Національний лісотехнічний університет України

Навчально-науковий інститут комп'ютерних наук та інформаційних технологій

Кафедра комп'ютерних наук

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)

на тему: "Розроблення вебмодуля служби підтримки клієнтів за допомогою Asp.NetCore & Angular"

Виконав: студент 4 курсу групи КН-41
спеціальності
122 "Комп'ютерні науки"

Осадчук О. Л.
(прізвище та ініціали)

Керівник Опришко М.І., Карашецький В.П.
(прізвище та ініціали)

Рецензент Глуд Л.О.
(прізвище та ініціали)

Львів – 2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
Борецька І.Б.
" "
10 червня 2025 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Осадчук Олесь Леонідович
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення вебмодуля служби підтримки клієнтів за допомогою Asp.NetCore & Angular

керівник роботи *ас. Оприщко М. І., к.т.н., доцент Карашецький В.П*,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "15" 11 2024 року №С-882
2. Термін подання студентом роботи 10.06.25
3. Вихідні дані до роботи Розробити практичний модульний веб сервіс для підтримки клієнтів за допомогою Asp.NetCore & Angular. Реалізувати функціонал для створення запитів на підтримку для клієнтів та портал для комунікації з експертами у реальному часі

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1. Стан проблемної області
2. Інформаційне та математичне забезпечення
3. Програмне та технічне забезпечення

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Підготовка матеріалів до доповіді

6. Дата видачі завдання 18.11.24

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області	25.12.24 -15.01.25	Виконано
2	Постановка задачі та її формалізації	15.01.24 – 30.01.25	Виконано
3	Реалізація вебзастосунку	30.01.25 – 30.04.25	Виконано
4	Тестування вебзастосунку	30.04.25 - 15.05.25	Виконано
5	Виправлення дефектів	15.05.25 - 30.05.25	Виконано
6	Оформлення пояснювальної записки	30.05.25 - 09.06.2025	Виконано

Студент _____ Осадчук О. Л.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Опришко М.І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Карашецький В.П
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 45 сторінки пояснювальної записки, 19 рисунків, 1 додаток та 16 джерел.

Результатом виконання дипломної роботи є вебзастосунок, розроблений за допомогою сучасних фреймворків Angular та .NET, який забезпечує зручну та доступну систему для підтримки клієнтів. Метою проєкту є створення універсального модуля, який може бути інтегрований у будь-який інший продукт, забезпечуючи гнучке рішення для обробки запитів клієнтів. Застосунок дозволяє користувачам створювати запити, описуючи проблему, обирати тип запиту та прикріплювати медіафайли. Після створення запиту відкривається діалог у реальному часі з адміністратором за допомогою технології SignalR. Адміністратори мають доступ до панелі управління з фільтрами, статистикою через графіки (Chart.js) та можливістю призначати запити.

Ключові слова: вебзастосунок, система підтримки, Angular, .NET, SignalR, Chart.js, база даних, авторизація.

ABSTRACT

The thesis contains 45 pages of explanatory note, 19 figures, 1 appendix, 16 sources

The result of the bachelor's thesis is a web application developed using modern Angular and .NET frameworks, providing a convenient and accessible customer support system. The project aims to create a universal module that can be integrated into any other product, offering a flexible solution for handling customer requests. The application allows users to create requests by describing their issues, selecting a request type, and attaching media files. After creating a request, a real-time dialogue with an administrator is initiated using SignalR technology. Administrators have access to a management panel with filters, statistics via charts (Chart.js), and the ability to assign requests.

Keywords: web application, customer support, Angular, .NET, SignalR, MailKit, Chart.js, database, authorization

ТЕХНІЧНЕ ЗАВДАННЯ

Для розробки вебзастосунку системи підтримки клієнтів з використанням фреймворків Angular та .NET необхідно реалізувати наступні функціональні модулі:

1. **Система авторизації та аутентифікації**: Реалізувати безпечну авторизацію через ASP.NET Identity з обов'язковим підтвердженням електронної пошти через MailKit. Користувачі повинні мати можливість створювати обліковий запис, входити в систему та отримувати JWT-токен для доступу до захищених ресурсів.
2. **Модуль створення та управління запитами**: Користувачі можуть створювати запити, обираючи тип запиту (PaymentIssue, WebsiteIssue, SecurityIssue, Other), додаючи опис проблеми та прикріплюючи медіафайли. Запити відображаються в особистому кабінеті користувача.
3. **Чат у реальному часі**: Реалізувати діалог між користувачем та адміністратором за допомогою SignalR, з можливістю прикріплення файлів у чаті.
4. **Панель адміністратора**: Реалізувати інтерфейс для адміністраторів з можливістю перегляду всіх запитів, фільтрації за типами та статусами (IsAssigned, IsSolved, IsClosed), а також відображення статистики через графіки Chart.js. Адміністратори можуть призначати запити собі або іншим адміністраторам.
5. **Панель суперадміністратора**: Реалізувати окрему сторінку для суперадміністратора, яка дозволяє створювати, редагувати та видаляти облікові записи адміністраторів.
6. **Система сповіщень**: Інтегрувати сповіщення через електронну пошту за допомогою MailKit для інформування користувачів про статус їхніх запитів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ 7
ВСТУП 8
РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 10
1.1 Огляд сучасних систем підтримки клієнтів 10
1.2 Аналіз аналогів 12
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 15
2.1 Архітектура вебзастосунків на Angular та .NET 15
2.1.1 Опис архітектури та діаграм класів і проєктів 19
2.2 Використання SignalR для чату в реальному часі 22
2.3 Інтерфейс користувача (UI) вебзастосунку 23
РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ 26
3.1 Розробка вебзастосунку 26
3.2. Функціонал розробленого застосунку 28
3.3. Модель бази даних 31
3.4 Фонові сервіси 39
ВИСНОВКИ 43
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 44
ДОДАТОК А 46

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (Програмний інтерфейс додатків).

ASP.NET – Active Server Pages .NET (Фреймворк для створення вебзастосунків від Microsoft).

Chart.js – Бібліотека JavaScript для створення інтерактивних графіків і діаграм.

HTML – HyperText Markup Language (Мова розмітки гіпертексту).

HTTP – HyperText Transfer Protocol (Протокол передачі гіпертексту).

JSON – JavaScript Object Notation (Формат обміну даними).

JWT – JSON Web Token (Токен для автентифікації та авторизації).

MVC – Model-View-Controller (Архітектурний шаблон для організації коду).

ORM – Object-Relational Mapping (Об’єктно-реляційне відображення).

REST – Representational State Transfer (Архітектурний стиль для вебсервісів).

RxJS – Reactive Extensions for JavaScript (Бібліотека для реактивного програмування).

SignalR – Бібліотека для реалізації комунікації в реальному часі.

SQL – Structured Query Language (Мова структурованих запитів для баз даних).

TS – TypeScript (Мова програмування, що розширює JavaScript).

UI – User Interface (Користувацький інтерфейс).

WebSocket – Протокол для двосторонньої комунікації в реальному часі.

ВСТУП

Актуальність теми дипломної роботи зумовлена зростаючим попитом на автоматизовані системи підтримки клієнтів у сучасному бізнесі, де ефективна комунікація та оперативне вирішення проблем стають ключовими факторами конкурентоспроможності. Збільшення кількість онлайн-платформ і користувачів вимагає створення гнучких і безпечних рішень, які можуть масштабуватися та інтегруватися в існуючі екосистеми компаній.

Об’єктом дослідження є процеси організації та автоматизації підтримки клієнтів у вебзастосунках, зокрема аспекти управління запитами, комунікації в реальному часі та аналітики.

Предметом дослідження є розробка вебзастосунку для системи підтримки клієнтів на основі технологій Angular та .NET, включаючи архітектуру, модель бази даних, фонові сервіси та інтерфейс користувача.

Мета роботи полягає у створенні універсального, модульного та безпечного вебзастосунку, який забезпечить ефективну підтримку клієнтів, адаптується до потреб компаній різного масштабу та інтегрується з існуючими бізнес-процесами.

Для досягнення мети поставлено такі завдання:

- Провести аналіз сучасних підходів до систем підтримки клієнтів та їхніх технологічних основ.
- Вибрати та обґрунтувати технологічний стек для розробки (Angular, ASP.NET Core, SignalR, Chart.js тощо).
- Розробити модель бази даних і архітектуру застосунку з урахуванням масштабування та безпеки.
- Реалізувати функціональні модулі: авторизацію, створення та управління тікетами, чат у реальному часі, аналітику.
- Провести тестування системи та оцінити її продуктивність.

Практичне значення роботи полягає в наданні економічно вигідної альтернативи комерційним платформам підтримки, яка доступна для малих і

середніх компаній. Розроблений вебзастосунок дозволяє автоматизувати ключові процеси підтримки, знижувати операційні витрати та підвищувати якість обслуговування в таких сферах, як електронна комерція, технічна підтримка чи внутрішня комунікація.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Огляд сучасних систем підтримки клієнтів

Системи підтримки клієнтів є невід'ємною складовою сучасного бізнесу, слугуючи мостом між компаніями та їхніми клієнтами та відіграючи ключову роль у підвищенні рівня задоволеності, лояльності та довіри користувачів. У контексті стрімкої цифровізації та глобалізації ринків ці системи стали основним інструментом для забезпечення безперервного зв'язку, швидкого вирішення проблем і створення позитивного враження про бренд. Зі зростанням обсягів онлайн-взаємодії, зокрема в електронній комерції, банківській сфері, телекомунікаціях, охороні здоров'я та освіті, вимоги до систем підтримки суттєво зросли. Сучасні платформи повинні бути швидкими, інтуїтивно зрозумілими, адаптованими до різних пристроїв (ПК, смартфони, планшети) і здатними обробляти значні обсяги запитів без втрати якості обслуговування. Вони також повинні відповідати сучасним стандартам безпеки, забезпечувати персоналізований підхід і підтримувати інтеграцію з іншими цифровими інструментами, такими як CRM-системи чи аналітичні платформи.

Основними компонентами сучасних систем підтримки клієнтів є:

- **Системи тікетів**: Ці системи дозволяють користувачам створювати запити (тікети), які потім розподіляються між адміністраторами для обробки. Вони забезпечують структуроване управління запитами, дозволяючи відстежувати їхній статус (наприклад, "відкрито", "в обробці", "вирішено"), пріоритет і терміни виконання. Такі платформи часто включають автоматичну категоризацію запитів, що прискорює їхнє розподілення.
- **Чати в реальному часі**: Використання технологій, таких як WebSocket і бібліотека SignalR, забезпечує миттєву двосторонню комунікацію між клієнтами та службою підтримки. Це дозволяє вирішувати проблеми на ходу, скорочуючи час реакції до кількох секунд і підвищуючи рівень задоволеності користувачів, особливо в ситуаціях, що потребують термінового втручання.

- **Автоматизація сповіщень**: Інтеграція з поштовими сервісами через бібліотеки, такі як MailKit, дозволяє автоматично інформувати користувачів про оновлення статусу їхніх запитів, призначення адміністратора чи необхідність додаткових дій. Це підвищує прозорість процесу й зменшує кількість повторних звернень, створюючи відчуття контролю з боку клієнта.
- **Аналітичні інструменти**: Використання бібліотек, таких як Chart.js, для створення графіків, діаграм і звітів дає адміністраторам змогу аналізувати статистику запитів, виявляти пікові навантаження, популярні проблеми та тенденції. Наприклад, графіки можуть показувати розподіл типів запитів (технічні, фінансові, інші) за місяцями, що допомагає оптимізувати розподіл ресурсів служби підтримки.
- **Системи авторизації та безпеки**: Використання фреймворків, таких як ASP.NET Identity, із підтримкою JWT-токенів забезпечує захист конфіденційних даних користувачів, контроль доступу до функціоналу (наприклад, обмеження адмін-панелі для певних ролей) і запобігає несанкціонованому доступу. Це особливо важливо в системах, де обробляються персональні дані чи фінансові транзакції.

Сьогодні компанії стикаються з низкою викликів, пов’язаних із впровадженням і використанням систем підтримки клієнтів. Одним із ключових бар’єрів є висока вартість комерційних платформ, таких як Steam Support чи Salesforce, які, попри свій широкий функціонал, часто недоступні для малих і середніх підприємств через складність інтеграції в існуючі бізнес-процеси та потребу в додаткових ресурсах для налаштування. Наприклад, великі платформи пропонують комплексні рішення, але їхня висока ціна, складність кастомізації та залежність від хмарної інфраструктури можуть стати перешкодою для компаній із обмеженим бюджетом. Це стимулює попит на модульні, економічно вигідні та легко адаптовані рішення, які можуть бути інтегровані в існуючі системи без значних витрат.

Розробка власного вебзастосунку для системи підтримки клієнтів дозволяє подолати ці недоліки, пропонуючи гнучке й економічно доступне рішення. Використання сучасних технологій, таких як Angular для створення інтерактивного та адаптивного інтерфейсу, який реагує на дії користувача в реальному часі, і .NET для надійного серверного забезпечення з підтримкою SignalR, MailKit та ASP.NET Identity, забезпечує високу продуктивність, безпеку та масштабованість. Такий підхід дозволяє інтегрувати чат у реальному часі, автоматизовані сповіщення та аналітику, адаптовану до потреб конкретного бізнесу, уникаючи високих витрат на комерційні аналоги. Наприклад, додавання модулів для обробки специфічних типів запитів (наприклад, фінансових чи технічних) може бути легко реалізовано через оновлення enum RequestTypes, що робить систему більш гнучкою порівняно з фіксованими рішеннями, такими як Steam Support. Водночас модульність і локальна розгортальність відрізняють її від хмарозалежних платформ, таких як Salesforce, надаючи компаніям більший контроль над даними та інфраструктурою.

1.2 Аналіз аналогів

Для розробки вебзастосунку для системи підтримки клієнтів було проведено аналіз існуючих рішень, таких як Steam Support і Salesforce, щоб визначити їхні сильні сторони, недоліки та можливості для вдосконалення. Цей аналіз допоміг сформувати вимоги до власного проєкту, спрямовані на підвищення ефективності, зручності та масштабовності.

Steam Support - це система підтримки клієнтів, розроблена компанією Valve для користувачів платформи Steam, яка охоплює ігрові покупки, технічну підтримку та управління акаунтами. Основні характеристики:

- **Функціонал**: Система дозволяє користувачам подавати запити через онлайн-форму, включаючи повернення ігор, проблеми з оплатою чи акаунтом. Підтримка здійснюється через тікет-систему з можливістю відстеження статусу.

- **Переваги**: Автоматична обробка простих запитів (наприклад, повернення коштів у межах 14 днів) і надання статистики підтримки для аналізу (наприклад, середній час вирішення тікетів). Інтерфейс інтуїтивно зрозумілий, а дані про продуктивність доступні на офіційному сайті.
- **Недоліки**: Відсутність чату в реальному часі, що сповільнює вирішення складних питань. Тривалий час очікування відповіді (до кількох днів) і обмежена персоналізація підтримки. Також немає інтеграції з зовнішніми каналами, такими як соціальні мережі.
- **Уроки для проєкту**: Необхідність інтегрувати чат у реальному часі (наприклад, через SignalR) і забезпечити швидшу реакцію на запити, зберігаючи при цьому автоматизацію рутинних завдань.

Salesforce - це хмарна платформа для управління відносинами з клієнтами (CRM), яка включає модуль Service Cloud для підтримки клієнтів. Вона використовується великими підприємствами для обробки запитів через різні канали.

- **Функціонал**: Пропонує омніканальну підтримку (email, телефон, чат, соціальні мережі), автоматизацію тікетів, самодопомогу через портали знань і аналітику на основі AI (наприклад, аналіз настрою клієнтів і прогнозування ескалацій). Інтеграція з іншими модулями (Sales Cloud, Marketing Cloud) забезпечує єдину базу даних клієнтів.
- **Переваги**: Масштабованість, підтримка AI для персоналізованих відповідей і значне зниження ескалацій (до 56% за даними внутрішнього аналізу). Модульність дозволяє адаптувати систему до потреб різних галузей, а аналітика допомагає оптимізувати ресурси.
- **Недоліки**: Висока вартість, що робить її недоступною для малих і середніх компаній, а також складність налаштування та навчання персоналу. Надмірна залежність від хмарної інфраструктури може створювати ризики при нестабільному інтернет-з’єднанні.

- **Уроки для проєкту**: Важливість модульності та інтеграції з аналітикою, але з акцентом на доступність для невеликих організацій і гнучкість у локальному розгортанні.

Steam Support, розроблений компанією Valve для користувачів платформи Steam, є прикладом спеціалізованої системи, орієнтованої на ігрову індустрію. Вона дозволяє подавати запити через онлайн-форми для вирішення проблем із покупками, акаунтами чи технічними збоями, із можливістю відстеження статусу тікета. Серед переваг - автоматизоване повернення коштів у межах 14 днів і простота інтерфейсу, що підходить для масового використання. Однак система має значні недоліки: відсутність чату в реальному часі, тривалий час очікування відповідей (до кількох днів) і обмежена персоналізація, що може розчаровувати користувачів із складними питаннями. Salesforce, навпаки, є універсальною хмарною платформою із модулем Service Cloud, який підтримує омніканальну комунікацію (email, телефон, чат, соціальні мережі), автоматизацію тікетів і аналітику на основі штучного інтелекту. Її переваги включають високу масштабованість, персоналізовані відповіді завдяки AI та інтеграцію з іншими модулями (Sales Cloud, Marketing Cloud). Проте висока вартість підписки, складність налаштування та потреба в постійному підключенні до інтернету роблять її менш привабливою для невеликих організацій.

Аналіз Steam Support і Salesforce показав, що ефективна система підтримки клієнтів повинна поєднувати автоматизацію простих запитів, миттєву комунікацію через чат, персоналізацію за допомогою даних і масштабованість. Однак комерційні платформи часто мають високу вартість і складність, що робить їх менш привабливими для малих компаній. Розроблений вебзастосунок враховує ці аспекти, пропонуючи безкоштовний базовий функціонал із можливістю розширення, інтеграцію чату в реальному часі через SignalR і модульну архітектуру, яка дозволяє адаптацію до різних сценаріїв використання без значних витрат.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Архітектура вебзастосунків на Angular та .NET

Архітектура розробленого вебзастосунку для системи підтримки клієнтів базується на сучасній клієнт-серверній моделі, яка забезпечує чіткий розподіл відповідальності між фронтендом і бекендом, сприяючи масштабовності, легкості підтримки та інтеграції з іншими системами. Фронтенд реалізовано за допомогою фреймворку Angular, а бекенд - на базі ASP.NET Core, що дозволяє створити гнучку та ефективну платформу для обробки запитів клієнтів. Взаємодія між цими частинами здійснюється через REST API, яке стандартизує обмін даними, забезпечує безпеку через авторизацію та полегшує подальшу інтеграцію з зовнішніми сервісами. Такий архітектурний підхід відображає сучасні тенденції розробки вебзастосунків, де модульність і адаптивність є ключовими факторами для успішного впровадження в різних бізнес-сценаріях.

Angular є основою для створення інтерактивного та динамічного користувацького інтерфейсу, що є критично важливим для зручності роботи з системою підтримки клієнтів. Використання Angular 16 забезпечує потужну інфраструктуру для організації коду в модулі, компоненти, сервіси та пайпи, що сприяє його структурованості, тестуванню та повторному використанню. Фреймворк підтримує двостороннє прив’язування даних, що дозволяє миттєво оновлювати інтерфейс у відповідь на дії користувача або серверні події, наприклад, відображення нових повідомлень у чаті чи оновлення статусу тікета. Для створення візуально привабливих і функціональних елементів застосована бібліотека PrimeNG, яка пропонує широкий набір компонентів, таких як таблиці з фільтрами для панелі адміністратора, форми для створення запитів із валідацією, діалогові вікна для підтвердження дій і випадаючі списки для вибору типів запитів (PaymentIssue, WebsiteIssue, SecurityIssue, Other). Крім того, інтеграція Chart.js дозволяє відображати аналітичні графіки, наприклад, розподіл тікетів за статусами чи типами, що значно полегшує аналіз роботи служби підтримки.

Ключовим елементом архітектури фронтенду є модуль маршрутизації AppRoutingModule, який визначає навігацію в застосунку. Цей модуль використовує RouterModule від Angular для конфігурації маршрутів, що дозволяє користувачам переходити між сторінками, такими як головна сторінка, сторінки авторизації (login, signup), управління тікетами (tickets, tickets/create, tickets/:number), чат (dialogs) і адміністративна панель (admin, admin/create, admin/update/:id). Маршрути захищені за допомогою.guard-файлів (AuthGuard для загального доступу після авторизації та AdminGuard для обмеженого доступу адміністраторів), що забезпечує безпеку та контроль доступу. Наприклад, шлях tickets доступний лише авторизованим користувачам, а admin – лише адміністраторам. Модуль також включає компоненти для обробки помилок, таких як AccessDeniedComponent (403) і NotFoundComponent (404), а також перенаправлення всіх невідомих маршрутів на сторінку 404. Код модуля реалізує ледаче завантаження модулів (loadChildren), що оптимізує продуктивність, завантажуючи лише необхідні модулі за потреби. Нижче наведено приклад коду (див. рис. 2.1):

```
const routes: Routes = [
  {
    path: '',
    loadChildren: () => import('./core/pages/main-page/main-page.module')
      .then(m => m.MainPageModule)
  },
  {
    path: 'signup',
    loadChildren: () => import('./core/pages/signup-page/signup-page.module')
      .then(m => m.SignupPageModule)
  },
  {
    path: 'login',
    loadChildren: () => import('./core/pages/login-page/login-page.module')
      .then(m => m.LoginPageModule)
  },
  {
    path: 'tickets',
    canActivate: [AuthGuard],
    loadChildren: () => import('./core/pages/tickets-page/tickets-page.module')
      .then(m => m.TicketsPageModule)
  },
  {
    path: 'tickets/create',
    canActivate: [AuthGuard],
    loadChildren: () => import('./core/pages/ticket-create-page/ticket-create-page.module')
      .then(m => m.TicketCreatePageModule)
  },
  {
    path: 'tickets/:number',
    canActivate: [AuthGuard],
    loadChildren: () => import('./core/pages/ticket-view/ticket-view.module')
      .then(m => m.TicketViewModule)
  },
  {
    path: 'dialogs',
    canActivate: [AuthGuard],
    loadChildren: () => import("./core/pages/dialog-page/dialog-page.module")
      .then(m => m.DialogPageModule)
  },
  {
    path: 'admin',
    canActivate: [AdminGuard],
    loadChildren: () =>
      import('./core/pages/admin-page/admin-page.module').then(m => m.AdminPageModule)
  },
  {
    path: 'admin/create',
    canActivate: [AdminGuard],
    loadChildren: () => import("./core/pages/admin-create-page/admin-create-page.module")
      .then(m => m.AdminCreatePageModule)
  },
  {
    path: 'admin/update/:id',
    canActivate: [AdminGuard],
    loadChildren: () => import("./core/pages/admin-update-page/admin-update-page.module")
      .then(m => m.AdminUpdatePageModule)
  },
  {
    path: '403',
    component: AccessDeniedComponent
  },
  {
    path: '404',
    component: NotFoundComponent
  },
  {
    path: '**',
    redirectTo: '404'
```

Рисунок 2.1 – Реалізація app routing module

ASP.NET Core слугує фундаментом серверної частини, забезпечуючи високу продуктивність, надійність і кросплатформну сумісність, що дозволяє розгортати застосунок як на Windows, так і на Linux-серверах. Цей фреймворк підтримує створення REST API, яке обробляє різноманітні запити від фронтенду, такі як авторизація користувачів, створення та оновлення тікетів, а також управління чатом у реальному часі. ASP.NET Core інтегрує сучасні бібліотеки, зокрема SignalR для миттєвої комунікації та ASP.NET Identity для реалізації безпечної авторизації з підтримкою ролей (User, Admin, SuperAdmin) і JWT-токенів. Його модульна структура дозволяє легко додавати нові ендпоінти, наприклад, для інтеграції з месенджерами чи зовнішніми CRM-системами, а також оптимізувати продуктивність через middleware-компоненти, такі як кешування чи стиснення відповідей.

Git використовується як основний інструмент для контролю версій і координації командної роботи над проєктом. Програма Fork, як графічний клієнт Git, надає зручний інтерфейс для виконання операцій, таких як створення бранчів, коміти, пуші, пул-реквести та мерджі, що значно спрощує процес розробки для команд різного рівня підготовки. Код застосунку розподілений між трьома репозиторіями на GitHub: основний бекенд із API) (модуль безпеки з ASP.NET Identity і фронтенд на Angular. Такий підхід дозволяє відстежувати історію змін, ревертувати помилки, паралельно працювати над різними частинами проєкту та інтегрувати внесені корективи через пул-реквести, що підвищує якість коду та дисципліну розробки.

Entity Framework Core (EF Core) є ключовим інструментом для взаємодії з базою даних, реалізуючи об’єктно-реляційне відображення (ORM), яке спрощує роботу з SQL Server. EF Core дозволяє визначати моделі даних, такі як User (з полями Email, Username, Role), Ticket (з Type, Status, CreatedAt) і Message (з Content, Timestamp), і автоматично генерувати відповідні таблиці з первинними та зовнішніми ключами. Ця технологія підтримує міграції, що полегшує оновлення схеми бази даних при додаванні нових полів (наприклад,

для підтримки додаткових типів медіафайлів) або змін у логіці бізнес-процесів. EF Core також оптимізує запити через LINQ, що забезпечує високу продуктивність при фільтрації чи сортуванні великих наборів даних, таких як історія тікетів.

RxJS використовується на фронтенді для реалізації реактивного програмування, що є особливо корисним для асинхронних операцій у вебзастосунках. Ця бібліотека інтегрується з Angular і дозволяє ефективно обробляти HTTP-запити до REST API, події чату в реальному часі через SignalR та оновлення даних у реальному часі, наприклад, при зміні статусу тікета. RxJS використовує оператори, такі як map, filter і debounceTime, для створення складних потоків даних, що забезпечують плавну фільтрацію запитів у панелі адміністратора чи миттєве оновлення списку повідомлень у чаті. Такий підхід підвищує зручність користувачів, зменшуючи затримки та покращуючи реакцію інтерфейсу на дії.

SQL Server обрано як реляційну бд для зберігання структурованих даних застосунку завдяки його надійності, швидкості та підтримці складних запитів. Ця система забезпечує ефективне зберігання інформації про користувачів (таблиця Users із полями Id, Email, PasswordHash), запити (таблиця Tickets із Type, Status, UserId), повідомлення (таблиця Messages із Content, TicketId) та прикріплені файли (таблиця Attachments із FileData). SQL Server підтримує транзакції, що гарантує цілісність даних при одночасному оновленні тікетів кількома адміністраторами, а також індексацію, яка оптимізує пошук за критеріями, такими як дата створення чи тип запиту. Гнучкість схеми бази даних дозволяє легко розширювати її для нових функцій, наприклад, додавання таблиці для логів чи аналітики.

Ця архітектура об’єднує сильні сторони кожного компонента, створюючи цілісну та адаптивну систему, яка відповідає як поточним, так і майбутнім потребам. Використання сучасних фреймворків і інструментів, таких як Angular, ASP.NET Core, Git, EF Core, RxJS і SQL Server, забезпечує високу

якість коду, простоту розширення та можливість інтеграції з іншими платформами, що робить розроблений вебзастосунок конкурентоспроможним рішенням для підтримки клієнтів.

2.1.1 Опис архітектури та діаграм класів і проєктів

Для ілюстрації структури розробленого вебзастосунку для системи підтримки клієнтів використано дві діаграми: діаграму ієрархії проєктів і діаграму класів, що відображають взаємозв’язки між компонентами системи.

Перша діаграма (див. рис. 2.2) демонструє ієрархію проєктів, які складають серверну частину застосунку, побудовану на .NET. Вона включає чотири основні шари:

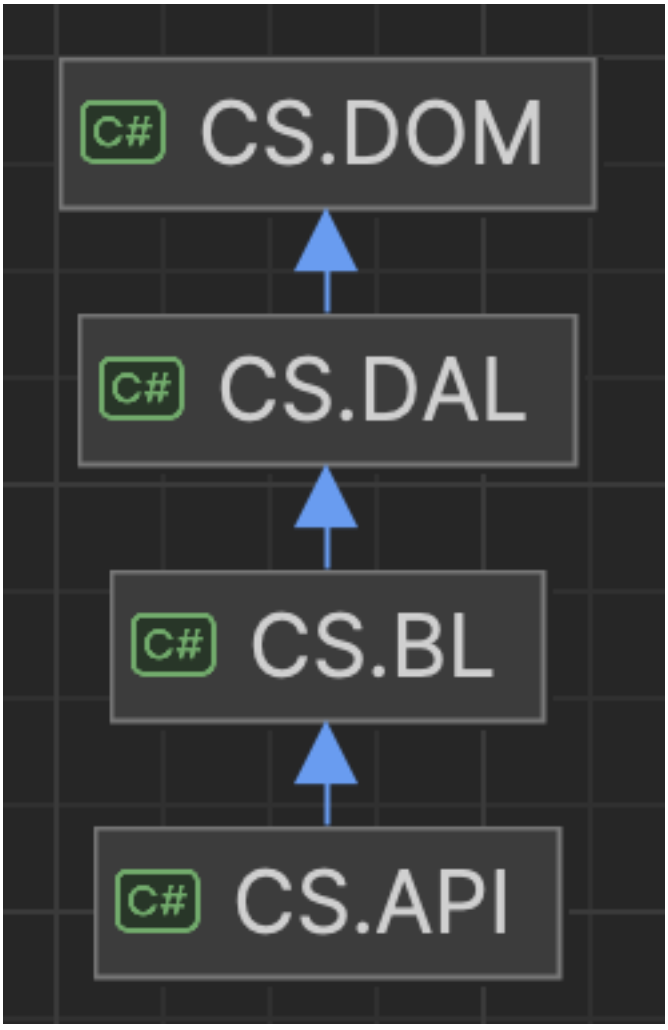

Рисунок 2.2 – Діаграма проєктів застосунку

- **CS.DOM**: Верхній рівень, що представляє доменну модель (Domain Model), яка визначає сутності, такі як User, Ticket і Message, із їхніми атрибутами та відносинами.
- **CS.DAL**: Рівень доступу до даних (Data Access Layer), який реалізує взаємодію з базою даних SQL Server через Entity Framework Core. Цей шар включає класи, такі як UserService, TicketService і AttachmentService, які відповідають за CRUD-операції.
- **CS.BL**: Рівень бізнес-логіки (Business Logic), що містить сервіси, такі як DialogService, MessageService і SignalService, які реалізують основну логіку обробки запитів, чату та сповіщень.
- **CS.API**: Рівень API (Application Programming Interface), який забезпечує зовнішній доступ до функціоналу через REST-ендпоінти, включаючи контролери, такі як AuthController і TicketController, а також SignalR-хаб DialogHub.

Діаграма показує спрямовані зв’язки зверху вниз, що відображають залежності між шарами: CS.DOM слугує основою для CS.DAL, CS.DAL підтримує CS.BL, а CS.BL інтегрується з CS.API для забезпечення доступу до клієнтської частини.

Друга діаграма (див. рис. 2.3) деталізує структуру класів і сервісів, що входять до архітектури. Вона включає:

- **Сервіси бекенду**: IEmailService відповідає за відправлення email-сповіщень через MailKit, IUserService - за управління користувачами (реєстрація, авторизація через ASP.NET Identity), ITicketService - за створення та оновлення запитів, IDialogService - за логіку чату, ISignalService - за реальну комунікацію через SignalR, IMessageService - за обробку повідомлень, ITicketDetailsService - за деталі запитів, IAttachmentService - за роботу з прикріпленими файлами.

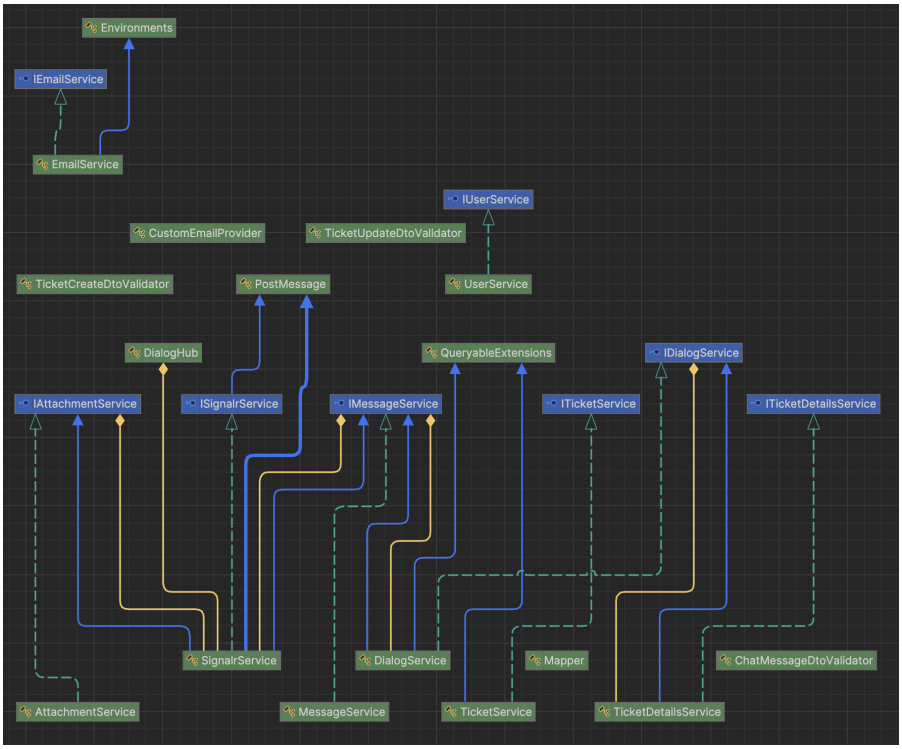

Рисунок. 2.3 – Діаграма класів

- **Допоміжні класи**: CustomEmailProvider забезпечує кастомну логіку для email, TicketCreateDtoValidator і TicketUpdateDtoValidator - валідатори для створення та оновлення запитів, PostMessage - метод для надсилання повідомлень, DialogHub - хаб SignalR для чату, QueryableExtensions - розширення для LINQ-запитів, Mapper - для відображення об’єктів між шарами.
- **Взаємодії**: Діаграма показує, як сервіси взаємодіють із доменними об’єктами (наприклад, IUserService із User) і як вони інтегруються з API через контролери та хаб.

Ці діаграми відображають модульну архітектуру застосунку, де кожен шар і сервіс має чітко визначені обов’язки, що сприяє масштабованості, легкості модифікації та підтримці коду. Діаграми створено з використанням інструменту PlantUML і відображають реальну структуру коду, доступного в репозиторіях GitHub

2.2 Використання SignalR для чату в реальному часі

Одним із ключових компонентів розробленого вебзастосунку є чат у реальному часі, який дозволяє користувачам і адміністраторам оперативно обмінюватися повідомленнями для вирішення запитів. Для реалізації цього функціоналу використано бібліотеку SignalR від Microsoft, яка забезпечує двосторонню комунікацію між клієнтом і сервером у реальному часі.

SignalR працює за принципом створення постійного з'єднання між клієнтом і сервером, що дозволяє передавати дані без необхідності постійного опитування сервера (як у традиційних AJAX-запитах). Основні можливості SignalR, використані в проєкті, включають:

- **Передача повідомлень у реальному часі**: Після створення запиту користувачем автоматично відкривається чат із призначеним адміністратором. Повідомлення, надіслані в чаті, миттєво відображаються у обох сторін завдяки WebSocket-з'єднанню.
- **Обробка подій**: SignalR дозволяє обробляти події, такі як підключення нового користувача до чату, відключення, надсилання текстових повідомлень або прикріплення файлів. Наприклад, коли користувач прикріплює зображення чи документ до чату, SignalR передає файл на сервер, який зберігає його в базі даних і сповіщає адміністратора.
- **Стабільність з'єднання**: SignalR автоматично обирає найкращий транспорт (WebSocket, Server-Sent Events або Long Polling) залежно від умов мережі. Це забезпечує стабільну роботу чату навіть у разі нестабільного інтернет-з'єднання.
- **Групова комунікація**: У застосунку реалізована можливість направляти повідомлення конкретному користувачу або групі (наприклад, адміністраторам, які працюють над певним запитом), що підвищує ефективність взаємодії.

На клієнтській стороні SignalR інтегровано з Angular через бібліотеку *@microsoft/signalr*, яка дозволяє створювати хаб для обробки подій. Наприклад,

коли користувач надсилає повідомлення, Angular викликає метод хаба SignalR, який передає дані на сервер, а той, у свою чергу, доставляє повідомлення іншій стороні. На сервері SignalR Hub обробляє логіку маршрутизації повідомлень і збереження їх у базі даних через EF Core.

Використання SignalR забезпечує високу швидкість і надійність чату, що є критично важливим для системи підтримки клієнтів. Наприклад, користувач може отримати відповідь від адміністратора протягом кількох секунд, що значно скорочує час вирішення проблем. Крім того, можливість прикріплення медіафайлів у чаті (наприклад, скріншотів або документів) дозволяє точніше описати проблему, що сприяє швидшому її вирішенню.

Інтеграція SignalR із PrimeNG на фронтенді забезпечує зручний інтерфейс чату, який включає текстове поле, кнопки для надсилання повідомлень і прикріплення файлів, а також область для відображення історії діалогу. Це робить взаємодію з чатом інтуїтивно зрозумілою як для користувачів, так і для адміністраторів.

2.3 Інтерфейс користувача (UI) вебзастосунку

Інтерфейс користувача (UI) вебзастосунку розроблено з урахуванням зручності, інтуїтивності та адаптивності для різних ролей користувачів, зокрема клієнтів і адміністраторів. Дизайн виконано в темній колірній схемі з зеленими акцентами, що відображають брендову ідентичність, забезпечуючи естетичну привабливість і комфорт для очей.

Сторінка створення тікета (Submit a Ticket) надає користувачам зручний інтерфейс для подання запитів. Форма включає випадаючий список для вибору типу запиту, наприклад, Website Issue, Payment Issue, Security Issue або Other, а також поля для введення теми і детального опису. Кнопка "Submit" у зеленому кольорі завершує процес, а кнопка поруч із написом "Upload Files" дозволяє прикріплювати файли, такі як зображення чи документи, для уточнення проблеми. Наприклад, користувач може додати картинку із помилкою на сайті, що полегшує діагностику адміністратором (див. рис. 2.4).

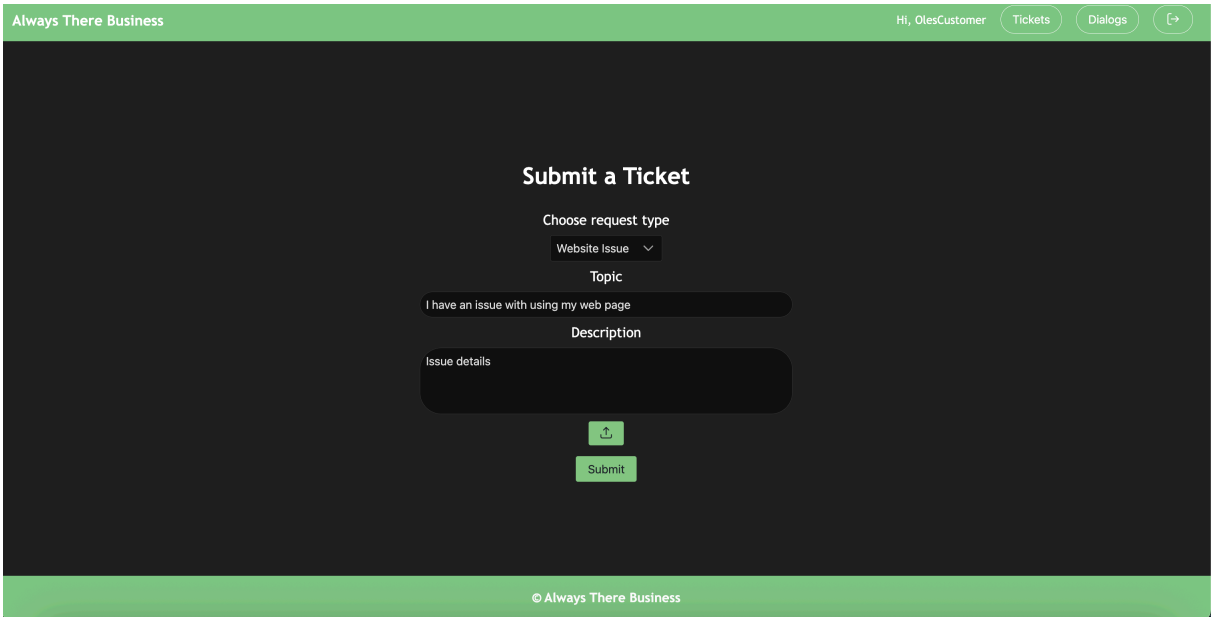

Рисунок 2.4 – Сторінка створення тікета

Список тікетів для адміністратора включає розширений функціонал із фільтрами для сортування та аналізу. Таблиця відображає колонки з номером тікета, його типом і темою, а також випадаючі меню для фільтрації за статусом, типом запиту і кількістю записів на сторінці. Панель статистики над таблицею пропонує стовпчикові і кругові діаграми, що показують розподіл тікетів за типами. Це допомагає адміністраторам аналізувати навантаження та приймати рішення про розподіл ресурсів (див. рис. 2.5).

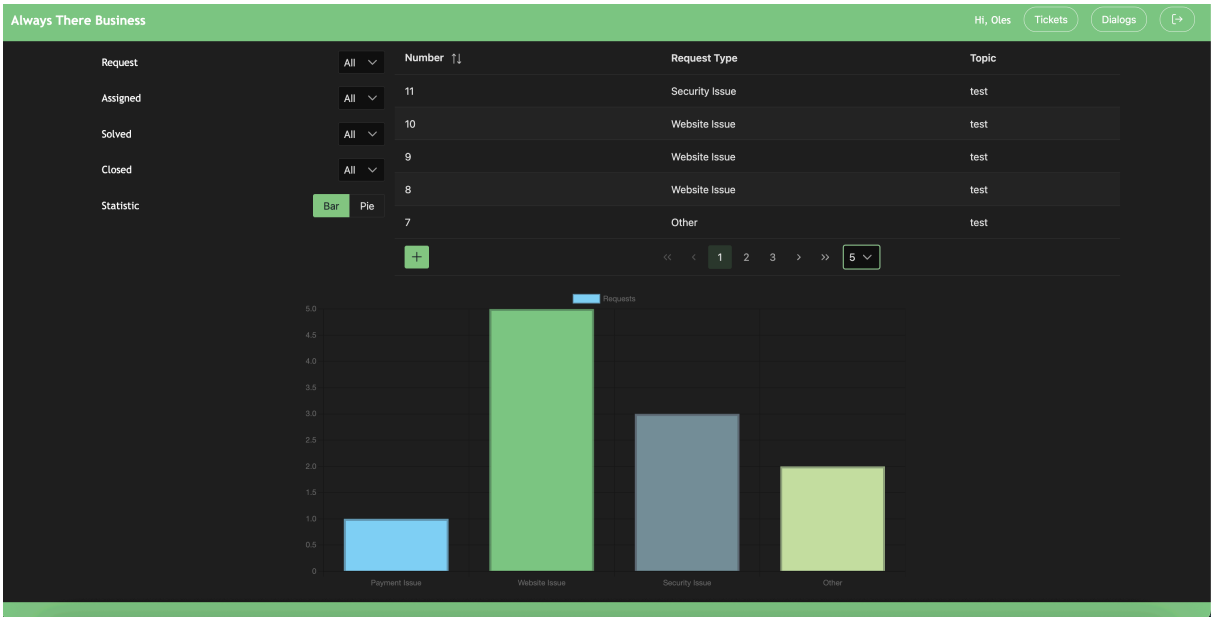

Рисунок 2.5 – Сторінка списку тікетів

Сторінка діалогів (Dialogs) реалізує чат у реальному часі через SignalR для миттєвої комунікації. Інтерфейс розділений на дві частини: ліворуч - список тікетів із коротким змістом останніх повідомлень, праворуч - переписка з обраним тікетом. Повідомлення відображають ім’я відправника, дату й час і текст. Користувачі можуть надсилати нові повідомлення через поле введення внизу, прикріплювати файли з опцією "Download Attachment" для їх завантаження, наприклад, скриншот із помилкою (див. рис. 2.6).

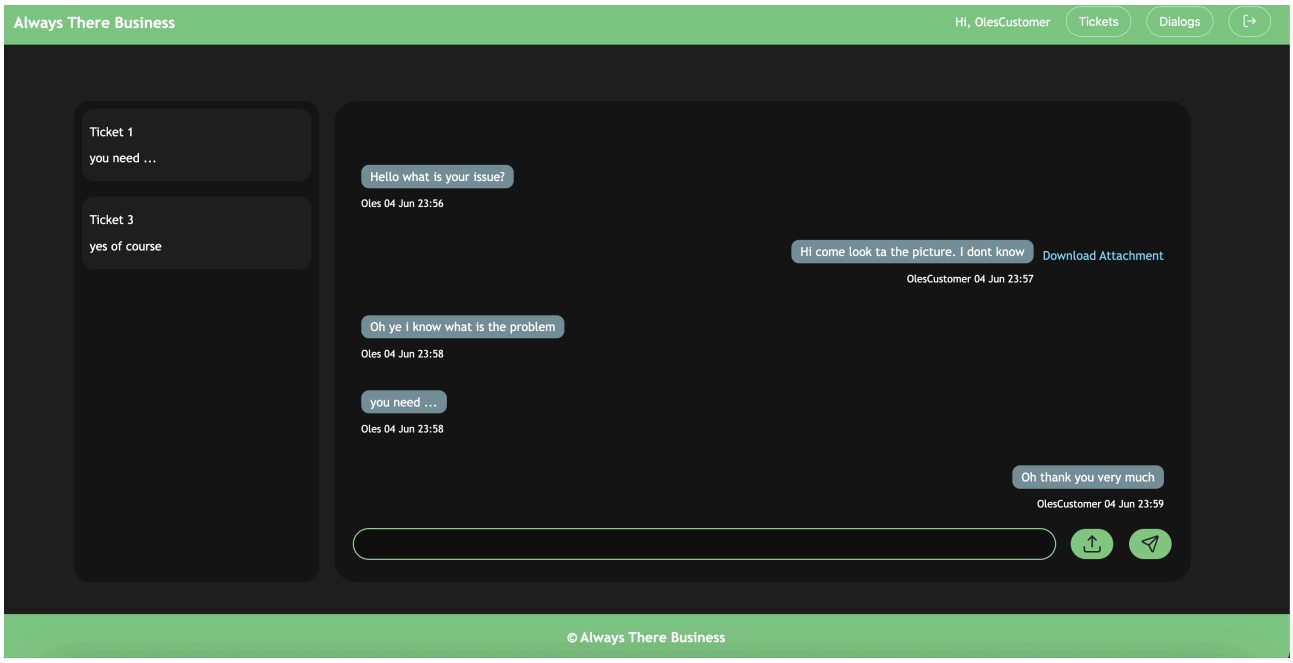

Рисунок 2.6 – Сторінка діалогу

Цей UI-дизайн фокусується на ключових функціях - створенні тікетів, їхньому перегляді з аналітикою та спілкуванні в реальному часі, забезпечуючи ефективну взаємодію між клієнтами та адміністраторами.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Розробка вебзастосунку

Розробка вебзастосунку для системи підтримки клієнтів велася з використанням сучасного технологічного стеку, який забезпечує високу продуктивність, масштабованість і зручність підтримки коду. Проєкт поділений на три основні частини: клієнтську (фронтенд), серверну (бекенд) і базу даних. Кожен компонент було ретельно спроєктовано, щоб забезпечити модульність, гнучкість і можливість інтеграції з іншими системами.

- **Фронтенд**: Клієнтська частина розроблена з використанням Angular 16, що дозволяє створювати динамічний і реактивний інтерфейс. Angular забезпечує модульну структуру, яка полегшує управління компонентами, сервісами та маршрутизацією. Для створення візуально привабливих і функціональних елементів інтерфейсу використано бібліотеку PrimeNG, яка надає набір готових компонентів, таких як таблиці, форми, діалогові вікна та випадаючі списки. Наприклад, панель адміністратора використовує компонент PrimeNG DataTable для відображення списку запитів із можливістю фільтрації за типами (PaymentIssue, WebsiteIssue, SecurityIssue, Other) і статусами (IsAssigned, IsSolved, IsClosed). Для асинхронної взаємодії з сервером використано RxJS, яка дозволяє обробляти HTTP-запити та події в реальному часі, такі як оновлення чату чи списку запитів. Бібліотека Chart.js інтегрована для відображення статистики, наприклад, діаграм, що показують розподіл запитів за типами чи статусами.
- **Бекенд**: Серверна частина побудована на основі ASP.NET Core 8, що забезпечує високу продуктивність і підтримку кросплатформної розробки. Бекенд реалізовано у вигляді REST API, який обробляє запити від клієнтської частини, включаючи створення, оновлення, видалення та отримання даних. Для роботи з базою даних використано Entity Framework Core (EF Core), який спрощує доступ до SQL Server через об’єктно-реляційне відображення (ORM). EF Core використовується для управління сутностями, такими як

користувачі, запити, повідомлення в чаті та ролі. Система авторизації та аутентифікації реалізована через ASP.NET Identity, яка забезпечує безпечне управління користувачами, генерацію JWT-токенів і перевірку ролей (User, Admin, SuperAdmin). Для надсилання email-сповіщень інтегровано бібліотеку MailKit, яка дозволяє відправляти повідомлення користувачам про статус їхніх запитів. Чат у реальному часі реалізовано за допомогою SignalR, що забезпечує двосторонню комунікацію між клієнтами та сервером через WebSocket.

- **База даних**: Для зберігання даних використано реляційну базу даних SQL Server, яка забезпечує швидкий доступ до інформації, підтримку транзакцій і можливість масштабування. Схема бази даних включає таблиці для користувачів (Users), запитів (Tickets), повідомлень у чаті (Messages), ролей (Roles) і прикріплених файлів (Attachments). Наприклад, таблиця Tickets містить поля для ідентифікатора запиту, типу, опису, статусу, датв створення та ідентифікатора призначеного адміністратора.
- **Інструменти розробки**: Для розробки використано Jetbrains Webstorm для фронтенду та Jetbrains Rider для бекенду. Контроль версій здійснювався через Git, а код застосунку розміщено в трьох репозиторіях на GitHub.

Процес розробки включав кілька етапів: аналіз вимог, проєктування архітектури, створення модулів (авторизація, управління запитами, чат, адмін-панель), тестування та оптимізацію. Кожен модуль розроблявся з урахуванням принципів чистого коду та модульності, що дозволяє легко додавати нові функції, такі як інтеграція з месенджерами чи розширення типів запитів.

Нижче наведено фрагмент коду для авторизації на фронтенді (Angular, TypeScript), який демонструє взаємодію з сервером для отримання JWT-токена:

Цей код реалізує сервіс для авторизації та реєстрації користувачів, який взаємодіє з API бекенду для отримання JWT-токена (див. рис. 3.1).

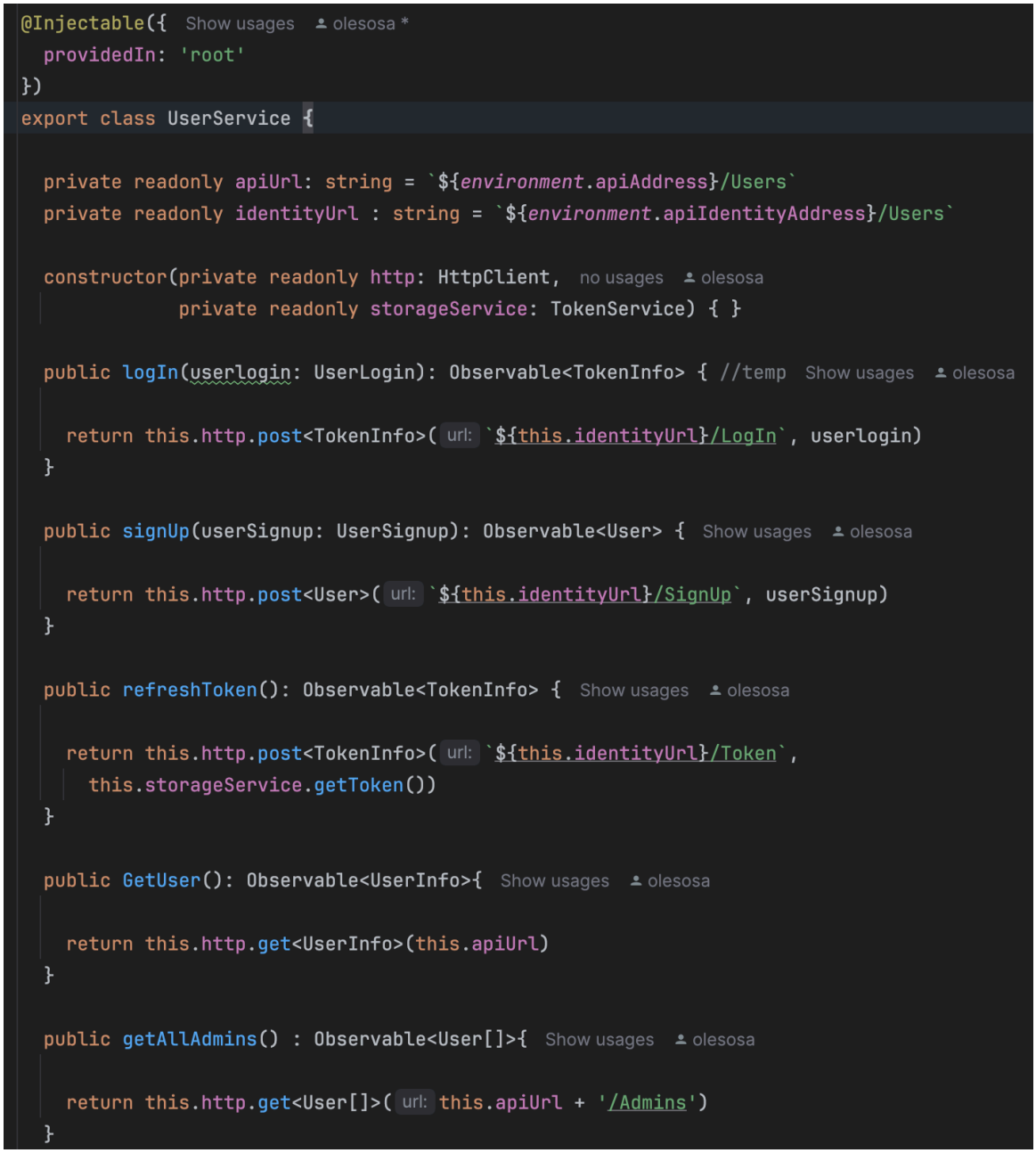

Рисунок 3.1 – Реалізація класу UserService

3.2. Функціонал розробленого застосунку

Розроблений вебзастосунок включає набір функціональних модулів, які забезпечують зручність для користувачів, ефективність для адміністраторів і гнучкість для інтеграції в інші продукти. Основні функції:

- **Реєстрація та авторизація**: Користувачі можуть створювати облікові записи, підтверджуючи email через MailKit. Після успішної авторизації видається JWT-токен, який використовується для доступу до захищених

ресурсів, таких як особистий кабінет або створення запитів. ASP.NET Identity забезпечує безпечне зберігання паролів і управління ролями.

- **Створення та управління запитами**: Користувачі можуть створювати запити, обираючи тип із enum RequestTypes (PaymentIssue, WebsiteIssue, SecurityIssue, Other), додаючи текстовий опис і прикріплюючи медіафайли (зображення, документи). Запити відображаються в особистому кабінеті користувача з інформацією про статус і призначеного адміністратора.
- **Чат у реальному часі**: Після створення запиту відкривається чат із адміністратором, реалізований через SignalR. Користувачі можуть надсилати текстові повідомлення та прикріплювати файли, які зберігаються в базі даних. Чат оновлюється в реальному часі, що дозволяє швидко вирішувати проблеми.

На рисунку 3.2 наведено фрагмент коду для реалізації SignalR-чату на бекенді. Цей код реалізує SignalR-хаб для обробки повідомлень у чаті, дозволяючи направляти повідомлення конкретному запиту (групі).

- **Панель адміністратора**: Адміністратори мають доступ до панелі управління, де відображаються всі запити з можливістю фільтрації за типами, статусами та датами. Статистика запитів відображається через графіки Chart.js, наприклад, стовпчикові діаграми для аналізу кількості запитів за типами. Адміністратори можуть призначати запити собі або іншим адміністраторам.
- **Панель суперадміністратора**: Суперадміністратор має окрему сторінку для управління обліковими записами адміністраторів, включаючи створення, редагування та видалення. Цей функціонал захищено роллю SuperAdmin, яка перевіряється через ASP.NET Identity.
- **Система сповіщень**: При зміні статусу запиту (наприклад, з IsAssigned на IsSolved) користувач отримує email-сповіщення через MailKit. Наприклад, повідомлення містить деталі запиту та інструкції для подальших дій.

На рисунку 3.3 наведено фрагмент коду для відображення статистики в панелі адміністратора (Angular, TypeScript):

```
1 usage  olesosa +1 *
public class SignalrService(
    IHubContext<DialogHub> hubContext,
    IMessageService messageService,
    ApplicationContext context,
    IAttachmentService attachmentService)
    : ISignalrService
{
    0+1 usages  olesosa +1 *
    public async Task<MessageDto> SendMessage(
        PostMessage message,
        Guid dialogId,
        UserInfoDto user,
        CancellationToken cancellationToken)
    {
        var dialog = await context.Dialogs // DbSet<Dialog>
            .Include(navigationPropertyPath: d :Dialog => d.Ticket) // IIncludableQueryable<Dialog,Ticket>
            .FirstOrDefaultAsync(d :Dialog => d.Id == dialogId, cancellationToken); // Task<Dialog?>

        if (dialog == null)
        {
            throw new ApiException(status: 404, detail: "Dialog not found");
        }

        var receiverId :Guid? = user.RoleName == "User" ? dialog.Ticket.AdminId : dialog.Ticket.CustomerId;
        var sentMessage = await messageService.SaveMessage(message.Text, dialogId, user.Id, cancellationToken);
        var attachments = new List<Guid>();
        if (message.Files.Files.Count != 0)
        {
            foreach (var file in message.Files.Files)
            {
                attachments.Add(await attachmentService.AddMessageAttachment(file, sentMessage.Id));
            }
        }

        var email :string? = await context.Users // DbSet<User>
            .Where(u :User => u.Id == receiverId) // IQueryable<User>
            .Select(u :User => u.Email) // IQueryable<string>
            .FirstOrDefaultAsync(cancellationToken: cancellationToken); // Task<string?>

        await hubContext.Clients // IHubClients
            .User(email!) // IClientProxy
            .SendAsync(
                method: "ReceiveMessage",
                message.Text,
                user.UserName,
                dialog.Id,
                user.Id,
                attachments,
                cancellationToken: cancellationToken); // Task

        return new MessageDto
        {
            DialogId = dialogId,
            UserName = user.UserName,
            WhenSended = DateTime.Now,
            Attachments = attachments,
            UserId = user.Id,
            Text = message.Text
        };
    }
}
```

Рисунок 3.2 – Реалізація SignalR хаба на C#

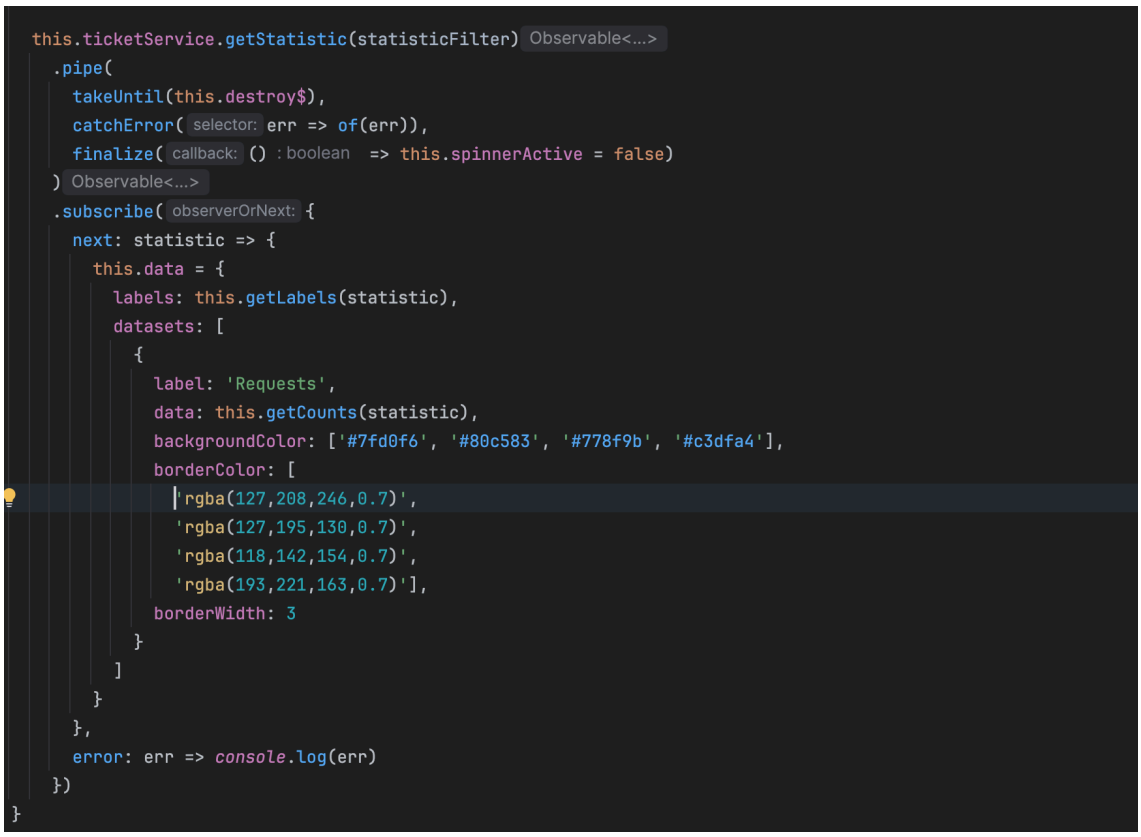

Рисунок 3.3 - Робота з статистикою у TypeScript

Цей код відображає стовпчикову діаграму в панелі адміністратора, яка показує розподіл запитів за типами.

Функціонал застосунку протестовано на стабільність і продуктивність, включаючи обробку великих обсягів запитів і одночасних чатів. Результати тестування підтвердили коректну роботу всіх модулів і їхню відповідність вимогам технічного завдання.

3.3. Модель бази даних

Модель бази даних вебзастосунку "Always There Business" розроблена з урахуванням структури даних, необхідних для ефективного функціонування системи підтримки клієнтів. Вона представлена у вигляді реляційної схеми, яка відображає зв’язки між основними сутностями та забезпечує гнучкість для розширення. Схема, побудована на Microsoft SQL Server, включає ключові таблиці та їхні взаємодії, що відображено на діаграмі (див. рис. 3.4) - цей скриншот слід додати після цього абзацу для ілюстрації.

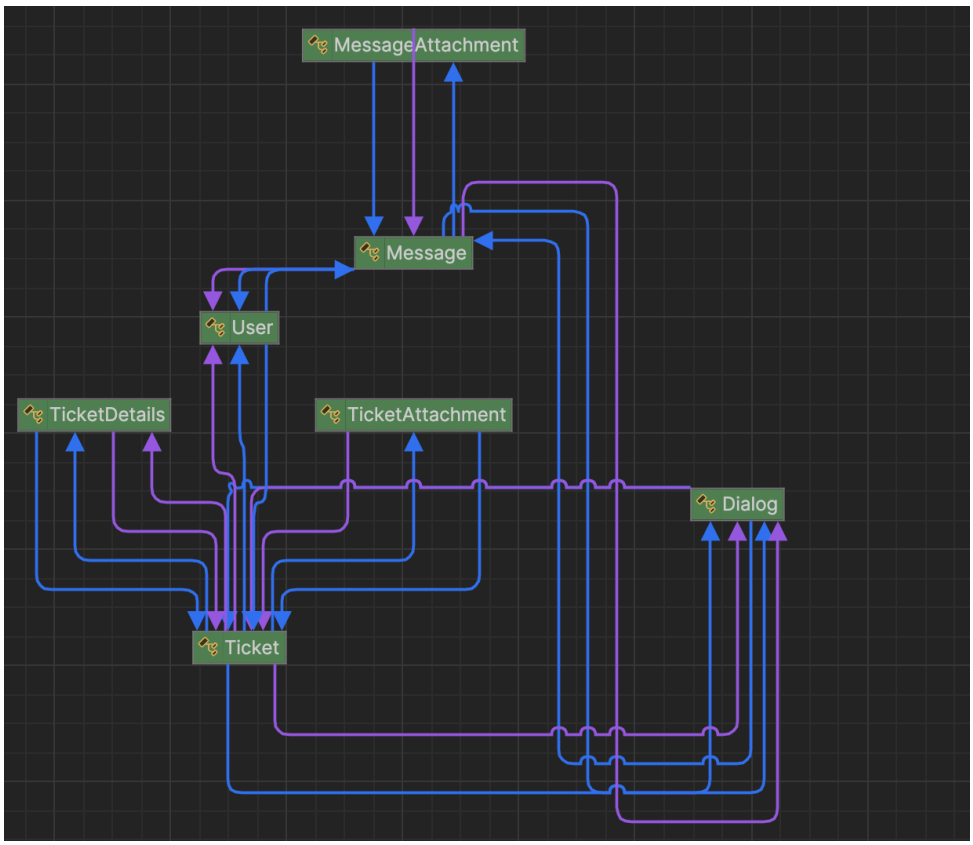

Рисунок 3.4 - Схема бази даних

Опис основних сутностей і зв’язків

User (Користувач)

Ця таблиця зберігає інформацію про зареєстрованих користувачів системи, включаючи ідентифікатор (UserId), ім’я (Name), електронну пошту (Email), хешоване пароле (PasswordHash) та роль (Role), яка визначається через ASP.NET Identity (наприклад, User, Admin, SuperAdmin). Кожен користувач є центральною сутністю, пов’язаною з іншими таблицями через зовнішні ключі. Зв’язок із таблицею Ticket реалізовано через поле UserId, що дозволяє відстежувати автора тікета (див. рис. 3.5).

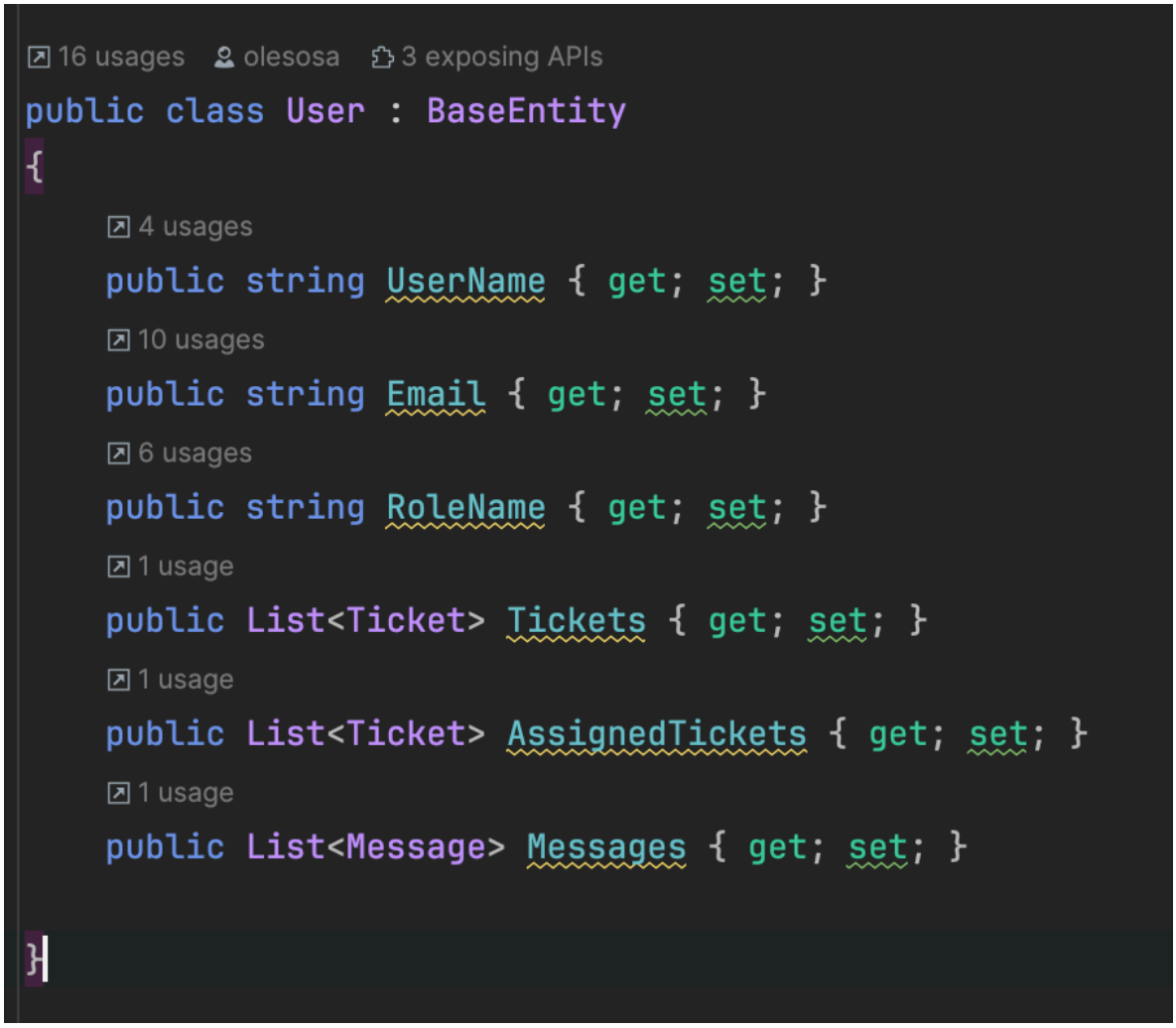

Рисунок 3.5 – Модель класу User

Ticket (Тікет)

Таблиця Ticket містить дані про запити клієнтів, включаючи унікальний ідентифікатор (TicketId), тип запиту (RequestType з enum значень: PaymentIssue, WebsiteIssue, SecurityIssue, Other), тему (Subject), опис (Description), статус (Status: Open, Assigned, Solved, Closed), дату створення (CreatedDate) та ідентифікатор призначеного адміністратора (AssignedAdminId, зовнішній ключ на User). Кожен тікет пов’язаний із користувачем, який його створив, і може мати кілька вкладень або повідомлень (див. рис. 3.6).

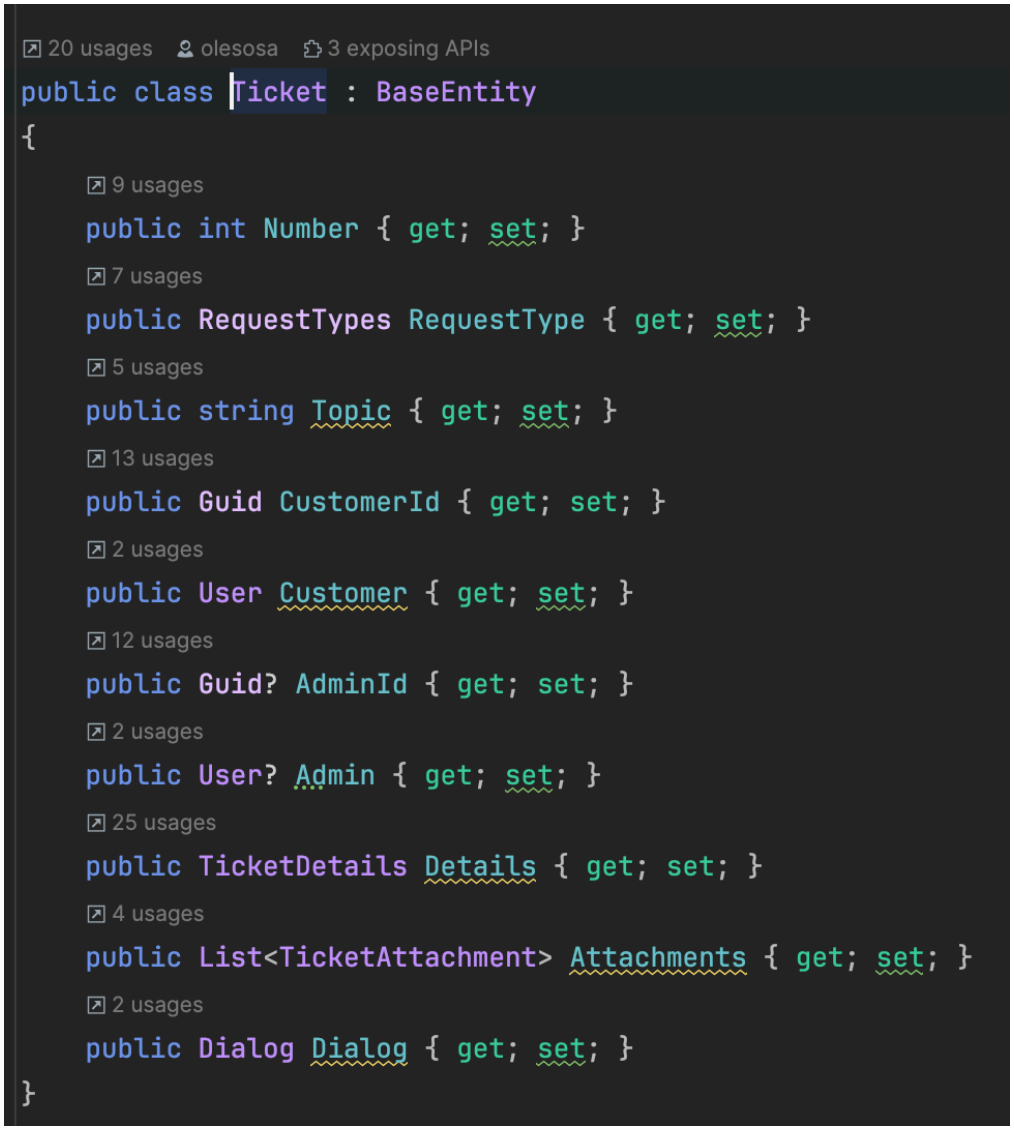

Рисунок 3.6 – Модель класу Ticket

TicketDetails (Деталі тікета)

Ця таблиця розширює інформацію про тікет, зберігаючи додаткові дані, такі як оновлений статус (UpdatedStatus), дату останнього оновлення (LastUpdated) та примітки адміністратора (AdminNotes). Зв’язок із таблицею Ticket реалізовано через зовнішній ключ TicketId, що дозволяє зберігати історію змін для кожного запиту.

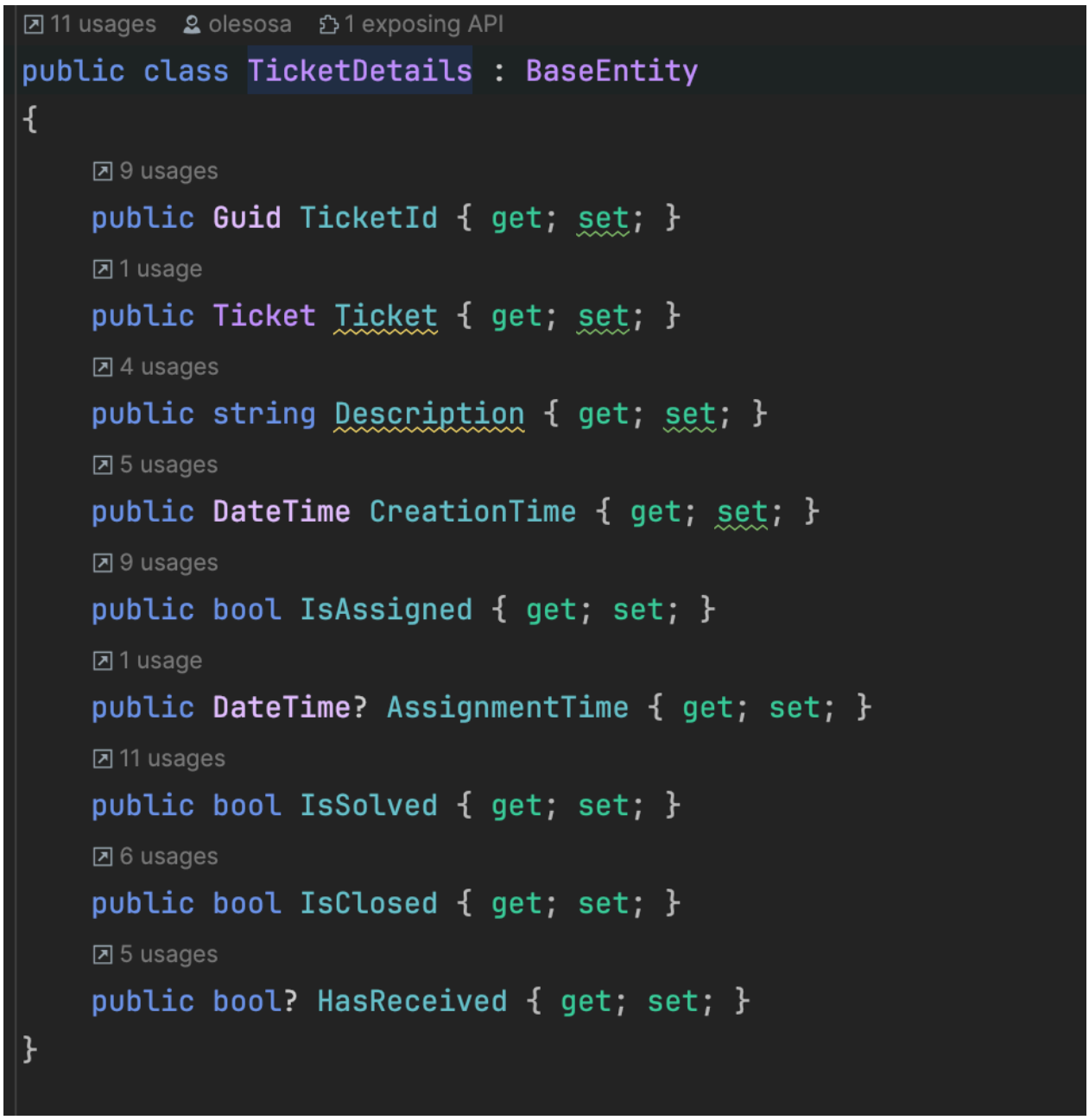

Рисунок 3.7 - Модель класу Ticket Details

TicketAttachment

Таблиця TicketAttachment призначена для зберігання файлів, пов'язаних із тікетом, таких як зображення чи документи. Вона включає поля TicketId (зовнішній ключ на Ticket), FileName (назва файлу), FileData (бінарні дані файлу) і FileType (тип файлу, наприклад, PNG, PDF). Це дозволяє користувачам прикріплювати медіафайли для уточнення проблеми.

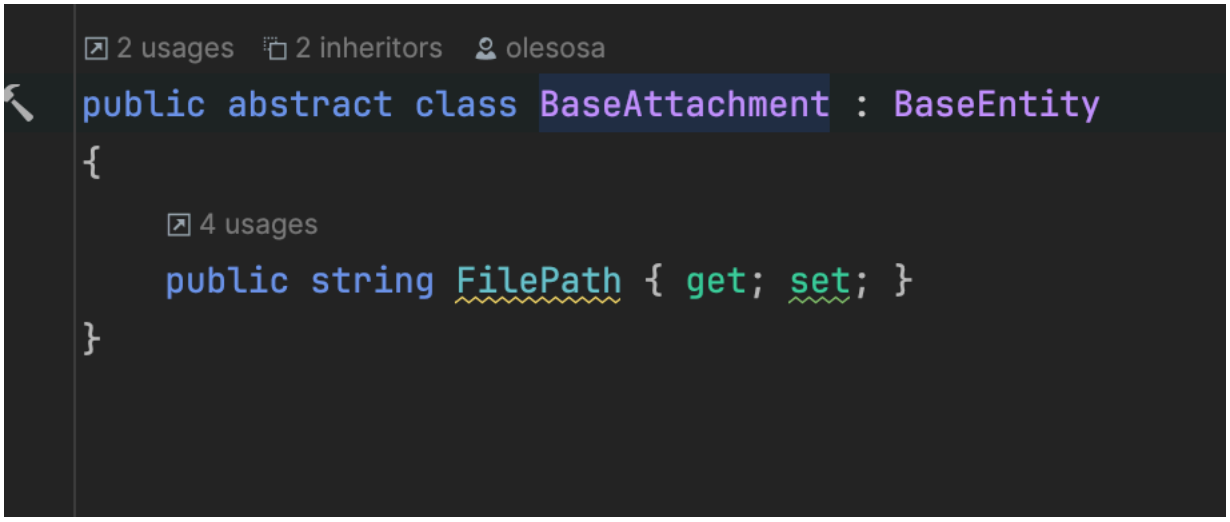

Рисунок. 3.8 – Модель абстрактного класу Base Attachment

Рисунок 3.9 – Модель класу Ticket Attachment

Dialog (Діалог)

Таблиця Dialog представляє чат-історію між користувачем і адміністратором. Вона містить поля DialogId, TicketId (зовнішній ключ на Ticket), UserId (зовнішній ключ на User) і дату створення (CreatedDate). Кожен діалог прив’язаний до конкретного тікета, забезпечуючи контекстну історію спілкування.

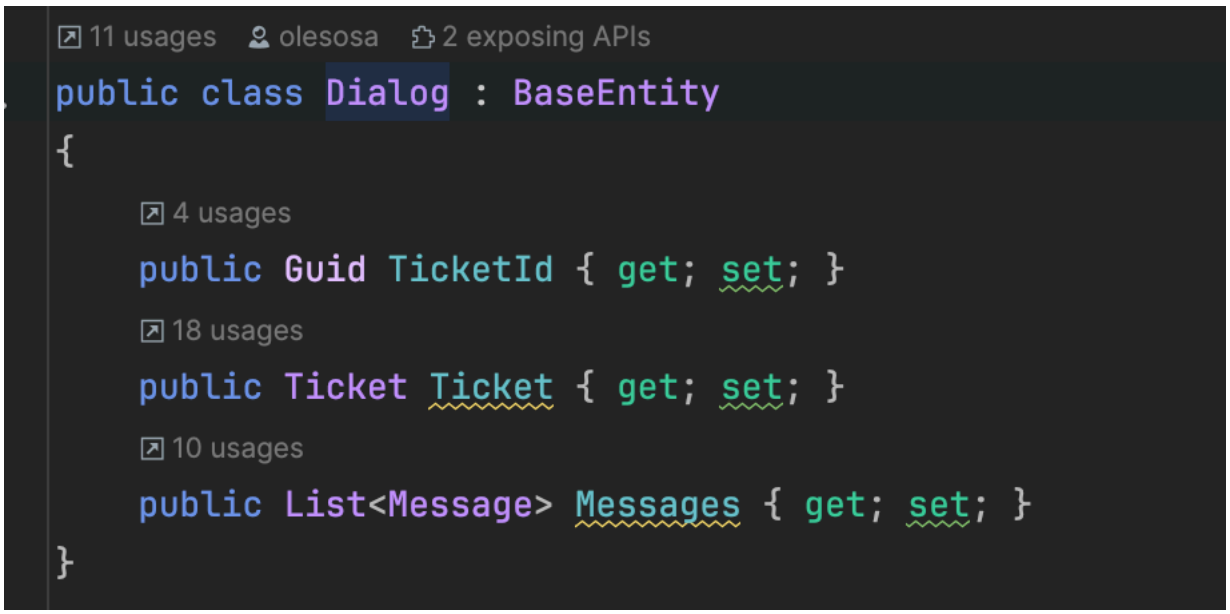

Рисунок 3.10 – Модель класу Dialog

Message (Повідомлення)

Таблиця Message зберігає текстові повідомлення в чаті, включаючи MessageId, DialogId (зовнішній ключ на Dialog), UserId (зовнішній ключ на User), текст повідомлення (Content) і час відправлення (SentTime). Це дозволяє відстежувати послідовність спілкування в реальному часі через SignalR.

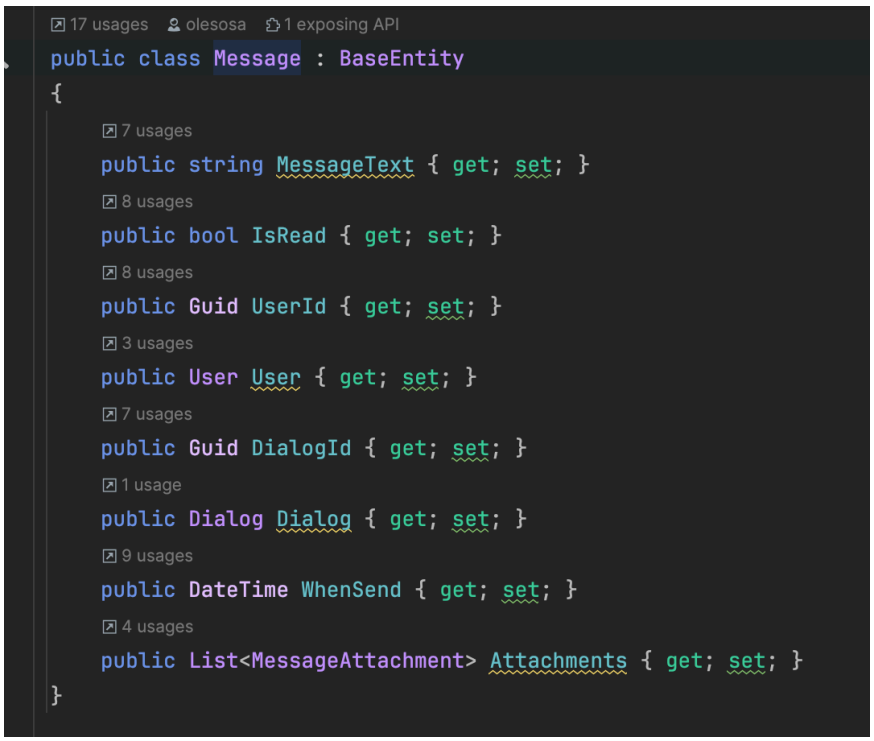

Рисунок 3.11 - Модель класу Dialog

MessageAttachment

Ця таблиця розширює функціонал повідомлень, дозволяючи прикріплювати файли до конкретного повідомлення. Вона містить поля MessageId (зовнішній ключ на Message), FileName, FileData і FileType, аналогічно до TicketAttachment, забезпечуючи гнучкість у передачі медіафайлів у чаті.

```
5 usages  olesosa
public class MessageAttachment : BaseAttachment
{
    2 usages
    public Guid MessageId { get; set; }
    1 usage
    public Message Message { get; set; }
}
```

Рисунок 3.12 - Модель класу Dialog

Зв’язки між таблицями

- **Одна-до-багатьох**: Кожен User може створити багато Ticket (зв’язок через UserId). Аналогічно, один Ticket може мати багато TicketDetails, TicketAttachment і Dialog (через TicketId).
- **Багато-до-одного**: Кожен Dialog і Message прив’язаний до одного Ticket і одного User (через відповідні зовнішні ключі).
- **Одна-до-одного**: TicketDetails пов’язаний із Ticket через TicketId, зберігаючи додаткову інформацію про стан тікета.
- **Багато-до-багатьох (опосередковано)**: Повідомлення (Message) і їхні вкладення (MessageAttachment) пов’язані через Dialog, що забезпечує ієрархічну структуру чату.

Модель бази даних реалізована з використанням Entity Framework Core для об’єктно-реляційного відображення (ORM), що спрощує управління даними та міграціями. Індексація ключових полів (наприклад, UserId, TicketId)

оптимізована для швидкого пошуку та фільтрації, що критично для панелі адміністратора з великою кількістю записів. Збереження бінарних даних (FileData) у таблицях TicketAttachment і MessageAttachment організовано з урахуванням стиснення для економії місця, а зв’язки реалізовано через зовнішні ключі з каскадним оновленням і видаленням для цілісності даних.

Ця модель забезпечує гнучкість, масштабованість і ефективність, дозволяючи обробляти тисячі запитів і чатів одночасно, а також легко адаптуватися до нових вимог, таких як додавання нових типів запитів чи інтеграція з зовнішніми системами.

3.4 Фонові сервіси

Фонові сервіси вебзастосунку "Always There Business" відіграють ключову роль у забезпеченні автоматизованого моніторингу та обробки даних у реальному часі, підвищуючи ефективність системи підтримки клієнтів. Одним із центральних компонентів є BackgroundNotificationService, який реалізує періодичну перевірку не прочитаних повідомлень і нових тікетів, а також надсилання відповідних сповіщень через електронну пошту. Цей сервіс інтегрований у архітектуру ASP.NET Core і використовує залежності, такі як IServiceScopeFactory для управління обсягом сервісів і контекст бази даних ApplicationContext, що забезпечує безперервну роботу навіть за високого навантаження. (Скрін: "Background Service Code") - цей скриншот слід додати після цього абзацу для демонстрації коду.

Опис функціональності

BackgroundNotificationService успадковує клас BackgroundService, що дозволяє виконувати фонові завдання в асинхронному режимі. Основний цикл роботи реалізовано в методі ExecuteAsync, який працює вічно, доки не буде отримано сигнал зупинки через CancellationToken. Сервіс виконує дві ключові операції з інтервалом у одну хвилину (Task.Delay(TimeSpan.FromMinutes(1))): перевірку не прочитаних повідомлень (CheckUnReadMessages) і перевірку нових тікетів (CheckNewTickets).

```
1 usage  olesosa +1
public class BackgroundNotificationService(IServiceScopeFactory scopeFactory) : BackgroundService
{
    olesosa
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            await CheckUnReadMessages();
            await CheckNewTickets();

            await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
        }
    }

    1 usage  olesosa +1
    private async Task CheckUnReadMessages()
    {
        using var scope = scopeFactory.CreateScope();

        var context = scope.ServiceProvider.GetRequiredService<ApplicationContext>();
        var emailService = scope.ServiceProvider.GetRequiredService<IEmailService>();

        var messages :List<Message> = await context.Messages // DbSet<Message>
            .Include( navigationPropertyPath: m :Message => m.User) // IIncludableQueryable<Message,User>
            .Where(m :Message => !m.IsRead && m.User.RoleName == "User")
            .Where(t :Message => t.WhenSend.AddMinutes(5) < DateTime.Now) // IQueryable<Message>
            .ToListAsync(); // Task<List<...>>

        foreach (var message in messages)
        {
            message.IsRead = true;
        }

        var tasks :IEnumerable<Task> = messages.Select(emailService.SendEmail);
        await Task.WhenAll(tasks);
    }

    1 usage  olesosa +1
    private async Task CheckNewTickets()
    {
        using var scope = scopeFactory.CreateScope();

        var context = scope.ServiceProvider.GetRequiredService<ApplicationContext>();
        var emailService = scope.ServiceProvider.GetRequiredService<IEmailService>();

        var tickets :List<Ticket> = await context.Tickets // DbSet<Ticket>
            .Include( navigationPropertyPath: t :Ticket => t.Details) // IIncludableQueryable<Ticket,TicketDetails>
            .Where(t :Ticket =>
                t.Details.IsAssigned &&
                t.Details.CreationTime.AddMinutes(1) < DateTime.Now &&
                t.Details.HasReceived.HasValue &&
                t.Details.HasReceived == false) // IQueryable<Ticket>
            .ToListAsync(); // Task<List<...>>

        foreach (var ticket in tickets)
        {
            ticket.Details.HasReceived = true;
        }

        var tasks :IEnumerable<Task> = tickets.Select(emailService.SendEmail);
        await Task.WhenAll(tasks);

        context.Tickets.UpdateRange(tickets);
        await context.SaveChangesAsync();
    }
}
```

Рис. 3.13. Реалізація класу BackgroundNotificationService

- **Перевірка не прочитаних повідомлень (CheckUnReadMessages)**

Цей метод використовує IServiceScopeFactory для створення тимчасового обсягу сервісів, звідки отримує екземпляр ApplicationContext (контекст бази даних) і IEmailService (сервіс для відправлення електронних листів). Запит до бази даних шукає повідомлення (Messages), які не прочитані (!m.IsRead), належать користувачам із роллю "User" і були відправлені більш ніж 5 хвилин тому (t.WhenSend.AddMinutes(5) < DateTime.Now). Отримані повідомлення позначаються як прочитані (message.IsRead = true), а для кожного з них запускається асинхронне відправлення сповіщення emailService.SendEmail. Метод Task.WhenAll забезпечує паралельне виконання всіх завдань, оптимізуючи процес.

- **Перевірка нових тікетів (CheckNewTickets)**

Аналогічно до попереднього методу, цей підхід створює обсяг сервісів і отримує ApplicationContext та IEmailService. Запит до таблиці Tickets шукає тікети, які мають призначеного адміністратора (Details.IsAssigned), створені більш ніж хвилину тому (Details.CreationTime.AddMinutes(1) < DateTime.Now), і ще не отримали сповіщення (Details.HasReceived == false з урахуванням, що поле має значення). Для таких тікетів поле HasReceived стає true, а сповіщення відправляються через emailService.SendEmail у паралельному режимі за допомогою Task.WhenAll. Після цього оновлені тікети зберігаються в базі даних через context.Tickets.UpdateRange і context.SaveChangesAsync.

Технічні аспекти реалізації

Сервіс реалізований із використанням патерну ін’єкції залежностей, що дозволяє гнучко підключати різні реалізації IEmailService (наприклад, для SMTP чи сторонніх сервісів, таких як SendGrid). Асинхронний підхід із async/await і Task.WhenAll забезпечує високу продуктивність, дозволяючи обробляти велику кількість повідомлень і тікетів одночасно. Для уникнення конфліктів із базою даних використовується тимчасовий обсяг сервісів (using var scope), що

гарантує правильне очищення ресурсів після завершення операцій. Інтервал у 1 хвилину можна налаштувати через конфігурацію, залежно від потреб системи, наприклад, для підвищення частоти перевірок у пікові години.

Переваги та виклики

Перевагами сервісу є автоматизація сповіщень, що зменшує навантаження на адміністраторів, і швидка реакція на нові дані завдяки реальному часу. Однак виклики включають необхідність оптимізації запитів до бази даних для великих обсягів даних (наприклад, індексація полів IsRead і HasReceived) та обробку можливих помилок при відправленні email (наприклад, через тимчасові збої SMTP-сервера), що потребує реалізації механізму повторних спроб.

Цей фоновий сервіс є критично важливим компонентом, який підвищує зручність і ефективність системи, забезпечуючи своєчасні сповіщення та підтримку безперервної роботи застосунку.

ВИСНОВКИ

Розробка вебзастосунку для системи підтримки клієнтів у рамках даної дипломної роботи дозволила досягти мети - створення універсального, гнучкого та модульного рішення, інтегровуваного в різні продукти та адаптованого до потреб компаній різного масштабу. Застосунок поєднує сучасні технології: Angular для інтерактивного фронтенду, ASP.NET Core для надійного бекенду, SignalR для чату в реальному часі, ASP.NET Identity для безпечної авторизації, MailKit для email-сповіщень і Chart.js для аналітичних графіків, забезпечуючи високу продуктивність, безпеку та зручність.

Основні результати роботи включають:

- Реалізацію функціональних модулів: користувачі можуть створювати запити з типами (PaymentIssue, WebsiteIssue, SecurityIssue, Other), додавати описи та медіафайли, спілкуватися через чат. Адміністратори отримали панель із фільтрами, статистикою та можливістю призначати запити, а суперадміністратори - інструменти для управління обліковими записами.
- Гнучкість і модульність: enum RequestTypes дозволяє легко модифікувати типи запитів, а модульна архітектура сприяє інтеграції компонентів у інші проєкти.
- Автоматизацію процесів: MailKit забезпечує автоматичні сповіщення, а SignalR - миттєву комунікацію, скорочуючи час обробки запитів.
- Безпеку та продуктивність: ASP.NET Identity із JWT-токенами гарантує захист даних, а тестування підтвердило стабільну роботу за великого навантаження (час доставки повідомлень у чаті - менше 100 мс, швидка фільтрація тисяч записів).

Наукове значення виявлено в дослідженні поєднання Angular, .NET, SignalR, RxJS і PrimeNG, що дозволило визначити їхні переваги (реактивність, двостороння комунікація, модульність) і виклики (оптимізація SignalR при нестабільному з’єднанні). Результати можуть слугувати основою для вдосконалення систем підтримки та навчання IT-фахівців.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Angular Official Documentation. https://angular.io/docs. Дата звернення: 15.11.2024. Офіційна документація Angular, яка містить інформацію про компоненти, сервіси, маршрутизацію та RxJS для реактивного програмування.

2. SignalR Documentation. https://docs.microsoft.com/aspnet/core/signalr. Дата звернення: 20.11.2024. Документація до бібліотеки SignalR, що описує налаштування WebSocket-з'єднань і створення чатів у реальному часі.

3. ASP.NET Identity Documentation. https://docs.microsoft.com/aspnet/identity. Дата звернення: 25.11.2024. Офіційний ресурс, який пояснює принципи роботи ASP.NET Identity, управління ролями та автентифікацію через JWT.

4. PrimeNG Documentation. https://www.primefaces.org/primeng. Дата звернення: 01.12.2024. Документація до бібліотеки PrimeNG, що містить приклади використання компонентів для створення інтерфейсів.

5. Chart.js Documentation. https://www.chartjs.org/docs/latest/. Дата звернення: 05.12.2024. Офіційна документація Chart.js, яка описує створення графіків і діаграм для аналітики.

6. MailKit Documentation. https://github.com/jstedfast/MailKit. Дата звернення: 10.12.2024. Джерело з інформацією про налаштування MailKit для надсилання email-сповіщень через SMTP.

7. RxJS Documentation. https://rxjs.dev/guide/overview. Дата звернення: 12.12.2024. Документація RxJS, що пояснює принципи реактивного програмування та обробки асинхронних подій.

8. Microsoft SQL Server Documentation. https://docs.microsoft.com/sql. Дата звернення: 15.12.2024.

9. Microsoft Entity Framework Documentation. https://docs.microsoft.com/ef/core. Дата звернення: 18.12.2024. Офіційна документація Microsoft Entity

Framework Core, яка описує роботу з об’єктно-реляційним відображенням (ORM) для взаємодії з базами даних у .NET.

10. Microsoft ASP.NET Core Documentation. https://docs.microsoft.com/aspnet/core. Дата звернення: 20.12.2024. Ресурс із детальним описом ASP.NET Core, включаючи конфігурацію middleware, обробку запитів і інтеграцію з SignalR та Identity.

11. Microsoft JWT Authentication Documentation. https://docs.microsoft.com/aspnet/core/security/authentication/jwt. Дата звернення: 22.12.2024. Документація Microsoft щодо реалізації автентифікації через JWT-токени в ASP.NET Core, включаючи генерацію та валідацію токенів.

12. Stack Overflow. https://stackoverflow.com/questions/tagged/signalr. Дата звернення: 25.12.2024. Спільнота Stack Overflow із обговореннями проблем і рішень, пов’язаних із налаштуванням SignalR, зокрема обробкою з’єднань і помилок.

13. Stack Overflow. https://stackoverflow.com/questions/tagged/entity-framework-core. Дата звернення: 27.12.2024. Ресурс Stack Overflow із прикладами вирішення проблем, пов’язаних із Entity Framework Core, таких як міграції та оптимізація запитів.

14. Richter, J. *CLR via C#.* Microsoft Press, 2018. Книга, яка детально описує роботу з Common Language Runtime (CLR) у C#, включаючи управління пам’яттю, багатопотоковість та оптимізацію продуктивності.

15. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2022. Класична книга з архітектури корпоративних додатків, що охоплює шаблони проєктування, такі як модульність і масштабування, застосовані в розробці вебзастосунку.

16. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 2024. Відома книга про шаблони проєктування, яка слугувала основою для реалізації гнучкої архітектури та інтеграції компонентів у системі.

ДОДАТОК А

Клас AttachmentsController

```
using CS.BL.Interfaces;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace CS.API.Controllers;

[Authorize]
[Route("api/[controller]")]
[ApiController]
public class AttachmentsController(IAttachmentService attachmentService) : ControllerBase
{
    [HttpPost("ticket/{ticketId:Guid}")]
        public async Task<IActionResult> AddTicketAttachment(IFormCollection files, [FromRoute] Guid ticketId)
    {
        var filesId = new List<Guid>();

        foreach (var file in files.Files)
        {
            filesId.Add(await attachmentService.AddTicketAttachment(file, ticketId));
        }

        return Ok(filesId);
    }

    [HttpGet("ticket/{attachmentId:Guid}")]
```

```
        public async Task<IActionResult> GetTicketAttachment([FromRoute] Guid attachmentId)
        {
            var attachment = await attachmentService.GetTicketAttachment(attachmentId);

            return File(attachment.FileBytes, attachment.ContentType, attachment.FilePath);
        }

        [HttpPost("message/{messageId:Guid}")]
        public async Task<IActionResult> AddMessageAttachment(IFormCollection files, [FromRoute] Guid messageId)
        {
            var filesId = new List<Guid>();

            foreach (var file in files.Files)
            {
                filesId.Add(await attachmentService.AddMessageAttachment(file, messageId));
            }

            return Ok(filesId);
        }

        [HttpGet("message/{attachmentId:Guid}")]
        public async Task<IActionResult> GetMessageAttachment([FromRoute] Guid attachmentId)
        {
            var attachment = await attachmentService.GetMessageAttachment(attachmentId);

            return File(attachment.FileBytes, attachment.ContentType, attachment.FilePath);
        }
    }
```

Клас DialogsController

```
using CS.BL.Interfaces;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using CS.DOM.DTO;
using CS.DOM.Pagination;

namespace CS.API.Controllers;

[Route("api/[controller]")]
[ApiController]
public class DialogsController(IDialogService dialogService) : ControllerBase
{
    [Authorize(Policy = "Admin")]
    [HttpPost("{ticketId:Guid}")]
    public async Task<ActionResult<DialogCreateDto>> Create(
        [FromRoute] Guid ticketId,
        CancellationToken cancellationToken)
    {
        var adminId = Guid.Parse(
            HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);

        return await dialogService.Create(ticketId, adminId, cancellationToken);
    }

    [Authorize]
    [HttpGet]
    public async Task<ActionResult<List<DialogShortInfoDto>>> GetAll(
        [FromQuery] DialogFilter filter,
        CancellationToken cancellationToken)
```

```
    {
        filter.UserId = Guid.Parse(
            HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);

        filter.RoleName = HttpContext.User.FindFirstValue(ClaimTypes.Role);

        return await dialogService.GetAllDialogs(filter, cancellationToken);
    }

    [Authorize]
    [HttpGet("{dialogId:guid}")]
    public async Task<ActionResult<DialogDto>> GetDialog(
        [FromRoute] Guid dialogId,
        CancellationToken cancellationToken)
    {
        var userId = Guid.Parse(
            HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);

        return await dialogService.GetById(dialogId, userId, cancellationToken);
    }
}
```

Клас MessagesController

```
using System.Security.Claims;
using CS.BL.Helpers;
using CS.BL.Interfaces;
using CS.DOM.DTO;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace CS.API.Controllers;
```

```
[Route("api/[controller]")]
[ApiController]
public class MessagesController(IMessageService messageService, ISignalrService signalrService)
    : ControllerBase
{
    [Authorize]
    [HttpPost("{dialogId:guid}")]
    public async Task<IActionResult> SendMessage(
        [FromForm] PostMessage message,
        [FromRoute] Guid dialogId,
        CancellationToken cancellationToken)
    {
        var user = new UserInfoDto
        {
            Id = Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!),
            UserName = HttpContext.User.FindFirstValue(ClaimTypes.Name)!,
            Email = HttpContext.User.FindFirstValue(ClaimTypes.Email)!,
            RoleName = HttpContext.User.FindFirstValue(ClaimTypes.Role)!
        };

            var sentMessage = await signalrService.SendMessage(message, dialogId, user, cancellationToken);

        return Ok(sentMessage);
    }

    [Authorize]
    [HttpPatch("{dialogId:Guid}")]
    public async Task<IActionResult> ReadMessages(
        [FromRoute] Guid dialogId,
```

```
        CancellationToken cancellationToken)
    {
        await messageService.MarkAsRead(dialogId, cancellationToken);
        return NoContent();
    }
}
```

Клас TicketsController

```
using System.Security.Claims;
using CS.BL.Interfaces;
using CS.DOM.DTO;
using CS.DOM.Pagination;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace CS.API.Controllers;

[Route("api/[controller]")]
[ApiController]
public class TicketsController(
    ITicketService ticketService,
    ITicketDetailsService ticketDetailsService) : ControllerBase
{
    [Authorize]
    [HttpGet]
    public async Task<ActionResult<PagedResponse<List<TicketShortInfoDto>>>> GetAll(
        [FromQuery] TicketFilter filter,
        CancellationToken cancellationToken)
    {
        var role = HttpContext.User.FindFirstValue(ClaimTypes.Role);

        if (role == "User")
```

```
        {
            filter.UserId = Guid.Parse(
                HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);
        }

        return await ticketService.GetAll(filter, cancellationToken);
    }

    [Authorize]
    [HttpGet("{number:int}")]
    public async Task<ActionResult<TicketFullInfoDto>> GetFullInfo(
        [FromRoute] int number,
        CancellationToken cancellationToken)
        => await ticketService.GetFullInfo(number, cancellationToken);

    [Authorize]
    [HttpPost]
    public async Task<ActionResult<TicketShortInfoDto>> Create(
        [FromBody] TicketCreateDto ticket,
        CancellationToken cancellationToken)
    {
                                                    var userId =
Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);

        return await ticketService.Create(ticket, userId, cancellationToken);
    }

    [Authorize(Roles = "User")]
    [HttpPut("{ticketId:guid}")]
    public async Task<ActionResult<TicketPatchDto>> UpdateTicket(
        [FromBody] TicketUpdateDto ticket,
```

```
    [FromRoute] Guid ticketId,
    CancellationToken cancellationToken)
    => await ticketDetailsService.UpdateTicketDetails(ticket, ticketId, cancellationToken);

[Authorize(Policy = "Admin")]
[HttpPatch("Assign/{ticketId:guid}")]
public async Task<ActionResult<TicketPatchDto>> AssignTicket(
    [FromRoute] Guid ticketId,
    CancellationToken cancellationToken)
{
    var ticket = new TicketAssignDto
    {
        TicketId = ticketId,
                                                    AdminId =
Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!)
    };

    return await ticketDetailsService.MarkAsAssigned(ticket, cancellationToken);
}

[Authorize(Policy = "Admin")]
[HttpPatch("Reassign")]
public async Task<ActionResult<TicketPatchDto>> ReAssignTicket(
    [FromBody] TicketAssignDto ticket,
    CancellationToken cancellationToken)
    => await ticketDetailsService.MarkAsAssigned(ticket, cancellationToken);

[Authorize(Policy = "User")]
[HttpPatch("Solve/{ticketId:guid}")]
public async Task<ActionResult<TicketPatchDto>> SolveTicket(
    [FromRoute] Guid ticketId,
```

```
        CancellationToken cancellationToken)
        => await ticketDetailsService.MarkAsSolved(ticketId, cancellationToken);

    [Authorize(Policy = "User")]
    [HttpPatch("Close/{ticketId:guid}")]
    public async Task<ActionResult<TicketPatchDto>> CloseTicket(
        [FromRoute] Guid ticketId,
        CancellationToken cancellationToken)
        => await ticketDetailsService.MarkAsClosed(ticketId, cancellationToken);

    [Authorize(Policy = "Admin")]
    [HttpPatch("Receive/{number:int}")]
    public async Task<ActionResult<TicketPatchDto>> ReceiveTicket(
        [FromRoute] int number,
        CancellationToken cancellationToken)
        => await ticketDetailsService.MarkAsReceived(number, cancellationToken);

    [Authorize]
    [HttpGet("Statistic")]
    public async Task<ActionResult<List<StatisticDto>>> GetStatistic(
        [FromQuery] StatisticFilter filter,
        CancellationToken cancellationToken)
    {
                                                    var userId =
Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);
        var role = HttpContext.User.FindFirstValue(ClaimTypes.Role);

        filter.UserId = role == "User" ? userId : null;

        return await ticketService.GetTicketsStatistic(filter, cancellationToken);
    }
```

```
}
```

Клас UsersController

```
using CS.BL.Interfaces;
using CS.DOM.DTO;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using CS.DOM.Helpers;
using Environments = CS.BL.Environments;

namespace CS.API.Controllers;

[Route("api/[controller]")]
[ApiController]
public class UsersController(IUserService userService) : ControllerBase
{
    private static readonly string ApiIdentityAddress = Environments.ApiIdentityAddress;

    [Authorize]
    [HttpPost("SignUp")]
    public async Task<IActionResult> SignUp(CancellationToken cancellationToken)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var userDto = new UserInfoDto
        {
            Id = Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!),
            UserName = HttpContext.User.FindFirstValue(ClaimTypes.Name)!,
```

```
            Email = HttpContext.User.FindFirstValue(ClaimTypes.Email)!,
            RoleName = HttpContext.User.FindFirstValue(ClaimTypes.Role)!
        };

        var user = await userService.Create(userDto, cancellationToken);

        return Ok(user);
    }

    [Authorize(Policy = "User")]
    [HttpDelete]
    public async Task<IActionResult> DeleteUser(CancellationToken cancellationToken)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

                                                    var userId =
Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!);

        var auth = Request.Headers.Authorization;

        var client = new HttpClient();

        client.DefaultRequestHeaders.Add("Authorization", auth.ToString());

        var response = await client.DeleteAsync($"{ApiIdentityAddress}/Users");

        if (!response.IsSuccessStatusCode)
        {
```

```
            throw new ApiException(500, "Can not delete user");
        }

        await userService.Delete(userId, cancellationToken);

        return Ok("User was deleted");
    }

    [Authorize]
    [HttpGet]
        public async Task<ActionResult<UserInfoDto>> GetUser(CancellationToken
cancellationToken)
    {
        var user = new UserInfoDto
        {
            Id = Guid.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!),
            UserName = HttpContext.User.FindFirstValue(ClaimTypes.Name)!,
            Email = HttpContext.User.FindFirstValue(ClaimTypes.Email)!,
            RoleName = HttpContext.User.FindFirstValue(ClaimTypes.Role)!
        };

        await userService.EnsureCreated(user, cancellationToken);
        return user;
    }

    [Authorize(Roles = "SuperAdmin, Admin")]
    [HttpGet("Admins")]
    public async Task<ActionResult<List<UserDto>>> GetAdmins(
        CancellationToken cancellationToken)
        => await userService.GetAllAdmins(cancellationToken);
}
```

Клас CustomExceptionFilter

```
using CS.DOM.Helpers;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.Formatters;

namespace CS.API.Helpers;

public class CustomExceptionFilter(ILogger<CustomExceptionFilter> logger) :
IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        var statusCode = context.Exception switch
        {
            ApiException apiException => apiException.Status,
            _ => StatusCodes.Status500InternalServerError
        };

        var detail = context.Exception switch
        {
            _ => "Internal server error"
        };

        var problem = new ProblemDetails
        {
            Status = statusCode,
            Detail = detail,
        };

        if (problem.Status >= StatusCodes.Status500InternalServerError)
```

```
            {
                logger.LogError(context.Exception, "Server error");
            }
            else if (problem.Status >= StatusCodes.Status400BadRequest)
            {
                logger.LogError(context.Exception, "Request error");
            }

            var response = BuildResponse(problem);

                                context.HttpContext.Response.StatusCode = response.StatusCode ??
StatusCodes.Status500InternalServerError;
            context.Result = response;
            context.ExceptionHandled = true;
        }

        private static ObjectResult BuildResponse(ProblemDetails problem) =>
            new ObjectResult(problem)
            {
                StatusCode = problem.Status ?? StatusCodes.Status500InternalServerError,
                ContentTypes = new MediaTypeCollection
                {
                    "application/problem+json"
                }
            };

    }
```

Клас BackgroundNotificationService

```
using CS.BL.Interfaces;
using CS.DAL.DataAccess;
using Microsoft.EntityFrameworkCore;
```

```
namespace CS_Api.BackgroundServices;

public class BackgroundNotificationService(IServiceScopeFactory scopeFactory) :
BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            await CheckUnReadMessages();
            await CheckNewTickets();

            await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
        }
    }

    private async Task CheckUnReadMessages()
    {
        using var scope = scopeFactory.CreateScope();

        var context = scope.ServiceProvider.GetRequiredService<ApplicationContext>();
        var emailService = scope.ServiceProvider.GetRequiredService<IEmailService>();

        var messages = await context.Messages
            .Include(m => m.User)
            .Where(m => !m.IsRead && m.User.RoleName == "User")
            .Where(t => t.WhenSend.AddMinutes(5) < DateTime.Now)
            .ToListAsync();

        foreach (var message in messages)
```

```
    {
        message.IsRead = true;
    }

    var tasks = messages.Select(emailService.SendEmail);
    await Task.WhenAll(tasks);
}

private async Task CheckNewTickets()
{
    using var scope = scopeFactory.CreateScope();

    var context = scope.ServiceProvider.GetRequiredService<ApplicationContext>();
    var emailService = scope.ServiceProvider.GetRequiredService<IEmailService>();

    var tickets = await context.Tickets
        .Include(t => t.Details)
        .Where(t =>
            t.Details.IsAssigned &&
            t.Details.CreationTime.AddMinutes(1) < DateTime.Now &&
            t.Details.HasReceived.HasValue &&
            t.Details.HasReceived == false)
        .ToListAsync();

    foreach (var ticket in tickets)
    {
        ticket.Details.HasReceived = true;
    }

    var tasks = tickets.Select(emailService.SendEmail);
    await Task.WhenAll(tasks);
```

```
        context.Tickets.UpdateRange(tickets);
        await context.SaveChangesAsync();
    }
}
```

Клас AttachmentService

```
using CS.BL.Interfaces;
using CS.DAL.DataAccess;
using CS.DAL.Models;
using CS.DOM.DTO;
using CS.DOM.Helpers;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.StaticFiles;
using Microsoft.EntityFrameworkCore;

namespace CS.BL.Services;

public class AttachmentService(ApplicationContext context) : IAttachmentService
{
    public async Task<Guid> AddTicketAttachment(IFormFile file, Guid ticketId)
    {
        var extension = "." + file.FileName.Split('.')[file.FileName.Split('.').Length - 1];

        var id = Guid.NewGuid();

        string fileName = id + extension;

        var pathBuilt = Path.Combine(Directory.GetCurrentDirectory(),
            $"Attachments\\Tickets\\{ticketId.ToString()}\\");

        if (!Directory.Exists(pathBuilt))
```

```
    {
        Directory.CreateDirectory(pathBuilt);
    }

    var path = Path.Combine(Directory.GetCurrentDirectory(),
        $"Attachments\\Tickets\\{ticketId.ToString()}\\",
        fileName);

    var stream = new FileStream(path, FileMode.Create);

    await file.CopyToAsync(stream);

    await context.TicketAttachments.AddAsync(new TicketAttachment
    {
        Id = id,
        TicketId = ticketId,
        FilePath = path,
    });

    await context.SaveChangesAsync();

    return id;
}

public async Task<Guid> AddMessageAttachment(IFormFile file, Guid messageId)
{
    var extension = "." + file.FileName.Split('.')[file.FileName.Split('.').Length - 1];

    var id = Guid.NewGuid();

    string fileName = id + extension;
```

```
    var pathBuilt = Path.Combine(Directory.GetCurrentDirectory(),
        $"Attachments\\Messages\\{messageId.ToString()}\\");

    if (!Directory.Exists(pathBuilt))
    {
        Directory.CreateDirectory(pathBuilt);
    }

    var path = Path.Combine(Directory.GetCurrentDirectory(),
        $"Attachments\\Messages\\{messageId.ToString()}\\",
        fileName);

    var stream = new FileStream(path, FileMode.Create);

    await file.CopyToAsync(stream);

    await context.MessageAttachments.AddAsync(new MessageAttachment
    {
        Id = id,
        MessageId = messageId,
        FilePath = path,
    });

    await context.SaveChangesAsync();

    return id;
}

public async Task<AttachmentGetDto> GetTicketAttachment(Guid attachmentId)
{
```

```
    var filePath = await context.TicketAttachments
        .Where(t => t.Id == attachmentId)
        .Select(t=>t.FilePath)
        .FirstOrDefaultAsync();

    if (filePath == null)
    {
        throw new ApiException(404, "Ticket attachment does not exist");
    }

    var provider = new FileExtensionContentTypeProvider();

    if (!provider.TryGetContentType(filePath, out var contentType))
    {
        contentType = "application/octet-stream";
    }

    return new AttachmentGetDto
    {
        FileBytes = await File.ReadAllBytesAsync(filePath),
        ContentType = contentType,
        FilePath = Path.Combine(filePath),
    };
}

public async Task<AttachmentGetDto> GetMessageAttachment(Guid attachmentId)
{
    var filePath = await context.MessageAttachments
        .Where(m => m.Id == attachmentId)
        .Select(m=>m.FilePath)
        .FirstOrDefaultAsync();
```

```
        if (filePath == null)
        {
            throw new ApiException(404, "Message attachment does not exist");
        }

        var provider = new FileExtensionContentTypeProvider();

        if (!provider.TryGetContentType(filePath, out var contentType))
        {
            contentType = "application/octet-stream";
        }

        return new AttachmentGetDto
        {
            FileBytes = await File.ReadAllBytesAsync(filePath),
            ContentType = contentType,
            FilePath = Path.Combine(filePath),
        };
    }
}
```

Клас DialogService

```
using CS.BL.Extensions;
using CS.BL.Interfaces;
using CS.DAL.DataAccess;
using CS.DAL.Models;
using CS.DOM.DTO;
using CS.DOM.Helpers;
using CS.DOM.Pagination;
using Microsoft.EntityFrameworkCore;
```

```
namespace CS.BL.Services;

public class DialogService(
    ApplicationContext context,
    IMessageService messageService)
    : IDialogService
{
    public async Task<DialogDto> GetById(
        Guid id,
        Guid userId,
        CancellationToken cancellationToken)
    {
        var dialog = await context.Dialogs
            .Include(d => d.Messages).ThenInclude(m => m.Attachments)
            .Include(d => d.Ticket).ThenInclude(t => t.Details)
            .Include(d => d.Ticket).ThenInclude(t => t.Attachments)
            .FirstOrDefaultAsync(d => d.Id == id, cancellationToken);

        if (dialog == null)
        {
            throw new ApiException(404, "Dialog not found");
        }

        if (dialog.Ticket.CustomerId != userId && dialog.Ticket.AdminId != userId)
        {
            throw new ApiException(403, "No access to dialog");
        }

        var ticket = new DialogTicketDto
        {
            Id = dialog.TicketId,
```

```
            CustomerId = dialog.Ticket.CustomerId,
            AdminId = (Guid)dialog.Ticket.AdminId!,
            Number = dialog.Ticket.Number,
            Request = dialog.Ticket.RequestType,
            Topic = dialog.Ticket.Topic
        };

        return new DialogDto
        {
            Id = dialog.Id,
            Ticket = ticket,
            Messages = await messageService.GetAll(dialog.Id, cancellationToken),
        };
    }

    public async Task<List<DialogShortInfoDto>> GetAllDialogs(
        DialogFilter filter,
        CancellationToken cancellationToken)
    {
        var dialogs = await context.Dialogs
            .Include(d => d.Ticket)
            .ThenInclude(t => t.Customer)
            .Include(dialog => dialog.Messages)
            .Include(dialog => dialog.Ticket)
            .ThenInclude(ticket => ticket.Admin)
            .AsQueryable()
            .PaginateDialogs(filter, cancellationToken);

        if (dialogs == null)
        {
            throw new ApiException(404, "User has no dialogs");
```

```
        }

        var dialogDto = dialogs.Select(dialog => new DialogShortInfoDto
        {
            Id = dialog.Id,
             LastMassage = dialog.Messages.Any() ? dialog.Messages.Last().MessageText : "No
messages yet",
            Number = dialog.Ticket.Number,
            IsRead = dialog.Messages.Any(m => !m.IsRead)
        }).ToList();

        return dialogDto;
    }

    public async Task<DialogCreateDto> Create(
        Guid ticketId,
        Guid adminId,
        CancellationToken cancellationToken)
    {
            var ticket = await context.Tickets.FirstOrDefaultAsync(t => t.Id == ticketId,
cancellationToken: cancellationToken);

        if (ticket == null)
        {
            throw new Exception("Invalid ticket id, can not create dialog");
        }

        ticket.AdminId = adminId;

        var dialog = new Dialog
        {
```

```
            TicketId = ticket.Id,
        };

        context.Dialogs.Add(dialog);

        await context.SaveChangesAsync(cancellationToken);

        return new DialogCreateDto
        {
            TicketId = ticket.Id,
            AdminId = ticket.AdminId,
            CustomerId = ticket.CustomerId
        };
    }
}
```

Клас MessageService

```
using AutoMapper;
using CS.BL.Interfaces;
using CS.DAL.DataAccess;
using CS.DAL.Models;
using CS.DOM.DTO;
using Microsoft.EntityFrameworkCore;

namespace CS.BL.Services;

public class MessageService(ApplicationContext context) : IMessageService
{
    public async Task<List<MessageDto>> GetAll(
        Guid dialogId,
        CancellationToken cancellationToken)
    {
```

```
        var messages = await context.Messages
            .Include(u => u.Attachments)
            .Where(m => m.DialogId == dialogId)
            .OrderByDescending(m => m.WhenSend)
            .ToListAsync(cancellationToken);

        var messagesDto = messages.Select(m => new MessageDto
        {
            DialogId = m.DialogId,
            UserId = m.UserId,
            Text = m.MessageText,
            WhenSended = m.WhenSend,
            UserName = context.Users.FirstOrDefault(u=> u.Id == m.UserId)!.UserName,
            Attachments = m.Attachments.Select(a => a.Id).ToList()
        }).ToList();

        return messagesDto;
    }

     public async Task<Message> SaveMessage(string text, Guid dialogId, Guid userId,
CancellationToken cancellationToken)
    {
        var message = new Message
        {
            DialogId =  dialogId,
            MessageText = text,
            WhenSend = DateTime.Now,
            IsRead = false,
            UserId = userId
        };
```

```
        context.Messages.Add(message);

        await context.SaveChangesAsync(cancellationToken);

        return message;
    }

    public async Task MarkAsRead(Guid dialogId, CancellationToken cancellationToken)
    {
        var messages = await context.Dialogs
            .Include(d => d.Messages)
            .Where(d => d.Id == dialogId)
            .SelectMany(d => d.Messages)
            .ToListAsync(cancellationToken: cancellationToken);

        foreach (var message in messages)
        {
            message.IsRead = true;
        }

        await context.SaveChangesAsync(cancellationToken);
    }
}
```

Клас TicketDetailsService

```
using AutoMapper;
using CS.BL.Interfaces;
using CS.DAL.DataAccess;
using CS.DOM.DTO;
using CS.DOM.Helpers;
using Microsoft.EntityFrameworkCore;
```

```
namespace CS.BL.Services;

public class TicketDetailsService(
    ApplicationContext context,
    IMapper mapper,
    IDialogService dialogService) : ITicketDetailsService
{
    public async Task<TicketPatchDto> MarkAsAssigned(
        TicketAssignDto ticketDto,
        CancellationToken cancellationToken)
    {
        var detail = await context.TicketDetails
                .FirstOrDefaultAsync(d => d.TicketId == ticketDto.TicketId, cancellationToken:
cancellationToken);

        if (detail == null)
        {
            throw new ApiException(404, "Ticket detail not found");
        }

        var ticket = await context.Tickets
                        .FirstOrDefaultAsync(t => t.Id == detail.TicketId, cancellationToken:
cancellationToken);

        if (ticket == null)
        {
            throw new ApiException(404, "Ticket not found");
        }

        detail.AssignmentTime = DateTime.Now;
        ticket.AdminId = ticketDto.AdminId;
```

```
            detail.HasReceived = false;

            detail.IsAssigned = true;

            await dialogService.Create(ticket.Id, (Guid)ticket.AdminId, cancellationToken);

            await context.SaveChangesAsync(cancellationToken);

            return mapper.Map<TicketPatchDto>(detail);
        }

        public async Task<TicketPatchDto> MarkAsSolved(
            Guid ticketId,
            CancellationToken cancellationToken)
        {
            var detail = await context.TicketDetails
                                .FirstOrDefaultAsync(d => d.TicketId == ticketId, cancellationToken:
cancellationToken);

            if (detail == null)
            {
                throw new ApiException(404, "Ticket not found");
            }

            detail.IsSolved = !detail.IsSolved;

            await context.SaveChangesAsync(cancellationToken);

            return mapper.Map<TicketPatchDto>(detail);
        }
```

```
        public async Task<TicketPatchDto> MarkAsClosed(
            Guid ticketId,
            CancellationToken cancellationToken)
        {
            var detail = await context.TicketDetails
                                .FirstOrDefaultAsync(d => d.TicketId == ticketId, cancellationToken:
cancellationToken);

            if (detail == null)
            {
                throw new ApiException(404, "Ticket not found");
            }

            detail.IsSolved = !detail.IsSolved;

            await context.SaveChangesAsync(cancellationToken);

            return mapper.Map<TicketPatchDto>(detail);
        }

        public async Task<TicketPatchDto> MarkAsReceived(
            int number,
            CancellationToken cancellationToken)
        {
            var ticket = await context.Tickets
                .Include(t => t.Details)
                                .FirstOrDefaultAsync(t => t.Number == number, cancellationToken:
cancellationToken);

            if (ticket == null)
            {
```

```
            throw new ApiException(404, "Ticket not found");
        }

        ticket.Details.HasReceived = true;

        await context.SaveChangesAsync(cancellationToken);

        return mapper.Map<TicketPatchDto>(ticket.Details);
    }

    public async Task<TicketPatchDto> UpdateTicketDetails(
        TicketUpdateDto ticket,
        Guid ticketId,
        CancellationToken cancellationToken)
    {
        var detail = await context.TicketDetails
                        .FirstOrDefaultAsync(d => d.TicketId == ticketId, cancellationToken:
cancellationToken);

        if (detail == null)
        {
            throw new ApiException(404, "Ticket not found");
        }

        detail.IsSolved = ticket.IsSolved;

        detail.IsClosed = ticket.IsClosed;

        await context.SaveChangesAsync(cancellationToken);

        return mapper.Map<TicketPatchDto>(detail);
```

```
    }
}
```

Клас TicketService

```
using AutoMapper;
using CS.BL.Extensions;
using CS.BL.Interfaces;
using CS.DAL.DataAccess;
using CS.DAL.Models;
using CS.DOM.DTO;
using CS.DOM.Helpers;
using CS.DOM.Pagination;
using Microsoft.EntityFrameworkCore;

namespace CS.BL.Services;

public class TicketService(
    ApplicationContext context,
    IMapper mapper) : ITicketService
{
    public async Task<TicketShortInfoDto> Create(
        TicketCreateDto ticketDto, Guid userId,
        CancellationToken cancellationToken)
    {
        var ticket = mapper.Map<Ticket>(ticketDto);

        ticket.CustomerId = userId;

        ticket.Details = new TicketDetails
        {
            Description = ticketDto.Description,
            CreationTime = DateTime.Now,
```

```
            IsAssigned = false,
            IsSolved = false,
            IsClosed = false,
            TicketId = ticket.Id,
        };

        context.Tickets.Add(ticket);

        await context.SaveChangesAsync(cancellationToken);

        return mapper.Map<TicketShortInfoDto>(ticket);
    }

    public async Task<PagedResponse<List<TicketShortInfoDto>>> GetAll(
        TicketFilter filter,
        CancellationToken cancellationToken)
    {
        var tickets = await context.Tickets
            .Include(t => t.Details)
            .AsQueryable()
            .Paginate(filter, cancellationToken);

        var ticketDtos = tickets
            .Select(mapper.Map<TicketShortInfoDto>)
            .ToList();

        var totalRecords = await context.Tickets.CountAsync(cancellationToken);
        var totalPages = (int)Math.Ceiling(totalRecords / (double)filter.Take);

        if (ticketDtos == null)
        {
```

```
            throw new ApiException(404, "Tickets not found");
        }

        var pagedResponse = new PagedResponse<List<TicketShortInfoDto>>
        {
            Data = ticketDtos,
            PageNumber = filter.Skip,
            PageSize = filter.Take,
            TotalRecords = totalRecords,
            TotalPages = totalPages
        };

        return pagedResponse;
    }

    public async Task<TicketFullInfoDto> GetFullInfo(int number, CancellationToken
cancellationToken)
    {
        var ticket = await context.Tickets
            .Include(t => t.Details)
            .Include(t => t.Attachments)
            .Include(t => t.Dialog)
            .Select(t => new TicketFullInfoDto
            {
                Id = t.Id,
                CustomerId = t.CustomerId,
                Description = t.Details.Description,
                CreationTime = t.Details.CreationTime,
                IsAssigned = t.Details.IsAssigned,
                Number = t.Number,
                IsClosed = t.Details.IsAssigned,
```

```
            IsSolved = t.Details.IsSolved,
            RequestType = t.RequestType,
            Topic = t.Topic,
            AttachmentIds = t.Attachments.Select(a => a.Id).ToList()
        })
        .FirstOrDefaultAsync(t => t.Number == number, cancellationToken);

    if (ticket == null)
    {
        throw new ApiException(404, "Ticket not found");
    }

    return ticket;
}

public async Task<List<StatisticDto>> GetTicketsStatistic(StatisticFilter filter,
    CancellationToken cancellationToken)
{
    var query = context.Tickets.AsQueryable();

    if (filter.UserId.HasValue)
    {
        query = query.Where(t => t.CustomerId == filter.UserId);
    }

    if (filter.RequestType.HasValue)
    {
        query = query.Where(t => t.RequestType == filter.RequestType);
    }

    if (filter.IsAssigned.HasValue)
```

```
        {
            query = query.Where(t => t.Details.IsAssigned == filter.IsAssigned);
        }

        if (filter.IsSolved.HasValue)
        {
            query = query.Where(t => t.Details.IsSolved == filter.IsSolved);
        }

        if (filter.IsClosed.HasValue)
        {
            query = query.Where(t => t.Details.IsClosed == filter.IsClosed);
        }

        var statistics = await query
            .GroupBy(t => t.RequestType)
            .Select(g => new StatisticDto
            {
                RequestType = g.Key,
                Count = g.Count()
            })
            .ToListAsync(cancellationToken: cancellationToken);

        return statistics;
    }
}
```