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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-платформи для диджиталізації бізнес-процесів сервісу з ремонту побутової техніки»

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

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

_____ Алєся ЦИГАНОК
(підпис)

Виконав: здобувачка вищої освіти групи ПД-44

Алєся ЦИГАНОК

Керівник: Олександр ЯРОШЕВСЬКИЙ
асист.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Циганок Алєсі Миколаївні

1. Тема кваліфікаційної роботи: «Розробка web-платформи для диджиталізації бізнес-процесів сервісу з ремонту побутової техніки»

керівник кваліфікаційної роботи асистент кафедри ІПЗ Олександр ЯРОШЕВСЬКИЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

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

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про діджиталізацію бізнес-процесів сервісних підприємств, методи та підходи до автоматизації сервісного обслуговування, принципи побудови веб-платформ для управління заявками та взаємодії користувачів, технічна документація щодо використання засобів розробки програмного забезпечення.

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

1. Огляд та аналіз існуючих систем і технологій автоматизації сервісних центрів з ремонту побутової техніки..
2. Проєктування веб-платформи для діджиталізації бізнес-процесів сервісного центру побутової техніки.
3. Програмна реалізація та опис функціонування веб-платформи для автоматизації обробки заявок, управління ремонтом та взаємодії між користувачами.
4. Тестування веб-платформи та аналіз результатів її роботи.

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

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02.2026 - 28.02.2026	
2	Аналіз особливостей організації роботи сервісного центру ремонту побутової техніки	01.03-12.03	
3	Порівняльний аналіз програмних рішень та визначення їх недоліків	13.03-16.03	
4	Формування функціональних і нефункціональних вимог до веб-платформи	17.03-18.03	
5	Аналіз технологій та інструментів для розробки веб-застосунку, вибір архітектурного підходу та структури системи	19.03-24.03	
6	Розподіл проєкту на архітектурні шари	25.03	
7	Розробка шару Domain	26.03-29.03	
8	Розробка шару Application	30.03-01.04	
9	Розробка шару Infrastructure	02.04-05.04	
10	Створення проєкту модульного тестування Test	06.04	
11	Розробка та покриття системи модульними тестами	07.04-11.04	

12	Проєктування структури та створення бази даних	12.04-18.04	
13	Налаштування та застосування міграцій бази даних	20.04-23.04	
14	Проєктування та реалізація користувацького інтерфейсу	24.04-29.04	
15	Узагальнення результатів розробки та побудова UML-діаграм	30.04-02.05	
16	Розробка демонстраційних матеріалів	08.05	
17	Попередній захист роботи	11.05-22.05	

Здобувачка вищої освіти _____ Алєся ЦИГАНОК
(підпис)

Керівник
кваліфікаційної роботи _____ Олександр ЯРОШЕВСЬКИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 60 стор., 1 табл., 10 рис., 23 джерел.

Мета роботи – удосконалення роботи сервісного центру побутової техніки на основі діджиталізації процесів.

Об'єкт дослідження – бізнес-процеси сервісного центру з ремонту побутової техніки, пов'язані з прийомом, обробкою та виконанням заявок на ремонт, а також взаємодією між клієнтами та персоналом.

Предмет дослідження – засоби, методи та підходи до діджиталізації бізнес-процесів сервісного центру, що забезпечують автоматизацію обробки заявок, підтримку життєвого циклу обслуговування та координацію взаємодії між учасниками процесу, а також принципи побудови програмних систем для реалізації зазначених функцій у вигляді веб-платформи.

Короткий зміст роботи: У роботі проведено аналіз предметної області сервісних центрів з ремонту побутової техніки та досліджено особливості організації процесів обробки заявок. Виконано аналіз існуючих програмних рішень для автоматизації сервісного обслуговування, зокрема RepairDesk, Orderry та RoApp, визначено їх переваги, недоліки та обмеження. На основі проведеного аналізу сформовано функціональні та нефункціональні вимоги до веб-платформи.

У роботі спроєктовано архітектуру системи на основі принципів Clean Architecture та MVC-підходу. Розроблено доменну модель системи, ER-діаграму бази даних та діаграми варіантів використання. Реалізовано веб-платформу для автоматизації діяльності сервісного центру, яка забезпечує створення та обробку заявок, підтримку рольової моделі користувачів, внутрішню комунікацію між учасниками процесу, ведення чек-листів виконаних робіт та контроль життєвого циклу заявки.

Під час реалізації застосунку використано ASP.NET Core MVC, Entity

Framework Core, Microsoft SQL Server, SignalR, FluentValidation та ASP.NET Core Identity. Проведено тестування основних функціональних можливостей системи та перевірено коректність роботи окремих модулів.

Сферою використання розробленої веб-платформи є автоматизація діяльності сервісних центрів з ремонту побутової техніки, оптимізація обробки заявок та покращення взаємодії між клієнтами й персоналом.

КЛЮЧОВІ СЛОВА: ВЕБ-ПЛАТФОРМА, СЕРВІСНИЙ ЦЕНТР, РЕМОНТ ПОБУТОВОЇ ТЕХНІКИ, АВТОМАТИЗАЦІЯ БІЗНЕС-ПРОЦЕСІВ, ASP.NET CORE MVC, CLEAN ARCHITECTURE, ENTITY FRAMEWORK CORE, СИСТЕМА УПРАВЛІННЯ ЗАЯВКАМИ, РОЛЬОВА МОДЕЛЬ ДОСТУПУ, БАЗА ДАНИХ, CQRS, SIGNALR, ВЕБ-ЗАСТОСУНОК.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасні сервісні центри з ремонту побутової техніки працюють в умовах постійного збільшення кількості заявок та необхідності швидкої взаємодії між клієнтами й персоналом. У багатьох випадках процеси прийому та обробки заявок виконуються вручну або за допомогою розрізнених інструментів, що призводить до втрати інформації, затримок у роботі та складності контролю виконання ремонтів. Тому виникає потреба у впровадженні сучасних цифрових рішень для автоматизації діяльності сервісних центрів.

Проєкт передбачає розробку веб-платформи для діджиталізації бізнес-процесів сервісного центру побутової техніки. Основною ідеєю системи є перенесення ключових процесів — створення та обробки заявок, виконання ремонту, комунікації між учасниками — у єдине цифрове середовище. Це дозволяє підвищити ефективність роботи персоналу, зменшити кількість помилок та забезпечити контроль усіх етапів обслуговування.

Діджиталізація у даному випадку полягає не лише у створенні веб-сайту, а у трансформації бізнес-процесів сервісного центру. Система забезпечує централізоване управління заявками, підтримує життєвий цикл ремонту, внутрішню комунікацію та взаємодію між клієнтами, консультантами й майстрами.

Об’єктом дослідження є бізнес-процеси сервісного центру з ремонту побутової техніки, пов’язані з прийомом, обробкою та виконанням заявок на ремонт, а також взаємодією між клієнтами та персоналом.

Предметом дослідження є засоби, методи та підходи до діджиталізації бізнес-процесів сервісного центру, що забезпечують автоматизацію обробки заявок, підтримку життєвого циклу обслуговування та координацію взаємодії між учасниками процесу, а також принципи побудови програмних систем для реалізації зазначених функцій у вигляді веб-платформи.

Метою роботи є удосконалення роботи сервісного центру побутової техніки на основі діджиталізації процесів.

Методи дослідження. У роботі проведено аналіз предметної області та існуючих програмних рішень для автоматизації сервісного обслуговування. Для реалізації системи використано ASP.NET Core MVC, C#, Entity Framework Core, Microsoft SQL Server, HTML, CSS та JavaScript. Архітектуру застосунку побудовано на основі Clean Architecture та CQRS. Для реалізації внутрішньої комунікації використано SignalR. Проєктування системи виконано за допомогою UML-діаграм та ER-моделі бази даних.

Наукова новизна роботи полягає у розробці веб-платформи, яка поєднує систему управління заявками, рольову модель доступу, внутрішню комунікацію та механізми контролю виконання ремонтних робіт у межах єдиного цифрового середовища.

Практична значущість результатів полягає у можливості використання розробленої веб-платформи для автоматизації діяльності сервісних центрів, оптимізації обробки заявок та покращення взаємодії між клієнтами й персоналом.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Особливості організації роботи сервісного центру ремонту побутової техніки

Сервісний центр з ремонту побутової техніки — це підприємство, яке займається діагностикою, технічним обслуговуванням та ремонтом побутових приладів, таких як пральні машини, холодильники, мікрохвильові печі та інша техніка. Робота центру включає не лише виконання ремонтних робіт, а й організацію процесу обробки заявок, взаємодію з клієнтами та контроль виконання ремонту [18, 19].

Робота сервісного центру починається зі звернення клієнта, під час якого фіксуються контактні дані, тип техніки та опис несправності. Після цього виконується діагностика обладнання, визначається характер поломки та орієнтовна вартість ремонту. Після погодження вартості майстер виконує ремонт, а після завершення техніка повертається клієнту. Для збереження та обробки інформації про заявки та клієнтів використовуються системи керування базами даних [16, 17].

Особливістю діяльності сервісного центру є багатоетапність обробки заявок. Кожна заявка проходить декілька стадій — від створення до завершення ремонту, тому важливо контролювати зміну статусів та фіксувати результати виконаних робіт. Важливу роль також відіграє правильний розподіл заявок між майстрами з урахуванням їх спеціалізації та поточного навантаження [5, 18].

У роботі сервісного центру значну роль відіграє взаємодія між клієнтом, консультантом і майстром. Консультант координує обробку звернень, контролює стан ремонту та забезпечує інформаційну підтримку клієнтів. Для ефективної роботи необхідне єдине інформаційне середовище, яке дозволяє зберігати історію заявок, статуси ремонту та результати виконаних робіт [4, 23].

Цифровізація діяльності сервісного центру передбачає використання спеціалізованого програмного забезпечення для автоматизації обробки заявок, управління даними та розмежування прав доступу користувачів [1, 3, 14].

1.2 Аналіз існуючих програмних рішень

На сьогоднішній день, сервісні центри з ремонту побутової техніки все частіше використовують спеціалізовані програмні системи для автоматизації внутрішніх процесів. Такі рішення допомагають впорядкувати роботу із заявками, контролювати етапи ремонту, організувати взаємодію між працівниками та покращити якість обслуговування клієнтів. У більшості випадків подібні системи належать до класу Field Service Management (FSM) або CRM-систем, які орієнтовані на управління сервісними процесами та роботу з клієнтською базою. До основних функцій подібних програмних засобів належать:

1. Управління заявками, включаючи їх створення, редагування та контроль статусів виконання;
2. Розподіл навантаження між майстрами з урахуванням поточної зайнятості та спеціалізації;
3. Ведення клієнтської бази та збереження історії звернень;
4. Облік виконаних робіт, використаних матеріалів і формування вартості ремонту;
5. Організація комунікації між працівниками сервісного центру та клієнтами;
6. Формування аналітики, звітів і статистичних показників діяльності підприємства.

Попри велику кількість програмних рішень, більшість із них мають обмеження. Частина систем орієнтована на великі компанії й містить надлишковий функціонал, що ускладнює їхнє використання у невеликих сервісних центрах. Інші ж забезпечують лише базову автоматизацію і не підтримують складні сценарії: розподіл навантаження, централізовану комунікацію чи гнучке налаштування процесів.

Ще однією поширеною проблемою є недостатній рівень інтеграції процесів. У багатьох системах комунікація з клієнтами відбувається через сторонні сервіси або месенджери, що ускладнює збереження повної історії взаємодії та знижує контроль над інформацією. Також не всі рішення підтримують чіткий розподіл ролей користувачів, зокрема виділення консультанта як окремого учасника процесу між клієнтом і майстром.

Для більш детального аналізу було розглянуто три програмні системи, які використовуються у сфері сервісного обслуговування побутової техніки: RepairDesk, Orderry та RoApp. Вибір цих рішень зумовлений їх поширеністю, різними підходами до організації бізнес-процесів та відмінностями у функціональних можливостях. Їх порівняння дозволяє визначити сильні та слабкі сторони існуючих систем, а також сформувати обґрунтовані вимоги до розробки власної веб-платформи для автоматизації діяльності сервісного центру.

1.2.1 RepairDesk

RepairDesk — це хмарна система, призначена для автоматизації роботи сервісних центрів і майстерень з ремонту техніки. Вона орієнтована на управління ключовими процесами сервісного бізнесу, зокрема прийомом заявок, веденням клієнтської бази, контролем етапів ремонту та фінансовими операціями.

У системі передбачено створення заявок на ремонт із зазначенням даних клієнта, типу та моделі пристрою, опису несправності та поточного статусу виконання робіт. Кожна заявка супроводжується історією змін, а також може бути закріплена за конкретним майстром. Обробка заявки відбувається поетапно — від реєстрації та діагностики до завершення ремонту і видачі пристрою клієнту [5].

Окремий блок функціоналу стосується управління складом та фінансами. RepairDesk дозволяє вести облік запасних частин, контролювати використання матеріалів у процесі ремонту, формувати рахунки та здійснювати облік оплат. Також система підтримує інтеграції із зовнішніми сервісами для проведення платежів і надсилання повідомлень клієнтам. Для аналітики доступні звіти щодо виконаних ремонтів, продуктивності працівників і фінансових показників [9, 19].

Цифровізація діяльності сервісного центру передбачає використання спеціалізованого програмного забезпечення для автоматизації обробки заявок, управління даними та розмежування прав доступу користувачів [3, 14, 18].

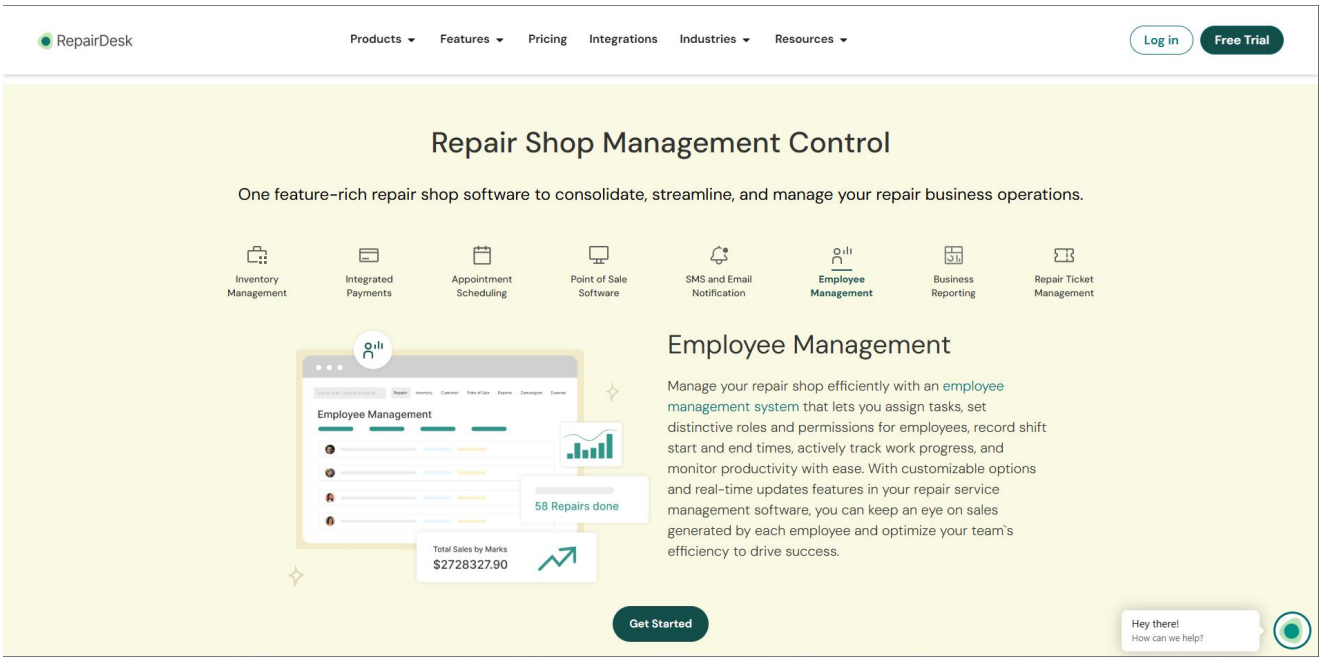


Рис. 1.1 Головна сторінка RepairDesk

Попри широкий функціонал, система має ряд недоліків. Основна проблема — перевантажений інтерфейс, складний в освоєнні для нових користувачів, що потребує додаткового навчання [18, 19]. Оскільки частина функцій орієнтована на великі компанії, для малого та середнього бізнесу вони є надлишковими й лише ускладнюють роботу [23].

Інший мінус — недостатня гнучкість. RepairDesk працює за визначеною логікою, тому адаптація під особливості конкретного сервісного центру обмежена. Зміна сценаріїв обробки заявок чи налаштування специфічних ролей часто недоступні, що створює труднощі для компаній із нестандартними процесами [4].

Проблемною є й комунікація. Взаємодія з клієнтами йде через зовнішні канали (SMS та email), тому історія спілкування не централізована в заявці. Відсутність внутрішнього чату ускладнює контроль за діалогом «клієнт–майстер–менеджер» [11, 13]. Крім того, немає окремої ролі консультанта як посередника, що обмежує організацію роботи персоналу [18].

Попри високу технологічну зрілість, система RepairDesk демонструє обмежену релевантність запитам мікропідприємств та сервісних центрів з нетиповою організаційною структурою [23].

1.2.2 Orderry

Orderry — це CRM/FSM-система, орієнтована на автоматизацію діяльності сервісних центрів і ремонтних майстерень. Вона використовується для організації роботи із заявками, ведення клієнтської бази та контролю окремих внутрішніх процесів підприємства. Система надає можливість створювати заявки на ремонт, змінювати їх статуси, призначати відповідальних виконавців та переглядати історію виконання робіт. Також передбачено збереження інформації про клієнтів, їх попередні звернення та виконані замовлення.

Orderry підтримує роботу зі складським обліком і запчастинами, дозволяє формувати фінансову документацію та переглядати базову статистику діяльності сервісного центру. Система має веб-інтерфейс, завдяки чому доступ до неї можливий з різних пристроїв без встановлення додаткового програмного забезпечення.

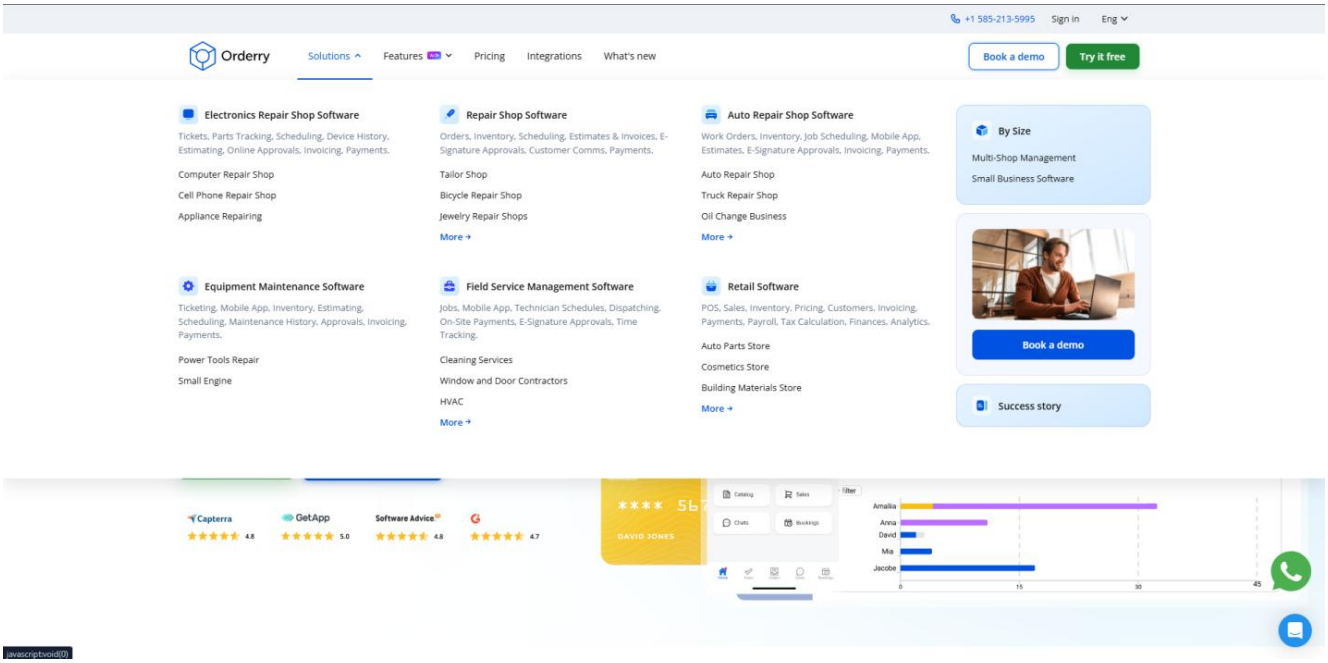


Рис. 1.2 Головна сторінка Orderry

Попри широкий функціонал, система має ряд недоліків. Основна проблема — перевантажений інтерфейс та надмірна кількість функцій, частина з яких може бути непотрібною для невеликих сервісних центрів. Orderry — це CRM/FSM-система, орієнтована на автоматизацію діяльності сервісних центрів і ремонтних майстерень. Вона використовується для організації роботи із заявками, ведення клієнтської бази та контролю окремих внутрішніх процесів підприємства [18, 19].

Система надає можливість створювати заявки на ремонт, змінювати їх статуси, призначати відповідальних виконавців та переглядати історію виконання робіт. Також передбачено збереження інформації про клієнтів, їх попередні звернення та виконані замовлення [5, 18].

Orderry підтримує роботу зі складським обліком і запчастинами, дозволяє формувати фінансову документацію та переглядати базову статистику діяльності сервісного центру. Система має веб-інтерфейс, завдяки чому доступ до неї можливий з різних пристроїв без встановлення додаткового програмного забезпечення [1, 3].

1.2.3 RoApp

RoApp — це веб-орієнтована система для управління сервісним бізнесом, яка призначена для базової автоматизації роботи сервісних центрів та ремонтних майстерень. Система орієнтована переважно на невеликі компанії, яким необхідний простий інструмент для ведення заявок, контролю виконання робіт та збереження інформації про клієнтів [18, 19]. Робота здійснюється через браузер, тому система не потребує складного встановлення або окремої серверної інфраструктури [1, 3].

Основний функціонал RoApp пов’язаний із роботою із заявками на ремонт. Користувач може створювати нові заявки, вносити інформацію про клієнта, тип обладнання, опис несправності та відповідального виконавця. Для кожної заявки передбачено зміну статусу виконання, що дозволяє відстежувати етапи ремонту [5,

18]. Також система підтримує ведення клієнтської бази, збереження історії звернень та формування базових фінансових документів [17, 18].

Окрему увагу в системі приділено комунікації з клієнтами. RoApp підтримує інтеграцію із зовнішніми сервісами, такими як SMS та месенджери, що дозволяє інформувати користувачів про зміну статусу ремонту або готовність техніки до видачі [7, 18]. Крім цього, система має базові засоби аналітики та звітності, які дозволяють переглядати інформацію щодо виконаних робіт і фінансових операцій [19].

Інтерфейс системи є відносно простим та не перевантаженим великою кількістю функцій, що спрощує початкове освоєння. Саме завдяки цьому RoApp часто розглядається як рішення для швидкого запуску автоматизації без тривалого навчання персоналу.

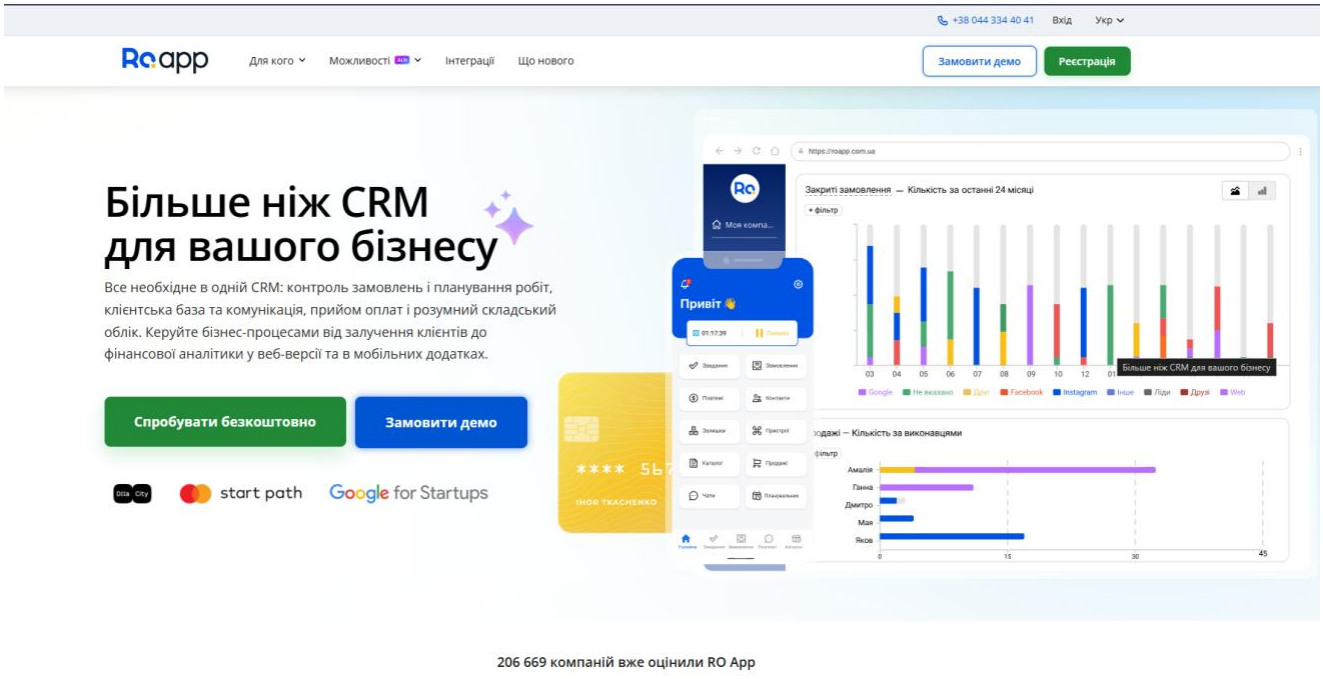


Рис. 1.3 Головна сторінка RoApp

Разом із цим система має ряд недоліків, які обмежують діджиталізацію бізнес-процесів. Насамперед це низький рівень автоматизації: розподіл заявок, контроль навантаження та координація робіт потребують ручного керування. Це перевантажує персонал при великому потоці замовлень.

Інший мінус — слабка підтримка складної бізнес-логіки. Система підходить для простих сценаріїв і важко адаптується під специфічні процеси, нестандартні етапи обробки заявок чи додатковий контроль[4, 6] .

Проблемною є й комунікація. Через відсутність централізованого чату в межах заявки взаємодія йде через зовнішні сервіси. Як наслідок, історія спілкування розпорошується, що ускладнює контроль за домовленостями. Також у системі не реалізовано балансування навантаження — вона не аналізує зайнятість майстрів і не розподіляє завдання автоматично. Крім того, у RoApp фактично відсутня роль консультанта як посередника між клієнтом та майстром, що важливо для багатьох сервісних центрів [5, 19].

Висновок: RoApp є простим рішенням для базової автоматизації сервісного бізнесу та може використовуватися для невеликих майстерень із нескладними процесами роботи. Проте обмежені можливості автоматизації, слабка гнучкість налаштування та відсутність централізованої комунікації не дозволяють розглядати систему як повноцінне рішення для комплексної діджиталізації сучасного сервісного центру.

1.3 Порівняльний аналіз систем

Для узагальнення аналізу та висновків щодо відповідності програмних рішень потребам сервісного центру було розроблено систему критеріїв порівняння. Кожен із них відображає практично важливий аспект функціонування системи.

Перелік містить як базові функції, так і розширені можливості, що визначають рівень автоматизації та гнучкості. Зокрема, окремо виділено наявність централізованого чату та підтримку ролей, оскільки ці аспекти є слабкими місцями більшості рішень на ринку і прямо впливають на якість комунікації. Критерій масштабованості відображає здатність системи працювати стабільно при зростанні кількості заявок і користувачів. Критерій простоти використання оцінює доступність інтерфейсу для персоналу без спеціальної підготовки, що

критично для його швидкого впровадження.

Оцінювання проводилося за трирівневою шкалою: «+» — функція реалізована повністю та доступна без додаткового налаштування; «±» — функція реалізована частково, потребує налаштування або має суттєві обмеження у використанні; «–» — функція відсутня або недоступна в рамках стандартного функціоналу системи. Результати порівняння наведено у таблиці 1.1.

Таблиця 1.1

Порівняльний аналіз систем

Критерій	RepairDesk	Orderry	RoApp
Управління заявками	+	+	+
Життєвий цикл заявки	+	+	±
Автоматичний розподіл	±	–	–
Гнучкість налаштувань	±	+	–
Комунікація (вбудована)	±	–	±
Централізований чат	–	–	–
Облік послуг і вартості	+	+	+
Аналітика	+	+	±
Простота використання	–	+	+
Масштабованість	+	+	±
Підтримка ролей	±	±	–
Інтеграції	+	±	±

Аналіз таблиці показує, що управління заявками та облік послуг і вартості присутні у всіх трьох системах — цей функціонал є базовим стандартом для рішень такого класу. Автоматичний розподіл заявок між виконавцями реалізований лише частково у RepairDesk, тоді як у Orderry та RoApp він відсутній повністю, що означає ручне призначення майстрів у переважній більшості випадків. Централізований чат у контексті заявки відсутній у всіх трьох системах, через що комунікація між учасниками процесу відбувається поза межами системи

і не зберігається в єдиному просторі. Підтримка ролей реалізована лише частково у RepairDesk та Orderry, а у RoApp відсутня зовсім — жодна із систем не виділяє роль консультанта як окремого учасника з чітко визначеними функціями.

Проведений порівняльний аналіз показує, що жодна з розглянутих систем не задовольняє повною мірою потреби сервісного центру. Відсутність централізованої комунікації в контексті заявки, недостатня підтримка рольової моделі та обмежена автоматизація розподілу навантаження є спільними недоліками всіх трьох рішень.

1.4 Визначення вимог до платформи

Сервісні центри побутової техніки часто ведуть заявки вручну або в таблицях, через що виникають затримки, помилки та складнощі з контролем ремонту. Для спрощення роботи потрібна єдина система, у якій можна створювати заявки, призначати майстрів, вести історію ремонту та спілкуватися всередині системи.

Метою роботи є розробка веб-платформи для автоматизації сервісного центру побутової техніки. Система дозволяє клієнтам створювати заявки, консультантам їх обробляти, майстрам виконувати ремонт, а адміністраторам контролювати роботу системи.

Функціональні вимоги:

1. Система має забезпечувати реєстрацію та авторизацію користувачів із розмежуванням доступу на основі ролей Client, Consultant, Master та Admin.

2. Користувач із роллю Client повинен мати можливість створювати заявки на ремонт, вказуючи тип обладнання, опис несправності, адресу та додаткові файли.

3. Система повинна забезпечувати можливість створення, оновлення та отримання заявок із бази даних.

4. Система повинна обмежувати доступ до заявок відповідно до ролей користувачів.

5. Система повинна підтримувати призначення майстра для заявки у ручному та автоматичному режимах.

6. Система повинна забезпечувати ведення чек-листа виконаних робіт для кожної заявки.

7. Система повинна забезпечувати внутрішню комунікацію у вигляді чату, прив’язаного до заявки.

Нефункціональні вимоги:

1. Час обробки CRUD-запитів ≤ 3 сек.

2. Збереження чат-повідомлення ≤ 1 сек.

3. Підтримка ≥ 10 одночасних користувачів.

4. Час обробки заявки ≤ 5 хв.

5. Безпека даних (Identity, ролі доступу, захист від XSS/CSRF/SQL-ін’єкцій).

6. Асинхронна робота всіх операцій з БД та I/O (async/await).

7. Модульна архітектура (Clean Architecture + DI + тестованість).

2 ЗАСОБИ РЕАЛІЗАЦІЇ ПЛАТФОРМИ

2.1 Середовище розробки

Для розробки веб-платформи використовувалися сучасні середовища розробки, які забезпечують створення, редагування та налагодження програмного забезпечення. Вони підтримують роботу як із серверною, так і з клієнтською частиною системи в межах одного проєкту. Основний акцент робився на зручності розробки, підтримці технологій .NET та інтеграції із системами контролю версій. Також середовища надають інструменти для пошуку помилок, аналізу коду та виконання типових задач [1, 2, 3].

2.1.1 Visual Studio Code

Visual Studio Code — легке та зручне середовище розробки, яке використовувалося переважно для роботи з клієнтською частиною системи, конфігураційними файлами та JavaScript-кодом. У ньому створювалися HTML-сторінки, CSS-стилі, JavaScript-скрипти [1, 18].

Середовище підтримує IntelliSense, що прискорює написання коду та зменшує кількість помилок. Також воно містить вбудований термінал і інтеграцію з Git, що дозволяє керувати репозиторієм та відстежувати зміни безпосередньо в середовищі.

2.1.2 Visual Studio

Visual Studio — інтегроване середовище розробки від Microsoft, яке використовувалося для створення серверної частини веб-платформи на основі ASP.NET Core MVC. Середовище забезпечує зручну роботу зі структурою проєкту, підтримку багаторівневої архітектури та можливості налагодження [1]. У ньому реалізовано основні шари системи: Application, Domain та Infrastructure, а також їх взаємодію.

Visual Studio підтримує Entity Framework Core, що дозволяє працювати з базою даних, створювати міграції та описувати сутності. Також використовувалися інструменти Dependency Injection, авторизації та конфігурації сервісів. Вбудований дебагінг і підтримка Git спрощують процес розробки та тестування [1, 3].

2.2 Мови програмування

Під час розробки веб-платформи використовувалися сучасні мови програмування для створення як серверної, так і клієнтської частини системи [2, 3]. Основна мова застосовувалася для реалізації бізнес-логіки, обробки заявок, роботи з даними та взаємодії між різними частинами системи. Додаткова використовується для забезпечення інтерактивності інтерфейсу, зокрема для динамічного оновлення статусів заявок, валідації форм, обробки дій користувача та асинхронної взаємодії з сервером через API без перезавантаження сторінки.

Такий підхід дозволив розділити логіку системи на окремі частини, що зробило проєкт більш структурованим, зрозумілим і зручним для подальшого розвитку.

2.2.1 C#

C# — об’єктно-орієнтована мова програмування платформи .NET, яка використовується для реалізації серверної частини веб-платформи. У проєкті на її основі реалізовано основну бізнес-логіку: створення та обробка заявок, зміна статусів, призначення майстрів, формування чек-листів, розрахунок вартості та обробка повідомлень. Також мова застосовується для реалізації авторизації користувачів і взаємодії з базою даних через Entity Framework Core [2].

Підтримка асинхронності дозволяє ефективно обробляти одночасні запити, що підвищує продуктивність системи при великій кількості користувачів. У

поєднанні з ASP.NET Core це забезпечує чітке розділення логіки на компоненти та спрощує масштабування архітектури застосунку [3].

2.2.2 JavaScript

JavaScript — об’єктно-орієнтована мова програмування, яка використовується для реалізації клієнтської частини веб-застосунків та керування HTML/CSS інтерфейсом. У проєкті вона забезпечує інтерактивність користувацького інтерфейсу: динамічне оновлення статусів заявок без перезавантаження сторінки, відображення чек-листів, валідацію форм та візуалізацію розрахунку вартості. Також мова відповідає за обробку дій користувача, анімацію елементів та взаємодію з сервером через API [18, 19].

Підтримка асинхронності дозволяє виконувати запити до сервера без блокування інтерфейсу, що забезпечує більш плавну та швидку роботу веб-додатку. У поєднанні з сучасними підходами до побудови UI це дає можливість розділяти інтерфейс на незалежні компоненти та спрощує масштабування фронтенд-частини системи [19].

2.3 Веб-фреймворки

Веб-фреймворки використовуються як основа для розробки серверної частини веб-застосунків і визначають спосіб організації коду системи. Вони задають структуру роботи із запитами, маршрутизацією та формуванням відповідей, що дозволяє розробляти систему більш впорядковано та зрозуміло [1].

Використання фреймворків спрощує реалізацію бізнес-логіки та взаємодію між різними частинами застосунку. Також вони підтримують підхід розділення відповідальностей, що полегшує подальше супроводження та розвиток системи. Окрім цього, фреймворки забезпечують можливість інтеграції з базами даних, системами авторизації та іншими сервісами [3].

2.4 Архітектурні підходи

У проєкті використано сучасні архітектурні підходи, що забезпечують чітке розділення відповідальностей у системі. Вони дозволяють відокремити бізнес-логіку від технічної реалізації та інтерфейсу користувача. Завдяки цьому система стає більш гнучкою та легко масштабованою. Архітектурні принципи забезпечують незалежність компонентів, що спрощує їх тестування та модифікацію. Окремі компоненти можна змінювати незалежно один від одного, що зменшує ризик впливу змін на всю систему.

2.4.1 Clean Architecture

Clean Architecture — це архітектурний підхід, який передбачає поділ системи на незалежні шари з чітким розмежуванням відповідальностей. У розробленій платформі система була поділена на шари Presentation, Application, Domain та Infrastructure. Основна ідея підходу полягає у відокремленні бізнес-логіки від технічної реалізації. Завдяки цьому доменні сутності не залежать від бази даних, інтерфейсу користувача чи зовнішніх сервісів. Clean Architecture також забезпечує можливість незалежної зміни окремих частин системи без впливу на інші компоненти [4, 23].

2.4.2 CQRS (з використанням MediatR)

CQRS є архітектурним підходом, який передбачає розділення операцій читання та зміни даних. У межах проєкту цей підхід реалізовано за допомогою бібліотеки MediatR, яка забезпечує реалізацію патерну посередника та спрощує організацію взаємодії між компонентами системи [15, 4].

Операції, які змінюють стан системи (наприклад створення заявки або оновлення статусу), виконуються через Commands. Операції, які лише отримують дані, реалізовані через Queries. MediatR використовується як проміжний шар між контролерами та бізнес-логікою. Він допомагає зменшити залежність між компонентами системи та централізувати обробку запитів [15, 4].

2.4.3 ASP.NET Core MVC

ASP.NET Core MVC використовується як основа серверної частини системи та реалізує патерн Model-View-Controller. Фреймворк розділяє систему на модель, представлення та контролер. Контролери обробляють HTTP-запити, передають їх до Application Layer і повертають результат у вигляді HTML або JSON [1].

ASP.NET Core MVC підтримує Dependency Injection, middleware та маршрутизацію, а також інтегрується з Entity Framework Core, SignalR та Identity. Це дозволяє будувати гнучку та масштабовану архітектуру сучасних веб-застосунків [3].

2.5 Технології доступу до даних

Технології доступу до даних забезпечують взаємодію застосунку з реляційною базою даних. Вони дозволяють працювати з даними у вигляді об’єктів, що спрощує реалізацію бізнес-логіки. Завдяки цьому зменшується потреба у написанні складних SQL-запитів вручну. Такі технології підтримують механізми міграцій, що дозволяє автоматично синхронізувати структуру бази даних із моделлю системи, також забезпечують цілісність даних і ефективну роботу з великими обсягами інформації [12, 17].

2.5.1 Entity Framework Core

Entity Framework Core використовується як ORM для взаємодії з базою даних через доменні сутності [12]. У проєкті застосовувався DbContext та набір сутностей, що забезпечує узгодженість між моделлю системи та базою даних. Міграції дозволяють автоматично оновлювати структуру БД відповідно до змін у коді [12].

2.5.2 Microsoft SQL Server

Microsoft SQL Server — це реляційна система керування базами даних, яка використовується для збереження інформації веб-платформи. У базі даних зберігаються заявки на ремонт, користувачі системи, повідомлення чату, перелік послуг, типи техніки та інші дані, необхідні для роботи сервісного центру. SQL Server забезпечує підтримку реляційної моделі даних, цілісність інформації та механізми роботи із зовнішніми ключами та зв’язками між таблицями [9, 17].

SQL Server використовується разом з Entity Framework Core, що дозволяє поєднати зручну об’єктну модель з реляційною базою даних та спростити доступ до даних у веб-застосунку [12].

2.6 Бібліотеки та технології

У розробці застосовувалися спеціалізовані бібліотеки, що спрощують реалізацію бізнес-логіки та технічних процесів. Вони дозволяють стандартизувати обробку даних, валідацію та комунікацію між компонентами системи [4, 15]. Частина бібліотек використовується для реалізації реального часу взаємодії між користувачами. Інші забезпечують централізовану перевірку даних і зменшують кількість помилок у системі [4].

2.6.1 FluentValidation

FluentValidation — це бібліотека для перевірки коректності вхідних даних у застосунках на платформі .NET. У межах проєкту вона використовується для валідації даних заявок, повідомлень чату, облікових записів користувачів та інших сутностей системи [2, 18].

Основна ідея бібліотеки полягає в тому, що всі правила перевірки виносяться в окремі класи-валідатори. Це дозволяє централізовано описувати логіку перевірки та відокремлювати її від основної бізнес-логіки системи [2]. FluentValidation легко інтегрується з ASP.NET Core MVC та CQRS-підходом,

забезпечуючи автоматичну валідацію даних перед виконанням бізнес-операцій [3, 2].

2.6.2 SignalR / WebSockets

SignalR є технологією платформи ASP.NET Core для реалізації обміну даними в режимі реального часу. В її основі використовується WebSockets, що забезпечує постійне двостороннє з’єднання між сервером і клієнтом [13].

У проєкті SignalR застосовується для реалізації чату між клієнтом, консультантом і майстром. Завдяки цьому повідомлення передаються миттєво, без перезавантаження сторінки [1].

2.7 Системи автентифікації та авторизації

Системи автентифікації та авторизації забезпечують безпечний доступ користувачів до функціоналу платформи. Вони відповідають за перевірку особи користувача та визначення його прав у системі. Реалізовано рольову модель доступу, що дозволяє розмежувати функції між різними типами користувачів.

2.7.1 ASP.NET Core Identity

ASP.NET Core Identity — це вбудована підсистема платформи .NET, призначена для реалізації повного циклу керування користувачами у веб-застосунках. Вона забезпечує функціонал реєстрації, автентифікації, управління профілями користувачів, ролями та правами доступу [2, 3].

У розроблюваній веб-платформі дана технологія використовується як базовий рівень ідентифікації користувачів та контролю доступу до функціоналу системи. ASP.NET Core Identity забезпечує безпечне зберігання облікових даних шляхом використання криптографічного хешування паролів, що виключає їх зберігання у відкритому вигляді. Додатково система підтримує механізми блокування облікових записів, відновлення доступу, а також розширення користувацьких профілів [1].

Важливою перевагою є глибока інтеграція з ASP.NET Core MVC, що дозволяє безпосередньо використовувати механізми автентифікації у контролерах та бізнес-логіці, також Identity формує основу багаторівневої та гнучкої системи безпеки веб-платформи.

2.8 Клієнтські технології

Клієнтські технології використовуються для створення інтерфейсу користувача веб-платформи. Вони забезпечують структуру сторінок, їх стилізацію та інтерактивність. Завдяки цим технологіям реалізується взаємодія користувача із системою через браузер. Вони дозволяють динамічно оновлювати дані без перезавантаження сторінкита забезпечують адаптивність інтерфейсу під різні пристрої.

2.8.1 HTML, CSS

HTML (HyperText Markup Language) є базовою мовою розмітки, яка використовується для створення структури веб-сторінок. У межах розроблюваної системи HTML застосовується для формування основних інтерфейсних компонентів: форм створення та обробки заявок, таблиць відображення даних, сторінок профілів користувачів, а також навігаційної структури веб-застосунку. HTML визначає логічну структуру сторінки та виступає основою для подальшого застосування стилізації та інтерактивності [3]. У системі CSS забезпечує візуальне оформлення всіх сторінок, включаючи кольорову схему, типографіку, розміщення елементів, відступи та адаптивність інтерфейсу під різні пристрої [1].

2.9 Система контролю версій

Система контролю версій використовується для управління вихідним кодом проєкту. Вона дозволяє відстежувати зміни, що вносяться під час розробки, та зберігати історію модифікацій. Також система підтримує командну роботу над

проєктом. Вона дозволяє організовувати різні гілки розробки та об’єднувати їх у єдину структуру.

2.9.1 Git

Git — це розподілена система контролю версій, яка використовується для управління вихідним кодом програмного забезпечення та відстеження змін у процесі розробки [18]. У межах проєкту Git застосовується для ведення історії змін коду, організації командної роботи та управління різними гілками розробки. Використання Git також забезпечує прозорість процесу розробки, оскільки всі зміни фіксуються та можуть бути проаналізовані. Інтеграція з середовищами розробки, такими як Visual Studio та VS Code, дозволяє автоматизувати процеси коміту, злиття гілок та синхронізації з віддаленими репозиторіями [18, 19].

2.10 Інструменти дизайну інтерфейсу

Інструменти дизайну інтерфейсу використовуються для проєктування зовнішнього вигляду веб-платформи. Вони дозволяють створювати макети сторінок та моделювати користувацький досвід ще до етапу програмної реалізації. Такі інструменти допомагають визначити логіку розміщення елементів інтерфейсу. Вони також дозволяють швидко вносити зміни в дизайн без впливу на код.

2.10.1 Figma

Figma — це хмарний інструмент для проєктування інтерфейсів користувача та створення інтерактивних прототипів. У рамках розробки веб-платформи Figma використовується для проєктування структури сторінок, визначення логіки розміщення елементів інтерфейсу та попереднього моделювання користувацького досвіду. Завдяки Figma вдалося сформувати загальне бачення інтерфейсу ще до початку програмування, що зменшило кількість змін під час розробки і зробило структуру системи більш узгодженою.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Діаграми варіантів використання

Для формалізації функціональної поведінки веб-платформи сервісного центру ремонту побутової техніки була розроблена діаграма варіантів використання (Use Case Diagram). Вона відображає взаємодію користувачів із системою, визначає межі функціональності та основні бізнес-процеси обробки заявок на ремонт. Діаграма дозволяє встановити зв’язок між ролями користувачів та відповідними функціональними можливостями системи [18].

У системі виділено чотири ролі: Клієнт, Консультант, Майстер та Адміністратор, кожна з яких має власний набір сценаріїв відповідно до своїх обов’язків [19].

Діаграма варіантів використання для клієнта описує основні можливості взаємодії користувача із веб-платформою сервісного центру ремонту побутової техніки. У системі передбачено два типи користувачів: незареєстрований клієнт та зареєстрований клієнт. Кожен із них має власний набір функціональних можливостей.

Незареєстрований користувач може виконувати реєстрацію в системі через сценарій «Реєстрація». Під час цього процесу користувач вводить необхідні дані для створення облікового запису, після чого система формує нового користувача з роллю клієнта. Також незареєстрований користувач має доступ до перегляду переліку послуг та їх вартості без необхідності авторизації [11]. Це дозволяє ознайомитися з функціоналом сервісного центру та приблизною ціною послуг ще до створення заявки.

Після проходження реєстрації користувач отримує можливість входу в систему через сценарій «Вхід в систему». Авторизований клієнт може переглядати власні заявки на ремонт, створювати нові заявки та взаємодіяти із консультантом і майстром через чат [14].

Основним сценарієм для зареєстрованого клієнта є «Перегляд власних заявок». У межах цього процесу користувач отримує інформацію про створені заявки, їх поточний статус, відповідального майстра та історію виконання робіт. Даний сценарій розширюється через use case «Створити нову заявку», який дозволяє клієнту ініціювати новий процес ремонту. Під час створення заявки користувач вказує тип техніки, опис несправності та іншу необхідну інформацію.

Також сценарій «Перегляд власних заявок» розширюється варіантом використання «Чат-спілкування з консультантом та майстром». Це забезпечує можливість централізованої комунікації між учасниками процесу ремонту в межах конкретної заявки. Користувач може уточнювати деталі ремонту, отримувати інформацію про стан виконання робіт та погоджувати додаткові послуги.

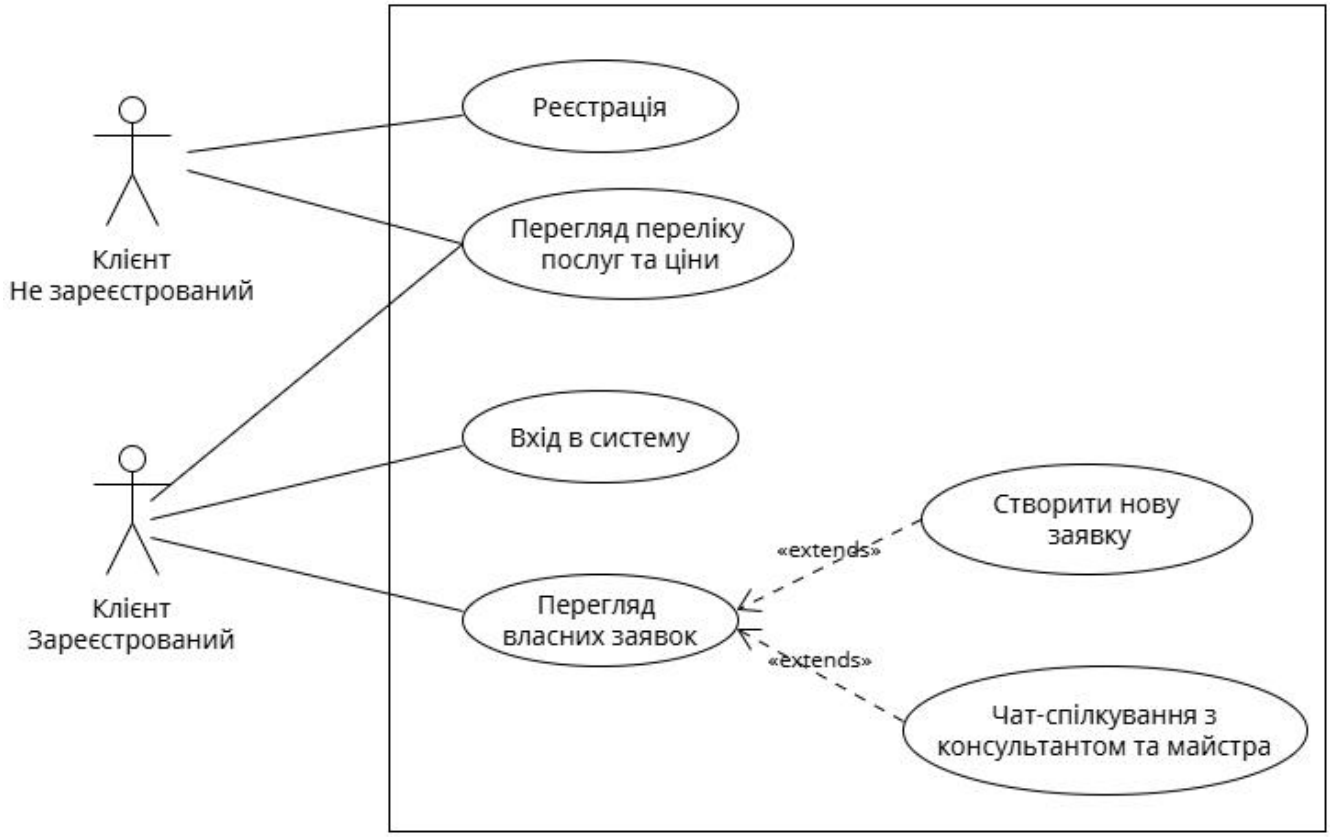


Рис. 3.1 Діаграма варіантів використання для клієнта

Діаграма варіантів використання для майстра та консультанта відображає процеси обробки заявок на ремонт і координації роботи сервісного центру. Дані ролі відповідають за організацію та виконання ремонтних робіт, контроль статусів заявок і взаємодію з клієнтами.

Майстер має доступ до сценарію «Перегляд власних активних заявок», який дозволяє отримувати список призначених йому заявок на ремонт. Через цей процес майстер може контролювати поточні завдання, переглядати інформацію про клієнта, опис несправності та історію обробки заявки.

Сценарій «Перегляд заявки» є розширенням процесу перегляду активних заявок та забезпечує детальне відображення інформації щодо конкретного ремонту. Після відкриття заявки майстер переходить до сценарію «Опрацювати заявку», який є центральним процесом його роботи.

У межах сценарію «Опрацювати заявку» передбачено декілька додаткових функцій. Через use case «Додати послуги» майстер може формувати перелік виконаних робіт та послуг, які були застосовані під час ремонту. Це дозволяє формувати чек-лист ремонту та обчислювати загальну вартість заявки.

Сценарії «Змінити статус заявки» та «Завершити заявку» розширюють процес опрацювання заявки. Майстер може змінювати статус ремонту відповідно до етапу виконання робіт, наприклад перевести заявку у стан діагностики, ремонту або завершення. Після завершення ремонту заявка переводиться у фінальний статус, а система фіксує завершення робіт.

Додатково реалізовано сценарій «Чат — спілкування з клієнтом», який забезпечує комунікацію між майстром та клієнтом без використання сторонніх сервісів. Це дозволяє уточнювати технічні деталі, погоджувати ремонтні роботи та централізовано зберігати історію повідомлень.

Консультант у системі виконує координаційну функцію. Основним сценарієм є «Обробка заявки», у межах якого консультант контролює надходження нових звернень та організовує їх подальшу обробку.

Сценарій «Призначення майстру нової заявки» включається до процесу обробки заявки та дозволяє закріпити заявку за конкретним майстром відповідно до його спеціалізації або поточного навантаження.

Окрім цього, консультант має доступ до сценарію «Моніторинг активних заявок», який розширює функціонал обробки заявок. Він дозволяє контролювати поточний стан ремонтів, відстежувати проблемні заявки та координувати взаємодію між клієнтами та технічними спеціалістами.

Також консультант бере участь у сценарії «Планування графіку роботи», який включає функцію «Планування робочого дня». Це забезпечує можливість формування графіків роботи майстрів, розподілу навантаження та організації роботи сервісного центру.

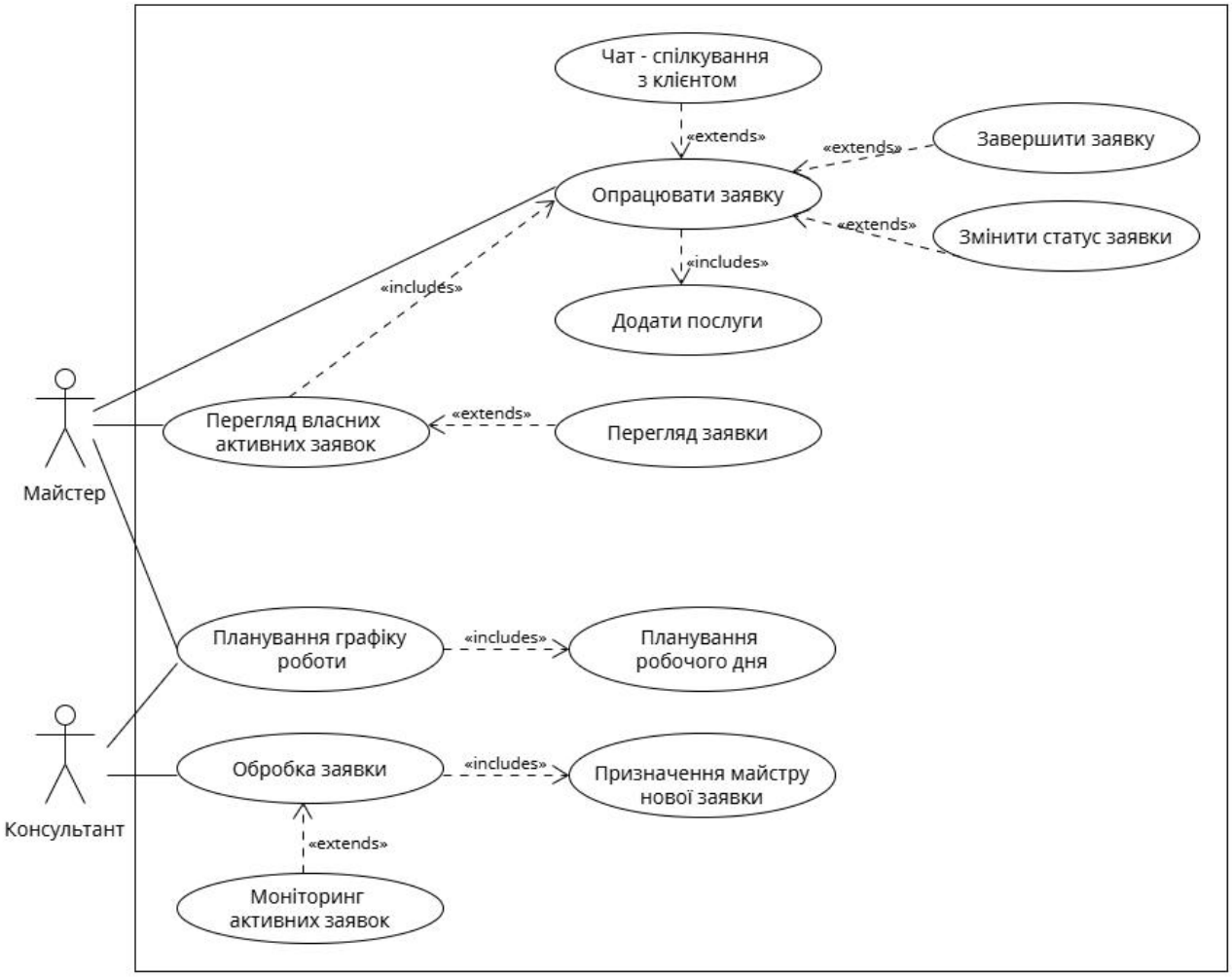


Рис. 3.2 Діаграма варіантів використання для клієнта

Адміністратор має доступ до сценарію «Переглянути послуги», який дозволяє отримувати список доступних сервісних послуг та керувати ними. Даний сценарій розширюється трьома додатковими варіантами використання: «Додати послугу», «Редагувати послугу» та «Видалити послугу». Сценарій «Додати послугу» забезпечує створення нової сервісної послуги із зазначенням її назви, опису та вартості. Це дозволяє формувати актуальний перелік ремонтних робіт, доступних у сервісному центрі. Через сценарій «Редагувати послугу» адміністратор може змінювати характеристики вже існуючих послуг, наприклад оновлювати ціну або змінювати назву послуги відповідно до потреб підприємства. Сценарій «Видалити послугу» дозволяє прибирати неактуальні або застарілі послуги з системи. Це забезпечує підтримку актуальності довідника та коректність формування вартості ремонтів. Окрім керування послугами, адміністратор має доступ до сценарію «Переглянути типи техніки». Сценарій «Додати новий тип техніки» розширює функціонал перегляду та дозволяє створювати нові категорії побутової техніки, наприклад пральні машини, холодильники, кондиціонери або мікрохвильові печі.

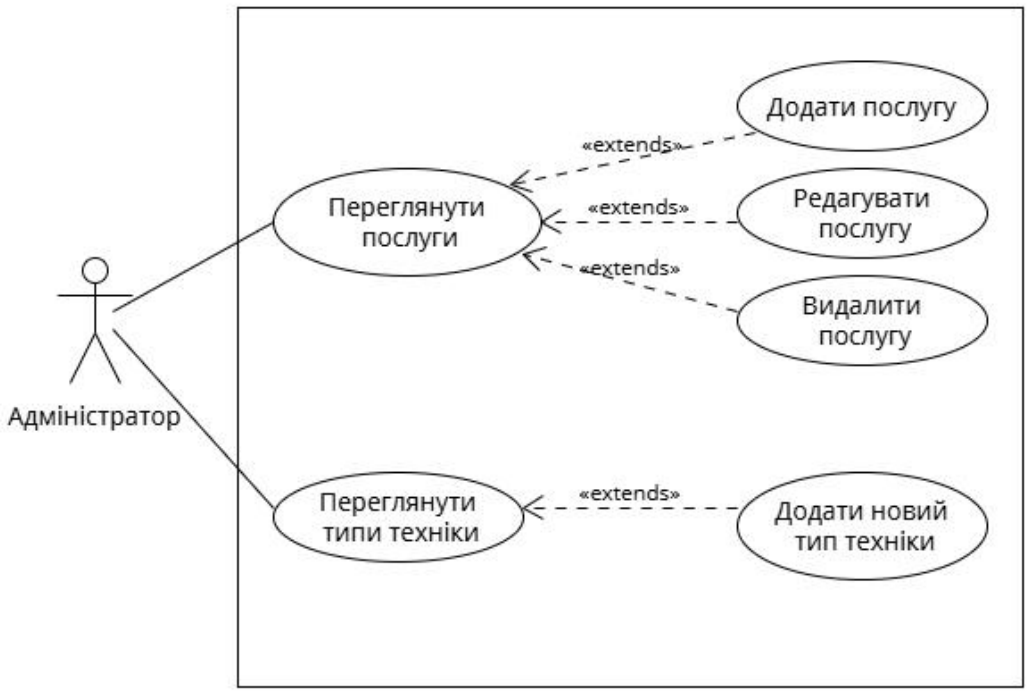


Рис. 3.3 Діаграма варіантів використання для адміністратора

3.2 Діаграма компонентів

Діаграма компонентів відображає архітектурну структуру веб-платформи сервісного центру ремонту побутової техніки та взаємодію між основними програмними модулями системи. Архітектура побудована за багатошаровим підходом, що дозволяє розділити бізнес-логіку, інтерфейс користувача, доступ до даних та тестування на окремі незалежні компоненти. Основними шарами системи є Presentation Layer, Application Layer, Domain Layer, Infrastructure Layer та Testing Layer.

Шар Presentation Layer відповідає за взаємодію користувача із системою та формування інтерфейсу веб-застосунку. До його складу входять компоненти Controllers, Views та View Models.

Компонент Controllers приймає HTTP-запити від користувачів, викликає необхідну бізнес-логіку Application Layer та повертає результати у вигляді представлень. Контролери створюють View Models, які використовуються для передачі даних між серверною частиною та інтерфейсом користувача. Компонент Views відповідає за відображення інформації користувачу у вигляді веб-сторінок. Представлення отримують View Models від контролерів та формують кінцевий інтерфейс системи. Компонент View Models використовується для структурування даних, які передаються між контролерами та представленнями. Це дозволяє відокремити моделі інтерфейсу від доменних сутностей системи. Presentation Layer залежить від Application Layer, оскільки саме через нього виконується доступ до бізнес-логіки та сценаріїв роботи системи.

Application Layer реалізує прикладну логіку системи та координує виконання бізнес-сценаріїв. Даний шар містить компоненти UseCases, Commands, Queries, DTO, Validators та Mappers. Компонент UseCases реалізує основні сценарії роботи системи, наприклад створення заявки, зміна статусу ремонту або призначення майстра. UseCases виступають центральною точкою координації між Presentation Layer, Domain Layer та Infrastructure Layer. Commands використовуються для операцій зміни даних у системі, тоді як Queries

відповідають за отримання інформації без зміни стану даних. Такий підхід дозволяє розділити операції читання та запису. Компонент DTO використовується для передачі даних між шарами системи без прямої залежності від доменних сутностей. Це забезпечує гнучкість архітектури та зменшує зв’язність між компонентами. Validators реалізують перевірку коректності вхідних даних перед виконанням бізнес-операцій. Валідація застосовується до Commands та Queries для запобігання помилковому введенню даних. Mappers відповідають за перетворення даних між DTO, доменними моделями та іншими структурами даних системи. Application Layer залежить від Domain Layer, оскільки використовує доменні сутності та інтерфейси для реалізації бізнес-процесів.

Domain Layer містить основну бізнес-модель системи та реалізує логіку предметної області сервісного центру. До цього шару входять компоненти Entities, Interfaces, Value Objects та Enums. Entities представляють основні сутності системи, наприклад заявку на ремонт, користувача, послугу або тип техніки. Саме в сутностях зберігається ключова бізнес-логіка та стан об’єктів. Value Objects використовуються для представлення допоміжних структур даних, які не мають власної ідентичності та описуються лише значеннями. Enums містять перелік фіксованих значень, які використовуються у системі, наприклад статуси заявок або ролі користувачів. Interfaces визначають контракти взаємодії між Application Layer та Infrastructure Layer. Зокрема, через інтерфейси реалізовується робота репозиторіїв та доступу до даних. Domain Layer є незалежним від інших шарів та не містить прямої залежності від бази даних чи інтерфейсу користувача.

Infrastructure Layer відповідає за технічну реалізацію доступу до даних та взаємодію із зовнішніми ресурсами. До його складу входять компоненти Repositories, Db Contexts, Data Access та Database Initializer. Repositories реалізують інтерфейси, визначені у Domain Layer, та забезпечують роботу з базою даних. Через репозиторії система виконує збереження, оновлення та отримання даних. Db Contexts відповідають за взаємодію із SQL Server та управління сутностями бази даних. Data Access реалізує механізми доступу до даних і забезпечує взаємодію між репозиторіями та контекстом бази даних. Database

Initializer використовується для початкового налаштування бази даних, створення таблиць та ініціалізації стартових даних системи. Infrastructure Layer взаємодіє із SQL Server, який виступає основним сховищем даних веб-платформи.

Testing Layer призначений для тестування функціональності системи та перевірки коректності роботи окремих компонентів. До його складу входять Unit, Integration та E2E тести. Unit тести використовуються для перевірки окремих методів, сервісів та бізнес-логіки без взаємодії із зовнішніми компонентами. Integration тести перевіряють взаємодію між декількома модулями системи, зокрема роботу Application Layer та Infrastructure Layer разом із базою даних. E2E тести імітують повний цикл роботи користувача із системою та перевіряють функціонування веб-застосунку як єдиного цілого. Testing Layer залежить від інших шарів системи, оскільки виконує перевірку їх функціональності та взаємодії.

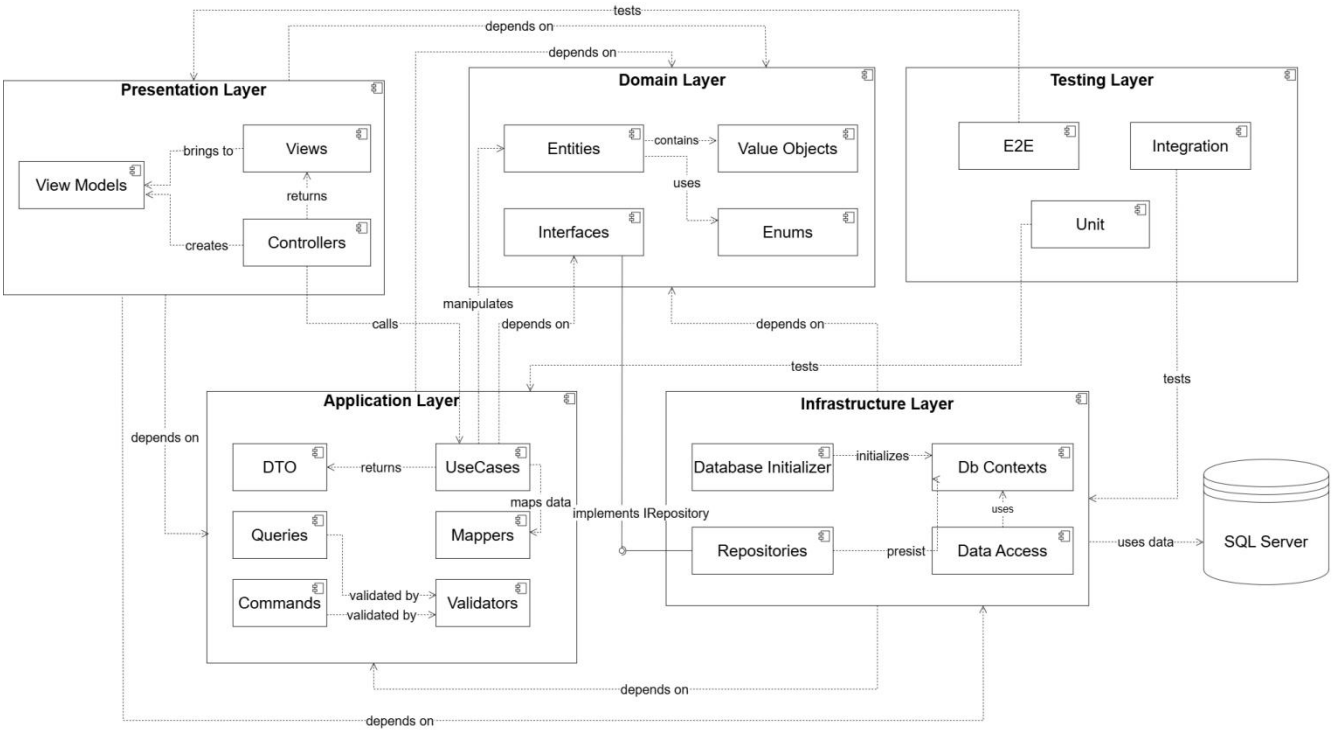


Рис. 3.4 Діаграма компонентів

3.3 Діаграма класів доменного шару

Для формалізації предметної області веб-платформи сервісного центру ремонту побутової техніки була розроблена діаграма класів доменної моделі. Вона відображає основні бізнес-сутності системи, їх атрибути, поведінку та взаємозв'язки між ними. Дана діаграма є ключовою складовою доменного шару, оскільки саме вона визначає структуру бізнес-логіки, а також усі подальші рівні архітектури будуються на основі цієї моделі. У межах доменної моделі визначено основні сутності та їх атрибути.

Сутність User представляє користувача системи незалежно від його статусу. Вона містить основні ідентифікаційні та контактні дані: унікальний ідентифікатор, повне ім'я, email, номер телефону та роль. Роль визначається через перелік UserRole і може набувати значень: клієнт, майстер, консультант або адміністратор.

Ця сутність є базовою для всіх типів користувачів і використовується у ключових процесах: створенні заявок, виконанні робіт, комунікації та отриманні сповіщень. Також передбачено допоміжну операцію перевірки ролі, необхідну для контролю доступу до функціоналу.

Сутність RepairRequest є центральним елементом доменної моделі та виступає агрегатним коренем (Aggregate Root). Вона описує заявку на ремонт побутової техніки й містить усю інформацію для її обробки.

До основних атрибутів належать ідентифікатори клієнта, консультанта й майстра, тип обладнання, опис проблеми, статус, дати створення та закриття, а також загальна вартість ремонту та адреса клієнта (реалізована як value object).

Бізнес-логіка сутності включає методи обчислення вартості заявки, додавання елементів чек-листа та зміни майстра. RepairRequest інкапсулює правила життєвого циклу заявки й забезпечує цілісність даних.

Сутність Service представляє довідник послуг сервісного центру. Вона містить назву послуги та її базову вартість. Дана сутність використовується для формування чек-листа виконаних робіт та розрахунку кінцевої вартості ремонту.

ChecklistItem є елементом чек-листа, який відображає конкретну виконану

послугу в межах заявки на ремонт. Сутність містить посилання на заявку та відповідну послугу, кількість виконаних робіт та ціну. Бізнес-логіка передбачає можливість розрахунку вартості елемента як добутку кількості на ціну послуги.

Сутність ChatMessage використовується для збереження повідомлень у системі чату, який прив’язаний до конкретної заявки. Вона містить текст повідомлення, ідентифікатор відправника та час відправлення. Дана сутність забезпечує реалізацію комунікації між учасниками процесу обслуговування заявки та дозволяє зберігати повну історію взаємодії.

Notification відповідає за систему сповіщень у межах платформи. Вона містить інформацію про отримувача, текст повідомлення, статус прочитання та дату створення. Ця сутність використовується для інформування користувачів про зміни статусу заявки, призначення майстра та інші системні події.

Сутність EquipmentType є довідником типів побутової техніки. Вона містить лише назву типу обладнання та використовується при створенні заявки для класифікації пристрою, що потребує ремонту.

WorkSchedule описує графік роботи користувача системи, зокрема майстрів. Сутність містить інформацію про день тижня, час початку та завершення роботи, а також ознаку робочого дня. Дана модель використовується для планування завантаженості майстрів та автоматичного розподілу заявок.

Address є value object, який описує адресу клієнта. Він містить вулицю, місто та поштовий індекс. Дана сутність не існує самостійно та завжди є частиною RepairRequest, що відповідає принципу композиції в DDD.

Окрім визначення атрибутивного складу об’єктів, важливим є визначення зв’язків між сутностями, яка забезпечується через встановлення логічних зв’язків та типів відношень між представленими класами. У межах доменної моделі встановлено наступні зв’язки між сутностями.

Кожна заявка RepairRequest пов'язана з одним клієнтом (сутність User), може бути призначена одному майстру (зв'язок необов'язковий) та закріплена за консультантом, який відповідає за її супровід.

Заявка містить множинний зв'язок з елементами чек-листа ChecklistItem, що відображають виконані роботи. Зв'язок реалізовано як композицію, оскільки елементи чек-листа не існують поза заявкою. Кожен елемент чек-листа пов'язаний з конкретною послугою Service, що визначає тип виконаної роботи.

Повідомлення чату ChatMessage належать конкретній заявці та мають зв'язок із користувачем-відправником. Сповіщення Notification пов'язані з користувачем, для якого вони створюються, а графік роботи WorkSchedule — з конкретним користувачем системи.

Адреса клієнта Address реалізована як вбудований об'єкт у межах RepairRequest і не існує як окрема сутність, що відповідає принципу value object.

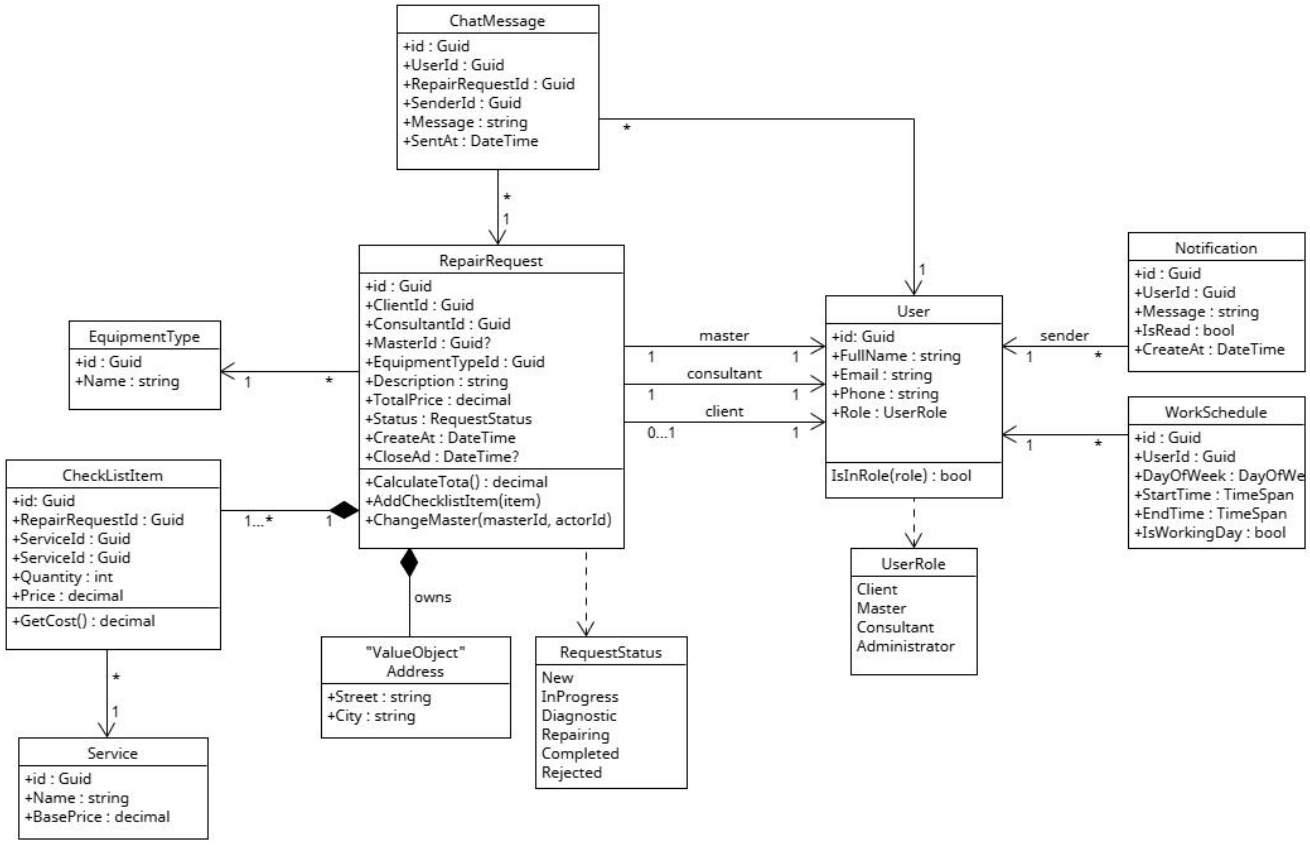


Рис. 3.5 Діаграма класів доменного шару

Отримана доменна модель відображає ключові бізнес-сутності сервісного центру та їх взаємодію.

3.4 ER-діаграма бази даних

Для проєктування структури зберігання даних веб-платформи сервісного центру було розроблено ER-діаграму бази даних. Вона побудована за принципами реляційного моделювання і формалізує логічну структуру даних для забезпечення їхньої цілісності.

Діаграма відображає основні сутності, їхні атрибути та зв'язки, необхідні для реалізації бізнес-процесів центру. Вона організовує централізоване зберігання інформації про користувачів, заявки на ремонт, виконані послуги (чек-листи), комунікацію учасників (повідомлення), графіки роботи персоналу та систему сповіщень.

Центральною сутністю є таблиця Users, яка зберігає інформацію про всіх користувачів платформи. Вона містить первинний ключ Id (тип uniqueidentifier), а також поля FullName, Email, Phone та Role. Поле Role визначає роль у системі згідно з переліком UserRole (клієнт, консультант, майстер або адміністратор). Сутність Users є базовою, адже з нею пов’язані заявки, повідомлення, сповіщення та графіки роботи.

Сутність RepairRequests призначена для зберігання інформації про заявки на ремонт. Кожна заявка має унікальний Id та набір зовнішніх ключів: ClientId, ConsultantId та MasterId (необов'язкове) для зв'язку з користувачами, а також EquipmentTypeId — для зв'язку з довідником типів обладнання.

Структура також містить опис несправності (Description), адресу клієнта (ClientAddress_Street, ClientAddress_City, ClientAddress_ZipCode), загальну вартість (TotalPrice) та поточний стан заявки (Status відповідно до RequestStatus). Поля CreatedAt та ClosedAt фіксують час створення й завершення ремонту для аналітики та аудиту. Сутність EquipmentTypes реалізує довідник типів обладнання та містить поля Id і Name. Використання окремої таблиці дозволяє уникнути дублювання даних і забезпечити класифікацію техніки.

Сутність Services призначена для зберігання довідника сервісних послуг. Таблиця містить первинний ключ Id, назву послуги Name та базову вартість

BasePrice. Цей довідник використовується під час формування чек-листа ремонтної заявки.

Сутність ChecklistItems реалізує структуру чек-листа заявки та виступає проміжною таблицею між заявками та послугами. Кожен запис містить зовнішні ключі RepairRequestId і ServiceId, а також поля Quantity та Price. Це дозволяє гнучко формувати вартість ремонту та підтримувати зв’язок «один-до-багатьох» між заявкою та набором послуг.

Сутність ChatMessages забезпечує реалізацію внутрішньої системи комунікації між учасниками процесу ремонту. Повідомлення пов’язується з конкретною заявкою через RepairRequestId та з користувачем через SenderId. Поле Message містить текст повідомлення, а SentAt — дату та час його відправлення. Наявність окремої сутності повідомлень дозволяє зберігати історію комунікації та підтримувати роботу в реальному часі через SignalR або WebSocket.

Сутність Notifications використовується для реалізації механізму сповіщень у системі. Кожне сповіщення містить зовнішній ключ UserId, текст повідомлення Message, ознаку прочитання IsRead та дату створення CreatedAt. Це забезпечує підтримку асинхронної взаємодії між системою та користувачами.

Сутність WorkSchedules реалізує механізм управління графіками роботи працівників сервісного центру. Таблиця містить зовнішній ключ UserId, поля DayOfWeek, StartTime, EndTime та IsWorkingDay. Наявність цієї сутності забезпечує можливість планування навантаження працівників і контролю доступності майстрів.

Між сутностями ER-діаграми реалізовано декілька типів зв’язків, що забезпечують цілісність і логічну структуру даних. Сутність Users має множинні зв’язки із RepairRequests, ChatMessages, Notifications та WorkSchedules. Сутність RepairRequests пов’язана з ChatMessages та ChecklistItems зв’язком «один-до-багатьох». Сутність Services також має зв’язок «один-до-багатьох» із ChecklistItems, а EquipmentTypes пов’язана із RepairRequests для забезпечення класифікації техніки

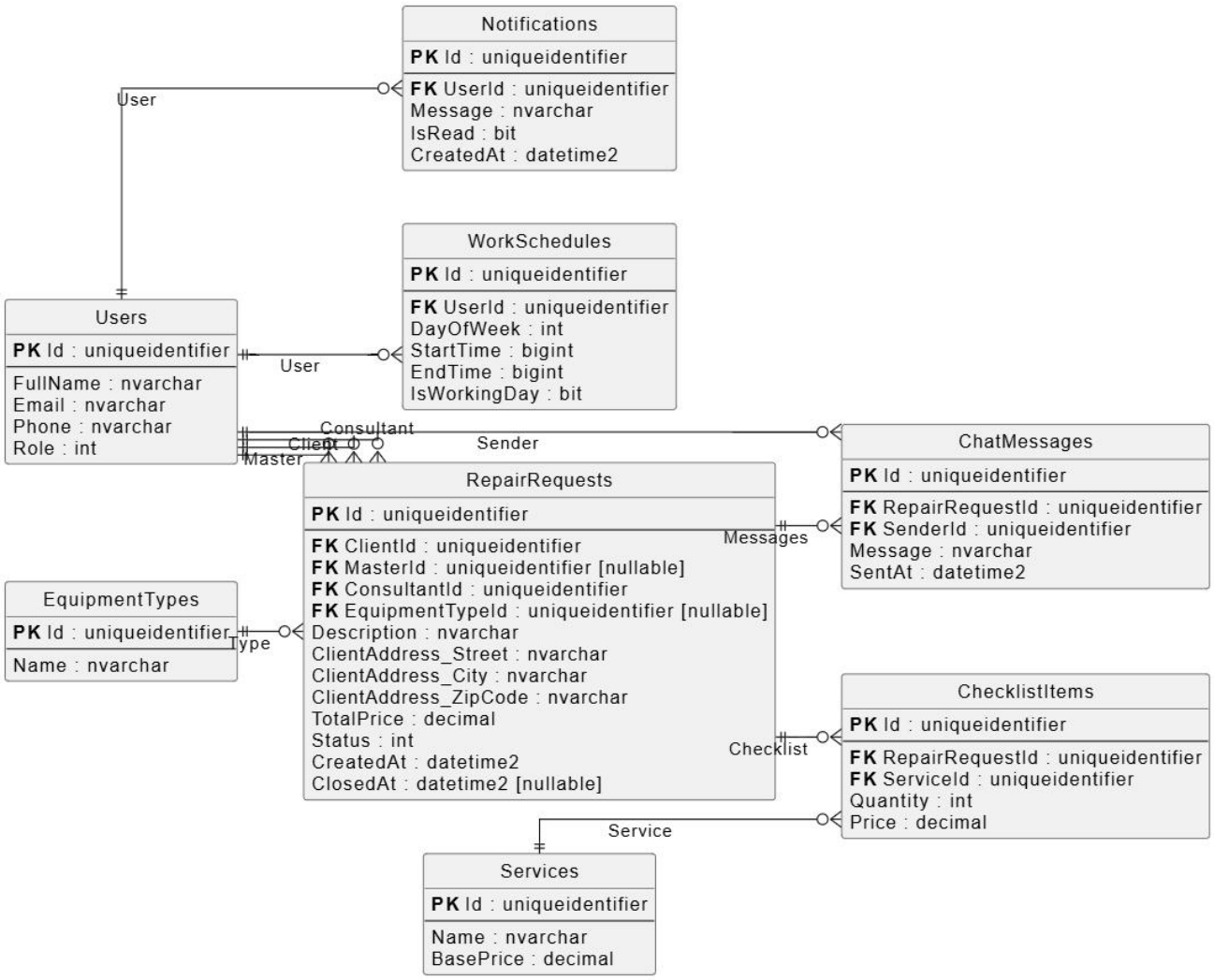


Рис. 3.6 ER-діаграма бази даних

Побудована структура бази даних забезпечує високий рівень нормалізації, мінімізує дублювання даних і підтримує масштабованість системи. Використання GUID-ідентифікаторів забезпечує унікальність записів та спрощує інтеграцію з розподіленими сервісами.

3.5 ER-діаграма системних таблиць авторизації

У веб-платформі реалізовано механізми автентифікації та авторизації на основі ASP.NET Identity , де вся взаємодія з користувачами базується на cookie-based підході. Основним класом користувача виступає ApplicationUser, який наслідує стандартний IdentityUser та розширений додатковим полем для зв’язку з

доменною моделлю через DomainUserId. Це дозволяє розділити технічні дані облікового запису та бізнес-інформацію користувача, яка зберігається у доменному шарі. У базі даних цей клас представлений у стандартній таблиці AspNetUsers, яка разом з іншими таблицями Identity забезпечує зберігання інформації про користувачів, ролі та їх зв’язки.

Для роботи з автентифікацією використовується окремий контекст AuthDbContext, який наслідує IdentityDbContext і відповідає за доступ до всіх таблиць ASP.NET Identity. У межах інфраструктурного шару налаштовані міграції для цих таблиць, що дозволяє керувати структурою користувачів, ролей та токенів відновлення пароля. Створення та управління користувачами здійснюється через сервіс AuthService, який використовує UserManager та SignInManager.

Під час реєстрації система створює нового користувача Identity, виконує хешування пароля та паралельно створює відповідну доменну сутність User, встановлюючи між ними зв’язок через DomainUserId. Під час входу користувача перевіряються облікові дані, і у разі успішної автентифікації формується cookie, яка використовується для подальшої ідентифікації користувача у системі.

Ролі користувачів у системі реалізовані через стандартний механізм IdentityRole, а їх початкова ініціалізація виконується під час запуску веб-платформи через RoleInitializer або DatabaseInitializer, де створюються основні ролі, такі як Client, Master, Consultant та Admin.

Реєстрація Identity-сервісів відбувається у Program.cs шляхом підключення AddIdentity та налаштування Entity Framework store, після чого в middleware-пайплайн додаються компоненти UseAuthentication та UseAuthorization, які забезпечують перевірку користувача та його прав доступу для кожного HTTP-запиту.

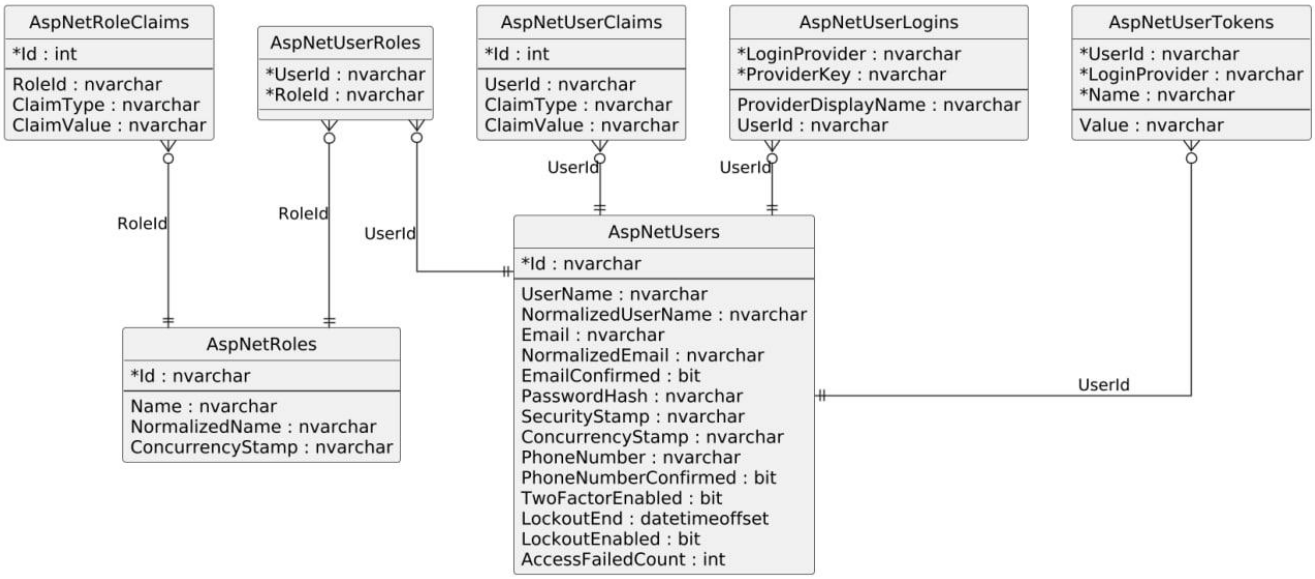


Рис. 3.7 Даграма таблиць авторизації та автентифікації

Авторизація у системі базується на ролях та політиках доступу, які застосовуються безпосередньо у контролерах або на рівні Use Cases через атрибути Authorize.

Після входу користувача його дані зберігаються у ClaimsPrincipal, що дозволяє отримувати ідентифікатор користувача та його ролі у межах виконання запитів і використовувати ці дані для перевірки бізнес-правил.

Уся взаємодія відбувається за класичним сценарієм cookie-аутентифікації: браузер надсилає запит разом із cookie, middleware автентифікує користувача, встановлює його контекст, після чого система перевіряє права доступу та дозволяє або забороняє виконання відповідних дій.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Шар представлення Presentation Layer

Presentation Layer є верхнім шаром ServiceCenterApp та відповідає за взаємодію користувача з веб-платформою MVC. До нього належать контролери HomeController і RepairRequestController, Razor Views (Index.cshtml, Create.cshtml), RepairRequestViewModel, шаблон _Layout.cshtml та статичні ресурси в папці wwwroot. Через цей шар користувач переходить на головну сторінку сервісного центру, створює нові заявки на ремонт обладнання та переглядає доступні послуги [1], [3].

Реалізація Presentation Layer розпочинається з налаштування маршрутизації в Program.cs, де визначаються основні маршрути для контролерів. HomeController приймає GET-запит на маршруті "/" та повертає View Index.cshtml, завантажуючи список типів обладнання з AppDbContext у ViewBag для відображення користувачеві. Це передбачає асинхронні операції доступу до бази даних, що є типовим підодом у ASP.NET Core застосунках [1].

RepairRequestController реалізує два методи для обробки заявок на ремонт. GET-метод Create() завантажує типи обладнання з бази даних та передає їх у RepairRequestViewModel для відображення у dropdown-списку форми. POST-метод Create() реалізує повний цикл обробки подання форми: перевірку ModelState, мапінг даних у CreateRepairRequestCommand та передачу команди в UseCase через await _useCase.Execute(command). Такий підхід відповідає принципам Clean Architecture та розділення відповідальностей [4], [6].

Для передачі даних між представленням та контролерами використовується RepairRequestViewModel, що містить поля форми Description, EquipmentTypeId, ServiceFormat, Street, City, ZipCode та колекцію EquipmentTypes для dropdown-списку. Поля мають атрибут [Required], що забезпечує базову серверну валідацію даних відповідно до підходів ASP.NET Core [3].

ViewBag використовується для передачі довідникових даних (типи обладнання, послуги), які не змінюються в межах одного запиту. Інтерфейс реалізовано через Razor Views: _Layout.cshtml містить загальний шаблон сторінок, Index.cshtml — головну сторінку системи, а Create.cshtml — форму створення заявки з підтримкою завантаження файлів.

Шар Presentation також містить статичні ресурси site.css та site.js, які підключаються через middleware app.UseStaticFiles(). Це стандартний механізм роботи зі статичними файлами в ASP.NET Core [1].

Залежність між шарами реалізується через Application Layer, куди передаються команди на виконання бізнес-логіки (CreateRepairRequest). Взаємодія між шарами відповідає принципам багатошарової архітектури та Dependency Injection [4].

4.2 Шар застосування Application Layer

Application Layer є середнім шаром системи ServiceCenterApp, який відповідає за реалізацію бізнес-логіки та сценаріїв використання системи.

Реалізація даного шару базується на архітектурному підході CQRS (Command Query Responsibility Segregation), який передбачає чітке розділення операцій запису (Commands) та читання (Queries). Такий підхід дозволяє ізолювати логіку зміни стану системи від логіки отримання даних, що підвищує зрозумілість коду та спрощує масштабування системи [5], [18].

Основним компонентом Application Layer є UseCase класи, які інкапсулюють окремі бізнес-сценарії системи. Кожен UseCase реалізує принцип єдиної відповідальності (Single Responsibility Principle) і відповідає лише за одну конкретну операцію. Типова реалізація UseCase включає отримання Command або Query як вхідного параметра методу Execute, виконання валідації, взаємодію з репозиторіями через інтерфейси та повернення результату виконання.

UseCase CreateRepairRequest реалізує сценарій створення нової заявки на ремонт. Він приймає відповідну команду, виконує перевірку існування клієнта

через репозиторій, створює доменну сутність RepairRequest зі статусом New, формує Value Object Address на основі вхідних даних, прив’язує заявку до клієнта та виконує збереження через _repository.AddAsync(request). У результаті повертається ідентифікатор створеної заявки (Guid), який використовується на рівні Presentation Layer. Додатково враховуються крайні випадки, зокрема відсутність користувача або некоректні дані адреси [4], [6].

UseCase GetClientRequests реалізує отримання всіх заявок конкретного клієнта. Він приймає ідентифікатор користувача, виконує фільтрацію заявок через репозиторій за ClientId, перевіряє права доступу (клієнт може отримувати тільки власні заявки), після чого виконує мапінг доменних сутностей у DTO через _mapper.ToDto(). Для оптимізації передбачено підтримку пагінації при великій кількості записів [5], [9].

UseCase AssignMaster відповідає за складну бізнес-логіку призначення майстра до заявки. У межах виконання перевіряється доступність майстра, відповідність його спеціалізації типу техніки, а також поточне навантаження. Після успішної перевірки система оновлює заявку та змінює її статус. Реалізація потребує взаємодії з кількома репозиторіями та узгодження даних між різними доменними сутностями [6], [8].

Для взаємодії між Presentation Layer та UseCase використовуються Command та Query класи. Commands відповідають за операції зміни стану системи (CreateRepairRequest, UpdateRequestStatus, CancelRepairRequest), а Queries — за отримання даних без зміни стану (GetClientRequests, GetRequestDetails, GetMasterWorkload). Кожна Command містить повний набір даних, необхідних для виконання операції, що дозволяє ізолювати бізнес-логіку від зовнішніх залежностей [5], [18].

Queries реалізуються з акцентом на оптимізоване читання даних. Вони використовують спеціалізовані методи репозиторіїв та дозволяють уникати завантаження зайвих пов’язаних сутностей. Розділення Commands та Queries забезпечує можливість окремої оптимізації операцій: для Commands

застосовується транзакційна обробка, а для Queries — кешування та оптимізовані LINQ-запити [9].

Валідація вхідних даних реалізується через Validator класи, які імплементують IValidator<T>. Вони перевіряють бізнес-правила перед виконанням UseCase. Наприклад, CreateRepairRequestValidator перевіряє коректність текстових полів, існування типу обладнання та допустимість створення заявки. Це дозволяє уникнути виконання операцій з некоректними даними ще до рівня доменної логіки [18].

Рівні валідації розділені логічно: базова перевірка типів даних виконується на рівні DTO через DataAnnotations у Presentation Layer, бізнес-правила перевіряються у Validators Application Layer, а доменні обмеження реалізуються у Domain Layer. Такий підхід забезпечує багаторівневий захист цілісності даних [4], [19].

Validators викликаються на початку виконання UseCase через метод Validate(command). У випадку помилок формується колекція ValidationResult, яка передається назад у Presentation Layer для відображення користувачу. Це дозволяє уникнути часткового виконання операцій та зберігає консистентність системи [18].

Мапінг між доменними сутностями та DTO реалізується через Mapper класи, які реалізують інтерфейс IMapper<TDomain, TDto>. Наприклад, RepairRequestMapper перетворює доменну сутність у DTO для передачі на клієнт, приховуючи внутрішні поля системи. Це включає трансформацію вкладених об’єктів, таких як Address (Value Object), список послуг та користувачів [5], [6].

DTO класи містять лише необхідні для клієнтської частини дані та ізолюють внутрішню структуру домену. Наприклад, RepairRequestDto містить основні параметри заявки, інформацію про клієнта, майстра та список послуг, але не містить технічних деталей збереження у базі даних. Це забезпечує стабільність API при змінах внутрішньої архітектури системи [5].

Загальна архітектура Application Layer побудована на принципі розділення відповідальностей: UseCase керують бізнес-сценаріями, Commands та Queries виражають намір операцій, Validators забезпечують перевірку даних, а Mappers

виконують трансформацію структур. Взаємодія з іншими шарами здійснюється через інтерфейси репозиторіїв, що забезпечує слабку зв’язаність системи [4], [6].

Реєстрація залежностей виконується у методі DependencyInjection.AddApplication(), де всі UseCase, Validators та Mappers додаються як Scoped сервіси. Це дозволяє ізолювати конфігурацію DI від Program.cs та підвищує модульність системи [3].

Обробка помилок у UseCase передбачає перехоплення винятків, які виникають під час роботи з репозиторіями (відсутній користувач, некоректні дані тощо). Ці помилки трансформуються у зрозумілі повідомлення для Presentation Layer, яка у свою чергу повертає відповідні HTTP-коди (400, 404) [1], [18].

4.3 Доменний шар Domain Layer

Domain Layer є ядром ServiceCenterApp, яке містить бізнес-логіку та моделі, незалежні від фреймворків і баз даних.

Реалізація Domain Layer передбачає розробку чистих моделей, які представляють поняття предметної області без залежностей від технічних деталей персистенції чи фреймворків. Кожен клас у Domain Layer фокусується на інкапсуляції даних та логіки, пов'язаної з цим поняттям, формуючи мікро-масштабні моделі зі своїми правилами та обмеженнями [4].

Сутності домену представляють основні об'єкти системи з набором властивостей та методів, які відображають реальні процеси ремонтного центру. Центральною сутністю є RepairRequest, яка інкапсулює весь життєвий цикл заявки: утримує інформацію про клієнта (Client), майстра (Master), консультанта (Consultant), тип техніки (EquipmentType), опис проблеми (Description), адресу (Address), список послуг (Services), загальну вартість (TotalPrice), статус (Status) та дати створення (CreatedAt) і завершення (CompletedAt) заявки.

Реалізація RepairRequest включає методи для зміни стану заявки у відповідності до визначених правил переходу (заявка переходить зі стану New до Assigned, потім до InProgress, Completed або Cancelled). Ці методи реалізують

бізнес-правила на рівні домену, гарантуючи, що невалідні переходи не можуть статися. Наприклад, неможливо перевести заявку до Completed, якщо вона ще не призначена майстру.

User зберігає основні дані користувача (Name, Email, Phone), його роль (Role — Client, Master, Consultant або Administrator), дату створення та інші дані профілю. Реалізація User передбачає методи для перевірки належності користувача до групи (IsClient, IsMaster) та отримання прав на виконання операцій.

Service описує послугу, яку надає центр (Name — наприклад, "Заміна компресора"), та її базову вартість (BasePrice). Реалізація дозволяє системі розраховувати фактичну вартість залежно від типу обладнання та складності роботи через коефіцієнти [5].

EquipmentType використовується для категоризації техніки (Refrigerator, WashingMachine, MicrowaveOven) та дозволяє зв’язувати послуги з типами обладнання, які їх підтримують. Реалізація включає переліки підтримуваних послуг для кожного типу.

ChecklistItem пов'язує послуги із заявкою та зберігає відомості про їх виконання (ServiceId, RepairRequestId, IsCompleted, CompletedAt). Це дозволяє відслідковувати хід виконання комплексних заявок, які складаються з декількох послуг.

ChatMessage відповідає за повідомлення в чаті між клієнтом та майстром (SenderId, ReceiverId, Content, CreatedAt), що забезпечує комунікацію учасників без необхідності виходити із системи. Реалізація передбачає відзначення прочитаних повідомлень для організації діалогу.

WorkSchedule задає графік роботи майстра (MasterId, DayOfWeek, StartTime, EndTime) та дозволяє системі перевіряти доступність майстра перед призначенням. Реалізація включає методи для отримання вільних часових інтервалів на день.

Notification використовується для сповіщень користувачів про важливі події (UserId, Title, Message, Type, IsRead). Реалізація підтримує різні типи сповіщень

(RequestCreated, RequestAssigned, RequestCompleted) для різних категорій користувачів.

Типи Enum визначають допустимі значення доменних понять та забезпечують строгу типізацію замість рядків чи чисел. UserRole містить ролі Client, Master, Consultant та Administrator з відповідними рівнями дозволів. RequestStatus описує усі можливі стани заявки (New, Assigned, InProgress, Completed, Cancelled), представляючи її життєвий цикл. WorkRole використовується для керування доступом до функцій системи та привілеїв.

Реалізація Enums дозволяє компілятору перевіряти коректність значень на етапі розробки, запобігаючи помилкам.

Value Object Address інкапсулює адресу клієнта, складаючись з трьох властивостей (Street, City, ZipCode), без окремого Id. На відміну від сутностей, які мають унікальний ідентифікатор, Value Objects ідентифікуються за своїм змістом. Для Value Object реалізовано методи Equals() та GetHashCode(), що забезпечує порівняння за значенням, а не за посиланням. Два об'єкти Address вважаються рівними, якщо мають однакові місто, вулицю та поштовий індекс, незалежно від того, що це різні об'єкти в пам’яті [6].

Інтерфейси репозиторіїв визначають контракти доступу до даних без прив'язки до конкретної реалізації з використанням Entity Framework або іншої технології персистенції. Базовий IRepository<T> містить CRUD-операції (Create, Read, Update, Delete), дозволяючи сутностям взаємодіяти з даними без знання про спосіб збереження.

IRequestRepository розширює базовий репозиторій методами отримання заявок за конкретними критеріями. IUserRepository забезпечує пошук користувача за Email та роллю, що необхідно для аутентифікації та розділення доступу. Інші інтерфейси визначають операції для відповідних сутностей.

Реалізація інтерфейсів у Infrastructure Layer забезпечує те, що Domain Layer залишається незалежним від деталей реалізації та дозволяє легко замінити реалізацію при тестуванні за допомогою mock-об'єктів.

Архітектура шару побудована на принципах Domain-Driven Design та інверсії залежностей. Бізнес-логіка зосереджена в доменних класах, які не мають залежностей від фреймворків, забезпечуючи портативність та чистоту коду. Enums забезпечують строгу типізацію доменних понять, а інтерфейси репозиторіїв дозволяють шарам Application та Infrastructure залежати від Domain Layer, а не навпаки, що відповідає принципам Clean Architecture [4], [6], [9].

Реалізація Domain Layer дозволяє тестувати бізнес-правила без залежностей від бази даних, оскільки всі бізнес-операції залишаються у коді, який не взаємодіє з інфраструктурою.Отже, Domain Layer відповідає за чисті бізнес-моделі, правила взаємозв'язків та контракти доступу, гарантуючи повну незалежність від деталей реалізації та забезпечуючи ясність бізнес-процесів.

4.4 Шар інфраструктури Infrastructure Layer

Infrastructure Layer реалізує доступ до даних та взаємодію із зовнішніми сервісами, забезпечуючи виконання контрактів, визначених у Domain Layer. Шар побудований на основі Entity Framework Core та MS SQL Server, що забезпечує масштабованість, надійність і чітке розділення відповідальностей відповідно до архітектурних принципів сучасних enterprise-застосунків [18], [19]. Реалізація передбачає трансформацію доменних моделей у реляційну схему бази даних та управління життєвим циклом об’єктів у контексті бази даних.

AppDbContext є центральним ORM-компонентом системи та розширює DbContext. Він містить DbSet для всіх основних сутностей системи: Users, RepairRequests, Services, ChatMessages, ChecklistItems, Notifications, EquipmentTypes та WorkSchedules. Реалізація AppDbContext передбачає визначення всіх пов’язаних сутностей та управління їх відносинами відповідно до підходів ORM-моделювання [17].

У методі OnModelCreating(ModelBuilder) налаштовуються детальні конфігурації зв’язків між сутностями та конверсії типів для специфічних випадків персистенції. Value Object ClientAddress вбудовується в таблицю RepairRequests

через OwnsOne(), що означає, що адреса зберігається як вбудовані колонки без окремої таблиці. Навігаційні властивості RepairRequest (Client, Master, Consultant) конфігуруються через HasOne().WithMany(), визначаючи зв'язки «один до багатьох» між сутностями відповідно до патернів проєктування баз даних [5].

Видалення з обмеженнями (Restrict deletion) налаштовується для запобігання видалення користувача, якщо у нього є активні заявки. Для WorkSchedule enum DayOfWeek конвертується з Enum у int для зберігання у базі даних, а TimeSpan зберігається через Ticks (кількість 100-наносекундних інтервалів), оскільки SQL Server не має вбудованого типу для часу.

Repository<T> є базовим генеричним репозиторієм, який реалізує IRepository<T> та надає CRUD-операції: AddAsync додає нову сутність до контексту та зберігає її, DeleteAsync видаляє сутність, GetByIdAsync отримує сутність за Id, GetAllAsync отримує всі сутності, UpdateAsync оновлює існуючу сутність. Реалізація використовує асинхронні операції для запобігання блокуванню потоку при доступі до бази даних та підвищення продуктивності системи [18].

RequestRepository реалізує IRequestRepository та містить методи отримання заявок за ClientId, MasterId та статусом заявки. Реалізація включає Eager Loading пов'язаних сутностей (Include().ThenInclude()) для завантаження Client, Master, Consultant та Services в одному запиті, що зменшує кількість звернень до БД. GetByClientIdAsync з пагінацією дозволяє отримувати заявки порціями для великих наборів даних.

UserRepository реалізує IUserRepository і додає пошук користувача за Email для аутентифікації та пошук за роллю для отримання списків користувачів певної категорії (майстрів, консультантів). FirstOrDefaultAsync використовується для запитів, які повинні повернути один запис. Така реалізація відповідає принципам побудови шаруватої архітектури та інкапсуляції доступу до даних [5].

ServiceRepository використовує лише базову CRUD-функціональність без додаткових методів для фільтрації.

WorkScheduleRepository окремо реалізує IWorkScheduleRepository і забезпечує роботу з графіками майстрів, включаючи методи для отримання графіка майстра на конкретний день або період. NotificationRepository надає методи отримання сповіщень користувача (включаючи непрочитані), підрахунку непрочитаних сповіщень та позначення як прочитаних.

AppDbContextFactory реалізує IDesignTimeDbContextFactory<AppDbContext> та використовується для підтримки EF Core міграцій під час розробки. Коли розробник запускає dotnet ef migrations add, фреймворк використовує цей factory для створення контексту без залежності від Program.cs, що дозволяє автоматизувати створення SQL-скриптів для змін схеми.

DatabaseInitializer виконує автоматичне створення таблиць на основі моделей DbSet при запуску застосунку та початкове наповнення бази даних тестовими записами для Services та EquipmentTypes. Це гарантує, що базова конфігурація сервісу присутня з моменту запуску без необхідності ручного створення.

DependencyInjection.AddInfrastructure() реєструє DbContext, репозиторії та сервіси через IServiceCollection у Program.cs. DbContext реєструється як Scoped-сервіс, що означає, що новий контекст створюється для кожного HTTP-запиту, забезпечуючи ізоляцію даних між запитами. Репозиторії також підключаються як Scoped-сервіси, що дозволяє їм спільно використовувати один DbContext у межах одного запиту.

DatabaseInitializer використовується як Hosted Service, що означає, що він запускається під час старту застосунку та виконує ініціалізацію бази даних.

Infrastructure Layer залежить від Entity Framework Core як ORM, SQL Server Provider для взаємодії з MS SQL Server, BCrypt.Net-Next для хешування паролів користувачів та Microsoft.Extensions.Configuration для зчитування параметрів підключення до БД з appsettings.json. Такий набір технологій забезпечує стабільну роботу системи та безпечне зберігання даних [1], [2].

Реалізація передбачає спеціальні оптимізації для запобігання проблемам N+1: при завантаженні списку заявок з клієнтами використовується Include, який завантажує пов’язані сутності в одному запиті замість множинних звернень до бази даних. Indexed запити на часто фільтровані поля (ClientId, MasterId, Status) забезпечують швидкий доступ при великих обсягах даних, що відповідає практикам проєктування продуктивних систем [9].

Реалізація асинхронних операцій дозволяє серверу обслуговувати більше одночасних запитів, оскільки потоки не блокуються при очікуванні відповіді від бази даних.

Отже, Infrastructure Layer відповідає за персистенцію, конкретну реалізацію репозиторіїв на основі Entity Framework Core, управління доступом до MS SQL Server та ініціалізацію схеми бази даних, гарантуючи, що Domain Layer залишається незалежним від фреймворку й конкретної БД. Реалізація забезпечує оптимізацію запитів, асинхронні операції та чистоту персистентного коду.

4.5 Шар тестування Test Layer

Test Layer є набором автоматизованих тестів, що забезпечують якість та стабільність роботи ServiceCenterApp через одиничне, інтеграційне та функціональне тестування всіх шарів архітектури. Загальний підхід до побудови тестового середовища відповідає сучасним принципам розробки програмного забезпечення та практикам забезпечення якості, описаним у підходах до багатошарової архітектури та тестування програмних систем [18], [19].

Для реалізації тестування використовуються xUnit як основний фреймворк для запуску тестів, FluentAssertions для написання більш читабельних та зрозумілих перевірок, NSubstitute для створення mock-об'єктів залежностей, AutoFixture для автоматичного генерування тестових даних та InMemory Database для ізольованого тестування Repository без використання реальної бази даних.

Реалізація Test Layer базується на принципі "Test Pyramid", відповідно до якого більшість тестів становлять unit-тести, менша частка припадає на

інтеграційні тести, а найменша — на функціональні або end-to-end тести. Такий підхід дозволяє досягти оптимального балансу між швидкістю виконання тестів та рівнем покриття бізнес-логіки, що є рекомендованою практикою в інженерії програмного забезпечення [18], [19].

Тести структуровані у декількох модулях відповідно до їх призначення. Модуль DataAccess.MsSql містить інтеграційні тести роботи з базою даних, які можуть виконуватися як на реальному MS SQL Server, так і з використанням In-Memory Database для пришвидшеного тестування. У межах цього модуля перевіряються CRUD-операції (створення, отримання, оновлення та видалення записів), коректність LINQ-запитів при фільтрації даних за різними критеріями, а також правильність конфігурації контексту та маппінгу пов’язаних сутностей [16], [17]. Реалізація тестів DataAccess передбачає використання In-Memory Database для забезпечення швидкого виконання без залежності від зовнішньої інфраструктури. Кожен тест створює окремий контекст виконання, що виключає взаємний вплив тестових сценаріїв та забезпечує ізоляцію даних.

Модуль UseCases охоплює unit-тестування бізнес-логіки із використанням мокування залежностей через NSubstitute. Замість реальних репозиторіїв використовуються mock-об'єкти, які повертають заздалегідь визначені дані. Наприклад, сценарій CreateRepairRequest перевіряється на коректне створення нової заявки зі статусом New та правильним заповненням полів, AssignMaster перевіряється на правильність призначення майстра та оновлення статусу заявки, а GetClientRequests перевіряється на повернення коректного списку заявок конкретного клієнта.

Окрему увагу приділено перевірці граничних випадків. Зокрема, спроба створити заявку для неіснуючого клієнта повинна викликати виключення, так само як і спроба призначити недоступного майстра [18]. AutoFixture використовується для автоматичного створення тестових об'єктів із випадковими або напіввипадковими даними, що значно спрощує написання тестів та підвищує їх універсальність. Це дозволяє зменшити дублювання коду та підвищити підтримуваність тестової бази.

Модуль Mappers відповідає за перевірку коректності перетворення даних між доменними сутностями та DTO. Зокрема, перевіряється правильність трансформації Value Objects, пов’язаних сутностей та enum-значень, таких як статус заявки. Mapper-тести також перевіряють двостороннє перетворення даних, де Entity → DTO та DTO → Entity повинні зберігати цілісність інформації без втрат, окрім свідомо виключених полів.

Модуль Validators містить тести перевірки вхідних команд, включаючи обов’язкові поля (наприклад, опис заявки не може бути порожнім), форматні обмеження (наприклад, коректність email), обмеження довжини рядків та бізнес-правила (наприклад, заборона негативної вартості послуг). Реалізація тестів охоплює як позитивні сценарії, так і негативні, де невалідні дані повинні відхилятися із відповідним повідомленням про помилку.

Запуск усіх тестів здійснюється за допомогою команди dotnet test, яка автоматично виконує весь набір тестів та формує звіт про результати виконання. Тести організовані структуровано відповідно до компонентів системи, що полегшує навігацію та підтримку тестового коду (наприклад, TestCreateRepairRequestUseCase, TestRepairRequestRepository).

Окремо передбачено контроль покриття коду (code coverage), який використовується для оцінки якості тестування. Мінімальний рівень покриття встановлено на рівні 80% для всього проєкту та підвищений для критичних компонентів, таких як UseCases та Validators, оскільки саме вони містять ключову бізнес-логіку системи.Отже, Test Layer відповідає за валідацію коректності ServiceCenterApp через автоматизовані тести, запобігання регресіям при змінах коду та документування очікуваної поведінки. Використання xUnit, FluentAssertions, NSubstitute та AutoFixture забезпечує сучасний, простий у прочитанні та підтримці набір тестів, який гарантує надійність і якість всіх шарів архітектури. Реалізація unit-тестів для UseCase та Mapper, інтеграційних тестів для Repository та валідаційних тестів для Validators дозволяє розробникам впевнено робити зміни без страху перед непередбачуваними побічними ефектами.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було досліджено особливості діджиталізації бізнес-процесів сервісних центрів з ремонту побутової техніки та проаналізовано сучасні підходи до автоматизації сервісного обслуговування. Проведено аналіз предметної галузі та визначено основні проблеми функціонування сервісних центрів, серед яких фрагментарність обробки заявок, недостатній контроль життєвого циклу ремонту, відсутність централізованої системи управління та складність координації взаємодії між клієнтами і персоналом.

У роботі виконано аналіз існуючих програмних рішень для сервісних центрів, зокрема RepairDesk, Orderry та RemOnline, що дозволило визначити їх переваги, недоліки та сформувати функціональні й нефункціональні вимоги до майбутньої веб-платформи. На основі проведеного аналізу визначено об’єкт, предмет та мету дослідження.

У процесі роботи обґрунтовано вибір технологічного стеку та засобів розробки програмного забезпечення. Для реалізації системи використано платформу .NET, мову програмування C#, ASP.NET Core MVC, Entity Framework Core та Microsoft SQL Server. Архітектуру програмного забезпечення побудовано з використанням принципів Clean Architecture та CQRS, що забезпечує масштабованість, модульність і підтримуваність системи.

У межах роботи спроєктовано структуру веб-платформи, розроблено ER-модель бази даних, UML-діаграми та основні функціональні модулі системи відповідно до ролей користувачів і сценаріїв життєвого циклу заявки. Реалізовано механізми автентифікації та авторизації користувачів, систему управління заявками, внутрішню комунікацію між учасниками процесу та підтримку контролю виконання ремонтних робіт.

Результати роботи можуть бути використані для автоматизації діяльності сервісних центрів побутової техніки, оптимізації процесів обробки заявок,

підвищення ефективності роботи персоналу та покращення взаємодії з клієнтами.

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

1. Циганок А. М., Ярошевський О.В. Інструменти та методи взаємодії з базою даних у веб-розробці. Матеріали Всеукраїнської науково-технічної конференції "Виклики та рішення в програмній інженерії" , 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.526-528.

2. Циганок А. М., Ярошевський О.В. Автоматизаця процесу обробки заявок у сервісному центрі з ремонту побутової техніки. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. К.: ДУІКТ, 2025. С.324-328.

ПЕРЕЛІК ПОСИЛАНЬ

1. Freeman A. Pro ASP.NET Core 8. 11th ed. New York : Apress, 2024. 1045 p.

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

3. Lock A. ASP.NET Core in Action. 3rd ed. Shelter Island : Manning Publications, 2023. 950 p.

4. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Boston : Pearson Education, 2018. 432 p.

5. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2019. 560 p.

6. Vernon V. Implementing Domain-Driven Design. Boston : Addison-Wesley, 2019. 656 p.

7. Richardson C. Microservices Patterns. Shelter Island : Manning Publications, 2018. 520 p.

8. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston : Addison-Wesley, 2018. 560 p.

9. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O’Reilly Media, 2017. 616 p.

10. Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma та ін. Boston : Addison-Wesley, 2019. 416 p.

11. ASP.NET Core documentation. Microsoft Corporation. URL: https://learn.microsoft.com/aspnet/core/ (дата звернення: 17.05.2026).

12. Entity Framework Core documentation. Microsoft Corporation. URL: https://learn.microsoft.com/ef/core/ (дата звернення: 17.05.2026).

13. SignalR documentation. Microsoft Corporation. URL: https://learn.microsoft.com/aspnet/core/signalr/ (дата звернення: 17.05.2026).

14. ASP.NET Core Identity documentation. Microsoft Corporation. URL: https://learn.microsoft.com/aspnet/core/security/authentication/identity/ (дата звернення: 17.05.2026).

15. MediatR documentation. URL: https://github.com/jbogard/MediatR (дата звернення: 17.05.2026).

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

17. Silberschatz A., Korth H. F., Sudarshan S. Database System Concepts. 7th ed. New York : McGraw-Hill Education, 2019. 1376 p.

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

19. Sommerville I. Software Engineering. 10th ed. Boston : Pearson, 2020. 816 p.

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

21. Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide. 2nd ed. Boston : Addison-Wesley, 2019. 496 p.

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

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

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка веб-платформи для діджиталізації бізнес-процесів сервісу з ремонту побутової техніки

Виконала студентка 4 курсу
групи ПД-44
Циганок Алєся Миколаївна
Керівник роботи
асист. Ярошевський Олександр Вікторович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи підвищення ефективності роботи сервісного центру побутової техніки шляхом діджиталізації процесів обробки заявок та управління сервісним обслуговуванням.

Об'єкт дослідження бізнес-процеси сервісного центру з ремонту побутової техніки, пов'язані з прийомом, обробкою та виконанням заявок на ремонт, а також взаємодією між клієнтами та персоналом.

Предмет дослідження засоби, методи та підходи до діджиталізації бізнес-процесів сервісного центру, що забезпечують автоматизацію обробки заявок, підтримку життєвого циклу обслуговування та координацію взаємодії між учасниками процесу, а також принципи побудови програмних систем для реалізації зазначених функцій у вигляді веб-платформи.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести огляд та аналіз існуючих систем автоматизації сервісних центрів побутової техніки, визначити їх основні функціональні можливості, переваги та недоліки, а також виявити проблеми сучасних програмних рішень у сфері управління сервісним обслуговуванням.
2. Дослідити бізнес-процеси сервісного центру побутової техніки, проаналізувати процеси прийому та обробки заявок, розподілу навантаження між працівниками, комунікації з клієнтами та контролю виконання ремонтних робіт, а також сформувати функціональні й нефункціональні вимоги до веб-платформи для автоматизації та діджиталізації діяльності сервісного центру.
3. Спроєктувати архітектуру веб-платформи для автоматизації роботи сервісного центру побутової техніки з використанням сучасних архітектурних підходів, визначити структуру системи, взаємодію між її компонентами та механізми реалізації основної бізнес-логіки.
4. Розробити структуру бази даних веб-платформи, спроєктувати рольову модель доступу користувачів, визначити взаємозв'язки між основними сутностями системи, а також реалізувати механізми взаємодії між клієнтами, консультантами, майстрами та адміністраторами.
5. Виконати програмну реалізацію веб-платформи для автоматизації обробки заявок і управління сервісним обслуговуванням, реалізувати функціонал створення та супроводу заявок, внутрішньої комунікації, управління ремонтними роботами та провести тестування працездатності й коректності функціонування системи.

3

АНАЛІЗ АНАЛОГІВ

Критерій	RepairDesk	Orderry	RoApp	ServCenter
Організація обробки заявок	Багатоетапна система зі складною структурою статусів	Стандартна CRM-обробка заявок	Базове ведення заявок	Централізоване керування життєвим циклом заявки
Механізм комунікації	Email та SMS інтеграції	Email та SMS інтеграції	Месенджери та SMS	Вбудований чат у межах системи
Організація ролей користувачів	Базові ролі персоналу	Частковий розподіл доступу	Спрощена рольова модель	Розмежування ролей Client, Consultant, Master, Admin
Розподіл навантаження між майстрами	Переважно ручне призначення	Часткова підтримка керування	Ручне керування заявками	Контроль навантаження та координація виконавців
Налаштування бізнес-процесів	Обмежена адаптація сценаріїв	Часткова кастомізація	Мінімальна гнучкість	Гнучка структура процесів та статусів
Підтримка історії взаємодії	Частково зберігається	Історія заявок без повної комунікації	Базова історія звернень	Повна історія змін та взаємодії
Архітектура доступу до системи	Хмарна SaaS-платформа	Веб-орієнтована CRM/FSM	Веб-застосунок із базовою структурою	Багаторівнева веб-платформа
Адаптація під специфіку підприємства	Складна через перевантаженість	Потребує технічних змін	Обмежена функціонально	Орієнтована на кастомізацію процесів

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Реєстрація, авторизація та керування користувачами з підтримкою ролей.
2. Створення та обробка заявок на ремонт із додаванням опису, адреси, типу обладнання.
3. Призначення майстрів на заявки з урахуванням навантаження та статусу ремонту.
4. Ведення чек-листів, результатів діагностики та історії змін заявки.
5. Внутрішній чат між клієнтом, консультантом і майстром у режимі реального часу.
6. Розрахунок вартості ремонту з урахуванням робіт, матеріалів і послуг.
7. Контроль життєвого циклу заявки та зміни статусів ремонту.
8. Збереження історії взаємодії користувачів і виконаних операцій у системі.

Нефункціональні вимоги:

1. Час обробки запиту ≤ 3 сек
2. Підтримка ≥ 10 одночасних користувачів
3. Час обробки заявки ≤ 5 хв
4. Безпека даних (ролі доступу)
5. Час збереження повідомлення в чаті ≤ 1 сек

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Мова програмування
C#

Backend / Серверна розробка
ASP.NET Core

Робота з базою даних
Microsoft SQL Server
Entity Framework Core

Аутентифікація та авторизація
ASP.NET Identity

Архітектурні патерни та взаємодія компонентів
MediatR

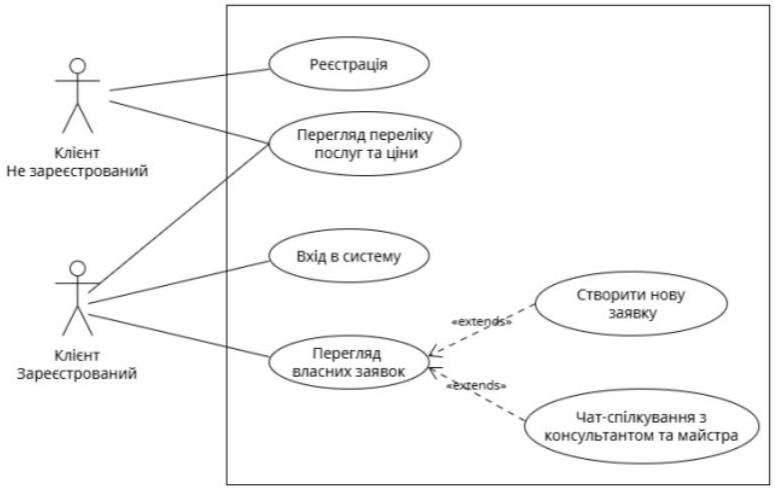
Підтримка внутрішньої комунікації
WebSocket

Валідація даних
FluentValidation

Frontend / Клієнтська частина
HTML, CSS, JavaScript

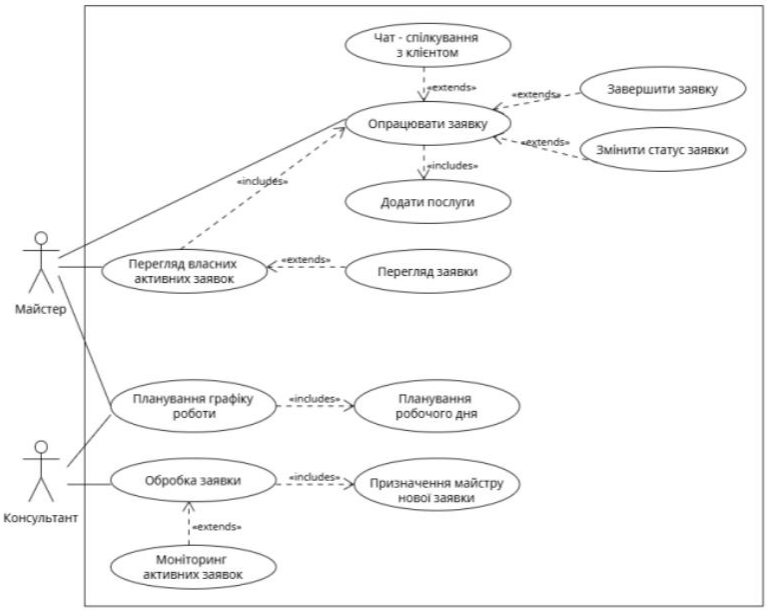
6

Діаграма варіантів використання клієнта



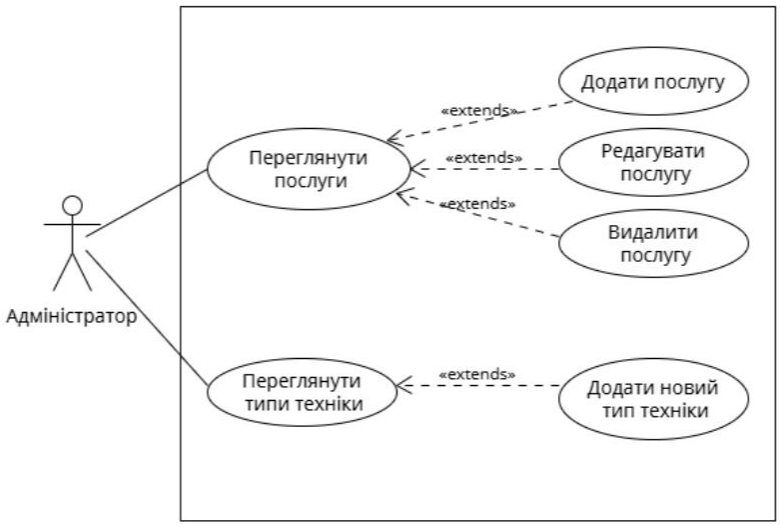
7

Діаграма варіантів використання майстра та консультанта



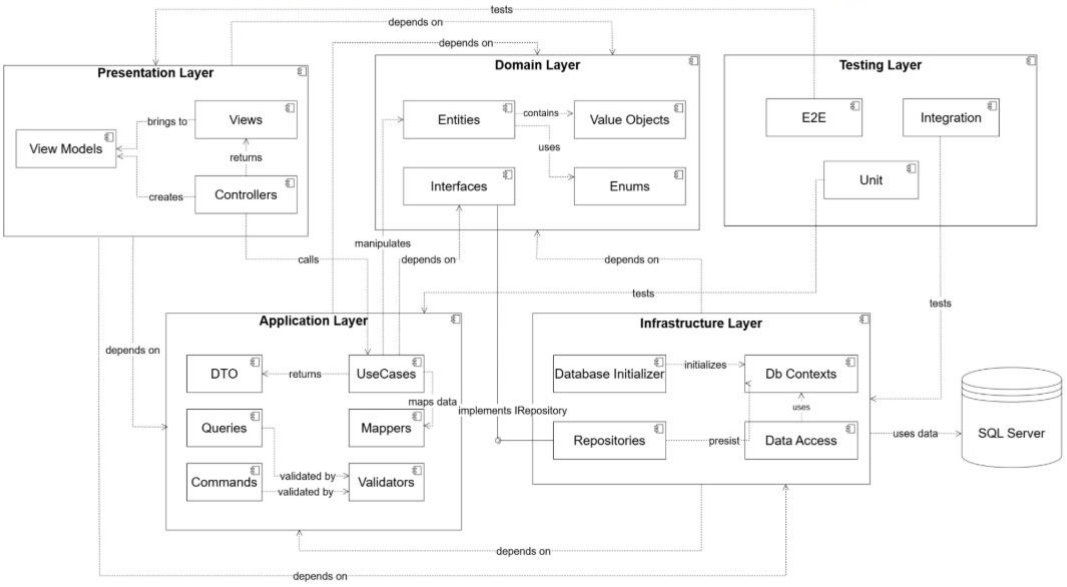
8

Діаграма варіантів використання адміністратора



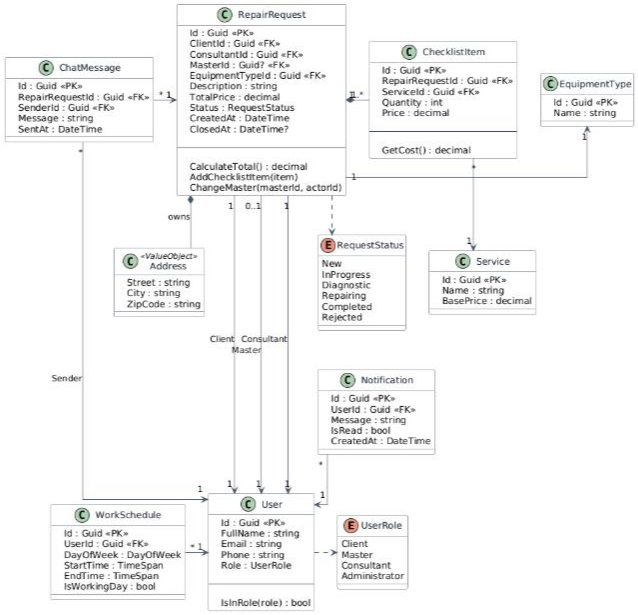
9

Діаграма компонентів веб-додатку



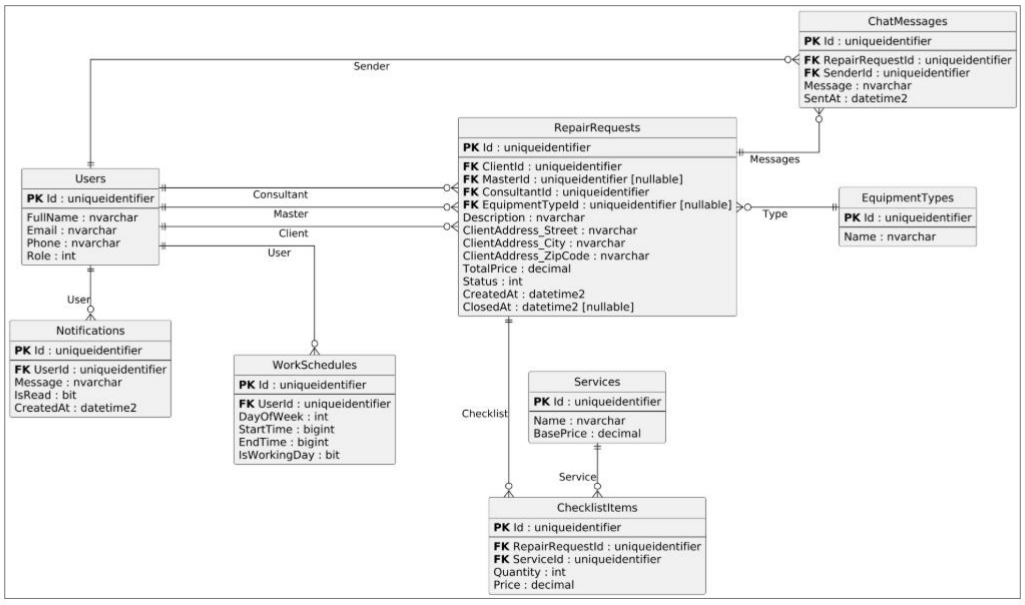
10

Діаграма класів



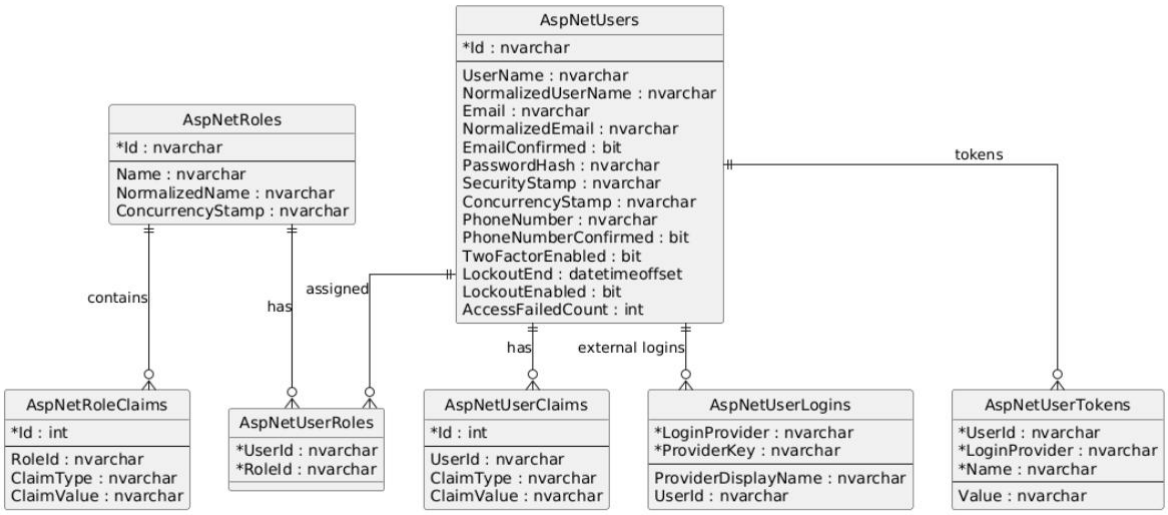
11

ER- діаграма баз даних

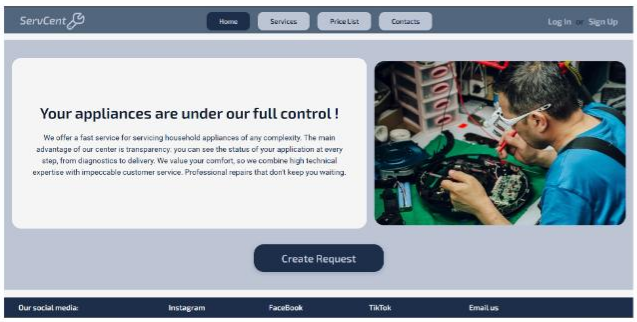


12

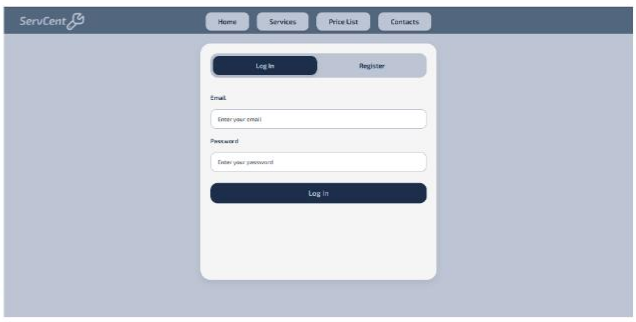
ER-діаграма бази даних Identity



ЕКРАННІ ФОРМИ

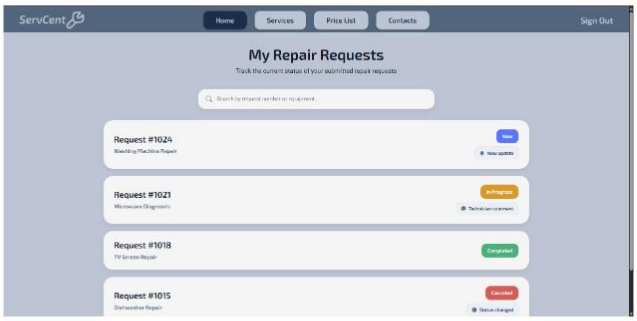


Головна сторінка

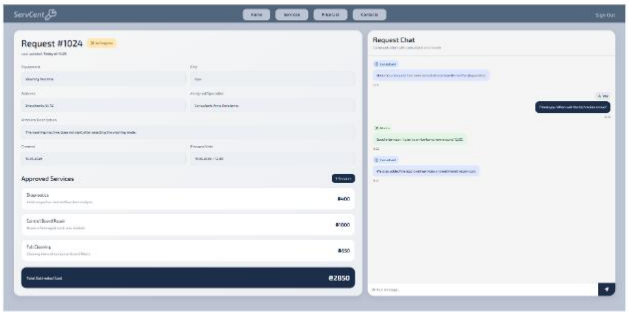


Сторінка входу/реєстрації користувача

ЕКРАННІ ФОРМИ



Перегляд наявних заявок клієнта

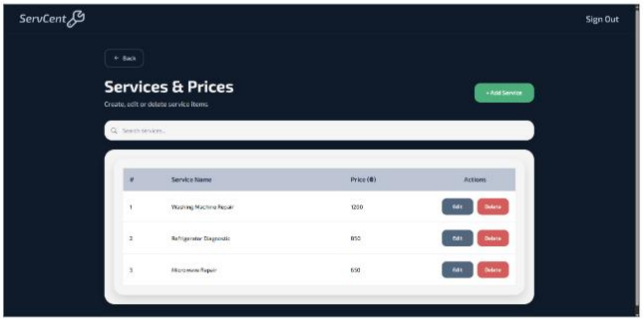


Детальний огляд заявки та чат з майстром/консультантом

15

ЕКРАННІ ФОРМИ

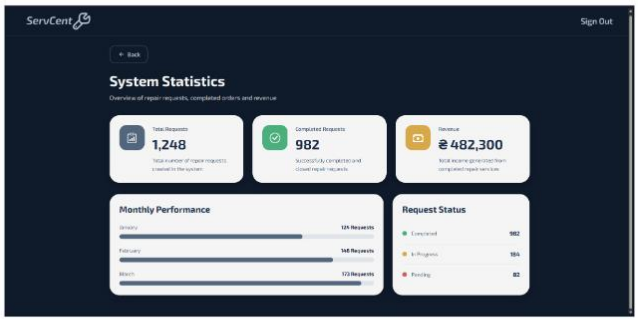
Головне меню адміністратора



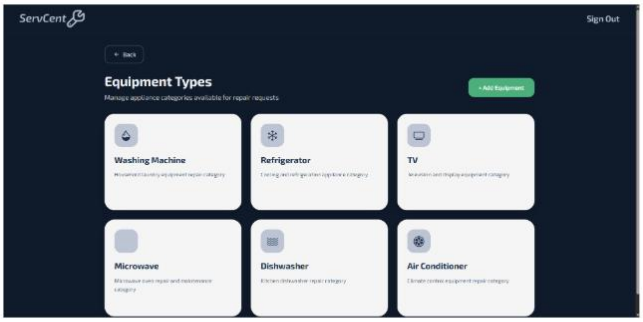
Огляд редагування, видалення та створення послуг

16

ЕКРАННІ ФОРМИ



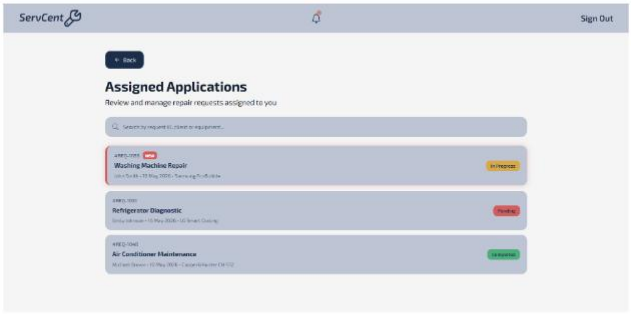
Перегляд статистики сервісного центру: дохід ,кількість заявок та ін.



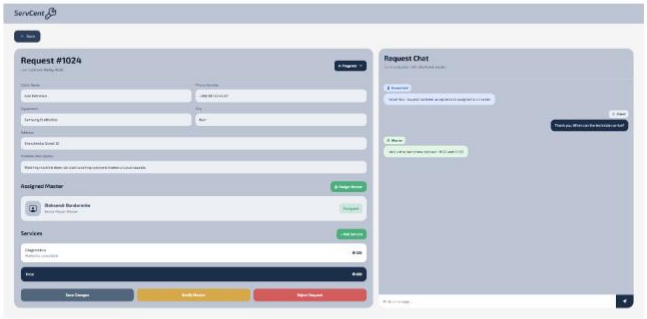
Перегляд та додавання типів техніки.

17

ЕКРАННІ ФОРМИ



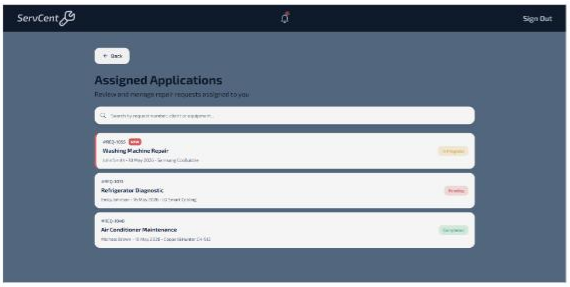
Огляд списку наявних та нових заявок



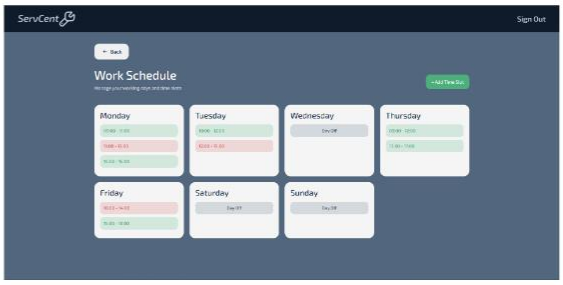
Огляд заявки , призначення майстра, чат з клієнтом

18

ЕКРАННІ ФОРМИ



Огляд списку наявних та нових заявок



Перегляд графіку робочого дня/тижня майстра

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Циганок А. М., Ярошевський О.В. Інструменти та методи взаємодії з базою даних у веб-розробці. Матеріали II Всеукраїнської науково-технічної конференції “Виклики та рішення в програмній інженерії” , 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 26 листопада 2025 року. С.526-528.
2. Циганок А. М., Ярошевський О.В. Автоматизація процесу обробки заявок у сервісному центрі з ремонту побутової техніки. Матеріали VII Всеукраїнської науково-технічної конференції “Застосування програмного забезпечення в інформаційно-комунікаційних технологіях”, 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. *С.324-328.*

ВИСНОВКИ

1. Проведено огляд та аналіз існуючих систем автоматизації сервісних центрів побутової техніки, зокрема RepairDesk, Orderry та RoApp. Визначено їх основні переваги, недоліки та обмеження у контексті діджиталізації бізнес-процесів сервісного обслуговування.
2. Виконано аналіз бізнес-процесів сервісного центру та визначено функціональні й нефункціональні вимоги до веб-платформи. Встановлено основні проблеми предметної галузі, серед яких фрагментарність обробки заявок, недостатній контроль життєвого циклу ремонту, відсутність централізованої комунікації та складність координації взаємодії між учасниками процесу.
3. Спроєктовано архітектуру веб-платформи для автоматизації роботи сервісного центру побутової техніки з використанням принципів багаторівневої побудови програмного забезпечення та сучасних підходів до організації бізнес-логіки системи.
4. Розроблено структуру бази даних, рольову модель доступу та механізми взаємодії між користувачами системи. Спроєктовано ER-модель бази даних, UML-діаграми та основні функціональні модулі відповідно до ролей користувачів і сценаріїв обробки заявок.
5. Реалізовано веб-платформу для автоматизації обробки заявок і управління сервісним обслуговуванням із використанням технологій ASP.NET Core MVC, Entity Framework Core та Microsoft SQL Server. У системі реалізовано механізми автентифікації та авторизації, управління заявками, контроль статусів ремонту та внутрішню комунікацію між учасниками процесу. Проведено тестування основних функціональних можливостей системи.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
public class User
{
   public Guid Id { get; set; }
   public string FullName { get; set; } = string.Empty;
   public string Email { get; set; } = string.Empty;
   public string Phone { get; set; } = string.Empty;
   public UserRole Role { get; set; }
}
```

```
public class User
 {
    public Guid Id { get; set; }
    public string FullName { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;
    public string Phone { get; set; } = string.Empty;
    public UserRole Role { get; set; }
 }
```

```
public class RepairRequest
{
   public Guid Id { get; set; }
   public Guid ClientId { get; set; }
   public User Client { get; set; } = null!;
   public Guid? MasterId { get; set; }
   public User? Master { get; set; }
   public Guid ConsultantId { get; set; }
   public User Consultant { get; set; } = null!;
   public Guid EquipmentTypeId { get; set; }
   public EquipmentType? EquipmentType { get; set; }
   public string Description { get; set; } = string.Empty;
   public Address ClientAddress { get; set; } = null!;

   public List<ChecklistItem> Checklist { get; set; } =
new();
   public decimal TotalPrice { get; set; }
   public RequestStatus Status { get; set; }
   public DateTime CreatedAt { get; set; }
   public DateTime? ClosedAt { get; set; }
}
```

```
 public class Notification
 {
    public Guid Id { get; set; }
    public Guid UserId { get; set; }
    public User User { get; set; } = null!;
    public string Message { get; set; } = string.Empty;
    public bool IsRead { get; set; } = false;
    public DateTime CreatedAt { get; set; }
 }
```

```
public class EquipmentType
{
   public Guid Id { get; set; }
   public string Name { get; set; } = string.Empty;
}
```

```
public class ChecklistItem
{
   public Guid Id { get; set; }
   public Guid RepairRequestId { get; set; }
   public RepairRequest RepairRequest { get; set; } = null!;
   public Guid ServiceId { get; set; }
   public Service Service { get; set; } = null!;
   public int Quantity { get; set; } = 1;
   public decimal Price { get; set; }
}
```

```
 public class ChatMessage
 {
    public Guid Id { get; set; }
    public Guid RepairRequestId { get; set; }
    public RepairRequest RepairRequest { get; set; } =
null!;
    public Guid SenderId { get; set; }
    public User Sender { get; set; } = null!;
    public string Message { get; set; } = string.Empty;
    public DateTime SentAt { get; set; }
 }
```

```
public class CreateRepairRequestValidator :
IValidator<CreateRepairRequestCommand>
{
   public void Validate(CreateRepairRequestCommand
command)
   {
      if (command.ClientId == Guid.Empty)
         throw new Exception("ClientId is required");

      if (command.EquipmentTypeId == Guid.Empty)
         throw new Exception("EquipmentTypeId is
required");

      if (string.IsNullOrWhiteSpace(command.Description))
         throw new Exception("Description is required");

      if (string.IsNullOrWhiteSpace(command.City))
         throw new Exception("City is required");

      if (string.IsNullOrWhiteSpace(command.Street))
         throw new Exception("Street is required");
   }
}
```

```
public class CreateServiceValidator :
IValidator<CreateServiceCommand>
{
   public void Validate(CreateServiceCommand command)
   {
      if (string.IsNullOrWhiteSpace(command.Name))
         throw new Exception("Name is required");

      if (command.BasePrice < 0)
         throw new Exception("Price cannot be negative");
```

```
    }
}
```

```
using ServiceCenterApp.Application.Commands;using
ServiceCenterApp.Application.Interfaces;
namespace ServiceCenterApp.Application.Validators
{
    public class AddChecklistItemValidator :
IValidator<AddChecklistItemCommand>
    {
        public void Validate(AddChecklistItemCommand
command)
        {
            if (command.RequestId == Guid.Empty)
                throw new Exception("RequestId is required");

            if (command.ServiceId == Guid.Empty)
                throw new Exception("ServiceId is required");

            if (command.Quantity <= 0)
                throw new Exception("Quantity must be greater
than 0");
        }
    }
}
```

```
using ServiceCenterApp.Application.Commands;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Enums;using
ServiceCenterApp.Domain.Interfaces;using
ServiceCenterApp.Domain.ValueObjects;
public class CreateRepairRequest
{
    private readonly IRequestRepository _repository;

    public CreateRepairRequest(IRequestRepository
repository)
    {
        _repository = repository;
    }

    public async Task<Guid>
Execute(CreateRepairRequestCommand command)
    {
        var request = new RepairRequest
        {
            Id = Guid.NewGuid(),
            ClientId = command.ClientId,
            EquipmentTypeId = command.EquipmentTypeId,
            Description = command.Description,
            Status = RequestStatus.New,
            CreatedAt = DateTime.UtcNow,
            TotalPrice = 0,

            ClientAddress = new Address(
                command.Street,
                command.City,
                command.ZipCode
            )
        };

        await _repository.AddAsync(request);
        return request.Id;
    }
}
```

```
using ServiceCenterApp.Application.Interfaces;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Interfaces;
public class GetClientRequests
{
    private readonly IRequestRepository _repo;
    private readonly IMapper<RepairRequest,
RepairRequestDto> _mapper;

    public GetClientRequests(
        IRequestRepository repo,
        IMapper<RepairRequest, RepairRequestDto> mapper)
    {
        _repo = repo;
        _mapper = mapper;
    }

    public async Task<List<RepairRequestDto>>
Execute(Guid clientId)
    {
        var data = await _repo.GetByClientIdAsync(clientId);

        return data.Select(_mapper.ToDto).ToList();
    }
}
```

```
using ServiceCenterApp.Domain.Interfaces;
public class CalculateTotalPrice
{
    private readonly IRequestRepository _repo;

    public CalculateTotalPrice(IRequestRepository repo)
    {
        _repo = repo;
    }

    public async Task Execute(Guid requestId)
    {
        var request = await _repo.GetByIdAsync(requestId);
        if (request == null) return;

        request.TotalPrice = request.Checklist.Sum(c =>
c.Price * c.Quantity);

        await _repo.UpdateAsync(request);
    }
}
```

```
using System;using System.Collections.Generic;using
System.Text;
namespace ServiceCenterApp.Application.Commands
{
    public class CreateRepairRequestCommand
    {
        public Guid ClientId { get; set; }
        public Guid EquipmentTypeId { get; set; }
```

```
        public string Description { get; set; } = string.Empty;

        public string Street { get; set; } = string.Empty;
        public string City { get; set; } = string.Empty;
        public string ZipCode { get; set; } = string.Empty;

        public string ServiceFormat { get; set; } =
string.Empty;
    }
}


using ServiceCenterApp.Application.DTO;using
ServiceCenterApp.Application.Interfaces;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Enums;
namespace ServiceCenterApp.Application.Mappers
{
    public class RepairRequestMapper :
IMapper<RepairRequest, RepairRequestDto>
    {
        public RepairRequestDto ToDto(RepairRequest
domain)
        {
            return new RepairRequestDto
            {
                Id = domain.Id,
                ClientName = domain.Client.FullName,
                EquipmentType =
domain.EquipmentType?.Name ?? string.Empty,
                Status = domain.Status.ToString(),
                TotalPrice = domain.TotalPrice,
                CreatedAt = domain.CreatedAt
            };
        }

        public RepairRequest ToDomain(RepairRequestDto
dto)
        {
            return new RepairRequest
            {
                Id = dto.Id,
                TotalPrice = dto.TotalPrice,
                Status =
Enum.Parse<RequestStatus>(dto.Status),
                CreatedAt = dto.CreatedAt
            };
        }
    }
}


using ServiceCenterApp.Application.DTO;using
ServiceCenterApp.Application.Interfaces;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Enums;
namespace ServiceCenterApp.Application.Mappers
{
    public class RequestDetailsMapper :
IMapper<RepairRequest, RequestDetailsDto>
    {
        private readonly ChecklistItemMapper
_checklistMapper = new ChecklistItemMapper();

        public RequestDetailsDto ToDto(RepairRequest
domain)
        {
            return new RequestDetailsDto
            {
                Id = domain.Id,
                Description = domain.Description,
                Status = domain.Status.ToString(),
                ClientName = domain.Client.FullName,
                MasterName = domain.Master?.FullName ??
string.Empty,
                Checklist = domain.Checklist.Select(c =>
_checklistMapper.ToDto(c)).ToList()
            };
        }

        public RepairRequest ToDomain(RequestDetailsDto
dto)
        {
            return new RepairRequest
            {
                Id = dto.Id,
                Description = dto.Description,
                Status =
Enum.Parse<RequestStatus>(dto.Status),
                Checklist = dto.Checklist.Select(c =>
_checklistMapper.ToDomain(c)).ToList()
            };
        }
    }
}
using Microsoft.EntityFrameworkCore;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.ValueObjects;
namespace
ServiceCenterApp.Infrastructure.DataAccess.MsSql.Conte
xt
{
    public class AppDbContext : DbContext
    {
        public
AppDbContext(DbContextOptions<AppDbContext>
options)
            : base(options) { }

        public DbSet<User> Users { get; set; }
        public DbSet<RepairRequest> RepairRequests { get;
set; }
        public DbSet<Service> Services { get; set; }
        public DbSet<ChatMessage> ChatMessages { get;
set; }
        public DbSet<ChecklistItem> ChecklistItems { get;
set; }
        public DbSet<Notification> Notifications { get; set; }
        public DbSet<EquipmentType> EquipmentTypes
{ get; set; }
        public DbSet<WorkSchedule> WorkSchedules { get;
set; }

        protected override void
OnModelCreating(ModelBuilder modelBuilder)
        {
```

```
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<RepairRequest>()
                .OwnsOne(r => r.ClientAddress);

            modelBuilder.Entity<RepairRequest>()
                .HasOne(r => r.Client)
                .WithMany()
                .HasForeignKey(r => r.ClientId)
                .OnDelete(DeleteBehavior.Restrict);

            modelBuilder.Entity<RepairRequest>()
                .HasOne(r => r.Consultant)
                .WithMany()
                .HasForeignKey(r => r.ConsultantId)
                .OnDelete(DeleteBehavior.Restrict);

            modelBuilder.Entity<RepairRequest>()
                .HasOne(r => r.Master)
                .WithMany()
                .HasForeignKey(r => r.MasterId)
                .OnDelete(DeleteBehavior.Restrict);

            modelBuilder.Entity<WorkSchedule>(entity =>
            {
                entity.ToTable("WorkSchedules");

                entity.HasKey(x => x.Id);


                entity.Property(x => x.Id)
                    .ValueGeneratedOnAdd();

                entity.Property(x => x.UserId)
                    .IsRequired();

                entity.Property(x => x.DayOfWeek)
                    .HasConversion<int>()
                    .IsRequired();

                entity.Property(x => x.StartTime)
                    .HasConversion(
                        v => v.Ticks,
                        v => TimeSpan.FromTicks(v))
                    .IsRequired();

                entity.Property(x => x.EndTime)
                    .HasConversion(
                        v => v.Ticks,
                        v => TimeSpan.FromTicks(v))
                    .IsRequired();

                entity.Property(x => x.IsWorkingDay)
                    .IsRequired();

                entity.HasOne(x => x.User)
                    .WithMany()
                    .HasForeignKey(x => x.UserId);
            });
        }
    }
}
```

```
using Microsoft.EntityFrameworkCore;using
Microsoft.EntityFrameworkCore.Design;using
ServiceCenterApp.Infrastructure.DataAccess.MsSql.Conte
xt;
namespace
ServiceCenterApp.Infrastructure.DataAccess.MsSql
{
    public class AppDbContextFactory :
IDesignTimeDbContextFactory<AppDbContext>
    {
        public AppDbContext CreateDbContext(string[] args)
        {
            var optionsBuilder = new
DbContextOptionsBuilder<AppDbContext>();

            optionsBuilder.UseSqlServer(

"Server=(localdb)\\MSSQLLocalDB;Database=ServiceCe
nterDb;Trusted_Connection=True;TrustServerCertificate=
True;"
            );

            return new AppDbContext(optionsBuilder.Options);
        }
    }
}
```

```
using ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Infrastructure.DataAccess.MsSql.Conte
xt;using System.Linq;using
Microsoft.AspNetCore.Identity;
namespace
ServiceCenterApp.Infrastructure.DataAccess.MsSql
{
    public class DatabaseInitializer
    {
        private readonly AppDbContext _context;

        public DatabaseInitializer(AppDbContext context)
        {
            _context = context;
        }



        public void Initialize()
        {
            _context.Database.EnsureCreated();

            if (!_context.Services.Any())
            {
                _context.Services.AddRange(
                    new Service
                    {
                        Id = Guid.NewGuid(),
                        Name = "Repair",
                        BasePrice = 500
                    },
                    new Service
                    {
                        Id = Guid.NewGuid(),
                        Name = "Diagnostics",
```

```
                BasePrice = 200
            }
        );
    }

    if (!_context.EquipmentTypes.Any())
    {
        _context.EquipmentTypes.AddRange(
            new EquipmentType
            {
                Id = Guid.NewGuid(),
                Name = "Laptop"
            },
            new EquipmentType
            {
                Id = Guid.NewGuid(),
                Name = "Phone"
            }
        );
    }

    _context.SaveChanges();
  }
 }
}
```

```
using Microsoft.EntityFrameworkCore;using
ServiceCenterApp.Domain.Interfaces;using
ServiceCenterApp.Infrastructure.DataAccess.MsSql.Conte
xt;
namespace
ServiceCenterApp.Infrastructure.DataAccess.MsSql.Repos
itories
{
  public class Repository<T> : IRepository<T> where T :
class
  {
    protected readonly AppDbContext _context;

    public Repository(AppDbContext context)
    {
      _context = context;
    }

    public async Task AddAsync(T entity)
    {
      await _context.Set<T>().AddAsync(entity);
      await _context.SaveChangesAsync();
    }

    public async Task DeleteAsync(T entity)
    {
      _context.Set<T>().Remove(entity);
      await _context.SaveChangesAsync();
    }

    public async Task<List<T>> GetAllAsync()
      => await _context.Set<T>().ToListAsync();

    public async Task<T?> GetByIdAsync(Guid id)
      => await _context.Set<T>().FindAsync(id);

    public async Task UpdateAsync(T entity)
    {
      _context.Set<T>().Update(entity);
      await _context.SaveChangesAsync();
    }
  }
}
```

```
using FluentAssertions;using NSubstitute;using
ServiceCenterApp.Application.Commands;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Enums;using
ServiceCenterApp.Domain.Interfaces;using Xunit;
namespace ServiceCenterApp.Tests.UseCases;
public class CreateRepairRequestTests
{
  private readonly IRequestRepository _repo =
    Substitute.For<IRequestRepository>();

  private readonly CreateRepairRequest _useCase;

  public CreateRepairRequestTests()
  {
    _useCase = new CreateRepairRequest(_repo);
  }

  [Fact]
  public async Task
Should_Create_Request_With_New_Status()
  {
    var command = new CreateRepairRequestCommand
    {
      ClientId = Guid.NewGuid(),
      EquipmentTypeId = Guid.NewGuid(),
      Description = "Broken screen",
      Street = "Main",
      City = "Kyiv",
      ZipCode = "01001"
    };

    var repo = Substitute.For<IRequestRepository>();
    var useCase = new CreateRepairRequest(repo);

    var resultId = await useCase.Execute(command);

    await
repo.Received(1).AddAsync(Arg.Is<RepairRequest>(r =>
      r.ClientId == command.ClientId &&
      r.Status == RequestStatus.New &&
      r.Description == command.Description &&
      r.ClientAddress.City == command.City &&
      r.EquipmentTypeId ==
command.EquipmentTypeId &&
      r.Id != Guid.Empty
    ));

    resultId.Should().NotBe(Guid.Empty);
  }
}
```

```
using NSubstitute;using Xunit;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Interfaces;
```

```
namespace ServiceCenterApp.Tests.UseCases;
public class CreateServiceTests
{
    private readonly IRepository<Service> _repo =
        Substitute.For<IRepository<Service>>();

    private readonly CreateService _useCase;

    public CreateServiceTests()
    {
        _useCase = new CreateService(_repo);
    }

    [Fact]
    public async Task Should_Add_Service()
    {
        var dto = new ServiceDto
        {
            Name = "Repair",
            BasePrice = 100
        };

        await _useCase.Execute(dto);

        await
_repo.Received(1).AddAsync(Arg.Is<Service>(s =>
            s.Name == "Repair" &&
            s.BasePrice == 100 &&
            s.Id != Guid.Empty
        ));
    }
}
```

```
using NSubstitute;using Xunit;using
ServiceCenterApp.Domain.Entities;using
ServiceCenterApp.Domain.Interfaces;
namespace ServiceCenterApp.Tests.UseCases;
public class DeleteServiceTests
{
    private readonly IRepository<Service> _repo =
        Substitute.For<IRepository<Service>>();

    private readonly DeleteService _useCase;

public DeleteServiceTests()
    {
        _useCase = new DeleteService(_repo);
    }

    [Fact]
    public async Task
Should_Delete_Service_When_Exists()
    {
        var service = new Service
        {
            Id = Guid.NewGuid(),
            Name = "Test"
        };

        _repo.GetByIdAsync(service.Id).Returns(service);

        await _useCase.Execute(service.Id);

        await _repo.Received(1).DeleteAsync(service);
    }

    [Fact]
    public async Task
Should_Do_Nothing_When_Service_Not_Found()
    {
        _repo.GetByIdAsync(Arg.Any<Guid>())
            .Returns((Service?)null);

        await _useCase.Execute(Guid.NewGuid());

        await
_repo.DidNotReceive().DeleteAsync(Arg.Any<Service>())
;
    }
}
```