний аналіз предметної галузі. Вивчено еволюцію концепції CRM від ранніх ідей Беррі і Гаммессона до стратегічної моделі Пейна і Фроу. Проаналізовано причини масових невдач CRM-впроваджень початку 2000-х і їх вплив на сучасний підхід. Класифіковано CRM-системи за типом розгортання, функціональним спрямуванням і масштабом. Виконано порівняльний аналіз п'яти ринкових рішень – Salesforce, Microsoft Dynamics 365, HubSpot CRM, Pipedrive, Bitrix24 – і показано, що жодне з них не є оптимальним для цільового сегменту малого бізнесу. Це підтвердило доцільність власної розробки.

2. Обрано і обґрунтовано технологічний стек: платформа .NET 8 з мовою C# 12; фреймворк Blazor Web App із чотирма режимами рендерингу; ASP.NET Core із вбудованим DI; ORM Entity Framework Core 8 із підходом Code First; СУБД SQLite для розгортання без окремого серверного процесу; бібліотека логування Serilog; платформа тестування xUnit. Поєднання цих технологій забезпечує необхідну функціональність і мінімальну операційну складність розгортання.

3 Спроектовано архітектуру модуля за патерном N-tier із service layer. Розроблено модель даних із семи сутностей (Customer, Product, Order, OrderItem, CustomerInteraction, AuditLog, User) із визначеними атрибутами, зв'язками та обмеженнями цілісності. Обґрунтовано ключові архітектурні рішення: зберігання ціни у позиції замовлення, відокремлення AuditLog від зовнішніх ключів, визначення сервісів через інтерфейси. Спроектовано ролеву модель безпеки із трьома рівнями доступу і незмінним журналом аудиту.

4. Реалізовано повнофункціональний модуль CRM-системи. Серверний проєкт містить дев'ять класів моделей, контекст CrmDbContext із Fluent API конфігурацією, одинадцять сервісів бізнес-логіки та п'ять REST-контролерів. Клієнтський проєкт містить 21 Blazor-компонент із підтримкою темної теми, адаптивного дизайну, debounce-пошуку і JavaScript Interop для Chart.js. Налаштовано cookie-аутентифікацію з AES-256, Serilog-логування із щоденною ротацією і Swagger UI.

5. Розроблено модуль автоматизованого тестування: 61 тест у семи тестових класах, що охоплюють усі основні сервіси системи. Всі тести виконуються успішно за менше ніж 2 секунди, що підтверджує коректність реалізації.

Практичне значення розробки визначається її автономністю, простотою розгортання (один виконуваний файл і файл бази даних), функціональною достатністю та відкритою архітектурою для подальшого розширення.

Перспективними напрямками розвитку є: перехід на PostgreSQL або SQL Server; реалізація email-сповіщень про зміну статусу замовлення; двофакторна аутентифікація через TOTP; розширення аналітики на базі ML.NET; розробка мобільного застосунку .NET MAUI.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Buttle F. Customer Relationship Management: Concepts and Technologies. 3rd ed. London : Routledge, 2019. 452 p.

2. Gummeson E. Total Relationship Marketing. 3rd ed. Oxford : Butterworth-Heinemann, 2008. 376 p.

3. Payne A., Frow P. A Strategic Framework for Customer Relationship Management. Journal of Marketing. 2005. Vol. 69, № 4. P. 167–176.

4. Peppers D., Rogers M. Managing Customer Relationships: A Strategic Framework. 2nd ed. Hoboken : Wiley, 2011. 512 p.

5. Rigby D. K., Reichheld F. F., Schefter P. Avoid the Four Perils of CRM. Harvard Business Review. 2002. Vol. 80, № 2. P. 101–109.

6. Greenberg P. CRM at the Speed of Light. 4th ed. New York : McGraw-Hill, 2010. 668 p.

7. Chen I. J., Popovich K. Understanding Customer Relationship Management. Business Process Management Journal. 2003. Vol. 9, № 5. P. 672–688.

8. Chaffey D. Digital Business and E-Commerce Management. 7th ed. London : Pearson, 2019. 752 p.

9. Foss B., Stone M. CRM in Financial Services. London : Kogan Page, 2002. 784 p.

10. Reinartz W., Krafft M., Hoyer W. D. The Customer Relationship Management Process. Journal of Marketing Research. 2004. Vol. 41, № 3. P. 293–305.

11. Salesforce Documentation. URL: https://developer.salesforce.com/docs (дата звернення: 15.04.2026).

12. Microsoft Dynamics 365 Documentation. URL: https://learn.microsoft.com/en-us/dynamics365 (дата звернення: 15.03.2025).

13. HubSpot CRM Overview. URL: https://www.hubspot.com/products/crm (дата звернення: 16.04.2026).

14. Pipedrive Features Overview. URL: https://www.pipedrive.com/en/features (дата звернення: 16.03.2025).

15. Bitrix24 Help Center. URL: https://helpdesk.bitrix24.com (дата звернення: 17.04.2026).

16. Price M. C# 12 and .NET 8 – Modern Cross-Platform Development. Birmingham : Packt Publishing, 2023. 828 p.

17. Microsoft. .NET 8 What's New. URL: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8 (дата звернення: 20.04.2026).

18. Smith J. ASP.NET Core in Action. 3rd ed. Shelter Island : Manning, 2023. 752 p.

19. Microsoft. Blazor render modes in ASP.NET Core 8.0. URL: https://learn.microsoft.com/en-us/aspnet/core/blazor/components/render-modes (дата звернення: 21.04.2026).

20. Microsoft. Entity Framework Core Documentation. URL: https://learn.microsoft.com/en-us/ef/core (дата звернення: 22.04.2026).

21. Hipp D. R. Appropriate Uses For SQLite. URL: https://www.sqlite.org/whentouse.html (дата звернення: 25.04.2026).

22. Serilog. Structured Logging for .NET. URL: https://serilog.net (дата звернення: 24.04.2026).

23. xUnit.net. Getting Started with xUnit.net. URL: https://xunit.net/docs/getting-started (дата звернення: 27.03.2025).

24. Sainty C. Blazor in Action. Shelter Island : Manning, 2022. 448 p.

25. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston : Addison-Wesley, 2003. 560 p.

26. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River : Prentice Hall, 2018. 432 p.

27. Онищенко, В. В., & Бондарчук, А. П. (2015). Програмна інженерія: проблеми та перспективи. Зв’язок, (1).

28. Sommerville I. Software Engineering. 10th ed. London : Pearson, 2016. 816 p.

29. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.

30. Microsoft. ASP.NET Core Security Documentation. URL: https://learn.microsoft.com/en-us/aspnet/core/security (дата звернення: 05.04.2025).

31. Microsoft. EF Core – Configuring DbContext. URL: https://learn.microsoft.com/en-us/ef/core/dbcontext-configuration (дата звернення: 16.04.2026).

32. Карпунін, І., Зінченко, Н. (2023). Когнитивне моделювання інтелектуальних систем аналізу фінансового стану суб'єкта господарювання. Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка», 2023, 1(21), 75–85

33. Машкіна, І. В., Абрамов, В. О., Носенко, Т. І., Іваніченко, Є. В., & Яскевич, В. О. (2024). Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп'ютерні науки для здобувачів вищої освіти денної форми навчання.